

Abraham Silberschatz
Peter Baer Galvin
Greg Gagne

SISTEMI OPERATIVI

Concetti ed esempi

Nona edizione

Edizione italiana a cura di
Riccardo Melen

Pearson Learning Solution

Codice di accesso a eText

SISTEMI OPERATIVI

Nona edizione

Abraham Silberschatz

Peter Baer Galvin

Greg Gagne

SISTEMI OPERATIVI

Nona edizione

Edizione italiana a cura di:

Riccardo Melen
Università di Milano Bicocca

© 2014 Pearson Italia – Milano, Torino

Authorized translation from the English language edition, entitled Operating System Concepts, 9th Edition, (9781118063330/11180633309) by Peter Galvin, Greg Gagne and Abraham Silberschatz, published by John Wiley & Sons, Inc, Copyright © 2013.

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

Italian language edition published by Pearson Italia S.p.A., Copyright © 2014.

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Italia S.p.A. o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro. Per i passi antologici, per le citazioni, per le riproduzioni grafiche, cartografiche e fotografiche appartenenti alla proprietà di terzi, inseriti in quest'opera, l'editore è a disposizione degli aventi diritto non potuti reperire nonché per eventuali non volute omissioni e/o errori di attribuzione nei riferimenti.

È vietata la riproduzione, anche parziale o ad uso interno didattico, con qualsiasi mezzo, non autorizzata.

Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941, n. 633.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEARED, Corso di Porta Romana 108, 20122 Milano, e-mail info@clearedi.org e sito web www.clearedi.org

Edizione italiana a cura di: Riccardo Melen

Traduzione: Pietro Codara

Copy editing: Donatella Pepe

Composizione: Andrea Astolfi

Grafica di copertina: Maurizio Garofalo

Stampa: Tip.Le.Co – S.Bonico (PC)

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

9788865183717

Printed in Italy

9^a edizione: marzo 2014

Ristampa

00 01 02 03 04

Anno

14 15 16 17 18

*Ai miei figli, Lemor, Silvan e Aaron
e alla mia Nicolette*

Avi Silberschatz

*A Brendan ed Ellen,
a Barbara, Anne e Harold, a Walter e Rebecca*

Peter Baer Galvin

Alla mia mamma e al mio papà,

Greg Gagne

Sommario

Prefazione all'edizione italiana	XIX
Prefazione	XXI

Capitolo 1	Introduzione	3
1.1	Che cosa fa un sistema operativo	4
1.1.1	Punto di vista dell'utente	5
1.1.2	Punto di vista del sistema	5
1.1.3	Definizione di sistema operativo	6
1.2	Organizzazione di un sistema elaborativo	7
1.2.1	Funzionamento di un sistema elaborativo	7
1.2.2	Struttura della memoria	10
1.2.3	Struttura di I/O	13
1.3	Architettura degli elaboratori	14
1.3.1	Sistemi monoprocessoress	15
1.3.2	Sistemi multiprocessoress	15
1.3.3	Cluster di elaboratori	18
1.4	Struttura del sistema operativo	21
1.5	Attività del sistema operativo	23
1.5.1	Duplice modalità di funzionamento	24
1.5.2	Timer	26
1.6	Gestione dei processi	27
1.7	Gestione della memoria	28
1.8	Gestione della memoria di massa	29
1.8.1	Gestione dei file	29
1.8.2	Gestione della memoria di massa	30
1.8.3	Cache	31
1.8.4	Sistemi di I/O	33
1.9	Protezione e sicurezza	34
1.10	Strutture dati del kernel	35
1.10.1	Liste, stack e code	35
1.10.2	Alberi	37
1.10.3	Funzioni e mappe hash	38
1.10.4	Bitmap	39
1.11	Ambienti d'elaborazione	39
1.11.1	Elaborazione tradizionale	39
1.11.2	Mobile computing	41
1.11.3	Sistemi distribuiti	42
1.11.4	Computazione client-server	43
1.11.5	Elaborazione peer-to-peer	44
1.11.6	Virtualizzazione	45
1.11.7	Cloud computing	47
1.11.8	Sistemi embedded real-time	48

1.12	Sistemi operativi open-source	49
1.12.1	Storia	50
1.12.2	Linux	51
1.12.3	UNIX BSD	52
1.12.4	Solaris	52
1.12.5	Sistemi open-source come strumenti didattici	53
1.13	Sommario	54

Esercizi di ripasso, 56 • Esercizi, 57 • Note bibliografiche, 59 • Bibliografia, 59

Capitolo 2	Strutture dei sistemi operativi	61
2.1	Servizi di un sistema operativo	62
2.2	Interfaccia con l'utente del sistema operativo	64
2.2.1	Interprete dei comandi	64
2.2.2	Interfaccia grafica con l'utente	66
2.2.3	Scelta dell'interfaccia	67
2.3	Chiamate di sistema	69
2.4	Categorie di chiamate di sistema	73
2.4.1	Controllo dei processi	73
2.4.2	Gestione dei file	78
2.4.3	Gestione dei dispositivi	79
2.4.4	Gestione delle informazioni	79
2.4.5	Comunicazione	80
2.4.6	Protezione	81
2.5	Programmi di sistema	81
2.6	Progettazione e realizzazione di un sistema operativo	83
2.6.1	Scopi della progettazione	83
2.6.2	Meccanismi e politiche	84
2.6.3	Realizzazione	85
2.7	Struttura del sistema operativo	86
2.7.1	Struttura semplice	86
2.7.2	Metodo stratificato	88
2.7.3	Microkernel	90
2.7.4	Moduli	91
2.7.5	Sistemi ibridi	93
2.8	Debugging dei sistemi operativi	95
2.8.1	Analisi dei malfunzionamenti	96
2.8.2	Regolazione delle prestazioni	96
2.8.3	DTrace	98
2.9	Generazione di sistemi operativi	101
2.10	Avvio del sistema	102
2.11	Sommario	104

Esercizi di ripasso, 105 • Esercizi, 106 • Problemi di programmazione, 107 • Progetti di programmazione, 107 • Note bibliografiche, 112 • Bibliografia, 112

Capitolo 3	Processi	117
3.1	Concetto di processo	118
3.1.1	Processo	118
3.1.2	Stato del processo	119
3.1.3	Blocco di controllo dei processi	120
3.1.4	Thread	122

3.2 Scheduling dei processi	122
3.2.1 Code di scheduling	122
3.2.2 Scheduler	125
3.2.3 Cambio di contesto	127
3.3 Operazioni sui processi	127
3.3.1 Creazione di un processo	128
3.3.2 Terminazione di un processo	133
3.4 Comunicazione tra processi	135
3.4.1 Sistemi a memoria condivisa	137
3.4.2 Sistemi a scambio di messaggi	140
3.5 Esempi di sistemi di IPC	144
3.5.1 Un esempio: memoria condivisa in posix	144
3.5.2 Un esempio: Mach	147
3.5.3 Un esempio: Windows	149
3.6 Comunicazione nei sistemi client-server	150
3.6.1 Socket	150
3.6.2 Chiamate di procedure remote	154
3.6.3 Pipe	157
3.7 Sommario	163

Esercizi di ripasso, 164 • Esercizi, 166 • Problemi di programmazione, 169 • Progetti di programmazione, 173 • Note bibliografiche, 178 • Bibliografia, 178

Capitolo 4 Thread	179
4.1 Introduzione	180
4.1.1 Motivazioni	180
4.1.2 Vantaggi	181
4.2 Programmazione multicore	182
4.2.1 Le sfide della programmazione	184
4.2.2 Tipi di parallelismo	185
4.3 Modelli di supporto al multithreading	185
4.3.1 Modello da molti a uno	186
4.3.2 Modello da uno a uno	186
4.3.3 Modello da molti a molti	187
4.4 Librerie dei thread	188
4.4.1 Pthreads	189
4.4.2 Thread in Windows	190
4.4.3 Thread Java	194
4.5 Threading隐式	196
4.5.1 Gruppi di thread	197
4.5.2 OpenMP	198
4.5.3 Grand Central Dispatch	200
4.5.4 Altri approcci	201
4.6 Problematiche di programmazione multithread	201
4.6.1 Chiamate di sistema <code>fork()</code> ed <code>exec()</code>	201
4.6.2 Gestione dei segnali	201
4.6.3 Cancellazione dei thread	203
4.6.4 Dati specifici dei thread	205
4.6.5 Attivazione dello scheduler	205
4.7 Esempi di sistemi operativi	207
4.7.1 Thread di Windows	207
4.7.2 Thread di Linux	208

4.8 Sommario **209**

Esercizi di ripasso, 210 • Esercizi, 210 • Problemi di programmazione, 213 • Progetti di programmazione, 216 • Note bibliografiche, 218 • Bibliografia, 219

Capitolo 5	Sincronizzazione dei processi	221
5.1	Introduzione	222
5.2	Problema della sezione critica	224
5.3	Soluzione di Peterson	226
5.4	Hardware per la sincronizzazione	228
5.5	Lock mutex	231
5.6	Semafori	233
5.6.1	Uso dei semafori	233
5.6.2	Implementazione dei semafori	234
5.6.3	Stallo e attesa indefinita	236
5.6.4	Inversione di priorità	237
5.7	Problemi tipici di sincronizzazione	238
5.7.1	Produttore/consumatore con memoria limitata	238
5.7.2	Problema dei lettori-scrittori	240
5.7.3	Problema dei cinque filosofi (dining philosophers)	242
5.8	Monitor	243
5.8.1	Uso del costrutto monitor	245
5.8.2	Soluzione al problema dei cinque filosofi per mezzo di monitor	248
5.8.3	Realizzazione di un monitor per mezzo di semafori	249
5.8.4	Ripresa dei processi all'interno di un monitor	250
5.9	Esempi di sincronizzazione	252
5.9.1	Sincronizzazione in Windows	253
5.9.2	Sincronizzazione dei processi in Linux	255
5.9.3	Sincronizzazione in Solaris	256
5.9.4	Sincronizzazione in Pthreads	258
5.10	Approcci alternativi	260
5.10.1	Memoria transazionale	260
5.10.2	OpenMP	262
5.10.3	Linguaggi di programmazione funzionali	263
5.11	Sommario	264
<p>Esercizi di ripasso, 265 • Esercizi, 265 • Problemi di programmazione, 271 • Progetti di programmazione, 274 • Note bibliografiche, 281 • Bibliografia, 281</p>		
Capitolo 6	Scheduling della CPU	283
6.1	Concetti fondamentali	284
6.1.1	Ciclicità delle fasi d'elaborazione e di I/O	284
6.1.2	Scheduler della CPU	284
6.1.3	Scheduling con prelazione	286
6.1.4	Dispatcher	287
6.2	Criteri di scheduling	287
6.3	Algoritmi di scheduling	289
6.3.1	Scheduling in ordine d'arrivo	289
6.3.2	Scheduling shortest-job-first	290
6.3.3	Scheduling con priorità	293
6.3.4	Scheduling circolare	295
6.3.5	Scheduling a code multilivello	298
6.3.6	Scheduling a code multilivello con retroazione	299

6.4	Scheduling dei thread	301
6.4.1	Ambito della contesa	301
6.4.2	Scheduling di Pthread	302
6.5	Scheduling per sistemi multiprocessore	304
6.5.1	Approcci allo scheduling per multiprocessori	304
6.5.2	Predilezione per il processore	304
6.5.3	Bilanciamento del carico	306
6.5.4	Processori multicore	306
6.6	Scheduling real-time della CPU	308
6.6.1	Minimizzazione della latenza	308
6.6.2	Scheduling basato sulla priorità	310
6.6.3	Scheduling con priorità proporzionale alla frequenza	312
6.6.4	Scheduling EDF	314
6.6.5	Scheduling a quote proporzionali	315
6.6.6	Scheduling real-time di POSIX	316
6.7	Esempi di sistemi operativi	318
6.7.1	Un esempio: scheduling di Linux	318
6.7.2	Un esempio: scheduling di Windows	320
6.7.3	Un esempio: scheduling di Solaris	324
6.8	Valutazione degli algoritmi	326
6.8.1	Modellazione deterministica	327
6.8.2	Reti di code	328
6.8.3	Simulazioni	329
6.8.4	Realizzazione	331
6.9	Sommario	332
Esercizi di ripasso, 333 • Esercizi, 335 • Note bibliografiche, 340 •		
Bibliografia, 340		

Capitolo 7	Stallo dei processi	343
7.1	Modello del sistema	344
7.2	Caratterizzazione delle situazioni di stallo	347
7.2.1	Condizioni necessarie	347
7.2.2	Grafo di assegnazione delle risorse	347
7.3	Metodi per la gestione delle situazioni di stallo	350
7.4	Prevenire le situazioni di stallo	351
7.4.1	Mutua esclusione	352
7.4.2	Possesso e attesa	352
7.4.3	Assenza di prelazione	353
7.4.4	Attesa circolare	353
7.5	Evitare le situazioni di stallo	356
7.5.1	Stato sicuro	357
7.5.2	Algoritmo con grafo di assegnazione delle risorse	358
7.5.3	Algoritmo del banchiere	359
7.6	Rilevamento delle situazioni di stallo	363
7.6.1	Istanza singola di ciascun tipo di risorsa	363
7.6.2	Più istanze di ciascun tipo di risorsa	364
7.6.3	Uso dell'algoritmo di rilevamento	366
7.7	Ripristino da situazioni di stallo	367
7.7.1	Terminazione dei processi	367
7.7.2	Prelazione delle risorse	368

7.8 Sommario **369**

Esercizi di ripasso, 370 • Esercizi, 372 • Note bibliografiche, 377 •
Bibliografia, 377

Capitolo 8	Memoria centrale	381
8.1	Introduzione	382
8.1.1	Hardware di base	382
8.1.2	Associazione degli indirizzi	384
8.1.3	Spazi di indirizzi logici e fisici	386
8.1.4	Caricamento dinamico	388
8.1.5	Linking dinamico e librerie condivise	388
8.2	Avvicendamento dei processi (swapping)	389
8.2.1	Avvicendamento standard	389
8.2.2	Avvicendamento di processi nei sistemi mobili	391
8.3	Allocazione contigua della memoria	392
8.3.1	Protezione della memoria	393
8.3.2	Allocazione della memoria	394
8.3.3	Frammentazione	395
8.4	Segmentazione	396
8.4.1	Metodo di base	396
8.4.2	Hardware di segmentazione	398
8.5	Paginazione	400
8.5.1	Metodo di base	400
8.5.2	Supporto hardware alla paginazione	405
8.5.3	Protezione	409
8.5.4	Pagine condivise	411
8.6	Struttura della tabella delle pagine	412
8.6.1	Paginazione gerarchica	412
8.6.2	Tabella delle pagine di tipo hash	415
8.6.3	Tabella delle pagine invertita	416
8.6.4	Oracle SPARC Solaris	418
8.7	Esempio: le architetture Intel a 32 e 64 bit	419
8.7.1	Architettura IA-32	419
8.7.2	Architettura x86-64	423
8.8	Esempio: architettura ARM	424
8.9	Sommario	425
<p>Esercizi di ripasso, 426 • Esercizi, 427 • Problemi di programmazione, 431 • Note bibliografiche, 431 • Bibliografia, 432</p>		
Capitolo 9	Memoria virtuale	433
9.1	Introduzione	434
9.2	Paginazione su richiesta	437
9.2.1	Concetti fondamentali	438
9.2.2	Prestazioni della paginazione su richiesta	442
9.3	Copiatura su scrittura	445
9.4	Sostituzione delle pagine	447
9.4.1	Sostituzione di pagina	448
9.4.2	Sostituzione delle pagine secondo l'ordine d'arrivo (FIFO)	451
9.4.3	Sostituzione ottimale delle pagine	453
9.4.4	Sostituzione delle pagine usate meno recentemente (LRU)	454

9.4.5	Sostituzione delle pagine per approssimazione a LRU	457
9.4.6	Sostituzione delle pagine basata su conteggio	459
9.4.7	Algoritmi con buffering delle pagine	460
9.4.8	Applicazioni e sostituzione della pagina	460
9.5	Allocazione dei frame	461
9.5.1	Numero minimo di frame	462
9.5.2	Algoritmi di allocazione	463
9.5.3	Allocazione globale e allocazione locale	464
9.5.4	Accesso non uniforme alla memoria	465
9.6	Thrashing	466
9.6.1	Cause del thrashing	466
9.6.2	Modello del working set	469
9.6.3	Frequenza dei page fault	470
9.6.4	Osservazioni conclusive	471
9.7	File mappati in memoria	471
9.7.1	Meccanismo di base	472
9.7.2	Memoria condivisa nella API Windows	474
9.7.3	Mappatura in memoria dell'I/O	476
9.8	Allocazione di memoria del kernel	478
9.8.1	Sistema buddy	478
9.8.2	Allocazione a lastre	479
9.9	Altre considerazioni	482
9.9.1	Prepaginazione	482
9.9.2	Dimensione delle pagine	482
9.9.3	Portata della TLB	484
9.9.4	Tabella delle pagine invertita	485
9.9.5	Struttura dei programmi	486
9.9.6	Vincolo di I/O e vincolo delle pagine	487
9.10	Esempi di sistemi operativi	489
9.10.1	Windows	489
9.10.2	Solaris	490
9.11	Sommario	492
Esercizi di ripasso, 493 • Esercizi, 496 • Problemi di programmazione, 502 • Progetti di programmazione, 503 • Note bibliografiche, 506 • Bibliografia, 507		

Capitolo 10	Memoria secondaria	511
10.1	Struttura dei dispositivi di memorizzazione	512
10.1.1	Dischi magnetici	512
10.1.2	Dischi a stato solido	513
10.1.3	Nastri magnetici	514
10.2	Struttura dei dischi	515
10.3	Connessione dei dischi	516
10.3.1	Memoria secondaria connessa alla macchina	516
10.3.2	Memoria secondaria connessa alla rete	516
10.3.3	Storage-area network	517
10.4	Scheduling del disco	518
10.4.1	Scheduling in ordine d'arrivo – FCFS	519
10.4.2	Scheduling – SSTF	520
10.4.3	Scheduling – SCAN	521
10.4.4	Scheduling – C-SCAN	522
10.4.5	Scheduling LOOK	522
10.4.6	Scelta di un algoritmo di scheduling	523

10.5	Gestione dell'unità a disco	525
10.5.1	Formattazione del disco	525
10.5.2	Blocco d'avviamento	526
10.5.3	Blocchi difettosi	527
10.6	Gestione dell'area d'avvicendamento	529
10.6.1	Uso dell'area d'avvicendamento	529
10.6.2	Collocazione dell'area d'avvicendamento	530
10.6.3	Gestione dell'area d'avvicendamento: un esempio	531
10.7	Strutture RAID	532
10.7.1	Miglioramento dell'affidabilità tramite la ridondanza	533
10.7.2	Miglioramento delle prestazioni tramite il parallelismo	534
10.7.3	Livelli RAID	534
10.7.4	Scelta di un livello RAID	540
10.7.5	Estensioni	541
10.7.6	Problemi connessi a RAID	542
10.8	Realizzazione della memoria stabile	544
10.9	Sommario	545

Esercizi di ripasso, 546 • Esercizi, 547 • Problemi di programmazione, 550 • Note bibliografiche, 551 • Bibliografia, 551

Capitolo 11	Interfaccia del file system	553
11.1	Concetto di file	554
11.1.1	Attributi dei file	554
11.1.2	Operazioni sui file	555
11.1.3	Tipi di file	559
11.1.4	Struttura dei file	562
11.1.5	Struttura interna dei file	563
11.2	Metodi d'accesso	564
11.2.1	Accesso sequenziale	564
11.2.2	Accesso diretto	564
11.2.3	Altri metodi d'accesso	566
11.3	Struttura della directory e del disco	567
11.3.1	Struttura della memorizzazione di massa	568
11.3.2	Aspetti generali delle directory	569
11.3.3	Directory a un livello	570
11.3.4	Directory a due livelli	571
11.3.5	Directory con struttura ad albero	573
11.3.6	Directory con struttura a grafo aciclico	575
11.3.7	Directory con struttura a grafo generale	578
11.4	Montaggio di un file system	579
11.5	Condivisione di file	581
11.5.1	Utenti multipli	582
11.5.2	File system remoti	582
11.5.3	Semantica della coerenza	586
11.6	Protezione	587
11.6.1	Tipi d'accesso	588
11.6.2	Controllo degli accessi	589
11.6.3	Altri metodi di protezione	591
11.7	Sommario	593

Esercizi di ripasso, 594 • Esercizi, 595 • Note bibliografiche, 596 • Bibliografia, 596

Capitolo 12	Realizzazione del file system	597
12.1	Struttura del file system	598
12.2	Realizzazione del file system	600
12.2.1	Introduzione	600
12.2.2	Partizioni e montaggio	603
12.2.3	File system virtuali	605
12.3	Realizzazione delle directory	607
12.3.1	Lista lineare	607
12.3.2	Tabella hash	608
12.4	Metodi di allocazione	609
12.4.1	Allocazione contigua	609
12.4.2	Allocazione concatenata	612
12.4.3	Allocazione indicizzata	615
12.4.4	Prestazioni	617
12.5	Gestione dello spazio libero	618
12.5.1	Vettore di bit	619
12.5.2	Lista concatenata	619
12.5.3	Raggruppamento	620
12.5.4	Conteggio	620
12.5.5	Mappe di spazio	621
12.6	Efficienza e prestazioni	622
12.6.1	Efficienza	622
12.6.2	Prestazioni	623
12.7	Ripristino	627
12.7.1	Verifica della coerenza	627
12.7.2	File system con log delle modifiche	628
12.7.3	Altre soluzioni	629
12.7.4	Copie di riserva e recupero dei dati	630
12.8	NFS	631
12.8.1	Generalità	631
12.8.2	Protocollo di montaggio	633
12.8.3	Protocollo NFS	634
12.8.4	Traduzione dei nomi di percorso	636
12.8.5	Operazioni remote	637
12.9	Esempio: il file system WAFL	638
12.10	Sommaario	641
Esercizi di ripasso, 642 • Esercizi, 643 • Problemi di programmazione, 645 • Note bibliografiche, 646 • Bibliografia, 647		

Capitolo 13	Sistemi di I/O	649
13.1	Introduzione	650
13.2	Hardware di I/O	650
13.2.1	Polling	653
13.2.2	Interruzioni	654
13.2.3	Accesso diretto alla memoria (DMA)	659
13.2.4	Concetti principali dell'hardware di I/O	660
13.3	Interfaccia di I/O per le applicazioni	661
13.3.1	Dispositivi con trasferimento a blocchi e a caratteri	664
13.3.2	Dispositivi di rete	665
13.3.3	Orologi e timer	665
13.3.4	I/O non bloccante e asincrono	666
13.3.5	I/O vettorizzato	668

13.4	Sottosistema di I/O del kernel	669
13.4.1	Scheduling dell'I/O	669
13.4.2	Gestione dei buffer	670
13.4.3	Cache	672
13.4.4	Code di spooling e riservazione dei dispositivi	672
13.4.5	Gestione degli errori	673
13.4.6	Protezione dell'I/O	674
13.4.7	Strutture dati del kernel	675
13.4.8	Concetti principali del sottosistema di I/O del kernel	675
13.5	Trasformazione delle richieste di I/O in operazioni hardware	676
13.6	STREAMS	679
13.7	Prestazioni	681
13.8	Sommario	685
Esercizi di ripasso, 686 • Esercizi, 686 • Note bibliografiche, 688 • Bibliografia, 688		

Capitolo 14	Protezione	691
14.1	Scopi della protezione	692
14.2	Principi di protezione	693
14.3	Domini di protezione	694
14.3.1	Struttura dei domini di protezione	694
14.3.2	Un esempio: UNIX	696
14.3.3	Un esempio: MULTICS	697
14.4	Matrice d'accesso	699
14.5	Realizzazione della matrice d'accesso	703
14.5.1	Tabella globale	704
14.5.2	Liste d'accesso per oggetti	704
14.5.3	Liste delle abilitazioni per domini	704
14.5.4	Meccanismo chiave-serratura	705
14.5.5	Confronto	706
14.6	Controllo dell'accesso	707
14.7	Revoca dei diritti d'accesso	707
14.8	Sistemi basati su abilitazioni	709
14.8.1	Un esempio: Hydra	710
14.8.2	Un esempio: sistema Cambridge CAP	711
14.9	Protezione basata sul linguaggio	712
14.9.1	Controllo realizzato dal compilatore	713
14.9.2	Protezione nel linguaggio Java	716
14.10	Sommario	719
Esercizi di ripasso, 719 • Esercizi, 720 • Note bibliografiche, 722 • Bibliografia, 722		

Capitolo 15	Sicurezza	725
15.1	Il problema della sicurezza	726
15.2	Minacce legate ai programmi	730
15.2.1	Cavalli di Troia	730
15.2.2	Trabocchetti	732
15.2.3	Bomba logica	732

15.2.4	Stack e buffer overflow	732
15.2.5	Virus	737
15.3	Minacce relative al sistema e alla rete	740
15.3.1	Worm	741
15.3.2	Scansione delle porte	744
15.3.3	Attacchi denial-of-service	745
15.4	Crittografia come strumento per la sicurezza	746
15.4.1	Cifratura	747
15.4.2	Implementazione della crittografia	755
15.4.3	Un esempio: SSL	756
15.5	Autenticazione degli utenti	758
15.5.1	Password	758
15.5.2	Vulnerabilità delle password	759
15.5.3	Password cifrate	760
15.5.4	Password monouso	762
15.5.5	Tecniche biometriche	763
15.6	Realizzazione delle misure di sicurezza	763
15.6.1	Politiche di sicurezza	764
15.6.2	Verifica delle vulnerabilità	764
15.6.3	Rilevamento delle intrusioni	766
15.6.4	Protezione dai virus	769
15.6.5	Verifica, accounting e log	770
15.7	Firewall a protezione di sistemi e reti	772
15.8	Classificazione della sicurezza dei calcolatori	773
15.9	Un esempio: Windows 7	775
15.10	Sommario	778

Esercizi di ripasso, 779 • Note bibliografiche, 780 • Bibliografia, 782

Capitolo 16	Macchine virtuali	787
16.1	Introduzione	788
16.2	Storia	790
16.3	Vantaggi e caratteristiche	791
16.4	Blocchi costituenti	793
16.4.1	Trap-and-Emulate	794
16.4.2	Traduzione binaria	795
16.4.3	Assistenza hardware	797
16.5	Tipologie di macchine virtuali e loro implementazioni	800
16.5.1	Il ciclo di vita della macchina virtuale	800
16.5.2	Hypervisor di tipo 0	801
16.5.3	Hypervisor di tipo 1	801
16.5.4	Hypervisor di tipo 2	803
16.5.5	Paravirtualizzazione	803
16.5.6	Virtualizzazione dell'ambiente di programmazione	804
16.5.7	Emulazione	805
16.5.8	Contenitori di applicazioni	806
16.6	Virtualizzazione e componenti dei sistemi operativi	806
16.6.1	Scheduling della CPU	806
16.6.2	Gestione della memoria	808
16.6.3	I/O	810
16.6.4	Gestione dello storage	811
16.6.5	Migrazione in tempo reale	812

16.7 Esempi	815
16.7.1 VMware	815
16.7.2 Java virtual machine	815
16.8 Sommario	817
Esercizi di ripasso, 818 • Note bibliografiche, 819 • Bibliografia, 819	
Capitolo 17 Sistemi distribuiti	821
17.1 Vantaggi dei sistemi distribuiti	822
17.1.1 Condivisione delle risorse	822
17.1.2 Velocizzazione delle elaborazioni	823
17.1.3 Affidabilità	823
17.1.4 Comunicazione	823
17.2 Tipi di sistemi operativi distribuiti	824
17.2.1 Sistemi operativi di rete	824
17.2.2 Sistemi operativi distribuiti	826
17.3 Tipi di reti	828
17.3.1 Reti locali	828
17.3.2 Reti geografiche	830
17.4 Struttura della comunicazione	832
17.4.1 Naming e risoluzione dei nomi	832
17.4.2 Strategie d'instradamento	835
17.4.3 Strategie riguardanti l'invio dei pacchetti	836
17.4.4 Strategie di connessione	837
17.5 Protocolli di comunicazione	838
17.6 Un esempio: TCP/IP	842
17.7 Robustezza	844
17.7.1 Rilevamento dei guasti	844
17.7.2 Riconfigurazione	845
17.7.3 Ripristino dopo un guasto	846
17.7.4 Tolleranza ai guasti	846
17.8 Problemi di progettazione	847
17.9 File system distribuiti	849
17.9.1 Naming e trasparenza	850
17.9.2 Accesso ai file remoti	854
17.10 Sommario	858
Esercizi di ripasso, 859 • Esercizi, 860 • Note bibliografiche, 861 • Bibliografia, 862	
Indice analitico	863

Disponibili on-line



- Capitolo 18 Linux
- Capitolo 19 Windows 7
- Capitolo 20 Prospettiva storica
- Appendice A BSD Unix
- Appendice B The Mach System

Prefazione all'edizione italiana

Questo libro, giunto ormai alla nona edizione, si conferma come uno dei più autorevoli e diffusi testi didattici sui sistemi operativi. Il suo successo è dovuto alla continua opera di aggiornamento svolta da parte degli autori, che in ogni edizione non solamente introducono e approfondiscono le nuove tematiche più rilevanti nell'evoluzione dello scenario tecnologico (in questa edizione viene per esempio dedicato un intero capitolo alla virtualizzazione), ma hanno anche il coraggio di cambiare l'impostazione data ad argomenti più consolidati per migliorarne continuamente la fruibilità dal punto di vista didattico.

Un altro grande merito del libro risiede nel suo approccio all'insegnamento della materia, che enfatizza le attività pratiche, rappresentate sia dai numerosi esercizi di programmazione sia da progetti complessi, che permettono allo studente di affrontare in maniera strutturata problemi di dimensioni più realistiche.

Nel preparare la traduzione di questa edizione sono state anche riviste le parti già presenti nelle edizioni precedenti, allo scopo di modernizzare e uniformare il più possibile la terminologia utilizzata, rendendola più simile a quella correntemente utilizzata nella pratica professionale dell'informatica.

Prof. Riccardo Melen

Dipartimento di Informatica,
Sistemistica e Comunicazione
Università di Milano Bicocca

Pearson Learning Solution



Il codice di registrazione che trovate sulla copertina di questo libro consente l'accesso per diciotto mesi al Pearson eText, l'edizione digitale del libro con alcuni utili strumenti che permettono di evidenziare, commentare e appuntare note sulle sezioni che giudicate significative. Questo nuovo approccio digitale al testo consente una lettura non lineare, personalizzabile e agganciata ai contenuti di supporto.

Tra i supplementi di questo eText trovate:

- Tre capitoli aggiuntivi (Capitolo 18 Linux, Capitolo 19 Windows 7, Capitolo 20 Prospettiva storica)
- L'Appendice A BSD Unix
- L'Appendice B The Mach System
- Il simulatore della Linux Virtual Machine
- Codice sorgente Java e C
- Una raccolta di testi di approfondimento su Java

Prefazione

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

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

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

I concetti fondamentali e gli algoritmi trattati in questo testo spesso si basano su quelli impiegati sia nei sistemi operativi commerciali sia in quelli open-source. Si è in ogni modo cercato di presentare questi concetti e algoritmi in una forma generale, non legata a un particolare sistema operativo. Nonostante ciò, nel libro sono presenti molti esempi che riguardano i sistemi operativi più diffusi, oltre che i più innovativi, tra cui Linux, Microsoft Windows, Apple Mac OS X e Solaris. Sono inclusi anche alcuni esempi riguardanti Android e iOS, i due sistemi operativi attualmente dominanti nel settore dei dispositivi mobili.

La struttura di questo testo rispecchia la lunga esperienza degli Autori, in qualità di docenti, nei rispettivi corsi di sistemi operativi. Inoltre, il testo recepisce nei contenuti quanto suggerito sull'insegnamento dei sistemi operativi dalla IEEE Computing Society e dall'Association for Computing Machinery (ACM). Nel redigere il testo si è tenuto conto delle valutazioni dei revisori del testo, nonché dei commenti e dei suggerimenti inviati dai lettori delle precedenti edizioni e dai nostri studenti.

Contenuti del libro

Il testo è suddiviso in otto parti principali, l'ultima delle quali, dedicata ai casi di studio su Linux e Windows 7, è disponibile sul sito web del volume.

- **Generalità.** Nei Capitoli 1 e 2 si spiega che cosa sono i sistemi operativi, che cosa fanno e come sono *progettati* e *realizzati*, attraverso un'analisi delle comuni caratteristiche dei sistemi operativi e di quei servizi che un sistema operativo deve fornire agli utenti. Vengono considerati sia sistemi operativi per PC e server sia sistemi operativi per dispositivi mobili. La presentazione è di tipo descrittivo e mira a motivare lo studio di queste tematiche. Essendo privi di riferimenti al funzionamento interno, questi capitoli sono consigliabili a chiunque voglia sapere che cos'è un sistema operativo, senza soffermarsi sui dettagli degli algoritmi che ne controllano il funzionamento.
- **Gestione dei processi.** Nei Capitoli dal 3 al 7 si descrivono i concetti di processo e concorrenza che costituiscono il fondamento dei moderni sistemi operativi. Per processo s'intende l'unità di lavoro di un sistema, che consiste quindi in un insieme di processi eseguiti in modo concorrente, alcuni dei quali sono parte del sistema operativo (quelli incaricati di eseguire il codice di sistema), mentre i rimanenti sono i processi utente (il cui scopo è, appunto, l'esecuzione del codice utente). In questi capitoli si affrontano i differenti metodi impiegati per lo scheduling e la sincronizzazione dei processi, la comunicazione tra processi e la gestione delle situazioni di stallo. Tra questi argomenti è compresa un'analisi dei thread e un esame dei temi relativi ai sistemi multicore e alla programmazione concorrente.
- **Gestione della memoria.** Nei Capitoli 8 e 9 è trattata la gestione della memoria centrale durante l'esecuzione di un processo. Al fine di migliorare sia l'utilizzo della CPU sia la velocità di risposta ai propri utenti, un calcolatore deve essere in grado di mantenere contemporaneamente più processi in memoria. Esistono molti schemi per la gestione della memoria centrale; questi schemi riflettono diverse strategie di gestione della memoria e l'efficacia dei diversi algoritmi dipende dal particolare contesto in cui si applicano.
- **Gestione della memoria secondaria.** Nei Capitoli dal 10 al 13 si spiega come gli elaboratori moderni gestiscano il file system, la memoria di massa e l'I/O. Il file system fornisce il meccanismo per la memorizzazione e l'accesso ai dati e ai programmi che risiedono sui dischi. Vengono descritti i fondamentali algoritmi interni e le strutture di gestione della memoria e viene fornita una solida conoscenza pratica degli algoritmi utilizzati, analizzandone le proprietà, i vantaggi e gli svantaggi. Dal momento che i dispositivi di I/O collegabili a un calcolatore sono del più vario genere, è necessario che il sistema operativo fornisca funzionalità ampie e diversificate alle applicazioni, cosicché queste possano tenere sotto controllo i dispositivi in ogni loro aspetto. La trattazione dell'I/O mira ad approfondirne progettazione, interfacce, strutture e funzioni interne del sistema. Per molti aspetti, i dispositivi di I/O sono i più lenti fra i componenti principali del calcolatore: vengono dunque analizzati i problemi che derivano da questo collo di bottiglia nelle prestazioni.
- **Protezione e sicurezza.** I Capitoli 14 e 15 illustrano i meccanismi necessari a garantire protezione e sicurezza dei sistemi elaborativi. Tutti i processi di un sistema

operativo devono essere reciprocamente protetti; a tale scopo bisogna garantire che solo i processi autorizzati dal sistema operativo possano impiegare le risorse del sistema, come file, segmenti di memoria, CPU e altre risorse. La protezione è il meccanismo attraverso il quale il sistema controlla l'accesso alle risorse da parte di programmi, processi o utenti. Tale meccanismo deve fornire un metodo per definire i controlli e i vincoli ai quali gli utenti vanno sottoposti, e i mezzi per realizzarli. La sicurezza, invece, consiste nel proteggere sia le informazioni memorizzate all'interno del sistema (dati e codice) sia le risorse fisiche del sistema elaborativo da accessi non autorizzati, tentativi di alterazione o distruzione e dall'introduzione di incongruenze nel funzionamento.

- **Argomenti avanzati.** I Capitoli 16 e 17 trattano le macchine virtuali e i sistemi distribuiti. Il Capitolo 16 è un capitolo nuovo che fornisce una panoramica delle macchine virtuali e della loro relazione con i sistemi operativi moderni; vengono inoltre illustrate le tecniche hardware e software che rendono possibile la virtualizzazione. Il Capitolo 17 è un condensato, aggiornato, dei tre capitoli sul calcolo distribuito presenti nella precedente edizione. Questa modifica ha lo scopo di rendere possibile ai docenti la copertura di questi argomenti nel poco tempo disponibile in un semestre e di fornire più velocemente agli studenti una conoscenza delle idee alla base del calcolo distribuito.
- **Casi di studio.** I Capitoli 18 e 19 (disponibili on-line all'indirizzo www.pearson.it/place), presentano nel dettaglio alcuni casi di studio di sistemi operativi reali come Linux, Windows 7, FreeBSD e Mach. Sia Linux sia Windows 7 sono trattati anche nel resto del volume, ma i casi di studio sono più dettagliati. È di particolare interesse il confronto tra i progetti di questi due sistemi molto differenti tra loro. Il Capitolo 20 descrive brevemente alcuni tra i sistemi operativi che hanno maggiormente influenzato l'evoluzione di questo settore.

Nona edizione

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

1. sistemi multicore;
2. dispositivi mobili;
3. virtualizzazione.

Per dare enfasi ai suddetti argomenti sono state fatte in questa nuova edizione diverse integrazioni e, nel caso della virtualizzazione, è stato scritto un capitolo interamente nuovo. Inoltre, è stato rivisto il contenuto di quasi tutti i capitoli, aggiornando gli argomenti più datati ed eliminando il materiale non più rilevante.

È stata anche modificata sostanzialmente l'organizzazione, per esempio eliminando il capitolo sui sistemi real-time e coprendo questo argomento negli altri capitoli, riordinando i capitoli sulla gestione della memoria secondaria e spostando la presentazione della sincronizzazione dei processi prima dello scheduling. La maggior parte di queste modifiche si basa sulla nostra esperienza di insegnamento nel corso di sistemi operativi.

Presentiamo di seguito un breve elenco dei principali cambiamenti.

- Il **Capitolo 1, Introduzione**, è stato arricchito includendo i sistemi multiprocessore e multicore e comprende anche un nuovo paragrafo sulle strutture dati utilizzate nel kernel. La parte sulle piattaforme di calcolo include ora i sistemi mobili e il cloud computing ed è stata inserita una panoramica dei sistemi real-time.
- Il **Capitolo 2, Strutture dei sistemi operativi**, dà spazio alle interfacce utente per i dispositivi mobili, presenta i sistemi iOS e Android ed espande la parte dedicata a Mac OS X come esempio di sistema ibrido.
- Il **Capitolo 3, Processi**, include ora il multitasking nei sistemi operativi per dispositivi mobili, presenta il modello multiprocesso del browser Google Chrome e tratta i processi orfani e zombie in UNIX.
- Il **Capitolo 4, Thread**, vede ampliata la trattazione del parallelismo e della legge di Amdahl. È inoltre presente una nuova sezione sul threading implicito che include OpenMP e GCD (Grand Central Dispatch) di Apple.
- Il **Capitolo 5** (ex Capitolo 6), **Sincronizzazione di processi**, vede aggiunta un nuovo paragrafo dedicato ai lock mutex, descrive la sincronizzazione con l'uso di OpenMP e introduce i linguaggi funzionali.
- Il **Capitolo 6** (ex Capitolo 5), **Scheduling della CPU**, contiene ora una trattazione dello scheduler CFS di Linux e dello scheduling in modalità utente di Windows. Sono state inoltre integrate le parti relative agli algoritmi di scheduling real-time.
- Il **Capitolo 7, Stallo dei processi**, non ha subito modifiche di rilievo.
- Il **Capitolo 8, Memoria centrale**, introduce ora anche l'avvicendamento (*swapping*) su dispositivi mobili e le architetture Intel a 32 e 64 bit. Un nuovo paragrafo è dedicato all'architettura ARM.
- Il **Capitolo 9, Memoria virtuale**, è stato aggiornato nella parte relativa alla gestione della memoria del kernel e include ora gli allocator SLUB e SLOB di Linux.
- Il **Capitolo 10** (ex Capitolo 12), **Memoria secondaria**, introduce ora anche i dischi a stato solido (SSD).
- Il **Capitolo 11** (ex Capitolo 10), **Interfaccia del file system**, è stato aggiornato con informazioni relative alle tecnologie correnti.
- Il **Capitolo 12** (ex Capitolo 11), **Realizzazione del file system**, è stato aggiornato alle tecnologie correnti.
- Il **Capitolo 13, Sistemi di I/O**, vede l'aggiunta di nuove tecnologie e l'aggiornamento dei dati sulle prestazioni. Sono state inoltre ampliate le parti relative all'I/O sincrono/asincrono e bloccante/non bloccante ed è stato aggiunto un paragrafo sull'I/O vettorizzato.
- Il **Capitolo 14, Protezione**, non ha subito modifiche sostanziali.
- Il **Capitolo 15, Sicurezza**, contiene un paragrafo sulla crittografia totalmente revisionato con l'utilizzo di una notazione moderna e con una spiegazione migliorata dei vari metodi di crittografia e del loro utilizzo. Il capitolo include ora una parte sulla sicurezza in Windows 7.
- Il **Capitolo 16, Macchine virtuali**, è un capitolo nuovo che offre una panoramica della virtualizzazione e di come tale tecnica sia in relazione con gli attuali sistemi operativi.

- Il **Capitolo 17, Sistemi distribuiti**, è un capitolo nuovo che combina e aggiorna parte del contenuto dei Capitoli 16, 17 e 18 della precedente edizione.
- Il **Capitolo 18** (ex Capitolo 21), **Linux**, (disponibile sul sito web del volume) è stato aggiornato e introduce ora la versione 3.2 del kernel.
- Il **Capitolo 19, Windows 7**, (disponibile sul sito web del volume) è un capitolo nuovo dedicato al sistema operativo Windows 7.
- Il **Capitolo 20, Prospettiva storica**, (disponibile sul sito web del volume), non ha subìto modifiche di rilievo.

Ambienti di programmazione

Per illustrare i concetti fondamentali della materia, in questo libro sono esaminati molti sistemi operativi realmente esistenti. Si è dedicata particolare attenzione ai sistemi operativi Linux e Microsoft Windows, ma si fa riferimento anche alle varie versioni di UNIX (tra cui Solaris, BSD e Mac OS X).

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

- **POSIX.** La sigla POSIX (ossia *interfaccia portabile del sistema operativo*) rappresenta una serie di standard creati essenzialmente per sistemi operativi della famiglia UNIX. Sebbene il sistema operativo Windows possa eseguire alcuni programmi POSIX, la nostra trattazione è prettamente incentrata sui sistemi UNIX e Linux. I sistemi compatibili con POSIX devono implementare lo standard di base (POSIX.1): è questo il caso di Linux, Solaris e Mac OS X. Esistono poi numerose estensioni degli standard di base; tra queste, l'estensione real-time (POSIX1.b) e quella per i thread (POSIX1.c, meglio nota come Pthreads). Vari programmi, scritti in C, fungono da esempio per chiarire non solo il funzionamento dell'interfaccia API di base, ma anche quello di Pthreads e dello standard per la programmazione real-time. Questi programmi dimostrativi sono stati testati sulle versioni 2.6 e 3.2 di Linux, su Mac OS X 10.7 e su Solaris 10 con l'ausilio del compilatore `gcc` versione 4.0.
- **Java.** Java è un linguaggio di programmazione largamente utilizzato, dotato di una ricca API, oltre che di funzionalità integrate per la creazione e la gestione dei thread. I programmi Java sono eseguibili da qualsiasi sistema operativo, purché vi sia installata una macchina virtuale Java (o JVM). Si illustrano vari concetti in relazione ai sistemi operativi e alle architetture di rete, grazie a programmi testati con la JVM 1.6.
- **Sistemi Windows.** Il più rilevante ambiente di programmazione per i sistemi Windows è l'API di Windows, che dispone di un insieme completo di funzioni per la gestione di processi, thread, memoria e periferiche. Per illustrare l'uso di questa API ci si è avvalsi di vari programmi in C. I programmi dimostrativi sono stati testati su piattaforme Windows XP e Windows 7.

Abbiamo scelto i tre suddetti ambienti di programmazione poiché li ritengiamo i più adatti a rappresentare i due modelli di sistemi operativi più diffusi, Windows e UNIX/Linux, al pari dell'ambiente Java, ampiamente diffuso. I programmi dimostrativi, scritti prevalentemente in C, presuppongono una certa dimestichezza da parte dei let-

tori con tale linguaggio; i lettori con buona padronanza di Java, oltre che del linguaggio C, dovrebbero comprendere senza problemi la maggior parte dei programmi.

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

Macchina virtuale Linux

Per aiutare gli studenti nella comprensione del sistema Linux mettiamo a disposizione una macchina virtuale Linux, con il codice sorgente del sistema operativo, scaricabile dal sito web di supporto a questo testo. Questa macchina virtuale comprende anche un ambiente di sviluppo gcc con compilatori ed editor. La maggior parte degli esercizi di programmazione contenuti nel libro può essere risolta su questa macchina virtuale, a eccezione di quelli che richiedono Java o le API di Windows.

Sono inoltre forniti tre esercizi di programmazione sulla modifica del kernel di Linux con l’uso di moduli del kernel:

1. aggiunta di un modulo al kernel di Linux
2. aggiunta di un modulo che utilizza diverse strutture dati del kernel
3. aggiunta di un modulo che consente di scorrere le attività in un sistema Linux in esecuzione.

Contatti

Sono benvenuti i suggerimenti su come migliorare il libro. Riceviamo inoltre volentieri ogni contributo al sito web che possa essere di aiuto agli altri lettori, come esercizi di programmazione, progetti, laboratori e tutorial on-line e consigli per l’insegnamento.

Inviate le vostre email all’indirizzo os-book-authors@cs.yale.edu

Ringraziamenti

Questo testo deriva dalle precedenti edizioni, le prime tre delle quali sono state scritte insieme con James Peterson. Tra le altre persone che sono state d’aiuto per quanto riguarda le precedenti edizioni ci sono Hamid Arabnia, Rida Bazzi, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, Roy Cambell, P.C. Capon, John Carpenter, Gil Carrick, Thomas Casavant, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Rasit Eskicioglu, Hans Flack, Robert Fowler, G. Scott Graham, Richard Guy, Max Hailparin, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Bruce Hillyer, Mark Holliday, Dean Hougen, Michael Huang, Ahmed Kamel, Richard Kieburz, Carol Kroll, Morty Kwe-

stel, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Yoichi Muraoka, Jim M. Ng, Banu Ozden, Ed Posnak, Boris Putanec, Charles Qualline, John Quarterman, Mike Reiter, Gustavo Rodriguez-Rivera, Carolyn J.C. Schauble, Thomas P. Skinner, Yannis Smaragdakis, Jesse St. Laurent, John Stankovich, Adam Stauffer, Steven Stepanek, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, John Werth, James M. Westall, J.S. Weston e Yang Xiang.

Robert Love ha aggiornato il Capitolo 18 e le parti riguardanti Linux presenti nel testo, oltre ad aver risposto a tutte le nostre domande relative ad Android. Il Capitolo 19 è stato scritto da Dave Probert sulla base del Capitolo 22 dell'ottava edizione. Jonathan Katz ha contribuito al Capitolo 15. Richard West ci ha dato suggerimenti per il Capitolo 16. Slahuddin Khan ha aggiornato il Paragrafo 15.9 per introdurre la sicurezza in Windows 7.

Parte del Capitolo 17 deriva da un articolo di Levy e Silbershatz del 1990. Il Capitolo 18 è basato su un lavoro non pubblicato di Stephen Tweedie. Cliff Martin ci ha aiutato nell'introdurre FreeBSD nell'appendice su UNIX. Alcuni esercizi con le relative soluzioni ci sono stati forniti da Arvind Krishnamurthy. Andrew DeNicola ha preparato la guida allo studio che è disponibile sul sito web. Alcune delle slide sono state realizzate da Marilyn Turnamian.

Mike Shapiro, Bryan Cantrill e Jim Mauro hanno risposto a diverse domande riguardanti Solaris. Bryan Cantrill della Sun Microsystems ci ha aiutato per la parte su ZFS. Josh Dees e Rob Reynolds hanno contribuito alla parte su Microsoft .NET. John Trono, dell'Università di Saint Michael di Colchester, Vermont, ha contribuito al progetto sulle code di messaggi POSIX.

Judi Paige ha aiutato nella creazione delle immagini e delle slide per le presentazioni. Thomas Gagne ha predisposto la nuova grafica per questa edizione. Mark Wogahn si è preoccupato del corretto funzionamento del software utilizzato per la stesura di questo volume (Latex e font) e Ranjar Kumar Keher ha riscritto per noi parte del software Latex. Il nostro editor esecutivo, Neth Lang Golub, ci ha guidato in maniera esperta nel corso della preparazione di questa edizione, assistita da Katherine Willis, che ha gestito con attenzione molti dettagli di questo progetto. L'editor di produzione, Ken Santor, è stato determinato nella gestione di tutti i dettagli di produzione.

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

Abraham Silberschatz, New Haven, CT, 2012

Peter Baer Galvin, Boston, MA, 2012

Greg Gagne, Salt Lake City, UT, 2012

Gli Autori

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

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

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

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

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

Generalità

Un sistema operativo è un programma che agisce come intermediario tra l'utente e l'hardware di un calcolatore. Scopo di un sistema operativo è fornire un ambiente nel quale un utente possa eseguire programmi in modo conveniente ed efficiente.

Un sistema operativo è il software che gestisce l'hardware di un calcolatore. L'hardware deve fornire meccanismi idonei che assicurino il corretto funzionamento dell'elaboratore e lo preservino da eventuali interferenze improprie da parte dei programmi utenti.

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

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

**OBIETTIVI
DEL CAPITOLO**

- Descrivere l'organizzazione di base di un computer.
- Offrire una panoramica dei più importanti componenti di un sistema operativo.
- Presentare le svariate tipologie di ambienti elaborativi.
- Esplorare diversi sistemi operativi open-source.

Introduzione

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

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

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

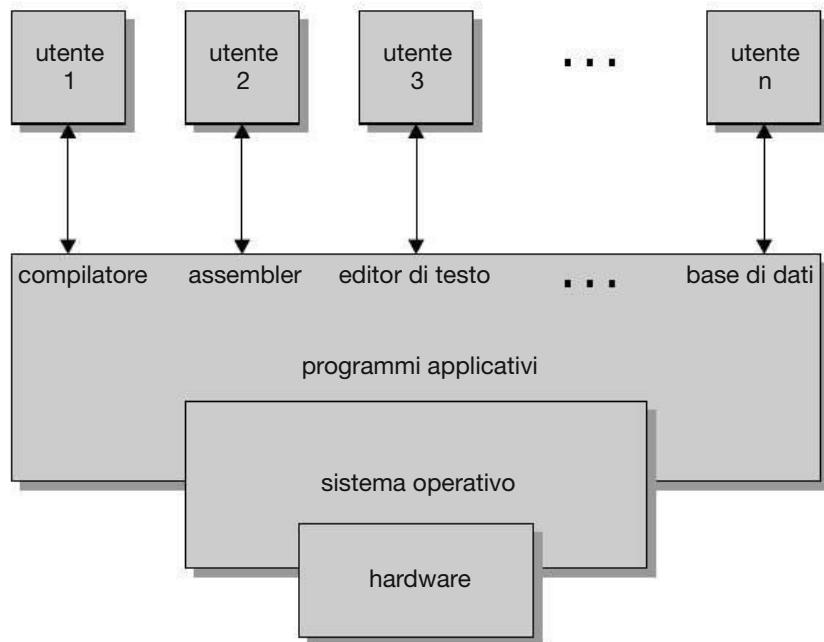


Figura 1.1 Componenti di un sistema elaborativo.

1.1 Che cosa fa un sistema operativo

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

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

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

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

1.1.1 Punto di vista dell'utente

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

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

In altri casi ci sono utenti che usano **stazioni di lavoro**, connesse a reti di altre stazioni di lavoro e **server**; dispongono di risorse loro riservate, ma altre devono condividerle, come la rete e i server (per servizi di accesso a file, stampa ed elaborazione). Tutto ciò richiede che il sistema operativo sia progettato ponderando l'adeguatezza all'uso individuale e l'utilizzo ottimale delle risorse.

Negli ultimi tempi si sono diffuse diverse tipologie di dispositivi mobili, come gli smartphone e i tablet. La maggior parte di questi dispositivi è costituita da unità stand-alone utilizzate da un singolo utente, spesso collegate a Internet mediante tecnologia cellulare o Wi-Fi. I dispositivi mobili stanno rimpiazzando i computer portatili e desktop tra gli utenti interessati principalmente all'utilizzo della posta elettronica e alla navigazione web. L'interfaccia utente dei dispositivi mobili è generalmente costituita da un **touch screen** che permette di interagire con il sistema attraverso movimenti delle dita sullo schermo, senza utilizzo di tastiera e mouse.

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

1.1.2 Punto di vista del sistema

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

e utenti affinché il sistema elaborativo operi in modo equo ed efficiente. Come abbiamo visto, l’assegnazione delle risorse è importante soprattutto nel caso in cui molti utenti accedono agli stessi mainframe o minicalcolatori.

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

1.1.3 Definizione di sistema operativo

A questo punto avrete già capito che il termine *sistema operativo* definisce diversi ruoli e funzionalità. Ciò è dovuto, almeno in parte, ai differenti modelli di computer e ai loro utilizzi: i computer sono infatti presenti ovunque, dai tostapane alle automobili, dalle navi ai veicoli spaziali, dalle abitazioni alle aziende e sono la base di piattaforme per videogiochi, lettori multimediali, decoder per la televisione e sistemi di controllo industriale. Anche se i computer hanno una storia piuttosto breve, la loro evoluzione è stata rapida. La storia del calcolo automatico iniziò come esperimento per capire quello che era possibile fare ed evolse velocemente in sistemi dedicati per scopi militari, come la decifrazione di codici o il disegno di traiettorie, e per usi governativi, come i censimenti. Da questi primi computer si passò a mainframe multifunzione per uso generale: fu in questa fase che nacquero i sistemi operativi. Negli anni ’60 la **legge di Moore** predisse che il numero dei transistor nei circuiti integrati sarebbe raddoppiato ogni 18 mesi e la predizione si dimostrò vera. I computer acquisirono nuove funzionalità e si ridussero di dimensioni, dando avvio a un gran numero di utilizzi e alla creazione di una vasta gamma di sistemi operativi.

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

Non esiste neppure una definizione universalmente accettata di che cosa faccia parte di un sistema operativo. Un punto di vista semplice è che esso comprenda tutto quello che il rivenditore fornisce quando gli si ordina “il sistema operativo”. Tuttavia queste funzioni variano molto da sistema a sistema. Alcuni sistemi usano meno di un megabyte di memoria e non possiedono neppure un *editor* a pieno schermo, mentre altri richiedono gigabyte di memoria e sono interamente basati su interfacce grafiche a finestre. Una definizione più comune è quella secondo cui il sistema operativo è il

solo programma che funziona sempre nel calcolatore, generalmente chiamato **kernel** (*nucleo*). (Oltre al kernel vi sono due tipi di programmi: i **programmi di sistema**, associati al sistema operativo, ma che non fanno necessariamente parte del kernel, e i **programmi applicativi**, che includono tutti i programmi non correlati al funzionamento del sistema).

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

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

1.2 Organizzazione di un sistema elaborativo

Prima di addentrarci nello studio dei dettagli di funzionamento di un sistema elaborativo è necessaria una conoscenza generale della struttura di un calcolatore. Lo scopo del presente paragrafo è una trattazione introduttiva dei diversi elementi di questa struttura. La trattazione riguarda principalmente l’organizzazione dei sistemi elaborativi, quindi chi già conosce questo argomento può saltare il paragrafo o limitarsi a un semplice sguardo.

1.2.1 Funzionamento di un sistema elaborativo

Un moderno calcolatore general-purpose è composto da una o più CPU e da un certo numero di controllori di dispositivi connessi attraverso un canale di comunicazione comune (*bus*) che permette l’accesso alla memoria condivisa dal sistema (Figura 1.2). Ciascuno di questi controllori si occupa di un particolare tipo di dispositivo fisico (per esempio, unità disco, dispositivi audio e unità video). La CPU e questi controllori possono operare in parallelo, contendendosi i cicli d’accesso alla memoria. La sincronizzazione degli accessi alla memoria è garantita dalla presenza di un controllore di memoria.

L’avviamento del sistema, conseguente all’accensione del calcolatore, così come il riavvio di un calcolatore già acceso, richiedono la presenza di uno specifico programma iniziale, di solito non troppo complesso, detto **programma d’avviamento**

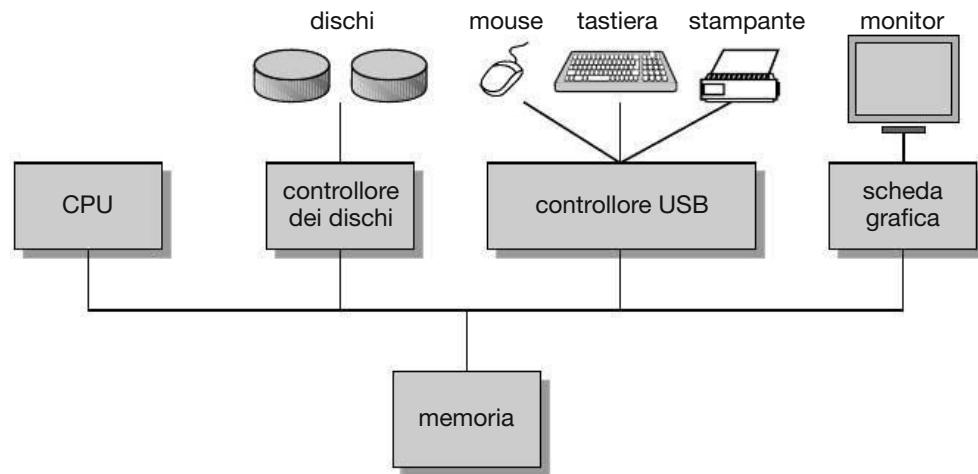


Figura 1.2 Moderno sistema elaborativo.

(*bootstrap program*), tipicamente memorizzato nell’hardware in memorie a sola lettura (*read only memory*, ROM) o in memorie programmabili e cancellabili elettricamente (EEPROM). Questo programma e le memorie in cui è contenuto sono noti con il termine generale di **firmware**. La sua funzione consiste nell’inizializzare i diversi componenti del sistema, dai registri della CPU ai controllori dei diversi dispositivi, fino al contenuto della memoria centrale. Il programma d’avviamento deve caricare nella memoria il sistema operativo e avviare l’esecuzione; a tale scopo individua e carica nella memoria il kernel del sistema operativo.

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

Un evento è di solito segnalato da un’interruzione dell’attuale sequenza d’esecuzione della CPU (*interrupt*), che può essere causata dall’hardware o da un programma. Nel primo caso i controllori dei dispositivi e altri elementi dell’architettura possono inviare alla CPU **segnali d’interruzione**, di solito attraverso il bus di sistema. Nel secondo caso il software genera un’interruzione eseguendo una speciale istruzione detta **chiamata di sistema** (*system call*, a volte chiamata anche *monitor call*).

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

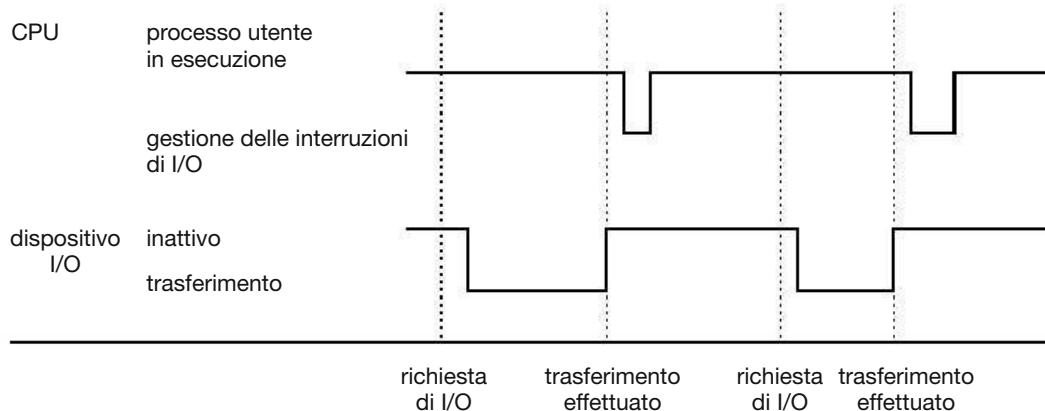


Figura 1.3 Diagramma temporale delle interruzioni per un singolo processo che emette dati.

Le interruzioni sono un elemento importante nell’architettura di un calcolatore. Ciascun tipo di calcolatore ha il proprio meccanismo di gestione delle interruzioni; ciò nonostante molte funzioni sono comuni. Ogni interruzione deve causare il trasferimento del controllo all’appropriata procedura di servizio. Il modo più semplice per gestire quest’operazione è quello di impiegare una procedura generale che esamini le informazioni associate all’interruzione, e a sua volta invochi la procedura di gestione dello specifico evento. D’altra parte, la gestione di un’interruzione deve essere

MEMORIA: DEFINIZIONI E NOTAZIONE

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

molto rapida, perciò, considerando che il numero delle possibili interruzioni è predefinito, si può usare una tabella di puntatori alle specifiche procedure. In questo modo, l'attivazione delle procedure di servizio delle interruzioni avviene in modo indiretto attraverso questa tabella, senza procedure intermedie. In genere questa tabella di puntatori è memorizzata nella memoria agli indirizzi più bassi (per esempio, le prime 100 locazioni). L'accesso a questa sequenza d'indirizzi, detta **vettore delle interruzioni**, avviene per mezzo di un indice, codificato nello stesso segnale d'interruzione, allo scopo di fornire l'indirizzo della procedura di servizio relativa all'evento segnalato dall'interruzione. Sistemi operativi molto differenti, come Windows e UNIX, usano questo stesso meccanismo di gestione delle interruzioni.

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

1.2.2 Struttura della memoria

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

I computer utilizzano anche altri tipi di memoria. Abbiamo già menzionato la ROM, memoria di sola lettura, e la EEPROM, memoria di sola lettura elettricamente cancellabile e programmabile. Dal momento che la ROM non può essere modificata, solo i programmi statici, come il programma d'avviamento, vi sono salvati. L'immutabilità della ROM è utile nelle cartucce per i videogiochi. Le EEPROM non possono essere riscritte di frequente; per questo contengono per lo più programmi statici. Sulle EEPROM degli smartphone, per esempio, sono memorizzati i programmi installati inizialmente dalla fabbrica.

Tutte le tipologie di memoria forniscono un vettore di byte. Ciascun byte possiede un proprio indirizzo. L'interazione avviene per mezzo di una sequenza di istruzioni **load** e **store** opportunamente indirizzate. L'istruzione **load** trasferisce il contenuto di un byte o una parola della memoria centrale in uno dei registri interni della CPU, mentre la **store** copia il contenuto di uno di questi registri nella locazione di memo-

ria specificata. Oltre agli accessi esplicativi, tramite `load` e `store`, la CPU preleva automaticamente dalla memoria centrale le istruzioni da eseguire.

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

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

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

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

Il dispositivo più comunemente impiegato a questo scopo è l'unità a **disco magnetico**, adoperato per la memorizzazione sia di programmi sia di dati. La maggior parte dei programmi (applicativi e di sistema) è mantenuta in un disco sino al momento del caricamento nella memoria. Molti programmi fanno uso del disco come sorgente e destinazione delle informazioni elaborate. Quindi, come si spiega nel Capitolo 10, una corretta gestione delle unità a disco è di fondamentale importanza per un sistema elaborativo.

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

L'ampio ventaglio dei sistemi di memorizzazione può essere ordinato secondo una scala gerarchica (Figura 1.4), sulla base della velocità e del costo. I livelli più alti sono più veloci, ma anche più dispendiosi. Andando verso il basso il costo per bit ge-

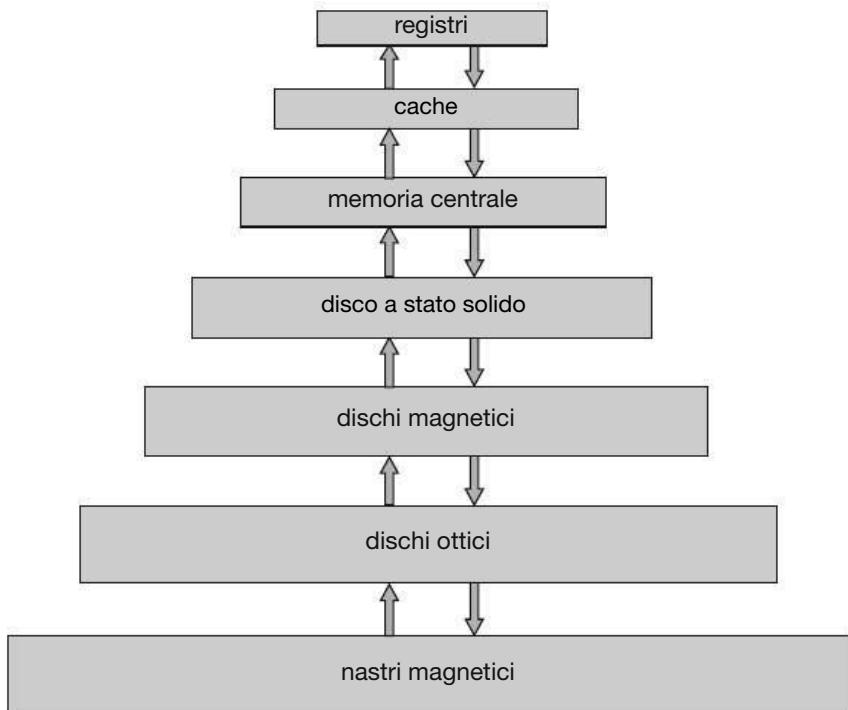


Figura 1.4 Scala gerarchica dei sistemi di memorizzazione.

neralmente diminuisce, mentre il tempo di accesso tende ad aumentare. Si tratta di un compromesso ragionevole: se un certo sistema di memorizzazione, a parità di condizioni, fosse più veloce e nel contempo meno costoso rispetto ad altri, verrebbe meno qualunque motivo per usare la soluzione più lenta e costosa. In effetti, molti sistemi di memorizzazione della prima ora, quali i nastri perforati e le memorie a nuclei magneticci, sono ormai relegati nei musei, essendo stati soppiantati dai nastri magnetici e dalle **memorie a semiconduttori**, più rapidi ed economici. I dispositivi nei quattro livelli superiori della Figura 1.4 possono essere materialmente costruiti con memorie a semiconduttori.

Oltre a differire per velocità e costo, i sistemi di memorizzazione si suddividono in volatili e non volatili. Come si è accennato, la **memoria volatile** perde i dati nel caso di interruzione dell’alimentazione. Qualora non si disponga di dispensosi sistemi di backup dell’alimentazione, basati su batterie o generatori elettrici, per preservare i dati è necessario salvarli su **memoria non volatile**. Nella gerarchia della Figura 1.4, i sistemi di memorizzazione al di sopra del disco a stato solido sono volatili, mentre quelli dal disco a stato solido in giù sono non volatili.

Esistono diverse varianti dei dischi a stato solido, ma in generale sono tutti più veloci dei dischi magnetici e sono non volatili. Una tipologia di disco a stato solido memorizza i dati in un grande array DRAM durante le normali operazioni, ma contiene anche, in posizione nascosta, un disco magnetico e una batteria di backup. Nel caso di un’interruzione della corrente elettrica, il controllore del disco a stato solido copia i dati dalla RAM sul disco magnetico, per poi eseguire l’operazione inversa al ripristino

della corrente elettrica. Un altro tipo di disco a stato solido è rappresentato dalla memoria flash, impiegata in vari prodotti, dalle macchine fotografiche agli smartphone e ai robot; la sua diffusione nei computer general-purpose è in costante aumento. La memoria flash è più lenta della DRAM, ma non necessita di alimentazione esterna. Un'altra possibilità per la memorizzazione non volatile è la NVRAM, vale a dire la DRAM provvista di batterie di backup. Questo tipo di memoria arriva a eguagliare la DRAM in velocità, ma la sua non volatilità ha durata limitata.

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

1.2.3 Struttura di I/O

La memoria è solo uno dei numerosi dispositivi di I/O di un elaboratore. Una percentuale cospicua del codice di un sistema operativo è dedicata alla gestione dell'I/O; ciò è dovuto sia alla sua importanza per il progetto di un sistema affidabile ed efficiente, sia alla differente tipologia dei dispositivi. Presentiamo adesso una panoramica generale dell'I/O.

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

Per avviare un'operazione di I/O, il driver del dispositivo carica gli appropriati registri all'interno del controllore, il quale, dal canto suo, esamina i contenuti di questi registri per determinare l'azione da intraprendere (per esempio "leggi un carattere dalla tastiera"). Il controllore comincia a trasferire i dati dal dispositivo al proprio buffer. A trasferimento completato, il controllore informa il driver, tramite un'interruzione, di avere terminato l'operazione. Il driver passa quindi il controllo al sistema operativo, restituendo i dati (o un puntatore a essi) se l'operazione è di lettura; per altre operazioni, il driver restituisce delle informazioni di stato.

Questa forma di I/O guidato dalle interruzioni è adatto al trasferimento di piccole quantità di dati, ma in caso di trasferimenti massicci può generare un pesante sovraccarico (*overhead*); si pensi, per esempio, all'I/O da e verso il disco. Per risolvere questo problema si utilizza la tecnica dell'**accesso diretto alla memoria (DMA)**. Una volta impostati i buffer, i puntatori e i contatori necessari al dispositivo di I/O, il con-

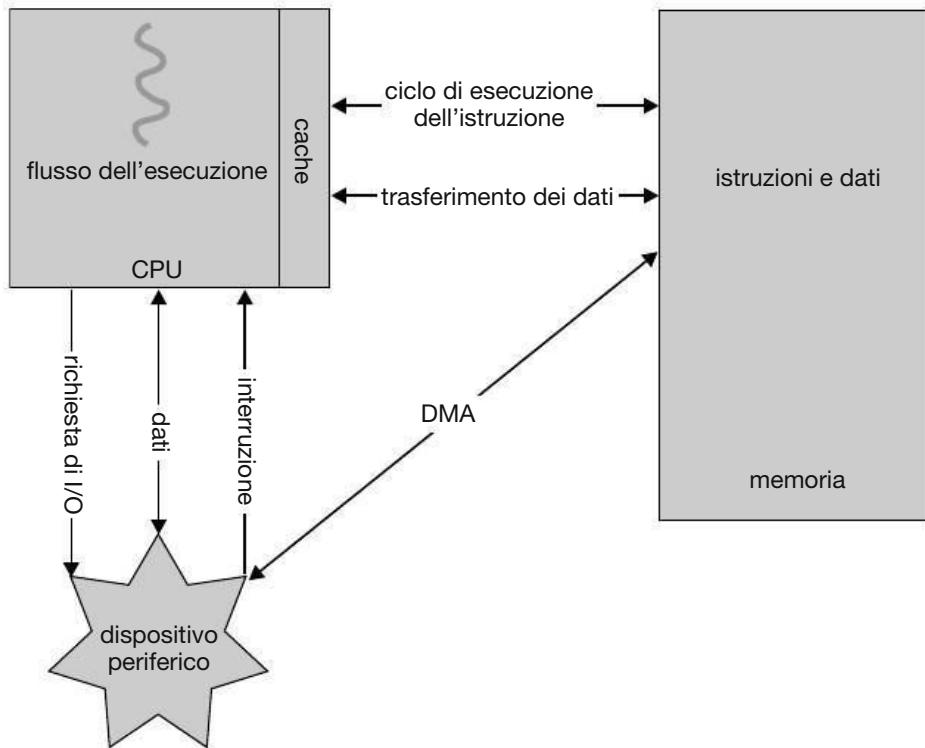


Figura 1.5 Funzionamento di un moderno sistema operativo.

trollore trasferisce un intero blocco di dati dal proprio buffer direttamente nella memoria centrale, o viceversa, senza alcun intervento da parte della CPU. In questo modo l'operazione richiede una sola interruzione per ogni blocco di dati trasferito, piuttosto che per ogni byte, come avviene nel caso dei dispositivi più lenti. Così, mentre il controllore del dispositivo effettua le operazioni descritte, la CPU rimane libera di occuparsi di altri compiti.

Alcuni sistemi ad alte prestazioni hanno abbandonato la configurazione basata sul bus per adottare un'architettura incentrata sugli switch, in cui più dispositivi fisici possono interagire con varie parti del sistema concorrentemente, piuttosto che contendersi un unico bus condiviso. In questo caso, il DMA risulta ancora più efficace. La Figura 1.5 mostra l'interazione di tutti i componenti di un elaboratore.

1.3 Architettura degli elaboratori

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

1.3.1 Sistemi monoprocessoress

Fino a pochi anni fa molti sistemi utilizzavano un solo processore. Un sistema monoprocessoress è dotato di una CPU principale in grado di eseguire un insieme di istruzioni di natura generale, comprese quelle necessarie ai processi utenti. Quasi tutti i sistemi monoprocessoress, inoltre, possiedono altri processori specializzati, deputati a compiti particolari. Essi possono assumere la forma di processori specifici dedicati a un dispositivo, quali i controllori del disco, della tastiera o del video; oppure, nel caso dei mainframe, possono essere processori di tipo più generale, come i processori di I/O che spostano dati da un componente a un altro del sistema.

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

1.3.2 Sistemi multiprocessoress

Negli ultimi anni i **sistemi multiprocessoress** (conosciuti anche come **sistemi paralleli** o **sistemi multicore**) hanno iniziato a dominare il mondo della computazione. Questo tipo di sistemi dispone di più unità d'elaborazione in stretta comunicazione, che dividono il bus e talvolta il clock di sistema, la memoria e i dispositivi periferici. I sistemi multiprocessoress vennero inizialmente impiegati in macchine server e si diffusero poi su desktop e computer portatili. Negli anni più recenti i multiprocessoressi vengono usati anche su dispositivi mobili come smartphone e tablet.

I sistemi multiprocessoress hanno tre vantaggi principali.

1. **Maggiore capacità elaborativa (*throughput*)**. Aumentando il numero di unità d'elaborazione è possibile svolgere un lavoro maggiore in meno tempo. Con n unità d'elaborazione la velocità non aumenta tuttavia di n volte, ma in misura minore. Infatti, se più unità d'elaborazione collaborano nell'esecuzione di un compito, è necessario un certo sovraccarico (*overhead*) per garantire che tutti i componenti funzionino correttamente. Questo overhead, unito alla contesa per le risorse condivise, riduce il guadagno atteso dalla disponibilità di più unità d'elaborazione: analogamente un gruppo di n programmati che lavorano insieme non produce n volte quanto produrrebbe un solo programmatore.

2. **Economia di scala.** I sistemi multiprocessore possono costare meno rispetto a più sistemi monoprocessoressi, poiché vengono condivisi periferiche, memorie di massa e sistemi di alimentazione. Se più programmi devono operare sullo stesso insieme di dati, è economicamente più conveniente mantenerli in dischi condivisi da tutte le unità d'elaborazione, piuttosto che avere più calcolatori con i rispettivi dischi locali e più copie degli stessi dati.
3. **Incremento dell'affidabilità.** Se le funzioni si possono distribuire adeguatamente tra più unità d'elaborazione, un guasto di una di esse non blocca il sistema, semplicemente lo rallenta; ciascuna delle unità d'elaborazione rimanenti assume su di sé una parte del lavoro che svolgeva l'unità d'elaborazione guasta; l'intero sistema non si ferma, ma funziona a una velocità ridotta.

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

I sistemi multiprocessore attualmente in uso sono di due tipi. Alcuni impiegano la **multielaborazione asimmetrica** (*asymmetric multiprocessing*), in cui a ogni processore si assegna un compito specifico. Un processore principale controlla il sistema, gli altri attendono istruzioni dal processore principale oppure hanno compiti predefiniti. Questo schema definisce una relazione gerarchica; il processore principale organizza e assegna il lavoro ai processori secondari.

I sistemi più comuni utilizzano la **multielaborazione simmetrica** (*symmetric multiprocessing*, SMP), in cui ogni processore è abilitato all'esecuzione di tutte le operazioni del sistema. La tecnica SMP pone tutti i processori su un piano di parità; tra essi non vi è subordinazione gerarchica. La Figura 1.6 illustra una tipica architettura SMP. Si noti che ogni processore ha il proprio set di registri e una cache privata (locale). Un esempio di sistema SMP è AIX, una versione commerciale di UNIX realizzata da IBM. Tale sistema può impiegare dozzine di processori. Il vantaggio offerto da questo modello è che molti processi sono eseguibili contemporaneamente (n processi se si hanno n CPU) senza causare un rilevante calo delle prestazioni. Per essere certi che i dati raggiungano le unità d'elaborazione giuste occorre però controllare con molta

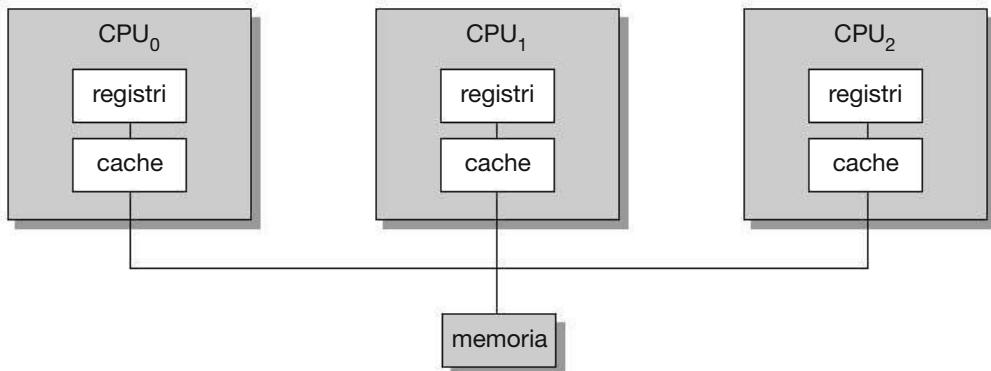


Figura 1.6 Architettura per la multielaborazione simmetrica.

attenzione le operazioni di I/O. Inoltre, poiché le unità d'elaborazione sono separate, una potrebbe essere inattiva mentre un'altra è sovraccarica, e ciò determinerebbe un'inefficienza evitabile se le unità d'elaborazione condividessero alcune strutture dati. Un sistema multiprocessore di questo tipo permetterebbe infatti di condividere dinamicamente processi e risorse (per esempio la memoria) tra i vari processori, riducendo la varianza tra di essi. Come si vedrà nel Capitolo 5, un sistema di questo tipo deve essere scritto con molta cura. Tutti i sistemi operativi moderni (tra i quali Windows, Mac OS X e Linux) supportano la multielaborazione simmetrica.

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

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

Una tendenza recente nella progettazione della CPU è raggruppare diverse unità di calcolo (*core*) in un singolo chip. Tali CPU sono dette **multicore**. Questi sistemi possono essere più efficienti rispetto a più chip dotati di una singola unità di calcolo, perché la comunicazione all'interno di un singolo chip è più veloce rispetto a quella

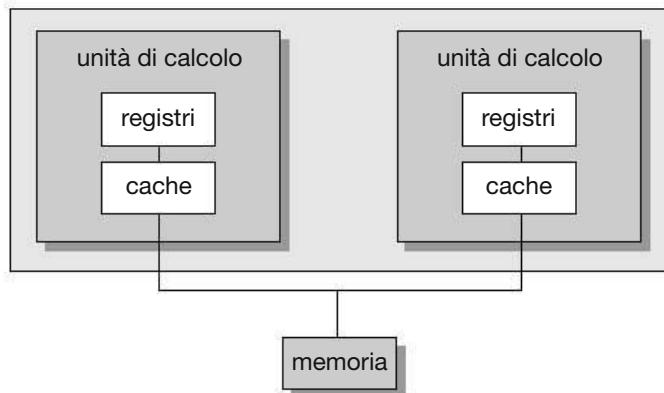


Figura 1.7 Architettura dual-core, con due unità sullo stesso chip.

tra un chip e un altro. Inoltre, un chip dotato di diversi core usa molta meno potenza di diversi chip con un singolo core.

È importante notare che i sistemi multicore sono sistemi multiprocessore, ma non è sempre vero il contrario, come vedremo nel prossimo paragrafo. In questo testo, se non altrimenti specificato, useremo il termine *sistemi multicore*, più moderno, per parlare di sistemi multiprocessore, escludendo così alcune tipologie di sistemi.

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

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

1.3.3 Cluster di elaboratori

I **cluster di elaboratori** (*clustered systems*) o **cluster** sono un altro tipo di sistemi multiprocessore, basati sull'uso congiunto di più CPU, ma differiscono dai sistemi multiprocessore descritti nel paragrafo precedente per il fatto che sono composti di due o più calcolatori completi – detti nodi – collegati tra loro. Sistemi di questo tipo sono detti **debolmente accoppiati** (*loosely coupled*). Ogni nodo può essere costituito da un sistema monoprocesso o da un sistema multicore. Va osservato che per tali

sistemi non esiste una definizione precisa; c'è un dibattito aperto tra i fornitori delle varie offerte commerciali su tale definizione e sul perché una soluzione sia migliore di un'altra. La definizione generalmente accettata è che si tratta di calcolatori che condividono la memoria di massa, connessi per mezzo di una rete locale (LAN), come descritto nel Capitolo 17, o da connessioni più veloci come InfiniBand.

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

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

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

Altre forme di clustering sono i cluster paralleli e quelli di sistemi connessi attraverso reti geografiche (WAN), come descritto nel Capitolo 17. I primi permettono a più calcolatori di accedere agli stessi dati nella memoria di massa condivisa. Poiché la maggior parte dei sistemi operativi non supporta quest'accesso simultaneo ai dati da parte di più calcolatori, si ricorre a programmi specifici e particolari versioni delle applicazioni. Per esempio, l'Oracle Real Application Cluster è una versione del sistema di gestione delle basi di dati Oracle, progettata per funzionare su cluster paralleli. Ogni calcolatore esegue l'applicazione Oracle e uno strato software controlla l'accesso ai dischi condivisi, in questo modo ogni calcolatore del sistema ha accesso alla base di dati. Per ottenere questo accesso condiviso ai dati, il sistema deve anche prevedere il controllo dell'accesso e la mutua esclusione, in modo da evitare sul nascere i conflitti tra operazioni. Questa funzione, detta **gestione distribuita dei lock** (*distributed lock manager*, DLM), è fornita da alcune tecnologie di clustering.

CLUSTER “BEOWULF”

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

La tecnologia in quest’ambito sta attraversando una fase di rapida evoluzione. Alcuni prodotti di questo genere ospitano dozzine di sistemi in un solo cluster, e possono accoppare in un unico cluster nodi distanti chilometri. Tali progressi si devono in buona misura a **reti di storage** (*storage-area network*, SAN), illustrate nel Paragrafo 10.3.3, che permettono a molti sistemi di accedere a un’unica area di memorizzazione secondaria. Se le applicazioni e i relativi dati sono memorizzati in una SAN, il software del cluster può smistare l’applicazione verso una qualunque delle macchine che vi hanno accesso. Qualora venga meno una di loro, può subentrare qualsiasi altra macchina. Se una base di dati è implementata su un cluster, decine di macchine possono condividerne il contenuto, con notevole aumento delle prestazioni e dell’affidabilità. La Figura 1.8 rappresenta la struttura generale di un cluster.

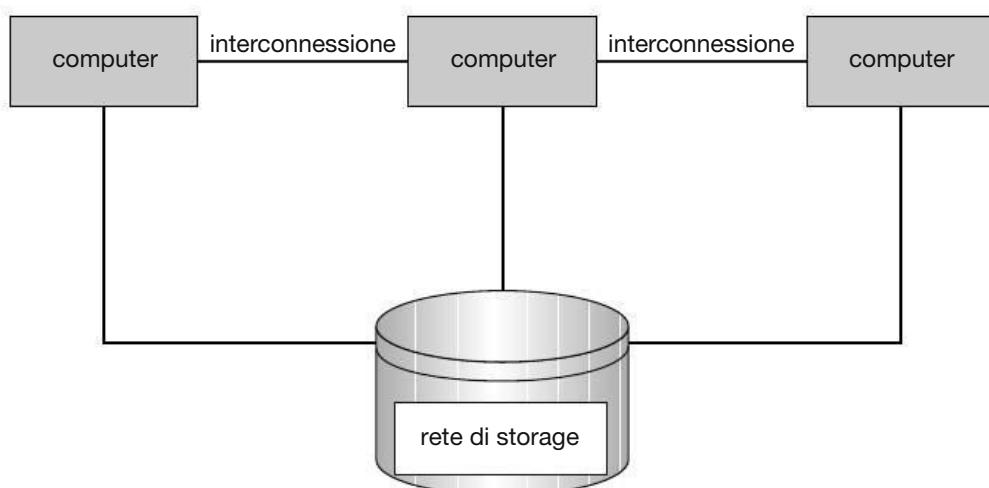


Figura 1.8 Struttura generale di un cluster.

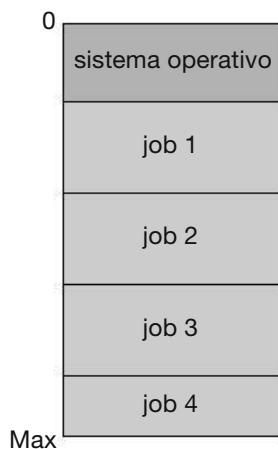


Figura 1.9 Configurazione della memoria per un sistema con multiprogrammazione.

1.4 Struttura del sistema operativo

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

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

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

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

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

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

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

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

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

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

Il time sharing e la multiprogrammazione richiedono la contemporanea presenza di diversi job in memoria. Se alcuni job sono pronti per il trasferimento in memoria centrale, ma lo spazio disponibile non è sufficiente per accoglierli tutti, il sistema deve fare una selezione; questa scelta, illustrata nel Capitolo 6, si chiama **job scheduling**, ossia *pianificazione dei lavori*. Quando il sistema operativo seleziona un job dall’insieme dei lavori, lo carica in memoria perché sia eseguito. La coesistenza di un certo numero di programmi in memoria nello stesso lasso di tempo richiede una qualche forma di gestione della memoria, argomento esposto nei Capitoli 8 e 9.

Inoltre, l'esistenza di diversi processi pronti per l'esecuzione nello stesso istante impone al sistema di scegliere quale processo viene eseguito per primo: i problemi inerenti a questa scelta, ovvero lo **scheduling della CPU**, sono descritti nel Capitolo 6. Infine, l'esecuzione concorrente di più processi rende necessario limitare le loro interferenze reciproche in ogni aspetto del funzionamento del sistema, compresi lo scheduling dei processi e la gestione del disco e della memoria. Tali problematiche sono affrontate di volta in volta nel libro.

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

I sistemi time sharing devono inoltre fornire un *file system* (Capitoli 11 e 12) residente in un insieme di dischi, i quali a loro volta necessitano di una gestione (Capitolo 10). Inoltre questi sistemi dispongono di meccanismi per la protezione delle risorse rispetto a utilizzi non autorizzati (Capitolo 14). Per assicurare che tale esecuzione sia disciplinata il sistema deve fornire meccanismi per la comunicazione e sincronizzazione dei processi (Capitolo 5) e garantire che i processi non si blocchino in una situazione di stallo, in un'indefinita attesa reciproca (Capitolo 7).

1.5 Attività del sistema operativo

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

Dal momento che il sistema operativo e gli utenti condividono risorse hardware e software dell'elaboratore, è necessario assicurarsi che un errore provocato da un programma utente non possa arrecare danno ad altri programmi. In un ambiente condi-

viso, infatti, l'errore di un solo programma potrebbe avere un effetto negativo su molti altri processi. Se, per esempio, un processo rimane bloccato in un ciclo infinito, potrebbe essere impedito il corretto funzionamento di molti altri processi. I sistemi multiprogrammati, inoltre, sono esposti a rischi più sottili; si pensi a un programma viziato da errori, capace di modificare interamente un altro programma, i suoi dati e persino lo stesso sistema operativo.

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

1.5.1 Dupliche modalità di funzionamento

Per garantire il corretto funzionamento del sistema è necessario distinguere tra l'esecuzione di codice del sistema operativo e di codice utente. Il metodo seguito dalla maggioranza dei sistemi operativi consiste nel disporre di un supporto hardware che consente di distinguere differenti modalità di funzionamento.

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

All'avviamento del sistema, il bit è posto in modalità di sistema. Si carica il sistema operativo che provvede all'esecuzione delle applicazioni utenti in modalità utente. Ogni volta che si verifica un'interruzione o un'eccezione, l'hardware passa dalla mo-

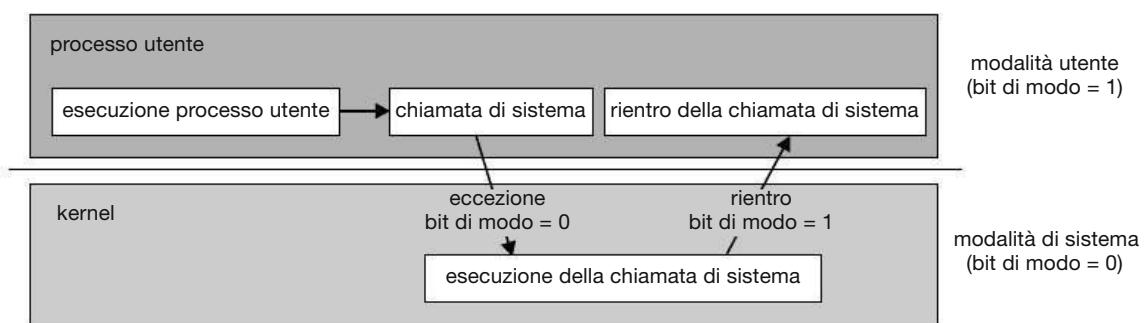


Figura 1.10 Transizione da modalità utente a modalità di sistema.

dalità utente a quella di sistema, cioè pone a 0 il bit di modo. Perciò, quando il sistema operativo riprende il controllo del calcolatore, esso si trova in modalità di sistema. Prima di passare il controllo al programma utente, il sistema ripristina la modalità utente riportando a 1 il valore del bit.

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

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

Il concetto di modalità può essere esteso oltre le due possibilità (in questo caso la CPU utilizza più di un bit per impostare e testare la modalità). Le CPU che supportano la virtualizzazione (si veda il Paragrafo 16.1) hanno spesso una modalità distinta per indicare che il **gestore della macchina virtuale (VMM)** – e il software per la gestione della virtualizzazione - hanno il controllo del sistema. In questa modalità il VMM ha più privilegi dei processi utente, ma meno privilegi del kernel. Questo livello di privilegio è necessario per creare e gestire le macchine virtuali, modificando lo stato della CPU. Talvolta sono utilizzate modalità distinte anche dalle varie componenti del kernel. Va osservato che i progettisti di CPU possono utilizzare altri metodi per la gestione dei privilegi. La famiglia di CPU Intel 64, per esempio, supporta quattro **livelli di privilegio** e ha il supporto per la virtualizzazione senza però avere una modalità distinta per gestirla.

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

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

Quando un programma utente esegue una chiamata di sistema, questa è solitamente gestita dalla CPU come un'interruzione software. Il controllo passa, tramite il vettore

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

Se la CPU non supporta il dual-mode il sistema operativo può andare incontro a serie limitazioni. Il sistema MS-DOS per esempio è stato sviluppato per l'architettura 8088 Intel priva del bit di modo e, quindi, del dual-mode. Un programma utente potrebbe cancellare il sistema operativo scrivendo nuove informazioni nelle locazioni in cui questo è memorizzato, e più programmi potrebbero scrivere contemporaneamente nello stesso dispositivo, con risultati potenzialmente disastrosi. Le versioni più moderne delle CPU Intel sono dotate della duplice modalità di funzionamento. I più recenti sistemi operativi sviluppati per tali architetture, come Microsoft Windows 7, Unix e Linux traggono vantaggio da questa caratteristica e garantiscono una maggiore protezione del sistema operativo.

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

1.5.2 Timer

Occorre assicurare che il sistema operativo mantenga il controllo della CPU, che consiste nell'impedire che un programma utente entri in un ciclo infinito o non richieda servizi del sistema senza più restituire il controllo al sistema operativo. A tale scopo si può usare un **timer**, programmabile affinché invii un segnale d'interruzione alla CPU a intervalli di tempo specificati, che possono essere fissi (per esempio, di 1/60 di secondo) o variabili (per esempio, da un millisecondo a un secondo). Un **timer**

variabile di solito si realizza mediante un clock a frequenza fissa e un contatore. Il sistema operativo assegna un valore al contatore, che si decrementa a ogni impulso e quando raggiunge il valore 0 si genera un segnale d'interruzione. Per esempio, un contatore di 10 bit con un clock con periodo di 1 millisecondo consente la generazione di interruzioni a intervalli compresi tra 1 e 1024 millisecondi, con incrementi di 1 millisecondo.

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

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

1.6 Gestione dei processi

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

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

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

Il processo è l’unità di lavoro di un sistema. Un sistema è costituito di un gruppo di processi, alcuni dei quali del sistema operativo (eseguono codice di sistema), mentre altri sono processi utenti (eseguono codice utente). Tutti questi processi si possono potenzialmente eseguire in modo concorrente, per esempio avvicendandosi nell’uso di una sigla CPU.

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

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

Le tecniche di gestione dei processi sono trattate nei Capitoli dal 3 al 5.

1.7 Gestione della memoria

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

Per eseguire un programma è necessario che questo sia associato a indirizzi assoluti e sia caricato nella memoria. Durante l’esecuzione del programma, questo accede

alle proprie istruzioni e ai dati provenienti dalla memoria, generando i suddetti indirizzi assoluti. Quando il programma termina, si rende disponibile il suo spazio di memoria; a questo punto si può caricare ed eseguire il programma successivo.

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

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

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

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

1.8 Gestione della memoria di massa

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

1.8.1 Gestione dei file

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

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

Il sistema operativo realizza il concetto astratto di file gestendo i mezzi di memoria di massa, come nastri e dischi, e i dispositivi che li controllano. I file sono generalmente organizzati in directory, che ne facilitano l'uso. Infine, se più utenti hanno ac-

cesso ai file, si potrebbe voler controllare chi ha la possibilità di accedervi e in che modo (per esempio, lettura, scrittura, aggiunta).

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

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

Le tecniche di gestione dei file sono trattate nei Capitoli 11 e 12.

1.8.2 Gestione della memoria di massa

Come si è visto, giacché la memoria centrale è troppo piccola per contenere tutti i dati e tutti i programmi, e il suo contenuto va perduto se il sistema si spegne, il calcolatore deve disporre di una memoria secondaria a sostegno della memoria centrale. La maggior parte dei moderni sistemi elaborativi impiega i dischi come principale mezzo di memorizzazione secondaria, sia per i programmi sia per i dati. I dischi contengono la maggior parte dei programmi, compresi i compilatori, gli assemblatori, i word processor e gli editor. Questi programmi rimangono nel disco fino al momento del caricamento in memoria e si servono del disco sia come sorgente sia come destinazione delle loro elaborazioni. Per tale ragione è fondamentale che la registrazione nei dischi sia gestita correttamente. Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei dischi:

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

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

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

La memoria terziaria ha scarso impatto sulle prestazioni del sistema, ma deve pur essere gestita. Alcuni sistemi si assumono tale onere direttamente, mentre altri lo delegano a programmi applicativi. Tra le funzioni che i sistemi possono fornire citiamo

l'installazione e la rimozione dei media dei dispositivi, l'allocazione dei dispositivi ai processi che ne richiedano l'uso esclusivo, e il trasferimento alla memoria terziaria dei dati provenienti da quella secondaria.

Le tecniche di gestione della memoria secondaria e terziaria saranno approfondite nel Capitolo 10.

1.8.3 Cache

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

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

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

La memoria centrale si può considerare una cache per la memoria secondaria, giacché i dati in essa contenuti devono essere riportati nella memoria centrale per poter essere usati e qui devono risiedere prima di essere trasferiti nella memoria secondaria. I dati del file system, che risiedono in modo permanente nella memoria secondaria, possono apparire a diversi livelli della gerarchia di memoria. A livello più alto, il sistema operativo può mantenere una cache dei dati del file system in memoria centrale. Anche i dischi a stato solido si possono impiegare come veloci sistemi di memoria cui si accede tramite l'interfaccia del file system. La memorizzazione secondaria si effettua generalmente in dischi magnetici; copie di riserva del loro contenuto spesso si registrano in nastri magnetici o dischi rimovibili per prevenirne la perdita.

Livello	1	2	3	4	5
Nome	registri	cache	memoria centrale	disco a stato solido	disco magnetico
Dimensione tipica	< 1 KB	< 16 MB	< 64 GB	< 1 TB	< 10 TB
Tecnologia	memoria dedicata con porte multiple (CMOS)	CMOS SRAM (on-chip o off-chip)	CMOS DRAM	memoria flash	disco magnetico
Tempo d'accesso (ns)	0,25 – 0,5	0,5 – 25	80 – 250	25.000-50.000	5.000,000
Aampiezza di banda (MB/s)	20.000 – 100.000	5000 – 10.000	1000 – 5000	500	20 – 150
Gestito da	compilatore	hardware	sistema operativo	sistema operativo	sistema operativo
Supportato da	cache	memoria centrale	disco	disco	disco o nastro

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

in caso di guasto dei dischi magnetici. Alcuni sistemi, per ridurre i costi di memorizzazione, archiviano automaticamente i vecchi file di dati trasferendoli dalla memoria secondaria alla memoria terziaria, costituita per esempio da *juke-box* di nastri (Capitolo 10).

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

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

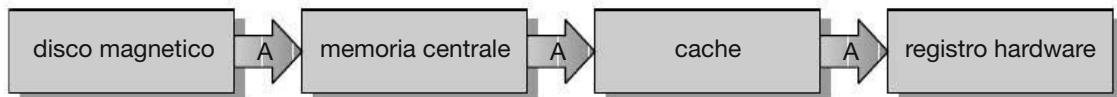


Figura 1.12 Migrazione di un intero A da un disco a un registro.

riabile. Le diverse copie di A avranno lo stesso valore solamente dopo che il nuovo valore sarà stato riportato dal registro interno della CPU alla copia di A residente nel disco.

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

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

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

1.8.4 Sistemi di I/O

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

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

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

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

1.9 Protezione e sicurezza

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

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

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

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

dell'allarmante incremento di incidenti legati alla sicurezza, le problematiche della sicurezza del sistema operativo costituiscono un settore che vede crescere rapidamente la ricerca e la sperimentazione. La sicurezza è analizzata nel Capitolo 15.

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

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

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

1.10 Strutture dati del kernel

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

1.10.1 Liste, stack e code

Un array è una semplice struttura dati in cui ogni elemento è direttamente accessibile. La memoria principale, per esempio, è costruita come un array. Quando un dato memorizzato è più grande di un byte possono essere allocati più byte per lo stesso dato e il dato sarà accessibile utilizzando il suo numero progressivo moltiplicato per la sua dimensione. Come procedere però in caso di dati di dimensione variabile? Come si

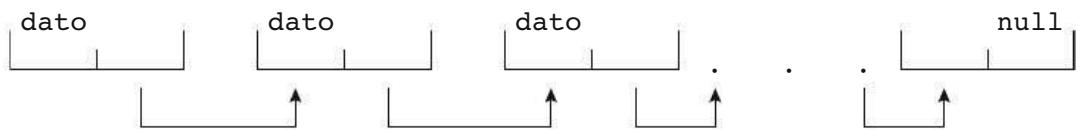


Figura 1.13 Lista semplicemente concatenata.

può rimuovere un elemento quando si deve preservare la posizione relativa degli elementi rimanenti? In queste situazioni gli array devono lasciare il posto ad altre strutture dati.

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

- In una lista semplicemente concatenata ogni elemento punta all’elemento successivo, come mostrato nella Figura 1.13.
- In una lista doppiamente concatenata ogni elemento contiene riferimenti sia al suo successore sia al suo predecessore, come mostrato nella Figura 1.14.
- In una lista circolare l’ultimo elemento punta al primo elemento piuttosto che a un valore null, come è mostrato nella Figura 1.15.

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

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

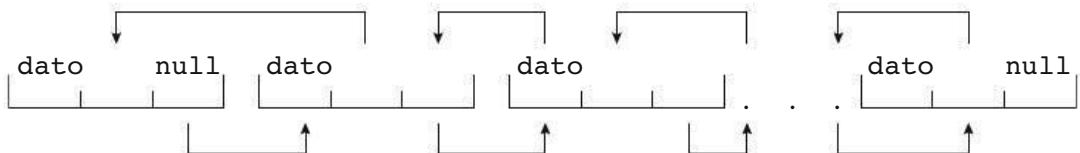


Figura 1.14 Lista doppiamente concatenata.

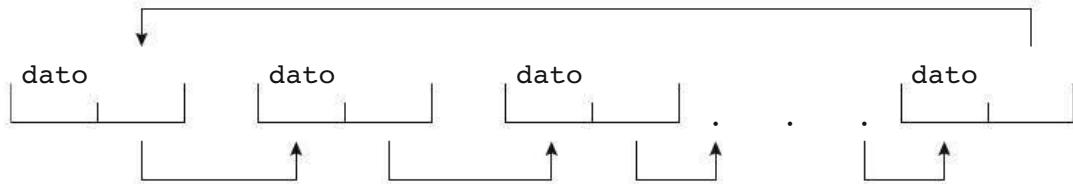


Figura 1.15 Lista circolare.

locali e l’indirizzo di ritorno vengono inseriti, mediante operazioni di push, nello stack. Al ritorno dalla funzione gli elementi inseriti nello stack vengono rimossi mediante operazioni di pop.

Una **coda**, a differenza dello stack, è una struttura dati ordinata sequenzialmente che adotta una politica di accesso di tipo FIFO (*first in first out*): gli elementi vengono rimossi da una coda nell’ordine in cui sono stati inseriti. Troviamo diversi esempi di code nella vita quotidiana, tra cui le persone in fila alla cassa di un negozio o le auto al semaforo in attesa del verde. Le code sono anche piuttosto comuni nei sistemi operativi, per esempio nel caso dei documenti inviati a una stampante che vengono di solito processati nell’ordine in cui sono stati ricevuti. Come vedremo nel Capitolo 6 i task in attesa di essere eseguiti su una CPU sono spesso organizzati in code.

1.10.2 Alberi

Un **albero** è una struttura dati utilizzabile per rappresentare i dati in maniera gerarchica. Gli elementi di un albero sono strutturati secondo una relazione padre-figlio. In un generico albero un padre può avere un numero illimitato di figli, mentre in un **albero binario** un padre ha al massimo due figli, chiamati figlio sinistro e figlio destro. Un **albero di ricerca binario** soddisfa un requisito aggiuntivo: i figli sono ordinati in modo che figlio-sinistro \leq figlio-destro. La Figura 1.16 rappresenta un esempio di albero di ricerca binario. Quando si cerca un elemento in un albero binario

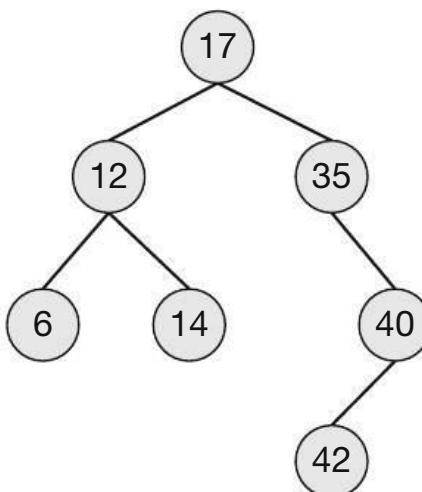


Figura 1.16 Albero binario di ricerca.

di ricerca le prestazioni nel caso peggiore sono $O(n)$ (lasciamo al lettore di capire quando si verifica il caso peggiore). Per rimediare a questa situazione possiamo utilizzare un algoritmo per la creazione di un albero binario di ricerca bilanciato. Un tale albero con n elementi ha al più $\log n$ livelli, assicurando così prestazioni di $O(\log n)$ nel caso peggiore. Come vedremo nel Paragrafo 6.7.1 Linux utilizza un albero binario di ricerca bilanciato nel suo algoritmo di scheduling della CPU.

1.10.3 Funzioni e mappe hash

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

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

Una funzione hash può essere utilizzata per creare una **tabella hash** (o **mappa hash**) che associa (o mappi) coppie [chiave:valore]. Possiamo per esempio mappare la chiave *operativo* al valore *sistema*. Una volta stabilita la mappa, è possibile applicare la funzione hash alla chiave per ottenere dalla tabella hash il valore (Figura 1.17). Si supponga, per esempio, che il valore di una username sia mappato in una password. Per l'autenticazione si procede come segue: l'utente inserisce username e password; viene applicata la funzione hash alla username, in modo da recuperare la password dalla tabella; la password viene confrontata con quella inserita dall'utente per l'autenticazione.

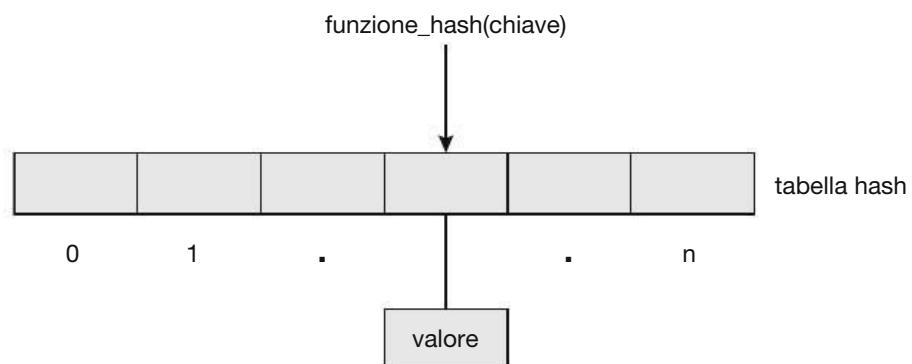


Figura 1.17 Tabella hash.

LE STRUTTURE DATI DEL KERNEL DI LINUX

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

1.10.4 Bitmap

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

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

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

1.11 Ambienti d'elaborazione

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

1.11.1 Elaborazione tradizionale

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

ancora diffusi in molte aziende, con ancora meno possibilità d’accesso remoto e di portabilità.

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

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

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

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

1.11.2 Mobile computing

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

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

Per offrire accesso ai servizi on-line i dispositivi mobili utilizzano di solito lo standard wireless 802.11 o le reti cellulari. La capacità di memoria e la velocità del processore dei dispositivi mobili sono limitate rispetto a quelle dei PC. Uno smartphone o un tablet possono avere 64 GB di memoria secondaria, mentre non è insolito trovare anche 1 TB di spazio su computer desktop. Analogamente, a causa dell'importanza di mantenere bassi i consumi, i dispositivi mobili utilizzano spesso processori più piccoli, più lenti e dotati di meno core rispetto ai processori desktop e portatili.

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

1.11.3 Sistemi distribuiti

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

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

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

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

Taluni sistemi operativi hanno interpretato l’idea delle reti e dei sistemi distribuiti secondo un’ottica più vasta rispetto alla semplice fornitura della connettività di rete. Un **sistema operativo di rete** è dotato di caratteristiche quali la condivisione dei file attraverso la rete e di un modello di comunicazione per i processi attivi su elaboratori

diversi, che possono così scambiarsi messaggi. Un computer che funziona con un sistema operativo di rete agisce in autonomia rispetto a tutti gli altri computer della rete, benché sia consci della presenza della rete e, dunque, possa interagire con gli altri computer che ne fanno parte. Un sistema operativo distribuito offre un ambiente meno autonomo: la stretta comunicazione che si instaura tra i diversi elaboratori partecipanti tende a dare l'impressione che vi sia un solo sistema incaricato di controllare la rete.

Il Capitolo 17 è dedicato alle reti di calcolatori e ai sistemi distribuiti.

1.11.4 Computazione client-server

Di pari passo all'aumento di velocità, potenza ed economicità dei PC, i progettisti hanno abbandonato il modello di architettura centralizzata per i sistemi. I PC e i dispositivi mobili stanno prendendo il posto dei terminali connessi ai grandi sistemi centralizzati. Analogamente, accade sempre più spesso che i PC gestiscano in proprio, spesso per mezzo di un'interfaccia web, la funzionalità d'interfaccia con l'utente, di cui prima si occupavano i sistemi a livello centrale. Ecco perché molti sistemi odierni fungono da sistemi server per le richieste che ricevono dai sistemi client. Questa particolare variante dei sistemi distribuiti, che prende il nome di sistema client-server, ha la struttura generale rappresentata nella Figura 1.18.

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

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

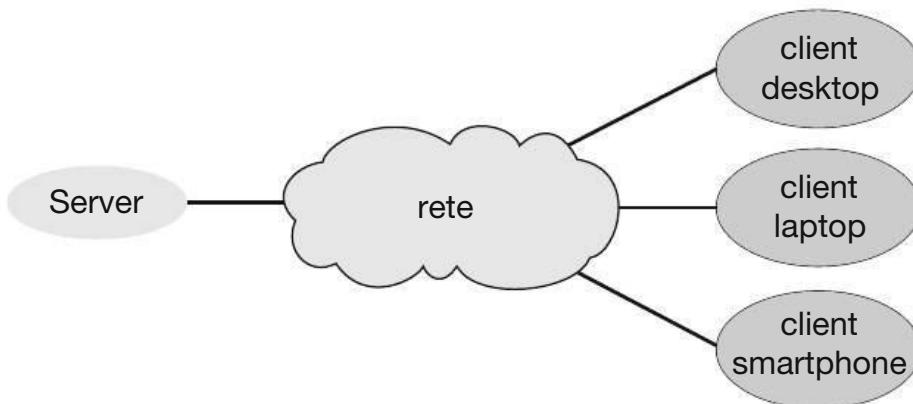


Figura 1.18 Struttura generale di un sistema client-server.

1.11.5 Elaborazione peer-to-peer

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

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

- Al momento di unirsi a una rete, un nodo iscrive il proprio servizio in un registro centralizzato di consultazione della rete. Quando un nodo vuole ottenere un servizio, esso contatta in via preliminare il registro centralizzato, per verificare quali nodi lo forniscono. Il resto della comunicazione ha luogo tra il client e il fornitore del servizio.
- Uno schema alternativo non utilizza alcun servizio di consultazione centralizzato. Un nodo che operi come client deve innanzitutto accertare quale nodo fornisca il servizio desiderato, inoltrando la propria richiesta di servizio a tutti gli altri nodi della rete. I nodi che possono fornire tale servizio rispondono al nodo da cui è partita la richiesta. Questa procedura richiede un *protocollo di scoperta*, che deve permettere ai nodi di scoprire i servizi forniti dagli altri nodi della rete. La Figura 1.19 illustra un tale scenario.

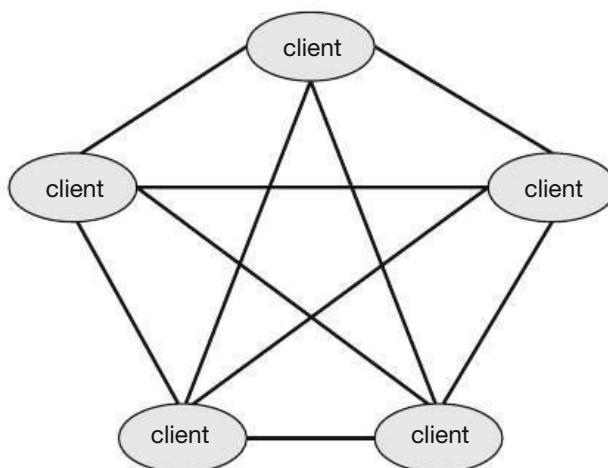


Figura 1.19 Sistema peer-to-peer senza servizi centralizzati.

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

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

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

1.11.6 Virtualizzazione

La virtualizzazione è una tecnica che permette di eseguire un sistema operativo come applicazione all'interno di un altro sistema operativo. Può sembrare, a prima vista, che ci siano poche ragioni per utilizzare questa tecnologia, ma il mercato della virtualizzazione è grande e in continua crescita, a testimonianza dell'utilità e dell'importanza di questa tecnologia.

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

Un classico esempio di emulazione si ha quando un linguaggio non viene compilato nel codice nativo, ma viene eseguito così com'è oppure tradotto in una forma intermedia. Questo meccanismo è noto come **interpretazione**. Alcuni linguaggi, come il BASIC, possono essere sia compilati sia interpretati, Java viene invece interpretato.

L'interpretazione è una forma di emulazione, in quanto il codice in linguaggio di alto livello viene tradotto nelle istruzioni native della CPU che emulano una macchina virtuale teorica su cui il linguaggio può essere eseguito nativamente. Possiamo quindi eseguire programmi Java su "macchine virtuali" Java, ma tecnicamente queste macchine virtuali sono emulatori Java.

Con la **virtualizzazione** invece un sistema operativo compilato per una particolare architettura viene eseguito all'interno di un altro sistema operativo progettato per la stessa CPU. La virtualizzazione fu inizialmente utilizzata su mainframe IBM come metodo per permettere a diversi utenti di eseguire task in maniera concorrente. L'esecuzione di diverse macchine virtuali permise (e permette tuttora) di eseguire task di diversi utenti su un sistema progettato per un solo utente. Più tardi, in risposta ad alcuni problemi nell'esecuzione contemporanea di più applicazioni Windows XP su CPU Intel x86, VMware creò una nuova tecnologia di virtualizzazione per Windows XP. Si trattava di un'applicazione che eseguiva una o più copie ospiti (guest) di Windows o altri sistemi operativi per x86, ognuna in grado di eseguire le sue applicazioni (si veda la Figura 1.20). Windows era l'applicazione ospitante (host) e l'applicazione VMware era il gestore della macchina virtuale (VMM). Compito del VMM era eseguire i sistemi operativi ospiti, gestire le loro risorse e proteggere ciascun ospite dagli altri.

Anche se i moderni sistemi operativi sono in grado di eseguire più applicazioni in maniera affidabile, l'uso della virtualizzazione è in continua crescita. Un VMM permette all'utente di installare su un computer portatile o desktop più sistemi operativi e di eseguire applicazioni scritte per sistemi operativi diversi da quello installato nativamente. Per esempio, un portatile Apple con un sistema Mac OS X in esecuzione su CPU x86 può mandare in esecuzione un sistema ospite Windows per permettere l'esecuzione di applicazioni Windows. Le società che sviluppano software per diversi

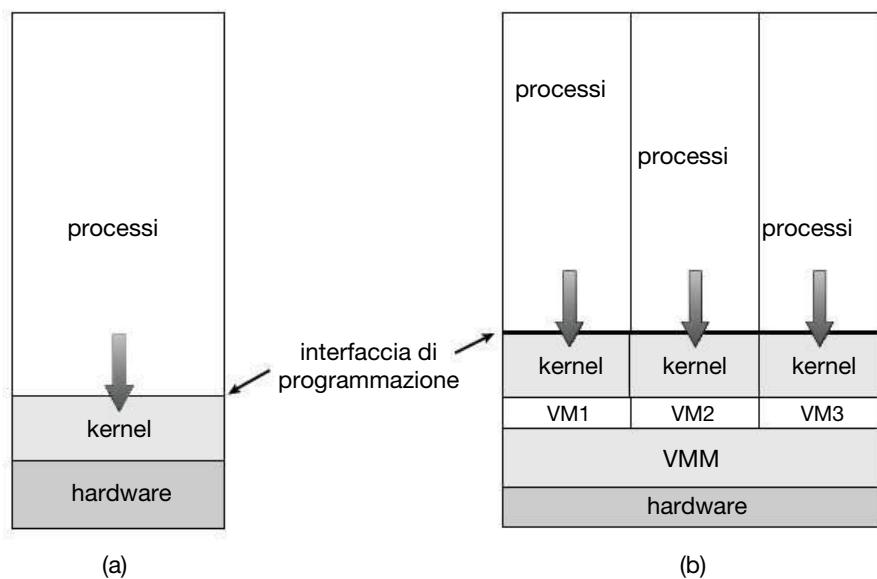


Figura 1.20 VMware.

sistemi operativi possono utilizzare la virtualizzazione per eseguire tutti questi sistemi operativi su un unico server fisico destinato allo sviluppo, al test e al debugging. Nei data center la virtualizzazione è diventata un metodo diffuso per eseguire e gestire gli ambienti elaborativi. Oggi i VMM come VMware, ESX e Citrix XenServer non sono più soltanto programmi in esecuzione sul sistema host, ma svolgono essi stessi il ruolo di host. Maggiori dettagli sugli strumenti per la virtualizzazione e sulla sua implementazione si trovano nel Capitolo 16.

1.11.7 Cloud computing

Il **cloud computing** è una tecnica che permette di fornire capacità elaborativa, storage e persino applicazioni come servizi di rete. In un certo senso si tratta di un'estensione logica della virtualizzazione, perché utilizza la virtualizzazione come strumento base per offrire le sue funzionalità. Per esempio, **EC2** (*elastic compute cloud*) di Amazon dispone di migliaia di server, milioni di macchine virtuali e petabyte di spazio dati a disposizione di ogni utente connesso a Internet. Gli utenti pagano una tariffa mensile che dipende dalla quantità di risorse che utilizzano.

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

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

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

Ci sono certamente sistemi operativi tradizionali dentro molti tipi di infrastrutture cloud. Al di sopra di questi sistemi ci sono i VMM che gestiscono le macchine virtuali su cui vengono eseguiti i processi utente. A un livello più alto gli stessi VMM sono controllati da strumenti per la gestione del cloud, come VMware vCloud Director e l'open-source Eucalyptus. Questi sistemi gestiscono le risorse all'interno di un dato cloud e forniscono interfacce verso i componenti cloud, offrendo così buoni argomenti per considerarli un nuovo tipo di sistema operativo.

La Figura 1.21 mostra un cloud pubblico che offre il servizio IaaS. Si noti che sia i servizi cloud sia l'interfaccia utente sono protetti da firewall.

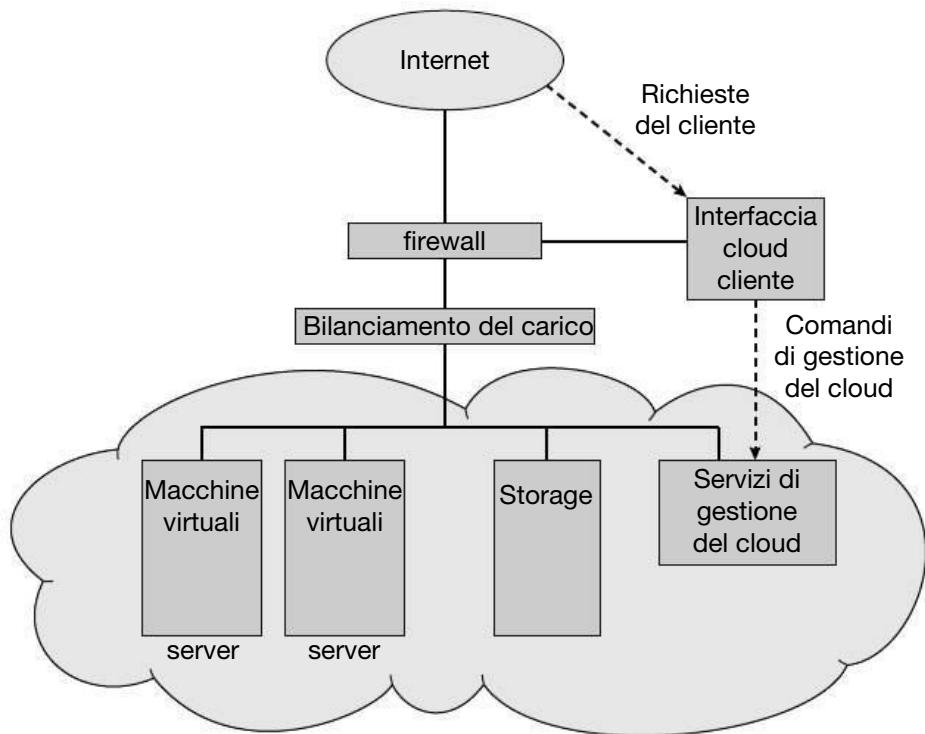


Figura 1.21 Cloud computing.

1.11.8 Sistemi embedded real-time

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

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

La diffusione dei sistemi embedded è in continua espansione. Indubbiamente, sono destinate a crescere anche le potenzialità di questi congegni, sia come unità indipendenti sia in qualità di membri delle reti e di Internet. È già oggi possibile l’automatizzazione di intere abitazioni, cosicché un computer centrale – sia che si tratti di un computer general purpose o di un sistema embedded – possa essere in grado di controllare il riscaldamento, l’illuminazione o i sistemi di allarme. Attraverso la rete,

si può comandare a distanza il riscaldamento della propria casa prima di rientrarvi. Un giorno, il frigorifero potrà forse prendere autonomamente l'iniziativa di chiamare il negozio di alimentari per il rifornimento del latte.

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

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

Il Capitolo 6 esamina la tipologia di scheduling da adottare affinché un sistema operativo possa eseguire applicazioni real-time. La gestione della memoria nell'ambito dell'elaborazione in tempo reale è illustrata nel Capitolo 9. Infine, i Capitoli 18 e 19 (disponibili sul sito web del volume) sono dedicati ai componenti real-time dei sistemi operativi Linux e Windows 7.

1.12 Sistemi operativi open-source

Lo studio dei sistemi operativi, come già sottolineato, è semplificato dalla disponibilità di un vasto numero di programmi open-source. I **sistemi operativi open-source** sono disponibili in formato sorgente anziché come codice binario compilato. Linux è il più noto sistema operativo open-source, mentre Microsoft Windows è un ben noto esempio dell'approccio opposto, a **sorgente chiuso (closed source)**. I sistemi operativi Mac OS X e iOS hanno un approccio ibrido: sono composti da un kernel open-source chiamato Darwin, ma includono componenti chiuse e proprietarie.

Avere a disposizione il codice sorgente permette al programmatore di produrre il codice binario, eseguibile da un sistema. Il processo inverso, chiamato processo di **reverse-engineering**, che permette di ricavare il codice sorgente partendo dal binario, è molto più oneroso; molti elementi utili, per esempio i commenti, non possono essere ripristinati. Apprendere il funzionamento dei sistemi operativi esaminandone il codice sorgente originale può essere molto utile. Avendo a disposizione il codice sorgente, uno studente può modificare il sistema operativo per poi compilare ed eseguire il codice, verificando così i cambiamenti che vi ha apportato. Procedere in questo modo è sicuramente di notevole aiuto per l'apprendimento. Questo libro include degli eser-

cizi che richiedono la modifica del codice sorgente di un sistema operativo. Alcuni algoritmi, inoltre, sono descritti ad alto livello, per essere sicuri di coprire tutti gli argomenti importanti che riguardano i sistemi operativi. Nel libro si possono inoltre trovare riferimenti a esempi di codice open-source, per eventuali approfondimenti.

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

1.12.1 Storia

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

Successivamente le società informatiche cercarono di limitare l’utilizzo del loro software a computer autorizzati e clienti paganti. Riuscirono a raggiungere questo obiettivo rendendo disponibili solo i file in binario compilati a partire dal codice sorgente, ma non il codice stesso. Esse protessero così allo stesso tempo il proprio codice e le proprie idee dai concorrenti. Un altro problema è quello che riguardava il materiale protetto da copyright. I sistemi operativi e altri programmi possono limitare la possibilità di accedere a film, musica e a libri elettronici solo a computer autorizzati. Tale **protezione o gestione dei diritti digitali** (*digital rights management*, DRM) non sarebbe efficace se il codice sorgente che implementa questi limiti fosse pubblicato. Le leggi di molti paesi, incluso il Digital Millennium Copyright Act (DMCA) negli Stati Uniti, rendono illegale ricavare un codice DRM tramite il reverse-engineering o provare a eludere la protezione del materiale.

Per contrastare la limitazione nell’utilizzo e nella redistribuzione di software, nel 1983 Richard Stallman diede vita al progetto GNU con la finalità di creare un sistema operativo gratuito, open-source e compatibile con UNIX. Nel 1985 pubblicò lo GNU Manifesto, sostenendo che tutto il software avrebbe dovuto essere gratuito e open-source.

Costituì inoltre la **Free Software Foundation (FSF)** con lo scopo di incoraggiare il libero scambio dei codici sorgente e il libero utilizzo del software. Anziché proteggere il proprio software con copyright, la FSF distribuisce il software con i relativi diritti (“copyleft”) incoraggiandone la condivisione e il miglioramento. La **General Public License** della **GNU (GPL)** codifica questa distribuzione ed è un’autorizzazione pubblica per il rilascio di software. Fondamentalmente la GPL richiede che il codice sorgente sia distribuito assieme all’eseguibile binario e che qualsiasi cambiamento apportato a quel sorgente sia anch’esso reso disponibile con la stessa autorizzazione GPL.

1.12.2 Linux

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

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

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

1. Scaricate il software di virtualizzazione gratuito “VMware player” all’indirizzo
<http://www.vmware.com/download/player/>
e installatelo sul vostro sistema.
2. Scegliete una distribuzione di Linux tra le centinaia di immagini di macchine virtuali (*appliances*), disponibili sul sito VMware all’indirizzo
<http://www.vmware.com/appliances/>
Queste immagini hanno preinstallati sistemi operativi e applicazioni e includono diverse varianti di Linux.
3. Avviate la macchina virtuale in ambiente VMware player.

In allegato a questo testo viene fornita un’immagine con la distribuzione Debian di Linux. L’immagine contiene il codice sorgente Linux e alcuni strumenti per lo sviluppo software. Nel testo sono contenuti esempi che riguardano questa immagine e, nel Capitolo 18 (sul sito web), un caso di studio dettagliato.

1.12.3 UNIX BSD

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

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

Darwin, il componente kernel fondamentale di Mac OS X, è basato su UNIX BSD ed è anch’esso open-source. Il sorgente è disponibile all’indirizzo <http://www.opensource.apple.com/>. Lo stesso sito contiene tutte le componenti open-source delle distribuzioni di Mac OS X. Il nome del pacchetto contenente il kernel comincia con “xnu”. Apple fornisce anche diversi strumenti per sviluppatori, documentazione e supporto all’indirizzo <http://connect.apple.com>.

1.12.4 Solaris

Solaris è il sistema operativo commerciale basato su UNIX della Sun Microsystems. In origine il sistema operativo della Sun, il SunOS, si basava su UNIX BSD. A partire dal 1991 Sun iniziò a utilizzare come base del suo sistema operativo UNIX System V di AT&T. Nel 2005 la Sun, nell’ambito del progetto OpenSolaris, rese open-source gran parte del codice di Solaris. Nel 2009 l’acquisto di Sun da parte di Oracle ha reso incerto lo stato di questo progetto. Il codice sorgente del 2005 è ancora accessibile attraverso un apposito browser o lo si può scaricare all’indirizzo

<http://src.opensolaris.org/source>.

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

LO STUDIO DEI SISTEMI OPERATIVI

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

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

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

Inoltre, l'incremento della virtualizzazione quale funzione principale (e spesso gratuita) del computer permette di far funzionare più sistemi operativi su un unico sistema (*core system*). Per esempio VMware (<http://www.vmware.com>) fornisce un programma gratuito per Windows sul quale possono girare centinaia di applicazioni virtuali gratuite. Virtualbox (<http://www.virtualbox.com>) mette a disposizione un gestore di macchine virtuali per diversi sistemi operativi, gratuito e open-source. Grazie a questi strumenti gli studenti possono sperimentare centinaia di sistemi operativi senza bisogno di un hardware dedicato.

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

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

1.12.5 Sistemi open-source come strumenti didattici

Il movimento a sostegno dei software gratuiti spinge legioni di programmati a creare migliaia di progetti open-source, inclusi sistemi operativi.

Siti come <http://freshmeat.net/> e <http://distrowatch.com/> offrono portali per molti di questi progetti. Come abbiamo detto, i progetti open-source permettono agli studenti di utilizzare il codice sorgente come strumento di apprendimento, dando loro l'opportunità di modificare i programmi, testarli, contribuire all'individuazione di bug e alla loro eliminazione, oltre a esplorare sistemi operativi maturi e completi, con i relativi compilatori, strumenti, interfacce utente e altri programmi. La disponibilità

del codice sorgente per progetti storici, come Multics, può aiutare gli studenti a comprendere quei progetti e a costruire conoscenze utili per nuovi progetti.

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

1.13 Sommario

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

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

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

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

Per utilizzare al meglio la CPU, i sistemi operativi moderni impiegano la multiprogrammazione, grazie alla quale diversi processi possono occupare contemporaneamente la memoria, assicurando che la CPU non resti mai inattiva. Con i sistemi time sharing il concetto di multiprogrammazione è stato ulteriormente esteso per mezzo di algoritmi

di scheduling della CPU che fanno rapidamente la spola tra un processo e l'altro, dando così l'impressione che molti processi siano in esecuzione allo stesso tempo.

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

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

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

I sistemi operativi utilizzano intensivamente diverse strutture dati fondamentali per l'informatica, tra cui: liste, pile, code, alberi, funzioni hash e bitmap.

Vi sono diversi tipi di ambienti elaborativi. L'elaborazione tradizionale riguarda i desktop e i computer portatili, solitamente connessi a una rete. Il mobile computing fa riferimento all'elaborazione su dispositivi come smartphone e tablet, in grado di offrire funzioni uniche nel loro genere. I sistemi distribuiti permettono agli utenti di condividere risorse su macchine geograficamente lontane connesse a una rete. I servizi possono essere forniti utilizzando un modello client-server o un modello peer-to-peer. La virtualizzazione ha a che fare con l'astrazione dell'hardware di un computer in molteplici distinti ambienti di esecuzione. Il cloud computing utilizza un sistema distribuito per offrire servizi in una "nuvola", in modo che gli utenti possano accedere ai servizi da postazioni remote. I sistemi operativi real-time sono progettati per sistemi embedded come elettronica di consumo, automobili e robotica.

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

GNU/Linux e UNIX BSD sono sistemi operativi open-source, ma ognuno ha obiettivi, utilità, licenze e scopi propri. I vantaggi del software gratuito e dell'open-source possono portare a un aumento del numero e della qualità dei progetti open-source, comportando un incremento del numero di individui e le aziende che li utilizzano.

Esercizi di ripasso

- 1.1** Quali sono i tre scopi principali di un sistema operativo?
- 1.2** Abbiamo sottolineato come il sistema operativo sia volto all'uso efficiente dell'hardware. Quando è opportuno che il sistema operativo rinunci a questo principio e “sprechi” risorse? Perché un sistema simile non può essere considerato davvero inefficiente?
- 1.3** Qual è la difficoltà principale che deve superare un programmatore nello scrivere un sistema operativo per un ambiente real-time?
- 1.4** Considerate le varie definizioni di sistema operativo. Valutate se sia opportuno che il sistema operativo includa o meno applicazioni quali browser e programmi di posta elettronica. Argomentate entrambe le possibilità, fornendo delle motivazioni.
- 1.5** Come funziona la distinzione tra modalità di sistema (modalità kernel) e modalità utente quale rudimentale forma di protezione (sicurezza) del sistema?
- 1.6** Quale delle seguenti istruzioni dovrebbe essere privilegiata?
 - a. Impostare il timer.
 - b. Leggere il clock.
 - c. Cancellare la memoria.
 - d. Invocare un’istruzione trap.
 - e. Disattivare le interruzioni.
 - f. Modificare le informazioni nella tabella che indica lo status dei dispositivi.
 - g. Passare da modalità utente a modalità di sistema.
 - h. Accedere a un dispositivo I/O.
- 1.7** Alcuni dei primi computer proteggevano il sistema operativo posizionandolo in una partizione della memoria che non poteva essere modificata né dall’utente né dal sistema operativo. Descrivete due difficoltà che secondo voi potrebbero sorgere da uno schema simile.
- 1.8** Alcune CPU offrono più di due modalità di operazione. Quali sono due possibili impieghi di queste modalità multiple?
- 1.9** I timer potrebbero essere utilizzati anche per calcolare l’ora corrente. Spiegate brevemente come.

- 1.10** Fornite due ragioni per l'utilizzo delle cache. Quali problemi risolvono? Quali problemi creano? Se una cache potesse essere costruita grande quanto il dispositivo per il quale lavora (per esempio una cache grande come un disco), perché non costruirla così grande ed eliminare il dispositivo?
- 1.11** Confrontate i modelli di sistemi distribuiti di tipo client-server e di tipo peer-to-peer.

Esercizi

- 1.12** In un ambiente multiprogrammato e time sharing, diversi utenti condividono il sistema simultaneamente. Tale situazione può generare alcuni problemi di sicurezza.
- Individuate due possibili problemi.
 - Si può garantire il medesimo grado di sicurezza in una macchina time sharing e in una macchina dedicata? Motivate la risposta.
- 1.13** La tematica dell'utilizzo delle risorse è una costante dei sistemi operativi, se pure in modi diversi a seconda del tipo di sistema considerato. Si dettaglino le risorse da gestire con cura nelle seguenti situazioni:
- mainframe o minicomputer;
 - stazioni di lavoro connesse a server;
 - dispositivi mobili.
- 1.14** Quando e perché sarebbe più conveniente per un utente scegliere un sistema time sharing anziché un PC o una stazione di lavoro individuale?
- 1.15** Descrivete le differenze tra multielaborazione simmetrica e asimmetrica. Elencate tre vantaggi e uno svantaggio dei sistemi multiprocessore.
- 1.16** Quali differenze presentano i cluster di elaboratori rispetto ai sistemi multiprocessore? Che cosa è necessario perché due macchine appartenenti a un cluster cooperino in modo da offrire un servizio altamente affidabile?
- 1.17** Ipotizziamo che sui due nodi di un cluster di elaboratori sia attiva una base di dati. Descrivete due modalità con cui il software per la gestione del cluster può regolare l'accesso ai dati sul disco, analizzando i pro e i contro di ognuna.
- 1.18** In che cosa i network computer si differenziano dagli elaboratori tradizionali? Enunciate qualche caso concreto in cui sia preferibile utilizzare un network computer.
- 1.19** Qual è lo scopo delle interruzioni? Quali differenze vi sono tra un'eccezione e un'interruzione? Un programma utente può generare un'eccezione di proposito? In caso affermativo, con quale scopo?

- 1.20** L'accesso diretto alla memoria (DMA) è usato per dispositivi I/O ad alta velocità per evitare di sovraccaricare la CPU.
- In che modo la CPU si interfaccia con il dispositivo per coordinare il trasferimento?
 - In che modo la CPU apprende che il trasferimento in memoria è completo?
 - La CPU è abilitata all'esecuzione di altri programmi mentre il controllore DMA procede al trasferimento dei dati. Può tale trasferimento interferire con la corretta esecuzione dei programmi utenti? Se la risposta è positiva, descrivete in quale forma può sorgere l'interferenza.
- 1.21** L'hardware di alcuni sistemi elaborativi non possiede una modalità hardware di funzionamento riservata al sistema operativo. Per questi calcolatori è possibile realizzare sistemi operativi sicuri? Fornite motivazioni in favore e contro questa possibilità.
- 1.22** Molti sistemi SMP hanno livelli distinti di cache: un livello interno a ogni core e un livello condiviso tra tutti i core. Perché i sistemi di cache sono progettati in questo modo?
- 1.23** Considerate un sistema SMP simile a quello della Figura 1.6. Illustrate con un esempio come i dati presenti nella memoria potrebbero avere un valore differente in ognuna delle cache locali.
- 1.24** Illustrate, con l'ausilio di esempi, come si manifesta il problema della coerenza dei dati memorizzati nella cache nei seguenti ambienti di elaborazione:
- sistemi a processore unico;
 - sistemi multiprocessore;
 - sistemi distribuiti.
- 1.25** Descrivete un meccanismo di protezione della memoria grazie al quale sia possibile impedire a un programma la modifica della memoria di pertinenza di altri programmi.
- 1.26** Quale configurazione di rete – LAN o WAN – si adatta meglio alle seguenti situazioni?
- Un edificio in un campus universitario.
 - Diversi campus localizzati all'interno di una stessa regione.
 - Il quartiere di una città.
- 1.27** Descrivete alcune problematiche da affrontare nel progetto di sistemi operativi per dispositivi mobili in confronto ai PC tradizionali.
- 1.28** Descrivete alcuni vantaggi dei sistemi peer-to-peer rispetto ai sistemi client-server.
- 1.29** Descrivete alcune applicazioni distribuite adatte a un sistema peer-to-peer.

1.30 Identificate vantaggi e svantaggi dei sistemi operativi open-source. Discutete anche le tipologie di persone che potrebbero definire determinati aspetti come vantaggi oppure svantaggi.

Note bibliografiche

[Brooksheat 2012] fornisce una panoramica a grandi linee dell'informatica in generale.

Uno studio dettagliato delle strutture dati si può trovare in [Cormen et al. 2009].

L'opera di [Solomon e Russinovich 2009], oltre a dare un'idea generale di Windows, è ricca di dettagli tecnici sulla struttura interna e sui componenti del sistema. [McDougall e Mauro 2007] descrive la struttura interna del sistema operativo Solaris. I componenti di Mac OS X vengono trattati in [Singh 2007]. [Love 2010] offre una panoramica del sistema Linux con numerosi dettagli sulle strutture dati utilizzate nel kernel.

I sistemi operativi trovano spazio in numerosi libri di testo generali, tra cui [Stalling 2011], [Deitel et al. 2004] e [Tanenbaum 2007].

[Kurose e Ross 2013] offre una trattazione generale delle reti di calcolatori, inclusi i sistemi client-server e peer-to-peer. [Tarkoma e Lagerspetz 2011] prende in esame diversi sistemi operativi per dispositivi mobili, tra cui Android e iOS.

[Hennessy e Patterson 2012] tratta i sistemi e i bus di I/O e più in generale l'architettura di un sistema. [Bryant and O'Hallaron 2010] fornisce una panoramica esauriva di un sistema di elaborazione dal punto di vista del programmatore. I dettagli dell'insieme delle istruzioni di Intel 64 e le modalità del processore si possono trovare in [Intel 2011].

La storia dell'open-source e dei suoi vantaggi e sfide si può trovare in [Raymond 1999]. La Free Software Foundation ha pubblicato la sua filosofia su <http://www.gnu.org/philosophy/free-software-for-freedom.html>. I sorgenti liberi di Mac OS X sono disponibili all'indirizzo <http://www.apple.vom/opensource>. Wikipedia contiene una voce su Richard Stallman all'indirizzo http://en.wikipedia.org/wiki/Richard_Stallman.

Bibliografia

[Brooksheat 2012] J.G. Brooksheat, *Computer Science: An Overview*, 11° ed., Addison-Wesley, 2012

[Bryant e O'Hallaron 2010] R. Bryant e D. O'Hallaron, *Computer Systems: A Programmers Perspective*, 2° ed., Addison-Wesley, 2010

[Cormen et al. 2009] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, *Introduction to Algorithms*, 3° ed., MIT Press, 2009.

[Deitel et al. 2004] H. Deitel, P. Deitel e D. Choffnes, *Operating Systems*, 3° ed., Prentice Hall, 2004

- [**Hennessy e Patterson 2012**] J. Hennessy e D. Patterson, *Computer architecture: a quantitative Approach*, 5° ed., Morgan Kaufmann, 2012.
- [**Intel 2011**] Intel 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 3A and 3B, Intel Corporation, 2011.
- [**Kurose e Ross 2013**] J. Kurose e K. Ross, *Computer Networking – A Top-Down Approach*, 6° ed., Addison-Wesley, 2013
- [**Love 2010**] R. Love, *Linux Kernel Development*, 3° ed., Developer’s Library, 2010
- [**McDougall e Mauro 2007**] R. McDougall e J. Mauro, *Solaris Internals*, 2° ed., Prentice-Hall, 2007.
- [**Raymond 1999**] E. S. Raymond, *The Cathedral and the Bazaar*, O’Reilly and Associates, 1999.
- [**Russinovich e Solomon 2009**] M. E. Russinovich e D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*, 5° ed., Microsoft Press, 2009.
- [**Singh 2007**] A. Singh, *Mac OS X Internals: A System Approach*, Addison-Wesley, 2007.
- [**Stallings 2011**] W. Stallings, *Operating Systems*, 7° ed., Prentice Hall, 2011.
- [**Tanenbaum 2007**] A.S. Tanenbaum, *Modern Operating Systems*, 3° ed., Prentice-Hall, 2007
- [**Tarkoma e Lagerspetz 2011**] S. Tarkoma e E. Lagerspetz, *Arching over the Mobile Computing Chasm: Platforms and Runtimes*, IEEE Computer, Vol. 44, p. 22-28, 2011

CAPITOLO

2

OBIETTIVI DEL CAPITOLO

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

Strutture dei sistemi operativi

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

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

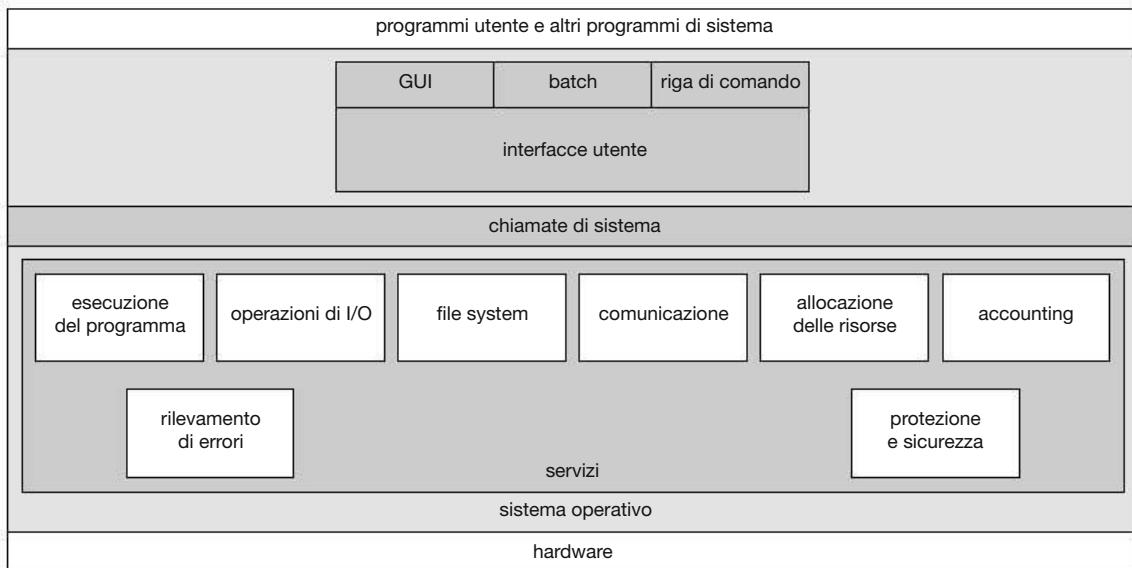


Figura 2.1 Panoramica dei servizi del sistema operativo.

2.1 Servizi di un sistema operativo

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

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

- **Interfaccia con l’utente.** Quasi tutti i sistemi operativi hanno un’**interfaccia con l’utente** (UI). Essa può assumere diverse forme. Un’**interfaccia a riga di comando** (CLI) è basata su stringhe che codificano i comandi, insieme a un metodo per inserirli (per esempio, una tastiera per digitare i comandi in uno specifico formato con determinate opzioni). Un’**interfaccia batch (a lotti)**, invece, prevede che comandi e relative direttive siano codificati in file, che vengono poi eseguiti. La forma senz’altro più diffusa è un’**interfaccia utente grafica** (GUI), ossia un sistema a finestre dotato di un dispositivo puntatore (per esempio, il mouse) per comandare operazioni di I/O e selezionare opzioni dai menu, insieme a una tastiera per inserire del testo. Certi sistemi offrono alcune o anche tutte queste soluzioni.
- **Esecuzione di un programma.** Il sistema deve poter caricare un programma in memoria ed eseguirlo. Il programma deve poter terminare la propria esecuzione in modo normale o anomalo (indicando l’errore).
- **Operazioni di I/O.** Un programma in esecuzione può richiedere un’operazione di I/O che implica l’uso di un file o di un dispositivo di I/O. Per particolari dispositivi possono essere necessarie funzioni speciali, come la registrazione su un CD o DVD,

oppure la cancellazione di uno schermo. Per motivi di efficienza e protezione, di solito un utente non può controllare direttamente i dispositivi di I/O, quindi il sistema operativo deve offrire mezzi adeguati.

- **Gestione del file system.** Il file system riveste un interesse particolare. I programmi richiedono l'esecuzione di operazioni di lettura e scrittura su file, oltre a creare e cancellare file e directory. Essi hanno anche bisogno di creare e cancellare i file, di eseguire la ricerca di un file con un certo nome, e disporre di informazioni relative al file stesso. Alcuni sistemi operativi, infine, gestiscono i permessi di accesso ai file sulla base della proprietà del file interessato. Molti sistemi operativi offrono all'utente la scelta di file system diversi con funzionalità e prestazioni specifiche.
- **Comunicazioni.** In molti casi un processo ha bisogno di scambiare informazioni con un altro processo. Ciò avviene principalmente in due modi: tra processi in esecuzione nello stesso calcolatore e tra processi in esecuzione in calcolatori diversi collegati per mezzo di una rete. La comunicazione si può realizzare tramite una **memoria condivisa**, che permette a due o più processi di leggere e scrivere in una porzione di memoria che condividono, o attraverso lo **scambio di messaggi**, in questo caso il sistema operativo trasferisce pacchetti d'informazioni in un formato predefinito tra i vari processi.
- **Rilevamento di errori.** Il sistema operativo deve essere sempre capace di rilevare e correggere eventuali errori che possono verificarsi nella CPU e nei dispositivi di memoria (come un errore di memoria o un guasto all'alimentazione elettrica), nei dispositivi di I/O (come un errore di parità in un nastro, il guasto di una connessione di rete, la mancanza di carta nella stampante) e in un programma utente (come una divisione per zero, un tentativo d'accesso a una locazione di memoria illegale, un uso eccessivo del tempo di CPU). Per assicurare un'elaborazione corretta e coerente il sistema operativo deve saper intraprendere l'azione giusta per ciascun tipo d'errore. Talvolta l'unica scelta possibile è l'arresto del sistema, altre volte è possibile terminare il processo che è causa d'errore o restituire un codice d'errore a un processo in modo che da solo cerchi di rilevare e correggere l'errore.

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

- **Allocazione delle risorse.** Se sono attivi più utenti o sono contemporaneamente in esecuzione più processi, il sistema operativo provvede all'assegnazione delle risorse necessarie a ciascuno di essi. Alcune di queste risorse, come i cicli di CPU, la memoria centrale e la memoria del file system, possono avere un software di gestione molto specifico, mentre altre, come i dispositivi di I/O, possono avere routine di richiesta e di rilascio più generali. Per esempio, per determinare come utilizzare al meglio la CPU, i sistemi operativi impiegano le procedure di schedu-

ling della CPU, che tengono conto della velocità, dei processi da eseguire, del numero di registri disponibili e di altri fattori. Esistono anche procedure per l’assegnazione di stampanti, driver di memorizzazione USB e altre periferiche.

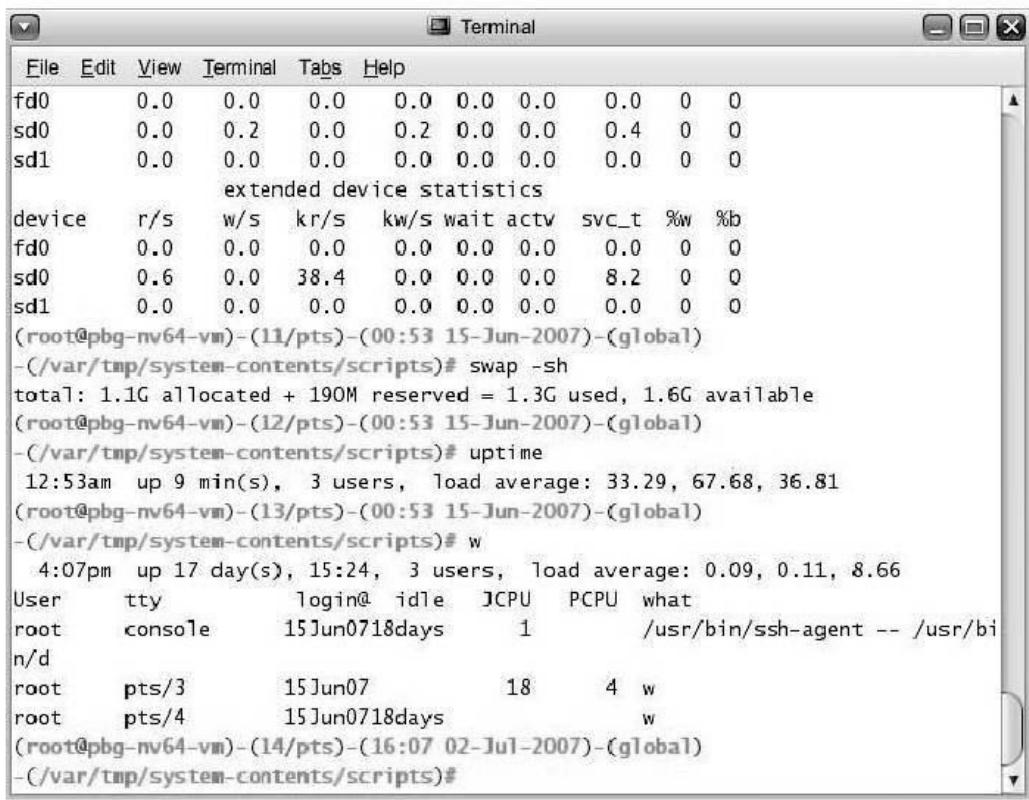
- **Accounting dell’uso delle risorse.** Vogliamo mantenere traccia di quali utenti usano il calcolatore, segnalando quali e quante risorse impiegano. Questo tipo di registrazione si può usare per contabilizzare l’uso delle risorse, in modo da addibitare il costo agli utenti, oppure per redigere statistiche;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

2.2 Interfaccia con l’utente del sistema operativo

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

2.2.1 Interprete dei comandi

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



```

Terminal

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

```

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

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

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

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

```
rm file.txt
```

cerca un file chiamato `rm`, lo carica in memoria e lo esegue con il parametro `file.txt`. La funzione corrispondente al comando `rm` è interamente definita dal co-

dice del file `rm`. In questo modo i programmatori possono aggiungere nuovi comandi al sistema, semplicemente creando nuovi file con il nome appropriato. Il codice dell’interprete dei comandi, che può quindi essere abbastanza piccolo, non necessita di alcuna modifica quando s’introducono nuovi comandi.

2.2.2 Interfaccia grafica con l’utente

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

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

Dato che l’utilizzo del mouse non è pratico per la maggior parte dei dispositivi mobili, gli smartphone e i tablet utilizzano di solito un’interfaccia touch screen che permette agli utenti di interagire attraverso opportuni gesti sullo schermo (*gesture*), per esempio premendo o strisciando le dita. La Figura 2.3 mostra lo schermo touch di un iPad. Anche se i primi smartphone erano dotati di tastiera fisica, attualmente la maggior parte dei dispositivi simula la tastiera sullo schermo.

Nei sistemi UNIX, tradizionalmente, le interfacce a riga di comando hanno avuto un ruolo preponderante, quantunque vi sia disponibilità di alcune interfacce GUI, come il CDE (*common desktop environment*) e i sistemi X-Windows, che sono diffusi fra le versioni commerciali di UNIX quali Solaris e il sistema AIX di IBM. Nella creazione di interfacce grafiche, inoltre, un impulso determinante è giunto da vari progetti **open-source** come il KDE (*K desktop environment*) e il desktop GNOME del progetto GNU. Ambedue i desktop, KDE e GNOME, sono compatibili con Linux e con vari sistemi UNIX, e sono regolate da licenza open-source, vale a dire che il loro codice sorgente è reso disponibile per consultazioni e per modifiche soggette a specifiche condizioni di licenza.



Figura 2.3 Lo schermo touch di un iPad.

2.2.3 Scelta dell'interfaccia

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

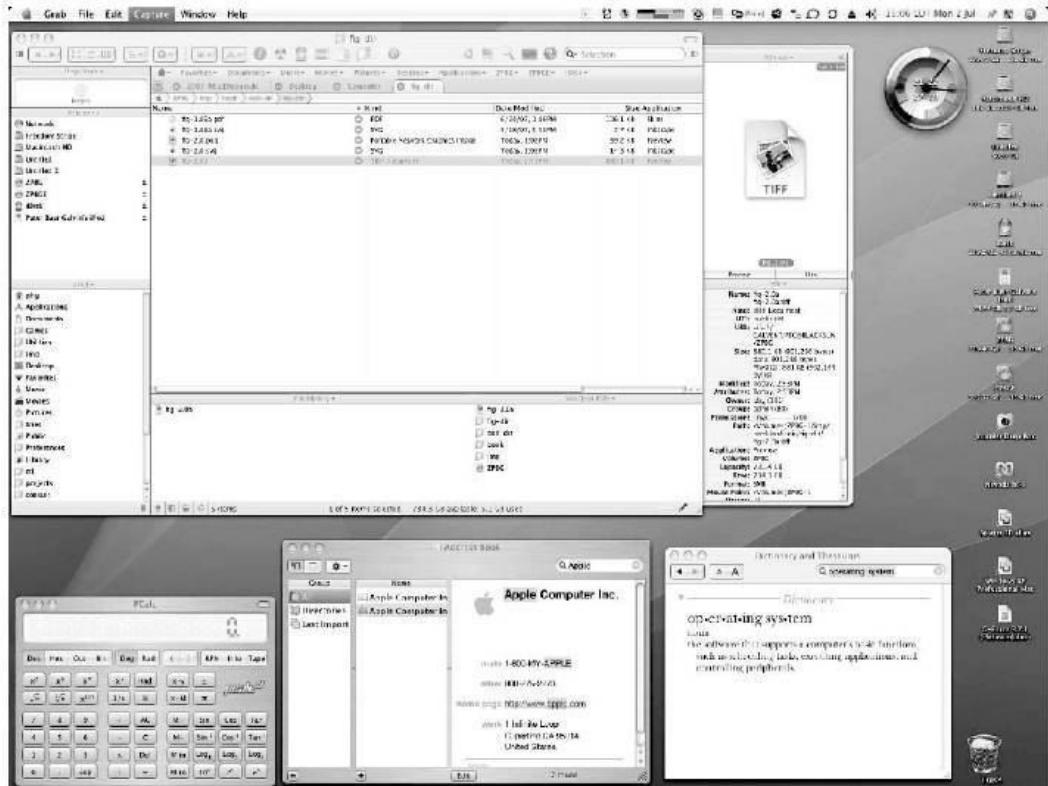


Figura 2.4 Interfaccia grafica di Mac OS X.

Molti utenti Windows, d’altro canto, sono soddisfatti dell’ambiente Windows GUI, e per questo motivo non usano quasi mai la shell dell’interfaccia MS-DOS. I sistemi operativi Macintosh, con i molti cambiamenti subiti, costituiscono un utile caso di studio da confrontare alla situazione di UNIX e Windows. Fino a tempi recenti, il Mac OS non disponeva di un’interfaccia a riga di comando, e vincolava l’interazione degli utenti con il sistema alla propria interfaccia GUI. Tuttavia, con l’introduzione del Mac OS X (realizzato, in parte, sfruttando il kernel UNIX), il sistema operativo contiene ora sia l’interfaccia grafica Aqua sia un’interfaccia a riga di comando. La Figura 2.4 mostra una schermata dell’interfaccia grafica di Mac OS X.

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

2.3 Chiamate di sistema

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

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

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

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

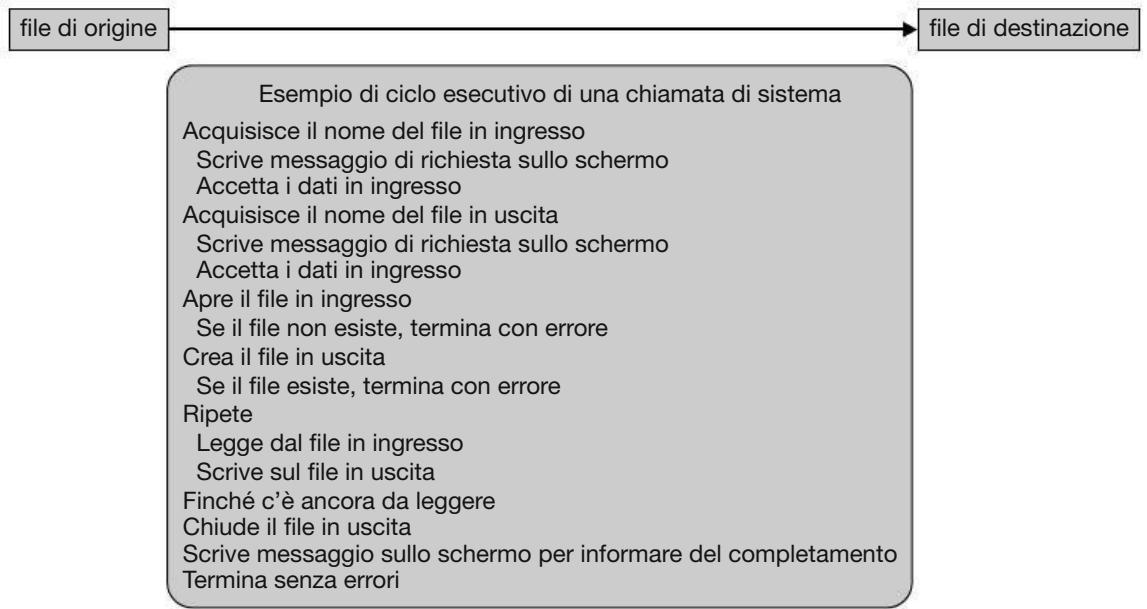


Figura 2.5 Esempio d’uso delle chiamate di sistema.

verificare vari errori, la cui natura dipende dal dispositivo impiegato (per esempio l’esaurimento dello spazio su disco).

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

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

Dietro le quinte, le funzioni fornite da un’API invocano le chiamate di sistema per conto del programmatore. Per esempio la funzione Windows `CreateProcess()`,

che ovviamente serve a generare un nuovo processo, invoca in effetti `NTCreateProcess()`, una chiamata di sistema del kernel di Windows.

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

Nella maggior parte dei linguaggi di programmazione il sistema di supporto all’esecuzione (*run-time support system*), un insieme di funzioni strutturate in librerie incluse nel compilatore, fornisce un’**interfaccia alle chiamate di sistema** che collega il linguaggio alle system call rese disponibili dal sistema operativo. L’interfaccia intercetta le chiamate a funzioni nella API, e invoca le relative system call. Di solito, ogni chiamata di sistema è codificata da un numero. L’interfaccia alle chiamate di sistema mantiene una tabella delle chiamate e invoca di volta in volta la chiamata richiesta, che risiede nel kernel del sistema, restituendo al chiamante lo stato della chiamata e i valori di ritorno.

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

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

Per passare parametri al sistema operativo si usano tre metodi generali. Il più semplice consiste nel passare i parametri in *registri*; si possono però presentare casi in cui vi sono più parametri che registri. In questi casi generalmente si memorizzano i parametri in un *blocco* o tabella di memoria e si passa l’indirizzo del blocco, come parametro, in un registro (Figura 2.7). È il metodo seguito dai sistemi operativi Linux

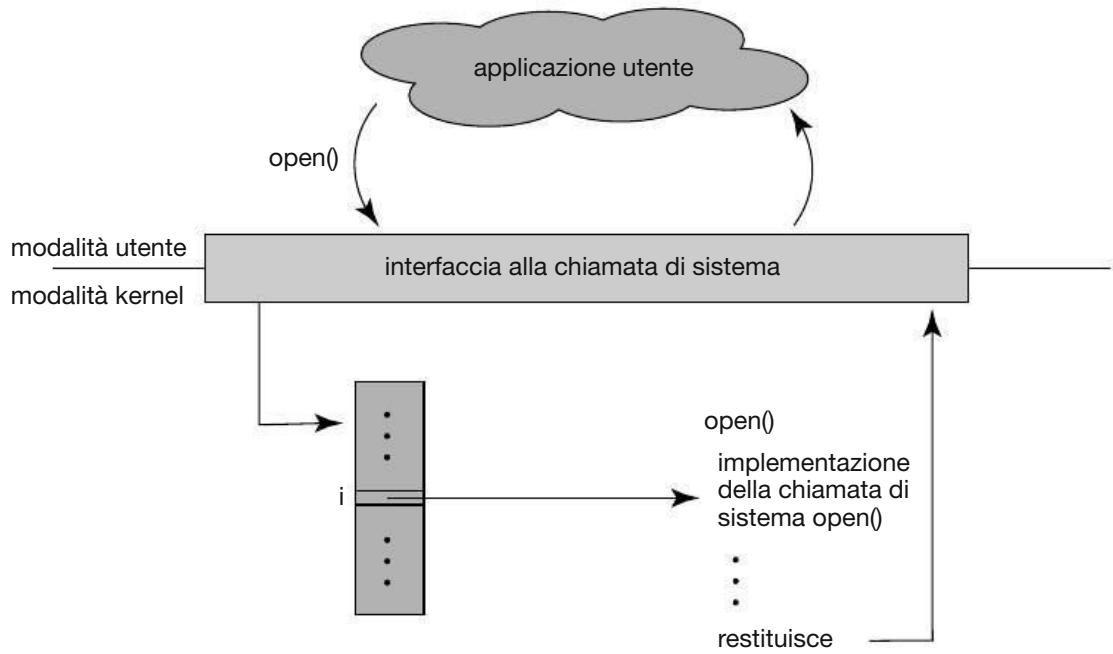


Figura 2.6 Gestione della chiamata di sistema `open()` invocata da un'applicazione utente.

ESEMPIO DI API STANDARD

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

`man read`

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

<code>#include <unistd.h></code>		
	<code>ssize_t</code>	<code>read(int fd, void *buf, size_t count)</code>
Valore restituito	Nome della funzione	Parametri

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

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

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

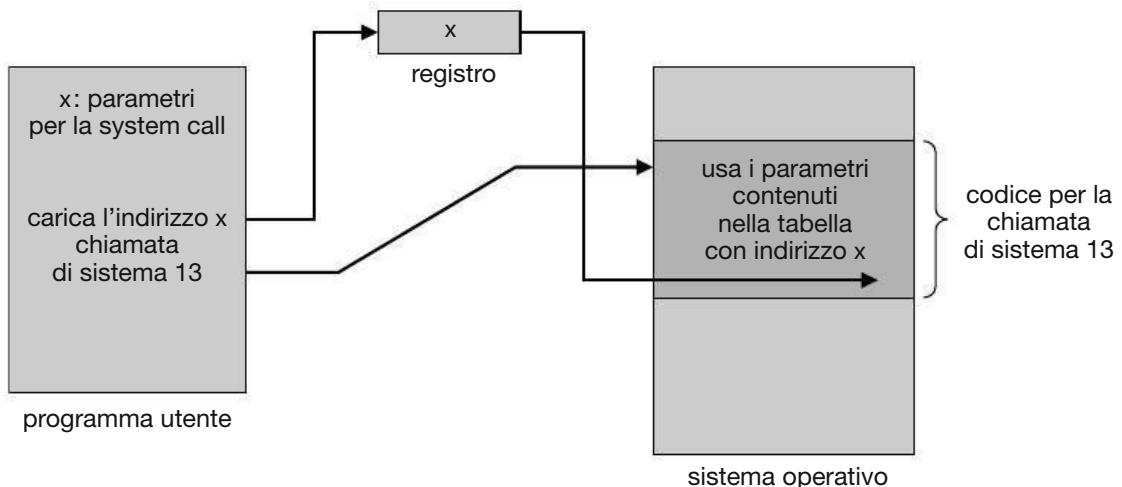


Figura 2.7 Passaggio di parametri in forma di tabella.

e Solaris. Il programma può anche collocare (*push*) i parametri nello stack da cui sono prelevati (*pop*) dal sistema operativo. Alcuni sistemi operativi preferiscono i metodi del blocco o dello stack, poiché non limitano il numero o la lunghezza dei parametri da passare.

2.4 Categorie di chiamate di sistema

Le chiamate di sistema sono classificabili approssimativamente in sei categorie principali: **controllo dei processi**, **gestione dei file**, **gestione dei dispositivi**, **gestione delle informazioni**, **comunicazioni e protezione**. Nei Paragrafi dal 2.4.1 al 2.4.6 sono illustrati brevemente i tipi di chiamate di sistema forniti da un sistema operativo. La maggior parte di queste chiamate di sistema implica o presuppone concetti e funzioni trattati in capitoli successivi. La Figura 2.8 riassume i tipi di chiamate di sistema forniti normalmente da un sistema operativo. Come già detto, in questo testo solitamente facciamo riferimento alle chiamate di sistema con denominazioni generiche. In tutto il libro, tuttavia, forniamo esempi delle controparti reali delle chiamate per i sistemi Windows, UNIX e Linux.

2.4.1 Controllo dei processi

Un programma in esecuzione deve potersi fermare in modo sia normale (`end()`) sia anomalo (`abort()`). Talvolta, se si ricorre a una chiamata di sistema per terminare in modo anomalo un programma in esecuzione, oppure se il programma incontra un problema e segnala l'errore con un'eccezione, un'immagine del contenuto della memoria viene copiata in un file (`dump`) e viene generato un messaggio d'errore. Il programmatore può usare uno specifico programma di ricerca e correzione di errori o **bug (debugger)** per esaminare tali informazioni e determinare le cause del problema. Sia in condizioni normali sia anomale il sistema operativo deve trasferire il controllo

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

Figura 2.8 Tipi di chiamate di sistema.

all’interprete dei comandi che legge il comando successivo. In un sistema interattivo l’interprete dei comandi continua semplicemente a interpretare il comando successivo; si suppone che l’utente invii un comando idoneo per rispondere a qualsiasi errore. In un sistema a interfaccia GUI una finestra avverte l’utente dell’errore e richiede indicazioni. In un sistema a lotti l’interprete dei comandi generalmente abortisce il lavoro corrente e prosegue con il successivo. Quando si presenta un errore, alcuni sistemi possono permettere specifiche azioni di recupero. Se il programma scopre un errore nei dati ricevuti e intende terminare in modo anomalo, può anche definire un livello d’errore. Più grave è l’errore, più alto è il livello del parametro che lo individua. È quindi possibile indicare in maniera uniforme una terminazione normale e una

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

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

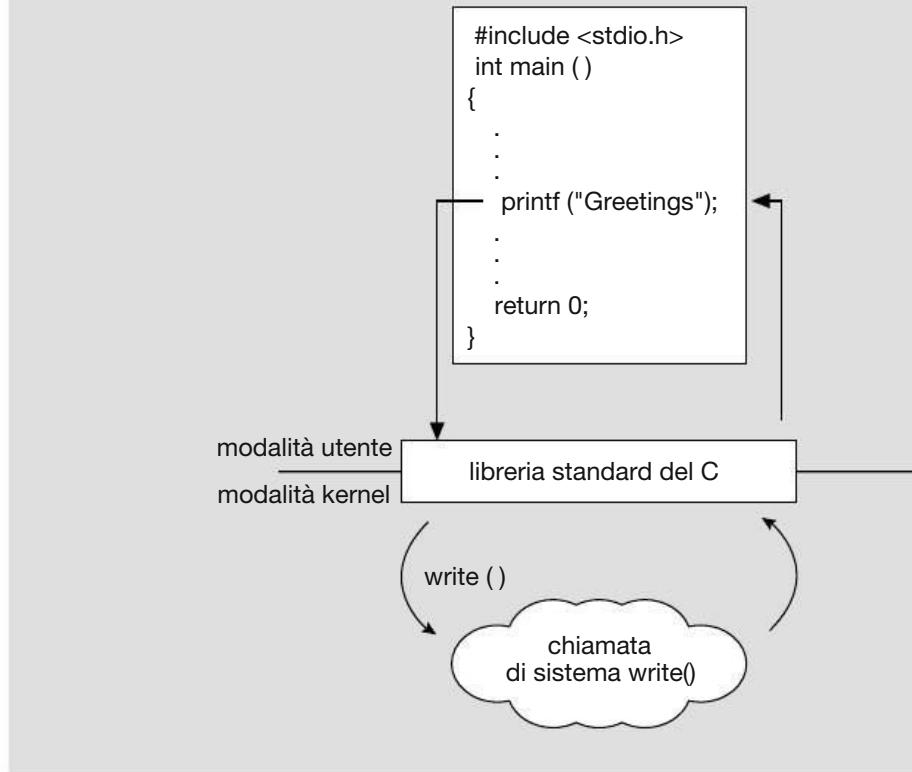
Un processo che esegue un programma può richiedere di caricare (`load()`) ed eseguire (`execute()`) un altro programma. In questo modo l'interprete dei comandi esegue un programma in seguito a una richiesta impartita, per esempio, da un comando utente, oppure dal clic di un mouse o da un comando batch.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

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

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

ESEMPIO DI LIBRERIA STANDARD DEL LINGUAGGIO C

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



tributi di un processo, compresi la sua priorità, il suo tempo massimo d’esecuzione e così via (`get_process_attributes()` e `set_process_attributes()`). Inoltre, può essere necessario terminare un processo creato, se si riscontra che non è corretto o se la sua esecuzione non è più utile (`terminate_process()`).

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

Molto spesso due o più processi possono condividere dati. Per assicurare l’integrità dei dati che vengono condivisi, spesso i sistemi operativi forniscono chiamate di sistema che consentono a un processo di bloccare (lock) dati condivisi, di modo che nessun altro processo possa accedere ai dati fino al rilascio del blocco. In genere tali chiamate di sistema includono `acquire_lock()` e `release_lock()`. Chiamate di sistema di questo tipo, che trattano cioè il coordinamento di processi concorrenti, sono esaminate in profondità nel Capitolo 5.

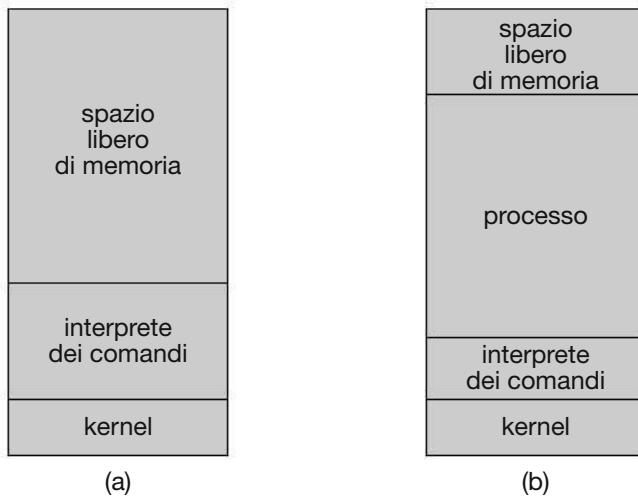


Figura 2.9 Esecuzione nell'MS-DOS. (a) All'avviamento del sistema. (b) Durante l'esecuzione di un programma.

Il controllo dei processi presenta così tanti aspetti e varianti che per chiarire questi concetti conviene ricorrere a due esempi, riguardanti uno un sistema monoprogrammato e l'altro un sistema multitasking. Il sistema operativo MS-DOS è un sistema che dispone di un interprete di comandi, attivato all'avviamento del calcolatore, e che esegue un solo programma alla volta (Figura 2.9(a)). MS-DOS opera in maniera semplice senza creare alcun nuovo processo; carica il programma in memoria, riscrivendo anche la maggior parte della memoria che esso stesso occupa, in modo da lasciare al programma quanta più memoria è possibile (Figura 2.9(b)); quindi imposta il contatore di programma alla prima istruzione del programma da eseguire. A questo punto si esegue il programma e si possono verificare due situazioni: un errore causa un segnale di eccezione, oppure il programma esegue una chiamata di sistema per terminare la propria esecuzione. In entrambi i casi si registra il codice d'errore in memoria di sistema per un eventuale uso successivo, quindi quella piccola parte dell'interprete che non era stata sovrascritta riprende l'esecuzione e il suo primo compito consiste nel ricaricare dal disco la parte rimanente dell'interprete stesso. Eseguito questo compito, quest'ultimo mette a disposizione dell'utente, o del programma successivo, il codice d'errore registrato.

Nel FreeBSD (derivato da UNIX Berkeley), quando un utente inizia una sessione di lavoro, il sistema esegue un interprete dei comandi (*shell*) scelto dall'utente. Quest'interprete è simile a quello dell'MS-DOS nell'accettare i comandi e nell'eseguire programmi richiesti dall'utente. Tuttavia, poiché il FreeBSD è un sistema multitasking, l'interprete dei comandi può continuare l'esecuzione mentre si esegue l'altro programma (Figura 2.10).

Per avviare un nuovo processo, la shell (interprete dei comandi) esegue la chiamata di sistema `fork()`; si carica il programma selezionato in memoria tramite la chiamata di sistema `exec()` e infine si esegue il programma. A seconda di come il

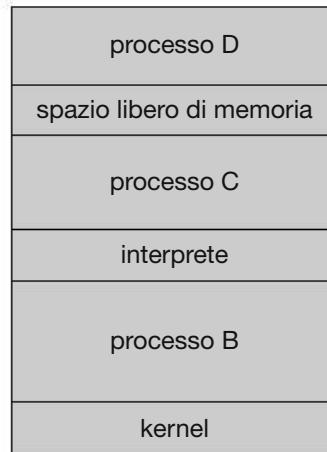


Figura 2.10 Esecuzione di più programmi nel sistema operativo FreeBSD.

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

2.4.2 Gestione dei file

Il file system è esaminato più in profondità nei Capitoli 11 e 12, tuttavia possiamo già identificare diverse chiamate di sistema riguardanti i file.

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

Queste stesse operazioni possono essere necessarie anche per le directory, nel caso in cui il file system sia strutturato in directory. È inoltre necessario poter determinare i valori degli attributi dei file o delle directory ed eventualmente modificarli. Tra gli attributi dei file figurano: nome, tipo, codici di protezione, informazioni di accounting, e così via. Per questa funzione sono richieste almeno due chiamate di sistema,

e precisamente `get_file_attributes()` e `set_file_attributes()`. Alcuni sistemi operativi forniscono molte più chiamate di sistema, per esempio per spostare (`move()`) e copiare (`copy()`) file. Altri forniscono API che eseguono operazioni di questo tipo tramite codice appropriato combinato a chiamate di sistema, e altri ancora mettono semplicemente a disposizione programmi di sistema che le eseguono. Se i programmi di sistema sono invocabili da altri programmi, ognuno di loro funge da API per altri programmi.

2.4.3 Gestione dei dispositivi

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

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

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

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

2.4.4 Gestione delle informazioni

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

Un altro insieme di chiamate di sistema è utile per il debugging di programmi. Molti sistemi operativi forniscono chiamate di sistema per ottenere un'immagine del-

la memoria (effettuare il `dump()`). Questa funzionalità è utile per il debugging. La `trace` di un programma fornisce la sequenza delle chiamate di sistema eseguite. Anche i microprocessori offrono una modalità conosciuta come *a singolo passo* (*single step*) nella quale viene eseguita una *trap* dopo ogni istruzione. La *trap* viene solitamente catturata dal debugger.

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

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

2.4.5 Comunicazione

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

sistema `read_message()` e `write_message()`; la chiamata di sistema `close_connection()` pone fine alla comunicazione.

Nel **modello a memoria condivisa**, invece, i processi usano chiamate di sistema `shared_memory_create()` e `shared_memory_attach()` per creare e accedere alle aree di memoria possedute da altri processi. Occorre ricordare che, normalmente, il sistema operativo tenta di impedire a un processo l'accesso alla memoria di un altro processo. Il modello a memoria condivisa richiede che più processi concordino nel superare tale limite; a questo punto tali processi possono scambiarsi le informazioni leggendo e scrivendo i dati nelle aree di memoria condivise. Il formato e la posizione dei dati sono determinati esclusivamente da questi processi e non sono sotto il controllo del sistema operativo. I processi sono anche responsabili del rispetto della condizione di non scrivere contemporaneamente nella stessa posizione. Questi meccanismi sono discussi nel Capitolo 5. Nel Capitolo 4 si illustra anche una variante del modello di processo, detto *thread*, che prevede a priori la condivisione della memoria.

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

2.4.6 Protezione

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

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

La protezione è trattata nel Capitolo 14. Il Capitolo 15 esamina la sicurezza in una prospettiva più ampia.

2.5 Programmi di sistema

Un'altra caratteristica importante di un sistema moderno è quella che riguarda l'insieme di programmi di sistema. Facendo riferimento alla Figura 1.1, in cui s'illustra la gerarchia logica di un calcolatore, si può notare che il livello più basso è occupato dall'hardware. Seguono, nell'ordine, il sistema operativo, i programmi di sistema e i

programmi applicativi. I **programmi di sistema**, detti anche **utilità di sistema**, offrono un ambiente più conveniente per lo sviluppo e l'esecuzione dei programmi; alcuni sono semplici interfacce per le chiamate di sistema, altri sono considerevolmente più complessi; in generale si possono classificare nelle seguenti categorie.

- **Gestione dei file.** Questi programmi creano, cancellano, copiano, rinominano, stampano, elencano e in genere compiono operazioni sui file e le directory.
- **Informazioni di stato.** Alcuni programmi richiedono semplicemente al sistema di indicare data, ora, quantità di memoria disponibile o spazio nei dischi, numero degli utenti e simili informazioni di stato. Altri, più complessi, forniscono informazioni dettagliate su prestazioni, accessi al sistema e debug. In genere mostrano le informazioni su terminale, o tramite altri dispositivi per l'uscita dei dati, o, ancora, all'interno di una finestra della GUI. Alcuni sistemi comprendono anche un **registro** (*registry*), al fine di archiviare e poter poi consultare informazioni sulla configurazione del sistema.
- **Modifica dei file.** Diversi editor sono disponibili per creare e modificare il contenuto di file memorizzati su dischi o altri dispositivi, oltre a comandi speciali per l'individuazione di contenuti di file o per particolari trasformazioni del testo.
- **Ambienti di supporto alla programmazione.** Compilatori, assemblatori, debugger e interpreti dei comuni linguaggi di programmazione, come C, C++, Java e PERL, sono spesso forniti insieme con il sistema operativo oppure disponibili per il download.
- **Caricamento ed esecuzione dei programmi.** Una volta assemblato o compilato, per essere eseguito, un programma deve essere caricato in memoria. Il sistema può mettere a disposizione caricatori assoluti, caricatori rilocabili, editor dei collegamenti (*linkage editor*) e caricatori di sezioni sovrapponibili di programmi (*overlay loader*). Sono necessari anche i sistemi d'ausilio all'individuazione e correzione degli errori (*debugger*) per i linguaggi d'alto livello o per il linguaggio macchina.
- **Comunicazioni.** Questi programmi offrono i meccanismi con cui si possono creare collegamenti virtuali tra processi, utenti e calcolatori diversi. Permettono agli utenti d'inviare messaggi agli schermi d'altri utenti, di consultare il Web, d'inviare messaggi di posta elettronica, di effettuare il login su calcolatori remoti, di trasferire file da un calcolatore a un altro.
- **Servizi in background.** Tutti i sistemi general-purpose hanno metodi per lanciare alcuni programmi di sistema al momento dell'avvio. Alcuni di questi processi terminano dopo aver completato i loro compiti, mentre altri restano in esecuzione fino a quando il sistema viene arrestato. I processi di sistema costantemente in esecuzione sono noti come **servizi**, **sottosistemi** oppure **demoni**. Un esempio è il demonio di rete discusso nel Paragrafo 2.4.5. In tale esempio un sistema aveva bisogno di un servizio per rilevare le connessioni di rete e quindi assegnare le richieste ai

processi corretti. Altri esempi includono le utilità di scheduling dei processi, che avviano i processi secondo una pianificazione specifica, i servizi di monitoraggio di errori di sistema e i server di stampa. Un sistema tipico ha decine di demoni. I sistemi operativi che eseguono attività importanti in contesto utente invece che in kernel possono utilizzare appositi demoni per eseguire queste attività.

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

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

2.6 Progettazione e realizzazione di un sistema operativo

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

2.6.1 Scopi della progettazione

Il primo problema che s'incontra nella progettazione di un sistema riguarda la definizione degli obiettivi e delle specifiche del sistema stesso. Al più alto livello, la progettazione del sistema è influenzata in modo decisivo dalla scelta dell'architettura fisica e del tipo di sistema: a lotti (batch) o time sharing, mono o multiutente, distribuito, per elaborazioni in real-time o general-purpose.

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

Gli utenti desiderano che un sistema abbia alcune caratteristiche ovvie: deve essere utile, facile da imparare e usare, affidabile, sicuro e veloce; queste caratteristiche non

sono particolarmente utili nella progettazione di un sistema, poiché non tutti concordano sui metodi da applicare per raggiungere questi scopi.

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

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

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

2.6.2 Meccanismi e politiche

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

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

I sistemi operativi basati su microkernel (Paragrafo 2.7.3) portano alle estreme conseguenze la separazione dei meccanismi dalle politiche, fornendo un insieme di funzioni fondamentali da impiegare come elementi di base; tali funzioni, quasi completamente indipendenti dalle politiche, consentono l'aggiunta di meccanismi e criteri più complessi tramite moduli del kernel creati dagli utenti o anche tramite programmi utente. Come esempio, si consideri l'evoluzione di UNIX. In principio, aveva uno scheduler basato sul time sharing. Nelle ultime versioni di Solaris, lo scheduling è controllato da tabelle caricabili: a seconda della tabella corrente, il sistema può essere time sharing, a lotti (batch), in tempo reale, fair share, o adottare una combinazione delle strategie precedenti. Tale parametrizzazione dei meccanismi di scheduling permette di attuare cambiamenti di vasta portata alle politiche del sistema con l'esecu-

zione di un singolo comando di caricamento (`load-new-table`) di una nuova tabella. All’altro estremo si trovano sistemi come Windows, nei quali sia i criteri sia i meccanismi sono fissati a priori e cablati nel sistema, al fine di fornire agli utenti un’unica immagine globale. Tutte le applicazioni hanno interfacce simili, perché l’interfaccia stessa fa parte del kernel e delle librerie del sistema. Il sistema Mac OS X è di tipo analogo.

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

2.6.3 Realizzazione

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

I primi sistemi operativi erano scritti in linguaggio assembler. Oggi, anche se alcuni sistemi operativi sono ancora scritti in linguaggio assembler, la maggior parte è scritta in un linguaggio di alto livello come il C o in linguaggi di livello ancora più alto come il C++. In realtà un sistema operativo può essere scritto in più linguaggi, per esempio usando il linguaggio assembler per i livelli più bassi del kernel, il C per routine di livello superiore e il C, il C++ o linguaggi di scripting interpretati come PERL, Python o gli script di shell per i programmi di sistema. Ogni distribuzione di Linux include probabilmente programmi scritti in ciascuno di questi linguaggi.

Il primo sistema scritto in un linguaggio di alto livello fu probabilmente il Master Control Program (MCP) per i calcolatori Burroughs; l’MCP fu scritto infatti in una variante del linguaggio ALGOL; il MULTICS, sviluppato al MIT, fu scritto prevalentemente nel linguaggio di programmazione di sistema PL/1; i kernel di Linux e Windows sono scritti per lo più in C, sebbene si trovino alcune brevi sezioni di codice assembler che riguardano i driver dei dispositivi e il salvataggio e il ripristino dei registri del sistema.

I vantaggi derivanti dall’uso di un linguaggio di alto livello, o perlomeno di un linguaggio orientato in modo specifico allo sviluppo di sistemi, per implementare un sistema operativo, sono gli stessi che si ottengono quando il linguaggio si usa per i programmi applicativi: il codice si scrive più rapidamente, è più compatto ed è più facile da capire e mettere a punto. Inoltre il perfezionamento delle tecniche di compilazione consente di migliorare il codice generato per l’intero sistema operativo con una semplice ricompilazione. Infine, un sistema operativo scritto in un linguaggio di alto livello è più facile da adattare a un’altra architettura (*porting*). L’MS-DOS, per esempio, fu scritto nel linguaggio assembly dell’Intel 8088, quindi è disponibile solo per la famiglia di CPU Intel (si noti che sebbene MS-DOS funzioni nativamente solo su Intel X86, gli emulatori del set di istruzioni x86 consentono al sistema operativo di girare su altre CPU, ma più lentamente e con un uso delle risorse più elevato. Come abbiamo accennato nel Capitolo 1, gli **emulatori** sono programmi che riproducono le funzionalità di un sistema su un altro sistema). Il sistema operativo Linux, scritto

prevalentemente in C, è invece disponibile su diversi tipi di CPU, tra cui Intel 80x86, Oracle SPARC e IBM PowerPC.

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

Come in altri sistemi, i miglioramenti principali nelle prestazioni dei sistemi operativi sono dovuti più a strutture dati e algoritmi migliori che a un'ottima codifica in linguaggio assembler. Inoltre, sebbene i sistemi operativi siano molto grandi, solo una piccola parte del codice assume un'importanza critica per le prestazioni: il gestore degli interrupt, il gestore dell'I/O, il gestore della memoria e lo scheduler della CPU sono probabilmente le procedure più critiche. Una volta scritto il sistema e verificato il suo corretto funzionamento, è possibile identificare le procedure che possono costituire colli di bottiglia e sostituirle con procedure equivalenti scritte in linguaggio assembler.

2.7 Struttura del sistema operativo

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

2.7.1 Struttura semplice

Molti sistemi operativi non hanno una struttura ben definita; spesso sono nati come sistemi piccoli, semplici e limitati, e solo in un secondo tempo si sono accresciuti superando il loro scopo originale. Un sistema di questo tipo è l'MS-DOS, originariamente progettato e realizzato da poche persone che non avrebbero mai immaginato una simile diffusione. Non fu suddiviso attentamente in moduli poiché lo scopo prioritario era fornire la massima funzionalità nel minimo spazio di memoria. La Figura 2.11 riporta la sua struttura.

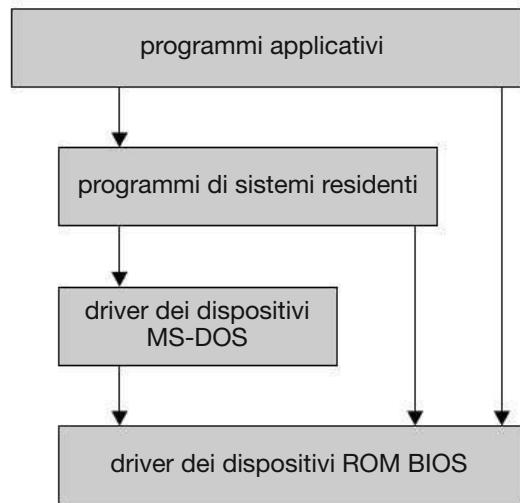


Figura 2.11 Struttura degli strati dell'MS-DOS.

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

Anche il sistema UNIX originale è poco strutturato, a causa delle limitazioni dell'hardware disponibile al tempo della sua progettazione. Il sistema consiste di due parti separate, il kernel e i programmi di sistema. A sua volta, il kernel è diviso in una serie di interfacce e driver dei dispositivi, aggiunti ed espansi nel corso dell'evoluzione di UNIX. Possiamo vedere UNIX come un sistema parzialmente stratificato, come mostrato nella Figura 2.12: tutto ciò che sta al di sotto dell'interfaccia alle chiamate di sistema e al di sopra dell'hardware costituisce il kernel. Esso comprende il file system, lo scheduling della CPU, la gestione della memoria, e le altre funzionalità del sistema rese disponibili tramite chiamate di sistema. Tutto sommato, si tratta di un'enorme massa di funzionalità diverse combinate in un solo livello. Questa struttura monolitica rendeva difficile l'implementazione e la manutenzione, ma offriva comunque un vantaggio in termini di prestazioni: un overhead molto ridotto nell'interfaccia alle chiamate di sistema o nella comunicazione all'interno del kernel. Possiamo ancora vedere dei segni di questa semplice struttura monolitica nei sistemi operativi UNIX, Linux e Windows.

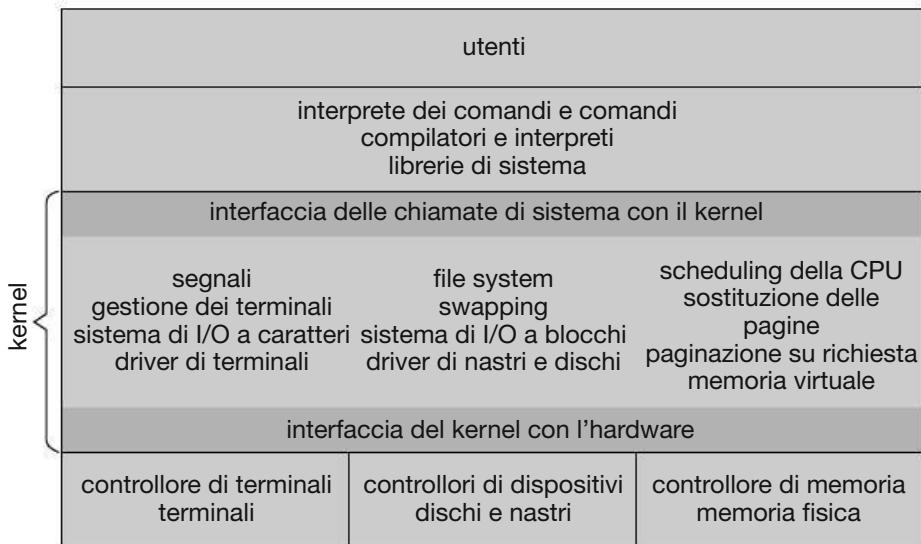


Figura 2.12 Struttura del sistema UNIX.

2.7.2 Metodo stratificato

In presenza di hardware appropriato, i sistemi operativi possono essere suddivisi in moduli più piccoli e gestibili di quanto non fossero quelli delle prime versioni di MS-DOS e UNIX. Ciò permette al sistema operativo di mantenere un controllo molto più stretto sul calcolatore e sulle applicazioni che lo utilizzano. Inoltre, gli sviluppatori del sistema godono di maggiore libertà nel modificare i meccanismi interni del sistema e nel suddividere il sistema in moduli. Secondo un approccio *top-down*, le specifiche del sistema partono dall'individuazione delle sue funzionalità e delle sue caratteristiche complessive, che sono poi suddivise in componenti distinte. L'attenzione all'incapsulamento delle informazioni è anche importante, in quanto dà libertà agli sviluppatori di implementare le routine di basso livello nel modo che ritengono più opportuno, fintanto che l'interfaccia esterna alla routine rimanga invariata, e la routine stessa esegua il compito per cui è stata prevista.

Vi sono molti modi per rendere modulare un sistema operativo. Uno di essi è il **metodo stratificato**, secondo il quale il sistema è suddiviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (strato 0), il più alto all'interfaccia con l'utente (strato N). Si veda la Figura 2.13.

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

Il vantaggio principale offerto da questo metodo è dato dalla semplicità di progettazione e di debugging. Gli strati sono composti in modo che ciascuno usi solo funzioni (operazioni) e servizi appartenenti a strati di livello inferiore. Questo approccio semplifica il debugging e la verifica del sistema. Il primo strato si può mettere a punto

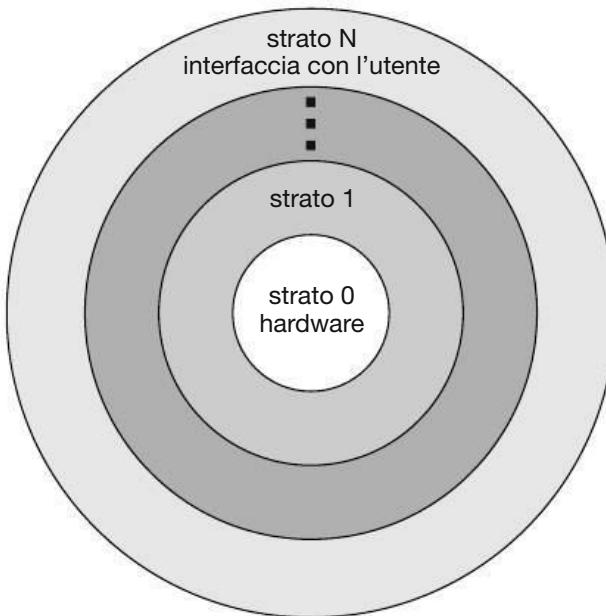


Figura 2.13 Struttura a strati di un sistema operativo.

senza intaccare il resto del sistema, poiché per realizzare le proprie funzioni usa, per definizione, solo lo strato hardware, che si presuppone sia corretto. Passando al secondo strato si presume, dopo la messa a punto, la correttezza del primo. Il procedimento si ripete per ogni strato. Se si riscontra un errore, questo deve trovarsi in quello strato, poiché gli strati inferiori sono già stati corretti; quindi la suddivisione in strati semplifica la progettazione e la realizzazione di un sistema.

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

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

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

Un ulteriore problema che si pone con la struttura stratificata è che essa tende a essere meno efficiente delle altre; per esempio, per eseguire un'operazione di I/O un

programma utente invoca una chiamata di sistema che è intercettata dallo strato di I/O che, a sua volta, esegue una chiamata allo strato di gestione della memoria, che a sua volta richiama lo strato di scheduling della CPU e che quindi è passata all'opportuno dispositivo di I/O. In ciascuno strato i parametri sono modificabili, può rendersi necessario il passaggio di dati, e così via; ciascuno strato aggiunge un overhead alla chiamata di sistema. Ne risulta una chiamata di sistema che richiede molto più tempo di una chiamata di sistema corrispondente in un sistema non stratificato.

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

2.7.3 Microkernel

Abbiamo visto che, a mano a mano che il sistema operativo UNIX è stato esteso, il kernel è cresciuto notevolmente, diventando sempre più difficile da gestire. Verso la metà degli anni '80 un gruppo di ricercatori della Carnegie Mellon University progettò e realizzò un sistema operativo, **Mach**, col kernel strutturato in moduli secondo il cosiddetto orientamento a **microkernel**. Seguendo questo orientamento si progetta il sistema operativo rimuovendo dal kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema. Ne risulta un kernel di dimensioni assai inferiori. Non c'è un'opinione comune su quali servizi debbano rimanere nel kernel e quali si debbano realizzare nello spazio utente. Tuttavia, in generale, un microkernel offre i servizi minimi di gestione dei processi, della memoria e di comunicazione. La Figura 2.14 mostra l'architettura tipica di un microkernel.

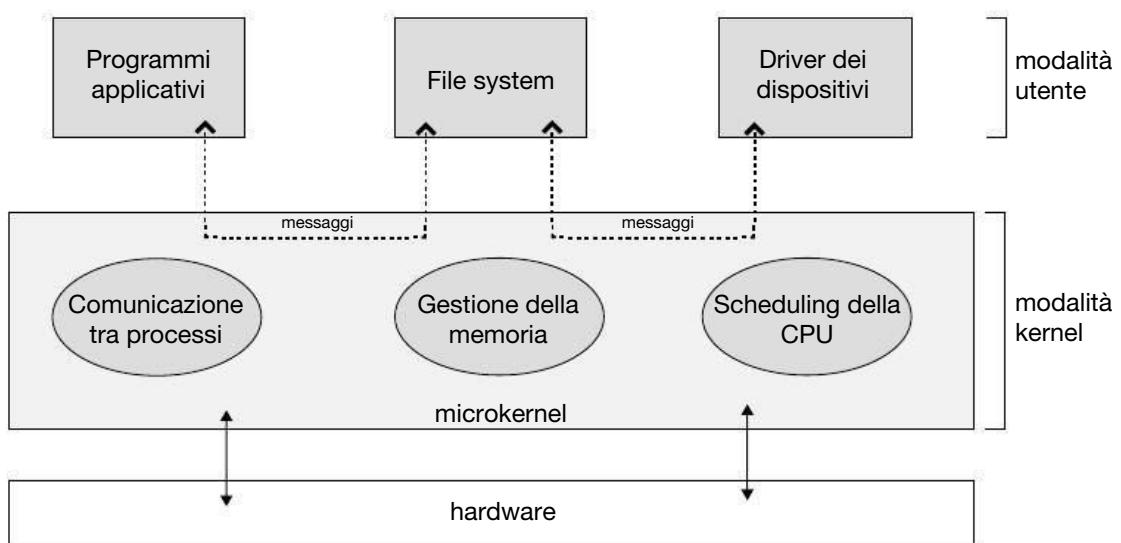


Figura 2.14 Architettura tipica di un microkernel.

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

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

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

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

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

2.7.4 Moduli

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

L'idea di base è che il kernel deve fornire direttamente i servizi principali, mentre gli altri servizi sono implementati in modo dinamico, quando il kernel è in esecuzione. Il collegamento dinamico dei servizi è preferibile all'aggiunta di nuove funzionalità direttamente nel kernel, che richiederebbe di ricompilare il kernel ogni volta

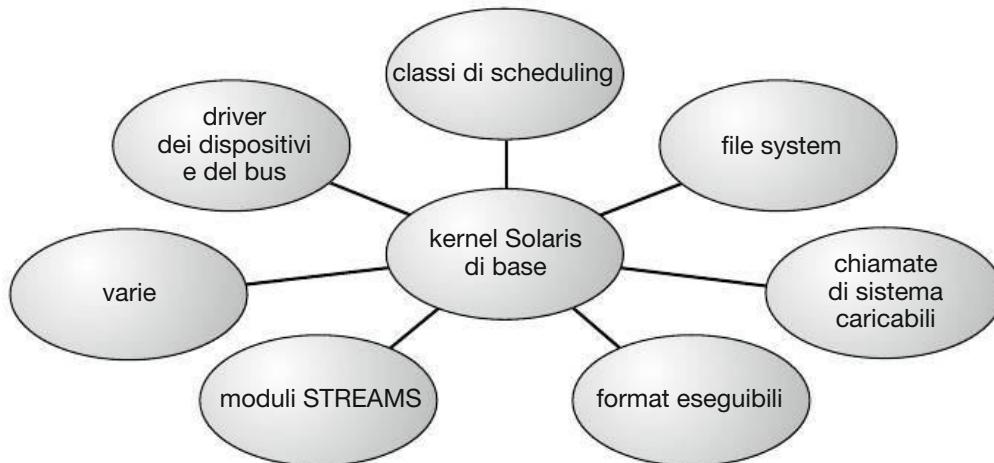


Figura 2.15 Moduli caricabili di Solaris.

che si effettua un cambiamento. Così, per esempio, si potrebbero includere lo scheduling della CPU e gli algoritmi di gestione della memoria direttamente nel kernel e aggiungere il supporto per diversi file system attraverso moduli caricabili.

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

La struttura di Solaris illustrata dalla Figura 2.15, si incentra su un kernel dotato dei seguenti sette tipi di moduli caricabili.

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

Anche Linux utilizza moduli del kernel caricabili dinamicamente, principalmente per supportare i driver delle periferiche e il file system. Analizzeremo la creazione di moduli caricabili del kernel in Linux in un esercizio di programmazione alla fine di questo capitolo.

2.7.5 Sistemi ibridi

Nella pratica pochi sistemi operativi adottano un'unica struttura ben definita. I sistemi operativi, piuttosto, combinano strutture diverse, che portano a sistemi ibridi indirizzati alle prestazioni, alla sicurezza e all'usabilità. Per esempio, sia Linux che Solaris sono monolitici, perché avere il sistema operativo in un unico spazio di indirizzamento garantisce prestazioni migliori, ma sono anche modulari, quindi una nuova funzionalità può essere aggiunta dinamicamente al kernel. Windows è in gran parte monolitico (ancora una volta principalmente per questioni di prestazioni), ma conserva alcuni comportamenti tipici dei sistemi microkernel, tra cui il supporto per sottosistemi separati (conosciuti come personalità del sistema operativo) che vengono eseguiti come processi in modalità utente. I sistemi Windows forniscono anche il supporto per i moduli del kernel caricabili dinamicamente. Forniamo casi di studio di Linux e di Windows 7 nei Capitoli 18 e 19, rispettivamente (nel sito web). Nel resto di questo paragrafo esploriamo la struttura di tre sistemi ibridi: Apple Mac OS X e i due principali sistemi operativi per dispositivi mobili, iOS e Android.

2.7.5.1 Mac OS X

Il sistema operativo Apple Mac OS X adotta una struttura ibrida. Come mostrato nella Figura 2.16 è un sistema stratificato. Gli strati superiori comprendono l'interfaccia utente **Aqua** (Figura 2.4) e una collezione di ambienti e servizi applicativi. In particolare, l'ambiente **Cocoa** definisce un'API per il linguaggio di programmazione Objective-C, utilizzato per la scrittura di applicazioni Mac OS X.

Il kernel si trova in uno strato sottostante, ed è costituito dal microkernel Mach e dal kernel BSD. Il primo cura la gestione della memoria, le chiamate di procedure remote (RPC), la comunicazione tra processi (IPC) – compreso lo scambio di messaggi – e lo scheduling dei thread. Il secondo mette a disposizione un'interfaccia BSD a riga

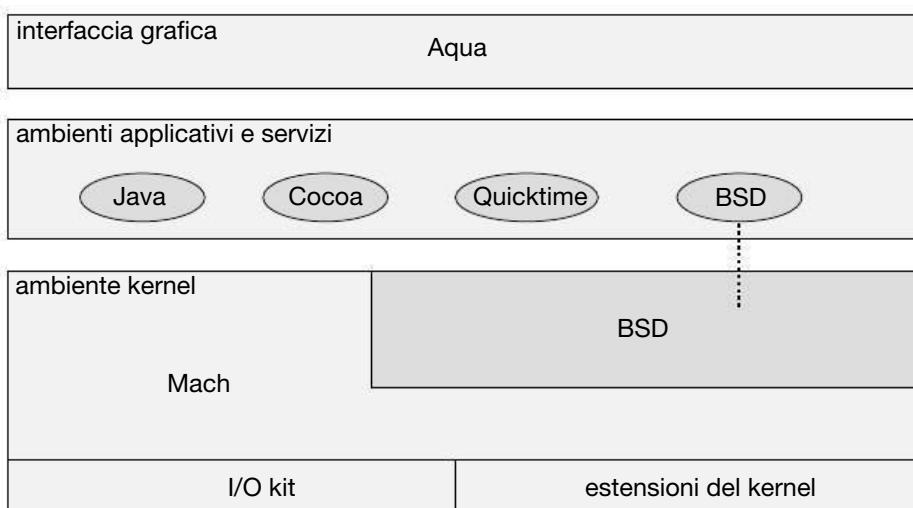


Figura 2.16 La struttura di Mac OS X.

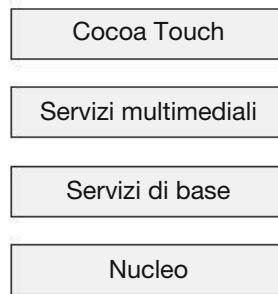


Figura 2.17 Architettura di Apple iOS.

di comando, i servizi legati al file system e alla rete, e una implementazione delle API POSIX, compresa Pthreads. Oltre a Mach e BSD, il kernel possiede un kit di strumenti (*I/O kit*) per lo sviluppo di driver dei dispositivi e di moduli caricabili dinamicamente, detti **estensioni del kernel** nel gergo di Mac OS X. Come si vede dalla Figura 2.16, l’ambiente applicativo BSD può accedere direttamente ai servizi del kernel BSD.

2.7.5.2 iOS

iOS è un sistema operativo mobile progettato da Apple per essere eseguito sul suo smartphone, l'**iPhone**, e sul suo tablet, l'**iPad**. iOS è strutturato sul sistema operativo Mac OS X, con l’aggiunta di funzionalità per dispositivi mobili, ma non esegue direttamente le applicazioni per Mac OS X. La struttura di iOS è illustrata nella Figura 2.17.

Cocoa Touch è un’API per Objective-C che fornisce diversi framework per lo sviluppo di applicazioni che girano su dispositivi iOS. La differenza fondamentale tra Cocoa, citato in precedenza, e Cocoa Touch è che quest’ultimo fornisce il supporto per le caratteristiche hardware peculiari dei dispositivi mobili, come i touch screen.

Lo strato dei **servizi multimediali** fornisce servizi per la grafica, l’audio e il video.

Lo strato dei servizi di base fornisce una varietà di funzioni, tra cui il supporto per il cloud computing e i database. Lo strato inferiore rappresenta il nucleo del sistema operativo, basato sull’ambiente kernel mostrato nella Figura 2.16.

2.7.5.3 Android

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

Android è simile a iOS in quanto è una pila di strati software che fornisce un ricco insieme di ambienti per lo sviluppo di applicazioni mobili. In fondo a questa pila vi è il kernel di Linux modificato da Google e attualmente al di fuori della normale distribuzione delle versioni di Linux.

Linux è utilizzato principalmente per la gestione dei processi e della memoria e per il supporto dei driver di periferica ed è stato ampliato per includere la gestione dei con-

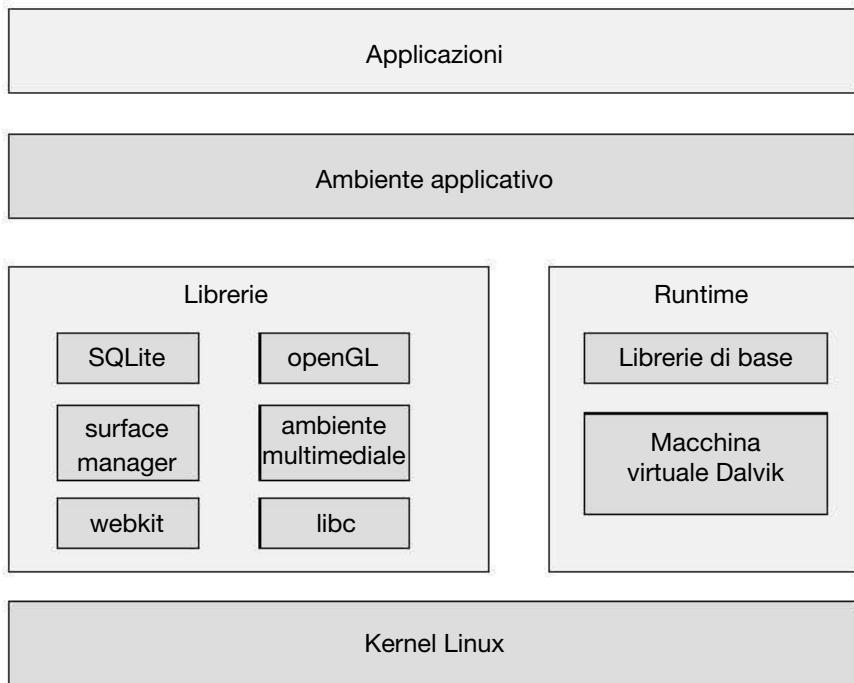


Figura 2.18 Architettura di Google Android.

sumi energetici. L'ambiente di runtime Android include un insieme di librerie di base e la macchina virtuale Dalvik. I progettisti di software per i dispositivi Android sviluppano applicazioni in linguaggio Java, ma invece di utilizzare le API standard di Java usano le API di Android, definite da Google. Le classi Java sono prima compilate in bytecode Java e poi tradotte in un file eseguibile che viene eseguito sulla macchina virtuale Dalvik. Dalvik è stata progettata per Android ed è ottimizzata per i dispositivi mobili con memoria limitata e ridotta capacità di elaborazione della CPU.

L'insieme di librerie disponibili per le applicazioni Android comprende ambienti per lo sviluppo di browser web (webkit), ambienti di supporto di database (SQLite) e ambienti multimediali. La libreria libc è simile alla libreria C standard, ma è molto più piccola ed è stata progettata per le CPU più lente, tipiche dei dispositivi mobili.

2.8 Debugging dei sistemi operativi

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

2.8.1 Analisi dei malfunzionamenti

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

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

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

2.8.2 Regolazione delle prestazioni

Abbiamo detto in precedenza che la regolazione delle prestazioni ha lo scopo di migliorare le prestazioni eliminando i colli di bottiglia. Per identificare eventuali colli di bottiglia dobbiamo essere in grado di monitorare le prestazioni del sistema. A tale scopo il sistema operativo deve comprendere strumenti per effettuare misurazioni sul comportamento del sistema e mostrare i risultati. In molti casi il sistema operativo produce delle tracce del comportamento del sistema; tutti gli eventi di rilievo sono descritti indicandone l'ora e i parametri importanti e sono scritti in un file. Successivamente, un programma di analisi può esaminare il file di log al fine di determinare le prestazioni

LA LEGGE DI KERNIGHAN

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

del sistema e di identificarne colli di bottiglia e inefficienze. Le stesse tracce possono essere utilizzate come input per la simulazione di una miglioria al sistema operativo e inoltre possono contribuire alla scoperta di errori nel comportamento dello stesso.

Un altro approccio per l'ottimizzazione delle prestazioni utilizza strumenti interattivi ad hoc che consentono agli utenti e agli amministratori di interrogare lo stato dei vari componenti del sistema alla ricerca di colli di bottiglia. Uno di questi strumenti utilizza il comando UNIX `top` per mostrare le risorse di sistema impiegate, nonché un elenco ordinato dei principali processi che utilizzano le risorse. Altri strumenti mostrano lo stato dell'I/O su disco, la memoria allocata e il traffico di rete. Gli autori di questi strumenti dotati di un'unica finalità provano a indovinare le necessità dell'utente che analizza il sistema e offrono queste informazioni.

Il **task manager** di Windows è uno strumento simile per i sistemi Windows. Il task manager include informazioni sulle applicazioni e i processi in esecuzione, sull'utilizzo della CPU e della memoria, e le statistiche di rete. Una schermata del task manager appare nella Figura 2.19.

Rendere i sistemi operativi esistenti più facili da comprendere e semplificare il debugging e la regolazione delle prestazioni sono un'area attiva della ricerca e dell'implementazione dei sistemi. Una nuova generazione di strumenti di analisi delle

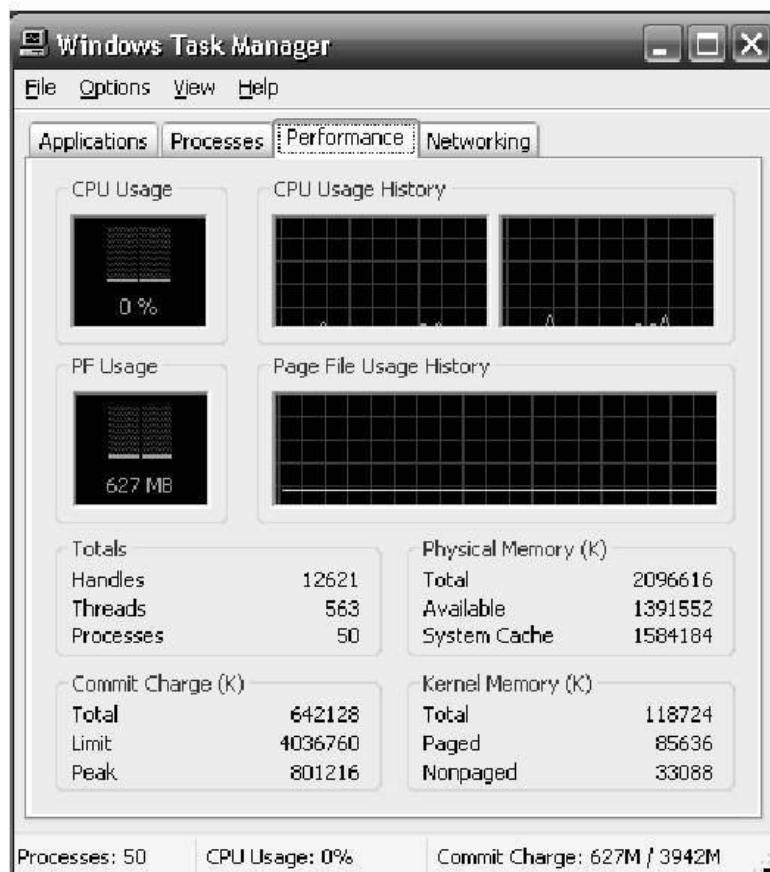


Figura 2.19 Il task manager di Windows.

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                                U
  0 -> _XEventsQueued                             U
  0 -> _X11TransBytesReadable                      U
  0 <- _X11TransBytesReadable                      U
  0 -> _X11TransSocketBytesReadable                U
  0 <- _X11TransSocketBytesreadable               U
  0 -> ioctl                                      U
  0 -> ioctl                                      K
  0 -> getf                                       K
  0 -> set_active_fd                            K
  0 <- set_active_fd                            K
  0 -> getf                                       K
  0 -> get_udatamodel                          K
  0 <- get_udatamodel                          K
...
  0 -> releasef                                 K
  0 -> clear_active_fd                         K
  0 <- clear_active_fd                         K
  0 -> cv_broadcast                           K
  0 <- cv_broadcast                           K
  0 -> releasef                                 K
  0 <- ioctl                                    K
  0 -> ioctl                                    U
  0 <- XEventsQueued                           U
  0 <- XEventsQueued                           U
```

Figura 2.20 DTrace, su Solaris 10, segue una chiamata di sistema all'interno del kernel.

prestazioni situati nel kernel ha introdotto notevoli miglioramenti nel modo in cui questo obiettivo può essere raggiunto. Introduciamo ora un esempio rilevante di tali strumenti: l'utilità per il tracciamento dinamico DTrace in Solaris 10.

2.8.3 DTrace

DTrace è un'utilità che aggiunge dinamicamente delle sonde al sistema operativo, sia nei processi utente sia nel kernel. Queste sonde possono essere interrogate attraverso il linguaggio di programmazione D per determinare una quantità stupefacente di informazioni sul kernel, sullo stato del sistema e sulle attività dei processi. Per esempio, nella Figura 2.20 si segue il comportamento di un'applicazione mentre esegue una chiamata di sistema (`ioctl()`) e si mostrano le chiamate di funzioni all'interno del kernel che vengono eseguite per completare la chiamata di sistema. Le linee che terminano con “U” sono eseguite in modalità utente, quelle che terminano con “K” in modalità kernel. È pressoché impossibile eseguire il debugging dell'interazione tra livello utente e codice kernel senza avere a disposizione un insieme di strumenti che capiscano entrambi i tipi di codice e possano tenere traccia di questa interazione con strumenti appropriati.

Affinché un tale gruppo di strumenti risulti veramente utile esso deve essere in grado di eseguire il debugging di tutte le aree del sistema, incluse quelle scritte originalmente senza prendere in considerazione il debugging, e deve poterlo fare senza con-

dizionare l'affidabilità del sistema. Questo strumento deve avere inoltre un impatto minimo sulle prestazioni: idealmente, non dovrebbe avere alcun impatto mentre non è in funzione e un impatto minimo durante l'utilizzo. L'utilità **DTrace** soddisfa questi requisiti offrendo uno strumento di debugging dinamico, sicuro e a basso impatto.

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

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

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

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

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

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

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

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

```

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

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

```

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

Figura 2.21 Risultato del codice D.

La Figura 2.21 mostra il risultato del programma, cioè quali sono i processi e per quanto tempo (in nanosecondi) occupano il processore.

Poiché DTrace fa parte della versione open-source OpenSolaris del sistema operativo Solaris 10, è stato incorporato in altri sistemi operativi in casi in cui non vi siano conflitti fra le licenze. Per esempio, DTrace è già stato aggiunto a Mac OS X e a FreeBSD. È probabile che esso si diffonderà ulteriormente in futuro per merito delle sue capacità uniche. Anche altri sistemi operativi, in special modo i derivati da Linux, si stanno incorporando funzionalità di tracciamento del kernel. Altri sistemi stanno cominciando a includere strumenti di monitoraggio delle prestazioni e di tracciamento derivati dalla ricerca in vari istituti, come nel caso del progetto Paradyn.

2.9 Generazione di sistemi operativi

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

I sistemi operativi sono normalmente distribuiti per mezzo di dischi, CD-ROM o DVD-ROM, o come immagine ISO, ossia sotto forma di un file con il formato di un CD-ROM o DVD-ROM. Per generare un sistema è necessario usare un programma speciale che può leggere da un file o richiedere all'operatore le informazioni riguardanti la configurazione specifica del sistema o anche esplorare il sistema elaborativo per determinarne i componenti. In questo modo devono essere raccolte le seguenti informazioni.

- La CPU che si deve impiegare e le opzioni installate (instruction set esteso, aritmetica in virgola mobile, e così via). Nel caso di sistemi multiprocessore occorre anche descrivere ciascuna CPU.

- Come verrà formattato il disco d'avvio? In quante sezioni o “partizioni” verrà suddiviso e che cosa ci sarà in ciascuna partizione?
- La quantità di memoria disponibile. Alcuni sistemi determinano autonomamente questi valori, accedendo a tutte le locazioni della memoria, fino alla generazione di un errore di indirizzo illegale. Questa procedura definisce l'indirizzo legale finale e quindi la quantità di memoria disponibile.
- I dispositivi disponibili. Il sistema deve conoscere come indirizzare ogni dispositivo (numero di dispositivo), deve conoscere il numero del codice d'interruzione del dispositivo, il tipo e il modello del dispositivo e tutte le sue caratteristiche specifiche.
- Le opzioni del sistema operativo richieste o i valori dei parametri che è necessario usare. Queste informazioni possono contenere il numero delle aree di memoria da usare per le operazioni di I/O e la loro dimensione, l'algoritmo di scheduling della CPU richiesto, il numero massimo di processi da sostenere, e così via.

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

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

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

2.10 Avvio del sistema

Dopo che un sistema operativo è stato scritto, bisogna renderlo utilizzabile da parte dell'hardware. Ma come fa l'hardware dell'elaboratore a sapere dove si trova il kernel e a caricarlo? La procedura d'avviamento di un calcolatore con il caricamento del

kernel è nota come **avviamento** (*booting*) del sistema: nella maggior parte dei calcolatori un piccolo segmento di codice, noto come **programma d'avvio** (*bootstrap program*) o **caricatore d'avvio** (*bootstrap loader*), che individua il kernel, lo carica in memoria e ne avvia l'esecuzione. Alcuni sistemi, come i PC, eseguono tale compito in due fasi: un caricatore d'avvio molto semplice preleva dal disco un più complesso programma d'avvio, che a sua volta carica il kernel.

Quando una CPU viene resettata – per esempio, quando l'elaboratore viene acceso o riavviato – il registro delle istruzioni è caricato con una locazione di memoria predefinita, da cui ha inizio l'esecuzione. Il programma di avvio inizia da questa locazione. Esso è contenuto in una **memoria a sola lettura** (**read-only memory**, ROM), poiché non si conosce lo stato della RAM all'avvio del sistema; inoltre, la ROM presenta il vantaggio di non dover essere inizializzata e di essere immune ai virus.

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

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

Per sistemi operativi di grandi dimensioni (quasi tutti quelli a carattere generale, come Windows, Mac OS X, UNIX) o per sistemi che cambiano di frequente, il caricatore di avvio è memorizzato nel firmware e il sistema operativo risiede su disco. In questo caso, il programma di avvio applica gli strumenti di diagnosi e utilizza una parte di codice per la lettura di un blocco singolo che occupa una locazione fissa del disco (per esempio, il blocco zero); quindi, lo trasferisce in memoria per eseguire il codice da quel **blocco di avvio** (*boot block*). Il programma custodito dal blocco di avvio può essere abbastanza complesso per caricare in memoria l'intero sistema operativo e dare avvio alla sua esecuzione. Più spesso, è un programma semplice (deve risiedere in un singolo blocco del disco) che conosce unicamente la lunghezza e l'in-

dirizzo sul disco del codice residuo di cui è composto l'intero programma d'avvio. **GRUB** è un esempio di programma di bootstrap open-source per sistemi Linux. Tutto il codice di avviamento su disco, e il sistema operativo stesso, possono essere sostituiti facilmente, scrivendone nuove versioni sul disco. Un disco che contenga una partizione di avvio è chiamato **disco di avvio** (*boot disk*) o **disco di sistema**; si veda il Paragrafo 10.5.1 sull'argomento.

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

2.11 Sommario

I sistemi operativi offrono diversi servizi: al livello più basso, le chiamate di sistema permettono al programma in esecuzione di fare richieste direttamente al sistema operativo; a un livello superiore, l'interprete dei comandi (o *shell*) mette a disposizione un meccanismo che consente a un utente di effettuare una richiesta senza scrivere un programma. I comandi possono provenire da file, in un'esecuzione a lotti (*batch*) oppure direttamente da un terminale o un'interfaccia grafica, in modo interattivo o a partizione del tempo. I programmi di sistema offrono agli utenti i servizi più comuni.

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

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

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

Una volta che un sistema operativo è progettato si deve procedere allo sviluppo. Ormai i sistemi operativi sono quasi tutti scritti in un linguaggio per lo sviluppo di sistemi o in un linguaggio di alto livello; questa caratteristica facilita realizzazione, manutenzione e portabilità.

Un sistema grande e complesso come un sistema operativo moderno deve essere progettato con cura. La modularità è importante. La progettazione di un sistema come una sequenza di strati o utilizzando un microkernel è considerata una buona tecnica. Molti sistemi operativi supportano oggi i moduli caricati dinamicamente, che per-

mettono di aggiungere funzionalità a un sistema operativo mentre è in esecuzione. Generalmente i sistemi operativi adottano un approccio ibrido che combina diverse tipologie di strutture.

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

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

Esercizi di ripasso

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

Esercizi

- 2.12** I servizi e le funzioni offerti da un sistema operativo possono essere divisi in due categorie. Procedete a una loro breve descrizione, analizzandone le differenze.
- 2.13** Descrivete tre metodi generali per passare parametri al sistema operativo.
- 2.14** Descrivete come si possa ottenere un profilo statistico del tempo consumato da un programma per eseguire le differenti parti del proprio codice. Argomentate l'importanza di simili profili statistici.
- 2.15** Quali sono le cinque attività principali di un sistema operativo relative alla gestione dei file?
- 2.16** Quali sono i vantaggi e gli svantaggi di usare la medesima interfaccia alle chiamate di sistema sia per i file sia per i dispositivi?
- 2.17** Sarebbe possibile per l'utente sviluppare un nuovo interprete dei comandi utilizzando le chiamate di sistema offerte dal sistema operativo?
- 2.18** Quali sono i due modelli della comunicazione tra processi? Quali i loro punti di forza e di debolezza?
- 2.19** Perché è auspicabile separare i meccanismi dalle politiche?
- 2.20** Talvolta è difficile realizzare un'architettura a strati se due componenti del sistema operativo dipendono l'uno dall'altro. Identificate una situazione in cui non risulti immediatamente evidente come stratificare due componenti del sistema che hanno funzionalità strettamente connesse.
- 2.21** Quale vantaggio si riscontra nell'architettura orientata al microkernel? In che modo interagiscono i programmi utenti e i servizi del sistema in tale architettura? Quali sono gli svantaggi?
- 2.22** Quali sono i vantaggi dell'utilizzo di moduli del kernel caricabili dinamicamente?
- 2.23** In che cosa sono simili iOS e Android? In che cosa differiscono?
- 2.24** Spiegate perché i programmi Java in esecuzione su sistemi Android non usano le API e la macchina virtuale standard di Java.
- 2.25** Il sistema operativo sperimentale Synthesis ha un assemblatore incorporato nel kernel. Per ottimizzare le prestazioni delle chiamate di sistema, il kernel assembla le procedure nello spazio del kernel, al fine di ridurre al minimo il percorso che le chiamate di sistema devono seguire attraverso il kernel. Tale metodo è in antitesi al metodo stratificato che, per rendere più semplice la costruzione di un sistema operativo, determina un prolungamento del percorso delle chiamate di sistema attraverso il kernel. Valutate i pro e i contro di tale metodo nella progettazione di un kernel e nell'ottimizzazione delle prestazioni di un sistema.

Problemi di programmazione

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

Progetti di programmazione

Moduli del kernel di Linux

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

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

Parte I - Creazione di moduli del kernel

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

```
lsmod
```

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

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

```

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

/* Questa funzione viene chiamata quando viene caricato
   il modulo. */
int simple_init (void)
{
    printk(KERN_INFO "Loading Module\n");
    return 0;
}

/* Questa funzione viene chiamata quando il modulo viene
   rimosso. */
void simple_exit(void)
{
    printk(KERN_INFO "Removing Module\n");
}

/* Macro per la registrazione di ingresso e di uscita
   del modulo. */
module_init(simple_init);
module_exit(simple_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");

```

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

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

```

module_init()
module_exit()

```

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

Le ultime linee, `MODULE_LICENSE()`, `MODULE_DESCRIPTION()` e `MODULE_AUTHOR()`, forniscono dettagli riguardanti la licenza software, la descrizio-

ne del modulo, e l'autore. Queste informazioni non sono importanti per i nostri scopi, ma le includiamo perché è una pratica comune nello sviluppo di moduli del kernel.

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

```
make
```

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

Caricamento e rimozione dei moduli del kernel

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

```
sudo insmod simple.ko
```

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

```
dmesg
```

Si dovrebbe vedere il messaggio "Loading Module."

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

```
sudo rmmmod simple
```

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

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

```
sudo dmesg -c
```

Parte I – Esercizio

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

Parte II – Strutture dati del kernel

La seconda parte di questo progetto prevede la modifica del modulo del kernel in modo da utilizzare una lista concatenata.

Nel Paragrafo 1.10 abbiamo trattato varie strutture dati utilizzate comunemente nei sistemi operativi. Il kernel di Linux fornisce molte di queste strutture. Vediamo

come utilizzare la lista circolare doppiamente concatenata, che è disponibile per gli sviluppatori del kernel. La maggior parte di ciò che tratteremo si trova nel codice sorgente di Linux, in questo caso nel file di `include <linux/list.h>`. Consigliamo di esaminare il file man mano che si procede attraverso i seguenti passaggi.

Inizialmente, è necessario definire una struttura contenente gli elementi che devono essere inseriti nella lista concatenata. La seguente `struct` del linguaggio C definisce i compleanni:

```
struct birthday {
    int day;
    int month;
    int year;
    struct list_head list;
}
```

Si presti attenzione al campo `struct list_head list`. La struttura `list_head` è definita nel file `<linux/types.h>` e serve a incorporare la lista concatenata all'interno dei nodi che compongono la lista. La struttura `list_head` è piuttosto semplice: è formata da due soli campi, `next` e `prev`, che puntano all'elemento precedente e a quello successivo della lista. Incorporando la lista concatenata all'interno della struttura, Linux consente di gestire la struttura dati mediante una serie di funzioni macro.

Inserimento di elementi nella lista concatenata

Possiamo dichiarare un oggetto `list_head` da usare come riferimento alla testa della lista utilizzando la macro `LIST_HEAD()` in questo modo:

```
static LIST_HEAD(birthday_list);
```

Questa macro definisce e inizializza la variabile `birthday_list`, di tipo `struct list_head`.

Possiamo creare e inizializzare le istanze di `struct birthday` come segue:

```
struct birthday *person;

person = kmalloc(sizeof(*person), GFP_KERNEL);
person->day = 2;
person->month= 8;
person->year = 1995;
INIT_LIST_HEAD(&person->list);
```

La funzione `kmalloc()` è l'equivalente a livello kernel della funzione `malloc()`, utilizzata a livello utente per l'allocazione di memoria, a eccezione del fatto che con la `kmalloc()` viene allocata memoria nell'area del kernel. (Il flag `KERNEL_GFP` indica l'allocazione di memoria da parte di una funzione del kernel). La macro `INIT_LIST_HEAD()` inizializza il campo `list` nella struttura `struct birthday`. Possiamo quindi aggiungere l'istanza alla fine della lista concatenata utilizzando la macro `list_add_tail`:

```
list_add_tail(&person->list, &birthday_list);
```

Attraversamento della lista concatenata

L'attraversamento della lista concatenata comporta l'uso della macro `list_for_each_entry()`, che accetta tre parametri:

- un puntatore alla struttura sulla quale si effettua l'iterazione;
- un puntatore alla testa della lista su cui si effettua l'iterazione;
- il nome della variabile contenente la struttura `list_head`.

Il codice seguente illustra questa macro:

```
struct birthday *ptr;

list_for_each_entry(ptr, &birthday_list, list) {
    /* durante ogni iterazione ptr punta */
    /* alla struttura birthday successiva */
}
```

Rimozione di elementi dalla lista concatenata

La rimozione di elementi dalla lista comporta l'uso della macro `list_del()`, a cui viene passato un puntatore a `struct list_head`:

```
list_del(struct list_head *element)
```

Questo passaggio rimuove *element* dalla lista mantenendo la struttura del resto della lista.

L'approccio più semplice per la rimozione di tutti gli elementi di una lista concatenata è forse quello di rimuovere ogni elemento mentre la si attraversa. La macro `list_for_each_entry_safe()` si comporta in modo simile alla `list_for_each_entry()`, ma riceve un argomento aggiuntivo che mantiene il valore del puntatore all'elemento successivo a quello da eliminare. (Ciò è necessario per preservare la struttura della lista). Il seguente codice di esempio illustra questa macro:

```
struct birthday *ptr, *next

list_for_each_entry_safe(ptr,next,&birthday_list,list) {
    /* durante ogni iterazione ptr punta */
    /* alla struttura birthday successiva */
    list_del(&ptr->list);
    kfree(ptr);
}
```

Si noti che, dopo l'eliminazione di ogni elemento, restituiamo al kernel la memoria precedentemente allocata con `kmalloc()` per mezzo della funzione `kfree()`. Una gestione attenta della memoria, che comprende il rilascio della memoria per prevenire perdite (*memory leaks*), è di fondamentale importanza nello sviluppo di codice a livello kernel.

Parte II – Esercizio

Create, nel punto di ingresso del modulo, una lista concatenata contenente cinque elementi `struct birthday`. Attraversate la lista concatenata e scrivete il suo contenuto nel buffer di log del kernel. Richiamate il comando `dmesg` e verificate, una volta che il modulo del kernel è stato caricato, se la lista è stata correttamente costruita.

Nel punto di uscita del modulo, eliminate gli elementi dalla lista concatenata e restituite la memoria al kernel. Ancora una volta, usate il comando `dmesg` per verificare se la lista è stata rimossa quando il modulo del kernel è stato scaricato.

Note bibliografiche

L’orientamento stratificato alla progettazione dei sistemi operativi è stato sostenuto da [Dijkstra 1968]. [Brinch-Hansen 1970] è stato uno dei primi sostenitori della costruzione di un sistema operativo intorno a un nucleo (il kernel), sulla base del quale sviluppare sistemi più completi. [Tarkoma e Lagerspetz 2011] forniscono una panoramica dei vari sistemi operativi mobili, tra cui Android e iOS.

L’MS-DOS, Versione 3.1, è descritto in [Microsoft 1986]. Windows NT e Windows 2000 sono esaminati in [Solomon 1998], nonché in [Solomon e Russinovich 2000]. Il sistema Windows XP è trattato in [Russinovich e Solomon 2009]. [Hart 2005] discute in dettaglio la programmazione di sistemi Windows. Il Berkeley UNIX (BSD) è descritto in [McKusick et al. 1996]. [Love 2010] e [Mauerer 2008] descrivono a fondo il kernel di Linux. In particolare, [Love 2010] tratta i moduli e le strutture dati del kernel. Diversi sistemi UNIX, tra cui Mach, sono discussi in dettaglio in [Vahalia (1996)]. Il Mac OS X è presentato all’indirizzo <http://www.apple.com/macosx> e in [Singh 2007]. Solaris è dettagliatamente descritto da [Mauro e McDougall 2001].

DTrace è discusso in [Gregg e Mauro 2011]. Il codice sorgente di DTrace è disponibile presso <http://src.opensolaris.org/source/>.

Bibliografia

- [Brinch-Hansen 1970] P. Brinch-Hansen, “The Nucleus of a Multiprogramming System”, *Communications of the ACM*, Volume 13, Nr. 4, p. 238–241 e 250, 1970.
- [Dijkstra 1968] E. W. Dijkstra, “The Structure of the THE Multiprogramming System”, *Communications of the ACM*, Volume 11, Nr. 5, p. 341–346, 1968.
- [Gregg e Mauro 2011] B. Gregg e J. Mauro, *DTrace—Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*, Prentice Hall, 2011.
- [Hart 2005] J. M. Hart, *Windows System Programming*, 3° ed., Addison-Wesley, 2005.
- [Love 2010] R. Love, *Linux Kernel Development*, 3° ed., Developer’s Library, 2010.
- [Mauerer 2008] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons, 2008.

- [McDougall e Mauro 2007]** R. McDougall e J. Mauro, *Solaris Internals*, 2º ed., Prentice Hall, 2007.
- [McKusick et al. 1996]** M. K. McKusick, K. Bostic, e M. J. Karels, *The Design and Implementation of the 4.4 BSD UNIX Operating System*, John Wiley e Sons, 1996.
- [Microsoft 1986]** *Microsoft MS-DOS User's Reference and Microsoft MS-DOS Programmer's Reference*, Microsoft Press, 1986.
- [Russinovich e Solomon 2009]** M. E. Russinovich e D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*, 5º ed., Microsoft Press, 2009.
- [Singh 2007]** A. Singh, *Mac OS X Internals: A Systems Approach*, Addison-Wesley, 2007.
- [Solomon 1998]** D. A. Solomon, *Inside Windows NT*, 2º ed., Microsoft Press, 1998.
- [Solomon e Russinovich 2000]** D. A. Solomon e M. E. Russinovich, *Inside Microsoft Windows 2000*, 3º ed., Microsoft Press, 2000.
- [Tarkoma e Lagerspetz 2011]** S. Tarkoma e E. Lagerspetz, “Arching over the Mobile Computing Chasm: Platforms and Runtimes”, *IEEE Computer*, Volume 44, p. 22–28, 2011.
- [Vahalia 1996]** U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall, 1996.

Gestione dei processi

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

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

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

Il sistema operativo è responsabile di diversi aspetti importanti relativi alla gestione dei processi e dei thread di sistema e utenti: creazione e cancellazione, scheduling, realizzazione dei meccanismi di sincronizzazione e comunicazione, gestione delle situazioni di stallo.

CAPITOLO

3

OBIETTIVI DEL CAPITOLO

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

Processi

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

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

3.1 Concetto di processo

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

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

3.1.1 Processo

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

Sottolineiamo che un programma di per sé non è un processo; un programma è un’entità *passiva*, come un file su disco contenente una lista di istruzioni (normal-

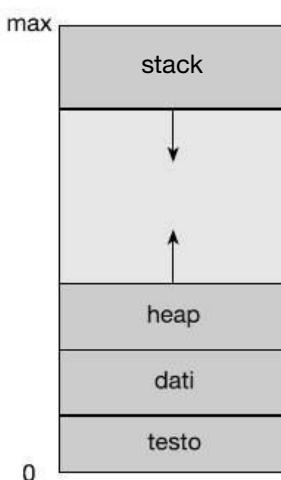


Figura 3.1 Processo in memoria.

mente chiamato **file eseguibile**), mentre un processo è un'entità *attiva*, con un contatore di programma che specifica qual è l'istruzione successiva da eseguire e un insieme di risorse associate. Un programma diventa un processo allorquando il file eseguibile è caricato in memoria. Due tecniche comuni per ottenere questo effetto sono il doppio clic sull'icona del file eseguibile e la digitazione del nome del file eseguibile nella riga di comando (scrivendo per esempio: `prog.exe` oppure `a.out`).

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

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

```
java Program
```

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

3.1.2 Stato del processo

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

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

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

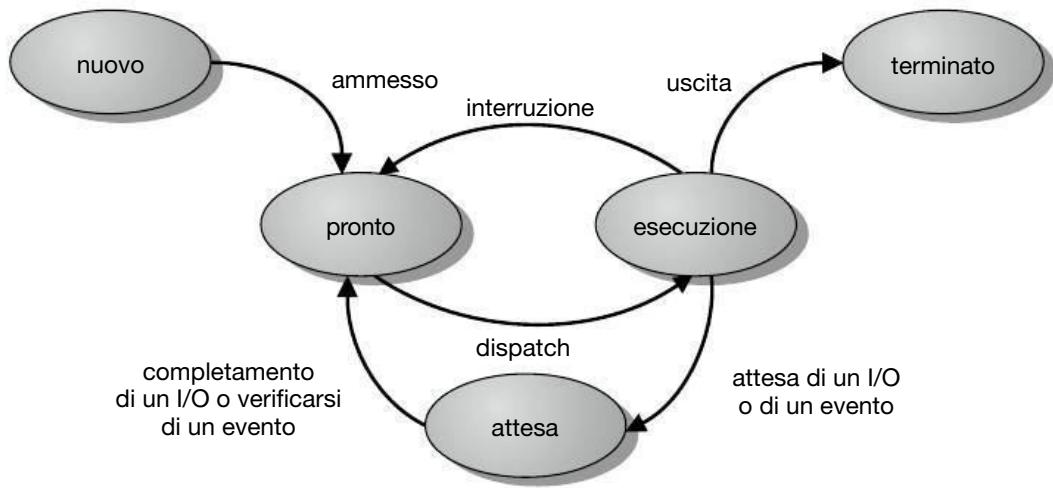


Figura 3.2 Diagramma di transizione degli stati di un processo.

3.1.3 Blocco di controllo dei processi

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

- **Stato del processo.** Lo stato può essere: nuovo, pronto, esecuzione, attesa, arresto, e così via.
- **Contatore di programma.** Il contatore di programma contiene l'indirizzo della successiva istruzione da eseguire per tale processo.
- **Registri della CPU.** I registri variano in numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, registri indice, puntatori alla cima dello stack (*stack pointer*), registri d'uso generale e registri contenenti i codici di condizione (*condition codes*). Quando si verifica un'interruzione della CPU si de-

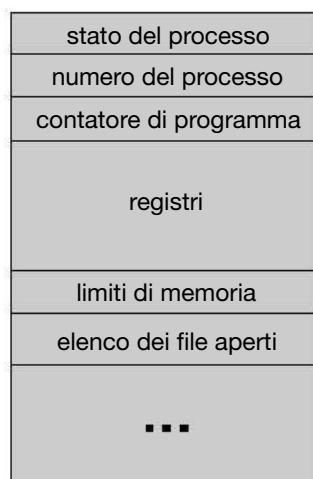


Figura 3.3 Blocco di controllo di un processo (PCB).

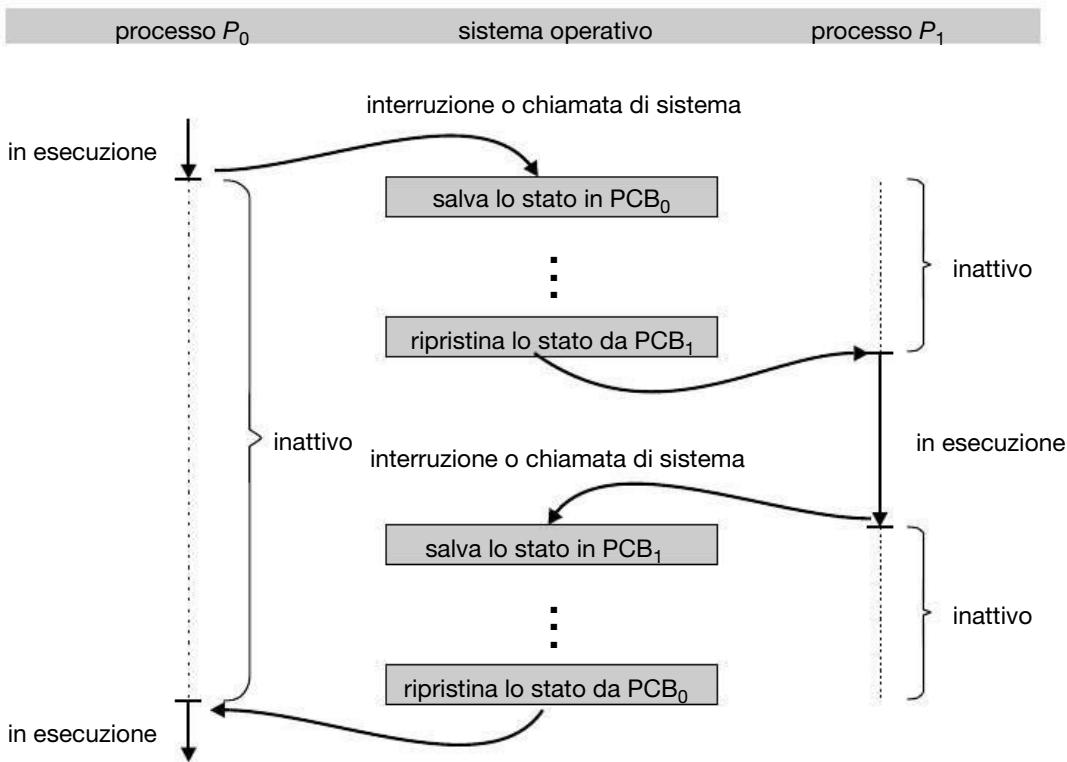


Figura 3.4 La CPU può essere commutata tra i processi.

vono salvare tutte queste informazioni insieme con il contatore di programma, in modo da permettere la corretta esecuzione del processo in un momento successivo (Figura 3.4).

- **Informazioni sullo scheduling di CPU.** Queste informazioni comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling (nel Capitolo 6 si descrive lo scheduling dei processi).
- **Informazioni sulla gestione della memoria.** Queste informazioni possono includere elementi quali il valore dei registri di base e di limite, le tabelle delle pagine o le tabelle dei segmenti, a seconda del sistema di gestione della memoria usato dal sistema operativo (Capitolo 8).
- **Informazioni di accounting.** Queste informazioni comprendono la quota di uso della CPU e il tempo d'utilizzo della stessa, i limiti di tempo, i numeri dei processi, e così via.
- **Informazioni sullo stato dell'I/O.** Queste informazioni comprendono la lista dei dispositivi di I/O assegnati a un determinato processo, l'elenco dei file aperti, e così via.

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

3.1.4 Thread

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

3.2 Scheduling dei processi

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

3.2.1 Code di scheduling

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

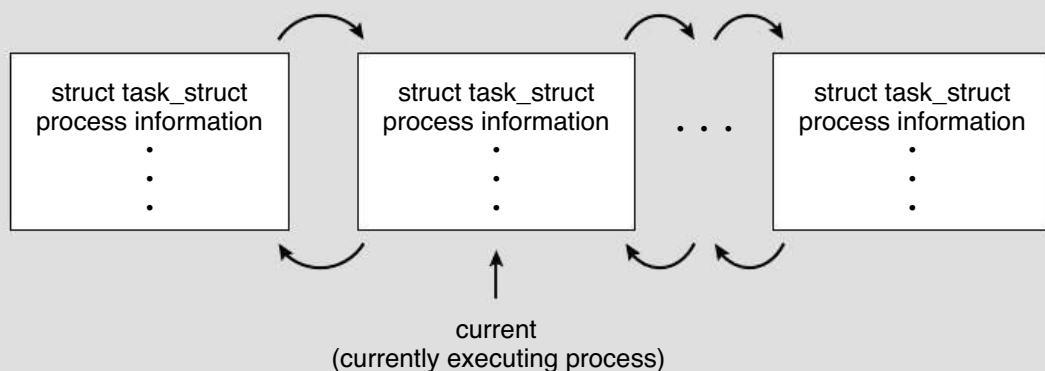
Il sistema operativo ha anche altre code. Quando si assegna la CPU a un processo, quest'ultimo rimane in esecuzione per un certo tempo e prima o poi termina, viene interrotto, oppure si ferma nell'attesa di un evento particolare, come il completamento di una richiesta di I/O. Una richiesta di I/O può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

RAPPRESENTAZIONE DEI PROCESSI IN LINUX

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

```
long state; /* stato del processo */
struct sched_entity se; /* informazioni per lo scheduling */
struct task_struct *parent; /* processo padre */
struct list_head children; /* processi figlio*/
struct files_struct *files; /* lista dei file aperti */
struct mm_struct *mm; /* spazio degli indirizzi del processo */
```

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



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

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

nibilità di un particolare dispositivo di I/O si chiama **coda del dispositivo**; ogni dispositivo ha la propria coda (Figura 3.5).

Una comune rappresentazione dello scheduling dei processi è data da un **diagramma di accodamento** come quello illustrato nella Figura 3.6. Ogni riquadro rappresenta una coda. Sono presenti due tipi di coda: la ready queue e un insieme di code di dispositivi. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso di processi nel sistema.

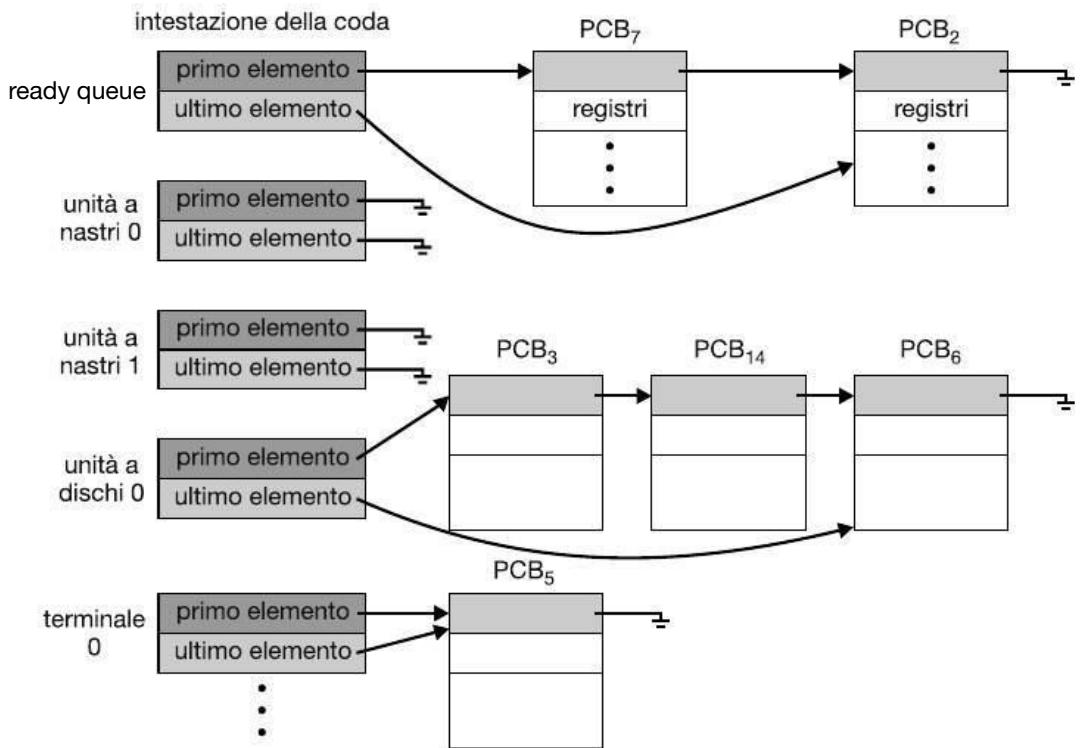


Figura 3.5 Ready queue e diverse code dei dispositivi di I/O.

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

- il processo può emettere una richiesta di I/O e quindi essere inserito in una coda di I/O;

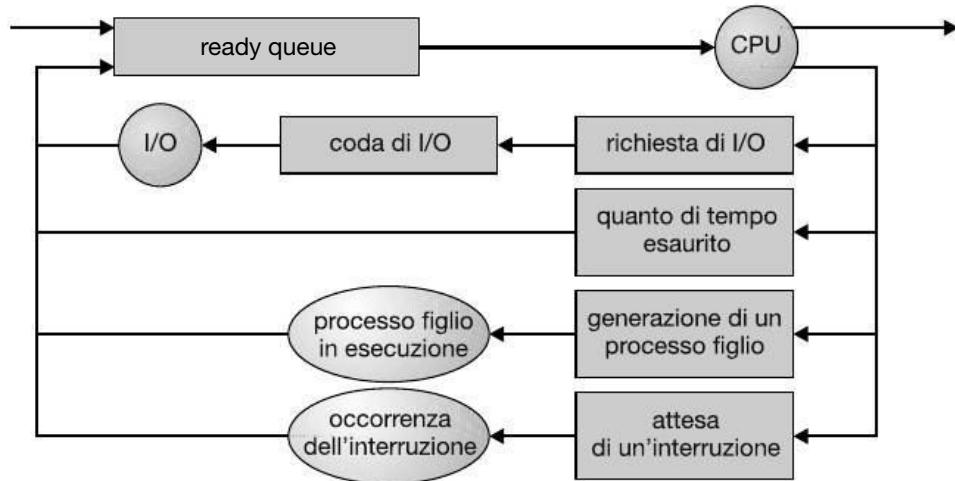


Figura 3.6 Diagramma di accodamento per lo scheduling dei processi.

- il processo può creare un nuovo processo figlio e attenderne la terminazione;
- il processo può essere rimosso forzatamente dalla CPU a causa di un'interruzione, ed essere reinserito nella coda dei processi pronti.

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

3.2.2 Scheduler

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

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

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

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

È importante che lo scheduler a lungo termine faccia un'accurata selezione dei processi. In generale, la maggior parte dei processi si può caratterizzare come avente una prevalenza di I/O, o come avente una prevalenza d'elaborazione. Un **processo con prevalenza di I/O** (*I/O bound*) impiega la maggior parte del proprio tempo nell'esecuzione di operazioni di I/O. Un **processo con prevalenza d'elaborazione**

(*CPU bound*), viceversa, richiede poche operazioni di I/O e impiega la maggior parte del proprio tempo nelle elaborazioni. È fondamentale che lo scheduler a lungo termine selezioni una buona **combinazione di processi** I/O bound e CPU bound. Se tutti i processi fossero con prevalenza di I/O, la coda dei processi pronti sarebbe quasi sempre vuota e lo scheduler a breve termine resterebbe pressoché inattivo. Se tutti i processi fossero con prevalenza d’elaborazione, la coda d’attesa per l’I/O sarebbe quasi sempre vuota, i dispositivi non sarebbero utilizzati, e il sistema sarebbe sbilanciato anche in questo caso. Le prestazioni migliori sono date da una combinazione equilibrata di processi I/O bound e CPU bound.

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

In alcuni sistemi operativi come quelli in time-sharing, si può introdurre un livello di scheduling intermedio. Questo **scheduler a medio termine** è rappresentato schematicamente nella Figura 3.7. L’idea alla base di un tale scheduler è che a volte può essere vantaggioso eliminare processi dalla memoria (e dalla contesa per la CPU), riducendo il grado di multiprogrammazione del sistema. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta.

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

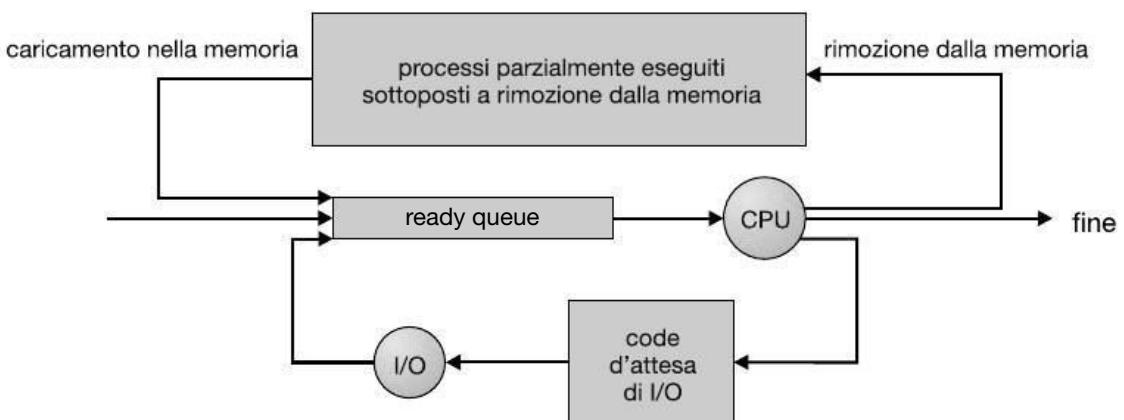


Figura 3.7 Aggiunta di scheduling a medio termine al diagramma di accodamento.

3.2.3 Cambio di contesto

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

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

La durata del cambio di contesto dipende molto dall'architettura; per esempio, alcune CPU (come la Sun UltraSPARC) sono dotate di più set di registri, quindi il cambio di contesto prevede la semplice modifica di un puntatore al gruppo di registri corrente. Naturalmente, se il numero dei processi attivi è maggiore di quello dei set di registri disponibili, il sistema rimedia copiando i dati dei registri nella e dalla memoria, come prima. Inoltre, più complesso è il sistema operativo, più lavoro si deve svolgere durante un cambio di contesto. Come vedremo nel Capitolo 8, l'uso di tecniche avanzate di gestione della memoria può richiedere lo spostamento di ulteriori dati a ogni cambio di contesto. Per esempio, si deve preservare lo spazio d'indirizzi del processo corrente mentre si prepara lo spazio per il processo successivo. Il modo in cui si preserva tale spazio e la relativa quantità di lavoro dipendono dal metodo di gestione della memoria del sistema operativo.

3.3 Operazioni sui processi

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

MULTITASKING NEI SISTEMI PER DISPOSITIVI MOBILI

A causa dei vincoli imposti sui dispositivi mobili, le prime versioni di iOS non fornivano il multitasking per le applicazioni utente; una sola applicazione veniva eseguita in foreground, mentre tutte le altre erano sospese. I processi del sistema operativo erano invece gestiti in multitasking, perché essendo scritti da Apple si comportavano in maniera corretta. A partire da iOS 4 Apple offre una forma limitata di multitasking per le applicazioni utente, consentendo così all'applicazione in foreground di funzionare contemporaneamente a più applicazioni in background. Su un dispositivo mobile, l'applicazione **in foreground** è l'applicazione aperta al momento e che appare sul display. L'applicazione **in background** rimane in memoria, ma non occupa lo schermo del display. L'API iOS 4 fornisce il supporto per il multitasking, permettendo così a un processo di rimanere in esecuzione in background senza essere sospeso. Tuttavia, questa possibilità è limitata e disponibile soltanto per un numero ridotto di tipologie di applicazioni, tra cui

- le applicazioni che eseguono un unico task di lunghezza finita (per esempio il completamento di un download di contenuti da una rete);
- le applicazioni che ricevono le notifiche sul verificarsi di un evento (per esempio un nuovo messaggio di posta elettronica);
- le applicazioni con attività in background di lunga durata (per esempio un lettore audio).

Probabilmente Apple pone un limite al multitasking a causa delle problematiche inerenti la durata della batteria e l'uso della memoria. La CPU ha certamente le caratteristiche per supportare il multitasking, ma Apple sceglie di non approfittare di alcune per poter gestire al meglio l'utilizzo delle risorse.

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

3.3.1 Creazione di un processo

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

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

La Figura 3.8 mostra un tipico albero dei processi del sistema operativo Linux, con il nome e il pid di ogni processo. (Qui usiamo genericamente il termine processo anche se in Linux si preferisce il termine task). Il processo **init** (che ha sempre PID

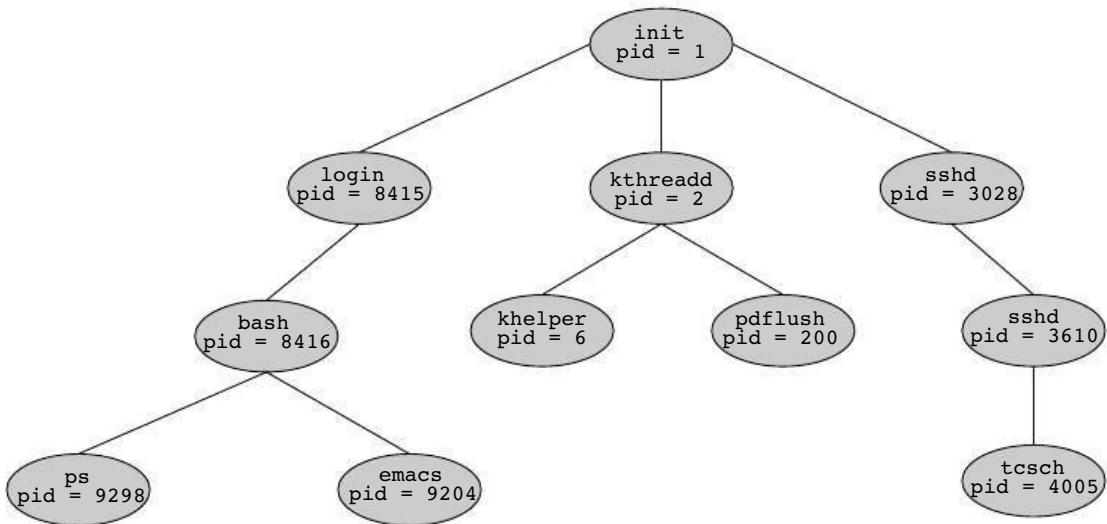


Figura 3.8 Esempio di albero dei processi di un tipico sistema Linux.

uguale a 1) svolge il ruolo di processo padre di tutti i processi utente. Una volta che il sistema si è avviato, il processo `init` può creare vari processi utente, per esempio un server web o per la stampa, un server `ssh`, e processi simili. Nella Figura 3.8 vediamo due figli di `init` - `kthreadd` e `sshd`. Il processo `kthreadd` è responsabile della creazione dei processi aggiuntivi che eseguono attività per conto del kernel (nel nostro caso `khelper` e `pdfflush`). Il processo `sshd` è responsabile della gestione dei client che si connettono al sistema tramite `ssh` (abbreviazione di *secure shell*). Il processo di `login` è responsabile della gestione dei client che accedono direttamente al sistema. In questo esempio, un client ha effettuato l'accesso e utilizza la shell `bash`, a cui è stato assegnato pid 8416. Utilizzando l'interfaccia a riga di comando `bash`, questo utente ha avviato il processo `ps` e l'editor `emacs`.

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

```
ps -el
```

si otterranno informazioni complete su tutti i processi attualmente attivi nel sistema; è facile costruire un albero come quello nella figura identificando ricorsivamente i processi genitore fino a giungere a `init`.

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

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

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

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

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

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

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

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

```

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

int main()
{
pid_t pid;

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

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

return 0;
}

```

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

Il programma C della Figura 3.9 illustra le chiamate di sistema UNIX descritte precedentemente. Abbiamo due processi distinti ciascuno dei quali esegue una copia dello stesso programma. Il valore assegnato alla variabile `pid` è zero nel processo figlio, e un numero intero maggiore di zero (il `pid` del processo figlio) nel processo genitore. Il processo figlio eredita privilegi, attributi di scheduling e alcune risorse, come i file aperti, dal processo genitore. Usando la chiamata di sistema `execlp()` – una versione della chiamata di sistema `exec()` – il processo figlio sovrappone il proprio spazio d’indirizzi con il comando `/bin/ls` di UNIX (che si usa per ottenere l’elenco del contenuto di una directory). Eseguendo la chiamata di sistema `wait()`, il processo genitore attende che il processo figlio termini. Quando ciò accade (implicitamente o esplicitamente mediante l’invocazione di `exit()`), il processo genitore chiude la propria fase d’attesa dovuta alla chiamata di sistema `wait()` e termina usando la chiamata di sistema `exit()`. Questo schema è illustrato nella Figura 3.10.

Naturalmente, non c’è modo di impedire al figlio di non invocare `exec()` e restare in esecuzione come una copia del processo genitore. In questo scenario, il genitore e

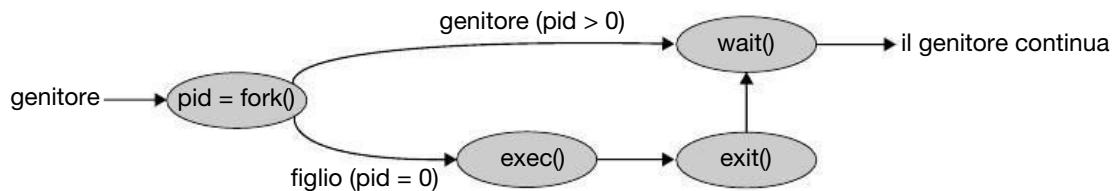


Figura 3.10 Creazione di un processo utilizzando la chiamata di sistema `fork()`.

il figlio sono processi concorrenti che eseguono lo stesso codice. Poiché il figlio è una copia del genitore, ogni processo ha la propria copia dei dati.

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

Il programma nella Figura 3.11, scritto in C, illustra la funzione `CreateProcess()`; essa genera un processo figlio che carica l'applicazione `mspaint.exe`. In questo esempio, i dieci parametri passati a `CreateProcess()` hanno in molti casi il loro valore di default.

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

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

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

```

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

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

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

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

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

```

Figura 3.11 Creazione di un nuovo processo con la API Windows.

3.3.2 Terminazione di un processo

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la chiamata di sistema `exit()`; a questo punto, il processo figlio può riportare un'informazione di stato al processo genitore, che la riceve attraverso la chiamata di sistema `wait()`. Tutte le

risorse del processo, incluse la memoria fisica e virtuale, i file aperti e le aree della memoria per l’I/O, sono liberate dal sistema operativo.

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

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

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

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

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

```
/ * Uscita con stato 1 * /  
exit(1);
```

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

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

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

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

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

Consideriamo ora che cosa accadrebbe se un genitore terminasse senza invocare la `wait()`, lasciando così orfani i suoi figli. Linux e UNIX affrontano questa situazione assegnando al processo `init` il ruolo di nuovo genitore dei processi orfani. (Ricordiamo dalla Figura 3.8 che il processo `init` è la radice della gerarchia dei processi nei sistemi UNIX e Linux). Il processo `init` invoca periodicamente `wait()`, consentendo in tal modo di raccogliere lo stato di uscita di qualsiasi processo orfano e rilasciando il suo PID e la relativa voce nella tabella dei processi.

3.4 Comunicazione tra processi

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

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

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

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di **comunicazione tra processi** (**IPC**, *interprocess communication*). I modelli

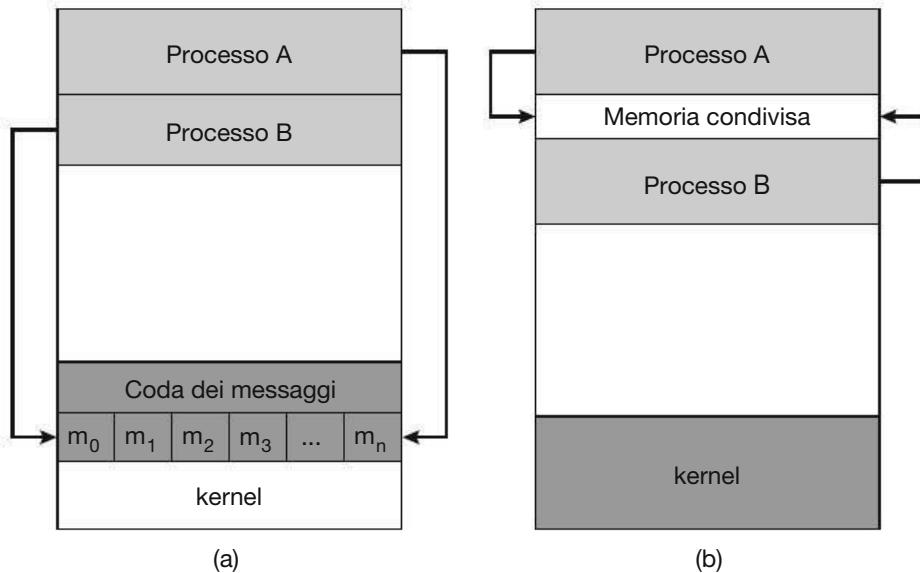


Figura 3.12 Modelli di comunicazione. (a) Scambio di messaggi. (b) Memoria condivisa.

fondamentali della comunicazione tra processi sono due: a **memoria condivisa** e a **scambio di messaggi**. Nel modello a memoria condivisa, si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona. Nel secondo modello la comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti. I due modelli sono messi a confronto nella Figura 3.12.

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

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

Recenti ricerche su sistemi con più core di elaborazione indicano che su tali sistemi lo scambio di messaggi fornisce prestazioni migliori rispetto alla memoria condivisa. La memoria condivisa soffre di problemi di coerenza della cache che sorgono a causa della migrazione dei dati condivisi tra le varie cache. È possibile che al crescere del numero di core vedremo lo scambio di messaggi diventare il meccanismo preferito per la comunicazione tra processi.

Nel resto di questo paragrafo esploriamo con maggior dettaglio i sistemi a memoria condivisa e a scambio di messaggi.

ARCHITETTURA MULTIPROCESSO – IL BROWSER CHROME

Per offrire un'esperienza di navigazione ricca e dinamica, in molti siti web sono presenti contenuti attivi come JavaScript, Flash e HTML5. Purtroppo queste applicazioni web possono anche contenere alcuni bug software che talvolta rallentano i tempi di risposta o provocano il crash del browser web. Non si tratterebbe di un grave problema in un browser che visualizza il contenuto di un solo sito web, ma i browser più moderni supportano la navigazione a schede, che permette di aprire in una singola istanza del browser diversi siti web allo stesso tempo, ogni sito in una scheda separata. Per passare da un sito all'altro, l'utente deve solo fare clic sulla scheda appropriata. Questa disposizione è illustrata di seguito:



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

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

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

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

3.4.1 Sistemi a memoria condivisa

La comunicazione tra processi basata sulla condivisione della memoria richiede che i processi comunicanti allochino una zona di memoria condivisa, di solito residente nello spazio degli indirizzi del processo che la alloca: gli altri processi che desiderano usarla per comunicare dovranno annetterla al loro spazio degli indirizzi. Si ricordi che, normalmente, il sistema operativo tenta di impedire a un processo l'accesso alla memoria di altri processi. La condivisione della memoria richiede che due o più processi raggiungano un accordo per superare questo limite, in modo da poter comuni-

care tramite scritture e letture dell’area condivisa. Il tipo e la collocazione dei dati sono determinati dai processi, e rimangono al di fuori del controllo del sistema operativo. I processi hanno anche la responsabilità di non scrivere nella stessa locazione simultaneamente.

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

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

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

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

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} elemento;

elemento buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```

item next_produced;
while (true) {
    /* produce un elemento in next_produced */
    while (((in + 1) % BUFFER_SIZE) == out); /* non fa niente */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

```

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

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

Il codice per il processo produttore è illustrato nella Figura 3.13, quello per il processo consumatore nella Figura 3.14. Il processo produttore ha una variabile locale `next_produced` contenente il nuovo elemento da produrre. Il processo consumatore ha una variabile locale `next_consumed` in cui si memorizza l'elemento da consumare.

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

Una questione ignorata dalla precedente analisi è il caso in cui sia il produttore sia il consumatore tentano di accedere al buffer concorrentemente. Nel Capitolo 5 si vedrà come sia possibile implementare efficacemente la sincronizzazione tra processi cooperanti nel modello a memoria condivisa.

```

item next_consumed;

while (true) {
    while (in == out)
        ; /* non fa niente */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consuma l'elemento in next_consumed */
}

```

Figura 3.14 Processo consumatore con l'utilizzo della memoria condivisa.

3.4.2 Sistemi a scambio di messaggi

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

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

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

Se i processi P e Q vogliono comunicare devono inviare e ricevere messaggi tra loro; deve, dunque, esistere un **canale di comunicazione** (*communication link*), realizzabile in molti modi. In questo paragrafo non si tratta della realizzazione fisica del canale (come la memoria condivisa, i bus o le reti, illustrati nel Capitolo 17) ma della sua realizzazione logica. Ci sono diversi metodi per realizzare a livello logico un canale di comunicazione e le operazioni `send()` e `receive()`:

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

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

3.4.2.1 Naming

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

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

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

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

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

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

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

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

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

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

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

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

A questo punto, si supponga che i processi P_1 , P_2 e P_3 condividano la mailbox A . Il processo P_1 invia un messaggio ad A , mentre sia P_2 sia P_3 eseguono una `receive()` da A . Sorge il problema di sapere quale processo riceverà il messaggio.

La soluzione dipende dallo schema prescelto:

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

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

Invece, una mailbox posseduta dal sistema operativo ha una vita autonoma, è indipendente e non è legata ad alcun processo particolare. Il sistema operativo offre un meccanismo che permette a un processo le seguenti operazioni:

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

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

3.4.2.2 Sincronizzazione

La comunicazione tra processi avviene attraverso chiamate delle primitive `send()` e `receive()`. Ci sono diverse possibilità nella definizione di ciascuna primitiva. Lo scambio di messaggi può essere **sincrono** (o **bloccante**) oppure **asincrono** (o **non bloccante**). (Per tutto il testo incontreremo i concetti di comportamento sincrono e asincrono in relazione a vari algoritmi del sistema operativo).

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

```

message next_produced;

while (true) {
    /* produce un elemento in next_produced */

    send(next_produced);
}

```

Figura 3.15 Processo produttore con l'utilizzo dello scambio di messaggi.

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

Le Figure 3.15 e 3.16 mostrano queste operazioni.

3.4.2.3 Code di messaggi

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

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

Il caso con capacità zero è talvolta chiamato *sistema a scambio di messaggi senza buffering*; gli altri due, *sistemi con buffering automatico*.

```

message next_consumed;

while (true) {
    receive(next_consumed);
    /* consuma l'elemento in next_consumed */
}

```

Figura 3.16 Processo consumatore con l'utilizzo dello scambio di messaggi.

3.5 Esempi di sistemi di IPC

In questo paragrafo analizzeremo tre sistemi per la comunicazione fra processi: la API POSIX basata sulla memoria condivisa, lo scambio di messaggi nel sistema operativo Mach e concludiamo con Windows, che ha la caratteristica interessante di usare la memoria condivisa come meccanismo per supportare alcune forme di message passing.

3.5.1 Un esempio: memoria condivisa in POSIX

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

La memoria condivisa POSIX è organizzata utilizzando i file mappati in memoria, che associano la regione di memoria condivisa a un file. Un processo deve prima creare un oggetto memoria condivisa con la chiamata di sistema `shm_open()`, come segue:

```
shm_fd = shm_open (name, O_CREAT | O_RDWR, 0666);
```

Il primo parametro specifica il nome dell'oggetto memoria condivisa. I processi che desiderano accedere a questa memoria condivisa devono fare riferimento all'oggetto mediante questo nome. I parametri successivi specificano che l'oggetto memoria condivisa deve essere creato se non esiste ancora (`O_CREAT`) e che l'oggetto è aperto in lettura e scrittura (`O_RDWR`). L'ultimo parametro definisce le autorizzazioni di directory dell'oggetto memoria condivisa. Una chiamata a `shm_open()` con esito positivo restituisce un intero, il descrittore di file per l'oggetto memoria condivisa.

Una volta che l'oggetto è creato, viene utilizzata la funzione `ftruncate()` per specificare la dimensione dell'oggetto in byte. La chiamata

```
ftruncate(shm_fd, 4096);
```

imposta la dimensione dell'oggetto a 4.096 byte.

Infine, la funzione `mmap()` crea un file mappato in memoria che contiene l'oggetto memoria condivisa e restituisce un puntatore al file mappato in memoria che viene utilizzato per accedere all'oggetto memoria condivisa.

I programmi mostrati nelle Figure 3.17 e 3.18 usano la memoria condivisa per implementare il modello produttore-consumatore. Il produttore definisce un oggetto memoria condivisa e scrive sulla memoria condivisa, il consumatore legge dalla memoria condivisa.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()

{
/* dimensione, in byte, dell'oggetto memoria condivisa */
const int SIZE 4096;
/* nome dell'oggetto memoria condivisa */
const char *name = "OS";
/* stringa scritta nella memoria condivisa */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* descrittore del file di memoria condivisa */
int shm_fd;
/* puntatore all'oggetto memoria condivisa */
void *ptr;

/* crea l'oggetto memoria condivisa */
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

/* configura la dimensione dell'oggetto memoria condivisa */
ftruncate(shm_fd, SIZE);

/* mappa in memoria l'oggetto memoria condivisa */
ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

/* scrive sull'oggetto memoria condivisa */
sprintf(ptr,"%s",message_0);
ptr += strlen(message_0);
sprintf(ptr,"%s",message_1);
ptr += strlen(message_1);

return 0;
}
```

Figura 3.17 Processo produttore che illustra l'API per la memoria condivisa POSIX.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* dimensione, in byte, dell'oggetto memoria condivisa */
    const int SIZE 4096;
    /* nome dell'oggetto memoria condivisa */
    const char *name = "OS";
    /* descrittore del file di memoria condivisa */
    int shm_fd;
    /* puntatore all'oggetto memoria condivisa */
    void *ptr;

    /* apre l'oggetto memoria condivisa */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* mappa in memoria l'oggetto memoria condivisa */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* legge dall'oggetto memoria condivisa */
    printf("%s", (char *)ptr);

    /* rimuove l'oggetto memoria condivisa */
    shm_unlink(name);

    return 0;
}
```

Figura 3.18 Processo consumatore che illustra l'API per la memoria condivisa POSIX.

Il produttore, mostrato in Figura 3.17, crea un oggetto memoria condivisa denominato OS e scrive la famigerata stringa “Hello world!” sulla memoria condivisa. Il programma mappa in memoria un oggetto memoria condivisa del formato specificato e consente di scrivere sull’oggetto. (Ovviamente, al produttore serve solo la scrittura). Il flag `MAP_SHARED` specifica che le modifiche apportate all’oggetto memoria condivisa saranno visibili a tutti i processi che condividono l’oggetto. Si noti che la scrittura avviene chiamando la funzione `sprintf()` e scrivendo la stringa formattata nella posizione puntata da `ptr`. Dopo ogni scrittura dobbiamo incrementare il puntatore secondo il numero di byte scritti.

Il processo consumatore, mostrato nella Figura 3.18, legge e visualizza il contenuto della memoria condivisa. Il consumatore invoca anche la funzione `shm_unlink()`,

che rimuove il segmento di memoria condivisa dopo l’accesso del consumatore. Tra gli esercizi di programmazione, alla fine di questo capitolo, ce ne sono alcuni che utilizzano la API POSIX per la memoria condivisa. Una trattazione più dettagliata della mappatura della memoria è presente nel Paragrafo 9.7 (Capitolo 9).

3.5.2 Un esempio: Mach

Come esempio di scambio di messaggi consideriamo ora il sistema operativo Mach. Abbiamo già presentato questo sistema operativo nel Capitolo 2, come parte di Mac OS X. Il suo kernel consente la creazione e la soppressione di più *task*, simili ai processi, ma che hanno più thread di controllo e meno risorse associate. La maggior parte delle comunicazioni – comprese tutte le comunicazioni tra task – si compie per mezzo di **messaggi**. I messaggi s’inviano e si ricevono attraverso mailbox, chiamate **porte**.

Anche le chiamate di sistema s’invocano per mezzo di messaggi. Al momento della creazione di ogni task si creano due porte speciali: la porta **Kernel** e la porta **Notify**. Il kernel usa la porta Kernel per comunicare con il task e notifica l’occorrenza di un evento alla porta Notify. Per il trasferimento dei messaggi sono necessarie solo tre chiamate di sistema: `msg_send()`, che invia un messaggio a una porta; `msg_receive()`, per ricevere un messaggio; `msg_rpc()`, per le **chiamate di procedura remota** (*remote procedure call*, **RPC**), invia un messaggio e ne attende esattamente uno di risposta per il trasmittente (in questo modo la RPC riproduce l’usuale chiamata di procedura, ma può operare tra sistemi diversi, da cui il termine *remota*). Le chiamate di procedura remote sono trattate dettagliatamente nel Paragrafo 3.6.2.

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

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

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

La parte variabile del messaggio è composta da una lista di dati tipizzati. Ogni elemento della lista ha un tipo, una dimensione e un valore. Il tipo degli oggetti specificati nel messaggio è importante, poiché oggetti definiti dal sistema operativo, co-

me diritti di proprietà o di ricezione, stati del task e segmenti di memoria, si possono inviare all'interno dei messaggi.

Anche le operazioni di trasmissione e ricezione sono piuttosto flessibili. Per esempio, quando s'invia un messaggio a una porta che non è già piena, lo si copia al suo interno e il thread trasmittente prosegue la sua esecuzione. Se la porta è piena, il thread trasmittente ha le seguenti possibilità.

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

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

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

Il sistema Mach è stato progettato per i sistemi distribuiti (descritti nel Capitolo 17), ma è adatto anche a sistemi dotati di meno core elaborativi, come prova la sua inclusione nel sistema Mac OS X. I problemi più gravi dei sistemi a scambio di messaggi sono generalmente causate dalle scarse prestazioni dovute alla doppia operazione di copiatura dei messaggi dal trasmittente alla porta e quindi dalla porta al ricevente. Il sistema di messaggi di Mach cerca di evitare doppie operazioni di copiatura impiegando tecniche di gestione della memoria virtuale (Capitolo 9). Fondamentalmente Mach mappa lo spazio d'indirizzi contenente il messaggio del trasmittente nello spazio d'indirizzi del ricevente. Il messaggio stesso non è mai effettivamente copiato, quindi le prestazioni del sistema migliorano notevolmente anche se solo nel caso di messaggi all'interno dello stesso sistema. Il sistema Mach viene discusso con maggior dettaglio nell'Appendice B, disponibile sul sito web.

3.5.3 Un esempio: Windows

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

La funzione di scambio di messaggi di Windows è detta **chiamata di procedura locale avanzata** (*advanced local procedure call*, ALPC) e si usa per la comunicazione tra due processi presenti nello stesso calcolatore. È simile al meccanismo standard della chiamata di procedura remota, ma è ottimizzata per questo sistema (le chiamate di procedura remota sono trattate in dettaglio nel Paragrafo 3.6.2). Come in Mach, nel sistema Windows s'impiega un oggetto porta per stabilire e mantenere una connessione tra due processi. Windows usa due tipi di porte: le **porte di connessione** e le **porte di comunicazione**.

I processi server pubblicano gli oggetti porte di connessione che sono visibili a tutti i processi. Quando un client vuole i servizi di un sottosistema, apre un handle all'oggetto porta di connessione del server e invia una richiesta di connessione a quella porta. Il server crea quindi un canale e restituisce un handle al client. Il canale è costituito da una coppia di porte di comunicazione private: una per i messaggi client-server, l'altra per i messaggi server-client. Inoltre, i canali di comunicazione supportano un meccanismo di callback che permette a client e server di accettare le richieste anche quando sarebbero normalmente in attesa di una risposta.

Quando viene creato un canale ALPC viene scelta una tra le seguenti tre tecniche di scambio di messaggi.

1. Per piccoli messaggi (fino a 256 byte) la coda di messaggi della porta viene utilizzata come deposito intermedio e i messaggi vengono copiati da un processo all'altro.
2. Messaggi più grandi devono essere fatti passare attraverso un **oggetto sezione**, che è una regione di memoria condivisa associata al canale.
3. Quando la quantità di dati è troppo grande per essere contenuta in un oggetto sezione, è disponibile una API che consente ai processi server di leggere e scrivere direttamente nello spazio degli indirizzi di un client.

Il client deve decidere, quando attiva il canale, se, dovrà inviare messaggi lunghi: in tal caso richiede la creazione di un oggetto sezione. Allo stesso modo, se il server stabilisce che le risposte sono lunghe, provvede alla creazione di un oggetto sezione. Per usare gli oggetti sezione s'invia un breve messaggio contenente un puntatore e le informazioni sulle sue dimensioni. Questo metodo è un po' più complicato del primo metodo sopra descritto, ma evita la copiatura dei dati. La struttura delle chiamate di procedura locale avanzate in Windows è illustrata nella Figura 3.19.

È importante notare che il meccanismo ALPC di Windows non è parte della API Windows e quindi non è accessibile ai programmatori di applicazioni. Le applicazioni

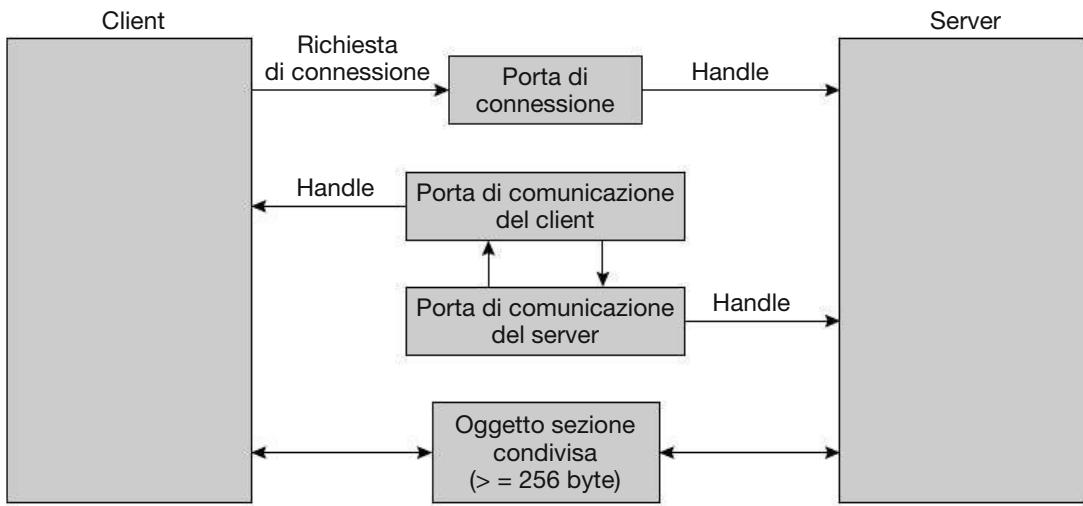


Figura 3.19 Chiamate di procedura locale avanzate in Windows.

che utilizzano l'API Windows devono comunque usare RPC standard; nel caso in cui la chiamata si riferisca a un processo residente sulla stessa macchina del chiamante, il sistema la implementa tramite una ALPC. Inoltre molti servizi del kernel utilizzano ALPC per comunicare con processi client.

3.6 Comunicazione nei sistemi client-server

Nel Paragrafo 3.4 ci siamo soffermati su come i processi possano comunicare usando memoria condivisa e scambio di messaggi. Tali tecniche sono utilizzabili anche per la comunicazione in sistemi client/server (Paragrafo 1.11.4). Consideriamo qui altre tre strategie: socket, chiamate di procedura remota (RPC) e pipe.

3.6.1 Socket

Una *socket* è definita come l'estremità di un canale di comunicazione. Una coppia di processi che comunicano attraverso una rete usa una coppia di socket, una per ogni processo, e ogni socket è identificata da un indirizzo IP concatenato a un numero di porta. In generale, le socket impiegano un'architettura client-server; il server attende le richieste dei client, stando in ascolto a una porta specificata; quando il server riceve una richiesta, accetta la connessione proveniente dalla socket del client, e si stabilisce la comunicazione. I server che svolgono servizi specifici (come telnet, FTP e HTTP) stanno in ascolto su porte ben note (i server telnet alla porta 23, i server FTP alla porta 21, e i server Web, o HTTP, alla porta 80). Tutte le porte al di sotto del valore 1024 sono considerate *ben note (well known)* e si usano per realizzare servizi standard.

Quando un processo client richiede una connessione, il calcolatore che lo ospita assegna una porta specifica, che consiste di un numero arbitrario maggiore di 1024. Si supponga per esempio che un processo client presente nel calcolatore x con indi-

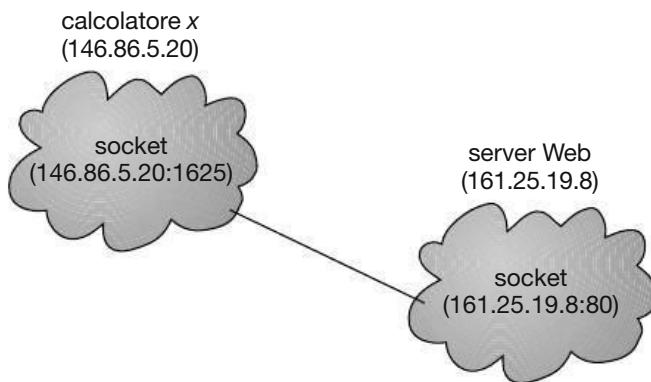


Figura 3.20 Comunicazione tramite socket.

rizzo IP 146.86.5.20 voglia stabilire una connessione con un server Web (in ascolto alla porta 80) all'indirizzo 161.25.19.8; il calcolatore x potrebbe assegnare al client, per esempio, la porta 1625. La connessione sarebbe composta di una coppia di socket: (146.86.5.20:1625) nel calcolatore x e (161.25.19.8:80) nel server Web. La Figura 3.20 mostra questa situazione. La consegna dei pacchetti al processo giusto avviene secondo il numero della porta di destinazione.

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

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

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

Nel nostro esempio un server usa socket TCP orientate alla connessione per fornire l'ora e la data correnti ai client. Il server si pone in ascolto alla porta 6013 – potrebbe essere un altro numero arbitrario, purché maggiore di 1024. Quando giunge una richiesta di connessione, il server restituisce la data e l'ora al client.

Il codice del server è mostrato nella Figura 3.21. Il server crea una `ServerSocket` che ascolta alla porta 6013. Il server si pone in ascolto tramite la chiamata bloccante `accept()` e rimane in attesa fino all'arrivo della richiesta di un client. A quel punto, `accept()` restituisce la socket che il server può usare per comunicare con il client.

```

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

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

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

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

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

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

```

Figura 3.21 Server che fornisce al client la data corrente.

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

Un client comunica con il server creando una socket e collegandosi per suo tramite alla porta su cui il server è in ascolto. L’implementazione è mostrata nella Figura 3.22. Il client crea una `Socket` e richiede una connessione al server alla porta 6013 dell’indirizzo IP 127.0.0.1. Stabilita la connessione, il client può leggere dalla socket tramite le ordinarie istruzioni di I/O. Dopo aver ricevuto la data dal server, il client chiude la socket e termina. L’indirizzo IP 127.0.0.1, noto come **loopback**, è peculiare: è

```

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

public class DateClient
{

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

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* legge la data dalla socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* chiude la socket */
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

Figura 3.22 Client che riceve dal server la data corrente.

usato da una macchina per riferirsi a se stessa. Tramite questo stratagemma, un client e un server residenti sulla stessa macchina sono in grado di comunicare tramite il protocollo TCP/IP. L'indirizzo IP 127.0.0.1 si potrebbe sostituire con l'indirizzo IP di una qualunque macchina che ospiti il server che fornisce la data. È anche possibile usare un nome simbolico, come www.westminstercollege.edu.

La comunicazione tramite socket è diffusa ed efficiente, ma è considerata una forma di comunicazione di basso livello fra sistemi distribuiti. Infatti, le socket permettono unicamente la trasmissione di un flusso non strutturato di byte: è responsabilità del client e del server interpretare e organizzare i dati in strutture più complesse. Nei due paragrafi successivi sono illustrati due metodi di comunicazione di più alto livello, le chiamate di procedure remote (RPC) e le pipe.

3.6.2 Chiamate di procedure remote

Uno tra i più diffusi tipi di servizio remoto è il paradigma della RPC, presentato brevemente nel Paragrafo 3.5.2. La RPC è stata progettata per astrarre il meccanismo della chiamata di procedura affinché si possa usare tra sistemi collegati tramite una rete. Per molti aspetti è simile al meccanismo IPC descritto nel Paragrafo 3.4, ed è generalmente costruita su un sistema di questo tipo. Tuttavia in questo caso, poiché in un sistema distribuito si eseguono i processi su sistemi distinti, per offrire un servizio remoto occorre impiegare uno schema di comunicazione basato sullo scambio di messaggi.

Contrariamente ai messaggi IPC, i messaggi scambiati per la comunicazione RPC sono ben strutturati e non semplici pacchetti di dati. Si indirizzano a un demone RPC, in ascolto a una porta del sistema remoto, e contengono un identificatore della funzione da eseguire e i parametri da passare a tale funzione. Nel sistema remoto si esegue questa funzione e s'invia ogni risultato al richiedente in un messaggio distinto.

La **porta** è semplicemente un numero inserito all'inizio del messaggio. Mentre un singolo sistema ha normalmente un solo indirizzo di rete, all'interno dell'indirizzo può avere molte porte che servono a distinguere i numerosi servizi che può fornire in rete. Se un processo remoto richiede un servizio, indirizza i propri messaggi alla porta corrispondente; per esempio, per permettere ad altri di ottenere un elenco dei suoi attuali utenti, un sistema deve possedere un demone in ascolto a una porta, per esempio la porta 3027, che realizzi una siffatta RPC. Qualsiasi sistema remoto può ottenere l'informazione richiesta, vale a dire l'elenco degli utenti, inviando un messaggio RPC alla porta 3027 del server; i dati si ricevono in un messaggio di risposta.

La semantica delle RPC permette a un client di richiamare una procedura presente in un sistema remoto nello stesso modo in cui invocherebbe una procedura locale. Il sistema delle RPC nasconde i dettagli necessari che consentono la comunicazione, fornendo uno **stub** al lato client. Esiste in genere uno stub per ogni diversa procedura remota. Quando il client la invoca, il sistema delle RPC richiama l'appropriato stub, passando i parametri della procedura remota. Lo stub individua la porta del server e struttura i parametri; la strutturazione dei parametri (*marshalling*) implica l'assemblaggio dei parametri in una forma che si può trasmettere tramite una rete. Lo stub quindi trasmette un messaggio al server usando lo scambio di messaggi. Un analogo stub nel server riceve questo messaggio e invoca la procedura nel server; se è necessario, riporta i risultati al client usando la stessa tecnica. Sui sistemi Windows, il codice dello stub è compilato a partire da una specifica, scritta nel **Microsoft Interface Definition Language** (IDL), utilizzato per definire le interfacce tra i programmi client e server.

Una questione da affrontare riguarda le differenze nella rappresentazione dei dati nel client e nel server. Si consideri la rappresentazione a 32 bit dei numeri interi; alcuni sistemi, noti come *big-endian*, usano l'indirizzo di memoria minore per contenere il byte più significativo; altri, noti come *little-endian*, lo usano per contenere il byte meno significativo. Nessuna delle due opzioni è migliore di per sé, si tratta di una scelta arbitraria nella definizione dell'architettura del computer. Per risolvere tali

differenze molti sistemi di RPC definiscono una rappresentazione dei dati indipendente dalla macchina. Uno di questi sistemi di rappresentazione è noto come **rappresentazione esterna dei dati** (*external data representation*, XDR). Nel client la strutturazione dei parametri riguarda la conversione dei dati, prima di inviarli al server, dal formato della specifica macchina nel formato XDR; nel server, i dati nel formato XDR si convertono nel formato della macchina server.

Un'altra questione importante riguarda la semantica delle chiamate. Infatti, mentre le chiamate locali falliscono in circostanze estreme, le RPC possono non riuscire, o risultare duplicate e dunque eseguite più volte, semplicemente a causa di comuni errori della rete. Un modo per affrontare il problema è che il sistema operativo garantisca che i messaggi vengono elaborati *esattamente una volta*, e non *al massimo una volta*. Questo tipo di semantica è comune per le chiamate locali, ma è più difficile da implementare per le RPC.

Si consideri prima la semantica “al massimo una volta”. Essa può essere implementata marcando ogni messaggio con la sua ora di emissione (*timestamp*). Il server dovrà mantenere l'archivio di tutti gli orari di emissione dei messaggi già ricevuti e trattati, o perlomeno un archivio che sia abbastanza ampio da identificare tutti i messaggi duplicati. I messaggi in entrata con orario di emissione già presente nell'archivio sono ignorati. I client avranno allora la sicurezza che la procedura remota sarà eseguita al massimo una volta, anche nel caso di un invio di più copie di uno stesso messaggio.

Per implementare la semantica “esattamente una volta” (*exactly once*), occorre eliminare il rischio che il server non riceva mai la richiesta. Per ottenere questo risultato, il server deve implementare la semantica “al massimo una volta” (*at most once*), ma integrarla con l'invio al client di un riscontro che attesti l'avvenuta esecuzione della procedura. Riscontri di questo tipo sono molto diffusi nella comunicazione tramite reti. Il client dovrà inviare periodicamente la richiesta RPC finché non ottenga il relativo riscontro.

Un altro argomento importante riguarda la comunicazione tra server e client. Con le ordinarie chiamate di procedure, durante la fase di collegamento, caricamento o esecuzione di un programma (Capitolo 8), ha luogo una forma di associazione che sostituisce il nome della procedura chiamata con l'indirizzo di memoria della procedura stessa. Lo schema delle RPC richiede una corrispondenza di questo genere tra il client e la porta del server, ma come fa il client a conoscere i numeri di porta sul server? Nessun sistema dispone d'informazioni complete sull'altro, poiché essi non condividono memoria.

Per risolvere questo problema s'impiegano per lo più due metodi. Con il primo, l'informazione sulla corrispondenza tra il client e la porta del server si può predeterminare fissando gli indirizzi delle porte: una RPC si associa nella fase di compilazione a un numero di porta fisso; il server non può modificare né il numero di porta né il servizio richiesto. Con il secondo metodo la corrispondenza si può effettuare dinamicamente tramite un meccanismo di *rendezvous*. Generalmente il sistema operativo fornisce un demone di rendezvous (*matchmaker*) a una porta di RPC fissata. Un client

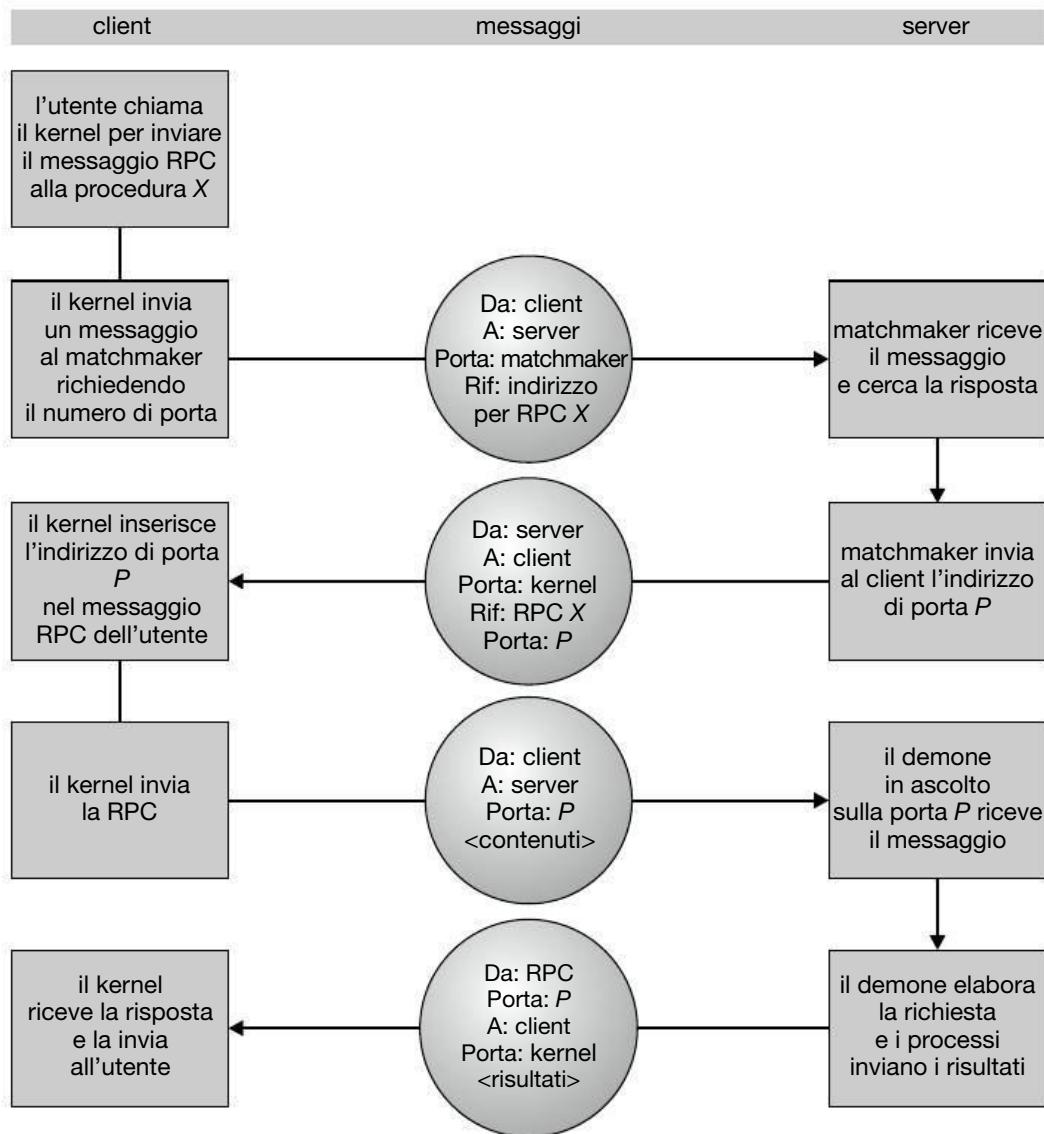


Figura 3.23 Esecuzione di una chiamata di procedura remota (RPC).

invia un messaggio, contenente il nome della RPC, al demone di rendezvous per richiedere l'indirizzo della porta della RPC che deve eseguire. Il demone risponde col numero di porta, e la richiesta d'esecuzione della RPC si può inviare a quella porta fino al termine del processo (o fino alla caduta del server). Questo metodo richiede un ulteriore carico a causa della richiesta iniziale, ma è più flessibile del primo metodo. La Figura 3.23 illustra un esempio d'interazione.

Lo schema della RPC è utile nella realizzazione di un file system distribuito (Capitolo 17); un sistema di questo tipo si può realizzare come un insieme di demoni e client di RPC. I messaggi s'indirizzano alla porta del file system distribuito su un server in cui deve avvenire l'operazione sui file. I messaggi contengono le operazioni da svolgere sui dischi: `read`, `write`, `rename`, `delete` o `status`, corrispondenti

alle normali chiamate di sistema che si usano per i file. Il messaggio di risposta contiene i dati risultanti da quella chiamata, che il demone del file system distribuito esegue su incarico del client. Un messaggio può, per esempio, contenere una richiesta di trasferimento di un intero file a un client, oppure semplici richieste di blocchi. Nel secondo caso per trasferire un intero file possono essere necessarie parecchie richieste di questo tipo.

3.6.3 Pipe

Una **pipe** agisce come canale di comunicazione tra processi. Le pipe sono state uno dei primi meccanismi di comunicazione tra processi (IPC) nei primi sistemi UNIX e generalmente forniscono ai processi uno dei metodi più semplici per comunicare l'uno con l'altro, sebbene con qualche limitazione. Quando si implementa una pipe devono essere prese in considerazione quattro questioni.

1. La comunicazione permessa dalla pipe è unidirezionale o bidirezionale?
2. Se è ammessa la comunicazione bidirezionale, essa è di tipo *half duplex* (i dati possono viaggiare in un'unica direzione alla volta) o *full duplex* (i dati possono viaggiare contemporaneamente in entrambe le direzioni)?
3. Deve esistere una relazione (del tipo *padre-figlio*) tra i processi in comunicazione?
4. Le pipe possono comunicare in rete o i processi comunicanti devono risiedere sulla stessa macchina?

Nei paragrafi seguenti esploriamo due tipi comuni di pipe utilizzate sia in UNIX sia in Windows: le pipe convenzionali e le named pipe.

3.6.3.1 Pipe convenzionali

Le pipe convenzionali permettono a due processi di comunicare secondo una modalità standard chiamata del produttore-consumatore. Il produttore scrive a una estremità del canale (**l'estremità dedicata alla scrittura**, o *write-end*) mentre il consumatore legge dall'altra estremità (**l'estremità dedicata alla lettura**, o *read-end*). Le pipe convenzionali sono quindi unidirezionali, perché permettono la comunicazione in un'unica direzione. Se viene richiesta la comunicazione bidirezionale devono essere utilizzate due *pipe*, ognuna delle quali manda i dati in una direzione. Illustreremo la costruzione di pipe convenzionali sia in UNIX sia in Windows. In entrambi i programmi di esempio un processo scrive sulla pipe il messaggio **Greetings**, mentre l'altro lo legge dall'altra estremità della pipe.

Nei sistemi UNIX le pipe convenzionali sono costruite utilizzando la funzione

```
pipe(int fd[ ])
```

Essa crea una pipe alla quale si può accedere tramite i descrittori del file `int fd[] : fd[0]` è l'estremità dedicata alla lettura, mentre `fd[1]` è l'estremità dedicata alla scrittura. Il sistema UNIX considera una pipe come un tipo speciale di file; si può così accedere alle pipe tramite le usuali chiamate di sistema `read()` e `write()`.

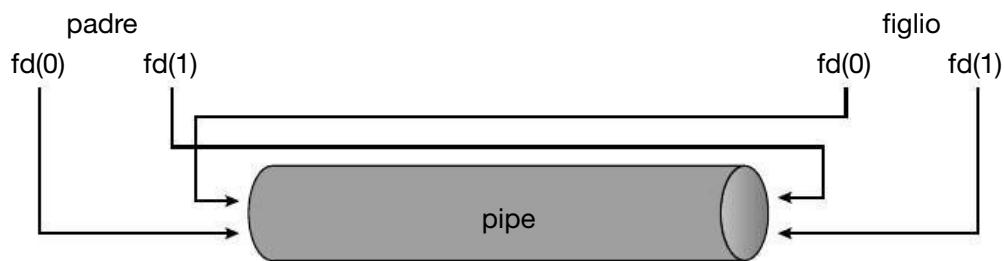


Figura 3.24 Descrittori di file per una pipe convenzionale.

Non si può accedere a una pipe al di fuori del processo che la crea. Solitamente un processo padre crea una pipe e la utilizza per comunicare con un processo figlio generato con il comando `fork()`. Come già indicato nel Paragrafo 3.3.1, il processo figlio eredita i file aperti dal processo padre. Dal momento che la pipe è un tipo speciale di file, il figlio eredita la pipe dal proprio processo padre. La Figura 3.24 illustra la relazione del descrittore di file `fd` rispetto ai processi padre e figlio.

Nel programma UNIX mostrato nella Figura 3.25 il processo padre crea una pipe e in seguito esegue una chiamata `fork()`, generando un processo figlio. Ciò che succede dopo la chiamata `fork()` dipende da come i dati devono fluire nel canale. In questo caso, il padre scrive sulla pipe e il figlio legge da essa. È importante sottolineare come sia il processo padre sia il processo figlio chiudano inizialmente le estremità inutilizzate del canale. Sebbene il programma mostrato nelle Figure 3.25 e 3.26 non richieda questa azione è importante assicurare che un processo che legge dalla pipe possa rilevare l'end-of-file (`read()` restituisce 0) quando chi scrive ha chiuso la sua estremità della pipe.

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;
```

Figura 3.25 Pipe convenzionali in UNIX. Il programma continua nella Figura 3.26.

```

/* crea la pipe */
if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
}

/* crea tramite fork un processo figlio */
pid = fork();

if (pid < 0) { /* errore */
    fprintf(stderr, "Fork Failed");
    return 1;
}

if (pid > 0) { /* processo padre */
    /* chiude l'estremità inutilizzata della pipe */
    close(fd[READ_END]);

    /* scrive sulla pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* chiude l'estremità della pipe dedicata alla scrittura */
    close(fd[WRITE_END]);
}

else { /* processo figlio */
    /* chiude l'estremità inutilizzata della pipe */
    close(fd[WRITE_END]);

    /* legge dalla pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s",read_msg);

    /* chiude l'estremità della pipe dedicata alla lettura */
    close(fd[READ_END]);
}

return 0;
}

```

Figura 3.26 Continuazione della Figura 3.25.

Nei sistemi Windows le pipe convenzionali sono denominate **pipe anonime** e si comportano analogamente alle loro equivalenti in UNIX: sono unidirezionali e utilizzano relazioni del tipo padre-figlio tra i processi coinvolti nella comunicazione. Inoltre, l'attività di lettura e scrittura sulla pipe può essere eseguita con le usuali funzioni

```

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;

```

Figura 3.27 Pipe anonime in Windows (processo genitore). Il programma continua nella Figura 3.28.

ReadFile() e WriteFile(). L’API Windows per creare le pipe è la funzione CreatePipe(), che riceve in ingresso quattro parametri. I parametri forniscono handle distinti per (1) leggere e (2) scrivere sulla pipe, oltre a (3) una istanza della struttura STARTUPINFO, usata per specificare che il processo figlio erediti gli handle della pipe. Inoltre, può essere specificata (4) la dimensione della pipe (in byte).

Le Figure 3.27 e 3.28 illustrano un processo padre che crea una pipe anonima per comunicare con il proprio figlio. A differenza dei sistemi UNIX, dove un processo figlio eredita automaticamente una pipe creata dal proprio padre, Windows richiede al programmatore di specificare quali attributi saranno ereditati dal processo figlio. Ciò avviene innanzitutto inizializzando la struttura SECURITY_ATTRIBUTES per permettere che gli handle siano ereditati e poi reindirizzando lo standard input o lo standard output del processo figlio verso l’handle di scrittura o di lettura della pipe, rispettivamente. Dato che il figlio leggerà dalla pipe, il padre deve reindirizzare lo standard input del figlio verso l’handle di lettura della pipe stessa. Inoltre, dal momento che le pipe sono half duplex, è necessario proibire al figlio di ereditare l’handle di scrittura della pipe. La creazione del processo figlio avviene come nel programma nella Figura 3.11, fatta eccezione per il quinto parametro che in questo caso viene impostato al valore TRUE per indicare che il processo figlio eredita degli handle specifici dal proprio padre. Prima di scrivere sulla pipe, il padre ne chiude l’estremità di lettura, che è inutilizzata. Il processo figlio che legge dalla pipe è mostrato nella Figura 3.29. Prima di leggere dalla pipe, questo programma ottiene l’handle di lettura invocando GetStdHandle().

Si noti bene che le pipe convenzionali richiedono una relazione di parentela padre-figlio tra i processi comunicanti, sia in UNIX sia in Windows. Ciò significa che queste pipe possono essere utilizzate soltanto per la comunicazione tra processi in esecuzione sulla stessa macchina.

```

/*imposta gli attributi di sicurezza in modo che le pipe siano
 ereditate */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* alloca la memoria */
ZeroMemory(&pi, sizeof(pi));

/* crea la pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
    fprintf(stderr, "Create Pipe Failed");
    return 1;
}

/* prepara la struttura START_INFO per il processo figlio */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* reindirizza lo standard input verso l'estremità della pipe
 dedicata alla lettura */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* non permette al processo figlio di ereditare l'estremità
 della pipe dedicata alla scrittura */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* crea il processo figlio */
CreateProcess(NULL, "child.exe", NULL, NULL,
    TRUE, /* inherit handles */
    0, NULL, NULL, &si, &pi);

/* chiude l'estremità inutilizzata della pipe */
CloseHandle(ReadHandle);

/* il padre scrive sulla pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
    fprintf(stderr, "Error writing to pipe.");

/* chiude l'estremità della pipe dedicata alla scrittura */
CloseHandle(WriteHandle);

/* attende la terminazione del processo figlio */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}

```

Figura 3.28 Continuazione della Figura 3.27.

```

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

/* riceve l'handle di lettura della pipe */
ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

/* il figlio legge dalla pipe */
if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
    printf("child read %s",buffer);
else
    fprintf(stderr, "Error reading from pipe");
return 0;
}

```

Figura 3.29 Pipe anonime in Windows (processo figlio).

3.6.3.2 Named pipe

Le pipe convenzionali offrono un meccanismo semplice di comunicazione tra una coppia di processi. Tuttavia, le pipe convenzionali esistono solo mentre i processi stanno comunicando fra loro. Sia in UNIX sia in Windows, una volta che i processi hanno finito di comunicare e terminano, le pipe convenzionali cessano di esistere.

Le **named pipe** costituiscono uno strumento di comunicazione molto più potente; la comunicazione può essere bidirezionale, e la relazione di parentela padre-figlio non è necessaria. Una volta che si sia creata la named pipe, diversi processi possono utilizzarla per comunicare. In uno scenario tipico una named pipe ha infatti diversi scrittori. In più, le named pipe continuano a esistere anche dopo che i processi comunicanti sono terminati. Sia UNIX sia Windows mettono a disposizione le named pipe, nonostante ci siano grandi differenze nei dettagli dell’implementazione. Di seguito, prendiamo in esame le named pipe in ciascuno dei due sistemi.

Nei sistemi UNIX le named pipe sono dette FIFO. Una volta create, esse appaiono come normali file all’interno del file system. Una FIFO viene creata mediante una chiamata di sistema `mkfifo()` e viene poi manipolata con le usuali chiamate di sistema `open()`, `read()`, `write()` e `close()`; essa continuerà a esistere finché non sarà esplicitamente eliminata dal file system. Nonostante le FIFO permettano la comunicazione bidirezionale, l’unica tipologia di trasmissione consentita è quella half duplex. Nel caso in cui i dati debbano viaggiare in entrambe le direzioni, vengono solitamente utilizzate due FIFO. Per utilizzare le FIFO i processi comunicanti devono



LE PIPE IN PRATICA

Le pipe sono usate molto spesso dalla riga di comando di UNIX in situazioni nelle quali l'output di un comando serve da input per un altro comando. Per esempio, il comando `ls` di UNIX elenca il contenuto di una directory. Per directory particolarmente grandi, l'output può scorrere su diverse schermate. Il comando `more` gestisce l'output mostrando solo una schermata alla volta; l'utente deve premere la barra spaziatrice per muoversi da una schermata all'altra. Creando una pipe tra i comandi `ls` e `more` (in esecuzione come singoli processi) si fa in modo che l'output di `ls` venga inviato all'input di `more`, permettendo così all'utente di vedere il contenuto di una grande directory una schermata alla volta. Dalla riga di comando si può costruire una pipe utilizzando il carattere `|`. Il comando completo è quindi

```
ls | more
```

In questo scenario, il comando `ls` funge da produttore, e il suo output è consumato dal comando `more`.

I sistemi Windows offrono un comando `more` per la shell DOS con una funzionalità analoga al corrispettivo di UNIX. Anche la shell DOS utilizza il carattere `|` per creare una pipe. L'unica differenza è costituita dal fatto che, per restituire il contenuto di una directory, DOS utilizza il comando `dir` anziché `ls`. Il comando equivalente in DOS è quindi

```
dir | more
```

risiedere sulla stessa macchina: se è richiesta la comunicazione tra più macchine devono essere impiegate le socket (si veda il Paragrafo 3.6.1).

Rispetto alle loro controparti in UNIX, le named pipe su un sistema Windows offrono un meccanismo di comunicazione più ricco. È permessa la comunicazione full duplex e i processi comunicanti possono risiedere sia sulla stessa macchina sia su macchine diverse. Inoltre, attraverso una FIFO di UNIX possono essere trasmessi solo dati byte-oriented, mentre i sistemi Windows permettono la trasmissione di dati sia byte-oriented sia message-oriented. Le named pipe vengono create con la funzione `CreateNamedPipe()` e un client può connettersi a una named pipe tramite `ConnectNamedPipe()`. La comunicazione attraverso le named pipe avviene grazie alle funzioni `ReadFile()` e `WriteFile()`.

3.7 Sommario

Un processo è un programma in esecuzione. Nel corso delle sue attività, un processo cambia stato, e tale stato è definito dall'attività corrente del processo stesso. Ogni processo può trovarsi in uno tra i seguenti stati: nuovo, pronto, esecuzione, attesa o terminato. In un sistema operativo ogni processo è rappresentato dal proprio blocco di controllo del processo (PCB).

Un processo, quando non è in esecuzione, è inserito in una coda d'attesa. Le due classi principali di code in un sistema operativo sono le code di richieste di I/O e la coda dei processi pronti per l'esecuzione (o *ready queue*), quest'ultima contenente

tutti i processi pronti per l'esecuzione che si trovino in attesa della CPU. Ogni processo è rappresentato da un PCB.

Il sistema operativo deve selezionare i processi da diverse code di scheduling. Lo scheduling a lungo termine (o *job scheduling*) consiste nella scelta dei processi che si contenderanno la CPU. Normalmente lo scheduling a lungo termine è influenzato in modo consistente da considerazioni riguardanti l'assegnazione delle risorse, in particolar modo quelle concernenti la gestione della memoria. Lo scheduling a breve termine (o scheduling della CPU) consiste nella selezione di un processo dalla ready queue.

I sistemi operativi devono implementare un meccanismo per generare processi figli da un processo genitore. Genitore e figli possono girare in concomitanza, oppure il genitore potrà essere posto in attesa della terminazione dei figli. L'esecuzione corrente è motivata dalla necessità di condividere informazioni e dal potenziale aumento della velocità di calcolo, oltre che da considerazioni legate alla modularità e alla convenienza.

I processi in esecuzione nel sistema operativo possono essere indipendenti o cooperanti. I processi cooperanti devono avere i mezzi per comunicare tra loro. Fondamentalmente esistono due schemi di comunicazione: memoria condivisa e scambio di messaggi. Nel metodo con memoria condivisa i processi in comunicazione devono condividere alcune variabili, per mezzo di cui i processi scambiano informazioni; il compito della comunicazione è lasciato ai programmatore di applicazioni; il sistema operativo deve semplicemente offrire la memoria condivisa. Il metodo con scambio di messaggi permette di compiere uno scambio di messaggi tra i processi; in questo caso il compito di attuare la comunicazione è del sistema operativo. Questi due schemi non sono mutuamente esclusivi, e si possono impiegare insieme in uno stesso sistema.

La comunicazione nei sistemi client/server può impiegare (1) socket, (2) chiamate di procedure remote (RPC) o (3) pipe. Una socket è definita come l'estremo di una comunicazione. Una connessione tra una coppia di applicazioni consiste di una coppia di socket, ciascuna a un'estremità del canale di comunicazione. Le RPC sono un'altra forma di comunicazione distribuita: una RPC si verifica quando un processo (o un thread) invoca una procedura in un'applicazione remota. Le pipe offrono un meccanismo relativamente semplice per la comunicazione tra processi. Le pipe convenzionali permettono la comunicazione tra processi padre e figlio, mentre le named pipe permettono la comunicazione tra processi non in relazione tra loro.

Esercizi di ripasso

- 3.1** Con riferimento al programma mostrato nella Figura 3.30, descrivete l'output prodotto alla LINEA A.
- 3.2** Considerando anche il processo padre iniziale, quanti processi vengono creati dal programma della Figura 3.31?
- 3.3** Le versioni originali del sistema operativo Apple iOS non fornivano alcuno strumento di elaborazione concorrente. Discutete tre principali complicazioni che l'elaborazione concorrente aggiunge a un sistema operativo.

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

int value = 5;

int main()
{
pid_t pid;

pid = fork();

if (pid == 0) { /* processo figlio */
    value += 15;
    return 0;
}
else if (pid > 0) { /* processo padre */
    wait(NULL);
    printf("PARENT: value = %d", value); /* LINEA A */
    return 0;
}
}
```

Figura 3.30 Quale output sarà prodotto alla linea A?

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

int main()
{
    /* crea mediante fork un processo figlio */
    fork();

    /* crea un altro processo figlio */
    fork();

    /* e ne crea un altro ancora */
    fork();

    return 0;
}
```

Figura 3.31 Quanti processi vengono creati?

- 3.4** Il processore Sun UltraSPARC ha diversi set di registri. Descrivete ciò che avviene quando si ha un cambio di contesto nel caso in cui il contesto successivo sia già caricato in un determinato set di registri. Che cosa succede se il contesto successivo è nella memoria (invece che in un set di registri) e tutti i registri sono in uso?
- 3.5** Quando un processo crea un nuovo processo utilizzando l’istruzione `fork()`, quale dei seguenti stati è condiviso tra il processo padre e il processo figlio?
- Stack
 - Heap
 - Segmenti di memoria condivisa
- 3.6** A proposito del meccanismo RPC, considerate la semantica “esattamente una volta” (*exactly once*). L’algoritmo che implementa questa semantica funziona correttamente anche se il messaggio ACK che restituisce al client va perso a causa di un problema di rete? Descrivete la sequenza di messaggi scambiati e indicate se la semantica “esattamente una volta” viene ancora preservata.
- 3.7** Assumendo che un sistema distribuito sia soggetto a malfunzionamento del server, quali meccanismi sarebbero richiesti per garantire la semantica “esattamente una volta” per l’esecuzione di RPC?

Esercizi

- 3.8** Descrivete le differenze tra scheduling a breve termine, a medio termine e a lungo termine.
- 3.9** Descrivete le azioni intraprese dal kernel nell’esecuzione di un cambio di contesto tra processi.
- 3.10** Costruite un albero di processi simile a quello nella Figura 3.8. Per ottenere informazioni sui processi su sistemi UNIX o Linux utilizzate il comando `ps aux`. Utilizzate il comando `man ps` per ottenere più informazioni sul comando `ps`. Il task manager di Windows non mostra l’ID del processo padre, ma lo strumento *process monitor*, disponibile su technet.microsoft.com, è in grado di mostrare l’albero dei processi.
- 3.11** Spiegate il ruolo del processo `init` su sistemi UNIX e Linux, con riferimento alla terminazione dei processi.
- 3.12** Compreso il processo padre iniziale, quanti processi vengono creati dal programma mostrato nella Figura 3.32?
- 3.13** Dite in quali circostanze la linea di codice contrassegnata `printf("LINE J")` nella Figura 3.33 sarà raggiunta.
- 3.14** In riferimento al programma nella Figura 3.34, identificate i valori dei `pid` alle linee A, B, C e D. (Assumete che i `pid` effettivi del padre e del figlio siano rispettivamente 2600 e 2603).

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

int main()
{
    int i;

    for (i = 0; i < 4; i++)
        fork();

    return 0;
}
```

Figura 3.32 Quanti processi vengono creati?

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

int main()
{
pid_t pid;

    /* crea un processo figlio */
    pid = fork();

    if (pid < 0) { /* si è verificato un errore */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* processo figlio */
        execlp("/bin/ls","ls",NULL);
        printf("LINE J");
    }
    else { /* processo padre */
        /* il padre attende che il figlio termini */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Figura 3.33 Quando sarà raggiunta la linea `printf("LINE J")`?

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

int main()
{
pid_t pid, pid1;

/* crea mediante fork un processo figlio */
pid = fork();

if (pid < 0) { /* errore */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* processo figlio */
    pid1 = getpid();
    printf("child: pid = %d",pid); /* A */
    printf("child: pid1 = %d",pid1); /* B */
}
else { /* processo padre */
    pid1 = getpid();
    printf("parent: pid = %d",pid); /* C */
    printf("parent: pid1 = %d",pid1); /* D */
    wait(NULL);
}

return 0;
}
```

Figura 3.34 Quali sono i valori dei pid?

3.15 Fornite un esempio di situazione nella quale le pipe convenzionali siano più adatte delle named pipe e un esempio di situazione nella quale le named pipe siano invece più indicate delle pipe convenzionali.

3.16 In riferimento al meccanismo RPC, descrivete i possibili effetti negativi della mancata implementazione delle semantiche “al massimo una volta” o di quella “esattamente una volta”. Considerate una possibile applicazione di un meccanismo che non le implementi.

3.17 Descrivete l’output del programma nella Figura 3.35 alle LINEE X e Y.

3.18 Analizzate vantaggi e svantaggi delle tecniche elencate di seguito, considerando sia il punto di vista del sistema sia quello del programmatore.

- a. Comunicazione sincrona e asincrona.
- b. Gestione automatica o esplicita del buffer.
- c. Trasmissione per copia e trasmissione per riferimento.
- d. Messaggi a lunghezza fissa e variabile.

```

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

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
    int i;
    pid_t pid;

    pid = fork();

    if (pid == 0) {
        for (i = 0; i < SIZE; i++) {
            nums[i] *= -i;
            printf("CHILD: %d ",nums[i]); /* LINEA X */
        }
    }
    else if (pid > 0) {
        wait(NULL);
        for (i = 0; i < SIZE; i++)
            printf("PARENT: %d ",nums[i]); /* LINEA Y */
    }

    return 0;
}

```

Figura 3.35 Quale output sarà prodotto alla LINEA X e alla LINEA Y?

Problemi di programmazione

- 3.19** Scrivete, in un sistema UNIX o Linux, un programma C che crei un processo figlio che alla fine diventi un processo zombie. Questo processo zombie deve rimanere nel sistema per almeno 10 secondi. Gli stati dei processi possono essere ottenuti con il comando

```
ps -l
```

Gli stati sono riportati nella colonna S, i processi con stato Z sono processi zombie. L'identificatore di processo (PID) del processo figlio viene mostrato nella colonna PID, quello del genitore nella colonna PPID.

Il modo più semplice per verificare che il processo figlio sia davvero uno zombie è probabilmente quello di eseguire il programma che avete scritto in background (utilizzando l'opzione &) e quindi utilizzare il comando `ps -1` per determinare se il figlio è un processo zombie. Poiché non vogliamo troppi processi zombie presenti nel sistema, è necessario rimuovere quello che si è creato. Il modo più semplice per farlo è quello di terminare il genitore con il comando `kill`. Per esempio, se l'ID di processo del genitore è 4884, dobbiamo scrivere

```
kill -9 4884
```

3.20 Il **pid manager** di un sistema operativo è responsabile della gestione degli identificatori di processo. Quando un processo viene creato, il gestore dei PID gli assegna un PID univoco, che viene restituito al gestore quando il processo termina la sua esecuzione, in modo che il gestore possa in seguito riassegnare il PID ad altri processi. Gli identificatori di processo vengono ampiamente descritti nel Paragrafo 3.3.1. L'aspetto più importante da comprendere è che gli identificatori di processo devono essere univoci; non possono esistere due processi attivi con lo stesso PID.

Utilizzate le seguenti costanti per individuare l'intervallo di valori possibili dei PID:

```
# define MIN_PID 300
# define MAX_PID 5000
```

È possibile utilizzare qualsiasi struttura dati per rappresentare la disponibilità di identificatori di processo. Una strategia è quella di seguire l'esempio di Linux e utilizzare una bitmap in cui il valore 0 alla posizione i indica che il PID di valore i è disponibile e un valore pari a 1 indica che il PID di valore i è attualmente in uso.

Implementate le seguenti API per ottenere e rilasciare un PID:

- `int allocate_map(void)` – Crea e inizializza una struttura dati per rappresentare i pid; restituisce -1 in caso di insuccesso e 1 in caso di successo.
- `int allocate_pid(void)` – Alloca e restituisce un pid, restituisce -1 se non è possibile assegnare un PID (tutti i PID sono in uso).
- `void release_pid(int pid)` – Rilascia un pid

Questo esercizio di programmazione sarà modificato in seguito, nei Capitoli 4 e 5.

3.21 La congettura di Collatz ha a che fare con quello che succede quando si prende un qualsiasi numero intero n positivo e si applica il seguente algoritmo:

$$n = \begin{cases} n / 2, & \text{se } n \text{ è pari} \\ 3 \times n + 1, & \text{se } n \text{ è dispari} \end{cases}$$

La congettura afferma che quando questo algoritmo viene applicato iterativamente, per ogni intero positivo in ingresso alla fine si raggiungerà il valore 1. Per esempio, se $n = 35$, la sequenza prodotta è

35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Scrivete un programma C che utilizzi la chiamata di sistema `fork()` per generare questa sequenza nel processo figlio. Il numero di partenza sarà fornito dalla riga di comando. Per esempio, se il valore 8 viene passato come parametro da riga di comando, il processo figlio restituirà 8, 4, 2, 1. Poiché i processi padre e figlio hanno ognuno le proprie copie dei dati, la sequenza verrà necessariamente restituita dal figlio. Fate in modo che il genitore invochi la chiamata `wait()` per attendere che il processo figlio termini prima di uscire dal programma. Eseguite il necessario controllo degli errori per garantire che dalla riga di comando venga passato un numero intero positivo.

3.22 Nel Problema 3.21 il processo figlio deve restituire la sequenza di numeri generati dall'algoritmo specificato dalla congettura di Collatz, perché il genitore e il figlio hanno ognuno le proprie copie dei dati. Un altro approccio per la realizzazione di questo programma è quello di creare un oggetto di memoria condivisa tra il padre e il figlio. Questa tecnica permette al figlio di scrivere il contenuto della sequenza nell'oggetto memoria condivisa. La sequenza può quindi essere restituita dal genitore nel momento in cui il figlio completa la sua attività. Poiché la memoria è condivisa, tutte le modifiche che il figlio applica si rifletteranno nel processo padre.

Questo programma sarà strutturato utilizzando memoria condivisa POSIX come descritto nel Paragrafo 3.5.1. Il processo padre procederà attraverso le seguenti fasi:

- Costruire l'oggetto memoria condivisa (`shm_open()`, `ftruncate()`, `mmap()`).
- Creare il processo figlio e attendere la sua terminazione.
- Restituire il contenuto della memoria condivisa.
- Rimuovere l'oggetto memoria condivisa.

Uno dei problemi che possono verificarsi con l'utilizzo di processi cooperanti riguarda la sincronizzazione. In questo esercizio i processi padre e figlio devono essere coordinati in modo che il genitore non restituisca la sequenza finché il figlio non termini l'esecuzione. I due processi vengono sincronizzati utilizzando la chiamata di sistema `wait()`: il processo padre invocherà una `wait()` e verrà così sospeso fino al termine dell'esecuzione del processo figlio.

3.23 Il Paragrafo 3.6.1 descrive i numeri di porta inferiori a 1024 come numeri ben noti, perché forniscono servizi standard. La porta 17 offre il servizio *quote-of-the-day* (citazione del giorno). Quando un client si connette alla porta 17 di un server, il server risponde restituendo la citazione per quel giorno.

Modificate il server di data mostrato nella Figura 3.21 in modo che restituisca una citazione del giorno al posto della data corrente. Le citazioni devono essere in caratteri ASCII stampabili e devono contenere meno di 512 caratteri, ma sono ammesse più righe. Poiché la porta 17 è ben nota e quindi non disponibile, mettete il server in ascolto sulla porta 6017. Il client mostrato nella Figura 3.22 può essere utilizzato per leggere le citazioni restituite dal server.

3.24 Un haiku è una poesia di tre righe in cui la prima riga contiene cinque sillabe, la seconda riga contiene sette sillabe e la terza riga contiene cinque sillabe. Scrivete un server di haiku che resta in ascolto sulla porta 5575: quando un client si connette a questa porta il server risponde con un haiku. Il client mostrato nella Figura 3.22 può essere utilizzato per leggere le stringhe restituite dal server di haiku.

3.25 Un server eco è un server che restituisce ai client esattamente ciò che essi gli inviano. Se un client, per esempio, invia al server la stringa *Ciao !*, il server risponderà con gli stessi dati: ossia, invierà al client la stringa *Ciao !*. Scrivete un server eco usando la API Java descritta nel Paragrafo 3.6.1. Il server rimarrà in attesa dei client tramite il metodo `accept()`. Dopo aver accettato una connessione, il client eseguirà il ciclo seguente:

- leggerà i dati dalla socket connessa con il client, ponendoli in un buffer;
- scriverà i contenuti del buffer sulla socket connessa con il client.

Il server uscirà dal ciclo una volta stabilito che il client ha chiuso la connessione. Il server nella Figura 3.21 impiega la classe Java `java.io.BufferedReader`, che estende la classe `java.io.Reader`, usata per leggere flussi di caratteri. Il server eco, però, potrebbe ricevere dati di altro tipo dal client – per esempio dati binari. La classe `java.io.InputStream` tratta i dati come byte, e non come caratteri. È quindi necessario che il server eco impieghi una classe che estenda `java.io.InputStream`.

Il metodo `read()` di `java.io.InputStream` restituisce `-1` quando il client ha chiuso la connessione.

3.26 Utilizzando pipe convenzionali, scrivete un programma nel quale un processo manda una stringa a un secondo processo, il quale cambia le lettere maiuscole del messaggio in minuscole, e viceversa, per poi restituire il risultato al primo processo. Per esempio, se il primo processo manda il messaggio `Hi There`, il secondo restituirà `hI tHERE`. Ciò richiede l'utilizzo di due pipe, una per mandare il messaggio originale dal primo al secondo processo e l'altra per mandare il messaggio modificato dal secondo processo al primo. Potete scrivere il programma utilizzando pipe in Windows o in UNIX.

3.27 Scrivete un programma `filecopy` per la copia di file, usando le pipe convenzionali. Questo programma riceverà in ingresso due parametri: il primo è il nome del file che deve essere copiato, il secondo è il nome del file copia. Il programma creerà una pipe convenzionale e scriverà i contenuti del file da copiare

nella pipe. Il processo figlio leggerà il file dalla pipe e lo scriverà nel suo file di destinazione. Se per esempio invochiamo il programma come segue:

```
filecopy input.txt copy.txt
```

il file `input.txt` sarà scritto sulla pipe. Il processo figlio leggerà i contenuti del file e li scriverà nel file di destinazione `copy.txt`. Potete scrivere il programma utilizzando pipe in Windows o in UNIX.

Progetti di programmazione

Progetto 1 – Shell di UNIX e cronologia

Questo progetto consiste nel realizzare in linguaggio C un’interfaccia shell che accetta comandi utente ed esegue ogni comando in un processo separato. Questo progetto può essere completato in qualsiasi sistema Linux, UNIX o Mac OS X.

Un’interfaccia shell fornisce all’utente un prompt e resta in attesa di un comando. L’esempio seguente mostra il prompt `osh>` e il successivo comando dell’utente: `cat prog.c`. (Questo comando consente di visualizzare il file `prog.c` sul terminale mediante il comando UNIX `cat`).

```
osh> cat prog.c
```

Una tecnica per implementare un’interfaccia shell è di fare in modo che il processo padre legga prima di tutto ciò che l’utente scrive sulla linea di comando (in questo caso `cat prog.c`) e quindi crei un processo figlio separato che esegua il comando. Se non diversamente specificato, il processo padre attende che il figlio termini prima di continuare. Questo è simile come funzionalità alla creazione di un nuovo processo illustrata nella Figura 3.10. Tuttavia, le shell UNIX di solito permettono anche che il processo figlio resti in esecuzione in background o in maniera concorrente. Per fare ciò, si aggiunge una `&` alla fine del comando.

Così, se riscriviamo il comando precedente come

```
osh> cat prog.c &
```

i processi padre e figlio resteranno in esecuzione concorrentemente.

Il processo figlio viene creato utilizzando la chiamata di sistema `fork()` e il comando dell’utente viene eseguito utilizzando una delle chiamate di sistema della famiglia `exec()` (come descritto nel Paragrafo 3.3.1).

Un programma in C che fornisce le operazioni generali di una shell a riga di comando è mostrato nella Figura 3.36. La funzione `main()` presenta il prompt `osh>` e delinea le azioni da intraprendere dopo che l’input dell’utente è stato letto. La funzione `main()` continua a ciclare finché `should_run` resta uguale a 1; quando l’utente immette `exit` al prompt, il programma imposterà `should_run` a 0 e terminerà.

Questo progetto è organizzato in due parti: (1) creazione del processo figlio ed esecuzione del comando nel processo figlio, (2) modifica della shell per consentire di mantenere una cronologia.

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

#define MAX_LINE 80 /* Lunghezza massima di un comando */

int main(void)
{
    char *args[MAX_LINE/2 + 1]; /* Argomenti della riga di comando */
    int should_run = 1; /* Flag per determinare quando uscire dal
programma */

    while (should_run) {
        printf("osh>");
        fflush(stdout);

        /**
         * Dopo aver letto l'input dell'utente i passi sono:
         * (1) creare un processo figlio usando la fork()
         * (2) il processo figlio invoca la execvp()
         * (3) se il comando include la &, il padre invoca la wait()
         */
    }

    return 0;
}
```

Figura 3.36 Schema di una semplice shell.

Parte I – Creazione di un processo figlio

Il primo compito è quello di modificare la funzione `main()` nella Figura 3.36 in modo da creare un processo figlio che esegua il comando specificato dall’utente. Ciò richiederà l’analisi di ciò che l’utente ha inserito, la sua suddivisione in token separati e la memorizzazione dei token in un array di stringhe di caratteri (`args` nella Figura 3.36). Per esempio, se l’utente immette il comando `ps -ael` al prompt `osh>`, i valori memorizzati nell’array `args` sono:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

L’array `args` sarà passato alla funzione `execvp()` che ha il seguente prototipo:

```
execvp (char *command, char *params[]);
```

Qui, `command` rappresenta il comando da eseguire e `params` memorizza i parametri per questo comando. In questo progetto, la funzione `execvp()` va invocata come `execvp(args[0], args)`. Assicuratevi di controllare se l'utente ha incluso un `&` per determinare se il processo padre deve o meno attendere la terminazione del figlio.

Parte II – Creazione della cronologia

Il compito successivo è quello di modificare il programma dell'interfaccia shell in modo da fornire una funzionalità di cronologia per consentire all'utente di accedere ai comandi immessi più di recente. Utilizzando la cronologia, l'utente sarà in grado di accedere a un massimo di 10 comandi. I comandi saranno numerati a partire da 1, e la numerazione proseguirà oltre il 10. Per esempio, se l'utente ha inserito 35 comandi, i 10 comandi più recenti saranno numerati da 26 a 35.

L'utente sarà in grado di elencare la cronologia dei comandi digitando

`history`

al prompt `osh>`. Si supponga, per esempio, che la cronologia dei comandi sia costituita, dal più al meno recente, da:

`ps, ls -l, top, cal, who, date`

Il comando `history` restituirà:

```
6 ps
5 ls -l
4 top
3 cal
2 who
1 date
```

Il vostro programma deve supportare due tecniche per il recupero dei comandi dalla cronologia:

1. Quando l'utente inserisce `!!` viene eseguito il comando più recente.
2. Quando l'utente immette un singolo `!` seguito da un numero intero `N`, viene eseguito l'`N`-esimo comando nella cronologia.

Proseguendo il nostro esempio precedente, se l'utente inserisce `!!` verrà eseguito il comando `ps`; se l'utente immette `!3`, verrà eseguito il comando `cal`. Ogni comando eseguito in questo modo deve essere visualizzato sullo schermo dell'utente. Il comando deve anche essere ricollocato nel buffer della cronologia come ultimo comando eseguito.

Il programma deve effettuare una gestione di base degli errori. Se non ci sono comandi nella cronologia all'immissione di `!!` deve essere visualizzato un messaggio `Nessun comando nella cronologia`. Se non c'è nessun comando corrispondente al numero inserito dopo un singolo `!` il programma deve visualizzare il messaggio `Comando non presente nella cronologia`.

Progetto 2 – Modulo del kernel di Linux per l’elenco dei task

In questo progetto verrà scritto un modulo del kernel che elenca tutti i task correnti in un sistema Linux. Assicurarsi prima di iniziare questo progetto di rivedere il progetto di programmazione del Capitolo 2, che si occupa di creazione di moduli del kernel Linux. Il progetto può essere realizzato utilizzando la macchina virtuale Linux fornita con questo testo.

Parte I – Iterazione lineare sui task

Come illustrato nel Paragrafo 3.1, il blocco di controllo di un processo (PCB) in Linux è rappresentato dalla struttura `task_struct`, che si trova nel file di include `<linux/sched.h>`. In Linux la macro `for_each_process()` permette di realizzare in modo semplice l’iterazione su tutti i task in corso nel sistema:

```
#include <linux/sched.h>
struct task_struct *task;
for each process(task) {
    /* A ogni iterazione task punta al prossimo task */
}
```

I vari campi di `task_struct` possono essere visualizzati mentre il programma cicla per mezzo della macro `for each process()`.

Parte I – Esercizio

Progettate un modulo del kernel che consenta di scorrere tutti i task del sistema utilizzando la macro `for_each_process()`. In particolare, si visualizzi il nome (noto come nome del file eseguibile), lo stato e l’ID di processo di ogni task. (Dovrete probabilmente leggere la struttura `task_struct` in `<linux/sched.h>` per ottenere i nomi di questi campi). Scrivete questo codice nel punto di ingresso del modulo in modo che i suoi contenuti verranno visualizzati nel buffer di log del kernel, che può essere letto con il comando `dmesg`. Per verificare che il codice funzioni correttamente, confrontate il contenuto del buffer di log del kernel con l’output del seguente comando, che elenca tutti i task del sistema:

```
ps -el
```

I due valori dovrebbero essere molto simili. Poiché i task sono dinamici, tuttavia, è possibile che alcuni task appaiano in una lista ma non nell’altra.

Parte II – Iterazione sui task con un albero di ricerca in profondità

La seconda parte di questo progetto prevede l’iterazione su tutti i task nel sistema utilizzando una ricerca in profondità (DFS: Depth-First Search). A titolo di esempio: l’iterazione DFS sui processi nella Figura 3.8 è 1, 8415, 8416, 9298, 9204, 2, 6, 200, 3028, 3610, 4005.

Linux mantiene il suo albero dei processi come una serie di liste. Esaminando la struttura `task_struct` in `<linux/sched.h>`, vediamo due oggetti `struct list_head`:

```
children
e
sibling
```

Questi oggetti sono puntatori a una lista dei figli del task e a una lista dei suoi fratelli. Linux mantiene anche riferimenti al task `init` (`struct task_struct init_task`). Utilizzando queste informazioni e le macro per operare sulle liste, siamo in grado di scorrere i figli di `init` come segue:

```
struct task_struct *task;
struct list_head *list;
list_for_each(list, &init_task->children) {
    task = list_entry(list, struct task_struct, sibling);
    /* task punta al successivo figlio nella lista */
}
```

La macro `list_for_each()` riceve due parametri, entrambi di tipo `struct list_head`:

- un puntatore alla testa della lista da scorrere;
- un puntatore al nodo di testa della lista da scorrere.

A ogni passo di `list_for_each()` il primo parametro è impostato alla struttura `list` del figlio successivo. Questo valore viene poi utilizzato per ottenere ogni struttura nella lista usando la macro `list_entry()`.

Parte II – Esercizio

Progettate un modulo del kernel che itera su tutte le attività del sistema, a partire dal task `init`, utilizzando un albero DFS. Proprio come nella prima parte di questo progetto, stampate il nome, lo stato e il PID di ogni task. Eseguite questa iterazione nel modulo di ingresso del kernel in modo che il suo output venga visualizzato nel buffer di log.

Se si stampano tutti i task del sistema è possibile vedere molti più task rispetto a quelli ottenuti con il comando `ps -ael`, poiché alcuni thread appaiono come figli, ma non si presentano come processi ordinari. Pertanto, per verificare l'output dell'albero DFS, utilizzate il comando

```
ps -eLf
```

Questo comando elenca tutti i task del sistema, inclusi i thread. Per verificare di aver effettivamente eseguito l'iterazione DFS in maniera appropriata, si dovranno esaminare le relazioni tra i vari task restituiti dal comando `ps`.

Note bibliografiche

La creazione di un processo, la sua gestione e la comunicazione tra processi in UNIX e Windows, rispettivamente, sono discussi in [Robbins e Robbins 2003] e [Russinovich e Solomon 2009]. [Love 2010] tratta il supporto ai processi del kernel di Linux e [Hart 2005] copre in dettaglio la programmazione di sistema in Windows. Un’analisi del modello multiprocesso utilizzato da Google Chrome è disponibile all’indirizzo <http://blog.chromium.org/2008/09/multi-process-architecture.html>.

Lo scambio di messaggi nei sistemi multicore è discusso in [Holland e Seltzer 2011]. [Baumann et al. 2009] trattano le questioni relative alle prestazioni nei sistemi a memoria condivisa e a scambio di messaggi. [Vahalia 1996] descrive la comunicazione tra processi nel sistema Mach.

L’implementazione del meccanismo RPC è analizzata in [Birrell e Nelson 1984]. [Staunstrup 1982] confronta chiamate di procedura e scambio di messaggi. [Harold 2005] offre una panoramica della programmazione con le socket in Java.

[Hart 2005] e [Robbins e Robbins 2003] trattano il tema delle pipe nei sistemi Windows e UNIX, rispettivamente.

Bibliografia

- [Baumann et al. 2009] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, P. Simon, T. Roscoe, A. Schupbach e A. Singhania, “The multikernel: a new OS architecture for scalable multicore systems”, p. 29–44, 2009.
- [Birrell e Nelson 1984] A. D. Birrell e B. J. Nelson, “Implementing Remote Procedure Calls”, ACM Transactions on Computer Systems, Volume 2, N. 1, p. 39–59, 1984.
- [Harold 2005] E. R. Harold, *Java Network Programming*, 3° ed., O’Reilly & Associates, 2005.
- [Hart 2005] J. M. Hart, *Windows System Programming*, 3° ed., Addison-Wesley, 2005.
- [Holland e Seltzer 2011] D. Holland e M. Seltzer, “Multicore OSes: looking forward from 1991, er, 2011”, Proceedings of the 13th USENIX conference on Hot topics in operating systems, p. 33–33, 2011.
- [Love 2010] R. Love, *Linux Kernel Development*, 3° ed., Developer’s Library, 2010.
- [Robbins e Robbins 2003] K. Robbins e S. Robbins, *Unix Systems Programming: Communication, Concurrency and Threads*, 2° ed., Prentice Hall, 2003.
- [Russinovich e Solomon 2009] M. E. Russinovich e D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*, 5° ed., Microsoft Press, 2009.
- [Staunstrup 1982] J. Staunstrup, “Message Passing Communication Versus Procedure Call Communication”, Software—Practice and Experience, Volume 12, N. 3, p. 223–234, 1982.
- [Vahalia 1996] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall, 1996.

CAPITOLO

4

OBIETTIVI DEL CAPITOLO

- Introduzione del concetto di thread, l'unità fondamentale nell'utilizzo della CPU alla base dei moderni sistemi multithread.
- Analisi delle API per l'uso dei thread in Java, Windows e Pthread.
- Analisi di diverse strategie per il threading implicito.
- Studio delle questioni relative alla programmazione multithread.
- Descrizione del supporto ai thread nei sistemi operativi Windows e Linux.

Thread

Nel modello introdotto nel Capitolo 3 si è assunto che un processo sia un programma in esecuzione con un unico percorso di controllo. Quasi tutti i sistemi operativi moderni permettono tuttavia che un processo possa avere più percorsi di controllo chiamati *thread*. In questo capitolo sono introdotti diversi concetti associati ai sistemi elaborativi *multithread*, tra i quali un'approfondita descrizione delle API per le librerie di thread di Pthreads, Windows e Java. Si esaminano molti aspetti legati alla programmazione multithread, e il modo in cui essa influenza la progettazione dei sistemi operativi; infine, si analizza il modo in cui alcuni sistemi operativi moderni, come Windows e Linux, gestiscono i thread a livello kernel.

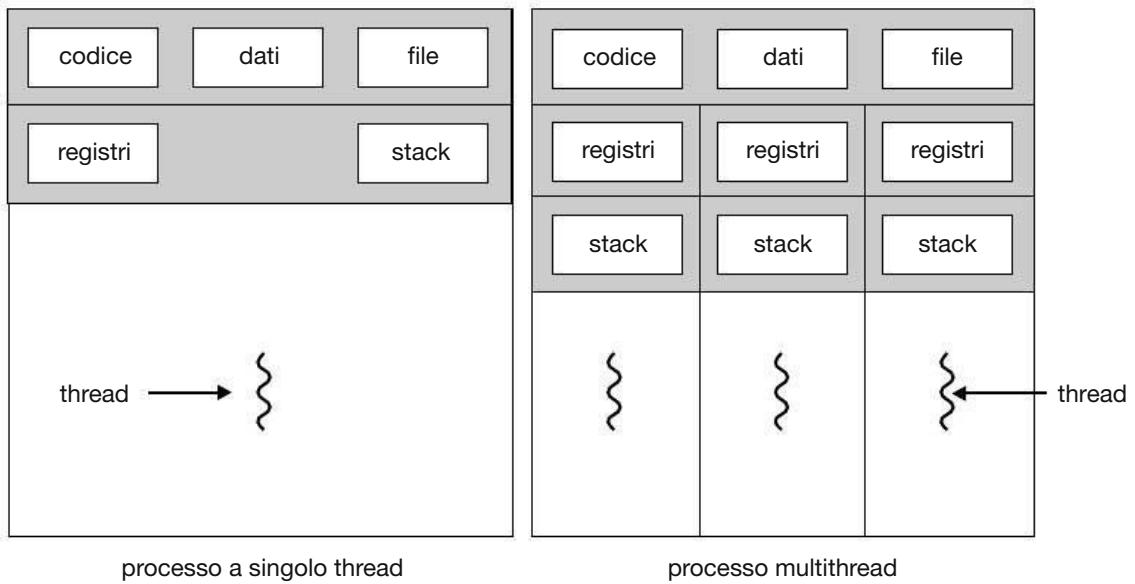


Figura 4.1 Processi a singolo thread e multithread.

4.1 Introduzione

Un thread è l’unità di base d’uso della CPU e comprende un identificatore di thread (ID), un contatore di programma, un insieme di registri, e una pila (*stack*). Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali. Un processo tradizionale, chiamato anche **processo pesante** (*heavyweight process*), è composto da un solo thread. Un processo multithread è in grado di svolgere più compiti in modo concorrente. La Figura 4.1 mostra la differenza tra un processo tradizionale, a **singolo thread**, e uno **multithread**.

4.1.1 Motivazioni

La maggior parte delle applicazioni per i moderni computer è **multithread**. Di solito, un’applicazione si codifica come un processo a sé stante comprendente più thread di controllo: un Web browser potrebbe avere un thread per la rappresentazione sullo schermo di immagini e testo, mentre un altro thread si occupa del reperimento dei dati dalla rete; un word processor potrebbe avere un thread per la rappresentazione grafica, uno per la risposta all’input da tastiera e uno per la correzione ortografica e grammaticale eseguita in background. Le applicazioni possono anche essere progettate per sfruttare le capacità di elaborazione sui sistemi multicore. Tali applicazioni possono eseguire diverse attività che utilizzano intensivamente la CPU in parallelo sui diversi core di elaborazione.

In alcune situazioni una singola applicazione deve poter gestire molti compiti simili tra loro. Per esempio, un server web accetta dai client richieste di pagine web, immagini, suoni e altro. Per un server web intensamente utilizzato potrebbero esservi

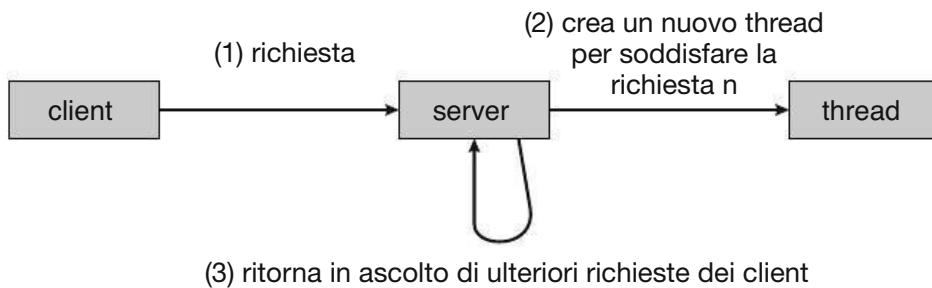


Figura 4.2 Architettura di server multithread.

molti (forse migliaia di) client che vi accedono in modo concorrente; se il server web fosse eseguito come un processo tradizionale a singolo thread, esso sarebbe in grado di soddisfare un solo client alla volta e un client potrebbe dover aspettare molto a lungo prima che la sua richiesta venga servita.

Una soluzione è eseguire il server come un singolo processo che accetta richieste. Quando ne riceve una, il server crea un processo separato per eseguirla. In effetti, questo metodo di creazione di processi era molto usato prima che si diffondesse la possibilità di gestione dei thread. La creazione dei processi è molto onerosa, sia a livello di tempi sia di risorse; se il nuovo processo si deve occupare degli stessi compiti del processo corrente, non c'è alcuna ragione di accettare l'intero carico che la sua creazione comporta. Generalmente è più conveniente impiegare un processo multithread. Nel caso del server web, se il processo è multithread, il server creerà un thread distinto per ricevere le richieste dei client; in presenza di una richiesta, anziché creare un altro processo, si creerebbe un altro thread per soddisfarla. Il tutto è illustrato nella Figura 4.2.

I thread hanno anche un ruolo primario nei sistemi che impiegano le RPC (*remote procedure call*); si tratta di un sistema che permette la comunicazione tra processi, fornendo un meccanismo di comunicazione simile alle normali chiamate di funzione o procedura (Capitolo 3). Di solito, i server RPC sono multithread. Quando riceve un messaggio, il server ne delega la gestione a un thread separato, in questo modo può gestire diverse richieste in modo concorrente.

Infine, molti kernel di sistemi operativi sono ormai multithread, con i singoli thread dedicati a specifici servizi – per esempio, la gestione dei dispositivi periferici, della memoria o delle interruzioni. È il caso di Solaris, il cui kernel utilizza un insieme di thread specializzati nella gestione delle interruzioni; Linux usa un thread a livello kernel per la gestione della memoria libera del sistema.

4.1.2 Vantaggi

I vantaggi della programmazione multithread si possono classificare in quattro categorie principali.

1. **Tempo di risposta.** Rendere multithread un'applicazione interattiva può permettere a un programma di continuare la sua esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo

di risposta all’utente. Questa caratteristica è particolarmente utile nella progettazione di interfacce utente. Per esempio, si consideri quello che succede quando un utente fa clic su un pulsante che provoca l’esecuzione di un’operazione che richieda diverso tempo. Un’applicazione a thread singolo resterebbe bloccata fino al completamento dell’operazione. Al contrario, se l’operazione viene eseguita in un thread separato, l’applicazione rimane attiva per l’utente.

2. **Condivisione delle risorse.** I processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa e lo scambio di messaggi. Queste tecniche devono essere esplicitamente messe in atto dal programmatore. Tuttavia, i thread condividono per default la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice e dei dati consiste nel fatto che un’applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d’indirizzi.
3. **Economia.** Assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto. È difficile misurare empiricamente la differenza nell’overhead richiesto per creare e gestire un processo invece che un thread, tuttavia la creazione e la gestione dei processi richiedono in generale molto più tempo. In Solaris, per esempio, la creazione di un processo richiede un tempo circa trenta volte maggiore di quello necessario per la creazione di un thread, un cambio di contesto per un processo richiede un tempo pari a circa cinque volte quello necessario per un thread.
4. **Scalabilità.** I vantaggi della programmazione multithread sono ancora maggiori nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo su distinti core di elaborazione. Invece un processo con un singolo thread può funzionare solo su un processore, indipendentemente da quanti ve ne siano a disposizione. Il multithreading su una macchina con più processori incrementa il parallelismo. Esploreremo questo tema nel prossimo paragrafo.

4.2 Programmazione multicore

Nei primi tempi della progettazione dei calcolatori, in risposta alla necessità di una maggiore potenza di calcolo, si è passati dai sistemi a singola CPU ai sistemi multi-CPU. Una simile tendenza, più recente, nel progetto dell’architettura dei sistemi consiste nel montare diversi *core* di elaborazione su un unico chip; ogni unità appare al sistema operativo come un processore separato (Paragrafo 1.3.2). Sia che i core appartengano allo stesso chip o a più chip, noi chiameremo questi sistemi multicore o multiprocessore. La programmazione multithread offre un meccanismo per un utilizzo più efficiente di questi multiprocessori e aiuta a sfruttare al meglio la concorrenza. Si consideri un’applicazione con quattro thread. In un sistema con un singolo core, “esecuzione concorrente” significa solo che l’esecuzione dei thread è avvicendata nel tempo (*interleaved*) (Figura 4.3), perché la CPU è in grado di eseguire un solo thread

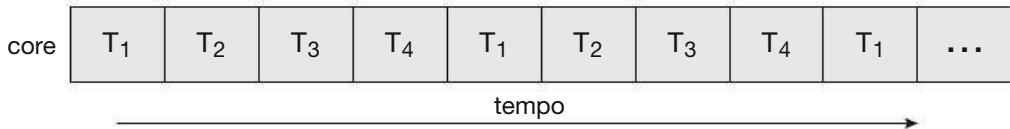


Figura 4.3 Esecuzione concorrente su un sistema a singolo core.

alla volta. Su un sistema multicore, invece, “esecuzione concorrente” significa che i thread possono funzionare in parallelo, dal momento che il sistema può assegnare thread diversi a ciascun core (Figura 4.4).

Si noti la distinzione tra *parallelismo* e *concorrenza*. Un sistema è parallelo se può eseguire simultaneamente più di un task. Un sistema concorrente, invece, supporta più di un task consentendo a tutti di progredire nell'esecuzione. È dunque possibile avere concorrenza senza parallelismo. Prima dell'avvento dei processori SMP e delle architetture multicore, la maggior parte dei sistemi era dotata di un singolo processore. Gli scheduler della CPU erano progettati per fornire l'illusione di parallelismo, mediante una rapida commutazione tra processi nel sistema, consentendo in tal modo a ogni processo di fare progressi. Tali processi erano eseguiti in maniera concorrente, ma non in parallelo.



LEGGE DI AMDAHL

La legge di Amdahl è una formula che permette di determinare i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di ulteriori core di elaborazione, nel caso di applicazioni che contengono sia componenti seriali (non parallele) sia componenti parallele. Indicando con S la porzione di applicazione che deve essere realizzata serialmente su un sistema con N core di elaborazione, la formula è la seguente:

$$\text{incremento di velocità} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Come esempio, supponiamo di avere un'applicazione che sia al 75% parallela e al 25% seriale. Se eseguiamo l'applicazione su un sistema con due core di elaborazione, possiamo ottenere un incremento di velocità pari a 1,6. Se aggiungiamo due core (per un totale di quattro core), l'incremento di velocità è pari a 2,28.

Un fatto interessante circa la legge di Amdahl è che per N che tende all'infinito, l'incremento di velocità converge a $1/S$. Per esempio, se il 40 per cento di un'applicazione viene eseguita in maniera seriale, il massimo aumento di velocità è di 2,5 volte, indipendentemente dal numero di core di elaborazione che aggiungiamo.

Questo è il principio fondamentale che sta dietro la legge di Amdahl: la porzione seriale di un'applicazione può avere un effetto dominante sulle prestazioni ottenibili con l'aggiunta di ulteriori core.

Alcuni sostengono che la legge di Amdahl non prende in considerazione i miglioramenti di prestazioni hardware ottenuti nella progettazione dei sistemi multicore moderni. Tali argomentazioni suggeriscono che la legge di Amdahl possa diventare inapplicabile al crescere del numero di core di elaborazione nei moderni computer.

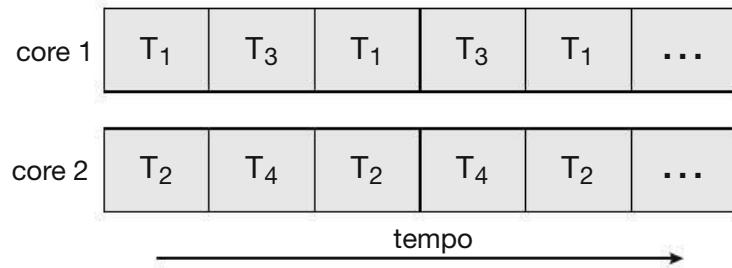


Figura 4.4 Esecuzione parallela su un sistema multicore.

Poiché i sistemi sono passati da decine a migliaia di thread, i progettisti di CPU hanno migliorato le prestazioni con l’aggiunta di hardware per rendere più efficiente l’esecuzione dei thread. Molte delle CPU Intel moderne offrono il supporto a due thread per core, mentre la CPU T4 di Oracle supporta otto thread per core. Ciò significa che più thread possono essere caricati nel core, garantendo una commutazione più rapida. I computer multicore continueranno senza dubbio ad aumentare il numero di core e a migliorare il supporto hardware ai thread.

4.2.1 Le sfide della programmazione

La tendenza verso i sistemi multicore tiene costantemente sotto pressione i progettisti di sistemi operativi e i programmatore di applicazioni, affinché utilizzino al meglio core multipli. I progettisti di sistemi operativi devono scrivere algoritmi di scheduling che utilizzano diversi core per permettere un’esecuzione parallela come quella mostrata nella Figura 4.4. Per i programmatore di applicazioni, la sfida consiste nel modificare programmi esistenti e progettare nuovi programmi multithread. In generale, possiamo individuare nelle cinque aree seguenti le principali sfide della programmazione dei sistemi multicore.

- Identificazione dei task.** Consiste nell’esaminare le applicazioni al fine di individuare aree separabili in task distinti e concorrenti che possano essere eseguiti in parallelo su core distinti.
- Bilanciamento.** Nell’identificare i task eseguibili in parallelo, i programmatore devono far sì che i vari task esegano compiti di mole e valore confrontabili. In alcuni casi si verifica che un determinato task non contribuisca al processo complessivo tanto quanto gli altri; in questi casi per eseguire il task può non valere la pena di utilizzare core separati.
- Suddivisione dei dati.** Proprio come le applicazioni sono divise in task separati, i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati core distinti.
- Dipendenze dei dati.** I dati a cui i task accedono devono essere esaminati per verificare le dipendenze tra due o più task. In casi in cui un task dipende dai dati forniti da un altro, i programmatore devono assicurare che l’esecuzione dei task sia

sincronizzata in modo da soddisfare queste dipendenze. Esamineremo queste strategie nel Capitolo 5.

5. **Test e debugging.** Quando un programma funziona in parallelo su unità multiple, vi sono diversi possibili flussi di esecuzione. Effettuare i test e il debugging di questi programmi è per natura più difficile rispetto al caso di applicazioni con un singolo thread.

Molti sviluppatori di software sostengono, in ragione dei problemi appena esposti, che l'avvento dei sistemi multicore richiederà in futuro un approccio interamente nuovo al progetto dei sistemi software. (Allo stesso modo, molti docenti di informatica ritengono che lo sviluppo del software debba essere insegnato ponendo maggiore enfasi sulla programmazione parallela).

4.2.2 Tipi di parallelismo

In generale, esistono due tipi di parallelismo: parallelismo dei dati e parallelismo delle attività. Il **parallelismo dei dati** riguarda la distribuzione di sottoinsiemi dei dati su più core di elaborazione e l'esecuzione della stessa operazione su ogni core. Si consideri, per esempio, l'operazione di somma dei valori contenuti in un vettore di dimensione N . In un sistema con un singolo core, un thread sommerebbe semplicemente gli elementi da $[0]$ a $[N - 1]$. In un sistema dual-core, invece, il thread A , in esecuzione sul core 0, potrebbe sommare gli elementi da $[0]$ a $[N/2 - 1]$, mentre il thread B , in esecuzione sul core 1, potrebbe sommare gli elementi da $[N/2]$ a $[N - 1]$. I due thread sarebbero in esecuzione in parallelo su core di elaborazione distinti.

Il **parallelismo delle attività** prevede la distribuzione di attività (thread), e non di dati, su più core. Ogni thread realizza un'operazione distinta e thread differenti possono operare sugli stessi dati o su dati diversi. Si consideri ancora il nostro esempio. A differenza della situazione precedente, un esempio di parallelismo delle attività potrebbe coinvolgere due thread, ciascuno dei quali esegue un'unica operazione statistica sull'array di elementi. I thread operano ancora in parallelo su core separati, ma ciascuno sta eseguendo un'operazione diversa.

Fondamentalmente, quindi, il parallelismo dei dati comporta la distribuzione dei dati su più core e il parallelismo delle attività la distribuzione dei task su più core. In pratica, tuttavia, poche applicazioni applicano rigorosamente parallelismo dei dati o parallelismo delle attività. Nella maggior parte dei casi le applicazioni utilizzano un ibrido di queste due strategie.

4.3 Modelli di supporto al multithreading

I thread possono essere distinti in **thread a livello utente** e **thread a livello kernel**: i primi sono gestiti sopra il livello del kernel e senza il suo supporto; i secondi, invece, sono gestiti direttamente dal sistema operativo. Praticamente tutti i sistemi operativi moderni supportano i thread nel kernel, compresi Windows, Linux, Mac OS X e Solaris.

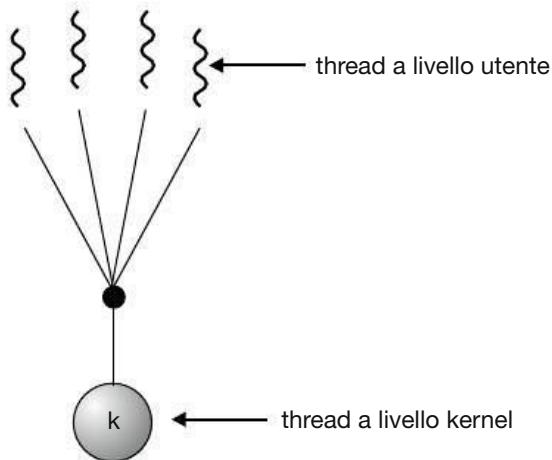


Figura 4.5 Modello da molti a uno.

In ogni caso, deve esistere una relazione tra i thread a livello utente e i thread a livello kernel. In questo paragrafo analizziamo tre opzioni comuni: il modello da molti a uno, il modello da uno a uno e il modello da molti a molti.

4.3.1 Modello da molti a uno

Il modello da molti a uno (Figura 4.5) fa corrispondere molti thread a livello utente a un singolo thread a livello kernel. La gestione dei thread risulta efficiente perché viene effettuata da una libreria di thread nello spazio utente (le librerie di supporto ai thread verranno discusse nel Paragrafo 4.4). Tuttavia l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante. Inoltre, poiché un solo thread alla volta può accedere al kernel, è impossibile eseguire thread multipli in parallelo in sistemi multicore; la libreria **green threads**, disponibile per Solaris e adottata nelle prime versioni di Java, usa questo modello. Tuttavia, pochissimi sistemi utilizzano ancora questo modello a causa della sua incapacità di trarre vantaggio dalla presenza di più core.

4.3.2 Modello da uno a uno

Il modello da uno a uno (Figura 4.6) mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel. Questo modello offre un grado di concorrenza maggiore rispetto al precedente, poiché anche se un thread invoca una chiamata di sistema bloccante, è possibile eseguire un altro thread; il modello permette anche l'esecuzione dei thread in parallelo nei sistemi multiprocessore. L'unico svantaggio di questo modello è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel. Poiché il carico dovuto alla creazione di un thread a livello kernel può sovraccaricare le prestazioni di un'applicazione, la maggior parte delle realizzazioni di questo modello limita il numero di thread supportabili dal sistema. I sistemi operativi Linux, insieme alla famiglia dei sistemi operativi Windows, adottano il modello da uno a uno.

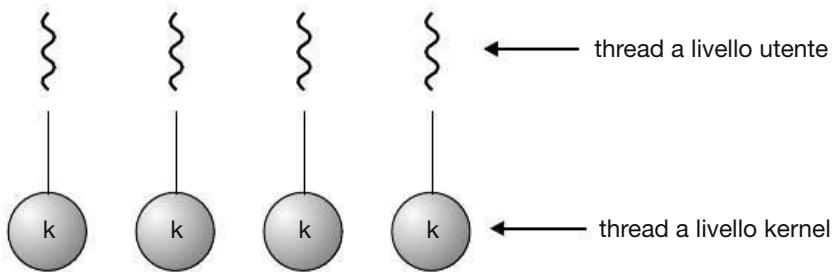


Figura 4.6 Modello da uno a uno.

4.3.3 Modello da molti a molti

Il modello da molti a molti (Figura 4.7) mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel; quest'ultimo può essere specifico per una certa applicazione o per un particolare calcolatore (ad un'applicazione potrebbero essere assegnati più thread a livello kernel in un'architettura multiprocessore rispetto quanti ne verrebbero assegnati in una con singola CPU).

Consideriamo l'effetto di questo modello sulla concorrenza. Nonostante il modello da molti a uno permetta ai programmati di creare tanti thread a livello utente quanti ne desiderino, non viene garantita una concorrenza reale, poiché il meccanismo di scheduling del kernel può scegliere un solo thread alla volta. Il modello da uno a uno permette una maggiore concorrenza, ma i programmati devono stare attenti a non creare troppi thread all'interno di un'applicazione (in qualche caso si possono avere limitazioni sul numero di thread che si possono creare). Il modello da molti a molti non ha alcuno di questi difetti: i programmati possono creare liberamente i thread che ritengono necessari, e i corrispondenti thread a livello kernel si possono eseguire in parallelo nelle architetture multiprocessore. Inoltre, se un thread impiega una chiamata di sistema bloccante, il kernel può schedulare un altro thread.

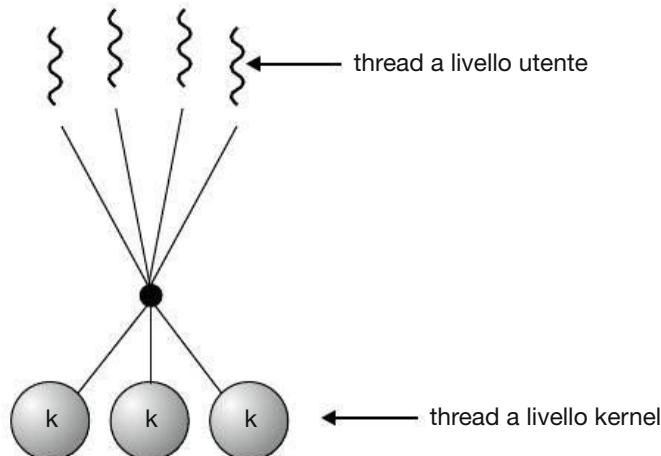


Figura 4.7 Modello da molti a molti.

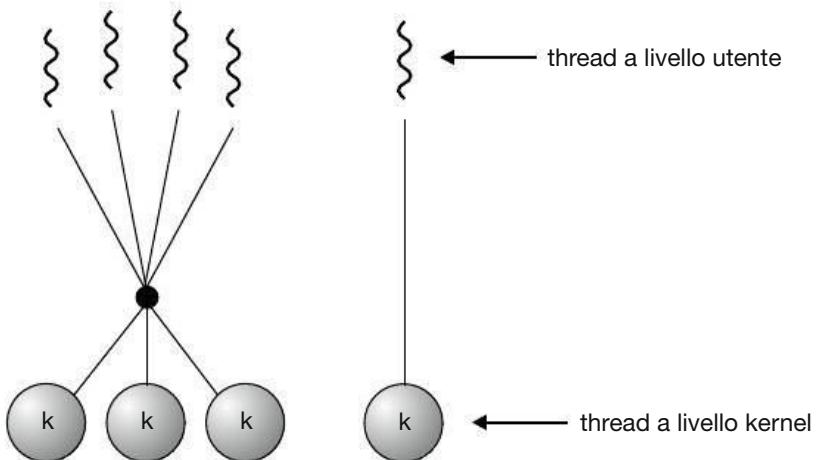


Figura 4.8 Modello a due livelli.

Una variante del modello da molti a molti mantiene la corrispondenza fra più thread utente e un numero minore o uguale di thread del kernel, ma permette anche di vincolare un thread utente a un solo thread del kernel. Questa variante è anche chiamata **modello a due livelli** (Figura 4.8). Solaris supporta il modello a due livelli nelle versioni precedenti a Solaris 9, a partire dalla quale il sistema segue il modello da uno a uno.

4.4 Librerie dei thread

Una **libreria dei thread** fornisce al programmatore una API per la creazione e la gestione dei thread. I metodi con cui implementare una libreria dei thread sono essenzialmente due. Nel primo, la libreria è collocata interamente a livello utente, senza fare ricorso al kernel. Il codice e le strutture dati per la libreria risiedono tutti nello spazio utente. Questo implica che invocare una funzione della libreria si traduce in una chiamata locale a una funzione nello spazio utente e non in una chiamata di sistema.

Il secondo metodo consiste nell'implementare una libreria a livello kernel, supportata direttamente dal sistema operativo. In questo caso, il codice e le strutture dati per la libreria si trovano nello spazio del kernel. Invocare una funzione della API per la libreria provoca, generalmente, una chiamata di sistema al kernel.

Attualmente, sono tre le librerie di thread maggiormente in uso: Pthreads di POSIX, Windows e Java. Pthreads, estensione dello standard POSIX, può essere realizzata sia come libreria a livello utente sia a livello kernel. La libreria di thread Windows è una libreria a livello kernel per i sistemi Windows. La API per la creazione dei thread in Java è gestibile direttamente dai programmi Java. Tuttavia, data la peculiarità di funzionamento della JVM, quasi sempre eseguita sopra un sistema operativo che la ospita, la API di Java per i thread è solitamente implementata per mezzo di una libreria dei thread del sistema ospitante. Perciò, i thread di Java sui sistemi Windows sono in ef-

fetti implementati mediante la API Windows; sui sistemi UNIX e Linux, invece, si adopera spesso Pthreads.

Nel threading di POSIX e di Windows tutti i dati dichiarati a livello globale, ovvero al di fuori di ogni funzione, sono condivisi tra tutti i thread appartenenti allo stesso processo. Poiché Java non ha alcuna nozione di dati globali, l'accesso ai dati condivisi deve essere assegnato in modo esplicito tra i thread. I dati locali di una funzione sono in genere memorizzati nello stack. Dal momento che ogni thread ha un proprio stack, ogni thread ha la propria copia di dati locali.

Nel seguito affrontiamo i fondamenti della generazione dei thread in queste tre librerie. Come esempio dimostrativo progetteremo un programma multithread che calcola la somma dei primi N interi non negativi in un thread separato, in simboli:

$$sum = \sum_{i=0}^N i$$

Se, per esempio, $N = 5$, si avrebbe $sum = 15$, la somma dei numeri da 0 a 5. Ciascuno dei tre programmi funzionerà inserendo nella riga di comando l'indice superiore N della sommatoria; inserendo 8, quindi, si otterrà come risultato la somma dei valori interi da 0 a 8.

Prima di procedere con i nostri esempi di creazione di thread, introduciamo due strategie generali per la creazione di più thread: il *threading asincrono* e il *threading sincrono*. Nel threading asincrono, una volta che il genitore crea un thread figlio, riprende la sua esecuzione, in modo che genitore e figlio restino in esecuzione concorrentemente. Ogni thread viene eseguito in modo indipendente rispetto agli altri thread e il thread genitore non ha bisogno di conoscere quando suo figlio termina. Poiché i thread sono indipendenti, vi è di solito poca condivisione dei dati tra i thread. Il threading asincrono è la strategia utilizzata nel server multithread illustrata nella Figura 4.2.

Il threading sincrono si verifica quando il thread genitore crea uno o più figli e attende che tutti terminino prima di riprendere l'esecuzione. Tale strategia è detta **fork-join**. In questo caso, i thread creati dal genitore svolgono il lavoro in maniera concorrente, ma il genitore non può continuare fino al completamento di questo lavoro. Una volta che un thread ha completato il suo lavoro esso termina e si unisce (*join*) con il genitore. Solo dopo che tutti i figli si sono uniti al genitore, questo può riprendere l'esecuzione. In genere, il threading sincrono comporta una significativa condivisione dei dati tra i thread. Per esempio, il thread genitore può combinare i risultati calcolati dai suoi figli. Tutti i seguenti esempi utilizzano il threading sincrono.

4.4.1 Pthreads

Col termine **Pthreads** ci si riferisce allo standard POSIX (IEEE 1003.1c) che definisce una API per la creazione e la sincronizzazione dei thread. Non si tratta di una *implementazione*, ma di una *specifica* del comportamento dei thread; i progettisti di sistemi operativi possono realizzare la specifica come meglio credono. Sono molti i sistemi che implementano le specifiche Pthreads; per la maggior parte si tratta di sistemi di tipo UNIX, tra cui Linux, Mac OS X e Solaris. Anche se Windows non supporta nativamente Pthreads, esiste un'implementazione disponibile.

vamente Pthread, sono disponibili alcune implementazioni di terze parti per Windows.

Il programma C multithread nella Figura 4.9 esemplifica la API Pthreads tramite il calcolo di una sommatoria eseguito da un thread apposito. Nei programmi Pthreads, i nuovi thread iniziano l'esecuzione a partire da una funzione specificata. Nel programma in esame si tratta della funzione `runner()`. All'inizio dell'esecuzione del programma c'è un unico thread di controllo che parte da `main()`; dopo una fase dinizializzazione, `main()` crea un secondo thread che inizia l'esecuzione dalla funzione `runner()`. Entrambi i thread condividono la variabile globale `sum`.

Esaminiamo il programma più attentamente. Tutti i programmi che impiegano la libreria Pthreads devono includere il file d'intestazione `pthread.h`. La dichiarazione di variabili `pthread_t tid` specifica l'identificatore per il thread da creare. Ogni thread ha un insieme di attributi che includono la dimensione dello stack e informazioni di scheduling. La dichiarazione `pthread_attr_t attr` riguarda la struttura dati per gli attributi del thread, i cui valori si assegnano con la chiamata di funzione `pthread_attr_init(&attr)`. Poiché non sono stati esplicitamente forniti valori per gli attributi, si usano quelli predefiniti. (Nel Capitolo 6 saranno esaminati alcuni degli attributi di scheduling offerti dalle API Pthreads). La chiamata di funzione `pthread_create()` crea un nuovo thread. Oltre all'identificatore del thread e ai suoi attributi, si passa anche il nome della funzione da cui il nuovo thread inizierà l'esecuzione, in questo caso la funzione `runner()`, e il numero intero fornito come parametro alla riga di comando, e individuato da `argv[1]`.

A questo punto il programma ha due thread: il thread iniziale (o genitore), in `main()`; e il thread che esegue la somma (o figlio), in `runner()`. Il programma segue dunque la strategia fork-join descritta in precedenza: dopo aver creato il figlio, il thread padre ne attende il completamento chiamando la funzione `pthread_join()`. Il secondo thread termina quando invoca la funzione `pthread_exit()`. Quando il thread che esegue la somma termina, il thread padre produce in uscita il valore condiviso `sum` della sommatoria.

Questo programma di esempio crea un solo thread. Con il crescente predominio dei sistemi multicore, la scrittura di programmi che contengono più thread è diventata sempre più comune. Un metodo semplice per l'attesa su più thread usando `pthread_join()` è di racchiudere l'operazione all'interno di un semplice ciclo `for`. Per esempio, è possibile effettuare il join di dieci thread utilizzando il codice Pthread mostrato nella Figura 4.10.

4.4.2 Thread in Windows

La tecnica usata dalla libreria Windows per la creazione dei thread può richiamare, per molti versi, quella di Pthreads. Illustriamo la API Windows nel programma C mostrato dalla Figura 4.11. Si noti che per utilizzare la API Windows è necessario includere il file d'intestazione `windows.h`.

Come nella versione di Pthreads della Figura 4.9, i dati condivisi da thread separati – nella fattispecie, `Sum` – sono globali (il tipo `DWORD` è un intero di 32 bit privo di

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

int sum; /* questo dato è condiviso dai thread */
void *runner(void *param); /* i thread chiamano questa funzione */

int main(int argc, char *argv[])
{
    pthread_t tid; /* identificatore del thread */
    pthread_attr_t attr; /* insieme di attributi del thread */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }

    /* reperisce gli attributi predefiniti */
    pthread_attr_init(&attr);
    /* crea il thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* attende la terminazione del thread */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* Il thread comincia l'esecuzione da questa funzione */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Figura 4.9 Programma multithread in linguaggio C che impiega la API Pthreads.

```
#define NUM_THREADS 10

/* array di thread da unire */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Figura 4.10 Codice Pthread per effettuare il join di 10 thread.

segno). Dobbiamo inoltre definire la funzione `Summation()` che il nuovo thread eseguirà. A questa funzione è passato un puntatore a `void`, che in Windows è `LPVOID`. Il thread che esegue questa funzione imposta la variabile globale `Sum` al valore della sommatoria da 0 fino al parametro passato a `Summation()`.

Nella API Windows i nuovi thread si generano tramite la funzione `CreateThread()`, che – proprio come in Pthreads – accetta una serie di attributi del thread come parametri. Tali attributi includono le informazioni sulla sicurezza, la dimensione dello stack e un indicatore (*flag*) per segnalare se il thread debba avere inizio nello stato d’attesa. Ci serviremo, nel programma, dei valori di default di questi attributi, che inizialmente non pongono il thread in stato d’attesa, bensì lo rendono eseguibile dallo scheduler della CPU. Una volta creato il nuovo thread, il thread iniziale deve attenderne il completamento prima di produrre in uscita il valore di `Sum`, poiché esso è computato dal nuovo thread. Come si ricorderà, nel programma Pthread (Figura 4.9) il thread iniziale era posto in attesa della terminazione del nuovo thread tramite la funzione `pthread_join()`. La chiamata equivalente nella API Windows è `WaitForSingleObject()`, che causa la sospensione del thread iniziale fintanto che il nuovo thread non abbia terminato.

In situazioni che richiedono l’attesa della terminazione di più thread viene utilizzata la funzione `WaitForMultipleObjects()`, alla quale vengono passati quattro parametri:

1. il numero di oggetti da attendere;
2. un puntatore al vettore di oggetti;
3. un flag che indica se tutti gli oggetti sono stati segnalati;
4. la durata del timeout (o il valore `INFINITE`).

Per esempio, se `THandles` è un array di `HANDLE` di thread di dimensione N , il thread principale può attendere che tutti i suoi figli terminino con la seguente istruzione:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

```

#include <windows.h>
#include <stdio.h>
DWORD Sum; /* il dato è condiviso tra i thread */

/* il thread viene eseguito in questa funzione separata */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }

    /* crea il thread */
    ThreadHandle = CreateThread(
        NULL, /* attributi di sicurezza di default */
        0, /* dimensione di default dello stack */
        Summation, /* funzione del thread */
        &Param, /* parametri alla funzione del thread */
        0, /* flag di crezione di default */
        &ThreadId); /* restituisce l'identificatore del thread */

    if (ThreadHandle != NULL) {
        /* adesso aspetta la fine del thread */
        WaitForSingleObject(ThreadHandle,INFINITE);

        /* chiude l'handle del thread */
        CloseHandle(ThreadHandle);

        printf("sum = %d\n",Sum);
    }
}

```

Figura 4.11 Programma multithread in C con l'utilizzo dell'API Windows.

4.4.3 Thread Java

I thread rappresentano il paradigma fondamentale per l'esecuzione dei programmi in ambiente Java; il linguaggio Java, con la propria API, è provvisto di una ricca gamma di funzionalità per la generazione e la gestione dei thread. Tutti i programmi scritti in Java incorporano almeno un thread di controllo – persino un semplice programma, costituito soltanto da un metodo `main()`, è eseguito dalla JVM come un singolo thread. I thread Java sono disponibili su tutti i sistemi che dispongono di una JVM, tra cui Windows, Linux e Mac OS X. L'API Java per i thread è anche disponibile per le applicazioni Android.

In un programma Java vi sono due tecniche per la generazione dei thread. Una è creare una nuova classe derivata dalla classe `Thread` e “sovrascrivere” (*override*) il suo metodo `run()`. L'alternativa, usata più comunemente, consiste nella definizione di una classe che implementi l'interfaccia `Runnable`, definita come segue:

```
public interface Runnable  
{  
    public abstract void run();  
}
```

Per implementare `Runnable`, una classe è tenuta a definire il metodo `run()`. Il codice che implementa `run()` sarà eseguito in un thread distinto.

La Figura 4.12 mostra la versione Java di un programma multithread che calcola la somma degli interi da 0 a N . La classe `Summation` implementa l'interfaccia `Runnable`. La generazione del thread prevede che si crei un'istanza della classe `Thread` passando al costruttore un oggetto `Runnable`.

La creazione di un oggetto di classe `Thread` non equivale a generare un nuovo thread: è il metodo `start()` che avvia effettivamente il nuovo thread. L'invocazione del metodo `start()` ha il duplice effetto di:

1. allocare la memoria e inizializzare un nuovo thread nella JVM;
2. chiamare il metodo `run()`, cosa che rende il thread eseguibile dalla JVM. Si osservi come il metodo `run()` non sia mai chiamato per via diretta, ma solo tramite la chiamata del metodo `start()`.

All'avvio del programma che calcola la somma, la JVM crea due thread. Il primo è il thread genitore, che inizia a essere eseguito dal metodo `main()`. Il secondo thread, figlio del primo, ha origine quando il metodo `start()` è invocato sull'oggetto di classe `Thread`. L'esecuzione di questo thread figlio inizia dal metodo `run()` della classe `Summation`. Dopo aver restituito il valore della somma, il thread termina all'uscita dal proprio metodo `run()`.

La condivisione dei dati tra thread diversi è semplice in Windows e in Pthreads: i dati condivisi sono semplicemente dichiarati globali. In quanto linguaggio orientato agli oggetti puro, Java non contempla la nozione di variabile globale: la condivisione dei dati fra più thread in Java avviene tramite il passaggio di riferimenti a uno stesso oggetto. Nel programma in Java della Figura 4.12 il thread principale e quello che

```

class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}

public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>"); }
    }
}

```

Figura 4.12 Programma Java per il calcolo della sommatoria dei primi N interi.

calcola la somma condividono un oggetto di classe `Sum`. A tale oggetto condiviso si accede attraverso gli appositi metodi `getSum()` e `setSum()`. Ci si potrebbe chiedere perché non si usi un oggetto di classe `Integer` anziché definire una nuova classe `sum`. La ragione è che la classe `Integer` è **immutabile** – ovvero, una volta impostato il valore di una sua istanza, non può più cambiare.

Ricordiamo che i thread genitori nelle librerie Pthreads e Windows usano, rispettivamente, `pthread_join()` e `WaitForSingleObject()` per attendere la conclusione del thread che esegue la somma prima di procedere. Il metodo `join()` in Java fornisce una simile funzionalità. (Si noti che `join()` può sollevare una `InterruptedException`, che nel codice abbiamo deciso di non gestire). Se il genitore deve attendere la terminazione di diversi thread, il metodo `join()` può essere racchiuso in un ciclo `for` simile a quello mostrato per Pthreads nella Figura 4.10.

4.5 Threading implicito

Con la continua crescita di elaborazione multicore si profilano all’orizzonte applicazioni contenenti centinaia, o anche migliaia, di thread. La progettazione di tali applicazioni non è un’impresa semplice: i programmati devono affrontare non solo le sfide descritte nel Paragrafo 4.2, ma anche ulteriori difficoltà. Queste difficoltà, che riguardano la correttezza dei programmi, sono trattate nei Capitoli 5 e 7.

Un modo per affrontare tali ostacoli e gestire al meglio la progettazione di applicazioni multithread è il trasferimento della creazione e della gestione del threading dagli sviluppatori di applicazioni ai compilatori e alle librerie di runtime.

LA JVM E IL SISTEMA OPERATIVO OSPITE

La macchina virtuale del linguaggio Java (JVM) è solitamente implementata sulla base di un sistema operativo sottostante (Figura 16.10). Questo assetto permette alla JVM di nascondere i dettagli del sistema operativo e di offrire un ambiente astratto e coerente che consente ai programmi scritti in Java di essere eseguiti su qualsiasi piattaforma che disponga di una JVM. Le specifiche della JVM non prescrivono come i thread Java debbano corrispondere ai servizi del sistema operativo sottostante, lasciando i dettagli all’implementazione. Il sistema operativo Windows XP, per esempio, adotta il modello da uno a uno; perciò, ogni thread Java di una JVM installata su questo sistema corrisponde a un thread a livello kernel. Sui sistemi che adottano il modello da molti a molti, come il Tru64 UNIX, i thread di Java sono associati a thread sottostanti secondo il modello da molti a molti. La JVM per Solaris impiegava inizialmente il modello da molti a uno (la libreria *green threads*, citata in precedenza); versioni successive della JVM per Solaris hanno utilizzato il modello da molti a molti. A partire da Solaris 9, i thread di Java vengono associati ai thread sottostanti secondo il modello da uno a uno. Oltre a ciò, può esistere una relazione tra la libreria dei thread di Java e la libreria dei thread del sistema operativo residente. Le varie versioni della JVM per la famiglia di sistemi operativi Windows, per esempio, potrebbero ricorrere alla API Windows al fine di implementare i thread di Java; i sistemi Linux, Solaris e Mac OS X potrebbero impiegare la API Pthreads.

Questa strategia, chiamata **threading implicito**, è diventata oggi molto comune. In questo paragrafo esploriamo tre approcci alternativi per la progettazione di programmi multithread in grado di sfruttare i processori multicore attraverso il threading implicito.

4.5.1 Gruppi di thread

Nel Paragrafo 4.1 abbiamo descritto un server web multithread in cui, per ogni richiesta ricevuta, il server crea un thread distinto per fornire il servizio richiesto. Nonostante la creazione di un thread distinto sia molto più vantaggiosa della creazione di un nuovo processo, tuttavia un server multithread presenta diversi problemi. Il primo riguarda il tempo richiesto per la creazione del thread prima di poter soddisfare la richiesta, considerando anche il fatto che questo thread sarà terminato non appena avrà completato il proprio lavoro. La seconda questione è più problematica: se si permette che tutte le richieste concorrenti siano servite da un nuovo thread, non si è posto un limite al numero di thread concorrentemente attivi nel sistema. Un numero illimitato di thread potrebbe esaurire le risorse del sistema, come il tempo di CPU o la memoria. L'impiego dei **gruppi di thread** (*thread pool*) è una possibile soluzione a questo problema.

L'idea generale è quella di creare un certo numero di thread alla creazione del processo, e organizzarli in un gruppo (*pool*) in cui attendano di eseguire il lavoro che gli sarà richiesto. Quando un server riceve una richiesta, attiva un thread del gruppo – se ce n'è uno disponibile – e gli passa la richiesta; dopo aver completato il suo lavoro, il thread rientra nel gruppo d'attesa. Se il gruppo non contiene alcun thread disponibile, il server attende fino a che un thread non si libera. I vantaggi sono i seguenti:

1. il servizio di una richiesta tramite un thread esistente è più rapido dell'attesa della creazione di un nuovo thread;
2. un gruppo di thread limita il numero di thread esistenti a un certo istante; ciò è particolarmente rilevante per sistemi che non possono sostenere un elevato numero di thread concorrenti.
3. Separare il task da svolgere dalla meccanica della sua creazione ci permette di utilizzare diverse strategie per l'esecuzione di tale task. Per esempio, si potrebbe pianificare l'esecuzione del task dopo un ritardo di tempo o periodicamente.

Il numero di thread di un gruppo si può determinare tramite euristiche che considerano fattori come il numero di CPU nel sistema, la quantità di memoria fisica e il numero atteso di richieste concorrenti da parte dei client. Architetture più raffinate per la gestione dei gruppi di thread possono correggere dinamicamente il numero di thread di un gruppo secondo l'utilizzazione del sistema. Queste architetture hanno l'ulteriore vantaggio di utilizzare gruppi più piccoli – comportando quindi un minore impegno di memoria – quando il carico del sistema è basso. Introdurremo una di queste architetture, Grand Central Dispatch di Apple, nei prossimi paragrafi.

La API Windows mette a disposizione diverse funzioni legate ai gruppi di thread, il cui uso è simile alla creazione di un thread tramite la funzione `Thread_Create()`

presentata al Paragrafo 4.4.2. Si definisce una funzione da eseguire in un nuovo thread, per esempio:

```
DWORD WINAPI PoolFunction(AVOID Param) {  
    /**/  
     * Questa funzione gira come nuovo thread.  
    **/  
}
```

Un puntatore a `PoolFunction()` è poi passato a una delle apposite funzioni nella API per i gruppi di thread, il che avvierà uno dei thread del gruppo. Una di tali apposite funzioni è `QueueUserWorkItem()`, che accetta tre parametri:

- `LPTHREAD_START_ROUTINE Function` – un puntatore alla funzione da eseguire in un nuovo thread;
- `PVOID Param` – il parametro passato a `Function`;
- `ULONG Flags` – indica come il gruppo di thread debba creare e gestire l'esecuzione del nuovo thread.

Un esempio di chiamata è:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

A seguito di questa invocazione, uno dei thread del gruppo richiamerà `PoolFunction()` per conto del programmatore. In questo esempio, `PoolFunction()` non riceve parametri, e il valore 0 di `Flags` indica l'assenza di indicazioni particolari per la creazione del thread.

Altre funzioni della API Windows per i gruppi di thread offrono servizi di invocazione periodica di funzioni, o invocazioni legate al completamento di un'operazione di I/O asincrona. Anche il package `java.util.concurrent` di API Java offre funzionalità per la gestione di gruppi di thread.

4.5.2 OpenMP

OpenMP è un insieme di direttive del compilatore e una API per programmi scritti in C, C++ o FORTRAN che fornisce il supporto per la programmazione parallela in ambienti a memoria condivisa. OpenMP definisce le **regioni parallele** come blocchi di codice eseguibili in parallelo. Gli sviluppatori di applicazioni inseriscono direttive del compilatore nei punti del codice dove vi sono regioni parallele e queste direttive istruiscono la libreria di runtime OpenMP per l'esecuzione della regione in parallelo. Il seguente programma C illustra l'uso di una direttiva del compilatore inserita prima della regione parallela contenente l'istruzione `printf()`:

```

#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* codice sequenziale */
    #pragma omp parallel
    {
        printf("I am a parallel region.");

    }
    /* codice sequenziale */
    return 0;
}

```

Quando OpenMP incontra la direttiva

```
# pragma omp parallel
```

crea tanti thread quanti sono i core di elaborazione del sistema. In questo modo, in un sistema dual-core vengono creati due thread, in un sistema quad-core quattro thread, e così via. Tutti i thread eseguono poi contemporaneamente la regione parallela. Quando un thread esce dalla regione parallela viene terminato.

OpenMP fornisce diverse direttive aggiuntive per l'esecuzione di porzioni di codice in parallelo, tra cui i cicli parallelizzati. Per esempio, supponiamo di avere due array **a** e **b** di dimensione **N**. Desideriamo sommare i valori in essi contenuti e inserire i risultati nell'array **c**. Possiamo eseguire questo task in parallelo utilizzando il seguente frammento di codice, contenente la direttiva del compilatore per parallelizzare i cicli **for**:

```

#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}

```

OpenMP divide il compito contenuto nel ciclo **for** fra i thread che ha creato in risposta alla direttiva:

```
# pragma omp parallel for
```

Oltre a fornire direttive per la parallelizzazione, OpenMP consente agli sviluppatori di scegliere tra diversi livelli di parallelismo. Si può per esempio impostare manualmente il numero di thread o indicare se i dati sono condivisi tra i thread o sono riser-

vati a un solo thread. OpenMP è disponibile su diversi compilatori open source e commerciali per sistemi Linux, Windows e Mac OS X. Esortiamo i lettori interessati a saperne di più su OpenMP a consultare la bibliografia alla fine del capitolo.

4.5.3 Grand Central Dispatch

Grand Central Dispatch (GCD), una tecnologia per i sistemi operativi Mac OS X e iOS di Apple, è una combinazione di estensioni del linguaggio C, una API e una libreria di runtime che permette agli sviluppatori di applicazioni di individuare sezioni di codice da eseguire in parallelo. Come OpenMP, GCD gestisce la maggior parte dei dettagli del threading.

GCD definisce estensioni ai linguaggi C e C++ chiamate **blocchi**. Un blocco è semplicemente un’unità di lavoro autocontenuta e viene specificato da un accento circonflesso ^ inserito prima di una coppia di parentesi graffe {}. Un semplice esempio di blocco è il seguente:

```
^{ printf ( "I am a block" ); }
```

GCD pianifica l’esecuzione runtime dei blocchi inserendoli in una **coda di dispacciamiento** (*dispatch queue*). Quando GCD rimuove un blocco dalla coda, lo assegna a un thread disponibile tra quelli presenti nel gruppo di thread che gestisce. GCD definisce due tipi di code di dispacciamento: *seriali* e *concorrenti*.

I blocchi posti in una coda seriale vengono prelevati secondo un ordine FIFO. Una volta che un blocco è stato rimosso dalla coda, deve essere completata la sua esecuzione prima del prelievo di un altro blocco. Ogni processo ha una propria coda seriale, chiamata **coda principale** (*main queue*). Gli sviluppatori possono creare code seriali locali a processi particolari. Le code seriali sono utili per assicurare l’esecuzione sequenziale delle diverse attività.

Anche i blocchi posti in una coda concorrente vengono rimossi secondo un ordine FIFO, ma è possibile prelevare più blocchi alla volta, permettendo così la loro esecuzione in parallelo. Ci sono tre code di spedizione concorrenti a livello di sistema, distinte in base alla priorità: priorità bassa, di default, e alta. Le priorità rappresentano un’approssimazione dell’importanza relativa dei blocchi. Com’è facile intuire, i blocchi con una priorità più alta vanno inseriti nella coda di spedizione con priorità alta.

Il seguente frammento di codice illustra come ottenere la coda concorrente con priorità di default e passare un blocco alla coda utilizzando la funzione `dispatch_async()`:

```
dispatch_queue_t queue = dispatch_get_global_queue
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{
    printf("I am a block.");
});
```

Internamente, il gruppo di thread di GCD è composto da thread POSIX. GCD gestisce attivamente il gruppo, consentendo così al numero di thread di crescere e rimpicciolirsi in base alle richieste delle applicazioni e alla capacità del sistema.

4.5.4 Altri approcci

I gruppi di thread, OpenMP e Grand Central Dispatch sono solo alcune delle molte tecnologie emergenti per la gestione di applicazioni multithread. Altri approcci commerciali includono librerie parallele e concorrenti, per esempio Threading Building Blocks (TBB) di Intel e diversi prodotti di Microsoft. Anche il linguaggio e l'API Java hanno visto significativi sviluppi verso il supporto alla programmazione concorrente. Un esempio degno di nota è il pacchetto `java.util.concurrent`, che supporta il threading隐式的.

4.6 Problematiche di programmazione multithread

In questo paragrafo si affrontano alcune problematiche legate al progetto di programmi multithread.

4.6.1 Chiamate di sistema `fork()` ed `exec()`

Nel Capitolo 3 è stato descritto l'uso della chiamata di sistema `fork()` per la creazione di un nuovo processo tramite la duplicazione di un processo esistente. In un programma multithread la semantica delle chiamate di sistema `fork()` ed `exec()` cambia.

Se un thread in un programma invoca la chiamata di sistema `fork()`, il nuovo processo potrebbe, in generale, contenere un duplciato di tutti i thread oppure del solo thread invocante. Alcuni sistemi UNIX includono entrambe le versioni.

La chiamata di sistema `exec()` di solito funziona nello stesso modo descritto nel Capitolo 3: se un thread invoca la chiamata di sistema `exec()`, il programma specificato come parametro della `exec()` sostituisce l'intero processo, inclusi tutti i thread.

L'uso delle due versioni della `fork()` dipende dall'applicazione. Se s'invoca la `exec()` immediatamente dopo la `fork()`, la duplicazione dei thread non è necessaria, poiché il programma specificato nei parametri della `exec()` sostituirà il processo. In questo caso conviene duplicare il solo thread chiamante. Tuttavia, se la `exec()` non segue immediatamente la `fork()`, il nuovo processo dovrebbe duplicare tutti i thread del processo genitore.

4.6.2 Gestione dei segnali

Nei sistemi UNIX si usano i **segnali** per comunicare ai processi il verificarsi di determinati eventi. Un segnale si può ricevere in modo sincrono o asincrono, secondo la sorgente e la ragione della segnalazione dell'evento. Indipendentemente dal modo di ricezione sincrono o asincrono, tutti i segnali seguono lo stesso schema:

1. all'occorrenza di un particolare evento si genera un segnale;
2. s'invia il segnale a un processo;
3. una volta ricevuto, il segnale deve essere gestito.

Come esempio, un accesso illegale alla memoria o una divisione per zero generano segnali sincroni. In questi casi, se un programma in esecuzione compie le suddette azioni, viene generato un segnale. I segnali sincroni s’inviano allo stesso processo che ha eseguito l’operazione causa del segnale (questo è il motivo per cui sono considerati sincroni).

Quando un segnale è causato da un evento esterno al processo in esecuzione, tale processo riceve il segnale in modo asincrono. Esempi di segnali di questo tipo sono la terminazione di un processo richiesta con specifiche combinazioni di tasti (come <control><C>) oppure la scadenza di un timer. Di solito un segnale asincrono s’invia a un altro processo.

Ogni segnale si può gestire in due modi:

1. tramite un gestore predefinito di segnali;
2. tramite un gestore di segnali definito dall’utente.

Per ogni segnale esiste un **gestore predefinito del segnale** che il kernel esegue quando deve gestire il segnale. La gestione predefinita è sostituibile da un **gestore del segnale definito dall’utente**, richiamato per gestire il segnale. Sia i segnali sincroni sia quelli asincroni sono gestibili in modi diversi: alcuni si possono semplicemente ignorare (per esempio, il ridimensionamento di una finestra); altri si possono gestire terminando l’esecuzione del programma (per esempio, un accesso illegale alla memoria).

Per i processi a singolo thread la gestione dei segnali è semplice: i segnali vengono sempre inviati al processo. Per i processi multithread si pone il problema del thread cui si deve inviare il segnale. In generale esistono le seguenti possibilità:

1. inviare il segnale al thread cui il segnale si riferisce;
2. inviare il segnale a ogni thread del processo;
3. inviare il segnale a specifici thread del processo;
4. definire un thread specifico per ricevere tutti i segnali diretti al processo.

Il metodo per recapitare un segnale dipende dal tipo di segnale. I segnali sincroni, per esempio, si devono inviare al thread che ha generato l’evento causa del segnale e non ad altri thread nel processo. Se si tratta di segnali asincroni la situazione non è invece così chiara; alcuni segnali asincroni, come il segnale che termina un processo (come <control><C>), si devono inviare a tutti i thread.

La funzione standard per l’invio di un segnale in UNIX è

```
kill(pid_t pid, int signal)
```

Questa funzione specifica il processo (`pid`) al quale un particolare segnale (`signal`) deve essere recapitato. La maggior parte delle versioni multithread di UNIX permette che per ciascun thread si indichino i segnali da accettare e quelli da bloccare. Quindi, alcuni segnali asincroni si potrebbero recapitare soltanto ai thread che non li bloccano. Tuttavia, poiché i segnali vanno gestiti una sola volta, di solito un segnale è recapitato solo al primo thread che non lo blocca. La API Pthreads POSIX dispone della funzione

```
pthread_kill(pthread_t tid, int signal)
```

che permette di specificare il thread (`tid`) cui recapitare il segnale.

Sebbene Windows non preveda la gestione esplicita dei segnali, questi si possono emulare con le **chiamate di procedure asincrone** (*asynchronous procedure call, APC*). Le funzioni APC permettono a un thread a livello utente di specificare la funzione da richiamare quando il thread riceve la comunicazione di un particolare evento. Come s'intuisce dal nome, una APC è grosso modo equivalente a un segnale asincrono di UNIX. Mentre tuttavia in un ambiente multithread UNIX necessita di un criterio di gestione dei segnali, il sistema delle APC è più semplice, poiché una APC è rivolta a un particolare thread e non a un processo.

4.6.3 Cancellazione dei thread

La **cancellazione dei thread** è l'operazione che permette di terminare un thread prima che completi il suo compito. Per esempio, se più thread eseguono una ricerca in modo concorrente in una base di dati e un thread riporta il risultato, gli altri thread possono essere cancellati. Una situazione analoga potrebbe verificarsi quando un utente preme il pulsante di terminazione di un browser Web per interrompere il caricamento di una pagina. Spesso il caricamento di una pagina è gestito da più thread: ogni immagine è carica da un thread separato; quando l'utente preme il pulsante di terminazione, tutti i thread che stanno caricando la pagina vengono cancellati.

Un thread da cancellare è spesso chiamato **thread bersaglio** (*target thread*). La cancellazione di un thread bersaglio può avvenire in due modi diversi:

1. **cancellazione asincrona.** Un thread fa immediatamente terminare il thread bersaglio;
2. **cancellazione differita.** Il thread bersaglio controlla periodicamente se deve terminare, in modo da effettuare la terminazione in maniera ordinata.

Si presentano difficoltà con la cancellazione nei casi in cui ci siano risorse assegnate a un thread cancellato, o se si cancella un thread mentre sta aggiornando dei dati che condivide con altri thread. Quest'ultimo caso è particolarmente problematico se si tratta di cancellazione asincrona. Il sistema operativo di solito si riappropria delle risorse di sistema usate da un thread cancellato, ma spesso non si riappropria di tutte le risorse. Quindi, la cancellazione di un thread in modo asincrono potrebbe non liberare una risorsa necessaria per tutto il sistema.

La cancellazione differita invece funziona tramite un thread che segnala la necessità di cancellare un certo thread bersaglio; la cancellazione avviene soltanto quando il thread bersaglio verifica se debba essere o meno cancellato. Questo metodo permette di programmare la verifica in un punto dell'esecuzione in cui il thread sia cancellabile senza problemi.

In Pthreads la cancellazione del thread viene avviata tramite la funzione `pthread_cancel()`. L'identificatore del thread da cancellare viene passato come parametro alla funzione. Il codice seguente illustra la creazione e la cancellazione di un thread:

```

pthread_t tid;

/* Crea il thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* Cancella il thread */
pthread_cancel(tid);

```

La chiamata della `pthread_cancel()` comporta solamente una richiesta di cancellazione del thread di destinazione: l'effettiva cancellazione dipende da come il thread di destinazione è impostato per la gestione della richiesta. Pthreads supporta tre modalità di cancellazione, ognuna definita da uno stato e da un tipo, come illustrato nella tabella che segue. Un thread può impostare il suo stato di cancellazione e il suo tipo usando una API.

Modalità	Stato	Tipo
Off	Disabilitato	-
Differita	Abilitato	Differita
Asincrona	Abilitato	Asincrona

Come risulta dalla tabella, Pthreads permette di disabilitare o abilitare la cancellazione dei thread. Ovviamente, un thread non può essere cancellato se la cancellazione è disabilitata; tuttavia, le richieste di cancellazione rimangono in sospeso, in modo che il thread possa in seguito abilitare la cancellazione e rispondere alla richiesta.

Il tipo di cancellazione predefinito è la cancellazione differita, secondo cui la cancellazione avviene solo quando un thread raggiunge un **punto di cancellazione** (*cancellation point*). Una tecnica per la creazione di un punto di cancellazione consiste nell'invocare la funzione `pthread_testcancel()`. In caso di richiesta di cancellazione in attesa, viene richiamata una funzione nota come **gestore della pulizia** (*cleanup handler*), che permette di rilasciare tutte le risorse che un thread può aver acquisito prima di eliminarlo.

Il codice seguente illustra la risposta di un thread a una richiesta di annullamento mediante cancellazione differita:

```

while (1) {
    /* fai qualche lavoro per un po' di tempo */
    /* . . . */

    /* verifica se vi sia una richiesta di cancellazione */
    pthread_testcancel();
}

```

A causa dei problemi descritti in precedenza, nella documentazione di Pthreads viene sconsigliata la cancellazione asincrona e per questa ragione noi non la tratteremo. Una nota interessante: su sistemi Linux, la cancellazione dei thread con l'uso dell'API Pthreads è gestita attraverso segnali (Paragrafo 4.6.2).

4.6.4 Dati specifici dei thread

Uno dei vantaggi principali della programmazione multithread è dato dal fatto che i thread appartenenti allo stesso processo ne condividono i dati. Tuttavia, in particolari circostanze, ogni thread può necessitare di una copia privata di certi dati, chiamati **dati specifici dei thread** (o TLS). Per esempio, in un sistema transazionale si può svolgere ciascuna transazione tramite un thread distinto e assegnare un identificatore unico per ogni transazione. Per associare ciascun thread al relativo identificatore si possono usare dati specifici dei thread.

È facile confondere i dati specifici dei thread con le variabili locali. Mentre le variabili locali sono visibili solo durante la chiamata di una singola funzione, i dati specifici sono visibili attraverso tutte le chiamate. In un certo senso, i dati specifici assomigliano ai dati statici, con la differenza che i dati specifici sono unici per ogni thread. La maggior parte delle librerie di thread – incluse Windows e Pthreads – e l'ambiente del linguaggio Java consentono l'impiego dei dati specifici di thread.

4.6.5 Attivazione dello scheduler

Un'ultima questione da affrontare in merito ai programmi multithread riguarda la comunicazione tra la libreria del kernel e la libreria per i thread, che può rendersi necessaria nel modello a due livelli e in quello da molti a molti (Paragrafo 4.3.3). È proprio grazie a questa forma di coordinamento che il numero dei thread nel kernel è modificabile dinamicamente, con l'obiettivo di conseguire le migliori prestazioni.

Molti sistemi che implementano o il modello da molti a molti o quello a due livelli collocano una struttura dati intermedia tra i thread del kernel e dell'utente. Questa struttura dati, normalmente nota come **processo leggero** o LWP (acronimo di *light-weight process*) è mostrata nella Figura 4.13. Dal punto di vista della libreria di thread a livello utente, l'LWP si presenta come un *processore virtuale* a cui l'applicazione può richiedere lo scheduling di un thread a livello utente. Ciascun LWP è associato a un thread del kernel, e sono proprio i thread del kernel che il sistema operativo pone in esecuzione sui processori fisici. Se un thread del kernel si blocca (mentre attende il completamento di un'operazione di I/O, per esempio) anche l'LWP si blocca. L'effetto a catena risale fino al thread a livello utente associato all'LWP, che si blocca anch'esso.

Per un'efficiente esecuzione un'applicazione può aver bisogno di un numero imprecisato di LPW. Si consideri un processo con prevalenza di elaborazione eseguito da un singolo processore. In questa situazione è eseguibile solo un thread per volta, dunque un LWP è sufficiente. Un'applicazione con prevalenza di I/O potrebbe, tuttavia, richiedere l'esecuzione di molteplici LWP. Di solito è necessario un LWP per ogni

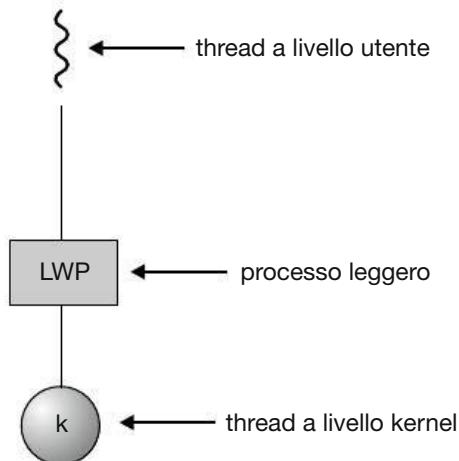


Figura 4.13 Processo leggero (LWP).

chiamata di sistema concorrente bloccante. Supponiamo, per esempio, che giungano allo stesso tempo cinque richieste differenti per la lettura di file. Sono necessari cinque LWP, nel caso in cui tutte le richieste restino in attesa del completamento dell’I/O nel kernel. Se un processo ha soltanto quattro LWP, la quinta richiesta deve attendere che uno degli LWP sia rilasciato dal kernel.

Uno dei modelli di comunicazione tra la libreria a livello utente e il kernel è conosciuto come **attivazione dello scheduler**. Il suo funzionamento è il seguente: il kernel fornisce all’applicazione una serie di processori virtuali (LWP), mentre l’applicazione esegue lo scheduling dei thread dell’utente sui processori virtuali disponibili. Inoltre, il kernel deve informare l’applicazione se si verificano determinati eventi, seguendo una procedura nota come **upcall**. Le upcall sono gestite dalla libreria dei thread mediante un apposito gestore, eseguito su un processore virtuale. Una situazione capace di innescare una upcall si verifica quando il thread di un’applicazione è sul punto di bloccarsi. In questo caso il kernel, tramite una upcall, informa l’applicazione che un thread è prossimo a bloccarsi, e identifica il thread in oggetto. Il kernel, quindi, assegna all’applicazione un nuovo processore virtuale. L’applicazione esegue un gestore della upcall su questo nuovo processore: il gestore salva lo stato del thread bloccante e rilascia il processore virtuale su cui era stato eseguito. Il gestore della upcall pianifica allora l’esecuzione di un altro thread sul nuovo processore virtuale. Quando si verifica l’evento atteso dal thread bloccante, il kernel fa un’altra upcall alla libreria dei thread per comunicare che il thread bloccato è nuovamente in condizione di essere eseguito. Il gestore di questa upcall necessita anch’esso di un processore virtuale: il kernel può crearne uno *ex novo*, o sottrarlo a un thread utente per prelazione. L’applicazione contrassegna il thread fino ad allora bloccato come pronto per l’esecuzione, ed esegue lo scheduling di un thread pronto per l’esecuzione su un processore virtuale disponibile.

4.7 Esempi di sistemi operativi

Abbiamo fin qui esaminato una serie di concetti e problematiche relativi ai thread. Concludiamo il capitolo descrivendo come i thread siano implementati nei sistemi Windows e Linux.

4.7.1 Thread di Windows

Il sistema operativo Windows offre la API Windows; si tratta dell'API principale della famiglia dei sistemi operativi di Microsoft (Windows 98, NT, 2000, XP e Windows 7). La maggior parte del contenuto di questo paragrafo riguarda questa intera famiglia di sistemi operativi.

Un'applicazione per l'ambiente Windows si esegue come un processo separato; ogni processo può contenere uno o più thread. La API Windows per la creazione dei thread è trattata nel Paragrafo 4.4.2. Il sistema Windows impiega il modello da uno a uno, descritto nel Paragrafo 4.3.2, secondo cui ogni thread a livello utente si associa a un thread del kernel.

I componenti generali di un thread includono:

- un identificatore di thread (ID), che identifica univocamente il thread;
- un insieme di registri che rappresenta lo stato del processore;
- uno stack utente, usato quando il thread è eseguito in modalità utente, e uno stack kernel, usato quando il thread è eseguito in modalità kernel;
- un'area di memoria privata, usata da diverse librerie run-time e dinamiche (DLL).

L'insieme di registri, gli stack e la memoria privata sono detti **contesto** del thread. Le strutture dati principali di un thread includono:

- ETHREAD (*executive thread block*);
- KTHREAD (*kernel thread block*);
- TEB (*thread environment block*).

I componenti chiave dell'ETHREAD sono un puntatore al processo a cui il thread appartiene e l'indirizzo della funzione in cui il thread inizia l'esecuzione. La struttura ETHREAD contiene anche un puntatore alla corrispondente struttura KTHREAD. Quest'ultima include informazioni per il thread relative allo scheduling e alla sincronizzazione. Inoltre, KTHREAD contiene lo stack kernel (usato quando il thread viene eseguito in modalità kernel) e un puntatore alla struttura TEB.

Le strutture ETHREAD e KTHREAD risiedono interamente nello spazio del kernel; ciò implica che solo il kernel vi può accedere. La struttura dati TEB appartiene invece allo spazio utente e vi si accede quando il thread è eseguito in modalità utente. Tra gli altri campi, il TEB contiene l'identificatore del thread, uno stack per la modalità utente e un vettore di dati specifici del thread. La struttura di un thread di Windows è illustrata nella Figura 4.14.

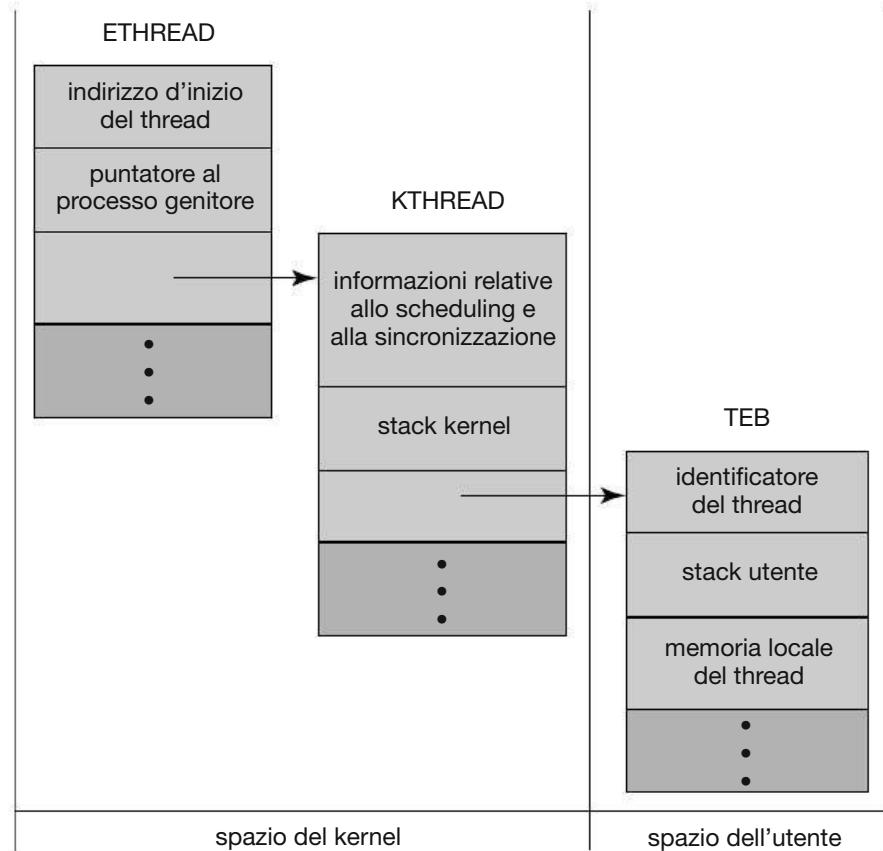


Figura 4.14 Strutture dati di un thread Windows.

4.7.2 Thread di Linux

Come si è visto nel Capitolo 3 Linux offre la chiamata di sistema `fork()` per duplicare un processo, e prevede inoltre la chiamata di sistema `clone()` per generare nuovo thread. Tuttavia Linux non distingue tra processi e thread, impiegando generalmente al loro posto il termine **task** in riferimento a un flusso del controllo nell’ambito di un programma. Quando `clone()` è invocata, riceve come parametro un insieme di indicatori (*flag*), al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio.

Alcuni di questi flag sono illustrati nella Figura 4.15. Per esempio, qualora

flag	significato
<code>CLONE_FS</code>	Condivisione delle informazioni sul file system
<code>CLONE_VM</code>	Condivisione dello stesso spazio di memoria
<code>CLONE_SIGHAND</code>	Condivisione dei gestori dei segnali
<code>CLONE_FILES</code>	Condivisione dei file aperti

Figura 4.15 Alcuni dei flag passati quando viene invocata la funzione `clone()`.

`clone()` riceva i flag `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, il task genitore e il task figlio condivideranno le medesime informazioni sul file system (come la directory attiva), lo stesso spazio di memoria, gli stessi gestori dei segnali e lo stesso insieme di file aperti. Adoperare `clone()` in questo modo è equivalente a creare thread come descritto in questo capitolo, dal momento che il task genitore condivide la maggior parte delle proprie risorse con il task figlio. Tuttavia, se nessuno dei flag è impostato al momento dell'invocazione di `clone()`, non si ha alcuna condivisione, e la funzionalità ottenuta diventa simile a quella fornita dalla chiamata di sistema `fork()`.

Questa condivisione a intensità variabile è resa possibile dal modo in cui un task è rappresentato nel kernel di Linux. Per ogni task, nel kernel esiste un'unica struttura dati (e precisamente, `struct task_struct`). Questa struttura, invece di memorizzare i dati del task relativo, utilizza dei puntatori ad altre strutture dove i dati sono effettivamente contenuti: per esempio, strutture dati che rappresentano l'elenco dei file aperti, le informazioni per la gestione dei segnali e la memoria virtuale. Quando si invoca `fork()`, si crea un nuovo task insieme con una *copia* di tutte le strutture dati del task genitore. Anche quando s'invoca la chiamata `clone()` si crea un nuovo task, ma anziché ricevere una copia di tutte le strutture dati, il nuovo task può *puntare* alle strutture dati del task genitore, a seconda dell'insieme di flag passati a `clone()`.

4.8 Sommario

Un thread è un percorso di controllo d'esecuzione all'interno di un processo. Un processo multithread contiene più percorsi di controllo diversi, ma che condividono lo stesso spazio d'indirizzi. I vantaggi della programmazione multithread includono un miglioramento del tempo di risposta, la condivisione di risorse all'interno del processo, il risparmio e la capacità di sfruttare le architetture dotate di più unità d'elaborazione.

I thread a livello utente sono thread visibili al programmatore e sconosciuti al kernel. Il kernel del sistema operativo gestisce thread a livello kernel. In generale, i thread a livello utente richiedono minor tempo per essere creati e gestiti rispetto a quelli a livello kernel, senza necessità d'intervento da parte del kernel. Ci sono tre diversi modelli che descrivono le relazioni fra thread a livello utente e a livello kernel: il modello da molti a uno associa più thread a livello utente a un singolo thread a livello kernel; il modello da uno a uno associa ciascun thread a livello utente a un corrispondente thread a livello kernel; il modello da molti a molti associa La maggioranza dei sistemi operativi moderni prevede la gestione dei thread a livello kernel: tra questi i sistemi Windows, Mac OS X, Linux e Solaris.

Per la creazione e la gestione dei thread le relative librerie forniscono una API al programmatore di applicazioni. Le tre più comuni librerie di thread sono: POSIX Pthreads, i thread di Windows e i thread Java.

Oltre a creare esplicitamente i thread utilizzando l'API fornita da una libreria, si può utilizzare il threading implicito, in cui la creazione e la gestione dei thread sono

demandate ai compilatori e a librerie run-time. Fra le soluzioni di threading implicito vi sono i gruppi di thread, OpenMP e Grand Central Dispatch.

I programmi multithread presentano molti aspetti critici per il programmatore, tra cui la semantica delle chiamate di sistema `fork()` ed `exec()`; altri aspetti sono per esempio la gestione dei segnali, la cancellazione, i dati privati dei thread e le attivazioni dello scheduler.

Esercizi di ripasso

- 4.1** Fornite due esempi di programmi nei quali il multithread offra prestazioni migliori rispetto a soluzioni con un singolo thread.
- 4.2** Quali sono due differenze tra i thread a livello utente e i thread a livello kernel? In quali circostanze un tipo è meglio dell’altro?
- 4.3** Descrivete le azioni intraprese da un kernel per cambiare contesto tra i thread a livello kernel.
- 4.4** Quali risorse vengono utilizzate quando si crea un thread? Come differiscono da quelle utilizzate quando si crea un processo?
- 4.5** Assumete che un sistema operativo mappi i thread a livello utente sul kernel utilizzando il modello molti a molti e che la mappatura avvenga tramite LWP. Assumete inoltre che il sistema permetta agli sviluppatori di creare dei thread real-time da utilizzare in sistemi real-time. È necessario vincolare un thread real-time a un LWP? Fornite una spiegazione.

Esercizi

- 4.6** Descrivete due esempi di programmazione in cui il multithreading *non* offre prestazioni migliori rispetto alla corrispondente soluzione a singolo thread.
- 4.7** In quali circostanze una soluzione basata sulla programmazione multithread che sfrutta thread multipli del kernel offre prestazioni migliori di una soluzione a singolo thread su un sistema monoprocessore?
- 4.8** Quali tra i seguenti componenti dello stato di un programma sono condivisi tra thread in un processo multithread?
 - a. Valori dei registri.
 - b. Memoria heap.
 - c. Variabili globali.
 - d. Stack.
- 4.9** È possibile che una soluzione multithread, impiegando thread multipli a livello utente, consegua prestazioni migliori su un sistema multiprocessore piuttosto che su un sistema a singolo processore? Motivate la risposta.

4.10 Nel Capitolo 3 abbiamo illustrato il browser Google Chrome e il suo modo di aprire ogni nuovo sito web in un processo separato. Si sarebbero ottenuti gli stessi vantaggi se Chrome fosse stato progettato per aprire ogni nuovo sito web in un thread separato? Motivate la vostra risposta.

4.11 È possibile avere la concorrenza, ma non il parallelismo? Motivate la vostra risposta.

4.12 Usando la legge di Amdahl, calcolate il guadagno in termini di velocità di un'applicazione che ha una componente parallela del 60 per cento in caso di utilizzo di (a) due core di elaborazione e (b) quattro core di elaborazione.

4.13 Determinate se i seguenti problemi presentano un parallelismo dei dati o un parallelismo delle attività.

- Il programma statistico multithread descritto nell'Esercizio 4.21.
- Il validatore di soluzioni di Sudoku multithread descritto nel Progetto 1 di questo capitolo.
- Il programma di ordinamento multithread descritto nel Progetto 2 di questo capitolo.
- Il server web multithread descritto nel Paragrafo 4.1.

4.14 Un sistema con due processori dual-core ha quattro processori disponibili per lo scheduling. Su questo sistema è in esecuzione un'applicazione con uso intensivo della CPU. Tutta la fase di input, in cui deve essere aperto un unico file, viene gestita all'avvio del programma. Allo stesso modo, tutto l'output è gestito appena prima che il programma termini, mediante il salvataggio dei risultati del programma in un unico file. Tra l'avvio e la terminazione, il programma utilizza esclusivamente la CPU. Il vostro compito è quello di migliorare le prestazioni di questa applicazione rendendola multithread. L'applicazione viene eseguita in un sistema che utilizza il modello da uno a uno (ogni thread utente viene mappato in un thread del kernel).

- Quanti thread creerete per gestire l'input e l'output? Spiegatelo.
- Quanti thread creerete per la porzione di applicazione a uso intensivo della CPU? Spiegatelo.

4.15 Si consideri il seguente frammento di codice:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* processo figlio */
    fork();
    thread_create(...);
}
fork();
```

- a. Quanti processi distinti vengono creati?
- b. Quanti thread distinti vengono creati?

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

#include <types.h>

int value = 0;
void *runner(void *param); /* il thread */

int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;

    pid = fork();

    if (pid == 0) { /* processo figlio */
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf("CHILD: value = %d", value); /* LINEA C */
    }
    else if (pid > 0) { /* processo padre */
        wait(NULL);
        printf("PARENT: value = %d", value); /* LINEA P */
    }
}

void *runner(void *param) {
    value = 5;
    pthread_exit(0);
}
```

Figura 4.16 Programma C dell’Esercizio 4.17.

4.16 Come descritto nel Paragrafo 4.7.2 Linux non fa distinzione tra processi e thread, cosicché un task apparirà più affine a un processo, oppure a un thread, a seconda dell’insieme di flag passati alla chiamata di sistema `clone()`. Tuttavia altri sistemi operativi, come Windows, trattano processi e thread in maniera diversa. In genere, per descrivere un processo, questi sistemi usano strutture dati contenenti un puntatore per ciascun thread appartenente al processo. Ponete a confronto queste due tecniche per rappresentare i processi e i thread nel kernel.

4.17 Il programma contenuto nella Figura 4.16 utilizza la API Pthreads. Quali dati in uscita verrebbero prodotti dal programma alla LINEA C e alla LINEA P?

4.18 Considerate un sistema multicore e un programma multithread scritto con il modello da molti a molti. Ipotizziamo un numero più alto di thread a livello utente nel programma rispetto al numero di processori nel sistema. Analizzate che cosa implica, in termini di prestazioni, ciascuna delle seguenti possibilità.

- a. Il numero di thread del kernel assegnati al programma è minore del numero di core.
- b. I thread del kernel assegnati al programma sono in numero uguale al numero dei core.
- c. Il numero di thread del kernel assegnati al programma è maggiore del numero di core, ma minore del numero di thread a livello utente.

4.19 Pthread fornisce una API per la gestione della cancellazione dei thread. La funzione `pthread_setcancelstate()` viene utilizzata per impostare lo stato di cancellazione. Il suo prototipo si presenta così:

```
pthread_setcancelstate(int state, int *oldstate)
```

I due possibili valori per lo stato sono `PTHREAD_CANCEL_ENABLE` e `PTHREAD_CANCEL_DISABLE`. Utilizzando il frammento di codice illustrato nella Figura 4.17, fornite esempi di due operazioni che sarebbero adatte da eseguire tra le chiamate di disabilitazione e abilitazione della cancellazione.

```
int oldstate;

pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);

/* Quali operazioni eseguire in questo punto? */

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Figura 4.17 Programma C dell'Esercizio 4.19.

Problemi di programmazione

4.20 Modificate Il Problema di programmazione 3.20 del Capitolo 3, che richiede di progettare un gestore di PID, scrivendo un programma multithread per testare la vostra soluzione all'Esercizio 3.20. Dovrete creare un certo numero di thread, per esempio 100, e ogni thread richiederà un PID, resterà sospeso per un periodo di tempo casuale e quindi rilascerà il PID. (La sospensione per un periodo di tempo casuale approssima il tipico utilizzo dei PID, in cui un PID viene assegnato a un nuovo processo, il processo viene eseguito e quindi termina, e il PID, al termine del processo, viene rilasciato). Sui sistemi UNIX e Linux questa sospensione è realizzata attraverso la funzione `sleep()`, alla quale viene passato

un intero che rappresenta il numero di secondi. Questo problema verrà modificato nel Capitolo 5.

- 4.21** Scrivete un programma multithread che calcoli diversi valori statistici su una lista di numeri. Questo programma riceverà una serie di numeri dalla riga di comando e quindi creerà tre thread di lavoro separati. Il primo thread determinerà la media dei numeri, il secondo determinerà il valore massimo e il terzo determinerà il valore minimo. Si supponga per esempio che al programma vengano passati i numeri interi:

90 81 78 95 79 72 85

Il programma risponderà:

```
Il valore medio è 82
Il valore minimo è 72
Il valore massimo è 95
```

Le variabili che rappresentano la media, il minimo e il massimo saranno salvate a livello globale. I thread di lavoro potranno impostare questi valori, e il thread genitore li restituirà una volta che i figli avranno terminato. (Si potrebbe ovviamente espandere questo programma con la creazione di thread aggiuntivi per determinare altri valori statistici, come la mediana e la deviazione standard).

- 4.22** Un modo interessante per calcolare π è quello di utilizzare una tecnica nota come *Monte Carlo*, che coinvolge la randomizzazione. Questa tecnica funziona così: supponiamo di avere un cerchio inscritto in un quadrato, come mostrato nella Figura 4.18. (Si assuma che il raggio di questo cerchio sia 1). Innanzitutto, generiamo una serie di punti casuali come semplici coppie (x, y) di coordinate. Questi punti devono cadere nel quadrato. Del numero totale di punti casuali che vengono generati, alcuni cadranno anche all'interno del cerchio.

Successivamente, eseguendo il seguente calcolo, stimiamo il valore di π :

$$\pi = 4 \times (\text{numero di punti interni al cerchio}) / (\text{numero totale di punti})$$

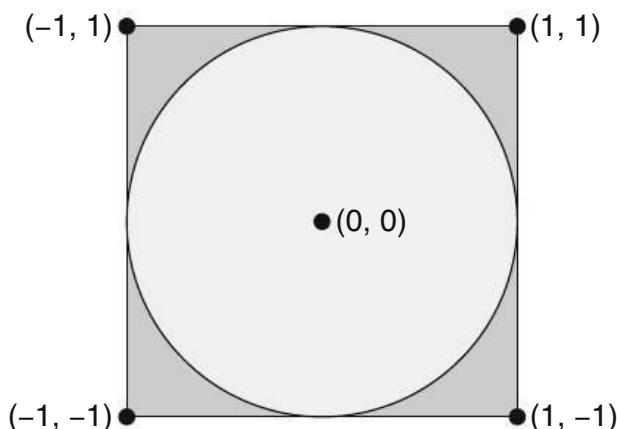


Figura 4.18 La tecnica di Monte Carlo per il calcolo di π .

Scrivete una versione multithread di questo algoritmo che crea un thread separato per generare un certo numero di punti casuali. Il thread conterà il numero di punti che cadono all'interno del cerchio e salverà il risultato in una variabile globale. Alla terminazione di questo thread, il thread genitore calcolerà e mostrerà il valore stimato di π . Vale la pena effettuare diversi esperimenti variando il numero di punti casuali generati. Come regola generale, maggiore è il numero di punti, migliore è l'approssimazione che si ottiene.

Tra il codice sorgente per questo testo mettiamo a disposizione un esempio di programma che fornisce una tecnica per generare numeri casuali, oltre a determinare se il punto casuale (x, y) cada all'interno del cerchio.

I lettori interessati ai dettagli sul metodo di *Monte Carlo* per la stima di π possono consultare la bibliografia alla fine di questo capitolo. Nel Capitolo 5 modificheremo questo esercizio grazie all'uso di nuovo materiale introdotto in tale capitolo.

- 4.23** Ripetete il Problema 4.22, ma invece di usare un thread separato per generare punti casuali, utilizzate OpenMP per parallelizzare la generazione di punti. Fate attenzione a non inserire il calcolo di π nella regione parallela, perché tale valore deve essere calcolato una volta soltanto.
- 4.24** Scrivete un programma multithread che generi numeri primi. Il programma deve avere il seguente comportamento: l'utente avvia l'esecuzione del programma e inserisce un numero alla riga di comando; il programma crea un thread distinto che riporta tutti i numeri primi minori o uguali al numero inserito dall'utente.
- 4.25** Modificate il server basato sulle socket della Figura 3.21 nel Capitolo 3 in modo che esso dedichi un thread separato a ciascuna richiesta del client.
- 4.26** La successione di Fibonacci inizia con 0, 1, 1, 2, 3, 5, 8,

Essa è definita da:

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2}$$

Scrivete un programma multithread che generi la successione di Fibonacci. Il programma deve avere il seguente comportamento: l'utente avvia l'esecuzione del programma e inserisce alla riga di comando il numero di termini della successione di Fibonacci che il programma deve generare. Il programma crea un thread separato per la generazione dei numeri di Fibonacci, ma colloca i termini della successione in dati condivisi dai thread (un vettore è probabilmente la struttura dati più adatta). Quando il thread figlio conclude l'esecuzione, il thread genitore emette la sequenza generata dal figlio. Poiché il thread genitore non

può produrre in uscita la sequenza prima che il thread figlio abbia terminato, sarà necessario applicare la tecnica illustrata nel Paragrafo 4.4, per sincronizzare il thread genitore con il thread figlio.

4.27 Nel Problema 3.25 del Capitolo 3 si spiega come progettare un server echo tramite la API di Java per i thread. Tale server, tuttavia, è a singolo thread, ossia non è in grado di rispondere a richieste simultanee di più client finché il client attualmente servito non termini. Si modifichi la soluzione del Problema 3.25 affinché il server echo possa servire separatamente ciascun client.

Progetti di programmazione

Progetto 1 – Validatore di soluzioni di Sudoku

Un *Sudoku* è una griglia 9×9 in cui ogni colonna, ogni riga e ciascuna delle nove sottogrigli 3 \times 3 devono contenere tutte le cifre da 1 a 9. La Figura 4.19 mostra un esempio di *Sudoku* valido. Questo esercizio consiste nel progettare un'applicazione multithread per determinare se la soluzione di un *Sudoku* è valida. Ci sono diversi modi per rendere multithread una tale applicazione. Una strategia che suggeriamo è quella di creare thread per controllare i seguenti criteri:

- Un thread per verificare che ogni colonna contenga le cifre da 1 a 9
- Un thread per verificare che ogni riga contenga le cifre da 1 a 9
- Nove thread per verificare che ciascuna sottogriglia 3 \times 3 contenga le cifre da 1 a 9

Per la convalida di un *Sudoku* si utilizzerebbe in questo caso un totale di undici thread distinti. Siete comunque liberi di creare ancora più thread. Per esempio, invece di creare un thread che controlli tutte e nove le colonne, potete creare nove thread, ognuno dei quali controlla una singola colonna.

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figura 4.19 Soluzione di un *Sudoku* 9×9 .

Passaggio di parametri a ogni thread

Il thread genitore crea i thread di lavoro, passando a ognuno la posizione che deve controllare nella griglia del Sudoku. Questo passo richiede il passaggio di diversi parametri a ogni thread. L'approccio più semplice è quello di creare una struttura dati utilizzando una `struct`. Per esempio, una struttura per passare riga e colonna da cui un thread deve iniziare la convalida potrebbe essere la seguente:

```
/* struttura per il passaggio di dati ai thread */
typedef struct
{
    int row;
    int column;
} parameters;
```

Sia i programmi Pthreads sia quelli Windows creeranno thread di lavoro utilizzando una strategia simile a quella mostrata di seguito:

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;
/* Ora create il thread passandogli data come parametro */
```

Il puntatore `data` sarà passato alla funzione `pthread_create()` (Pthreads) o alla funzione `CreateThread()` (Windows), che a loro volta lo passeranno come parametro alla funzione che deve essere eseguita in un thread separato.

Restituzione dei risultati al thread genitore

A ogni thread di lavoro viene assegnato il compito di determinare la validità di una particolare regione del Sudoku. Una volta che uno di questi thread ha eseguito il controllo, esso deve restituire i risultati al genitore. Un buon modo per gestire tale situazione è quello di creare un array di valori interi visibile a ogni thread. L'indice *i-esimo* di questo array corrisponde al thread di lavoro *i-esimo*. Un thread imposta il corrispondente valore a 1 per indicare che la regione del Sudoku da lui controllata sia valida; il valore 0 indica invece il contrario. Quando tutti i thread di lavoro hanno completato l'esecuzione, il thread genitore controlla ogni voce dell'array risultato per determinare se il Sudoku sia valido.

Progetto 2 – Programma di ordinamento multithread

Scrivete un programma di ordinamento multithread che funziona come segue. Un vettore di numeri interi viene suddiviso in due vettori più piccoli di dimensioni uguali. Due thread separati (che chiameremo thread di ordinamento) ordinano ogni sottovettore utilizzando un algoritmo di ordinamento di vostra scelta. I due sottovettori sono

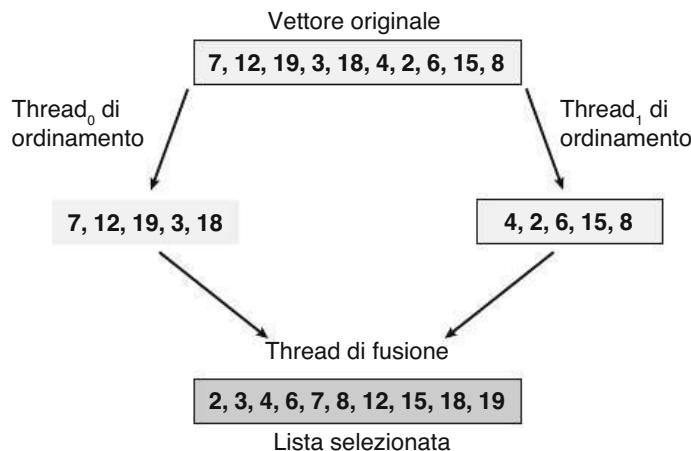


Figura 4.20 Ordinamento multithread.

poi uniti da un terzo thread (il **thread di fusione**) che forma così un unico vettore ordinato.

Il modo più semplice per gestire i dati è forse quello di creare un array globale, poiché i dati globali sono condivisi tra tutti i thread. Ogni thread di ordinamento lavorerà su una metà di questo array. Verrà inoltre creato un secondo array globale di pari dimensione, utilizzato dal thread di fusione per inserire l'unione dei due sottovettori. Graficamente, questo programma è strutturato come mostrato nella Figura 4.20.

Questo progetto di programmazione richiederà il passaggio di parametri a ciascuno dei thread di ordinamento. In particolare, sarà necessario individuare l'indice da cui ciascun thread deve iniziare l'ordinamento.

Fare riferimento alle istruzioni contenute nel Progetto 1 per i dettagli sul passaggio di parametri a un thread.

Il thread genitore darà in output l'array ordinato una volta che ogni thread di lavoro abbia terminato la sua esecuzione.

Note bibliografiche

Dopo l'esordio come “concorrenza a basso costo” nei linguaggi di programmazione, i thread hanno avuto una lunga evoluzione, passando ai “processi leggeri”, con i primi esempi che comprendono il sistema Thoth (Cheriton et al. [1979] e il sistema Pilot (Redell et al. [1980]). Binding [1985] ha descritto come i thread siano stati incorporati all'interno del kernel UNIX. Mach (Accetta et al. [1986], Tevanian et al. [1987a] e V (Cheriton [1988]) ha fatto ampio uso dei thread. Alla fine di questo percorso quasi tutti i principali sistemi operativi hanno implementato i thread in una forma o nell'altra.

[Vahalia 1996] tratta l'uso dei thread in diverse versioni di UNIX. [Mauro e McDougall 2007] descrivono gli sviluppi relativi all'uso dei thread nel kernel di Solaris. [Russinovich e Solomon 2009] descrivono la realizzazione dei thread nei sistemi

operativi Windows. [Mauerer 2008] e [Love 2010] spiegano come Linux gestisce il threading, [Singh 2007] tratta i thread in Mac OS X.

Informazioni sulla programmazione Pthreads si trovano in [Lewis e Berg 1998], oltre che in [Butenhof 1997]. [Oaks e Wong 1999], [Lewis e Berg 2000] e [Holub 1998] analizzano la programmazione multithread nel linguaggio Java. [Goetz et al. 2006] presentano una trattazione dettagliata della programmazione concorrente in Java. [Hart 2005] descrive il multithreading con l'uso di Windows. I dettagli sull'uso di OpenMP sono disponibili all'indirizzo <http://openmp.org>.

Un'analisi sulla dimensione ottimale di un gruppo di thread può essere trovata in [Ling et al. 2000]. Le attivazioni dello scheduler sono state presentate prima in [Anderson et al. 1991] e successivamente in [Williams 2002], che ne discute in ambiente NetBSD. [Breshears 2009] e [Pacheco 2011] trattano la programmazione parallela in dettaglio. [Hill e Marty (2008)] studiano la legge di Amdahl relativamente ai sistemi multicore. La tecnica di Monte Carlo per la stima di π è discussa in:

<http://math.fullerton.edu/mathews/n2003/montecarlopimod.html>.

Bibliografia

- [Accetta et al. 1986] M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian e M. Young, “Mach: A New Kernel Foundation for UNIX Development”, *Proceedings of the Summer USENIX Conference*, p. 93–112, 1986.
- [Anderson et al. 1991] T. E. Anderson, B. N. Bershad, E. D. Lazowska e H. M. Levy, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, *Proceedings of the ACM Symposium on Operating Systems Principles*, p. 95–109, 1991.
- [Binding 1985] C. Binding, “Cheap Concurrency in C”, *SIGPLAN Notices*, Volume 20, N. 9, p. 21–27, 1985
- [Breshears 2009] C. Breshears, *The Art of Concurrency*, O'Reilly & Associates, 2009.
- [Butenhof 1997] D. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.
- [Cheriton 1988] D. Cheriton, “The V Distributed System”, *Communications of the ACM*, Volume 31, N. 3, p. 314–333, 1988.
- [Cheriton et al. 1979] D. R. Cheriton, M. A. Malcolm, L. S. Melen e G. R. Sager, “Thoth, a Portable Real-Time Operating System”, *Communications of the ACM*, Volume 22, N. 2, p. 105–115, 1979.
- [Goetz et al. 2006] B. Goetz, T. Peirls, J. Bloch, J. Bowbeer, D. Holmes e D. Lea, *Java Concurrency in Practice*, Addison-Wesley, 2006.
- [Hart 2005] J. M. Hart, *Windows System Programming*, 3° Ed., Addison-Wesley, 2005.
- [Hill e Marty 2008] M. Hill e M. Marty, “Amdahl's Law in the Multicore Era”, *IEEE Computer*, Volume 41, N. 7, p. 33–38, 2008.

- [**Lewis e Berg 1998**] B. Lewis e D. Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press, 1998.
- [**Lewis e Berg 2000**] B. Lewis e D. Berg, *Multithreaded Programming with Java Technology*, Sun Microsystems Press, 2000.
- [**Ling et al. 2000**] Y. Ling, T. Mullen e X. Lin, “Analysis of Optimal Thread Pool Size”, *Operating System Review*, Volume 34, N. 2, p. 42–55, 2000.
- [**Love 2010**] R. Love, *Linux Kernel Development*, 3° Ed., Developer’s Library, 2010.
- [**Mauerer 2008**] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons, 2008.
- [**McDougall e Mauro 2007**] R. McDougall e J. Mauro, *Solaris Internals*, 2° Ed., Prentice Hall, 2007.
- [**Oaks e Wong 1999**] S. Oaks e H. Wong, *Java Threads*, 2° Ed., O’Reilly & Associates, 1999.
- [**Pacheco 2011**] P. S. Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011.
- [**Redell et al. 1980**] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray e S. P. Purcell, “Pilot: An Operating System for a Personal Computer”, *Communications of the ACM*, Volume 23, N. 2, p. 81–92, 1980.
- [**Russinovich e Solomon 2009**] M. E. Russinovich e D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*, 5° Ed., Microsoft Press, 2009.
- [**Singh 2007**] A. Singh, *Mac OS X Internals: A Systems Approach*, Addison-Wesley, 2007.
- [**Tevanian et al. 1987**] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper e M. W. Young, “Mach Threads and the Unix Kernel: The Battle for Control”, *Proceedings of the Summer USENIX Conference*, 1987.
- [**Vahalia 1996**] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall, 1996.
- [**Williams 2002**] N. Williams, “An Implementation of Scheduler Activations on the NetBSD Operating System”, 2002 *USENIX Annual Technical Conference*, FREE-NIX Track, 2002.

CAPITOLO

5

OBIETTIVI DEL CAPITOLO

- Introduzione al problema della sezione critica, le cui soluzioni sono utilizzabili per assicurare la coerenza dei dati condivisi.
- Introduzione alle soluzioni hardware e software al problema della sezione critica.
- Descrizione di diversi problemi classici di sincronizzazione di processi.
- Analisi di diversi strumenti utilizzati per risolvere problemi di sincronizzazione dei processi.

Sincronizzazione dei processi

Un **processo cooperante** è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l'influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi (cioè, codice e dati) oppure condividere dati soltanto attraverso file o messaggi. Nel primo caso si fa uso dei thread, presentati nel Capitolo 4. L'accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati. In questo capitolo si trattano vari meccanismi atti ad assicurare un'ordinata esecuzione dei processi cooperanti, che condividono uno spazio logico di indirizzi, così da mantenere la coerenza dei dati.

5.1 Introduzione

Abbiamo già visto che i processi possono essere eseguiti in modo concorrente o in parallelo. Il Paragrafo 3.2.2 ha introdotto il ruolo dello scheduling dei processi e ha descritto come lo scheduler della CPU passi rapidamente da un processo all’altro per offrire un’esecuzione concorrente. Questo significa che un processo può avere completato solo in parte la sua esecuzione quando viene schedulato un altro processo. In effetti, un processo può essere interrotto in qualsiasi punto del proprio flusso d’esecuzione, assegnando il core di elaborazione all’esecuzione di istruzioni di un altro processo. Il Paragrafo 4.2 ha inoltre introdotto l’esecuzione parallela, in cui due flussi di istruzioni (che rappresentano processi differenti) vengono eseguiti contemporaneamente su core distinti. In questo capitolo viene spiegato come l’esecuzione concorrente o parallela possa contribuire a problematiche che riguardano l’integrità dei dati condivisi da più processi.

Prendiamo in considerazione un esempio di come ciò può accadere. Nel Capitolo 3 è stato descritto un modello di sistema costituito da un certo numero di processi sequenziali cooperanti o thread, tutti in esecuzione asincrona e con la possibilità di condividere dati. Tale modello è stato illustrato attraverso l’esempio del produttore/consumatore, che ben rappresenta molte situazioni che riguardano i sistemi operativi. Nel Paragrafo 3.4.1, in particolare, si è descritto come un buffer limitato sia utilizzabile per permettere ai processi la condivisione della memoria.

Torniamo al concetto di buffer limitato. Com’è stato sottolineato, la nostra soluzione consentiva la presenza contemporanea di non più di `BUFFER_SIZE - 1` elementi. Si supponga di voler modificare l’algoritmo per rimediare a questa carenza. Una possibilità consiste nell’aggiungere una variabile intera, `contatore`, inizializzata a 0, che si incrementa ogniqualvolta s’inserisce un nuovo elemento nel buffer e si decrementa ogniqualvolta si preleva un elemento dal buffer. Il codice per il processo produttore si può modificare come segue:

```
while (true) {
    /* produce un elemento in next_produced */
    while (contatore == BUFFER_SIZE)
        ; /* non fa niente */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    contatore++;
}
```

Il codice per il processo consumatore si può modificare come segue:

```
while (true) {
    while (contatore == 0)
        ; /* non fa niente */
```

```

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    contatore--;
    /* consuma un elemento in next_consumed */
}

```

Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non funzionare altrettanto correttamente se si eseguono in modo concorrente. Si supponga per esempio che il valore della variabile contatore sia attualmente 5, e che i processi produttore e consumatore eseguano le istruzioni `contatore++` e `contatore--` in modo concorrente. Terminata l'esecuzione delle due istruzioni, il valore della variabile contatore potrebbe essere 4, 5 o 6! Il solo risultato corretto è `contatore == 5`, che si ottiene se si eseguono separatamente il produttore e il consumatore.

Si può dimostrare che il valore di contatore può essere scorretto: l'istruzione `contatore++` si può codificare in un tipico linguaggio macchina, come

```

registro1 := contatore
registro1 := registro1 + 1
contatore := registro1

```

dove `registro1` è un registro locale della CPU. Analogamente, l'istruzione `contatore--` si può codificare come

```

registro2 := contatore
registro2 := registro2 - 1
contatore := registro2

```

dove `registro2` è un registro locale della CPU. Anche se `registro1` e `registro2` possono essere lo stesso registro fisico, per esempio un accumulatore, occorre ricordare che il contenuto di questo registro viene salvato e recuperato dal gestore dei segnali d'interruzione (Paragrafo 1.2.3).

L'esecuzione concorrente delle istruzioni `contatore++` e `contatore--` equivale a un'esecuzione sequenziale delle istruzioni del linguaggio macchina introdotte precedentemente, intercalate (*interleaved*) in una qualunque sequenza che però conservi l'ordine interno di ogni singola istruzione di alto livello. Una di queste sequenze è

T_0 :	<i>produttore</i>	esegue	$registro_1 := \text{contatore}$	{ $registro_1 = 5$ }
T_1 :	<i>produttore</i>	esegue	$registro_1 := registro_1 + 1$	{ $registro_1 = 6$ }
T_2 :	<i>consumatore</i>	esegue	$registro_2 := \text{contatore}$	{ $registro_2 = 5$ }
T_3 :	<i>consumatore</i>	esegue	$registro_2 := registro_2 - 1$	{ $registro_2 = 4$ }
T_4 :	<i>produttore</i>	esegue	$\text{contatore} := registro_1$	{ $\text{contatore} = 6$ }
T_5 :	<i>consumatore</i>	esegue	$\text{contatore} := registro_2$	{ $\text{contatore} = 4$ }

e conduce al risultato errato in cui `contatore == 4`; si registra la presenza di 4 elementi nel buffer, mentre in realtà gli elementi sono 5. Se si invertisse l'ordine delle istruzioni in T_4 e T_5 si giungerebbe allo stato errato in cui `contatore == 6`.

Si è arrivati a questo stato non corretto perché si è permesso a entrambi i processi di manipolare concorrentemente la variabile `contatore`. Per evitare le situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (le cosiddette **race condition**) occorre assicurare che un solo processo alla volta possa modificare la variabile `contatore`. Questa garanzia richiede una forma di sincronizzazione dei processi.

Tali situazioni si verificano spesso nei sistemi operativi, nei quali diversi componenti del sistema compiono operazioni su risorse condivise. Inoltre, come evidenziato nei precedenti capitoli, la crescente importanza dei sistemi multicore ha dato maggior enfasi allo sviluppo di applicazioni multithread in cui diversi thread, che probabilmente possono condividere dei dati, sono in esecuzione in parallelo su core distinte. Ovviamente tali operazioni non devono interferire reciprocamente in modi indesiderati. Data l'importanza della questione, la maggior parte di questo capitolo è dedicata ai problemi della **sincronizzazione** e **coordinamento dei processi**.

5.2 Problema della sezione critica

Iniziamo la nostra discussione sui processi di sincronizzazione illustrando il problema della sezione critica. Si consideri un sistema composto di n processi $\{P_0, P_1, \dots, P_{n-1}\}$ ciascuno avente un segmento di codice, chiamato **sezione critica** (detto anche *regione critica*), in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via. La caratteristica fondamentale del sistema è che, quando un processo è in esecuzione nella propria sezione critica, non si consente a nessun altro processo di essere in esecuzione nella propria sezione critica. Il problema della *sezione critica* consiste nel progettare un protocollo che i processi possono usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione di codice che realizza questa richiesta è la **sezione d'ingresso**. La sezione critica può essere seguita da una **sezione d'uscita**, e la restante parte del codice è detta **sezione non critica**. La Figura 5.1 mostra la struttura generale di un tipico processo P_i . La sezione d'ingresso e quella d'uscita sono state inserite nei riquadri per evidenziare questi importanti segmenti di codice.

Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti.

1. **Mutua esclusione.** Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. **Progresso.** Se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori delle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.

```

do{

    sezione d'ingresso

    sezione critica

    sezione d'uscita

    sezione non critica

} while (true);

```

Figura 5.1 Struttura generale di un tipico processo P_i .

3. **Attesa limitata.** Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

Si suppone che ogni processo sia eseguito a una velocità diversa da zero. Tuttavia, non si può fare alcuna ipotesi sulla velocità relativa degli n processi.

In un dato momento, numerosi processi in modalità kernel possono essere attivi nel sistema operativo. Se ciò si verifica, il **codice del kernel**, che implementa il sistema operativo, è soggetto a molte possibili race condition. Si consideri per esempio una struttura dati del kernel che mantenga una lista di tutti i file aperti nel sistema. Tale lista deve essere modificata quando un nuovo file è aperto, e quindi aggiunto all'elenco, oppure chiuso, e quindi tolto dall'elenco. Se due o più processi dovessero aprire dei file simultaneamente, potrebbero ingenerare nel sistema una race condition legata ai necessari aggiornamenti della lista. Altre strutture dati del kernel soggette a problemi analoghi sono quelle per l'allocazione della memoria, per la gestione delle interruzioni e le liste dei processi. La responsabilità di preservare il sistema operativo da simili problemi compete a chi sviluppa il kernel.

La due strategie principali per la gestione delle sezioni critiche nei sistemi operativi sono: **kernel con diritto di prelazione** e **kernel senza diritto di prelazione**. Un kernel con diritto di prelazione consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. Un kernel senza diritto di prelazione non consente di applicare la prelazione a un processo attivo in modalità di sistema: l'esecuzione di questo processo seguirà finché lo stesso esca da tale modalità, si blocchi o ceda volontariamente il controllo della CPU. Ovviamente, i kernel senza diritto di prelazione sono immuni da race condition sulle strutture dati del kernel, visto che un solo processo per volta impegna il kernel. Altrettanto non si può dire dei kernel con diritto di prelazione, motivo per cui bisogna avere cura, nella progettazione, di mantenerli al riparo dai problemi nell'accesso alle strutture dati del kernel. I kernel con diritto di prelazione presentano particolari difficoltà di progettazione quando sono destinati ad architetture SMP, poiché in tali ambienti due pro-

cessi nella modalità di sistema possono essere eseguiti in contemporanea su processori differenti.

Perché, allora, i kernel con diritto di prelazione dovrebbero essere preferiti a quelli senza diritto di prelazione? I kernel con diritto di prelazione possono vantare una maggior prontezza nelle risposte, grazie al basso rischio di eseguire i processi in modalità di sistema per un tempo eccessivamente lungo, prima di liberare la CPU per i processi in attesa (certamente il rischio può essere minimizzato anche progettando codice kernel che non si comporta in questo modo). Inoltre, i kernel con diritto di prelazione sono più adatti alla programmazione real-time, dal momento che permettono ai processi in tempo reale di effettuare la prelazione di un processo attivo nel kernel. Vedremo più avanti come diversi sistemi operativi usino la prelazione all'interno del kernel.

5.3 Soluzione di Peterson

Illustriamo adesso una classica soluzione software al problema della sezione critica, nota come **soluzione di Peterson**. A causa del modo in cui i moderni elaboratori eseguono le istruzioni elementari del linguaggio macchina, quali `load` e `store`, non è affatto certo che la soluzione di Peterson funzioni correttamente su tali sistemi. Tuttavia si è scelto di presentarla ugualmente perché rappresenta un buon algoritmo per il problema della sezione critica che illustra alcune complessità legate alla progettazione di programmi che soddisfino i tre requisiti di mutua esclusione, progresso e attesa limitata.

La soluzione di Peterson è limitata a due processi, P_0 e P_1 , ognuno dei quali esegue alternativamente la propria sezione critica e la sezione non critica. Per il seguito, è utile convenire che se P_i denota uno dei due processi, P_j denoti l'altro; ossia, che $j = 1 - i$.

La soluzione di Peterson richiede che i processi condividano i seguenti dati:

```
int turn;
boolean flag[2];
```

La variabile `turn` segnala, per l'appunto, di chi sia il turno d'accesso alla sezione critica; quindi, se `turn == i`, il processo P_i è autorizzato a eseguire la propria sezione critica. L'array `flag`, invece, indica se un processo *sia pronto* a entrare nella sezione critica. Per esempio, se `flag[i]` è `true`, P_i è pronto a entrare nella propria sezione critica. Chiarito il ruolo delle strutture dati, possiamo ora analizzare l'algoritmo descritto nella Figura 5.2.

Per accedere alla sezione critica, il processo P_i assegna innanzitutto a `flag[i]` il valore `true`; quindi attribuisce a `turn` il valore j , conferendo così all'altro processo la facoltà di entrare nella sezione critica.

Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a turno sia il valore i sia il valore j . Soltanto uno dei due permane: l'altro sarà immediatamente sovrascritto. Il valore definitivo di `turn` stabilisce quale dei due processi sia autorizzato a entrare per primo nella propria sezione critica.

```

do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

    sezione critica

    flag[i] = false;

    sezione non critica

} while (true);

```

Figura 5.2 Struttura del processo P_i nella soluzione di Peterson.

Dimostriamo ora la correttezza di questa soluzione. Dobbiamo provare che:

1. la mutua esclusione è preservata;
2. il requisito del progresso è soddisfatto;
3. il requisito dell'attesa limitata è rispettato.

Per dimostrare la proprietà 1, si osservi come ogni P_i acceda alla propria sezione critica solo se $\text{flag}[j] == \text{false}$ oppure $\text{turn} == i$. Si noti anche che, se entrambi i processi sono eseguibili in concomitanza nelle rispettive sezioni critiche, allora $\text{flag}[0] == \text{flag}[1] == \text{true}$. Si desume da queste due osservazioni che P_0 e P_1 sono impossibilitati a eseguire con successo le rispettive istruzioni `while` approssimativamente nello stesso momento: `turn`, infatti, può valere 0 o 1, ma non entrambi. Pertanto, uno dei processi – poniamo P_j – deve aver eseguito con successo l’istruzione `while`, mentre P_i aveva da eseguire almeno un’istruzione aggiuntiva (“`turn == j`”). Tuttavia, poiché in quel momento, e fino al termine della permanenza di P_j nella propria sezione critica, restano valide le asserzioni `flag[j] == true` e `turn == j`, ne consegue che la mutua esclusione è preservata.

Per dimostrare le proprietà 2 e 3, osserviamo come l’ingresso di un processo P_i nella propria sezione critica possa essere impedito solo se il processo è bloccato nella sua iterazione `while`, con le condizioni `flag[j] == true` e `turn == j`; questa è l’unica possibilità. Qualora P_j non sia pronto a entrare nella sezione critica, `flag[j] == false`, e P_i può accedere alla propria sezione critica. Se P_j ha impostato `flag[j]` a `true` e sta eseguendo il proprio ciclo `while`, `turn == i`, oppure `turn == j`. Se `turn == i`, P_i entrerà nella propria sezione critica. Se `turn == j`, P_j entrerà nella propria sezione critica. Tuttavia, al momento di uscire dalla propria sezione critica, P_j reimposta `flag[j]` a `false`, consentendo a P_i di entrarvi. Se P_j reimposta `flag[j]` a `true`, deve anche attribuire alla variabile `turn` il valore `i`. Poiché tuttavia P_i non modifica il valore della variabile `turn` durante l’esecuzione dell’istruzione `while`, P_i entrerà nella sezione critica (progresso) dopo che P_j abbia effettuato non più di un ingresso (attesa limitata).

5.4 Hardware per la sincronizzazione

Abbiamo appena descritto una soluzione software al problema della sezione critica. Tuttavia, come già detto, soluzioni basate sul software come quella di Peterson non garantiscono il loro funzionamento su architetture moderne.

Nel seguito esploriamo diverse altre soluzioni al problema della sezione critica che utilizzano tecniche che vanno da quelle hardware alle API software disponibili sia per gli sviluppatori del kernel sia per i programmati di applicazioni. Tutte queste soluzioni si basano sul concetto di **lock** (lucchetto), ovvero sulla protezione di regioni critiche attraverso l'uso di lock. Come vedremo, la progettazione di un lock può essere piuttosto complessa.

Iniziamo presentando alcune semplici istruzioni hardware disponibili in molti sistemi e mostrando come queste possano essere efficacemente utilizzate per risolvere il problema della sezione critica. Le funzionalità hardware possono rendere più facile il compito del programmatore e migliorare l'efficienza del sistema.

In un sistema dotato di una singola CPU il problema della sezione critica si potrebbe risolvere semplicemente se si potessero disabilitare le interruzioni mentre si modificano le variabili condivise. In questo modo si assicurerebbe un'esecuzione nell'ordine e senza possibilità di prelazione della corrente sequenza di istruzioni; non si potrebbe eseguire nessun'altra istruzione, quindi non si potrebbe apportare alcuna modifica inaspettata alle variabili condivise. È questo l'approccio seguito dai kernel senza diritto di prelazione.

Sfortunatamente questa soluzione non è altrettanto praticabile in un sistema multiprocessore; la disabilitazione delle interruzioni nei sistemi multiprocessore può comportare sprechi di tempo dovuti alla necessità di trasmettere un messaggio di disabilitazione a tutte le unità d'elaborazione. Tale trasmissione ritarda l'accesso a ogni sezione critica determinando una diminuzione dell'efficienza. Si considerino, inoltre, gli effetti su un clock di sistema se questo viene aggiornato tramite interruzioni.

Per questo motivo molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, in modo **atomico** – cioè come un'unità non interrompibile. Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice. Anziché discutere una specifica istruzione di una particolare architettura, è preferibile astrarre i concetti principali descrivendo le istruzioni `test_and_set()` e `compare_and_swap()`.

L'istruzione `test_and_set()` si può definire come nella Figura 5.3. La caratteristica fondamentale di questa istruzione è che viene eseguita **atomicamente**; quindi,

```
boolean test_and_set(boolean *obiettivo) {
    boolean valore = *obiettivo;
    *obiettivo = true;
    return valore;
}
```

Figura 5.3 Definizione dell'istruzione `test_and_set()`.

```

do {
    while (test_and_set(&lock));
        sezione critica
    lock = false;
        sezione non critica
} while (true);

```

Figura 5.4 Realizzazione di mutua esclusione con `test_and_set()`.

se si eseguono contemporaneamente due istruzioni `test_and_set()`, ciascuna in un'unità d'elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario. Se si dispone dell'istruzione `test_and_set()`, si può realizzare la mutua esclusione dichiarando una variabile booleana globale `lock`, inizializzata a `false`. La struttura del processo P_i è illustrata nella Figura 5.4.

L'istruzione `compare_and_swap()`, definita nella Figura 5.5, a differenza dell'istruzione `test_and_set()` utilizza tre operandi. L'operando `value` viene impostato a `new_value` solo se l'espressione `(*value == expected)` è vera. A parte in questo caso, la `compare_and_swap()` restituisce sempre il valore originale della variabile `value`. Come l'istruzione di `test_and_set()`, la `compare_and_swap()` viene eseguita atomicamente. La mutua esclusione può essere realizzata come segue. Viene dichiarata e inizializzata a 0 una variabile globale (`lock`). Il primo processo che richiama `compare_and_swap()` imposterà `lock` a 1. Entrerà poi nella sua sezione critica, poiché il valore originale di `lock` era pari al valore atteso 0. Le chiamate successive di `compare_and_swap()` non avranno successo, perché ora `lock` non è uguale al valore atteso 0. Quando un processo esce dalla sezione critica, imposta di nuovo `lock` al valore 0, per permettere a un altro processo di entrare nella propria sezione critica. La struttura del processo P_i è illustrata nella Figura 5.6.

```

int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}

```

Figura 5.5 Definizione dell'istruzione `compare_and_swap()`.

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* non fa niente */

    /* sezione critica */

    lock = 0;

    /* sezione non critica */
} while (true);
```

Figura 5.6 Realizzazione di mutua esclusione con `compare_and_swap()`.

Questi algoritmi soddisfano il requisito della mutua esclusione, ma non quello dell'attesa limitata. La Figura 5.7 mostra un altro algoritmo che sfrutta l'istruzione `test_and_set()` che soddisfare tutti e tre i requisiti desiderati. Le strutture dati condivise sono

```
boolean waiting[n];
boolean lock;
```

e sono inizializzate al valore `false`. Per dimostrare che l'algoritmo soddisfa il requisito di mutua esclusione, si noti che il processo P_i può entrare nella propria sezione

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* sezione critica */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* sezione non critica */
} while (true);
```

Figura 5.7 Mutua esclusione con attesa limitata con `test_and_set()`.

critica solo se `waiting[i] == false` oppure `chiave == false`. Il valore di `chiave` può diventare `false` solo se si esegue `test_and_set()`. Il primo processo che esegue `test_and_set()` trova `chiave == false`; tutti gli altri devono attendere. La variabile `waiting[i]` può diventare `false` solo se un altro processo esce dalla propria sezione critica; solo una variabile `waiting[i]` vale `false`, il che consente di rispettare il requisito di mutua esclusione.

Per dimostrare che l'algoritmo soddisfa il requisito di progresso, basta osservare che le argomentazioni fatte per la mutua esclusione valgono anche in questo caso; infatti un processo che esce dalla sezione critica imposta `lock` al valore `false` oppure `waiting[j]` al valore `false`; entrambe consentono a un processo in attesa l'ingresso nella propria sezione critica.

Per dimostrare che l'algoritmo soddisfa il requisito di attesa limitata occorre osservare che un processo, quando lascia la propria sezione critica, scandisce il vettore `waiting` in ordine ciclico ($i + 1, i + 2, \dots, n - 1, 0, \dots, i - 1$) e designa il primo processo in questo ordinamento presente nella sezione d'ingresso (`waiting[j] == true`) come il primo processo che deve entrare nella propria sezione critica. Qualsiasi processo che attende l'ingresso nella propria sezione critica può farlo entro $n - 1$ turni.

Una descrizione dettagliata dell'implementazione delle istruzioni atomiche `test_and_set()` e `compare_and_swap()` è reperibile nei testi di architettura dei calcolatori.

5.5 Lock mutex

Le soluzioni hardware al problema della sezione critica presentate nel Paragrafo 5.4 sono complicate e generalmente inaccessibili ai programmati di applicazioni. In alternativa, i progettisti di sistemi operativi implementano strumenti software per risolvere lo stesso problema. Il più semplice di questi strumenti è il **lock mutex** (il termine *mutex* è in realtà l'abbreviazione di *mutual exclusion*, cioè mutua esclusione). Usiamo il lock mutex per proteggere le regioni critiche e quindi prevenire le race condition. In pratica un processo deve acquisire il lock prima di entrare in una sezione critica e rilasciarlo quando esce dalla sezione critica. La funzione `acquire()` acquisisce il lock e la funzione `release()` lo rilascia, come illustrato nella Figura 5.8.

```
do {
    acquisisci lock
        sezione critica
    rilascia lock
        sezione non critica
} while (true);
```

Figura 5.8 Soluzione al problema della sezione critica con lock mutex.

Un lock mutex ha una variabile booleana `available` il cui valore indica se il lock è disponibile o meno. Se il lock è disponibile la chiamata di `acquire()` ha successo e il lock viene da questo momento considerato non disponibile. Un processo che tenta di acquisire un lock indisponibile viene bloccato fino al rilascio del lock.

La definizione della funzione `acquire()` è la seguente:

```
acquire() {
    while (!available)
        ; /* attesa attiva */
    available = false;
}
```

La definizione di `release()` è la seguente:

```
release() {
    available = true;
}
```

Le chiamate alle funzioni `acquire()` e `release()` devono essere eseguite atomicamente. Per questa ragione i lock mutex sono spesso realizzati utilizzando uno dei meccanismi hardware descritti nel Paragrafo 5.4. Lasciamo la descrizione di questa tecnica come esercizio.

Il principale svantaggio dell’implementazione che abbiamo fornito è che richiede **attesa attiva** (*busy waiting*). Mentre un processo si trova nella sua sezione critica, ogni altro processo che cerca di entrare nella sezione critica deve ciclare continuamente effettuando la chiamata `acquire()`. Questo tipo di lock mutex è anche chiamato **spinlock**, perché il processo continua a “girare” (spin), in attesa che il lock diventi disponibile. (Osserviamo lo stesso problema negli esempi di codice che illustrano le istruzioni di `test_and_set()` e `compare_and_swap()`). Questo continuo ciclare è chiaramente un problema in un sistema multiprogrammato, dove una singola CPU è condivisa tra diversi processi. L’attesa attiva spreca cicli di CPU che qualche altro processo potrebbe utilizzare in modo produttivo.

Tuttavia, gli spinlock hanno il vantaggio di non rendere necessario alcun cambio di contesto (operazione che può richiedere molto tempo) quando un processo deve attendere un lock e tornano quindi utili quando si prevede che i lock verranno trattati per tempi brevi. Gli spinlock sono spesso impiegati in sistemi multiprocessore in cui un thread può “girare” su un processore, mentre un altro thread esegue la sua sezione critica su un altro processore.

Più avanti in questo capitolo (Paragrafo 5.7), esamineremo come i lock mutex possono essere utilizzati per risolvere classici problemi di sincronizzazione. Discuteremo anche di come questi lock siano usati in diversi sistemi operativi e in Pthreads.

5.6 Semafori

Come abbiamo accennato in precedenza, i lock mutex sono generalmente considerati il più semplice degli strumenti di sincronizzazione. In questo paragrafo esaminiamo uno strumento più robusto in grado di comportarsi in modo simile a un lock mutex, ma capace anche di fornire metodi più complessi per la sincronizzazione delle attività dei processi.

Un **semaforo** *s* è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: `wait()` e `signal()`. Queste operazioni erano originariamente chiamate *P* (per `wait()`; dall'olandese *proberen*, verificare) e *V* (per `signal()`; da *verhogen*, incrementare). La definizione di `wait()` in pseudocodice è la seguente:

```
wait(s) {
    while(s <= 0)
        ;//attesa attiva
    s--;
}
```

La definizione di `signal()` in pseudocodice è la seguente:

```
signal(s) {
    s++;
}
```

Tutte le modifiche al valore del semaforo contenute nelle operazioni `wait()` e `signal()` si devono eseguire in modo indivisibile: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Inoltre, nel caso della `wait(s)` si devono eseguire senza interruzione anche la verifica del valore intero di *s* (*s* \leq 0) e la sua possibile modifica (*s*--). Nel Paragrafo 5.6.2 si spiega come si possono realizzare queste operazioni. Ora vediamo come si possono utilizzare i semafori.

5.6.1 Uso dei semafori

Si usa distinguere tra **semafori contatore**, il cui valore è un numero intero, e i **semafori binari**, il cui valore è limitato a 0 o 1. I semafori binari sono dunque simili ai lock mutex e vengono utilizzati al loro posto per la mutua esclusione nei sistemi dove i lock mutex non sono disponibili.

I semafori contatore trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di risorse disponibili. I processi che desiderino utilizzare una risorsa invocano `wait()` sul semaforo, decrementandone così il valore; i processi che restituiscono una risorsa, invece, invocano `signal()` sul semaforo, incrementandone il valore. Quando il semaforo vale 0, tutte le risorse sono occupate, e i processi che ne richiedano l'uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

I semafori sono utilizzabili anche per risolvere diversi problemi di sincronizzazione. Si considerino, per esempio, due processi in esecuzione concorrente: P_1 con un’istruzione S_1 e P_2 con un’istruzione S_2 . Si supponga di voler eseguire S_2 solo dopo che S_1 è terminata. Questo schema si può prontamente realizzare facendo condividere a P_1 e P_2 un semaforo comune, `synch`, inizializzato a 0, e inserendo nel processo P_1 le istruzioni

```
S1;  
signal(synch);
```

e nel processo P_2 le istruzioni

```
wait(synch);  
S2;
```

Poiché `synch` è inizializzato a 0, P_2 esegue S_2 solo dopo che P_1 ha eseguito `signal(synch)`, che si trova dopo S_1 .

5.6.2 Implementazione dei semafori

Ricordiamo che la realizzazione dei lock mutex trattati nel Paragrafo 5.5 presentava il problema dell’attesa attiva. Le definizioni delle operazioni sui semafori `wait()` e `signal()` appena descritte presentano lo stesso problema.

Per superare la necessità dell’attesa attiva si possono modificare le definizioni delle operazioni `wait()` e `signal()` come segue: quando un processo invoca l’operazione `wait()` e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell’attesa attiva può *bloccare* se stesso. L’operazione di *bloccaggio* pone il processo in una coda d’attesa associata al semaforo e pone lo stato del processo a *waiting*. Quindi, si trasferisce il controllo allo scheduler della CPU che sceglie un altro processo pronto per l’esecuzione.

Un processo bloccato, che attende a un semaforo `s`, sarà riavviato in seguito all’esecuzione di un’operazione `signal()` su `s` da parte di qualche altro processo. Il processo si riavvia tramite un’operazione `wakeup()`, che modifica lo stato del processo da *waiting* a *ready*. Il processo entra nella coda dei processi pronti. (L’uso della CPU può essere o non essere commutato dal processo in esecuzione al processo appena divenuto pronto, a seconda del criterio di scheduling).

Per realizzare i semafori secondo quel che s’è detto si può definire il semaforo come segue:

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

A ogni semaforo sono associati un valore intero (`value`) e una lista di processi (`list`), contenente i processi in attesa a un semaforo; l’operazione `signal()` preleva un processo da tale lista e lo attiva.

L'operazione `wait()` del semaforo si può definire come segue:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        aggiungi questo processo a S->list;
        block();
    }
}
```

L'operazione `signal()` del semaforo si può definire come segue:

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        togli un processo P da S->list;
        wakeup(P);
    }
}
```

L'operazione `block()` sospende il processo che la invoca; l'operazione `wakeup(P)` pone in stato di pronto per l'esecuzione un processo P bloccato. Queste due operazioni sono fornite dal sistema operativo come chiamate di sistema.

Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, questa definizione può condurre a valori negativi. Se il valore del semaforo è negativo, il suo valore assoluto rappresenta il numero dei processi che attendono a quel semaforo. Ciò è dovuto all'inversione dell'ordine del decremento e della verifica nel codice dell'operazione `wait()`.

La lista dei processi che attendono a un semaforo si può facilmente realizzare con un campo puntatore in ciascun blocco di controllo del processo (PCB). Ogni semaforo contiene un valore intero e un puntatore a una lista di PCB. Un modo per aggiungere e togliere processi dalla lista assicurando un'attesa limitata è usare una coda FIFO, della quale il semaforo contiene i puntatori al primo e all'ultimo elemento. In generale tuttavia si può usare *qualsiasi* criterio d'accodamento; il corretto uso dei semafori non dipende dal particolare criterio adottato.

Le operazioni sui semafori devono essere eseguite in modo atomico. Si deve garantire che nessuna coppia di processi possa eseguire operazioni `wait()` e `signal()` contemporaneamente sullo stesso semaforo. Si tratta di un problema di accesso alla sezione critica, e in un contesto monoprocesso lo si può risolvere semplicemente inibendo le interruzioni durante l'esecuzione di `signal()` e `wait()`. Nei sistemi con una sola CPU, infatti, le interruzioni sono i soli elementi di disturbo: non vi sono istruzioni eseguite da altri processori. Finché non si riattivino le interru-

zioni, dando la possibilità allo scheduler di riprendere il controllo della CPU, il processo corrente continua indisturbato la sua esecuzione.

Nei sistemi multiprocessore sarebbe necessario disabilitare le interruzioni di tutti i processori, perché altrimenti le istruzioni dei diversi processi in esecuzione su processori distinti potrebbero interferire fra loro. Tuttavia, disabilitare le interruzioni di tutti i processori può essere complesso, e causare un notevole calo delle prestazioni. È per questo che – per garantire l'esecuzione atomica di `wait()` e `signal()` – i sistemi SMP devono mettere a disposizione altre tecniche di realizzazione dei lock (per esempio, `compare_and_swap()` e gli spinlock).

È importante rilevare che questa definizione delle operazioni `wait()` e `signal()` non consente di eliminare completamente l'attesa attiva, ma piuttosto di rimuoverla dalle sezioni d'ingresso dei programmi applicativi. Inoltre, l'attesa attiva si limita alle sezioni critiche delle operazioni `wait()` e `signal()`, che sono abbastanza brevi; se sono convenientemente codificate, non sono più lunghe di 10 istruzioni. Quindi, la sezione critica non è quasi mai occupata e l'attesa attiva si presenta raramente e per breve tempo. Una situazione completamente diversa si verifica con i programmi applicativi le cui sezioni critiche possono essere lunghe minuti o anche ore, oppure occupate spesso. In questi casi l'attesa attiva è assai inefficiente.

5.6.3 Stallo e attesa indefinita

La realizzazione di un semaforo con coda d'attesa può condurre a situazioni in cui due o più processi attendono indefinitamente un evento – l'esecuzione di un'operazione `signal()` – che può essere generato solo da uno dei processi in attesa. Quando si verifica una situazione di questo tipo si dice che i processi sono in **stallo** (*deadlocked*).

Per illustrare questo fenomeno si considerino due processi, P_0 e P_1 , ciascuno dei quali ha accesso a due semafori, S e Q , impostati al valore 1:

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
⋮	⋮
⋮	⋮
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

Si supponga che P_0 esegua `wait(S)` e quindi P_1 esegua `wait(Q)`; eseguita `wait(Q)`, P_0 deve attendere che P_1 esegua `signal(Q)`; analogamente, quando P_1 esegue `wait(S)`, deve attendere che P_0 esegua `signal(S)`. Poiché queste operazioni `signal()` non si possono eseguire, P_0 e P_1 sono in stallo.

Un insieme di processi è in stallo se ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. In questo contesto si considerano principalmente gli *eventi di acquisizione e rilascio di risorse*, tuttavia an-



INVERSIONE DI PRIORITÀ E IL MARS PATHFINDER

L'inversione di priorità può essere più di un semplice inconveniente nello scheduling. Su sistemi con vincoli temporali molto restrittivi (come i sistemi real-time) l'inversione di priorità può far sì che un processo non venga eseguito nei tempi richiesti. Quando ciò succede possono innescarsi errori a cascata in grado di provocare un malfunzionamento del sistema.

Si consideri il caso del Mars Pathfinder, una sonda spaziale della NASA che nel 1997 portò un robot, il Sojourner rover, su Marte per condurre un esperimento. Poco dopo che il Sojourner ebbe iniziato il suo lavoro, cominciarono ad aver luogo numerosi reset del sistema. Ciascun reset inizializzava di nuovo sia hardware che software, inclusi gli strumenti preposti alla comunicazione. Se il problema non fosse stato risolto, il Sojourner avrebbe fallito la sua missione.

Il problema era causato dal fatto che un processo ad alta priorità, di nome "bc_dist", impiegava più tempo del dovuto a portare a termine il suo compito. Questo processo era forzatamente in attesa di una risorsa condivisa utilizzata da un processo a priorità inferiore, denominato "ASI/MET", a sua volta prelazionato da diversi processi di priorità media. Il processo "bc_dist" andava quindi in stallo in attesa della risorsa condivisa, e il processo "bc_sched", rilevando il problema, eseguiva il reset. Il Sojourner soffriva dunque di un tipico caso di inversione di priorità.

Il sistema operativo installato sul Sojourner era il sistema operativo real-time VxWorks, che disponeva di una variabile globale per abilitare l'ereditarietà delle priorità su tutti i semafori. Dopo alcuni test, il valore della variabile del Sojourner (su Marte!) fu impostato correttamente, e il problema fu risolto.

Un resoconto completo del problema, della sua scoperta, e della sua soluzione è stato scritto dal responsabile del team software ed è disponibile all'indirizzo

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html.

che altri tipi di eventi possono produrre situazioni di stallo (si veda il Capitolo 7, che descrive anche i meccanismi che servono ad affrontare questo tipo di problema).

Un'altra questione connessa alle situazioni di stallo è quella dell'**attesa indefinita** (nota anche col termine **starvation**). Con tale termine si definisce una situazione d'attesa indefinita nella coda di un semaforo, che si può per esempio presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio LIFO (last-in, first-out).

5.6.4 Inversione di priorità

Nello scheduling dei processi si possono incontrare difficoltà ognualvolta un processo a priorità più alta abbia bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa. Visto che i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse. La situazione si complica ulteriormente se il processo a priorità più bassa viene prelazionato da un processo a priorità più alta.

Assumiamo, per esempio, che vi siano tre processi L , M e H , le cui priorità seguono l'ordine $L < M < H$. Assumiamo che il processo H richieda la risorsa R alla quale

sta accedendo il processo L . Usualmente il processo H resterebbe in attesa che L liberi la risorsa R . Supponiamo però che M diventi eseguibile, con prelazione sul processo L . Avviene quindi, indirettamente, che un processo con priorità più bassa, il processo M , influenzi il tempo che H attenderà in attesa della risorsa R .

Questo problema è noto come **inversione della priorità**. Dato che l'inversione di priorità si verifica solo su sistemi con più di due priorità, una delle soluzioni è limitare a due il numero di priorità. Tuttavia, questa soluzione non è accettabile nella maggior parte dei sistemi a uso generale. Solitamente questi sistemi risolvono il problema implementando un **protocollo di ereditarietà delle priorità**, secondo il quale tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non finiscono di utilizzare le risorse in questione. Quando hanno terminato, la loro priorità ritorna al valore originale. Nell'esempio discusso in precedenza, un protocollo di ereditarietà delle priorità avrebbe permesso al processo L di ereditare temporaneamente la priorità di H , impedendo così al processo M di prelazionare la sua esecuzione. In un tale caso, una volta che il processo H avrà terminato con la risorsa R , rinuncerà alla priorità ereditata da H assumendo di nuovo la priorità originale. Poiché R sarà a questo punto disponibile, il processo H , e non il processo M , sarà il successivo processo eseguito.

5.7 Problemi tipici di sincronizzazione

In questo paragrafo s'illustrano diversi problemi di sincronizzazione come esempi di una vasta classe di problemi connessi al controllo della concorrenza. Questi problemi sono utili per verificare quasi tutte le nuove proposte di schemi di sincronizzazione. Nelle soluzioni che proponiamo ai problemi s'impiegano i semafori, perché per tradizione le soluzioni vengono così presentate. Le implementazioni reali possono tuttavia utilizzare i lock mutex al posto dei semafori binari.

5.7.1 Produttore/consumatore con memoria limitata

Il problema del *produttore/consumatore con memoria limitata*, introdotto nel Paragrafo 5.1, si usa generalmente per illustrare la potenza delle primitive di sincronizzazione. In questa sede si presenta uno schema generale di soluzione, senza far riferimento a nessuna realizzazione particolare. In conclusione del capitolo proponiamo al riguardo un progetto di programmazione.

Nel nostro problema produttore e consumatore condividono le seguenti strutture dati:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

```

do {
    . . .
    /* produce un elemento in next_produced */
    . . .
    wait(empty);
    wait(mutex);
    . . .
    /* inserisci next_produced in buffer */
    . . .
    signal(mutex);
    signal(full);
} while (true);

```

Figura 5.9 Struttura generale del processo produttore.

Si supponga di disporre di una certa quantità di memoria rappresentata da un buffer con n posizioni, ciascuna capace di contenere un elemento. Il semaforo `mutex` garantisce la mutua esclusione degli accessi al buffer ed è inizializzato al valore 1. I semafori `vuote` e `piene` conteggiano rispettivamente il numero di posizioni vuote e il numero di posizioni piene nel buffer. Il semaforo `vuote` si inizializza al valore n ; il semaforo `piene` si inizializza al valore 0.

La Figura 5.9 riporta la struttura del processo produttore, la Figura 5.10 quella del processo consumatore. È interessante notare la simmetria esistente tra il produttore e il consumatore. Il codice si può interpretare nel senso di produzione, da parte del produttore, di posizioni piene per il consumatore; oppure di produzione, da parte del consumatore, di posizioni vuote per il produttore.

```

do {
    wait(full);
    wait(mutex);
    . . .
    /* rimuovi un elemento da buffer e mettilo in next_consumed */
    . . .
    signal(mutex);
    signal(empty);
    . . .
    /* consuma l'elemento contenuto in next_consumed */
    . . .
} while (true);

```

Figura 5.10 Struttura generale del processo consumatore.

5.7.2 Problema dei lettori-scrittori

Si supponga che una base di dati sia da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto della base dati, mentre altri ne possono richiedere un aggiornamento, vale a dire una lettura e una scrittura. Questi due tipi di processi vengono distinti chiamando **lettori** quelli interessati alla sola lettura e **scrittori** gli altri. Naturalmente, se due lettori accedono nello stesso momento all’insieme di dati condiviso, non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro processo (lettore o scrittore) accedono contemporaneamente alla stessa base di dati, ne può derivare il caos.

Per impedire l’insorgere di difficoltà di questo tipo è necessario che gli scrittori abbiano un accesso esclusivo in fase di scrittura alla base di dati condivisa. Questo problema di sincronizzazione è conosciuto come **problema dei lettori-scrittori**. Da quando tale problema fu enunciato, è stato usato per verificare quasi tutte le nuove primitive di sincronizzazione. Il problema dei lettori-scrittori ha diverse varianti, che implicano tutte l’esistenza di priorità; la più semplice, cui si fa riferimento come al *primo* problema dei lettori-scrittori, richiede che nessun lettore attenda, a meno che uno scrittore abbia già ottenuto il permesso di usare l’insieme di dati condiviso. In altre parole, nessun lettore deve attendere che altri lettori terminino l’operazione solo perché uno scrittore attende l’accesso ai dati. Il *secondo* problema dei lettori-scrittori richiede che uno scrittore, una volta pronto, esegua il proprio compito di scrittura al più presto. In altre parole, se uno scrittore attende l’accesso all’insieme di dati, nessun nuovo lettore deve iniziare la lettura.

La soluzione del primo problema e quella del secondo possono condurre a uno stato d’attesa indefinita (*starvation*), degli scrittori, nel primo caso; dei lettori, nel secondo. Per questo motivo sono state proposte altre varianti. Nel seguito si presenta una soluzione del primo problema dei lettori-scrittori; indicazioni attinenti a soluzioni del secondo problema immuni all’attesa indefinita si trovano nelle note bibliografiche.

La soluzione del primo problema dei lettori-scrittori prevede dunque la condivisione da parte dei processi lettori delle seguenti strutture dati:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

I semafori `mutex` e `rw_mutex` sono inizializzati a 1; `read_count` è inizializzato a 0. Il semaforo `rw_mutex` è comune a entrambi i tipi di processi (lettori e scrittori). Il semaforo `mutex` si usa per assicurare la mutua esclusione al momento dell’aggiornamento di `read_count`. La variabile `read_count` contiene il numero dei processi che stanno attualmente leggendo l’insieme di dati. Il semaforo `rw_mutex` funziona come semaforo di mutua esclusione per gli scrittori e serve anche al primo o all’ultimo lettore che entra o esce dalla sezione critica. Non serve, invece, ai lettori che entrano o escono mentre altri lettori si trovano nelle rispettive sezioni critiche.

La Figura 5.11 illustra la struttura di un processo scrittore; la Figura 5.12 presenta la struttura di un processo lettore. Occorre notare che se uno scrittore si trova nella

```

do {
    wait(rw_mutex);
    . . .
    /* esegui l'operazione di scrittura */
    . . .
    signal(rw_mutex);
} while (true);

```

Figura 5.11 Struttura generale di un processo scrittore.

sezione critica e n lettori attendono di entrarvi, si accoda un lettore a `rw_mutex` e $n - 1$ lettori a `mutex`. Inoltre, se uno scrittore esegue `signal(rw_mutex)` si può riprendere l'esecuzione dei lettori in attesa, oppure di un singolo scrittore in attesa. La scelta è fatta dallo scheduler.

Le soluzioni al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire **lock di lettura-scrittura**. Per acquisire un tale lock, è necessario specificarne la modalità, scrittura o lettura: se il processo desidera solo leggere i dati condivisi, richiede un lock di lettura-scrittura in modalità lettura; se invece desidera anche modificare i dati, lo richiede in modalità scrittura. È permesso a più processi di acquisire lock di lettura-scrittura in modalità lettura, ma solo un processo alla volta può avere il lock di lettura-scrittura in modalità scrittura, visto che nel caso della scrittura è necessario garantire l'accesso esclusivo.

I lock di lettura-scrittura sono utili soprattutto nelle situazioni seguenti.

- Nelle applicazioni in cui è facile identificare i processi che si limitano alla lettura di dati condivisi e quelli che si limitano alla scrittura di dati condivisi.

```

do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    . . .
    /* esegui l'operazione di lettura */
    . . .
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);

```

Figura 5.12 Struttura generale di un processo lettore.

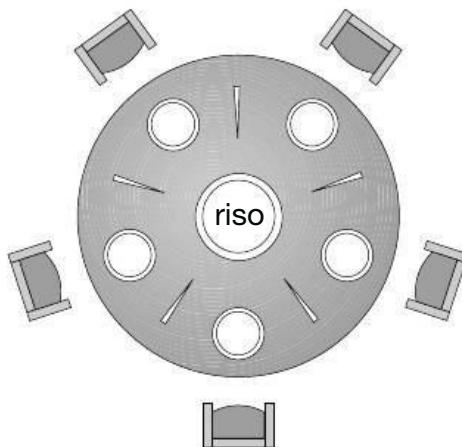


Figura 5.13 Situazione dei cinque filosofi (dining philosophers).

- Nelle applicazioni che prevedono più lettori che scrittori. Infatti, i lock di lettura-scrittura comportano in genere un carico di lavoro aggiuntivo rispetto ai semafori o ai lock mutex, compensato però dalla possibilità di eseguire molti lettori in con-correnza.

5.7.3 Problema dei cinque filosofi (dining philosophers)

Si considerino cinque filosofi che trascorrono la loro esistenza pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso, e la tavola è apparecchiata con cinque bacchette (in inglese chopstick). Si veda la Figura 5.13. Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino. Quando un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare.

Il *problema dei cinque filosofi* (*dining philosophers*) è considerato un classico problema di sincronizzazione, non certo per la sua importanza pratica, e neanche per antipatia verso i filosofi da parte degli informatici, ma perché rappresenta una vasta classe di problemi di controllo della concorrenza, in particolare i problemi caratterizzati dalla necessità di assegnare varie risorse a diversi processi evitando situazioni di stallo e d'attesa indefinita.

Una semplice soluzione consiste nel rappresentare ogni bacchetta con un semaforo: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione `wait()` su quel semaforo e la posa eseguendo operazioni `signal()` sui semafori appropriati. Quindi, i dati condivisi sono

```
semaphore chopstick[5];
```

```

do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    . . .
    /* mangia */
    . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    . . .
    /* pensa */
    . . .
} while (true);

```

Figura 5.14 Struttura del filosofo *i*.

dove tutti gli elementi `chopstick` sono inizializzati a 1. La struttura del filosofo *i* è illustrata nella Figura 5.14.

Questa soluzione garantisce che due vicini non mangino contemporaneamente, ma è insufficiente poiché non esclude la possibilità che si abbia una situazione di stallo. Si supponga che tutti e cinque i filosofi abbiano fame contemporaneamente e che ciascuno afferri la bacchetta di sinistra; tutti gli elementi di `chopstick` diventano uguali a zero, perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo. Tali situazioni di stallo possono essere evitate con i seguenti espedienti:

- solo quattro filosofi possono stare contemporaneamente a tavola;
- un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (quest'operazione si deve eseguire in una sezione critica);
- si adotta una soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

Nel Paragrafo 5.8 presentiamo una soluzione al problema dei cinque filosofi che assicura l'assenza di situazioni di stallo. Si noti tuttavia che qualsiasi soluzione soddisfacente per il problema dei cinque filosofi deve escludere la possibilità di situazioni d'attesa indefinita, in altre parole che uno dei filosofi muoia di fame (da qui il termine *starvation*) – una soluzione immune alle situazioni di stallo non esclude necessariamente la possibilità di situazioni d'attesa indefinita.

5.8 Monitor

Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione che non si verificano sempre.

Un esempio di tali errori si nell'utilizzo dei contatori della nostra soluzione al problema produttore/consumatore (Paragrafo 5.1). In quella circostanza, il problema di sincronizzazione appariva solo sporadicamente, e anche il valore del contatore si manteneva entro limiti ragionevoli, essendo sfasato tutt'al più di 1. Ciononostante, le soluzioni di questo tipo restano inaccettabili, ed è per ottenere soluzioni soddisfacenti che sono stati inventati i semafori.

Neanche l'uso dei semafori, purtroppo, esclude la possibilità che si verifichi qualche errore di sincronizzazione. Per capire perché, analizziamo la soluzione al problema della sezione critica. Tutti i processi condividono una variabile semaforo `mutex`, inizializzata a 1. Ogni processo deve eseguire `wait(mutex)` prima di entrare nella sezione critica e `signal(mutex)` al momento di uscirne. Se questa sequenza non è rispettata, può accadere che due processi occupino simultaneamente le rispettive sezioni critiche. Esaminiamo le difficoltà che possono insorgere. Si noti che tali difficoltà possono insorgere anche nel caso che *un solo* processo si comporti in maniera non corretta. L'inconveniente può nascere da un involontario errore di programmazione o essere causato dalla mancata collaborazione del programmatore.

- Supponiamo che un processo capovolga l'ordine in cui sono eseguite le istruzioni `wait()` e `signal()`, in questo modo:

```
signal(mutex);  
.  
.  
.  
sezione critica  
.  
.  
.  
wait(mutex);
```

In questa situazione, numerosi processi possono eseguire le proprie sezioni critiche allo stesso tempo, violando il requisito della mutua esclusione. Questo errore può essere scoperto solo qualora diversi processi siano attivi simultaneamente nelle rispettive sezioni critiche. Si osservi che tale situazione potrebbe non essere sempre riproducibile.

- Ipotizziamo che un processo sostituisca `signal(mutex)` con `wait(mutex)`, cioè che esegua

```
wait(mutex);  
.  
.  
.  
sezione critica  
.  
.  
.  
wait(mutex);
```

Si genera, in questo caso, uno stallo (*deadlock*).

- Si supponga che un processo ometta `wait(mutex)`, `signal(mutex)`, o entrambi. In questo caso si viola la mutua esclusione oppure si genera uno stallo.

Questi esempi chiariscono come sia facile incorrere in errori allorché i programmatori utilizzino i semafori in maniera scorretta, nel tentativo di risolvere il problema delle sezioni critiche. Problemi di natura simile possono insorgere negli altri modelli di sincronizzazione, esaminati nel Paragrafo 5.7.

Per rimediare a questi errori, i ricercatori hanno sviluppato costrutti per linguaggi ad alto livello. Un costrutto fondamentale di sincronizzazione ad alto livello – il tipo **monitor** – è descritto nel paragrafo successivo.

5.8.1 Uso del costrutto monitor

Un **tipo di dato astratto** (ADT) incapsula i dati mettendo a disposizione un insieme di funzioni per operare su di essi; tali funzioni sono indipendenti dalla specifica implementazione del tipo di dato. Il monitor è un ADT che comprende un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione. Il tipo monitor contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza del tipo, oltre al corpo delle procedure o funzioni che operano su tali variabili. La sintassi di un monitor è mostrata nella Figura 5.15. La rappresentazione di un tipo monitor non può essere usata direttamente dai vari processi. Pertanto, una funzione definita all'interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, situate nel monitor, e ai relativi parametri formali. In modo analogo, alle variabili locali di un monitor possono accedere solo le procedure locali.

```
monitor monitor name
{
    /* dichiarazione di variabili condivise */
    function P1 ( . . . ) {
        . . .
    }
    function P2 ( . . . ) {
        . . .
    }
    .
    .
    .
    function Pn ( . . . ) {
        . . .
    }
    initialization_code ( . . . ) {
        . . .
    }
}
```

Figura 5.15 Sintassi di un monitor.

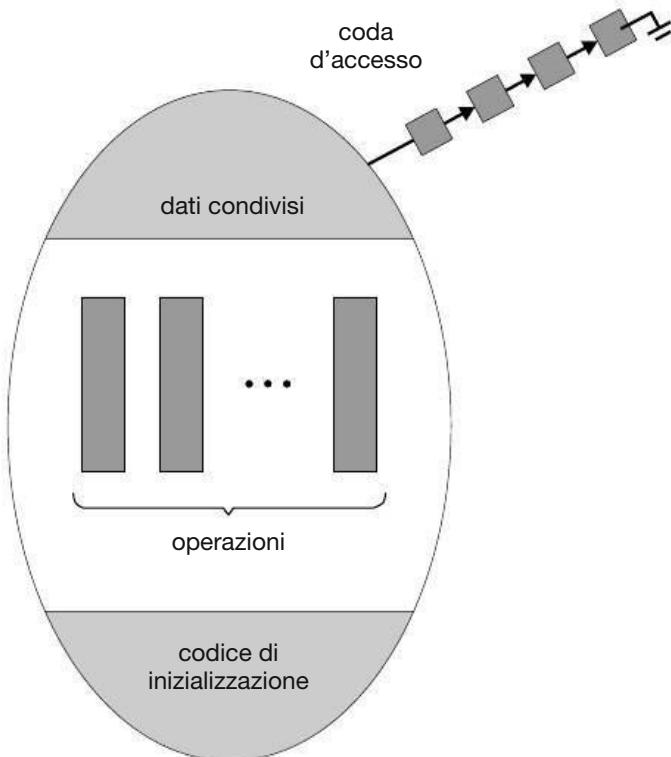


Figura 5.16 Schema di un monitor.

Il costrutto monitor assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, sicché non si deve codificare esplicitamente il vincolo di mutua esclusione (Figura 5.16). Tuttavia la definizione di monitor presentata finora non è abbastanza potente per esprimere alcuni schemi di sincronizzazione, sono perciò necessari ulteriori meccanismi forniti dal costrutto `condition`. Un programmatore che necessita di implementare un proprio particolare schema di sincronizzazione può definire uno più variabili di tipo `condition`:

```
condition x, y;
```

Le uniche operazioni eseguibili su una variabile `condition` sono `wait()` e `signal()`. L'operazione

```
x.wait();
```

implica che il processo che la invoca rimanga sospeso finché un altro processo non invochi l'operazione

```
x.signal();
```

che risveglia esattamente un processo sospeso. Se non esistono processi sospesi l'operazione `signal()` non ha alcun effetto, vale a dire che lo stato di `x` resta immutato, come se l'operazione non fosse stata eseguita; la situazione è descritta nella Figura 5.17. Tutto ciò contrasta con l'operazione `signal()` associata ai semafori, poiché questa influisce sempre sullo stato del semaforo.

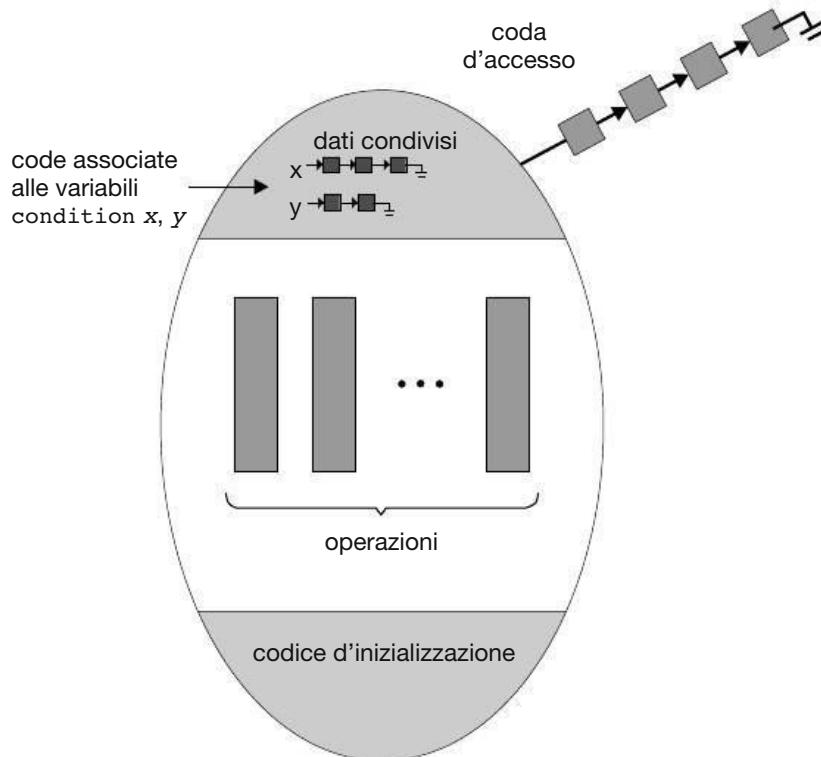


Figura 5.17 Monitor con variabili condition.

Si supponga, per esempio, che quando un processo P invoca l'operazione `x.signal()`, esista un processo sospeso Q associato alla variabile `x` di tipo `condition`. Chiaramente, se al processo sospeso Q si permette di riprendere l'esecuzione, il processo segnalante P è costretto ad attendere, altrimenti P e Q sarebbero contemporaneamente attivi all'interno del monitor. Occorre tuttavia notare che, concettualmente, entrambi i processi possono continuare l'esecuzione. Sussistono due possibilità:

1. **segnalare e attendere.** P attende che Q lasci il monitor o attenda su un'altra variabile `condition`;
2. **segnalare e proseguire.** Q attende che P lasci il monitor o attenda su un'altra variabile `condition`.

Si possono fornire argomenti ragionevoli a favore dell'uno o dell'altra opzione. Da un lato, visto che P era già in esecuzione all'interno del monitor, il secondo metodo appare più ragionevole. D'altro canto, se si lascia proseguire il thread P , la condizione attesa da Q potrebbe non valere più al momento in cui quest'ultimo riprende l'esecuzione. Il linguaggio Concurrent Pascal ha scelto un compromesso: quando il thread P esegue l'operazione `signal()`, lascia subito il monitor; pertanto, Q riprende immediatamente l'esecuzione.

Molti linguaggi di programmazione incorporano l'idea di monitor descritta in questo paragrafo. Tra questi vi sono Java e C# (da leggersi “C-sharp”). Altri linguaggi,

come Erlang, forniscono alcune tipologie di supporto alla concorrenza usando un meccanismo simile.

5.8.2 Soluzione al problema dei cinque filosofi per mezzo di monitor

Illustriamo quindi i concetti relativi al costrutto monitor presentando una soluzione esente da stallo del problema dei cinque filosofi (*dining philosophers*). La soluzione impone il vincolo che un filosofo possa prendere le sue bacchette solo quando siano entrambe disponibili. Per codificare questa soluzione si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. A tale scopo si introduce la seguente struttura dati:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Il filosofo i può impostare la variabile `state[i] = EATING` solo se i suoi due vicini non stanno mangiando:

```
((state[(i + 4) % 5] != EATING) && (state[(i + 1) % 5] != EATING)).
```

Inoltre, occorre dichiarare la seguente struttura dati:

```
condition self[5];
```

che permette al filosofo i di ritardare se stesso quando ha fame, ma non riesce a ottenere le bacchette di cui ha bisogno.

A questo punto si può descrivere la soluzione al problema dei cinque filosofi. La distribuzione delle bacchette è controllata dal monitor `DiningPhilosophers` (Figura 5.18). Ciascun filosofo, prima di cominciare a mangiare, deve invocare l'operazione `pickup()`; ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione, il filosofo può mangiare; in seguito, il filosofo invoca l'operazione `putdown()` e comincia a pensare. Il filosofo i deve quindi chiamare le operazioni `pickup()` e `putdown()` nella seguente sequenza:

```
DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);
```

È facile dimostrare che questa soluzione assicura che due vicini non mangino contemporaneamente e che non si verifichino situazioni di stallo. Occorre però notare che un filosofo può attendere indefinitamente. La soluzione di questo problema è lasciata come esercizio per il lettore.

```

monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }
    Initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}

```

Figura 5.18 Una soluzione con monitor al problema dei cinque filosofi.

5.8.3 Realizzazione di un monitor per mezzo di semafori

A questo punto si considera una possibile realizzazione del meccanismo del monitor usando i semafori. A ogni monitor si associa un semaforo `mutex`, inizializzato a 1; un processo deve eseguire `wait(mutex)` prima di entrare nel monitor, e `signal(mutex)` dopo aver lasciato il monitor.

Poiché un processo che esegue una `signal()` deve attendere finché il processo risvegliato si metta in attesa o lasci il monitor, si introduce un altro semaforo, `next`, inizializzato a 0, su cui i processi che eseguono una `signal()` possono autosospen-

dersi. Per contare i processi sospesi al semaforo `next`, si usa una variabile intera `next_count`. Quindi, ogni procedura esterna `F` si sostituisce col seguente codice:

```
wait(mutex);
...
corpo di F
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

In questo modo si assicura la mutua esclusione all'interno del monitor.

A questo punto si può descrivere la realizzazione delle variabili `condition`. Per ogni variabile `x` di tipo `condition` si introducono un semaforo `x_sem` e una variabile intera `x_count`, entrambi inizializzati a 0. L'operazione `x.wait()` si può realizzare come segue:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

L'operazione `x.signal()` si può realizzare come segue:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Questa soluzione è applicabile alle definizioni di monitor date sia da Hoare sia da Brinch-Hansen (si vedano le note bibliografiche alla fine del capitolo). In alcuni casi, tuttavia, questo livello di generalità della codifica non è necessario, e si possono apportare notevoli miglioramenti all'efficienza. La soluzione a questo problema è lasciata al lettore nell'Esercizio 5.30.

5.8.4 Ripresa dei processi all'interno di un monitor

Ritorniamo ora al problema dell'ordine di ripresa dei processi all'interno di un monitor. Se più processi sono sospesi alla condizione `x`, e se qualche processo esegue l'operazione `x.signal()`, è necessario stabilire quale tra i processi sospesi si debba riattivare per primo. Una semplice soluzione consiste nell'usare un ordinamento FCFS

```

monitor assegnazione_risorse
{
    boolean occupato;
    condition x;

    acquisizione(int tempo) {
        if (occupato)
            x.wait(tempo);
        occupato = true;
    }

    rilascio() {
        occupato = false;
        x.signal();
    }

    codice di inizializzazione() {
        occupato = false;
    }
}

```

Figura 5.19 Un monitor per l'assegnazione di una singola risorsa.

(first-come, first-served), secondo cui il processo che attende da più tempo viene ripreso per primo. Tuttavia, in molti casi uno schema di scheduling di questo tipo non risulta adeguato; in questi casi si può usare un costrutto di **attesa condizionale** della forma

```
x.wait(c);
```

dove con **c** si indica un'espressione intera che si valuta al momento dell'esecuzione dell'operazione `wait()`. Il valore di **c**, chiamato **numero di priorità**, viene poi memorizzato col nome del processo sospeso. Quando si esegue `x.signal()`, si riprende il processo cui è associato il numero di priorità più basso.

Per comprendere questo nuovo meccanismo, si consideri il monitor illustrato nella Figura 5.19; tale monitor ha il compito di assegnare una particolare risorsa a processi in competizione. Quando richiede l'assegnazione di una delle sue risorse, ogni processo specifica il tempo massimo per il quale prevede di usare la risorsa. Il monitor assegna la risorsa al processo con la richiesta di assegnazione più breve.

Per accedere alla risorsa in questione il processo deve rispettare la sequenza:

```

R.acquire(t);
...
accesso alla risorsa;
...
R.release();

```

dove **R** è un'istanza di tipo `assegnazione_risorse`.

Sfortunatamente il concetto di monitor non può garantire che la precedente sequenza d’accesso sia rispettata. In particolare, può accadere quanto segue:

- un processo può accedere alla risorsa senza prima ottenere il permesso d’accesso;
- una volta che ne ha ottenuto l’accesso, un processo può non rilasciare più la risorsa;
- un processo può tentare di rilasciare una risorsa che non ha mai richiesto;
- un processo può richiedere due volte la stessa risorsa, senza rilasciarla prima della seconda richiesta.

Le stesse difficoltà si sono incontrate nell’uso dei semafori e sono, in realtà, simili alle difficoltà che condussero allo sviluppo del costrutto monitor. In precedenza ci si è preoccupati del corretto uso dei semafori, ora ci si deve preoccupare del corretto uso delle operazioni ad alto livello definite dal programmatore, senza poter avere, a questo livello, l’assistenza del compilatore.

Una possibile soluzione del problema precedente prevede l’inclusione delle operazioni d’accesso alle risorse all’interno del monitor `assegnazione_risorse`. Tuttavia, adottando questa soluzione, per lo scheduling delle risorse si userebbe l’algoritmo base di scheduling del monitor anziché quello che abbiamo codificato.

Per garantire che i processi rispettino le sequenze appropriate, è necessario controllare tutti i programmi che usano il monitor `assegnazione_risorse` e la risorsa da esso gestita. Per stabilire la correttezza del sistema è necessario verificare le seguenti due condizioni: la prima, che i processi utenti devono sempre impiegare il monitor secondo una sequenza corretta; la seconda, che è necessario assicurare che un processo non cooperativo non cerchi di aggirare la mutua esclusione offerta dal monitor, e tenti di accedere direttamente alla risorsa condivisa senza usare i protocolli d’accesso. Soltanto se si assicurano queste due condizioni, si può garantire l’assenza di errori di sincronizzazione e che l’algoritmo di scheduling sia rispettato.

Questo controllo è possibile per sistemi statici di piccole dimensioni, mentre non è ragionevolmente applicabile a sistemi di grandi dimensioni o a sistemi dinamici. Questo problema di controllo dell’accesso si può risolvere solo introducendo ulteriori meccanismi, descritti nel Capitolo 14.

5.9 Esempi di sincronizzazione

Si descrivono ora i meccanismi di sincronizzazione forniti dai sistemi operativi Windows, Linux e Solaris, e le API Pthreads. Abbiamo prescelto questi tre sistemi perché offrono buoni esempi di approcci differenti rispetto alla sincronizzazione del kernel; le API Pthreads sono state incluse nella trattazione perché ampiamente utilizzate dagli sviluppatori per la creazione e la sincronizzazione dei thread su piattaforme UNIX e Linux. Come si vedrà nel corso del paragrafo i metodi di sincronizzazione messi a disposizione da questi sistemi variano in modo sottile ma significativo.

MONITOR IN JAVA

Per la sincronizzazione dei thread Java fornisce un meccanismo affine a quello del monitor. Ciascun oggetto, in Java, ha associato un singolo lock. Quando si dichiara un metodo `synchronized`, per invocare il metodo su un oggetto occorre possedere il lock dell'oggetto.

Si dichiara `synchronized` un metodo inserendo la parola chiave nella definizione del metodo. Il codice che segue, per esempio, dichiara `metodoSicuro()` come `synchronized`:

```
public class SempliceClasse {
    ...
    public synchronized void metodoSicuro() {
        ...
        /* Implementazione di metodoSicuro() */
        ...
    }
}
```

Si supponga di creare un'istanza di `SempliceClasse()` nel modo seguente:

```
SempliceClasse sc = new SempliceClasse();
```

Per richiamare il metodo `sc.metodoSicuro()` è necessario possedere il lock dell'istanza `sc`. Se il lock è già proprietà di un thread diverso, il thread che invoca il metodo dichiarato `synchronized` si blocca ed è collocato nella lista d'attesa per il lock. La lista d'attesa è formata dall'insieme di thread che attendono la disponibilità del lock. Se, al momento dell'invocazione di un metodo `synchronized`, il lock è disponibile, il thread chiamante diviene il proprietario del lock dell'oggetto e può accedere al metodo. Il lock ritorna disponibile quando il thread termina l'esecuzione del metodo; un thread in lista d'attesa, quindi, è selezionato come nuovo proprietario del lock.

Java offre inoltre i metodi `wait()` e `notify()`, che funzionano analogamente alle istruzioni `wait()` e `signal()` per i monitor. La API Java comprende il supporto di semafori, variabili condizionali e semafori mutex (tra gli altri meccanismi per la concorrenza) nel package `java.util.concurrent`.

5.9.1 Sincronizzazione in Windows

Il sistema operativo Windows ha un kernel multithread che offre anche il supporto alle applicazioni in tempo reale e alle architetture multiprocessore. Quando il kernel di Windows accede a una risorsa globale in un sistema con monoprocesso, disabilita temporaneamente le interruzioni con interrupt handler che potrebbero accedere alla stessa risorsa globale. In un sistema multiprocessore, Windows protegge l'accesso alle risorse globali con i semafori ad attesa attiva (*spinlock*), anche se il kernel usa i semafori ad attesa attiva solo per proteggere segmenti di codice brevi. Inoltre, per ragioni di efficienza, il kernel impedisce che un thread sia sottoposto a prelazione mentre detiene uno spinlock.

Per la sincronizzazione fuori dal kernel, Windows offre gli **oggetti dispatcher**, che permettono ai thread di sincronizzarsi servendosi di diversi meccanismi, inclusi

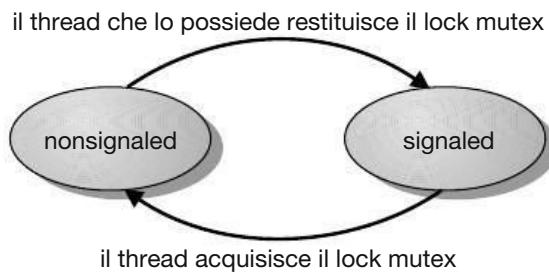


Figura 5.20 Oggetto dispatcher di tipo mutex.

lock mutex, semafori, eventi e timer. I dati condivisi vengono protetti richiedendo che un thread entri in possesso di un mutex prima di potervi accedere, e rilasci il mutex al completamento dell’elaborazione di quei dati. Il comportamento dei semafori è analogo a quello illustrato nel Paragrafo 5.6. Gli **eventi** sono un meccanismo di sincronizzazione utilizzabile in modo simile alle variabili condizionali; cioè, possono notificare il verificarsi di una determinata condizione a un thread che l’attendeva. Infine i timer sono usati per informare un thread (o più di uno) della scadenza di uno specifico periodo di tempo.

Gli oggetti dispatcher possono essere nello stato *signaled* o nello stato *nonsignaled*. Uno **stato signaled** indica che l’oggetto è disponibile e che un thread che tentasse di accedere all’oggetto non sarebbe bloccato; uno **stato nonsignaled** indica che l’oggetto non è disponibile e che qualsiasi thread che tentasse di accedervi sarebbe bloccato. La Figura 5.20 illustra le transizioni di stato di un oggetto dispatcher di tipo lock mutex.

C’è una relazione tra lo stato di un *oggetto dispatcher* e lo stato di un thread. Se un thread si blocca a un *oggetto dispatcher* nello stato *nonsignaled*, il suo stato cambia da pronto per l’esecuzione ad attesa, e il thread viene messo nella coda d’attesa per quell’oggetto. Quando lo stato dell’*oggetto dispatcher* diventa *signaled*, il kernel verifica se ci sono thread nella coda di attesa dell’oggetto, e in tal caso ne fa passare uno, o più d’uno, dallo stato di attesa allo stato di pronto per l’esecuzione, dal quale può riprendere l’esecuzione. Il numero dei thread che il kernel seleziona dalla coda d’attesa dipende dal tipo di *oggetto dispatcher* su cui attendono. Il kernel seleziona un solo thread dalla coda d’attesa nel caso di un mutex, poiché un oggetto mutex può essere posseduto da un solo thread. Nel caso di un oggetto evento, il kernel seleziona tutti i thread che attendono l’evento stesso.

Consideriamo un lock mutex come esempio per illustrare gli *oggetti dispatcher* e gli stati dei thread. Se un thread cerca di acquisire un *oggetto dispatcher* di tipo mutex che è nello stato *nonsignaled*, viene sospeso e messo in una coda d’attesa per l’oggetto mutex. Quando il mutex passa allo stato *signaled* (come risultato del rilascio del lock mutex da parte di un altro thread), il thread in attesa in testa alla coda del mutex passa dallo stato d’attesa allo stato di pronto per l’esecuzione e acquisisce il lock mutex.

Un **oggetto sezione critica** è un mutex in modalità utente che può spesso essere acquisito e rilasciato senza l'intervento del kernel. In un sistema multiprocessore, un oggetto sezione critica usa inizialmente uno spinlock in attesa che l'altro thread rilasci l'oggetto. Se l'attesa è troppo lunga, il thread alloca un mutex del kernel e cede la sua CPU. Gli oggetti sezione critica sono particolarmente efficienti perché il mutex del kernel viene allocato solo quando c'è contesa per l'oggetto. Nella pratica vi sono ben poche contese, dunque il risparmio è notevole.

Alla fine di questo capitolo si trova un progetto di programmazione che utilizza i lock mutex e i semafori nella API di Windows.

5.9.2 Sincronizzazione dei processi in Linux

Prima della versione 2.6, Linux adoperava un kernel senza prelazione; ciò significa che un processo in esecuzione in modalità kernel non poteva essere prelazionato – neppure nel caso in cui processi con priorità più alta fossero pronti per l'esecuzione. Ora, per contro, il kernel di Linux ha adottato compiutamente il procedimento della prelazione, cosicché i task attivi nel kernel possono essere sottoposti a prelazione.

Linux fornisce diversi meccanismi per la sincronizzazione nel kernel. Dato che la maggior parte delle architetture fornisce istruzioni per le versioni atomiche di semplici operazioni matematiche, la tecnica di sincronizzazione più semplice nel kernel di Linux è l'intero atomico, rappresentato mediante il tipo di dato opaco `atomic_t`. Come suggerito da questo nome, tutte le operazioni matematiche che usano numeri interi atomici vengono eseguite senza interruzioni. Il codice seguente mostra la dichiarazione di un intero atomico `counter` e l'esecuzione di varie operazioni atomiche:

```
atomic_t counter;
int value;

atomic_set(&counter,5); /* counter = 5 */
atomic_add(10, &counter); /* counter = counter + 10 */
atomic_sub(4, &counter); /* counter = counter - 4 */
atomic_inc(&counter); /* counter = counter + 1 */
value = atomic_read(&counter); /* value = 12 */
```

Gli interi atomici sono particolarmente efficienti in situazioni in cui deve essere aggiornata una variabile intera, per esempio un contatore, in quanto non risentono dell'overhead dei meccanismi di lock. Il loro utilizzo è tuttavia limitato a questi tipi di scenario. In situazioni in cui vi sono diverse variabili che contribuiscono a una possibile race condition, devono essere utilizzati strumenti di lock più sofisticati.

In Linux sono disponibili i lock mutex, utili per proteggere le sezioni critiche all'interno del kernel. In caso di loro utilizzo, un task deve invocare la funzione `mutex_lock()` prima di entrare in una sezione critica e la funzione `mutex_unlock()` dopo l'uscita dalla sezione critica. Se il lock mutex non è disponibile, il

task che ha invocato la `mutex_lock()` viene sospeso; verrà risvegliato quando il proprietario del lock invoca `mutex_unlock()`.

Linux fornisce anche spinlock e semafori (nonché la variante lettore-scrittore di questi due meccanismi) per implementare i lock a livello kernel. Su macchine SMP, il meccanismo fondamentale è lo spinlock; il kernel è progettato in modo da mantenere attivi gli spinlock solo per brevi periodi di tempo. Sulle macchine monoprocesore, come nel caso di sistemi embedded con un solo core di elaborazione, gli spinlock sono inadatti, e si ricorre all’abilitazione e inibizione del diritto di prelazione nel kernel. Su tali macchine, in pratica, anziché attivare uno spinlock, il kernel inibisce la prelazione; anziché rimuovere lo spinlock, abilita la prelazione. In sintesi:

monoprocessore	multiprocessore
Inibisce la prelazione a livello kernel.	Attiva spinlock.
Abilita la prelazione a livello kernel.	Rimuove spinlock.

Il modo impiegato da Linux per abilitare e inibire il diritto di prelazione nel kernel è di particolare interesse; si basa su due semplici chiamate di sistema, `preempt_disable()` e `preempt_enable()`. Tuttavia non è possibile sottoporre il kernel a prelazione se un task attivo nel kernel possiede un lock. Per implementare questa regola, ogni task nel sistema possiede una struttura, `thread_info`, in cui un contatore, `preempt_count`, indica il numero dei lock detenuti dal task. Quando un lock viene acquisito, `preempt_count` aumenta di uno, mentre diminuisce di uno quando viene rilasciato. Qualora il valore di `preempt_count` per il task in esecuzione sia maggiore di zero, sarebbe rischioso sottoporre a prelazione il kernel, dato che il task possiede un lock. Se il valore è 0, il kernel può subire l’interruzione (assumendo che non vi siano chiamate in sospeso a `preempt_disable()`).

Gli spinlock, insieme all’abilitazione e inibizione della prelazione, sono utilizzati nel kernel solo quando si ricorre per breve tempo a un lock (o alla disabilitazione della prelazione del kernel). Quando vi sia necessità di mantenere un lock attivo più a lungo, è opportuno utilizzare i semafori o i lock mutex.

5.9.3 Sincronizzazione in Solaris

Per regolare l’accesso alle sezioni critiche, Solaris mette a disposizione lock mutex adattivi, variabili condizionali, semafori, lock di lettura-scrittura e i cosiddetti tornelli (*turnstiles*). Solaris implementa i semafori e le variabili condizionali come descritto nei Paragrafi 5.6 e 5.7. Qui descriviamo lock mutex adattivi, lock di lettura-scrittura e tornelli.

Un **mutex adattivo** (*adaptive mutex*) protegge l’accesso a ogni dato critico; in un sistema multiprocessore si attiva come un semaforo ordinario realizzato come uno spinlock. Se i dati sono soggetti a lock e quindi già in uso, nel mutex adattivo si possono verificare due situazioni: se i dati sono posseduti da un thread correntemente in esecuzione in un’altra unità d’elaborazione, il thread che ha fatto la nuova richiesta

entra in uno stato d'attesa attiva, mentre aspetta la disponibilità del lock, poiché è probabile che il thread in possesso dei dati termini la propria elaborazione in breve tempo; viceversa, se quest'ultimo non si trova nello stato d'esecuzione, il thread richiedente si sospende nello stato d'attesa fino al rilascio del lock. In questo modo si evita il ciclo d'attesa attiva, poiché probabilmente il lock non sarà rilasciato in un tempo breve. Si ha questa situazione, per esempio, quando il lock è detenuto da un thread bloccato. Poiché un sistema dotato di una singola CPU può eseguire un solo thread alla volta, il thread che possiede il lock non è mai in esecuzione nell'istante in cui un altro thread verifica la presenza del lock. Quindi, in un sistema con una CPU, un thread che incontra un lock sospende sempre la propria esecuzione anziché ciclare in attesa.

Solaris adotta il metodo del mutex adattivo per proteggere soltanto i dati cui si accede da segmenti di codice corti; in pratica si usa solo se un lock viene detenuto per meno di qualche centinaio di istruzioni. Se il segmento di codice fosse più lungo, l'uso dei cicli d'attesa sarebbe inefficiente. Per segmenti di codice più lunghi il sistema ricorre all'impiego di semafori e variabili condizionali. Se il lock richiesto è già posseduto da altri thread, il thread richiedente invoca una `wait()` e si sospende. Quando il thread che possiede il lock ne rilascia il controllo, invia un segnale al successivo thread presente nella coda d'attesa. L'ulteriore costo richiesto dalla sospensione e dalla successiva riattivazione del thread, compresi i relativi cambi di contesto, è minore di quello dovuto alla esecuzione di centinaia di istruzioni in uno spinlock.

I lock di lettura-scrittura si usano per proteggere i dati cui si accede spesso, ma di solito per la sola lettura. In tali circostanze essi sono più efficienti dei semafori, poiché più thread possono leggere i dati in modo concorrente, mentre un semaforo avrebbe imposto la serializzazione di questi accessi. Poiché la loro realizzazione introduce un costo aggiuntivo, anche i lock di lettura-scrittura si applicano solo alle sezioni di codice lunghe.

Solaris utilizza i *tornelli* per ordinare la lista dei thread che attendono di ottenere un mutex adattivo o un lock di lettura-scrittura. Un **tornello** (*turnstile*) è una struttura a coda contenente i thread che attendono il rilascio di un lock. Per esempio, se un thread possiede il lock di un oggetto sincronizzato, tutti gli altri thread che cercano di acquisirlo si bloccano ed entrano nel tornello relativo a quel lock. Quando il lock è rimosso, il kernel seleziona un thread dal tornello per concedergli la proprietà del lock. Ogni oggetto sincronizzato con almeno un thread che attende di acquisire il lock richiede un tornello separato. Tuttavia, anziché associare un tornello a ciascun oggetto sincronizzato, Solaris assegna un tornello a ogni thread a livello kernel.

Il tornello del primo thread che si blocca su un oggetto sincronizzato diventa il tornello per l'oggetto stesso, i thread successivi si aggiungono allo stesso tornello. Quando il thread iniziale infine rilascia il lock, acquisisce un nuovo tornello da una lista di tornelli liberi mantenuta dal kernel. Per prevenire un'**inversione delle priorità**, i tornelli sono organizzati secondo un **protocollo di ereditarietà delle priorità**. Ciò significa che, se un thread detiene il lock di un oggetto su cui è in attesa un thread a priorità maggiore, il thread a priorità minore eredita temporaneamente la

priorità del thread a priorità maggiore. Al rilascio del lock, il thread riassume la priorità originaria.

Si noti che i meccanismi di gestione dei lock usati dal kernel sono disponibili anche per i thread a livello utente, sicché gli stessi tipi di lock sono disponibili sia all'interno sia all'esterno del kernel. Una differenza cruciale nella loro realizzazione è il protocollo di ereditarietà delle priorità: i lock del kernel adottano i metodi di ereditarietà delle priorità del kernel usati dallo scheduler (Paragrafo 5.6.4); quelli dei thread a livello utente non offrono questa funzionalità.

Per ottimizzare le prestazioni di Solaris, gli sviluppatori hanno via via perfezionato e calibrato l'implementazione dei lock. Poiché i lock si usano frequentemente, e spesso per funzioni cruciali del kernel, la loro ottimizzazione può condurre a notevoli incrementi delle prestazioni.

5.9.4 Sincronizzazione in Pthreads

Anche se i meccanismi di lock utilizzati in Solaris sono disponibili sia per thread a livello di utente sia per thread del kernel, in pratica i metodi di sincronizzazione discussi finora riguardano la sincronizzazione all'interno del kernel. L'API Pthreads è invece disponibile per i programmatori a livello utente e non è parte di alcun particolare kernel. Questa API fornisce lock mutex, variabili condizionali e lock di lettura-scrittura per la sincronizzazione dei thread.

I lock mutex rappresentano, in ambiente Pthreads, la tecnica di sincronizzazione fondamentale. La loro finalità è di proteggere le sezioni critiche del codice: un thread acquisisce un lock prima di entrare in una sezione critica, quindi, al momento di uscirne, lo rilascia. Pthreads utilizza il tipo di dato `pthread_mutex_t` per i lock mutex. Un mutex viene creato mediante la funzione `mutex_pthread_init()`. Il primo parametro è un puntatore al mutex. Passando `NULL` come secondo parametro si inizializza il mutex agli attributi predefiniti, come illustrato di seguito:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* crea il lock mutex */
pthread_mutex_init(&mutex,NULL);
```

Il mutex viene acquisito e rilasciato con le funzioni `pthread_mutex_lock()` e `pthread_mutex_unlock()`, rispettivamente. Se il lock mutex non è disponibile quando viene invocata la `pthread_mutex_lock()`, il thread chiamante viene bloccato finché il proprietario richiama `pthread_mutex_unlock()`. Il codice seguente mostra come proteggere una sezione critica con i lock mutex.

```

/* acquisisci il lock mutex */
pthread_mutex_lock(&mutex);

/* sezione critica */

/* rilascia il lock mutex */
pthread_mutex_unlock(&mutex);

```

Tutte le funzioni mutex restituiscono 0 in caso di corretto funzionamento; se si verifica un errore restituiscono un codice di errore diverso da zero. Le variabili condizionali e i lock di lettura-scrittura si comportano in modo simile a quanto descritto nei Paragrafi 5.8 e 5.7.2, rispettivamente.

Diversi sistemi che implementano Pthreads forniscono anche i semafori, sebbene i semafori non appartengono allo standard Pthreads, ma all'estensione POSIX SEM. POSIX definisce due tipi di semafori: *con nome* e *senza nome*. La distinzione fondamentale tra i due è che un semaforo con nome ha un effettivo nome nel file system e può essere condiviso da più processi indipendenti. I semafori senza nome possono invece essere utilizzati solo da thread appartenenti allo stesso processo. In questo paragrafo descriviamo i semafori senza nome.

Il codice seguente illustra la funzione `sem_init()`, utilizzata per la creazione e l'inizializzazione di un semaforo senza nome:

```

#include <semaphore.h>
sem_t sem;

/* Crea il semaforo e inizializzalo a 1 */
sem_init(&sem, 0, 1);

```

La funzione `sem_init()` riceve tre parametri:

1. un puntatore al semaforo
2. un flag che indica il livello di condivisione
3. il valore iniziale del semaforo.

In questo esempio, passando il flag 0 indichiamo che questo semaforo può essere condiviso solo da thread appartenenti al processo che ha creato il semaforo. Un valore diverso da 0 consentirebbe anche ad altri processi di accedere al semaforo. Inoltre, nel nostro esempio, inizializziamo il semaforo al valore 1.

Nel Paragrafo 5.6 abbiamo descritto le tipiche operazioni sui semafori `wait()` e `signal()`. Pthreads chiama queste operazioni rispettivamente `sem_wait()` e `sem_post()`.

Il seguente codice mostra come proteggere una sezione critica utilizzando il semaforo creato in precedenza:

```
/* acquisisci il semaforo */
sem_wait(&sem);

/* sezione critica */

/* rilascia il semaforo */
sem_post(&sem);
```

Proprio come per i lock mutex, tutte le funzioni sui semafori restituiscono 0 in caso di successo e un valore diverso da zero quando si verifica una condizione di errore.

Ci sono altre estensioni alla API Pthreads, tra cui gli spinlock, ma è importante notare che non tutte le estensioni sono portabili da un'implementazione all'altra. Alla fine di questo capitolo vengono proposti diversi problemi e progetti di programmazione in cui si utilizzano lock mutex e variabili condizionali Pthreads, oltre ai semafori POSIX.

5.10 Approcci alternativi

Con l'emergere dei sistemi multicore si è visto un aumento della pressione per lo sviluppo di applicazioni multithread in grado di sfruttare la presenza di più core di elaborazione. Le applicazioni multithread presentano però un rischio maggiore di race condition e situazioni di stallo. Tradizionalmente, per risolvere questi problemi sono state utilizzate tecniche come i lock mutex, i semafori e i monitor, ma al crescere del numero di core diventa sempre più difficile progettare applicazioni multithread che siano esenti da race condition e stalli. In questo paragrafo esploriamo diverse funzionalità presenti sia nei linguaggi di programmazione sia a livello hardware a supporto del progetto di applicazioni concorrenti sicure.

5.10.1 Memoria transazionale

Molto spesso in informatica le idee provenienti da un'area di studio possono essere utilizzate per risolvere problemi in altre aree. Il concetto di **memoria transazionale**, che ha avuto origine nella teoria dei database, per esempio, fornisce una strategia per la sincronizzazione dei processi. Una **transazione di memoria** è una sequenza atomica di operazioni di lettura-scrittura. Se tutte le operazioni di una transazione sono eseguite, la transazione di memoria viene completata, altrimenti le operazioni devono essere annullate e deve essere ripristinata la situazione precedente l'inizio della transazione. I vantaggi della memoria transazionale possono essere sfruttati mediante l'aggiunta di nuove funzionalità a un linguaggio di programmazione.

Si consideri il seguente esempio. Si supponga di avere a disposizione una funzione `update()` che modifica dati condivisi. Questa funzione potrebbe essere scritta in maniera tradizionale usando i lock mutex (o i semafori):

```
void update ()
{
    acquire();

    /* modifica dati condivisi */

    release();
}
```

Tuttavia, l'utilizzo di meccanismi di sincronizzazione come lock mutex e semafori implica molti potenziali problemi, incluso lo stallo dei processi. Inoltre, in seguito alla crescita del numero di thread, i meccanismi tradizionali basati sui lock non si adattano al meglio, perché il livello di contesa tra i thread per la proprietà del lock diventa molto elevato.

Un'alternativa consiste nell'aggiungere ai linguaggi di programmazione nuove funzionalità che sfruttano il vantaggio dato dalla memoria transazionale. Nel nostro esempio, supponiamo di aggiungere il costrutto `atomic{S}` che assicura che le operazioni in `S` siano eseguite come transazione. Potremo riscrivere il metodo `update()` così:

```
void update ()
{
    atomic {
        /* modifica dati condivisi */
    }
}
```

Il vantaggio di utilizzare tale meccanismo al posto dei lock sta nel fatto che è il sistema di memoria transazionale, e non il programmatore, a garantire l'atomicità. Inoltre, poiché non sono coinvolti i lock, non possono verificarsi situazioni di stallo. Inoltre, questo sistema è in grado di identificare le istruzioni nei blocchi atomici che possono essere eseguite in concorrenza, come per esempio accessi concorrenti in lettura a una variabile condivisa. È comunque certamente possibile per un programmatore identificare queste situazioni e utilizzare i lock di lettura-scrittura, ma il compito si complica al crescere del numero di thread in un'applicazione.

La memoria transazionale può essere implementata via software oppure via hardware. La **memoria transazionale software** (STM), come il nome suggerisce, implementa la memoria transazionale esclusivamente via software, senza la necessità di hardware particolare. In questo schema si inserisce del codice ausiliario nelle transazioni.

Esso è prodotto e inserito da un compilatore e gestisce ciascuna transazione esaminando quali istruzioni possono essere eseguite in concorrenza, e quando sono necessari dei lock a basso livello. La **memoria transazionale hardware** (HTM) utilizza gerarchie di cache hardware e protocolli di coerenza della cache per gestire e risolvere conflitti riguardanti dati condivisi residenti in memorie cache di processori distinti. HTM non richiede una particolare strumentazione software e ha quindi un minor overhead rispetto a STM. Tuttavia, richiede che le gerarchie di cache esistenti e i protocolli di coerenza della cache siano modificati per il supporto di memorie transazionali.

Le memorie transazionali esistono da diversi anni, ma per diverso tempo non hanno conosciuto una vasta diffusione. Soltanto di recente il crescente utilizzo dei sistemi multicore e la maggior enfasi sulla programmazione concorrente e parallela hanno focalizzato molta ricerca su questo settore, sia da parte delle istituzioni accademiche sia da parte dei produttori di hardware e software.

5.10.2 OpenMP

Nel Paragrafo 4.5.2 abbiamo introdotto OpenMP e il suo supporto alla programmazione parallela in ambienti a memoria condivisa. Ricordiamo che OpenMP comprende una serie di direttive del compilatore e una API. Il codice che segue la direttiva `#pragma omp parallel` viene identificato come regione parallela e viene eseguito da un numero di thread pari al numero di core di elaborazione del sistema. Il vantaggio di OpenMP (e di strumenti analoghi) è che la creazione e la gestione dei thread è gestita dalla libreria OpenMP e non è sotto la responsabilità degli sviluppatori di applicazioni.

Oltre alla direttiva del compilatore `#pragma omp parallel`, OpenMP include la direttiva `#pragma omp critical`, che specifica che la regione di codice dopo la direttiva è una sezione critica in cui solo un thread alla volta può essere attivo. In questo modo OpenMP fornisce il supporto per garantire che i thread non generino race condition.

Come esempio dell'uso della direttiva per la sezione critica, si assuma che la variabile condivisa `counter` possa essere modificata dalla funzione `update()` come segue:

```
void update(int value)
{
    counter += value;
}
```

Se la funzione `update()` potesse far parte di una sezione critica (o essere invocata da questa), sarebbe possibile una race condition sulla variabile `counter`.

La direttiva del compilatore per la sezione critica può essere usata per porre rimedio a questa race condition nel seguente modo:

```
void update(int value)
{
    #pragma omp critical
    {
        counter += value;
    }
}
```

La direttiva per la sezione critica si comporta come un semaforo binario o un lock mutex, assicurando che solo un thread alla volta sia attivo nella sezione critica. Se un thread tenta di entrare in una sezione critica mentre un altro thread è attualmente attivo in quella sezione (cioè, **possiede** la sezione), il thread chiamante viene bloccato fino all'uscita del thread proprietario. Se è necessario utilizzare più sezioni critiche, a ciascuna sezione può essere assegnato un nome distinto e una regola può specificare che non più di un thread può essere attivo simultaneamente in una sezione critica con lo stesso nome.

Un vantaggio nell'utilizzo della direttiva del compilatore per la sezione critica in OpenMP è che è generalmente considerato più semplice rispetto ai lock mutex standard. Tuttavia, tocca ancora agli sviluppatori di applicazioni di individuare possibili race condition e proteggere adeguatamente i dati condivisi utilizzando la direttiva del compilatore. Inoltre, poiché la direttiva per la sezione critica si comporta in modo molto simile a un lock mutex, sono possibili situazioni di stallo quando vengono definite due o più sezioni critiche.

5.10.3 Linguaggi di programmazione funzionali

La maggior parte dei linguaggi di programmazione noti, come per esempio C, C++, Java e C#, sono **linguaggi imperativi** (o **procedurali**). I linguaggi imperativi sono utilizzati per l'implementazione di algoritmi basati sugli stati. In questi linguaggi il flusso dell'algoritmo è fondamentale per il suo corretto funzionamento e uno stato è rappresentato dalle variabili e dalle altre strutture dati. Naturalmente lo stato del programma cambia, visto che il valore delle variabili può essere modificato nel tempo.

Con l'enfasi che viene posta attualmente sulla programmazione concorrente e parallela per sistemi multicore, è aumentata l'attenzione verso i **linguaggi di programmazione funzionale**, che seguono un paradigma di programmazione molto diverso da quello proposto dai linguaggi imperativi. La differenza fondamentale tra i linguaggi imperativi e quelli funzionali è che i linguaggi funzionali non mantengono uno stato. In altre parole, una volta che una variabile è stata definita e inizializzata il suo valore è immutabile, non può cambiare. Poiché i linguaggi funzionali non permettono

la mutazione di stato, non si devono preoccupare di questioni come le race condition e gli stalli. In sostanza, la maggior parte dei problemi affrontati in questo capitolo non esiste nei linguaggi funzionali.

Diversi linguaggi funzionali sono attualmente in uso e noi menzioniamo molto brevemente due di questi: Erlang e Scala. Il linguaggio Erlang è stato oggetto di una significativa attenzione grazie al suo supporto alla concorrenza e alla facilità con cui può essere utilizzato per sviluppare applicazioni per sistemi paralleli. Scala è un linguaggio funzionale orientato agli oggetti. La sintassi di Scala è in gran parte simile a quella dei popolari linguaggi orientati agli oggetti Java e C#. I lettori interessati a Erlang e Scala, e in generale a ulteriori dettagli sui linguaggi funzionali, sono invitati a consultare la bibliografia alla fine del capitolo.

5.11 Sommario

Dato un gruppo di processi sequenziali cooperanti che condividono dati, è necessario implementare la mutua esclusione per assicurare che una sezione critica di codice sia utilizzabile da un solo processo o thread alla volta. Di solito l'hardware di un calcolatore fornisce diverse operazioni che assicurano la mutua esclusione, ma per la maggior parte dei programmatore queste soluzioni hardware sono troppo complicate da utilizzare. I lock mutex e i semafori consentono di superare questa difficoltà; entrambi sono utilizzabili per risolvere diversi problemi di sincronizzazione e sono realizzabili in modo efficiente, soprattutto se è disponibile un'architettura che permette le operazioni atomiche.

Sono stati presentati diversi problemi di sincronizzazione – come il problema produttore/consumatore con memoria limitata, dei lettori-scrittori e dei cinque filosofi (dining philosophers) – che costituiscono esempi rappresentativi di una vasta classe di problemi di controllo della concorrenza. Questi problemi sono stati usati per verificare quasi tutti gli schemi di sincronizzazione proposti.

Il sistema operativo deve fornire mezzi di protezione contro gli errori di sincronizzazione. A tal fine sono stati proposti parecchi costrutti di linguaggio. I monitor offrono un meccanismo di sincronizzazione per la condivisione di tipi di dati astratti. Una variabile condizionale consente a una procedura di monitor di sospendere la propria esecuzione finché non riceve un segnale.

Windows, Linux e Solaris sono esempi di sistemi operativi moderni che offrono vari meccanismi come semafori, lock mutex, spinlock e variabili condizionali per il controllo dell'accesso ai dati condivisi. La API Pthreads fornisce supporto a lock mutex, semafori e variabili condizionali.

Vi sono diversi approcci alternativi per gestire la sincronizzazione su sistemi multicores. Un approccio utilizza la memoria transazionale, che può risolvere i problemi di sincronizzazione utilizzando tecniche software o tecniche hardware. Un altro approccio utilizza le estensioni del compilatore offerte da OpenMP. Infine, i linguaggi di programmazione funzionale affrontano i problemi di sincronizzazione vietando le variazioni di stato.

Esercizi di ripasso

- 5.1** Nel Paragrafo 5.4 si afferma che disabilitando di frequente le interruzioni si può influenzare l'orologio di sistema. Spiegate perché ciò può succedere e come tali effetti possono essere mitigati.
- 5.2** Spiegate perché Windows, Linux e Solaris implementano multipli meccanismi di locking. Descrivete le circostanze in cui questi sistemi operativi utilizzano spinlock, mutex, semafori, lock mutex adattivi e variabili condizionali. Spiegate, in ciascun caso, perché occorre un tale meccanismo.
- 5.3** Qual è il significato della locuzione *attesa attiva*? Esistono attese di altro genere in un sistema operativo? È possibile evitare completamente l'attesa attiva? Spiegate le vostre risposte.
- 5.4** Spiegate perché gli spinlock non sono adatti ai sistemi monoprocesso, ma sono spesso usati nei sistemi multiprocesso.
- 5.5** Si dimostri che, se le operazioni `wait()` e `signal()` dei semafori non sono eseguite in modo atomico, la mutua esclusione rischia di essere violata.
- 5.6** Mostrate come si può utilizzare un semaforo binario per implementare la mutua esclusione tra n processi.

Esercizi

- 5.7** Le race condition sono possibili in diversi sistemi. Si consideri un sistema bancario che gestisce un conto corrente con due funzioni: `deposit(amount)` e `withdraw(amount)`. Le due funzioni ricevono in ingresso l'importo (*amount*) che deve essere depositato (*deposit*) o prelevato (*withdraw*) da un conto corrente bancario. Assumete che un conto corrente sia condiviso tra marito e moglie, e che in maniera concorrente il marito chiama la funzione `withdraw()` e la moglie la funzione `deposit()`. Descrivete com'è possibile il verificarsi di una race condition e che cosa dovrebbe essere fatto per evitarla.
- 5.8** L'algoritmo seguente, concepito da Dekker, è la prima soluzione software nota del problema della sezione critica per due processi. I due processi, P_0 e P_1 , condividono le seguenti variabili:

```
boolean flag[2]; /* inizialmente falsa* /
int turn;
```

La struttura del processo P_i ($i == 0$ oppure 1), dove P_j ($j == 1$ oppure 0) è l'altro processo, è mostrata nella Figura 5.21. Dimostrate che l'algoritmo soddisfa tutti e tre i requisiti per il problema della sezione critica.

```
do {
    flag[i] = true;

    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; /* non fa niente */
            flag[i] = true;
        }
    }

    /* sezione critica */

    turn = j;
    flag[i] = false;

    /* sezione non critica */
} while (true);
```

Figura 5.21 Struttura del processo P_i nell'algoritmo di Dekker.

- 5.9** La prima soluzione software nota del problema della sezione critica per n processi con un limite di $n - 1$ turni d'attesa è stata proposta da Eisenberg e McGuire. I processi condividono le seguenti variabili:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

Ciascun elemento di `flag` è inizialmente `idle` (inattivo); il valore iniziale di `turn` è irrilevante (compreso tra 0 e $n - 1$). La struttura del processo P_i è illustrata nella Figura 5.22. Dimostrate che tale algoritmo soddisfa tutti e tre i requisiti del problema della sezione critica.

- 5.10** Chiarite perché, per implementare le primitive di sincronizzazione, non è corretto disabilitare le interruzioni di un sistema monoprocesso se le primitive stesse sono destinate a programmi utenti.
- 5.11** Spiegate perché le interruzioni non costituiscono un metodo appropriato per implementare le primitive di sincronizzazione nei sistemi multiprocessore.
- 5.12** Nel kernel di Linux un processo non può trattenere uno spinlock mentre tenta di acquisire un semaforo. Giustificate l'esistenza di questa politica di gestione.
- 5.13** Descrivete due strutture dati di un kernel in cui possono verificarsi le cosiddette race condition. Assicuratevi di includere una descrizione della modalità in cui queste situazioni possono verificarsi.

```

do {
    while (true) {
        flag[i] = want_in;
        j = turn;

        while (j != i) {
            if (flag[j] != idle) {
                j = turn;
            } else
                j = (j + 1) % n;
        }

        flag[i] = in_cs;
        j = 0;

        while ((j < n) && (j == i || flag[j] != in_cs))
            j++;
        if ((j >= n) && (turn == i || flag[turn] == idle))
            break;
    }

    /* sezione critica */

    j = (turn + 1) % n;

    while (flag[j] == idle)
        j = (j + 1) % n;

    turn = j;
    flag[i] = idle;

    /* sezione non critica */
} while (true);

```

Figura 5.22 Struttura del processo P_i nell'algoritmo di Eisenberg e McGuire.

5.14 Descrivete come l'istruzione `compare_and_swap()` consenta di ottenere una mutua esclusione che soddisfa il requisito dell'attesa limitata.

5.15 Si consideri l'implementazione di un lock mutex con l'utilizzo di un'istruzione hardware atomica. Supponete che sia disponibile la seguente struttura per definire il lock mutex:

```

typedef struct {
    int available;
} lock;

```

(`available == 0`) indica che il lock è disponibile e un valore pari a 1 indica che il lock non è disponibile. Utilizzando questa struttura, illustrate come le seguenti funzioni possono essere implementate con l'utilizzo di `test_and_set()` e `compare_and_swap()`:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Assicuratevi di includere tutte le inizializzazioni necessarie.

5.16 L'implementazione dei lock mutex esposta nel Paragrafo 5.5 soffre di attesa attiva. Descrivete quali cambiamenti sono necessari per fare in modo che un processo in attesa di acquisire un lock mutex venga bloccato e messo in una coda di attesa fino a quando il lock diventa disponibile.

5.17 Si supponga che un sistema disponga di più core di elaborazione. Per ciascuno dei seguenti scenari, individuate il miglior meccanismo di lock, tra uno spinlock e un lock mutex in cui i processi in attesa restano sospesi fino a quando il lock diventa disponibile:

- Il lock viene trattenuto per tempi brevi.
- Il lock viene trattenuto per tempi lunghi.
- Un thread può essere sospeso mentre è in possesso del lock.

5.18 Si supponga che un cambio di contesto richieda un tempo T . Stabilite un limite superiore (in termini di T) per il possesso di uno spinlock, tale per cui se lo spinlock viene mantenuto per un tempo maggiore, un lock mutex (in cui i thread in attesa vengono sospesi) diventa un'alternativa migliore.

5.19 Un server web multithread desidera tenere traccia del numero di richieste servite (questo numero è noto come `hits`). Considerate le due seguenti strategie per prevenire una race condition sulla variabile `hits`. La prima strategia è quella di utilizzare un lock mutex per l'aggiornamento di `hits`:

```
int hits;  
mutex_lock hit_lock;  
  
hit_lock.acquire();  
hits++;  
hit_lock.release();
```

Una seconda strategia è quella di utilizzare un numero intero atomico:

```
atomic_t hits;  
atomic_inc(&hits);
```

Spiegate quale di queste due strategie sia più efficiente.

```

#define MAX_PROCESSES 255
int number_of_processes = 0;

/* l'implementazione della fork() chiama questa funzione */
int allocate_process() {
    int new_pid;

    if (number_of_processes == MAX_PROCESSES)
        return -1;
    else {
        /* alloca le risorse di processo necessarie */
        ++number_of_processes;

        return new_pid;
    }
}

/* l'implementazione della exit() chiama questa funzione */
void release_process() {
    /* libera le risorse di processo */
    --number_of_processes;
}

```

Figura 5.23 Allocazione e rilascio di processi.

5.20 Si consideri l'esempio di codice per l'assegnazione e il rilascio di processi mostrato nella Figura 5.23.

- Individuate la/le race condition.
- Supponete di avere un lock mutex chiamato `mutex` con le operazioni di `acquire()` e `release()`. Indicate il punto in cui inserire il meccanismo di lock per evitare la/le race condition.
- Possiamo sostituire la variabile intera

```
int number_of_processes = 0
```

con il numero intero atomico

```
atomic_t number_of_processes = 0
```

per prevenire la/le race condition?

5.21 I server possono essere progettati in modo da limitare il numero di connessioni aperte. In un dato momento, per esempio, un server può ammettere solo N connessioni socket per volta; dopo questo limite, il server non accetterà alcuna connessione entrante prima che una connessione esistente sia chiusa. Spiegate come il server possa usare i semafori per limitare il numero delle connessioni concorrenti.

- 5.22** Windows Vista fornisce uno strumento di sincronizzazione leggero chiamato **lock SRW** (*slim reader-writer*). Mentre la maggior parte delle implementazioni dei lock di lettura-scrittura favorisce la lettura o la scrittura, oppure ordina i thread in attesa utilizzando una politica FIFO, i lock SRW non favoriscono né le letture né le scritture e i thread in attesa non vengono ordinati utilizzando una politica FIFO. Spiegate i vantaggi di un tale strumento di sincronizzazione.
- 5.23** Mostrate come implementare le operazioni `wait()` e `signal()` dei semafori negli ambienti multiprocessore tramite l'istruzione `test_and_set()`. La soluzione dovrebbe comportare un'attesa attiva minima.
- 5.24** L'Esercizio 4.26 richiede che il thread padre attenda che il thread figlio termini la sua esecuzione prima di visualizzare i valori calcolati. Ipotizziamo di voler permettere che il thread padre acceda ai numeri di Fibonacci non appena questi vengono calcolati dal figlio, piuttosto che attendere che il figlio termini. Spiegate quali modifiche sono necessarie alla soluzione dell'esercizio. Implementate la soluzione modificata.
- 5.25** Dimostrate che monitor e semafori sono equivalenti in quanto consentono di risolvere gli stessi problemi di sincronizzazione.
- 5.26** Progettate un algoritmo per un monitor a buffer limitato in cui i buffer siano incorporati nel monitor stesso.
- 5.27** All'interno di un monitor, la mutua esclusione stretta fa sì che il monitor con memoria limitata dell'Esercizio 5.26 sia adatto soprattutto buffer piccoli.
- Spiegate perché questa affermazione è vera.
 - Progettate un nuovo schema idoneo a buffer grandi.
- 5.28** Discutete il tradeoff tra equità e throughput delle operazioni nel problema dei lettori-scrittori. Proponete un metodo per risolvere il problema dei lettori-scrittori senza che si determini attesa indefinita.
- 5.29** Come si differenzia l'operazione `signal()` dei monitor dalla corrispondente operazione dei semafori?
- 5.30** Ipotizziamo che l'istruzione `signal()` possa apparire solo per ultima in una procedura monitor. Proponete un modo per semplificare, in questa situazione, l'implementazione descritta nel Paragrafo 5.8.
- 5.31** Considerate un sistema formato dai processi P_1, P_2, \dots, P_n , aventi priorità distinte, rappresentate da numeri interi. Scrivete un monitor grazie al quale tre identiche stampanti siano assegnate a tali processi, stabilendo l'ordine di allocazione in base alle priorità.
- 5.32** Un file deve essere condiviso fra processi diversi, ognuno dei quali è identificato da un numero univoco. Al file possono accedere simultaneamente vari processi, a patto che osservino la seguente prescrizione: la somma degli identifi-

catori di tutti i processi che accedono al file deve essere minore di n . Scrivete un monitor per coordinare l’accesso al file.

5.33 Se un processo invoca `signal()` al verificarsi di una condizione all’interno di un monitor, esso può continuare la propria esecuzione, oppure cedere il controllo al processo che ha ricevuto il segnale. Come cambia la soluzione al precedente esercizio in queste due circostanze?

5.34 Supponete di sostituire le operazioni `wait()` e `signal()` dei monitor con un solo costrutto, `await(B)`, dove B è un’espressione booleana generale, che obbliga il processo che lo esegue ad attendere finché B diventi vera.

- a. Si metta a punto, in base a questo modello, un monitor che affronti il problema dei lettori-scrittori.
- b. Chiarite che cosa impedisce di implementare efficientemente questo costrutto.
- c. Quali restrizioni è necessario imporre all’istruzione `await(B)` perché possa essere implementata efficientemente? (Suggerimento: limitate la forma di B ; si veda [Kessels 1977].)

5.35 Scrivete un monitor per realizzare una “sveglia” con cui un programma chiamante possa differire la propria esecuzione per il numero prescelto di unità di tempo (“battiti o tic”). Si può ipotizzare l’esistenza di un orologio hardware che invochi una procedura `battito()` sul vostro monitor a intervalli regolari.

Problemi di programmazione

5.36 Nell’Esercizio di programmazione 3.20 veniva richiesto di progettare un gestore di PID che assegnava un identificatore unico a ogni processo. Nell’Esercizio 4.20 veniva richiesto di modificare la soluzione all’Esercizio 3.20, scrivendo un programma che creava un certo numero di thread che richiedevano e rilasciavano identificatori di processo. Modificate ora la soluzione all’Esercizio 4.20, garantendo che la struttura dati utilizzata per rappresentare la disponibilità dei PID sia al sicuro da race condition. Utilizzate i lock mutex Pthreads descritti nel Paragrafo 5.9.4.

5.37 Assumete di dover gestire un numero finito di risorse, appartenenti tutte allo stesso tipo. I processi possono richiedere una parte di queste risorse, per poi restituirle quando hanno terminato. Un esempio proviene dai programmi commerciali, molti dei quali includono un certo numero di **licenze**, che indicano il numero di applicazioni concorrentemente eseguibili. All’avvio di un’applicazione, il totale delle licenze a disposizione diminuisce. Se un’applicazione termina, il totale delle licenze aumenta. Quando tutte le licenze sono in uso, le nuove richieste per l’avvio di un’applicazione non avranno esito. Si ottempera a tali richieste solo nel momento in cui il proprietario di una licenza lascia libera un’applicazione e restituisce la relativa licenza.

Il seguente frammento di un programma serve per la gestione di un numero finito di istanze di una risorsa disponibile. Il numero massimo di risorse e quello delle risorse disponibili vengono dichiarati come segue:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

Quando un processo mira a ottenere alcune risorse, invoca la funzione `decrease_count()`:

```
/* decrementa il numero di risorse disponibili */
/* di una quantità pari a count; restituisce 0 */
/* se vi sono risorse sufficienti, -1 altrimenti */
int decrease_count(int_count) {
    if (available_resources < count)
        return -1;
    else {
        available_resources -= count;
        return 0;
    }
}
```

Quando un processo intende restituire le risorse che possiede, chiama la funzione `increase_count()`:

```
/* incrementa il numero di risorse disponibili */
/* di una quantità pari a count*/
int increase_count(int_count) {
    available_resources += count;
    return 0;
}
```

Il programma precedente provoca una *race condition*, per ovviare alla quale dovete:

- a) individuare i dati interessati dal conflitto;
- b) identificare, nel codice, il punto (o i punti) che danno luogo al conflitto;
- c) adoperare un semaforo o un lock mutex eliminare il conflitto. È accettabile modificare la funzione `decrease_count()` in modo da bloccare il processo richiedente fino a che sono disponibili risorse sufficienti.

5.38 La funzione `decrease_count()` dell'esercizio precedente restituisce 0 qualora vi siano sufficienti risorse a disposizione e -1 in caso contrario. Questo

complica la programmazione di un processo che intenda richiedere un certo numero di risorse:

```
while (decrease_count(count) == -1)
;
```

Riscrivete il segmento di codice dedicato alla gestione delle risorse, servendovi di un monitor e di variabili condizionali, in modo che la funzione `decrease_count()` sospenda il processo finché non siano disponibili risorse sufficienti. Ciò consentirà a un processo di invocare `decrease_count()` semplicemente così:

```
decrease_count(count);
```

Il processo rientra da questa chiamata solo quando vi siano risorse sufficienti disponibili.

5.39 Nell’Esercizio 4.22 veniva richiesto di scrivere un programma multithread per stimare il valore di π utilizzando la tecnica di Monte Carlo. Vi era richiesto, in particolare, di creare un singolo thread per generare punti casuali, memorizzando il risultato in una variabile globale. Una volta che il thread terminava, il thread genitore eseguiva i calcoli necessari per la stima di π . Modificate il programma in modo da creare più thread, ognuno dei quali genera punti casuali e determina se i punti cadono all’interno del cerchio. Ogni thread dovrà aggiornare il conteggio globale del numero di punti che cadono all’interno del cerchio. Proteggete da race condition gli aggiornamenti alla variabile globale condivisa utilizzando lock mutex.

5.40 L’Esercizio 4.23 vi chiedeva di progettare un programma per la stima di π con la tecnica di Monte Carlo utilizzando OpenMP. Esaminate la vostra soluzione in cerca di possibili race condition. Se si individua una race condition, proteggete il vostro programma utilizzando la strategia descritta nel Paragrafo 5.10.2.

5.41 Una **barriera** è uno strumento per sincronizzare l’attività di un certo numero di thread. Quando un thread raggiunge un punto di barriera non può procedere fino a quando anche tutti gli altri thread hanno raggiunto lo stesso punto. Quando l’ultimo thread raggiunge il punto di barriera, tutti i thread vengono rilasciati e possono riprendere l’esecuzione concorrente.

Si supponga che la barriera sia inizializzata a N (numero di thread che devono attendere al punto di barriera):

```
init(N);
```

Ogni thread esegue un po’ di lavoro, fino a raggiungere il punto di barriera:

```
/* esegui un po' di lavoro */
barrier_point();
/* esegui un po' di lavoro */
```

Utilizzando gli strumenti di sincronizzazione descritti in questo capitolo, costruite una barriera che implementa la seguente API:

- `int init(int n)` – inizializza la barriera alla dimensione specificata.
- `int barrier_point(void)` – identifica il punto di barriera. Tutti i thread sono rilasciati dalla barriera quando l’ultimo thread raggiunge questo punto.

Il valore restituito da ogni funzione viene utilizzato per individuare le condizioni di errore. Ogni funzione restituisce 0 in condizioni normali di funzionamento e -1 in caso di errore. Tra il codice sorgente da scaricare vi è uno strumento di test per verificare la vostra implementazione della barriera.

Progetti di programmazione

Progetto 1 – L’assistente che dorme

Un dipartimento di informatica ha un assistente (TA) che aiuta gli studenti universitari nei loro esercizi di programmazione, in orario d’ufficio. L’ufficio del TA è piuttosto piccolo e ha spazio solo per una scrivania con una sedia e per un computer. Ci sono tre sedie in corridoio, davanti all’ufficio, dove gli studenti possono sedersi e aspettare quando il TA è impegnato nell’aiuto a un altro studente. Quando non ci sono studenti che hanno bisogno di aiuto, il TA si siede alla scrivania e fa un pisolino. Se uno studente arriva in orario d’ufficio e trova il TA che dorme, deve sveglierlo per chiedergli aiuto. Se uno studente arriva e trova il TA che sta aiutando un altro studente, si siede su una delle sedie in corridoio e aspetta. Se non ci sono sedie disponibili, lo studente dovrà tornare in un secondo momento.

Utilizzando i thread POSIX, i lock mutex e i semafori implementate una soluzione per coordinare le attività del TA e degli studenti. Riportiamo di seguito i dettagli di questo progetto.

Gli studenti e il TA

Utilizzando Pthread (Paragrafo 4.4.1), iniziate dalla creazione di n studenti. Ogni studente e il TA verranno eseguiti come thread distinti. I thread degli studenti alterneranno un periodo di tempo di programmazione alla ricerca di aiuto dal TA. Se il TA è disponibile, otterranno aiuto. In caso contrario, si siederanno su una sedia nel corridoio o, se non ci sono sedie disponibili, riprenderanno l’attività di programmazione e cercheranno aiuto in un secondo momento. Se uno studente arriva e si accorge che il TA sta dormendo, deve notificare il suo arrivo al TA con un semaforo. Quando il TA finisce di aiutare uno studente deve controllare in corridoio per vedere se ci sono studenti in attesa di aiuto. In caso affermativo, il TA deve aiutare, a turno, tutti gli studenti. Se non sono presenti studenti, il TA può tornare a dormire.

Probabilmente l’opzione migliore per simulare gli studenti che programmano e il TA che fornisce aiuto a uno studente è di sospendere i thread coinvolti per un periodo di tempo casuale.

La sincronizzazione POSIX

I lock mutex POSIX e i semafori sono trattati nel Paragrafo 5.9.4. Consultate tale paragrafo per maggiori dettagli.

Progetto 2 – Il problema dei cinque filosofi (dining philosophers)

Nel Paragrafo 5.7.3 abbiamo fornito uno schema di soluzione al problema dei dining philosophers con l'utilizzo dei monitor. Questo progetto richiede l'implementazione di una soluzione usando i lock mutex Pthreads e le variabili condizionali.

I filosofi

Iniziate dalla creazione dei cinque filosofi, ciascuno identificato da un numero tra 0 e 4. Ogni filosofo verrà eseguito come un thread separato. La creazione di thread con l'uso di Pthreads è trattata nel Paragrafo 4.4.1. I filosofi alternano il pensare al mangiare. Per simulare entrambe le attività, sospendete i thread per un periodo di tempo casuale compreso tra uno e tre secondi. Quando un filosofo vuole mangiare, invoca la funzione

```
pickup_forks (int philosopher_number)
```

dove `philosopher_number` identifica il numero del filosofo che desidera mangiare. Quando un filosofo finisce di mangiare, invoca

```
return_forks (int philosopher_number)
```

Variabili condizionali Pthreads

Le variabili condizionali in Pthreads si comportano in modo simile a quelle descritte nel Paragrafo 5.8. Tuttavia, in tale paragrafo, le variabili condizionali sono utilizzate nel contesto di un monitor, che fornisce un meccanismo di locking per assicurare l'integrità dei dati. Visto che Pthreads è tipicamente usato nei programmi C e che il C non ha un monitor, realizziamo il lock associando una variabile condizionale a un lock mutex. I lock mutex Pthreads sono trattati nel Paragrafo 5.9.4, trattiamo qui le variabili condizionali Pthreads.

Le variabili condizionali in Pthreads usano il tipo di dato `pthread_cond_t` e vengono inizializzate mediante la funzione `pthread_cond_init()`. Il codice seguente crea e inizializza una variabile condizionale e il lock mutex a essa associato:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

Per l'attesa su una variabile condizionale viene usata la funzione:

```
pthread_cond_wait()
```

Il seguente codice illustra come un thread può aspettare il verificarsi della condizione `a == b` utilizzando una variabile condizionale Pthreads:

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait (&mutex, &cond_var);
pthread_mutex_unlock (&mutex);
```

Il lock mutex associato alla variabile condizionale deve essere bloccato prima della chiamata `pthread_cond_wait()`, in quanto viene utilizzato per proteggere i dati nella clausola condizionale da una possibile race condition. Una volta acquisito il lock, il thread può verificare la condizione. Se la condizione non è vera, il thread richiama `pthread_cond_wait()`, passando il lock mutex e la variabile condizionale come parametri. La chiamata a `pthread_cond_wait()` rilascia il lock mutex, consentendo in tal modo a un altro thread di accedere al dato condiviso ed eventualmente aggiornare il suo valore in modo che la clausola condizionale restituisca true. (Per proteggersi contro eventuali errori, è importante collocare la clausola condizionale all'interno di un ciclo in modo che la condizione sia ricontrollata dopo essere stata segnalata.)

Un thread che modifica i dati condivisi può richiamare la funzione `pthread_cond_signal()`, in modo da mandare un segnale a un thread in attesa sulla variabile condizionale, come mostrato di seguito:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

È importante notare che la chiamata a `pthread_cond_signal()` non rilascia il lock mutex, ma lo fa la chiamata successiva a `pthread_mutex_unlock()`. Una volta che il lock mutex viene rilasciato, il thread che ha ricevuto il segnale diventa proprietario del lock e il controllo riprende dalla chiamata alla `pthread_cond_wait()`.

Progetto 3 - Il problema produttore/consumatore

Nel Paragrafo 5.7.1 abbiamo proposto una soluzione al problema produttore/consumatore che si basa su semafori e si avvale di un buffer limitato. In questo progetto daremo soluzione al problema del buffer limitato mediante i processi produttore e consumatore schematizzati nelle Figure 5.9 e 5.10, rispettivamente. La soluzione presentata nel Paragrafo 5.7.1 utilizza tre semafori: vuote e piene, che enumerano le posizioni vuote e piene del buffer, e mutex, un semaforo binario, o a mutua esclusione, che protegge l'inserimento nel (o l'estrazione dal) buffer. In questo progetto i semafori vuote e piene saranno semafori contatori standard, mentre, per rappresentare mutex, si userà un lock mutex in luogo di un semaforo binario. Il produttore e il consumatore – eseguiti come thread separati – trasferiscono gli oggetti a un buffer, oppure li prelevano da esso; il buffer è sincronizzato con le strutture vuote, piene e mutex. Potete scegliere di risolvere il problema con Pthreads o con la API Windows.

```

#include "buffer.h"
/* il buffer */
buffer_item buffer[BUFFER_SIZE];
int insert_item(buffer_item item) {
    /* inserisce un elemento nel buffer
       restituisce 0 se ha successo,
       altrimenti -1 per segnalare l'errore */
}
int remove_item(buffer_item *item) {
    /* estraе un elemento dal buffer
       e lo pone in item
       restituisce 0 se ha successo,
       altrimenti -1 per segnalare l'errore */
}

```

Figura 5.24 Schema delle operazioni sul buffer.

Buffer

Internamente, il buffer è costituito da un array di dimensione fissa avente tipo `buffer_item`, definito tramite `typedef`. L'array di oggetti `buffer_item` sarà trattato come una coda circolare. Definizione e dimensione relative possono essere poste in un file di intestazione come il seguente:

```

/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5

```

Il buffer sarà manipolato da due funzioni, `insert_item()` e `remove_item()`, richiamate, rispettivamente, dal thread del produttore e dal thread del consumatore. Tali funzioni, in un primo abbozzo schematico, appaiono nella Figura 5.24.

Le funzioni `insert_item()` e `remove_item()` sincronizzeranno il produttore e il consumatore usando gli algoritmi delineati nelle Figure 5.9 e 5.10. Il buffer richiederà, inoltre, una funzione per inizializzare mutex, come pure i semafori vuote e piene.

La funzione `main()` inizializzerà il buffer, creando due thread distinti per il produttore e il consumatore. Dopo aver creato tali thread, la funzione `main()` rimarrà inattiva per un certo tempo e, alla ripresa dell'attività, terminerà l'applicazione. La funzione `main()` riceverà tre parametri dalla riga di comando, indicanti:

1. il tempo di inattività prima di terminare;
2. il numero di thread produttori;
3. il numero di thread consumatori.

La struttura di questa funzione è illustrata nella Figura 5.25.

```
#include "buffer.h"
int main(int argc, char *argv[]) {
    /* 1. Ottiene i parametri argv[0], argv[1], argv[2]
        dalla riga di comando */
    /* 2. Inizializza il buffer */
    /* 3. Crea i thread di tipo produttore */
    /* 4. Crea i thread di tipo consumatore */
    /* 5. Attende */
    /* 6. Termina */
```

Figura 5.25 Scheletro del programma.

Thread produttori e consumatori

Il thread produttore alternerà periodi di tempo di inattività casuali all'inserimento di un intero casuale nel buffer. I numeri casuali saranno generati tramite la funzione `rand()`, che dà luogo a valori interi casuali compresi tra 0 e `RAND_MAX`. Anche il consumatore rimarrà in stato di d'inattività per un intervallo di tempo casuale; tornato attivo, tenterà di estrarre un oggetto dal buffer. Lo schema dei thread produttori e consumatori è illustrato nella Figura 5.26.

Come osservato in precedenza, è possibile risolvere questo problema utilizzando sia Pthreads sia l'API di Windows. Nei paragrafi seguenti forniamo ulteriori informazioni su ciascuna di queste scelte.

Creazione e sincronizzazione dei thread in Pthreads

La creazione di thread utilizzando l'API Pthreads è trattata nel Paragrafo 4.4.1. I lock mutex e i semafori in Pthreads sono introdotti nel Paragrafo 5.9.4. Fare riferimento a questi paragrafi per informazioni specifiche sulla creazione e sulla sincronizzazione dei thread in Pthreads.

Lock mutex di Windows

I lock mutex sono oggetti dispatcher, come si è visto nel Paragrafo 5.9.1. Di seguito si può osservare come creare un lock mutex utilizzando la funzione `CreateMutex()`:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

Il primo parametro si riferisce a un attributo di sicurezza per il lock mutex. Impostando a `NULL` questo attributo, i processi figli del creatore del lock mutex non ereditano il riferimento al mutex stesso. Il secondo parametro indica se il processo che ha creato il lock mutex sia il possessore iniziale del lock: passando il valore `FALSE`, si asserisce che il thread non è il possessore iniziale; vedremo fra breve come acquisire i lock

```

#include <stdlib.h> /* necessario per rand() */
#include "buffer.h"

void *producer(void *param) {
    buffer_item item;

    while (true) {
        /* resta inattivo per un intervallo di tempo casuale */
        sleep(...);
        /* genera un numero casuale */
        item = rand();
        if (insert_item(item))
            fprintf("report error condition");
        else
            printf("producer produced %d\n",item);
    }
}

void *consumer(void *param) {
    buffer_item item;

    while (true) {
        /* resta inattivo per un intervallo di tempo casuale */
        sleep(...);
        if (remove_item(&item))
            fprintf("report error condition");
        else
            printf("consumer consumed %d\n",item);
    }
}

```

Figura 5.26 Schema dei thread dei produttori e consumatori.

mutex. Il terzo parametro consente di attribuire un nome al mutex. Tuttavia, poiché si è scelto un valore NULL, non gli attribuiremo un nome. Se `CreateMutex()` ha avuto successo, restituirà il riferimento HANDLE al lock; altrimenti, NULL.

Nel Paragrafo 5.9.1 abbiamo descritto i due possibili stati, detti *signaled* e *nonsignaled*, degli oggetti dispatcher. Quando un oggetto dispatcher (per esempio un lock mutex) è nello stato *signaled* se ne può entrare in possesso; una volta acquisito passa allo stato *nonsignaled*. L'oggetto diviene *signaled* non appena risulta nuovamente disponibile.

I lock mutex si acquisiscono richiamando la funzione `WaitForSingleObject()` con due parametri: il riferimento HANDLE al lock, e un flag che indica la durata dell'attesa. Il codice mostra come si può acquisire il lock mutex creato in precedenza.

```
WaitForSingleObject(Mutex, INFINITE);
```

Il valore `INFINITE` significa che non vi è limite a quanto si è disposti ad attendere che il lock diventi disponibile. Con altri valori il thread chiamante potrebbe troncare

l'operazione, qualora il lock non tornasse disponibile entro un tempo definito. Se il lock è in stato *signaled*, `WaitForSingleObject()` restituisce immediatamente il controllo, e il lock diviene *nonsignaled*. Per rendere disponibile un lock (ossia, per far sì che entri nello stato *signaled*) si invoca `ReleaseMutex()`:

```
ReleaseMutex(Mutex);
```

Semafori di Windows

Nella API di Windows anche i semafori sono oggetti dispatcher e pertanto adoperano lo stesso meccanismo di segnalazione dei lock mutex. I semafori si creano tramite il codice:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

In analogia a quanto descritto per i lock mutex, il primo e l'ultimo parametro designano un attributo di sicurezza e un nome per il semaforo. Il secondo e il terzo parametro indicano il valore iniziale e il valore massimo del semaforo. In questo caso, il valore iniziale del semaforo è 1, mentre il suo valore massimo è 5. Se ha successo, `CreateSemaphore()` restituisce un riferimento HANDLE al semaforo; in caso contrario, restituisce NULL.

I semafori si acquisiscono con la stessa funzione trattata per i lock mutex, cioè `WaitForSingleObject()`. Per acquisire il semaforo Sem di questo esempio, si invoca:

```
WaitForSingleObject(Semaphore, INFINITE);
```

Se il valore del semaforo è > 0 , esso si trova nello stato *signaled*, e viene quindi acquisito dal thread chiamante; altrimenti, il thread chiamante è sospeso (per un tempo imprecisato, a causa del parametro `INFINITE`) in attesa che il semaforo ritorni nello stato *signaled*.

Per i semafori di Windows, l'equivalente dell'operazione `signal()` è la funzione `ReleaseSemaphore()`. Essa accetta tre parametri:

1. il riferimento HANDLE al semaforo;
2. il valore di cui incrementare il semaforo;
3. un puntatore al valore precedente del semaforo.

Si può aumentare `Sem` di 1 nel modo seguente:

```
ReleaseSemaphore(Sem, 1, NULL);
```

`ReleaseSemaphore()` e `ReleaseMutex()` restituiscono 0 se eseguite senza errori e un valore non nullo in caso contrario.

Note bibliografiche

Il problema della mutua esclusione fu discusso per la prima volta in un classico lavoro di [Dijkstra 1965]. L'algoritmo di Dekker (Esercizio 5.8), la prima soluzione software corretta al problema della mutua esclusione per due processi, fu sviluppato dal matematico olandese T. Dekker. L'algoritmo è anche esaminato in [Dijkstra 1965]; una soluzione più semplice al problema della mutua esclusione per due processi si trova in [Peterson 1981] (Figura 5.2). Il concetto di semaforo è stato proposto da [Dijkstra 1965].

I classici problemi del coordinamento dei processi presentati in questo capitolo fungono da paradigma per un'ampia classe di problemi di controllo della concorrenza. Il problema del buffer limitato e dei cinque filosofi sono proposti in [Dijkstra 1965a] e [Dijkstra 1971]; quello dei lettori-scrittori è dovuto a [Courtois et al. 1971].

Il concetto di sezione critica è dovuto a [Hoare 1972] e [Brinch-Hansen 1972]. Il concetto di monitor è sviluppato da [Brinch-Hansen 1973], e una sua completa descrizione si trova in [Hoare 1974].

Alcuni dettagli sui lock di Solaris sono presentati da [Mauro e McDougall 2007]. Come sottolineato in precedenza, i lock usati dal kernel sono implementati anche a beneficio dei thread utenti: lo stesso tipo di lock è disponibile all'interno e all'esterno del kernel. I dettagli sulla sincronizzazione in Windows 2000 si trovano in [Solomon e Russinovich 2000]. [Love 2010] descrive la sincronizzazione nel kernel di Linux.

Informazioni sulla programmazione Pthreads possono essere trovate in [Lewis e Berg 1998] e [Butenhof 1997]. [Hart 2005] descrive la sincronizzazione dei thread utilizzando Windows. [Goetz et al. 2006] presenta una trattazione dettagliata della programmazione concorrente in Java e del pacchetto `java.util.concurrent`. [Breshears 2009] e [Pacheco 2011] trattano in dettaglio le problematiche di sincronizzazione in relazione alla programmazione parallela. [Lu et al. 2008] fornisce uno studio dei bug nell'utilizzo della concorrenza in applicazioni reali.

[Adl-Tabatabai et al. (2007)] tratta il tema della memoria transazionale. I dettagli sull'uso di OpenMP sono disponibili all'indirizzo <http://openmp.org>. La programmazione funzionale con Erlang e Scala è trattata in [Armstrong 2007] e [Odersky et al.], rispettivamente.

Bibliografia

- [Adl-Tabatabai et al. 2007] A.-R. Adl-Tabatabai, C. Kozyrakis e B. Saha, “Unlocking Concurrency”, *Queue*, Volume 4, N. 10, p. 24–33, 2007.
- [Armstrong 2007] J. Armstrong, *Programming Erlang Software for a Concurrent World*, The Pragmatic Bookshelf, 2007.
- [Breshears 2009] C. Breshears, *The Art of Concurrency*, O'Reilly & Associates, 2009.
- [Brinch-Hansen 1972] P. Brinch-Hansen, “Structured Multiprogramming”, *Communications of the ACM*, Volume 15, N. 7, p. 574–578, 1972

- [**Brinch-Hansen 1973**] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall, 1973.
- [**Butenhof 1997**] D. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.
- [**Courtois et al. 1971**] P. J.Courtois, F.Heymans e D.L.Parnas, “Concurrent Control with ‘Readers’ and ‘Writers’”, *Communications of the ACM*, Volume 14, N. 10, p. 667–668, 1971.
- [**Dijkstra 1965**] E. W. Dijkstra, “Cooperating Sequential Processes”, Technical report, Technological University, Eindhoven, the Netherlands, 1965.
- [**Dijkstra 1971**] E.W.Dijkstra, “Hierarchical Ordering of Sequential Processes”, *Acta Informatica*, Volume 1, N. 2, p. 115–138, 1971.
- [**Goetz et al. 2006**] B. Goetz, T. Peirls, J. Bloch, J. Bowbeer, D. Holmes e D. Lea, *Java Concurrency in Practice*, Addison-Wesley, 2006.
- [**Hart 2005**] J. M. Hart, *Windows System Programming*, Third Edition, Addison-Wesley, 2005.
- [**Hoare 1972**] C. A. R. Hoare, “Towards a Theory of Parallel Programming”, in [Hoare e Perrott 1972], p. 61–71, 1972.
- [**Hoare 1974**] C. A. R. Hoare, “Monitors: An Operating System Structuring Concept”, *Communications of the ACM*, Volume 17, N. 10, p. 549–557, 1974.
- [**Kessels 1977**] J. L. W. Kessels, “An Alternative to Event Queues for Synchronization in Monitors”, *Communications of the ACM*, Volume 20, N. 7, p. 500–503, 1977.
- [**Lewis e Berg 1998**] B. Lewis e D. Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press, 1998.
- [**Love 2010**] R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library, 2010.
- [**Lu et al. 2008**] S. Lu, S. Park, E. Seo e Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics”, *SIGPLAN Notices*, Volume 43, N. 3, p. 329–339, 2008.
- [**Mauro e McDougall 2007**] J. Mauro e R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall, 2007.
- [**Odersky et al.**] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. Mcdirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon e M. Zenger.
- [**Pacheco 2011**] P. S. Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011.
- [**Peterson 1981**] G. L. Peterson, “Myths About the Mutual Exclusion Problem”, *Information Processing Letters*, Volume 12, N. 3, 1981.
- [**Solomon e Russinovich 2000**] D. A. Solomon e M. E. Russinovich, *Inside Microsoft Windows 2000*, 3° ed., Microsoft Press, 2000.

6

**OBIETTIVI
DEL CAPITOLO**

- Introduzione allo scheduling della CPU, base dei sistemi operativi multiprogrammati.
- Descrizione di vari algoritmi di scheduling della CPU.
- Criteri di valutazione nella scelta degli algoritmi di scheduling della CPU per particolari sistemi.
- Studio degli algoritmi di scheduling di diversi sistemi operativi.

Scheduling della CPU

Lo scheduling della CPU è alla base dei sistemi operativi multiprogrammati: attraverso la commutazione della CPU tra i vari processi, il sistema operativo può rendere più produttivo il calcolatore. In questo capitolo s'introducono i concetti fondamentali dello scheduling e si descrivono vari algoritmi di scheduling della CPU. Si affronta inoltre il problema della scelta dell'algoritmo da impiegare per un dato sistema.

Nel Capitolo 4 abbiamo arricchito il concetto di processo introducendo i thread. Nei sistemi operativi che li supportano, in effetti sono i thread, e non i processi, l'oggetto dell'attività di scheduling. Ciononostante, le locuzioni **scheduling dei processi** e **scheduling dei thread** sono spesso considerate equivalenti. In questo capitolo useremo la prima nell'analizzare i principi generali dello scheduling, e la seconda nel trattare idee specificamente inerenti ai thread.

6.1 Concetti fondamentali

In un sistema monoprocesso si può eseguire al massimo un processo alla volta; gli altri processi, se ci sono, devono attendere che la CPU sia libera e possa essere nuovamente sottoposta a scheduling. L'obiettivo della multiprogrammazione è avere sempre un processo in esecuzione, in modo da massimizzare l'utilizzazione della CPU.

L'idea è relativamente semplice. Un processo è in esecuzione finché non deve attendere un evento, generalmente il completamento di qualche richiesta di I/O; durante l'attesa, in un sistema di calcolo semplice, la CPU resterebbe inattiva, e tutto il tempo d'attesa sarebbe sprecato. Con la multiprogrammazione si cerca d'impiegare questi tempi d'attesa in modo produttivo: si tengono contemporaneamente più processi in memoria, e quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della CPU per cederlo a un altro processo.

Lo scheduling è una funzione fondamentale dei sistemi operativi; si sottopongono a scheduling quasi tutte le risorse di un calcolatore. Naturalmente la CPU è una delle risorse principali, e il suo scheduling è alla base della progettazione dei sistemi operativi.

6.1.1 Ciclicità delle fasi d'elaborazione e di I/O

Il successo dello scheduling della CPU dipende dall'osservazione della seguente proprietà dei processi: l'esecuzione del processo consiste in un **ciclo** d'elaborazione (svolta dalla CPU) e d'attesa del completamento delle operazioni di I/O. I processi si alternano tra questi due stati. L'esecuzione di un processo comincia con una sequenza di operazioni d'elaborazione svolte dalla CPU (*CPU burst*), seguita da una sequenza di operazioni di I/O (*I/O burst*), quindi un'altra sequenza di operazioni della CPU, di nuovo una sequenza di operazioni di I/O, e così via. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione (Figura 6.1).

Le durate delle sequenze di operazioni della CPU (dette anche “burst della CPU”) sono state misurate in molte sperimentazioni, e sebbene varino molto secondo il processo e secondo il calcolatore, la loro curva di frequenza è simile a quella illustrata nella Figura 6.2. La curva è generalmente di tipo esponenziale o iperesponenziale, con molte brevi sequenze di operazioni della CPU, e poche sequenze di operazioni della CPU molto lunghe. Un programma con prevalenza di I/O (*I/O bound*) produce generalmente molte sequenze di operazioni della CPU di breve durata. Un programma con prevalenza d'elaborazione (*CPU bound*), invece, può produrre varie sequenze di operazioni della CPU molto lunghe. Questa distribuzione può essere utile nella scelta di un appropriato algoritmo di scheduling della CPU.

6.1.2 Scheduler della CPU

Ogniqualvolta la CPU passa nello stato d'inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella ready queue. In particolare, è lo **scheduler a breve termine**, o scheduler della CPU che, tra i processi in memoria pronti per l'esecuzione, sceglie quello cui assegnare la CPU.

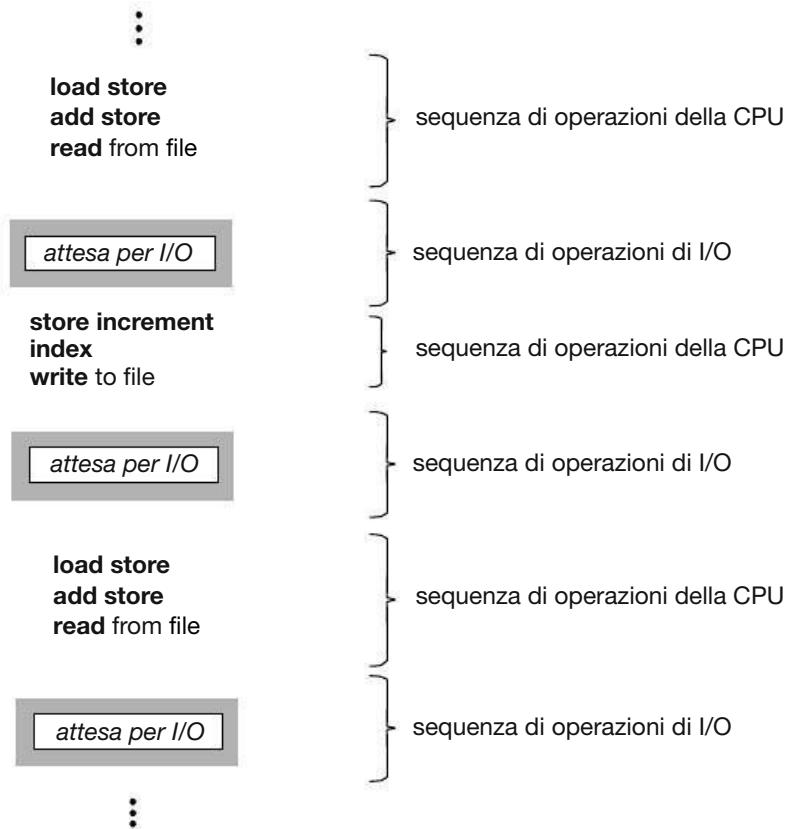


Figura 6.1 Serie alternata di sequenze di operazioni della CPU e di sequenze di operazioni di I/O.



Figura 6.2 Diagramma delle durate delle sequenze di operazioni della CPU.

La ready queue non è necessariamente una coda in ordine d'arrivo (*first-in, first-out* o FIFO). Come vedremo analizzando i diversi algoritmi di scheduling, una ready queue si può realizzare come una coda FIFO, una coda con priorità, un albero o semplicemente una lista concatenata non ordinata. Tuttavia, concettualmente tutti i pro-

cessi della ready queue sono posti nella lista d'attesa per accedere alla CPU. Generalmente gli elementi delle code sono i *process control block* (PCB) dei processi.

6.1.3 Scheduling con prelazione

Le decisioni riguardanti lo scheduling della CPU vengono prese nelle seguenti quattro circostanze:

1. un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di I/O o invocazione di `wait()` per la terminazione di uno dei processi figli);
2. un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale d'interruzione);
3. un processo passa dallo stato di attesa allo stato pronto (per esempio, al completamento di un'operazione di I/O);
4. un processo termina.

I casi 1 e 4 non danno alternative in termini di scheduling: si deve comunque scegliere un nuovo processo da eseguire, sempre che ce ne sia almeno uno nella ready queue per l'esecuzione. Una scelta si può invece fare nei casi 2 e 3.

Quando lo scheduling interviene solo nelle condizioni 1 e 4, si dice che lo schema di scheduling è **senza prelazione** (*nonpreemptive*) o **cooperativo** (*cooperative*); altrimenti, lo schema di scheduling è **con prelazione** (*preemptive*). Nel caso dello scheduling senza prelazione, quando si assegna la CPU a un processo, questo rimane in possesso della CPU fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al passaggio nello stato di attesa. Questo metodo di scheduling è stato adottato da Microsoft Windows 3.x; lo scheduling con prelazione è stato introdotto a partire da Windows 95 e in tutte le versioni successive. È adottato anche dal sistema Mac OS X; le versioni precedenti del sistema operativo Macintosh si basavano sullo scheduling cooperativo. Poiché, diversamente dallo scheduling con prelazione, non richiede dispositivi particolari (per esempio i timer), lo scheduling cooperativo è l'unico metodo utilizzabile su alcune piattaforme.

Sfortunatamente lo scheduling con prelazione può portare a race condition quando i dati sono condivisi tra diversi processi. Si consideri il caso in cui due processi condividono dati; mentre uno di questi aggiorna i dati si ha la sua prelazione in favore dell'esecuzione dell'altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato incoerente dal primo processo. Tali questioni sono state trattate in dettaglio nel Capitolo 5.

La capacità di prelazione si ripercuote anche sulla progettazione del kernel del sistema operativo. Durante l'elaborazione di una chiamata di sistema, il kernel può essere impegnato in attività per conto di un processo; tali attività possono comportare la necessità di modifiche a importanti dati del kernel, come le code di I/O. Se si ha la prelazione del processo durante tali modifiche e il kernel (o un driver di dispositivo) deve leggere o modificare gli stessi dati, si può avere il caos. Alcuni sistemi operativi, tra cui la maggior parte delle versioni dello UNIX, affrontano questo problema atten-

dendo il completamento della chiamata di sistema o un blocco dell'I/O prima di eseguire un cambio di contesto. Ciò garantisce che la struttura del kernel sia semplice, dato che il kernel non può esercitare la prelazione su un processo mentre le strutture dati del kernel si trovano in uno stato incoerente. Sfortunatamente, questo modello d'esecuzione del kernel non è adeguato alle computazioni in tempo reale, in cui i task devono essere conclusi entro un intervallo fissato di tempo. I requisiti di scheduling per sistemi real-time sono descritti nel Paragrafo 6.6.

Poiché le interruzioni si possono, per definizione, verificare in ogni istante e il kernel non può sempre ignorare, le sezioni di codice eseguite per effetto delle interruzioni devono essere protette da un uso simultaneo. Il sistema operativo deve ignorare raramente le interruzioni, altrimenti si potrebbero perdere dati in ingresso, o si potrebbero sovrascrivere dati in uscita. Per evitare che più processi accedano in modo concorrente a tali sezioni di codice, queste disattivano le interruzioni al loro inizio e le riattivano alla fine. Le sezioni di codice che disabilitano le interruzioni si verificano tuttavia raramente e, in genere, non contengono molte istruzioni.

6.1.4 Dispatcher

Un altro elemento coinvolto nella funzione di scheduling della CPU è il **dispatcher**; si tratta del modulo che passa effettivamente il controllo della CPU al processo scelto dallo scheduler a breve termine. Questa funzione comprende:

- il cambio di contesto;
- il passaggio alla modalità utente;
- il salto alla giusta posizione del programma utente per riavviare l'esecuzione.

Poiché si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere quanto più rapido è possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è noto come **latenza di dispatch**.

6.2 Criteri di scheduling

Diversi algoritmi di scheduling della CPU hanno proprietà differenti e possono favorire una particolare classe di processi. Prima di scegliere l'algoritmo da usare in una specifica situazione occorre considerare le caratteristiche dei diversi algoritmi.

Per il confronto tra gli algoritmi di scheduling della CPU sono stati suggeriti molti criteri. Le caratteristiche usate per tale confronto possono incidere in modo rilevante sulla scelta dell'algoritmo migliore. Di seguito si riportano alcuni criteri.

- **Utilizzo della CPU.** La CPU deve essere più attiva possibile. Teoricamente, l'utilizzo della CPU può variare dallo 0 al 100 per cento. In un sistema reale dovrebbe variare dal 40 per cento, per un sistema con poco carico, al 90 per cento, per un sistema con utilizzo intenso.

- **Produttività.** La CPU è attiva quando si svolge del lavoro. Una misura del lavoro svolto è data dal numero dei processi completati nell'unità di tempo: tale misura è detta **produttività** (*throughput*). Per processi di lunga durata il suo valore può essere di un processo all'ora, mentre per brevi transazioni è possibile avere un throughput di 10 processi al secondo.
- **Tempo di completamento.** Dal punto di vista di uno specifico processo, il criterio più importante è il tempo necessario per eseguire il processo stesso. L'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato **tempo di completamento** (*turnaround time*), ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella ready queue (cioè la ready queue), durante l'esecuzione nella CPU e nelle operazioni di I/O.
- **Tempo d'attesa.** L'algoritmo di scheduling della CPU non influisce sul tempo impiegato per l'esecuzione di un processo o di un'operazione di I/O; influisce solo sul tempo d'attesa nella ready queue. Il **tempo d'attesa** è la somma degli intervalli d'attesa passati in questa coda.
- **Tempo di risposta.** In un sistema interattivo il tempo di completamento può non essere il miglior criterio di valutazione: spesso accade che un processo emetta dati abbastanza presto, e continui a calcolare i nuovi risultati mentre quelli precedenti sono in fase di output per l'utente. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la effettuazione di una richiesta e la prima risposta prodotta. Questa misura è chiamata **tempo di risposta**, ed è data dal tempo necessario per iniziare la risposta, ma non dal suo tempo necessario per completare l'output. Generalmente il tempo di completamento è limitato dalla velocità del dispositivo di output.

È auspicabile aumentare al massimo utilizzo e produttività della CPU, mentre il tempo di completamento, il tempo d'attesa e il tempo di risposta si devono ridurre al minimo. Nella maggior parte dei casi si ottimizzano i valori medi; tuttavia in alcune circostanze è più opportuno ottimizzare i valori minimi o massimi, anziché i valori medi; per esempio, per garantire che tutti gli utenti ottengano un buon servizio, possiamo voler ridurre il massimo tempo di risposta.

Per i sistemi interattivi, come i sistemi desktop, alcuni analisti suggeriscono che sia più importante ridurre al minimo la varianza del tempo di risposta anziché il tempo medio di risposta. Un sistema il cui tempo di risposta sia ragionevole e prevedibile può essere considerato migliore di un sistema mediamente più rapido, ma molto variabile. Tuttavia, è stato fatto poco sugli algoritmi di scheduling della CPU al fine di ridurre al minimo la varianza.

Nell'analizzare i diversi algoritmi di scheduling della CPU, dobbiamo esemplificare il funzionamento. Una descrizione approfondita richiederebbe il ricorso a molti processi, ognuno dei quali costituito da parecchie centinaia di sequenze di operazioni della CPU e di sequenze di operazioni di I/O. Per motivi di semplicità, negli esempi si considera una sola sequenza di operazioni della CPU (la cui durata è espressa in millisecondi) per ogni processo. La misura di confronto adottata è il tempo d'attesa medio. Meccanismi di valutazione più raffinati sono trattati nel Paragrafo 6.8.

6.3 Algoritmi di scheduling

Lo scheduling della CPU riguarda la scelta di quale processo presente nella ready queue a cui assegnare la CPU. In questo paragrafo si descrivono alcuni fra i tanti algoritmi di scheduling della CPU.

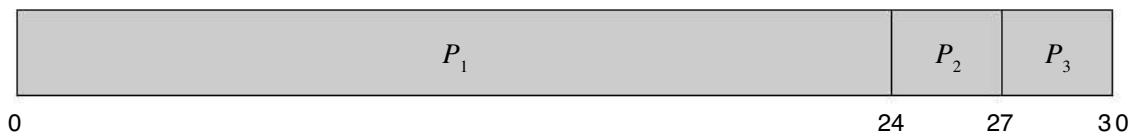
6.3.1 Scheduling in ordine d'arrivo

Il più semplice algoritmo di scheduling della CPU è l'algoritmo di **scheduling in ordine d'arrivo** (*scheduling first-come, first-served* o FCFS). Con questo schema la CPU si assegna al processo che la richiede per primo. La realizzazione del criterio FCFS si basa su una coda FIFO. Quando un processo entra nella ready queue, si collega il suo PCB all'ultimo elemento della coda. Quando la CPU è libera, viene assegnata al processo che si trova alla testa della coda, rimuovendolo da essa. Il codice per lo scheduling FCFS è semplice sia da scrivere sia da capire.

Un aspetto negativo è che il tempo medio d'attesa per l'algoritmo FCFS è spesso abbastanza lungo. Si consideri il seguente insieme di processi, che si presenta al momento 0, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se i processi arrivano nell'ordine P_1, P_2, P_3 e sono serviti in ordine FCFS, si ottiene il risultato illustrato nel seguente **diagramma di Gantt**, un diagramma a barre che illustra una data pianificazione includendo i tempi d'inizio e fine di ogni processo partecipante.



Il tempo d'attesa è 0 millisecondi per il processo P_1 , 24 millisecondi per il processo P_2 e 27 millisecondi per il processo P_3 . Quindi, il tempo d'attesa medio è $(0 + 24 + 27)/3 = 17$ millisecondi. Se i processi arrivassero nell'ordine P_2, P_3, P_1 , i risultati sarebbero quelli illustrati nel seguente diagramma di Gantt:



Il tempo di attesa medio è ora di $(6 + 0 + 3)/3 = 3$ millisecondi. Si tratta di una notevole riduzione. Quindi, il tempo medio d'attesa in condizioni di FCFS non è in genere minimo, e può variare grandemente all'aumentare della variabilità dei CPU burst dei vari processi.

Si considerino inoltre le prestazioni dello scheduling FCFS in una situazione dinamica. Si supponga di avere un processo con prevalenza d'elaborazione e molti processi con prevalenza di I/O. Via via che i processi fluiscono nel sistema si può verificare la seguente situazione. Il processo con prevalenza d'elaborazione occupa la CPU. Durante questo periodo tutti gli altri processi terminano le proprie operazioni di I/O e si spostano nella ready queue, nell'attesa della CPU. Mentre i processi si trovano nella ready queue, i dispositivi di I/O sono inattivi. Successivamente il processo con prevalenza d'elaborazione termina la propria sequenza di operazioni della CPU e passa a una fase di I/O. Tutti i processi con prevalenza di I/O, caratterizzati da sequenze di operazioni della CPU molto brevi, sono eseguiti rapidamente e tornano alle code di I/O, lasciando inattiva la CPU. Il processo con prevalenza d'elaborazione torna nella ready queue e riceve il controllo della CPU; così, finché non termina l'esecuzione del processo con prevalenza d'elaborazione, tutti i processi con prevalenza di I/O si trovano nuovamente ad attendere nella ready queue. Si ha un **effetto convoglio**, tutti i processi attendono che un lungo processo liberi la CPU, il che causa una riduzione dell'utilizzo della CPU e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi.

L'algoritmo di scheduling FCFS è senza prelazione; una volta che la CPU è assegnata a un processo, questo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o alla richiesta di un'operazione di I/O. L'algoritmo FCFS risulta particolarmente problematico nei sistemi in time sharing, dove è importante che ogni utente disponga della CPU a intervalli regolari. Permettere a un solo processo di occupare la CPU per un lungo periodo condurrebbe a risultati disastrosi.

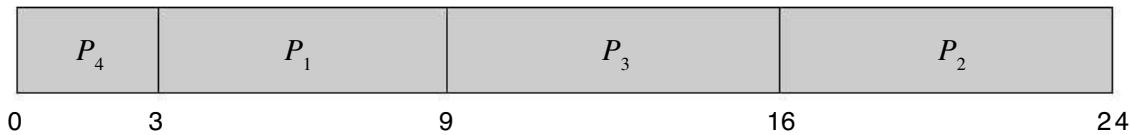
6.3.2 Scheduling shortest-job-first

Un criterio diverso di scheduling della CPU è l'algoritmo di **scheduling per brevità** (*shortest-job-first*, SJF). Questo algoritmo associa a ogni processo la lunghezza della successiva sequenza di operazioni della CPU. Quando è disponibile, si assegna la CPU al processo che ha la più breve lunghezza della successiva sequenza di operazioni della CPU. Se due processi hanno le successive sequenze di operazioni della CPU della stessa lunghezza si applica lo scheduling FCFS. Si noti che sarebbe più appropriato il termine *shortest next CPU burst*, infatti lo scheduling si esegue esaminando la lunghezza della successiva sequenza di operazioni della CPU del processo e non la sua lunghezza totale. Tuttavia, poiché è comunemente usato ed è presente nella maggior parte dei libri di testo, anche qui si fa uso del termine SJF.

Come esempio si consideri il seguente insieme di processi, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
P_1	6
P_2	8
P_3	7
P_4	3

Con lo scheduling SJF questi processi si ordinerebbero secondo il seguente diagramma di Gantt.



Il tempo d'attesa è di 3 millisecondi per il processo P_1 , di 16 millisecondi per il processo P_2 , di 9 millisecondi per il processo P_3 e di 0 millisecondi per il processo P_4 . Quindi, il tempo d'attesa medio è di $(3 + 16 + 9 + 0)/4 = 7$ millisecondi. Usando lo scheduling FCFS, il tempo d'attesa medio sarebbe di 10,25 millisecondi.

Si può dimostrare che l'algoritmo di scheduling SJF è *ottimale*, nel senso che rende minimo il tempo d'attesa medio per un dato insieme di processi. Spostando un processo breve prima di un processo lungo, il tempo d'attesa per il processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza, il tempo d'attesa *medio* diminuisce.

La difficoltà reale implicita nell'algoritmo SJF consiste nel conoscere la durata della successiva richiesta della CPU. Per lo scheduling a lungo termine (*job scheduling*) di un sistema batch si può usare come durata il tempo limite d'elaborazione che gli utenti specificano nel sottoporre il job. In questa situazione gli utenti sono quindi motivati a stabilire con precisione tale limite, poiché un valore inferiore può significare una risposta più rapida, ma un valore troppo basso causerebbe un errore di superamento del tempo limite e richiederebbe una nuova esecuzione. Lo scheduling SJF si usa spesso nello scheduling a lungo termine.

Sebbene sia ottimale, l'algoritmo SJF non si può realizzare a livello dello scheduling della CPU a breve termine, poiché non esiste alcun modo per conoscere la lunghezza della successiva sequenza di operazioni della CPU. Un possibile approccio a questo problema consiste nel tentare di approssimare lo scheduling SJF: se non è possibile *conoscere* la lunghezza della prossima sequenza di operazioni della CPU, si può cercare di *predire* il suo valore; è probabile, infatti, che sia simile alle precedenti. Quindi, calcolando un valore approssimato della lunghezza, si può scegliere il processo con la più breve fra tali lunghezze previste.

La lunghezza della successiva sequenza di operazioni della CPU generalmente si stima calcolando la **media esponenziale** delle lunghezze misurate delle precedenti

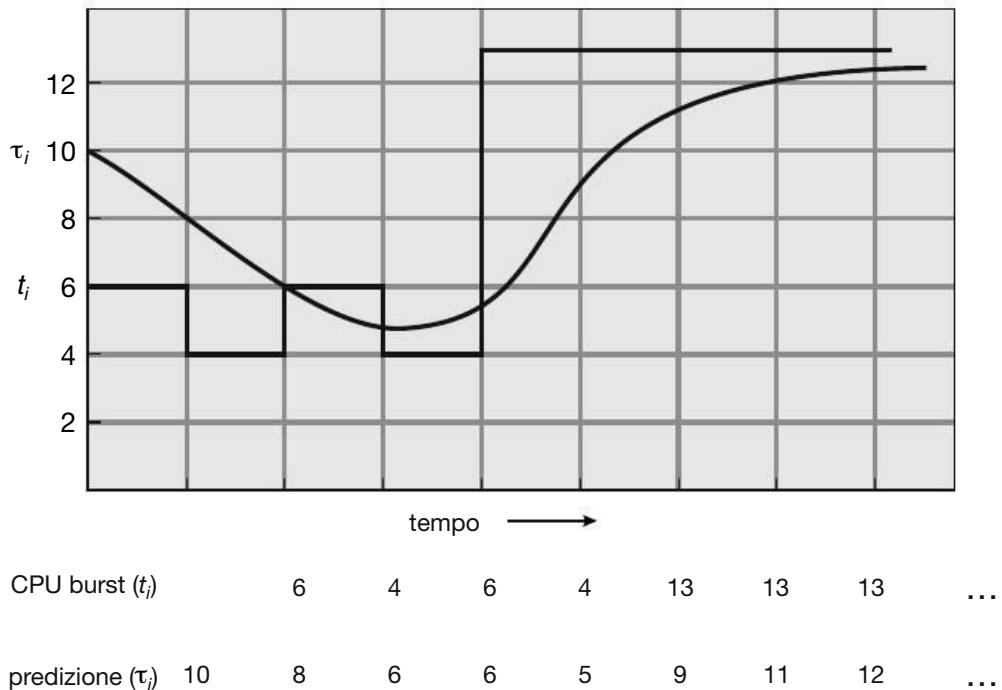


Figura 6.3 Predizione della lunghezza della successiva sequenza di operazioni della CPU (CPU burst).

sequenze di operazioni della CPU. La media esponenziale si definisce con la formula seguente. Siano t_n la lunghezza dell' n -esima sequenza di operazioni della CPU e τ_{n+1} il valore previsto per la successiva sequenza. Allora, dato α tale che $0 \leq \alpha \leq 1$, si definisce

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Il valore di t_n contiene le informazioni più recenti; τ_n registra la storia passata. Il parametro α controlla il peso relativo sulla predizione della storia recente e di quella passata. Se $\alpha = 0$, allora, $\tau_{n+1} = \tau_n$, e la storia recente non ha effetto (si suppone che le condizioni attuali siano transitorie); se $\alpha = 1$, allora $\tau_{n+1} = t_n$, e ha significato solo la più recente sequenza di operazioni della CPU (si suppone che la storia sia irrilevante). Più comune è la condizione in cui $\alpha = 1/2$, valore che indica che la storia recente e la storia passata hanno lo stesso peso. Il τ_0 iniziale si può definire come una costante o come una media complessiva del sistema. Nella Figura 6.3 è illustrata una media esponenziale con $\alpha = 1/2$ e $\tau_0 = 10$.

Per comprendere il comportamento della media esponenziale, si può sviluppare la formula per τ_{n+1} sostituendo il valore di τ_n , in modo da ottenere

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

Di solito α è minore di 1. Quindi, anche $(1 - \alpha)$ è minore di 1 e ogni termine ha peso inferiore a quello del suo predecessore.

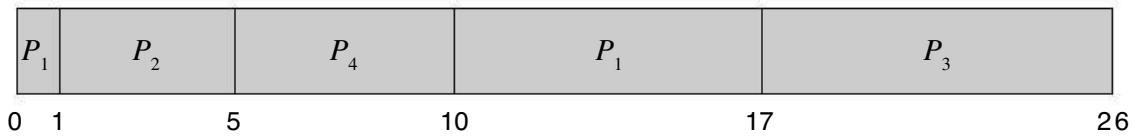
L'algoritmo SJF può essere sia *con prelazione* sia *senza prelazione*. La scelta si presenta quando alla ready queue arriva un nuovo processo mentre un altro processo

è ancora in esecuzione. Il nuovo processo può richiedere una sequenza di operazioni della CPU più breve di quella che resta al processo correntemente in esecuzione. Un algoritmo SJF con prelazione sostituisce il processo attualmente in esecuzione, mentre un algoritmo SJF senza prelazione permette al processo correntemente in esecuzione di portare a termine la propria sequenza di operazioni della CPU. (Lo scheduling SJF con prelazione è talvolta chiamato scheduling *shortest-remaining-time-first*).

Come esempio, si considerino i quattro processi seguenti, dove la durata delle sequenze di operazioni della CPU è data in millisecondi:

Processo	Istante d'arrivo	Durata della sequenza
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Se i processi arrivano alla ready queue nei momenti indicati e richiedono i tempi di CPU illustrati, dallo scheduling SJF con prelazione risulta la sequenza indicata dal seguente diagramma di Gantt.



All'istante 0 si avvia il processo P_1 , poiché è l'unico che si trova nella coda. All'istante 1 arriva il processo P_2 . Il tempo necessario per completare il processo P_1 (7 millisecondi) è maggiore del tempo richiesto dal processo P_2 (4 millisecondi), perciò si effettua la prelazione del processo P_1 sostituendolo col processo P_2 . Il tempo d'attesa medio per questo esempio è $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6,5$ millisecondi. Con uno scheduling SJF senza prelazione si otterebbe un tempo d'attesa medio di 7,75 millisecondi.

6.3.3 Scheduling con priorità

L'algoritmo SJF è un caso particolare del più generale **algoritmo di scheduling con priorità**: si associa una priorità a ogni processo e si assegna la CPU al processo con priorità più alta; i processi con priorità uguali si ordinano secondo uno schema FCFS. Un algoritmo SJF è semplicemente un algoritmo con priorità in cui la priorità (p) è l'inverso della lunghezza (prevista) della successiva sequenza di operazioni della CPU. A una maggiore lunghezza corrisponde una minore priorità, e viceversa.

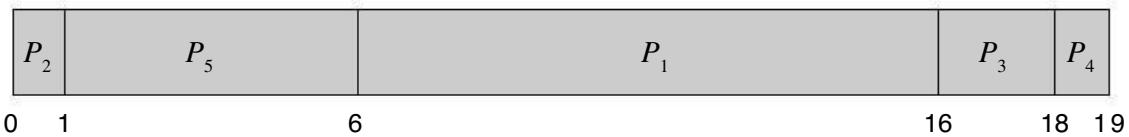
Occorre notare che la discussione si svolge in termini di priorità *alta* e priorità *bassa*. Generalmente le priorità sono indicate da un intervallo fisso di numeri, come da 0 a 7, oppure da 0 a 4.095. Tuttavia, non c'è un orientamento comune sull'attribuire allo 0 la priorità più alta o quella più bassa; alcuni sistemi usano numeri bassi

per rappresentare priorità basse, altri usano numeri bassi per priorità alte, il che può generare confusione. In questo testo i numeri bassi indicano priorità alte.

Come esempio, si consideri il seguente insieme di processi, che si suppone siano arrivati al tempo 0, nell'ordine $P_1, P_2 \dots P_5$, e dove la durata delle sequenze di operazioni della CPU è espressa in millisecondi:

Processo	Durata della sequenza	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Usando lo scheduling con priorità, questi processi sarebbero ordinati secondo il seguente diagramma di Gantt.



Il tempo d'attesa medio è di 8,2 millisecondi.

Le priorità si possono definire sia *internamente* sia *esternamente*. Quelle definite internamente usano una o più quantità misurabili per calcolare la priorità del processo; per esempio sono stati utilizzati i limiti di tempo, i requisiti di memoria, il numero dei file aperti e il rapporto tra la lunghezza media delle sequenze di operazioni di I/O e la lunghezza media delle sequenze di operazioni della CPU. Le priorità esterne si definiscono secondo criteri esterni al sistema operativo, come l'importanza del processo, il tipo e la quantità dei fondi pagati per l'uso del calcolatore, il dipartimento che promuove il lavoro e altri fattori, spesso di ordine politico.

Lo scheduling con priorità può essere sia con prelazione sia senza prelazione. Quando un processo arriva alla ready queue, si confronta la sua priorità con quella del processo attualmente in esecuzione. Un algoritmo di scheduling con priorità con diritto di prelazione sottrae la CPU al processo attualmente in esecuzione se la priorità del processo appena arrivato è superiore. Un algoritmo senza prelazione si limita a porre l'ultimo processo arrivato alla testa della ready queue.

Un problema importante relativo agli algoritmi di scheduling con priorità è **l'attesa indefinita** (*starvation*). Un processo pronto per l'esecuzione, ma che non dispone della CPU, si può considerare bloccato nell'attesa della CPU. Un algoritmo di scheduling con priorità può lasciare processi a bassa priorità nell'attesa indefinita della CPU. In un sistema con carico elevato, un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla CPU. Generalmente accade che o il processo è eseguito, alle ore 2 del mattino della domenica,

quando il sistema ha finalmente ridotto il proprio carico, oppure il calcolatore, prima o poi, va in crash e perde tutti i processi a bassa priorità non terminati. Corre voce che, quando fu fermato l'IBM 7094 al MIT, nel 1973, si scoprì che un processo con bassa priorità sottoposto nel 1967 non era ancora stato eseguito.

Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'**invecchiamento** (*aging*); si tratta di una tecnica di aumento graduale delle priorità dei processi che attendono nel sistema da parecchio tempo. Per esempio, se le priorità variano da 127 (bassa) a 0 (alta), si potrebbe decrementare di 1 ogni 15 minuti la priorità di un processo in attesa. Alla fine anche un processo con priorità iniziale 127 può ottenere la priorità massima nel sistema e quindi essere eseguito: un processo con priorità 127 non impiegherebbe più di 32 ore per raggiungere la priorità 0.

6.3.4 Scheduling circolare

L'**algoritmo di scheduling circolare** (*round-robin*, **RR**) è stato progettato appositamente per i sistemi time sharing; è simile allo scheduling FCFS, ma aggiunge la capacità di prelazione in modo che il sistema possa commutare fra i vari processi. Ciascun processo riceve una piccola quantità fissata del tempo della CPU, chiamata **quanto di tempo o porzione di tempo** (*time slice*), che varia generalmente da 10 a 100 millisecondi; la ready queue è trattata come una coda circolare. Lo scheduler della CPU scorre la ready queue, assegnando la CPU a ciascun processo per un intervallo di tempo della durata massima di un quanto di tempo.

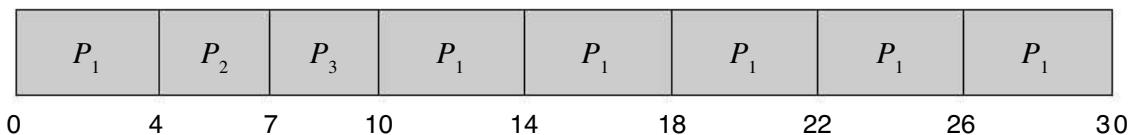
Per implementare lo scheduling RR si gestisce la ready queue come una coda FIFO. I nuovi processi si aggiungono alla fine della ready queue. Lo scheduler della CPU prende il primo processo dalla ready queue, imposta un timer in modo che invii un segnale d'interruzione alla scadenza di un intervallo pari a un quanto di tempo, e attiva il dispatcher per eseguire il processo.

A questo punto si può verificare una delle due seguenti situazioni: il processo ha una sequenza di operazioni della CPU di durata minore di un quanto di tempo, quindi il processo stesso rilascia volontariamente la CPU e lo scheduler passa al processo successivo della ready queue; oppure la durata della sequenza di operazioni è più lunga di un quanto di tempo; in questo caso il timer scade e invia un segnale d'interruzione al sistema operativo, che esegue un cambio di contesto e mette il processo alla fine della ready queue. Lo scheduler quindi seleziona il processo successivo nella ready queue.

Il tempo d'attesa medio per il criterio di scheduling RR è spesso abbastanza lungo. Si consideri il seguente insieme di processi che si presenta al tempo 0, con la durata delle sequenze di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se si usa un quanto di tempo di 4 millisecondi, il processo P_1 ottiene i primi 4 millisecondi ma, poiché richiede altri 20 millisecondi, è soggetto a prelazione dopo il primo quanto di tempo e la CPU passa al processo successivo della coda, il processo P_2 . Poiché il processo P_2 non necessita di 4 millisecondi, termina prima che il suo quanto di tempo si esaurisca, così si assegna immediatamente la CPU al processo successivo, il processo P_3 . Una volta che tutti i processi hanno ricevuto un quanto di tempo, si assegna nuovamente la CPU al processo P_1 per un ulteriore quanto di tempo. Dallo scheduling RR risulta quanto segue.



Calcoliamo ora il tempo di attesa medio per questa sequenza. P_1 resta in attesa per 6 millisecondi (10-4), P_2 per 4 millisecondi e P_3 per 7 millisecondi. Il tempo d'attesa medio è di $17/3 = 5,66$ millisecondi.

Nell'algoritmo di scheduling RR la CPU si assegna a un processo per non più di un quanto di tempo per volta (a meno che questo sia l'unico processo ready). Se la durata della sequenza di operazioni della CPU di un processo eccede il quanto di tempo, il processo viene sottoposto a prelazione e riportato nella ready queue. L'algoritmo di scheduling RR è pertanto con prelazione.

Se nella ready queue esistono n processi e il quanto di tempo è pari a q , ciascun processo ottiene $1/n$ -esimo del tempo di elaborazione della CPU in frazioni di, al massimo, q unità di tempo. Ogni processo non deve attendere per più di $(n - 1) \times q$ unità di tempo per il suo successivo quanto di tempo. Per esempio, dati cinque processi e un quanto di tempo di 20 millisecondi, ogni processo usa fino a 20 millisecondi ogni 100 millisecondi.

Le prestazioni dell'algoritmo RR dipendono molto dalla dimensione del quanto di tempo. Nel caso limite in cui il quanto di tempo sia molto lungo, il criterio di scheduling RR si riduce al criterio di scheduling FCFS. Se il quanto di tempo è molto breve (per esempio, un millisecondo), il criterio RR può portare a un numero elevato di cambi di contesto. Assumiamo che, per esempio, ci sia un solo processo della durata di 10 unità di tempo, se il quanto di tempo è di 12 unità, il processo impiega meno di un quanto di tempo; se però il quanto di tempo è di 6 unità, il processo richiede 2 quanti di tempo e un cambio di contesto; e se il quanto di tempo è di un'unità di tempo, occorrono nove cambi di contesto, con proporzionale rallentamento dell'esecuzione del processo (Figura 6.4).

Quindi il quanto di tempo deve essere ampio rispetto alla durata del cambio di contesto; se, per esempio, questa è pari al 10 per cento del quanto di tempo, allora s'impiega in cambi di contesto circa il 10 per cento del tempo d'elaborazione della CPU. In pratica, nella maggior parte dei sistemi moderni un quanto di tempo va dai 10 ai 100 millisecondi. Il tempo richiesto per un cambio di contesto non eccede so-

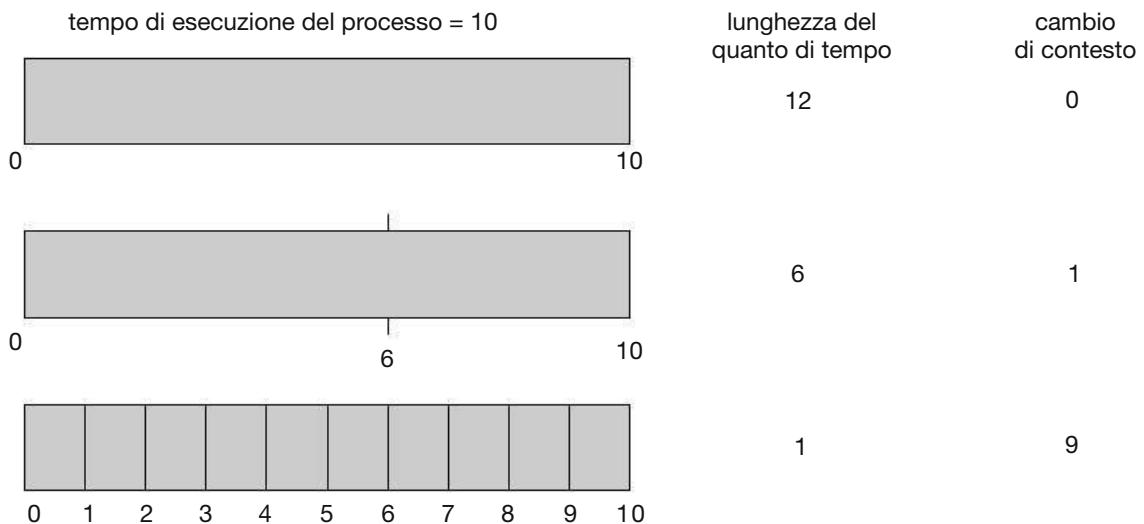


Figura 6.4 Aumento del numero dei cambi di contesto al diminuire del quanto di tempo.

litamente i 10 microsecondi, risultando quindi una modesta frazione del quanto di tempo.

Anche il tempo di completamento (*turnaround time*) dipende dalla dimensione del quanto di tempo: com'è evidenziato nella Figura 6.5, il tempo di completamento medio di un insieme di processi non migliora necessariamente con l'aumento della

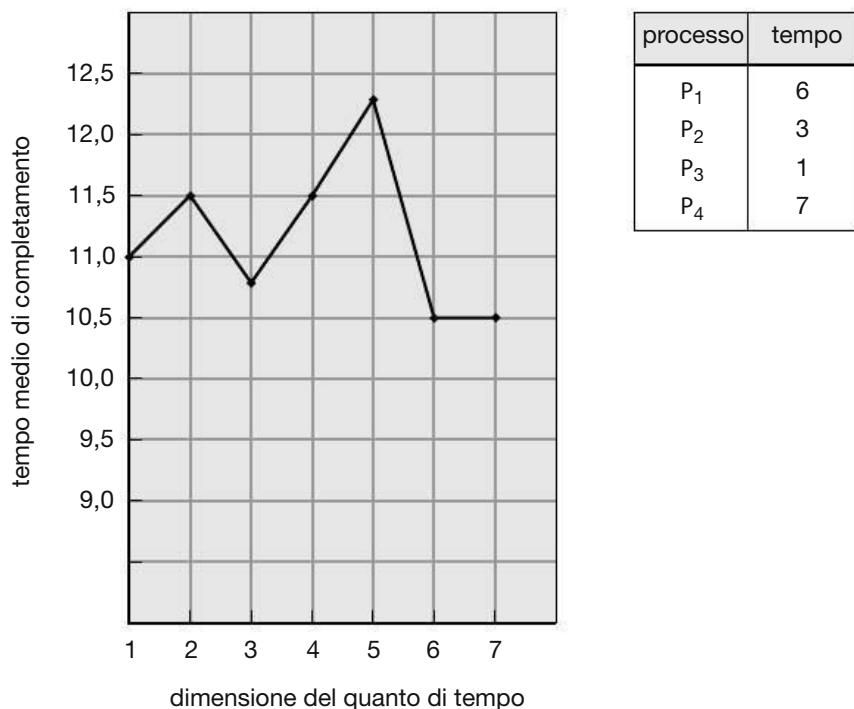


Figura 6.5 Variazione del tempo di completamento in funzione del quanto di tempo.

dimensione del quanto di tempo. In generale, il tempo di completamento medio può migliorare se la maggior parte dei processi termina la successiva sequenza di operazioni della CPU in un solo quanto di tempo. Per esempio, dati tre processi della durata di 10 unità di tempo ciascuno e un quanto di una unità di tempo, il tempo di completamento medio è di 29 unità. Se però il quanto di tempo è di 10 unità, il tempo di completamento medio scende a 20 unità. Aggiungendo il tempo del cambio di contesto, con un piccolo quanto di tempo, il tempo di completamento medio aumenta ancora poiché sono richiesti più cambi di contesto.

Benchè il quanto di tempo debba essere grande in confronto al tempo necessario al cambio di contesto, non deve essere tuttavia troppo grande. Se il quanto di tempo è molto ampio, il criterio di scheduling RR, come puntualizzato in precedenza, tende al criterio FCFS. Empiricamente si può stabilire che l'80 per cento delle sequenze di operazioni della CPU debba essere più breve del quanto di tempo.

6.3.5 Scheduling a code multilivello

È stata creata una classe di algoritmi di scheduling adatta a situazioni in cui i processi si possono classificare facilmente in gruppi diversi. Una distinzione comune è per esempio quella che si fa tra i processi che si eseguono in **foreground** (*primo piano*), o **interattivi**, e i processi che si eseguono in **background** (*sottofondo*), o **batch** (a lotti). Questi due tipi di processi hanno differenti requisiti sui tempi di risposta e possono quindi avere diverse necessità di scheduling. Inoltre, i processi in foreground possono avere una priorità, definita esternamente, sui processi di background.

Un **algoritmo di scheduling a code multilivello** (*multilevel queue scheduling algorithm*) suddivide la ready queue in code distinte (Figura 6.6). I processi si assegnano in modo permanente a una coda, generalmente secondo qualche caratteristica del processo, come la quantità di memoria richiesta, la priorità o il tipo di processo. Ogni coda ha il proprio algoritmo di scheduling. Per esempio, per i processi in foreground e i processi in background si possono usare code distinte. La coda dei processi in foreground si può gestire con un algoritmo RR, mentre quella dei processi in background si può gestire con un algoritmo FCFS.

In questa situazione è inoltre necessario avere uno scheduling tra le code; si tratta comunemente di uno scheduling a priorità fissa con prelazione. Per esempio, la coda dei processi in foreground può avere la priorità assoluta sulla coda dei processi in background.

Si consideri come esempio il seguente algoritmo di scheduling a code multilivello, in ordine di priorità:

1. processi di sistema;
2. processi interattivi;
3. processi interattivi di *editing*;
4. processi batch;
5. processi degli studenti.

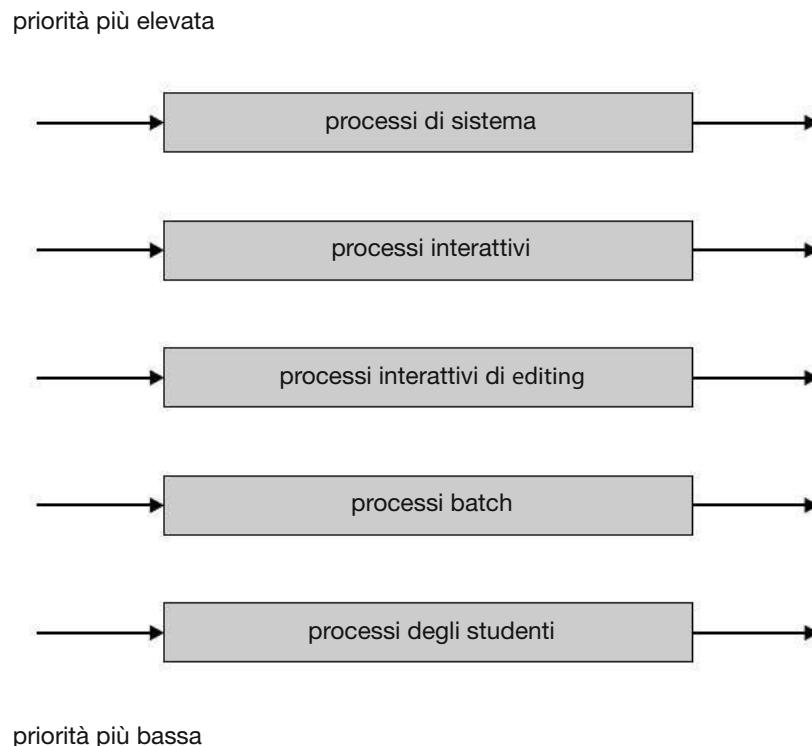


Figura 6.6 Scheduling a code multilivello.

Ogni coda ha la priorità assoluta sulle code di priorità più bassa; ad esempio nessun processo della coda dei processi batch può iniziare l'esecuzione finché le code per i processi di sistema, interattivi e interattivi di *editing* non siano tutte vuote. Se un processo interattivo di *editing* entrasse nella ready queue durante l'esecuzione di un processo in background, si avrebbe la prelazione su quest'ultimo.

Un'altra possibilità è definire porzioni di tempo per le code. Ad ogni coda si assegna una parte del tempo d'elaborazione della CPU, suddivisibile a sua volta tra i processi che la costituiscono. Nel caso foreground/background, per esempio, si può assegnare l'80 per cento del tempo d'elaborazione della CPU alla coda dei processi in foreground, con scheduling RR tra i suoi processi; mentre per la coda dei processi in background si riserva il 20 per cento del tempo d'elaborazione della CPU, assegnato col criterio FCFS.

6.3.6 Scheduling a code multilivello con retroazione

Di solito in un algoritmo di scheduling a code multilivello i processi si assegnano in modo permanente a una coda all'entrata nel sistema, e non si possono spostare tra le code. Se, per esempio, esistono code distinte per i processi che si eseguono in foreground e quelli che si eseguono in background, i processi non possono passare da una coda all'altra, poiché non possono cambiare la loro natura di processi in foreground o in background. Quest'impostazione è rigida, ma ha il vantaggio di avere un basso carico di scheduling.

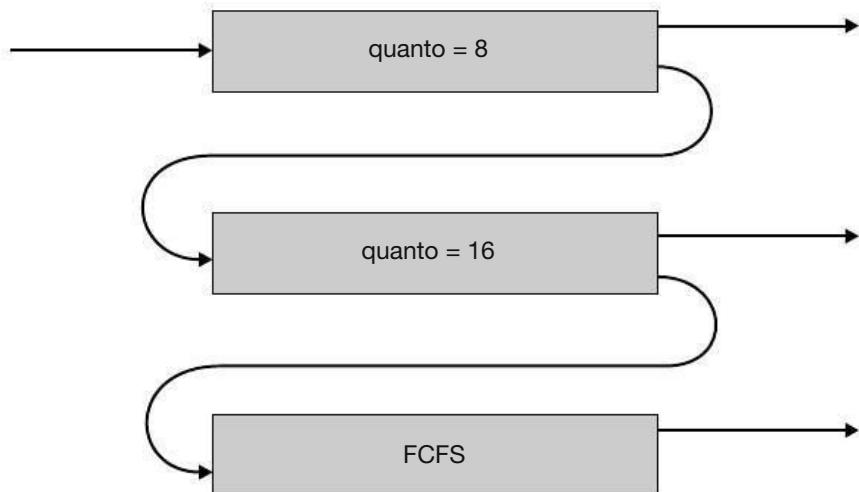


Figura 6.7 Code multilivello con retroazione.

Lo **scheduling a code multilivello con retroazione** (*multilevel feedback queue scheduling*), invece, permette ai processi di spostarsi fra le code. L’idea consiste nel separare i processi che hanno caratteristiche diverse in termini di CPU burst. Se un processo usa troppo tempo di elaborazione della CPU, viene spostato in una coda con priorità più bassa. Questo schema mantiene i processi con prevalenza di I/O e i processi interattivi nelle code con priorità più elevata. Inoltre, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo in una coda a priorità bassa. Questa forma di aging impedisce il verificarsi di un’attesa indefinita.

Si consideri, per esempio, uno scheduler a code multilivello con retroazione con tre code, numerate da 0 a 2, come nella Figura 6.7. Lo scheduler fa eseguire tutti i processi presenti nella coda 0; quando la coda 0 è vuota, si eseguono i processi nella coda 1; analogamente, i processi nella coda 2 si eseguono solo se le code 0 e 1 sono vuote. Un processo in ingresso nella coda 1 ha la prelazione sui processi della coda 2; un processo in ingresso nella coda 0, a sua volta, ha la prelazione sui processi della coda 1.

All’ingresso nella ready queue, i processi vengono assegnati alla coda 0 e ottengono un quanto di tempo di 8 millisecondi; i processi che non terminano entro tale quanto di tempo, vengono spostati alla fine della coda 1. Se la coda 0 è vuota, si assegna un quanto di tempo di 16 millisecondi al processo alla testa della coda 1, ma se questo non riesce a completare la propria esecuzione, viene sottoposto a prelazione e messo nella coda 2. Se le code 0 e 1 sono vuote, si eseguono i processi della coda 2 secondo il criterio FCFS.

Questo algoritmo di scheduling dà la massima priorità ai processi con una sequenza di operazioni della CPU della durata di non più di 8 millisecondi. I processi di questo tipo ottengono rapidamente la CPU, terminano la propria sequenza di operazioni della CPU e passano alla successiva sequenza di operazioni di I/O; anche i processi che necessitano di più di 8 ma di non più di 24 millisecondi (coda 1) vengono serviti

rapidamente, anche se con una priorità inferiore. I processi più lunghi finiscono automaticamente nella coda 2 e sono serviti secondo il criterio FCFS all'interno dei cicli di CPU lasciati liberi dai processi delle code 0 e 1.

Generalmente uno scheduler a code multilivello con retroazione è caratterizzato dai seguenti parametri:

- numero di code;
- algoritmo di scheduling per ciascuna coda;
- metodo usato per determinare quando spostare un processo in una coda con priorità maggiore;
- metodo usato per determinare quando spostare un processo in una coda con priorità minore;
- metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio.

La definizione di uno scheduler a code multilivello con retroazione costituisce il più generale criterio di scheduling della CPU, che nella fase di progettazione si può adeguare a un sistema specifico. Sfortunatamente corrisponde anche all'algoritmo più complesso; la definizione dello scheduler migliore richiede infatti particolari metodi per la selezione dei valori dei diversi parametri.

6.4 Scheduling dei thread

Nel Capitolo 4 abbiamo arricchito il modello dei processi con i thread, distinguendo quelli *a livello utente* da quelli *a livello kernel*. Sui sistemi operativi che prevedono la loro presenza, il sistema non effettua lo scheduling dei processi, ma dei thread a livello kernel. I thread a livello utente sono gestiti da una libreria: il kernel non è consapevole della loro esistenza. Di conseguenza, per eseguire i thread a livello utente occorre associare loro dei thread a livello kernel. Tale associazione può essere indiretta, ossia realizzata con un processo leggero (LWP). Trattiamo adesso le questioni dello scheduling che riguardano i thread a livello utente e a livello kernel, offrendo esempi specifici dello scheduling per Pthreads.

6.4.1 Ambito della contesa

Una distinzione fra thread a livello utente e a livello kernel riguarda il modo in cui vengono schedulati. Nei sistemi che impiegano il modello da molti a uno (Paragrafo 4.3.1) e il modello da molti a molti (Paragrafo 4.3.3), la libreria dei thread pianifica l'esecuzione dei thread a livello utente su un LWP libero; si parla allora di **ambito della contesa ristretto al processo** (*process-contention scope*, PCS), perché la contesa per aggiudicarsi la CPU ha luogo fra thread dello stesso processo. In realtà, affermando che la libreria dei thread pianifica l'esecuzione dei thread a livello utente associandoli agli LWP liberi, non si intende che il thread sia in esecuzione su una CPU; ciò avviene solo quando il sistema operativo pianifica l'esecuzione di thread del

kernel su un processore fisico. Per determinare quale thread a livello kernel debba essere eseguito da una CPU, il kernel esamina i thread di tutto il sistema; si parla allora di **ambito della contesa allargato al sistema** (*system-contention scope*, SCS). Quindi, nel caso di SCS, tutti i thread del sistema competono per l'uso della CPU. I sistemi caratterizzati dal modello da uno a uno (Paragrafo 4.3.2) quali Windows, Linux e Solaris, schedulano i thread unicamente sulla base di SCS.

Nel caso del PCS, lo scheduling è solitamente basato sulle priorità: lo scheduler sceglie per l'esecuzione il thread con priorità più alta. Le priorità dei thread a livello utente sono stabilite dal programmatore, e la libreria dei thread non le modifica; alcune librerie danno facoltà al programmatore di cambiare la priorità di un thread. Si noti che quando l'ambito della contesa è ristretto al processo si è soliti applicare la prelazione al thread in esecuzione, a vantaggio di thread con priorità più alta; tuttavia, se i thread sono dotati della medesima priorità, non vi è garanzia di *time slicing* (porzione di tempo) (Paragrafo 6.3.4).

6.4.2 Scheduling di Pthread

La generazione dei thread con POSIX Pthreads è stata introdotta nel Paragrafo 4.4.1, insieme a un programma esemplificativo. Ci accingiamo ora a esaminare la API Pthread di POSIX, che consente di specificare PCS o SCS nella fase di generazione dei thread. Per specificare l'ambito della contesa Pthreads usa i valori seguenti:

- PTHREAD_SCOPE_PROCESS pianifica i thread con lo scheduling PCS
- PTHREAD_SCOPE_SYSTEM pianifica i thread tramite lo scheduling SCS

Nei sistemi che si avvalgono del modello da molti a molti la politica PTHREAD_SCOPE_PROCESS pianifica i thread a livello utente sugli LWP disponibili. Il numero di LWP viene mantenuto dalla libreria dei thread, che può servirsi delle attivazioni dello scheduler (Paragrafo 4.6.5). La seconda politica, PTHREAD_SCOPE_SYSTEM, crea, in corrispondenza di ciascun thread a livello utente, un LWP a esso vincolato, realizzando così, in effetti, una corrispondenza secondo il modello da molti a uno.

L'IPC di Pthread offre due funzioni per leggere (e impostare) l'ambito della contesa:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

Il primo parametro per entrambe le funzioni è un puntatore agli attributi del thread. Il secondo parametro della funzione `pthread_attr_setscope()` riceve uno dei valori PTHREAD_SCOPE_SYSTEM o PTHREAD_SCOPE_PROCESS, che stabiliscono come deve essere impostato l'ambito della contesa. Nel caso di `pthread_attr_getscope()` il secondo parametro contiene un puntatore ad un intero che contiene l'attuale valore dell'ambito della contesa. Qualora si verifichi un errore, ambedue le funzioni restituiscono valori non nulli.

Nella Figura 6.8 illustriamo l'API di scheduling Pthread mediante un programma che determina l'ambito della contesa in vigore e lo imposta a PTHREAD_SCOPE_SYSTEM; quindi crea cinque thread distinti, che andranno in esecuzione secondo il

```

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* ottiene gli attributi di default */
    pthread_attr_init(&attr);

    /* per prima cosa appura l'ambito della contesa */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Impossibile appurare l'ambito della contesa\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Valore d'ambito della contesa non ammesso.\n");
    }

    /* imposta l'algoritmo di scheduling a PCS o SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* genera i thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* adesso aspetta la terminazione di tutti i thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* ogni thread inizia l'esecuzione da questa funzione */
void *runner(void *param)
{
    /* fai qualcosa ... */

    pthread_exit(0);
}

```

Figura 6.8 API di scheduling Pthread.

modello di scheduling SCS. Si noti che, nel caso di alcuni sistemi, sono possibili solo determinati valori per l'ambito della contesa. I sistemi Linux e Mac OS X, per esempio, consentono soltanto PTHREAD_SCOPE_SYSTEM.

6.5 Scheduling per sistemi multiprocessore

Fin qui la trattazione ha riguardato lo scheduling della CPU in un sistema a processore singolo. Se sono disponibili più unità d'elaborazione, è possibile la distribuzione del carico (*load sharing*), ma il problema dello scheduling diviene proporzionalmente più complesso. Si sono sperimentate diverse possibilità e, come s'è visto nella trattazione dello scheduling di una sola CPU, “la soluzione migliore” non esiste. Nel seguito si analizzano brevemente alcuni problemi attinenti lo scheduling per sistemi multiprocessore. Si considerano i sistemi in cui le unità d'elaborazione sono funzionalmente identiche (**sistemi omogenei**): si può usare qualunque unità d'elaborazione disponibile per eseguire qualsiasi processo presente nella coda. Anche i multiprocessori omogenei, talvolta, possono incorrere in limitazioni nello scheduling. Si consideri un sistema con un dispositivo di I/O collegato a un processore mediante un bus privato; ogni processo che intenda avvalersi di tale dispositivo dovrà essere pianificato per l'esecuzione su quel processore.

6.5.1 Approcci allo scheduling per multiprocessori

Una prima strategia di scheduling della CPU per i sistemi multiprocessore affida tutte le decisioni, l'elaborazione dell'I/O e altre attività del sistema a un solo processore, il cosiddetto *master server*. Gli altri processori eseguono soltanto il codice utente. Si tratta della **multielaborazione asimmetrica**, che riduce la necessità di condividere dati grazie all'accesso di un solo processore alle strutture dati del sistema.

Quando invece ciascun processore provvede al proprio scheduling, si parla di **multielaborazione simmetrica** (*symmetric multiprocessing*, SMP). In questo caso i processi pronti per l'esecuzione sono situati tutti in una ready queue comune, oppure vi è un'apposita coda per ogni processore. In entrambi i casi, lo scheduler di ciascun processore esamina la coda appropriata, da cui seleziona un processo da eseguire. Come abbiamo visto nel Capitolo 5, l'accesso concorrente di più processori a una struttura dati comune rende delicata la programmazione dello scheduler, al fine di evitare che due processori scelgano il medesimo processo o che un processo in coda non vada perso. La SMP è messa a disposizione da quasi tutti i sistemi operativi moderni, quali Windows, Linux e Mac OS X.

Nei paragrafi successivi discuteremo le questioni concernenti i sistemi SMP.

6.5.2 Predilezione per il processore

Si consideri che cosa accade alla memoria cache dopo che un processo sia stato eseguito da uno specifico processore: i dati che il processore ha trattato più recentemente permangono nella cache e, di conseguenza, i successivi accessi alla memoria da parte del processo tendono a utilizzare spesso la memoria cache. Si consideri ora che cosa succede se un processo si sposta su un altro processore: i contenuti della memoria cache devono essere invalidati sul processore di partenza, mentre la cache del processore di arrivo deve essere nuovamente riempita. A causa degli alti costi di svuotamento e riempimento della cache, molti sistemi SMP tentano di impedire il passaggio di

processi da un processore all'altro, mirando a mantenere un processo sempre sullo stesso processore. Si parla di **predilezione per il processore** (*processor affinity*), intendendo con ciò che un processo ha una predilezione per il processore su cui è in esecuzione.

La predilezione per il processore può assumere varie forme. Quando un sistema operativo si propone di mantenere un processo su un singolo processore, ma non garantisce che sarà così, si parla di **predilezione debole** (*soft affinity*). In questo caso il sistema operativo tenta di mantenere il processo su un singolo processore, ma è possibile che un processo migri da un processore all'altro. Alcuni sistemi dispongono di chiamate di sistema per realizzare la **predilezione forte** (*hard affinity*), permettendo così a un processo di specificare un sottoinsieme di processori su cui può essere eseguito. Molti sistemi supportano sia la predilezione debole che la predilezione forte. Linux, per esempio, implementa la predilezione debole, ma fornisce anche la chiamata `sched_setaffinity()` per il supporto della predilezione forte.

L'architettura della memoria principale di un sistema può influenzare le questioni relative alla predilezione. La Figura 6.9 mostra un'architettura con accesso non uniforme alla memoria (NUMA) in cui la CPU ha un accesso più rapido ad alcune zone di memoria rispetto ad altre. Di solito una situazione di questo tipo si ha in sistemi dove sono presenti diverse schede, ognuna con CPU e memoria. Le CPU su una scheda possono accedere alla memoria sulla stessa scheda con meno ritardo rispetto a quella su schede diverse. Se lo scheduler della CPU di un sistema operativo e gli algoritmi di allocazione della memoria lavorano insieme, allora ad un processo con predilezione per una determinata CPU può essere allocata memoria residente sulla stessa scheda in cui è montata la CPU. Questo esempio mostra anche come i sistemi operativi non siano definiti e implementati in maniera tanto nitida quanto è descritto nei libri di testo. Le linee di demarcazione tra le componenti di un sistema operativo, lungi dall'essere nette, sono sfumate; opportuni algoritmi instaurano poi connessioni fra le varie parti allo scopo di ottimizzare le prestazioni e l'affidabilità.

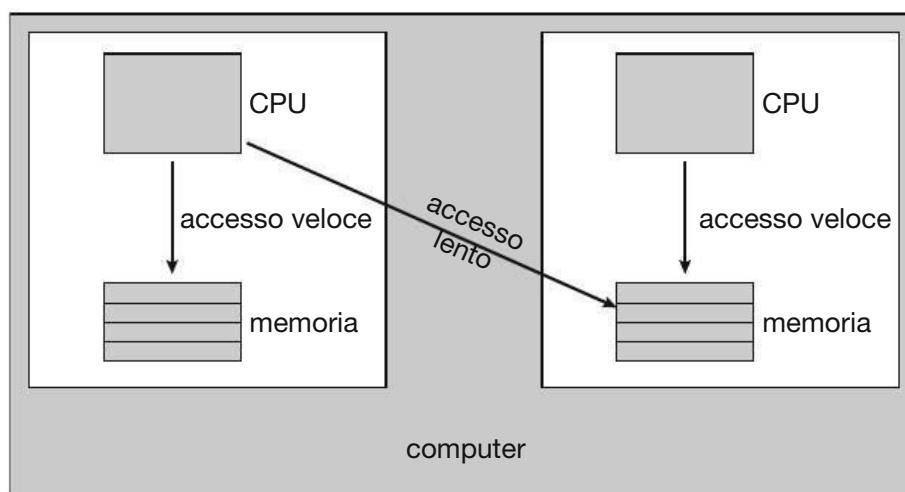


Figura 6.9 NUMA e lo scheduling della CPU.

6.5.3 Bilanciamento del carico

Sui sistemi SMP è importante che il carico di lavoro sia distribuito equamente tra tutte le unità elaborative per sfruttare appieno i vantaggi di avere più processori. Se ciò non avviene, alcuni processori potrebbero restare inattivi mentre altri verrebbero intensamente sfruttati con una coda di processi in attesa. Il **bilanciamento del carico** tenta di ripartire il carico di lavoro uniformemente tra tutti i processori di un sistema SMP. Bisogna notare che il bilanciamento è necessario, di norma, solo nei sistemi in cui ciascun processore ha una coda privata di processi eseguibili. Nei sistemi che mantengono una coda comune, il bilanciamento del carico è sovente superfluo: un processore inattivo passerà immediatamente all'esecuzione di un processo dalla coda comune dei processi eseguibili. Va ricordato, tuttavia, che in quasi tutti i sistemi operativi attuali che supportano l'SMP, ogni processore è effettivamente dotato di una propria coda di processi eseguibili.

Il bilanciamento del carico può seguire due approcci: la **migrazione push** e la **migrazione pull**. La prima prevede che un processo apposito controlli periodicamente il carico di ogni processore: se identifica uno sbilanciamento, riporta il carico in equilibrio, spostando i processi dal processore saturo ad altri più liberi, o inattivi. La migrazione pull, invece, si ha quando un processore inattivo prende un processo in attesa a un processore sovraccarico. I due tipi di migrazione non sono mutuamente esclusivi, e trovano spesso applicazione contemporanea nei sistemi con bilanciamento del carico. Lo scheduler Linux, per esempio, che vedremo nel Paragrafo 6.7.1, e lo scheduler ULE dei sistemi FreeBSD, si avvalgono di entrambe le tecniche.

Una singolare proprietà del bilanciamento del carico è di andare spesso a discapito del vantaggio derivante dalla predilezione del processore, analizzata nel Paragrafo 6.5.2. Il vantaggio di poter eseguire un processo dall'inizio alla fine sul medesimo processore è che in tal modo la memoria cache contiene i dati necessari per quel processo. Con la migrazione dei processi necessaria al bilanciamento del carico, questo vantaggio si perde. Anche in questo frangente, come di consueto nell'ingegneria dei sistemi, non vi sono regole che prescrivano una strategia ottimale: in alcuni sistemi, un processore inattivo sottrae sempre un processo da un processore impegnato; in altri, i processi sono spostati solo se la disparità del carico supera una data soglia.

6.5.4 Processori multicore

Tradizionalmente i sistemi SMP hanno reso possibile la concorrenza tra thread con l'utilizzo di diversi processori fisici. Tuttavia, la pratica recente nel progetto hardware dei calcolatori è di inserire più unità di calcolo in un unico chip fisico, dando origine a un **processore multicore**. Ogni core mantiene il proprio stato e appare dunque al sistema operativo come un processore fisico separato. I sistemi SMP che usano processori multicore sono più veloci e consumano meno energia dei sistemi in cui ciascun processore è costituito da un singolo chip.

I processori multicore possono complicare i problemi relativi allo scheduling. Proviamo a vedere che cosa può succedere. Le ricerche hanno permesso di scoprire che

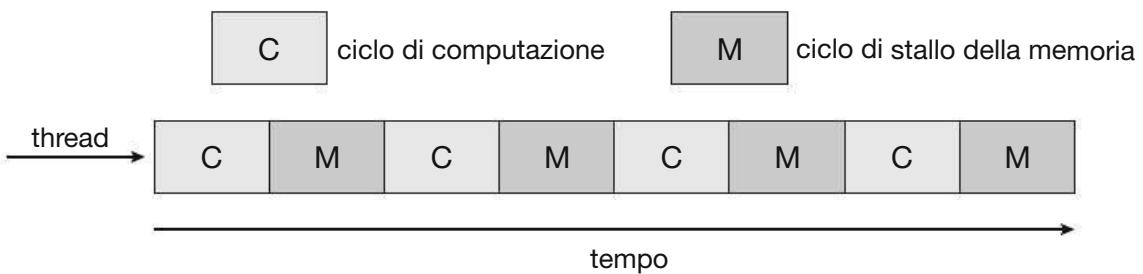


Figura 6.10 Stallo della memoria.

quando un processore accede alla memoria, una quantità significativa di tempo trascorre in attesa della disponibilità dei dati. Questa situazione, nota come **stallo della memoria**, può verificarsi per varie ragioni, per esempio la mancanza dei dati richiesti nella cache. La Figura 6.10 mostra uno stallo della memoria. In questo scenario, il processore può trascorrere fino al 50 per cento del suo tempo attendendo che i dati siano disponibili in memoria. Per rimediare a questa situazione, molti dei progetti hardware recenti implementano delle unità di calcolo multithread in cui due o più thread hardware sono assegnati a un singolo core. In questo modo, se un thread è in situazione di stallo in attesa della memoria, il core può passare ad eseguire un altro thread. La Figura 6.11 mostra un processore a due thread nel quale l'esecuzione dei thread 0 e 1 sono avvicendate nel tempo. Dal punto di vista del sistema operativo, ogni thread hardware appare come un processore logico in grado di eseguire un thread software. In un sistema a due thread con due unità di calcolo il sistema operativo vede dunque quattro processori logici. La CPU UltraSPARC T3 monta sedici core per singolo chip, e otto thread hardware per ogni unità di calcolo; dalla prospettiva del sistema operativo si vedono 128 processori logici.

In generale, ci sono due modi per rendere un core multithread: attraverso il **multithreading a grana grossa** (*coarse-grained*) o il **multithreading a grana fine** (*fine-grained*). Nel multithreading coarse-grained un thread resta in esecuzione su un processore fino al verificarsi di un evento a lunga latenza, per esempio uno stallo di memoria. A causa dell'attesa introdotta dall'evento a lunga latenza, il processore deve passare a un altro thread e iniziare a eseguirlo. Tuttavia, il costo per cambiare il thread

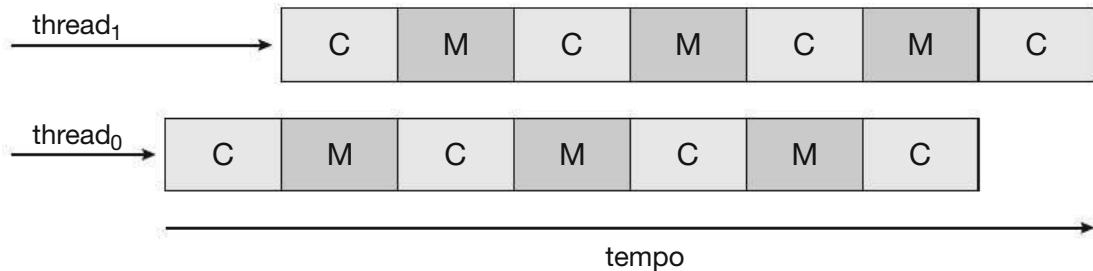


Figura 6.11 Sistema multicore e multithread.

in esecuzione è alto, perché occorre ripulire la pipeline delle istruzioni prima che il nuovo thread possa iniziare a essere eseguito sull'unità di calcolo. Una volta che il nuovo thread è in esecuzione inizia a riempire la pipeline con le sue istruzioni. Il multithreading fine (anche detto *interleaved multithreading*) passa da un thread a un altro ad un livello molto più fine di granularità (tipicamente al termine di un ciclo di istruzione). Tuttavia, il progetto di sistemi a multithreading fine include una logica dedicata al cambio di thread. Ne risulta così che il costo del passaggio da un thread a un altro è piuttosto basso.

Va osservato che un processore multicore multithreaded richiede due diversi livelli di scheduling. A un primo livello vi sono le decisioni di scheduling che devono essere prese dal sistema operativo per stabilire quale thread software mandare in esecuzione su ciascun thread hardware (processore logico). Per realizzare questo tipo di scheduling il processore può scegliere qualunque algoritmo, per esempio quelli descritti nel Paragrafo 6.3. Un secondo livello di scheduling specifica come ogni core decida quale thread hardware eseguire. Ci sono diverse strategie che si possono adottare in questa situazione. Il processore UltraSPARC T3, menzionato prima, usa un semplice algoritmo circolare (round-robin) per lo scheduling degli otto thread hardware su ogni core. Un altro esempio è l'Intel Itanium, una CPU dual-core con due thread gestiti dall'hardware per ogni core. Un valore dinamico di *urgency*, compreso tra 0 e 7, dove 0 rappresenta l'urgenza minore e 7 la più alta, viene assegnato a ogni thread hardware del processore. L'Itanium identifica cinque diversi eventi che possono far scattare un cambio di thread. Quando uno di questi eventi si verifica, la logica preposta al cambio di thread confronta l'urgenza dei due thread che dovrebbero scambiarsi e sceglie di mandare in esecuzione sull'unità di calcolo il thread con il valore più alto.

6.6 Scheduling real-time della CPU

Lo scheduling della CPU nei sistemi real-time ha peculiarità proprie. In generale, possiamo distinguere tra sistemi real-time soft e sistemi real-time hard. I **sistemi real-time soft** non offrono garanzie sul momento in cui un processo critico sarà eseguito, ma assicurano solamente che sarà data precedenza a quest'ultimo piuttosto che ad altri processi non critici. I **sistemi real-time hard** hanno vincoli più rigidi: i task vanno eseguiti entro una scadenza prefissata ed eseguirli dopo tale scadenza è del tutto inutile. In questo paragrafo analizziamo diversi aspetti relativi allo scheduling nei sistemi real-time soft e hard.

6.6.1 Minimizzazione della latenza

I sistemi real-time sono per loro natura guidati dagli eventi: generalmente, il sistema attende che si verifichi un evento in tempo reale. Quest'ultimo può avere luogo a livello software – come quando scatta un timer – o hardware – come quando un veicolo controllato a distanza si accorge di essere vicino a un ostacolo. In presenza di un even-

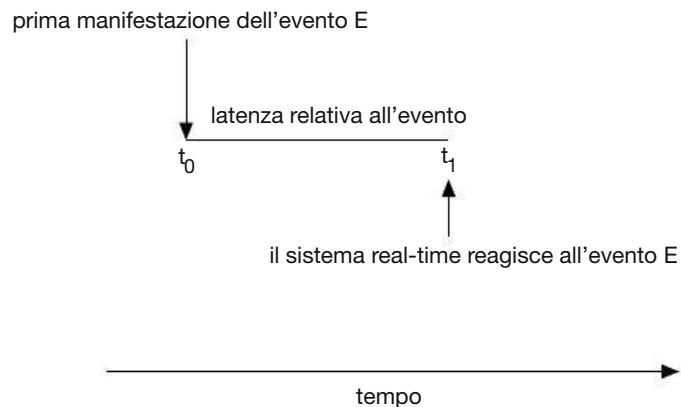


Figura 6.12 Latenza relativa all'evento.

to, il sistema deve rispondere con la massima velocità possibile. Chiameremo **latenza relativa all'evento** (*event latency*) il periodo di tempo che intercorre tra l'occorrenza dell'evento e il momento in cui il sistema ne effettua la gestione (Figura 6.12).

In genere, eventi diversi hanno diversi requisiti di latenza. Per esempio, la latenza per un sistema antiblocco dei freni potrebbe essere da tre a cinque millisecondi; ciò significa che dal momento in cui una ruota avverte che sta scivolando, il meccanismo che controlla il blocco dei freni ha da tre a cinque millesimi di secondo per reagire e controllare la situazione. Una risposta più lenta potrebbe causare lo sbandamento e la perdita di controllo del veicolo. Un sistema integrato di controllo del radar in un aeroplano passeggeri, invece, potrebbe tollerare una latenza di alcuni secondi.

Le categorie di latenza che influiscono sul funzionamento dei sistemi real-time sono due:

1. latenza relativa alle interruzioni;
2. latenza relativa al dispatch.

La **latenza relativa all'interruzione** si riferisce al periodo di tempo compreso tra la notifica di un'interruzione alla CPU e l'avvio della routine che gestisce l'interruzione. Quando sopraggiunge un'interruzione, il sistema operativo deve in primo luogo portare a compimento l'istruzione che sta eseguendo, e determinare di quale tipo di interruzione si tratti. Successivamente deve salvare lo stato del processo che è in atto, prima di occuparsi dell'interruzione tramite l'apposita procedura di gestione (ISR). Il tempo complessivamente impiegato per svolgere questi task è definito come la latenza relativa all'interruzione (Figura 6.13).

Evidentemente, per garantire l'immediata presa in consegna delle operazioni real-time è cruciale che un sistema real-time riduca al minimo la latenza relativa alle interruzioni. Nei sistemi real-time hard la latenza relativa alle interruzioni non deve essere semplicemente minimizzata, ma deve rispettare un limite superiore, per rispondere alle rigide esigenze di questo tipo di sistemi.

Un fattore importante che incide sulla latenza relativa alle interruzioni è l'intervallo di tempo in cui le interruzioni sono disattivate durante l'aggiornamento delle

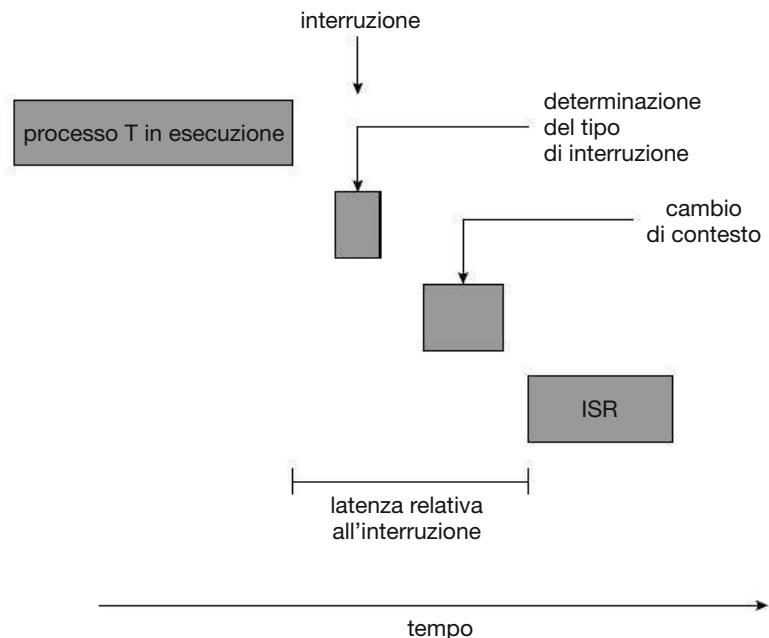


Figura 6.13 Latenza relativa alle interruzioni.

strutture dati del kernel. I sistemi operativi real-time esigono che le interruzioni siano inibite soltanto per intervalli di tempo molto piccoli.

Il periodo di tempo necessario al dispatcher per bloccare un processo e avviare un altro è noto come *latenza di dispatch*. Per garantire ai processi real-time l'accesso immediato alla CPU è necessario che i sistemi operativi minimizzino anche questo tipo di latenza. La tecnica più efficace per mantenere bassa la latenza relativa al dispatch consiste nell'implementare kernel con prelazione.

Il grafico nella Figura 6.14 mostra la composizione della latenza relativa al dispatch. La fase di conflitto della latenza di dispatch consiste di due componenti:

1. prelazione di ogni processo in esecuzione nel kernel;
 2. cessione, da parte dei processi a bassa priorità, delle risorse richieste dal processo ad alta priorità.

Nel sistema Solaris, per esempio, la latenza relativa al dispatch con prelazione inibita è di oltre 100 millisecondi; se la prelazione è attiva, si riduce a meno di un millisecondo.

6.6.2 Scheduling basato sulla priorità

La caratteristica più importante di un sistema operativo real-time è di rispondere immediatamente a un processo in tempo reale, non appena questo processo richiede la CPU. Lo scheduler di un sistema operativo real-time deve dunque utilizzare un algoritmo con prelazione basato su priorità. Ricordiamo che gli algoritmi di scheduling con priorità assegnano a ogni processo una priorità in base alla sua importanza; ai task più importanti vengono assegnate priorità più elevate rispetto a quelle assegnate

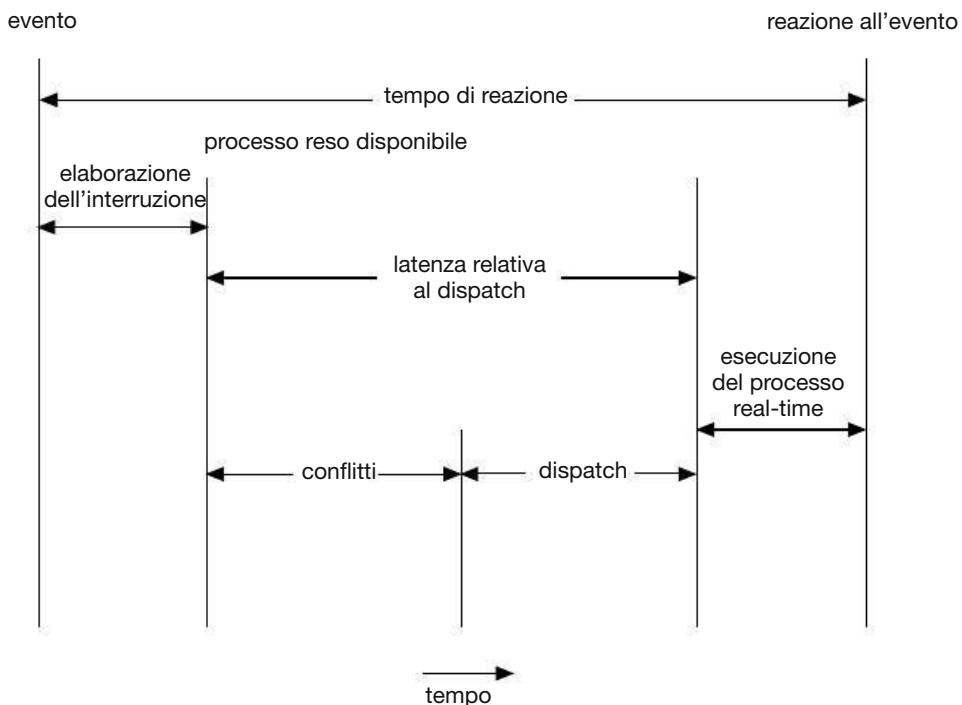


Figura 6.14 Latenza relativa al dispatch.

ai processi ritenuti meno importanti. Se lo scheduler supporta anche la prelazione, un processo in esecuzione sulla CPU viene interrotto se diventa disponibile per l'esecuzione un processo con priorità più alta.

Gli algoritmi di scheduling con prelazione basati sulla priorità sono discussi in dettaglio nel Paragrafo 6.3.3 e il Paragrafo 6.7 presenta esempi delle funzionalità di scheduling real-time soft nei sistemi operativi Linux, Windows e Solaris. Tutti questi sistemi assegnano ai processi in tempo reale la massima priorità. Per esempio, Windows ha 32 livelli di priorità diversi. I livelli più alti, con valori di priorità da 16 a 31, sono riservati ai processi real-time. Solaris e Linux hanno schemi di priorità simili.

Si noti che uno scheduler con prelazione basato sulla priorità garantisce solo la funzionalità soft real-time. I sistemi real-time hard devono anche garantire che le attività in tempo reale saranno servite rispettando le scadenze. Per soddisfare a questo requisito lo scheduler deve avere funzionalità addizionali. Nel resto di questo paragrafo ci occupiamo di algoritmi di scheduling per sistemi real-time hard.

Prima di procedere con i dettagli dei singoli scheduler, dobbiamo tuttavia definire alcune caratteristiche dei processi che devono essere schedulati.

Innanzitutto, i processi sono considerati **periodici**, nel senso che richiedono la CPU a intervalli costanti di tempo (periodi). Ciascun processo periodico, una volta avuto accesso alla CPU, ha un tempo di elaborazione fisso t , una scadenza d entro cui la CPU deve completare la sua esecuzione e un periodo p . La relazione tra il tempo di elaborazione, la scadenza e il periodo si può riassumere nelle diseguaglianze $0 \leq t \leq d \leq p$. La frequenza di un processo periodico è data da $1/p$. La Figura 6.15 mostra

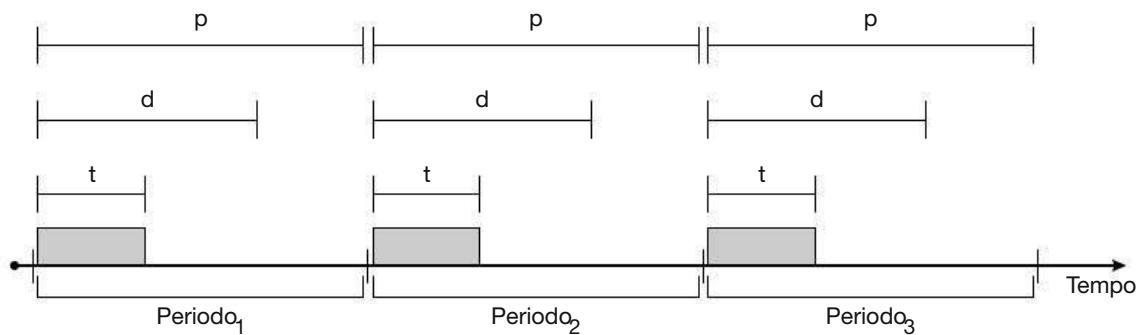


Figura 6.15 Processo periodico.

l'esecuzione di un processo periodico nel corso del tempo. Lo scheduler può trarre vantaggio da queste caratteristiche, assegnando le priorità in base alla scadenza o ai requisiti di frequenza di un processo.

L'aspetto inusuale di questo tipo di scheduling è il fatto che un processo può dover dichiarare allo scheduler la propria scadenza d . Sulla base di questa informazione, grazie a una tecnica nota come **algoritmo di controllo dell'ammissione**, lo scheduler decide se accettare il processo – con la garanzia di eseguirne le richieste in tempo – o rifiutarlo, qualora non sia certo di poterne soddisfare le richieste entro la scadenza relativa.

6.6.3 Scheduling con priorità proporzionale alla frequenza

L'algoritmo di scheduling con priorità proporzionale alla frequenza programma i task periodici applicando un modello statico di attribuzione delle priorità con prelazione. Se un processo con priorità elevata diventa pronto per l'esecuzione mentre è in esecuzione un processo con priorità più bassa, il primo avrà diritto di prelazione. Entrando nel sistema, ciascun task periodico si vede assegnare una priorità inversamente proporzionale al proprio periodo: più breve è il periodo, più alta la priorità; più lungo il periodo, più bassa la priorità. La logica di questa regola consiste nell'assegnare priorità più elevate ai processi che fanno uso più frequente della CPU. Per di più, lo scheduling con priorità proporzionale alla frequenza si fonda sul presupposto che il tempo di elaborazione di un processo periodico resti uguale per ogni CPU burst: ogni volta che il processo utilizza la CPU, dunque, la durata della sua esecuzione è la stessa.

Consideriamo, per esempio, due processi P_1 e P_2 , i cui periodi siano rispettivamente $p_1 = 50$ e $p_2 = 100$. Supponiamo che i tempi di elaborazione siano $t_1 = 20$ e $t_2 = 35$, rispettivamente. La scadenza di ogni processo impone loro di terminare l'esecuzione entro l'inizio del periodo seguente.

Per prima cosa dobbiamo chiederci se sia possibile schedulare l'esecuzione di ciascun task in modo da rispettare la sua scadenza. Se misuriamo l'utilizzo percentuale della CPU da parte di un processo P_i con il quoziente t_i/p_i , l'utilizzo della CPU da parte

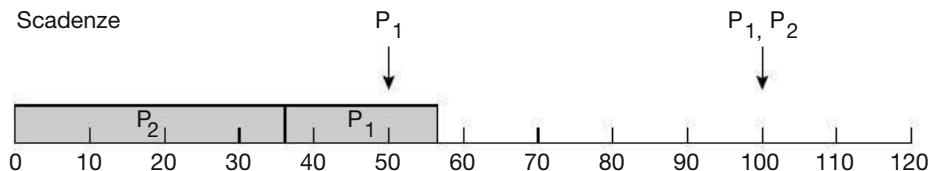


Figura 6.16 Scheduling dei task in caso P_2 abbia priorità maggiore di P_1 .

di P_1 è $20/50 = 0,40$, mentre quello di P_2 è $35/100 = 0,35$, per un utilizzo totale della CPU del 75 per cento. Sembra quindi che il problema di scheduling sia risolubile rispettando le scadenze dei processi, e lasciando anche dei cicli di CPU disponibili.

Per cominciare, ipotizziamo di assegnare a P_2 una priorità più alta che a P_1 . L'esecuzione di P_1 e P_2 è illustrata dalla Figura 6.16. Come si può vedere, P_2 inizia l'esecuzione per primo e la termina dopo 35 millisecondi. A questo punto, è il turno di P_1 , che completa la sua esecuzione al millisecondo 55; ma la prima scadenza di P_1 scoccava al millisecondo 50: questa politica di scheduling non permette a P_1 di rispettare la propria scadenza.

Proviamo adesso ad applicare lo scheduling con priorità proporzionale alla frequenza, in base al quale assegniamo a P_1 una priorità più elevata rispetto a P_2 , dato che il suo periodo è più breve. L'esecuzione di questi processi è mostrata nella Figura 6.17. Dapprima è eseguito P_1 , che termina al millisecondo 20, rispettando così la sua prima scadenza. Quindi, è il turno di P_2 , che prosegue fino al millisecondo 50. A questo punto, nonostante debba ancora usare la CPU per 5 millisecondi, P_2 lascia il posto per prelazione a P_1 . Quest'ultimo conclude la sua elaborazione al millisecondo 70, momento in cui lo scheduler riavvia P_2 , che termina al millisecondo 75, rispettando anch'esso la prima scadenza. Il sistema rimane inattivo fino al millisecondo 100, quando P_1 è nuovamente eseguito.

Lo scheduling con priorità proporzionale alla frequenza è un algoritmo ottimale, nel senso che se non è in grado di pianificare l'esecuzione di una serie di processi rispettandone i vincoli temporali, nessun altro algoritmo che assegna priorità statiche vi riuscirà. Consideriamo, ora, un insieme di processi la cui esecuzione non può essere schedulata dall'algoritmo con priorità proporzionale alla frequenza rispettandone i vincoli temporali.

Supponiamo che il processo P_1 abbia periodo $p_1 = 50$ e tempo d'esecuzione $t_1 = 25$, e che il processo P_2 abbia invece parametri corrispondenti pari a $p_2 = 80$ e $t_2 = 35$. Lo scheduling con priorità proporzionale alla frequenza assegnerebbe a P_1 la

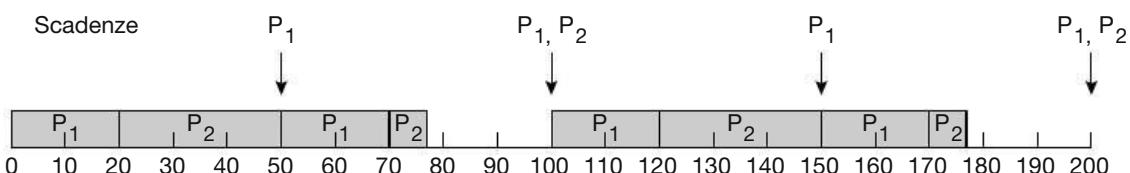


Figura 6.17 Scheduling con priorità proporzionale alla frequenza.

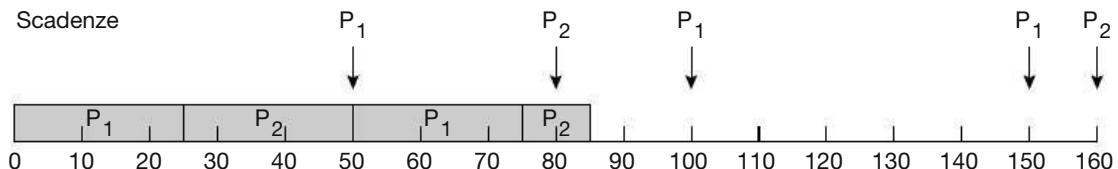


Figura 6.18 Scadenze non rispettate con lo scheduling con priorità proporzionale alla frequenza.

priorità più elevata, visto che ha il periodo più breve. Nel complesso la CPU è utilizzata dai due processi per $(25/50)+(35/80) = 0,94$, quindi sembra ragionevole attendersi che il problema di scheduling abbia soluzione, lasciando alla CPU un 6 per cento di tempo libero. La Figura 6.18 illustra lo scheduling dei processi P₁ e P₂. Nella fase iniziale, P₁ termina la sua prima esecuzione al millisecondo 25. Il processo P₂, invece, va avanti fino al millisecondo 50, e qui si arresta perché P₁ fa valere il diritto di prelazione. A questo punto P₂ ha bisogno di altri 10 millisecondi per terminare la sua esecuzione. Il processo P₁ continua a girare fino al millisecondo 75, ma P₂ viola la sua scadenza, che scoccava al millisecondo 80.

Lo scheduling con priorità proporzionale alla frequenza, dunque, benché ottimale nel senso descritto sopra, presenta una limitazione: l'utilizzo della CPU è limitato, per cui non sempre è possibile ricavarne un rendimento ottimale. Il caso peggiore di utilizzo della CPU per lo scheduling di N processi ammonta a

$$N(2^{1/N} - 1)$$

Con un solo processo nel sistema, la CPU è sfruttata al 100 per cento, ma la percentuale scende al 69 per cento circa quando il numero di processi tende all'infinito. Con due processi l'utilizzo della CPU è limitato a circa l'83 per cento. L'utilizzo congiunto della CPU per i due processi programmati nelle Figure 6.16 e 6.17 ammonta al 75 per cento; pertanto, lo scheduling con priorità proporzionale alla frequenza ne garantisce l'esecuzione entro le rispettive scadenze. Per i due processi nella Figura 6.18 l'utilizzo complessivo della CPU è del 94 per cento circa; di conseguenza, lo scheduling con priorità proporzionale alla frequenza non è in grado di pianificare l'esecuzione nel rispetto dei loro vincoli temporali.

6.6.4 Scheduling EDF

Lo scheduling EDF (earliest-deadline-first, ossia “per prima la scadenza più ravvicinata”), attribuisce le priorità dinamicamente, sulla base delle scadenze. Più vicina è la scadenza, maggiore è la priorità; una scadenza più lontana implica una priorità più bassa. Nel modello EDF, un processo pronto per l'esecuzione deve notificare al sistema la propria scadenza. Potrà essere allora necessario modificare le priorità vigenti per tenere conto della scadenza del nuovo processo eseguibile. Si osservi come questo procedimento sia diverso dallo scheduling con priorità proporzionale alla frequenza, che ipotizza priorità fisse.

Per illustrare lo scheduling EDF ci serviamo nuovamente dei processi raffigurati

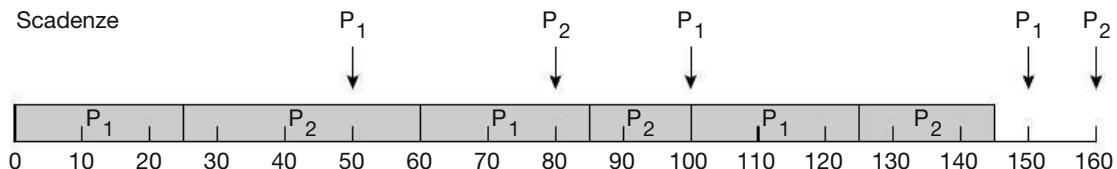


Figura 6.19 Scheduling EDF.

nella Figura 6.18, per i quali lo scheduling con priorità proporzionale alla frequenza risultava inadeguato. Ricordiamo che i parametri di P_1 e P_2 sono $p_1 = 50$, $t_1 = 25$ e $p_2 = 80$, $t_2 = 35$, rispettivamente. Lo scheduling EDF di questi processi è mostrato nella Figura 6.19. Il processo P_1 ha la scadenza più vicina, quindi parte con una priorità più alta rispetto a P_2 . Non appena P_1 termina l'esecuzione, il processo P_2 inizia a girare. Laddove, però, lo scheduling con priorità proporzionale alla frequenza consente a P_1 di esercitare prelazione nei confronti di P_2 all'inizio del suo periodo successivo, cioè al millisecondo 50, lo scheduling EDF lascia che P_2 continui a girare. Infatti, P_2 ha adesso priorità più alta rispetto a P_1 , perché la sua prossima scadenza (al millisecondo 80) precede quella di P_1 (al millisecondo 100). Pertanto, sia P_1 sia P_2 hanno onorato le loro prime scadenze. Il processo P_1 ritorna in esecuzione al millisecondo 60, e termina al millisecondo 85, rispettando anche la seconda scadenza, che scatta al millisecondo 100. È a questo punto che P_2 comincia a girare, ma subisce prelazione da parte di P_1 , che è all'inizio del suo periodo successivo, al millisecondo 100. La prelazione si applica perché la scadenza di P_1 (al millisecondo 150) è più vicina di quella di P_2 (al millisecondo 160). Al millisecondo 125 P_1 termina l'esecuzione e P_2 riparte, completando al millisecondo 145, e ottemperando così alla sua scadenza. Il sistema rimane inattivo fino al millisecondo 150, allorché P_1 è ancora una volta pronto per l'esecuzione.

A differenza dell'algoritmo con priorità proporzionale alla frequenza, lo scheduling EDF non postula la periodicità dei processi, e non prevede neanche di impiegare sempre la stesso tempo della CPU per ogni burst. L'unico obbligo a carico dei processi è di notificare allo scheduler la propria prossima scadenza nel momento in cui divengano eseguibili. Il vantaggio dello scheduling EDF è di essere teoricamente ottimo. Idealmente, esso garantisce l'esecuzione di tutti i processi entro le proprie scadenze, sfruttando inoltre la CPU al 100 per cento. Nella pratica, tuttavia, un rendimento così alto della CPU è impossibile, per il rallentamento dovuto ai cambi di contesto tra processi e alla gestione delle interruzioni.

6.6.5 Scheduling a quote proporzionali

Lo **scheduler a quote proporzionali** opera distribuendo un certo numero di quote, diciamo T , fra tutte le applicazioni. Un'applicazione può ricevere N quote di tempo, assicurandosi così l'uso di una frazione N/T del tempo totale del processore. Poniamo, per esempio, di avere un totale di quote $T = 100$, da ripartire fra tre processi, A , B e C .

A può disporre di 50 quote, B di 15, mentre C ha 20 quote. Questa suddivisione assicura che A possa sfruttare il 50 percento del tempo totale del processore, B il 15 percento e C il 20 percento.

Lo scheduler a quote proporzionali deve lavorare in sinergia con un meccanismo di controllo dell’ammissione, per garantire che ogni applicazione possa effettivamente ricevere le quote di tempo che le sono state destinate. Il meccanismo di controllo dell’ammissione accetterà le richieste da parte dei processi solo a condizione che il numero di quote disponibili sia sufficiente. Nell’esempio sopra citato, abbiamo assegnato $50 + 15 + 20 = 85$ quote su un totale di 100; se un nuovo processo D ne richiedesse 30, il meccanismo di controllo dell’ammissione gli negherebbe l’accesso al sistema.

6.6.6 Scheduling real-time di POSIX

Lo standard POSIX fornisce un’estensione per l’elaborazione real-time, detta POSIX.1b. In questo paragrafo si descrivono alcune API di POSIX relative alla programmazione real-time dei thread. POSIX definisce due classi di scheduling per i thread real-time:

- SCED_FIFO;
- SCED_RR.

SCED_FIFO pianifica l’esecuzione dei thread secondo la politica FCFS esposta al Paragrafo 6.3.1. Non è prevista alcuna forma di time slicing fra thread di pari priorità. Pertanto, il thread real-time con priorità più elevata in prima posizione nella coda FIFO avrà accesso alla CPU fin quando termina o si arresta. SCED_RR adotta una politica round-robin ed è simile a SCED_FIFO, eccetto che prevede il time slicing fra thread di uguale priorità. POSIX fornisce una terza classe di scheduling – SCED_OTHER – la cui implementazione è indefinita e dipendente dal sistema; il suo comportamento può variare da sistema a sistema.

La API POSIX specifica le due funzioni seguenti per leggere e impostare le politiche di scheduling:

- `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
- `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

In entrambe le funzioni il primo parametro è un puntatore all’insieme degli attributi del thread. Il secondo parametro è, nel primo caso, un puntatore a un intero impostato alla politica di scheduling corrente; nel secondo caso, un valore intero che denota SCED_FIFO, SCED_RR o SCED_OTHER. Entrambe le funzioni restituiscono valori non nulli in caso di errore.

La Figura 6.20 contiene un programma POSIX Pthread che usa questa API. Il programma appura in primo luogo la politica di scheduling vigente, per scegliere poi l’algoritmo di scheduling SCED_FIFO.

```

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* appura gli attributi di default */
    pthread_attr_init(&attr);

    /* appura la politica di scheduling corrente */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    /* imposta la politica di scheduling - FIFO, RR o OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)
        fprintf(stderr, "Unable to set policy.\n");

    /* genera i thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* ora attende la terminazione di ciascun thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* ciascun thread comincia l'esecuzione da questa funzione */
void *runner(void *param)

{
    /* fai qualcosa ... */

    pthread_exit(0);
}

```

Figura 6.20 Scheduling real-time con la API POSIX.

6.7 Esempi di sistemi operativi

Procediamo ora nella descrizione dei criteri di scheduling per i sistemi operativi Linux, Windows e Solaris.

È importante notare che viene qui utilizzato il termine scheduling dei processi in senso generale. In realtà nei sistemi Solaris e Windows stiamo descrivendo lo *scheduling dei thread del kernel* e in Linux lo *scheduling dei task*.

6.7.1 Un esempio: scheduling di Linux

Lo scheduling dei processi in Linux ha avuto una storia interessante. Prima della versione 2.5, il kernel Linux utilizzava una variante dell'algoritmo tradizionale di scheduling di UNIX.

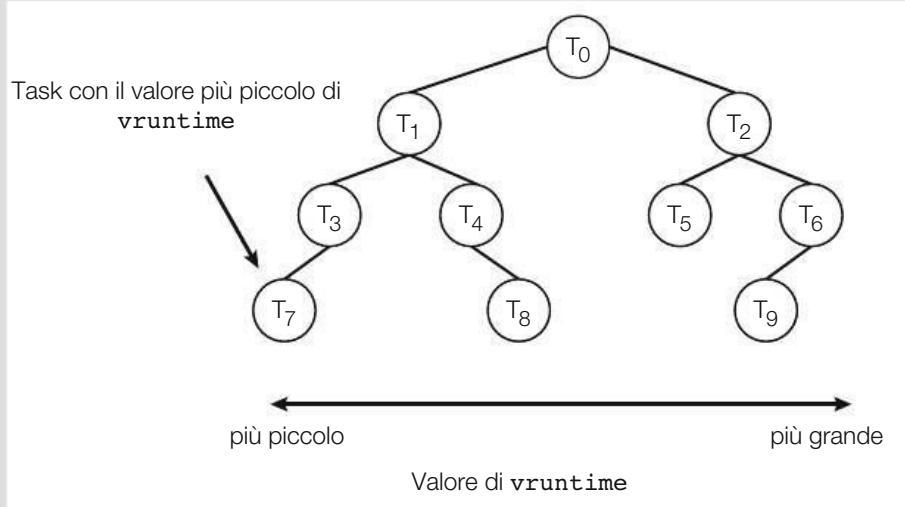
Tuttavia questo algoritmo, progettato senza pensare ai sistemi SMP, non supporta adeguatamente sistemi multiprocessore e fornisce prestazioni scarse nei sistemi in grado di eseguire un gran numero di processi. Con la versione 2.5 del kernel, lo scheduler è stato rivisitato per includere un algoritmo di scheduling noto come $O(1)$, che veniva eseguito in tempo costante indipendentemente dal numero di task nel sistema. Lo scheduler $O(1)$ forniva anche un maggiore supporto ai sistemi SMP fra cui la gestione della predilezione e il bilanciamento del carico. Tuttavia, in pratica, anche se lo scheduler $O(1)$ forniva eccellenti prestazioni su sistemi SMP, portava a tempi di risposta troppo scarsi sui processi interattivi, comuni in molti sistemi desktop. Durante lo sviluppo del kernel 2.6, lo scheduler è stato nuovamente rivisto e dalla versione 2.6.23 del kernel, lo scheduler CFS (*completely fair scheduler*) è diventato l'algoritmo predefinito di scheduling Linux.

Nei sistemi Linux lo scheduling si basa sulle **classi di scheduling**. A ogni classe è assegnata una specifica priorità. Utilizzando diverse classi di scheduling, il kernel può utilizzare algoritmi distinti in base alle esigenze del sistema e dei suoi processi. I criteri di scheduling per un server Linux, per esempio, possono essere diversi da quelli per un dispositivo mobile che utilizza Linux. Per decidere quale task eseguire, lo scheduler seleziona il task con priorità più alta appartenente alla classe di scheduling a priorità più elevata. Il kernel Linux standard implementa due classi di scheduling, che verranno discusse nel seguito: (1) una classe di scheduling predefinita che utilizza l'algoritmo CFS e (2) una classe di scheduling real-time. Possono naturalmente essere aggiunte ulteriori classi.

Invece di utilizzare regole rigide che associano un valore di priorità relativo alla lunghezza di un quanto di tempo, lo scheduler CFS assegna a ogni task una percentuale del tempo di elaborazione della CPU. Questa percentuale è calcolata sulla base dei valori **nice value** assegnati a ciascun task. I nice value vanno da -20 a +19, dove un valore numerico più basso indica una priorità relativa superiore. I task con nice value minori ricevono una maggiore percentuale di tempo di elaborazione della CPU rispetto ai task con nice value più alti. Il nice value di default è 0. Il termine nice value deriva dall'idea che se un task aumenta il suo nice value, per esempio da 0 a +10, si comporta in maniera gentile – in inglese nice – rispetto agli altri task del sistema, ab-

PRESTAZIONI DI CFS

Lo scheduler CFS di Linux fornisce un efficiente algoritmo per la selezione del prossimo task da eseguire. Ogni task eseguibile è posto in un R-B albero, un albero binario di ricerca bilanciato, la cui chiave si basa sul valore di `vruntime`. L'albero è il seguente:



Quando un task diventa eseguibile viene aggiunto all'albero. Se un task dell'albero non è eseguibile (per esempio, se è bloccato in attesa di I/O), viene rimosso. In generale, i task a cui è stato dato meno tempo di elaborazione (valori minori di `vruntime`) si trovano nel lato sinistro dell'albero e i task a cui è stato dato più tempo di elaborazione si trovano sul lato destro. Secondo le proprietà di un albero binario di ricerca il nodo più a sinistra ha il valore della chiave più piccolo, il che significa, per lo scheduler CFS, che è il task con la massima priorità. Poiché l'albero R-B è bilanciato, la ricerca del nodo più a sinistra richiede $O(\log N)$ operazioni (dove N è il numero di nodi dell'albero). Tuttavia, per ragioni di efficienza, lo scheduler Linux memorizza questo valore nella variabile `rb_leftmost` in modo da poter determinare qual è il prossimo task da eseguire recuperando semplicemente il valore memorizzato.

bassando la sua priorità relativa. CFS non utilizza valori discreti per i quanti di tempo, ma piuttosto definisce una **latenza obiettivo** (*targeted latency*), cioè un intervallo di tempo entro il quale ogni task eseguibile dovrebbe andare in esecuzione almeno una volta. Le porzioni di tempo di CPU vengono assegnate a partire dal valore della latenza obiettivo. La latenza obiettivo, che ha valore di default e un valore minimo, può aumentare qualora il numero di task attivi nel sistema cresca oltre una certa soglia.

Lo scheduler CFS non assegna direttamente le priorità, ma registra per quanto tempo ogni task è stato eseguito mantenendo il **tempo di esecuzione virtuale** di ogni task nella variabile `vruntime`. Il tempo di esecuzione virtuale è associato a un fattore di decadimento che dipende dalla priorità di un task: task a bassa priorità hanno fattori di decadimento più alti di task ad alta priorità. Per i task con priorità normale (nice value pari a 0), il tempo di esecuzione virtuale coincide con il tempo effettivo di esecuzione. Quindi, se un task con priorità normale viene eseguito per 200 millisecondi,

anche il suo `vruntime` sarà di 200 millisecondi, ma se un task con priorità inferiore viene eseguito per 200 millisecondi, il suo `vruntime` sarà superiore a 200 millisecondi. Analogamente, se un task con priorità più alta viene eseguito per 200 millisecondi, il suo `vruntime` sarà inferiore a 200 millisecondi. Per decidere il prossimo task da eseguire, lo scheduler seleziona semplicemente il task con il valore `vruntime` più piccolo. Inoltre, un task con priorità più alta che diventa disponibile per l'esecuzione può avere prelazione su un task con priorità inferiore.

Esaminiamo lo scheduler CFS in azione. Si supponga che due task abbiano lo stesso nice value. Un task è I/O-bound e l'altro è CPU-bound. In genere, un task I/O-bound verrà eseguito solo per brevi periodi di tempo prima di bloccarsi in attesa di I/O, mentre un task CPU-bound esaurirà il suo periodo di tempo ogni volta che avrà l'opportunità di essere in esecuzione su un processore. Pertanto, il valore di `vruntime` del task I/O-bound diventerà inferiore rispetto a quello del task CPU-bound, dando così al task I/O-bound una priorità superiore. A questo punto, se il task CPU-bound è in esecuzione quando il task I/O-bound diventa pronto per l'esecuzione (per esempio, quando l'I/O che il task sta attendendo diventa disponibile), il task I/O-bound effettuerà la prelazione del task CPU-bound.

Linux implementa anche lo scheduling in tempo reale utilizzando lo standard POSIX, come descritto nel Paragrafo 6.6.6. Qualsiasi task pianificato utilizzando le politiche real-time `SCHED_FIFO` o `SCHED_RR` viene eseguito con una priorità più alta rispetto ai normali task (non real-time). Linux usa due intervalli di priorità distinti, uno per i task real-time e l'altro per i normali task. Ai task real-time vengono assegnate priorità statiche nell'intervallo 0–99, mentre ai task non real-time vengono assegnate priorità nell'intervallo 100–139. Questi due intervalli sono mappati in uno schema di priorità globale in cui i valori numericamente inferiori indicano priorità relative più alte. Ai task normali vengono assegnate priorità in base ai loro nice value: il valore -20 viene mappato nella priorità 100 e il valore 19 nella priorità 139. Questo schema è mostrato in Figura 6.21.

6.7.2 Un esempio: scheduling di Windows

Il sistema operativo Windows compie lo scheduling dei thread servendosi di un algoritmo basato su priorità e prelazione. Lo scheduler di Windows assicura che si eseguano sempre i thread a più alta priorità. La porzione del kernel che si occupa dello scheduling si chiama *dispatcher*. Una volta selezionato dal *dispatcher*, un thread viene

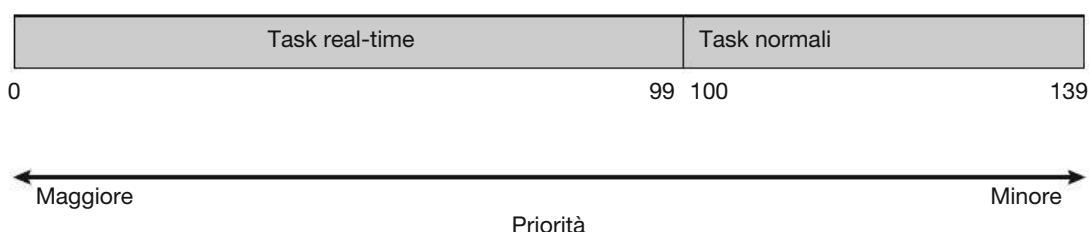


Figura 6.21 Priorità di scheduling in Linux.

eseguito finché non sia sottoposto a prelazione da un altro thread a priorità più alta oppure termini, esaurisca il suo quanto di tempo o esegua una chiamata di sistema bloccante, per esempio un'operazione di I/O. Se un thread d'elaborazione in tempo reale, ad alta priorità, diventa pronto per l'esecuzione mentre è in esecuzione un thread a bassa priorità, quest'ultimo viene sottoposto a prelazione. Ciò realizza un accesso preferenziale alla CPU per i thread d'elaborazione in tempo reale che ne hanno necessità.

Per determinare l'ordine d'esecuzione dei thread il *dispatcher* impiega uno schema di priorità a 32 livelli. Le priorità sono suddivise in due classi: la **classe variable** raccoglie i thread con priorità da 1 a 15, mentre la **classe real-time** raccoglie i thread con priorità tra 16 e 31 (esiste anche un thread, per la gestione della memoria, che si esegue con priorità 0). Il *dispatcher* adopera una coda per ciascuna priorità di scheduling e percorre l'insieme delle code da quella a priorità più alta a quella a priorità più bassa, finché trova un thread pronto per l'esecuzione. In assenza di tali thread, il *dispatcher* manda in esecuzione un thread speciale detto **idle thread**.

C'è una relazione tra le priorità numeriche del kernel del sistema operativo Windows e quelle dell'API Windows. Secondo l'API Windows un processo può appartenere a una delle seguenti classi di priorità:

- IDLE_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- REALTIME_PRIORITY_CLASS

Di solito, i processi appartengono alla classe NORMAL_PRIORITY_CLASS, sempre che il processo genitore non appartenga alla classe IDLE_PRIORITY_CLASS, o sia stata specificata un'altra classe alla creazione del processo. La classe di priorità di un processo può essere modificata mediante la funzione `SetPriorityClass()` dell'API Windows. Le priorità di ciascuna classe eccetto REALTIME_PRIORITY_CLASS sono priorità di classe variabile, quindi la priorità di un thread appartenente a queste classi può cambiare.

Un thread, nell'ambito di una classe di priorità, ha a sua volta una priorità relativa, i cui valori comprendono i seguenti:

- IDLE
- LOWEST
- BELOW_NORMAL
- NORMAL
- ABOVE_NORMAL
- HIGHEST
- TIME_CRITICAL

	realtime	high	above_normal	normal	below_normal	idle_priority
time_critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above_normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below_normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figura 6.22 Priorità dei thread in Windows.

La priorità di ciascun thread dipende dalla priorità della classe cui appartiene e dalla priorità relativa che il thread ha all'interno della stessa classe. Questa relazione è rappresentata nella Figura 6.22. I valori di ciascuna classe di priorità sono riportati nella prima riga in alto. La prima colonna a sinistra contiene i valori delle diverse priorità relative. Per esempio, se la priorità relativa di un thread nella classe ABOVE_NORMAL_PRIORITY_CLASS è NORMAL, la priorità numerica di quel thread è 10.

Inoltre, ogni thread ha una priorità di base che rappresenta un valore nell'intervallo di priorità della classe di appartenenza. Il valore predefinito per la priorità di base in una classe è quello della priorità relativa NORMAL per quella classe. Le priorità di base per ciascuna classe di priorità sono le seguenti:

- REALTIME_PRIORITY_CLASS–24
- HIGH_PRIORITY_CLASS–13
- ABOVE_NORMAL_PRIORITY_CLASS–10
- NORMAL_PRIORITY_CLASS–8
- BELOW_NORMAL_PRIORITY_CLASS–6
- IDLE_PRIORITY_CLASS–4.

Di solito la priorità iniziale di un thread è la priorità di base del processo a cui il thread appartiene, anche se può essere utilizzata la funzione `SetThreadPriority()` nella API di Windows per modificare la priorità di base di un thread.

Quando il quanto di tempo di un thread si esaurisce, il thread viene interrotto e se il thread fa parte della classe a priorità variabile, la sua priorità viene ridotta. Tuttavia, la priorità non si abbassa mai sotto la priorità di base. L'abbassamento della priorità tende a limitare l'uso della CPU da parte dei thread con prevalenza d'elaborazione. Se un thread a priorità variabile è rilasciato da un'operazione d'attesa, il *dispatcher* aumenta la sua priorità. L'entità di questo aumento dipende dal tipo d'evento che il

thread attendeva: un thread che attendeva dati dalla tastiera riceve un forte aumento di priorità, uno che attendeva operazioni relative a un disco riceve un aumento più moderato. Questa strategia mira a fornire buoni tempi di risposta per i thread interattivi, con interfacce basate su mouse e finestre; permette inoltre ai thread con prevalenza di I/O di tenere occupati i dispositivi di I/O, e rende nel contempo possibile l'utilizzo in background dei cicli di CPU inutilizzati da parte dei thread con prevalenza d'elaborazione. Questa strategia si segue in molti sistemi operativi in time sharing, compreso UNIX. Inoltre, per migliorare il tempo di risposta, la finestra attraverso cui l'utente sta interagendo ottiene un incremento di priorità.

Quando un utente richiede l'esecuzione di un programma interattivo, il sistema deve fornire al relativo processo prestazioni particolarmente elevate. Per questa ragione, il sistema Windows segue una regola specifica di scheduling per i processi della classe NORMAL_PRIORITY_CLASS. Il sistema operativo Windows distingue tra il *processo in foreground*, correntemente selezionato sullo schermo e i *processi in background*, che non sono attualmente selezionati. Quando un processo passa in foreground, Windows aumenta il suo quanto di tempo di un certo fattore, tipicamente pari a 3; ciò fa sì che il processo in foreground possa continuare la propria esecuzione per un tempo tre volte più lungo, prima che si abbia una prelazione dovuta al time sharing.

Windows 7 ha introdotto lo **scheduling in modalità utente** (UMS), che consente alle applicazioni di creare e gestire i thread in maniera indipendente dal kernel. Un'applicazione può quindi creare e schedulare più thread senza coinvolgere l'utilità di scheduling del kernel Windows. Per le applicazioni che creano un gran numero di thread lo scheduling in modalità utente è molto più efficiente rispetto a quello in modalità kernel, proprio perché non è necessario alcun intervento del kernel.

Le versioni precedenti di Windows fornivano una caratteristica simile che permetteva a diversi thread in modalità utente (detti **fibre**) di essere associati a un singolo thread del kernel. Tuttavia, le fibre erano di utilità pratica limitata. Una fibra non era in grado di effettuare chiamate alle API di Windows, perché tutte le fibre dovevano condividere il blocco di ambiente (TEB, *thread environment block*) del thread su cui erano in esecuzione. Ciò presentava un problema quando una funzione dell'API di Windows inseriva informazioni di stato per una fibra nel TEB, perché queste informazioni potevano essere sovrascritte da altre fibre. UMS ha superato questi problemi, fornendo a ogni thread in modalità utente il proprio contesto privato.

Inoltre, a differenza delle fibre, UMS non è destinato a essere utilizzato direttamente dal programmatore. La programmazione di uno scheduler in modalità utente può essere molto impegnativa. UMS non include un tale scheduler: gli scheduler provengono invece da librerie costruite su UMS. Per esempio, Microsoft fornisce la libreria ConcRT (*concurrency runtime*), un framework per la programmazione concorrente C++ progettato per il parallelismo basato sui task (Paragrafo 4.2) su processori multicore. ConcRT fornisce uno scheduler in modalità utente e alcune funzionalità per decomporre i programmi in task da pianificare successivamente sui core disponibili. Ulteriori dettagli su UMS possono essere trovati nel Paragrafo 19.7.3.7 (sul sito web del volume).

6.7.3 Un esempio: scheduling di Solaris

Solaris utilizza uno scheduling dei thread basato sulle priorità in cui ogni thread appartiene a una delle sei seguenti classi.

1. Time sharing (TS).
2. Interattivo (IA).
3. Real-time (RT).
4. Sistema (SYS).
5. Ripartizione equa (FSS, per *fair share*).
6. Priorità fissa (FP).

All'interno di ciascuna classe vi sono priorità e algoritmi di scheduling differenti.

La classe di scheduling predefinita per i processi è quella time sharing. È basata su un criterio di scheduling che modifica dinamicamente le priorità, assegnando porzioni di tempo variabili, grazie a una coda multilivello con retroazione. Per default, tra le priorità e le frazioni di tempo sussiste una relazione inversa: più alta è la priorità, minore la frazione di tempo associata; più bassa è la priorità, maggiore sarà la frazione di tempo. Di solito, i processi interattivi hanno priorità alta, mentre i processi con prevalenza d'elaborazione hanno priorità bassa. Questo criterio di scheduling offre un buon tempo di risposta per i processi interattivi e una buona produttività per i processi con prevalenza d'elaborazione. La classe interattiva utilizza gli stessi criteri di scheduling di quella time sharing, ma privilegia le applicazioni dotate di interfacce a finestre (come quelle create dagli ambienti KDE e GNOME), a cui attribuisce priorità più elevate per ottenere prestazioni migliori.

La Figura 6.23 mostra la tabella di dispatch per lo scheduling dei thread interattivi e a tempo ripartito. Queste due classi contemplano 60 livelli di priorità; ne elenchiemo solo alcuni. La tabella di dispatch della Figura 6.23 contiene i seguenti campi.

- **Priorità.** È la priorità dipendente dalle classi, nel caso di quelle interattiva e a tempo ripartito. Il valore cresce al crescere della priorità.
- **Quanto di tempo.** È il quanto di tempo della relativa priorità. Come si può notare, priorità e frazioni di tempo sono inversamente correlate: alla priorità 0, infatti, spetta il quanto di tempo più lungo (200 millisecondi), mentre alla priorità 59, cioè la più alta, corrisponde il quanto minimo (20 millisecondi).
- **Quanto di tempo esaurito.** È la nuova priorità dei thread che abbiano consumato l'intero quanto di tempo loro assegnato senza sospendersi. Tali thread sono considerati a prevalenza d'elaborazione; le loro priorità, come evidenziato dalla tabella, subiscono una diminuzione.
- **Ripresa dell'attività.** È la priorità di un thread che ritorni in attività dopo un periodo di attesa (dell'I/O, per esempio). Come si può vedere nella tabella, quando è disponibile l' I/O per un thread in attesa, la priorità del thread viene incrementata fino a un valore compreso tra 50 e 59. Si implementa così il criterio di scheduling che consiste nel fornire risposte sollecite ai processi interattivi.

priorità	quanto di tempo	quanto di tempo esaurito	ripresa dell'attività
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figura 6.23 Tabella di dispatch di Solaris per i thread interattivi e a tempo ripartito.

La classe dei thread real-time ha la priorità maggiore. Un processo real-time sarà eseguito prima di un processo appartenente a qualsiasi altra classe. Ciò fa sì che i processi real-time abbiano la garanzia di ottenere una risposta dal sistema entro limiti di tempo prefissati. In generale, tuttavia, pochi processi appartengono alla classe real-time.

Solaris sfrutta la classe sistema per eseguire i thread del kernel, quali lo scheduler e il demone per la paginazione. La priorità di un thread di sistema, una volta fissata, non cambia. La classe sistema è riservata all'uso da parte del kernel (i processi utenti eseguiti in modalità di sistema non appartengono alla classe sistema).

Le classi a priorità fissa e a ripartizione equa sono state introdotte in Solaris 9. I thread appartenenti alla classe a priorità fissa hanno lo stesso livello di priorità di quelli della classe time sharing, ma le loro priorità non vengono modificate dinamicamente. Per la classe a ripartizione equa le decisioni di scheduling vengono prese sulla base delle **quote** (*shares*) di CPU, e non sulla base delle priorità. Le quote di CPU sono assegnate a un insieme di processi, chiamato **progetto**, e indicano in che misura il progetto ha diritto all'uso delle risorse disponibili.

Ogni classe di scheduling include una scala di priorità. Tuttavia, lo scheduler converte le priorità specifiche della classe in priorità globali e sceglie per l'esecuzione il thread con la priorità globale più elevata. La CPU esegue il thread prescelto finché (1) si blocca, (2) esaurisce la propria frazione di tempo, o (3) è soggetto a prelazione da un thread con priorità più alta. Se vi sono più thread con la stessa priorità, lo scheduler utilizza una coda circolare RR. La Figura 6.24 mostra le relazioni fra le sei classi di

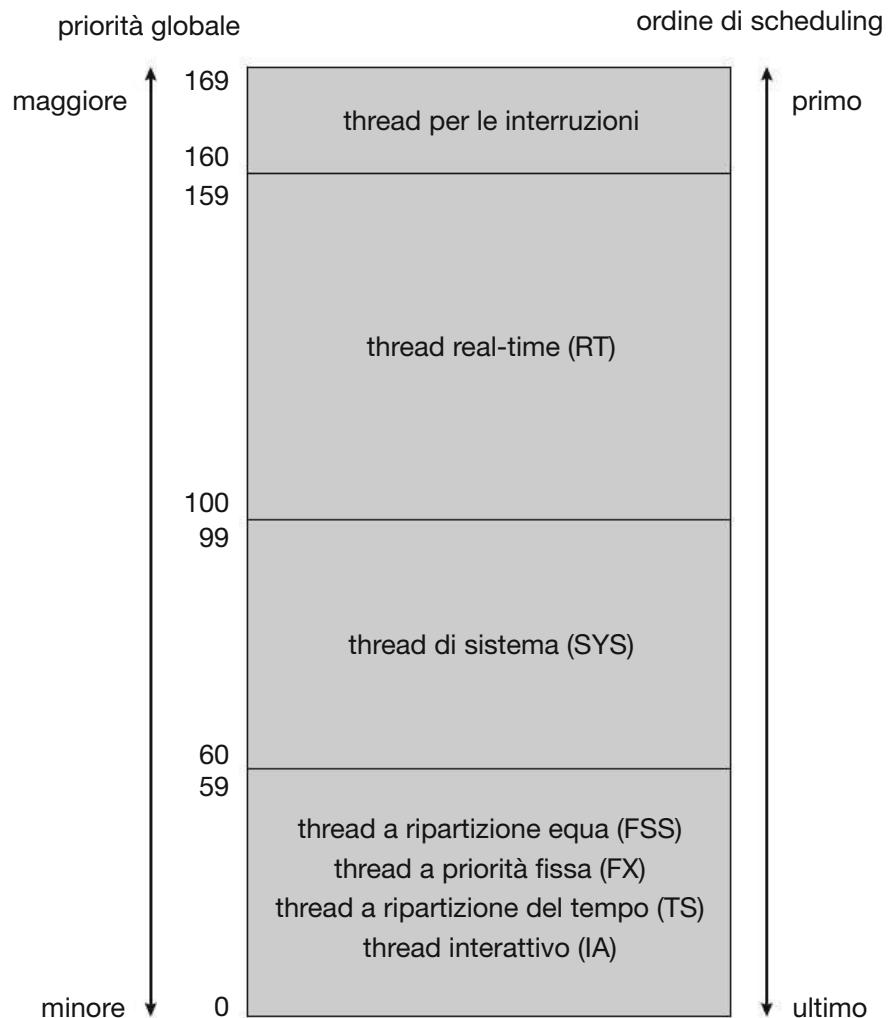


Figura 6.24 Scheduling di Solaris.

scheduling, e quali sono le priorità globali a loro assegnate. Va notato che il kernel utilizza 10 thread per servire le interruzioni. Questi thread non appartengono ad alcuna classe e sono eseguiti con massima priorità (160-169). Come già ricordato, tradizionalmente Solaris utilizzava il modello da molti a molti (Paragrafo 4.3.3), ma a partire da Solaris 9 è passato al modello da uno a uno (Paragrafo 4.3.2).

6.8 Valutazione degli algoritmi

Ci si può chiedere come scegliere un algoritmo di scheduling della CPU per un sistema particolare. Come abbiamo visto nel Paragrafo 6.3, esistono molti algoritmi di scheduling, ciascuno dotato dei propri parametri; quindi, la scelta di un algoritmo può essere abbastanza difficile.

Il primo problema da affrontare riguarda la definizione dei criteri da usare per la scelta dell'algoritmo. Nel Paragrafo 6.2 si spiega che i criteri si definiscono spesso

in termini di utilizzo della CPU, tempo di risposta o produttività. Per scegliere un algoritmo occorre innanzitutto stabilire l'importanza relativa di queste misure. Tra i criteri suggeriti si possono inserire diverse misure, per esempio le seguenti:

- rendere massimo l'utilizzo della CPU con il vincolo che il massimo tempo di risposta sia 1 secondo;
- rendere massima la produttività in modo che il tempo di completamento sia (in media) linearmente proporzionale al tempo d'esecuzione totale.

Una volta definiti i criteri di selezione, è necessario valutare gli algoritmi considerati. Di seguito si descrivono i vari possibili metodi di valutazione.

6.8.1 Modellazione deterministica

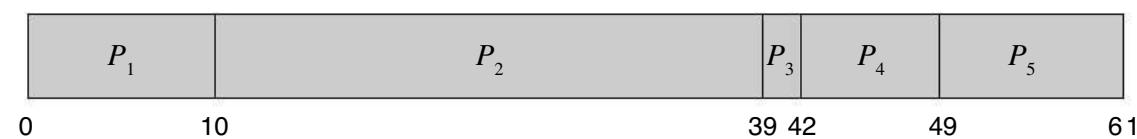
Fra i metodi di valutazione sono di grande importanza quelli che rientrano nella classe della valutazione analitica. La **valutazione analitica**, partendo dall'algoritmo dato e dal carico di lavoro del sistema, fornisce una formula o un numero che valuta le prestazioni dell'algoritmo per quel carico di lavoro.

La **modellazione deterministica** è un tipo di valutazione analitica, che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro.

Si supponga, per esempio, di avere il carico di lavoro illustrato di seguito; i cinque processi si presentano al tempo 0, nell'ordine dato, e la durata delle sequenze di operazioni della CPU è espressa in millisecondi:

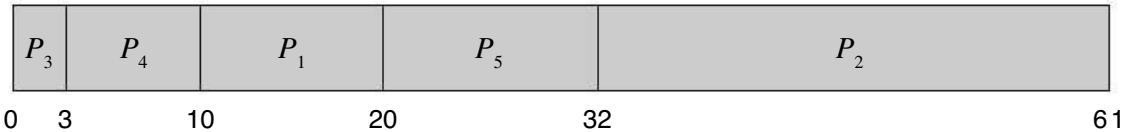
Processo	Durata della sequenza
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Si può stabilire con quale fra gli algoritmi di scheduling FCFS, SJF e RR (con quanto di tempo = 10 millisecondi) per questo insieme di processi si ottenga il minimo tempo medio d'attesa. Con l'algoritmo FCFS i processi si eseguono secondo lo schema seguente.



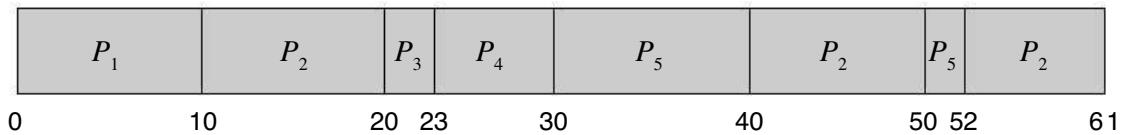
Il tempo d'attesa è di 0 millisecondi per il processo P_1 , di 10 millisecondi per il processo P_2 , di 39 millisecondi per il processo P_3 , di 42 millisecondi per il processo P_4 e di 49 millisecondi per il processo P_5 . Quindi, il tempo d'attesa medio è di $(0 + 10 + 39 + 42 + 49)/5 = 28$ millisecondi.

Con l'algoritmo SJF senza prelazione i processi si eseguono come segue.



Il tempo d'attesa è di 10 millisecondi per il processo P_1 , di 32 millisecondi per il processo P_2 , di 0 millisecondi per il processo P_3 , di 3 millisecondi per il processo P_4 e di 20 millisecondi per il processo P_5 . Quindi, il tempo d'attesa medio è di $(10 + 32 + 0 + 3 + 20)/5 = 13$ millisecondi.

Con l'algoritmo RR i processi si eseguono come segue.



Il tempo d'attesa è di 0 millisecondi per il processo P_1 , di 32 millisecondi per il processo P_2 , di 20 millisecondi per il processo P_3 , di 23 millisecondi per il processo P_4 e di 40 millisecondi per il processo P_5 . Quindi, il tempo d'attesa medio è di $(0 + 32 + 20 + 23 + 40)/5 = 23$ millisecondi.

È importante notare come, *in questo caso*, il criterio SJF fornisca come risultato un tempo medio d'attesa minore della metà del tempo corrispondente ottenuto con lo scheduling FCFS; l'algoritmo RR fornisce un risultato intermedio tra i precedenti.

La definizione e lo studio di un modello deterministico è semplice e rapida; i risultati sono numeri esatti che consentono il confronto tra gli algoritmi. Nondimeno, anche i parametri in ingresso devono essere numeri esatti e i risultati sono applicabili solo a quei casi. Il suo impiego principale consiste nella descrizione degli algoritmi di scheduling e nella presentazione d'esempi. Nei casi in cui vengano eseguiti ripetutamente gli stessi programmi e si possono misurare con precisione i requisiti d'elaborazione dei programmi, la modellazione deterministica è utilizzabile per scegliere un algoritmo di scheduling. Lo studio della modellazione deterministica su un insieme d'esempi può indicare tendenze che si possono poi analizzare e verificare separatamente. Si può per esempio mostrare che per l'ambiente descritto, vale a dire tutti i processi e i relativi tempi disponibili al tempo 0, con il criterio SJF si ottiene sempre il tempo d'attesa minimo.

6.8.2 Reti di code

In molti sistemi i processi eseguiti variano di giorno in giorno, quindi non esiste un insieme statico di processi (o di tempi) da usare nella modellazione deterministica. Si possono però determinare le distribuzioni delle sequenze di operazioni della CPU e delle sequenze di operazioni di I/O, poiché si possono misurare e quindi approssimare, o più semplicemente stimare. Si ottiene una formula matematica che indica la

probabilità di una determinata sequenza di operazioni della CPU. Comunemente questa distribuzione è di tipo esponenziale ed è descritta dalla sua media. Analogamente, è possibile caratterizzare anche la distribuzione degli istanti d'arrivo dei processi nel sistema. Da queste due distribuzioni si può calcolare la produttività media, l'utilizzo o il tempo d'attesa medi, e così via, per la maggior parte degli algoritmi.

Il sistema di calcolo si descrive come una rete di server, ciascuno con una coda d'attesa. La CPU è un server con la propria ready queue, così come il sistema di I/O con le sue code di attesa dei dispositivi. Se sono note le distribuzioni degli arrivi e dei servizi, si possono calcolare l'utilizzo, la lunghezza media delle code, il tempo medio d'attesa e così via. Questo tipo di studio si chiama **analisi delle reti di code** (*queueing-network analysis*).

Si consideri il seguente esempio: sia n la lunghezza media di una coda, escluso il processo correntemente servito, detti W il tempo medio d'attesa nella coda e λ il tasso medio d'arrivo dei nuovi processi nella coda (per esempio, 3 processi al secondo); si prevede che, nel tempo W durante il quale un processo attende nella coda, raggiungano la coda $\lambda \times W$ nuovi processi. Se il sistema è stabile, il numero dei processi che lasciano la coda deve essere uguale al numero dei processi che vi arrivano; quindi,

$$n = \lambda \times W$$

Quest'equazione è nota come **formula di Little** ed è utile soprattutto perché è valida per qualsiasi algoritmo di scheduling e distribuzione degli arrivi.

La formula di Little è utilizzabile per il calcolo di una delle tre variabili, quando sono note le altre due. Per esempio, sapendo che ogni secondo arrivano 7 processi (in media), e che normalmente nella coda ne sono presenti 14, si può calcolare che il tempo medio d'attesa per ogni processo è di 2 secondi.

L'analisi delle reti di code può essere utile per il confronto degli algoritmi di scheduling, ma presenta alcuni limiti. Attualmente le classi di algoritmi e distribuzioni trattabili sono piuttosto limitate. Poiché può essere difficile lavorare matematicamente con distribuzioni e algoritmi complicati, spesso le distribuzioni d'arrivo e servizio vengono definite in maniera matematicamente trattabile, ma non realistica. Generalmente è necessario stabilire anche un numero di assunzioni indipendenti che possono non essere precise. Come risultato di queste difficoltà, le reti di code spesso si limitano ad approssimare un sistema reale, rendendo discutibile la precisione dei risultati ottenuti.

6.8.3 Simulazioni

Per riuscire ad avere una valutazione più precisa degli algoritmi di scheduling ci si può servire di **simulazioni**. Le simulazioni implicano la programmazione di un modello del sistema di calcolo; le strutture dati rappresentano gli elementi principali del sistema. Il simulatore dispone di una variabile che rappresenta un clock; all'incremento del valore di questa variabile, il simulatore modifica lo stato del sistema in modo da tener conto delle attività dei dispositivi, dei processi e dello scheduler. Durante l'esecuzione della simulazione si raccolgono e si stampano statistiche che indicano le prestazioni degli algoritmi.

I dati necessari per condurre la simulazione si possono ottenere in vari modi. Il metodo più diffuso impiega un generatore di numeri casuali, programmato per generare processi, durate dei burst della CPU, arrivi, partenze dal sistema e così via, in modo conforme alle rispettive distribuzioni di probabilità. Queste sono definibili matematicamente (esponenziali, uniformi, di Poisson) oppure in modo empirico. Se la distribuzione deve essere definita in modo empirico, si fanno misure sul sistema reale in esame, e si usano i risultati per definire la distribuzione effettiva degli eventi nel sistema reale; questa distribuzione è poi utilizzata per generare l'input della simulazione.

Tuttavia, una simulazione condotta sulla base delle distribuzioni degli eventi può non essere precisa, a causa delle relazioni esistenti tra eventi successivi nel sistema reale. La distribuzione delle frequenze, infatti, si limita a indicare quanti eventi di una data categoria si verificano, senza fornire informazioni sul loro ordine. Per rimediare a questo problema si può sottoporre il sistema reale a un monitoraggio, con la registrazione della sequenza degli eventi effettivi – in questo modo si ottiene un cosiddetto **trace tape** – (Figura 6.25), che poi si usa per condurre la simulazione. Si tratta di uno strumento eccellente che permette di confrontare gli algoritmi rispetto allo stesso insieme di dati reali, e con cui si possono ottenere risultati molto precisi.

Poiché spesso richiedono diverse ore di tempo d'elaborazione, le simulazioni possono tuttavia essere molto onerose. Una simulazione più dettagliata dà risultati più precisi, ma richiede anche una maggiore quantità di tempo, e molto spazio di memoria per la registrazione degli eventi. Inoltre, la progettazione, la codifica e la messa a punto di un simulatore possono essere un compito assai impegnativo.

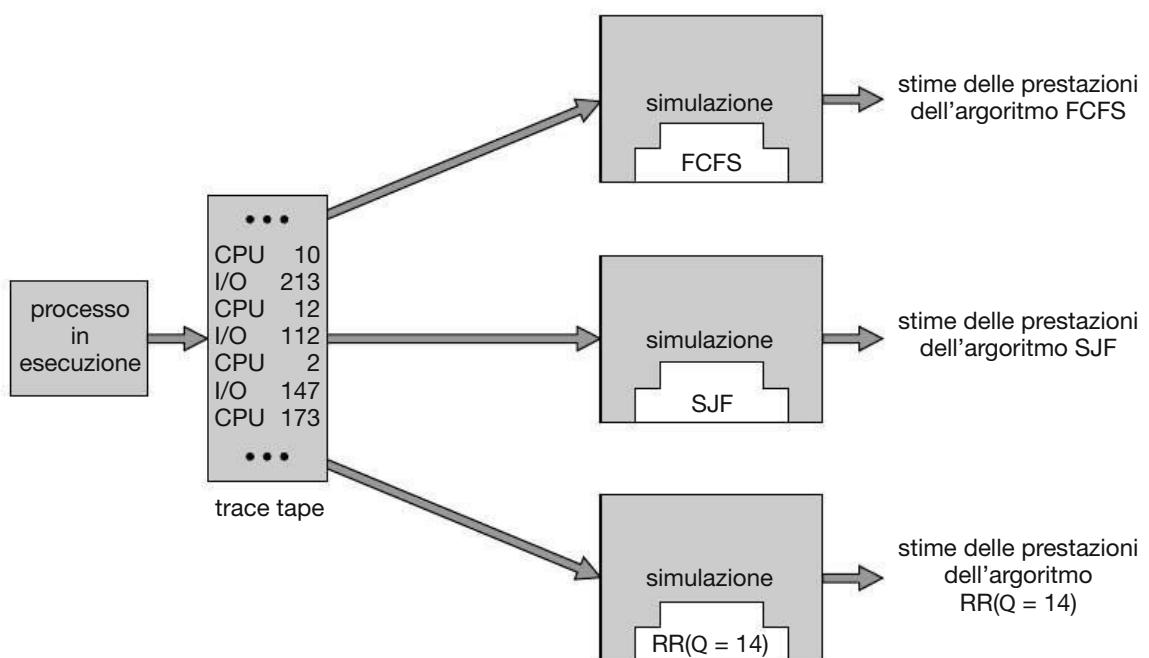


Figura 6.25 Valutazione di algoritmi di scheduling della CPU tramite una simulazione.

6.8.4 Realizzazione

Persino una simulazione ha dei limiti per quel che riguarda la precisione. L'unico modo assolutamente preciso per valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel sistema operativo e osservarne il comportamento nelle reali condizioni di funzionamento del sistema.

Il problema principale di questo approccio è il suo costo: le spese non sono dovute solo alla codifica dell'algoritmo e alle modifiche da fare al sistema operativo affinché possa gestire l'algoritmo con le sue strutture dati, ma anche alle reazioni degli utenti a fronte di continue modifiche del sistema operativo. Alla maggior parte degli utenti non interessa la realizzazione di un sistema operativo migliore, vogliono solo eseguire i propri processi e usare i risultati. Un sistema operativo che si trovi costantemente in fasi di modifica e messa a punto non aiuta gli utenti a svolgere il loro lavoro.

Un'altra difficoltà da affrontare per la valutazione è il cambiamento dell'ambiente in cui si usa l'algoritmo. L'ambiente non cambia solo nel modo consueto, cioè per la scrittura di nuovi programmi e nuovi problemi che si possono riscontrare, ma si modifica anche in seguito alle prestazioni dello scheduler. Se si dà la priorità ai processi brevi, gli utenti possono suddividere i processi più lunghi in gruppi di processi brevi. Se si dà la priorità ai processi interattivi rispetto ai processi non interattivi, gli utenti possono passare all'uso interattivo.

Per esempio, alcuni ricercatori progettarono un sistema che classificava automaticamente i processi nelle categorie interattiva e non interattiva sulla base della quantità di I/O eseguita dal o verso il terminale. I processi che non leggevano o scrivevano sul terminale per un intero secondo erano classificati come non interattivi, e conseguentemente spostati in una coda a bassa priorità. Un programmatore reagì a questa strategia modificando i suoi programmi di modo che scrivessero sul terminale un carattere a intervalli regolari di meno di un secondo. Il risultato fu che le applicazioni del programmatore ricevettero alta priorità dal sistema, sebbene il loro output fosse del tutto privo di senso.

Gli algoritmi di scheduling più flessibili sono quelli che possono essere tarati dagli amministratori del sistema o dagli utenti in modo da adattarsi a una specifica applicazione o insieme di applicazioni. Per esempio, le macchine impegnate in applicazioni grafiche sofisticate avranno necessità di scheduling diverse da quelle di un server web o di un file server. Alcuni sistemi operativi, e in particolare diverse versioni di UNIX, danno all'amministratore la possibilità di calibrare con precisione i parametri di scheduling a seconda delle particolari configurazioni del sistema. Solaris, per esempio, offre il comando `dispadm` per la modifica dei parametri che regolano le classi di scheduling discusse nel Paragrafo 6.7.3.

Un altro approccio è di usare delle API appropriate per modificare la priorità di processi e thread. Le API Java, POSIX e Windows offrono tali funzionalità. Questa tecnica ha però lo svantaggio che tarare un sistema o un'applicazione per migliorarne le prestazioni in un caso specifico spesso non ha lo stesso risultato in situazioni più generali.

6.9 Sommario

Lo scheduling della CPU consiste nella scelta di un processo dalla ready queue a cui assegnare la CPU. L’assegnazione della CPU al processo prescelto è eseguita dal dispatcher.

L’algoritmo di scheduling in ordine d’arrivo (FCFS) è il più semplice, ma può far sì che processi di breve durata attendano processi molto lunghi. Si dimostra che lo scheduling ottimale, che determina il minimo tempo medio d’attesa, è lo scheduling shortest-job-first (SJF). Realizzare lo scheduling SJF è complicato, poiché è difficile prevedere la lunghezza della successiva sequenza di operazioni della CPU. L’algoritmo SJF è un caso particolare dell’algoritmo generale di scheduling con priorità, che si limita ad assegnare la CPU al processo con priorità più elevata. Sia lo scheduling con priorità sia lo scheduling SJF possono condurre a situazioni d’attesa indefinita. L’invecchiamento (*aging*) è una tecnica che si usa per impedire che avvengano tali situazioni.

Lo scheduling circolare (RR) è più appropriato per un sistema time sharing: si assegna la CPU al primo processo della ready queue per q unità di tempo (quanto di tempo); dopo q unità di tempo, se il processo non ha rilasciato la CPU, viene prelazionato e messo in fondo alla ready queue. Il problema principale è la scelta della durata del quanto di tempo; se è troppo lungo, lo scheduling RR si riduce a uno scheduling FCFS; se è troppo breve, l’overhead di scheduling dovuto ai cambi di contesto diventa eccessivo.

L’algoritmo FCFS è senza prelazione; l’algoritmo RR è con prelazione; gli algoritmi SJF e con priorità possono essere sia con prelazione sia senza prelazione.

Lo scheduling a code multilivello permette l’uso di diversi algoritmi per diverse classi di processi. Il modello più comune prevede una coda dei processi interattivi, eseguiti in foreground, con scheduling RR, e la coda per processi batch, eseguiti in background, con scheduling FCFS. Le code multilivello con retroazione permettono ai processi di spostarsi da una coda all’altra.

Molti dei sistemi attuali supportano processori multipli permettendo a ciascun processore uno scheduling indipendente. In genere ogni processore mantiene una propria coda di processi (o di thread) pronti, tutti disponibili all’esecuzione. Tra gli altri aspetti relativi allo scheduling in sistemi multiprocessore vi sono la predilezione, il bilanciamento del carico e lo scheduling con processori multicore.

Un sistema real-time richiede che i risultati arrivino entro una scadenza, i risultati che arrivano dopo il termine sono inutili. I sistemi real-time hard devono garantire che le attività in tempo reale siano servite entro le scadenze. I sistemi real-time soft sono meno restrittivi e assegnano soltanto una maggiore priorità di scheduling ai task real-time.

Tra gli algoritmi di scheduling real-time vi sono lo scheduling con priorità proporzionale alla frequenza e lo scheduling EDF. Lo scheduling con priorità proporzionale alla frequenza assegna ai task che richiedono la CPU più spesso una priorità più

alta rispetto ai task che richiedono la CPU meno spesso. Lo scheduling EDF assegna la priorità in base alle successive scadenze: più vicina è la scadenza, maggiore è la priorità. Lo scheduling a quote proporzionali divide il tempo di processore in quote e assegna a ogni processo un certo numero di quote, garantendo così a ogni processo una quota proporzionale del tempo di CPU. La API POSIX Pthreads fornisce diverse funzionalità per lo scheduling di thread real-time.

I sistemi operativi che gestiscono i thread a livello kernel devono occuparsi dello scheduling dei thread, e non di quello dei processi. Tra questi vi sono il Solaris e il Windows; entrambi gestiscono lo scheduling dei thread impiegando un algoritmo di scheduling con prelazione, basato su priorità e che gestisce anche i thread in tempo reale. Anche lo scheduler dei processi di Linux impiega un algoritmo basato su priorità e che prevede la gestione dei processi in tempo reale. Gli algoritmi di scheduling di questi tre sistemi operativi favoriscono generalmente i processi interattivi rispetto ai processi con prevalenza d'elaborazione.

La vasta gamma di algoritmi di scheduling esistenti richiede la disponibilità di metodi per la loro selezione. I metodi analitici sfruttano strumenti matematici per valutare le prestazioni degli algoritmi. Le simulazioni determinano le prestazioni eseguendo gli algoritmi in presenza di un insieme “rappresentativo” di processi. Tuttavia la simulazione può tutt'al più dare un'approssimazione delle effettive prestazioni dei sistemi. L'unica tecnica affidabile per la valutazione delle prestazioni di un algoritmo di scheduling consiste nell'implementarlo su un vero sistema, e nel misurarne le prestazioni in un contesto reale.

Esercizi di ripasso

6.1 Un algoritmo di scheduling della CPU stabilisce un ordine per l'esecuzione dei processi pianificati. Dati n processi da mandare in esecuzione su di un singolo processore, quanti differenti scheduling sono possibili? Date una formula in funzione di n .

6.2 Spiegate la differenza tra lo scheduling con e senza prelazione.

6.3 Supponete che i seguenti processi siano pronti per l'esecuzione agli istanti di tempo indicati nella tabella che segue. Nella tabella è anche indicata la durata della sequenza di operazioni per ogni processo. Nel rispondere alle domande seguenti utilizzate uno scheduling senza prelazione e basate le vostre decisioni sulle informazioni che avete a disposizione nel momento in cui la decisione deve essere presa.

Processo	Istante di arrivo	Durata della sequenza
P_1	0,0	8
P_2	0,4	4
P_3	1,0	1

- a. Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling FCFS?
- b. Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling SJF?
- c. L'algoritmo SJF dovrebbe migliorare le prestazioni, ma si noti che al tempo 0 viene scelto il processo P_1 , perché non si sa ancora che presto arriveranno due processi più brevi. Calcolate il tempo di completamento medio ipotizzando che la CPU venga lasciata inattiva per il primo istante di tempo e che successivamente venga utilizzato l'algoritmo SJF. Ricordate che i processi P_1 e P_2 restano in attesa durante il periodo di inattività, e quindi il loro tempo di attesa può aumentare. Questo algoritmo si potrebbe chiamare scheduling con conoscenza del futuro.

6.4 Quali vantaggi si hanno nell'avere quanti di tempo di dimensioni differenti a differenti livelli in un sistema di code multilivello?

6.5 Molti algoritmi di scheduling della CPU sono parametrici. L'algoritmo RR, per esempio, richiede un parametro che specifichi l'intervallo di tempo. Le code multilivello con retroazione richiedono parametri per specificare il numero di code, l'algoritmo di scheduling da utilizzare per ogni coda, il criterio usato per muovere i processi tra le code, e così via.

Questi algoritmi sono dunque classi di algoritmi (per esempio, la classe di algoritmi RR per tutti gli intervalli di tempo, e così via). Una classe di algoritmi ne può includere un'altra (per esempio, l'algoritmo FCFS è un algoritmo RR con intervallo di tempo infinito). Quale relazione intercorre (se esiste una relazione) tra le seguenti coppie di classi di algoritmi?

- a. Priorità e SJF.
- b. Code multilivello con retroazione e FCFS.
- c. Priorità ed FCFS.
- d. RR e SJF.

6.6 Supponete che un algoritmo di scheduling (a livello dello scheduling della CPU a breve termine) favorisca quei processi che hanno usato meno CPU in un passato recente. Spiegate perché questo algoritmo favorirà programmi con prevalenza di I/O senza bloccare permanentemente i programmi con prevalenza di elaborazione.

6.7 Individuate le differenze tra gli scheduling PCS e SCS.

6.8 Supponete che un sistema operativo mappi thread a livello utente in thread a livello kernel utilizzando il modello molti-a-molti e che il mapping venga effettuato mediante l'utilizzo di LWP. Inoltre, il sistema consente agli utenti di creare thread real-time. È necessario legare rigidamente un thread real-time a un LWP?

6.9 Lo scheduler tradizionale di UNIX stabilisce una relazione inversa tra numeri e priorità: più alto è il numero, minore è la priorità. Lo scheduler ricalcola le priorità dei processi una volta al secondo utilizzando la seguente funzione:

$$\text{Priorità} = (\text{utilizzo recente della CPU} / 2) + \text{base}$$

dove base = 60 utilizzo e *utilizzo recente della CPU* è un valore che indica in che misura un processo ha utilizzato la CPU dall'ultima volta in cui le priorità sono state ricalcolate.

Si supponga che l' utilizzo recente della CPU sia 40 per il processo P_1 , 18 per il processo P_2 e 10 per il processo P_3 . Quali saranno le nuove priorità di questi tre processi dopo il ricalcolo? Sulla base di questa informazione, dite se lo scheduler tradizionale di UNIX alza o abbassa la priorità relativa di un processo CPU-bound.

Esercizi

6.10 Perché è importante, per lo scheduler, distinguere i programmi con prevalenza di I/O da quelli con prevalenza di elaborazione?

6.11 Considerate come le seguenti coppie di criteri per lo scheduling entrino in conflitto in certe situazioni:

- a. utilizzo della CPU e tempo di risposta;
- b. tempo di completamento medio e tempo di attesa massimo;
- c. utilizzo dei dispositivi di I/O e utilizzo della CPU.

6.12 Una tecnica per l'implementazione dello **scheduling a lotteria** consiste nell'assegnare ai processi dei biglietti della lotteria che vengono utilizzati per l'attribuzione del tempo di CPU. Ogni volta che si deve prendere una decisione riguardo allo scheduling viene sorteggiato casualmente un biglietto della lotteria e il processo che ne è in possesso ottiene la CPU. Il sistema operativo BTV implementa lo scheduling a lotteria effettuando 50 sorteggi ogni secondo e assegnando al vincitore 20 millisecondi di tempo di CPU (20×50 millisecondi = 1 secondo). Descrivete in che modo lo scheduler BTV può assicurare che i thread con priorità più alta ricevano più attenzione dalla CPU rispetto a quelli a bassa priorità.

6.13 Nel Capitolo 5 abbiamo discusso le possibili race condition su varie strutture dati del kernel. La maggior parte degli algoritmi di scheduling mantiene una coda di esecuzione, che elenca i processi ammissibili per l'esecuzione su un processore. Sui sistemi multicore vi sono due opzioni: (1) ogni core di elaborazione ha la sua coda di esecuzione, oppure (2) una sola coda di esecuzione è condivisa tra tutti i core. Descrivete vantaggi e svantaggi di ciascuno di questi approcci.

6.14 Considerate la formula della media esponenziale atta a predire la durata della sequenza successiva della CPU. Quali implicazioni scaturiscono dall’assegnazione dei seguenti valori ai parametri usati dall’algoritmo?

- a. $\alpha = 0$ e $\tau_0 = 100$ millisecondi
- b. $\alpha = 0,99$ e $\tau_0 = 10$ millisecondi

6.15 Una variante dello scheduler round-robin è lo scheduler **round-robin regressivo**.

Questo scheduler assegna a ogni processo un quanto di tempo e una priorità. Il valore iniziale di un quanto di tempo è pari a 50 millisecondi. Ogni volta che un processo viene assegnato alla CPU e usa interamente il suo quanto di tempo (non bloccandosi per I/O), vengono aggiunti 10 millisecondi al suo quanto e il suo livello di priorità viene aumentato. Il quanto di tempo di un processo può essere aumentato fino a un massimo di 100 millisecondi. Quando un processo si blocca prima di utilizzare per intero il suo tempo, il suo quanto viene ridotto di 5 millisecondi e la sua priorità rimane la stessa. Quale tipo di processo (CPU-bound o I/O-bound) viene favorito dallo scheduler round-robin regressivo? Giustificate la vostra risposta.

6.16 Considerate il seguente insieme di processi, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza	Priorità
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

Presumiamo che i processi siano arrivati nell’ordine P_1, P_2, P_3, P_4, P_5 , e siano tutti presenti al tempo 0.

- a. Disegnate quattro diagrammi di Gantt che illustrino l’esecuzione di questi processi con gli algoritmi di scheduling FCFS, SJF, con priorità senza prelazione (un numero di priorità più alto indica una priorità maggiore) e RR (quanto = 2).
- b. Calcolate il tempo di completamento di ciascun processo per ciascun algoritmo di scheduling di cui al punto a).
- c. Calcolate il tempo d’attesa di ciascun processo per ciascun algoritmo di scheduling di cui al punto a).
- d. Dite quale algoritmo ha il minimo tempo medio d’attesa (su tutti i processi).

6.17 I seguenti processi vengono pianificati utilizzando un algoritmo di scheduling round robin con prelazione. A ogni processo viene assegnato un valore numerico

di priorità, dove un valore più grande indica una priorità relativa superiore. Oltre ai processi di seguito elencati, il sistema ha un **task idle** (che non consuma risorse di CPU e viene identificato come P_{idle}). Questo task ha priorità 0 e viene schedulato ogni volta che il sistema non ha altri processi disponibili per l'esecuzione. La lunghezza di un quanto di tempo è di 10 unità. Se un processo è prelazionato da un processo a priorità superiore viene posto alla fine della coda.

Thread	Priorità	Durata della sequenza	Istante di arrivo
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

- a. Mostrate l'ordine di scheduling dei processi utilizzando un diagramma di Gantt.
 - b. Calcolate il tempo di completamento di ciascun processo.
 - c. Calcolate il tempo d'attesa di ciascun processo.
 - d. Calcolate il tasso di utilizzo della CPU.
- 6.18** Il comando `nice` viene usato per impostare il nice value di un processo su Linux, oltre che su altri sistemi UNIX. Spiegate perché alcuni sistemi permettono a qualsiasi utente di assegnare a un processo un nice value ≥ 0 , mentre consentono solo all'utente root di assegnare un nice value < 0 .
- 6.19** Quale tra i seguenti algoritmi di scheduling potrebbe generare un'attesa indefinita?
- a. In ordine di arrivo (FCFS).
 - b. Per brevità (SJF).
 - c. Circolare (RR).
 - d. Per priorità.
- 6.20** Data una variante dell'algoritmo di scheduling RR in cui gli elementi della ready queue sono puntatori ai PCB:
- a. descrivete l'effetto dell'inserimento di due puntatori allo stesso processo nella ready queue;
 - b. descrivete due vantaggi e due svantaggi di questo schema;
 - c. ipotizzate una modifica all'algoritmo RR ordinario che consenta di ottenere lo stesso effetto senza duplicare i puntatori.
- 6.21** Considerate un sistema su cui vengano eseguiti dieci processi con prevalenza di I/O e un processo con prevalenza di elaborazione. Supponiamo che i primi

richiedano un’operazione di I/O ogni millisecondo di elaborazione della CPU, e che ciascuna di tali operazioni sia completata in 10 millisecondi. Ipotizzate, inoltre, che il tempo necessario per il cambio di contesto sia di 0,1 millisecondi e che tutti i processi siano attivi per un lungo periodo. Calcolate l’utilizzo della CPU in presenza di scheduler circolare RR se:

- a. il quanto di tempo è pari a 1 millisecondo;
- b. il quanto di tempo è pari a 10 millisecondi.

6.22 Considerate un sistema che applichi un algoritmo di scheduling a code multilivello. A quale strategia può ricorrere l’utente che voglia ottenere per un suo processo la massima quantità di tempo dalla CPU?

6.23 Considerate un algoritmo di scheduling a priorità con prelazione, basato su priorità variabili dinamicamente. I numeri di priorità maggiori indicano una priorità più alta. Quando un processo attende la CPU (nella ready queue), la sua priorità varia a un tasso α ; quando è in esecuzione, la sua priorità varia a un tasso β . All’ingresso nella ready queue, si attribuisce la priorità 0 a tutti i processi. I parametri α e β si possono impostare in modo da fornire algoritmi di scheduling diversi.

- a. Descrivete l’algoritmo risultante da $\beta > \alpha > 0$.
- b. Descrivete l’algoritmo risultante da $\alpha < \beta < 0$.

6.24 Spiegate quanto i seguenti algoritmi di scheduling gestiscono preferenzialmente i processi di breve durata:

- a. in ordine di arrivo (FCFS);
- b. circolare (RR);
- c. a code multilivello con retroazione.

6.25 Usando l’algoritmo di scheduling di Windows, quale priorità numerica è assegnata, nei casi che seguono a:

- a. Un thread appartenente alla REALTIME_PRIORITY_CLASS con una priorità relativa NORMAL.
- b. Un thread appartenente alla ABOVE_NORMAL_PRIORITY_CLASS con una priorità relativa HIGHEST.
- c. Un thread appartenente alla BELOW_NORMAL_PRIORITY_CLASS con una priorità relativa ABOVE_NORMAL.

6.26 Supponendo che nessun thread appartenga alla REALTIME_PRIORITY_CLASS e che a nessuno possa essere assegnata una priorità TIME_CRITICAL, quale combinazione classe di priorità/priorità corrisponde al più alto livello possibile di priorità relativa nello scheduling Windows?

6.27 Considerate l'algoritmo di scheduling del sistema operativo Solaris per i thread time sharing.

- a. Qual è il quanto di tempo (in millisecondi) per un thread con priorità 15? E per uno con priorità 40?
- b. Ipotizzate che un thread con priorità 50 abbia consumato l'intera porzione di tempo riservatagli senza bloccarsi. Quale sarà la nuova priorità assegnata dallo scheduler a questo thread?
- c. Poniamo che un thread con priorità 20 si blocchi per l'I/O prima che la propria porzione di tempo sia finita. Qual è la nuova priorità che lo scheduler assegnerà a questo thread?

6.28 Supponiamo che due task A e B siano in esecuzione su un sistema Linux. I nice value di A e B sono rispettivamente -5 e 5. Utilizzando lo scheduler CFS come guida, descrivete come variano i valori di `vruntime` dei due processi in ciascuno dei seguenti scenari:

- A e B sono CPU-bound.
- A è I/O-bound e B è CPU-bound.
- A è CPU-bound e B è I/O-bound.

6.29 Descrivete le possibili soluzioni al problema di inversione della priorità in un sistema real-time. Dite anche se le soluzioni potrebbero essere implementate nel contesto di uno scheduler a quote proporzionali.

6.30 In quali circostanze lo scheduling con priorità proporzionale alla frequenza si comporta peggio dello scheduling EDF nel venire incontro alle scadenze associate ai processi?

6.31 Si considerino due processi P_1 e P_2 dove $P_1 = 50$, $t_1 = 25$, $p_2 = 75$ e $t_2 = 30$.

- a. È possibile schedulare questi due processi utilizzando lo scheduling a priorità proporzionale alla frequenza? Illustrate la vostra risposta utilizzando un diagramma di Gantt come quelli mostrati nelle Figure 6.16-6.19.
- b. Illustrate lo scheduling di questi due processi utilizzando lo scheduling EDF.

6.32 Spiegate perché nei sistemi real-time hard i tempi di latenza di interrupt e dispatch devono essere limitati.

Note bibliografiche

Le code con retroazione furono originariamente implementate sul sistema CTSS descritto in [Corbato et al. 1962]. Questa tecnica di scheduling è stata analizzata in [Schrage 1967]. L'algoritmo basato sulle priorità con prelazione dell'Esercizio 6.23 è stato suggerito da [Kleinrock 1975]. Gli algoritmi di scheduling per sistemi real-time hard, tra cui lo scheduling con priorità proporzionale alla frequenza e lo scheduling EDF, sono presentati in [Liu e Layland 1973].

[Anderson et al. 1989], [Lewis e Berg 1998] e [Philbin et al. 1996] analizzano lo scheduling dei thread. Lo scheduling di processori multicore è trattato da McNairy e Bhatia [2005] e Kongetira et al. [2005].

[Fisher 1981], [Hall et al. 1996] e [Lowney et al. 1993] descrivono tecniche di scheduling che tengono conto delle informazioni sui tempi di esecuzione dei processi ottenute da esecuzioni precedenti.

Lo scheduling basato sulle quote (*fair-share*) è trattato da [Henry 1984], [Woodside 1986], e [Kay e Lauder 1988].

Le strategie di scheduling di UNIX V sono descritte da [Bach 1987]; quelle di UNIX FreeBSD 5.2 da [McKusick et al. 2005] e Neville-Neil [2005]; quelle di Mach da [Black 1990]. [Love 2010] e [Maurer 2008] trattano lo scheduling in Linux. [Faggioli et al.] discutono dell'aggiunta di uno scheduling EDF al kernel di Linux. I dettagli dello scheduler ULE si trovano in Roberson [2003]. Lo scheduling di Solaris è trattato in [Mauro e McDougall 2007]. [Russinovich e Solomon 2009] trattano dello scheduling di Windows. [Butenhof 1997] e [Lewis e Berg 1998] si occupano dello scheduling in Pthread. Siddha et al. [2007] discutono gli obiettivi futuri dello scheduling in sistemi multicore.

Bibliografia

- [Anderson et al. 1989] T. E. Anderson, E. D. Lazowska e H. M. Levy, “The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors”, IEEE Transactions on Computers, Volume 38, N. 12, p. 1631–1644, 1989.
- [Bach 1987] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, 1987.
- [Black 1990] D. L. Black, “Scheduling Support for Concurrency and Parallelism in the Mach Operating System”, IEEE Computer, Volume 23, N. 5, p. 35–43, 1990.
- [Butenhof 1997] D. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.
- [Corbato et al. 1962] F. J. Corbato, M. Merwin-Daggett e R. C. Daley, “An Experimental Time-Sharing System”, Proceedings of the AFIPS Fall Joint Computer Conference, p. 335–344, 1962.
- [Faggioli et al. 2009] D. Faggioli, F. Checconi, M. Trimarchi e C. Scordino, “An EDF scheduling class for the Linux kernel”, Proceedings of the 11th Real-Time Linux Workshop, 2009.

- [Fisher 1981]** J. A. Fisher, “Trace Scheduling: A Technique for Global Microcode Compaction”, IEEE Transactions on Computers, Volume 30, N. 7, p. 478–490, 1981.
- [Hall et al. 1996]** L. Hall, D. Shmoys e J. Wein, “Scheduling To Minimize Average Completion Time: Off-line and On-line Algorithms”, SODA: ACM-SIAM Symposium on Discrete Algorithms, 1996.
- [Henry 1984]** G. Henry, “The Fair Share Scheduler”, AT&T Bell Laboratories Technical Journal, 1984.
- [Kay e Lauder 1988]** J. Kay e P. Lauder, “A Fair Share Scheduler”, Communications of the ACM, Volume 31, N. 1, p. 44–55, 1988.
- [Kleinrock 1975]** L. Kleinrock, *Queueing Systems*, Volume II: Computer Applications, Wiley-Interscience, 1975.
- [Kongetira et al. 2005]** P. Kongetira, K. Aingaran e K. Olukotun, “Niagara: A 32-Way Multithreaded SPARC Processor”, IEEE Micro Magazine, Volume 25, N. 2, p. 21–29, 2005.
- [Lewis e Berg 1998]** B. Lewis e D. Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press, 1998.
- [Liu e Layland 1973]** C. L. Liu e J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, Communications of the ACM, Volume 20, N. 1, p. 46–61, 1973.
- [Love 2010]** R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library, 2010.
- [Lowney et al. 1993]** P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O’Donnell e J. C. Ruttenberg, “The Multiflow Trace Scheduling Compiler”, Journal of Supercomputing, Volume 7, N. 1-2, p. 51–142, 1993.
- [Mauerer 2008]** W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons, 2008.
- [Mauro e McDougall 2007]** J. Mauro e R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall, 2007.
- [McKusick e Neville-Neil 2005]** M. K. McKusick e G. V. Neville-Neil, *The Design and Implementation of the FreeBSD UNIX Operating System*, Addison Wesley, 2005.
- [McNairy e Bhatia 2005]** C. McNairy e R. Bhatia, “Montecito: A Dual-Core, Dual-Threaded Itanium Processor”, IEEE Micro Magazine, Volume 25, N. 2, p. 10–20, 2005.
- [Philbin et al. 1996]** J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas e K. Li, “Thread Scheduling for Cache Locality”, Architectural Support for Programming Languages and Operating Systems, p. 60–71, 1996.
- [Roberson 2003]** J. Roberson, “ULE: A Modern Scheduler For FreeBSD”, Proceedings of the USENIX BSDCon Conference, p. 17–28, 2003.
- [Russinovich e Solomon 2009]** M. E. Russinovich e D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*, 5º Ed., Microsoft Press, 2009.

- [Schrage 1967]** L. E. Schrage, “The Queue M/G/I with Feedback to Lower Priority Queues”, *Management Science*, Volume 13, p. 466–474, 1967.
- [Siddha et al. 2007]** S. Siddha, V. Pallipadi e A. Mallick, “Process Scheduling Challenges in the Era of Multi-Core Processors”, *Intel Technology Journal*, Volume 11, N. 4, 2007.
- [Woodside 1986]** C. Woodside, “Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers”, *IEEE Transactions on Software Engineering*, Volume SE-12, N. 10, p. 1041–1048, 1986.

CAPITOLO

7

OBIETTIVI DEL CAPITOLO

- Descrizione delle situazioni di stallo (*deadlock*) che impediscono il completamento del lavoro a gruppi di processi concorrenti.
- Presentazione di vari metodi per prevenire o impedire le situazioni di stallo.

Stallo dei processi

In un ambiente con multiprogrammazione più processi possono competere per ottenere un numero finito di risorse; se una risorsa non è correntemente disponibile, il processo richiedente passa allo stato d'attesa. In alcuni casi, se le risorse richieste sono trattenute da altri processi, a loro volta nello stato d'attesa, il processo potrebbe non cambiare più il suo stato. Situazioni di questo tipo sono chiamate di **stallo** (*deadlock*). Questo argomento è stato trattato brevemente anche nel Capitolo 5, nello studio dei semafori.

Un efficace esempio di situazione di stallo si può ricavare da una legge dello stato del Kansas approvata all'inizio del ventesimo secolo, che in una sua parte recita: “Quando due treni convergono a un incrocio, ambedue devono arrestarsi, e nessuno dei due può ripartire prima che l'altro si sia allontanato”.

In questo capitolo si descrivono i metodi che un sistema operativo può usare per prevenire o affrontare le situazioni di stallo. Anche se esistono applicazioni che possono identificare i programmi suscettibili di stallo, la maggior parte dei sistemi operativi attuali non offre strumenti di prevenzione di queste situazioni, e progettare programmi che non rischiano lo stallo rimane una responsabilità dei programmatore. Date le tendenze correnti, il problema dello stallo può diventare soltanto più importante: aumento del numero dei processi e delle risorse all'interno di un sistema, programmi multithread e preferenza attribuita a sistemi server sempre attivi che gestiscono file e basi di dati, anziché ai sistemi batch.

7.1 Modello del sistema

Un sistema è composto da un numero finito di risorse da distribuire tra più processi in competizione. Le risorse possono essere suddivise in tipi (o classi) differenti, ciascuno formato da un certo numero di istanze identiche. Cicli di CPU, file e dispositivi di I/O (come stampanti e lettori DVD), sono tutti esempi di tipi di risorsa. Se un sistema ha due CPU, il tipo di risorsa *CPU* ha due istanze. Analogamente, il tipo di risorsa *stampante* può avere cinque istanze.

Se un processo richiede un’istanza relativa a un tipo di risorsa, l’assegnazione di *qualsiasi* istanza di quel tipo può soddisfare la richiesta. Se ciò non si verifica significa che le istanze non sono identiche e le classi di risorse non sono state definite correttamente. Un sistema può, per esempio, avere due stampanti; se a nessuno interessa sapere quale sia la stampante che viene utilizzata, le due stampanti si possono definire come appartenenti alla stessa classe di risorse; se, però, una stampante si trova al nono piano e l’altra al piano terra, allora le due stampanti si possono considerare non equivalenti, e può essere necessario definire classi di risorse distinte per ciascuna risorsa.

Nel Capitolo 5 abbiamo descritto vari strumenti di sincronizzazione, tra cui i lock mutex e i semafori. Anche questi strumenti sono considerati risorse di sistema e sono una tipica causa di situazioni di stallo. Tuttavia, un lock è solitamente destinato alla protezione di una specifica struttura dati, per esempio un lock può essere utilizzato per proteggere l’accesso a una coda, un altro per proteggere l’accesso a una lista concatenata, e così via. Per questo motivo, a ogni lock viene tipicamente assegnata la propria classe di risorse, senza problemi di definizione.

Prima di adoperare una risorsa, un sistema deve richiederla e, dopo averla usata, deve rilasciarla. Un processo può richiedere tutte le risorse necessarie per eseguire il compito assegnatogli, anche se ovviamente il numero delle risorse richieste non può superare quello totale delle risorse disponibili nel sistema: un processo non può richiedere tre stampanti se il sistema ne ha solo due.

Nelle ordinarie condizioni di funzionamento un processo può servirsi di una risorsa soltanto se rispetta la seguente sequenza.

1. **Richiesta.** Il processo richiede la risorsa; se la richiesta non si può soddisfare immediatamente – per esempio, perché la risorsa è attualmente in possesso di un altro processo – il processo richiedente deve attendere finché non possa acquisire tale risorsa.
2. **Uso.** Il processo può operare sulla risorsa (se, per esempio, la risorsa è una stampante, il processo può effettuare una stampa).
3. **Rilascio.** Il processo rilascia la risorsa.

La richiesta e il rilascio di risorse avvengono tramite chiamate di sistema, come illustrato nel Capitolo 2. Ne sono esempi le chiamate di sistema `request()` e `release()` di una periferica, `open()` e `close()` di un file, `allocate()` e `free()` di una porzione di memoria. In modo simile, come descritto nel Capitolo 5, la richiesta e il rilascio di semafori si possono eseguire per mezzo delle operazioni `wait()`

e `signal()` o con l'utilizzo delle funzioni `acquire()` e `release()` dei lock mutex. Quindi, ogni volta che si usa una risorsa gestita dal kernel, il sistema operativo controlla che il processo utente ne abbia fatto richiesta e che questa gli sia stata assegnata. Una tabella di sistema registra lo stato di ogni risorsa e, se questa è assegnata, indica il processo relativo. Se un processo richiede una risorsa già assegnata a un altro processo, il processo richiedente può essere accodato agli altri processi che attendono tale risorsa.

Un gruppo di processi si trova in stallo quando ogni processo del gruppo attende un evento che può essere causato solo da un altro processo che si trova nello stato di attesa. Gli eventi che interessano maggiormente in questo contesto sono l'acquisizione e il rilascio di risorse. Le risorse possono essere sia fisiche, per esempio, stampanti, unità a nastri, spazio di memoria e cicli di CPU, sia logiche, per esempio semafori, lock mutex e file. Tuttavia anche altri eventi possono condurre a situazioni di stallo, come le funzioni di IPC trattate nel Capitolo 3.

Per esaminare una situazione di stallo si consideri un sistema con tre lettori CD. Si supponga che tre processi ne possiedano uno ciascuno. Se ogni processo richiede un altro lettore CD, i tre processi entrano in stallo: ciascuno è in attesa del rilascio di un lettore, ma quest'evento può essere causato solo da uno degli altri processi che a loro volta sono in attesa. Quest'esempio descrive uno stallo causato da processi che competono per acquisire lo stesso tipo di risorsa.

Le situazioni di stallo possono implicare anche diversi tipi di risorsa. Si consideri, per esempio, un sistema con una stampante e un lettore DVD, e si supponga che il processo P_i sia in possesso del lettore e il processo P_j della stampante. Se P_i richiede la stampante e P_j il lettore DVD, si ha uno stallo.

Gli sviluppatori di applicazioni multithread devono essere consapevoli delle possibili situazioni di stallo. Gli strumenti di lock presentati nel Capitolo 5 sono progettati per evitare race condition, ma utilizzando questi strumenti gli sviluppatori devono



STALLO CON LOCK MUTEX

Vediamo come si possa incorrere in situazioni di stallo in un programma multithread di Pthread che usi i lock mutex. La funzione `pthread_mutex_init()` inizializza un semaforo mutex su cui non è attivo un lock. I lock mutex sono acquisiti e restituiti per mezzo di `pthread_mutex_lock()` e `pthread_mutex_unlock()`, rispettivamente. Se un thread tenta di acquisire un mutex già impegnato, l'invocazione di `pthread_mutex_lock()` blocca il thread finché il possessore del mutex non invochi `pthread_mutex_unlock()`.

Il segmento di codice seguente genera due lock mutex:

```
/* Crea e inizializza due lock mutex */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

► STALLO CON LOCK MUTEX (continua)

Si creano poi due thread, di nome `thread_one` e `thread_two`, che possono accedere a entrambi i lock mutex. Sono eseguiti dalle funzioni `do_work_one()` e `do_work_two()`, rispettivamente, come mostrato di seguito:

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /*
     * Fa qualcosa
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /*
     * Fa qualcosa
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

In questo esempio `thread_one` tenta di acquisire i lock mutex nell'ordine (1) `first_mutex`, (2) `second_mutex`, mentre `thread_two` tenta di acquisire i lock nell'ordine (1) `second_mutex`, (2) `first_mutex`. Lo stallo è possibile se `thread_one` acquisisce `first_mutex` mentre `thread_two` acquisisce `second_mutex`.

Si noti che lo stallo, pur essendo possibile, non si verifica se `thread_one` è in grado di acquisire e rilasciare entrambi i lock prima che `thread_two` tenti a sua volta di impossessarsene. Naturalmente, l'ordine in cui i thread vengono eseguiti dipende da come vengono gestiti dallo scheduler della CPU. L'esempio evidenzia un problema importante per la gestione dello stallo: è difficile identificare e sottoporre a test gli stalli che si verificano solo in determinate condizioni di scheduling.

prestare particolare attenzione a come i lock vengono acquisiti e rilasciati. In caso contrario possono verificarsi situazioni di stallo, come illustrato nel Paragrafo 5.7.3, nel Problema dei Cinque Filosofi.

7.2 Caratterizzazione delle situazioni di stallo

In una situazione di stallo i processi non terminano mai l'esecuzione, e le risorse del sistema vengono bloccate impedendo l'esecuzione di altri processi. Prima di trattarne le soluzioni, descriviamo più precisamente le caratteristiche di una situazione di stallo.

7.2.1 Condizioni necessarie

Si può avere una situazione di stallo *solo se* in un sistema si verificano contemporaneamente le seguenti quattro condizioni.

1. **Mutua esclusione.** Almeno una risorsa deve essere non condivisibile, vale a dire che è utilizzabile da un solo processo alla volta. Se un altro processo richiede tale risorsa, si deve ritardare il processo richiedente fino al rilascio della risorsa.
2. **Possesso e attesa.** Un processo deve essere in possesso di almeno una risorsa e attendere di acquisire risorse già in possesso di altri processi.
3. **Assenza di prelazione.** Le risorse non possono essere prelazionate, vale a dire che una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, dopo aver terminato il proprio compito.
4. **Attesa circolare.** Deve esistere un insieme $\{P_0, P_1, \dots, P_n\}$ di processi, tale che P_0 attende una risorsa posseduta da P_1 , P_1 attende una risorsa posseduta da P_2 , ..., P_{n-1} attende una risorsa posseduta da P_n e P_n attende una risorsa posseduta da P_0 .

Occorre sottolineare che tutte e quattro le condizioni devono essere vere, altrimenti non si può avere alcuno stallo. La condizione dell'attesa circolare implica la condizione di possesso e attesa, quindi le quattro condizioni non sono completamente indipendenti; tuttavia è utile considerare separatamente ciascuna condizione (Paragrafo 7.4).

7.2.2 Grafo di assegnazione delle risorse

Le situazioni di stallo si possono descrivere con maggior precisione avvalendosi di una rappresentazione detta **grafo di assegnazione delle risorse**. Si tratta di un insieme di vertici V e un insieme di archi E , con l'insieme di vertici V composto da due sottoinsiemi: $P = \{P_1, P_2, \dots, P_n\}$, che rappresenta tutti i processi del sistema, e $R = \{R_1, R_2, \dots, R_m\}$, che rappresenta tutti i tipi di risorsa del sistema.

Un arco diretto dal processo P_i al tipo di risorsa R_j si indica $P_i \rightarrow R_j$, e significa che il processo P_i ha richiesto un'istanza del tipo di risorsa R_j , e attualmente attende tale risorsa. Un arco diretto dal tipo di risorsa R_j al processo P_i si indica $R_j \rightarrow P_i$, e significa che un'istanza del tipo di risorsa R_j è assegnata al processo P_i . Un arco orientato $P_i \rightarrow R_j$ si chiama **arco di richiesta**, un arco orientato $R_j \rightarrow P_i$ si chiama **arco di assegnazione**.

Graficamente ogni processo P_i si rappresenta con un cerchio e ogni tipo di risorsa R_j si rappresenta con un rettangolo. Giacché il tipo di risorsa R_j può avere più di un'istanza, ciascuna di loro si rappresenta con un puntino all'interno del rettangolo. Occorre notare che un arco di richiesta è diretto soltanto verso il rettangolo R_j , mentre un arco di assegnazione deve designare anche uno dei puntini del rettangolo.

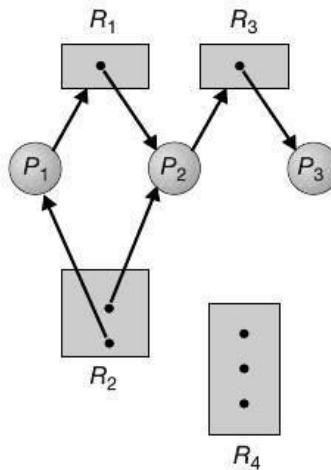


Figura 7.1 Grafo di assegnazione delle risorse.

Quando il processo P_i richiede un'istanza del tipo di risorsa R_j , si inserisce un arco di richiesta nel grafo di assegnazione delle risorse. Se questa richiesta può essere esaudita, si trasforma *immediatamente* l'arco di richiesta in un arco di assegnazione, che al rilascio della risorsa viene cancellato.

Nel grafo di assegnazione delle risorse della Figura 7.1 è illustrata la seguente situazione.

- Insiemi P , R ed E :
 - $P = \{P_1, P_2, P_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$
- Istanze delle risorse:
 - un'istanza del tipo di risorsa R_1
 - due istanze del tipo di risorsa R_2
 - un'istanza del tipo di risorsa R_3
 - tre istanze del tipo di risorsa R_4
- Stati dei processi:
 - il processo P_1 possiede un'istanza del tipo di risorsa R_2 e attende un'istanza del tipo di risorsa R_1
 - il processo P_2 possiede un'istanza dei tipi di risorsa R_1 ed R_2 e attende un'istanza del tipo di risorsa R_3
 - il processo P_3 possiede un'istanza del tipo di risorsa R_3

Data la definizione di grafo di assegnazione delle risorse, si può mostrare che, se il grafo non contiene cicli, nessun processo del sistema subisce uno stallo; se il grafo contiene un ciclo, può sopraggiungere uno stallo.

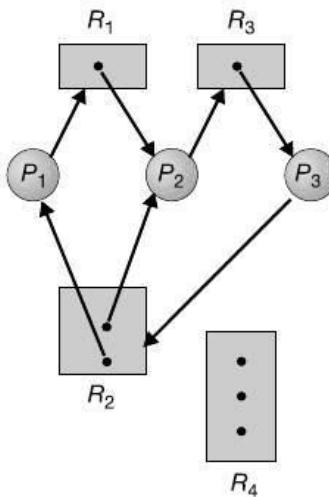


Figura 7.2 Grafo di assegnazione delle risorse con uno stallo.

Se ciascun tipo di risorsa ha esattamente un'istanza, allora l'esistenza di un ciclo implica la presenza di uno stallo; se il ciclo riguarda solo un insieme di tipi di risorsa, ciascuno dei quali ha solo un'istanza, si è verificato uno stallo. Ogni processo che si trovi nel ciclo è in stallo. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria e sufficiente per l'esistenza di uno stallo.

Se ogni tipo di risorsa ha più istanze, un ciclo non implica necessariamente uno stallo. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria ma non sufficiente per l'esistenza di uno stallo.

Per spiegare questo concetto conviene ritornare al grafo di assegnazione delle risorse della Figura 7.1. Si supponga che il processo \$P_3\$ richieda un'istanza del tipo di risorsa \$R_2\$. Poiché attualmente non è disponibile alcuna istanza di risorsa, si aggiunge un arco di richiesta \$P_3 \rightarrow R_2\$ al grafo, com'è illustrato nella Figura 7.2. A questo punto nel sistema ci sono due cicli minimi:

$$\begin{aligned} P_1 &\rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \\ P_2 &\rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \end{aligned}$$

I processi \$P_1\$, \$P_2\$ e \$P_3\$ sono in stallo: il processo \$P_2\$ attende la risorsa \$R_3\$, posseduta dal processo \$P_3\$; il processo \$P_3\$, invece, attende che il processo \$P_1\$ o \$P_2\$ rilasci la risorsa \$R_2\$; inoltre il processo \$P_1\$ attende che il processo \$P_2\$ rilasci la risorsa \$R_1\$.

Si consideri ora il grafo di assegnazione delle risorse della Figura 7.3. Anche in questo esempio c'è un ciclo:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

In questo caso, però, non si ha alcuno stallo: il processo \$P_4\$ può rilasciare la propria istanza del tipo di risorsa \$R_2\$, che si può assegnare al processo \$P_3\$, rompendo il ciclo.

Per concludere, l'assenza di cicli nel grafo di assegnazione delle risorse implica l'assenza di situazioni di stallo nel sistema. Viceversa, la presenza di un ciclo non è

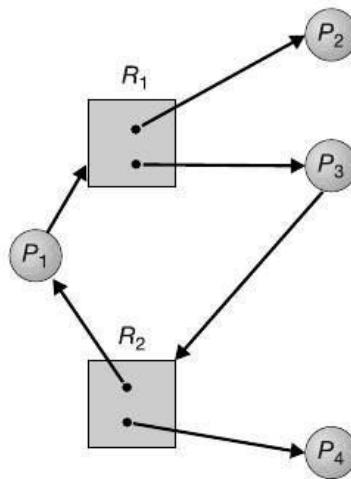


Figura 7.3 Grafo di assegnazione delle risorse con un ciclo, ma senza stallo.

sufficiente a implicare la presenza di uno stallo nel sistema. Questa osservazione è importante ai fini della gestione del problema delle situazioni di stallo.

7.3 Metodi per la gestione delle situazioni di stallo

Essenzialmente, il problema delle situazioni di stallo si può affrontare in tre modi:

- si può usare un protocollo per prevenire o evitare le situazioni di stallo, assicurando che il sistema non entri *mai* in stallo;
- si può permettere al sistema di entrare in stallo, individuarlo, e quindi eseguire il ripristino;
- si può ignorare del tutto il problema, *fingendo* che le situazioni di stallo non possano mai verificarsi nel sistema.

Quest'ultima è la soluzione usata dalla maggior parte dei sistemi operativi, compresi Linux e Windows. La scrittura di programmi che gestiscono gli stalli diventa quindi un problema dei programmatori applicativi.

Nel seguito sono spiegati brevemente tutti questi metodi. Gli algoritmi relativi sono presentati in modo dettagliato nei Paragrafi dal 7.4 al 7.7. Tuttavia, prima di procedere, vogliamo considerare anche l'opinione di quegli sviluppatori che hanno sostenuto il fatto che nessuno degli approcci di base si adatti da solo all'intero spettro di problemi di allocazione delle risorse nei sistemi operativi. Comunque tali approcci possono essere combinati, in modo da permettere la selezione del migliore per ciascuna classe di risorse del sistema.

Per assicurare che non si verifichi mai uno stallo, il sistema può servirsi di metodi di prevenzione o di metodi per evitare tale situazione. **Prevenire le situazioni di stallo** significa far uso di metodi atti ad assicurare che non si verifichi almeno una delle condizioni necessarie (Paragrafo 7.2.1). Questi metodi, discussi nel Para-

grafo 7.4, prevengono le situazioni di stallo controllando il modo in cui si devono fare le richieste delle risorse.

Per **evitare le situazioni di stallo** occorre che il sistema operativo abbia in anticipo informazioni aggiuntive riguardanti le risorse che un processo richiederà e userà durante le sue attività. Con queste informazioni aggiuntive il sistema operativo può decidere se una richiesta di risorse da parte di un processo si può soddisfare o se il processo debba invece attendere. In tale processo di decisione il sistema tiene conto delle risorse correntemente disponibili, di quelle correntemente assegnate a ciascun processo, e delle future richieste e futuri rilasci di ciascun processo. Questi metodi sono discussi nel Paragrafo 7.5.

Se un sistema non impiega né un algoritmo per prevenire né un algoritmo per evitare gli stalli, tali situazioni possono verificarsi. In un ambiente di questo tipo il sistema può servirsi di un algoritmo che ne esamini lo stato, al fine di stabilire se si è verificato uno stallo e in tal caso ricorrere a un secondo algoritmo per il ripristino del sistema. Tali argomenti sono discussi nei Paragrafi 7.6 e 7.7.

Se un sistema non fornisce alcun meccanismo per l'individuazione degli stalli e il ripristino del sistema, situazioni di stallo possono avvenire senza che ci sia la possibilità di capire cos'è successo. In questo caso la presenza di situazioni di stallo non rilevate causerà un degrado delle prestazioni del sistema; infatti vi sono risorse assegnate a processi che non si possono eseguire e un numero crescente di processi richiedono tali risorse ed entrano in stallo, fino al blocco totale del sistema che dovrà essere riavviato manualmente.

Anche se questo metodo può non sembrare una valida soluzione al problema delle situazioni di stallo viene comunque utilizzato nella maggior parte dei sistemi operativi, come accennato in precedenza. Gli aspetti economici sono importanti: ignorare la possibilità di situazioni di stallo è più conveniente rispetto ad altri approcci. Dal momento che in molti sistemi le situazioni di stallo si verificano raramente (diciamo una volta all'anno), sostenere una spesa aggiuntiva per utilizzare gli altri metodi non sembra essere così conveniente. Inoltre, i metodi utilizzati per risolvere altre situazioni possono essere utilizzati per gestire le situazioni di stallo. In alcune circostanze, un sistema si trova in uno stato di blocco (è congelato), ma non in una situazione di stallo. Si consideri, per esempio, la situazione determinata da un processo d'elaborazione in tempo reale che viene eseguito con la priorità più elevata (o qualsiasi processo in esecuzione in un sistema con scheduler senza prelazione) e che non restituisce il controllo al sistema operativo. Il sistema deve così disporre di meccanismi manuali di ripristino per queste situazioni, che può impiegare anche per le situazioni di stallo.

7.4 Prevenire le situazioni di stallo

Com'è evidenziato nel Paragrafo 7.2.1, affinché si abbia uno stallo si devono verificare quattro condizioni necessarie; perciò si può *prevenire* il verificarsi di uno stallo assicurando che almeno una di queste condizioni non possa capitare. Questo metodo è analizzato trattando separatamente ciascuna delle quattro condizioni necessarie.

7.4.1 Mutua esclusione

Deve valere la condizione di mutua esclusione: almeno una risorsa deve essere non condivisibile. Le risorse condivisibili, invece, non richiedono l’accesso mutuamente esclusivo, perciò non possono essere coinvolte in uno stallo. I file aperti per la sola lettura sono un buon esempio di risorsa condivisibile; se più processi richiedono l’apertura di un file a sola lettura, possono ottenere un accesso contemporaneo. Un processo non deve mai attendere una risorsa condivisibile. Ma poiché alcune risorse sono intrinsecamente non condivisibili, non si possono prevenire in generale le situazioni di stallo negando la condizione di mutua esclusione. Per esempio, un lock mutex non può essere contemporaneamente condiviso da diversi processi.

7.4.2 Possesso e attesa

Per assicurare che la condizione di possesso e attesa non si presenti mai nel sistema, occorre garantire che un processo che richiede una risorsa non ne possegga altre. Si può usare un protocollo che ponga la condizione che ogni processo, prima di iniziare la propria esecuzione, richieda tutte le risorse che gli servono e che esse gli siano assegnate. Questa condizione si può realizzare imponendo che le chiamate di sistema che richiedono risorse per un processo precedano tutte le altre.

Un protocollo alternativo è quello che permette a un processo di richiedere risorse solo se non ne possiede: un processo può richiedere risorse e adoperarle, ma prima di richiedere ulteriori risorse deve rilasciare tutte quelle che possiede.

Per spiegare la differenza tra questi due protocolli si può considerare un processo che copi i dati da un DVD a un file su disco, ordina il contenuto del file e quindi stampa i risultati. Nel caso del primo protocollo, il processo deve richiedere fin dall’inizio il lettore DVD, il file su disco e una stampante. La stampante rimane in suo possesso per tutta la durata dell’esecuzione, anche se si usa solo alla fine.

Il secondo protocollo prevede che il processo richieda inizialmente solo il lettore DVD e il file nel disco. Il processo copia i dati dal DVD al disco, quindi rilascia le due risorse. A questo punto richiede il file nel disco e la stampante, e dopo aver copiato il file nella stampante rilascia le due risorse e termina.

Entrambi i protocolli presentano due svantaggi principali. Innanzitutto, l’utilizzo delle risorse può risultare poco efficiente, poiché molte risorse possono essere assegnate, ma non utilizzate, per un lungo periodo di tempo. Nel caso del secondo metodo dell’esempio precedente si possono rilasciare il lettore DVD e il file nel disco, per poi richiedere il file e la stampante, solo se si ha la certezza che i dati nel file restino immutati tra il rilascio e la seconda richiesta. Se non si può garantire quest’ultima condizione, è necessario richiedere tutte le risorse all’inizio per entrambi i protocolli.

Il secondo svantaggio è dovuto al fatto che si possono verificare situazioni di attesa indefinita. Un processo che richieda più risorse molto utilizzate può trovarsi nella condizione di attenderne indefinitamente la disponibilità, poiché almeno una delle risorse di cui necessita è sempre assegnata a qualche altro processo.

7.4.3 Assenza di prelazione

La terza condizione necessaria prevede che non sia possibile avere la prelazione su risorse già assegnate. Per assicurare che questa condizione non sussista, si può impiegare il seguente protocollo. Se un processo che possiede una o più risorse ne richiede un'altra che non gli si può assegnare immediatamente (e quindi il processo deve attendere), allora si esercita la prelazione su tutte le risorse attualmente in suo possesso. Si ha cioè il rilascio implicito di queste risorse, che si aggiungono alla lista delle risorse che il processo sta attendendo; il processo viene nuovamente avviato solo quando può ottenere sia le vecchie risorse sia quella nuova che sta richiedendo.

In alternativa, quando un processo richiede alcune risorse, se ne verifica la disponibilità: se sono disponibili vengono assegnate, se non lo sono, si verifica se sono assegnate a un processo che attende altre risorse. In tal caso si sottraggono le risorse desiderate a quest'ultimo processo e si assegnano al processo richiedente. Se le risorse non sono disponibili né sono possedute da un processo in attesa, il processo richiedente deve attendere. Durante l'attesa si può avere la prelazione di alcune sue risorse; ciò può accadere solo se un altro processo le richiede. Un processo si può avviare nuovamente solo quando riceve le risorse che sta richiedendo e recupera tutte quelle a esso sottratte durante l'attesa.

Questo protocollo è applicato spesso per risorse il cui stato si può salvare e recuperare facilmente in un secondo tempo, come i registri della CPU e lo spazio di memoria, mentre non si può in generale applicare a risorse come i lock mutex e i semafori.

7.4.4 Attesa circolare

La quarta e ultima condizione necessaria per una situazione di stallo è l'attesa circolare. Un modo per assicurare che tale condizione d'attesa non si verifichi consiste nell'imporre un ordinamento totale all'insieme di tutti i tipi di risorse e imporre che ciascun processo richieda le risorse in ordine crescente.

Si supponga che $R = \{R_1, R_2, \dots, R_m\}$ sia l'insieme dei tipi di risorse. A ogni tipo di risorsa si assegna un numero intero unico che permetta di confrontare due risorse e stabilirne la relazione di precedenza nell'ordinamento. Formalmente, si definisce una funzione iniettiva, $F: R \rightarrow N$, dove N è l'insieme dei numeri naturali. Se, per esempio, l'insieme dei tipi di risorsa R contiene unità a nastri, unità disco e stampanti, la funzione F si può definire come segue:

$$F(\text{unità a nastri}) = 1$$

$$F(\text{unità disco}) = 5$$

$$F(\text{stampante}) = 12$$

Per prevenire il verificarsi di situazioni di stallo si può considerare il seguente protocollo: ogni processo può richiedere risorse solo seguendo un ordine crescente di numerazione. Ciò significa che un processo può richiedere inizialmente qualsiasi numero di istanze di un tipo di risorsa, per esempio R_i , dopo di che il processo può

richiedere istanze del tipo di risorsa R_j se e solo se $F(R_j) > F(R_i)$. Se sono necessarie più istanze dello stesso tipo di risorsa si deve presentare una singola richiesta per tutte le istanze. Per esempio, con la funzione definita precedentemente, un processo che deve impiegare contemporaneamente l'unità a nastri e la stampante deve prima richiedere l'unità a nastri e poi la stampante. In alternativa, si può stabilire che un processo, prima di richiedere un'istanza del tipo di risorsa R_j , rilasci qualsiasi risorsa R_i tale che $F(R_i) \geq F(R_j)$. Si noti inoltre che, se sono necessarie più istanze di uno stesso tipo di risorsa, queste istanze devono essere oggetto di **un'unica richiesta**.

Se si usa uno di questi due protocolli, la condizione di attesa circolare non può sussistere. Ciò si può dimostrare supponendo, per assurdo, che esista un'attesa circolare. Si supponga che l'insieme di processi coinvolti nell'attesa circolare sia $\{P_0, P_1, \dots, P_n\}$, dove P_i attende una risorsa R_i , posseduta dal processo P_{i+1} . (Sugli indici si usa l'aritmetica modulare, quindi P_n attende una risorsa R_n posseduta da P_0 .) Poiché il processo P_{i+1} possiede la risorsa R_i mentre richiede la risorsa R_{i+1} , è necessario che sia verificata la condizione $F(R_i) < F(R_{i+1})$ per tutti gli i , ma ciò implica che $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$. Per la proprietà transitiva, risulta che $F(R_0) < F(R_0)$, il che è impossibile; quindi, non può esservi attesa circolare.

Questo schema si può implementare in un programma applicativo imponendo un ordine a tutti gli oggetti di sincronizzazione del sistema. Tutte le richieste inerenti a tali oggetti dovranno seguire l'ordine crescente. Per esempio, se l'ordinamento dei lock nel programma Pthread della Figura 7.4 fosse

$$\begin{aligned}F(\text{first_mutex}) &= 1 \\F(\text{second_mutex}) &= 5\end{aligned}$$

allora `thread_two` non avrebbe potuto richiedere i lock nell'ordine inverso. Si tenga presente che la semplice esistenza di un ordinamento delle risorse non protegge dallo stallo: è infatti responsabilità degli sviluppatori di applicazioni scrivere programmi che rispettino tale ordinamento. Si noti anche che la funzione F andrebbe definita secondo l'ordine d'uso normale delle risorse del sistema. Per esempio, poiché generalmente l'unità a nastri si usa prima della stampante, è ragionevole definire $F(\text{unità a nastri}) < F(\text{stampante})$.

Anche se è responsabilità dei programmatore rispettare l'ordinamento delle risorse, è possibile sviluppare del software in grado di verificare che l'acquisizione dei lock avvenga nell'ordine corretto, e che emetta avvisi appropriati in caso contrario. Uno di tali verificatori, disponibile per le versioni BSD di UNIX (per esempio, FreeBSD), è noto con il nome di **witness**. Witness usa lock mutex per proteggere le sezioni critiche, come descritto nel Capitolo 5, e tiene traccia dinamicamente delle relazioni d'ordine fra i lock del sistema. Si consideri per esempio il programma della Figura 7.4. Se `thread_one` è il primo thread ad acquisire i lock, e se ciò avviene nell'ordine (1) `first_mutex`, (2) `second_mutex`, il programma witness registra il fatto che `first_mutex` deve essere acquisito prima di `second_mutex`. Se, in seguito, `thread_two` acquisisce i lock violando quest'ordine, il programma witness produce sulla console del sistema un messaggio di notifica.

```

/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /*
     * Fa qualcosa
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /*
     * Fa qualcosa
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}

```

Figura 7.4 Esempio di stallo.

È anche importante notare che imporre un ordinamento sui lock non garantisce l'assenza di situazioni di stallo quando i lock possono essere acquisiti dinamicamente. Per esempio, supponiamo di avere una funzione che consente di trasferire fondi tra due conti correnti. Per evitare una race condition, ogni conto ha un lock mutex associato che si può ottenere mediante la funzione `get_lock()`, come mostrato nella Figura 7.5.

Si potrebbe verificare una situazione di stallo se due thread invocassero contemporaneamente la funzione `transaction()`, passandole account distinti. In altre parole, un thread potrebbe invocare

```
transaction(checking_account, savings_account, 25);
```

e un altro potrebbe invocare

```
transaction(savings_account, checking_account, 50);
```

Lasciamo la risoluzione di questa situazione come esercizio.

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);

    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Figura 7.5 Esempio di stallo con ordinamento dei lock.

7.5 Evitare le situazioni di stallo

Gli algoritmi di prevenzione delle situazioni di stallo trattati nel Paragrafo 7.4 si basano sul controllo delle modalità di richiesta, così da assicurare che non si possa verificare almeno una delle condizioni necessarie perché si abbia uno stallo. Questo metodo può però causare effetti collaterali negativi, come uno scarso utilizzo dei dispositivi e una ridotta produttività del sistema.

Un metodo alternativo per evitare le situazioni di stallo consiste nel richiedere ulteriori informazioni sulle modalità di richiesta delle risorse. In un sistema con un’unità a nastri e una stampante, per esempio, il sistema potrebbe aver bisogno di sapere che il processo P intende richiedere prima l’unità a nastri e poi la stampante, prima di rilasciarle entrambe, mentre il processo Q richiederà prima la stampante e poi l’unità a nastri. Una volta acquisita la sequenza completa delle richieste e dei rilasci di ogni processo, il sistema può stabilire per ogni richiesta se il processo debba attendere o meno, per evitare una possibile situazione di stallo futura. In seguito a ogni richiesta, il sistema deve esaminare le risorse attualmente disponibili, le risorse attualmente assegnate a ogni processo e le richieste e i rilasci futuri per ciascun processo.

Gli algoritmi differiscono tra loro per la quantità e il tipo di informazioni richieste. Il modello più semplice e più utile richiede che ciascun processo dichiari il *numero massimo* delle risorse di ciascun tipo di cui necessita. Data questa informazione a priori, si può costruire un algoritmo capace di assicurare che il sistema non entri mai in stallo. Questo algoritmo per **evitare lo stallo** esamina dinamicamente lo stato di assegnazione delle risorse per garantire che non possa esistere una condizione di attesa circolare. Lo *stato* di assegnazione delle risorse è definito dal numero di risorse disponibili e assegnate e dalle richieste massime dei processi. Nei due paragrafi successivi esaminiamo due algoritmi per evitare le situazioni di stallo.

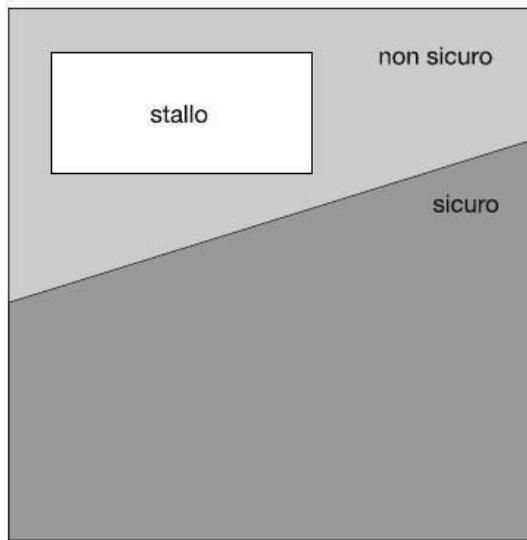


Figura 7.6 Spazi degli stati sicuri, non sicuri e di stallo.

7.5.1 Stato sicuro

Uno stato si dice *sicuro* se il sistema è in grado di assegnare risorse a ciascun processo (fino al suo massimo) in un certo ordine e impedire il verificarsi di uno stallo. Più formalmente, un sistema si trova in stato sicuro solo se esiste una **sequenza sicura**. Una sequenza di processi $\langle P_1, P_2, \dots, P_n \rangle$ è una sequenza sicura per lo stato di assegnazione attuale se, per ogni P_i , le richieste che P_i può ancora fare si possono soddisfare impiegando le risorse attualmente disponibili più le risorse possedute da tutti i P_j con $j < i$. In questa situazione, se le risorse necessarie al processo P_i non sono disponibili immediatamente, allora P_i può attendere che tutti i P_j abbiano finito, e a quel punto P_i può ottenere tutte le risorse di cui ha bisogno, completare il compito assegnato, restituire le risorse assegnate e terminare. Quando P_i termina, P_{i+1} può ottenere le risorse richieste, e così via. Se non esiste una sequenza di questo tipo, lo stato del sistema si dice *non sicuro*.

Uno stato sicuro non è di stallo. Viceversa, uno stato di stallo è uno stato non sicuro; tuttavia non tutti gli stati non sicuri sono stati di stallo (Figura 7.6). Uno stato non sicuro può condurre a uno stallo. Finché lo stato rimane sicuro, il sistema operativo può evitare il verificarsi di stati non sicuri e di stallo. In uno stato non sicuro il sistema operativo non può impedire ai processi di richiedere risorse in modo da causare uno stallo: ciò che accade negli stati non sicuri dipende dal comportamento dei processi.

	Richieste massime	Unità possedute
P_0	10	5
P_1	4	2
P_2	9	2

Per illustrare meglio quel che si è detto sopra, si consideri un sistema con 12 unità a nastri magnetici e 3 processi: P_0 , P_1 e P_2 . Il processo P_0 può richiedere 10 unità a nastri, il processo P_1 può richiederne 4 e il processo P_2 può richiedere fino a 9. Supponendo che all’istante t_0 il processo P_0 possieda 5 unità a nastri, e che i processi P_1 e P_2 ne possiedano 2 ciascuno, restano libere 3 unità a nastri.

All’istante t_0 , il sistema si trova in uno stato sicuro. La sequenza $\langle P_1, P_0, P_2 \rangle$ soddisfa la condizione di sicurezza, poiché al processo P_1 si possono assegnare immediatamente tutte le unità a nastri richieste, che saranno poi restituite (a questo punto sono disponibili 5 unità a nastri), quindi il processo P_0 può ottenere tutte le unità a nastri richieste e restituirlle (il sistema avrà 10 unità a nastri disponibili) e infine il processo P_2 potrebbe ottenere tutte le sue unità a nastri e restituirlle (rendendo disponibili tutte e 12 le unità a nastri).

Un sistema può passare da uno stato sicuro a uno stato non sicuro. Si supponga che all’istante t_1 il processo P_2 richieda un’ulteriore unità a nastri e che questa gli sia assegnata: il sistema non si trova più nello stato sicuro. A questo punto, si possono assegnare tutte le unità a nastri richieste soltanto al processo P_1 . Al momento della restituzione, il sistema avrà solo 4 unità a nastri disponibili. Poiché al processo P_0 sono assegnate 5 unità a nastri, ma il numero massimo è 10, il processo può richiederne altre 5; se lo fa, poiché queste non sono disponibili, il processo P_0 deve attendere. Analogamente, il processo P_2 può richiedere altre 6 unità a nastri ed essere costretto ad attendere; il risultato è una situazione di stallo. L’errore è stato commesso nel soddisfare la richiesta di un’ulteriore unità a nastri fatta dal processo P_2 . Se P_2 fosse stato costretto ad attendere il termine di uno degli altri processi e il conseguente rilascio delle sue risorse, la situazione di stallo si sarebbe potuta evitare.

Dato il concetto di stato sicuro, si possono definire algoritmi che permettano di evitare le situazioni di stallo. L’idea è semplice: è sufficiente assicurare che il sistema rimanga sempre in uno stato sicuro. Il sistema si trova inizialmente in uno stato sicuro; ogni volta che un processo richiede una risorsa disponibile, il sistema deve stabilire se la risorsa può essere allocata oppure se il processo debba attendere. Si soddisfa la richiesta solo se l’assegnazione lascia il sistema in uno stato sicuro.

In questo modo, se un processo richiede una risorsa attualmente disponibile può essere comunque costretto ad attendere. Quindi, l’utilizzo delle risorse può essere inferiore rispetto a quello che si avrebbe in assenza di un algoritmo per evitare le situazioni di stallo.

7.5.2 Algoritmo con grafo di assegnazione delle risorse

Quando il sistema per l’assegnazione delle risorse è tale che ogni tipo di risorsa ha una sola istanza, per evitare le situazioni di stallo si può far uso di una variante del grafo di assegnazione delle risorse definito nel Paragrafo 7.2.2. Oltre agli archi di richiesta e di assegnazione, si introduce un nuovo tipo di arco, l’arco di “rivendicazione” (*claim edge*). Un **arco di rivendicazione** $P_i \rightarrow R_j$ indica che il processo P_i può richiedere la risorsa R_j in un qualsiasi momento futuro. Quest’arco ha la stessa direzione dell’arco di richiesta, ma si rappresenta con una linea tratteggiata. Quando il

processo P_i richiede la risorsa R_j , l'arco di rivendicazione $P_i \rightarrow R_j$ diventa un arco di richiesta. Analogamente, quando P_i rilascia la risorsa R_j , l'arco di assegnazione $R_j \rightarrow P_i$ diventa un arco di rivendicazione $P_i \rightarrow R_j$.

Occorre sottolineare che le risorse devono essere rivendicate a priori nel sistema. Ciò significa che prima che il processo P_i inizi l'esecuzione, tutti i suoi archi di rivendicazione devono essere già inseriti nel grafo di assegnazione delle risorse. Questa condizione si può rendere meno stringente permettendo l'aggiunta di un arco di rivendicazione $P_i \rightarrow R_j$ al grafo solo se tutti gli archi associati al processo P_i sono archi di rivendicazione.

Si supponga che il processo P_i richieda la risorsa R_j . La richiesta si può soddisfare solo se la conversione dell'arco di richiesta $P_i \rightarrow R_j$ nell'arco di assegnazione $R_j \rightarrow P_i$ non causa la formazione di un ciclo nel grafo di assegnazione delle risorse. Possiamo verificare la condizione di sicurezza con un algoritmo di rilevamento dei cicli, che richiede un numero di operazioni dell'ordine di n^2 , dove n è il numero dei processi del sistema.

Se non esiste alcun ciclo, l'assegnazione della risorsa lascia il sistema in uno stato sicuro. Se invece si trova un ciclo, l'assegnazione conduce il sistema in uno stato non sicuro, e il processo P_i deve attendere che si soddisfino le sue richieste.

Per illustrare questo algoritmo si consideri il grafo di assegnazione delle risorse della Figura 7.7. Si supponga che P_2 richieda R_2 . Sebbene sia attualmente libera, R_2 non può essere assegnata a P_2 , poiché, com'è evidenziato nella Figura 7.8, quest'operazione creerebbe un ciclo nel grafo e un ciclo indica che il sistema è in uno stato non sicuro. Se, a questo punto, P_1 richiedesse R_2 , si avrebbe uno stallo.

7.5.3 Algoritmo del banchiere

L'algoritmo con grafo di assegnazione delle risorse non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. L'algoritmo per evitare le situazioni di stallo descritto nel seguito, pur essendo meno efficiente dello schema con grafo di assegnazione delle risorse, si può applicare a tali sistemi, ed è noto col nome di **algoritmo del banchiere**. Questo nome è stato scelto perché l'algoritmo si potrebbe impiegare in un sistema bancario per assicurare che la banca non

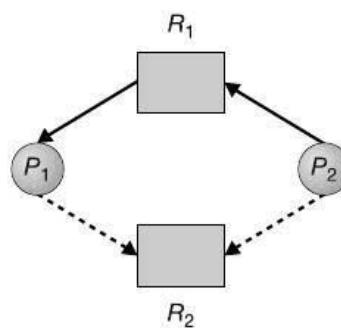


Figura 7.7 Grafo di assegnazione delle risorse per evitare le situazioni di stallo.

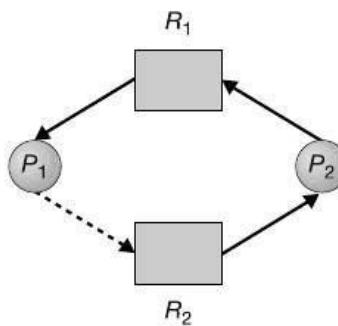


Figura 7.8 Stato non sicuro in un grafo di assegnazione delle risorse.

assegni mai tutto il denaro disponibile in modo da non poter più soddisfare le richieste di tutti i suoi clienti.

Quando si presenta al sistema, un nuovo processo deve dichiarare il numero massimo delle istanze di ciascun tipo di risorsa di cui potrà aver bisogno. Questo numero non può superare il numero totale delle risorse del sistema. Quando un utente richiede un gruppo di risorse, si deve stabilire se l'assegnazione di queste risorse lasci il sistema in uno stato sicuro. Se si rispetta tale condizione, si assegnano le risorse, altrimenti il processo deve attendere che qualche altro processo ne rilasci un numero sufficiente.

La realizzazione dell'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema. Sia n il numero di processi del sistema e m il numero dei tipi di risorsa. Sono necessarie le seguenti strutture dati.

- **Disponibili.** Un vettore di lunghezza m che indica il numero delle istanze disponibili per ciascun tipo di risorsa; $Disponibili[j] = k$, significa che sono disponibili k istanze del tipo di risorsa R_j .
- **Massimo.** Una matrice $n \times m$ che definisce la richiesta massima di ciascun processo; $Massimo[i, j] = k$ significa che il processo P_i può richiedere un massimo di k istanze del tipo di risorsa R_j .
- **Assegnate.** Una matrice $n \times m$ che definisce il numero delle istanze di ciascun tipo di risorsa attualmente assegnate a ogni processo; $Assegnate[i, j] = k$ significa che al processo P_i sono correntemente assegnate k istanze del tipo di risorsa R_j .
- **Necessità.** Una matrice $n \times m$ che indica la necessità residua di risorse relativa a ogni processo; $Necessità[i, j] = k$ significa che il processo P_i , per completare il suo compito, può avere bisogno di altre k istanze del tipo di risorsa R_j . Si osservi che $Necessità[i, j] = Massimo[i, j] - Assegnate[i, j]$.

Col trascorrere del tempo, queste strutture dati variano sia nelle dimensioni sia nei valori.

Per semplificare la presentazione dell'algoritmo del banchiere, si usano le seguenti notazioni: supponendo che X e Y siano vettori di lunghezza n , si può affermare che $X \leq Y$ se e solo se $X[i] \leq Y[i]$ per ogni $i = 1, 2, \dots, n$. Per esempio, se $X = (1, 7, 3, 2)$ e $Y = (0, 3, 2, 1)$, allora $Y \leq X$; inoltre $Y < X$ se $Y \leq X$ e $Y \neq X$.

Si possono trattare le righe delle matrici *Assegnate* e *Necessità* come vettori chiamandole rispettivamente $Assegnate_i$ e $Necessità_i$. Il vettore $Assegnate_i$ specifica le risorse correntemente assegnate al processo P_i , mentre il vettore $Necessità_i$ specifica le risorse che il processo P_i può ancora richiedere per completare il suo compito.

7.5.3.1 Algoritmo di verifica della sicurezza

L'algoritmo utilizzato per scoprire se il sistema è o non è in uno stato sicuro si può descrivere come segue.

1. Siano *Lavoro* e *Fine* vettori di lunghezza rispettivamente m e n , si inizializza $Lavoro = Disponibili$ e $Fine[i] = \text{falso}$, per $i = 1, 2, \dots, n - 1$;
2. si cerca un indice i tale che valgano contemporaneamente le seguenti relazioni:
 - a) $Fine[i] == \text{falso}$
 - b) $Necessità_i \leq Lavoro$
 se tale i non esiste, si esegue il passo 4;
3. $Lavoro = Lavoro + Assegnate_i$
 $Fine[i] = \text{vero}$
 si va al passo 2
4. se $Fine[i] == \text{vero}$ per ogni i , allora il sistema è in uno stato sicuro.

Per determinare se uno stato è sicuro tale algoritmo può richiedere un numero di operazioni dell'ordine di $m \times n^2$.

7.5.3.2 Algoritmo di richiesta delle risorse

Si descrive ora l'algoritmo che determina se le richieste possano essere soddisfatte mantenendo la condizione di sicurezza.

Sia $Richieste_i$ il vettore delle richieste per il processo P_i . Se $Richieste_i[j] == k$, allora il processo P_i richiede k istanze del tipo di risorsa R_j . Se il processo P_i effettua una richiesta di risorse, si svolgono le seguenti azioni:

1. se $Richieste_i \leq Necessità_i$, si va al passo 2, altrimenti si riporta una condizione d'errore, poiché il processo ha superato il numero massimo di richieste;
2. se $Richieste_i \leq Disponibili$, si esegue il passo 3, altrimenti P_i deve attendere poiché le risorse non sono disponibili;
3. si simula l'assegnazione al processo P_i delle risorse richieste modificando come segue lo stato di assegnazione delle risorse:

$$\begin{aligned} Disponibili &= Disponibili - Richieste_i \\ Assegnate_i &= Assegnate_i + Richieste_i \\ Necessità_i &= Necessità_i - Richieste_i \end{aligned}$$

4. Se lo stato di assegnazione delle risorse risultante è sicuro, la transazione è completata e al processo P_i si assegnano le risorse richieste. Tuttavia, se il nuovo stato è non sicuro, P_i deve attendere $Richieste_i$ e si ripristina il vecchio stato di assegnazione delle risorse.

7.5.3.3 Un esempio

Illustriamo l'uso dell'algoritmo del banchiere, considerando un sistema con cinque processi, da P_0 a P_4 , e tre tipi di risorse: A, B, e C. Il tipo di risorse A ha 10 istanze, il tipo B ha 5 istanze e il tipo C ha 7 istanze. Si supponga che all'istante T_0 si sia verificata la seguente situazione del sistema:

	Assegnate	Massimo	Disponibili
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Il contenuto della matrice *Necessità* è definito come *Massimo – Assegnate*:

	Necessità
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Possiamo affermare che il sistema si trova attualmente in uno stato sicuro; infatti, la sequenza $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ soddisfa i criteri di sicurezza. Si supponga ora che il processo P_1 richieda un'altra istanza del tipo di risorsa A e due istanze del tipo C, quindi $Richieste_1 = (1, 0, 2)$. Per stabilire se questa richiesta si possa soddisfare immediatamente verifichiamo la condizione $Richieste_1 \leq Disponibili$ (vale a dire $(1, 0, 2) \leq (3, 3, 2)$), che risulta vera. A questo punto simuliamo che questa richiesta sia stata soddisfatta, e otteniamo il seguente nuovo stato:

	Assegnate	Necessità	Disponibili
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Occorre stabilire se questo nuovo stato del sistema sia sicuro; a tale scopo si esegue l'algoritmo di verifica della sicurezza da cui risulta che la sequenza $\langle P_1, P_3, P_4, P_0, P_2 \rangle$

rispetta il requisito di sicurezza. Quindi si può soddisfare immediatamente la richiesta del processo P_1 .

Tuttavia, dovrebbe essere chiaro che, quando il sistema si trova in questo stato, una richiesta di $(3, 3, 0)$ da parte di P_4 non si può soddisfare perché non sono disponibili le risorse. Inoltre, una richiesta di $(0, 2, 0)$ da parte di P_0 non si può soddisfare, anche se le risorse sono disponibili, poiché lo stato risultante sarebbe non sicuro. L'implementazione dell'algoritmo del banchiere è lasciata come esercizio di programmazione.

7.6 Rilevamento delle situazioni di stallo

Se un sistema non si avvale di un algoritmo per prevenire o evitare lo stallo, è possibile che si verifichi una di queste situazioni. In un ambiente di questo genere, il sistema può fornire i seguenti algoritmi:

- un algoritmo che esamini lo stato del sistema per stabilire se si è verificato uno stallo;
- un algoritmo che ripristini il sistema dalla condizione di stallo.

Nell'analisi seguente sono trattati i suddetti argomenti sia per sistemi con una sola istanza di ciascun tipo di risorsa, sia per sistemi con più istanze. Tuttavia, a questo punto, occorre notare che uno schema di rilevamento e ripristino richiede un overhead che include non solo i costi per la memorizzazione delle informazioni necessarie e per l'esecuzione dell'algoritmo di rilevamento, ma anche i potenziali costi dovuti alle perdite di informazioni connesse al ripristino da una situazione di stallo.

7.6.1 Istanza singola di ciascun tipo di risorsa

Se tutte le risorse hanno una sola istanza si può definire un algoritmo di rilevamento di situazioni di stallo che fa uso di una variante del grafo di assegnazione delle risorse, detta **grafo d'attesa**, ottenuta dal grafo di assegnazione delle risorse togliendo i nodi dei tipi di risorse e componendo gli archi tra i processi.

Più precisamente, un arco da P_i a P_j del grafo d'attesa implica che il processo P_i attende che il processo P_j rilasci una risorsa di cui P_i ha bisogno. Un arco $P_i \rightarrow P_j$ esiste nel grafo d'attesa se e solo se il corrispondente grafo di assegnazione delle risorse contiene due archi $P_i \rightarrow R_q$ e $R_q \rightarrow P_j$ per qualche risorsa R_q . Nella Figura 7.9 sono illustrati un grafo di assegnazione delle risorse e il corrispondente grafo d'attesa.

Come in precedenza, nel sistema esiste uno stallo se e solo se il grafo d'attesa contiene un ciclo. Per individuare le situazioni di stallo, il sistema deve *mantenere aggiornato* il grafo d'attesa e *invocare periodicamente un algoritmo* che cerchi un ciclo all'interno del grafo. L'algoritmo per il rilevamento di un ciclo all'interno di un grafo richiede un numero di operazioni dell'ordine di n^2 , dove con n si indica il numero dei vertici del grafo.

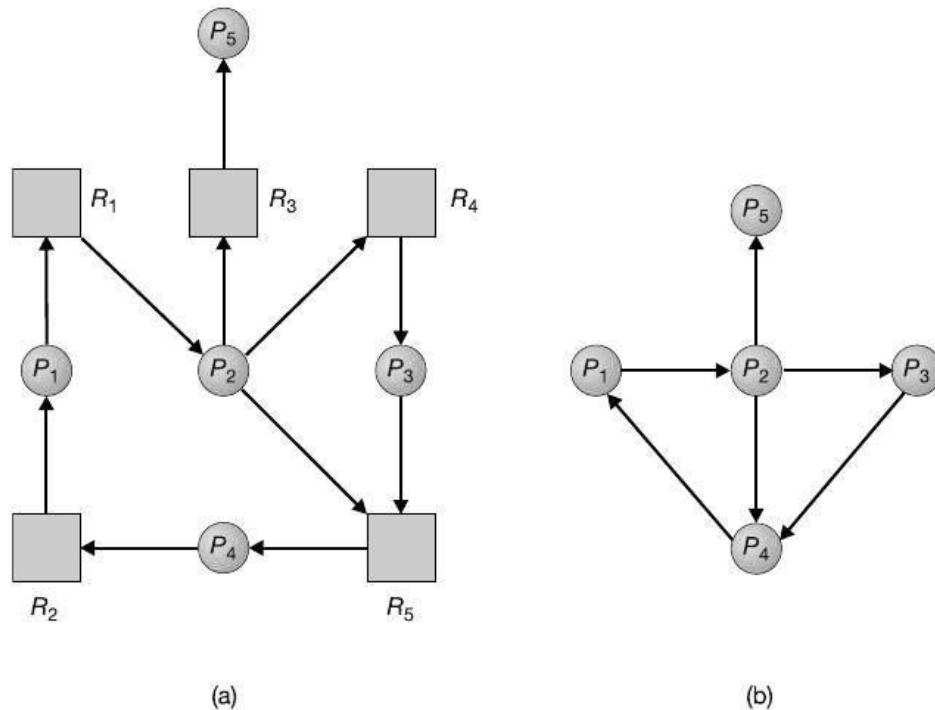


Figura 7.9 (a) Grafo di assegnazione delle risorse; (b) Grafo d'attesa corrispondente.

7.6.2 Più istanze di ciascun tipo di risorsa

Lo schema con grafo d'attesa non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. Il seguente algoritmo di rilevamento di situazioni di stallo è, invece, applicabile a tali sistemi. Esso si serve di strutture dati variabili nel tempo, simili a quelle adoperate nell'algoritmo del banchiere (Paragrafo 7.5.3).

- **Disponibili.** Vettore di lunghezza m che indica il numero delle istanze disponibili per ciascun tipo di risorsa.
- **Assegnate.** Matrice $n \times m$ che definisce il numero delle istanze di ciascun tipo di risorse correntemente assegnate a ciascun processo.
- **Richieste.** Matrice $n \times m$ che indica la richiesta attuale di ciascun processo. Se $Richieste[i, j] = k$, significa che il processo P_i sta richiedendo altre k istanze del tipo di risorsa R_j .

La relazione \leq tra due vettori si definisce come nel Paragrafo 7.5.3. Per semplificare la notazione, le righe delle matrici *Assegnate* e *Richieste* si trattano come vettori e, nel seguito, sono indicate rispettivamente come $Assegnate_i$ e $Richieste_i$. L'algoritmo di rilevamento descritto verifica ogni possibile sequenza di assegnazione per i processi che devono ancora essere completati. Questo algoritmo si può confrontare con quello del banchiere del Paragrafo 7.5.3.

1. Siano $Lavoro$ e $Fine$ vettori di lunghezza rispettivamente m e n , si inizializza $Lavoro = Disponibili$; per $i = 1, 2, \dots, n$, se $Assegnate_i \neq 0$, allora $Fine[i] = falso$, altrimenti $Fine[i] = vero$;
2. si cerca un indice i tale che valgano contemporaneamente le seguenti relazioni:
 - a) $Fine[i] == falso$
 - b) $Richieste_i \leq Lavoro$
se tale i non esiste, si esegue il passo 4;
3. $Lavoro = Lavoro + Assegnate_i$
 $Fine[i] = vero$
si torna al passo 2
4. se $Fine[i] == falso$ per qualche i , $0 \leq i < n$, allora il sistema è in stallo, inoltre, se $Fine[i] == falso$, il processo P_i è in stallo.

Tale algoritmo richiede un numero di operazioni dell'ordine di $m \times n^2$ per controllare se il sistema è in stallo.

Ci si può chiedere perché le risorse del processo P_i siano liberate (passo 3) non appena risulta valida la condizione $Richieste_i \leq Lavoro$ (passo 2.b). Sappiamo che P_i *non* è correntemente coinvolto in uno stallo, quindi, assumendo un atteggiamento ottimistico, si suppone che P_i non intenda richiedere altre risorse per completare il proprio compito, e che restituisca presto tutte le risorse. Se non si verifica l'ipotesi fatta, si potrà verificare uno stallo, che sarà rilevato quando si richiamerà nuovamente l'algoritmo di rilevamento.

Per illustrare questo algoritmo, si consideri un sistema con cinque processi, da P_0 a P_4 , e tre tipi di risorse: A , B , e C . Il tipo di risorsa A ha 7 istanze, il tipo B ha 2 istanze e il tipo C ne ha 6. Si supponga di avere, all'istante T_0 , il seguente stato di assegnazione delle risorse:

	Assegnate	Richieste	Disponibili
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Il sistema non è in stallo. Infatti eseguendo l'algoritmo troviamo che la sequenza $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ porta a $Fine[i] == vero$ per ogni i .

Si supponga ora che il processo P_2 richieda un'altra istanza di tipo C . La matrice *Richieste* viene modificata come segue:

	Richieste		
	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Ora il sistema è in stallo. Anche se si possono liberare le risorse possedute dal processo P_0 , il numero delle risorse disponibili non è sufficiente per soddisfare le richieste degli altri processi, quindi si verifica uno stallo composto dai processi P_1 , P_2 , P_3 e P_4 .

7.6.3 Uso dell'algoritmo di rilevamento

Per sapere quando è necessario ricorrere all'algoritmo di rilevamento si devono considerare i seguenti fattori:

1. *frequenza* presunta con la quale si verifica uno stallo;
2. *numero* dei processi che sarebbero influenzati da tale stallo.

Se le situazioni di stallo sono frequenti, è necessario ricorrere spesso all'algoritmo per il loro rilevamento. Le risorse assegnate a processi in stallo rimangono inattive fino all'eliminazione dello stallo. Inoltre, il numero dei processi coinvolti nel ciclo di stallo può aumentare.

Le situazioni di stallo si verificano solo quando qualche processo fa una richiesta che non si può soddisfare immediatamente; questa può essere la richiesta che chiude una catena di processi in attesa. All'estremo, si può invocare l'algoritmo di rilevamento ogni volta che non si può soddisfare immediatamente una richiesta di assegnazione. In questo caso non si identifica soltanto il gruppo di processi in stallo, ma anche lo specifico processo che ha “causato” lo stallo, anche se, in verità, ciascuno dei processi in stallo è un elemento del ciclo all'interno del grafo di assegnazione delle risorse, quindi tutti i processi sono, congiuntamente, responsabili dello stallo. Se esistono tipi di risorsa diversi, una singola richiesta può causare più cicli nel grafo delle risorse, ciascuno dei quali viene completato da quest'ultima richiesta, causata da un processo identificabile.

Naturalmente, l'uso dell'algoritmo di rilevamento per ogni richiesta aumenta notevolmente il carico computazionale. Un'alternativa meno dispendiosa è quella in cui l'algoritmo di rilevamento s'invoca a intervalli definiti, per esempio una volta ogni ora, oppure ogni volta che l'utilizzo della CPU scende sotto il 40 per cento, poiché uno stallo rende inefficienti le prestazioni del sistema e quindi porta ad una drastica

riduzione dell'utilizzo della CPU. Se l'algoritmo di rilevamento viene invocato in momenti arbitrari, poiché nel grafo delle risorse possono coesistere molti cicli, normalmente non si può dire quale fra i tanti processi in stallo abbia "causato" lo stallo.

7.7 Ripristino da situazioni di stallo

Una volta rilevato uno stallo, questo si può affrontare in diversi modi. Una soluzione consiste nell'informare l'operatore della presenza dello stallo, in modo che possa gestirlo manualmente. L'altra soluzione lascia al sistema il ripristino automatico dalla situazione di stallo. Uno stallo si può eliminare in due modi: il primo prevede semplicemente la terminazione di uno o più processi per interrompere l'attesa circolare; il secondo consiste nell'esercitare la prelazione su alcune risorse in possesso di uno o più processi in stallo.

7.7.1 Terminazione dei processi

Per eliminare le situazioni di stallo attraverso la terminazione di processi si possono adoperare due metodi; in entrambi il sistema recupera tutte le risorse assegnate ai processi terminati.

- **Terminazione di tutti i processi in stallo.** Chiaramente questo metodo interrompe il ciclo di stallo, ma l'operazione è molto onerosa; questi processi possono aver già fatto molti calcoli i cui risultati si scartano e probabilmente dovranno essere ricalcolati.
- **Terminazione di un processo alla volta fino all'eliminazione del ciclo di stallo.** Questo metodo comporta un notevole overhead, poiché, dopo aver terminato ogni processo, si deve invocare un algoritmo di rilevamento per stabilire se esistono ancora processi in stallo.

Procurare la terminazione di un processo può non essere un'operazione semplice: se il processo si trova nel mezzo dell'aggiornamento di un file, la terminazione lascia il file in uno stato scorretto; analogamente, se il processo si trova nel mezzo di una stampa di dati, prima di stampare un lavoro successivo, il sistema deve reimpostare la stampante riportandola a uno stato corretto.

Se si adopera il metodo di terminazione parziale, occorre determinare quale processo, o quali processi, in situazione di stallo devono essere terminati. Analogamente ai problemi di scheduling della CPU, si tratta di seguire un criterio. Le considerazioni sono essenzialmente economiche: si dovrebbero arrestare i processi la cui terminazione causa il minimo costo. Sfortunatamente, il termine *minimo costo* non è preciso. La scelta dei processi è influenzata da diversi fattori, tra cui i seguenti:

1. la priorità del processo;
2. il tempo trascorso in computazione e il tempo ancora necessario per completare il compito assegnato al processo;

3. la quantità e il tipo di risorse impiegate dal processo (per esempio, se si può effettuare facilmente la prelazione delle risorse);
4. la quantità di ulteriori risorse di cui il processo ha ancora bisogno per completare l'esecuzione;
5. il numero di processi che si devono terminare;
6. il tipo di processo: interattivo o batch.

7.7.2 Prelazione delle risorse

Per eliminare uno stallo si può esercitare la prelazione sulle risorse: le risorse si sottraggono in successione ad alcuni processi e si assegnano ad altri finché si ottiene l'interruzione del ciclo di stallo.

Se per gestire le situazioni di stallo s'impiega la prelazione, si devono considerare i seguenti problemi.

1. **Selezione di una vittima.** Occorre stabilire quali risorse e quali processi si devono sottoporre a prelazione. Come per la terminazione dei processi, è necessario stabilire l'ordine di prelazione allo scopo di minimizzare i costi. I fattori di costo possono includere parametri come il numero delle risorse possedute da un processo in stallo, e la quantità di tempo già spesa durante l'esecuzione del processo.
2. **Ristabilimento di un precedente stato sicuro.** Occorre stabilire che cosa fare con un processo cui è stata sottratta una risorsa. Poiché è stato privato di una risorsa necessaria, la sua esecuzione non può continuare normalmente, quindi deve essere ricondotto a un precedente stato sicuro (*rollback*) dal quale essere riavviato. Poiché, in generale, è difficile stabilire quale stato sia sicuro, la soluzione più semplice consiste nel terminare il processo e quindi riavviarlo. Benché sia più efficace effettuare il rollback solo fino al punto necessario allo scioglimento della situazione di stallo, questo metodo richiede che il sistema mantenga più informazioni sullo stato di tutti i processi in esecuzione.
3. **Attesa indefinita.** È necessario assicurare che non si verifichino situazioni d'attesa indefinita, occorre cioè garantire che le risorse non siano sottratte sempre allo stesso processo.

In un sistema in cui la scelta della vittima avviene soprattutto secondo fattori di costo, può accadere che si scelga sempre lo stesso processo; in questo caso il processo non riesce mai a completare il suo compito; si tratta di una situazione d'attesa indefinita che ogni sistema reale deve saper affrontare. Chiaramente è necessario assicurare che un processo possa essere prescelto come vittima solo un numero finito (e ridotto) di volte; la soluzione più diffusa prevede l'inclusione del numero di rollback tra i fattori di costo.

7.8 Sommario

Uno stallo (*deadlock*) si verifica quando, in un insieme di processi, ciascun processo attende un evento che può essere causato solo da uno dei processi in attesa. In linea di principio, i metodi di gestione degli stalli sono tre:

- impiegare un protocollo che prevenga o eviti le situazioni di stallo, assicurando che il sistema non vi entri mai;
- permettere al sistema di entrare in stallo, rilevarlo e quindi effettuarne il ripristino;
- ignorare del tutto il problema fingendo che nel sistema non si verifichino mai situazioni di stallo.

La terza soluzione è adottata dalla maggior parte dei sistemi operativi, compresi Linux e Windows.

Una situazione di stallo può presentarsi solo se all'interno del sistema si verificano contemporaneamente quattro condizioni necessarie: *mutua esclusione, possesso e attesa, assenza di prelazione e attesa circolare*. Per prevenire il verificarsi di situazioni di stallo è necessario assicurare che almeno una delle suddette condizioni necessarie non sia soddisfatta.

Un metodo per evitare le situazioni di stallo, piuttosto che prevenirle, consiste nel disporre di informazioni a priori su come ciascun processo intende impiegare le risorse. L'algoritmo del banchiere, per esempio, richiede la conoscenza del numero massimo di risorse di ogni classe che può essere richiesto da ciascun processo. Servendosi di queste informazioni si può definire un algoritmo per evitare le situazioni di stallo.

Se un sistema non usa alcun protocollo per assicurare che non avvengano situazioni di stallo, è necessario un metodo di rilevamento e ripristino. Per stabilire se si sia verificato uno stallo, è necessario ricorrere a un algoritmo di rilevamento; nel caso in cui si rilevi uno stallo, il sistema deve attuare il ripristino terminando alcuni processi coinvolti, oppure sottraendo risorse a qualcuno di essi.

Nel caso in cui la prelazione sia applicata per effettuare il ripristino dallo stallo, occorre tenere conto di tre questioni: la selezione di una vittima, il ristabilimento di un precedente stato sicuro e l'attesa indefinita. Nei sistemi che selezionano le vittime principalmente sulla base di fattori di costo, possono presentarsi situazioni di attesa indefinita, causando l'impossibilità del processo scelto di portare a termine il suo compito.

Infine, è stato sostenuto da parte di alcuni ricercatori che nessuno di questi approcci sia in grado, da solo, di risolvere appropriatamente l'intera gamma di problemi legati all'allocazione delle risorse nei sistemi operativi. Ciononostante, questi metodi di base possono essere combinati in modo da fornire strategie ottimali per ogni classe di risorse in un sistema.

Esercizi di ripasso

- 7.1** Elencate tre esempi di stallo che non riguardino l'informatica.
- 7.2** Supponete che un sistema sia in uno stato non sicuro. Mostrate che è possibile che i processi completino l'esecuzione senza entrare in una situazione di stallo.
- 7.3** Considerate la seguente situazione istantanea di un sistema:

	Assegnate	Massimo	Disponibili
	A B C D	A B C D	A B C D
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Rispondete alle seguenti domande servendovi dell'algoritmo del banchiere.

- a. Qual è il contenuto della matrice *Necessità*?
 - b. Il sistema è in uno stato sicuro?
 - c. Se arriva una richiesta dal processo P_1 di (0,4,2,0) tale richiesta può venire accolta immediatamente?
- 7.4** Un metodo possibile per prevenire gli stalli consiste nel disporre di una singola risorsa di ordine più elevato che deve essere richiesta prima di ogni altra risorsa. Lo stallo è possibile, per esempio, se più thread tentano di accedere agli oggetti di sincronizzazione $A \cdots E$ (tali oggetti di sincronizzazione possono includere mutex, semafori, variabili condition, e simili). Possiamo prevenire gli stalli aggiungendo un sesto oggetto F . Ogni volta che un thread vuole acquisire un lock di sincronizzazione per ciascun oggetto $A \cdots E$, deve prima acquisire il lock per l'oggetto F . Questa soluzione è conosciuta come **inclusione** (*containment*): i lock per gli oggetti $A \cdots E$ sono inclusi all'interno del lock per l'oggetto F . Paragonate questo schema con quello ad attesa circolare del Paragrafo 7.4.4.
- 7.5** Dimostrate che l'algoritmo di sicurezza presentato nel Paragrafo 7.5.3 richieda un numero di operazioni dell'ordine di $m \times n^2$.

- 7.6** Considerate un sistema informatico che esegua 5000 task al mese e non disponga né di uno schema per prevenire gli stalli né di uno schema per evitarli. Gli stalli si verificano circa due volte al mese e l'operatore deve terminare e rieseguire circa 10 task per ogni stallo. Ogni task costa circa 2 dollari (in tempo del processore), e i task terminati tendono a essere circa a metà del loro lavoro quando vengono interrotti.

Un programmatore di sistema ha stimato che un algoritmo per evitare gli stalli (come l'algoritmo del banchiere) potrebbe essere installato nel sistema con un

incremento del 10% circa del tempo medio di esecuzione per task. Siccome la macchina ha attualmente il 30% di periodo d'inattività, tutti i 5000 task al mese potrebbero ancora essere eseguiti, nonostante il tempo di completamento aumenterebbe in media del 20 per cento circa.

- Quali sono i vantaggi dell'installazione dell'algoritmo per evitare gli stalli?
- Quali sono gli svantaggi dell'installazione dell'algoritmo per evitare gli stalli?

7.7 Un sistema può individuare che alcuni dei suoi processi sono in una situazione di attesa indefinita? Se la vostra risposta è affermativa, spiegate com'è possibile. Se è negativa, spiegate come il sistema può trattare il problema dell'attesa indefinita.

7.8 Considerate la seguente politica di allocazione delle risorse. Le richieste e i rilasci di risorse sono sempre possibili. Se una richiesta di risorse non può essere soddisfatta perché queste non sono disponibili, si controllano tutti i processi bloccati in attesa di risorse. Se un processo bloccato possiede le risorse desiderate, queste vengono prelevate e assegnate al processo che le richiede. Il vettore delle risorse attese dal processo bloccato è aggiornato in modo da includere le risorse sottratte al processo.

Si consideri per esempio un sistema con tre tipi di risorse e il vettore *Disponibili* inizializzato a (4,2,2). Se il processo P_0 richiede (2,2,1), le ottiene. Se P_1 richiede (1,0,1), le ottiene. Poi, se P_0 chiede (0,0,1) viene bloccato (risorsa non disponibile). Se P_2 ora chiede (2,0,0) ottiene la risorsa disponibile (1,0,0) e quella che è stata assegnata a P_0 (poiché P_0 è bloccato). Il vettore *Assegnate* di P_0 scende a (1,2,1) e il suo vettore *Necessità* passa a (1,0,1).

- Possono verificarsi stalli? Se la risposta è affermativa, fornite un esempio. In caso di risposta negativa, specificate quale condizione necessaria non si può verificare.
- Può verificarsi un blocco indefinito? Spiegate la risposta.

7.9 Supponete di aver codificato l'algoritmo di sicurezza per evitare gli stalli e che ora vi sia richiesto di implementare l'algoritmo di rilevamento delle situazioni di stallo. È possibile farlo utilizzando semplicemente il codice dell'algoritmo di sicurezza e ridefinendo $Massimo_i = Attesa_i + Assegnate_p$, dove $Attesa_i$ è un vettore che specifica le risorse attese dal processo i e $Assegnate_i$ è definito come nel Paragrafo 7.5? Motivate la risposta.

7.10 È possibile che uno stallo coinvolga un solo processo con un singolo thread? Argomentate la risposta.

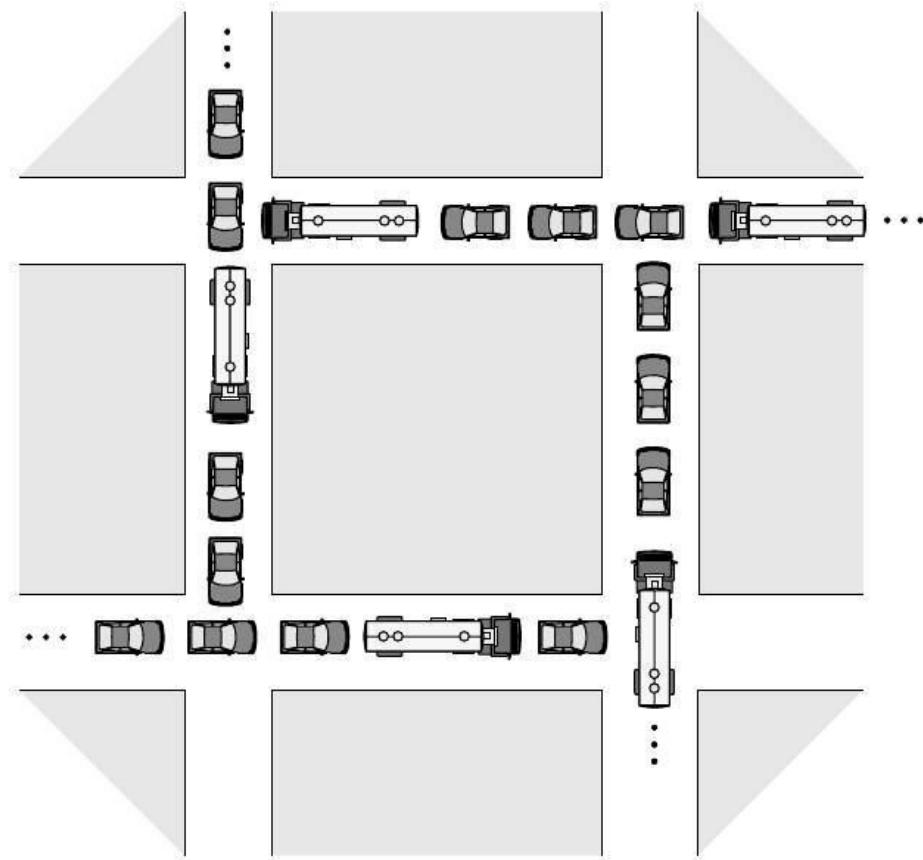


Figura 7.10 Stallo di traffico automobilistico per l’Esercizio 7.11.

Esercizi

- 7.11** Considerate lo stallo di traffico automobilistico illustrato nella Figura 7.10.
- dimostrate che le quattro condizioni necessarie per lo stallo valgono anche in questo esempio;
 - fissate una semplice regola che eviti le situazioni di stallo in questo sistema.
- 7.12** Supponiamo che un’applicazione multithread utilizzi per la sincronizzazione soltanto lock lettore-scrittore. Applicando le quattro condizioni necessarie per le situazioni di stallo, può ancora verificarsi uno stallo se si utilizzano più lock lettore-scrittore?
- 7.13** Il programma di esempio mostrato nella Figura 7.4 non conduce sempre a uno stallo. Descrivete il ruolo dello scheduler della CPU e come lo scheduler può contribuire a uno stallo in questo programma.
- 7.14** Nel Paragrafo 7.4.4 abbiamo descritto una situazione in cui preveniamo gli stalli facendo in modo che tutti i lock vengano acquisiti in un certo ordine. Tuttavia abbiamo anche notato che gli stalli sono possibili in questa situazione se due

thread invocano simultaneamente la funzione `transaction()`. Correggere la funzione `transaction()` in modo da prevenire gli stalli.

7.15 Confrontate il modello dell'attesa circolare con le varie strategie di neutralizzazione dello stallo (quali l'algoritmo del banchiere) rispetto alle seguenti problematiche:

- a. overhead di esecuzione;
- b. produttività del sistema.

7.16 In un sistema reale, né le risorse disponibili, né le richieste di risorse da parte dei processi sono stabili in lunghi periodi (mesi). Le risorse si guastano o sono sostituite, nuovi processi vanno e vengono, si acquistano e si aggiungono nuove risorse al sistema. Se le situazioni di stallo si controllano con l'algoritmo del banchiere, dite quali tra le seguenti modifiche si possono apportare con sicurezza, ossia senza introdurre la possibilità di situazioni di stallo, e in quali circostanze:

- a. aumento di *Disponibili* (aggiunta di nuove risorse);
- b. riduzione di *Disponibili* (risorsa rimossa definitivamente dal sistema);
- c. aumento di *Massimo* per un processo (il processo necessita di più risorse di quante siano permesse);
- d. riduzione di *Massimo* per un processo (il processo decide che non ha bisogno di tante risorse);
- e. aumento del numero di processi;
- f. riduzione del numero di processi.

7.17 Considerate un sistema composto da quattro risorse dello stesso tipo condivise da tre processi, ciascuno dei quali necessita di non più di due risorse. Dimostrate che non si possono verificare situazioni di stallo.

7.18 Considerate un sistema composto da m risorse dello stesso tipo, condivise da n processi. Le risorse possono essere richieste e rilasciate dai processi solo una alla volta. Dimostrate che non si possono verificare situazioni di stallo se si rispettano le seguenti condizioni:

- a. la richiesta massima di ciascun processo è compresa tra 1 e m risorse;
- b. la somma di tutte le richieste massime è minore di $m + n$.

7.19 Considerate il Problema dei cinque filosofi, supponendo che le bacchette siano disposte al centro del tavolo e che un filosofo ne possa utilizzare due qualsiasi. Supponete che le richieste di bacchette avvengano una per volta. Descrivete una semplice regola che determini se una certa richiesta possa essere soddisfatta senza causare stallo, data la distribuzione corrente delle bacchette tra i filosofi.

7.20 Considerate lo stesso contesto del problema precedente. Ipotizzate, ora, che ogni filosofo richieda tre bacchette per mangiare e che, anche qui, le richieste

siano avanzate separatamente. Descrivete delle semplici regole che determinino se una certa richiesta possa essere soddisfatta senza causare stallo, data la distribuzione corrente delle bacchette tra i filosofi.

7.21 Potete ricavare un algoritmo del banchiere che sia adatto a un solo tipo di risorsa dall’algoritmo generale del banchiere, semplicemente riducendo la dimensione dei vari array a 1. Dimostrate con un esempio che l’algoritmo del banchiere generale non può essere implementato applicando separatamente a ciascun tipo di risorsa l’algoritmo per un solo tipo di risorsa.

7.22 Si consideri la seguente situazione istantanea di un sistema:

	Allocazione				Massimo			
	A	B	C	D	A	B	C	D
P_0	3	0	1	4	5	1	1	7
P_1	2	2	1	0	3	2	1	1
P_2	3	1	2	1	3	3	2	1
P_3	0	5	1	0	4	6	1	2
P_4	4	2	1	2	6	3	2	5

Utilizzando l’algoritmo del banchiere, determinate se ciascuno dei seguenti stati è non sicuro. Se lo stato è sicuro, illustrate l’ordine in cui i processi possono essere completati. In caso contrario, spiegate perché lo stato è non sicuro.

- a. Disponibile = (0, 3, 0, 1)
- b. Disponibile = (1, 0, 0, 2)

7.23 Si consideri la seguente situazione istantanea di un sistema:

	Allocazione				Massimo			
	A	B	C	D	A	B	C	D
P_0	2	0	0	1	4	2	1	2
P_1	3	1	2	1	5	2	5	2
P_2	2	1	0	3	2	3	1	6
P_3	1	3	1	2	1	4	2	4
P_4	1	4	3	2	3	6	6	5

Rispondete alle seguenti domande usando l’algoritmo del banchiere:

- a. Mostrate che il sistema è in uno stato sicuro indicando un ordine in cui i processi possono essere completati.
- b. Se arriva una richiesta di P_1 per (1, 1, 0, 0), la richiesta può essere soddisfatta immediatamente?
- c. Se arriva una richiesta di P_4 per (0, 0, 2, 0), la richiesta può essere soddisfatta immediatamente?

7.24 Quale ipotesi ottimistica è alla base dell'algoritmo di rilevamento delle situazioni di stallo? Come potrebbe essere violata questa ipotesi?

7.25 Un ponte a corsia unica collega i due villaggi di North Tunbridge e South Tunbridge, nel Vermont. Gli agricoltori dei due villaggi usano tale ponte per portare i loro prodotti nella cittadina confinante. Il ponte rimane bloccato se un contadino diretto a nord e uno diretto a sud vi salgono contemporaneamente. (I contadini del Vermont sono ostinati: non farebbero mai marcia indietro.) Scrivete un algoritmo in pseudocodice che eviti lo stallo tramite i semafori e/o i lock mutex. Non curatevi, all'inizio, dell'attesa indefinita, ossia la situazione in cui i contadini diretti a nord impediscono a quelli diretti a sud di salire sul ponte, o viceversa.

7.26 Modificate la soluzione al problema dell'Esercizio 7.25 in modo da eliminare l'attesa indefinita.

Problemi di programmazione

7.27 Implementate la soluzione dell'Esercizio 7.25 usando la sincronizzazione POSIX. In particolare, rappresentate gli agricoltori diretti a nord e quelli diretti a sud come thread separati. Una volta che un contadino è sul ponte, il thread associato verrà sospeso per un periodo di tempo casuale, che rappresenta l'attraversamento del ponte. Progettate il vostro programma in modo da poter creare diversi thread per rappresentare gli agricoltori diretti a nord e quelli diretti a sud.

Progetto di programmazione

Algoritmo del banchiere

In questo progetto scriverete un programma multithread che implementa l'algoritmo del banchiere discusso nel Paragrafo 7.5.3. Diversi clienti richiedono e rilasciano risorse della banca. Il banchiere soddisferà una richiesta solo se questa lascerà il sistema in uno stato sicuro. Una richiesta che lascia il sistema in uno stato non sicuro viene negata. Questo esercizio di programmazione combina tre tematiche distinte: (1) il multithreading, (2) la prevenzione di race condition e (3) la prevenzione di situazioni di stallo.

Il banchiere

Il banchiere prenderà in considerazione richieste da parte di n clienti per m tipi di risorse, come indicato nel Paragrafo 7.5.3. Il banchiere terrà traccia delle risorse utilizzando le seguenti strutture dati:

```
/* possono assumere ogni valore >= 0 */
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 3
```

```
/* la quantità disponibile di ogni risorsa */
int available[NUMBER_OF_RESOURCES];

/* la richiesta massima di ogni cliente */
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* la quantità attualmente assegnata a ogni cliente */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* le rimanenti necessità di ogni cliente */
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
```

I clienti

Create n thread dei clienti che richiedono e rilasciano le risorse della banca. I clienti saranno continuamente attivi, richiedendo e poi rilasciando quantità casuali di risorse. Le richieste dei clienti saranno limitate dai rispettivi valori nell'array `need`. Il banchiere soddisfa una richiesta quando questa soddisfa l'algoritmo di sicurezza descritto nel Paragrafo 7.5.3.1. Se una richiesta non lascia il sistema in uno stato sicuro, il banchiere la nega. I prototipi delle funzione per la richiesta e il rilascio delle risorse sono i seguenti:

```
int request_resources(int customer_num, int request[]);
int release_resources(int customer_num, int release[]);
```

Queste due funzioni devono restituire 0 in caso di successo (la richiesta è stata accolta) e -1 in caso di insuccesso. Diversi thread (i clienti) faranno accesso in maniera concorrente ai dati condivisi per mezzo di queste due funzioni. L'accesso deve dunque essere controllato mediante lock mutex, per evitare race condition. Sia Pthreads che la API di Windows mettono a disposizione i lock mutex. L'uso dei lock mutex Pthreads è descritto nel Paragrafo 5.9.4, mentre i lock mutex in Windows sono descritti nel progetto dal titolo “Il problema dei produttori e dei consumatori” alla fine del Capitolo 5.

Implementazione

Si vuole invocare il programma passando da riga di comando il numero di risorse di ogni tipo. Per esempio, se ci sono tre tipi di risorse, con dieci istanze del primo tipo, cinque del secondo tipo e sette del terzo tipo, il programma va richiamato come segue:

```
./a.out 10 5 7
```

L'array `available` sarà inizializzato a questi valori. Potete inizializzare l'array `maximum` (che mantiene la richiesta massima di ogni cliente) utilizzando qualsiasi metodo troviate conveniente.

Note bibliografiche

Gran parte della ricerca sulle situazioni di stallo è stata condotta diversi anni or sono. [Dijkstra 1965a] è stato uno dei primi e più influenti studiosi del fenomeno dello stallo. [Holt 1972] è stato il primo a formalizzare la nozione di stallo tramite un modello basato sulla teoria dei grafi simile a quello presentato in questo capitolo; nello stesso lavoro, si è inoltre occupato dell'attesa indefinita. [Hyman 1985] presenta l'esempio di situazione di stallo tratto dalla legislazione del Kansas. Uno studio sulla gestione dello stallo è fornito da [Levine 2003].

I diversi algoritmi di prevenzione sono stati suggeriti da [Havender 1968], che ha progettato lo schema di ordinamento delle risorse per il sistema IBM OS/360.

L'algoritmo del banchiere è stato sviluppato per un singolo tipo di risorsa da [Dijkstra 1965a], ed è stato esteso a più tipi di risorse da [Habermann 1969].

L'algoritmo di rilevamento delle situazioni di stallo per istanze multiple di un tipo di risorsa, descritto nel Paragrafo 7.6.2, è presentato in [Coffman et al. 1971].

[Bach 1987] riporta quanti e quali sono gli algoritmi del kernel di UNIX tradizionale in grado di gestire le situazioni di stallo. Soluzioni ai problemi di stallo nelle reti sono trattate nei lavori di [Culler et al. 1998] e [Rodeheffer e Schroeder 1991].

Il programma witness per la verifica dell'ordine d'acquisizione dei lock è presentato in [Baldwin 2002].

Bibliografia

- [Bach 1987]** M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall 1987.
- [Baldwin 2002]** J. Baldwin, “Locking in the Multithreaded FreeBSD Kernel”, USENIX BSD 2002.
- [Coffman et al. 1971]** E. G. Coffman, M. J. Elphick e A. Shoshani, “System Deadlocks”, Computing Surveys, Volume 3, N. 2, p. 67–78, 1971.
- [Culler et al. 1998]** D. E. Culler, J. P. Singh e A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers Inc. 1998.
- [Dijkstra 1965]** E. W. Dijkstra, “Cooperating Sequential Processes”, Technical report, Technological University, Eindhoven, the Netherlands 1965.
- [Habermann 1969]** A. N. Habermann, “Prevention of System Deadlocks”, Communications of the ACM, Volume 12, N. 7, p. 373–377, 385, 1969.
- [Havender 1968]** J. W. Havender, “Avoiding Deadlock in Multitasking Systems”, IBM Systems Journal, Volume 7, N. 2 , p. 74–84, 1968.
- [Holt 1972]** R. C. Holt, “Some Deadlock Properties of Computer Systems”, Computing Surveys, Volume 4, N. 3, , p. 179–196, 1972.
- [Hyman 1985]** D. Hyman, *The Columbus Chicken Statute and More Bonehead Legislation*, S. Greene Press 1985.

[Levine 2003] G. Levine, “Defining Deadlock”, Operating Systems Review, Volume 37, N. 1, 2003.

[Rodeheffer e Schroeder 1991] T. L. Rodeheffer e M. D. Schroeder, “Automatic Re-configuration in Autonet”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 183–97, 1991.

Gestione della memoria

Scopo principale di un sistema di calcolo è eseguire programmi; durante l'esecuzione, i programmi e i dati cui essi accedono devono trovarsi almeno parzialmente in memoria centrale.

Per migliorare l'utilizzo della CPU e la velocità di risposta agli utenti, il calcolatore deve tenere in memoria parecchi processi. Esistono molti metodi di gestione della memoria e l'efficacia di ciascun algoritmo dipende dalla situazione. La scelta di un metodo di gestione della memoria, per un sistema specifico, dipende da molti fattori, in particolar modo dall'*architettura hardware* del sistema; ogni algoritmo richiede infatti uno specifico supporto hardware.

CAPITOLO

8

OBIETTIVI DEL CAPITOLO

- Descrizione dettagliata delle diverse organizzazioni dell'hardware della memoria.
- Analisi delle diverse tecniche di allocazione della memoria ai processi.
- Descrizione accurata del funzionamento della paginazione nei moderni sistemi elaborativi.

Memoria centrale

Nel Capitolo 6 si è descritto come è possibile condividere la CPU tra un insieme di processi. Uno dei risultati dello scheduling della CPU consiste nella possibilità di migliorare sia l'utilizzo della CPU sia la rapidità con cui il calcolatore risponde ai propri utenti; per ottenere questo aumento delle prestazioni, tuttavia, occorre tenere in memoria parecchi processi: la memoria deve, cioè, essere condivisa.

In questo capitolo si presentano diversi metodi di gestione della memoria. Gli algoritmi di gestione della memoria variano dall'approccio più semplice che accede all'hardware della macchina in maniera diretta ai metodi di paginazione e segmentazione. Ogni metodo presenta vantaggi e svantaggi. La scelta di un metodo specifico di gestione della memoria dipende da molti fattori, in particolar modo dall'architettura hardware del sistema, infatti molti algoritmi richiedono un supporto hardware. Per questa ragione in molti sistemi l'hardware e la gestione della memoria da parte del sistema operativo sono strettamente integrati.

8.1 Introduzione

Come abbiamo visto nel Capitolo 1, la memoria è fondamentale nelle operazioni di un moderno sistema di calcolo. La memoria consiste in un grande vettore di byte, ciascuno con il proprio indirizzo. La CPU preleva le istruzioni dalla memoria sulla base del contenuto del contatore di programma; tali istruzioni possono determinare ulteriori letture (*load*) e scritture (*store*) in specifici indirizzi di memoria.

Un tipico ciclo d'esecuzione di un'istruzione, per esempio, prevede che l'istruzione sia prelevata dalla memoria; quindi viene decodificata (e ciò può comportare il prelievo di operandi dalla memoria) e poi eseguita sugli eventuali operandi; i risultati si possono scrivere in memoria. La memoria vede soltanto un flusso d'indirizzi di memoria, e non sa come sono generati (contatore di programma, indicizzazione, riferimenti indiretti, indirizzamenti immediati e così via), oppure a che cosa servano (istruzioni o dati). Di conseguenza, è possibile ignorare *come* un programma genera un indirizzo di memoria, e prestare attenzione solo alla sequenza degli indirizzi di memoria generati dal programma in esecuzione.

L'analisi che sviluppiamo nel seguito comprende una panoramica degli aspetti basilari della gestione della memoria: l'hardware, la mappatura degli indirizzi simbolici in indirizzi fisici effettivi, e la distinzione fra indirizzamento logico e fisico. Concluderemo il paragrafo discutendo il linking dinamico e la condivisione delle librerie.

8.1.1 Hardware di base

La memoria centrale e i registri incorporati nel processore sono le sole aree di memorizzazione a cui la CPU può accedere direttamente. Vi sono istruzioni macchina che accettano gli indirizzi di memoria come argomenti, ma nessuna accetta gli indirizzi del disco. Pertanto, qualsiasi istruzione in esecuzione, e tutti i dati utilizzati dalle istruzioni, devono risiedere in uno di questi dispositivi di memorizzazione ad accesso diretto. I dati che non sono in memoria devono essere caricati prima che la CPU possa operare su di loro.

I registri incorporati nella CPU sono accessibili, in genere, nell'arco di un ciclo del clock della CPU. Molte CPU sono capaci di decodificare istruzioni ed effettuare semplici operazioni sui contenuti dei registri alla velocità di una o più operazioni per ciclo. Ciò non vale per la memoria centrale, cui si accede tramite una transazione sul bus della memoria che può richiedere molti cicli di clock. In tal caso il processore entra necessariamente in **stallo** (*stall*), poiché manca dei dati richiesti per completare l'istruzione che sta eseguendo. Questa situazione è intollerabile, perché gli accessi alla memoria sono frequenti. Il rimedio consiste nell'interposizione di una memoria veloce tra CPU e memoria centrale. Di solito questo buffer di memoria, detto **cache**, si trova sul chip della CPU, in modo da garantire un accesso più rapido. La cache è stata descritta nel Paragrafo 1.8.3. Nella gestione di una cache interna alla CPU l'hardware si occupa direttamente di accelerare l'accesso alla memoria, senza l'intervento del sistema operativo.

Non basta prestare attenzione alle velocità relative di accesso alla memoria fisica, ma occorre anche assicurare una corretta esecuzione delle operazioni. A tal fine, bisogna proteggere il sistema operativo dall'accesso dei processi utenti, e, in sistemi multiutente, salvaguardare i processi utenti l'uno dall'altro. Tale protezione deve essere messa in atto a livello hardware, perché solitamente il sistema operativo, per una questione di prestazioni, non interviene negli accessi della CPU alla memoria. Come si vedrà lungo tutto il capitolo, questa protezione può essere realizzata dall'hardware con meccanismi diversi. In questo paragrafo evidenziamo una delle possibili implementazioni.

Innanzitutto, bisogna assicurarsi che ciascun processo abbia uno spazio di memoria separato, in modo da proteggere i processi l'uno dall'altro: ciò è fondamentale per avere più processi caricati in memoria per l'esecuzione concorrente. Per separare gli spazi di memoria occorre poter determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi. Si può implementare il meccanismo di protezione tramite due registri, detti **registro base** e **registro limite**, come illustrato nella Figura 8.1. Il registro base contiene il più piccolo indirizzo legale della memoria fisica; il registro limite determina la dimensione dell'intervallo ammesso. Per esempio, se i registri base e limite contengono rispettivamente i valori 300040 e 120900, al programma si consente l'accesso agli indirizzi compresi tra 300040 e 420939, estremi inclusi.

Per mettere in atto il meccanismo di protezione, la CPU confronta ciascun indirizzo generato in modalità utente con i valori contenuti nei due registri. Qualsiasi tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di memoria riservata ad altri utenti comporta l'invio di una eccezione (*trap*) che restituisce il controllo al sistema operativo che, a sua volta, interpreta l'evento come un errore fatale (Figura 8.2). Questo schema impedisce a qualsiasi programma utente di alterare (accidentalmente o intenzionalmente) il codice o le strutture dati del sistema operativo o degli altri utenti.

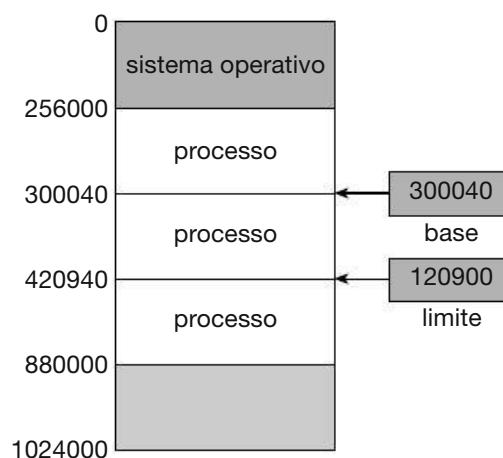


Figura 8.1 I registri base e limite definiscono lo spazio degli indirizzi logici.

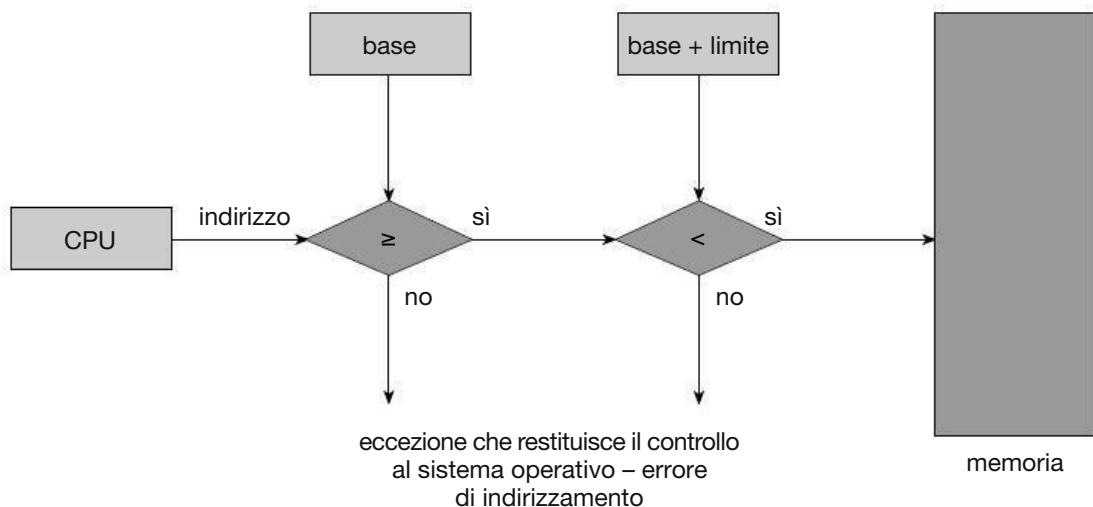


Figura 8.2 Protezione hardware degli indirizzi tramite registri base e limite.

Solo il sistema operativo può caricare i registri base e limite, grazie a una speciale istruzione privilegiata. Dal momento che le istruzioni privilegiate possono essere eseguite unicamente in modalità kernel, e poiché solo il sistema operativo può essere eseguito in tale modalità, tale schema gli consente di modificare il valore di questi registri, ma impedisce tale operazione ai programmi utenti.

Grazie all'esecuzione in modalità kernel, il sistema operativo ha la possibilità di accedere indiscriminatamente sia alla memoria a esso riservata sia a quella riservata agli utenti. Questo privilegio consente al sistema di caricare i programmi utenti nelle aree di memoria a loro riservate; di generare copie del contenuto di queste regioni di memoria (*dump*) a scopi diagnostici, qualora si verifichino errori; di modificare i parametri delle chiamate di sistema; di eseguire I/O da e verso la memoria utente e di fornire molti altri servizi. Consideriamo, per esempio, che un sistema operativo per un sistema multiprocesssing deve effettuare cambi di contesto, memorizzando lo stato di un processo dai registri nella memoria centrale prima di caricare il contesto del processo successivo dalla memoria centrale nei registri.

8.1.2 Associazione degli indirizzi

In genere un programma risiede in un disco sotto forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria e inserito nel contesto di un processo. Secondo il tipo di gestione della memoria adoperato, durante la sua esecuzione, il processo si può trasferire dalla memoria al disco e viceversa. L'insieme dei processi presenti nei dischi e che attendono d'essere trasferiti in memoria per essere eseguiti forma la **coda d'ingresso** (*input queue*).

La procedura normale, a task singolo, consiste nello scegliere uno dei processi appartenenti alla coda d'ingresso e nel caricarlo in memoria. Il processo durante l'esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina, si dichiara disponibile il suo spazio di memoria.

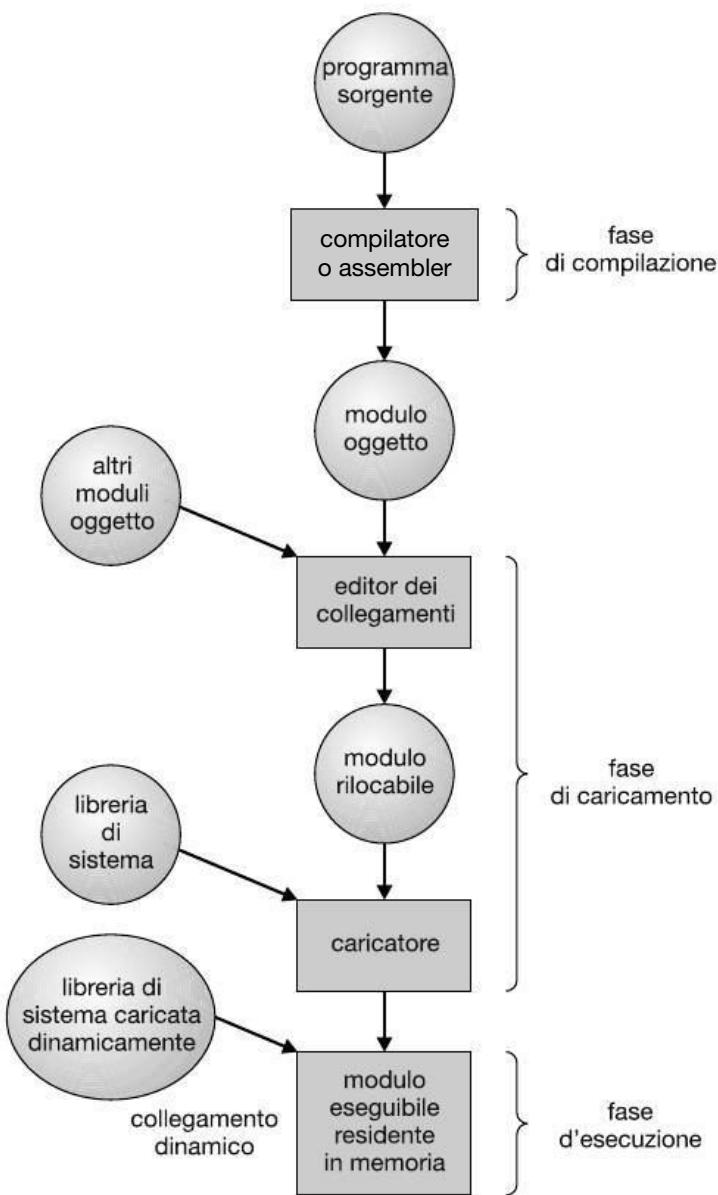


Figura 8.3 Fasi di elaborazione di un programma utente.

La maggior parte dei sistemi consente ai processi utenti di risiedere in qualsiasi parte della memoria fisica, quindi, anche se lo spazio d'indirizzi del calcolatore comincia all'indirizzo 00000, il primo indirizzo del processo utente non deve necessariamente essere 00000. Vedremo più avanti come un programma utente inserisce un processo nella memoria fisica.

Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari passi, alcuni dei quali possono essere facoltativi (Figura 8.3). In questi passi gli indirizzi sono rappresentabili in modi diversi. Generalmente gli indirizzi del programma sorgente sono simbolici (per esempio, la variabile `count`). Un compilatore di solito **associa** (bind) questi indirizzi simbolici a indirizzi rilocabili

(per esempio, “14 byte dall’inizio di questo modulo”). L’editor dei collegamenti (*linkage editor*), o il caricatore (*loader*), fa corrispondere a sua volta questi indirizzi rilocabili a indirizzi assoluti (per esempio, 74014). Ogni associazione rappresenta una corrispondenza da uno spazio d’indirizzi a un altro.

Generalmente, l’associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi passo del seguente percorso.

- **Compilazione.** Se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare **codice assoluto**. Se, per esempio, è noto a priori che un processo utente inizia alla locazione r , anche il codice generato dal compilatore comincia da quella locazione. Se, in un momento successivo, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice. I programmi per MS-DOS nel formato identificato dall’estensione .COM associano gli indirizzi fisici al tempo di compilazione.
- **Caricamento.** Se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare **codice rilocabile**. In questo caso si ritarda l’associazione finale degli indirizzi alla fase del caricamento. Se l’indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato.
- **Esecuzione.** Se durante l’esecuzione il processo può essere spostato da un segmento di memoria a un altro, si deve ritardare l’associazione degli indirizzi fino alla fase d’esecuzione. Per realizzare questo schema è necessario disporre di hardware specializzato; questo argomento è trattato nel Paragrafo 8.1.3. La maggior parte dei sistemi operativi general purpose impiega questo metodo.

Una gran parte di questo capitolo è dedicata alla spiegazione di come i vari tipi di associazione degli indirizzi si possano realizzare efficacemente in un calcolatore e, inoltre, alla discussione dell’appropriato supporto hardware per queste funzioni.

8.1.3 Spazi di indirizzi logici e fisici

Un indirizzo generato dalla CPU è normalmente chiamato **indirizzo logico**, mentre un indirizzo visto dall’unità di memoria, cioè caricato nel **registro dell’indirizzo di memoria** (*memory address register*, MAR) è normalmente chiamato **indirizzo fisico**.

I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Con il metodo di associazione nella fase d’esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. In questo caso ci si riferisce, di solito, agli indirizzi logici col termine **indirizzi virtuali**; in questo testo si usano tali termini in modo intercambiabile. L’insieme di tutti gli indirizzi logici generati da un programma è lo **spazio degli indirizzi logici**; l’insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è lo **spazio degli indirizzi fisici**. Quindi, con lo schema di associazione degli indirizzi nella fase d’esecuzione, lo spazio degli indirizzi logici differisce dallo spazio degli indirizzi fisici.

L’associazione nella fase d’esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto **unità di gestione della memoria** (*memory-management unit*, MMU).

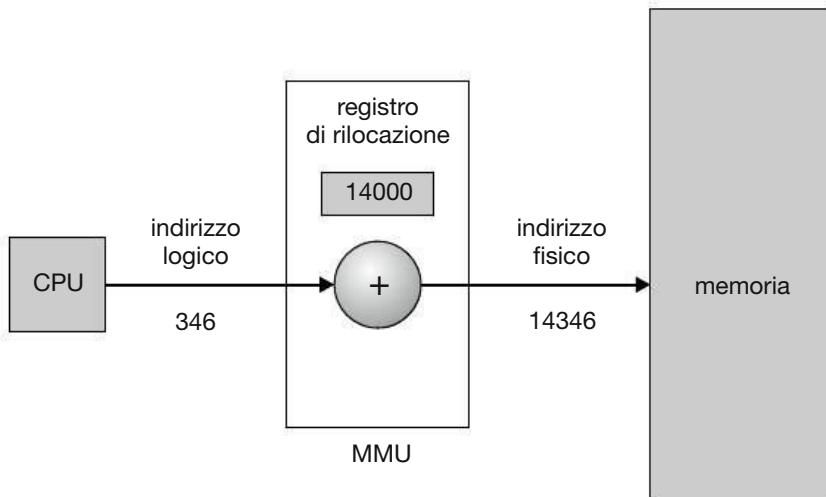


Figura 8.4 Rilocazione dinamica tramite un registro di rilocazione.

(*memory management unit*, MMU). Come viene discusso nei Paragrafi dal 8.3 al 8.5, si può scegliere tra diversi metodi di realizzazione di tale associazione. Per ora illustriamo un semplice schema di associazione degli indirizzi tramite MMU, che è una generalizzazione dello schema con registro di base descritto nel Paragrafo 8.1.1. Com'è illustrato nella Figura 8.4, il registro di base è ora denominato **registro di rilocazione**: quando un processo utente genera un indirizzo, prima dell'invio all'unità di memoria, si somma a tale indirizzo il valore contenuto nel registro di rilocazione. Per esempio, se il registro di rilocazione contiene il valore 14000, un tentativo da parte dell'utente di accedere alla locazione 0 è dinamicamente rilocato alla locazione 14000; un accesso alla locazione 346 è mappato alla locazione 14346.

Il programma utente non vede mai i reali indirizzi fisici. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò sempre come il numero 346. Solo quando assume il ruolo di un indirizzo di memoria (magari in una load o una store indiretta), si riloca il numero sulla base del contenuto del registro di rilocazione. Il programma utente tratta indirizzi logici, l'architettura del sistema converte gli indirizzi logici in indirizzi fisici. Questa forma di collegamento nella fase d'esecuzione è stata trattata nel Paragrafo 8.1.2. La locazione finale di un riferimento a un indirizzo di memoria non è determinata finché non si compie effettivamente il riferimento.

In questo caso esistono due diversi tipi di indirizzi: gli indirizzi logici (nell'intervallo da 0 a *max*) e gli indirizzi fisici (nell'intervallo da *r* + 0 a *r* + *max* per un valore di base *r*). Il programma utente genera solo indirizzi logici e pensa che il processo sia eseguito nelle posizioni da 0 a *max*. Tuttavia questi indirizzi logici devono essere mappati in indirizzi fisici prima d'essere usati. Il concetto di *spazio d'indirizzi logici* mappato su uno *spazio d'indirizzi fisici* separato è fondamentale per una corretta gestione della memoria.

8.1.4 Caricamento dinamico

Nella discussione svolta fin’ora, era necessario che l’intero programma e i dati di un processo fossero presenti nella memoria fisica perché il processo potesse essere eseguito. La dimensione di un processo era quindi limitata alle dimensioni della memoria fisica. Per migliorare l’utilizzo della memoria si può ricorrere al **caricamento dinamico** (*dynamic loading*), mediante il quale si carica una procedura solo quando viene richiamata; tutte le procedure si tengono su disco in un formato di caricamento rilocabile. Si carica il programma principale in memoria e quando, durante l’esecuzione, una procedura deve richiamarne un’altra, controlla innanzitutto che sia stata caricata. Se non è stata caricata, si richiama il *linking loader* rilocabile per caricare in memoria la procedura richiesta e aggiornare le tabelle degli indirizzi del programma in modo che registrino questo cambiamento. A questo punto il controllo passa alla procedura appena caricata.

Il vantaggio dato dal caricamento dinamico consiste nel fatto che una procedura viene caricata solo quando serve. Questo metodo è utile soprattutto quando servono grandi quantità di codice per gestire casi non frequenti, per esempio le procedure di gestione degli errori. In questi casi, anche se un programma può avere dimensioni totali elevate, la parte effettivamente usata, e quindi effettivamente caricata, può essere molto più piccola.

Il caricamento dinamico non richiede un supporto particolare del sistema operativo. Spetta agli utenti progettare i programmi in modo da trarre vantaggio da un metodo di questo tipo. Il sistema operativo può tuttavia aiutare il programmatore fornendo librerie di procedure che realizzano il caricamento dinamico.

8.1.5 Linking dinamico e librerie condivise

Le librerie collegate dinamicamente sono librerie di sistema che vengono collegate ai programmi utente quando questi vengono eseguiti (si faccia riferimento alla Figura 8.3). Alcuni sistemi operativi consentono solo il **collegamento statico** (*static linking*), in cui le librerie di sistema sono trattate come qualsiasi altro modulo oggetto e combinate dal caricatore nell’immagine binaria del programma. Il concetto di linking dinamico, invece, è analogo a quello di caricamento dinamico. Invece di differire il caricamento di una procedura fino al momento dell’esecuzione, si differisce il collegamento. Questa caratteristica si usa soprattutto con le librerie di sistema, per esempio le librerie di subroutine del linguaggio. Senza questo strumento tutti i programmi di un sistema dovrebbero disporre, all’interno dell’eseguibile, di una copia della libreria di linguaggio (o almeno delle procedure cui il programma fa riferimento). Tutto ciò sprezza spazio nei dischi e in memoria centrale.

Con il linking dinamico, invece, per ogni riferimento a una procedura di libreria s’inscrive all’interno dell’eseguibile una piccola porzione di codice di riferimento (*stub*), che indica come localizzare la giusta procedura di libreria residente in memoria o come caricare la libreria se la procedura non è già presente. Durante l’esecuzione, lo stub controlla se la procedura richiesta è già in memoria, altrimenti provvede a caricarla; in entrambi i casi lo stub sostituisce se stesso con l’indirizzo della proce-

dura, che viene poi eseguita. In questo modo, quando si raggiunge nuovamente quel segmento del codice, si esegue direttamente la procedura di libreria, senza costi aggiuntivi per il linking dinamico. Con questo metodo tutti i processi che usano una libreria del linguaggio eseguono la stessa copia del codice della libreria.

Questa funzionalità si può estendere anche agli aggiornamenti delle librerie, per esempio per la correzione di errori. Una libreria si può sostituire con una nuova versione, e tutti i programmi che fanno riferimento a quella libreria usano automaticamente la nuova versione. Senza il linking dinamico tutti questi programmi devono essere nuovamente linkati per accedere alla nuova libreria. Affinché i programmi non eseguano accidentalmente nuove versioni di librerie incompatibili, sia nel programma sia nella libreria si inserisce un'informazione relativa alla versione. È possibile caricare in memoria più di una versione della stessa libreria, e ciascun programma si serve dell'informazione sulla versione per decidere quale copia debba usare. Se le modifiche sono di piccola entità, il numero di versione resta invariato; se l'entità delle modifiche diviene rilevante, si aumenta anche il numero di versione. Perciò, solo i programmi compilati con la nuova versione della libreria subiscono gli effetti delle modifiche incompatibili incorporate nella libreria stessa. I programmi linkati prima dell'installazione della nuova libreria continuano ad avvalersi della vecchia libreria. Questo sistema è noto anche con il nome di **librerie condivise**.

A differenza del caricamento dinamico, il linking dinamico e le librerie condivise richiedono generalmente l'assistenza del sistema operativo. Se i processi presenti in memoria sono protetti l'uno dall'altro, il sistema operativo è l'unica entità che può controllare se la procedura richiesta da un processo è nello spazio di memoria di un altro processo, o che può consentire l'accesso di più processi agli stessi indirizzi di memoria. Questo concetto è sviluppato nell'analisi della paginazione, nel Paragrafo 8.5.4.

8.2 Avvicendamento dei processi (swapping)

Per essere eseguito, un processo deve trovarsi nella memoria centrale. Un processo, tuttavia, può essere temporaneamente tolto dalla memoria centrale e spostato in una **memoria ausiliaria** (*backing store*) e in seguito riportato in memoria per continuare l'esecuzione. Questo procedimento è illustrato nella Figura 8.5 e si chiama **avvicendamento dei processi in memoria** – o, più brevemente, **swapping**. Grazie all'avvicendamento dei processi lo spazio totale degli indirizzi fisici di tutti i processi può eccedere la reale dimensione della memoria fisica del sistema, aumentando così il grado di multiprogrammazione possibile.

8.2.1 Avvicendamento standard

L'avvicendamento standard riguarda lo spostamento dei processi tra la memoria centrale e una memoria ausiliaria (*backing store*). La memoria ausiliaria è di solito costituita da un disco veloce. Tale memoria deve essere abbastanza ampia da contenere le copie di tutte le immagini di memoria di tutti i processi utenti, e deve permettere un accesso diretto a dette immagini di memoria. Il sistema mantiene una

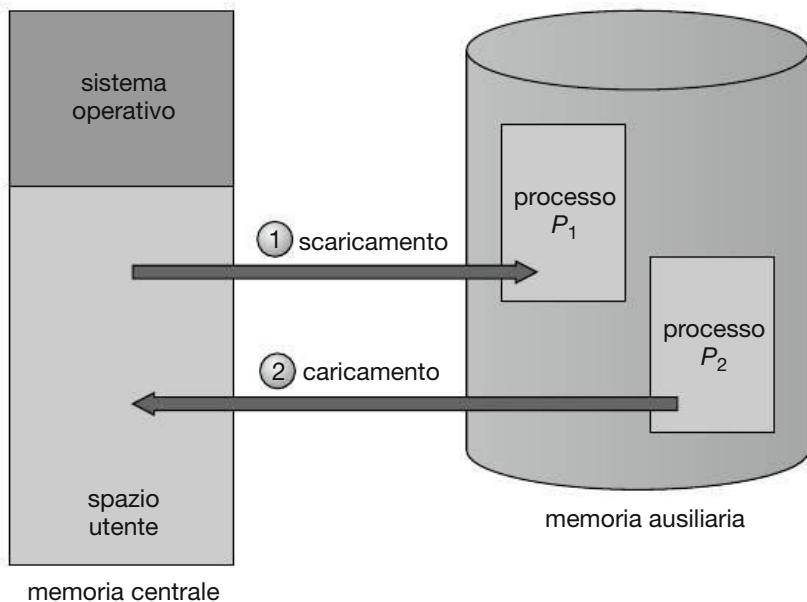


Figura 8.5 Avvicendamento (swapping) di due processi con un disco come memoria ausiliaria.

coda dei processi pronti (*ready queue*) formata da tutti i processi pronti per l'esecuzione, le cui immagini di memoria si trovano in memoria ausiliaria o in memoria centrale. Quando lo scheduler della CPU decide di eseguire un processo, richiama il dispatcher, che controlla se il primo processo della coda si trova in memoria centrale. Se non si trova in memoria, e in questa non c'è spazio libero, il dispatcher scarica un processo dalla memoria e vi carica il processo richiesto dallo scheduler della CPU, quindi ricarica normalmente i registri e trasferisce il controllo al processo selezionato.

In un tale sistema d'avvicendamento, il tempo di cambio di contesto (*context-switch time*) è piuttosto elevato. Per avere un'idea della sua durata si pensi a un processo utente di 100 MB e a una memoria ausiliaria costituita da un normale hard disk con velocità di trasferimento di 50 MB al secondo. Il trasferimento effettivo del processo di 100 MB da e in memoria richiede:

$$100 \text{ MB} / 50 \text{ MB al secondo} = 2 \text{ secondi}$$

Il tempo di avvicendamento è di 2000 millisecondi. Dal momento che dobbiamo effettuare lo spostamento in entrambe le direzioni (scaricare e poi ricaricare il processo), il tempo totale di avvicendamento è di circa 4.000 millisecondi. (Stiamo in questo caso ignorando altri aspetti relativi alle prestazioni del disco, che tratteremo nel Capitolo 10).

Occorre notare che la maggior parte del tempo d'avvicendamento è data dal tempo di trasferimento. Il tempo di trasferimento totale è direttamente proporzionale alla quantità di memoria trasferita. In un calcolatore con 4 GB di memoria centrale e un sistema operativo residente di 1 GB, la massima dimensione possibile per un processo utente è di 3 GB. Tuttavia molti processi utenti possono essere molto più piccoli, per

esempio 100 MB. Il trasferimento di un processo di 100 MB può concludersi in 2 secondi, mentre per il trasferimento di 3 GB sono necessari 60 secondi. Perciò sarebbe utile sapere esattamente quanta memoria *sia* effettivamente usata da un processo utente e non solo quanta questo *potrebbe* teoricamente usarne, poiché in questo caso è necessario trasferire solo quanto è effettivamente utilizzato, riducendo il tempo d'avvicendamento. Affinché questo metodo risulti efficace, l'utente deve tenere informato il sistema su tutte le modifiche apportate ai requisiti di memoria; quindi un processo con requisiti di memoria dinamici deve impiegare chiamate di sistema (`request_memory()` e `release_memory()`) per informare il sistema operativo delle modifiche da apportare alla memoria.

L'avvicendamento dei processi è soggetto ad altri vincoli. Per scaricare un processo dalla memoria è necessario essere certi che sia completamente inattivo. Particolare importanza ha l'attesa di I/O: quando decidiamo di scaricare un processo per liberare la memoria, tale processo può essere nell'attesa del completamento di un'operazione di I/O. Tuttavia, se un dispositivo di I/O accede in modo asincrono alle aree di I/O della memoria (*buffer*) utente, il processo non può essere scaricato. Si supponga che l'operazione di I/O sia stata accodata, perché il dispositivo era occupato. Se il processo P_2 s'avvicendasse al processo P_1 , l'operazione di I/O potrebbe tentare di usare la memoria che attualmente appartiene al processo P_2 . Questo problema si può risolvere in due modi: non scaricando dalla memoria un processo con operazioni di I/O pendenti, oppure eseguendo operazioni di I/O solo in buffer del sistema operativo. Trasferimenti fra tali aree del sistema operativo e la memoria assegnata al processo possono poi avvenire solo quando il processo è presente in memoria centrale. Si noti che questo meccanismo di memorizzazione (*double buffering*) aggiunge overhead. Abbiamo infatti bisogno di copiare nuovamente i dati, dalla memoria del kernel alla memoria utente, prima che il processo utente possa accedervi.

Attualmente l'avvicendamento nella sua forma standard si usa in pochi sistemi; richiede infatti un elevato tempo di trasferimento, e consente un tempo di esecuzione troppo breve per essere considerato una soluzione ragionevole al problema di gestione della memoria. Versioni modificate dell'avvicendamento dei processi si trovano comunque in molti sistemi, tra cui UNIX, Linux, e Windows. In una sua forma modificata l'avvicendamento è di norma disattivato e si avvia solo quando la quantità di memoria libera (memoria non utilizzata, disponibile per il sistema o i processi) scende al di sotto di una soglia fissata. L'avvicendamento dei processi viene nuovamente disabilitato qualora la quantità di memoria libera aumenta. Un'altra variante prevede lo spostamento di porzioni di processi anziché di interi processi, in modo da diminuire il tempo di avvicendamento. In genere, queste forme di avvicendamento lavorano in combinazione con la memoria virtuale, di cui ci occupiamo nel Capitolo 9.

8.2.2 Avvicendamento di processi nei sistemi mobili

Mentre la maggior parte dei sistemi operativi per PC e server supporta qualche versione modificata di avvicendamento, i sistemi mobili non supportano in genere alcuna forma di avvicendamento. I dispositivi mobili utilizzano solitamente, come memoria

di massa, la memoria flash al posto dei dischi rigidi, più voluminosi. Il vincolo che ne deriva in termini di spazio è una delle ragioni per cui i progettisti di sistemi operativi mobili evitano l'avvicendamento di processi. Tra le altre ragioni vi sono il numero limitato di scritture che una memoria flash può sopportare prima di diventare inaffidabile e la scarsa velocità di trasferimento tra memoria centrale e memoria flash.

Invece di usare l'avvicendamento dei processi, se la memoria disponibile scende al di sotto di una certa soglia, iOS di Apple *chiede* alle applicazioni di rinunciare volontariamente alla memoria allocata. I dati di sola lettura (come il codice) vengono rimossi dal sistema e successivamente ricaricati dalla memoria flash, se necessario. I dati che sono stati modificati (per esempio lo stack) non vengono rimossi. Tuttavia, tutte le applicazioni che non riescono a liberare memoria a sufficienza possono essere terminate dal sistema operativo.

Android non supporta l'avvicendamento e adotta una strategia simile a quella utilizzata da iOS. Anche Android può terminare un processo qualora la memoria libera disponibile non sia sufficiente. Tuttavia, prima di terminarlo, scrive lo stato dell'applicazione (*application state*) nella memoria flash, in modo che il processo possa essere rapidamente riavviato.

A causa di queste limitazioni, gli sviluppatori per dispositivi mobili devono allocare e rilasciare memoria con molta cura, in modo da garantire che le loro applicazioni non utilizzino troppa memoria e non soffrano di *memory leak* (“perdite di memoria”). Si noti che sia iOS sia Android supportano la paginazione e hanno quindi capacità di gestione della memoria. Si parlerà di paginazione più avanti, in questo capitolo.

8.3 Allocazione contigua della memoria

La memoria centrale deve contenere sia il sistema operativo sia i vari processi utenti; perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente. In questo paragrafo si tratta il primo metodo di allocazione della memoria, l'allocazione contigua.

La memoria centrale di solito si divide in due partizioni, una per il sistema operativo residente e una per i processi utente. Il sistema operativo si può collocare sia nella parte bassa sia nella parte alta della memoria. Il fattore che incide in modo decisivo su tale scelta è generalmente la posizione del vettore delle interruzioni. Poiché si trova spesso nella parte bassa dello spazio degli indirizzi fisici, i programmati collocano di solito anche il sistema operativo nella parte bassa della memoria. Per questo motivo nel seguito prendiamo in considerazione solo la situazione in cui il sistema operativo risiede in quest'area di memoria.

Generalmente si vuole che più processi utenti risiedano contemporaneamente in memoria centrale. Perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda d'ingresso che attendono di essere caricati in memoria. Con l'**allocazione contigua della memoria**, ciascun processo è contenuto in una singola sezione di memoria contigua a quella che contiene il processo successivo.

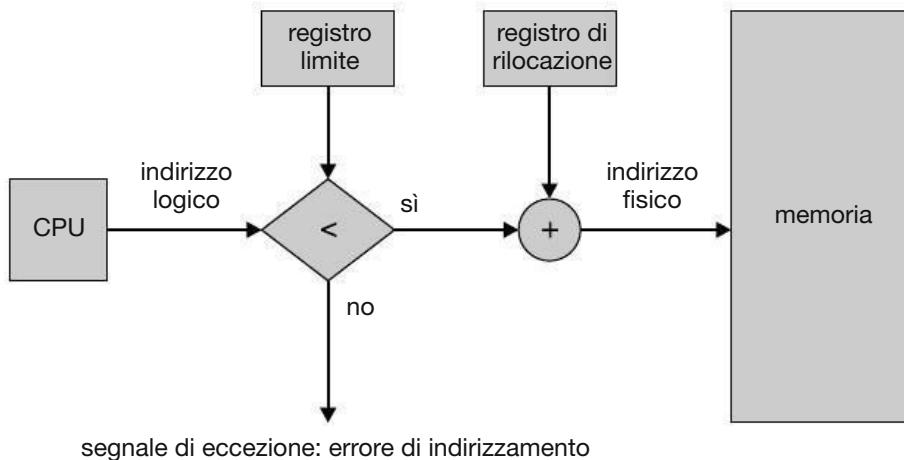


Figura 8.6 Registri di rilocazione e limite.

8.3.1 Protezione della memoria

Prima di trattare l’allocazione della memoria, dobbiamo soffermarci sul problema della protezione della memoria. Possiamo evitare che un processo acceda alla memoria che non gli appartiene combinando due idee discusse in precedenza. Se abbiamo un sistema con un registro di rilocazione (Paragrafo 8.1.3) e un registro limite (Paragrafo 8.1.1) abbiamo già raggiunto il nostro obiettivo. Il registro di rilocazione contiene il valore dell’indirizzo fisico minore; il registro limite contiene l’intervallo di indirizzi logici, per esempio, *rilocazione* = 100.040 e *limite* = 74.600. Ogni indirizzo logico deve cadere nell’intervallo specificato dal registro limite; la MMU fa corrispondere *dinamicamente* l’indirizzo fisico all’indirizzo logico sommando a quest’ultimo il valore contenuto nel registro di rilocazione (Figura 8.6), e invia l’indirizzo risultante alla memoria.

Quando lo scheduler della CPU seleziona un processo per l’esecuzione, il dispatcher, durante l’esecuzione del cambio di contesto, carica il registro di rilocazione e il registro limite con i valori corretti. Poiché si confronta ogni indirizzo generato dalla CPU con i valori contenuti in questi registri, si possono proteggere il sistema operativo, i programmi e i dati di altri utenti da tentativi di modifiche da parte del processo in esecuzione.

Lo schema con registro di rilocazione consente al sistema operativo di cambiare dinamicamente le proprie dimensioni. Tale flessibilità è utile in molte situazioni; il sistema operativo, per esempio, contiene codice e spazio di memoria per i driver dei dispositivi; se uno di questi, o un altro servizio del sistema operativo, non è comune-mente usato, è inutile tenerne in memoria codice e dati, poiché lo spazio occupato si potrebbe usare per altri scopi. Talvolta questo codice si chiama codice **transiente** del sistema operativo, poiché s’inscrive secondo le necessità; l’uso di tale codice cambia le dimensioni del sistema operativo durante l’esecuzione del programma.

8.3.2 Allocazione della memoria

Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in **partizioni** di dimensione fissa. Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato dal numero di partizioni. In questo **metodo delle partizioni multiple** quando una partizione è libera può essere occupata da un processo presente nella coda d'ingresso; terminato il processo, la partizione diviene nuovamente disponibile per un altro processo. Originariamente questo metodo, detto MFT, si usava nel sistema operativo IBM OS/360, ma attualmente non è più in uso. Il metodo descritto di seguito, detto MVT, è una generalizzazione del metodo con partizioni fisse e si usa soprattutto in ambienti batch. Si noti, tuttavia, che molte idee che lo riguardano sono applicabili agli ambienti time sharing nei quali si fa uso della segmentazione semplice per la gestione della memoria (Paragrafo 8.4).

Nello schema a partizione variabile il sistema operativo conserva una tabella in cui sono indicate le partizioni di memoria disponibili e quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utenti; si tratta di un grande blocco di memoria disponibile, un **buco** (*hole*). Nel lungo periodo, come si vede nel seguito, la memoria contiene una serie di buchi di diverse dimensioni.

Quando entrano nel sistema, i processi vengono inseriti in una coda d'ingresso. Per determinare a quali processi si debba assegnare la memoria, il sistema operativo tiene conto dei requisiti di memoria di ciascun processo e della quantità di spazio di memoria disponibile. Quando a un processo si assegna dello spazio, il processo stesso viene caricato in memoria e può quindi competere per il controllo della CPU. Al termine, rilascia la memoria che gli era stata assegnata, e il sistema operativo può impiegarla per un altro processo presente nella coda d'ingresso.

In ogni istante, quindi, abbiamo una lista delle dimensioni dei blocchi liberi e una coda d'ingresso. Il sistema operativo può ordinare la coda d'ingresso secondo un algoritmo di scheduling. La memoria si assegna ai processi della coda finché si possono soddisfare i requisiti di memoria del processo successivo, cioè finché esiste un blocco di memoria (o buco) disponibile, sufficientemente grande da accogliere quel processo. Se ciò non avviene, il sistema operativo può attendere che si renda disponibile un blocco sufficientemente grande, oppure può scorrere la coda d'ingresso per verificare se sia possibile soddisfare le richieste di memoria più limitate di qualche altro processo.

In generale, è sempre presente un insieme di buchi di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un buco di dimensioni sufficienti per contenerlo. Se è troppo grande, il buco viene diviso in due parti: si assegna una parte al processo in arrivo e si riporta l'altra nell'insieme dei buchi. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nell'insieme dei buchi; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formarne uno più grande. A questo punto il sistema può controllare se vi siano processi nell'attesa di spazio di memoria, e se la memoria appena liberata e ricombinata possa soddisfare le richieste di qualcuno fra tali processi.

Questa procedura è una particolare istanza del più generale problema di **allocazione dinamica della memoria**, che consiste nel soddisfare una richiesta di dimen-

sione n data una lista di buchi liberi. Le soluzioni sono numerose. I criteri più usati per scegliere un buco libero tra quelli disponibili nell'insieme sono i seguenti.

- **First-fit.** Si assegna il *primo* buco abbastanza grande. La ricerca può cominciare sia dall'inizio dell'insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi.
- **Best-fit.** Si assegna il *più piccolo* buco in grado di contenere il processo. Si deve compiere la ricerca in tutta la lista, a meno che questa non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più piccole.
- **Worst-fit.** Si assegna il buco *più grande*. Anche in questo caso si deve esaminare tutta la lista, a meno che non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più grandi, che possono essere più utili delle parti più piccole ottenute col criterio best-fit.

Con l'uso di simulazioni si è dimostrato che sia first-fit sia best-fit sono migliori rispetto a worst-fit in termini di risparmio di tempo e di utilizzo di memoria. D'altra parte nessuno dei due è chiaramente migliore dell'altro per quel che riguarda l'utilizzo della memoria ma, in genere, first-fit è più veloce.

8.3.3 Frammentazione

Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di **frammentazione esterna**. Caricando e rimuovendo i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna se lo spazio di memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria è frammentata in tanti piccoli buchi. Questo problema di frammentazione può essere molto grave; nel caso peggiore può verificarsi un blocco di memoria libera, sprecata, tra ogni coppia di processi. Se tutti questi piccoli pezzi di memoria costituissero in un unico blocco libero di grandi dimensioni, si potrebbero eseguire molti più processi.

Sia che si adotti l'uno o l'altro, l'impiego di un determinato criterio può influire sulla quantità di frammentazione: in alcuni sistemi dà migliori risultati il first-fit, in altri dà migliori risultati il best-fit; un altro elemento rilevante è quale sia l'estremità assegnata di un blocco libero (se la parte inutilizzata è quella in alto o quella in basso). A prescindere dal tipo di algoritmo usato, la frammentazione esterna è un problema.

La sua gravità dipende dalla quantità totale di memoria e dalla dimensione media dei processi. L'analisi statistica dell'algoritmo first-fit, per esempio, rivela che, pur con alcune ottimizzazioni, per n blocchi assegnati, si perdono altri $0,5 n$ blocchi a causa della frammentazione, ciò significa che potrebbe essere inutilizzabile un terzo della memoria. Questa caratteristica è nota come **regola del 50 per cento**.

La frammentazione può essere interna oltre che esterna. Si consideri uno schema a partizioni multiple con un buco di 18.464 byte. Supponendo che il processo successivo richieda 18.462 byte, assegnando esattamente il blocco richiesto rimane un buco di 2 byte. L'overhead necessario per tener traccia di questo buco è nettamente

più grande del buco stesso. Il metodo generale per superare questo problema prevede di suddividere la memoria fisica in blocchi di dimensione fissa, che costituiscono le unità d’allocazione. Con questo metodo la memoria assegnata può essere leggermente maggiore della memoria richiesta. La differenza tra questi due numeri è la **frammentazione interna** che consiste nella memoria inutilizzata all’interno di una partizione.

Una soluzione al problema della frammentazione esterna è data dalla **compattazione**. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco. La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è effettuata nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si effettua nella fase d’esecuzione. Se gli indirizzi sono rilocati dinamicamente, la rilocazione richiede solo lo spostamento del programma e dei dati, e quindi la modifica del registro di rilocazione in modo che rifletta il nuovo indirizzo di base. Quando è possibile eseguire la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un’estremità della memoria: tutti i buchi si spostano nell’altra direzione formando un grosso buco di memoria. Questo metodo può essere assai oneroso.

Un’altra possibile soluzione del problema della frammentazione esterna è data dal consentire la non contiguità dello spazio degli indirizzi logici di un processo, permettendo così di assegnare la memoria fisica ai processi dovunque essa sia disponibile. Due tecniche complementari conseguono questo risultato: la segmentazione (Paragrafo 8.4) e la paginazione (Paragrafo 8.5). Queste tecniche si possono anche combinare. La frammentazione è un problema generale che può verificarsi ogni volta che si opera su blocchi di dati. Questa tematica sarà approfondita ulteriormente nei capitoli sulla gestione della memoria secondaria (Capitoli 10 e 12).

8.4 Segmentazione

Come abbiamo già detto, l’immagine che un utente ha della memoria non coincide con la memoria fisica. Lo stesso si può dire sulla rappresentazione della memoria dal punto di vista del programmatore. Lavorare sulla memoria in termini di caratteristiche fisiche è scomodo sia per il sistema operativo sia per il programmatore. Che cosa si otterrebbe se l’hardware potesse fornire un meccanismo di memoria in grado di mappare il punto di vista del programmatore sulla memoria fisica? Il sistema avrebbe più libertà nella gestione della memoria e il programmatore avrebbe a disposizione un ambiente di programmazione più naturale. Un tale meccanismo ci è fornito dalla segmentazione.

8.4.1 Metodo di base

Ci si potrebbe chiedere se il programmatore pensi alla memoria come a un array lineare di byte, alcuni dei quali contengono istruzioni e altri dati. Molti programmatori risponderebbero di no. I programmatori la vedono piuttosto come un insieme di segmenti di dimensione variabile che non hanno fra loro un ordinamento particolare (Figura 8.7).

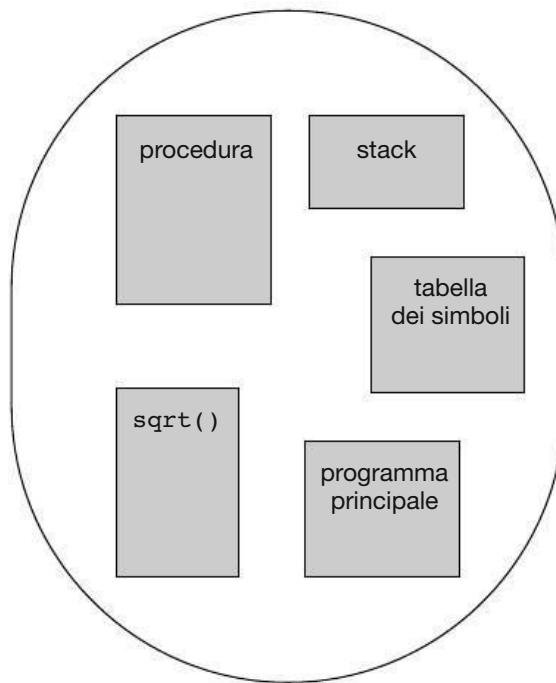


Figura 8.7 Un programma dal punto di vista del programmatore.

La tipica struttura di un programma con cui i programmatori hanno familiarità è costituita di un programma principale e di un gruppo di procedure, funzioni o moduli, insieme con diverse strutture dati come oggetti, array, stack, variabili e così via. Ciascuno di questi elementi si identifica con un nome. Il programmatore parla di “stack”, “libreria matematica”, “programma principale”, indipendentemente dagli indirizzi che questi elementi occupano in memoria. Non è necessario preoccuparsi del fatto che lo stack sia memorizzato prima o dopo la funzione `sqrt()`. Ciascuno di questi segmenti ha una lunghezza variabile, definita intrinsecamente dallo scopo che il segmento stesso ha all’interno del programma. Gli elementi che si trovano all’interno di un segmento sono identificati dal loro offset, misurato dall’inizio del segmento: la prima istruzione del programma, il settimo frame nello stack, la quinta istruzione della funzione `sqrt()`, e così via.

La **segmentazione** è uno schema di gestione della memoria che supporta questa rappresentazione della memoria dal punto di vista del programmatore. Uno spazio d’indirizzi logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indirizzi specificano sia il nome sia l’offset all’interno del segmento, quindi il programmatore fornisce ogni indirizzo come una coppia ordinata di valori: un nome di segmento e un offset. Per semplicità di implementazione, i segmenti sono numerati, e ogni riferimento si compie per mezzo di un numero anziché di un nome; quindi un indirizzo logico è una *coppia*:

<numero di segmento, offset>

Normalmente quando un programma viene compilato il compilatore costruisce automaticamente i segmenti in rapporto al programma sorgente. Un compilatore per il linguaggio C può creare segmenti distinti per i seguenti elementi di un programma:

1. il codice;
2. le variabili globali;
3. lo heap, da cui si alloca la memoria;
4. gli stack usati da ciascun thread;
5. la libreria standard del C.

Alle librerie collegate dal linker al momento della compilazione possono essere assegnati dei segmenti separati. Il loader preleva questi segmenti e assegna loro i numeri di segmento.

8.4.2 Hardware di segmentazione

Sebbene il programmatore possa far riferimento agli oggetti del programma per mezzo di un indirizzo bidimensionale, la memoria fisica è in ogni caso una sequenza di byte unidimensionale. Per questo motivo occorre tradurre gli indirizzi bidimensionali definiti dall'utente negli indirizzi fisici unidimensionali. Questa operazione si compie tramite una **tabella dei segmenti**; ogni suo elemento è una coppia ordinata: la *base del segmento* e il *limite del segmento*. La base del segmento contiene l'indirizzo fisico iniziale della memoria dove il segmento risiede, mentre il limite del segmento contiene la lunghezza del segmento.

L'uso della tabella dei segmenti è illustrato nella Figura 8.8. Un indirizzo logico è formato da due parti: un numero di segmento s e un offset in tale segmento d . Il nu-

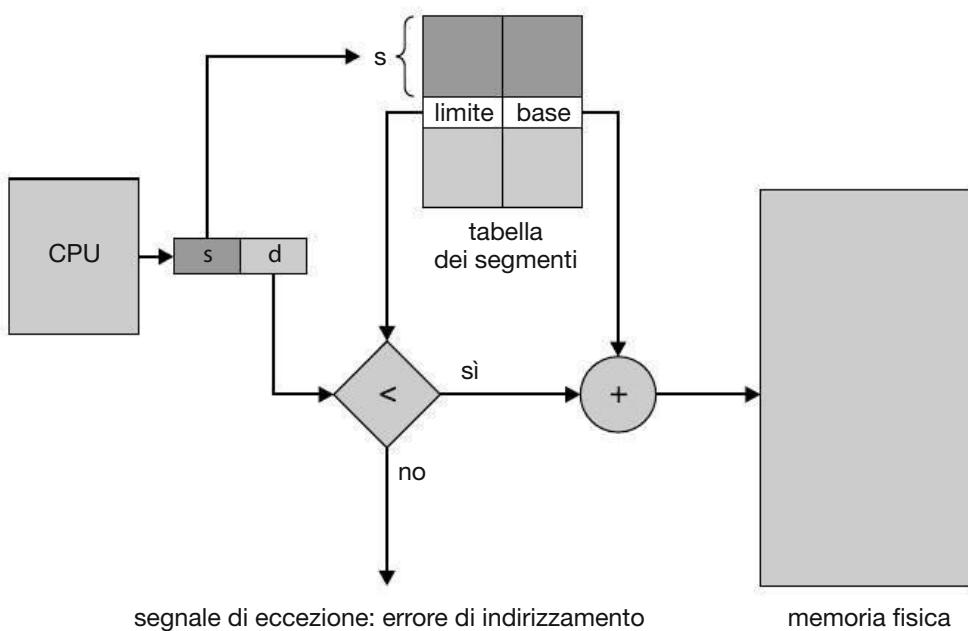


Figura 8.8 Hardware di segmentazione.

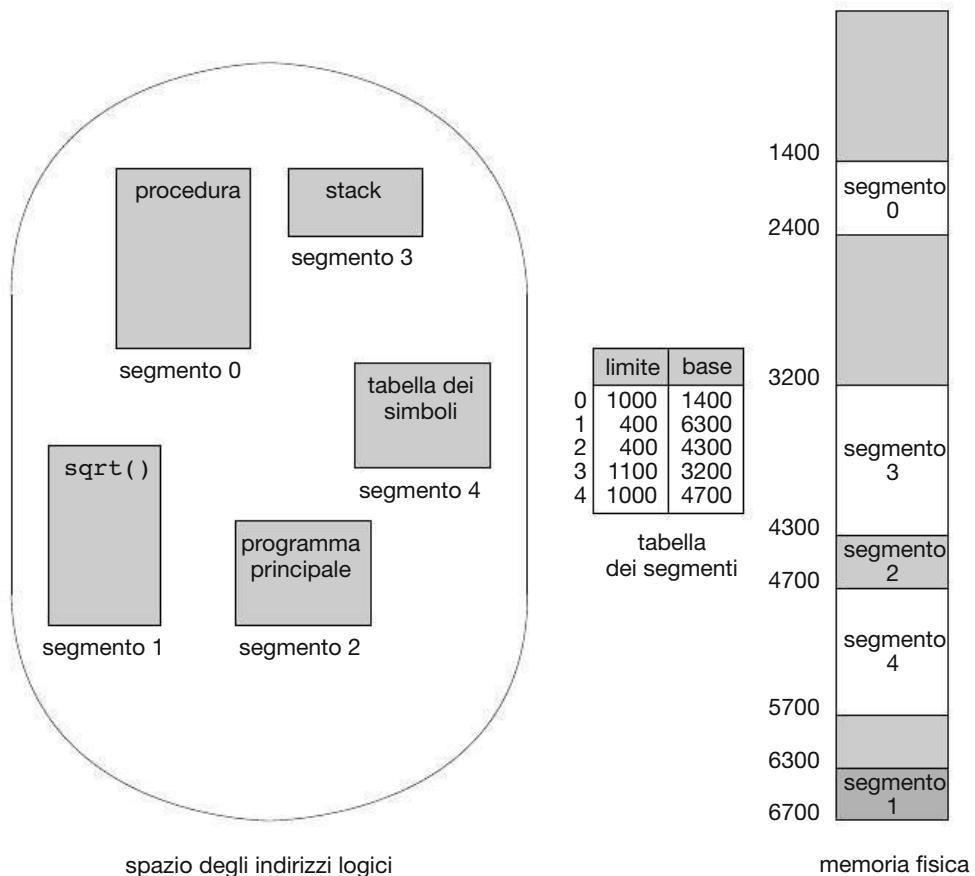


Figura 8.9 Esempio di segmentazione.

numero di segmento si usa come indice per la tabella dei segmenti; l'offset d dell'indirizzo logico deve essere compreso tra 0 e il limite del segmento, altrimenti si genera una eccezione per il sistema operativo (tentativo di indirizzamento logico oltre la fine del segmento). Se la condizione sull'offset è rispettata, questo viene sommato alla base del segmento per produrre l'indirizzo della memoria fisica dove si trova il byte desiderato. Quindi la tabella dei segmenti è fondamentalmente un vettore di coppie di registri di base e limite.

Come esempio si può considerare la situazione illustrata nella Figura 8.9. Sono dati cinque segmenti numerati da 0 a 4, posizionati in memoria fisica come indicato. La tabella dei segmenti ha un elemento distinto per ogni segmento, indicante l'indirizzo iniziale del segmento in memoria fisica (la base) e la lunghezza di quel segmento (il limite). Per esempio, il segmento 2 è lungo 400 byte e inizia alla locazione 4300, quindi un riferimento al byte 53 del segmento 2 si mappa sulla locazione $4300 + 53 = 4353$. Un riferimento al segmento 3, byte 852, si mappa sulla locazione 3200 (la base del segmento 3) + 852 = 4052. Un riferimento al byte 1222 del segmento 0 causa la generazione di una eccezione per il sistema operativo, poiché questo segmento è lungo solo 1000 byte.

8.5 Paginazione

La segmentazione permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. La **paginazione** (*paging*) è un altro schema di gestione della memoria che offre lo stesso vantaggio. Tuttavia, a differenza della segmentazione, la paginazione evita la frammentazione esterna e la necessità di compattazione. Inoltre, la paginazione elimina il gravoso problema della sistemazione di blocchi di memoria di diverse dimensioni in memoria ausiliaria. La maggior parte degli schemi di memory management utilizzati prima della introduzione della paginazione soffrivano di questo problema. Il problema insorge perché, quando alcuni frammenti di codice o dati residenti in memoria centrale devono essere scaricati, si deve trovare lo spazio necessario in memoria ausiliaria. I problemi di frammentazione visti in relazione alla memoria centrale valgono anche per la memoria ausiliaria, con la differenza che in questo caso l'accesso è molto più lento, quindi è impossibile eseguire la compattazione. Grazie ai vantaggi offerti rispetto ai metodi precedenti, la paginazione nelle sue varie forme è comunemente usata nella maggior parte dei sistemi operativi, dai sistemi per mainframe a quelli per smartphone. L'implementazione della paginazione è frutto della cooperazione tra il sistema operativo e l'hardware del computer.

8.5.1 Metodo di base

Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di dimensione fissa, detti **frame**, e nel suddividere la memoria logica in blocchi di pari dimensione, detti **pagine**. Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria o dal file system. La memoria ausiliaria è divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria o di blocchi composti da più frame. Questa idea piuttosto semplice fornisce grandi funzionalità e ha diverse ramificazioni. Per esempio, ora lo spazio degli indirizzi logici è totalmente separato dallo spazio degli indirizzi fisici e dunque un processo può avere uno spazio degli indirizzi logici a 64 bit anche se il sistema ha meno di 2^{64} byte di memoria fisica.

L'hardware di supporto alla paginazione è illustrato nella Figura 8.10; ogni indirizzo generato dalla CPU è diviso in due parti: un **numero di pagina** (p), e un **offset di pagina** (d). Il numero di pagina serve come indice per la **tabella delle pagine**, contenente l'indirizzo di base in memoria fisica di ogni pagina. Questo indirizzo di base si combina con l'offset di pagina per definire l'indirizzo della memoria fisica, che s'invia all'unità di memoria. La Figura 8.11 illustra il modello di paginazione della memoria.

La dimensione di una pagina, così come quella di un frame, è definita dall'hardware ed è, in genere, una potenza di 2 compresa tra 512 byte e 1 GB, a seconda dell'architettura. La scelta di una potenza di 2 come dimensione della pagina facilita notevolmente la traduzione di un indirizzo logico nei corrispondenti numero di pagina e offset di pagina. Se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagina è di 2^n byte, allora gli $m - n$ bit più significativi di un indirizzo

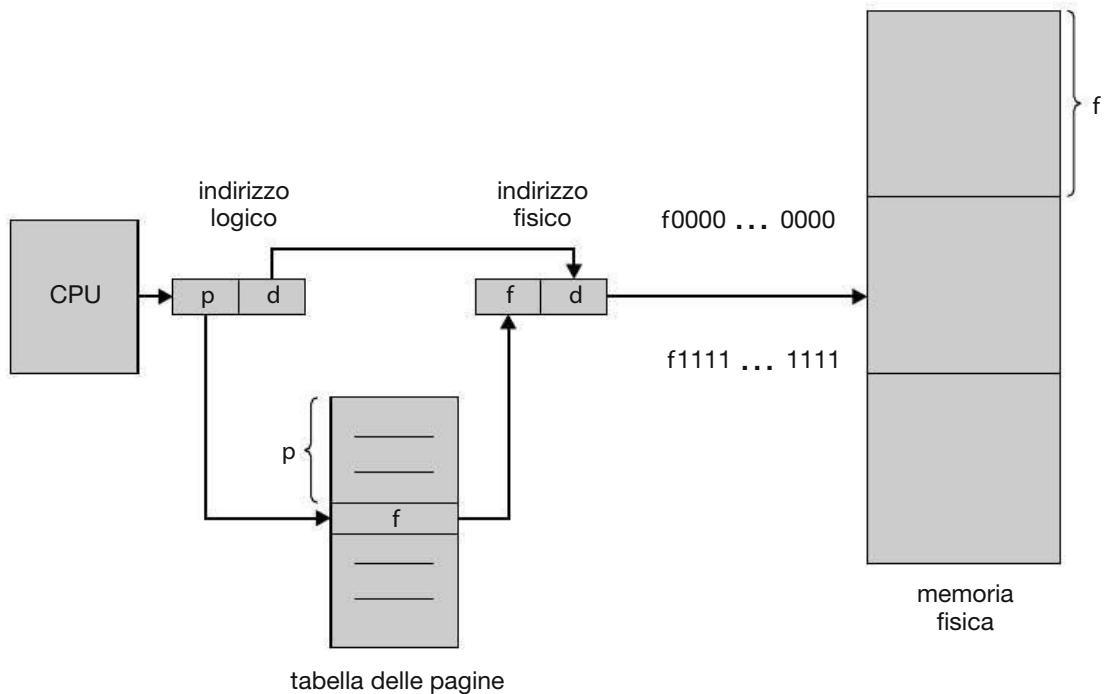


Figura 8.10 Hardware di paginazione.

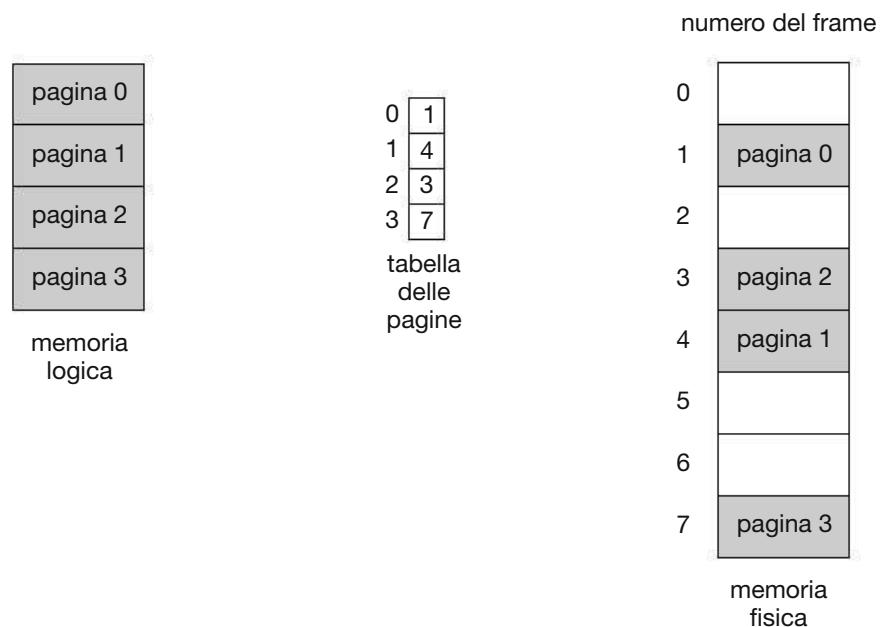


Figura 8.11 Modello di paginazione di memoria logica e memoria fisica.

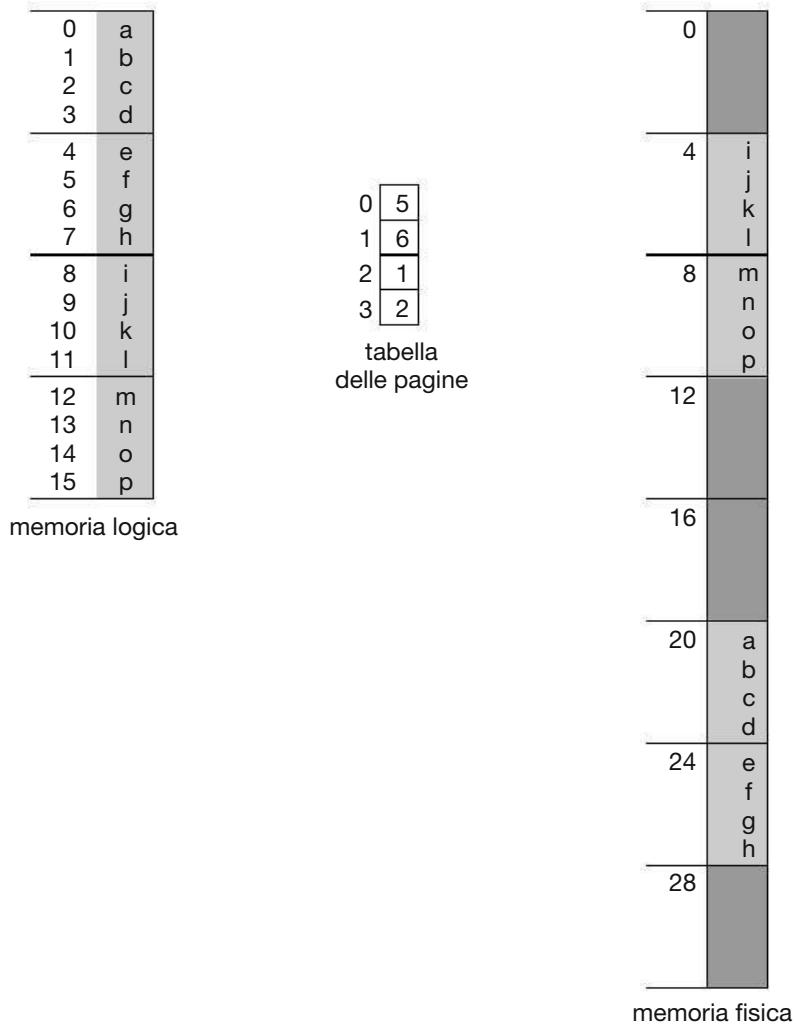


Figura 8.12 Esempio di paginazione per una memoria di 32 byte con pagine di 4 byte.

logico indicano il numero di pagina, e gli n bit meno significativi indicano l'offset di pagina. L'indirizzo logico ha quindi la forma seguente:

numero di pagina	offset di pagina
p	d
$m - n$	n

dove p è un indice della tabella delle pagine e d è l'offset all'interno della pagina indicata da p .

Come esempio concreto, anche se minimo, si consideri la memoria illustrata nella Figura 8.12; qui, nell'indirizzo logico, $n = 2$ e $m = 4$. Utilizzando pagine di 4 byte e una memoria fisica di 32 byte (8 pagine), vediamo come si fa corrispondere la memoria vista dal programmatore alla memoria fisica. L'indirizzo logico 0 è la pagina 0



OTTENERE LA DIMENSIONE DELLA PAGINA SU SISTEMI LINUX

Su un sistema Linux la dimensione della pagina varia a seconda dell'architettura. Vi sono diversi modi per ottenere le dimensioni della pagina. Un approccio è quello di utilizzare la chiamata di sistema `getpagesize()`. In alternativa è possibile eseguire da riga di comando:

```
getconf PAGESIZE
```

Ciascuna di queste tecniche restituisce la dimensione della pagina espressa in byte.

con offset 0. Secondo la tabella delle pagine, la pagina 0 si trova nel frame 5. Quindi all'indirizzo logico 0 corrisponde l'indirizzo fisico $20 [= (5 \times 4) + 0]$. All'indirizzo logico 3 (pagina 0, offset 3) corrisponde l'indirizzo fisico $23 [= (5 \times 4) + 3]$. Per quel che riguarda l'indirizzo logico 4 (pagina 1, offset 0), secondo la tabella delle pagine, alla pagina 1 corrisponde il frame 6, quindi, all'indirizzo logico 4 corrisponde l'indirizzo fisico $24 [= (6 \times 4) + 0]$. All'indirizzo logico 13 corrisponde l'indirizzo fisico 9.

Il lettore può aver notato che la paginazione non è altro che una forma di rilocazione dinamica: a ogni indirizzo logico l'architettura di paginazione fa corrispondere un indirizzo fisico. L'uso della tabella delle pagine è simile all'uso di una tabella di registri base (o di rilocazione), uno per ciascun frame.

Con la paginazione si evita la frammentazione esterna: *qualsiasi* frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia si può avere la frammentazione interna. I frame si assegnano come unità. Poiché in generale lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle pagine, l'*ultimo* frame assegnato può non essere completamente pieno. Se, per esempio, le pagine sono di 2048 byte, un processo di 72.766 byte necessita di 35 pagine più 1086 byte. Si assegnano 36 frame, quindi si ha una frammentazione interna di $2048 - 1086 = 962$ byte. Il caso peggiore si ha con un processo che necessita di n pagine più un byte: si assegnano $n + 1$ frame, quindi si ha una frammentazione interna di quasi un intero frame.

Se la dimensione del processo è indipendente dalla dimensione della pagina, si deve prevedere una frammentazione interna media di mezza pagina per processo. Questa considerazione suggerisce che conviene usare pagine di piccole dimensioni; tuttavia, a ogni elemento della tabella delle pagine è associato un overhead che si riduce all'aumentare delle dimensioni delle pagine. Inoltre, con un maggior numero di dati da trasferire, l'I/O su disco è più efficiente (Capitolo 10). Generalmente, nel tempo la dimensione delle pagine è cresciuta col crescere dei processi, dei dati e della memoria centrale; attualmente la dimensione tipica delle pagine è compresa tra 4 KB e 8 KB; in alcuni sistemi può essere anche maggiore. Alcune CPU e alcuni kernel di sistemi operativi gestiscono anche diverse dimensioni di pagina; il sistema Solaris per esempio usa pagine di 4 KB o 8 MB, secondo il tipo dei dati memorizzati nelle pagine. Sono in fase di studio e progettazione sistemi di paginazione che consentono la variazione dinamica della dimensione delle pagine.

Spesso, su una CPU a 32 bit, ogni voce della tabella delle pagine è lunga 4 byte, ma questa dimensione può variare. Un singola voce di 32 bit può puntare a uno dei 2^{32} frame di pagina fisici. Se la dimensione di un frame è di 4 KB (2^{12}), un sistema con voci della tabella delle pagine di 4 byte può indirizzare 2^{44} byte (o 16 TB) di memoria fisica. Va notato che la dimensione della memoria fisica in un sistema di memoria paginata è differente dalla dimensione logica massima di un processo. Quando analizzeremo ulteriormente la paginazione, introdurremo altre informazioni che devono essere mantenute nelle voci della tabella delle pagine. Queste informazioni riducono il numero di bit disponibili per indirizzare i frame. Un sistema con voci di 32 bit è quindi in grado di indirizzare una quantità di memoria fisica inferiore al valore massimo teoricamente possibile. Una CPU a 32 bit utilizza indirizzi a 32 bit, il che significa che lo spazio di memoria di un processo può essere al massimo di 2^{32} byte (4 GB). La paginazione ci consente quindi di utilizzare una memoria fisica che è decisamente più grande rispetto a quella che potrebbe essere indirizzata dalla lunghezza del puntatore agli indirizzi della CPU.

Quando si deve eseguire un processo, il sistema esamina la sua dimensione espresso in pagine. Poiché ogni pagina del processo necessita di un frame, se il processo richiede n pagine, devono essere disponibili almeno n frame che, se ci sono, vengono assegnate al processo. Si carica la prima pagina in uno dei frame assegnati e s'inserisce il numero del frame nella tabella delle pagine relativa al processo in questione. La pagina successiva si carica in un altro frame e, anche in questo caso, s'inserisce il numero del frame nella tabella delle pagine, e così via (Figura 8.13).

Un aspetto importante della paginazione è la netta distinzione tra la memoria vista dal programmatore e l'effettiva memoria fisica: il programmatore vede la memoria come un unico spazio contiguo, contenente solo il programma stesso; in realtà, il programma utente è sparso in una memoria fisica contenente anche altri programmi. La differenza tra la memoria vista dal programmatore e la memoria fisica è colmata dall'hardware di traduzione degli indirizzi, che fa corrispondere gli indirizzi fisici agli indirizzi logici generati dai processi utenti. Questa corrispondenza non è visibile ai programmatori ed è controllata dal sistema operativo. Si noti che un processo utente, per definizione, non può accedere alle zone di memoria che non gli appartengono. Non ha modo di accedere alla memoria oltre quel che è previsto dalla sua tabella delle pagine, e tale tabella contiene soltanto le pagine che appartengono al processo.

Poiché il sistema operativo gestisce la memoria fisica, deve essere informato dei dettagli della allocazione: quali frame sono assegnati, quali sono disponibili, il loro numero totale, e così via. In genere queste informazioni sono contenute in una struttura dati chiamata **tabella dei frame**, contenente un elemento per ogni frame, indicante se sia libero oppure assegnato e, se è assegnato, a quale pagina di quale processo o di quali processi.

Inoltre, il sistema operativo deve sapere che i processi utenti operano nello spazio utente, e tutti gli indirizzi logici si devono far corrispondere a indirizzi fisici. Se un utente usa una chiamata di sistema (per esempio per eseguire un'operazione di I/O) e fornisce un indirizzo come parametro (per esempio l'indirizzo di un buffer), si deve

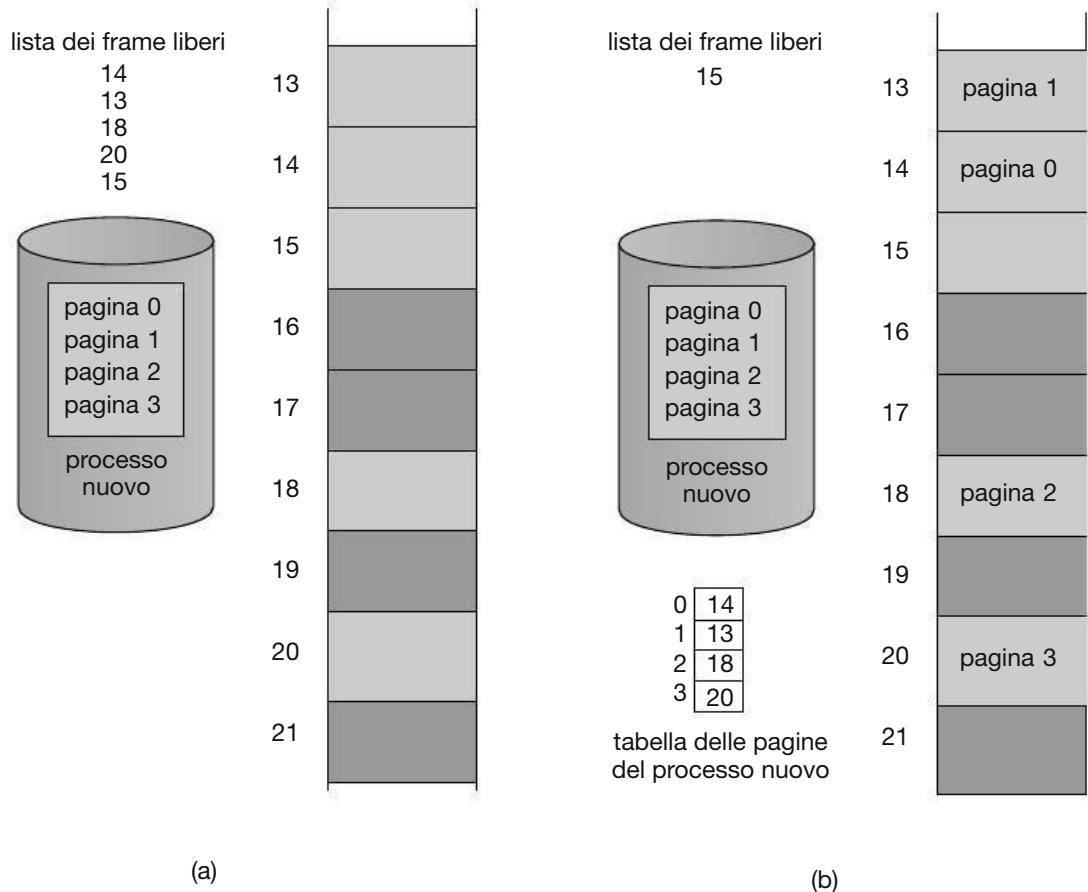


Figura 8.13 Frame liberi; (a) prima e (b) dopo l'allocazione.

tradurre questo indirizzo nell'indirizzo fisico corretto. Il sistema operativo conserva una copia della tabella delle pagine per ciascun processo, così come conserva una copia dei valori contenuti nel contatore di programma e nei registri. Questa copia si usa per tradurre gli indirizzi logici in indirizzi fisici ogni volta che il sistema operativo deve associare esplicitamente un indirizzo fisico a un indirizzo logico. La stessa copia è usata anche dal dispatcher della CPU per impostare l'hardware di paginazione quando a un processo sta per essere assegnata la CPU. La paginazione fa quindi aumentare la durata dei cambi di contesto.

8.5.2 Supporto hardware alla paginazione

Ogni sistema operativo ha il proprio metodo per memorizzare le tabelle delle pagine. Alcuni impiegano una tabella delle pagine per ciascun processo. Il PCB contiene, insieme col valore di altri registri, come il registro delle istruzioni, un puntatore alla tabella delle pagine. Per avviare un processo, il dispatcher ricarica i registri utente e imposta i corretti valori della page table hardware, usando la tabella delle pagine presente in memoria e relativa al processo. Altri sistemi operativi usano un numero li-

mitato di tabelle delle pagine (oppure una sola di queste) in modo da diminuire l'overhead per i cambi di contesto dei processi.

L'implementazione hardware della tabella delle pagine si può realizzare in modi diversi. Nel caso più semplice, si usa uno specifico insieme di **registri**. Per garantire un'efficiente traduzione degli indirizzi di paginazione, questi registri devono essere realizzati con una logica molto veloce. Tale efficienza è determinante, poiché ogni accesso alla memoria passa attraverso il sistema di paginazione. Il dispatcher della CPU ricarica questi registri proprio come ricarica gli altri, e ovviamente le istruzioni di caricamento e modifica dei registri della tabella delle pagine sono privilegiate, quindi soltanto il sistema operativo può modificare la mappa della memoria. Il DEC PDP-11 è un esempio di tale tipo di architettura. Un indirizzo è costituito di 16 bit e la dimensione di una pagina è di 8 KB; la tabella delle pagine consiste quindi di otto elementi che sono mantenuti in registri veloci.

L'uso di registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, nell'ordine, per esempio, di 256 elementi. Però la maggior parte dei calcolatori contemporanei usa tabelle molto grandi, per esempio di un milione di elementi, quindi non si possono impiegare i registri veloci per realizzare la tabella delle pagine; quest'ultima viene invece mantenuta nella memoria principale e un **registro di base della tabella delle pagine** (*page-table base register*, PTBR) punta alla tabella stessa. Il cambio della tabella delle pagine richiede soltanto di modificare questo registro, riducendo considerevolmente il tempo dei cambi di contesto.

Questo metodo presenta un problema connesso al tempo necessario di accesso a una locazione della memoria utente. Per accedere alla locazione i , occorre far riferimento alla tabella delle pagine usando il valore contenuto nel PTBR aumentato dell'offset relativo alla pagina di i , perciò si deve accedere alla memoria. Si ottiene il numero del frame che, associato all'offset rispetto all'inizio della pagina, produce l'indirizzo cercato; a questo punto è possibile accedere alla posizione di memoria desiderata. Con questo metodo, per accedere a un byte occorrono *due* accessi alla memoria (uno per l'elemento della tabella delle pagine e uno per il byte stesso), quindi l'accesso alla memoria è rallentato di un fattore 2. Nella maggior parte dei casi un tale ritardo è intollerabile; sarebbe più conveniente ricorrere allo swapping dei processi!

La soluzione tipica a questo problema consiste nell'impiego di una speciale, piccola cache hardware, detta **TLB** (*translation look-aside buffer*). La TLB è una memoria associativa ad alta velocità in cui ogni elemento consiste di due parti: una chiave (o *tag*) e un valore. Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. La ricerca è molto rapida e in un hardware moderno è parte della pipeline delle istruzioni: non induce dunque nessuna penalizzazione in termini di prestazioni. Per poter eseguire la ricerca in uno stadio della pipeline, tuttavia, la TLB deve essere di dimensioni ridotte, in genere contenute tra le 32 e le 1.024 voci. Alcune CPU implementano TLB separate per istruzioni e dati, in modo da poter raddoppiare il numero di voci TLB disponibili, poiché le due ricerche vengono effettuate in diversi stadi della

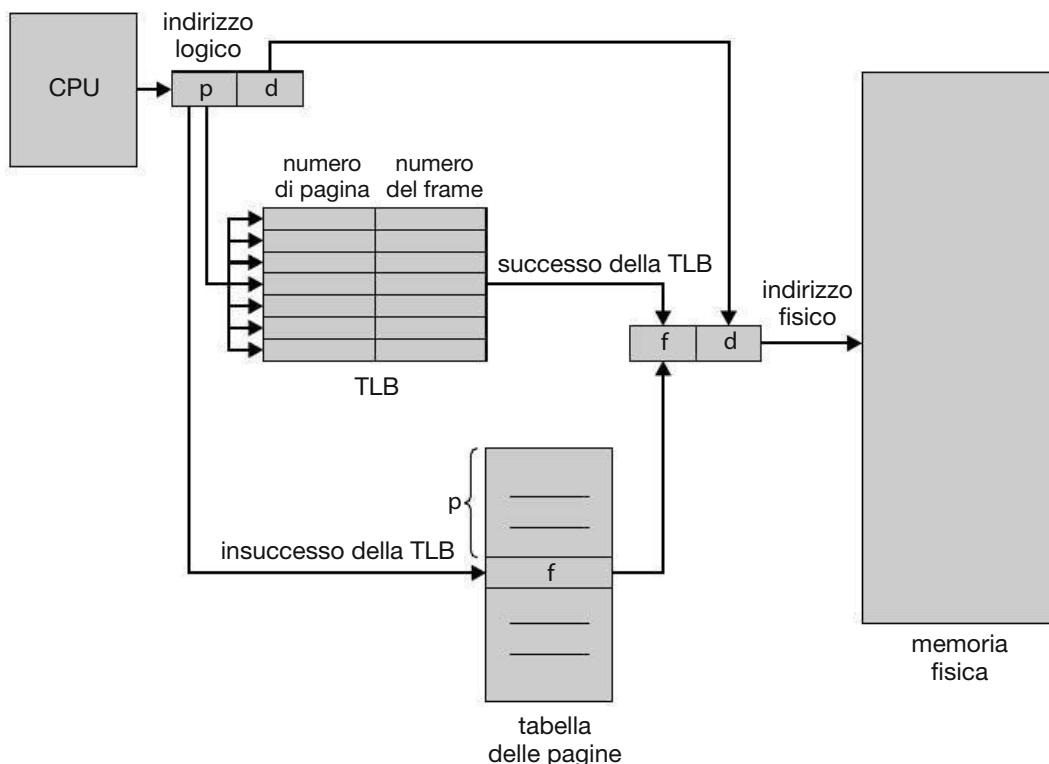


Figura 8.14 Hardware di paginazione con TLB.

pipeline. Questo sviluppo rappresenta un esempio di evoluzione della tecnologia delle CPU: i sistemi si sono evoluti passando dall'assenza di TLB fino ad avere più livelli di TLB, proprio come nel caso delle cache.

La TLB si usa insieme con le tabelle delle pagine nel modo seguente: la TLB contiene una piccola parte degli elementi della tabella delle pagine; quando la CPU genera un indirizzo logico, si presenta il suo numero di pagina alla TLB; se tale numero è presente, il corrispondente numero del frame è immediatamente disponibile e si usa per accedere alla memoria. Come abbiamo appena menzionato, queste operazioni vengono eseguite come parte della pipeline delle istruzioni all'interno della CPU, senza penalizzare il sistema rispetto ad altri sistemi che non implementano la paginazione.

Se nella TLB non è presente il numero di pagina, situazione nota come **insuccesso della TLB** (*TLB miss*), si deve consultare la tabella delle pagine in memoria. A seconda della CPU, questa operazione può essere effettuata automaticamente a livello hardware oppure per mezzo di un interrupt al sistema operativo. Il numero del frame così ottenuto viene usato per accedere alla memoria (Figura 8.14). Inoltre, i numeri della pagina e del frame vengono inseriti nella TLB, e al riferimento successivo la ricerca sarà molto più rapida. Se la TLB è già piena d'elementi, occorre sceglierne uno per sostituirlo. I criteri di sostituzione variano dalla scelta dell'elemento usato meno recentemente (LRU), a una politica round-robin, fino alla scelta casuale. Alcune CPU permettono al sistema operativo di partecipare alla sostituzione LRU di elementi, mentre altre

gestiscono in autonomia questa operazione. Inoltre alcune TLB consentono che certi elementi siano **vincolati** (*wired down*), cioè non si possano rimuovere dalla TLB; in genere si vincolano gli elementi per il codice chiave del kernel.

Alcune TLB memorizzano gli **identificatori dello spazio d'indirizzi** (*address-space identifier*, ASID) in ciascun elemento della TLB. Un ASID identifica in modo univoco ciascun processo e si usa per fornire al processo corrispondente la protezione del suo spazio d'indirizzi. Quando tenta di trovare i valori corrispondenti ai numeri delle pagine virtuali, la TLB si assicura che l'ASID per il processo attualmente in esecuzione corrisponda all'ASID associato alla pagina virtuale. La mancata corrispondenza dell'ASID viene trattata come un TLB miss. Oltre a fornire la protezione dello spazio d'indirizzi, l'ASID consente che la TLB contenga nello stesso istante elementi di diversi processi. Se la TLB non permette l'uso di ASID distinti, ogni volta che si seleziona una nuova tabella delle pagine, per esempio a ogni cambio di contesto, si deve **cancellare** (*flush*) la TLB, in modo da assicurare che il successivo processo in esecuzione non faccia uso di errate informazioni di traduzione. Potrebbero altrimenti esserci vecchi elementi della TLB contenenti indirizzi virtuali validi, ma con indirizzi fisici corrispondenti sbagliati o non validi, lasciati dal precedente processo.

La percentuale di volte che il numero di pagina di interesse si trova nella TLB è detta **tasso di successi** (*hit ratio*). Un tasso di successi dell'80 per cento significa che il numero di pagina desiderato si trova nella TLB nell'80 per cento dei casi. Se sono necessari 100 nanosecondi per accedere alla memoria, allora un accesso alla memoria mappata nella TLB richiede 100 nanosecondi. Se, invece, il numero non è contenuto nella TLB, occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero del frame (100 nanosecondi), quindi accedere al byte desiderato in memoria (100 nanosecondi); in totale sono necessari 200 nanosecondi. Stiamo in questo caso supponendo che una ricerca nella tabella delle pagine richieda un solo accesso alla memoria, ma, come vedremo in seguito, potrebbero talvolta essere necessari più accessi. Per calcolare il **tempo effettivo d'accesso alla memoria** occorre tener conto della probabilità dei due casi:

$$\text{tempo effettivo d'accesso} = 0,80 \times 100 + 0,20 \times 200 = 120 \text{ nanosecondi}$$

In questo esempio si verifica un rallentamento del 20 per cento nel tempo medio d'accesso alla memoria (da 100 a 120 nanosecondi).

Per un tasso di successi del 99 per cento, molto più realistico, si ottiene il seguente risultato:

$$\text{tempo effettivo d'accesso} = 0,99 \times 100 + 0,01 \times 200 = 101 \text{ nanosecondi}$$

Con questo tasso di successi, il rallentamento del tempo d'accesso alla memoria scende all'1 per cento.

Come abbiamo osservato in precedenza, le CPU di oggi possono fornire più livelli di TLB. Il calcolo dei tempi di accesso alla memoria nelle CPU moderne diventa quindi molto più complicato di quanto illustrato nell'esempio precedente. Per esempio, la CPU Intel Core i7 ha una TLB L1 da 128 elementi per le istruzioni e una TLB L1 da 64 elementi per i dati. In caso di insuccesso sulla TLB L1, sono necessari sei cicli di

CPU per cercare la voce sulla TLB L2 da 512 elementi. Un insuccesso sulla TLB L2 significa che la CPU deve attraversare le voci della tabella delle pagine in memoria per trovare l'indirizzo del frame associato, il che può richiedere centinaia di cicli, oppure generare un'interruzione per il sistema operativo affinché esegua lo stesso lavoro.

Un'analisi completa delle prestazioni in un tale sistema richiederebbe informazioni sul tasso di insuccesso di ogni livello di TLB. Da quanto abbiamo detto possiamo tuttavia dedurre che le caratteristiche hardware possono condizionare in maniera significativa le prestazioni della memoria e che le migliorie al sistema operativo (come la paginazione) possono indurre modifiche hardware e, a loro volta, essere da queste influenzate (come nel caso delle TLB). L'impatto del tasso di successi nelle TLB è ulteriormente analizzato nel Capitolo 9.

Le TLB sono una caratteristica hardware e in quanto tali potrebbero sembrare di scarso interesse per i sistemi operativi e i loro progettisti. I progettisti, tuttavia, hanno bisogno di capire la funzione e le caratteristiche delle TLB, che variano a seconda della piattaforma hardware. Per un funzionamento ottimale, il progetto di un sistema operativo per una data piattaforma deve implementare la paginazione basandosi sul progetto delle TLB della piattaforma. Analogamente, un cambiamento nel progetto delle TLB (per esempio, tra generazioni diverse di CPU Intel) può rendere necessaria una modifica nell'implementazione della paginazione dei sistemi operativi che utilizzano questa tecnica.

8.5.3 Protezione

In un ambiente paginato, la protezione della memoria è assicurata dai bit di protezione associati a ogni frame; normalmente tali bit si trovano nella tabella delle pagine.

Un bit può determinare se una pagina si può leggere e scrivere oppure soltanto leggere. Tutti i riferimenti alla memoria passano attraverso la tabella delle pagine per trovare il numero corretto del frame; quindi mentre si calcola l'indirizzo fisico, si possono controllare i bit di protezione per verificare che non si scriva in una pagina di sola lettura. Un tale tentativo causerebbe la generazione di un'eccezione hardware per il sistema operativo, si avrebbe cioè una violazione della protezione della memoria.

Questo metodo si può facilmente estendere per fornire un livello di protezione più perfezionato. Si può progettare un hardware che fornisca la protezione di sola lettura, di sola scrittura o di sola esecuzione. In alternativa, con bit di protezione distinti per ogni tipo d'accesso, si può ottenere una qualsiasi combinazione di tali tipi d'accesso; i tentativi illegali causano la generazione di un'eccezione per il sistema operativo.

Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto **bit di validità**. Tale bit, impostato a *valido*, indica che la pagina corrispondente è nello spazio d'indirizzi logici del processo, quindi è una pagina valida; impostato a *non valido*, indica che la pagina non è nello spazio d'indirizzi logici del processo. Il bit di validità consente quindi di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso un'eccezione. Il sistema operativo concede o impedisce l'accesso a una pagina impostando in modo appropriato tale bit.

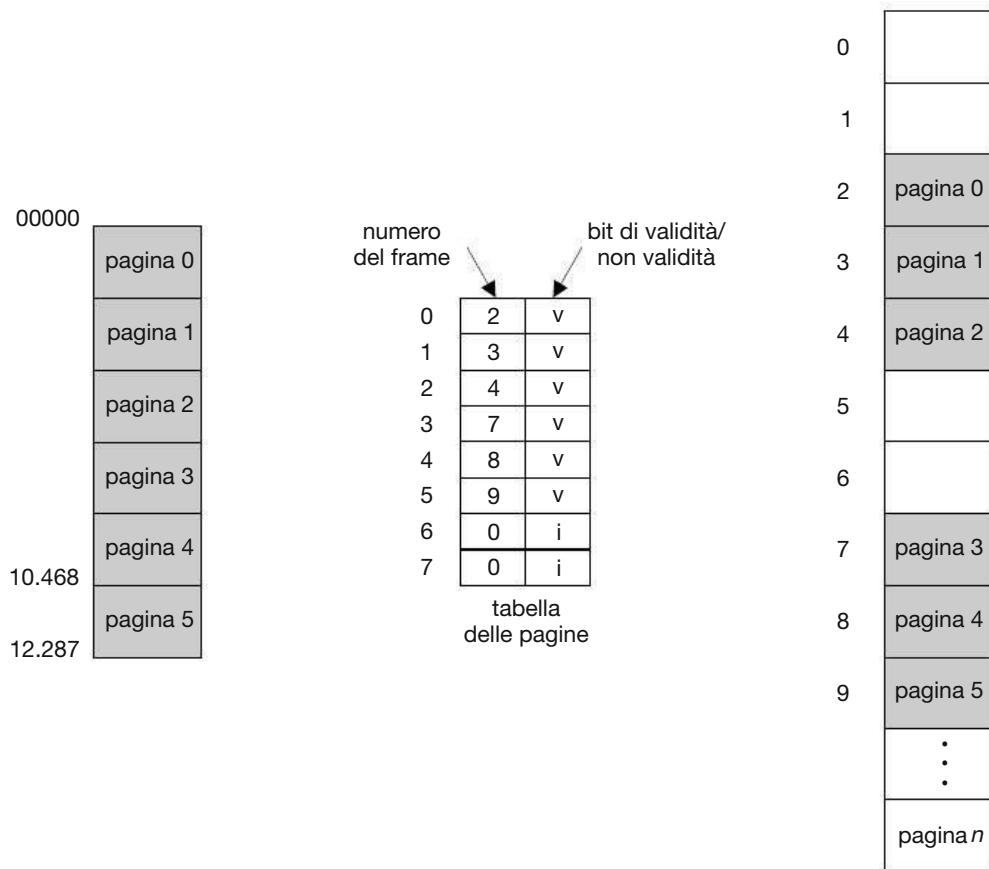


Figura 8.15 Bit di validità (v) o non validità (i) in una tabella delle pagine.

Per esempio, supponiamo che in un sistema con uno spazio di indirizzi di 14 bit (da 0 a 16.383) si abbia un programma che deve usare soltanto gli indirizzi da 0 a 10.468. Con una dimensione delle pagine di 2 KB si ha la situazione mostrata nella Figura 8.15. Gli indirizzi nelle pagine 0, 1, 2, 3, 4 e 5 sono tradotti normalmente tramite la tabella delle pagine. D'altra parte, ogni tentativo di generare un indirizzo nelle pagine 6 o 7 trova non valido il bit di validità; in questo caso il calcolatore invia un'eccezione al sistema operativo (riferimento di pagina non valido).

Questo schema ha creato un problema: poiché il programma si estende solo fino all'indirizzo 10.468, ogni riferimento oltre tale indirizzo è illegale; i riferimenti alla pagina 5 sono tuttavia classificati come validi, e ciò rende validi gli accessi sino all'indirizzo 12.287; solo gli indirizzi da 12.288 a 16.383 sono non validi. Tale problema è dovuto alla dimensione delle pagine di 2 KB e corrisponde alla frammentazione interna della paginazione.

Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, infatti molti processi utilizzano solo una piccola frazione dello spazio d'indirizzi di cui dispongono. In questi casi è un inutile spreco creare una tabella di pagine con elementi per ogni pagina dell'intervallo di indirizzi, poiché una gran parte di questa tabella resta inutilizzata e occupa prezioso spazio di memoria. Alcune architetture di-

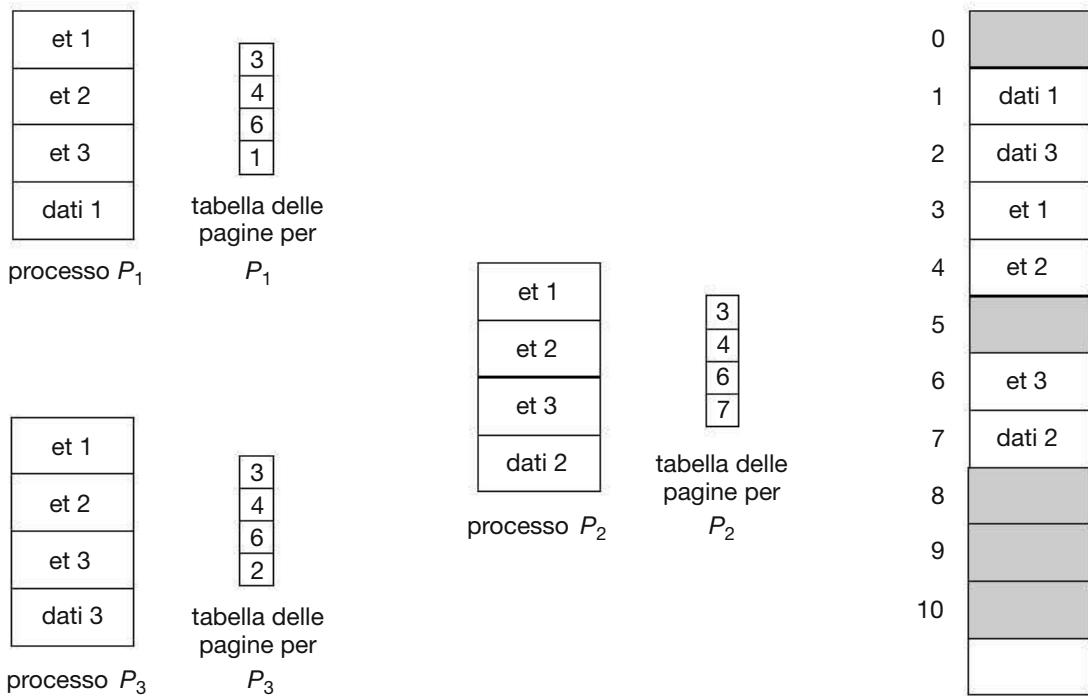


Figura 8.16 Condivisione di codice in un ambiente paginato.

spongono di registri, detti **registri di lunghezza della tabella delle pagine** (*page-table length register*, PTLR), per indicare le dimensioni della tabella. Questo valore si controlla rispetto a ogni indirizzo logico per verificare che quest'ultimo si trovi nell'intervallo valido per il processo. Un errore causa la generazione di un'eccezione per il sistema operativo.

8.5.4 Pagine condivise

Un altro vantaggio della paginazione consiste nella possibilità di *condividere* codice comune. Questa considerazione è importante soprattutto in un ambiente time sharing. Si consideri un sistema con 40 utenti, ciascuno dei quali usa un text editor. Se tale programma è formato da 150 KB di codice e 50 KB di spazio di dati, per gestire i 40 utenti sono necessari 8000 KB. Se però il codice è *rientrante*, può essere condiviso, come mostra la Figura 8.16: un text editor, di tre pagine di 50 KB ciascuna (l'ampia dimensione delle pagine ha lo scopo di rendere più chiara la rappresentazione), condiviso da tre processi, ciascuno dei quali dispone della propria pagina di dati.

Il **codice rientrante**, detto anche **codice puro**, è un codice non automodificante: non cambia durante l'esecuzione. Quindi, due o più processi possono eseguire lo stesso codice nello stesso momento. Ciascun processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari alla propria esecuzione. I dati per due differenti processi sono, ovviamente, diversi per ciascun processo.

In memoria fisica è presente una sola copia dell'editor: la tabella delle pagine di ogni utente fa corrispondere gli stessi frame contenenti l'elaboratore di testi, mentre

le pagine dei dati si fanno corrispondere a frame diversi. Quindi per gestire 40 utenti sono sufficienti una copia dell’elaboratore di testi (150 KB) e 40 copie dei 50 KB di spazio di dati per ciascun utente, per un totale di 2150 KB, invece di 8000 KB; il risparmio è notevole.

Si possono condividere anche altri programmi d’uso frequente: compilatori, interfacce a finestre, librerie a linking dinamico, sistemi di basi di dati e così via. Per essere condivisibile, il codice deve essere rientrante. La natura di sola lettura del codice condiviso non si può affidare alla sola correttezza intrinseca del codice stesso, ma deve essere fatta rispettare dal sistema operativo. La condivisione della memoria tra processi di un sistema è simile al modo in cui i thread condividono lo spazio d’indirizzi di un task (Capitolo 4). Inoltre con riferimento al Capitolo 3, dove si descrive la memoria condivisa come un metodo di comunicazione tra processi, alcuni sistemi operativi realizzano la memoria condivisa impiegando le pagine condivise.

Oltre a permettere che più processi condividano le stesse pagine fisiche, l’organizzazione della memoria in pagine offre numerosi altri vantaggi; ne vedremo alcuni nel Capitolo 9.

8.6 Struttura della tabella delle pagine

In questo paragrafo si descrivono alcune tra le tecniche più comuni per strutturare la tabella delle pagine: la paginazione gerarchica, la tabella delle pagine di tipo hash e la tabella delle pagine invertita.

8.6.1 Paginazione gerarchica

La maggior parte dei moderni calcolatori dispone di uno spazio d’indirizzi logici molto grande (da 2^{32} a 2^{64} elementi). In un ambiente di questo tipo la stessa tabella delle pagine diventa eccessivamente grande. Si consideri, per esempio, un sistema con uno spazio d’indirizzi logici a 32 bit. Se la dimensione di ciascuna pagina è di 4 KB (2^{12}), la tabella delle pagine potrebbe arrivare a contenere fino a 1 milione di elementi ($2^{32}/2^{12}$). Se ogni elemento consiste di 4 byte, ciascun processo potrebbe richiedere fino a 4 MB di spazio fisico d’indirizzi solo per la tabella delle pagine. Chiaramente, sarebbe meglio evitare di collocare la tabella delle pagine in modo contiguo in memoria centrale. Una semplice soluzione a questo problema consiste nel suddividere la tabella delle pagine in parti più piccole; questo risultato si può ottenere in molti modi.

Un metodo consiste nell’adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata (Figura 8.17). Si consideri il precedente esempio di macchina a 32 bit con dimensione delle pagine di 4 KB. Ciascun indirizzo logico è suddiviso in un numero di pagina di 20 bit e in un offset di pagina di 12 bit. Paginando la tabella delle pagine, anche il numero di pagina è a sua volta suddiviso in un numero

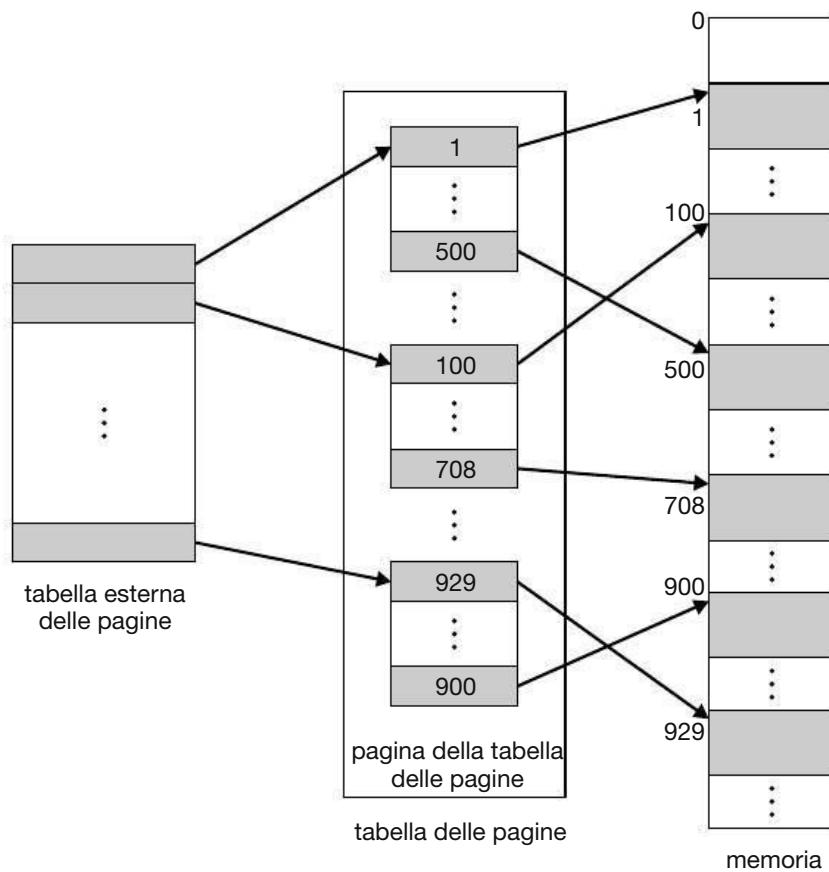


Figura 8.17 Schema di una tabella delle pagine a due livelli.

di pagina di 10 bit e un offset di pagina di 10 bit. Quindi, l’indirizzo logico è strutturato come segue:

numero di pagina		offset di pagina
p_1	p_2	d
10	10	12

dove p_1 è un indice della tabella delle pagine di primo livello, o tabella esterna delle pagine, e p_2 è l’offset all’interno della pagina indicata dalla tabella esterna delle pagine. Il metodo di traduzione degli indirizzi seguito da questa architettura è mostrato nella Figura 8.18. Poiché la traduzione degli indirizzi si svolge dalla tabella esterna delle pagine verso l’interno, questo metodo è anche noto come **tabella delle pagine ad associazione diretta** (*forward-mapped page table*).

Prendiamo in considerazione la gestione della memoria di uno dei sistemi classici, il minicomputer VAX della Digital Equipment Corporation (DEC). Il VAX era il mini-computer più popolare del suo tempo ed è stato venduto dal 1977 al 2000.

L’architettura VAX utilizza una variante della paginazione a due livelli. Il VAX è una macchina a 32 bit con pagine di 512 byte. Lo spazio d’indirizzi logici di un pro-

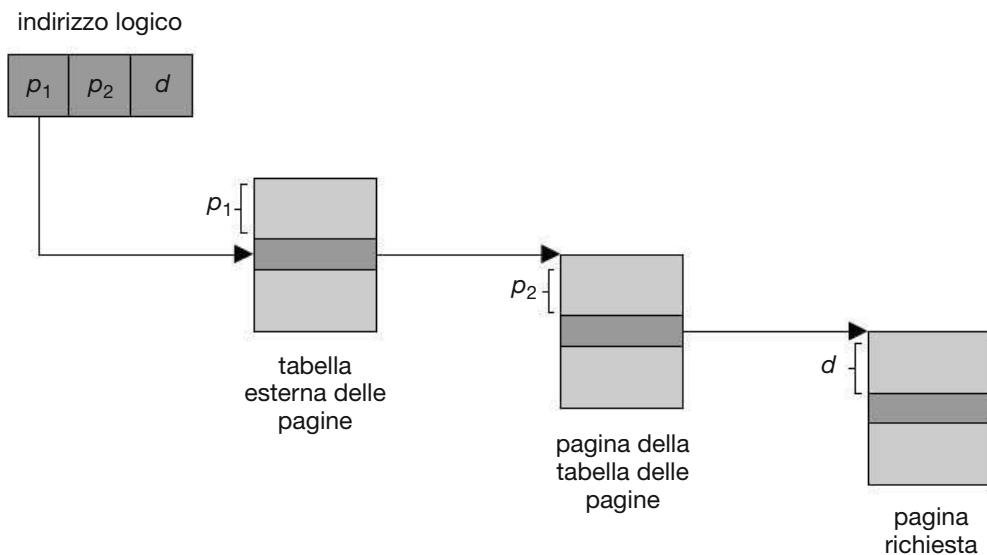


Figura 8.18 Traduzione degli indirizzi per un’architettura a 32 bit con paginazione a due livelli.

cesso è suddiviso in quattro sezioni uguali, ciascuna di 2^{30} byte; ogni sezione rappresenta una parte differente dello spazio d’indirizzi logici di un processo. I 2 bit più significativi dell’indirizzo logico identificano la sezione appropriata, i successivi 21 bit rappresentano il numero logico di pagina all’interno di tale sezione e gli ultimi 9 bit l’offset nella pagina richiesta. Suddividendo in questo modo la tabella delle pagine, il sistema operativo può lasciare inutilizzate le diverse partizioni fino al momento in cui un processo ne fa richiesta. Intere sezioni di spazio di indirizzi virtuali sono spesso inutilizzate. Le tabelle delle pagine multilivello non contengono voci relative a questi spazi, permettendo così di ridurre notevolmente la quantità di memoria necessaria per memorizzare le strutture dati della memoria virtuale.

Nell’architettura VAX un indirizzo ha quindi la forma seguente:

sezione	pagina	offset
s	p	d
2	21	9

dove s denota il numero della sezione, p è un indice per la tabella delle pagine e d è l’offset all’interno della pagina. Anche quando si usa questo schema, la dimensione di una tabella delle pagine a un livello per un processo in un sistema VAX che usa una sezione è ancora 2^{21} bit \times 4 byte per elemento = 8 MB. Per ridurre ulteriormente l’uso della memoria centrale, il VAX pagina le tabelle delle pagine dei processi utenti.

Lo schema di paginazione a due livelli non è più adatto nel caso di sistemi con uno spazio di indirizzi logici a 64 bit. Per spiegare questo aspetto, si supponga che la dimensione delle pagine di questo sistema sia di 4 KB (2^{12}). In questo caso, la tabella delle pagine conterrà fino a 2^{52} elementi. Adottando uno schema di paginazione a due

livelli, le tabelle interne delle pagine possono occupare convenientemente una pagina, o contenere 2^{10} elementi di 4 byte. Gli indirizzi si presentano come segue:

pagina esterna	pagina interna	offset
p_1	p_2	d
42	10	12

La tabella esterna delle pagine consiste di 2^{42} elementi, o 2⁴⁴ byte. La soluzione ovvia per evitare una tabella tanto grande consiste nel suddividere la tabella in parti più piccole. Questo metodo si adotta anche in alcuni processori a 32 bit allo scopo di fornire una maggiore flessibilità ed efficienza.

La tabella esterna delle pagine si può suddividere in vari modi. Per esempio, si può paginare la tabella esterna delle pagine, ottenendo uno schema di paginazione a tre livelli. Si supponga che la tabella esterna delle pagine sia costituita di pagine di dimensione ordinaria (2^{10} elementi, o 2¹² byte); uno spazio d'indirizzi a 64 bit è ancora enorme:

seconda pagina esterna	pagina esterna	pagina interna	offset
p_1	p_2	p_3	d
32	10	10	12

La tabella esterna delle pagine è ancora di 2³⁴ byte (16 GB).

Il passo successivo sarebbe uno schema di paginazione a quattro livelli, in cui si pagina anche la tabella esterna di secondo livello delle pagine, e così via. L'UltraSPARC a 64 bit richiederebbe sette livelli di paginazione – con un numero proibitivo di accessi alla memoria – per tradurre ciascun indirizzo logico. Da questo esempio è possibile capire perché, per le architetture a 64 bit, le page table gerarchiche sono in genere considerate inappropriate.

8.6.2 Tabella delle pagine di tipo hash

Un metodo di gestione molto comune degli spazi d'indirizzi oltre i 32 bit consiste nell'impiego di una **tabella delle pagine di tipo hash**, in cui l'argomento della funzione hash è il numero della pagina virtuale. Per la gestione delle collisioni, ogni elemento della tabella hash contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa locazione. Ciascun elemento è composto da tre campi: (1) il numero della pagina virtuale; (2) l'indirizzo del frame corrispondente alla pagina virtuale; (3) un puntatore al successivo elemento della lista.

L'algoritmo opera come segue: si applica la funzione hash al numero della pagina virtuale contenuto nell'indirizzo virtuale, identificando un elemento della tabella. Si confronta il numero di pagina virtuale con il campo (1) del primo elemento della lista concatenata corrispondente. Se i valori coincidono, si usa l'indirizzo del relativo fra-

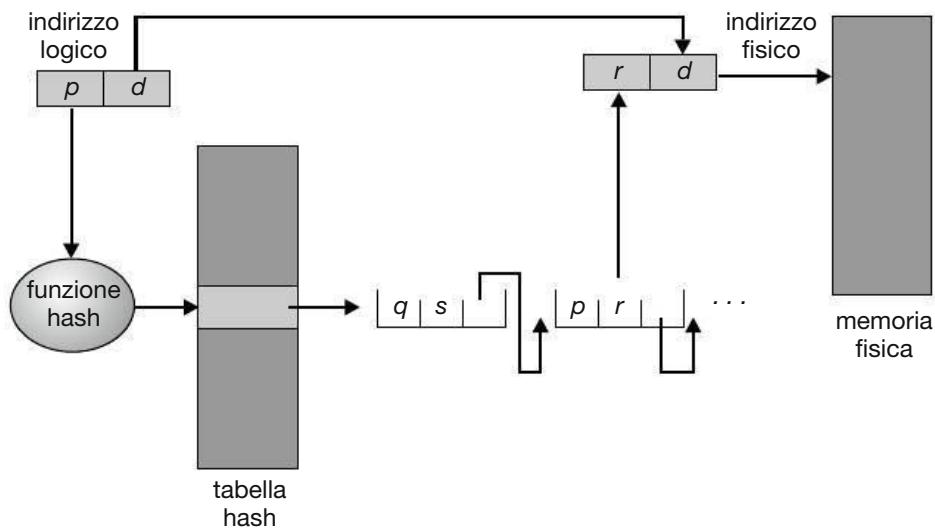


Figura 8.19 Tabella delle pagine di tipo hash.

me (campo 2) per generare l'indirizzo fisico desiderato. Altrimenti, l'algoritmo esamina allo stesso modo gli elementi successivi della lista concatenata (Figura 8.19).

Per questo schema è stata proposta una variante, adatta a spazi di indirizzamento a 64 bit. Si tratta della **tabella delle pagine a gruppi** (*clustered page table*), simile alla tabella delle pagine di tipo hash; la differenza è che ciascun elemento della tabella hash contiene i riferimenti alle pagine fisiche corrispondenti a un gruppo di pagine virtuali contigue (per esempio 16). Le tabelle delle pagine a gruppi sono particolarmente utili per gli spazi d'indirizzi **sparsi**, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio d'indirizzi.

8.6.3 Tabella delle pagine invertita

Generalmente, si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, ossia vi sono elementi corrispondenti ad ogni indirizzo virtuale a prescindere dalla validità di quest'ultimo. Questa rappresentazione tabellare è naturale, poiché i processi fanno riferimento alle pagine tramite gli indirizzi virtuali delle pagine stesse, che il sistema operativo deve poi tradurre in indirizzi di memoria fisica. Poiché la tabella è ordinata per indirizzi virtuali, il sistema operativo può calcolare in che punto della tabella si trova l'indirizzo fisico associato, e usare direttamente tale valore. Uno degli inconvenienti insiti in questo metodo è che ciascuna tabella delle pagine può contenere milioni di elementi e occupare grandi quantità di memoria fisica, necessaria solo per sapere com'è impiegata la rimanente memoria fisica.

Per risolvere questo problema si può fare uso della **tabella delle pagine invertita**. Una tabella delle pagine invertita ha un elemento per ogni pagina reale (o frame). Ciascun elemento è quindi costituito dell'indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale

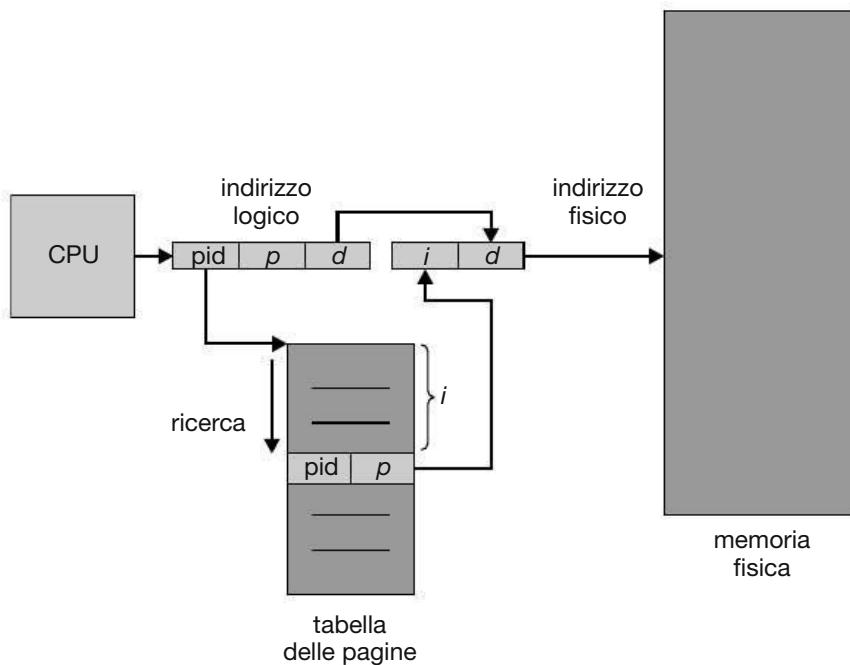


Figura 8.20 Tabella delle pagine invertita.

pagina. Quindi, nel sistema esiste una sola tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica. Nella Figura 8.20 sono mostrate le operazioni di una tabella delle pagine invertita; si confronti questa figura con la Figura 8.10, che illustra il modo di operare per una tabella delle pagine ordinaria. Le tabelle invertite richiedono spesso la memorizzazione di un identificatore dello spazio d'indirizzi (Paragrafo 8.5.2) in ciascun elemento della tabella delle pagine, perché essa contiene di solito molti spazi d'indirizzi diversi associati alla memoria fisica; l'identificatore garantisce che una data pagina logica relativa a un certo processo sia associata alla pagina fisica corrispondente. Esempi di sistemi che usano le tabelle delle pagine invertite sono l'UltraSPARC a 64 bit e il PowerPC.

Per illustrare questo metodo descriviamo una versione semplificata della tabella delle pagine invertita dell'IBM RT. IBM è stata la prima grande azienda a utilizzare le tabelle delle pagine invertite, a cominciare dal System 38, passando per il sistema RS/6000, fino alle attuali CPU IBM Power. Nell'IBM RT ciascun indirizzo virtuale è una tripla del tipo seguente:

$\langle id\text{-processo}, \text{numero di pagina}, offset \rangle$

Ogni elemento della tabella delle pagine invertita è una coppia $\langle id\text{-processo}, \text{numero di pagina} \rangle$ dove l' $id\text{-processo}$ assume il ruolo di identificatore dello spazio d'indirizzi. Quando si fa un riferimento alla memoria, si presenta una parte dell'indirizzo virtuale, formato da $\langle id\text{-processo}, \text{numero di pagina} \rangle$, al sottosistema di memoria. Quindi si cerca una corrispondenza nella tabella delle pagine invertita. Se si trova tale corrispondenza, per esempio sull'elemento i , si genera l'indirizzo fisico $\langle i, offset \rangle$. In caso contrario è stato tentato un accesso a un indirizzo illegale.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella quando si fa riferimento a una pagina. Poiché la tabella delle pagine invertita è ordinata per indirizzi fisici, mentre le ricerche si fanno per indirizzi virtuali, per trovare una corrispondenza potrebbe essere necessario esaminare tutta la tabella; questa ricerca richiederebbe troppo tempo. Per limitare l'entità del problema si può impiegare una tabella hash (come si descrive nel Paragrafo 8.6.2), che riduce la ricerca a un solo, o a pochi, elementi della tabella delle pagine. Naturalmente, ogni accesso alla tabella hash aggiunge al procedimento un riferimento alla memoria, quindi un riferimento alla memoria virtuale richiede almeno due letture della memoria reale: una per l'elemento della tabella hash e l'altro per la tabella delle pagine. Ricordiamo che la ricerca si effettua prima nella TLB, quindi si consulta la tabella hash, il che migliora le prestazioni.

Nei sistemi che adottano le tabelle delle pagine invertite, l'implementazione della memoria condivisa è difficoltosa. Difatti, la condivisione si realizza solitamente tramite indirizzi virtuali multipli (uno per ogni processo che partecipa alla condivisione) associati a un unico indirizzo fisico. Questo metodo però non può essere adottato in presenza di tabelle invertite, perché, essendovi un solo elemento indicante la pagina virtuale corrispondente a ogni pagina fisica, questa non può avere più di un indirizzo virtuale associato. Una semplice tecnica per superare il problema consiste nel porre nella tabella delle pagine una sola associazione fra un indirizzo virtuale e l'indirizzo fisico condiviso; ciò comporta un errore dovuto all'assenza della pagina (*page fault*) per ogni riferimento agli indirizzi virtuali non associati.

8.6.4 Oracle SPARC Solaris

Consideriamo come ultimo esempio una moderna CPU a 64 bit e un sistema operativo strettamente integrati tra loro per fornire una memoria virtuale a basso overhead. Il sistema Solaris in esecuzione sulla CPU SPARC è un sistema operativo completamente a 64 bit e come tale deve risolvere il problema della memoria virtuale senza esaurire tutta la sua memoria fisica, mantenendo livelli multipli di tabelle delle pagine. L'approccio di Solaris è piuttosto complesso, ma risolve il problema in modo efficiente utilizzando tabelle delle pagine di tipo hash. Vi sono due tabelle hash, una per il kernel e una per tutti i processi utente. Ogni tabella mappa indirizzi di memoria virtuale nella memoria fisica. Ogni elemento della tabella hash rappresenta un'area contigua di memoria virtuale mappata, il che è più efficiente rispetto ad avere voci separate per ciascuna pagina. Ogni voce ha un indirizzo di base e un intervallo (*span*) che indica il numero di pagine rappresentate da quella voce.

La traduzione da virtuale a fisico impiegherebbe troppo tempo se ogni indirizzo richiedesse la ricerca attraverso una tabella hash, quindi la CPU implementa una TLB che contiene le voci della tabella di traduzione (TTE), in modo da offrire ricerche hardware rapide. Una cache di queste TTE risiede in un buffer di memoria di traduzione (TSB), che include una voce per ogni pagina di recente accesso. In caso di riferimento a un indirizzo virtuale, l'hardware interroga la TLB per una traduzione. Se non vengono trovate traduzioni, l'hardware scorre il buffer TSB in cerca della TTE che

corrisponde all'indirizzo virtuale che ha causato la ricerca. Questa funzionalità, chiamata **TLB walk**, è presente su molte CPU moderne. Se viene trovata una corrispondenza nel TSB, la CPU copia la voce TSB nella TLB, e la traduzione viene completata. Se invece non viene trovata alcuna corrispondenza, viene interrotto il kernel per effettuare una ricerca nella tabella hash. Il kernel crea quindi una TTE dalla tabella hash appropriata e la memorizza nel TSB affinché l'unità di gestione della memoria della CPU possa effettuare il caricamento automatico nella TLB. Infine, il gestore di interrupt restituisce il controllo alla MMU, che completa la conversione dell'indirizzo e recupera il byte o la parola richiesta dalla memoria principale.

8.7 Esempio: le architetture Intel a 32 e 64 bit

L'architettura Intel ha dominato il mondo dei personal computer per diversi anni. Il processore Intel 8086, a 16 bit, apparve alla fine degli anni 70 e fu presto seguito da un altro chip a 16 bit, l'Intel 8088, noto per essere stato il chip utilizzato nel PC IBM originale. Sia il chip 8086 che il chip 8088 erano basati su una architettura segmentata. Più tardi Intel iniziò la produzione di una serie di chip a 32 bit, IA-32, che includeva la famiglia di processori Pentium. L'architettura IA-32 supportava paginazione e segmentazione. Più di recente Intel ha prodotto una serie di chip a 64 bit basati sull'architettura x86-64. Attualmente tutti i più popolari sistemi operativi per PC, tra cui Windows, Mac OS X e Linux (anche se Linux, ovviamente, gira su diverse altre architetture), vengono eseguiti su chip Intel. Va notato tuttavia che la posizione dominante di Intel non si è propagata ai sistemi mobili, su cui attualmente gode di notevole successo l'architettura ARM (si veda il Paragrafo 8.8).

In questo paragrafo esaminiamo la traduzione degli indirizzi nelle architetture IA-32 e x86-64. Prima di procedere è importante osservare che, poiché Intel ha rilasciato diverse versioni e diverse varianti delle sue architetture nel corso degli anni, non possiamo fornire una descrizione completa della struttura di gestione della memoria per tutti i suoi chip. Non è nemmeno nostra intenzione fornire tutti i dettagli della CPU, perché questi argomenti vengono trattati nei libri di architettura degli elaboratori. Quello che faremo, piuttosto, è di presentare i principali concetti della gestione della memoria di queste CPU Intel.

8.7.1 Architettura IA-32

La gestione della memoria nei sistemi IA-32 è suddivisa in due componenti: segmentazione e paginazione. La CPU genera indirizzi logici che vengono passati all'unità di segmentazione. L'unità di segmentazione produce un indirizzo lineare per ogni indirizzo logico. L'indirizzo lineare viene quindi passato all'unità di paginazione, che genera l'indirizzo fisico nella memoria principale. Dunque, l'unità di segmentazione e l'unità di paginazione formano insieme l'equivalente dell'unità di gestione della memoria (MMU). Questo schema è mostrato nella Figura 8.21.



Figura 8.21 Traduzione degli indirizzi logici in indirizzi fisici in IA-32.

8.7.1.1 Segmentazione in IA-32

Nell’architettura IA-32 un segmento può raggiungere la dimensione massima di 4 GB; il numero massimo di segmenti per processo è pari a 16 K. Lo spazio degli indirizzi logici di un processo è composto da due partizioni: la prima contiene fino a 8 K segmenti riservati al processo; la seconda contiene fino a 8 K segmenti condivisi fra tutti i processi. Le informazioni riguardanti la prima partizione sono contenute nella **tabella locale dei descrittori** (*local descriptor table*, LDT), quelle relative alla seconda partizione sono memorizzate nella **tabella globale dei descrittori** (*global descriptor table*, GDT). Ciascun elemento nella LDT e nella GDT è lungo 8 byte e contiene informazioni dettagliate riguardanti uno specifico segmento, oltre agli indirizzi base e limite.

Un indirizzo logico è una coppia (*selettore, offset*), dove il selettore è un numero di 16 bit:

s	g	p
13	1	2

in cui s indica il numero del segmento, g indica se il segmento si trova nella GDT o nella LDT e p contiene informazioni relative alla protezione. L’*offset* è un numero di 32 bit che indica la posizione del byte (o della parola) all’interno del segmento in questione.

La macchina ha sei registri di segmento che consentono a un processo di far riferimento contemporaneamente a sei segmenti; inoltre possiede sei registri di microprogramma di 8 byte per i corrispondenti descrittori prelevati dalla LDT o dalla GDT. Questa cache evita alla macchina di dover prelevare dalla memoria i descrittori per ogni riferimento alla memoria.

Un indirizzo lineare di IA-32 è lungo 32 bit e si genera come segue. Il registro di segmento punta all’elemento appropriato all’interno della LDT o della GDT; le informazioni relative alla base e al limite di tale segmento si usano per generare un **indirizzo lineare**. Innanzitutto si usa il valore del limite per controllare la validità dell’indirizzo; se non è valido, si ha un errore di riferimento alla memoria che causa la generazione di un’eccezione e la restituzione del controllo al sistema operativo; altrimenti, si somma il valore dell’*offset* al valore della base, ottenendo un indirizzo lineare di 32 bit. La Figura 8.22 illustra tale processo. Nel paragrafo successivo si considera come l’unità di paginazione trasforma questo indirizzo lineare in un indirizzo fisico.

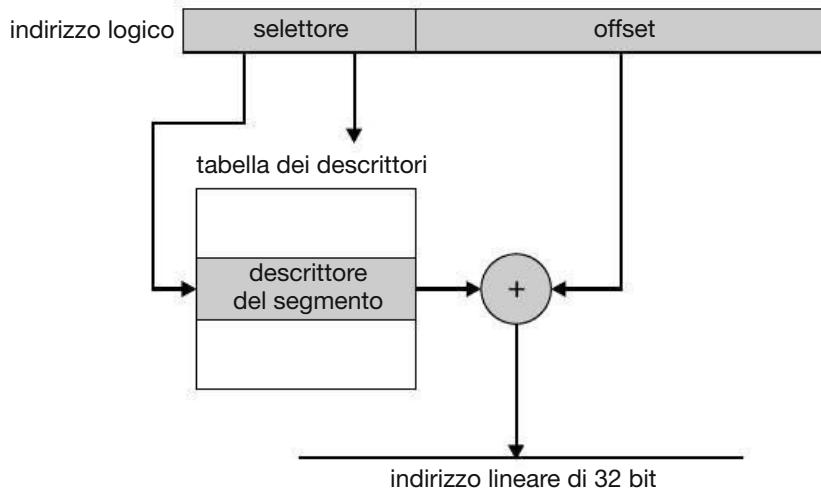


Figura 8.22 Segmentazione in IA-32.

8.7.1.2 Paginazione in IA-32

L'architettura IA-32 prevede che le pagine abbiano una misura di 4 KB oppure di 4 MB. Per le pagine di 4 KB, in IA-32 vige uno schema di paginazione a due livelli che prevede la seguente scomposizione degli indirizzi lineari a 32 bit:

numero di pagina	offset di pagina
p_1	p_2
10	10 12

Lo schema di traduzione degli indirizzi per questa architettura, simile a quello rappresentato nella Figura 8.18, è mostrato in dettaglio nella Figura 8.23. I dieci bit più significativi puntano a un elemento nella tabella delle pagine più esterna, detta in IA-32 **directory delle pagine**. (Il registro CR3 punta alla directory delle pagine del processo corrente.) Gli elementi della directory delle pagine puntano a una tabella delle pagine interne, indicizzata da dieci bit intermedi dell'indirizzo lineare. Infine, i bit meno significativi in posizione 0-11 contengono l'offset da applicare all'interno della pagina di 4 KB cui si fa riferimento nella tabella delle pagine.

Un elemento appartenente alla directory delle pagine è il flag Page Size; se impostato, indica che il frame non ha la dimensione standard di 4 MB, ma misura invece 4 KB. In questo caso, la directory di pagina punta direttamente al frame di 4 MB, scavalcando la tabella delle pagine interne; i 22 bit meno significativi nell'indirizzo lineare indicano l'offset nella pagina di 4 MB.

Per migliorare l'efficienza d'uso della memoria fisica, le tabelle delle pagine in IA-32 possono essere trasferite sul disco. In questo caso, si ricorre a un bit *invalid* in ciascun elemento della directory di pagina, per indicare se la tabella a cui l'elemento punta sia in memoria o sul disco. Se è su disco, il sistema operativo può usare i 31 bit

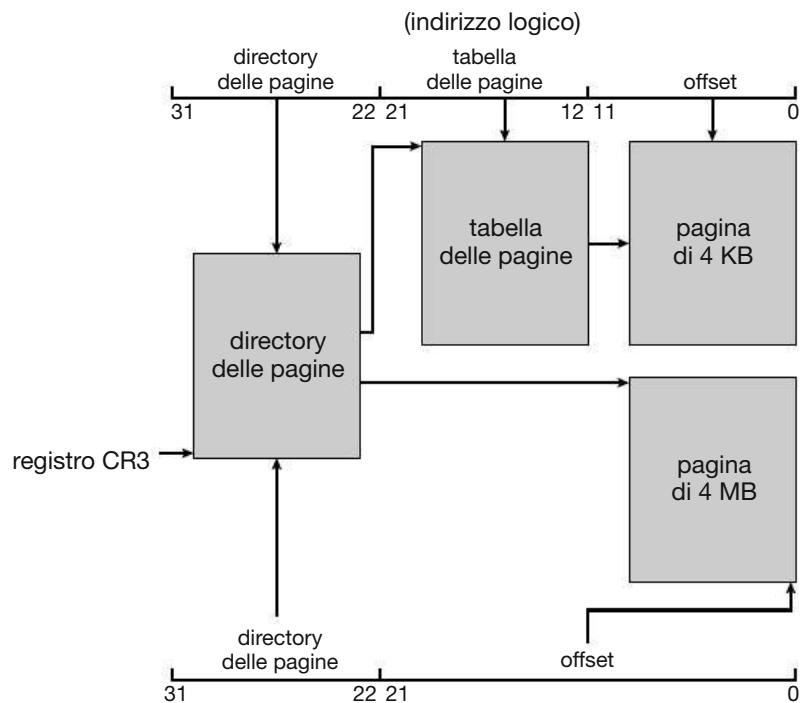


Figura 8.23 Paginazione nell'architettura IA-32.

rimanenti per specificare la collocazione della tabella sul disco; in questo modo, si può richiamare la tabella in memoria su richiesta.

Non appena gli sviluppatori di software hanno iniziato a soffrire della limitazione della memoria a 4 GB imposta dall'architettura a 32 bit, Intel ha introdotto l'**estensione di indirizzo della pagina** (PAE, *page address extension*), che consente ai processori a 32 bit di accedere a uno spazio di indirizzamento fisico più grande di 4 GB. La differenza fondamentale introdotta dal supporto PAE era il passaggio della paginazione da una schema a due livelli (come mostrato nella Figura 8.23) a uno schema a tre livelli, in cui i primi due bit fanno riferimento a una tabella di puntatori alle directory di pagina. La Figura 8.24 illustra un sistema PAE con pagine di 4 KB. PAE supporta anche pagine di 2 MB.

Con PAE è stata inoltre aumentata la dimensione degli elementi della directory delle pagine e della tabella delle pagine, che passa da 32 a 64 bit, permettendo di estendere l'indirizzo di base delle tabelle delle pagine e dei frame da 20 a 24 bit. In combinazione con i 12 bit di offset, l'aggiunta del supporto PAE a IA-32 ha aumentato lo spazio di indirizzamento a 36 bit, garantendo il supporto di un massimo di 64 GB di memoria fisica. È importante notare che per utilizzare PAE è necessario il supporto del sistema operativo. Linux e Intel Mac OS X supportano PAE. Tuttavia, le versioni a 32 bit dei sistemi operativi Windows per desktop supportano soltanto 4 GB di memoria fisica, anche se PAE è abilitato.

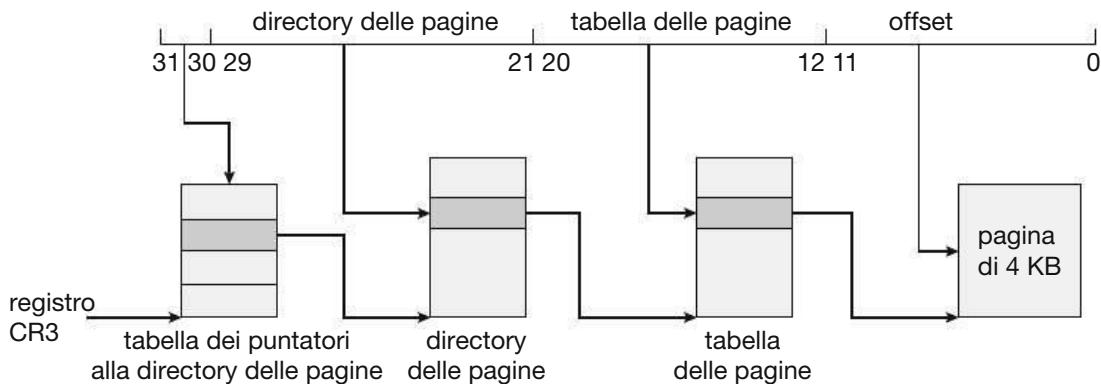


Figura 8.24 Estensione degli indirizzi di pagina.

8.7.2 Architettura x86-64

Lo sviluppo di architetture Intel a 64 bit ha avuto una storia curiosa. La prima di queste architetture era IA-64 (in seguito denominata **Itanium**), ma questa architettura non ha avuto un'ampia diffusione. Nel frattempo, un altro produttore di chip – AMD – ha iniziato a sviluppare un'architettura a 64 bit nota come x86-64, basata sull'estensione del set di istruzioni IA-32 esistente. L'architettura x86-64 supportava spazi di indirizzamento logico e fisico molto più grandi e introduceva diverse altre novità architettoniche. Storicamente AMD aveva spesso sviluppato chip basati sull'architettura Intel, ma in questo caso i ruoli si sono invertiti e Intel ha adottato l'architettura x86-64 di AMD. Nel discutere questa architettura, piuttosto che utilizzare le denominazioni commerciali **AMD64** e **Intel 64**, useremo il termine più generale **x86-64**.

Il supporto a uno spazio di indirizzamento a 64 bit permette di indirizzare la straordinaria quantità di 2^{64} byte di memoria – un numero superiore a 16 miliardi di miliardi (o 16 exabyte). Tuttavia, anche se i sistemi a 64 bit possono potenzialmente indirizzare una tale quantità di memoria, nella pratica vengono utilizzati nei progetti attuali assai meno di 64 bit per la rappresentazione di un indirizzo. L'architettura x86-64 utilizza attualmente un indirizzo virtuale di 48 bit con supporto ai formati di pagina di 4 KB, 2 MB o 1 GB utilizzando una paginazione a quattro livelli. La rappresentazione dell'indirizzo lineare è mostrata nella Figura 8.25. Poiché questo schema di indirizzamento può usare PAE, gli indirizzi virtuali sono di 48 bit, ma supportano indirizzi fisici a 52 bit (4096 terabyte).

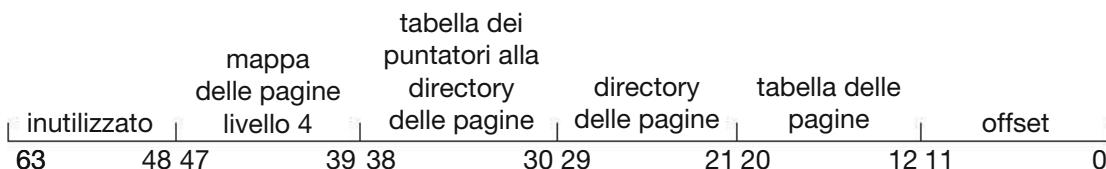


Figura 8.25 Indirizzo lineare in x86-64.

ELABORAZIONE A 64-BIT

La storia ci insegna che anche quando la capacità di memoria, la velocità della CPU e altre caratteristiche simili sembrano sufficienti a soddisfare la domanda futura, il progresso tecnologico riesce alla fine ad assorbire tutte le risorse disponibili. Ci si ritrova dunque ad aver bisogno di più memoria o di una maggior capacità di elaborazione e spesso ciò avviene prima del previsto. Cosa ci porteranno le future tecnologie per far sembrare troppo piccoli gli indirizzi di 64 bit?

8.8 Esempio: architettura ARM

Anche se i chip Intel hanno dominato il mercato dei personal computer per oltre 30 anni, i dispositivi mobili come smartphone e tablet montano spesso processori ARM a 32 bit. È interessante notare che mentre Intel progetta e produce i chip, ARM li progetta soltanto, concedendo poi in licenza i suoi progetti ai produttori di chip. Apple ha adottato ARM per i suoi dispositivi mobili iPhone e iPad e anche diversi smartphone basati su Android utilizzano processori ARM.

L'architettura ARM a 32 bit supporta i seguenti formati di pagina:

1. Pagine di 4 KB e 16 KB
2. Pagine di 1 MB e 16 MB (chiamate **sezioni**)

Il sistema di paginazione utilizzato dipende dal fatto che si faccia riferimento a una pagina piuttosto che a una sezione. Per sezioni di 1 e 16 MB viene utilizzata una paginazione a un livello, mentre per pagine di 4 KB e 16 KB si utilizza una paginazione a due livelli. La traduzione degli indirizzi con la MMU ARM è mostrata nella Figura 8.26.

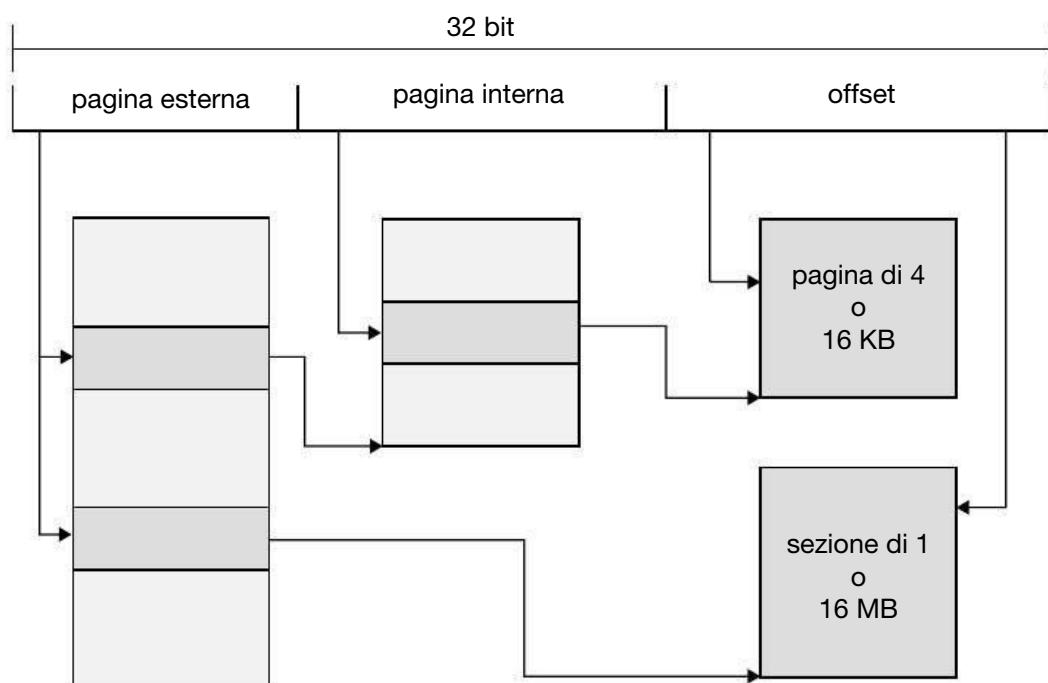


Figura 8.26 Traduzione degli indirizzi logici in ARM.

L'architettura ARM supporta inoltre due livelli di TLB. A livello esterno vi sono due micro TLB, una separata per i dati e una per le istruzioni. La micro TLB supporta gli ASID. A livello interno vi è un'unica TLB principale. La traduzione di un indirizzo inizia a livello micro TLB: in caso di insuccesso viene controllata la TLB principale. In caso di ulteriore insuccesso, ci si rivolge, via hardware, alla tabella delle pagine.

8.9 Sommario

Gli algoritmi di gestione della memoria per sistemi operativi multiprogrammati variano dal metodo più semplice, per sistema con singolo utente, fino alla segmentazione e alla paginazione. Il fattore più rilevante che incide sulla scelta del metodo da seguire in un sistema particolare è costituito dall'hardware disponibile. È necessario controllare la validità di ogni indirizzo di memoria generato dalla CPU; inoltre tali indirizzi, se sono corretti, devono essere tradotti in un indirizzo fisico. Tali controlli non possono essere realizzati efficientemente in software. Quindi dipendiamo dall'hardware disponibile.

Gli algoritmi di gestione della memoria (allocazione contigua, paginazione, segmentazione e combinazione di paginazione e segmentazione) differiscono per molti aspetti. Per confrontare i diversi metodi di gestione della memoria si possono considerare i seguenti elementi.

- **Supporto hardware.** Un semplice registro di base oppure una coppia di registri di base e limite è sufficiente per i metodi con partizione singola e con più partizioni, mentre la paginazione e la segmentazione necessitano di tabelle di traduzione per definire la corrispondenza degli indirizzi.
- **Prestazioni.** Aumentando la complessità dell'algoritmo di gestione della memoria, aumenta anche il tempo necessario per tradurre un indirizzo logico in un indirizzo fisico. Nei sistemi più semplici è sufficiente fare un confronto o sommare un valore all'indirizzo logico; si tratta di operazioni rapide. Paginazione e segmentazione possono essere altrettanto rapide se per realizzare la tabella s'impiegano registri veloci. Se però la tabella si trova in memoria, gli accessi alla memoria utente possono essere assai più lenti. Una TLB può limitare il calo delle prestazioni a un livello accettabile.
- **Frammentazione.** Un sistema multiprogrammato esegue le elaborazioni generalmente in modo più efficiente se ha un più elevato livello di multiprogrammazione. Per un dato gruppo di processi, il livello di multiprogrammazione si può aumentare solo compattando più processi in memoria. Per eseguire questo compito occorre ridurre lo spreco di memoria o la frammentazione. Sistemi con unità di allocazione di dimensione fissa, come lo schema con partizione singola e la paginazione, soffrono di frammentazione interna. Sistemi con unità di allocazione di dimensione variabile, come lo schema con più partizioni e la segmentazione, soffrono di frammentazione esterna.

- **Rilocazione.** Una soluzione al problema della frammentazione esterna è data dalla compattazione, che implica lo spostamento di un programma in memoria, senza che il programma stesso si accorga del cambiamento. Ciò richiede che gli indirizzi logici siano rilocati dinamicamente al momento dell'esecuzione. Se gli indirizzi si rilocano solo al momento del caricamento, non è possibile compattare la memoria.
- **Avvicendamento dei processi (swapping).** L'avvicendamento dei processi (*swapping*) si può incorporare in ogni algoritmo. I processi si copiano dalla memoria centrale alla memoria ausiliaria, e successivamente si ricopiano in memoria centrale a intervalli fissati dal sistema operativo, e generalmente stabiliti dai criteri di scheduling della CPU. Questo schema permette di eseguire contemporaneamente più processi di quanti si possano inserire in memoria. In generale i sistemi operativi per PC supportano lo swapping, mentre i sistemi per dispositivi mobili non lo supportano.
- **Condivisione.** Un altro mezzo per aumentare il livello di multiprogrammazione è quello della condivisione del codice e dei dati tra diversi utenti. La condivisione generalmente richiede l'uso della paginazione o della segmentazione, poiché deve fornire piccoli pacchetti d'informazioni (pagine o segmenti) condivisibili. La condivisione permette di eseguire molti processi con una quantità di memoria limitata, ma i programmi e i dati condivisi si devono progettare con estrema cura.
- **Protezione.** Con la paginazione o la segmentazione, diverse sezioni di un programma utente si possono dichiarare di sola esecuzione, di sola lettura oppure di lettura e scrittura. Questa limitazione è necessaria per il codice e i dati condivisi, ed è utile, in genere, in quanto fornisce semplici controlli nella fase d'esecuzione per l'individuazione degli errori di programmazione.

Esercizi di ripasso

8.1 Citate due differenze tra indirizzi logici e fisici.

8.2 Considerate un sistema nel quale un programma possa essere separato in due parti: codice e dati. Il processore sa se necessita di un'istruzione (prelievo di istruzione) o di un dato (prelievo o memorizzazione di dati). Perciò, vengono fornite due coppie di registri base e limite: una per le istruzioni e una per i dati. La coppia di registri base e limite per le istruzioni è automaticamente a sola lettura, di modo che i programmi possano essere condivisi tra i diversi utenti. Discutete i vantaggi e gli svantaggi di questo schema.

8.3 Perché la dimensione delle pagine è sempre una potenza di due?

8.4 Considerate uno spazio degli indirizzi logici di 64 pagine, ciascuna delle quali di 1024 parole, mappato su una memoria fisica di 32 frame.

- Quanti bit ci sono nell'indirizzo logico?
- Quanti bit ci sono nell'indirizzo fisico?

- 8.5** Quale effetto si verifica se si permette a due voci di una tabella delle pagine di puntare allo stesso frame di pagina della memoria? Spiegate come questo effetto potrebbe essere utilizzato per diminuire il tempo necessario per copiare una grande quantità di memoria da uno spazio a un altro. Nel caso in cui vengano aggiornati alcuni byte della prima pagina, quale effetto si avrebbe sulla seconda pagina?
- 8.6** Descrivete un meccanismo per il quale un segmento potrebbe appartenere allo spazio degli indirizzi di due processi differenti.
- 8.7** In un sistema di segmentazione linkato dinamicamente è possibile condividere segmenti tra processi senza richiedere che abbiano lo stesso numero di segmento.
- Definite un sistema che permetta il linking statico e la condivisione di segmenti senza richiedere che il numero del segmento sia lo stesso.
 - Descrivete uno schema di paginazione che permetta alle pagine di essere condivise senza richiedere ai numeri delle pagine di essere gli stessi.
- 8.8** Nell'IBM/370 la memoria viene protetta attraverso l'uso di *chiavi*. Una chiave è di 4 bit. Ogni blocco di memoria di 2 K è associato a una chiave (la chiave di memoria). Anche il processore è associato a una chiave (la chiave di protezione). Un'operazione di memorizzazione è permessa solo se entrambe le chiavi sono uguali oppure se una è uguale zero. Quale dei seguenti schemi di gestione della memoria potrebbe essere usato con successo con questo hardware?
- Macchina nuda.
 - Sistema a singolo utente.
 - Programmazione multipla con un numero fisso di processi.
 - Programmazione multipla con un numero variabile di processi.
 - Paginazione.
 - Segmentazione.

Esercizi

- 8.9** Spiegate la differenza tra frammentazione interna e frammentazione esterna.
- 8.10** Considerate il seguente ciclo di produzione di codice binario eseguibile. Si usa un compilatore per generare il codice oggetto dei singoli moduli, e un editor per gestire la fase di link, ossia per combinare diversi moduli oggetto in un unico codice binario eseguibile. Come può l'editor modificare l'associazione di istruzioni e dati, agli indirizzi di memoria? Quali informazioni devono passare dal compilatore all'editor per facilitare l'editor nell'esecuzione di tale associazione?

- 8.11** Date sei partizioni di memoria, pari a 300 KB, 600 KB, 350 KB, 200 KB, 750 KB e 125 KB, nell’ordine, e cinque processi di 115 KB, 500 KB, 358 KB, 200 KB e 375 KB, nell’ordine, come verrebbero allocati in memoria tali processi dagli algoritmi first-fit, best-fit e worst-fit? Ordinate gli algoritmi secondo la loro efficienza nell’utilizzo della memoria.
- 8.12** La maggioranza dei sistemi consente ai programmi di aumentare durante l’esecuzione la memoria allocata al proprio spazio di indirizzi. I dati posti nei segmenti heap dei programmi ne rappresentano un esempio. Di che cosa c’è bisogno per agevolare l’allocazione dinamica della memoria, in ognuno dei seguenti casi?
- Allocazione contigua della memoria.
 - Segmentazione pura.
 - Paginazione pura.
- 8.13** Confrontate i modelli della segmentazione pura, paginazione pura e allocazione contigua in riferimento alle seguenti tematiche:
- frammentazione esterna;
 - frammentazione interna;
 - capacità di condividere codice tra i processi.
- 8.14** Nei sistemi che si avvalgono della paginazione i processi non possono accedere alla memoria che non possiedono. Perché? In che modo potrebbe il sistema operativo concedere l’accesso a tale memoria estranea? Argomentate perché dovrebbe o non dovrebbe farlo.
- 8.15** Spiegate perché i sistemi operativi mobili come iOS e Android non supportano l’avvicendamento dei processi (swapping).
- 8.16** Anche se Android non supporta lo swapping sul suo disco di avvio, è possibile impostare uno spazio di swap usando una scheda di memoria non volatile (una scheda SD). Per quale ragione Android non consente lo swapping sul proprio disco di avvio per poi permetterlo su un disco secondario?
- 8.17** Confrontate la paginazione con la segmentazione, in riferimento alla quantità di memoria richiesta dalle strutture di traduzione degli indirizzi al fine di convertire gli indirizzi virtuali in indirizzi fisici.
- 8.18** Spiegate perché vengono utilizzati gli identificatori dello spazio d’indirizzi (ASID).
- 8.19** Il codice binario eseguibile di un programma, in molti sistemi, ha la seguente forma tipica. Il codice è memorizzato a partire da un piccolo indirizzo virtuale fisso, come, per esempio, 0. Al segmento del codice fa seguito il segmento dei dati, utilizzato per memorizzare le variabili del programma. Quando il program-

ma dà avvio all'esecuzione, lo stack è collocato all'altro estremo dello spazio degli indirizzi virtuali, e ha, dunque, la possibilità di espandersi verso gli indirizzi virtuali inferiori. Quale rilevanza assume la struttura ora descritta nelle seguenti circostanze?

- a. Allocazione contigua della memoria.
- b. Segmentazione pura.
- c. Paginazione pura.

8.20 Assumendo che la dimensione della pagina sia di 1 KB, quali sono i numeri di pagina e gli scostamenti per i seguenti indirizzi (indicati in numeri decimali):

- a. 3085
- b. 42095
- c. 215201
- d. 650000
- e. 2000001

8.21 Il sistema operativo BTV ha indirizzi virtuali a 21 bit, anche se su alcuni dispositivi embedded ha indirizzi fisici di solo 16 bit. La dimensione di pagina è di 2 KB. Quante voci ci sono in ognuna delle seguenti strutture dati?

- a. Una tabella delle pagine convenzionale, a singolo livello.
- b. Una tabella delle pagine invertita.

8.22 Considerate uno spazio di indirizzo logico di 256 pagine, con una dimensione di pagina di 4 KB, mappato su una memoria fisica di 64 frame.

- a. Quanti bit sono necessari all'indirizzo logico?
- b. Quanti bit sono necessari all'indirizzo fisico?

8.23 Prendete in considerazione un sistema con un indirizzo logico di 32 bit e una dimensione di pagina di 4 KB. Il sistema supporta fino a 512 MB di memoria fisica. Quante voci ci sono in:

- a. una tabella delle pagine convenzionale a singolo livello;
- b. una tabella delle pagine invertita.

8.24 Considerate un sistema di paginazione con la tabella delle pagine conservata in memoria.

- a. Se un riferimento alla memoria necessita di 50 nanosecondi, di quanto necessiterà un riferimento alla memoria paginata?
- b. Se si aggiungono TLB, e il 75 percento di tutti i riferimenti si trova in questi ultimi, quale sarà il tempo effettivo di riferimento alla memoria? (Ipotizzate che la ricerca di un elemento effettivamente presente nelle TLB richieda un tempo di 2 nanosecondi.)

8.25 Spiegate perché segmentazione e paginazione si combinino talvolta in un unico schema.

8.26 Spiegate perché sia più facile condividere un modulo di codice rientrante usando la segmentazione anziché la paginazione pura.

8.27 Considerate la seguente tabella dei segmenti:

Segmento	Base	Lunghezza
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

Calcolate gli indirizzi fisici corrispondenti ai seguenti indirizzi logici:

- a. 0,430
- b. 1,10
- c. 2,500
- d. 3,400
- e. 4,112

8.28 Qual è lo scopo di paginare le tabelle delle pagine?

8.29 Considerate il meccanismo gerarchico di paginazione impiegato dall'architettura VAX. Quante operazioni di memoria si effettuano quando un programma utente esegue un'operazione di lettura (*load*) in memoria?

8.30 Confrontate il meccanismo di paginazione segmentata con il modello delle tabelle hash delle pagine, per gestire spazi degli indirizzi grandi. In quali circostanze è preferibile optare per l'uno o per l'altro?

8.31 Considerate lo schema di traduzione degli indirizzi di Intel Pentium mostrato nella Figura 8.22.

- a. Descrivete tutti i passi eseguiti da Intel Pentium nel tradurre un indirizzo logico in un indirizzo fisico.
- b. Esponete i vantaggi offerti a un sistema operativo da un'architettura dotata di un così complesso sistema di traduzione degli indirizzi.
- c. Dite se ci sono svantaggi in questo sistema di traduzione degli indirizzi; se sì, dite quali sono; altrimenti spiegate perché non è impiegato da ogni costruttore.

Problemi di programmazione

8.32 Assumete che un sistema abbia un indirizzo virtuale di 32 bit con una dimensione della pagina di 4 KB. Scrivete un programma in C al quale viene passato un indirizzo virtuale (in decimale) dalla riga di comando e che fornisce in output il numero di pagina e l'offset per l'indirizzo dato. Per esempio, se il programma fosse invocato come segue

```
./a.out 19986
```

il risultato sarebbe:

```
The address 19986 contains:  
page number = 4  
offset = 3602
```

Scrivere questo programma richiederà di utilizzare tipi di dati appropriati per memorizzare 32 bit. Suggeriamo inoltre di utilizzare tipi di dati `unsigned`.

Note bibliografiche

L'allocazione dinamica della memoria è analizzata nel Paragrafo 2.5 di [Knuth 1973], il quale, tramite i risultati di simulazioni, scoprì che il criterio di scelta del primo buco abbastanza grande (*first-fit*) è in genere più vantaggioso del criterio di scelta del più piccolo tra i buchi abbastanza grandi (*best-fit*). Nella stessa opera, Knuth discute la regola del 50 per cento.

Il concetto di paginazione si può attribuire ai progettisti del sistema Atlas, descritto sia da [Kilburn et al. 1961] che da [Howarth et al. 1961]. Il concetto di segmentazione è stato discusso per la prima volta da [Dennis 1965]. Il primo sistema con segmentazione paginata è stato il GE 645, per il quale era stato originariamente progettato e realizzato il sistema operativo MULTICS ([Organick 1972] e [Daley e Dennis 1967]).

Le tabelle delle pagine invertite sono trattate in un articolo sulla gestione della memoria dell'IBM RT, di [Chang e Mergen 1988].

[Jacob e Mudge 1997] prendono in esame la traduzione software degli indirizzi.

[Hennessy e Patterson 2012] si occupano degli aspetti architetturali di TLB, cache e MMU. Le tabelle delle pagine per spazi di indirizzi a 64 bit sono trattate in [Talluri et al. 1995]. [Jacob and Mudge (2001)] espongono le tecniche di gestione delle TLB. Le soluzioni per la gestione di pagine grandi sono analizzate in [Fang et al. 2001].

<http://msdn.microsoft.com/en-us/library/windows/hardware/gg487512.aspx> tratta il supporto PAE nei sistemi Windows.

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html> contiene diversi manuali relativi alle architetture Intel 64 e IA-32.

<http://www.arm.com/products/processors/cortex-a/cortex-a9.php> offre una panoramica dell'architettura ARM.

Bibliografia

- [**Chang e Mergen 1988**] A. Chang e M. F. Mergen, “801 Storage: Architecture and Programming”, ACM Transactions on Computer Systems, Vol. 6, N. 1, p. 28–50, 1988.
- [**Daley e Dennis 1967**] R. C. Daley e J. B. Dennis, “Virtual Memory, Processes, and Sharing in Multics”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 121–128, 1967.
- [**Dennis 1965**] J. B. Dennis, “Segmentation and the Design of Multiprogrammed Computer Systems”, Communications of the ACM, Vol. 8, N.4, p. 589–602, 1965.
- [**Fang et al. 2001**] Z. Fang, L. Zhang, J. B. Carter, W.C. Hsieh e S. A. McKee, “Re-evaluating Online Superpage Promotion with Hardware Support”, Proceedings of the International Symposium on High-Performance Computer Architecture, Vol. 50, N. 5, 2001.
- [**Hennessy e Patterson 2012**] J. Hennessy e D. Patterson, *Computer Architecture: A Quantitative Approach*, Fifth Edition, Morgan Kaufmann, 2012.
- [**Howarth et al. 1961**] D. J. Howarth, R. B. Payne e F. H. Sumner, “The Manchester University Atlas Operating System, Part II: User’s Description”, Computer Journal, Vol. 4, N. 3, p. 226–229, 1961.
- [**Jacob e Mudge 2001**] B. Jacob e T. Mudge, “Uniprocessor Virtual Memory Without TLBs”, IEEE Transactions on Computers, Vol. 50, N. 5, 2001.
- [**Kilburn et al. 1961**] T. Kilburn, D. J. Howarth, R. B. Payne e F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, Computer Journal, Vol. 4, N. 3, p. 222–225, 1961.
- [**Knuth 1973**] D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, Second Edition, Addison-Wesley, 1973.
- [**Organick 1972**] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
- [**Talluri et al. 1995**] M. Talluri, M. D. Hill e Y. A. Khalidi, “A New Page Table for 64-bit Address Spaces”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 184–200, 1995.

CAPITOLO

9

OBIETTIVI DEL CAPITOLO

- Descrizione dei vantaggi derivanti dalla memoria virtuale.
- Definizione dei concetti di paginazione su richiesta, algoritmi di sostituzione di pagina e allocazione dei frame.
- Trattazione dei principi del modello del working-set.
- Studio della relazione tra memoria condivisa e file mappati in memoria.
- Analisi della gestione della memoria del kernel.

Memoria virtuale

Nel Capitolo 8 sono state esaminate varie strategie di gestione della memoria impiegate nei calcolatori. Hanno tutte lo stesso scopo: tenere contemporaneamente più processi in memoria per permettere la multiprogrammazione; tuttavia esse tendono a richiedere che l'intero processo si trovi in memoria prima di essere eseguito.

La **memoria virtuale** è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria. Il vantaggio principale offerto da questa tecnica è quello di permettere che i programmi siano più grandi della memoria fisica; inoltre la memoria virtuale astrae la memoria centrale in un vettore di memorizzazione molto grande e uniforme, separando la memoria logica, com'è vista dall'utente, da quella fisica. Questa tecnica libera i programmatore dai problemi di limitazione della memoria. La memoria virtuale permette inoltre ai processi di condividere facilmente file e di realizzare memorie condivise, e fornisce un meccanismo efficiente per la creazione dei processi. La memoria virtuale è però difficile da realizzare e, s'è usata scorrettamente, può ridurre di molto le prestazioni del sistema. In questo capitolo si esamina la memoria virtuale nella forma della paginazione su richiesta e se ne valutano complessità e costi.

9.1 Introduzione

Gli algoritmi di gestione della memoria delineati nel Capitolo 8 sono necessari a causa di un requisito fondamentale: le istruzioni da eseguire si devono trovare all'interno della memoria fisica. Il primo metodo per far fronte a tale requisito consiste nel collocare l'intero spazio d'indirizzi logici del processo in memoria fisica. Il caricamento dinamico può aiutare ad attenuare gli effetti di tale limitazione, ma richiede generalmente particolari precauzioni e un ulteriore impegno dei programmatore.

La condizione che le istruzioni debbano essere nella memoria fisica sembra tanto necessaria quanto ragionevole, ma purtroppo riduce le dimensioni dei programmi a valori strettamente correlati alle dimensioni della memoria fisica. In effetti, da un esame dei programmi reali risulta che in molti casi non è necessario avere in memoria l'intero programma; si considerino per esempio le seguenti situazioni.

- Spesso i programmi dispongono di codice per la gestione di condizioni d'errore insolite. Poiché questi errori sono rari, se non inesistenti, anche il relativo codice non si esegue quasi mai.
- Spesso ad array, liste e tavole si assegna più memoria di quanta sia effettivamente necessaria. Un array si può dichiarare di 100 per 100 elementi, anche se raramente contiene più di 10 per 10 elementi. La tabella dei simboli di un assemblatore può avere spazio per 3000 simboli, anche se un programma medio ne ha meno di 200.
- Alcune opzioni e caratteristiche di un programma sono utilizzabili solo di rado. Alcune routine di certi programmi amministrativi non vengono eseguite per anni.

Anche nei casi in cui è necessario disporre di tutto il programma, è possibile che non serva tutto in una volta.

La possibilità di eseguire un programma che si trova solo parzialmente in memoria porterebbe molti benefici.

- Un programma non è più vincolato alla quantità di memoria fisica disponibile. Gli utenti possono scrivere programmi per uno **spazio degli indirizzi virtuali** molto grande, semplificando così il compito della programmazione.
- Poiché ogni programma utente può impiegare meno memoria fisica, si possono eseguire molti più programmi contemporaneamente, ottenendo un corrispondente aumento dell'utilizzo e della produttività della CPU senza aumentare il tempo di risposta o di completamento.
- Per caricare (o avvicendare) ogni programma utente in memoria sono necessarie meno operazioni di I/O, quindi ogni programma utente è eseguito più rapidamente.

La possibilità di eseguire un programma che non si trovi completamente in memoria apporterebbe quindi vantaggi sia al sistema sia all'utente.

La **memoria virtuale** si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica. Questa separazione permette di offrire ai programmatore una memoria virtuale molto ampia, anche se la memoria fisica disponibile è

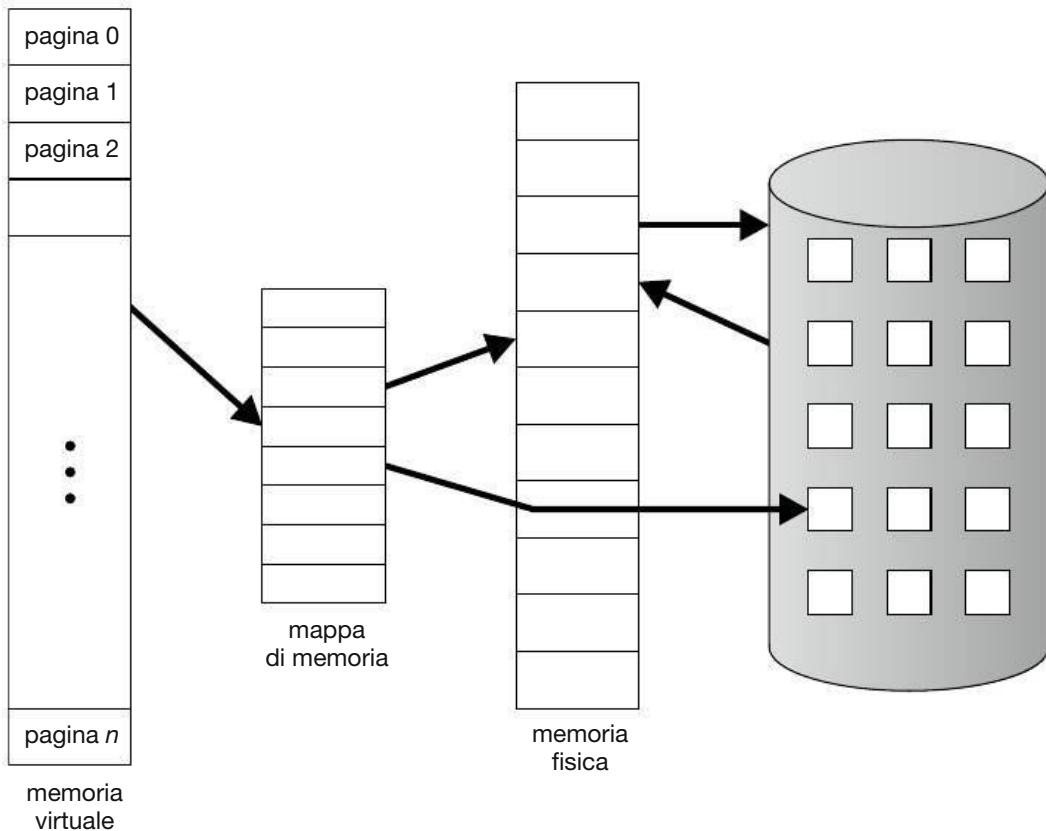


Figura 9.1 Schema che mostra una memoria virtuale più grande di quella fisica.

più piccola, com’è illustrato nella Figura 9.1. La memoria virtuale facilita la programmazione, poiché il programmatore non deve preoccuparsi della quantità di memoria fisica disponibile, ma può concentrarsi sul problema da risolvere con il programma.

L’espressione **spazio degli indirizzi virtuali** si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Tipicamente, da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico – per esempio, l’indirizzo 0 – e si estende in uno spazio di memoria contigua, come evidenziato dalla Figura 9.2. Come si ricorderà dal Capitolo 8, è tuttavia possibile organizzare la memoria fisica in frame di pagine; in questo caso i frame delle pagine fisiche assegnati ai processi possono non essere contigui. Spetta all’unità di gestione della memoria (MMU) associare in memoria le pagine logiche alle pagine fisiche.

Si noti come, nella Figura 9.2, allo heap sia lasciato sufficiente spazio per crescere verso l’alto nello spazio di memoria, poiché esso ospita la memoria allocata dinamicamente. In modo analogo, consentiamo allo stack di svilupparsi verso il basso nella memoria, quando vengono effettuate ripetute chiamate di funzione. L’ampio spazio vuoto (o buco) che separa lo heap dallo stack è parte dello spazio degli indirizzi virtuali, ma richiede pagine fisiche reali solo nel caso che lo heap o lo stack crescano. Uno spazio degli indirizzi virtuali che contiene buchi si definisce sparso. Un simile spazio degli indirizzi è utile, poiché i buchi possono essere riempiti grazie all’espansione.

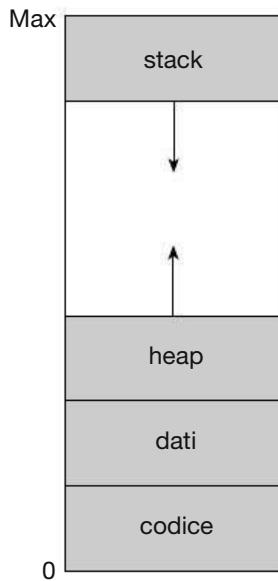


Figura 9.2 Spazio degli indirizzi virtuali.

sione dei segmenti heap o stack, oppure se vogliamo collegare dinamicamente delle librerie (o altri oggetti condivisi) durante l'esecuzione del programma.

Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre il vantaggio di condividere i file e la memoria fra due o più processi, mediante la condivisione delle pagine (Paragrafo 8.5.4). Ciò comporta i seguenti vantaggi.

- Le librerie di sistema sono condivisibili da diversi processi associando (“mappando”) l’oggetto di memoria condiviso a uno spazio degli indirizzi virtuali. Benché ciascun processo veda le librerie condivise come parte del proprio spazio degli indirizzi virtuali, le pagine che ospitano effettivamente le librerie nella memoria fisica sono in condivisione tra tutti i processi (Figura 9.3). In genere le librerie si associano allo spazio di ogni processo a loro collegato, in modalità di sola lettura.
- In maniera analoga, la memoria può essere condivisa tra processi distinti. Come si rammenterà dal Capitolo 3, due o più processi possono comunicare condividendo memoria. La memoria virtuale permette a un processo di creare una regione di memoria condivisibile da un altro processo. I processi che condividono questa regione la considerano parte del proprio spazio degli indirizzi virtuali, malgrado le pagine fisiche siano, in realtà, condivise, come illustrato sempre dalla Figura 9.3.
- Le pagine possono essere condivise durante la creazione di un processo mediante la chiamata di sistema `fork()`, così da velocizzare la generazione dei processi.
- Approfondiremo questi e altri vantaggi offerti dalla memoria virtuale nel corso di questo capitolo. In primo luogo, però, ci soffermeremo sulla memoria virtuale realizzata attraverso la paginazione su richiesta.

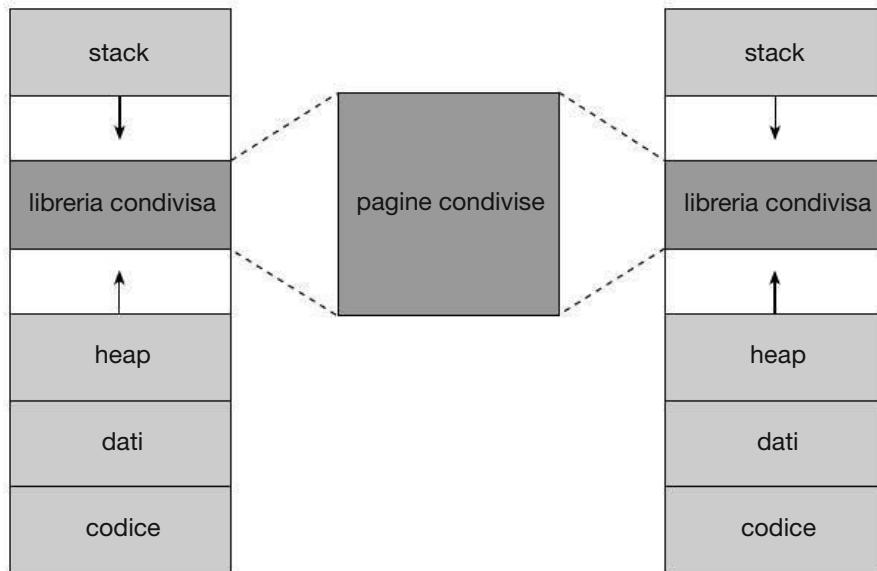


Figura 9.3 Condivisione delle librerie tramite la memoria virtuale.

9.2 Paginazione su richiesta

Si consideri il caricamento in memoria di un eseguibile residente su disco. Una possibilità è quella di caricare l'intero programma nella memoria fisica al momento dell'esecuzione. Il problema, però, è che all'inizio non è detto che serva avere tutto il programma in memoria: se il programma, per esempio, fornisce all'avvio una lista di opzioni all'utente, è inutile caricare il codice per l'esecuzione di *tutte* le opzioni previste, senza tener conto di quella effettivamente scelta dall'utente. Una strategia alternativa consiste nel caricare le pagine nel momento in cui servono realmente; si tratta di una tecnica, detta **paginazione su richiesta**, comunemente adottata dai sistemi con memoria virtuale. Secondo questo schema, le pagine sono caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica.

Un sistema di paginazione su richiesta è analogo a un sistema paginato con avvicendamento dei processi in memoria; si veda la Figura 9.4. I processi risiedono in memoria secondaria (generalmente su disco). Per eseguire un processo occorre caricarlo in memoria. Tuttavia, anziché caricare in memoria l'intero processo, si può seguire un metodo d'avvicendamento "pigro" (*lazy swapping*): non si carica mai in memoria una pagina che non sia necessaria. Nell'ambito dei sistemi con paginazione su richiesta l'uso del termine *avvicendamento* o *swapping* non è appropriato: uno swapper manipola interi processi mentre un **paginatore** (*pager*) gestisce le singole pagine dei processi.

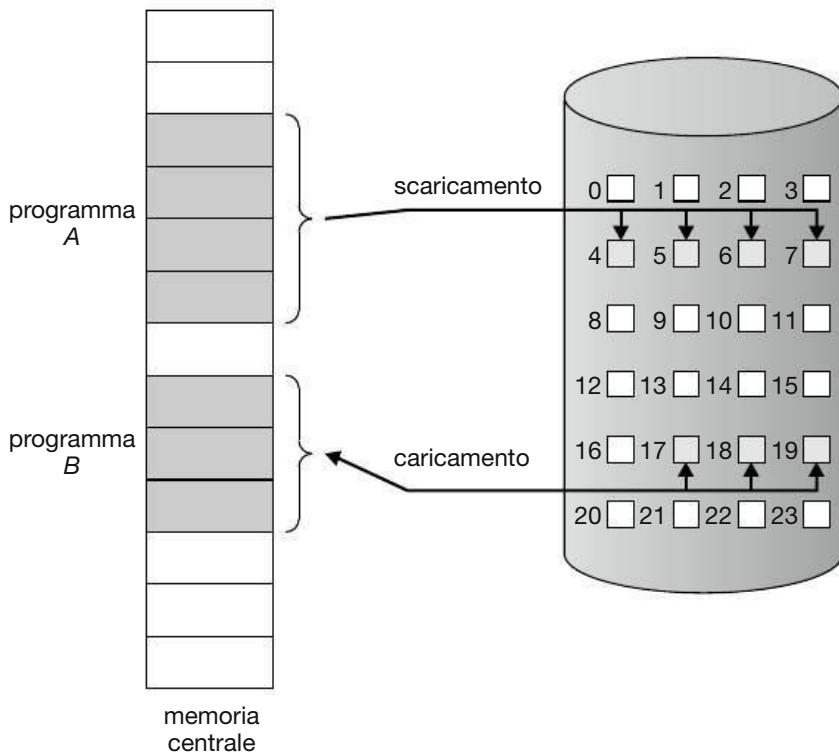


Figura 9.4 Trasferimento di una memoria paginata nello spazio contiguo di un disco.

9.2.1 Concetti fondamentali

Quando un processo sta per essere caricato in memoria, il paginatore ipotizza quali pagine saranno usate, prima che il processo sia nuovamente scaricato dalla memoria. Anziché caricare in memoria tutto il processo, il paginatore trasferisce in memoria solo le pagine che ritiene necessarie. In questo modo è possibile evitare il trasferimento in memoria di pagine che non sono effettivamente usate, riducendo il tempo d'avvicendamento e la quantità di memoria fisica richiesta.

Con tale schema è necessario che l'hardware fornisca un meccanismo che consenta di distinguere le pagine presenti in memoria da quelle nei dischi. A tal fine è utilizzabile lo schema basato sul bit di validità, descritto nel Paragrafo 8.5.3. In questo caso, però, il bit impostato come “valido” significa che la pagina corrispondente è valida ed è presente in memoria; il bit impostato come “non valido” indica che la pagina non è valida (cioè non appartiene allo spazio d'indirizzi logici del processo) oppure è valida ma è attualmente nel disco. L'elemento della tabella delle pagine di una pagina caricata in memoria s'imposta come al solito, mentre l'elemento della tabella delle pagine corrispondente a una pagina che attualmente non è in memoria è semplicemente contrassegnato come non valido oppure contiene l'indirizzo della pagina su disco. Tale situazione è illustrata nella Figura 9.5.

Occorre notare che indicare una pagina come non valida non sortisce alcun effetto se il processo non tenta mai di accedervi. Quindi, se l'ipotesi del paginatore è esatta

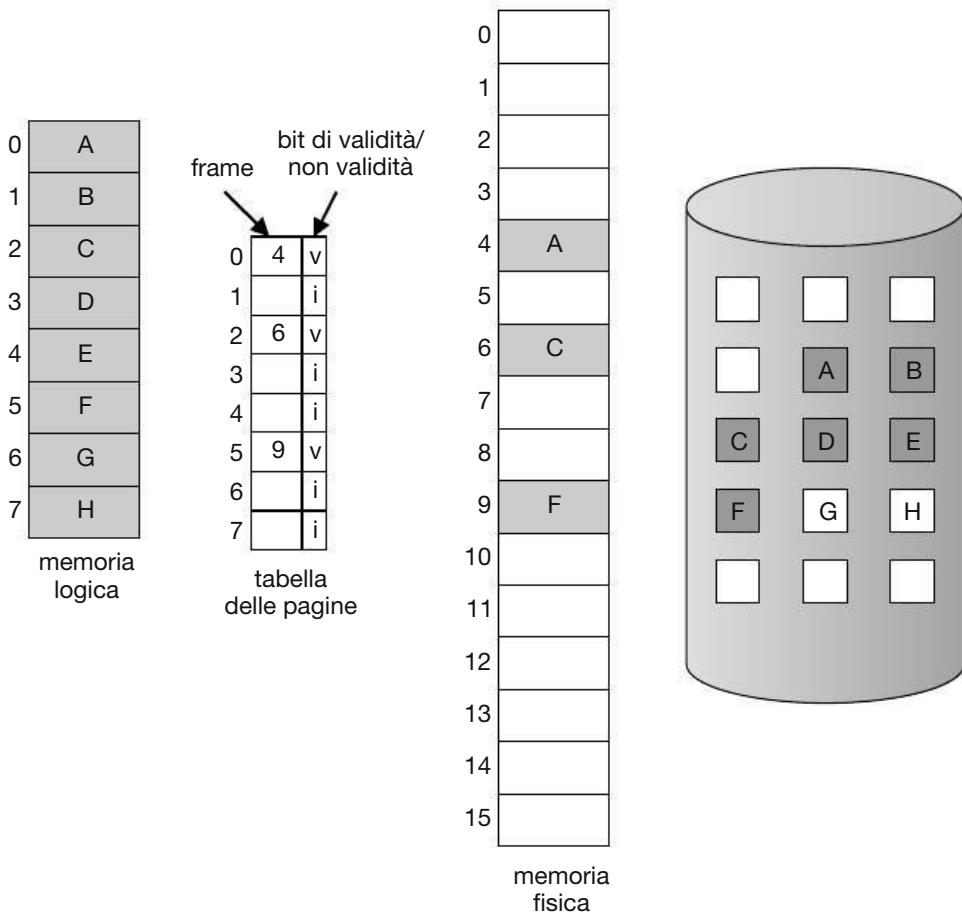


Figura 9.5 Tabella delle pagine quando alcune pagine non si trovano nella memoria centrale.

e si caricano tutte e solo le pagine che servono effettivamente, il processo è eseguito proprio come se fossero state caricate tutte le pagine. Durante l'esecuzione, finchè il processo accede alle pagine **residenti in memoria**, l'esecuzione procede come di consueto.

Che succede se il processo tenta l'accesso a una pagina che non era stata caricata in memoria? L'accesso a una pagina contrassegnata come non valida causa un evento o eccezione di **page fault** (*pagina mancante*). L'hardware di paginazione, traducendo l'indirizzo attraverso la tabella delle pagine, nota che il bit è non valido e genera una trap per il sistema operativo; tale eccezione è dovuta a un “insuccesso” del sistema operativo nella scelta delle pagine da caricare in memoria. La procedura di gestione dell'eccezione di page fault è lineare (Figura 9.6).

1. Si controlla una tabella interna per questo processo (in genere tale tabella è conservata insieme al blocco di controllo del processo) allo scopo di stabilire se il riferimento fosse un accesso alla memoria valido o non valido.
2. Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua il caricamento.

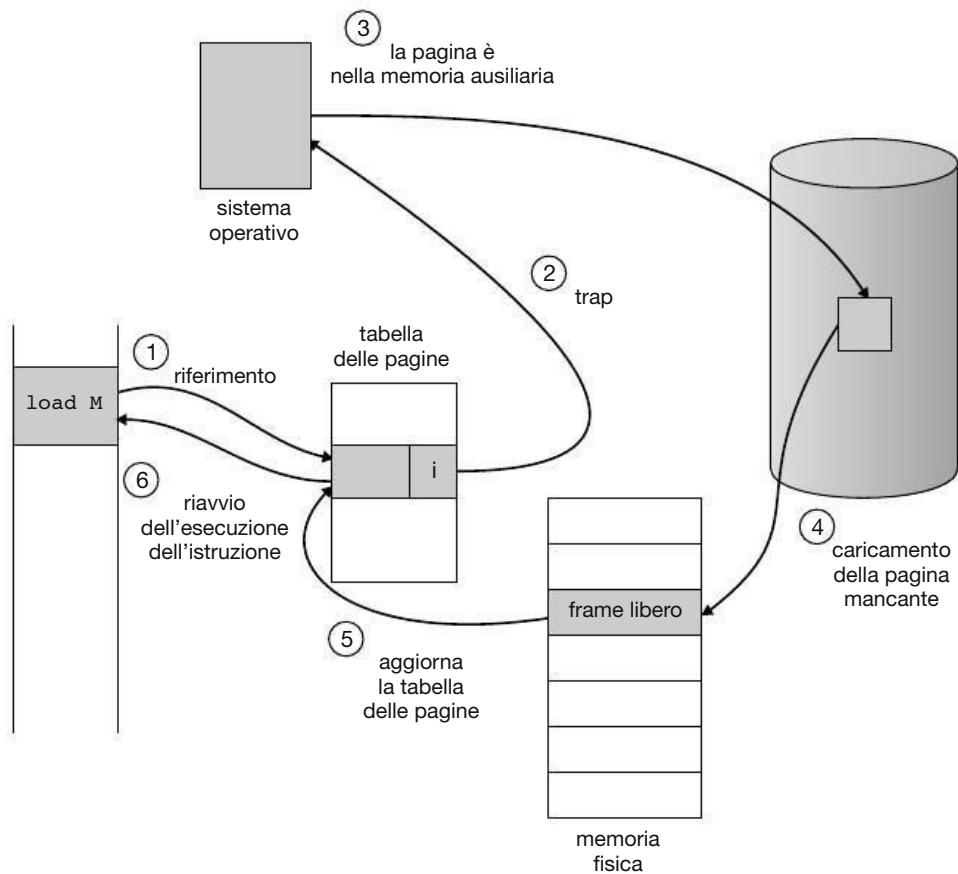


Figura 9.6 Fasi di gestione di un page fault.

3. Si individua un frame libero, per esempio prelevandone uno dalla lista dei frame liberi.
4. Si programma un'operazione sui dischi per trasferire la pagina desiderata nel frame appena assegnato.
5. Quando la lettura dal disco è completata, si modificano la tabella interna, conservata con il processo, e la tabella delle pagine per indicare che la pagina si trova attualmente in memoria.
6. Si riavvia l'istruzione interrotta dall'eccezione. A questo punto il processo può accedere alla pagina come se questa fosse stata sempre presente in memoria.

Come caso estremo, è possibile avviare l'esecuzione di un processo *senza* pagine in memoria. Quando il sistema operativo carica nel contatore di programma l'indirizzo della prima istruzione del processo, che è in una pagina non residente in memoria, il processo genera immediatamente un page fault. Una volta portata la pagina in memoria, il processo continua l'esecuzione, generando page fault fino a che tutte le pagine necessarie non si trovino effettivamente in memoria; a questo punto si può eseguire il processo senza ulteriori richieste. Lo schema descritto è una **paginazione su**

richiesta pura, vale a dire che una pagina non si trasferisce mai in memoria se non viene richiesta.

In teoria alcuni programmi possono accedere a diverse nuove pagine di memoria all'esecuzione di ogni istruzione (una pagina per l'istruzione e molte per i dati), eventualmente causando più page fault per ogni istruzione. In un caso simile le prestazioni del sistema sarebbero inaccettabili. Fortunatamente l'analisi dei programmi in esecuzione mostra che questo comportamento è estremamente improbabile. I programmi tendono ad avere una **località dei riferimenti**, descritta nel Paragrafo 9.6.1, quindi le prestazioni della paginazione su richiesta risultano ragionevoli.

L'hardware di supporto alla paginazione su richiesta è lo stesso che è richiesto per la paginazione e l'avvicendamento dei processi in memoria:

- **tabella delle pagine.** Questa tabella ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione;
- **memoria secondaria.** Questa memoria conserva le pagine non presenti in memoria centrale. Generalmente la memoria secondaria è costituita da un disco ad alta velocità detto dispositivo di swap; la sezione del disco usata a questo scopo si chiama **area di avvicendamento** (*swap space*). L'allocazione di quest'area è trattata nel Capitolo 10.

Uno dei requisiti cruciali della paginazione su richiesta è la possibilità di rieseguire una qualunque istruzione dopo un page fault. Avendo salvato lo stato del processo interrotto (registri, codici di condizione, contatore di programma) al momento del page fault, occorrerà riavviare il processo esattamente dallo stesso punto e con lo stesso stato, eccezion fatta per la presenza della pagina desiderata in memoria. Nella maggior parte dei casi questo requisito è facile da soddisfare. Un page fault si può verificare per qualsiasi riferimento alla memoria. Se si verifica durante la fase di fetch (prelievo) di un'istruzione, l'esecuzione si può riavviare effettuando nuovamente il fetch. Se si verifica durante il fetch di un operando, bisogna effettuare nuovamente fetch e decode dell'istruzione, quindi si può prelevare l'operando.

Come caso limite si consideri un'istruzione a tre indirizzi, come per esempio la somma (ADD) del contenuto di A al contenuto di B, con risultato posto in C. I passi necessari per eseguire l'istruzione sono i seguenti:

1. fetch e decodifica dell'istruzione (ADD);
2. prelievo del contenuto di A;
3. prelievo del contenuto di B;
4. addizione del contenuto di A al contenuto di B;
5. memorizzazione della somma in C.

Se il page fault avviene al momento della memorizzazione in C, poiché C si trova in una pagina che non è in memoria, occorre prelevare la pagina desiderata, caricarla in memoria, correggere la tabella delle pagine e riavviare l'istruzione. Il riavvio dell'i-

struzione richiede una nuova operazione di fetch, con nuova decodifica e nuovo prelievo dei due operandi; infine occorre ripetere l'addizione. In ogni modo il lavoro da ripetere non è molto, meno di un'istruzione completa, e la ripetizione è necessaria solo nel caso si verifichi un page fault.

La difficoltà maggiore si presenta quando un'istruzione può modificare parecchie locazioni diverse. Si consideri, per esempio, l'istruzione MVC (*move character*) del sistema IBM 360/370: quest'istruzione può spostare una sequenza di byte (fino a 256) da una locazione a un'altra (con possibilità di sovrapposizione). Se una delle sequenze (quella d'origine o quella di destinazione) esce dal confine di una pagina, si può verificare un page fault quando lo spostamento è stato effettuato solo in parte. Inoltre, se le sequenze d'origine e di destinazione si sovrappongono, è possibile che la sequenza d'origine sia stata modificata, in tal caso non è possibile limitarsi a riavviare l'istruzione.

Il problema si può risolvere in due modi. In una delle due soluzioni il microcodice computa e tenta di accedere alle estremità delle due sequenze di byte. Un'eventuale page fault si può verificare solo in questa fase, prima che si apporti qualsiasi modifica. A questo punto si può compiere lo spostamento senza rischio di page fault perché tutte le pagine interessate si trovano in memoria. L'altra soluzione si serve di registri temporanei per conservare i valori delle locazioni sovrascritte. Nel caso di un page fault, si riscrivono tutti i vecchi valori in memoria prima che sia generata la trap. Questa operazione riporta la memoria allo stato in cui si trovava prima che l'istruzione fosse avviata, perciò si può ripetere la sua esecuzione.

Sebbene non si tratti certo dell'unico problema da affrontare per estendere un'architettura esistente con la funzionalità della paginazione su richiesta, illustra alcune delle difficoltà da superare. Il sistema di paginazione si colloca tra la CPU e la memoria di un calcolatore e deve essere completamente trasparente al processo utente. L'opinione comune che la paginazione si possa aggiungere a qualsiasi sistema è vera per gli ambienti senza paginazione su richiesta, nei quali un'eccezione di page fault rappresenta un errore fatale, ma è falsa nei casi in cui un'eccezione di page fault implica solo la necessità di caricare in memoria un'altra pagina e quindi riavviare il processo.

9.2.2 Prestazioni della paginazione su richiesta

La paginazione su richiesta può avere un effetto rilevante sulle prestazioni di un calcolatore. Il motivo si può comprendere calcolando il **tempo d'accesso effettivo** per una memoria con paginazione su richiesta. Attualmente, nella maggior parte dei calcolatori il tempo d'accesso alla memoria, che si denota *ma*, varia da 10 a 200 nanosecondi. Finché non si verifichino page fault, il tempo d'accesso effettivo è uguale al tempo d'accesso alla memoria. Se però si verifica un page fault, occorre prima leggere dal disco la pagina interessata e quindi accedere alla parola della memoria desiderata.

Supponendo che p sia la probabilità che si verifichi un page fault ($0 \leq p \leq 1$), è probabile che p sia molto vicina allo zero, cioè che ci siano solo pochi fault. Il **tempo d'accesso effettivo** è dato dalla seguente espressione:

$$\text{tempo d'accesso effettivo} = (1 - p) - ma + p \times \text{tempo di gestione del page fault}$$

Per calcolare il tempo d'accesso effettivo occorre conoscere il tempo necessario alla gestione di un page fault. In tal caso si deve eseguire la seguente sequenza:

1. trap per il sistema operativo;
2. salvataggio dei registri utente e dello stato del processo;
3. verifica che l'interruzione sia dovuta o meno a un page fault;
4. controllo della correttezza del riferimento alla pagina e determinazione della locazione della pagina nel disco;
5. lettura dal disco e trasferimento in un frame libero:
 - a) attesa nella coda relativa a questo dispositivo finché la richiesta di lettura non sia servita;
 - b) attesa del tempo di posizionamento e latenza del dispositivo;
 - c) inizio del trasferimento della pagina in un frame libero;
6. durante l'attesa, allocazione della CPU a un altro processo utente (scheduling della CPU, facoltativo);
7. ricezione di un'interruzione dal controllore del disco (I/O completato);
8. salvataggio dei registri e dello stato dell'altro processo utente (se è stato eseguito il passo 6);
9. verifica della provenienza dell'interruzione dal disco;
10. aggiornamento della tabella delle pagine e di altre tabelle per segnalare che la pagina richiesta è attualmente presente in memoria;
11. attesa che la CPU sia nuovamente assegnata a questo processo;
12. ripristino dei registri utente, dello stato del processo e della nuova tabella delle pagine, quindi ripresa dell'istruzione interrotta.

Non sempre sono necessari tutti i passi sopra elencati. Nel passo 6, per esempio, si ipotizza che la CPU sia assegnata a un altro processo durante un'operazione di I/O. Tale possibilità permette la multiprogrammazione per mantenere occupata la CPU, ma una volta completato il trasferimento di I/O implica un dispendio di tempo per riprendere la procedura di servizio dell'eccezione di page fault.

In ogni caso, il tempo di servizio dell'eccezione di page fault ha tre componenti principali:

1. servizio del segnale di eccezione di page fault;
2. lettura della pagina da disco;
3. riavvio del processo.

La prima e la terza operazione si possono ridurre, per mezzo di un'accurata codifica, ad alcune centinaia di istruzioni. Ciascuna di queste operazioni può quindi richiedere da 1 a 100 microsecondi. D'altra parte, il tempo di trasferimento di pagina è probabilmente vicino a 8 millisecondi (un disco ha in genere un tempo di latenza di 3 milisecondi, un tempo di posizionamento di 5 milisecondi e un tempo di trasferimento di 0,05 milisecondi, quindi il tempo totale della paginazione è dell'ordine di 8 milisecondi, comprendendo le tempistiche hardware e software). Inoltre nel calcolo si è considerato solo il tempo di servizio del dispositivo. Se una coda di processi è in attesa del dispositivo è necessario considerare anche il tempo di accodamento del dispositivo, poiché occorre attendere che il dispositivo di paginazione sia libero per servire la richiesta, quindi il tempo di trasferimento aumenta ulteriormente.

Considerando un tempo medio di servizio dell'eccezione di page fault di 8 milisecondi e un tempo d'accesso alla memoria di 200 nanosecondi, il tempo effettivo d'accesso in nanosecondi è il seguente:

$$\begin{aligned}\text{tempo d'accesso effettivo} &= (1 - p) \times 200 + p \text{ (8 milisecondi)} \\ &= (1 - p) \times 200 + p \times 8.000.000 \\ &= 200 + 7.999.800 \times p\end{aligned}$$

Il tempo d'accesso effettivo è direttamente proporzionale al **tasso di page fault** (*page-fault rate*). Se un accesso su 1000 accusa un page fault, il tempo d'accesso effettivo è di 8,2 microsecondi. Impiegando la paginazione su richiesta, il calcolatore è rallentato di un fattore pari a 40! Se si desidera un rallentamento inferiore al 10 per cento, occorre contenere la probabilità di page fault al seguente livello:

$$\begin{aligned}220 &> 200 + 7.999.800 \times p \\ 20 &> 7.999.800 \times p \\ p &< 0,00000025\end{aligned}$$

Quindi, per mantenere a un livello ragionevole il rallentamento dovuto alla paginazione, si può permettere meno di un page fault ogni 399.990 accessi alla memoria. In un sistema con paginazione su richiesta, è cioè importante tenere basso il tasso di page fault, altrimenti il tempo effettivo d'accesso aumenta, rallentando molto l'esecuzione del processo.

Un altro aspetto della paginazione su richiesta è la gestione e l'uso generale dell'area di swap. L'I/O di un disco relativo all'area di swap è generalmente più rapido di quello relativo al file system (Capitolo 10): ciò si deve al fatto che lo spazio di swap è allocato in blocchi molto grandi e non vengono utilizzate ricerche e riferimenti indiretti. Perciò il sistema può migliorare l'efficienza della paginazione copiando tutta l'immagine di un file nell'area di swap all'avvio del processo e di lì eseguire la paginazione su richiesta. Un'altra possibilità consiste nel richiedere inizialmente le pagine al file system, ma scrivere le pagine nell'area di swap al momento della sostituzione. Questo metodo assicura che si leggano sempre dal file system solo le pagine necessarie, ma che tutta la paginazione successiva sia fatta dall'area di swap.

Alcuni sistemi tentano di limitare l'area di swap utilizzata per file binari: le pagine richieste per questi file si prelevano direttamente dal file system; tuttavia, quando è

richiesta una sostituzione di pagine, i frame possono semplicemente essere sovrascritti, dato che non sono mai stati modificati, e le pagine, se è necessario, possono essere nuovamente lette dal file system. Seguendo questo criterio, lo stesso file system funziona da memoria ausiliaria (*backing store*). L'area di swap si deve in ogni caso usare per le pagine che non sono relative ai file (la cosiddetta memoria *anonima*); queste comprendono lo stack e lo heap di un processo. Questa tecnica che sembra essere un buon compromesso si usa in diversi sistemi tra cui Solaris e UNIX BSD.

I sistemi operativi mobili non supportano, in genere, lo swapping, ma, in caso di carenza di memoria, richiedono pagine al file system e recuperano pagine di sola lettura (come il codice) dalle applicazioni. Se necessario, questi dati possono essere richiesti di nuovo al file system. In iOS non vengono mai riprese a un'applicazione le pagine di memoria anonima a meno che l'applicazione sia terminata o abbia rilasciato la memoria in maniera volontaria.

9.3 Copiatura su scrittura

Nel Paragrafo 9.2 si è visto come un processo possa cominciare rapidamente l'esecuzione richiedendo solo la pagina contenente la prima istruzione. La generazione dei processi tramite `fork()`, però, può inizialmente evitare la paginazione su richiesta per mezzo di una tecnica simile alla condivisione delle pagine (Paragrafo 8.5.4), che garantisce la celere generazione dei processi riuscendo anche a minimizzare il numero di nuove pagine allocate al processo appena creato.

Si ricordi che la chiamata di sistema `fork()` crea un processo figlio come duplicato del genitore. Nella sua versione originale la `fork()` creava per il figlio una copia dello spazio d'indirizzi del genitore, duplicando le pagine appartenenti al processo genitore. Considerando che molti processi figli eseguono subito dopo la loro creazione la chiamata di sistema `exec()`, questa operazione di copiatura può essere inutile. Come alternativa, si può impiegare una tecnica nota come **copiatura su scrittura** (*copy-on write*), il cui funzionamento si fonda sulla condivisione iniziale delle pagine da parte dei processi genitori e dei processi figli. Le pagine condivise si contrassegnano come pagine da copiare su scrittura, a significare che, se un processo (genitore o figlio) scrive su una pagina condivisa, il sistema deve creare una copia di tale pagina. La copia su scrittura è illustrata nella Figura 9.7 e nella Figura 9.8, che mostrano il contenuto della memoria fisica prima e dopo che il processo 1 abbia modificato la pagina C.

Si consideri per esempio un processo figlio che cerchi di modificare una pagina contenente parti della stack, quando le pagine sono contrassegnate come copy-on write. Il sistema operativo crea una copia della pagina nello spazio degli indirizzi del processo figlio. Il processo figlio modifica la sua copia della pagina e non la pagina appartenente al processo genitore. È chiaro che, adoperando la tecnica di copiatura su scrittura, si copiano soltanto le pagine modificate da uno dei due processi, mentre tutte le altre sono condivisibili dai processi genitore e figlio. Si noti inoltre che soltanto le pagine modificabili si devono contrassegnare come copy-on write, mentre quelle che non si possono modificare (per esempio, le pagine contenenti codice ese-

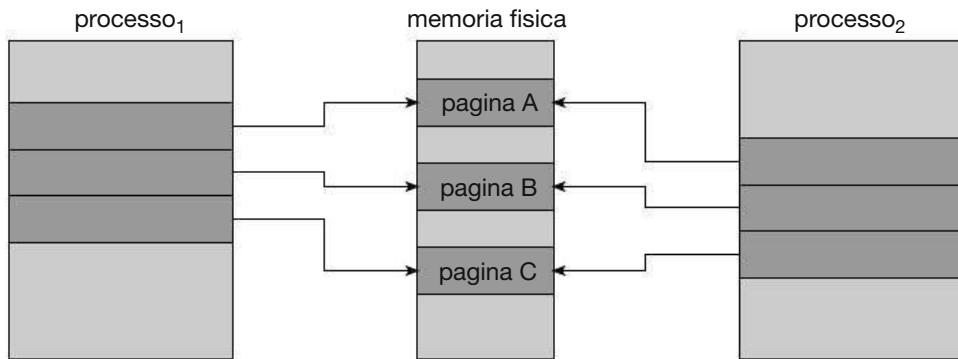


Figura 9.7 Prima della modifica alla pagina C da parte del processo 1.

guibile) sono condivisibili dai processi genitore e figlio. La tecnica di copiatura su scrittura è piuttosto comune e si usa in diversi sistemi operativi, tra i quali Windows XP, Linux e Solaris.

Quando è necessaria la duplicazione di una pagina secondo la tecnica di copiatura su scrittura, è importante capire da dove si attingerà la pagina libera necessaria. Molti sistemi operativi forniscono, per queste richieste, un gruppo (*pool*) di pagine libere, che di solito si assegnano quando lo stack o lo heap di un processo devono espandersi, oppure proprio per gestire pagine da copiare su scrittura. L'allocazione di queste pagine di solito avviene secondo una tecnica nota come **azzeramento su richiesta** (*zero-fill-on-demand*); prima dell'allocazione si riempiono di zeri le pagine, cancellandone in questo modo tutto il contenuto precedente.

Diverse versioni di UNIX (compreso Solaris e Linux) offrono anche una variante della chiamata di sistema `fork()` – detta `vfork()` (per *virtual memory fork*). La `vfork()` offre un'alternativa all'uso della `fork()` con copiatura su scrittura. Con la `vfork()` il processo genitore viene sospeso e il processo figlio usa lo spazio d'indirizzi del genitore. Poiché la `vfork()` non usa la copiatura su scrittura, se il processo figlio modifica qualche pagina dello spazio d'indirizzi del genitore, le pagine modi-

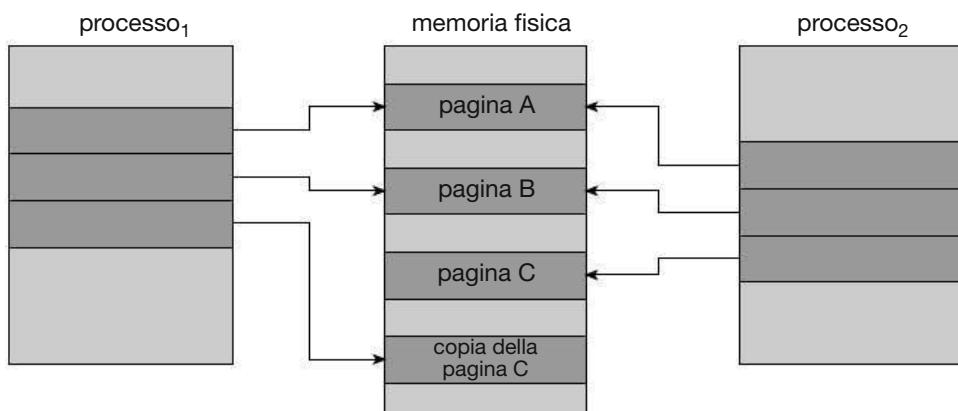


Figura 9.8 Dopo la modifica alla pagina C da parte del processo 1.

ficate saranno visibili al processo genitore non appena riprenderà il controllo. È quindi necessaria molta attenzione nell'uso di `vfork()`, per assicurarsi che il processo figlio non modifichi lo spazio d'indirizzi del genitore. La chiamata di sistema `vfork()` è adatta al caso in cui il processo figlio esegua una `exec()` immediatamente dopo la sua creazione. Poiché non richiede alcuna copiatura delle pagine, la `vfork()` è un metodo di creazione dei processi molto efficiente, in alcuni casi impiegato per realizzare le interfacce shell in UNIX.

9.4 Sostituzione delle pagine

Nelle descrizioni fatte finora sul tasso di page fault abbiamo supposto che ogni pagina poteva dar luogo al massimo a un fault, la prima volta in cui si effettuava un riferimento a essa. Tale rappresentazione tuttavia non è molto precisa. Se un processo di 10 pagine ne impiega effettivamente solo la metà, la paginazione su richiesta fa risparmiare l'I/O necessario per caricare le cinque pagine che non sono mai usate. Inoltre il grado di multiprogrammazione potrebbe essere aumentato eseguendo il doppio dei processi. Quindi, disponendo di 40 frame, si potrebbero eseguire otto processi anziché i quattro che si eseguirebbero se ciascuno di loro richiedesse 10 blocchi di memoria, cinque dei quali non sarebbero mai usati.

Aumentando il grado di multiprogrammazione, si *sovrassegna* la memoria. Eseguendo sei processi, ciascuno dei quali è formato da 10 pagine, di cui solo cinque sono effettivamente usate, s'incrementerebbero l'utilizzo e la produttività della CPU e si avrebbero ancora 10 frame disponibili. Tuttavia è possibile che ciascuno di questi processi, per un insieme particolare di dati, abbia improvvisamente necessità di impiegare tutte le 10 pagine, perciò sarebbero necessari 60 frame, mentre ne sono disponibili solo 40.

Si consideri inoltre che la memoria del sistema non si usa solo per contenere pagine di programmi: le aree di memoria per l'I/O impegnano una rilevante quantità di memoria. Ciò può aumentare le difficoltà agli algoritmi di allocazione della memoria. Decidere quanta memoria assegnare all'I/O e quanta alle pagine dei programmi è un problema complesso. Alcuni sistemi riservano una quota fissa di memoria per l'I/O, altri permettono sia ai processi utenti sia al sottosistema di I/O di competere per tutta la memoria del sistema.

La **sovrallocazione** (*over-allocation*) si può illustrare come segue. Durante l'esecuzione di un processo utente si verifica un page fault. Il sistema operativo determina la locazione del disco in cui risiede la pagina desiderata, ma poi scopre che la lista dei frame liberi è *vuota*: tutta la memoria è in uso (Figura 9.9).

A questo punto il sistema operativo può scegliere tra diverse possibilità, per esempio può terminare il processo utente. Tuttavia, la paginazione su richiesta è un tentativo che il sistema operativo fa per migliorare l'utilizzo e la produttività del sistema di calcolo. Gli utenti non dovrebbero sapere che i loro processi sono eseguiti su un sistema paginato. La paginazione deve essere logicamente trasparente per l'utente, quindi la terminazione del processo non costituisce la scelta migliore.

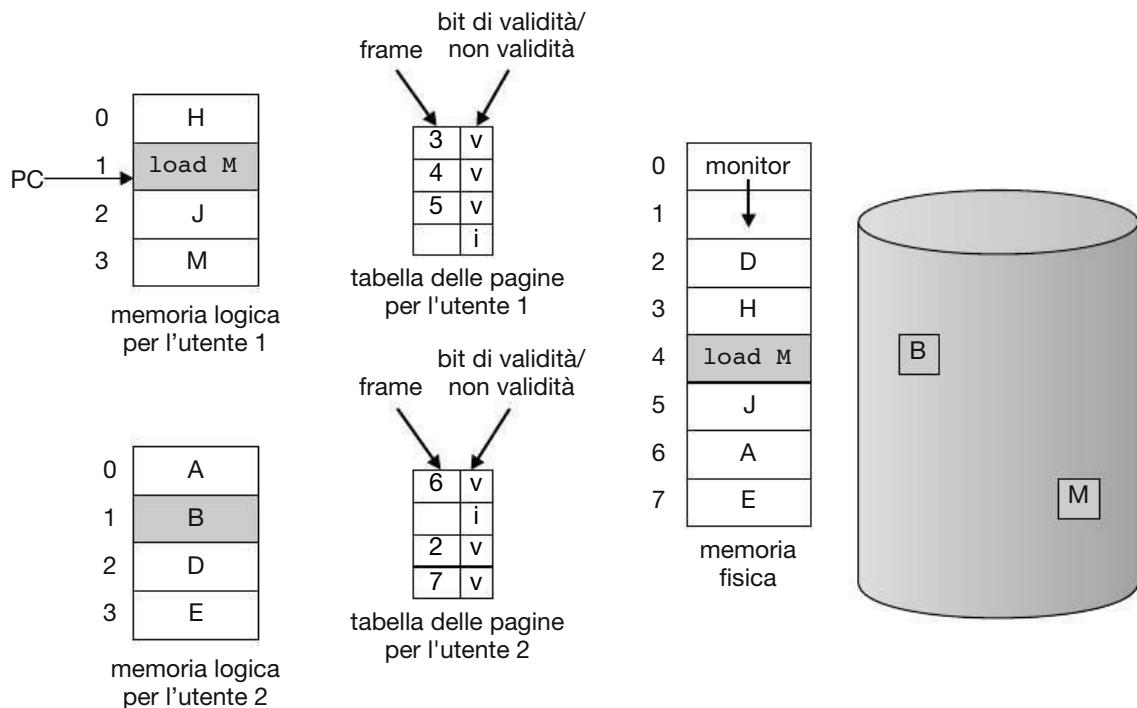


Figura 9.9 Necessità di sostituzione di pagine.

Il sistema operativo può scaricare dalla memoria un intero processo, liberando tutti i suoi frame e riducendo il livello di multiprogrammazione. Questa possibilità, buona in certe situazioni, è considerata nel Paragrafo 9.6. In questa sede analizziamo l'opzione più comune: la **sostituzione delle pagine** (*page replacement*).

9.4.1 Sostituzione di pagina

La sostituzione delle pagine segue il seguente criterio: se nessun frame è libero, ne viene liberato uno attualmente inutilizzato. È possibile liberarlo scrivendo il suo contenuto nell'area di swap e modificando la tabella delle pagine (e tutte le altre tabelle) per indicare che la pagina non si trova più in memoria (Figura 9.10). Il frame liberato si può usare per memorizzare la pagina che ha causato il fault. Si modifica la procedura di servizio dell'eccezione di page fault in modo da includere la sostituzione della pagina:

1. s'individua la locazione su disco della pagina richiesta;
2. si cerca un frame libero:
 - a. se esiste, lo si usa;
 - b. altrimenti si impiega un algoritmo di sostituzione delle pagine per scegliere un **frame vittima**;
 - c. si scrive la pagina “vittima” nel disco; si di conseguenza le tabelle delle pagine e quelle dei frame;

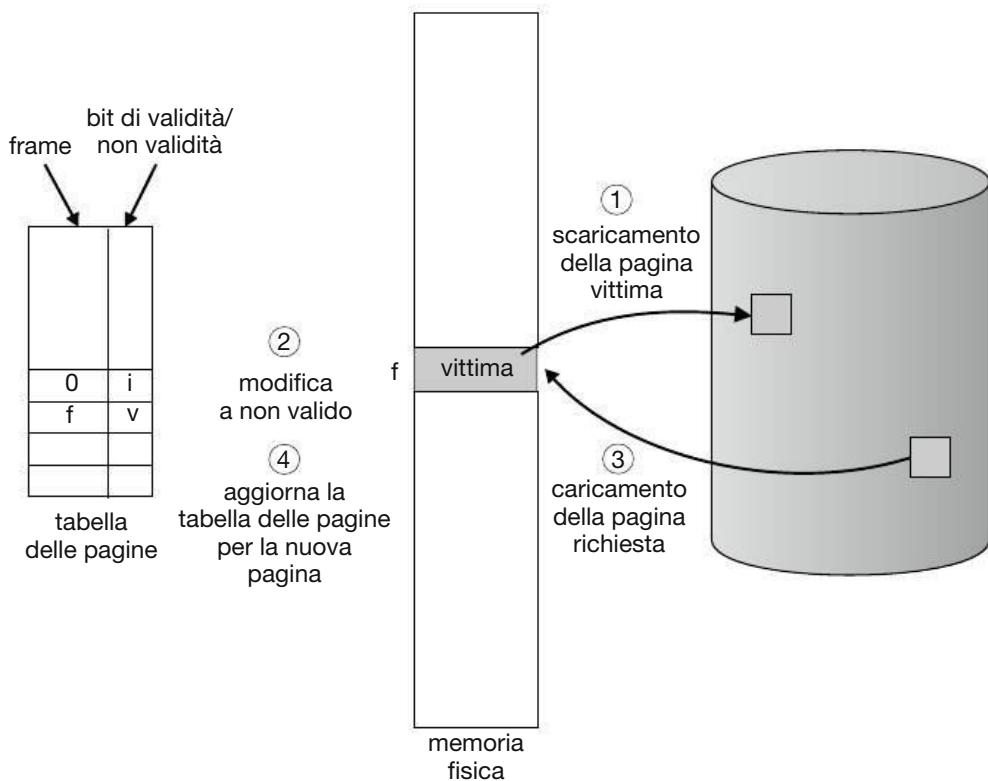


Figura 9.10 Sostituzione di una pagina.

3. si scrive la pagina richiesta nel frame appena liberato; si modificano le tabelle delle pagine e dei frame;
4. si riprende il processo utente dal punto in cui si è verificato il page fault.

Occorre notare che, se non esiste alcun frame libero sono necessari *due* trasferimenti di pagine, uno fuori e uno dentro la memoria. Questa situazione raddoppia il tempo di servizio del page fault e aumenta di conseguenza anche il tempo effettivo d’accesso.

Questo sovraccarico si può ridurre usando un **bit di modifica** (*modify bit* o *dirty bit*). In questo caso l’hardware del calcolatore dispone di un bit di modifica, associato a ogni pagina (o frame), che viene posto a 1 ogni volta che nella pagina si scrive un byte, indicando che la pagina è stata modificata. Quando si sceglie una pagina da sostituire si esamina il suo bit di modifica; se è a 1, significa che quella pagina è stata modificata rispetto a quando era stata letta dal disco; in questo caso la pagina deve essere scritta nel disco. Se il bit di modifica è rimasto a 0, significa che la pagina *non* è stata modificata da quando è stata caricata in memoria, quindi non è necessario scrivere nel disco la pagina di memoria: c’è già. Questa tecnica vale anche per le pagine di sola lettura, per esempio pagine di codice binario. Queste pagine non possono essere modificate, quindi si possono rimuovere in ogni momento. Questo schema può ridurre in modo considerevole il tempo per il servizio del page fault, poiché dimezza il tempo di I/O, se la pagina non è stata modificata.

La sostituzione di una pagina è fondamentale al fine della paginazione su richiesta, perché completa la separazione tra memoria logica e memoria fisica. Con questo meccanismo si può mettere a disposizione dei programmati una memoria virtuale enorme con una memoria fisica più piccola. Senza la paginazione su richiesta, gli indirizzi utente si fanno corrispondere a indirizzi fisici e i due insiemi di indirizzi possono essere diversi. Tuttavia tutte le pagine di un processo devono ancora essere in memoria fisica. Con la paginazione su richiesta la dimensione dello spazio degli indirizzi logici non è più limitata dalla memoria fisica. Per esempio, un processo utente formato da 20 pagine si può eseguire in 10 frame semplicemente usando la paginazione su richiesta e un algoritmo di sostituzione per localizzare un frame libero ogni volta sia necessario. Se una pagina modificata deve essere sostituita, si copia nel disco il suo contenuto. Un successivo riferimento a quella pagina causa un'eccezione di page fault. In quel momento, la pagina viene riportata in memoria, eventualmente sostituendo un'altra pagina.

Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali: occorre sviluppare un **algoritmo di allocazione dei frame** e un **algoritmo di sostituzione delle pagine**. Ossia, se sono presenti più processi in memoria, occorre decidere quanti frame vadano assegnati a ciascun processo. Inoltre, quando è richiesta una sostituzione di pagina, occorre selezionare i frame da sostituire. La progettazione di algoritmi idonei a risolvere questi problemi è un compito importante, poiché l'I/O nei dischi è molto oneroso. Anche miglioramenti minimi ai metodi di paginazione su richiesta apportano notevoli incrementi alle prestazioni del sistema.

Esistono molti algoritmi di sostituzione delle pagine; probabilmente ogni sistema operativo ha il proprio schema di sostituzione. È quindi necessario stabilire un criterio per selezionare un algoritmo di sostituzione particolare; comunemente si sceglie quello con il minimo tasso di page fault.

Un algoritmo si valuta effettuandone l'esecuzione su una particolare successione di riferimenti alla memoria e calcolando il numero di page fault. La successione dei riferimenti alla memoria è detta, appunto, **successione dei riferimenti**. Queste successioni si possono generare artificialmente (per esempio con un generatore di numeri casuali), oppure analizzando un dato sistema e registrando l'indirizzo di ciascun riferimento alla memoria. Quest'ultima opzione genera un numero elevato di dati, dell'ordine di un milione di indirizzi al secondo. Per ridurre questa quantità di dati occorre notare due fatti.

Innanzitutto, per una pagina di dimensioni date, generalmente fissate dall'architettura del sistema, si considera solo il numero della pagina anziché l'intero indirizzo. In secondo luogo, se si ha un riferimento a una pagina p , i riferimenti alla stessa pagina *immediatamente* successivi al primo non generano page fault: dopo il primo riferimento, la pagina p è presente in memoria.

Esaminando un processo si potrebbe per esempio registrare la seguente successione di indirizzi:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

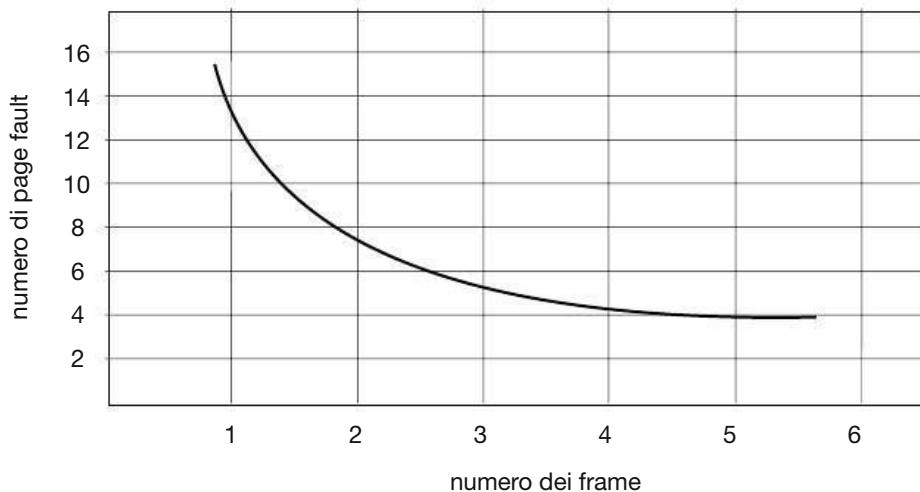


Figura 9.11 Grafico che illustra il numero di page fault rispetto al numero dei frame.

che, a 100 byte per pagina, si riduce alla seguente successione di riferimenti:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Per stabilire il numero di page fault relativo a una particolare successione di riferimenti e a un particolare algoritmo di sostituzione delle pagine, occorre conoscere anche il numero dei frame disponibili. Naturalmente, aumentando il numero di quest'ultimi diminuisce il numero di page fault. Per la successione dei riferimenti precedentemente esaminata, per esempio, dati tre o più blocchi di memoria avremmo solo tre fault: uno per il primo riferimento di ogni pagina. D'altra parte, se si dispone di un solo frame è necessaria una sostituzione per ogni riferimento, con il risultato di 11 page fault. In generale ci si aspetta una curva simile a quella della Figura 9.11. Aumentando il numero dei frame, il numero di page fault diminuisce fino al livello minimo. Naturalmente aggiungendo memoria fisica il numero dei frame aumenta.

Per illustrare gli algoritmi di sostituzione delle pagine impiegheremo la seguente successione di riferimenti

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

per una memoria con tre frame.

9.4.2 Sostituzione delle pagine secondo l'ordine d'arrivo (FIFO)

L'algoritmo di sostituzione delle pagine più semplice è un algoritmo FIFO. Questo algoritmo associa a ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo. Occorre notare che non è strettamente necessario registrare l'istante in

successione dei riferimenti

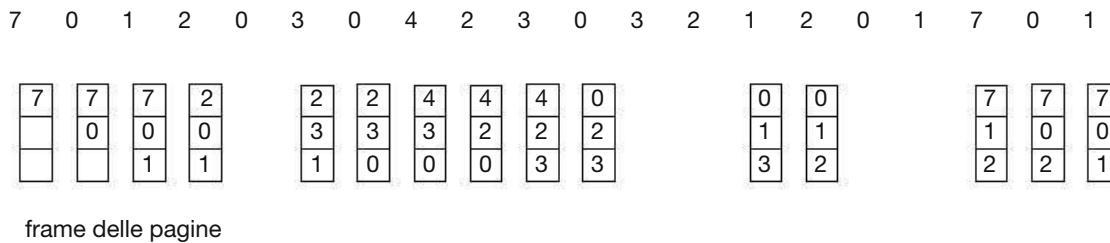


Figura 9.12 Algoritmo di sostituzione delle pagine FIFO.

cui si carica una pagina in memoria; infatti si può creare una coda FIFO di tutte le pagine presenti in memoria. In questo caso si sostituisce la pagina che si trova in testa alla coda. Quando si carica una pagina in memoria, la si inserisce nell'ultimo elemento della coda.

Nella successione di riferimenti di esempio, i nostri tre frame sono inizialmente vuoti. I primi tre riferimenti (7, 0, 1) accusano ciascuno un page fault con conseguente caricamento delle relative pagine nei frame vuoti. Il riferimento successivo (2) causa la sostituzione della pagina 7, perché essa è stata caricata per prima in memoria. Siccome 0 è il riferimento successivo e si trova già in memoria, per questo riferimento non ha luogo alcun page fault. Il primo riferimento a 3 causa la sostituzione della pagina 0, che ora è la prima pagina in coda. A causa di questa sostituzione il riferimento successivo, a 0, causerà un page fault. La pagina 1 è poi sostituita dalla pagina 0. Questo processo prosegue come è illustrato nella Figura 9.12. Ogni volta che si verifica un page fault sono indicate le pagine presenti nei tre frame. Complessivamente si hanno 15 page fault.

L'algoritmo FIFO di sostituzione delle pagine è facile da capire e da programmare; tuttavia la sue prestazioni non sono sempre buone. La pagina sostituita potrebbe essere un modulo di inizializzazione usato molto tempo prima e che non serve più, ma potrebbe anche contenere una variabile molto usata, inizializzata precedentemente, e ancora in uso.

Occorre notare che anche se si sceglie una pagina da sostituire che è in uso attivo, tutto continua a funzionare correttamente. Dopo aver rimosso una pagina attiva per inserirne una nuova, quasi immediatamente si verifica un'eccezione di page fault per riprendere la pagina attiva. Per riportare la pagina attiva in memoria è necessario sostituire un'altra pagina. Quindi, una cattiva scelta della pagina da sostituire aumenta il tasso di page fault e rallenta l'esecuzione del processo, ma non causa errori.

Per illustrare i problemi che possono insorgere con l'uso dell'algoritmo di sostituzione delle pagine FIFO, si consideri la seguente successione di riferimenti:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Nella Figura 9.13 è illustrata la curva dei page fault per questa successione di riferimenti in funzione del numero dei frame disponibili. Occorre notare che il numero dei

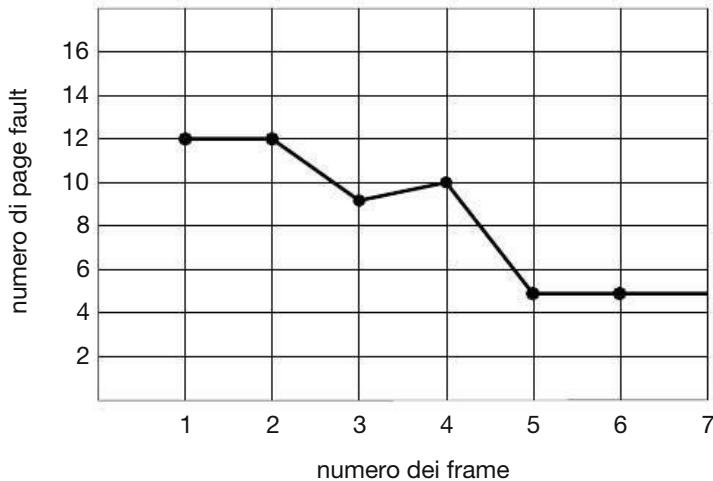


Figura 9.13 Curva dei page fault per la sostituzione FIFO su una successione di riferimenti.

page fault (10) per quattro frame è *maggior*e del numero dei page fault (9) per tre frame. Questo inatteso risultato è noto col nome di **anomalia di Belady**: con alcuni algoritmi di sostituzione delle pagine, il tasso di page fault può *aumentare* con l'incremento del numero dei frame assegnati. A prima vista sembra logico supporre che fornendo più memoria a un processo le prestazioni di quest'ultimo migliorino. In alcune delle prime ricerche sperimentali si notò invece che questo presupposto non sempre è vero; venne così individuata l'anomalia di Belady.

9.4.3 Sostituzione ottimale delle pagine

In seguito alla scoperta dell'anomalia di Belady si è ricercato un **algoritmo ottimale di sostituzione delle pagine**. Tale algoritmo è quello che fra tutti gli algoritmi presenta il tasso minimo di page fault e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato OPT o MIN. Consiste semplicemente nel:

sostituire la pagina che non verrà usata per il periodo di tempo più lungo.

L'uso di quest'algoritmo di sostituzione delle pagine assicura il tasso minimo di page fault per un dato numero di frame.

Per esempio, nella successione dei riferimenti d'esempio, l'algoritmo ottimale di sostituzione delle pagine produce nove page fault, come è mostrato nella Figura 9.14. I primi tre riferimenti causano page fault che riempiono i tre blocchi di memoria vuoti. Il riferimento alla pagina 2 determina la sostituzione della pagina 7, perché la pagina 7 non è usata fino al riferimento 18, mentre la pagina 0 viene usata al 5 e la pagina 1 al 14. Il riferimento alla pagina 3 causa la sostituzione della pagina 1, poiché la pagina 1 è l'ultima delle tre pagine in memoria cui si fa nuovamente riferimento. Con sole nove page fault, la sostituzione ottimale risulta assai migliore di quella ottenuta con un algoritmo FIFO, dove i page fault erano 15. Ignorando i primi tre page fault, che si verificano con tutti gli algoritmi, la sostituzione ottimale è due volte migliore

successione dei riferimenti

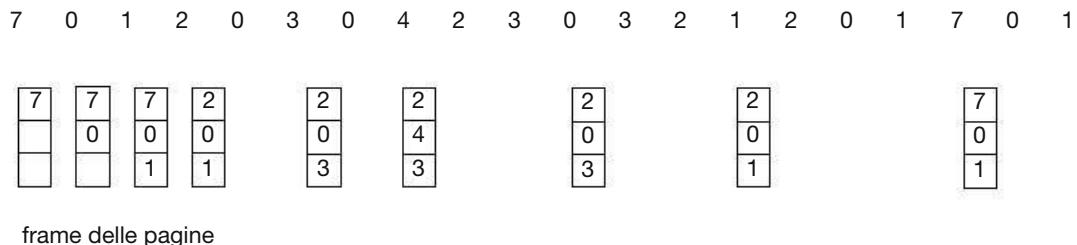


Figura 9.14 Algoritmo ottimale di sostituzione delle pagine.

rispetto all'algoritmo FIFO; nessun algoritmo di sostituzione può gestire questa successione di riferimenti a tre blocchi di memoria con meno di nove page fault.

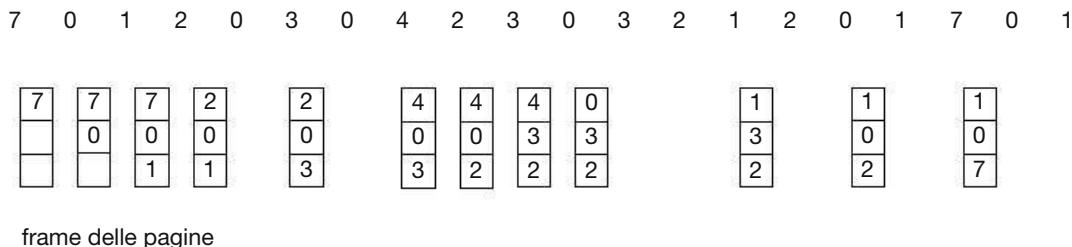
Sfortunatamente l'algoritmo ottimale di sostituzione delle pagine è difficile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti (una situazione analoga si è riscontrata con l'algoritmo SJF di scheduling della CPU, nel Paragrafo 6.3.2). Quindi, l'algoritmo ottimale si impiega soprattutto per studi comparativi. Per esempio, può risultare abbastanza utile sapere che, sebbene un algoritmo nuovo non sia ottimale, nel peggiore dei casi le sue prestazioni sono inferiori del 12,3 per cento rispetto a quelle dell'algoritmo ottimale, e mediamente questa percentuale è del 4,7 per cento.

9.4.4 Sostituzione delle pagine usate meno recentemente (LRU)

Se l'algoritmo ottimale non è realizzabile, è forse possibile realizzarne un'approssimazione. La distinzione fondamentale tra gli algoritmi FIFO e OPT, oltre quella di guardare avanti o indietro nel tempo, consiste nel fatto che l'algoritmo FIFO impiega l'istante in cui una pagina è stata caricata in memoria, mentre l'algoritmo OPT impiega l'istante in cui una pagina sarà *usata*. Usando come approssimazione di un futuro vicino un passato recente, si sostituisce la pagina che *non è stata usata* per il periodo più lungo. Il metodo appena descritto è noto come **algoritmo LRU** (*least recently used*).

La sostituzione LRU associa a ogni pagina l'istante in cui è stata usata per l'ultima volta. Quando occorre sostituire una pagina, l'algoritmo LRU sceglie quella che non è stata usata per il periodo più lungo. Possiamo interpretare questa strategia come l'algoritmo ottimale di sostituzione delle pagine con ricerca all'indietro nel tempo, anziché in avanti. Un risultato interessante per l'analogia dei due algoritmi è il seguente. Supponendo che S^R sia la successione inversa di una successione di riferimenti S , il tasso di page fault per l'algoritmo OPT su S è uguale a quello su S^R . Allo stesso modo, il tasso di page fault per l'algoritmo LRU su S è uguale a quella su S^R .

successione dei riferimenti



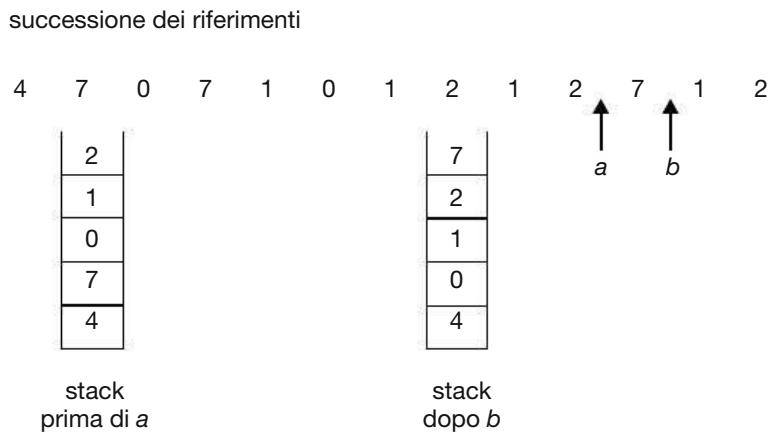
frame delle pagine

Figura 9.15 Algoritmo di sostituzione delle pagine LRU.

Il risultato dell'applicazione dell'algoritmo LRU alla successione dei riferimenti dell'esempio è illustrato nella Figura 9.15. L'algoritmo LRU produce 12 page fault. Occorre notare che i primi cinque page fault sono gli stessi della sostituzione ottimale. Quando si presenta il riferimento alla pagina 4, però, l'algoritmo LRU trova che, fra i tre blocchi di memoria, quello usato meno recentemente è della pagina 2. Quindi, l'algoritmo LRU sostituisce la pagina 2 senza sapere che sta per essere usata. Quando si verifica il fault della pagina 2, l'algoritmo LRU sostituisce la pagina 3, poiché, fra le tre pagine in memoria (0, 3, 4), la pagina 3 è quella usata meno recentemente. Nonostante questi problemi, la sostituzione LRU, con 12 page fault, è molto migliore della sostituzione FIFO, con 15 page fault.

Il criterio LRU si usa spesso come algoritmo di sostituzione delle pagine ed è considerato valido. Il problema principale riguarda la sua implementazione. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza da parte dell'hardware. Il problema consiste nel determinare un ordine per i frame definito dal momento dell'ultimo uso. Si possono realizzare le due seguenti soluzioni.

- **Contatori.** Nel caso più semplice, a ogni elemento della tabella delle pagine si associa un campo *momento di utilizzo*, e alla CPU si aggiunge un contatore che si incrementa a ogni riferimento alla memoria. Ogni volta che si fa un riferimento a una pagina, si copia il contenuto del registro contatore nel campo *momento di utilizzo* nella voce della page table relativa a quella pagina. In questo modo è sempre possibile conoscere il momento in cui è stato fatto l'ultimo riferimento a ogni pagina. Si sostituisce la pagina con il valore associato più piccolo. Questo schema implica una ricerca all'interno della tabella delle pagine per individuare la pagina usata meno recentemente (LRU), e una scrittura in memoria (nel campo *momento di utilizzo* della tabella delle pagine) per ogni accesso alla memoria. I riferimenti temporali si devono mantenere anche quando, a seguito dello scheduling della CPU, si modificano le tabelle delle pagine. Occorre infine considerare l'overflow del contatore.
- **Stack.** Un altro metodo per la realizzazione della sostituzione delle pagine LRU prevede l'utilizzo di uno stack dei numeri delle pagine. Ogni volta che si fa un riferimento a una pagina, la si estrae dallo stack e la si colloca in cima a quest'ulti-



9.4.5 Sostituzione delle pagine per approssimazione a LRU

Sono pochi i sistemi di calcolo che dispongono del supporto hardware per una vera sostituzione LRU delle pagine. Nei sistemi che non offrono alcun supporto hardware si devono impiegare altri algoritmi di sostituzione delle pagine, per esempio l'algoritmo FIFO. Molti sistemi tuttavia possono fornire un aiuto: un **bit di riferimento**. Il bit di riferimento a una pagina è impostato automaticamente dall'hardware del sistema ogni volta che si fa un riferimento a quella pagina, che sia una lettura o una scrittura su qualsiasi byte della pagina. I bit di riferimento sono associati a ciascun elemento della tabella delle pagine.

Inizialmente, il sistema operativo azzera tutti i bit. Quando s'inizia l'esecuzione di un processo utente, l'hardware imposta a 1 il bit associato a ciascuna pagina cui si fa riferimento. Dopo qualche tempo è possibile stabilire quali pagine sono state usate semplicemente esaminando i bit di riferimento. Non è però possibile conoscere l'*ordine d'uso*. Questa informazione è alla base di molti algoritmi per la sostituzione delle pagine che approssimano LRU.

9.4.5.1 Algoritmo con bit supplementari di riferimento

Ulteriori informazioni sull'ordinamento si possono ottenere registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella in memoria con, ad esempio, un byte per ogni pagina. A intervalli regolari, per esempio di 100 millisecondi, un segnale d'interruzione del timer trasferisce il controllo al sistema operativo. Questo inserisce il bit di riferimento per ciascuna pagina nel bit più significativo del byte, shiftando gli altri bit a destra di 1 bit e scartando il bit meno significativo. Questi registri a scorrimento di 8 bit contengono la storia dell'utilizzo delle pagine relativo agli ultimi otto periodi di tempo. Se il registro a scorrimento contiene la successione di bit 00000000, significa che la pagina associata non è stata usata da otto periodi di tempo; a una pagina usata almeno una volta per ogni periodo corrisponde la successione 11111111 nel registro a scorrimento. Una pagina cui corrisponde la successione 11000100, è stata usata più recentemente di una pagina a cui corrisponde 01110111. Interpretando queste successioni di bit come interi senza segno, la pagina cui è associato il numero minore è la pagina LRU, e può essere sostituita. Si noti che l'unicità dei numeri non è garantita. Si possono sostituire tutte le pagine con il valore minore, oppure si può ricorrere a una selezione FIFO.

Il numero dei bit può ovviamente essere variato: si sceglie secondo l'hardware disponibile per accelerarne al massimo la modifica. Nel caso limite tale numero si riduce a zero, lasciando soltanto il bit di riferimento. In questo caso l'algoritmo è noto come **algoritmo di sostituzione delle pagine con seconda chance**.

9.4.5.2 Algoritmo con seconda chance

L'algoritmo di base per la sostituzione con seconda chance è un algoritmo di sostituzione di tipo FIFO. Tuttavia, dopo aver selezionato una pagina, si controlla il bit di riferimento: se il suo valore è 0, si sostituisce la pagina; se il bit di riferimento è impostato a 1, si dà una seconda chance alla pagina e si passa alla successiva pagina FIFO. Quando una pagina riceve la seconda chance, si azzera il suo bit di riferimento e si

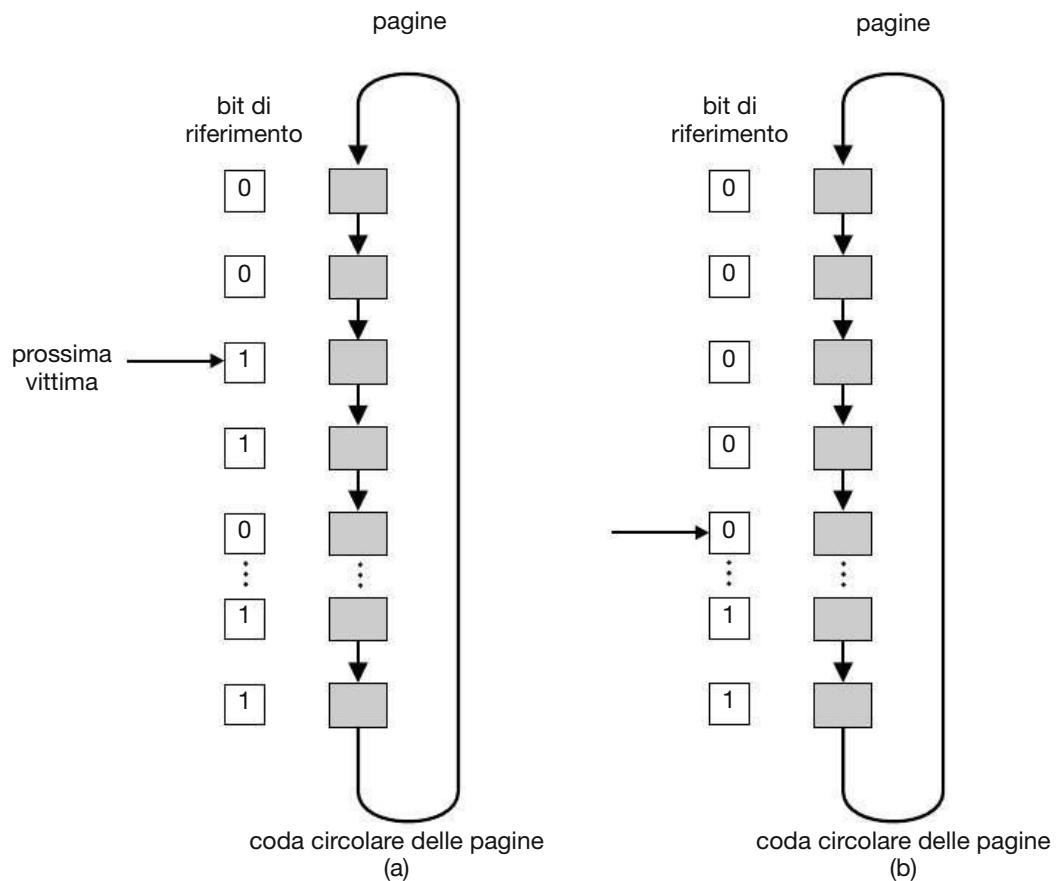


Figura 9.17 Algoritmo di sostituzione delle pagine con seconda chance (orologio).

aggiorna il suo istante d’arrivo al momento attuale. In questo modo, una pagina cui si offre una seconda chance non viene mai sostituita finché tutte le altre pagine non siano state sostituite, oppure non sia stata data loro una seconda chance. Inoltre, se una pagina è usata abbastanza spesso, da mantenere il suo bit di riferimento impostato a 1, non viene mai sostituita.

Un metodo per implementare l’algoritmo con seconda chance, detto anche a orologio (*clock*), è basato sull’uso di una coda circolare, in cui un puntatore (lancetta) indica qual è la prima pagina da sostituire. Quando serve un frame, si fa avanzare il puntatore finché non si trovi in corrispondenza di una pagina con il bit di riferimento 0; a ogni passo si azzera il bit di riferimento appena esaminato (Figura 9.17). Una volta trovata una pagina “vittima”, la si sostituisce e si inserisce la nuova pagina nella coda circolare nella posizione corrispondente. Si noti che nel caso peggiore, quando tutti i bit sono impostati a 1, il puntatore percorre un ciclo su tutta la coda, dando a ogni pagina una seconda chance. Prima di selezionare la pagina da sostituire, azzera tutti i bit di riferimento. Se tutti i bit sono a 1, la sostituzione con seconda chance si riduce a una sostituzione FIFO.

9.4.5.3 Algoritmo con seconda chance migliorato

L'algoritmo con seconda chance descritto precedentemente si può migliorare considerando i bit di riferimento e di modifica (descritto nel Paragrafo 9.4.1) come una coppia ordinata, con cui si possono ottenere le seguenti quattro classi:

1. (0, 0) né recentemente usato né modificato – migliore pagina da sostituire;
2. (0, 1) non usato recentemente, ma modificato – la pagina non così buona poiché prima di essere sostituita deve essere scritta in memoria secondaria;
3. (1, 0) usato recentemente ma non modificato – probabilmente la pagina sarà presto usata nuovamente;
4. (1, 1) usato recentemente e modificato – probabilmente la pagina sarà presto ancora usata e dovrà essere scritta in memoria secondaria prima di essere sostituita.

Ogni pagina rientra in una di queste quattro classi. Alla richiesta di una sostituzione di pagina, si usa lo stesso schema impiegato nell'algoritmo a orologio, ma anziché controllare se la pagina puntata ha il bit di riferimento impostato a 1, si esamina la classe a cui la pagina appartiene e si sostituisce la prima pagina che si trova nella classe minima non vuota. Si noti che si può dover scandire la coda circolare più volte prima di trovare una pagina da sostituire.

La differenza principale tra questo algoritmo e il più semplice algoritmo a orologio è che qui si dà la preferenza alle pagine modificate, al fine di ridurre il numero di I/O richiesti.

9.4.6 Sostituzione delle pagine basata su conteggio

Esistono molti altri algoritmi che si possono usare per la sostituzione delle pagine. Per esempio, si potrebbe usare un contatore del numero dei riferimenti fatti a ciascuna pagina, e sviluppare i due seguenti schemi.

- **Algoritmo di sostituzione delle pagine meno frequentemente usate** (*least frequently used, LFU*); richiede che si sostituisca la pagina con il conteggio più basso. La ragione di questa scelta è che una pagina usata attivamente deve avere un conteggio di riferimento alto. Si ha però un problema quando una pagina è usata molto intensamente durante la fase iniziale di un processo, ma poi non viene più usata. Poiché è stata usata intensamente il suo conteggio è alto, quindi rimane in memoria anche se non è più necessaria. Una soluzione può essere quella di spostare i valori dei contatori a destra di un bit a intervalli regolari, misurando l'utilizzo con un peso esponenziale decrescente.
- **Algoritmo di sostituzione delle pagine più frequentemente usate** (*most frequently used, MFU*); è basato sul fatto che, probabilmente, la pagina con il contatore più basso è stata appena inserita e non è stata ancora usata.

Le sostituzioni MFU e LFU non sono molto comuni, poiché la realizzazione di questi algoritmi è abbastanza onerosa; inoltre, tali algoritmi non approssimano bene la sostituzione OPT.

9.4.7 Algoritmi con buffering delle pagine

Oltre a uno specifico algoritmo per la sostituzione delle pagine, si usano spesso anche altre procedure; per esempio, i sistemi hanno generalmente un gruppo di frame liberi (*pool of free frames*). Quando si verifica un page fault, si sceglie innanzi tutto un frame vittima, ma prima che la vittima sia scritta in memoria secondaria, si trasferisce la pagina richiesta in un frame libero del gruppo. Questa procedura permette al processo di ricominciare al più presto, senza attendere che la pagina vittima sia scritta in memoria secondaria. Quando nel seguito si scrive la vittima in memoria secondaria, si aggiunge il suo frame al gruppo dei frame liberi.

Quest'idea si può estendere conservando una lista delle pagine modificate: ogni qualvolta il dispositivo di paginazione è inattivo, si sceglie una pagina modificata, la si scrive nel disco e si resetta il suo bit di modifica. Questo schema aumenta la probabilità che, al momento della selezione per la sostituzione, la pagina non abbia subito modifiche e non debba essere scritta in memoria secondaria.

Un'altra modifica consiste nell'usare un gruppo di frame liberi, ma ricordare quale pagina era contenuta in ciascun frame. Poiché quando il contenuto di un frame viene scritto su disco tale contenuto non cambia, la vecchia pagina è ancora utilizzabile prendendola dal gruppo dei frame liberi, se ce n'è bisogno prima che sia riusato quel frame. In questo caso non è necessario alcun I/O. Se si verifica un page fault si controlla prima se la pagina richiesta si trova nel gruppo dei frame liberi; se non c'è si deve individuare un frame libero e trasferirvi la pagina.

Questa tecnica, insieme con l'algoritmo di sostituzione FIFO, è usata dal sistema VAX/VMS. Quando l'algoritmo FIFO sostituisce per errore una pagina ancora in uso, la si ricupera rapidamente dal gruppo dei frame liberi senza ricorrere a operazioni di I/O. Il buffer dei frame liberi offre protezione contro l'algoritmo di sostituzione FIFO, relativamente elementare, ma di semplice implementazione. Questo metodo è necessario poiché le prime versioni del VAX non implementavano correttamente il bit di riferimento.

Alcune versioni di UNIX adottano questo metodo insieme all'algoritmo con seconda chance. In effetti, si tratta di un'utile integrazione a qualunque algoritmo di sostituzione, al fine di ridurre il prezzo pagato per l'eventuale errata scelta della pagina vittima.

9.4.8 Applicazioni e sostituzione della pagina

In taluni casi, le applicazioni che accedono ai dati tramite la memoria virtuale del sistema operativo hanno prestazioni peggiori di quelle che avrebbero se il sistema operativo non offrisse alcun buffering. Si pensi, quale esempio tipico, a un database che gestisce la memoria e il buffering dell'I/O in modo autonomo. Applicazioni come questa capiscono il proprio utilizzo della memoria e del disco meglio di quanto possa fare un sistema operativo, che applica algoritmi adatti a un uso generale. Se il sistema operativo adotta un buffer per l'I/O, e così pure fa l'applicazione, la quantità di memoria necessaria per l'I/O sarà inutilmente raddoppiata.

Un altro esempio proviene dai data warehouse, che effettuano spesso lunghe letture sequenziali del disco, seguite da calcoli e scritture. L'algoritmo LRU eliminerebbe le pagine vecchie per conservare le nuove, mentre in questo caso è ragionevole attenersi la lettura delle pagine vecchie in luogo di quelle nuove (quando l'applicazione inizia nuovamente la lettura sequenziale). In queste circostanze l'algoritmo MFU sarebbe più efficiente di LRU.

Per risolvere tali problemi, alcuni sistemi operativi permettono a certi programmi di utilizzare una partizione del disco come un array sequenziale di blocchi logici, senza ricorrere alle strutture di dati del file system. Un simile array è anche detto **disco di basso livello** (*raw disk*), e il relativo I/O è denominato **I/O di basso livello** (*raw I/O*). L'I/O di basso livello bypassa tutti i servizi del file system, come la paginazione su richiesta dell'I/O su file, il locking dei file, il prefetching, l'allocazione dello spazio, la gestione dei nomi dei file e le directory. Si noti però che, sebbene alcune applicazioni siano più efficienti quando gestiscono i propri servizi specifici di memorizzazione sul disco di basso livello, quasi tutte hanno una resa migliore quando operano con i servizi regolari del file system.

9.5 Allocazione dei frame

Consideriamo ora il problema dell'allocazione. Occorre stabilire un criterio per l'allocazione della memoria libera ai diversi processi. Come esempio, se abbiamo 93 frame liberi e due processi, quanti frame assegnamo a ciascuno?

Il caso più semplice è un sistema con utente singolo. Si consideri un sistema monoutente che disponga di 128 KB di memoria, con pagine di 1 KB. Complessivamente sono presenti 128 frame. Il sistema operativo può occupare 35 KB, lasciando 93 frame per il processo utente. In condizioni di paginazione su richiesta pura, tutti i 93 blocchi di memoria sono inizialmente posti nella lista dei frame liberi. Quando comincia l'esecuzione, il processo utente genera una sequenza di page fault. I primi 93 fault ricevono i frame liberi dalla lista. Una volta esaurita quest'ultima, per stabilire quale tra le 93 pagine presenti in memoria si debba sostituire con la novantaquattresima, si può usare un algoritmo di sostituzione delle pagine. Terminato il processo, si reinseriscono i 93 frame nella lista dei frame liberi.

Vi sono molte variazioni di questa semplice strategia. Si può richiedere che il sistema operativo assegna tutto lo spazio richiesto dalle proprie strutture dati attingendo dalla lista dei frame liberi. Quando questo spazio è inutilizzato dal sistema operativo può essere sfruttato per la paginazione utente. Un'altra variante prevede di riservare sempre tre frame liberi, in modo che quando si verifica un page fault sia sempre disponibile un frame libero in cui trasferire la pagina richiesta. Mentre ha luogo il trasferimento, si può scegliere una pagina da rimpiazzare, che viene poi scritta nel disco mentre il processo utente continua l'esecuzione. Sono possibili anche altre varianti, ma la strategia di base è chiara: al processo utente si assegna qualsiasi frame libero.

9.5.1 Numero minimo di frame

Le strategie di allocazione dei frame sono soggette a parecchi vincoli. Non si possono assegnare più frame di quanti siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre è necessario assegnare almeno un numero minimo di frame. Esaminiamo quest'ultimo requisito in maggiore dettaglio.

Una delle ragioni per allocare sempre un numero minimo di frame è legata alle prestazioni. Ovviamente, al decrescere del numero dei frame allocati a ciascun processo aumenta il tasso di page fault, con conseguente rallentamento dell'esecuzione dei processi. Inoltre va ricordato che, quando si verifica un page fault prima che sia stata completata l'esecuzione di un'istruzione, quest'ultima deve essere riavviata. Di conseguenza, i frame disponibili devono essere in numero sufficiente per contenere tutte le pagine cui ogni singola istruzione può far riferimento.

Si consideri, per esempio, un calcolatore in cui tutte le istruzioni di riferimento alla memoria hanno solo un indirizzo di memoria; in questo caso occorre almeno un frame per l'istruzione e uno per il riferimento alla memoria. Inoltre se è ammesso un indirizzamento indiretto a un livello (come nel caso di un'istruzione `load` presente nella pagina 16 che può far riferimento a un indirizzo della pagina 0, che costituisce a sua volta un riferimento indiretto alla pagina 23) la paginazione richiede allora almeno tre frame per ogni processo. Si immagini che cosa accadrebbe nel caso di un processo che disponga di due soli frame.

Il numero minimo di frame è definito dall'architettura del calcolatore. Per esempio, L'istruzione di `move` del PDP-11, per alcune modalità di indirizzamento, è costituita di più di una parola, quindi la stessa istruzione può stare a cavallo tra due pagine. Inoltre, ciascuno dei suoi due operandi può essere un riferimento indiretto, per un totale di sei frame. Un altro esempio è dato dall'istruzione `MVC` di IBM 370. Poiché l'istruzione è da memoria a memoria, può occupare 6 byte e stare a cavallo tra due pagine. Anche la sequenza di caratteri da spostare e l'area su cui effettuare lo spostamento possono essere a cavallo tra due pagine; questa situazione richiede quindi sei frame. In effetti, la situazione peggiore si presenta quando l'istruzione `MVC` è l'operando di un'istruzione `EXECUTE` che sta a cavallo di un limite di pagina; in questo caso occorrono otto frame.

Il caso peggiore si può presentare nelle architetture di calcolatori che permettono riferimenti indiretti a più livelli (per esempio quando ogni parola di 16 bit può contenere un indirizzo di 15 bit più un indicatore indiretto di 1 bit). In teoria, una semplice istruzione di caricamento può far riferimento a un indirizzo indiretto che a sua volta può far riferimento a un indirizzo indiretto (su un'altra pagina) anch'esso facente riferimento a un indirizzo indiretto su un'altra pagina ancora, e così via, finché tutte le pagine della memoria virtuale siano state chiamate in causa. Quindi, nel caso peggiore, tutta la memoria virtuale si deve trovare in memoria fisica. Per superare questa difficoltà occorre porre un limite al livello dei riferimenti indiretti, per esempio limitando un'istruzione a un massimo di 16 livelli. Quando si verifica il riferimento indiretto di primo livello, si imposta un contatore al valore 16, per decrementarlo a ciascun riferimento successivo in questa istruzione. Se il contatore si riduce a 0 si ve-

rifica un’eccezione (numero di riferimenti indiretti eccessivo). Tale limite riduce a 17 il numero massimo dei riferimenti alla memoria per ogni istruzione, richiedendo un pari numero di frame.

Il numero minimo di frame per ciascun processo è definito dall’architettura, mentre il numero massimo è definito dalla quantità di memoria fisica disponibile. In mezzo vi è un ampio spazio di scelta.

9.5.2 Algoritmi di allocazione

Il modo più semplice per suddividere m frame tra n processi è quello per cui a ciascuno si dà una parte uguale, m/n frame (ignorando per ora i frame di cui il sistema operativo ha bisogno). Dati 93 frame e cinque processi, ogni processo riceve 18 frame. I tre frame lasciati liberi si potrebbero usare come buffer di frame liberi. Questo schema è chiamato **allocazione uniforme**.

Un’alternativa consiste nel riconoscere che diversi processi hanno bisogno di quantità di memoria diverse. Si consideri un sistema con frame di 1 KB. Se un piccolo processo utente di 10 KB e una base di dati interattiva di 127 KB sono gli unici due processi in esecuzione su un sistema con 62 frame liberi, non ha senso allocare a ciascun processo 31 frame. Al processo utente non ne servono più di 10, quindi gli altri 21 sarebbero semplicemente sprecati.

Per risolvere questo problema è possibile ricorrere all’**allocazione proporzionale**, secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione. Si supponga che s_i sia la dimensione della memoria virtuale per il processo p_i . Si definisce la seguente quantità:

$$S = \sum s_i$$

Quindi, se il numero totale dei frame disponibili è m , al processo p_i si assegnano a_i frame, dove a_i è approssimativamente

$$a_i = s_i / S \times m.$$

Naturalmente è necessario scegliere ciascun a_i in modo che sia un intero maggiore del numero minimo di frame richiesti dall’instruction set e in modo che la somma di tutti gli a_i non sia maggiore di m .

Usando l’allocazione proporzionale, per suddividere 62 frame tra due processi, uno di 10 e uno di 127 pagine, si assegnano rispettivamente 4 e 57 frame, infatti:

$$\begin{aligned} 10/137 \times 62 &\approx 4 \\ 127/137 \times 62 &\approx 57. \end{aligned}$$

In questo modo entrambi i processi condividono i frame disponibili secondo le rispettive necessità, e non in modo uniforme.

Sia nell’allocazione uniforme sia in quella proporzionale, l’allocazione a ogni processo può variare rispetto al livello di multiprogrammazione. Se tale livello aumenta, ciascun processo perde alcuni frame per fornire la memoria necessaria per il nuovo processo. D’altra parte, se il livello di multiprogrammazione diminuisce, i frame allocati al processo rimosso si possono distribuire tra quelli che restano.

Occorre notare che sia con l’allocazione uniforme sia con l’allocazione proporzionale, un processo a priorità elevata è trattato come un processo a bassa priorità anche se, per definizione, si vorrebbe che al processo con elevata priorità fosse allocata più memoria per accelerarne l’esecuzione, a discapito dei processi a bassa priorità. Un soluzione prevede l’uso di uno schema di allocazione proporzionale in cui il rapporto dei frame non dipende dalle dimensioni relative dei processi, ma dalle priorità degli stessi oppure da una combinazione di dimensioni e priorità.

9.5.3 Allocazione globale e allocazione locale

Un altro importante fattore che riguarda il modo in cui si assegnano i frame ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione delle pagine si possono classificare in due categorie generali: **sostituzione globale** e **sostituzione locale**. La sostituzione globale permette che per un processo si scelga un frame per la sostituzione dall’insieme di tutti i frame, anche se quel frame è al momento allocato a un altro processo; un processo può dunque sottrarre un frame a un altro processo. La sostituzione locale richiede invece che per ogni processo si scelga un frame solo dal proprio insieme di frame.

Si consideri per esempio uno schema di allocazione che, per una sostituzione a favore dei processi ad alta priorità, permetta di sottrarre frame ai processi a bassa priorità. Un processo può scegliere per la sostituzione uno dei suoi rame o uno di quelli di qualsiasi processo con priorità minore. Questo metodo permette a un processo ad alta priorità di aumentare il proprio livello di allocazione dei frame a discapito dei processi a bassa priorità.

Con la strategia di sostituzione locale, il numero di blocchi di memoria assegnati a un processo non cambia. Con la sostituzione globale, invece, può accadere che per un certo processo si selezionino solo frame allocati ad altri processi, aumentando così il numero di frame assegnati a quel processo, purché altri non scelgano per la sostituzione i *suoi* frame.

L’algoritmo di sostituzione globale risente di un problema: un processo non può controllare il proprio tasso di page fault, infatti l’insieme di pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi. Quindi, lo stesso processo può comportarsi in modi molto diversi, per esempio impiegando 0,5 secondi per un’esecuzione e 10,3 secondi per quella successiva, a causa di circostanze del tutto esterne. Con l’algoritmo di sostituzione locale questo problema non si presenta. Infatti l’insieme di pagine in memoria per un processo subisce l’effetto del comportamento di paginazione di quel solo processo. Tuttavia la sostituzione locale può penalizzare un processo, non rendendogli disponibili altre pagine di memoria meno usate. Generalmente, la sostituzione globale genera una maggiore produttività del sistema, e perciò è il metodo più usato.

9.5.4 Accesso non uniforme alla memoria

Fino a questo momento trattando il tema della memoria virtuale abbiamo assunto che le diverse parti della memoria centrale siano uguali, o almeno che vi si potesse accedere nello stesso modo. In molti sistemi informatici non è così. Spesso in sistemi con processori multipli (Paragrafo 1.3.2) un certo processore può accedere ad alcune regioni della memoria più rapidamente rispetto ad altre. Tali differenze nelle prestazioni sono causate dalla modalità di interconnessione tra processori e memoria all'interno del sistema. Frequentemente un tale sistema è costituito da diverse schede madri, ognuna contenente più processori e una parte di memoria. Le schede sono connesse in vari modi, a partire dai bus di sistema fino a connessioni di rete ad alta velocità come InfiniBand. Come ci si aspetta, i processori di una particolare scheda possono accedere alla memoria della scheda stessa in meno tempo rispetto a quello necessario per accedere ad altre schede del sistema. I sistemi nei quali i tempi di accesso alla memoria variano in modo significativo sono generalmente detti **sistemi con accesso non uniforme alla memoria** (*non-uniform memory access*, NUMA) e, senza eccezioni, sono più lenti dei sistemi nei quali memoria e processori risiedono sulla stessa scheda madre.

Le decisioni su quali frame di pagina memorizzare in quale posizione possono condizionare in modo significativo le prestazioni nei sistemi NUMA. Se, in sistemi del genere, consideriamo uniforme la memoria, i processori potrebbero dover aspettare molto più a lungo per accedere alla memoria rispetto al caso in cui gli algoritmi di allocazione della memoria siano modificati per tenere in conto il NUMA. Analoghe modifiche devono essere apportate anche al sistema di scheduling. L'obiettivo di questi cambiamenti è quello di allocare i frame di memoria “il più vicino possibile” al processore sul quale il processo è in esecuzione, dove per “vicino” si intende “con latenza minima”, ovvero, di solito, sulla stessa scheda della CPU.

I cambiamenti negli algoritmi consistono nel fatto che lo scheduler tiene traccia dell'ultimo processore sul quale ciascun processo è stato eseguito. Se lo scheduler cerca di allocare ciascun processo sul suo processore precedente, e se il sistema di gestione della memoria cerca di allocare frame per il processo vicino al processore sul quale sta per essere mandato in esecuzione, si otterrà un incremento dei cache hit e una diminuzione del tempo di accesso alla memoria.

La questione diventa ancora più complicata con l'aggiunta dei thread. Per esempio, un processo con molti thread in esecuzione potrebbe vedere quei thread schedulati su differenti schede del sistema. Come viene allocata la memoria in questo caso? Solaris risolve il problema creando una entità **lgroup** (*latency group*, ovvero gruppo di latenza) nel kernel. Ogni lgroup raccoglie i processori e la memoria vicini fra loro. In effetti gli lgroup sono ordinati gerarchicamente sulla base del periodo di latenza tra i gruppi. Solaris tenta di pianificare tutti i thread di un processo e di allocare tutta la memoria di un processo nell'ambito di un solo lgroup. Se una tale soluzione non è possibile, per il resto delle risorse necessarie vengono utilizzati gli lgroup più vicini, in modo da minimizzare la latenza complessiva della memoria e massimizzare il tasso di successo della cache del processore.

9.6 Thrashing

Se il numero dei frame allocati a un processo con priorità bassa diviene inferiore al numero minimo richiesto dall’architettura del calcolatore, occorre sospendere l’esecuzione del processo, e quindi togliere le pagine restanti, liberando tutti i frame allocati. Questa operazione introduce un livello intermedio di scheduling per la gestione dell’entrata e dell’uscita dei processi dalla memoria centrale.

Infatti, si consideri un qualsiasi processo che non disponga di un numero di frame “sufficiente”. Se non dispone del numero di frame sufficiente per ospitare le pagine attive, il processo incorre rapidamente in un page fault. A questo punto si deve sostituire qualche pagina; ma, poiché tutte le sue pagine sono attive, si deve sostituire una pagina che sarà subito necessaria, e di conseguenza si verificano parecchi page fault poiché si sostituiscono pagine che saranno immediatamente riportate in memoria.

Questa intensa paginazione è nota come *thrashing*. Un processo in thrashing spende più tempo per la paginazione che per l’esecuzione dei processi.

9.6.1 Cause del thrashing

Il thrashing causa notevoli problemi di prestazioni. Si consideri il seguente scenario, basato sul comportamento effettivo dei primi sistemi di paginazione.

Il sistema operativo controlla l’utilizzo della CPU. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione delle pagine globale, che sostituisce le pagine senza tener conto del processo al quale appartengono. Ora si ipotizzi che un processo entri in una nuova fase d’esecuzione e richieda più frame; se ciò si verifica si ha una serie di page fault, cui segue la sottrazione di frame ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch’essi dei page fault, con conseguente sottrazione di frame ad altri processi. Per effettuare il caricamento e lo scaricamento delle pagine per questi processi si deve usare il dispositivo di paginazione. Mentre si mettono i processi in coda per il dispositivo di paginazione, la coda dei processi pronti per l’esecuzione si svuota, quindi l’utilizzo della CPU diminuisce.

Lo scheduler della CPU rileva questa riduzione dell’utilizzo della CPU e *aumenta* il grado di multiprogrammazione. Si tenta di avviare il nuovo processo sottraendo pagine ai processi in esecuzione, causando ulteriori page fault e allungando la coda per il dispositivo di paginazione. Come risultato l’utilizzo della CPU scende ulteriormente, e lo scheduler della CPU tenta di aumentare ancora il grado di multiprogrammazione. Si è in una situazione di thrashing che fa precipitare la produttività del sistema. Il tasso dei page fault aumenta enormemente, e di conseguenza aumenta il tempo effettivo d’accesso alla memoria. I processi non svolgono alcun lavoro, poiché si sta spendendo tutto il tempo nell’attività di paginazione.

Questo fenomeno è illustrato nella Figura 9.18, in cui si riporta l’utilizzo della CPU in funzione del grado di multiprogrammazione. Aumentando il grado di multiprogrammazione aumenta anche l’utilizzo della CPU, anche se più lentamente, fino a raggiungere un massimo. Se a questo punto si aumenta ulteriormente il grado di mul-

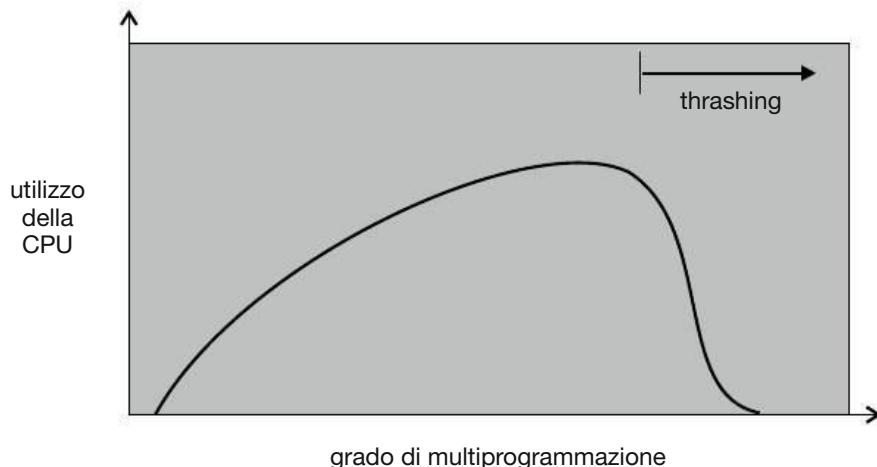


Figura 9.18 Thrashing.

tiprogrammazione, l'attività di paginazione degenera e fa crollare l'utilizzo della CPU. In questa situazione, per aumentare l'utilizzo della CPU e bloccare il thrashing occorre *ridurre* il grado di multiprogrammazione.

Gli effetti di questa situazione si possono limitare usando un **algoritmo di sostituzione locale**, o **algoritmo di sostituzione per priorità**. Con la sostituzione locale, se un processo entra in thrashing, non può sottrarre frame a un altro processo e quindi provocarne a sua volta la degenerazione. Tuttavia il problema non è completamente risolto. I processi in thrashing rimangono nella coda d'attesa del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio di un page fault aumenta a causa dell'allungamento della coda media d'attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo d'accesso in memoria aumenta anche per gli altri processi.

Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame “servano” a un processo si impiegano diverse tecniche. L'approccio del working-set, trattato nel Paragrafo 9.6.2, comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo approccio definisce il **modello di località** d'esecuzione del processo.

Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine usate attivamente insieme, com'è illustrato nella Figura 9.19. Generalmente un programma è formato di parecchie località diverse, che sono sovrapponibili.

Per esempio, quando s'invoca una procedura, essa definisce una nuova località. In questa località si fanno riferimenti alla memoria per le istruzioni della procedura, per le sue variabili locali e per un sottoinsieme delle variabili globali. Quando la procedura termina, il processo lascia questa località, poiché le variabili locali e le istruzioni della procedura non sono più usate attivamente. Potrà tornare più tardi in questa località.

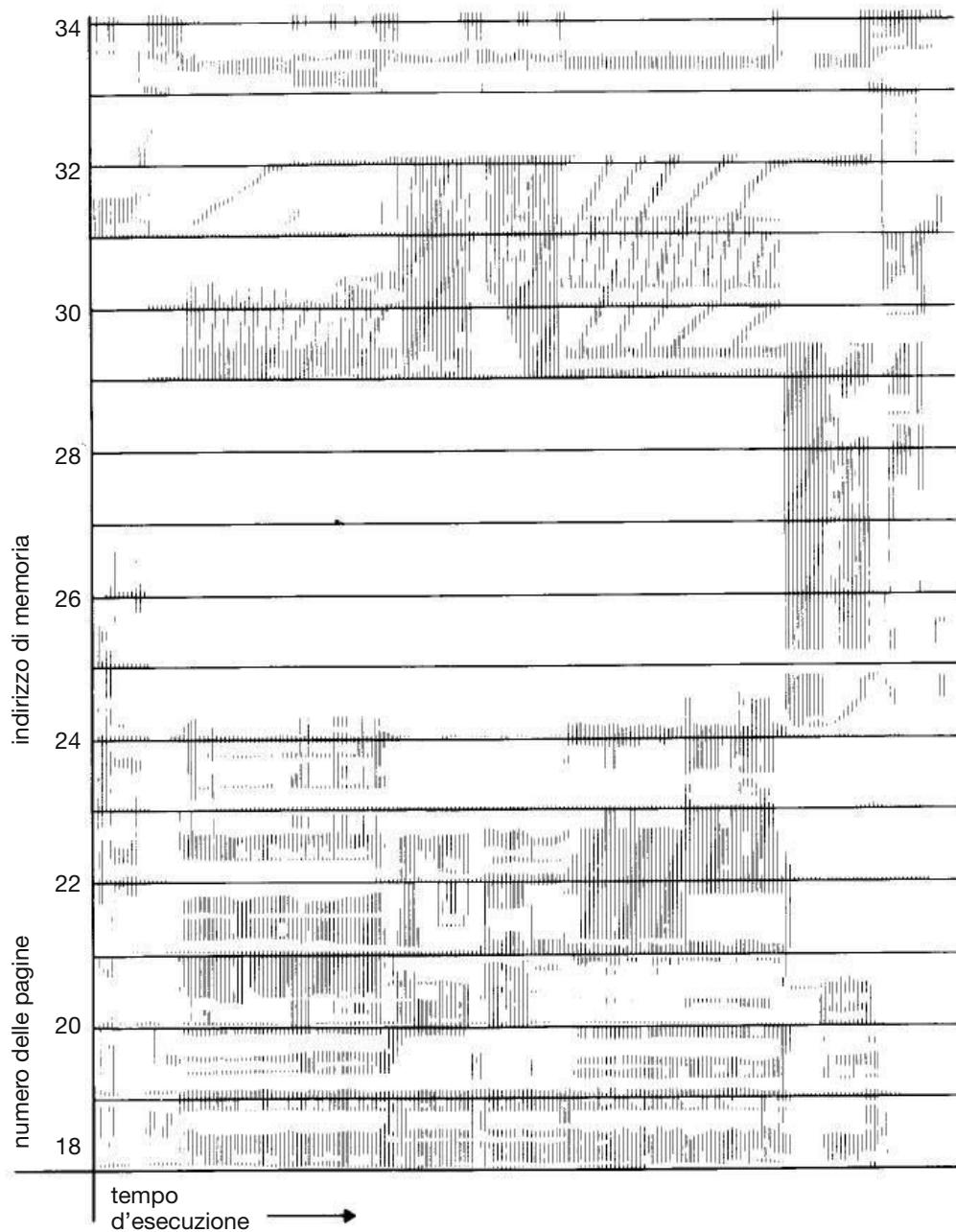


Figura 9.19 Località dei riferimenti alla memoria.

Quindi, le località sono definite dalla struttura del programma e dalle relative strutture dati. Il modello di località sostiene che tutti i programmi mostrino questa struttura di base di riferimenti alla memoria. Si noti che il modello di località è il principio non dichiarato sottostante all’analisi fin qui svolta sul caching. Se gli accessi ai vari tipi di dati fossero casuali, anziché strutturati in località, il caching sarebbe inutile.

Si supponga di allocare a un processo un numero di frame sufficiente per sistemare le sue località attuali. Finché tutte queste pagine non si trovano in memoria, si veri-

riferimenti alle pagine

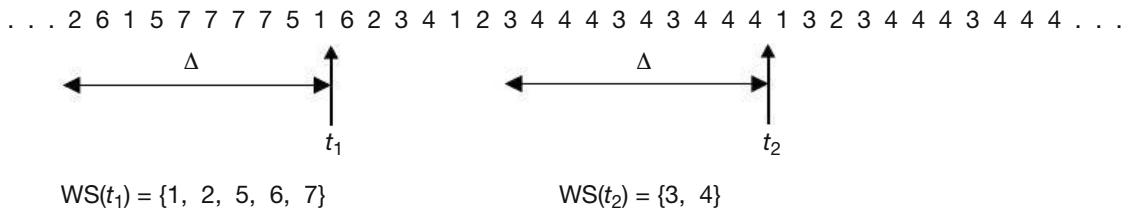


Figura 9.20 Modello del working set.

ficano le assenze delle pagine relative a tali località; quindi, finché le località non vengano modificate, non hanno luogo altri page fault. Se si assegnano meno frame rispetto alla dimensione della località attuale, la paginazione del processo degenera, poiché non si possono tenere in memoria tutte le pagine che il processo sta usando attivamente.

9.6.2 Modello del working set

Come già accennato, il **modello del working set** è basato sull’ipotesi di località. Questo modello usa un parametro, Δ , per definire la **finestra del working set**. L’idea consiste nell’esaminare i più recenti Δ riferimenti alle pagine. L’insieme di pagine nei più recenti Δ riferimenti è il working set (Figura 9.20). Se una pagina è in uso attivo si trova nel working set; se non è più usata esce dal working set Δ unità di tempo dopo il suo ultimo riferimento. Quindi, il working set non è altro che un’approssimazione della località del programma.

Per esempio, data la successione di riferimenti alla memoria mostrata nella Figura 9.20, se $\Delta = 10$ riferimenti alla memoria, il working set all’istante t_1 è $\{1, 2, 5, 6, 7\}$. All’istante t_2 il working set è diventato $\{3, 4\}$.

La precisione del working set dipende dalla scelta del valore di Δ . Se Δ è troppo piccolo non include l’intera località, se è troppo grande può sovrapporre più località. Al limite, se Δ è infinito il working set coincide con l’insieme di pagine cui il processo fa riferimento durante la sua esecuzione.

La caratteristica più importante del working set è la sua dimensione. Calcolandone la dimensione WSS_i , per ciascun processo p_i del sistema, si può determinare la richiesta totale di frame, cioè D :

$$D = \sum WSS_i$$

Ogni processo usa attivamente le pagine del proprio working set. Quindi, il processo i necessita di WSS_i frame. Se la richiesta totale è maggiore del numero totale di frame liberi ($D > m$), si avrà thrashing, poiché alcuni processi non dispongono di un numero sufficiente di frame.

Una volta scelto Δ , l’uso del modello del working set è abbastanza semplice. Il sistema operativo controlla il working set di ogni processo e gli assegna un numero di frame sufficiente, rispetto alle dimensioni del suo working set. Se i frame ancora li-

beri sono in numero sufficiente, si può iniziare un altro processo. Se la somma delle dimensioni dei working set aumenta, superando il numero totale dei frame disponibili, il sistema operativo individua un processo da sospendere. Scrive in memoria secondaria le pagine di quel processo e assegna i suoi frame ad altri processi. Il processo sospeso può essere ripreso successivamente.

Questa strategia impedisce il thrashing, mantenendo il grado di multiprogrammazione più alto possibile, quindi ottimizza l'utilizzo della CPU.

Poiché la finestra del working set è una finestra dinamica, la difficoltà insita in questo modello consiste nel tener traccia degli elementi che compongono il working set stesso. A ogni riferimento alla memoria, a un'estremità appare un riferimento nuovo e il riferimento più vecchio fuoriesce dall'altra estremità. Una pagina si trova nel working set se esiste un riferimento a essa in qualsiasi punto della finestra del working set.

Si può approssimare il modello con un interrupt da timer ad intervalli fissi ed un bit di riferimento. Si supponga, per esempio, che Δ sia pari a 10.000 riferimenti e che sia possibile ottenere un segnale d'interruzione dal timer ogni 5000 riferimenti. Quando si verifica uno di tali segnali d'interruzione, i valori dei bit di riferimento di ciascuna pagina vengono copiati in memoria e poi azzerati. Così, quando si verifica un page fault è possibile esaminare il bit di riferimento corrente e 2 bit in memoria per stabilire se una pagina sia stata usata entro gli ultimi 10.000-15.000 riferimenti. Se lo è stata, almeno uno di questi bit è attivo. Se non lo è stata, questi bit sono tutti inattivi. Le pagine con almeno un bit attivo si considerano appartenenti al working set. Occorre notare che questo schema non è del tutto preciso, poiché non è possibile stabilire dove si è verificato un riferimento entro un intervallo di 5000. L'incertezza si può ridurre aumentando il numero dei bit di cronologia e la frequenza dei segnali d'interruzione, per esempio, 10 bit e un'interruzione ogni 1000 riferimenti. Tuttavia, il costo per servire questi segnali d'interruzione più frequenti aumenta in modo corrispondente.

9.6.3 Frequenza dei page fault

Il modello del working set ha avuto successo, e la sua conoscenza può servire per la prepaginazione (Paragrafo 9.9.1), ma appare un modo alquanto goffo per controllare il thrashing. La strategia basata sulla **frequenza dei page fault** (*page fault frequency*, PFF) è più diretta.

Il problema specifico è la prevenzione del thrashing. La frequenza dei page fault in tale situazione è alta, ed è proprio questa che si deve controllare. Se la frequenza è eccessiva, significa che il processo necessita di più frame. Analogamente, se la frequenza dei page fault è molto bassa, il processo potrebbe disporre di troppi frame. Si può fissare un limite inferiore e un limite superiore per la frequenza desiderata dei page fault (Figura 9.21). Se la frequenza effettiva dei page fault per un processo oltrepassa il limite superiore, occorre allocare a quel processo un altro frame; se la frequenza scende sotto il limite inferiore, si sottrae un frame a quel processo. Quindi,

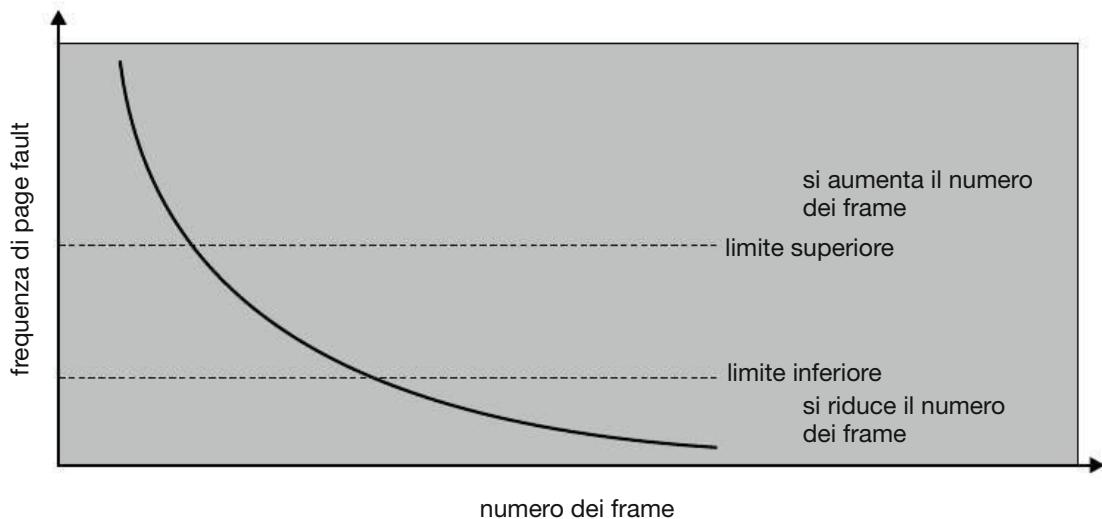


Figura 9.21 Frequenza dei page fault.

per prevenire il thrashing, si può misurare e controllare direttamente la frequenza dei page fault.

Come nel caso della strategia del working set, può essere necessaria lo swapping di un processo. Se la frequenza dei page fault aumenta e non ci sono frame disponibili, occorre selezionare un processo e spostarlo sul backing store. I frame liberati si distribuiscono ai processi con elevate frequenze di page fault.

9.6.4 Osservazioni conclusive

Il thrashing e l'avvicendamento dei processi hanno un pessimo impatto sulle prestazioni. La miglior regola adottata attualmente nella realizzazione dei computer è quella di includere una quantità di memoria fisica sufficiente ad evitare thrashing e avvicendamento, quando possibile. Sia quando si parla di smartphone che nel caso di mainframe, fornire una quantità di memoria sufficiente a mantenere tutti gli insiemi di lavoro contemporaneamente in memoria, eccetto in condizioni estreme, offre all'utente la migliore esperienza d'uso possibile.

9.7 File mappati in memoria

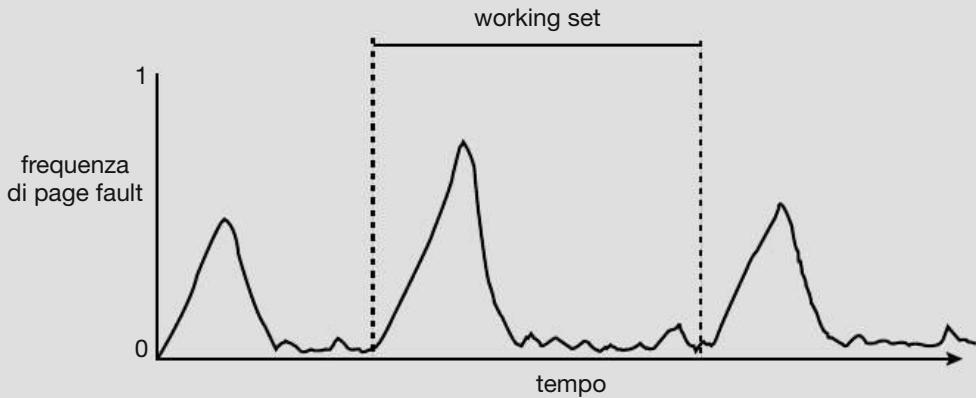
Si consideri la lettura sequenziale di un file sul disco per mezzo delle consuete chiamate di sistema: `open()`, `read()` e `write()`. Ciascun accesso al file richiede una chiamata di sistema e un accesso al disco. In alternativa, possiamo avvalerci delle tecniche di memoria virtuale analizzate sin qui per trattare l'I/O dei file come l'accesso ordinario alla memoria. Grazie a questa soluzione, nota come **mappatura dei file in memoria**, una parte dello spazio degli indirizzi virtuali può essere associata logicamente al file. Come vedremo, ciò può portare a un incremento significativo delle prestazioni.



WORKING SET E TASSI DI PAGE FAULT

Vi è una relazione diretta tra il working set di un processo e il tasso di page fault. Come mostra la Figura 9.20, il working set di un processo cambia nel corso del tempo, mentre i riferimenti ai dati e alle parti del codice passano da una località all'altra.

Assumendo memoria sufficiente a contenere il working set di un processo (ossia, a evitare il thrashing), il tasso di page fault oscillerà, in un dato periodo di tempo, tra picchi e valli. Questa tendenza generale è illustrata di seguito.



Un picco nel tasso di page fault si verifica alle richieste di paginazione relative a una nuova località. Tuttavia, una volta che il working set interessato è in memoria, tasso di page fault precipita. Quando il processo entra in un nuovo working set, il tasso si impenna ancora una volta verso un picco; quando il nuovo working set è caricato in memoria, il tasso crolla nuovamente. L'intervallo di tempo tra l'inizio di un picco e quello del picco successivo descrive la transizione da un working set a un altro.

9.7.1 Meccanismo di base

La mappatura di un file in memoria si realizza associando un blocco del disco a una o più pagine residenti in memoria. L'accesso iniziale al file avviene tramite una normale richiesta di paginazione, che causa un errore di page fault. Tuttavia, una porzione del file, che è pari a una pagina, è caricata dal file system in una pagina fisica (alcuni sistemi possono decidere di caricare porzioni più grandi di memoria). Ogni successiva lettura e scrittura del file è gestita come accesso ordinario alla memoria, semplificando e velocizzando così l'accesso al file e il suo utilizzo, in quanto si permette al sistema di manipolare i file attraverso la memoria anziché incorrere nell'overhead delle chiamate di sistema `read()` e `write()`.

Si osservi come le scritture sul file mappato in memoria non si traducano necessariamente in scritture immediate (sincrone) sul file del disco. Alcuni sistemi scelgono di aggiornare il file fisico quando il sistema operativo esegue un controllo periodico per l'accertamento di eventuali modifiche alle pagine. Quando il file viene chiuso, tutti i dati mappati in memoria sono scritti nuovamente su disco e rimossi dalla memoria virtuale del processo.

Alcuni sistemi operativi prevedono un'apposita chiamata di sistema per la mappatura dei file in memoria; le chiamate ordinarie sono riservate a tutte le altre operazioni di I/O su file. Altri sistemi possono mappare un file in memoria, anche in assenza di un'esplicita indicazione in tal senso. Prendiamo per esempio Solaris. Se si dichiara che un file deve essere mappato in memoria, tramite la chiamata di sistema `mmap()`, Solaris opera la mappatura del file nello spazio degli indirizzi del processo. Se si apre un file, e vi si accede con le chiamate di sistema ordinarie, quali `open()`, `read()` e `write()`, Solaris mappa ancora in memoria il file; tuttavia, il file è mappato nello spazio degli indirizzi del kernel. Quindi, a prescindere da come si apre il file, Solaris considera tutto l'I/O relativo ai file come mappato in memoria, sfruttando per l'accesso ai file l'efficiente sottosistema della memoria.

Per consentire la condivisione dei dati, più processi possono essere autorizzati a mappare contemporaneamente un file in memoria. Le scritture di uno di questi processi modificano i dati nella memoria virtuale e risultano visibili a tutti gli altri processi che mappano la stessa sezione del file. L'analisi sulla memoria virtuale svolta fin qui dovrebbe aver chiarito come la condivisione delle sezioni di memoria interessate dalla mappatura abbia luogo: la memoria virtuale di ciascun processo che partecipa alla condivisione punta alla stessa pagina della memoria fisica – la pagina che ospita una copia del blocco del disco. Tale situazione è illustrata nella Figura 9.22. Le chiamate di sistema per la mappatura in memoria possono inoltre offrire la funzionalità di copiatura su scrittura, che consente ai processi di condividere un file in

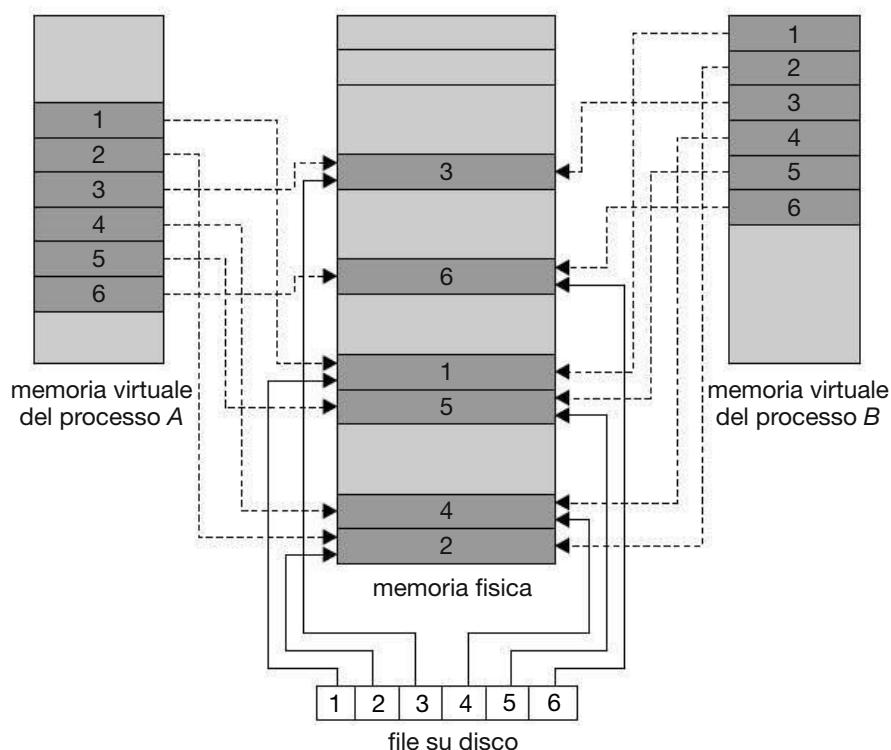


Figura 9.22 File mappati in memoria.

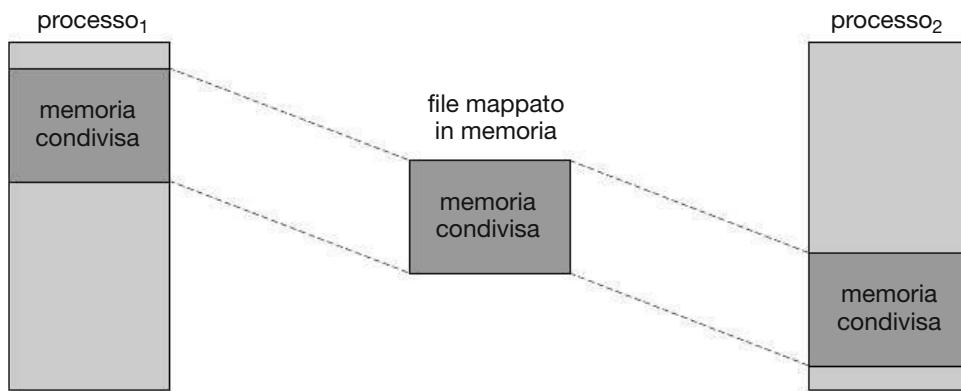


Figura 9.23 Condivisione della memoria tramite I/O mappato in memoria.

modalità di sola lettura ma di avere una copia propria dei dati che modificano. Affinché l’accesso ai dati condivisi sia coordinato, i processi interessati potrebbero usare uno dei meccanismi per la mutua esclusione, descritti nel Capitolo 5.

Spesso la memoria condivisa viene implementata utilizzando i file mappati in memoria. La comunicazione fra processi si ottiene in questi casi mappando in memoria uno stesso file negli spazi degli indirizzi virtuali dei processi coinvolti. Il file mappato in memoria funge da area di memoria condivisa tra i processi comunicanti (Figura 9.23). Abbiamo già analizzato questa situazione nel Paragrafo 3.4.1, in cui si crea un oggetto POSIX di memoria condivisa e ogni processo comunicante mappa l’oggetto in memoria nel proprio spazio degli indirizzi. Nel paragrafo successivo vedremo come la API Windows fornisca gli strumenti per condividere memoria tramite mappatura dei file in memoria.

9.7.2 Memoria condivisa nella API Windows

La procedura generale per la configurazione di una regione di memoria condivisa utilizzando i file mappati in memoria con la API Windows consiste, dapprima, nel creare un **file mapping** per il file interessato dall’operazione, per poi stabilire una vista (*view*) del file mappato nello spazio degli indirizzi virtuali di un processo. Un secondo processo, a questo punto, può aprire e creare una sua vista del file mappato nel proprio spazio degli indirizzi virtuali. Il file mappato è l’oggetto di memoria condiviso tramite il quale può svolgersi la comunicazione tra i processi.

Illustriamo ora queste fasi più da vicino. In questo esempio, il processo produttore crea dapprima un oggetto di memoria condiviso, sfruttando le funzionalità per la mappatura di memoria disponibili nella API Windows. Il produttore scrive quindi un messaggio nella memoria condivisa. Il processo consumatore in seguito crea a sua volta una mappatura del file, e legge il messaggio scritto dal produttore.

Per costruire un file mappato in memoria, il processo apre in primo luogo il file da mappare con la funzione `CreateFile()`, che restituisce un riferimento `HANDLE` al file. Il processo allora crea una mappatura di questo `HANDLE`, usando la funzione `CreateFileMapping()`. Una volta stabilita la mappatura del file, il processo genera

```

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

int main(int argc, char *argv[])
{
    HANDLE hFile, hMapFile;
    LPVOID lpMapAddress;

    hFile = CreateFile("temp.txt", /* nome del file */
                      GENERIC_READ | GENERIC_WRITE, /* accesso R/W */
                      0, /* nessuna condivisione del file*/
                      NULL, /* sicurezza di default */
                      OPEN_ALWAYS, /* apre il file (nuovo o esistente) */
                      FILE_ATTRIBUTE_NORMAL, /* attributi del file ordinari */
                      NULL); /* niente template del file*/

    hMapFile = CreateFileMapping(hFile, /* riferimento al file */
                                NULL, /* sicurezza di default */
                                PAGE_READWRITE, /* accesso R/W alle pagine mappate */
                                0, /* mappa l'intero file */
                                0,
                                TEXT("OggettoCondiviso")); /* oggetto condiviso con nome */

    lpMapAddress = MapViewOfFile(hMapFile, /* riferimento al file */
                                FILE_MAP_ALL_ACCESS, /* accesso R/W */
                                0, /* vista dell'intero file */
                                0,
                                0);
    /* scrive nella memoria condivisa */
    sprintf(lpMapAddress, "Messaggio nella memoria condivisa ");

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hFile);
    CloseHandle(hMapFile);

}

```

Figura 9.24 Produttore che scrive nella memoria condivisa tramite la API Windows.

nel proprio spazio degli indirizzi virtuali una vista del file mappato, con la funzione `MapViewOfFile()`. La vista costituisce la porzione di file che risiederà nello spazio degli indirizzi virtuali del processo (il file può essere mappato solo in parte o per intero). Il programma mostrato nella Figura 9.24 illustra questa procedura (molti controlli sugli errori sono stati omessi per esigenze di brevità).

L'invocazione a `CreateFileMapping()` genera un **oggetto di memoria condiviso con nome**, chiamato `SharedObject`. Il processo consumatore utilizzerà questo segmento di memoria condivisa per comunicare, creando una mappatura del medesimo oggetto con nome. Il produttore, quindi, crea nel proprio spazio degli indirizzi virtuali una vista del file mappato in memoria. Passando agli ultimi tre parametri il valore 0, si richiede che la porzione mappata sia l'intero file; si sarebbero anche potuti passare valori che specifichino l'inizio e la dimensione della porzione da mappare. (Si osservi che non è detto che l'intero file sia caricato in memoria al momento in cui se ne richiede la mappatura: è possibile che il caricamento avvenga tramite paginazione su richiesta, trasferendo perciò di volta in volta le pagine a cui si accede.) La funzione `MapViewOfFile()` restituisce un puntatore all'oggetto di memoria condiviso; ogni accesso a questa locazione di memoria è dunque un accesso al file mappato in memoria. In questo caso, nella memoria condivisa il processo produttore scrive il messaggio **"Messaggio nella memoria condivisa"**.

La Figura 9.25 contiene un programma che dimostra come il processo consumatore stabilisca una vista dell'oggetto di memoria condiviso con nome. Il codice è un po' più semplice di quello della Figura 9.24, poiché il processo deve solo mappare l'oggetto di memoria condiviso con nome già esistente. Come il processo produttore, anche il processo consumatore deve creare una vista del file mappato. Il consumatore, quindi, legge dalla memoria condivisa il messaggio scritto dal processo produttore.

Infine, entrambi i processi eliminano la vista del file mappato chiamando `UnmapViewOfFile()`. Alla fine del capitolo il lettore troverà un esercizio di programmazione per la API Windows incentrato sulla condivisione della memoria tramite mappatura dei file.

9.7.3 Mappatura in memoria dell'I/O

Per quanto riguarda l'I/O, come descritto nel Paragrafo 1.2.1, ogni controllore è dotato di registri contenenti i comandi e i dati in via di trasferimento. Solitamente esistono istruzioni espressamente dedicate all'I/O che consentono il trasferimento dei dati fra questi registri e la memoria del sistema. Le architetture di molti elaboratori, per rendere più agevole l'accesso ai dispositivi dell'I/O, forniscono la **mappatura in memoria dell'I/O** (*memory mapped I/O*). In questo caso, gli indirizzi di memoria compresi in certi intervalli devono essere riservati alla mappatura dei registri dei dispositivi. Le operazioni in lettura e scrittura su questi indirizzi di memoria fanno sì che i dati siano trasferiti fra i registri dei dispositivi e la memoria. Questo metodo è adatto a dispositivi che abbiano rapidi tempi di risposta, quali i controllori video. Nel PC IBM, a ciascuna posizione sullo schermo corrisponde una locazione mappata in memoria; visualizzare un testo sullo schermo è quasi altrettanto facile che scrivere il testo nelle relative locazioni mappate in memoria.

```

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

int main(int argc, char *argv[])
{
    HANDLE hMapFile;
    LPVOID lpMapAddress;

    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, /* R/W */
        FALSE, /* nessuna ereditarietà */
        TEXT("OggettoCondiviso")); /* nome del file mappato */

    lpMapAddress = MapViewOfFile(hMapFile, /*riferimento al file*/
        FILE_MAP_ALL_ACCESS, /* accesso in lettura/scrittura */
        0, /* vista dell'intero file */
        0,
        0);

    /* lettura dalla memoria condivisa */
    printf("Messaggio letto: %s", lpMapAddress);

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hMapFile);
}

```

Figura 9.25 Consumatore che legge dalla memoria condivisa tramite la API Windows.

La mappatura in memoria dell'I/O, inoltre, è conveniente per altri dispositivi, come le porte seriali e parallele usate per connettere i modem e le stampanti all'elaboratore. La CPU invia dati tramite tali dispositivi leggendo e scrivendo alcuni registri all'interno del dispositivo, detti **porte di I/O**. Per inviare una lunga sequenza di byte attraverso una porta seriale mappata in memoria, la CPU scrive un byte nel registro dei dati e imposta un bit nel registro di controllo per indicare che il byte è disponibile. Il dispositivo preleva il byte di dati e cancella il bit del registro di controllo per segnalare che è pronto a ricevere un nuovo byte; a questo punto, la CPU può trasferire il byte successivo. Se la CPU sottopone il bit di controllo a *polling* e cioè esegue costantemente un ciclo per rilevare quando il dispositivo divenga pronto, si parla di **I/O programmato** (*programmed I/O*, PIO). Se la CPU, invece, riceve dal dispositivo un'interruzione non appena è pronto per il byte successivo, si parla di trasferimento dati **basato sulle interruzioni**.

9.8 Allocazione di memoria del kernel

Quando un processo eseguito in modalità utente necessita di memoria aggiuntiva, le pagine sono allocate dalla lista dei frame disponibili mantenuta dal kernel. Per formare questa lista, si applica in genere uno degli algoritmi di sostituzione delle pagine esaminati nel Paragrafo 9.4; molto verosimilmente, la lista conterrà pagine non utilizzate sparse per tutta la memoria, come abbiamo visto in precedenza. Va inoltre ricordato che, se un processo utente richiede un solo byte di memoria, si ottiene frammentazione interna, poiché al processo viene garantito un intero frame.

Il kernel, per allocare la propria memoria, attinge spesso a una riserva di memoria libera differente dalla lista usata per soddisfare gli ordinari processi in modalità utente. Questo avviene principalmente per due motivi.

1. Il kernel richiede memoria per strutture dati dalle dimensioni variabili; alcune di loro corrispondono a meno di una pagina. Deve quindi fare un uso oculato della memoria, tentando di contenere al minimo gli sprechi dovuti alla frammentazione. Questo fattore è di particolare rilevanza, se si considera che, in molti sistemi operativi, il codice e i dati del kernel non sono paginabili.
2. Le pagine allocate ai processi in modalità utente non devono necessariamente essere contigue nella memoria fisica. Alcuni dispositivi, però, interagiscono direttamente con la memoria fisica, senza il vantaggio dell’interfaccia della memoria virtuale; di conseguenza, possono richiedere memoria che risieda in pagine fisicamente contigue.

Nei paragrafi successivi esaminiamo due strategie per la gestione della memoria libera assegnata ai processi del kernel: il cosiddetto “sistema buddy” e l’allocazione a lastre.

9.8.1 Sistema buddy

Il “sistema buddy” utilizza un segmento di grandezza fissa per l’allocazione della memoria, costituito da pagine fisicamente contigue. La memoria è assegnata mediante un cosiddetto **allocatore a potenze di 2**, che alloca memoria in unità di dimensioni pari a potenze di 2 (4 KB, 8 KB, 16 KB, e via di seguito). Le differenti quantità richieste sono arrotondate alla successiva potenza di 2. Per esempio, una richiesta di 11 KB viene soddisfatta con un segmento di 16 KB.

Consideriamo un semplice esempio e ipotizziamo che la grandezza di un segmento di memoria sia inizialmente di 256 KB e che il kernel richieda 21 KB di memoria. In primo luogo il segmento è suddiviso in due *buddy* (“compagni”), che chiameremo A_s e A_d , ciascuno dei quali misura 128 KB. Uno di questi è ulteriormente dimezzato in 2 *buddy* da 64 KB, diciamo B_s e B_d . Poiché la minima potenza di 2 che superi 21 KB è pari a 32 KB, occorre suddividere ancora B_s , oppure B_d , in due *buddy* la cui dimensione è 32 KB; chiamiamoli C_s e C_d . Uno di questi è utilizzato per soddisfare la richiesta di 21 KB. Lo schema è illustrato nella Figura 9.26, dove C_s è il segmento allocato per la richiesta di 21 KB.

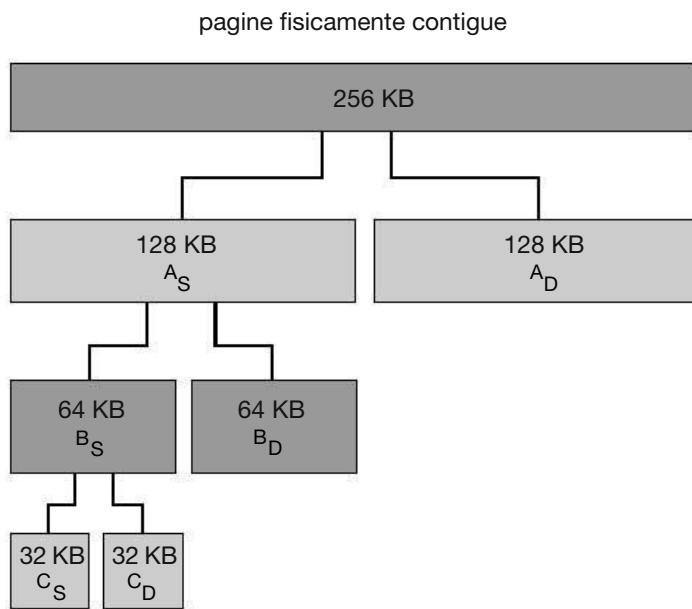


Figura 9.26 Sistema di allocazione buddy.

Questo sistema offre il vantaggio di poter congiungere rapidamente buddy adiacenti per formare segmenti più grandi tramite una tecnica nota come **fusione** (*coalescing*). Nella Figura 9.26, per esempio, quando il kernel rilascia l'unità C_s che gli era stata allocata, il sistema può fondere C_s e C_d in un segmento di 64 KB. Questo segmento, B_s , può a sua volta fondersi con il proprio compagno B_d , costituendo un segmento di 128 KB. Con l'ultima operazione di fusione si può ritornare al segmento originale di 256 KB.

L'ovvio inconveniente di questo sistema è che l'arrotondamento per eccesso a una potenza di 2 può facilmente generare frammentazione all'interno dei segmenti allocati. Una richiesta di 33 KB, per esempio, può essere soddisfatta solo con un segmento di 64 KB. Proprio per effetto della frammentazione interna, dunque, risulta impossibile garantire che lo spreco dell'unità allocata resterà al di sotto del 50%. Nel paragrafo successivo presentiamo una tecnica di allocazione della memoria priva dello spreco dovuto alla frammentazione.

9.8.2 Allocazione a lastre

Una seconda strategia per assegnare la memoria del kernel è detta **allocazione a lastre** (*slab allocation*). Una **lastra** è composta da una o più pagine fisicamente contigue. Una **cache** consiste di una o più lastre. Vi è una sola cache per ciascuna categoria di struttura dati del kernel: una cache dedicata alla struttura dati che rappresenta i descrittori dei processi, una dedicata agli oggetti che rappresentano i file, un'altra per i semafori, e così via. Ogni cache è popolata da **oggetti**, istanze della struttura dati del kernel rappresentata dalla cache. La cache che rappresenta i semafori, per esempio, memorizza istanze di oggetti semaforo; quella che rappresenta i descrittori dei processi

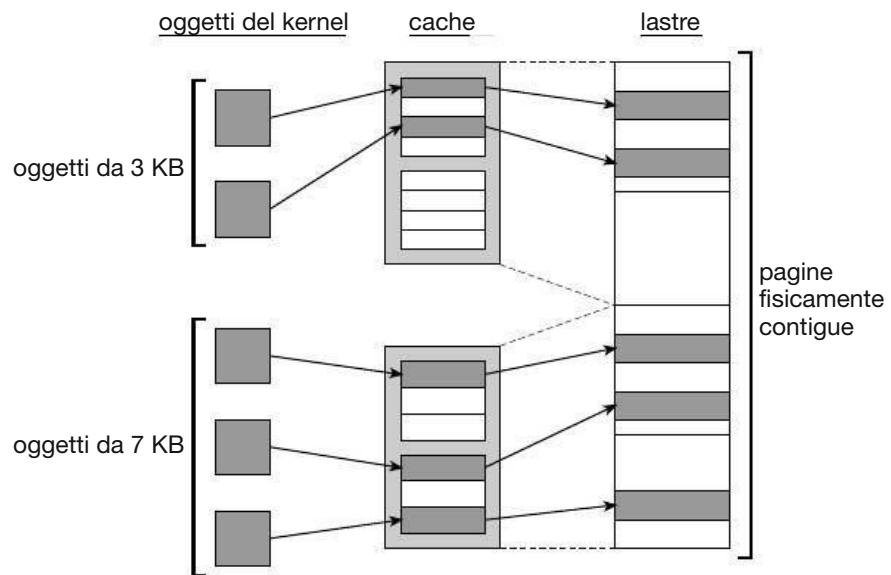


Figura 9.27 Allocazione a lastre (slab).

memorizza istanze di descrittori dei processi, e così via. La relazione fra lastre, cache e oggetti è illustrata nella Figura 9.27; essa mostra due oggetti del kernel che misurano 3 KB e tre oggetti che misurano 7 KB, memorizzati nelle rispettive cache.

L'algoritmo di allocazione a lastre utilizza le cache per memorizzare oggetti del kernel. Quando si crea una cache, un certo numero di oggetti, inizialmente dichiarati liberi, viene assegnato alla cache. Questo numero dipende dalla grandezza della lastra associata alla cache. Per esempio, una lastra di 12 KB (formata da tre pagine contigue di 4 KB) potrebbe contenere sei oggetti di 2 KB ciascuno. Al principio, tutti gli oggetti nella cache sono contrassegnati come **liberi**. Quando una struttura dati del kernel ha bisogno di un oggetto, per soddisfare la richiesta l'allocatore può selezionare dalla cache qualunque oggetto libero; l'oggetto tratto dalla cache è quindi contrassegnato come **usato**.

Consideriamo una situazione in cui il kernel richieda all'allocatore delle lastre la memoria per un oggetto rappresentante un descrittore dei processi. Nei sistemi Linux, un descrittore dei processi ha tipo `struct task_struct`, che richiede circa 1,7 KB di memoria. Quando il kernel di Linux crea un nuovo task, richiede alla propria cache la memoria necessaria per l'oggetto di tipo `struct task_struct`. La cache darà corso alla richiesta impiegando un oggetto di questo tipo che sia già stato allocato in una lastra e rechi il contrassegno “libero”.

In Linux una lastra può essere in uno dei seguenti stati.

1. **Piena.** Tutti gli oggetti della lastra sono contrassegnati come usati.
2. **Vuota.** Tutti gli oggetti della lastra sono contrassegnati come liberi.
3. **Parzialmente occupata.** La lastra contiene oggetti sia usati sia liberi.

L'allocatore a lastre, per soddisfare una richiesta, tenta in primo luogo di estrarre un oggetto libero da una lastra parzialmente occupata; se non ne esistono, assegna un

oggetto libero da una lastra vuota; in mancanza di lastre vuote disponibili, crea una nuova lastra da pagine fisiche contigue e la alloca a una cache; da tale lastra si attinge la memoria da allocare all'oggetto.

L'allocatore a lastre offre due vantaggi principali.

1. Annulla lo spreco di memoria derivante da frammentazione. La frammentazione non è un problema, poiché ogni struttura dati del kernel ha una cache associata; ciascuna delle cache è composta da un numero variabile di lastre, suddivise in spezzoni di grandezza pari a quella degli oggetti rappresentati. Pertanto, quando il kernel richiede memoria per un oggetto, l'allocatore a lastre restituisce la quantità esatta di memoria necessaria per rappresentare l'oggetto.
2. Le richieste di memoria possono essere soddisfatte rapidamente. La tecnica di allocazione a lastre si rivela particolarmente efficace quando, nella gestione della memoria, gli oggetti sono frequentemente allocati e deallocati, come spesso accade con le richieste del kernel. In termini di tempo, allocare e deallocate memoria può essere un processo dispendioso. Tuttavia, gli oggetti sono creati in anticipo e possono dunque essere allocati rapidamente dalla cache. Inoltre, quando il kernel rilascia un oggetto di cui non ha più bisogno, questo è dichiarato libero e restituito alla propria cache, rendendolo così immediatamente disponibile ad altre richieste del kernel.

L'allocatore a lastre ha fatto la sua prima apparizione nel kernel di Solaris 2.4. Per la sua natura generale è ora applicato da Solaris anche ad alcune richieste di memoria in modalità utente. Linux adottava, originariamente, il sistema buddy; tuttavia, a partire dalla versione 2.2, il kernel di Linux include l'allocazione a lastre.

Le distribuzioni recenti di Linux includono altri due allocator di memoria del kernel, SLOB e SLUB. (L'implementazione Linux dell'allocazione a lastre viene chiamata SLAB).

L'allocatore SLOB è progettato per sistemi con una quantità limitata di memoria, come per esempio i sistemi embedded. SLOB (acronimo di simple list of blocks, “semplice lista di blocchi”) funziona mantenendo tre liste di oggetti: piccoli (per gli oggetti più piccoli di 256 byte), medi (per gli oggetti più piccoli di 1.024 byte) e grandi (per gli oggetti oltre i 1.024 byte). Le richieste di memoria sono soddisfatte utilizzando una politica first-fit.

A partire dalla versione 2.6.24, l'allocatore SLUB ha sostituito SLAB come allocator di default per il kernel Linux. SLUB risolve i problemi di prestazioni dell'allocazione a lastre riducendo gran parte dell'overhead richiesto da SLAB. Uno dei cambiamenti costituisce nello spostamento dei metadati archiviati con ogni lastra nell'allocazione SLAB alla struttura page che il kernel Linux utilizza per ogni pagina. Inoltre, in SLUB sono state eliminate le code per singola CPU, mantenute da SLAB per gli oggetti in ogni cache. Per sistemi con un gran numero di processori, la quantità di memoria allocata a queste code non era del tutto insignificante. SLUB fornisce quindi prestazioni migliori al crescere del numero di processori.

9.9 Altre considerazioni

Le due scelte fondamentali nella progettazione dei sistemi di paginazione sono la definizione dell'algoritmo di sostituzione e della politica di allocazione, già analizzate in questo capitolo. Si devono però fare anche molte altre considerazioni, che riportiamo nel seguito.

9.9.1 Prepaginazione

Una caratteristica ovvia per un sistema di paginazione su richiesta puro, consiste nell'alto numero di page fault che si verificano all'avvio di un processo. Questa situazione è dovuta al tentativo di portare la località iniziale in memoria. La stessa situazione si può presentare anche in altri momenti. Per esempio, quando si riavvia un processo scaricato nell'area d'avvicendamento, tutte le sue pagine si trovano nel disco e ognuna di loro deve essere reinserita in memoria tramite la gestione di un proprio page fault. La **prepaginazione** rappresenta un tentativo di prevenire questo alto livello di paginazione iniziale. Alcuni sistemi operativi, e in particolare Solaris, applicano la prepaginazione ai frame dei file di dimensioni modeste.

In un sistema che usi il modello del working set, per esempio, a ogni processo si può associare una lista delle pagine contenute nel suo working set. Se occorre sospendere un processo a causa di un'attesa di I/O oppure dell'assenza di frame liberi, si memorizza il suo working set. Al momento di riprendere l'esecuzione del processo (perché l'I/O è terminato o un numero sufficiente di frame liberi è divenuto disponibile), prima di riavviare il processo, si riporta in memoria il suo intero working set.

In alcuni casi la prepaginazione può essere vantaggiosa. La questione riguarda semplicemente il suo costo, che deve essere inferiore al costo per servire i corrispondenti page faults. Può accadere che molte pagine trasferite in memoria dalla prepaginazione non siano usate.

Si supponga che siano prepaginate s pagine e sia effettivamente usata una frazione αs di queste s pagine ($0 \leq \alpha \leq 1$). Occorre sapere se il costo delle αs eccezioni di page fault risparmiate sia maggiore o minore del costo di prepaginazione di $(1 - \alpha)s$ pagine non necessarie. Se il parametro α è prossimo allo 0, la prepaginazione non è conveniente; se α è prossimo a 1, la prepaginazione certamente lo è.

9.9.2 Dimensione delle pagine

È raro che chi progetta un sistema operativo per un calcolatore esistente possa scegliere le dimensioni delle pagine. Tuttavia, se si devono progettare nuovi calcolatori, occorre stabilire quali siano le dimensioni migliori per le pagine. Come s'intuisce non esiste un'unica dimensione migliore, ma più fattori sono a sostegno delle diverse dimensioni. Le dimensioni delle pagine sono invariabilmente potenze di 2, in genere comprese tra 4096 (2^{12}) e 4.194.304 (2^{22}) byte.

Un fattore da considerare nella scelta delle dimensioni di una pagina è la dimensione della tabella delle pagine. Per un dato spazio di memoria virtuale, diminuendo la dimensione delle pagine aumenta il numero delle stesse e quindi la dimensione

della tabella delle pagine. Per una memoria virtuale di 4 MB (2^{22}), ci sarebbero 4096 pagine di 1024 byte, ma solo 512 pagine di 8192 byte. Poiché ogni processo attivo deve avere la propria copia della tabella delle pagine, conviene che le pagine siano grandi.

D'altra parte, la memoria è utilizzata meglio se le pagine sono piccole. Se a un processo si assegna una porzione della memoria che comincia alla locazione 00000 e continua fino alla quantità di cui necessita, è molto probabile che il processo non termini esattamente sul limite di una pagina, lasciando inutilizzata (le pagine sono unità di allocazione) una parte della pagina finale (frammentazione interna). Supponendo che le dimensioni del processo e delle pagine siano indipendenti è probabile che, in media, metà dell'ultima pagina di ogni processo sia sprecata. Questa perdita è di soli 256 byte per una pagina di 512 byte, ma di 4096 byte per una pagina di 8192 byte. Quindi, per ridurre la frammentazione interna occorrono pagine di piccole dimensioni.

Un altro problema è dovuto al tempo richiesto per leggere o scrivere una pagina. Il tempo di I/O è dato dalla somma dei tempi di posizionamento, latenza e trasferimento. Il tempo di trasferimento è proporzionale alla quantità trasferita, ossia alla dimensione delle pagine; tutto ciò farebbe supporre che siano preferibili pagine piccole. Come però vedremo nel Paragrafo 10.1.1, il tempo di trasferimento è normalmente molto piccolo se è confrontato con il tempo di latenza e il tempo di posizionamento. A una velocità di trasferimento di 2 MB al secondo, per trasferire 512 byte s'impiegano 0,2 millisecondi. D'altra parte, il tempo di latenza è di circa 8 millisecondi e quello di posizionamento 20 millisecondi. Perciò, del tempo totale di I/O (28,2 millisecondi), solo l'1 per cento è attribuibile al trasferimento effettivo. Raddoppiando le dimensioni delle pagine, il tempo di I/O aumenta solo fino a 28,4 millisecondi. S'impiegano 28,4 millisecondi per leggere una sola pagina di 1024 byte, ma 56,4 millisecondi per leggere la stessa quantità di byte su due pagine di 512 byte l'una. Quindi, per ridurre il tempo di I/O occorre avere pagine di dimensioni maggiori.

Tuttavia, con pagine di piccole dimensioni si dovrebbe ridurre l'I/O totale, poiché si migliorano le caratteristiche di località. Pagine di piccole dimensioni permettono di adattarsi con maggior precisione alla località del programma. Si consideri, per esempio, un processo di 200 KB, dei quali solo la metà (100 KB) sono effettivamente usati durante l'esecuzione. Se si dispone di una sola ampia pagina, occorre inserirla tutta, sicché vengono trasferiti e assegnati 200 KB. Disponendo di pagine di 1 byte, si potrebbero invece portare in memoria i soli 100 KB effettivamente usati, con trasferimento e allocazione di quei soli 100 KB. Con pagine di piccole dimensioni è possibile avere una migliore **risoluzione**, che permette di isolare solo la memoria effettivamente necessaria. Se le dimensioni delle pagine sono maggiori, occorre allocare e trasferire non solo quanto è necessario, ma tutto quel che è presente nella pagina, a prescindere dal fatto che sia o meno necessario. Quindi dimensioni più piccole danno come risultato una minore attività di I/O e una minore memoria totale allocata.

D'altra parte occorre notare che con pagine di 1 byte si verifica un page fault per *ciascun* byte. Un processo di 200 KB che usasse solo metà di tale memoria genere-

rebbe un solo page fault con una pagina di 200 KB, ma 102.400 page fault con le pagine di 1 byte. Ciascun page fault causa un rilevante sovraccarico necessario a elaborare l'eccezione, salvare i registri, sostituire una pagina, attendere nella coda del dispositivo di paginazione e aggiornare le tabelle. Per ridurre il numero di page fault al minimo sono necessarie pagine di grandi dimensioni.

Occorre considerare altri fattori, come la relazione tra la dimensione delle pagine e quella dei settori del mezzo di paginazione. Non esiste una risposta ottimale al problema considerato. Alcuni fattori (frammentazione interna, località) sono a favore delle piccole dimensioni, mentre altri (dimensione delle tabelle, tempo di I/O) sono a favore delle grandi dimensioni. Tuttavia la tendenza è storicamente verso l'aumento delle dimensioni delle pagine e questo vale anche per i sistemi mobili. Nella prima edizione di questo testo (1983) si considerava un valore di 4096 byte come limite superiore alla dimensione delle pagine. Nel 1990 tale dimensione delle pagine era la più comune. I sistemi moderni possono impiegare pagine di dimensioni assai maggiori, come vedremo nel paragrafo successivo.

9.9.3 Portata della TLB

Il **tasso di successi** (*hit ratio*) di una TLB – si veda in proposito il Capitolo 8 – si riferisce alla percentuale di traduzioni di indirizzi virtuali risolte dalla TLB anziché dalla tabella delle pagine. Il tasso di successi è evidentemente proporzionale al numero di elementi della TLB, e un modo per migliorarlo è aumentare il numero di voci nella TLB. Tuttavia, la memoria associativa che si usa per costruire le TLB è costosa e consuma molta energia.

Una metrica collegata al tasso di successi, detto **portata della TLB** (*TLB reach*), esprime la quantità di memoria accessibile dalla TLB, ed è dato semplicemente dal numero di elementi moltiplicato per la dimensione delle pagine. Idealmente, la TLB dovrebbe contenere il working set di un processo; altrimenti, il processo passerà molto tempo traducendo riferimenti alla memoria nella page table invece che nella TLB. Se si raddoppia il numero di elementi della TLB, se ne raddoppia la portata; per alcune applicazioni che comportano un uso intensivo della memoria ciò potrebbe rivelarsi ancora insufficiente per la memorizzazione del working set.

Un altro metodo per aumentare la portata della TLB consiste nell'aumentare la dimensione delle pagine oppure nell'impiegare pagine di diverse dimensioni. Se si aumenta la dimensione delle pagine, per esempio da 8 KB a 32 KB, la portata della TLB si quadruplica. Quest'aumento potrebbe però condurre a una maggiore frammenazione della memoria relativamente alle applicazioni che non richiedono pagine così grandi. Come alternativa, un sistema può utilizzare diverse dimensioni di pagina. UltraSPARC è un esempio di architettura che consente diverse dimensioni delle pagine: 8 KB, 64 KB, 512 KB e 4 MB. Di queste possibili dimensioni Solaris impiega sia quella di 8 KB sia quella di 4 MB. Con una TLB a 64 elementi, la portata della TLB per Solaris varia da 512 KB, con tutte le pagine di 8 KB, a 256 MB, con tutte le pagine di 4 MB. Per la maggior parte delle applicazioni una dimensione delle pagine di 8 KB è sufficiente, sebbene Solaris associa i primi 4 MB del codice e dei dati del kernel a

due pagine di 4 MB. Solaris permette anche alle applicazioni, come per esempio i sistemi di gestione delle basi di dati, di trarre vantaggio dalle grandi pagine di 4 MB. Per la maggior parte delle applicazioni è sufficiente la dimensione di 8 KB, sebbene Solaris mappi i primi 4 MB del codice del kernel e dei dati in due pagine da 4 MB. Solaris inoltre consente alle applicazioni – come le basi di dati – di trarre vantaggio dalla dimensione ampia della pagina da 4 MB.

L'uso di diverse dimensioni delle pagine richiede però che la gestione della TLB sia svolta dal sistema operativo e non direttamente dall'hardware. Per esempio, uno dei campi degli elementi della TLB deve indicare la dimensione della pagina fisica cui il contenuto di ciascun elemento fa riferimento. La gestione della TLB svolta dal sistema operativo e non esclusivamente dall'architettura comporta una penalizzazione delle prestazioni. Tuttavia, i vantaggi dovuti all'aumento del tasso di successi e della portata della TLB. Le recenti tendenze indicano infatti un'evoluzione verso TLB gestite dal sistema operativo e verso l'uso di pagine di diverse dimensioni.

9.9.4 Tabella delle pagine invertita

Nel Paragrafo 8.6.3 si è introdotto il concetto di tabella delle pagine invertita come sistema di gestione delle pagine che consente di ridurre la quantità di memoria fisica necessaria per tener traccia della corrispondenza tra gli indirizzi virtuali e gli indirizzi fisici. Tale riduzione si ottiene tramite una tabella con un elemento per pagina fisica, indicizzato dalla coppia *<id-processo, numero di pagina>*.

Poiché contiene informazioni su quale pagina di memoria virtuale è memorizzata in ciascuna pagina fisica, la tabella delle pagine invertita riduce la quantità di memoria fisica necessaria a memorizzare tali informazioni. Tuttavia essa non contiene le informazioni complete sullo spazio degli indirizzi logici di un processo, che sono necessarie se una pagina a cui si è fatto riferimento non è correntemente presente in memoria; la paginazione su richiesta richiede tali informazioni per elaborare le eccezioni di page fault. Per disporre di tali informazioni è necessaria una tabella delle pagine esterna per ciascun processo. Queste tabelle sono simili alle ordinarie tabelle delle pagine di ciascun processo e contengono le informazioni relative alla locazione di ciascuna pagina virtuale.

L'uso delle tabelle esterne delle pagine non pregiudica l'utilità della tabella delle pagine invertita; infatti si fa riferimento alle tabelle esterne solo nel caso di un page fault; quindi non è necessario che siano immediatamente disponibili ed esse stesse sono paginate dentro e fuori dalla memoria quando è necessario. Sfortunatamente, in questo modo un primo page fault può far sì che il gestore della memoria virtuale generi un altro page fault quando carica in memoria la tabella esterna delle pagine per individuare la pagina virtuale nel backing store. Questo caso particolare richiede un'accurata gestione da parte del kernel del sistema operativo e causa un ritardo nell'elaborazione della ricerca della pagina.

9.9.5 Struttura dei programmi

La paginazione su richiesta deve essere trasparente per il programma utente; spesso, l’utente non è neanche a conoscenza della natura paginata della memoria. In altri casi, però, le prestazioni del sistema si possono migliorare se l’utente (o il compilatore) è consapevole della sottostante presenza della paginazione su richiesta.

Come esempio limite, ma esplicativo, ipotizziamo pagine di 128 parole. Si consideri il seguente frammento di programma scritto in C la cui funzione è inizializzare a 0 ciascun elemento di una matrice di 128×128 elementi. Il tipico codice è il seguente:

```
int i, j;  
int[128][128] data;  
  
for (j = 0; j < 128; j++)  
    for (i = 0; i < 128; i++)  
        data[i][j] = 0;
```

Occorre notare che l’array è memorizzato per riga, vale a dire che è disposto in memoria secondo l’ordine `data[0][0]`, `data[0][1]`, ..., `data[0][127]`, `data[1][0]`, `data[1][1]`, ..., `data[127][127]`. In pagine di 128 parole, ogni riga occupa una pagina, quindi il frammento di codice precedente azzera una parola per pagina, poi un’altra parola per pagina, e così via. Se il sistema operativo assegna meno di 128 frame a tutto il programma, la sua esecuzione causa $128 \times 128 = 16.384$ page fault. D’altra parte, cambiando il codice in

```
int i, j;  
int[128][128] data;  
  
for (i = 0; i < 128; i++)  
    for (j = 0; j < 128; j++)  
        data[i][j] = 0;
```

si azzerano tutte le parole di una pagina prima che si inizi la pagina successiva, riducendo a 128 il numero di page fault.

Un’attenta scelta delle strutture dati e delle strutture di programmazione può aumentare la località e quindi ridurre il tasso di page fault e il numero di pagine del working set. Una buona località è quella di uno stack, poiché l’accesso avviene sempre alla sua parte superiore. Una tabella hash, invece, è progettata proprio per distribuire i riferimenti, causando una località non buona. Naturalmente, la località dei riferimenti rappresenta soltanto una misura dell’efficienza d’uso di una struttura dati. Altri fattori rilevanti sono rapidità di ricerca, numero totale dei riferimenti alla memoria e numero totale delle pagine coinvolte.

In uno stadio successivo, anche il compilatore e il loader possono avere un effetto notevole sulla paginazione. La separazione di codice e dati e la generazione di un codice rientrante significano che le pagine di codice si possono soltanto leggere e quindi non vengono modificate. Nel sostituire le pagine non modificate, non occorre scriverle in memoria ausiliaria. Il loader può evitare di collocare procedure lungo i limiti delle pagine, sistemando ogni procedura completamente all'interno di una pagina. Le procedure che si invocano a vicenda più volte si possono “impaccare” nella stessa pagina. Questa forma di impaccamento è una variante del problema del *bin-packing* della ricerca operativa: cercare di impaccare i segmenti di dimensione variabile in pagine di dimensione fissa, in modo da ridurre al minimo i riferimenti tra pagine diverse. Un metodo di questo tipo è utile soprattutto per pagine di grandi dimensioni.

9.9.6 Vincolo di I/O e vincolo delle pagine

Quando si usa la paginazione su richiesta, talvolta occorre permettere che alcune pagine si possano **vincolare** in memoria (*locked in memory*). Una situazione di questo tipo si presenta quando l'I/O si esegue verso o dalla memoria (virtuale) utente. Spesso il sistema di I/O comprende un processore dedicato; al controllore di un dispositivo di memorizzazione USB, per esempio, generalmente si indica il numero di byte da trasferire e un indirizzo di memoria per il buffer (Figura 9.28). Completato il trasferimento, la CPU riceve un segnale d'interruzione.

Occorre essere certi che non si verifichi la seguente successione di eventi: un processo effettua una richiesta di I/O ed è messo in coda per il relativo dispositivo. Nel frattempo si assegna la CPU ad altri processi che accusano page fault e, usando un al-

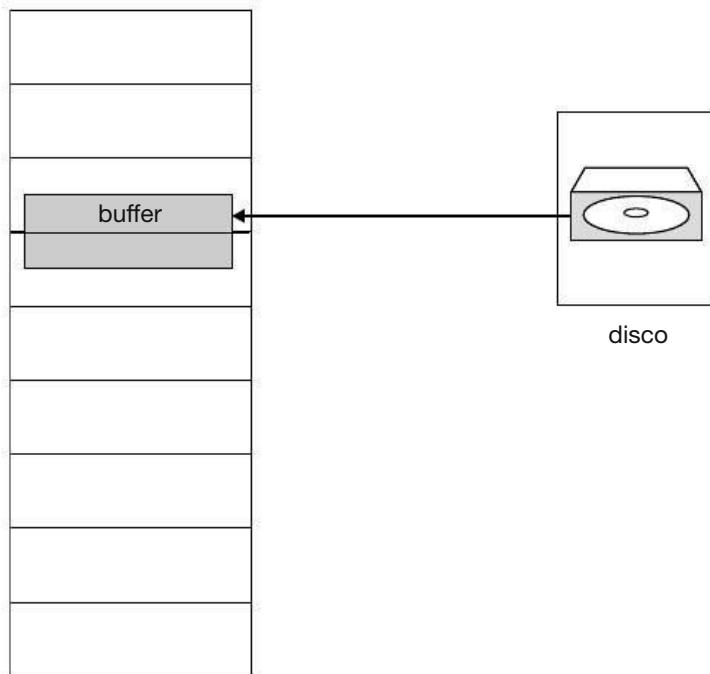


Figura 9.28 Ragione per i cui i frame usati per l'I/O devono essere presenti in memoria.

goritmo di sostituzione globale, uno di questi sostituisce la pagina contenente il buffer di I/O del primo processo, pagina che viene scaricata dalla memoria. Qualche tempo dopo, quando la richiesta di I/O raggiunge la prima posizione della coda d'attesa per il dispositivo, l'operazione di I/O avviene all'indirizzo specificato, ma questo frame è ora impiegato per una pagina appartenente a un altro processo.

Questo problema si può risolvere in due modi. Una soluzione prevede di non eseguire operazioni di I/O in memoria utente, ma di copiare i dati sempre tra la memoria di sistema e la memoria utente. In questo modo l'I/O avviene solo tra la memoria di sistema e il dispositivo di I/O. Per scrivere dati in un nastro, occorre prima copiarli in memoria di sistema, quindi trasferirli all'unità a nastro. Tale copia supplementare può causare un sovraccarico inaccettabile.

Un'altra soluzione consiste nel permettere che le pagine siano vincolate in memoria. A ogni frame si associa un bit di vincolo (*lock bit*); se tale bit è attivato, la pagina contenuta in tale frame non può essere selezionata per la sostituzione. Con questo metodo, per scrivere dati in un nastro occorre vincolare alla memoria le pagine contenenti tali dati, quindi il sistema può continuare come di consueto. Le pagine vincolate non si possono sostituire. Completato l'I/O, si rimuove il vincolo.

I bit di vincolo sono usati in varie situazioni. Spesso il kernel del sistema operativo, o una sua parte, è vincolato alla memoria. La maggior parte dei sistemi non può tollerare un page fault generato dal kernel o da un suo modulo, incluso il modulo incaricato della gestione della memoria. Anche i processi utente possono aver bisogno di vincolare pagine alla memoria. Per esempio, a un processo database potrebbe tornare utile la possibilità di gestire una porzione di memoria, spostando autonomamente blocchi tra il disco e la memoria, perché ha una migliore conoscenza di come vorrà utilizzare i suoi dati. Una simile gestione delle pagine in memoria (*pinning*) è abbastanza comune e la maggior parte dei sistemi operativi dispone di una chiamata di sistema che consente a una applicazione di richiedere che una regione del suo spazio di indirizzamento logico sia vincolata. Si noti che questa caratteristica potrebbe essere usata eccessivamente creando problemi agli algoritmi di gestione della memoria. Per questa ragione un'applicazione ha spesso bisogno di privilegi speciali per poter effettuare una richiesta di questo tipo.

Un altro uso del bit di vincolo riguarda la normale sostituzione di pagine. Si consideri la seguente successione d'eventi: un processo a bassa priorità subisce un page fault. Selezionando un frame per la sostituzione, il sistema di paginazione carica in memoria la pagina necessaria. Pronto per continuare, il processo con priorità bassa entra nella coda dei processi pronti per l'esecuzione e attende l'allocazione della CPU. Giacché si tratta di un processo con bassa priorità, può non essere selezionato dallo scheduler della CPU per un certo tempo. Mentre il processo con priorità bassa attende, un processo ad alta priorità ha un page fault. Durante la ricerca per la sostituzione, il sistema di paginazione individua una pagina in memoria alla quale non sono stati fatti riferimenti o modifiche; si tratta della pagina che il processo con bassa priorità ha appena caricato. Questa pagina sembra una sostituzione perfetta: non è stata modificata, non è necessario scriverla in memoria secondaria e, apparentemente, non è stata usata da molto tempo.

Stabilire se la pagina del processo con bassa priorità si debba sostituire a vantaggio del processo con alta priorità è un problema di politica di gestione. Dopo tutto, si ritarda semplicemente un processo con bassa priorità a vantaggio di quello con priorità alta. D'altra parte, però, si spreca il lavoro fatto per trasferire in memoria la pagina del processo con bassa priorità. Se si vuole evitare che una pagina appena caricata sia sostituita prima che sia usata almeno una volta si può usare il bit di vincolo. Se una pagina viene portata in memoria, si attiva il suo bit di vincolo: tale bit rimane attivato finché si esegue nuovamente il processo che ha avuto il page fault.

Tuttavia, l'uso dei bit di vincolo può essere pericoloso: se un bit non viene mai disattivato, per esempio a causa di un baco del sistema operativo, il frame relativo alla pagina vincolata diventa inutilizzabile. Su un sistema a singolo utente, l'abuso di tale meccanismo può causare danni soltanto allo stesso utente. Ciò non si può consentire nei sistemi multiutente. Il sistema operativo Solaris, per esempio, consente l'impiego di “suggerimenti” (*hint*) di vincolo delle pagine, che si possono però trascurare se l'insieme delle pagine libere diviene troppo piccolo o se un singolo processo richiede che troppe pagine siano vincolate in memoria.

9.10 Esempi di sistemi operativi

In questo paragrafo si descrive la realizzazione della memoria virtuale nei sistemi operativi Windows e Solaris.

9.10.1 Windows

Il sistema operativo Windows realizza la memoria virtuale impiegando la paginazione su richiesta per gruppi di pagine (*demand paging with clustering*). Tale tecnica consiste nel gestire i page fault caricando in memoria, non solo la pagina richiesta, ma più pagine a essa successive. Alla sua creazione, un processo riceve i valori del working set minimo e del working set massimo. Il **working set minimo** è il minimo numero di pagine caricate in memoria di un processo che il sistema garantisce di assegnare; se la memoria è sufficiente, però, il sistema potrebbe assegnare un numero di pagine fino al **working set massimo**. (In alcuni casi è anche possibile superare quest'ultimo valore). Il gestore della memoria virtuale mantiene una lista di pagine fisiche libere, con associato un valore di soglia che indica se è disponibile una quantità sufficiente di memoria libera oppure no. Se si verifica un page fault per un processo che è sotto il suo working set massimo, il gestore della memoria virtuale assegna una pagina dalla lista delle pagine libere; se invece un processo è già al suo massimo e si verifica un page fault, il gestore deve scegliere una pagina da sostituire usando un criterio di sostituzione LRU locale.

Nel caso in cui la quantità di memoria libera scenda sotto la soglia, il gestore della memoria virtuale usa un metodo noto come **regolazione automatica del working set** per riportare il valore sopra la soglia. Si tratta sostanzialmente di valutare il numero di pagine assegnate a ciascun processo; se a un processo sono state assegnate

più pagine del suo working set minimo, il gestore della memoria virtuale rimuove pagine fino a raggiungere quel valore; a un processo che è al suo working set minimo, può assegnare altre pagine prendendole dalla lista delle pagine fisiche libere, non appena è disponibile una quantità sufficiente di memoria libera. Windows realizza la regolazione automatica del working set sia in modalità utente che nel caso dei processi di sistema.

La memoria virtuale sarà trattata molto dettagliatamente nel Capitolo 19 (disponibile sul sito web), quando si approfondirà la conoscenza del sistema Windows.

9.10.2 Solaris

Il sistema operativo Solaris assegna una pagina a un thread ogni volta che si verifica un page fault, prendendola dalla lista delle pagine libere mantenuta dal kernel. È quindi essenziale che il kernel riesca a mantenere una quantità sufficiente di memoria libera. Un parametro, `lotsfree`, associato alla lista delle pagine libere, rappresenta una soglia per l'inizio del processo di paginazione. `lotsfree` è di solito fissato a 1/64 della dimensione della memoria fisica. Il kernel verifica, quattro volte al secondo, se la quantità di memoria libera è inferiore a `lotsfree`. Se il numero di pagine libere scende sotto `lotsfree`, si avvia un processo noto come **pageout**. Questo processo è simile all'algoritmo con seconda chance descritto nel Paragrafo 9.4.5.2, tranne per il fatto che non usa una ma due lancette per scorrere le pagine.

Il suo funzionamento prevede che la prima lancetta scorra lungo tutte le pagine della memoria, azzerandone il bit di riferimento; più tardi, la seconda lancetta esamina il bit di riferimento delle pagine in memoria, ponendo le pagine in cui il bit di riferimento è ancora nullo in coda alla lista delle pagine libere, e scrivendone i contenuti su disco in caso di modifica. Solaris mantiene una lista cache di pagine liberate, ma non ancora sovrascritte. La lista delle pagine libere contiene frame dal contenuto non valido. Le pagine possono essere **richiamate** dalla lista cache nel caso in cui occorra accedervi prima che siano trasferite nella lista delle pagine libere.

Per controllare la frequenza di scansione delle pagine (chiamata anche `scanrate`) l'algoritmo pageout si serve di diversi parametri. Questa frequenza è espressa in pagine al secondo ed è compresa tra i valori `slowscan` e `fastscan`. Quando la memoria libera scende sotto `lotsfree`, la scansione delle pagine avviene alla frequenza `slowscan`, e sale fino a `fastscan` a secondo della quantità di memoria libera disponibile. Il valore predefinito di `slowscan` è 100, mentre `fastscan` è di solito fissato a (*numero totale delle pagine fisiche*)/2 con un massimo di 8192 pagine al secondo. Questa variazione di frequenza è illustrata nella Figura 9.29 (con `fastscan` fissato al massimo).

La distanza (in pagine) tra le lancette dell'orologio è determinata dal parametro di sistema, `handspread`. L'intervallo tra l'azzeramento di un bit da parte della lancetta anteriore e l'esame del suo valore da parte della lancetta posteriore dipende sia da `scanrate` sia da `handspread`. Se il valore di `scanrate` è pari a 100 pagine al

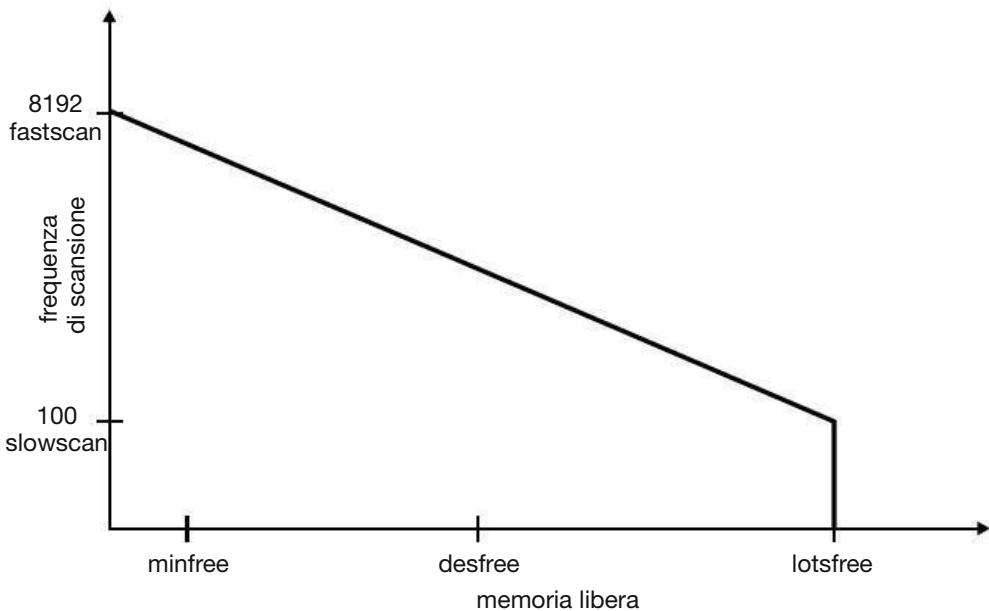


Figura 9.29 Scansione delle pagine in Solaris.

secondo e quello di `handspread` è pari a 1024 pagine, possono passare 10 secondi tra la scrittura di un bit da parte della lancetta anteriore e la sua verifica da parte di quella posteriore. Tuttavia, visti i requisiti imposti a un sistema di memoria, non sono rari valori di `scanrate` di diverse migliaia di pagine al secondo. Ciò significa che l'intervallo tra l'azzeramento e il controllo di un bit è spesso di pochi secondi.

Come si è descritto sopra, il processo `pageout` controlla la memoria quattro volte al secondo. Tuttavia, se la memoria libera scende sotto `desfree` (Figura 9.29) `pageout` sarà eseguito un centinaio di volte al secondo con lo scopo di tenere una quantità di memoria libera almeno pari a `desfree`. Se il processo `pageout` non riesce a mantenere al valore `desfree` la quantità media di memoria libera calcolata in un intervallo di 30 secondi, il kernel incomincia a effettuare lo `swap` di processi, liberando, in questo caso, tutte le pagine assegnate a un processo spostato dalla memoria. In generale, il kernel cerca i processi che sono rimasti inattivi per lunghi periodi. Infine, se il sistema non riesce a mantenere la quantità di memoria libera a `minfree`, invoca il processo `pageout` a ogni richiesta di una nuova pagina.

Le versioni recenti del kernel di Solaris hanno portato alcuni miglioramenti all'algoritmo di paginazione. Uno è il riconoscimento delle pagine che appartengono a librerie condivise da più processi: anche se sono potenzialmente richiedibili per la scansione, sono ignorate durante il processo d'esame delle pagine. Un altro miglioramento riguarda la capacità di distinguere le pagine allocate ai processi da quelle allocate ai file ordinari. Si tratta del meccanismo di **paginazione con priorità** descritto nel Paragrafo 12.6.2.

9.11 Sommario

È auspicabile poter eseguire processi il cui spazio degli indirizzi logici superi quello disponibile per gli indirizzi fisici. La memoria virtuale è una tecnica che permette di associare grandi spazi degli indirizzi logici a quantità più ridotte di memoria fisica. La memoria virtuale è una tecnica che consente di eseguire processi molto grandi e di aumentare il grado di multiprogrammazione, incrementando l'utilizzo della CPU. Inoltre, grazie a tale tecnica, i programmatore di applicazioni non devono più preoccuparsi della disponibilità di memoria. In più, grazie alla memoria virtuale, processi distinti possono condividere librerie di sistema e memoria. La memoria virtuale consente anche l'utilizzo di un tipo efficiente di creazione di processo conosciuto con il nome di copiatura su scrittura (*copy-on-write*), in cui i processi genitore e figlio condividono pagine effettive di memoria.

La memoria virtuale è comunemente implementata tramite paginazione su richiesta. La paginazione su richiesta pura trasferisce in memoria una pagina solo quando si incontra un riferimento alla pagina stessa; il primo riferimento produce un page fault. Il kernel del sistema operativo consulta una tabella interna per stabilire la localizzazione della pagina in memoria ausiliaria, quindi individua un frame libero e vi trasferisce la pagina prelevandola dalla memoria ausiliaria. La tabella delle pagine viene aggiornata per riflettere tale modifica e si riavvia l'istruzione che aveva causato l'eccezione di page fault. Questo metodo permette l'esecuzione di un processo anche se in memoria centrale non è interamente presente la sua immagine di memoria. Finché il tasso di page fault rimane ragionevolmente basso, le prestazioni si considerano accettabili.

La paginazione su richiesta si può usare per ridurre il numero dei frame assegnati a un processo. Questo metodo può aumentare il grado di multiprogrammazione, permettendo che più processi siano disponibili per l'esecuzione in un dato momento e, almeno in teoria, può migliorare l'utilizzo della CPU. Inoltre, consente l'esecuzione di processi i cui requisiti di spazio di memoria superano la memoria fisica disponibile. Tali processi si eseguono in memoria virtuale.

Se i requisiti di spazio di memoria superano la memoria fisica, può essere necessaria la sostituzione di pagine presenti in memoria allo scopo di liberare frame per nuove pagine. Gli algoritmi usati per la sostituzione delle pagine sono diversi: la sostituzione di tipo FIFO è facile da programmare, ma soffre dell'anomalia di Belady; la sostituzione ottimale delle pagine richiede la conoscenza dei futuri riferimenti alla memoria; la sostituzione delle pagine LRU è una approssimazione della politica ottimale, ma può essere di difficile realizzazione. Quasi tutti gli algoritmi di sostituzione delle pagine, come l'algoritmo con seconda chance, sono approssimazioni della sostituzione LRU.

Oltre un algoritmo di sostituzione delle pagine, occorre un criterio di allocazione dei frame. L'allocazione può essere statica, portando a una sostituzione di pagine locale, oppure dinamica, con una sostituzione di pagine globale. Il modello del working set presuppone che i processi siano eseguiti in località. Il working set è l'insieme delle pagine nella località corrente. Di conseguenza, a ogni processo si dovrebbero allocare

frame sufficienti al suo corrente working set. Se un processo non ha spazio di memoria sufficiente per il proprio working set, si ha *thrashing*. Se a ogni processo si devono fornire frame sufficienti per evitare tale degenerazione, sono necessarie attività d'avvicendamento (*swapping*) e scheduling dei processi.

La maggior parte dei sistemi operativi mette a disposizione degli strumenti per la mappatura in memoria dei file, che permettono di trattare gli I/O su file come normali accessi in memoria. La API Win32 implementa la condivisione della memoria tramite la mappatura in memoria di file.

Di solito, i processi del kernel richiedono l'allocazione di pagine fisicamente contigue. Il sistema buddy alloca memoria al kernel in segmenti di dimensioni pari a potenze di 2; ciò conduce facilmente a frammentazione. L'allocazione a lastre assegna le strutture dati del kernel a cache associate a lastre, le quali a loro volta sono costituite da una o più pagine fisiche contigue. Questa strategia non produce frammentazione e permette di servire rapidamente le richieste del kernel.

La corretta progettazione dei sistemi di paginazione non solo richiede la soluzione dei due problemi fondamentali della sostituzione delle pagine e dell'allocazione dei frame, ma porta anche a considerare questioni relative a prepaginazione, dimensione delle pagine, portata del TLB, page table invertite, struttura dei programmi, vincolo di I/O e delle pagine e altro ancora.

Esercizi di ripasso

9.1 In quali circostanze si verifica un page fault? Descrivete le azioni che vengono intraprese dal sistema operativo in questo caso.

9.2 Considerate una successione di riferimenti alle pagine di memoria per un processo con m frame (inizialmente tutti vuoti). La successione ha lunghezza p ; in essa vi sono n distinti numeri di pagina. Rispondete alle seguenti domande relative agli algoritmi di sostituzione delle pagine in generale:

- Qual è un limite inferiore del numero di page fault?
- Qual è un limite superiore del numero di page fault?

9.3 Considerate la tabella delle pagine per un sistema con indirizzi virtuali e fisici a 12 bit con pagine di 256 byte mostrata nella Figura 9.30. La lista dei frame di pagina liberi è D, E, F (dove D è in testa alla lista, E al secondo posto, ed F è in coda).

Convertite i seguenti indirizzi virtuali nei corrispondenti indirizzi fisici, in esadecimale. Tutti i numeri dati sono esadecimali. (Il trattino nella colonna dei frame di pagina indica che la pagina non è in memoria.)

- 9EF
- 111
- 700
- OFF

Pagina	Frame di pagina
0	–
1	2
2	C
3	A
4	–
5	4
6	3
7	–
8	B
9	0

Figura 9.30 Tabella delle pagine per l’Esercizio 9.3.

9.4 Considerate i seguenti algoritmi di sostituzione delle pagine e valutateli basandovi su di una scala a cinque valori da “pessimo” a “ottimo” a seconda del tasso di page fault. Separate gli algoritmi che soffrono dell’anomalia di Belady da quelli che non ne sono affetti:

- sostituzione delle pagine usate meno recentemente (LRU);
- sostituzione delle pagine secondo l’ordine d’arrivo (FIFO);
- sostituzione ottimale;
- sostituzione alla seconda chance.

9.5 Analizzate l’hardware necessario alla paginazione su richiesta.

9.6 Un sistema operativo supporta la memoria virtuale paginata, utilizzando un processore centrale con una durata di ciclo di 1 microsecondo. L’accesso a una pagina diversa da quella corrente richiede 1 ulteriore microsecondo. Le pagine hanno 1.000 parole e lo strumento di paginazione è un tamburo che ruota a 3.000 giri al minuto e trasferisce un milione di parole al secondo. Dal sistema si ottengono le seguenti misurazioni statistiche.

- L’1 per cento di tutte le istruzioni eseguite hanno avuto accesso a una pagina diversa dalla pagina corrente.
- L’80 per cento di queste istruzioni – che hanno cioè avuto accesso a un’altra pagina – hanno avuto accesso a una pagina già in memoria.
- Nel caso in cui sia stata richiesta una nuova pagina, la pagina sostituita è stata modificata nel 50 per cento dei casi.

Calcolate il tempo effettivo di esecuzione delle istruzioni di questo sistema, assumendo che il sistema stia eseguendo un unico processo e che il processore sia fermo durante i trasferimenti dal tamburo.

9.7 Considerate un array bidimensionale A:

```
int A [ ] [ ] = new int[100][100];
```

dove $A[0][0]$ si trova nella posizione 200 in un sistema di memoria paginato con pagine di dimensione 200. Un piccolo processo che manipola la matrice risiede alla pagina 0 (posizioni da 0 a 199); ogni fetch di istruzioni avverrà così dalla pagina 0.

Per tre frame di pagina, quanti page fault vengono generati dai seguenti cicli di inizializzazione dell'array? Utilizzate la sostituzione LRU e ipotizzate che un frame contenga il processo e gli altri due frame siano inizialmente vuoti.

- a.**

```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```
- b.**

```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

9.8 Considerate la seguente successione di riferimenti a pagine di memoria:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Quante eccezioni di page fault si verificherebbero per i seguenti algoritmi di sostituzione, assumendo uno, due, tre, quattro, cinque, sei e sette frame? Ricordate che tutti i frame sono inizialmente vuoti, per cui le vostre prime pagine uniche costeranno un'eccezione ciascuna.

- Sostituzione LRU.
- Sostituzione FIFO.
- Sostituzione ottimale.

9.9 Supponete di voler utilizzare un algoritmo di paginazione che richiede un bit di riferimento (come nella sostituzione alla seconda chance o nel modello del working set) che non viene però fornito dall'hardware. Delineate come potreste simulare un bit di riferimento anche se non fornito dall'hardware, oppure spiegate perché non è possibile mettere in pratica una tale ipotesi. Se possibile, calcolate il costo di questa soluzione.**9.10** Avete progettato un nuovo algoritmo per la sostituzione delle pagine che pensate possa essere ottimale. In alcuni complicati test di controllo si verifica l'anomalia di Belady. L'algoritmo può essere considerato ottimale? Argomentate la vostra risposta.**9.11** La segmentazione è simile alla paginazione, ma utilizza “pagine” di dimensione variabile. Definite due algoritmi di sostituzione dei segmenti basati sugli schemi di sostituzione delle pagine FIFO e LRU. Ricordate che, siccome i segmenti non hanno la stessa dimensione, il segmento scelto per essere sostituito può essere

tropo piccolo per poter contenere abbastanza locazioni di memoria consecutive per il segmento richiesto. Considerate strategie per sistemi nei quali i segmenti non possono essere rilocati e per sistemi nei quali ciò è invece possibile.

9.12 Considerate un sistema informatico a paginazione su richiesta nel quale il grado di multiprogrammazione sia attualmente fissato a quattro. Il sistema è stato recentemente sottoposto a misurazioni volte a determinare l'utilizzo del processore e del disco di paginazione. Le alternative che seguono mostrano tre possibili risultati. Per ognuno di questi casi, che cosa sta avvenendo? Il livello di multiprogrammazione può essere incrementato per migliorare l'utilizzo del processore? La paginazione sta dimostrandosi utile?

- a. Utilizzo del processore 13 per cento; utilizzo del disco 97 per cento.
- b. Utilizzo del processore 87 per cento; utilizzo del disco 3 per cento.
- c. Utilizzo del processore 13 per cento; utilizzo del disco 3 per cento.

9.13 Considerate un sistema operativo per una macchina che utilizza registri base e limite, ma supponete di aver modificato la macchina di modo che metta a disposizione una tabella delle pagine. Si possono configurare le tabelle in modo da simulare registri base e limite? Come? Oppure, perché la cosa non è possibile?

Esercizi

9.14 Supponete che un programma abbia appena fatto riferimento a un indirizzo nella memoria virtuale. Descrivete uno scenario nel quale si verifichi ognuno dei seguenti eventi. (Se non è possibile che si verifichi un particolare scenario, spiegatene il motivo).

- Insuccesso della TLB senza page fault.
- Insuccesso della TLB con page fault.
- Successo della TLB senza page fault.
- Successo della TLB con page fault.

9.15 Una visione semplificata degli stati di un thread è **Pronto** (*Ready*), **Esecuzione** (*Running*) e **Bloccato** (*Blocked*), dove un thread è pronto e in attesa di essere schedulato, oppure è in esecuzione nel processore, oppure è bloccato (per esempio è in attesa di I/O), come illustrato nella Figura 9.31. Assumendo che un thread si trovi nello stato di Esecuzione, rispondete alle seguenti domande, argomentando le risposte.

- a. Il thread cambierà di stato se incorrerà in un page fault? In caso affermativo, quale sarà il nuovo stato?
- b. Il thread cambierà di stato se genererà un insuccesso della TLB che viene risolto nella tabella delle pagine? In caso affermativo, quale sarà il nuovo stato?
- c. Il thread cambierà di stato se il riferimento all'indirizzo viene risolto nella tabella delle pagine? In caso affermativo, quale sarà il nuovo stato?

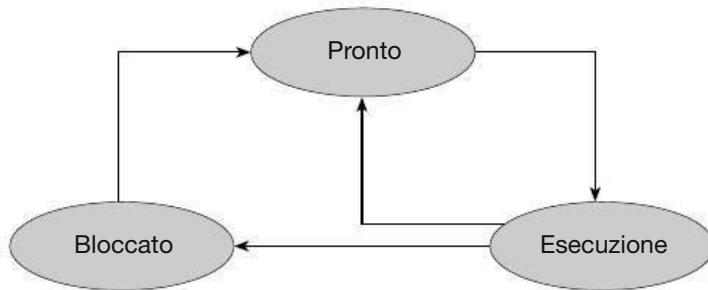


Figura 9.31 Diagramma di stato del thread per l'Esercizio 9.15.

9.16 Considerate un sistema che utilizza la paginazione su richiesta pura.

- Quando un processo inizia a essere eseguito, come caratterizzereste il tasso di page fault?
- Una volta che il working set di un processo viene caricato nella memoria, come caratterizzereste il tasso di page fault?
- Supponete che un processo cambi la propria località e che la dimensione del nuovo working set sia troppo grande per essere caricata nella memoria libera disponibile. Identificate alcune opzioni che i progettisti di sistemi potrebbero scegliere per gestire questa situazione.

9.17 Che cos'è la funzionalità della copiatura su scrittura? In quali circostanze l'uso di tale funzionalità è vantaggioso? Quale hardware è richiesto per implementarla?

9.18 Un elaboratore fornisce ai propri utenti uno spazio di memoria virtuale di 2^{32} byte. L'elaboratore dispone di 2^{22} byte di memoria fisica. La memoria virtuale è implementata tramite paginazione, e la dimensione delle pagine è di 4096 byte. Un processo utente genera l'indirizzo virtuale 11123456. Spiegate in che modo il sistema determina la corrispondente locazione fisica, distinguendo fra operazioni software e hardware.

9.19 Ipotizzate di avere una memoria paginata su richiesta. La tabella delle pagine è conservata in registri. Se un frame vuoto è disponibile o se la pagina sostituita non è modificata, per ovviare alla mancanza di una pagina sono necessari 8 millisecondi, mentre occorrono 20 millisecondi, qualora la pagina sostituita abbia subito modifiche. Il tempo di accesso alla memoria è pari a 100 nanosecondi.

Supponete che la pagina da sostituire subisca modifiche nel 70 per cento dei casi. Per un tempo effettivo di accesso non superiore a 200 nanosecondi, qual è il tasso massimo tollerabile di page fault?

9.20 Quando manca una pagina, il processo che ha richiesto la pagina deve bloccarsi mentre aspetta che la pagina venga portata dal disco alla memoria fisica. Posto che esista un processo con cinque thread a livello utente e che il mapping dei thread utente sui thread del kernel sia di molti a uno, se un thread utente incorre in un page fault quando accede al suo stack, allora anche gli altri thread utente

Pagina	Frame di pagine	Bit di riferimento
0	9	0
1	1	0
2	14	0
3	10	0
4	–	0
5	13	0
6	8	0
7	15	0
8	–	0
9	0	0
10	5	0
11	4	0
12	–	0
13	–	0
14	3	0
15	2	0

Figura 9.32 Tabella delle pagine per l'Esercizio 9.22.

appartenenti allo stesso processo sono coinvolti nel page fault – ossia devono anch'essi aspettare che la pagina mancante venga portata in memoria? Motivate la risposta.

9.21 Considerate la seguente successione di riferimenti alle pagine:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.

Assumendo di utilizzare la paginazione su richiesta con tre frame, quanti page fault si verificherebbero con i tre seguenti algoritmi di sostituzione?

- LRU
- FIFO
- ottimale

9.22 La tabella delle pagine mostrata nella Figura 9.32 è per un sistema con indirizzi virtuali e fisici a 16 bit e con pagine di 4096 byte. Il bit di riferimento è impostato a 1 se c'è stato un riferimento alla pagina. Un thread azzera periodicamente tutti i valori del bit di riferimento. Un trattino nella colonna frame di pagina indica che la pagina non è in memoria. L'algoritmo di sostituzione delle pagine è di tipo LRU locale. Tutti i numeri sono espressi in formato decimale.

- a. Convertite i seguenti indirizzi virtuali (espressi in esadecimale) negli equivalenti indirizzi fisici. Potete fornire le risposte sia in esadecimale che in decimale. Impostate anche il bit di riferimento nella voce appropriata della tabella delle pagine.
- 0xE12C
 - 0x3A9D
 - 0xA9D9
 - 0x7001
 - 0xACA1
- b. Utilizzando come guida i precedenti indirizzi, fornite un esempio di indirizzo logico (espresso in esadecimale), che porta a un page fault.
- c. Da quale insieme di frame di pagina l'algoritmo di sostituzione LRU sceglierà per risolvere un page fault?
- 9.23** Si supponga di monitorare la velocità con cui si muove il puntatore nell'algoritmo a orologio (che indica la pagina candidata per la sostituzione). Che cosa si può inferire sul sistema sapendo che:
- a. il puntatore si muove velocemente;
 - b. il puntatore si muove lentamente.
- 9.24** Esaminate a quali condizioni l'algoritmo di sostituzione delle pagine meno frequentemente usate (LFU) genera un numero inferiore di page fault rispetto all'algoritmo di sostituzione delle pagine usate meno recentemente (LRU). Descrivete anche le circostanze nelle quali è vero il contrario.
- 9.25** Considerate a quali condizioni l'algoritmo di sostituzione delle pagine più frequentemente usate (MFU) genera un numero inferiore di page fault rispetto all'algoritmo di sostituzione delle pagine usate meno recentemente (LRU). Descrivete anche le circostanze nelle quali è vero il contrario.
- 9.26** Il sistema VAX/VMS utilizza un algoritmo di sostituzione FIFO per le pagine residenti, nonché un gruppo di frame liberi costituito dalle pagine recentemente usate. Supponete che il gruppo di frame liberi sia gestito utilizzando il criterio di sostituzione LRU. Rispondete alle seguenti domande.
- a. Se si verifica il fault di una pagina che non si trova nel gruppo di frame, come si genera lo spazio libero per la pagina appena richiesta?
 - b. Se si verifica il fault di una pagina che si trova nel gruppo di frame, come va impostata la pagina residente e in che modo deve essere gestito il gruppo di frame per far spazio alla pagina richiesta?
 - c. In che cosa degenera il sistema di paginazione se il numero delle pagine residenti è impostato a uno?
 - d. In che cosa degenera il sistema di paginazione se il numero delle pagine nel gruppo di frame è zero?

9.27 Considerate un sistema con paginazione su richiesta con le seguenti utilizzazioni:

utilizzo della CPU	20%
disco di paginazione	97,7%
altri dispositivi di I/O	5%

Indicate, fra le seguenti operazioni, quelle che consentono (o è probabile che consentano) di migliorare l'utilizzo della CPU:

- a. installazione di una CPU più veloce;
- b. installazione di un disco di paginazione più grande;
- c. aumento del grado di multiprogrammazione;
- d. riduzione del grado di multiprogrammazione;
- e. installazione di una maggiore quantità di memoria centrale;
- f. installazione di un disco più veloce o di più controllori di unità con dischi multipli;
- g. aggiunta della prepaginazione agli algoritmi di fetch delle pagine;
- h. aumento della dimensione delle pagine.

Motivate le risposte.

9.28 Supponete che una macchina fornisca istruzioni che possono accedere alle locazioni di memoria attraverso lo schema di indirizzamento indiretto a un livello. Quale sequenza di page fault si riscontra allorché tutte le pagine di un programma sono non residenti e la prima istruzione del programma è un'operazione di caricamento indiretto dalla memoria? Che cosa succede se il sistema adopera una tecnica di allocazione dei frame per processo e soltanto due pagine sono allocate al processo in questione?

9.29 Si supponga che il criterio di sostituzione (in un sistema paginato) consista nel controllo regolare delle pagine, una per volta, eliminando ogni pagina che non sia stata usata dopo l'ultimo controllo. Che cosa offre in più tale criterio, e che cosa in meno, se paragonato all'algoritmo di sostituzione LRU o con seconda chance?

9.30 Un algoritmo di sostituzione delle pagine dovrebbe ridurre al minimo il numero di page fault. Questa minimizzazione si può ottenere distribuendo in modo uniforme su tutta la memoria le pagine maggiormente usate, anziché lasciarle competere per un piccolo numero di frame. A ogni frame si può associare un contatore del numero delle pagine relative a quel frame. Quindi, per sostituire una pagina, si cerca il frame con il contatore più basso.

- a. Definite un algoritmo di sostituzione delle pagine che si avvalga di questa idea di base. Affrontate in modo specifico i seguenti problemi:

1. qual è il valore iniziale dei contatori;
2. quando si incrementano i contatori;

3. quando si decrementano i contatori;
4. come si sceglie la pagina da sostituire.
- b. Se sono disponibili quattro frame, dite quanti page fault avvengono per l'algoritmo che avete progettato, con la seguente successione di riferimenti:
1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.
- c. Calcolate il numero minimo di page fault per una strategia di sostituzione delle pagine ottimale per la successione di riferimenti del punto b), con quattro frame.

9.31 Considerate un sistema di paginazione su richiesta con un disco di paginazione che abbia un tempo medio d'accesso e di trasferimento di 20 millisecondi. Gli indirizzi sono tradotti per mezzo di una tabella delle pagine che si trova in memoria centrale, con un tempo d'accesso di un microsecondo per ogni accesso alla memoria. Quindi, ogni riferimento alla memoria per mezzo della tabella delle pagine richiede due accessi. Per migliorare questo tempo, è stata aggiunta una memoria associativa che riduce il tempo d'accesso a un riferimento alla memoria, se l'elemento della tabella delle pagine si trova in memoria associativa.

Supponete che, per l'80 per cento degli accessi, l'elemento relativo si trovi in memoria associativa e che il 10 per cento dei restanti (cioè il 2 per cento del totale) causi un page fault. Calcolate il tempo effettivo d'accesso alla memoria.

9.32 Qual è la causa del thrashing? Come può il sistema accertarlo? E, una volta rivelato questo problema, che cosa può fare per eliminarlo?

9.33 Chiarite se un processo possa avere due working set, uno per rappresentare i dati e l'altro per rappresentare il codice.

9.34 Considerate il parametro Δ usato per definire la finestra del working set nell'ambito del modello omonimo. Impostando Δ a un valore basso, quale effetto ne deriva per la frequenza degli errori dovuti a page fault e per il numero di processi attivi (non sospesi) in esecuzione nel sistema? Qual è l'effetto quando Δ è impostato a un valore molto alto?

9.35 Ipotizzate di avere un segmento iniziale da 1024 KB allocato con il sistema buddy. Seguendo la Figura 9.26 come guida, tracciate l'albero che rappresenta l'allocazione di memoria derivante dalle richieste seguenti:

- richiesta di 6 KB;
- richiesta di 250 byte;
- richiesta di 900 byte;
- richiesta di 1500 byte;
- richiesta di 7 KB.

Modificate adesso l’albero in conformità ai seguenti rilasci di memoria; applicate la fusione ogni volta è possibile:

- rilascio di 250 byte;
- rilascio di 900 byte;
- rilascio di 1500 byte.

9.36 Considerate un sistema in grado di gestire thread sia a livello utente sia a livello kernel. Il mappaggio in questo sistema è di uno a uno (a ogni thread del kernel corrisponde un thread utente). Un processo a più thread consiste allora di (a) un working set per l’intero processo, oppure di (b) un working set per ciascun thread?

9.37 L’algoritmo di allocazione delle lastre (*slab*) riserva una cache a ciascun oggetto di tipo diverso. Assumendo di avere una cache per tipo di oggetto, spiegate perché il metodo non scala bene su sistemi multiprocessore. Quale potrebbe essere la soluzione a tale problema di scalabilità?

9.38 Considerate un sistema che assegna ai propri processi pagine di dimensioni differenti. Quali vantaggi presenta tale schema di paginazione? Quali sono le modifiche da apportare al sistema di memoria virtuale per ottenere questa funzionalità?

Problemi di programmazione

9.39 Scrivete un programma che codifichi gli algoritmi di sostituzione delle pagine FIFO, LRU e ottimale descritti in questo capitolo. Generate una successione di riferimenti casuale, in cui i numeri delle pagine siano compresi tra 0 e 9. Applicate ciascun algoritmo a tale successione e registrate i numeri di page fault che vengono generati. Codificate gli algoritmi di sostituzione delle pagine in modo che il numero dei frame possa variare tra 1 e 7. Supponete l’impiego della paginazione su richiesta.

9.40 Ripetete l’Esercizio 3.22 utilizzando la memoria condivisa di Windows. In particolare, utilizzando la strategia produttore-consumatore, progettate due programmi che comunicano con la memoria condivisa utilizzando l’API di Windows, come descritto nel Paragrafo 9.7.2. Il produttore genera i numeri specificati nella congettura di Collatz e li scrive in un oggetto di memoria condivisa. Il consumatore leggerà quindi la sequenza di numeri dalla memoria condivisa e ne effettuerà l’output.

In questo caso, al produttore sarà passato un parametro intero da riga di comando per specificare quanti numeri produrre (per esempio, passando 5 dalla riga di comando il processo produttore genererà i primi cinque numeri).

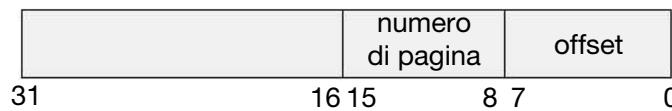


Figura 9.33 Struttura di un indirizzo.

Progetti di programmazione

Progettazione di un gestore di memoria virtuale

Questo progetto consiste nell'implementazione di un programma che traduce indirizzi logici in indirizzi fisici per uno spazio di indirizzamento virtuale di dimensione $2^{16} = 65.536$ byte. Il programma leggerà da un file contenente indirizzi logici e, utilizzando una TLB e una tabella delle pagine, tradurrà ogni indirizzo logico nel corrispondente indirizzo fisico e restituirà il valore del byte memorizzato all'indirizzo fisico. L'obiettivo principale di questo progetto è di simulare i passi necessari per tradurre indirizzi logici in indirizzi fisici.

Specifiche

Il vostro programma legge un file contenente diversi numeri interi a 32 bit che rappresentano indirizzi logici. Tratteremo tuttavia soltanto indirizzi a 16 bit: è quindi necessario mascherare i 16 bit più a destra di ogni indirizzo logico. Questi 16 bit sono suddivisi in (1) un numero di pagina di 8 bit e (2) un offset (scostamento) di pagina di 8 bit. La struttura degli indirizzi è mostrata nella Figura 9.33. Tra le altre specifiche vi sono le seguenti.

- 2^8 elementi nella tabella delle pagine
- Dimensione delle pagine di 2^8 byte
- 16 elementi nella TLB
- Dimensione dei frame di 2^8 byte
- 256 frame
- Memoria fisica di 65.536 byte (256 frame \times 256 byte di dimensione di ogni frame)

Inoltre, il programma deve preoccuparsi soltanto della lettura di indirizzi logici e della loro traduzione nei corrispondenti indirizzi fisici. Non è richiesto di supportare la scrittura sullo spazio di indirizzamento logico.

Traduzione degli indirizzi

Il programma tradurrà indirizzi logici in indirizzi fisici utilizzando una TLB e una tabella delle pagine, come descritto nel Paragrafo 8.5. Per prima cosa viene estratto dall'indirizzo logico il numero di pagina e si consulta la TLB. In caso di successo si ottiene il numero di frame dalla TLB; in caso di insuccesso occorre consultare la tabella delle pagine. In quest'ultimo caso, si ricava dalla tabella delle pagine il numero

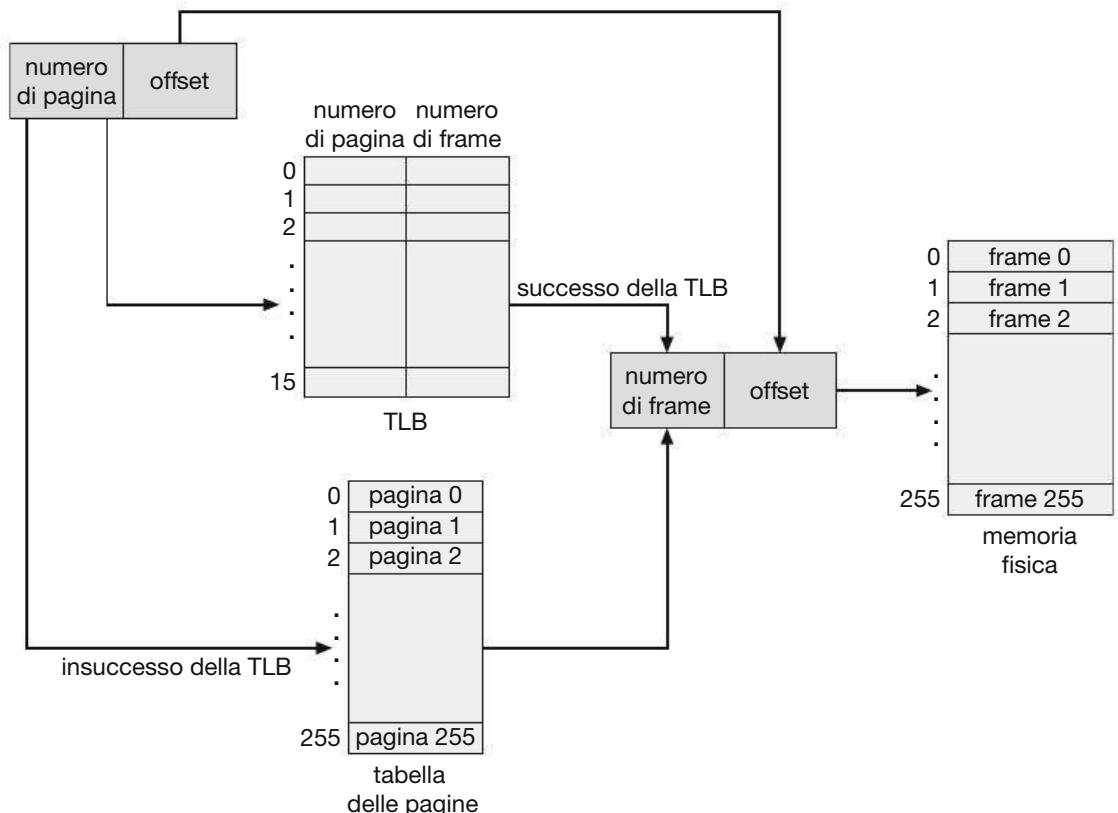


Figura 9.34 Rappresentazione del processo di traduzione degli indirizzi.

di frame, oppure si verifica un page fault. Nella Figura 9.34 viene rappresentato graficamente il processo di traduzione degli indirizzi.

Gestione dei page fault

Il programma implementerà la paginazione su richiesta, come descritto nel Paragrafo 9.2. L'archivio di backup è rappresentato dal file `BACKING_STORE.bin`, un file binario di dimensione 65.536 byte. Quando si verifica un errore di pagina occorre leggere una pagina di 256 byte del `BACKING_STORE` e memorizzarla in un frame disponibile della memoria fisica. Per esempio, se un indirizzo logico con numero di pagina 15 ha provocato un page fault, il programma legge dal `BACKING_STORE` la pagina 15 (ricordate che le pagine cominciano da 0 e hanno dimensione di 256 byte) e salva la pagina in un frame della memoria fisica. Una volta che il frame è memorizzato (e sono aggiornate la tabella delle pagine e la TLB), i successivi accessi alla pagina 15 saranno risolti grazie alla TLB o alla tabella delle pagine.

Sarà necessario trattare `BACKING_STORE.bin` come un file ad accesso casuale in modo da poter accedere in lettura in modo casuale a diverse posizioni nel file. Consigliamo di utilizzare, per realizzare l'I/O, le funzioni della libreria standard C, tra cui `fopen()`, `fread()`, `fseek()` e `fclose()`.

La dimensione della memoria fisica è la stessa della dimensione dello spazio degli indirizzi virtuali (65.536 byte), non c'è quindi bisogno di preoccuparsi della sostitu-

zione di pagina quando si verifica un page fault. Proporremo più avanti una versione modificata del presente progetto in cui si utilizza una minore quantità di memoria fisica. In tal caso sarà necessaria una strategia di sostituzione delle pagine.

File di test

Viene fornito il file `addresses.txt` che contiene valori interi che rappresentano indirizzi logici tra 0 e 65.535 (la dimensione dello spazio di indirizzamento virtuale). Il vostro programma aprirà questo file, leggerà ogni indirizzo logico e lo tradurrà nel corrispondente indirizzo fisico, producendo in output il valore del byte che si trova all'indirizzo trovato.

Come iniziare

In primo luogo, scrivete un semplice programma che estrae il numero di pagina e l'offset (basandovi sulla Figura 9.33) dai seguenti numeri interi:

1, 256, 32768, 32769, 128, 65534, 33153

Probabilmente il modo più semplice per farlo è quello di utilizzare gli operatori per il mascheramento e lo shift dei bit. Una volta che siete riusciti a ricavare correttamente il numero di pagina e l'offset da un numero intero, siete pronti per iniziare.

Suggeriamo di iniziare senza considerare la TLB e utilizzando soltanto una tabella delle pagine. È possibile integrare la TLB in un secondo tempo, una volta che la tabella delle pagine funziona correttamente. Ricordate che la traduzione degli indirizzi può funzionare senza una TLB: la TLB rende solo più veloce il processo. Quando siete pronti per inserire la TLB, ricordate che ha solo 16 voci e che quindi sarà necessario utilizzare una strategia di sostituzione quando si aggiorna una TLB piena. Potete aggiornare la vostra TLB utilizzando una politica FIFO o una politica LRU.

Come eseguire il programma

Il vostro programma deve essere lanciato nel modo seguente:

```
./ a.out addresses.txt
```

Il programma leggerà nel file `addresses.txt`, che contiene 1000 indirizzi logici compresi tra 0 e 65.535, convertirà ogni indirizzo logico in un indirizzo fisico e determinerà il contenuto del byte con segno memorizzato all'indirizzo fisico corretto. Ricordiamo che nel linguaggio C il tipo di dati `char` occupa un byte di memoria: si consiglia dunque di utilizzare valori `char`.

Il vostro programma deve restituire i seguenti valori.

1. L'indirizzo logico che viene tradotto (il valore intero letto dal file `addresses.txt`).
2. L'indirizzo fisico corrispondente (il risultato della traduzione dell'indirizzo logico).
3. Il valore del byte con segno memorizzato all'indirizzo fisico tradotto.

Viene fornito anche il file `correct.txt`, che contiene l'output corretto per il file `addresses.txt`. Dovete utilizzare questo file per verificare se il programma traduce correttamente gli indirizzi logici in indirizzi fisici.

Statistiche

Al termine dell'esecuzione il vostro programma dovrà produrre un report con le seguenti statistiche.

1. Page-fault rate: la percentuale di riferimenti a indirizzi che hanno provocato errori di page fault.
2. Tasso di successo della TLB (*TLB hit rate*): la percentuale di riferimenti a indirizzi che sono stati risolti nella TLB.

Dal momento che gli indirizzi logici in `addresses.txt` sono stati generati casualmente e non seguono alcun principio di località, non aspettatevi di avere un tasso di successo della TLB elevato.

Modifiche

Questo progetto presuppone che la memoria fisica abbia la stessa dimensione dello spazio degli indirizzi virtuali, mentre in realtà la memoria fisica è tipicamente molto più piccola di uno spazio degli indirizzi virtuali. Una modifica che suggeriamo è di usare uno spazio degli indirizzi fisici più piccolo. Consigliamo di utilizzare 128 frame di pagina piuttosto che 256. Questo cambiamento richiede di modificare il vostro programma in modo che tenga traccia dei frame di pagina liberi e che implementi una politica di sostituzione delle pagine di tipo FIFO o LRU (si veda il Paragrafo 9.4).

Note bibliografiche

La paginazione su richiesta è stata usata per la prima volta nel sistema operativo Atlas, realizzato per il calcolatore MUSE della Manchester University intorno al 1960 [Kilburn et al. 1961]. Un altro tra i primi sistemi di paginazione su richiesta è stato MULTICS, per il calcolatore GE 645 [Organick 1972]. La memoria virtuale è stata aggiunta a UNIX nel 1979 [Babaoglu e Joy (1981)]

[Belady et al. 1969] sono stati i primi ricercatori a osservare che la strategia di sostituzione FIFO poteva presentare l'anomalia che porta il nome di Belady. In [Mattson et al. 1970] si dimostra che gli algoritmi a stack non sono soggetti all'anomalia di Belady.

L'algoritmo di sostituzione ottimale è dovuto a [Belady 1966]. In [Mattson et al. 1970] si trova la dimostrazione che esso è ottimale. L'algoritmo ottimale di Belady si usa per l'allocazione statica; [Prieve e Fabry 1976] hanno proposto un algoritmo ottimale per situazioni in cui l'allocazione può variare.

L'algoritmo dell'orologio è trattato in [Carr e Hennessy 1981].

Il modello del working set è stato sviluppato da [Denning 1980], ed è discusso in [Denning 1980].

Lo schema di controllo del tasso di page fault è stato sviluppato da [Wulf 1969], che ha applicato con successo la propria tecnica al calcolatore Burroughs B5500. Gli allocatori di memoria con il sistema buddy sono stati descritti in [Knowlton 1965], [Peterson e Norman 1977], [Purdom Jr. e Stigler 1970]. [Bonwick 1994] ha trattato l'allocatore delle lastre e, insieme con Adams [2001] ha esteso la discussione ai processori multipli. Altri algoritmi del genere sono reperibili in [Stephenson 1983], [Bays 1977] e [Brent 1989]. Per una panoramica delle strategie di allocazione della memoria si può consultare [Wilson et al. 1995].

[Solomon e Russinovich 2000] e [Russinovich e Solomon 2005] descrivono come Windows implementi la memoria virtuale. [Mauro e McDougall 2007] trattano della memoria virtuale di Solaris. Le tecniche di memoria virtuale nei sistemi operativi Linux e FreeBSD sono state descritte, rispettivamente, in [Love 2010] e [McKusick e Neville-Neil 2005]. Le caratteristiche dei sistemi con pagine di dimensioni diverse sono trattate da [Ganapathy e Schimmel 1998], e da [Navarro et al. 2002].

Bibliografia

- [Babaoglu e Joy 1981]** O. Babaoglu e W. Joy, “Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Reference Bits”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 78–86, 1981.
- [Bays 1977]** C. Bays, “A Comparison of Next-Fit, First-Fit and Best-Fit”, Communications of the ACM, Vol. 20, Num. 3, p. 191–192, 1977.
- [Belady 1966]** L. A. Belady, “A Study of Replacement Algorithms for a Virtual-Storage Computer”, IBM Systems Journal, Vol. 5, Num. 2, p. 78–101, 1966.
- [Belady et al. 1969]** L.A.Belady,R. A. Nelson e G. S. Shedler, “An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine”, Communications of the ACM, Vol. 12, Num. 6, p. 349–353, 1969.
- [Bonwick 1994]** J. Bonwick, “The Slab Allocator: An Object-Caching Kernel Memory Allocator”, USENIX Summer, p. 87–98, 1994.
- [Bonwick e Adams 2001]** J. Bonwick e J. Adams, “Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources”, Proceedings of the 2001 USENIX Annual Technical Conference, 2001.
- [Brent 1989]** R. Brent, “Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation”, ACM Transactions on Programming Languages and Systems, Vol. 11, Num. 3, p. 388–403, 1989.
- [Carr e Hennessy 1981]** W. R. Carr e J. L. Hennessy, “WSClock—A Simple and Effective Algorithm for Virtual Memory Management”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 87–95, 1981.
- [Denning 1968]** P. J. Denning, “The Working Set Model for Program Behavior”, Communications of the ACM, Vol. 11, Num. 5, p. 323–333, 1968.
- [Denning 1980]** P. J. Denning, “Working Sets Past and Present”, IEEE Transactions on Software Engineering, Vol. SE-6, Num. 1, p. 64–84, 1980.

- [Ganapathy e Schimmel 1998]** N. Ganapathy e C. Schimmel, “General Purpose Operating System Support for Multiple Page Sizes”, Proceedings of the USENIX Technical Conference, 1998.
- [Kilburn et al. 1961]** T. Kilburn, D. J. Howarth, R. B. Payne e F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, Computer Journal, Vol. 4, Num. 3, p. 222–225, 1961.
- [Knowlton 1965]** K. C. Knowlton, “A Fast Storage Allocator”, Communications of the ACM, Vol. 8, Num. 10, p. 623–624, 1965.
- [Love 2010]** R. Love, *Linux Kernel Development*, 3° Ed., Developer’s Library, 2010.
- [Mattson et al. 1970]** R. L. Mattson, J. Gecsei, D. R. Slutz e I. L. Traiger, “Evaluation Techniques for Storage Hierarchies”, IBM Systems Journal, Vol. 9, Num. 2, p. 78–117, 1970.
- [McDougall e Mauro 2007]** R. McDougall e J. Mauro, *Solaris Internals*, 2° Ed., Prentice Hall, 2007.
- [McKusick e Neville-Neil 2005]** M. K. McKusick e G. V. Neville-Neil, *The Design and Implementation of the FreeBSD UNIX Operating System*, Addison-Wesley, 2005.
- [Navarro et al. 2002]** J. Navarro, S. Lyer, P. Druschel e A. Cox, “Practical, Transparent Operating System Support for SuperP.”, Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, 2002.
- [Organick 1972]** E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
- [Peterson e Norman 1977]** J. L. Peterson e T. A. Norman, “Buddy Systems”, Communications of the ACM, Vol. 20, Num. 6, p. 421–431, 1977.
- [Prieve e Fabry 1976]** B. G. Prieve e R. S. Fabry, “VMIN—An Optimal Variable Space Page-Replacement Algorithm”, Communications of the ACM, Vol. 19, Num. 5, p. 295–297, 1976.
- [Purdom, Jr. e Stigler 1970]** P. W. Purdom, Jr. e S. M. Stigler, “Statistical Properties of the Buddy System”, J. ACM, Vol. 17, Num. 4, p. 683–697, 1970.
- [Russinovich e Solomon 2005]** M. E. Russinovich e D. A. Solomon, *Microsoft Windows Internals*, 4° Ed., Microsoft Press, 2005.
- [Solomon e Russinovich 2000]** D. A. Solomon e M. E. Russinovich, *Inside Microsoft Windows 2000*, 3° Ed., Microsoft Press, 2000.
- [Stephenson 1983]** C. J. Stephenson, “Fast Fits: A New Method for Dynamic Storage Allocation”, Proceedings of the Ninth Symposium on Operating Systems Principles, p. 30–32, 1983.
- [Wilson et al. 1995]** P. R. Wilson, M. S. Johnstone, M. Neely e D. Boles, “Dynamic Storage Allocation: A Survey and Critical Review”, Proceedings of the International Workshop on Memory Management, p. 1–116, 1995.
- [Wulf 1969]** W. A. Wulf, “Performance Monitors for Multiprogramming Systems”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 175–181, 1969.

Gestione della memoria secondaria

Poiché la memoria centrale è in genere troppo piccola per contenere in modo permanente tutti i dati e tutti i programmi, il calcolatore deve disporre di una memoria secondaria a sostegno della memoria centrale. I calcolatori moderni impiegano i dischi come mezzo principale di registrazione delle informazioni, cioè programmi e dati. Il file system fornisce i meccanismi sia per l'accesso ai dati e ai programmi residenti nei dischi sia per la loro registrazione. Un file è una raccolta d'informazioni tra loro correlate definite dal suo creatore. Il sistema operativo gestisce la corrispondenza tra i file e i dispositivi che li contengono fisicamente; i file sono normalmente organizzati in directory che ne facilitano l'uso.

I dispositivi da collegare a un calcolatore possono variare sotto molti aspetti. Alcuni di loro trasferiscono un carattere o un blocco di caratteri per volta. Alcuni prevedono esclusivamente l'accesso sequenziale, altri solo l'accesso casuale. Talvolta il trasferimento dei dati è sincrono, talaltra è asincrono. Esistono dispositivi dedicati, mentre altri possono essere condivisi. E ancora, a differenza di quelli a sola lettura, alcuni dispositivi ammettono sia la lettura sia la scrittura. Pur essendo caratterizzati da notevoli differenze di velocità, i dispositivi rappresentano, nel loro insieme, la componente più lenta e voluminosa dell'elaboratore.

Il sistema operativo, per trattare efficacemente tutte queste varianti, deve offrire alle applicazioni funzionalità che consentano loro un minuzioso controllo dei dispositivi. Uno degli obiettivi cruciali del sottosistema di I/O è fornire la più semplice interfaccia possibile al resto del sistema. Poiché i dispositivi rappresentano un collo di bottiglia per le prestazioni, al fine di sfruttare l'accesso concorrente nel migliore dei modi, l'ottimizzazione dell'I/O è un altro elemento chiave.

CAPITOLO

10

OBIETTIVI DEL CAPITOLO

- Descrizione della struttura fisica dei dispositivi di memorizzazione secondaria e degli effetti che ne derivano per il loro utilizzo.
- Spiegazione delle caratteristiche prestazionali della memoria secondaria.
- Analisi degli algoritmi di scheduling delle unità a disco.
- Analisi dei servizi offerti dal sistema operativo per la memorizzazione secondaria, tra cui il RAID.

Memoria secondaria

Il file system, da un punto di vista logico, si può considerare composto da tre parti: nel Capitolo 11 sarà presentata l’interfaccia per il programmatore e per l’utente del file system; nel Capitolo 12 descriveremo le strutture dati interne e gli algoritmi usati dal sistema operativo per realizzare quest’interfaccia; nel presente capitolo si comincia l’analisi del file system dal livello più basso: la struttura della memoria secondaria. Si descrivono innanzitutto la struttura fisica dei dischi e dei nastri magnetici. Vengono poi discussi gli algoritmi di scheduling delle unità a disco, che ordinano la sequenza delle operazioni di I/O al fine di massimizzarne le prestazioni. Quindi si illustrano la formattazione dei dischi e la gestione dei blocchi d’avviamento, dei blocchi danneggiati e dell’area d’avvicendamento dei processi. Si analizza infine la struttura dei sistemi RAID.

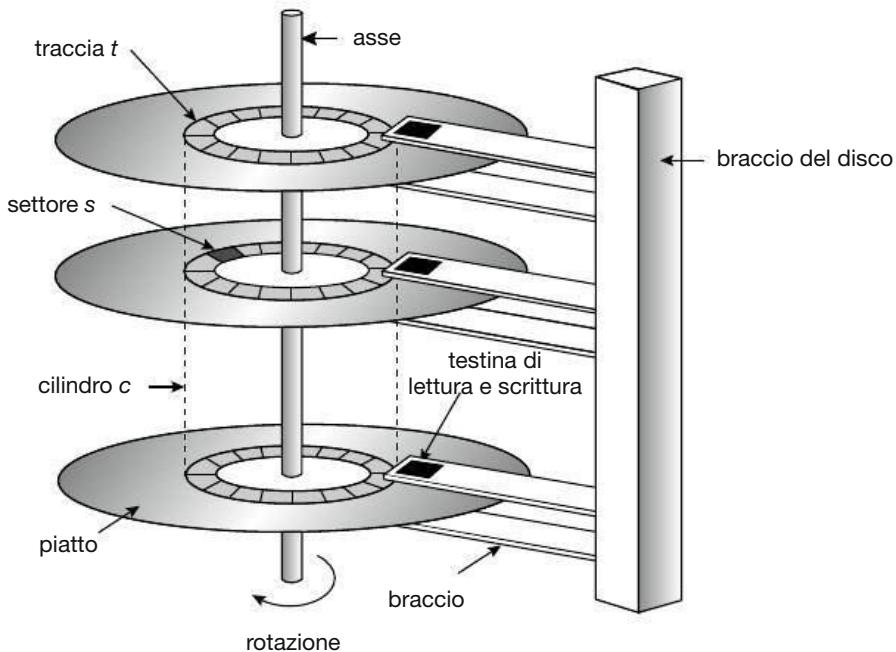


Figura 10.1 Schema di un disco a testine mobili.

10.1 Struttura dei dispositivi di memorizzazione

In questo paragrafo si presenta una rassegna generale della struttura fisica dei dispositivi di memorizzazione secondaria e terziaria.

10.1.1 Dischi magnetici

I **dischi magnetici** sono il supporto fondamentale di memoria secondaria dei moderni sistemi elaborativi. Concettualmente, i dischi (Figura 10.1) sono relativamente semplici: i **piatti** dei dischi hanno una forma piana e rotonda come quella dei CD, con un diametro che comunemente varia tra 1,8 e 3,5 pollici, e le due superfici ricoperte di materiale magnetico; le informazioni si memorizzano registrandole magneticamente sui piatti.

Una testina di lettura e scrittura è sospesa su ciascuna superficie d'ogni piatto. Le testine sono attaccate al **braccio del disco** che le muove in blocco. La superficie di un piatto è divisa logicamente in **tracce** circolari a loro volta suddivise in **settori**; l'insieme delle tracce corrispondenti a una posizione del braccio costituisce un **cilindro**. In un'unità a disco possono esservi migliaia di cilindri concentrici e ogni traccia può contenere centinaia di settori. La capacità di memorizzazione di una comune unità a disco si misura in gigabyte.

Quando un disco è in funzione, un motore lo fa ruotare ad alta velocità; la maggior parte dei dischi ruota a velocità comprese tra 60 e 250 giri al secondo. Questa velocità viene espresa in termini di giri al minuto (RPM): i comuni dischi possono lavorare alle velocità di 5400, 7200, 10000 o 15000 RPM. La velocità di un disco è caratteriz-

zata da due valori: la **velocità di trasferimento**, cioè la velocità con cui i dati fluiscono dall'unità a disco al calcolatore, e il **tempo di posizionamento**, detto anche tempo d'accesso casuale, che consiste di due componenti: il tempo necessario a spostare il braccio del disco in corrispondenza del cilindro desiderato, detto **tempo di ricerca** (*seek time*), e nel tempo necessario affinché il settore desiderato si porti, tramite la rotazione del disco, sotto la testina, detto **latenza di rotazione**. In genere i dischi possono trasferire parecchi megabyte di dati al secondo e hanno un tempo di ricerca e una latenza di rotazione di diversi millisecondi.

Poiché le testine di un disco sono sospese su un cuscino d'aria sottilissimo (dell'ordine dei micron), esiste il pericolo che la testina urti la superficie del disco; in tal caso, nonostante i piatti del disco siano ricoperti da un sottile strato protettivo, la testina può danneggiare la superficie magnetica. Tale incidente, detto **urto della testina**, di solito non può essere riparato e comporta la sostituzione dell'intera unità a disco.

Un disco può essere **rimovibile**: ciò permette che diversi dischi siano montati secondo le necessità. I dischi magnetici rimovibili consistono generalmente in un piatto contenuto in un involucro di materiale plastico, che serve a evitare danni che possono verificarsi quando il disco non è inserito nella propria unità. Sono dischi rimovibili anche i CD, i DVD, i Blu-ray e i dispositivi di memoria flash (che costituiscono una tipologia di drive a stato solido). Un'unità a disco è connessa a un calcolatore attraverso un insieme di fili detto **bus di I/O**; esistono diversi tipi di tale bus, tra i quali i bus ATA (*advanced technology attachment*), SATA (*serial ATA*), eSATA, USB (*universal serial bus*) e FC (*fiber channel*). Il trasferimento dei dati in un bus è eseguito da processori dedicati detti **controllori**: i **controllori di macchina** (*host controller*) sono i controllori posti all'estremità del bus lato calcolatore; i **controllori dei dischi** (*disk controller*) sono incorporati in ciascuna unità a disco. Per eseguire un'operazione di I/O il calcolatore inserisce un comando nell'adattatore, generalmente mediante porte di I/O mappate in memoria, com'è descritto nel Paragrafo 9.7.3; l'adattatore invia il comando mediante messaggi al controllore del disco, che agisce sull'hardware dell'unità a disco per portare a termine il comando. I controllori dei dischi di solito hanno una cache incorporata: il trasferimento dei dati nell'unità a disco avviene tra la cache e la superficie del disco; il trasferimento dei dati tra la cache e l'adattatore avviene alla velocità propria dei dispositivi elettronici.

10.1.2 Dischi a stato solido

A volte le vecchie tecnologie sono utilizzate in modo nuovo grazie ai cambiamenti economici e all'evolversi delle tecnologie stesse. Un esempio è la crescente importanza dei **dischi a stato solido**, o SSD. In breve, un SSD è una memoria non volatile che viene utilizzata come un disco rigido. Ci sono molte varianti di questa tecnologia, dalle DRAM dotate di batteria per permettere di mantenere lo stato in caso di caduta di tensione alle tecnologie di memoria flash come i chip SLC (*single-level cell*) e MLC (*multilevel cell*).

Gli SSD hanno le stesse caratteristiche dei dischi rigidi tradizionali, ma possono essere più affidabili, perché non hanno parti in movimento, e più veloci, perché non hanno tempo di ricerca o di latenza. Inoltre consumano meno energia. Tuttavia il costo al megabyte è superiore rispetto ai dischi rigidi tradizionali, hanno meno capacità rispetto ai dischi rigidi più grandi e possono avere una durata di vita più breve, quindi il loro uso resta un po' limitato. Un utilizzo degli SSD è negli storage array, per mantenere i metadati del file-system che richiedono prestazioni elevate. Vengono utilizzati anche in alcuni computer portatili, per renderli più piccoli, più veloci e più efficienti.

Poiché gli SSD possono essere molto più veloci rispetto alle unità disco magnetiche, le interfacce bus standard possono costituire un notevole limite per il throughput. Alcuni SSD sono progettati per essere collegati direttamente al bus di sistema (PCI, per esempio). Gli SSD stanno cambiando altri aspetti tradizionali nel progetto dei computer. Alcuni sistemi li usano in sostituzione delle unità disco tradizionali, mentre altri li usano come un nuovo livello di cache, spostando i dati tra dischi magnetici, SSD e memoria per ottimizzare le prestazioni.

Nel resto di questo capitolo alcuni paragrafi hanno a che fare con gli SSD, mentre altri non li riguardano. Per esempio, poiché gli SSD non hanno una testina, la maggior parte degli algoritmi di scheduling non si applica a questi dispositivi. I paragrafi che trattano di throughput e formattazione riguardano invece anche gli SSD.

10.1.3 Nastri magnetici

I **nastri magnetici** sono stati i primi supporti di memorizzazione secondaria. Pur avendo la capacità di memorizzare in modo permanente un'enorme quantità di dati, queste unità sono caratterizzate da un tempo d'accesso molto elevato rispetto a quello della memoria centrale e dei dischi magnetici. Inoltre il tempo d'accesso casuale dei nastri magnetici (essendo fisicamente ad accesso sequenziale) è un migliaio di volte maggiore di quello dei dischi magnetici, e ciò li rende inadatti come supporto di memoria secondaria. Gli usi principali dei nastri sono la creazione di copie di backup dei dati, la registrazione di dati poco usati e il trasferimento di informazioni tra diversi sistemi elaborativi.

Il nastro è avvolto in bobine e scorre su una testina di lettura e scrittura. Il posizionamento sul settore richiesto può richiedere alcuni minuti, anche se, una volta raggiunta la posizione desiderata, l'unità a nastro può leggere o scrivere informazioni a una velocità paragonabile a quella di un'unità a disco. La capacità varia secondo il particolare tipo di unità a nastro. I nastri di oggi hanno una capacità di diversi terabyte. Alcune unità sono dotate di funzionalità di compressione dei dati che permettono di più che raddoppiare la capacità effettiva. Le unità a nastro e i loro driver sono solitamente classificate per larghezza: misure tipiche sono 4, 8 o 19 millimetri, e 1/4 o 1/2 pollice. Alcune unità prendono il nome dalla tecnologia su cui si basano, come nel caso di LTO-5 e SDLT.



VELOCITÀ DI TRASFERIMENTO DEL DISCO

Come per molti altri aspetti dell'informatica, le prestazioni pubblicate relative ai dischi non coincidono con le cifre reali: sono sempre inferiori delle **velocità di trasferimento effettive**, per esempio. La velocità di trasferimento può essere considerata la velocità con cui la testina legge i bit dal supporto magnetico, ma questa va distinta dalla velocità con cui i blocchi sono consegnati al sistema operativo.

10.2 Struttura dei dischi

I moderni dischi magnetici sono indirizzati come grandi array monodimensionali di **blocchi logici**, dove un blocco logico è la minima unità di trasferimento. La dimensione di un blocco logico è di solito di 512 byte, sebbene alcuni dischi si possano **formattare a basso livello** allo scopo di ottenere una diversa dimensione dei blocchi logici, per esempio 1024 byte; per altre informazioni su quest'opzione, si veda il Paragrafo 10.5.1. L'array monodimensionale di blocchi logici è mappato in modo sequenziale sui settori del disco: il settore 0 è il primo settore della prima traccia sul cilindro più esterno; la corrispondenza prosegue ordinatamente lungo la prima traccia, quindi lungo le rimanenti tracce del primo cilindro, e così via di cilindro in cilindro dall'esterno verso l'interno.

Sfruttando questa corrispondenza sarebbe possibile – almeno in teoria – trasformare il numero di un blocco logico in un indirizzo fisico di vecchio tipo, consistente in un numero di cilindro, un numero di traccia concernente quel cilindro, e un numero di settore relativo a quella traccia. In pratica, però, vi sono due motivi che rendono difficile quest'operazione: in primo luogo, la maggior parte dei dischi contiene settori difettosi, ma la corrispondenza nasconde questo fatto sostituendo ai settori malfunzionanti settori sparsi in altre parti del disco; in secondo luogo, il numero di settori per traccia in certe unità a disco non è costante.

Nei supporti che impiegano la **velocità lineare costante** (*constant linear velocity*, CLV) la densità di bit per traccia è uniforme. Più è lontana dal centro del disco, tanto maggiore è la lunghezza della traccia, tanto maggiore è il numero di settori che essa può contenere. Spostandosi da aree esterne verso aree più interne il numero di settori per traccia diminuisce. Le tracce nell'area più esterna contengono in genere il 40 per cento in più dei settori contenuti nelle tracce dell'area più interna. L'unità aumenta la sua velocità di rotazione man mano che le testine si spostano dalle tracce esterne verso le tracce più interne per mantenere costante la quantità di dati che scorrono sotto le testine. Questo metodo si usa nelle unità per CD-ROM e DVD. In alternativa la velocità di rotazione dei dischi può rimanere costante, e la densità di bit decresce dalle tracce interne alle tracce più esterne per mantenere costante la quantità di dati che scorre sotto le testine. Questo metodo si usa nelle unità a disco magnetico ed è noto come **velocità angolare costante** (*constant angular velocity*, CAV).

Il numero di settori per traccia cresce con l'evoluzione della tecnologia dei dischi, e l'area più esterna di un disco di solito contiene diverse centinaia di settori per trac-

cia. Anche il numero di cilindri è andato aumentando; le grandi unità a disco contengono decine di migliaia di cilindri.

10.3 Connessione dei dischi

I calcolatori accedono alla memoria secondaria in due modi: nei sistemi di piccole dimensioni il modo più comune è tramite le porte di I/O (**memoria secondaria connessa alla macchina**); oppure ciò avviene tramite un host remoto in un file system distribuito (**memoria secondaria connessa alla rete**).

10.3.1 Memoria secondaria connessa alla macchina

Alla memoria secondaria connessa alla macchina (*host-attached storage*) si accede dalle porte locali di I/O. Queste porte sono disponibili in diverse tecnologie; i comuni PC impiegano un’architettura per il bus di I/O chiamata IDE o ATA. Quest’architettura consente di avere non più di due unità per ciascun bus di I/O. Un protocollo più moderno che prevede un cablaggio più semplice è il SATA.

Le stazioni di lavoro di fascia alta e i server impiegano architetture più raffinate come FC (*fibre channel*), un’architettura seriale ad alta velocità che può funzionare sia su fibra ottica sia su un cavo con 4 conduttori di rame. FC ha due varianti. La prima è una grande struttura di commutazione con uno spazio d’indirizzi a 24 bit. Per il futuro ci si aspetta che questo metodo prevalga, ed è la base per le **storage-area network** (*reti di memoria secondaria*), trattate nel Paragrafo 10.3.3. Grazie al vasto spazio d’indirizzi e alla natura commutata della comunicazione, si possono connettere più macchine e dispositivi di memorizzazione alla struttura a commutazione, permettendo una notevole flessibilità nella comunicazione di I/O. La seconda variante si chiama **FC-AL** (*arbitrated loop*) e può indirizzare fino a 126 dispositivi (unità e controlleri).

C’è un gran numero di dispositivi utilizzabili come memoria secondaria connessa alla macchina, tra questi le unità a disco, le unità RAID, le unità a CD, DVD e a nastri magnetici. I comandi di I/O che avviano trasferimenti di dati a un dispositivo di memoria connessa alla macchina sono letture e scritture di blocchi logici di dati, dirette a unità di memorizzazione specificamente identificate (per esempio, tramite bus ID e unità logica del dispositivo).

10.3.2 Memoria secondaria connessa alla rete

Un dispositivo di memoria secondaria connessa alla rete (*network-attached storage*, NAS) è un sistema di memoria specializzato al quale si accede in modo remoto per mezzo di una rete di trasmissione di dati (Figura 10.2). I client accedono alla memoria connessa alla rete tramite un’interfaccia RPC, come l’NFS nel caso dei sistemi UNIX o CIFS nel caso di sistemi Windows. Le chiamate di procedura remota (RPC) sono realizzate per mezzo dei protocolli TCP o UDP sopra una rete IP (di solito la stessa rete locale che porta tutto il traffico dati ai client). Può quindi risultare più semplice pen-

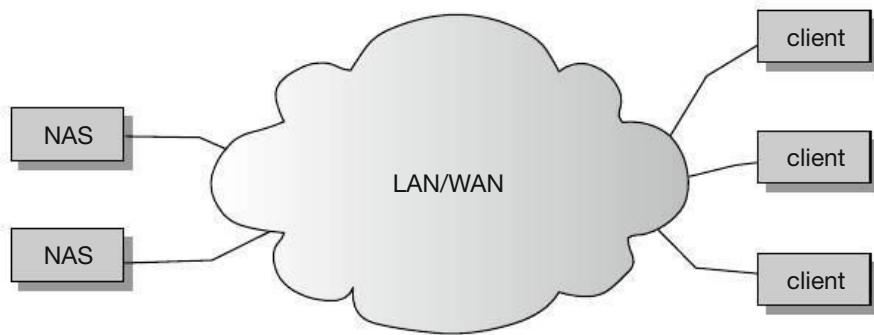


Figura 10.2 Memoria secondaria connessa alla rete.

sare a NAS come a un ulteriore protocollo di accesso alla memoria secondaria. L'unità di memoria è normalmente realizzata come una batteria RAID con programmi di controllo che implementano l'interfaccia per le RPC.

La memoria secondaria connessa alla rete fornisce un modo semplice per condividere spazio di storage per tutti i calcolatori di una LAN, con la stessa facilità di gestione dei nomi e degli accessi caratteristica della memoria secondaria connessa alla macchina. Tuttavia, un sistema di questo genere tende a essere meno efficiente e ad avere prestazioni inferiori rispetto ad alcuni sistemi con connessione diretta alla macchina.

iSCSI è il più recente protocollo per la memoria connessa alla rete. Essenzialmente sfrutta il protocollo IP della rete per il trasporto del protocollo SCSI. Ne consegue la possibilità di usare cavi di rete invece che cavi SCSI per connettere le diverse macchine alla memoria secondaria. Uno dei vantaggi di questa tecnica è che le macchine sono in grado di trattare la memoria secondaria come se fosse direttamente collegata, sebbene possa essere collocata a distanza.

10.3.3 Storage-area network

Uno svantaggio dei sistemi di memoria secondaria connessa alla rete è che le operazioni di I/O sulla memoria secondaria impegnano banda della rete e quindi aumentano la latenza della comunicazione nella rete. Questo problema può essere particolarmente grave per sistemi client-server di grandi dimensioni: l'ordinaria comunicazione tra i server e i client compete per la banda con la comunicazione tra i server e i dispositivi di memorizzazione.

Una storage-area network (SAN) è una rete privata (che impiega protocolli specifici per la memorizzazione anziché protocolli di rete) tra i server e le unità di memoria secondaria (Figura 10.3). La potenza di una SAN sta nella sua flessibilità: si possono connettere alla stessa SAN molte macchine e molti storage array; la memoria può essere allocata alle macchine dinamicamente. Uno switch SAN nega o concede alle macchine l'accesso alla memoria secondaria. Per fare un esempio, è possibile configurarlo di modo che allochi ulteriore memoria secondaria alle macchine che stanno esaurendo lo spazio sui propri dischi. Questi switch rendono anche possibile la condivisione del-

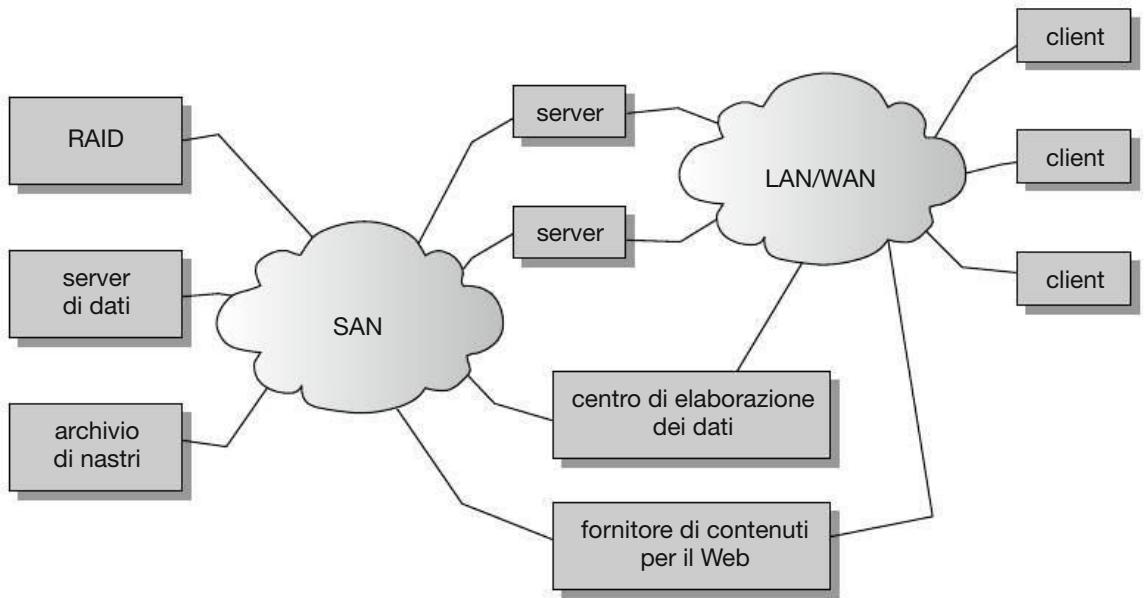


Figura 10.3 Storage area network.

la memoria di massa da parte di cluster di server, e il collegamento diretto di più macchine alla memoria di massa. Gli switch SAN hanno spesso porte in maggior numero, oltre che più costose, dei normali storage array.

La soluzione più comune per il collegamento tramite SAN è il FC, anche se la diffusione di iSCSI è in crescita, grazie alla sua semplicità. Una possibile alternativa è InfiniBand, un'architettura di bus specifica per SAN, che fornisce supporto hardware e software per reti d'interconnessione ad alta velocità tra server e unità di memoria secondaria.

10.4 Scheduling del disco

Una delle responsabilità del sistema operativo è quella di fare un uso efficiente dell'hardware. Nel caso delle unità a disco, far fronte a questa responsabilità significa garantire tempi d'accesso contenuti e ampiezze di banda elevate. Nel caso dei dischi magnetici il tempo d'accesso si può scindere in due componenti principali, come menzionato nel Paragrafo 10.1.1: il **tempo di ricerca** (*seek time*), cioè il tempo necessario affinché il braccio dell'unità a disco sposti le testine fino al cilindro contenente il settore desiderato, e la **latenza di rotazione** (*rotational latency*), e cioè il tempo aggiuntivo necessario perché il disco ruoti finché il settore desiderato si trovi sotto la testina. L'**ampiezza di banda** (*bandwidth*) del disco è il numero totale di byte trasferiti diviso il tempo totale intercorso fra la prima richiesta e il completamento dell'ultimo trasferimento. Gestendo l'ordine delle richieste di I/O relative al disco si possono migliorare sia il tempo d'accesso sia l'ampiezza di banda.

Ogni volta che deve compiere operazioni di I/O con un'unità a disco, un processo effettua una chiamata di sistema.

La richiesta contiene diverse informazioni:

- se l'operazione è di input o di output;
- l'indirizzo nel disco per il trasferimento;
- l'indirizzo di memoria per il trasferimento;
- il numero di settori da trasferire.

Se l'unità a disco desiderata e il controllore sono disponibili, la richiesta si può immediatamente soddisfare; altrimenti le nuove richieste si aggiungono alla coda di richieste inevase relativa a quell'unità. La coda relativa a un'unità a disco in un sistema con multiprogrammazione può spesso essere piuttosto lunga, quindi, al completamento di una richiesta, il sistema operativo sceglie quale fra le richieste inevase conviene servire prima. Come il sistema operativo affronta tale scelta? Per mezzo di uno dei vari algoritmi di scheduling che presentiamo nel seguito.

10.4.1 Scheduling in ordine d'arrivo – FCFS

La forma più semplice di scheduling è, naturalmente, l'algoritmo di servizio secondo l'ordine d'arrivo (*first come, first served*, FCFS). Si tratta di un algoritmo intrinsecamente equo, ma che in generale non garantisce la massima velocità del servizio. Si consideri, per esempio, una coda di richieste per l'unità a disco che riguardino blocchi sui seguenti cilindri (nell'ordine):

98, 183, 37, 122, 14, 124, 65, 67.

Se si trova inizialmente al cilindro 53, la testina dell'unità a disco dovrà prima spostarsi al cilindro 98, poi al 183, 37, 122, 14, 124, 65 e infine al 67, per un movimento totale della testina, misurato in numero di cilindri visitati, di 640 cilindri. La sequenza è rappresentata nella Figura 10.4.

Le defezioni di quest'algoritmo sono evidenziate dal grande salto da 122 a 14 e poi di nuovo a 124: se le richieste per i cilindri 37 e 14 si potessero soddisfare in sequenza, la distanza totale percorsa diminuirebbe notevolmente e le prestazioni migliorerebbero di conseguenza.

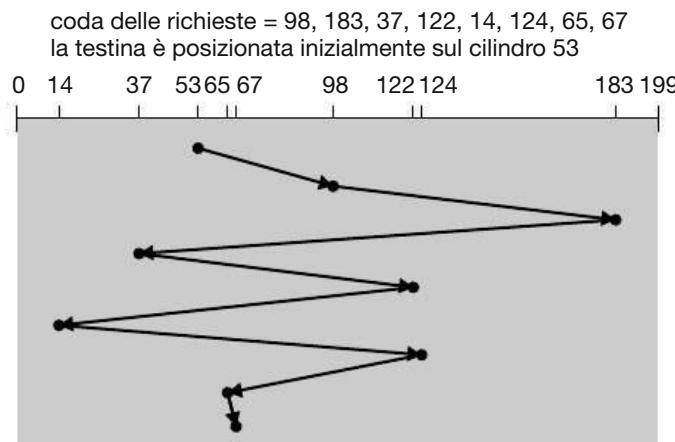


Figura 10.4 Scheduling FCFS.

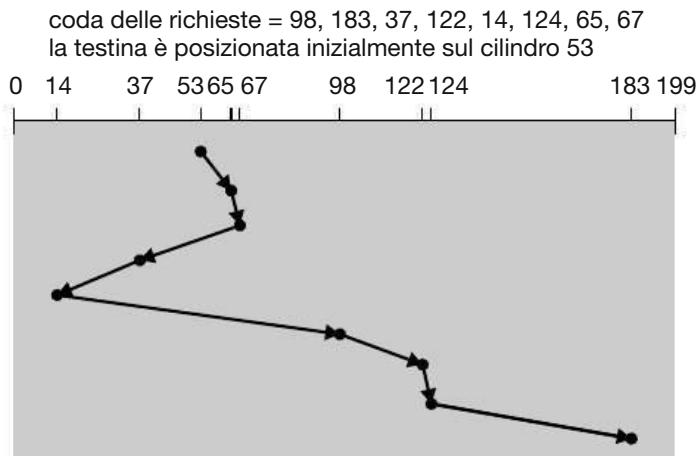


Figura 10.5 Scheduling SSTF.

10.4.2 Scheduling – SSTF

Sembra ragionevole servire tutte le richieste vicine alla posizione corrente della testina prima di spostarla in un’area lontana per soddisfarne altre: questa considerazione è alla base dell’**algoritmo di servizio secondo il più breve tempo di ricerca** (*shortest seek time first*, SSTF). SSTF sceglie la richiesta che dà il minimo tempo di ricerca rispetto alla posizione corrente della testina. In altre parole, l’algoritmo sceglie la richiesta relativa ai cilindri più vicini alla posizione della testina.

Se si considera nuovamente la sequenza di richieste dell’esempio sopra, il cilindro più vicino alla posizione iniziale della testina (cioè 53) è il 65, la successiva richiesta più vicina è quella relativa al cilindro 67; da questo cilindro, la richiesta relativa al cilindro 37 è più vicina di quella relativa al cilindro 98, ed è quindi servita per terza. Continuando allo stesso modo, sarà soddisfatta la richiesta relativa al cilindro 14, poi 98, 122, 124 e infine 183 (Figura 10.5). Questo metodo di scheduling implica un movimento totale della testina di soli 236 cilindri, poco più di un terzo di quello ottenuto con lo scheduling FCFS di questa coda di richieste: esso porta quindi sostanziali miglioramenti d’efficienza.

Lo scheduling SSTF è essenzialmente una forma di scheduling per brevità (*shortest job first*, SJF), e al pari di questo, può condurre a situazioni d’attesa indefinita (*starvation*) di alcune richieste. Si ricordi infatti che nuove richieste possono giungere in qualunque momento: si supponga di avere due richieste in coda, una per il cilindro 14 e l’altra per il 186, e che mentre si sta servendo la richiesta relativa al cilindro 14, arrivi una nuova richiesta per un cilindro vicino al 14. Questa sarà la prossima richiesta soddisfatta, mentre la richiesta per il cilindro 186 dovrà attendere. La stessa situazione potrebbe ripetersi, perché un’altra richiesta relativa a una posizione in prossimità del cilindro 14 potrà giungere mentre si sta servendo la precedente richiesta: in teoria, un flusso continuo di richieste riferite a posizioni vicine fra loro potrebbe causare l’attesa indefinita della richiesta relativa al cilindro 186. Quest’ipotesi diviene sempre più probabile man mano che la coda di richieste si allunga.

Sebbene l'algoritmo SSTF costituisca un miglioramento notevole rispetto al FCFS, esso non è ottimale: si può fare di meglio con uno spostamento dal cilindro 53 al 37, anche se questa non è la minima distanza possibile, e poi al 14, prima di invertire la marcia per servire 65, 67, 98, 122, 124 e 183. L'adozione di questa strategia riduce il movimento totale a 208 cilindri.

10.4.3 Scheduling – SCAN

Secondo l'**algoritmo SCAN** il braccio dell'unità a disco parte da un estremo del disco e si sposta verso l'altro estremo, servendo le richieste mentre attraversa i cilindri, finché non completa il tragitto: a questo punto, il braccio inverte la marcia, e la procedura continua. Le testine attraversano continuamente il disco nelle due direzioni. L'algoritmo SCAN è a volte chiamato **algoritmo dell'ascensore**, perché il braccio dell'unità a disco si comporta proprio come un ascensore che serva prima tutte le richieste in salita e poi tutte quelle in discesa.

Si consideri ancora l'esempio precedente. Prima di poter applicare lo scheduling SCAN alle richieste per i cilindri 98, 183, 37, 122, 14, 124, 65, e 67, oltre la posizione corrente (53), occorre conoscere la direzione del movimento delle testine. Se lo spostamento è nella direzione del cilindro 0, l'unità a disco servirà prima la richiesta 37 e poi la 14; una volta giunto al cilindro 0, il braccio invertirà il movimento verso l'altro estremo del disco, servendo le richieste 65, 67, 98, 122, 124 e 183 (Figura 10.6). Se arriva una nuova richiesta riferita a uno dei cilindri posti davanti alla testina essa sarà quasi immediatamente soddisfatta; ma se la richiesta è riferita a uno dei cilindri appena sorpassati, essa dovrà attendere fino a che la testina non giunga alla fine del disco, inverta la direzione del moto, e torni indietro.

Assumendo una distribuzione uniforme delle richieste per i cilindri da visitare, si consideri la densità di richieste quando il braccio giunge a un estremo e inverte la direzione del moto: in quel momento, relativamente poche richieste sono riferite a ci-

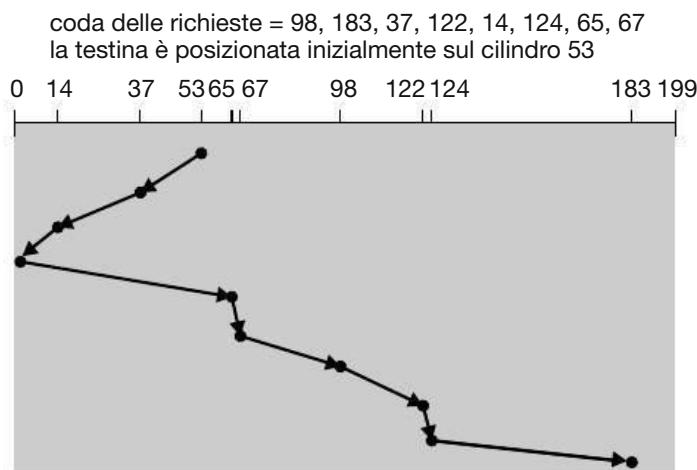


Figura 10.6 Scheduling SCAN.

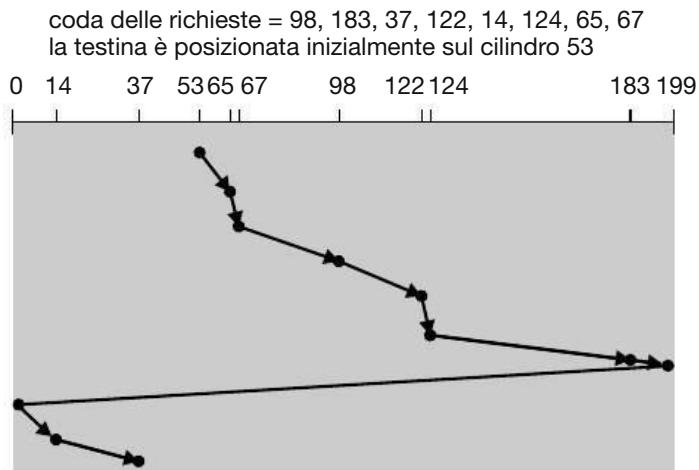


Figura 10.7 Scheduling C-SCAN.

lindri posti appena davanti alla testina, perché i cilindri in questione sono stati recentemente visitati. La massima densità di richieste si riferisce all’altro estremo del disco, e queste richieste sono anche quelle che hanno atteso più a lungo: sembra ragionevole spostarsi subito lì. Questa è l’idea dell’algoritmo seguente.

10.4.4 Scheduling – C-SCAN

L’algoritmo **SCAN circolare** (*circular SCAN*, C-SCAN) è una variante dello scheduling SCAN concepita per garantire un tempo d’attesa meno variabile. Anche l’algoritmo C-SCAN, come lo SCAN, sposta la testina da un estremo all’altro del disco, servendo le richieste lungo il percorso; tuttavia, quando la testina giunge all’altro estremo del disco, ritorna immediatamente all’inizio del disco stesso, senza servire richieste durante il viaggio di ritorno (Figura 10.7). L’algoritmo di scheduling C-SCAN, essenzialmente, tratta il disco come una lista circolare, cioè come se il primo e l’ultimo cilindro fossero adiacenti.

10.4.5 Scheduling LOOK

Secondo la descrizione appena data, sia l’algoritmo SCAN sia il C-SCAN spostano il braccio dell’unità attraverso tutta l’ampiezza del disco; in pratica, nessuno dei due algoritmi è implementato in questo modo: più comunemente, il braccio si sposta solo finché ci sono altre richieste da servire in quella direzione, dopo di che cambia immediatamente direzione, senza giungere all’estremo del disco. Queste versioni dello SCAN e del C-SCAN sono dette **LOOK** e **C-LOOK**, perché “guardano” (in inglese, *look*) se ci sono altre richieste da soddisfare lungo la direzione attuale prima di continuare a spostare la testina in quella direzione (Figura 10.8).

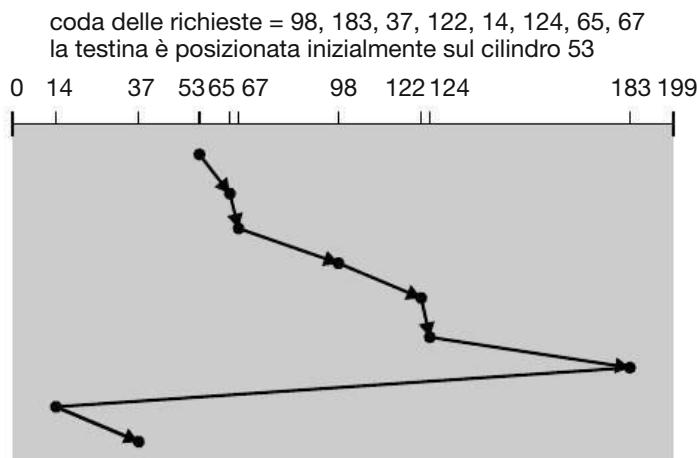


Figura 10.8 Scheduling C-LOOK.

10.4.6 Scelta di un algoritmo di scheduling

Dati tanti diversi algoritmi di scheduling, come scegliere il migliore? Un algoritmo molto comune e naturalmente attraente è l'SSTF poiché aumenta le prestazioni rispetto all'FCFS; lo SCAN e il C-SCAN danno migliori prestazioni in sistemi che sfruttano molto le unità a disco, perché conducono con minor probabilità a situazioni d'attesa indefinita. Per una data lista di richieste si può definire un ordine ottimo di servizio, ma la computazione richiesta può non essere giustificata dal miglioramento in prestazioni rispetto agli algoritmi SSTF o SCAN.

Per qualunque algoritmo di scheduling, le prestazioni dipendono comunque in larga misura dal numero e dal tipo di richieste. Per esempio, si supponga che la coda sia costituita in genere di una sola richiesta inevasa: tutti gli algoritmi danno allora luogo allo stesso comportamento, perché hanno una sola scelta possibile di spostamento della testina. In questo caso, tutti gli algoritmi si comportano come l'FCFS.

Le richieste di I/O per l'unità a disco possono essere notevolmente influenzate dal metodo adottato per l'allocazione dei file. Un programma che legga un file allocato in modo contiguo genererà molte richieste raggruppate, con un conseguente limitato spostamento della testina. Un file con allocazione concatenata o indicizzata, d'altro canto, potrebbe includere blocchi sparsi per tutto il disco, e richiedere quindi un maggiore movimento della testina.

Anche la posizione delle directory e dei blocchi indice è importante: poiché ogni file deve essere aperto per essere usato, e visto che l'apertura di un file richiede una ricerca attraverso la struttura delle directory, vi saranno frequenti accessi alle directory. Si supponga che un elemento di directory risieda nel primo cilindro e che i dati del file relativo si trovino nell'ultimo cilindro; la testina dovrà allora percorrere l'intera ampiezza del disco nel caso di accesso al file in questione. Se l'elemento della directory fosse nel cilindro di mezzo, la testina dovrebbe spostarsi al più della metà dell'ampiezza. Anche l'uso della memoria centrale come cache delle directory e dei

blocchi indice può contribuire a ridurre i movimenti del braccio dell’unità a disco, in particolare quando si tratta di operazioni di lettura.

A causa di queste complicazioni, l’algoritmo di scheduling del disco dovrebbe costituire un modulo a sé stante del sistema operativo, così da poter essere sostituito da un altro algoritmo qualora ciò fosse necessario; come algoritmo di partenza è ragionevole la scelta dell’SSTF o del LOOK.

Gli algoritmi di scheduling descritti tengono conto solamente del tempo di ricerca, mentre nelle moderne unità a disco la latenza di rotazione può essere lunga quasi quanto il tempo medio di ricerca. Tuttavia è difficile per il sistema operativo adottare una strategia di scheduling che porti a miglioramenti dei tempi di latenza di rotazione, perché le moderne unità a disco non rivelano la posizione fisica dei blocchi logici. I produttori di unità a disco hanno collaborato alla limitazione di questo problema incorporando algoritmi di scheduling all’interno dei controllori hardware delle unità a disco: se il sistema operativo invia un gruppo di richieste al controllore, esso può organizzarle in una coda e poi applicare algoritmi di scheduling che riducano sia i tempi di ricerca sia la latenza di rotazione.

Se l’unico elemento di cui tener conto fossero le prestazioni dell’I/O, il sistema operativo scaricherebbe volentieri la responsabilità dello scheduling sull’hardware del disco. In pratica, però, il sistema operativo può dover considerare altri vincoli relativi all’ordine in cui si devono servire le richieste: per esempio, la richiesta di una pagina di memoria virtuale potrebbe avere maggiore priorità rispetto all’I/O delle applicazioni, e le scritture divengono più urgenti delle letture quando la cache sta per esaurire le pagine disponibili. Inoltre, può essere auspicabile mantenere l’ordine naturale delle richieste di scrittura al fine di rendere il file system robusto rispetto ai crash di sistema: si consideri che cosa accadrebbe se il sistema operativoassegnasse un blocco di disco a un file, e un’applicazione scrivesse dati in quel blocco prima che il sistema operativo abbia la possibilità di aggiornare i metadati del file-system sul disco. Per conciliare queste esigenze, il sistema operativo può scegliere di accollarsi la responsabilità dello scheduling del disco e, per alcuni tipi di I/O, fornire le richieste al controllore una alla volta.

SSD E SCHEDULING

Gli algoritmi di scheduling del disco discussi in questo paragrafo hanno come obiettivo principale la riduzione della quantità di movimento della testina del disco nei dischi magnetici. Gli SSD, che non contengono testine in movimento, usano solitamente una semplice politica FCFS. Ad esempio, lo scheduler Noop di Linux utilizza una politica FCFS modificata per fondere richieste adiacenti. Il comportamento osservato degli SSD indica che il tempo richiesto per le letture è uniforme, ma che, a causa delle proprietà della memoria flash, il tempo richiesto per le scritture non è uniforme. Alcuni scheduler per i dischi SSD sfruttano questa proprietà e fondono soltanto le richieste di scrittura adiacenti, mentre le richieste di lettura sono servite in ordine FCFS.

10.5 Gestione dell'unità a disco

Il sistema operativo è anche responsabile di molti altri aspetti della gestione delle unità a disco. In questo paragrafo si discutono l'inizializzazione del disco, l'avviamento del sistema da disco, e la gestione dei blocchi difettosi.

10.5.1 Formattazione del disco

Un disco magnetico nuovo è *tabula rasa*: è semplicemente un piatto ricoperto di materiale magnetico; prima che possa memorizzare dati, deve essere diviso in settori che possano essere letti o scritti dal controllore. Questo processo si chiama formattazione di basso livello, o **formattazione fisica**. La **formattazione di basso livello** riempie il disco con una speciale struttura dati per ogni settore, tipicamente consistente di un'intestazione, un'area per i dati (di solito di 512 byte), e una coda. L'intestazione e la coda contengono informazioni usate dal controllore del disco, per esempio il numero del settore e un **codice per la correzione degli errori** (*error-correcting code*, **ECC**). Quando il controllore scrive dati in un settore nel corso di un'ordinaria operazione di I/O, aggiorna il valore dell'ECC secondo il contenuto dell'area di dati del settore. Quando il controllore legge dati da quel settore, calcola anche l'ECC e lo confronta con il suo valore memorizzato: se risulta una differenza, l'area dei dati del settore non è integra, e il settore del disco potrebbe essere difettoso (si veda il Paragrafo 10.5.3). L'ECC è un codice per la *correzione* degli errori: se solo alcuni bit di dati sono stati alterati, esso contiene sufficienti informazioni affinché il controllore possa identificare i bit in questione e ricalcolare il loro corretto valore. In questo caso il controllore riporta un **errore recuperabile** o **soft error**. Il controllore esegue automaticamente l'elaborazione dell'ECC ogni volta che accede a un settore del disco.

La formattazione fisica dei dischi è eseguita nella maggior parte dei casi dal costruttore come parte del processo produttivo; ciò permette al costruttore di provare il disco, e di inizializzare la corrispondenza fra blocchi logici e settori correttamente funzionanti del disco. In molte unità a disco, quando si richiede al controllore di formattare fisicamente il disco, si può anche specificare il numero di byte delle aree di dati comprese fra l'intestazione e la coda di un settore. La scelta è di solito ristretta a poche opzioni, come 256, 512 o 1024 byte. La formattazione in settori più grandi implica la presenza di meno settori su ogni traccia, ma anche meno intestazioni e code, e quindi maggior spazio per i dati utente. Alcuni sistemi operativi gestiscono solo settori di 512 byte.

Prima di usare un disco come contenitore di file, il sistema operativo deve ancora registrare le proprie strutture dati nel disco, cosa che fa in due passi. Il primo consiste nel suddividere il disco in uno o più gruppi di cilindri, detti **partizioni**. Il sistema operativo può trattare ogni partizione come se fosse un'unità a disco a sé stante: per esempio, una partizione può contenere una copia del codice eseguibile del sistema operativo, mentre un'altra contiene i file degli utenti. Il passo successivo è la **formattazione logica**, cioè la creazione di un file system: il sistema operativo registra nel disco le strutture dati iniziali relative al file system. Le strutture dati in questione pos-

sono includere descrizioni dello spazio libero e dello spazio allocato e una directory iniziale vuota.

Per una maggior efficienza, la maggior parte dei file system accorda i blocchi in gruppi, detti **cluster**. L’I/O del disco procede per blocchi, ma l’I/O del file system procede invece per cluster, di modo che l’I/O abbia caratteristiche di accesso più sequenziale che casuale.

Alcuni sistemi operativi danno l’opportunità a certi programmi speciali di impiegare una partizione del disco come un grande array sequenziale di blocchi logici, non contenente alcuna struttura dati relativa al file system. Questo array è detto a volte **disco di basso livello** (*raw disk*); l’I/O relativo si chiama **I/O di basso livello** (*raw I/O*). Alcuni sistemi per la gestione di basi di dati, per esempio, preferiscono questo tipo di I/O perché permette di controllare l’esatta posizione nel disco di ogni record. Il raw I/O bypassa tutti i servizi del file system: gestione delle *buffer cache*, locking dei file, prefetching, l’allocazione dello spazio, la gestione dei nomi dei file e le directory. È possibile rendere certe applicazioni più efficienti permettendogli di implementare servizi di memorizzazione specializzati che usino una partizione di basso livello, ma la maggior parte delle applicazioni funziona meglio quando usufruisce degli ordinari servizi del file system.

10.5.2 Blocco d’avviamento

Affinché un calcolatore possa entrare in funzione, per esempio quando viene acceso o riavviato, è necessario che esegua un programma iniziale; di solito, questo programma d’avviamento iniziale (*bootstrap*) è piuttosto semplice. Esso inizializza il sistema in tutti i suoi aspetti, dai registri della CPU ai controllori dei dispositivi e al contenuto della memoria centrale, quindi avvia il sistema operativo. Per far ciò, il programma d’avviamento trova il kernel del sistema operativo nei dischi, lo carica nella memoria, e salta a un indirizzo iniziale per avviare l’esecuzione del sistema operativo.

Per la maggior parte dei calcolatori, il programma d’avviamento è memorizzato in una **memoria a sola lettura** (*read-only memory*, ROM), il che è conveniente, perché la ROM non richiede inizializzazione, e ha un indirizzo iniziale fisso dal quale la CPU può cominciare l’esecuzione ogniqualvolta si accende o si riavvia la macchina. Inoltre, visto che la ROM è a sola lettura, non può essere contaminata da un virus informatico. Il problema, però, è che cambiare il programma d’avviamento richiede in questo caso la sostituzione dei circuiti integrati della ROM. A causa di questo inconveniente, molti sistemi memorizzano nella ROM un piccolo caricatore d’avviamento (*bootstrap loader*) il cui solo compito è quello di caricare da un disco il programma di bootstrap completo. Quest’ultimo si può facilmente modificare: se ne scrive semplicemente una nuova versione nel disco. Il programma d’avviamento completo è registrato nei “blocchi di avviamento” in una posizione fissa del disco; un disco contenente una tale partizione si chiama **disco d’avviamento** o **disco di sistema**.

Il codice contenuto nella ROM d’avviamento istruisce il controllore dell’unità a disco affinché trasferisca il contenuto dei blocchi d’avviamento nella memoria (si noti che a questo fine non si carica alcun driver di dispositivo), quindi comincia a ese-

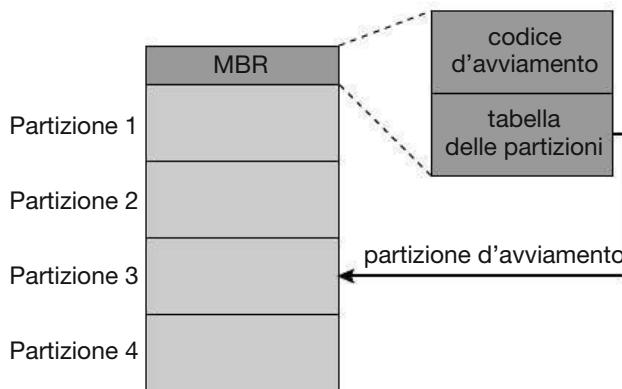


Figura 10.9 Avviamento dal disco di Windows.

guire il codice. Il programma d'avviamento completo è più complesso del suo caricatore, ed è capace di trasferire nella memoria l'intero sistema operativo inizialmente residente in un disco, in una locazione non fissata a priori, e di avviare il sistema operativo stesso. Anche così, il codice di bootstrap è abbastanza breve.

Consideriamo come esempio il processo d'avviamento in Windows. Osserviamo innanzitutto che Windows consente di suddividere il disco rigido in una o più partizioni; in una di esse, detta **partizione d'avviamento**, sono contenuti il sistema operativo e i driver dei dispositivi. Windows colloca il proprio codice d'avviamento nel primo settore del disco rigido, denominato **MBR** (*master boot record*). La procedura d'avviamento inizia con l'esecuzione del codice residente nella memoria ROM del sistema. Questo codice fa sì che il sistema legga il codice d'avviamento dall'MBR. Oltre al codice d'avviamento, l'MBR contiene una tabella che elenca le partizioni del disco rigido e un flag che indica da quale partizione si debba avviare il sistema, come è illustrato nella Figura 10.9. Dopo aver identificato la partizione d'avviamento, il sistema legge da tale partizione il primo settore (chiamato **settore d'avviamento**) e svolge le restanti procedure d'avviamento, tra cui il caricamento dei vari sottosistemi e dei servizi del sistema.

10.5.3 Blocchi difettosi

Le unità a disco sono strutturalmente portate ai malfunzionamenti perché sono costituite da parti mobili con basse tolleranze (si ricordi che una testina è sospesa appena sopra la superficie del disco). A volte si può verificare un guasto irreparabile, e l'unità a disco deve essere sostituita: il suo contenuto dovrà essere recuperato da una copia di backup e trasferito nella nuova unità a disco. Più di frequente, uno o più settori divengono malfunzionanti; in effetti, la maggior parte dei dischi messi in commercio contiene già **blocchi difettosi** (*bad blocks*). Essi sono trattati in diversi modi a seconda del controllore e del tipo di disco.

Nel caso di dischi semplici come quelli gestiti da un controllore IDE, i blocchi difettosi sono gestiti "manualmente". Una possibile strategia consiste nell'effettuare la scansione del disco durante la formattazione, per rilevare la presenza di blocchi di-

fettosi. Se si trova un blocco difettoso lo si marca come inutilizzabile al fine di segnalare alle procedure di allocazione del file system di non usarlo. Se qualche blocco diviene malfunzionante nel corso dell'ordinario uso del sistema, un programma speciale (per esempio il comando Linux `badblocks`) deve essere lanciato manualmente per individuare i blocchi difettosi e isolarli. Di solito, i dati residenti nei blocchi difettosi vanno perduti.

Unità a disco più complesse hanno strategie di recupero dei blocchi difettosi più raffinate. Il controllore mantiene una lista dei blocchi malfunzionanti dell'unità a disco che è inizializzata durante la formattazione fisica eseguita dal produttore, ed è aggiornata per tutto il periodo in cui l'unità a disco è operativa. La formattazione fisica mette anche da parte dei settori di riserva non visibili al sistema operativo: si può istruire il controllore affinché sostituisca da un punto di vista logico un settore difettoso con uno dei settori di riserva inutilizzati. Questa strategia è nota come **accantonamento di settori** (*sector sparing* o *sector forwarding*).

Un tipico esempio di attuazione di questa strategia è il seguente:

- il sistema operativo tenta di leggere il blocco logico 87;
- il controllore calcola l'ECC e scopre che il settore è difettoso; quindi segnala questo malfunzionamento al sistema operativo;
- la volta successiva che il sistema viene riavviato, si esegue un comando speciale al fine di comunicare al controllore la necessità di sostituire il settore difettoso con uno di riserva;
- dopo di ciò, ogni volta che il sistema tenta di leggere il contenuto del blocco 87, il controllore traduce la richiesta nell'indirizzo del settore di rimpiazzo.

Si noti che un tale reindirizzamento da parte del controllore potrebbe inficiare ogni ottimizzazione fornita dall'algoritmo di scheduling del disco del sistema operativo. Per questa ragione la maggior parte dei dischi viene formattata in modo tale da mantenere alcuni settori di riserva in ogni cilindro, e anche un intero cilindro di riserva. Quando un blocco difettoso viene rimappato, il controllore usa settori di riserva presenti nello stesso cilindro ogniqualvolta è possibile.

Un'alternativa all'accantonamento dei settori è data da quei controllori capaci di sostituire i settori difettosi tramite la **traslazione dei settori** (*sector slipping*). Si supponga per esempio che il blocco logico 17 divenga malfunzionante, e che il primo settore di riserva disponibile sia quello successivo al settore 202. La traslazione dei settori sposta in avanti di una posizione tutti i settori dal 17 al 202: quindi, il settore 202 viene copiato sul settore di riserva, il settore 201 sul 202, il 200 sul 201, e così via, fino a che il settore 18 non viene copiato sul 19. Questa traslazione dei settori libera lo spazio del settore 18, e il settore 17 può essere mappato su quest'ultimo.

La sostituzione di un blocco difettoso non è in genere un processo totalmente automatico, perché i dati contenuti nel blocco in questione di solito vanno perduti. I soft error possono attivare un processo in cui viene effettuata una copia dei dati del blocco e il blocco viene messo in riserva o traslato. Un *errore non recuperabile* o **hard error**, tuttavia, causa una perdita di dati. Un file che usava quel blocco deve quindi essere

riparato (per esempio, ricopiandolo da un nastro contenente le copie di riserva), e ciò richiede un intervento manuale.

10.6 Gestione dell'area d'avvicendamento

L'avvicendamento (*swapping*) è stato introdotto, inizialmente, nel Paragrafo 8.2, dove abbiamo trattato lo spostamento di interi processi tra disco e memoria centrale. In quel contesto, l'avvicendamento interviene quando l'ammontare della memoria fisica si abbassa fino al punto di raggiungere la soglia critica e i processi vengono trasferiti dalla memoria all'area d'avvicendamento, per liberare memoria. Nella pratica, pochissimi sistemi operativi moderni realizzano l'avvicendamento nel modo descritto: essi, infatti, combinano l'avvicendamento con tecniche di memoria virtuale (Capitolo 9) ed effettuano lo swapping di pagine, e non necessariamente interi processi. Infatti alcuni sistemi considerano *avvicendamento* e *paginazione* termini intercambiabili, a riprova della moderna tendenza a convergere di questi due concetti.

La **gestione dell'area d'avvicendamento** è un altro compito di basso livello del sistema operativo. La memoria virtuale usa lo spazio dei dischi come estensione della memoria centrale: poiché l'accesso alle unità a disco è molto più lento dell'accesso alla memoria centrale, l'uso dell'area d'avvicendamento riduce notevolmente le prestazioni del sistema. L'obiettivo principale nella progettazione e realizzazione di un'area di avvicendamento è di fornire il migliore throughput per il sistema di memoria virtuale. In questo paragrafo sono trattati l'uso, la collocazione nei dischi e la gestione dell'area d'avvicendamento.

10.6.1 Uso dell'area d'avvicendamento

L'area d'avvicendamento (*area di swapping*) è usata in modi diversi da sistemi operativi diversi, in funzione degli algoritmi di gestione della memoria utilizzati. I sistemi che adottano l'avvicendamento dei processi nella memoria, per esempio, possono usare l'area d'avvicendamento per mantenere l'intera immagine del processo, inclusi i segmenti dei dati e del codice; i sistemi a paginazione, invece, possono semplicemente memorizzarvi pagine non contenute nella memoria centrale. Lo spazio richiesto dall'area d'avvicendamento per un sistema può quindi variare da pochi megabyte a alcuni gigabyte, a seconda della quantità di memoria fisica, della quantità di memoria virtuale che esso deve sostenere, e del modo in cui quest'ultima è usata.

Si noti che una stima per eccesso delle dimensioni dell'area d'avvicendamento è più prudente di una per difetto, perché un sistema che esaurisca l'area d'avvicendamento potrebbe essere costretto a terminare forzatamente i processi o ad arrestarsi completamente: una stima per eccesso spreca spazio dei dischi che si potrebbe usare per i file, ma non provoca altri danni. Alcuni sistemi consigliano la quantità da riservare per l'area d'avvicendamento. Solaris, per esempio, raccomanda di riservare a tal fine uno spazio uguale alla quantità di memoria virtuale che eccede la memoria fisica paginabile. Linux ha in passato suggerito di raddoppiare l'area d'avvicendamento ri-

spetto alla memoria fisica, ma oggi questa limitazione è scomparsa e la maggior parte dei sistemi Linux usa un'area d'avvicendamento notevolmente minore.

Alcuni sistemi operativi, fra i quali Linux, permettono l'uso di aree d'avvicendamento multiple, che includono sia file che partizioni dedicate. Queste aree d'avvicendamento sono poste di solito in unità a disco distinte per distribuire su più dispositivi il carico della paginazione e dell'avvicendamento dei processi gravante sul sistema per l'I/O.

10.6.2 Collocazione dell'area d'avvicendamento

Le possibili collocazioni per un'area d'avvicendamento sono due: all'interno del normale file system, o in una partizione del disco a sé stante. Se l'area d'avvicendamento è semplicemente un grande file all'interno del file system, si possono usare le ordinarie funzioni del file system per crearla, assegnargli un nome, e allocare spazio per essa. Questo approccio, sebbene sia semplice da realizzare, è inefficiente: l'attraversamento della struttura delle directory e delle strutture dati per l'allocazione dello spazio nei dischi richiede tempo, oltre che, almeno potenzialmente, operazioni di accesso aggiuntive ai dischi. La frammentazione esterna può aumentare molto i tempi di swapping, causando seek multipli durante la scrittura o la lettura dell'immagine di un processo. Le prestazioni si possono migliorare impiegando la memoria fisica come cache per le informazioni relative alla posizione dei blocchi, e anche usando strumenti speciali per l'allocazione in blocchi fisicamente contigui del file d'avvicendamento, ma il costo dovuto all'attraversamento delle strutture dati del file system permane.

In alternativa, l'area d'avvicendamento si può creare in un'apposita **partizione del disco non formattata** (*raw partition*): in essa non è presente alcuna struttura relativa al file system e alle directory, ma si usa uno speciale gestore dell'area d'avvicendamento per allocare e rimuovere i blocchi. Esso adotta algoritmi ottimizzati rispetto alla velocità di accesso, e non rispetto allo spazio impiegato su disco, dato che all'area d'avvicendamento (quando viene utilizzata) si accede molto più frequentemente che al file system. La frammentazione interna può aumentare, ma questo prezzo da pagare è ragionevole perché i dati nell'area d'avvicendamento hanno una vita media molto più breve dei file ordinari. Poiché l'area d'avvicendamento è reinizializzata all'avvio del sistema, la frammentazione ha vita breve. Il metodo della raw partition assegna una dimensione fissa all'area d'avvicendamento al momento della creazione delle partizioni del disco, e l'aumento delle dimensioni dell'area d'avvicendamento deve quindi passare attraverso il ripartizionamento del disco (che implica lo spostamento o l'eliminazione e la sostituzione con copie di riserva delle altre partizioni del disco), oppure attraverso la creazione di un'altra area d'avvicendamento in qualche altra unità a disco del sistema.

Alcuni sistemi operativi non adottano una strategia rigida e possono costruire aree d'avvicendamento sia su partizioni specifiche sia all'interno del file system: Linux ne è un esempio. I metodi di gestione e la loro implementazione sono diversi nei due casi, e la scelta fra le due soluzioni è lasciata all'amministratore del sistema: sul piatto

della bilancia pesano da un lato la facilità dell'allocazione e della gestione dell'area d'avvicendamento all'interno del file system, e dall'altro le migliori prestazioni ottenibili grazie all'uso di una partizione non formattata.

10.6.3 Gestione dell'area d'avvicendamento: un esempio

Si può comprendere come l'area d'avvicendamento venga utilizzata ripercorrendo l'evoluzione dello swapping e della paginazione nei vari sistemi UNIX. Il kernel tradizionale di UNIX implementava inizialmente una tecnica d'avvicendamento che spostava interi processi tra regioni contigue del disco e la memoria. Più tardi, con la disponibilità dell'hardware per la paginazione, UNIX progredì verso una combinazione d'avvicendamento e paginazione.

Per migliorare l'efficienza e sfruttare le innovazioni tecnologiche, in Solaris 1 (SunOS) i progettisti cambiarono le tecniche tradizionali di UNIX. Quando un processo va in esecuzione, le pagine contenenti il codice eseguibile sono prelevate dal file system, poste in memoria centrale e, qualora vengano selezionate per essere sovrascritte, eliminate. Rileggere una pagina dal file system è più efficiente che scriverla nell'area d'avvicendamento e rileggerla da lì: l'area d'avvicendamento è adoperata esclusivamente come area di stoccaggio per le pagine della **memoria anonima**, di cui fanno parte la memoria allocata per lo stack, quella per lo heap e i dati non inizializzati dei processi.

Ulteriori cambiamenti sono stati introdotti nelle versioni più recenti di Solaris. Il più importante prevede che l'area d'avvicendamento sia allocata solo quando una pagina è portata fuori dalla memoria fisica, anziché quando la pagina è creata per la prima volta in memoria virtuale. Questa regola produce un miglioramento delle prestazioni nei calcolatori moderni, i quali, rispetto ai sistemi più vecchi, possono contare su una maggiore quantità di memoria fisica e limitare il ricorso alla paginazione.

Così come per Solaris, l'area d'avvicendamento di Linux è utilizzata soltanto per la memoria anonima, ovvero per la memoria che non corrisponde ad alcun file. Linux permette di istituire una o più aree d'avvicendamento, sia in file di avvicendamento (*swap file*) del file system regolare, sia in una partizione dedicata. Un'area d'avvicendamento è formata da una serie di moduli di 4 KB detti **slot delle pagine**, la cui funzione è di conservare le pagine avvicendate. Ogni area d'avvicendamento ha una **mappa d'avvicendamento** (*swap map*) associata, ovvero un array di contatori interi, ciascuno dei quali corrisponde a uno slot nell'area d'avvicendamento. Se un contatore segna il valore 0, la pagina che gli corrisponde è disponibile. Valori superiori a 0 indicano che lo slot è occupato da una delle pagine avvicendate. Il valore del contatore indica il numero di collegamenti alla pagina; se esso è 3, per esempio, allora la pagina avvicendata è associata a tre processi differenti, eventualità possibile nel caso che la pagina rappresenti una regione di memoria condivisa da tre processi. Le strutture dati relative all'avvicendamento nei sistemi Linux sono rappresentate dalla Figura 10.10.

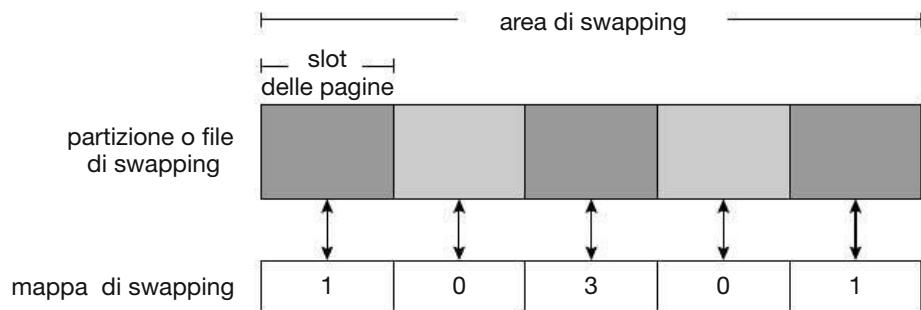


Figura 10.10 Strutture dati per lo swapping nei sistemi Linux.

10.7 Strutture RAID

L’evoluzione tecnologica ha reso le unità a disco progressivamente più piccole e meno costose, tanto che oggi è economicamente possibile equipaggiare un sistema elaborativo con molti dischi. La presenza di più dischi, qualora si possano usare in parallelo, rende possibile l’aumento della frequenza a cui i dati si possono leggere o scrivere. Inoltre, una configurazione di questo tipo permette di migliorare l’affidabilità della memoria secondaria, poiché diventa possibile memorizzare le informazioni in più dischi in modo ridondante. In questo caso, un guasto a uno dei dischi non comporta la perdita di dati. Ci sono varie tecniche per l’organizzazione dei dischi, note col nome comune di **batterie ridondanti di dischi** (*redundant array of independent disks*, RAID), che hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.

Nel passato, strutture RAID composte da piccoli dischi economici erano viste come un’alternativa economicamente vantaggiosa rispetto a costosi dischi di grande capacità; oggi, le strutture RAID s’impiegano per la loro maggiore affidabilità e velocità di trasferimento dei dati, piuttosto che per ragioni economiche. Quindi, la *I* in RAID viene attualmente letta *independent* anziché *inexpensive* com’era originariamente.

STRUTTURA DEI DISPOSITIVI RAID

Le memorie RAID si prestano a essere strutturate con modalità diverse. Un sistema, per esempio, può collegare direttamente i dischi ai propri bus, nel qual caso la funzionalità RAID può essere realizzata dal sistema operativo o dai programmi di sistema. In alternativa, un controllore intelligente può gestire diversi dischi collegati alla macchina e implementare una struttura RAID per quei dischi a livello hardware. Infine, si può ricorrere a una **batteria RAID** (RAID array), un’unità a sé stante, dotata di un controllore, di una cache (nella maggioranza dei casi) e di dischi autonomi. L’array è collegato alla macchina attraverso uno o più controllori (ad esempio FC). Questa diffusa organizzazione consente a programmi e sistemi operativi di per sé privi della funzionalità RAID di usufruirne comunque. Persino sistemi che, in realtà, implementano le funzionalità RAID a livello software adottano a volte la tecnica descritta, per via della sua semplicità e flessibilità.

10.7.1 Miglioramento dell'affidabilità tramite la ridondanza

Consideriamo in primo luogo l'affidabilità dei RAID. La possibilità che uno dei dischi in un insieme di n dischi si guasti è molto più alta della possibilità che uno specifico disco presenti un guasto. Si supponga che il **tempo medio di guasto** di un singolo disco sia 100.000 ore. In questo caso, il tempo medio di guasto per un qualsiasi disco in una batteria di 100 dischi sarebbe $100.000/100 = 1000$ ore, o 41,66 giorni: non molto tempo! Se si memorizzasse una sola copia dei dati, allora ogni guasto di un disco comporterebbe la perdita di una notevole quantità di dati; una frequenza di perdita di dati così alta sarebbe inaccettabile.

La soluzione al problema dell'affidabilità sta nell'introdurre una certa **ridondanza**, cioè nel memorizzare informazioni che non sono normalmente necessarie, ma che si possono usare nel caso di un guasto a un disco per ricostruire le informazioni perse.

Il metodo più semplice (ma anche il più costoso) di introduzione di ridondanza è quello della duplicazione di ogni disco. Questo metodo è detto **mirroring** (*copiatura speculare*): ogni disco logico consiste di due dischi fisici e ogni scrittura si effettua in entrambi i dischi. Si ottiene un disco duplicato (detto **mirrored volume**). Se uno dei dischi si guasta, i dati si possono leggere dall'altro. I dati si perdono solo se il secondo disco si guasta prima della sostituzione del disco già guasto.

Il tempo medio di guasto di un disco duplicato, dove per *guasto* s'intende ora la perdita di dati, dipende da due fattori: il tempo medio di guasto di un singolo disco e il **tempo medio di riparazione**, cioè il tempo richiesto (in media) per sostituire un disco guasto e ripristinarvi i dati. Supponendo che i possibili guasti dei due dischi siano **indipendenti**, vale a dire che il guasto di un disco non sia mai legato a quello dell'altro, se il tempo medio di guasto di un singolo disco è 100.000 ore e il tempo medio di riparazione è di 10 ore, allora il tempo medio di perdita di dati di un sistema con mirroring è $100.000^2/(2 * 10) = 500 * 10^6$ ore, che corrispondono a 57.000 anni!

Occorre però notare che non è possibile assumere l'ipotesi di indipendenza tra i guasti dei dischi. Improvvisi cali di tensione e disastri naturali, quali terremoti, incendi e alluvioni, danneggerebbero con tutta probabilità entrambi i dischi. Inoltre, difetti di fabbricazione in una partita di dischi possono causare guasti correlati. Con l'invecchiamento del disco, la probabilità di un guasto aumenta, accrescendo la probabilità che un secondo disco si guasti mentre il primo è in riparazione. Tuttavia, nonostante tutte queste considerazioni, i sistemi con mirroring offrono un'affidabilità assai più alta dei sistemi a disco singolo.

I casi di caduta di alimentazione elettrica costituiscono un problema particolarmente sentito, poiché avvengono con una frequenza molto più alta dei disastri naturali. Anche impiegando il mirroring, se si sta svolgendo un'operazione di scrittura nello stesso blocco in entrambi i dischi e si verifica una caduta di alimentazione prima che sia completata la scrittura dell'intero blocco, i due blocchi possono ritrovarsi in uno stato incoerente. Una soluzione prevede la scrittura di una delle due copie e solo successivamente la scrittura della seconda. Un'altra soluzione è aggiungere una memoria non volatile a stato solido (NVRAM, *non-volatile RAM*) alla batteria RAID, protetta dalla perdita di dati causata dalle cadute di alimentazione: se è dotata di forme

di correzione d’errore come ECC o mirroring, la scrittura dei dati nella cache può essere considerata completa anche in quei casi.

10.7.2 Miglioramento delle prestazioni tramite il parallelismo

Vediamo come l’accesso in parallelo a più dischi può migliorare le prestazioni. Con il mirroring, la frequenza con la quale si possono gestire le richieste di lettura raddoppia, poiché ciascuna richiesta si può inviare indifferentemente a uno dei due dischi (sempre che entrambi i dischi siano funzionanti, condizione che è quasi sempre soddisfatta). La capacità di trasferimento di ciascuna lettura è la stessa di quella di un sistema a singolo disco, ma il numero di letture per unità di tempo raddoppia.

Attraverso l’uso di più dischi è possibile anche (o alternativamente) migliorare la capacità di trasferimento distribuendo i dati in sezioni su più dischi. La forma più semplice di questa distribuzione (*data striping*), consiste nel distribuire i bit di ciascun byte su più dischi; in questo caso si parla di **sezionamento** o **striping a livello dei bit**. Per esempio, se il sistema impiega un array di otto dischi, si scriverà il bit i di ciascun byte nel disco i . L’array di otto dischi si può trattare come un unico disco avente settori che hanno una dimensione otto volte superiore a quella normale e, soprattutto, che hanno una capacità di trasferimento otto volte superiore. In un’organizzazione di questo tipo, ogni disco è coinvolto in ogni accesso (lettura o scrittura che sia), così che il numero di accessi che si possono gestire nell’unità di tempo è circa lo stesso di quello per un sistema a disco singolo, ma ogni accesso permette di leggere una quantità di dati pari a otto volte quella che si può leggere con un singolo disco.

Lo striping a livello dei bit si può generalizzare a un numero di dischi multiplo di 8 o che divide 8. Per esempio, se un sistema adopera un array di quattro dischi, i bit i e $i + 4$ di ciascun byte si memorizzano nel disco i . Inoltre, lo striping non si deve realizzare necessariamente a livello dei bit di un byte: nello **striping a livello di blocco**, per esempio, i blocchi di un file si distribuiscono su più dischi; con n dischi, il blocco i di un file si memorizza nel disco $(i \bmod n) + 1$. Sono possibili anche altri livelli di striping, come quelli basati sui byte di un settore o sui settori di un blocco, ma lo striping a livello di blocco è il più comune.

Riassumendo, gli obiettivi principali del parallelismo mediante striping in un sistema di dischi sono due:

1. l’aumento, tramite il bilanciamento del carico, del throughput per accessi multipli a piccole porzioni di dati (cioè accessi a pagine);
2. la riduzione del tempo di risposta relativo ad accessi a grandi quantità di dati.

10.7.3 Livelli RAID

La tecnica di mirroring offre un’alta affidabilità ma è costosa; la tecnica di striping (sezionamento) offre un’alta capacità di trasferimento dei dati, ma non migliora l’affidabilità. Sono stati proposti numerosi schemi per fornire ridondanza a basso costo usando l’idea dello striping combinata con i bit di parità (che vedremo a breve). Que-

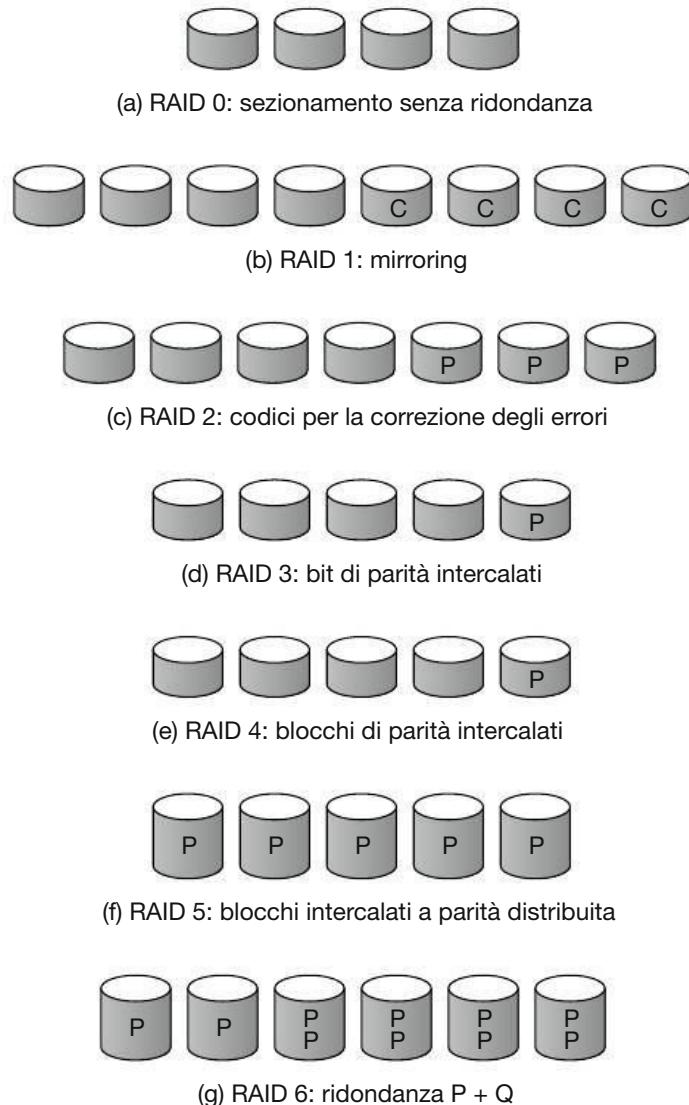


Figura 10.11 Livelli RAID.

sti schemi realizzano diversi compromessi tra costi e prestazioni e sono stati classificati in livelli chiamati **livelli RAID**, che la Figura 10.11 mostra graficamente (nella figura, la lettera *P* indica i bit di correzione degli errori, la lettera *C* indica una seconda copia dei dati). In tutti i casi riportati nella figura, sono presenti quattro dischi di dati, mentre i dischi supplementari s’impiegano per memorizzare le informazioni ridondanti per il ripristino dai guasti.

- **RAID di livello 0.** Il livello 0 si riferisce ad array di dischi con striping a livello di blocchi, ma senza ridondanza (come il mirroring o i bit di parità), come illustrato nella Figura 10.11(a).
- **RAID di livello 1.** Il livello 1 si riferisce alla tecnica di mirroring. La Figura 10.11(b) mostra un’organizzazione basata sul mirroring.

- **RAID di livello 2.** Il livello 2 è anche noto come **organizzazione con codici per la correzione degli errori di memoria**. Da molto tempo i sistemi di memorizzazione impiegano tecniche di riconoscimento degli errori basate su bit di parità. In un sistema di questo tipo, ogni byte di memoria può avere associato un bit di parità che indica se i bit con valore 1 nel byte sono in numero pari (parità = 0) oppure dispari (parità = 1). Se si altera uno dei bit nel byte (un valore 1 diventa 0 o viceversa), la parità del byte cambia e quindi non concorda più con la parità memorizzata. Analogamente, se si altera il bit di parità, esso non concorda più con la parità calcolata. In questo modo s'identificano tutti gli errori di un singolo bit nel sistema di memoria. Gli schemi di correzione degli errori memorizzano più bit supplementari e possono ricostruire i dati nel caso di un singolo bit danneggiato.

La stessa idea alla base degli ECC (*error-correcting codes*) si può usare immediatamente negli array di dischi eseguendo lo striping dei byte presenti nei dischi. Per esempio, il primo bit di ogni byte si potrebbe memorizzare nel disco 1, il secondo bit nel disco 2, e così via fino alla memorizzazione dell'ottavo bit nel disco 8 e alla memorizzazione dei bit di correzione degli errori in ulteriori dischi. Questo schema è rappresentato graficamente nella Figura 10.11(c), dove i dischi etichettati con la lettera *P* contengono i bit di correzione. Se uno dei dischi si guasta, i bit rimanenti del byte e i bit di correzione a esso associati si possono leggere dagli altri dischi e usare per ricostruire i dati danneggiati. Si noti che il RAID di livello 2 richiede tre soli dischi di overhead per quattro dischi di dati, a differenza del RAID di livello 1, che richiede quattro dischi in più.

- **RAID di livello 3.** Con il livello 3, o **organizzazione con bit di parità intercalati**, si migliora l'organizzazione del livello 2 considerando che, a differenza dei sistemi di memoria centrale, i controllori dei dischi possono rilevare se un settore è stato letto correttamente, così che un unico bit di parità si può usare sia per individuare gli errori sia per correggerli. L'idea è la seguente: se uno dei settori è danneggiato, si conosce esattamente di quale settore si tratta e, per ogni bit nel settore, è possibile determinare se debba avere valore 1 o 0 calcolando la parità dei bit corrispondenti dai settori negli altri dischi. Se la parità dei rimanenti bit è uguale a quella memorizzata, il bit mancante è 0, altrimenti è 1. Il RAID di livello 3 è altrettanto valido del livello 2 ma meno costoso in quanto richiede un solo disco supplementare, quindi il RAID di livello 2 non è utilizzato nella pratica. La Figura 10.11(d) mostra il livello 3.

Il livello 3 presenta due vantaggi rispetto al livello 1. Il primo vantaggio è che si usa un solo disco per la parità dei dati memorizzati in diversi dischi di dati, anziché un disco di mirroring per ciascun disco di dati come nel livello 1. Il secondo vantaggio è che, essendo le letture e le scritture dei byte distribuite su più dischi con uno striping a *n* vie, la velocità di trasferimento di un singolo blocco è pari a *n* volte quella dei RAID di livello 1. D'altro canto, però, il livello 3 permette meno operazioni di I/O al secondo, perché ogni disco è coinvolto da tutte le richieste.

Un altro problema di prestazioni riguardante il RAID di livello 3 (come per tutti i livelli RAID basati sui bit di parità) è il tempo richiesto dal calcolo e dalla scrittura

della parità. Questo tempo aggiuntivo determina operazioni di scrittura significativamente più lente rispetto ad array RAID senza parità. Per limitare questo calo di prestazioni, molti array RAID dispongono di un controllore capace di gestire il calcolo della parità. Questo sposta il carico dovuto al calcolo della parità dalla CPU all'array di dischi. L'array ha anche una cache NVRAM per memorizzare i blocchi mentre viene calcolata la parità e per memorizzare transitoriamente le scritture dal controllore ai dischi. Questa combinazione può rendere la tecnica RAID con parità altrettanto veloce di quella senza parità; infatti, un array RAID con cache e bit di parità può avere prestazioni migliori di un'organizzazione RAID senza cache e senza parità.

- **RAID di livello 4.** Nel livello 4, o **organizzazione con blocchi di parità intercalati**, s'impiega lo striping a livello di blocchi, come nel RAID di livello 0 e inoltre si tiene un blocco di parità in un disco separato per i blocchi corrispondenti presenti negli altri n dischi. Tale schema è illustrato nella Figura 10.11(e). Se uno dei dischi si guasta, il blocco di parità si può usare insieme ai blocchi corrispondenti degli altri dischi per ripristinare i blocchi nel disco guasto.

La lettura di un blocco richiede l'accesso a un solo disco, permettendo la gestione di altre richieste da parte di altri dischi. Quindi, la capacità di trasferimento dei dati per ciascun accesso è minore, ma gli accessi in lettura possono procedere in modo parallelo ottenendo una velocità complessiva di I/O più alta. La capacità di trasferimento per la lettura di molti dati è alta, poiché si possono leggere in modo parallelo tutti i dischi e anche le operazioni di scrittura di grandi quantità di dati presentano un'alta capacità di trasferimento, poiché i dati e i bit di parità si possono scrivere in parallelo.

Scritture indipendenti di modesta entità non si possono eseguire in parallelo. La scrittura da parte del sistema operativo di una quantità di dati inferiore a un blocco richiede la lettura del blocco, la sua modifica, e la scrittura del blocco modificato; anche il blocco di parità deve essere aggiornato. Si parla a questo proposito del **ciclo lettura-modifica-scrittura**. Una singola richiesta di scrittura comporta pertanto quattro accessi al disco, due in lettura e due in scrittura.

Nel Capitolo 12 sarà presentato WAFL: esso adotta RAID di livello 4, che permette l'aggiunta di dischi al sistema senza soluzione di continuità. Inizializzando i nuovi dischi a zero, la parità non cambia, e l'array RAID è ancora nello stato corretto.

- **RAID di livello 5.** Il livello 5, o **organizzazione con blocchi intercalati a parità distribuita**, differisce dal livello 4 per il fatto che, invece di memorizzare i dati in n dischi e la parità in un disco separato, i dati e le informazioni di parità sono distribuite tra gli $n + 1$ dischi. Per ogni blocco, uno dei dischi memorizza la parità e gli altri i dati. Per esempio, considerando una batteria di cinque dischi, la parità per il blocco m -esimo si memorizza nel disco $(m \bmod 5) + 1$, mentre i blocchi m -esimi degli altri quattro dischi contengono i dati effettivi per quel blocco. Questo schema è illustrato nella Figura 10.11(f), dove i simboli P sono distribuiti su tutti i dischi. Un blocco di parità non può contenere informazioni di parità per blocchi

che risiedono nello stesso disco, poiché un guasto al disco provocherebbe sia la perdita di dati sia la perdita dell'informazione di parità e quindi i dati non sarebbero ripristinabili. Con la distribuzione della parità sui diversi dischi, il RAID di livello 5 evita un uso intensivo del disco dove risiede la parità, che invece si ha con il RAID di livello 4. RAID 5 è il più comune sistema di parità RAID.

- **RAID di livello 6.** Il livello 6, detto anche **schema di ridondanza P + Q**, è molto simile al RAID di livello 5, ma memorizza ulteriori informazioni ridondanti per poter gestire guasti contemporanei di più dischi. Invece di usare la parità, s'impiegano codici per la correzione degli errori come i **codici di Reed-Solomon**. Nello schema mostrato nella Figura 10.11(g), sono memorizzati 2 bit di dati ridondanti ogni 4 bit di dati effettivi (a differenza di 1 bit di parità usato nel livello 5) e il sistema risultante può tollerare due guasti dei dischi.
- **RAID di livello 0 + 1 e 1 + 0.** Il livello 0 + 1 consiste in una combinazione dei livelli RAID 0 e 1. Il livello 0 fornisce le prestazioni, mentre il livello 1 l'affidabilità. Di solito, questo schema porta a prestazioni migliori rispetto a livello 5 e si usa prevalentemente negli ambienti in cui sono importanti sia le prestazioni sia l'affidabilità. Sfortunatamente, questo schema richiede, come il RAID di livello 1, un raddoppio del numero di dischi necessario per memorizzare i dati, quindi è anche relativamente costoso. Nel RAID di livello 0 + 1, si effettua lo striping su un insieme di dischi e si duplica ogni sezione con la tecnica del mirroring.

Un altro metodo che sta diventando disponibile commercialmente è il RAID di livello 1 + 0, in cui si fa prima il mirroring dei dischi a coppie, e poi lo striping di queste coppie. Questo schema RAID ha alcuni vantaggi teorici rispetto al RAID 0 + 1. Per esempio, se si guasta un singolo disco nel RAID 0 + 1, l'intera sezione di dati diventa inaccessibile, lasciando disponibile solo l'altra sezione. Con un guasto nel RAID 1 + 0, il singolo disco diventa inaccessibile, ma il suo duplicato è ancora disponibile, come tutti gli altri dischi (Figura 10.12).

Sono state proposte numerose altre varianti agli schemi RAID di base illustrati sopra e questo ha portato anche una certa confusione nelle precise definizioni dei diversi livelli RAID.

Un altro aspetto soggetto a molte varianti è l'implementazione del RAID. Esamiamo i diversi strati architetturali a cui è possibile implementare un sistema RAID.

- Il software per la gestione dei volumi può implementare un sistema RAID all'interno del kernel o a livello dei programmi di sistema. In questo caso, nonostante i dispositivi per la memorizzazione possano fornire funzionalità minime, è possibile ottenere un sistema RAID completo. Il metodo RAID con parità è alquanto lento se realizzato tramite software, quindi gli si preferisce solitamente RAID 0,1 oppure 0 + 1.
- Il RAID può essere implementato a livello hardware dall'adattatore del bus della macchina (*host bus adapter*, HBA). Solo i dischi connessi direttamente all'HBA possono costituire parte integrante di una dato array RAID. Questa soluzione è a basso costo, ma non è molto flessibile.

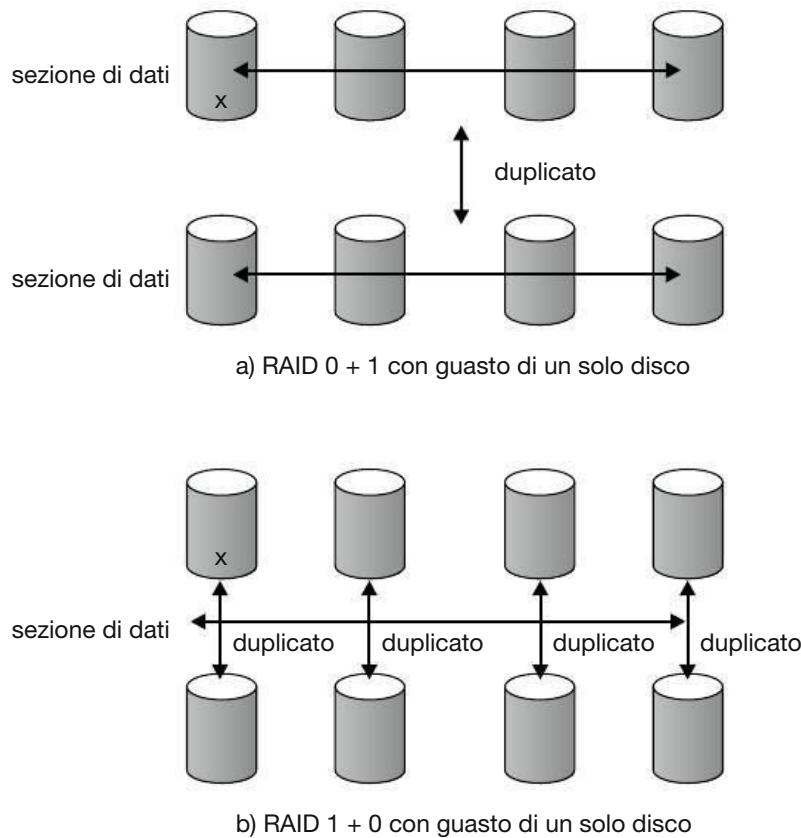


Figura 10.12 RAID 0 + 1 e 1 + 0.

- Il RAID può essere implementato a livello hardware dall’array di dischi. È così possibile creare sistemi RAID a vari livelli, e persino ricavare da essi volumi più piccoli, che sono quindi presentati al sistema operativo, che avrà solo da realizzare il file system su ciascuno dei volumi. Gli array possono disporre di connessioni multiple o far parte di una rete di memorizzazione secondaria (SAN), consentendo a varie macchine di sfruttare le funzionalità dell’array.
- Il RAID può essere implementato da dispositivi di virtualizzazione del disco a livello di interconnessione SAN. In questo caso, un dispositivo funge da intermediario tra le macchine e l’area di memorizzazione, accettando istruzioni dai server e gestendo l’accesso alla memoria secondaria. Esso potrebbe, per esempio, attuare il mirroring, trascrivendo ciascun blocco su due distinti dispositivi di memorizzazione.

Ulteriori funzionalità, come quella di istantanea e di replica, possono essere implementate a ognuno di questi livelli. Un’**istantanea (snapshot)** è un’immagine del file system così com’era prima dell’ultimo aggiornamento (le istantanee saranno trattate più in dettaglio nel Capitolo 12). La **replica** prevede la duplicazione automatica di scritture su siti diversi, per finalità di ridondanza, o di ripristino in caso di eventi disastrosi (*disaster recovery*). La replica può essere sincrona o asincrona. Se è sincrona,

ciascun blocco deve essere scritto sia localmente, sia in remoto, prima che la scrittura sia considerata completa; se è asincrona, si effettuano periodicamente scritture a gruppi. La replica asincrona espone al rischio di perdere i dati, se il sito principale fallisce, ma è più veloce e non ha limiti di distanza.

L'implementazione di queste funzionalità varia a seconda dello strato scelto per realizzare il sistema RAID. Qualora RAID, per esempio, sia implementato a livello software, ciascuna macchina può aver necessità di implementare e gestire la replica per proprio conto. Tuttavia, se la replica avviene a livello dell'array di dischi o dell'interconnessione SAN, si possono replicare i dati della macchina a prescindere dal suo sistema operativo e dalle relative funzionalità.

Un'altra caratteristica spesso presente nei sistemi RAID è la previsione di **dischi di scorta** (*hot spare*), che possono sostituire quelli normali in caso di guasti. Per esempio, un disco di scorta può essere usato per sostituire un disco danneggiato, ricostruendo l'integrità di una coppia in mirroring. In questo modo, si può ristabilire automaticamente lo stato corretto del livello RAID, senza attendere che il disco difettoso sia sostituito. È possibile riparare più di un guasto, senza l'intervento di un operatore, con l'allocazione di più dischi di scorta.

10.7.4 Scelta di un livello RAID

Viste le svariate possibilità esistenti, ci si potrebbe chiedere quali siano i criteri di scelta dei progettisti nei confronti del livello RAID. Una considerazione è il tempo di ricostruzione. Se un disco si guasta, il tempo necessario a ricostruire i dati che contiene può essere rilevante. Questo fattore può essere importante nel caso in cui venga richiesto un flusso continuo di dati, come nei database ad alte prestazioni o interattivi. Inoltre il tempo di ricostruzione influenza il tempo medio di guasto.

Il tempo di ricostruzione varia a seconda del livello RAID utilizzato. La ricostruzione più semplice si ha per RAID di livello 1, poiché i dati possono essere copiati da un altro disco; per gli altri livelli, per ricostruire i dati in un disco guasto è necessario accedere a tutti gli altri dischi dell'array. Il tempo necessario per la ricostruzione dei dati può essere di ore nel caso di sistemi RAID di livello 5 con molti dischi.

RAID di livello 0 si usa nelle applicazioni ad alte prestazioni in cui le perdite di dati non sono critiche. Il RAID di livello 1 si usa comunemente nelle applicazioni che richiedono un'alta affidabilità e un rapido ripristino. I livelli RAID 0 + 1 e 1 + 0 si usano dove sia le prestazioni sia l'affidabilità sono importanti, per esempio per piccole basi di dati. A causa dell'elevata richiesta di spazio del RAID di livello 1, per la memorizzazione di grandi quantità di dati, spesso si preferisce impiegare il RAID di livello 5. Il livello 6, attualmente non disponibile in molte implementazioni RAID, dovrebbe offrire una migliore affidabilità rispetto a livello 5.

Progettisti e amministratori di sistemi RAID devono prendere anche altre decisioni importanti, per esempio riguardo al numero ottimale di dischi in un array e al numero di bit che ciascun bit di parità deve proteggere. Maggiore è il numero di dischi in un array, maggiore sarà la capacità di trasferimento dei dati, ma il sistema sarà anche più costoso. Maggiore è il numero di bit protetti da un singolo bit di parità, minore

L'ARRAY InServ

L'innovazione, grazie al quale sono di continuo introdotte soluzioni migliori, più veloci e meno costose, ridefinisce spesso i confini che separano le tecnologie già esistenti. Si consideri, per esempio, l'array InServ di 3Par. A differenza di molti altri, esso non richiede la configurazione di un insieme di dischi a un livello RAID specifico, ma scomponete invece ogni disco in porzioni da 256 MB. Il metodo RAID è pertanto applicato a livello di queste porzioni. Di conseguenza, vari livelli RAID possono interessare lo stesso disco, e le sue porzioni vengono utilizzate per formare diversi volumi.

InServ mette inoltre a disposizione le istantanee, una funzionalità simile a quella del file system WAFL. Le istantanee di InServ prevedono sia il formato lettura-scrittura sia il formato a sola lettura, consentendo a utenti multipli di montare copie di un dato file system senza doverne possedere una copia completa. Le modifiche eventualmente apportate dagli utenti alla propria copia sono di copiatura su scrittura, ragion per cui non hanno effetto sulle altre copie.

Un'altra innovazione è il cosiddetto **utility storage**. Alcuni file system non possono essere espansi, né compressi. I file system di questo genere mantengono costantemente le dimensioni originali: per qualsiasi modifica è necessaria la copiatura di dati. Un amministratore può configurare InServ per fornire a una macchina cospicue quantità di memoria logica, che all'inizio occupano solamente un piccolo spazio di memoria fisica. Mentre la macchina comincia a usare lo spazio di memorizzazione, dischi non ancora utilizzati sono assegnati alla macchina, fino a raggiungere il livello logico originale. In tal modo, la macchina è indotta a credere di possedere un vasto spazio di memorizzazione permanente, dove creare i propri file system, e così via. InServ può aggiungere o rimuovere dischi dal file system senza che il file system se ne accorga. Questa caratteristica può ridurre il numero complessivo di unità a disco necessarie alle macchine, o perlomeno ritardare l'acquisizione di nuovi dischi finché non divengano realmente necessari.

sarà lo spazio richiesto dai bit di parità; sarà però maggiore anche la probabilità che un secondo disco si guasti prima che un disco guasto sia riparato e questo porterebbe alla perdita di dati.

10.7.5 Estensioni

I concetti relativi ai sistemi RAID sono stati generalizzati ad altri dispositivi di memorizzazione, comprese le unità a nastri e anche alla diffusione dei dati tramite sistemi senza fili (*wireless*). Con strutture RAID applicate alle unità a nastri si possono ripristinare i dati anche se uno dei nastri della batteria è danneggiato. Se applicate alla trasmissione dei dati, si divide ogni blocco di dati in unità più piccole che si trasmettono insieme a un'unità di parità; se per qualsiasi ragione una delle unità non viene ricevuta, può essere ricostruita dalle altre. Di solito, con l'uso di unità automatiche dotate di molte unità a nastro si esegue lo striping dei dati su tutte le unità per aumentare il throughput e diminuire il tempo di backup.

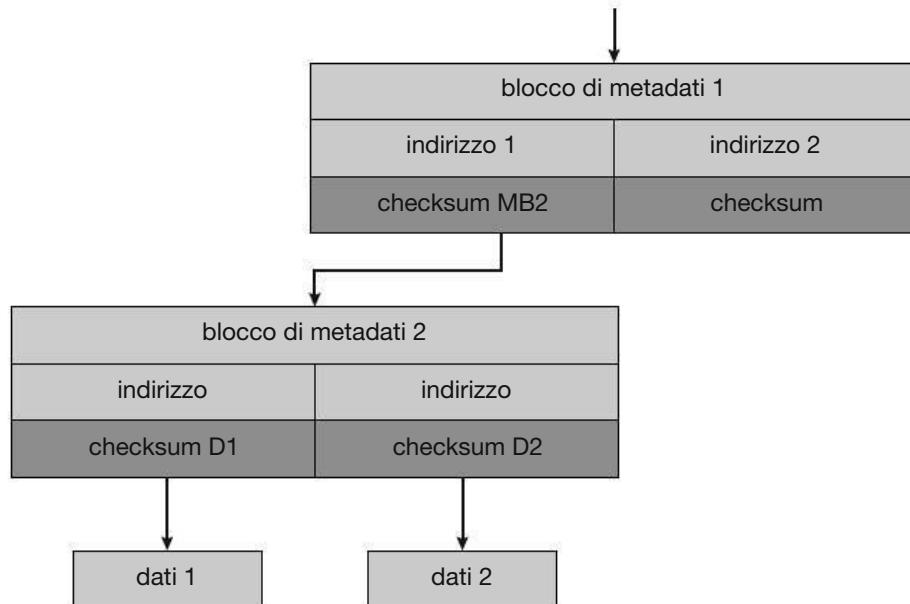


Figura 10.13 ZFS applica una checksum a tutti i metadati e i dati.

10.7.6 Problemi connessi a RAID

I sistemi RAID, purtroppo, non assicurano sempre la disponibilità dei dati al sistema operativo e agli utenti. Un puntatore a un file potrebbe essere errato, per esempio, e lo stesso potrebbe accadere ai puntatori nella struttura interna dei file. Le operazioni incomplete di scrittura, se non ripristinate in maniera adeguata, possono alterare i dati. Altri processi, inoltre, potrebbero scrivere accidentalmente sulle strutture del file system. RAID protegge dagli errori derivanti dai supporti fisici per la memorizzazione, ma non da altri tipi di errori dovuti all'hardware e ai programmi. I pericoli potenziali, per i dati di un sistema, si estendono alla totalità degli errori derivanti dal software e dall'hardware.

Per risolvere tali problemi, il file system **Solaris ZFS** ricorre a una strategia innovativa per verificare l'integrità dei dati. Esso applica una **checksum** (*somma di controllo*) interna a ogni blocco, dati e metadati inclusi. Le checksum non risiedono nel blocco sottoposto a controllo: ciascuna di esse, invece, è memorizzata insieme al puntatore a quel blocco (Figura 10.13). Si consideri un **inode** – una struttura dati per memorizzare i metadati del file system - con puntatori ai propri dati. All'interno dell'inode si trova la checksum per ciascun blocco di dati. Se si verifica un problema con i dati, la checksum darà un valore errato e il file system verrà a conoscenza del problema. Qualora sia attivo il mirroring, il sistema ZFS, in presenza di un blocco con una checksum corretta e di uno con una checksum errata, sostituirà automaticamente il blocco errato con quello valido. In maniera simile, l'elemento della directory che punta all'inode possiede una checksum relativa all'inode. Qualunque problema ri-

guardi l'inode, quindi, è rilevato al momento dell'accesso alla directory. Queste checksum, che sono applicate a tutte le strutture di ZFS, producono risultati molto più efficaci degli ambienti RAID o dei file system tradizionali, per livello di coerenza, rilevazione degli errori e capacità di correggerli. Il sovraccarico di gestione determinato dal calcolo delle checksum e dai cicli supplementari di lettura-modifica-scrittura dei blocchi non condizionano il funzionamento complessivo di ZFS, che mantiene un'alta velocità nelle prestazioni.

Un altro problema della maggior parte delle implementazioni RAID è la mancanza di flessibilità. Considerate un array di memorizzazione dotato di venti dischi divisi in quattro insiemi da cinque dischi. Ogni gruppo di cinque dischi è un insieme RAID di livello 5. Ne risultano quattro volumi separati, ciascuno contenente un proprio file system. Ma che cosa succede se un file system ha una dimensione troppo grande per un insieme RAID di livello 5 a cinque dischi? E se un altro file system necessita di un'area molto ridotta? Se tali fattori sono noti in anticipo, allora i dischi e i volumi possono essere allocati adeguatamente. Frequentemente, però, l'utilizzo del disco e le richieste variano nel tempo.

Anche se l'array di memorizzazione ha permesso all'intero insieme di venti dischi di essere creato come un grande insieme RAID, potrebbero insorgere altre problematiche. Nell'insieme potrebbero essere costruiti diversi volumi di dimensioni differenti. Alcuni gestori del volume non ci permettono però di cambiare la dimensione di un volume. In quel caso si ripresenterebbe la stessa situazione descritta in precedenza, ovvero dimensioni non adatte al file system. Alcuni gestori di volume permettono cambiamenti di dimensione, ma alcuni file system non permettono l'aumento o la diminuzione della dimensione del file system stesso. I volumi potrebbero cambiare dimensione, ma i file system dovrebbero essere ricreati per poter usufruire di quei cambiamenti.

ZFS combina la gestione dei file system e quella dei volumi in una unità in grado di offrire una maggiore funzionalità rispetto a quella permessa dalla tradizionale separazione di tali funzioni. I dischi, o le partizioni di dischi, sono riuniti in **gruppi di memorizzazione** (*pools of storage*) attraverso insiemi RAID. Un gruppo o **pool** può contenere uno o più file system ZFS. Tutta l'area libera di un pool è a disposizione di tutti i file system contenuti all'interno di quel pool. ZFS utilizza il modello di gestione della memoria basato su `malloc()` e `free()` per allocare e rilasciare memoria nei file system quando i blocchi vengono utilizzati e liberati all'interno del file system. Di conseguenza non ci sono limiti artificiali all'utilizzo della memoria e non sussiste la necessità di ridistribuire i file system tra i volumi né di ridimensionare i volumi. ZFS stabilisce delle quote per limitare la dimensione di un file system e definisce dei meccanismi di prenotazione per assicurarsi che il file system possa aumentare all'interno di una dimensione specificata, ma queste variabili possono essere sempre cambiate dal proprietario del file system. La Figura 10.14(a) illustra i volumi e i file system tradizionali, mentre la Figura 10.14(b) rappresenta il modello ZFS.

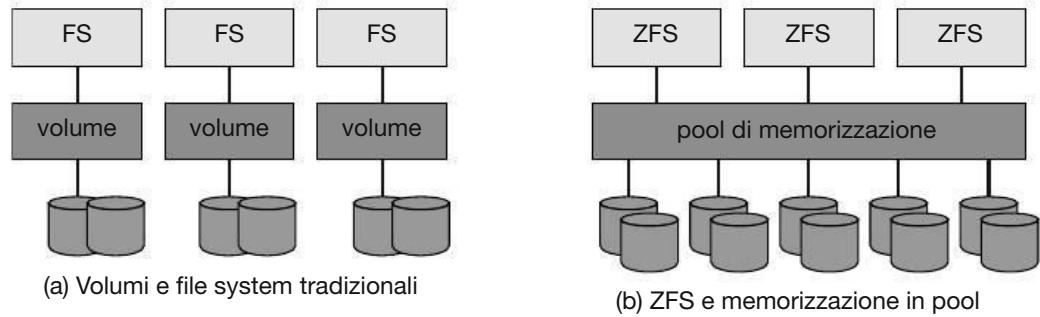


Figura 10.14 (a) Volumi e file system tradizionali. (b) Pool e file system ZFS.

10.8 Realizzazione della memoria stabile

In molte applicazioni relative alle basi di dati è necessario disporre di una **memoria stabile**. Per definizione, le informazioni residenti in questo tipo di memoria non vanno *mai* perse. Per realizzare un tale tipo di memoria si devono replicare le informazioni necessarie in più dispositivi di memorizzazione (di solito dischi) con meccanismi di guasto indipendenti. Inoltre è necessario coordinare l’aggiornamento delle informazioni in modo tale che un eventuale malfunzionamento durante l’aggiornamento non lasci tutte le copie in uno stato danneggiato, e che ripristinando le informazioni a seguito di un guasto sia possibile riportare ogni copia alla sua forma coerente e corretta anche se si verifica un altro malfunzionamento proprio durante il ripristino. Nel resto del paragrafo si presentano le possibili soluzioni a queste esigenze.

Un’operazione di scrittura in un disco può avere uno dei seguenti esiti.

1. **Operazione riuscita.** I dati sono stati scritti correttamente nel disco.
2. **Insuccesso parziale.** Si è verificato un guasto durante il trasferimento, e solo alcuni tra i settori coinvolti sono stati correttamente aggiornati, mentre il settore interessato dalla scrittura al momento del malfunzionamento può essere stato danneggiato.
3. **Insuccesso totale.** Il malfunzionamento è avvenuto prima dell’avvio del processo di scrittura nel disco; i dati già residenti nel disco sono rimasti inalterati.

Se si verifica un guasto durante la scrittura di un blocco, il sistema deve rilevarlo e invocare una procedura di ripristino per riportare il blocco in uno stato coerente. A tale scopo il sistema deve mantenere due blocchi fisici per ciascun blocco logico, eseguendo ogni scrittura nel modo seguente:

1. scrittura delle informazioni nel primo blocco fisico;
2. completata con successo la prima scrittura, scrittura delle stesse informazioni nel secondo blocco fisico;
3. l’operazione è considerata completa solamente dopo che la seconda scrittura è stata eseguita correttamente.

Durante il ripristino da un malfunzionamento si esamina ogni coppia di blocchi fisici; se essi contengono gli stessi dati e non c'è traccia d'errori, non è necessario intraprendere alcuna azione ulteriore. Se un blocco contiene un errore riscontrabile, se ne sostituisce il contenuto con quello del secondo blocco. Se in nessuno dei due blocchi si riscontra un errore, ma ciascuno contiene informazioni differenti, si sostituisce il contenuto del primo blocco con quello del secondo. Questa procedura di ripristino assicura che gli unici due esiti possibili di un'operazione di scrittura nella memoria stabile siano o la completa riuscita dell'operazione oppure il suo insuccesso totale, in quest'ultimo caso i dati memorizzati non subiscono alcun cambiamento.

Questa procedura si può facilmente estendere all'uso di un numero arbitrariamente grande di copie per ciascun blocco di memoria stabile. Benché un gran numero di queste copie riduca la probabilità di malfunzionamenti, di solito è ragionevole simulare la memoria stabile con due sole copie. I dati di una memoria stabile non andranno mai persi purché un guasto non distrugga tutte le copie.

Poiché le attese per il completamento delle scritture (I/O sincrono) richiedono molto tempo, molti array di memorizzazione aggiungono NVRAM come cache. Poiché la memoria è non volatile (di solito è dotata di una pila per l'alimentazione elettrica di riserva), si può ritenere affidabile per la memorizzazione dei dati in transito verso i dischi, e si può considerare parte della memoria stabile. Le scritture in NVRAM sono molto più rapide di quelle nei dischi, quindi le prestazioni migliorano notevolmente.

10.9 Sommario

Per la maggior parte dei calcolatori le unità a dischi sono il principale dispositivo di I/O di memoria secondaria. La gran parte dei dispositivi per la memorizzazione secondaria usa i dischi o i nastri magnetici, anche se i dischi a stato solido stanno acquisendo un'importanza sempre maggiore. Le moderne unità a disco sono strutturate come un grande array monodimensionale di blocchi logici, solitamente di 512 byte. I dischi possono essere collegati al computer in due modi: (1) tramite le porte per l'I/O locali della macchina o (2) tramite connessioni di rete.

Le richieste di I/O sui dischi sono generate sia dal file system sia dai sistemi di memoria virtuale, e ognuna di esse specifica l'indirizzo cui fare riferimento nel disco sotto forma di numero di un blocco logico. Gli algoritmi di scheduling per i dischi possono aumentare l'ampiezza di banda, e ridurre il tempo di risposta medio e la variabilità del tempo di risposta. Algoritmi come l'SSTF, SCAN, C-SCAN, LOOK e C-LOOK sono progettati per realizzare questi miglioramenti tramite criteri di ordinamento della coda di richieste di accesso ai dischi. Le prestazioni degli algoritmi di scheduling sui dischi magnetici possono variare notevolmente. Al contrario, nel caso dei dischi a stato solido, che non hanno parti in movimento, non ci sono grandi differenze di prestazioni tra i vari algoritmi e molto spesso viene utilizzata una semplice strategia FCFS.

Il sistema operativo gestisce i blocchi di un disco: innanzitutto si deve formattare fisicamente il disco per creare i settori direttamente sui suoi piatti – i dischi nuovi so-

no generalmente venduti già formattati; in seguito, il disco può essere diviso in partizioni, si può creare il file system, e si possono assegnare i blocchi d'avviamento che conterranno il programma d'avviamento del sistema; infine, quando un blocco diviene difettoso, il sistema deve essere in grado di isolarlo o di sostituirlo logicamente con un blocco di riserva.

Poiché un'area d'avvicendamento efficiente è essenziale per un sistema dalle buone prestazioni, molti sistemi bypassano il file system e usano un accesso di basso livello per l'I/O di paginazione. Certi sistemi riservano all'area d'avvicendamento una partizione di basso livello, altri impiegano un ordinario file all'interno del file system. Altri sistemi lasciano la scelta fra queste due possibilità all'utente o all'amministratore di sistema.

A causa della quantità di spazio di memorizzazione secondaria richiesta nei grandi sistemi, i dischi sono spesso resi ridondanti tramite algoritmi RAID. Questi algoritmi permettono l'uso di più dischi per una data operazione, e consentono la prosecuzione del funzionamento del sistema e anche il ripristino automatico dei dati a fronte del guasto di un disco. Gli algoritmi RAID sono classificati in livelli che offrono diverse combinazioni di affidabilità ed elevate velocità di trasferimento.

Esercizi di ripasso

- 10.1** Lo scheduling del disco, con algoritmi diversi dall'FCFS, è utile in un ambiente con un solo utente? Argomentate.
- 10.2** Spiegate perché l'SSTF tende a favorire i cilindri centrali rispetto a quelli più interni e più esterni.
- 10.3** Perché la latenza di rotazione non viene solitamente presa in considerazione nello scheduling del disco? Come potreste modificare lo scheduling SSTF, lo SCAN e quello C-SCAN per includervi l'ottimizzazione della latenza?
- 10.4** Perché è importante bilanciare gli I/O del file system tra i dischi e i controllori su un sistema in un ambiente multitasking?
- 10.5** Quali sono i tradeoff fra la rilettura delle pagine di codice da un file system e l'utilizzo dell'area di avvicendamento per memorizzarli?
- 10.6** Esiste un modo per realizzare una memorizzazione delle informazioni veramente stabile? Argomentate la risposta.
- 10.7** A volte un nastro è detto mezzo ad accesso sequenziale, mentre un disco magnetico è considerato un mezzo ad accesso diretto. In realtà, l'idoneità di un dispositivo di memorizzazione all'accesso diretto dipende dalla grandezza del trasferimento. Il termine *velocità di trasferimento in streaming* denota la velocità di un trasferimento di dati che è in corso, escluso l'effetto della latenza di accesso. La *velocità effettiva di trasferimento*, invece, è il rapporto dei byte totali per il totale dei secondi, inclusi i tempi di overhead come la latenza di accesso.

Ipotizzate che, in un computer, la cache di livello 2 abbia una latenza di accesso di 8 nanosecondi e una velocità di trasferimento in streaming di 800 megabyte al secondo, che la memoria principale abbia una latenza di accesso di 60 nanosecondi e una velocità di trasferimento in streaming di 80 megabyte al secondo, che il disco magnetico abbia una latenza di accesso di 15 millisecondi e una velocità di trasferimento in streaming di 5 megabyte al secondo, e che una unità a nastro abbia una latenza di accesso di 60 secondi e una velocità di trasferimento in streaming di 2 megabyte al secondo.

- a. L'accesso diretto causa la diminuzione della velocità effettiva di trasferimento di un dispositivo, perché non vengono trasferiti dati durante il tempo di accesso. Per il disco descritto, qual è la velocità di trasferimento effettiva se in media un accesso è seguito da un trasferimento in streaming di (1) 512 byte, (2) 8 kilobyte, (3) 1 megabyte, e (4) 16 megabyte?
- b. L'utilizzazione di un dispositivo è definita come il rapporto tra la velocità effettiva di trasferimento e la velocità di trasferimento in streaming. Calcolate l'utilizzazione dell'unità a disco per ognuna delle quattro grandezze di trasferimento indicate al punto a.
- c. Supponete che un'utilizzazione del 25 per cento o più sia considerata accettabile. Utilizzando i valori di prestazione indicati, calcolate la dimensione minima di trasferimento per un disco che dia un'utilizzazione accettabile.
- d. Completate la frase seguente: Un disco è un dispositivo ad accesso diretto per trasferimenti superiori a _____ byte ed è un dispositivo ad accesso sequenziale per trasferimenti inferiori.
- e. Calcolate le dimensioni minime di trasferimento che diano utilizzazioni accettabili per cache, memoria e nastro.
- f. Quando un nastro può essere considerato un dispositivo ad accesso diretto e quando invece è un dispositivo ad accesso sequenziale?

10.8 Può un'organizzazione RAID a livello 1 ottenere prestazioni migliori sulle richieste di lettura rispetto a un'organizzazione RAID a livello 0 (con striping senza ridondanza)? Se sì, come?

Esercizi

10.9 Eccetto l'FCFS, nessuno tra i criteri di scheduling del disco descritti è veramente *equo* (si può avere un'attesa indefinita).

- a. Spiegate perché questa affermazione è vera.
- b. Descrivete una maniera di modificare gli algoritmi come lo SCAN in modo che risultino equi.

- c. Spiegate perché l'equità è un obiettivo importante in un sistema a partizione del tempo.
- d. Date tre o più esempi di circostanze in cui è importante che il sistema operativo non sia equo nel servire le richieste di I/O.

10.10 Spiegate perché per i dischi SSD viene spesso utilizzato un algoritmo di scheduling FCFS.

10.11 Supponete che un'unità a disco abbia 5000 cilindri numerati da 0 a 4999. L'unità serve attualmente una richiesta relativa al cilindro 2150, e la richiesta precedente era relativa al cilindro 1850. La coda di richieste inevase, in ordine FIFO, è composta di richieste riguardanti i cilindri

2069, 1212, 2296, 2800, 544, 1618, 356, 1523, 4965, 3681

Partendo dalla posizione attuale della testina, calcolate la distanza totale (in cilindri) che il braccio del disco percorre per soddisfare tutte le richieste inevase usando i seguenti algoritmi di scheduling:

- a. FCFS;
- b. SSTF;
- c. SCAN;
- d. LOOK;
- e. C-SCAN;
- f. C-LOOK.

10.12 È noto dalla fisica elementare che quando un oggetto è sottoposto a un'accelerazione costante a , la relazione fra la distanza d e il tempo t è data da $d = 1/2 at^2$. Supponete che l'unità a disco dell'Esercizio 10.11, durante la ricerca di un cilindro, imprima al braccio del disco un'accelerazione costante per la prima metà del tragitto richiesto dalla ricerca, e imprima invece una decelerazione costante della stessa intensità per la seconda metà del tragitto. Ipotizzate che l'unità possa portare a termine in 1 millisecondo la ricerca di un cilindro adiacente, e in 18 millisecondi una ricerca a tutto raggio lungo i 5000 cilindri.

- a. La distanza di una ricerca è il numero di cilindri attraverso i quali la testina deve passare. Spiegate perché il tempo di ricerca è proporzionale alla radice quadrata della distanza percorsa.
- b. Scrivete il tempo di ricerca in funzione della distanza da percorrere. L'equazione dovrebbe avere la forma $t = x + y \sqrt{L}$, dove t è il tempo in millisecondi e L è la distanza da percorrere in cilindri.
- c. Calcolate il tempo totale di ricerca per ogni algoritmo di scheduling dell'Esercizio 10.11 relativamente alla coda di richieste lì descritta. Determinate quale algoritmo sia più veloce, cioè implica un tempo di ricerca totale minore.

d. L'*aumento percentuale di velocità* è il tempo risparmiato diviso il tempo originariamente necessario. Calcolate l'aumento percentuale di velocità dell'algoritmo più veloce rispetto all'FCFS.

10.13 Supponete che il disco dell'Esercizio 10.12 ruoti alla velocità di 7200 giri al minuto.

- a. Calcolate la latenza di rotazione media di quest'unità a disco.
- b. Dite quale distanza di ricerca è possibile coprire nel tempo calcolato al punto a).

10.14 Descrivete vantaggi e svantaggi dell'utilizzo di unità SSD come livello aggiuntivo di cache e come unico dispositivo rispetto all'utilizzo esclusivo di un disco magnetico.

10.15 Confrontate le prestazioni di SCAN e C-SCAN assumendo una distribuzione uniforme delle richieste di I/O. Considerate il tempo di risposta medio (cioè il tempo che intercorre fra l'arrivo di una richiesta e il completamento dell'operazione a essa associata), la variazione del tempo di risposta, e l'effettiva ampiezza di banda. Analizzate come le prestazioni dipendano dalle dimensioni relative del tempo di ricerca e della latenza di rotazione.

10.16 Le richieste non sono di solito uniformemente distribuite: per esempio un cilindro che contiene metadati del file system potrebbe essere visitato più spesso di un cilindro che contiene solo file. Supponete di sapere che il 50 per cento delle richieste sia relativo a un piccolo numero fisso di cilindri.

- a. Dite se uno fra gli algoritmi illustrati in questo capitolo sia particolarmente adatto a questa circostanza. Motivate la risposta.
- b. Proponete un algoritmo di scheduling che offra prestazioni anche migliori sfruttando questo "punto caldo" del disco.

10.17 Considerate un sistema RAID a livello 5, comprensivo di cinque dischi; nel quinto disco risiedono le parità per gli insiemi di quattro blocchi di quattro dischi. A quanti blocchi si deve accedere per effettuare le operazioni seguenti:

- a. scrittura di un blocco di dati;
- b. scrittura di sette blocchi contigui di dati.

10.18 Ponete a confronto il throughput ottenuto attraverso un sistema RAID a livello 5 con quello ottenuto mediante un sistema RAID a livello 1, per quel che riguarda:

- a. operazioni di lettura su blocchi singoli;
- b. operazioni di lettura su blocchi multipli contigui.

10.19 Paragonate le prestazioni raggiunte da un sistema RAID a livello 5 con quelle di un sistema RAID a livello 1, relativamente alle operazioni di scrittura.

10.20 Assumete di avere una configurazione mista, che comprenda dischi strutturati secondo il sistema RAID a livello 1, e altri dischi a livello RAID 5. Supponete che il sistema, per memorizzare un determinato file, sia libero di optare per l'una o l'altra soluzione. Quali file dovrebbero essere memorizzati nei dischi a livello RAID 1 e quali nei dischi a livello RAID 5, allo scopo di ottimizzare le prestazioni?

10.21 L'affidabilità di un'unità a disco generalmente si quantifica usando il **tempo medio fra due guasti** (*mean time between failures*, MTBF); sebbene sia chiamata “tempo”, questa quantità in effetti si misura in ore di funzionamento dell’unità per guasto.

- a. Dato un gruppo di 1000 unità a disco, ognuna delle quali ha un MTBF di 750.000 ore, dite con quale frequenza avverrà il guasto di un’unità del gruppo, scegliendo fra le seguenti possibilità quella che si adatta meglio alla situazione descritta: una volta ogni mille anni, una volta ogni cento anni, una volta ogni dieci anni, una volta al mese, una volta alla settimana, una volta al giorno, una volta ogni ora, una volta al minuto, una volta al secondo.
- b. Le statistiche di mortalità indicano che in media un individuo residente negli Stati Uniti d’America ha circa 1 probabilità su 1000 di morire fra i 20 e i 21 anni d’età. Deducete l’MTBF di un ventenne, e convertite il risultato da ore in anni. Spiegate che cosa dice questo MTBF sulle aspettative di vita di un ventenne.
- c. Un produttore asserisce che un certo modello di unità a disco abbia un MTBF di 1 milione di ore. Dite cosa si può concludere circa il numero di anni per cui una di queste unità a disco è coperta dalla garanzia.

10.22 Esponete vantaggi e svantaggi relativi all'accantonamento dei settori e alla traslazione dei settori.

10.23 Esponete i motivi per cui il sistema operativo potrebbe necessitare di informazioni accurate sulle modalità di memorizzazione dei blocchi sul disco. In termini di miglioramento delle prestazioni del file system, in che modo il sistema operativo può mettere a frutto queste informazioni?

Problemi di programmazione

10.24 Scrivete un programma che implementa i seguenti algoritmi di scheduling del disco:

- a. FCFS
- b. SSTF
- c. SCAN
- d. C-SCAN
- e. LOOK
- f. C-LOOK

Il vostro programma servirà un disco con 5000 cilindri numerati da 0 a 4.999. Il programma genererà una sequenza casuale di 1000 richieste e le servirà secondo ciascuno degli algoritmi sopra elencati. Al programma sarà passata, come parametro da riga di comando, la posizione iniziale della testina del disco. Il programma restituirà la quantità totale di movimenti della testina richiesti da ciascun algoritmo.

Note bibliografiche

[Services 2012] fornisce una panoramica dei sistemi di memorizzazione dei dati in una varietà di ambienti elaborativi moderni. [Teorey e Pinkerton 1972] presentano un'analisi comparativa di algoritmi di scheduling del disco utilizzando simulazioni che modellano un disco per il quale il tempo di ricerca è lineare nel numero di cilindri traversati. Le ottimizzazioni dello scheduling che sfruttano i tempi morti (idle time) dei dischi sono discusse in [Lumb et al. 2000]. [Kim et al. 2009] trattano gli algoritmi di scheduling per le unità SSD.

Le batterie ridondanti di dischi (RAID) sono presentate da [Patterson et al. 1988]

In [Russinovich e Solomon 2009], [McDougall e Mauro 2007] e [Love 2010] sono trattati in dettaglio i file system di Windows, Solaris e Linux, rispettivamente.

La dimensione dei trasferimenti richiesti e la casualità del carico di lavoro hanno un'influenza considerevole sulle prestazioni dell'unità a disco. [Ousterhout et al. 1985], e [Ruemmler e Wilkes 1993] riportano numerose interessanti caratteristiche dei carichi di lavoro, per esempio che la maggior parte dei file sono brevi, la maggior parte dei file creati di recente è eliminata assai presto, che il più delle volte i file aperti per lettura sono letti in modo sequenziale nella loro interezza, e che nella maggior parte dei casi le distanze di ricerca sono brevi.

Il concetto di gestione gerarchica della memorizzazione è stato studiato per più di quarant'anni: un articolo di [Mattson et al. 1970], per esempio, descrive un metodo matematico per prevedere le prestazioni di un sistema di gestione gerarchica della memorizzazione.

Bibliografia

- [Kim et al. 2009] J.Kim, Y.Oh, E.Kim, J.C.D. Lee e S.Noh, “Disk schedulers for solid state drivers”, p. 295–304, 2009.
- [Love 2010] R. Love, Linux Kernel Development, 3° Ed., Developer’s Library, 2010.
- [Lumb et al. 2000] C. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle e E. Riedel, “Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives”, Symposium on Operating Systems Design and Implementation, 2000.

- [**Mattson et al. 1970**] R. L. Mattson, J. Gecsei, D. R. Slutz e I. L. Traiger, “Evaluation Techniques for Storage Hierarchies”, *IBM Systems Journal*, Vol. 9, Num. 2, p. 78–117, 1970.
- [**McDougall e Mauro 2007**] R. McDougall e J. Mauro, *Solaris Internals*, 2° Ed., Prentice Hall, 2007.
- [**Ousterhout et al. 1985**] J.K.Ousterhout,H.D., Costa,D.Harrison, J.A.Kunze, M. Kupfer e J. G. Thompson, “A Trace-Driven Analysis of the UNIX 4.2 BSD File System”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 15–24, 1985.
- [**Patterson et al. 1988**] D. A. Patterson, G. Gibson e R. H. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID)”, Proceedings of the ACM SIGMOD International Conference on the Management of Data, p. 109–116, 1988.
- [**Ruemmler e Wilkes 1993**] C. Ruemmler e J. Wilkes, “Unix Disk Access Patterns”, Proceedings of the Winter USENIX Conference 1993, p. 405–420.
- [**Russinovich e Solomon 2009**] M. E. Russinovich e D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*, 5° Ed., Microsoft Press, 2009.
- [**Services 2012**] E. E. Services, *Information Storage and Management: Storing, Managing, and Protecting Digital Information in Classic, Virtualized, and Cloud Environments*, Wiley, 2012.
- [**Teorey e Pinkerton 1972**] T. J. Teorey e T. B. Pinkerton, “A Comparative Analysis of Disk Scheduling Policies”, *Communications of the ACM*, Vol. 15, Num. 3, p. 177–184, 1972.

CAPITOLO

11

OBIETTIVI DEL CAPITOLO

- Spiegazione della funzione dei file system.
- Descrizione delle interfacce dei file system.
- Presentazione dei tradeoff di progettazione dei file system, compresi metodi d'accesso, condivisione dei file, uso dei lock e strutture della directory.
- Analisi della protezione dei file system.

Interfaccia del file system

Per la maggior parte degli utenti il file system è l'aspetto più visibile di un sistema operativo. Esso fornisce il meccanismo per la memorizzazione in linea di dati e programmi appartenenti al sistema operativo e a tutti gli utenti del sistema elaborativo. Il file system consiste di due parti distinte: un insieme di *file*, ciascuno dei quali contiene i dati, e una *struttura della directory*, che organizza tutti i file nel sistema e fornisce le informazioni relative. I file system si basano sui dispositivi che abbiamo descritto nel precedente capitolo e che continueremo a trattare in seguito. In questo capitolo considereremo i vari aspetti dei file e i principali tipi di strutture della directory. Discuteremo anche la semantica della condivisione dei file fra più processi, utenti e calcolatori. Esamineremo inoltre la gestione della *protezione dei file*, necessaria in un ambiente in cui più utenti hanno accesso ai file, e dove si vuole controllare chi e in che modo vi ha accesso.

11.1 Concetto di file

I calcolatori possono memorizzare le informazioni su diversi supporti, come dischi, nastri magnetici e dischi ottici. Per rendere agevole l’uso del calcolatore, il sistema operativo offre una visione logica uniforme delle informazioni memorizzate; fornisce un’astrazione delle caratteristiche fisiche dei propri dispositivi di memoria definendo un’unità di memorizzazione logica, il *file*. Il sistema operativo associa i file a dispositivi fisici, che di solito sono non volatili, in modo che il loro contenuto non vada perduto a causa dei riavvii del sistema.

Un **file** è un insieme di informazioni correlate, registrate in memoria secondaria, cui è stato assegnato un nome. Dal punto di vista dell’utente, un file è la più piccola porzione di memoria logica secondaria; i dati si possono cioè scrivere in memoria secondaria soltanto all’interno di un file. Di solito i file rappresentano programmi, in forma sorgente e oggetto, e dati. I file di dati possono essere numerici, alfabetici, alfanumerici o binari. I file possono non possedere un formato specifico, come i file di testo, oppure essere rigidamente formattati. In genere un file è formato da una sequenza di bit, byte, righe o *record* il cui significato è definito dal creatore e dall’utente del file stesso. Il concetto di file è quindi estremamente generale.

Le informazioni contenute in un file sono definite dal suo creatore e possono essere di molti tipi: programmi sorgente, programmi oggetto, dati numerici, testo, foto, musica, video, e così via. Un file ha una **struttura** definita secondo il tipo: un file di *testo* è formato da una sequenza di caratteri organizzati in righe, ed eventualmente pagine; un file *sorgente* è formato da una sequenza di funzioni, ciascuna delle quali è a sua volta organizzata in dichiarazioni seguite da istruzioni eseguibili; un file *eseguibile* consiste di una serie di sezioni di codice che il caricatore può caricare in memoria ed eseguire.

11.1.1 Attributi dei file

Per comodità degli utenti, ogni file ha un nome che si usa come riferimento. Un nome, di solito, è una sequenza di caratteri come `esempio.c`. Alcuni sistemi, per quel che riguarda i nomi, distinguono le lettere maiuscole dalle minuscole, altri le considerano equivalenti. Una volta ricevuto il nome, il file diviene indipendente dal processo, dall’utente, e anche dal sistema da cui è stato creato. Per esempio, un utente potrebbe creare il file `esempio.c` e un altro utente potrebbe modificarlo specificandone il nome. Il proprietario del file potrebbe registrare il file in una chiavetta USB, inviarlo per posta elettronica come allegato o copiarlo attraverso la rete, ed esso potrebbe ancora chiamarsi `esempio.c` nel sistema di destinazione.

Un file ha attributi che possono variare secondo il sistema operativo, ma che tipicamente comprendono i seguenti.

- **Nome.** Il nome simbolico del file è l’unica informazione in forma umanamente leggibile.
- **Identificatore.** Si tratta di un’etichetta unica, di solito un numero, che identifica il file all’interno del file system; è il nome impiegato dal sistema per il file.

- **Tipo.** Questa informazione è necessaria ai sistemi che gestiscono tipi di file diversi.
- **Locazione.** Si tratta di un puntatore al dispositivo e alla locazione del file in tale dispositivo.
- **Dimensione.** Si tratta della dimensione corrente del file (in byte, parole o blocchi) ed eventualmente della massima dimensione consentita.
- **Protezione.** Le informazioni di controllo degli accessi controllano chi può leggere, scrivere o eseguire il file.
- **Ora, data e identificazione dell'utente.** Queste informazioni possono essere relative alla creazione, l'ultima modifica e l'ultimo uso. Questi dati possono essere utili ai fini della protezione, della sicurezza e del monitoraggio del suo utilizzo.

Alcuni file system più recenti supportano anche gli attributi estesi dei file, tra cui la codifica dei caratteri del file e funzioni di sicurezza come la checksum. La Figura 11.1 illustra una finestra di informazioni di un file su Mac OS X nella quale sono visualizzati gli attributi di un file.

Le informazioni sui file sono conservate nella struttura della directory, che risiede a sua volta in memoria secondaria. Di solito un elemento di directory consiste di un nome di file e di un identificatore unico, che a sua volta individua gli altri attributi del file. Un elemento di directory può richiedere più di un kilobyte per contenere queste informazioni per ciascun file. In un sistema con molti file, la dimensione della stessa directory può essere dell'ordine dei megabyte. Poiché le directory, come i file, devono essere non volatili, si devono registrare in memoria secondaria e caricare in memoria centrale un po' per volta, secondo le necessità.

11.1.2 Operazioni sui file

Un file è un **tipo di dato astratto**. Per definire adeguatamente un file è necessario considerare le operazioni che si possono eseguire su di esso. Il sistema operativo può offrire chiamate di sistema per creare, scrivere, leggere, spostare, cancellare e troncare un file. Esaminiamo ciò che deve fare un sistema operativo per ciascuna di queste sei operazioni di base. Ciò dovrebbero aiutare a vedere come si possano realizzare altre operazioni simili, per esempio la ridenominazione di un file.

- **Creazione di un file.** Per creare un file è necessario compiere due passaggi. In primo luogo si deve trovare lo spazio per il file nel file system; l'allocazione dello spazio per i file è rimandata al Capitolo 12. Inoltre, per il file si deve creare un nuovo elemento nella directory.
- **Scrittura di un file.** Per scrivere in un file viene effettuata una chiamata di sistema che specifica il nome del file e le informazioni che si vogliono scrivere. Dato il nome del file, il sistema ricerca la directory per individuare la posizione del file. Il file system deve mantenere un *puntatore di scrittura* alla locazione nel file in cui deve avvenire l'operazione di scrittura successiva. Il puntatore si deve aggiornare ogniqualvolta si esegue una scrittura.



Figura 11.1 La finestra di informazioni di un file su Mac OS X.

- **Lettura di un file.** Per leggere da un file è necessaria una chiamata di sistema che specifichi il nome del file e la posizione in memoria dove collocare il blocco del file da leggere. Anche in questo caso si cerca l'elemento corrispondente nella directory e il sistema deve mantenere un *puntatore di lettura* alla locazione nel file in cui deve avvenire la successiva operazione di lettura. Una volta completata la lettura, si aggiorna il puntatore. Di solito un processo o legge o scrive in un file, e la posizione corrente è mantenuta come un **puntatore alla posizione corrente del file** specifico del processo. Sia le operazioni di lettura sia quelle di scrittura adoperano lo stesso puntatore, risparmiando spazio e riducendo la complessità del sistema.
- **Riposizionamento in un file.** Si ricerca l'elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente nel file. Il riposizionamento non richiede alcuna operazione di I/O. Questa operazione è anche nota come *posizionamento* o *ricerca (seek)* nel file.

- **Cancellazione di un file.** Per cancellare un file si cerca l'elemento della directory associato al file designato, si rilascia lo spazio associato al file (in modo che possa essere adoperato per altri) e si elimina l'elemento della directory.
- **Troncamento di un file.** Si potrebbe voler cancellare il contenuto di un file, ma mantenere i suoi attributi. Invece di forzare gli utenti a cancellare il file e quindi ricrearlo, questa funzione consente di mantenere immutati gli attributi (a esclusione della lunghezza del file) pur azzerando la lunghezza del file e rilasciando lo spazio occupato.

Queste sei operazioni di base comprendono l'insieme minimo delle operazioni richieste per i file. Altre operazioni comuni comprendono l'*aggiunta (appending)* di nuove informazioni alla fine di un file esistente e la *ridenominazione* di un file esistente. Queste operazioni primitive si possono combinare per compiere altre operazioni. Per esempio, per creare una *copia* di un file, o per copiare il file in un altro dispositivo di I/O, come una stampante o un video, è sufficiente creare un nuovo file, leggere i dati dal file vecchio e scriverli nel nuovo. Sono inoltre necessarie operazioni che consentano a un utente di leggere e impostare i vari attributi di un file. Per esempio, si potrebbe volere un'operazione che consenta all'utente di determinare lo stato di un file, come la lunghezza, e che consenta di definire gli attributi di un file, come il proprietario.

La maggior parte delle operazioni sopra citate richiede una ricerca nella directory dell'elemento associato al file specificato. Per evitare questa continua ricerca, molti sistemi richiedono l'impiego di una chiamata di sistema `open()` prima che un file venga utilizzato. Il sistema operativo mantiene una tabella contenente informazioni riguardanti tutti i file aperti (detta, per l'appunto, **tabella dei file aperti**). Quando si richiede un'operazione su un file, questo viene individuato tramite un indice in tale tabella, in questo modo si evita qualsiasi ricerca. Quando il file non è più attivamente usato viene *chiuso* dal processo, e il sistema operativo rimuove l'elemento a esso associato dalla tabella dei file aperti. Le chiamate di sistema che lavorano su file chiusi invece che aperti sono `create()` e `delete()`.

Alcuni sistemi aprono implicitamente un file al primo riferimento e lo chiudono automaticamente quando il processo che lo ha aperto termina. La maggior parte dei sistemi invece esige che il programmatore richieda l'apertura del file in modo esplicito per mezzo di una chiamata di sistema `open()` prima che sia possibile adoperarlo. L'operazione `open()` riceve il nome del file, lo cerca nella directory e copia l'elemento della directory a esso associato nella tabella dei file aperti. La chiamata di sistema `open()` può accettare anche informazioni sui modi d'accesso: creazione, sola lettura, lettura e scrittura, sola aggiunta, ecc. Si controllano i permessi relativi al file, e se la modalità d'accesso richiesta è consentita, si apre il file per il processo. La chiamata di sistema `open()` riporta di solito un puntatore all'elemento nella tabella dei file aperti; questo puntatore si adopera al posto dell'effettivo nome del file in tutte le operazioni di I/O, evitando così successive operazioni di ricerca e semplificando l'interfaccia delle chiamate di sistema.

La realizzazione delle operazioni `open()` e `close()` è più complicata in un ambiente multiutente dove più processi possono aprire un file contemporaneamente. Di solito il sistema operativo introduce due livelli di tabelle interne: una tabella per ciascun processo e una tabella di sistema. La tabella del processo contiene i riferimenti a tutti i file aperti da quel processo. In questa tabella sono memorizzate le informazioni sull'uso del file da parte del processo; per esempio, si trovano in questa tabella il puntatore alla posizione corrente per ciascun file e le informazioni sui diritti d'accesso ai file.

Ciascun elemento della tabella associata a ciascun processo punta a sua volta a una tabella di sistema dei file aperti, contenente le informazioni indipendenti dai processi come la posizione dei file nei dischi, le date degli accessi e le dimensioni dei file. Quando un file è stato aperto da un processo, la tabella dei file aperti del sistema contiene un elemento relativo al file; una `open()` eseguita da un altro processo comporta solamente l'aggiunta di un nuovo elemento nella tabella dei file aperti associata al processo, che punta al corrispondente elemento della tabella di sistema. In genere, la tabella dei file aperti ha anche un *contatore delle aperture* associato a ciascun file, indicante il numero di processi che hanno aperto quel file. Ogni `close()` decremente questo *contatore*; quando raggiunge il valore zero il file non è più in uso e si elimina l'elemento corrispondente dalla tabella dei file aperti.

Riassumendo, a ciascun file aperto sono associate le diverse seguenti informazioni.

- **Puntatore al file.** Nei sistemi che non prevedono un offset come parametro delle chiamate di sistema `read()` e `write()`, il sistema deve tener traccia dell'ultima posizione di lettura e scrittura sotto forma di un puntatore alla posizione corrente nel file. Questo puntatore è unico per ogni processo che opera sul file e quindi deve essere tenuto separato dagli attributi del file residenti nel disco.
- **Contatore dei file aperti.** Man mano che si chiudono i file, per evitare di esaurire lo spazio associato alla propria tabella dei file aperti, il sistema operativo deve riutilizzarne gli elementi. Poiché più processi possono aprire uno stesso file, prima di rimuovere l'elemento corrispondente, il sistema deve attendere l'ultima chiusura del file. Questo contatore tiene traccia del numero di `open()` e `close()`, e raggiunge il valore zero dopo l'ultima chiusura, momento in cui il sistema può rimuovere l'elemento della tabella.
- **Posizione nel disco del file.** La maggior parte delle operazioni richiede al sistema di modificare i dati contenuti nel file. L'informazione necessaria per localizzare il file nel disco è mantenuta in memoria, per evitare di doverla prelevare dal disco a ogni operazione.
- **Diritti d'accesso.** Ciascun processo apre un file in una delle modalità d'accesso. Questa informazione è contenuta nella tabella del processo in modo che il sistema operativo possa permettere o negare le successive richieste di I/O.

Alcuni sistemi operativi offrono la possibilità di applicare lock a un file aperto (o a parti di esso). Quando un processo intende proteggere un file dall'accesso concorrente di altri processi, si serve dei lock. L'utilità dei lock dei file emerge nel caso di file condivisi da diversi processi: un file di log, per esempio, può subire modifiche da parte di molti processi nel sistema.

I lock dei file sono basati su una funzionalità simile ai lock di lettura-scrittura (Paragrafo 5.7.2). Un **lock condiviso** è assimilabile, per funzionamento, ai lock di lettura: entrambi consentono a più processi concorrenti di appropriarsene. Un **lock esclusivo** mostra invece analogie con i lock di scrittura, perché un solo processo per volta può acquisire questo tipo di lock. Si noti bene che non in tutti i sistemi operativi forniscono entrambi i tipi di lock; alcuni sistemi forniscono solamente lock esclusivi dei file.

Inoltre, il sistema operativo può fornire meccanismi di lock dei **file obbligatori** (*mandatory*), oppure **consultivi** (*advisory*). Se un lock è obbligatorio, il sistema operativo impedirà a qualunque altro processo di accedere al file interessato una volta che il suo lock sia stato acquisito. Poniamo, per esempio, che un processo acquisisca un lock esclusivo del file `system.log`. Se un altro processo – per esempio, un editor – tentasse di aprire `system.log`, il sistema operativo negherebbe l'accesso finché il lock esclusivo ritorni disponibile. Ciò accade anche se l'editor non è esplicitamente programmato per acquisire il lock. Qualora invece il lock è consultivo, il sistema operativo non impedirà l'accesso dell'editor a `system.log`. Tuttavia, per poter accedere al file, l'editor deve essere scritto in modo tale da acquisire esplicitamente il lock. In altri termini, se il lock è obbligatorio, il sistema operativo assicura l'integrità dei dati soggetti a lock; se il lock è consultivo, è compito dei programmatore garantire la corretta acquisizione e cessione dei lock. In linea generale, i sistemi operativi Windows adottano i lock obbligatori, mentre i sistemi UNIX impiegano i lock consultivi.

L'uso dei lock dei file richiede l'osservazione delle stesse accortezze della sincronizzazione dei processi. Per esempio, i programmatore impegnati a sviluppare su sistemi con lock obbligatori devono prestare attenzione a detenere i lock esclusivi solo per l'effettiva durata degli accessi ai file; in caso contrario, bloccheranno anche gli accessi da parte di altri processi. Occorre, inoltre, attuare misure appropriate al fine di evitare che due o più processi entrino in stallo nel tentativo di acquisire i lock per i file.

11.1.3 Tipi di file

Nella progettazione di un file system, ma anche dell'intero sistema operativo, si deve sempre considerare la possibilità o meno che quest'ultimo riconosca e gestisca i tipi di file. Un sistema operativo che riconosce il tipo di un file ha la possibilità di trattare il file in modo ragionevole. Per esempio, un errore abbastanza comune consiste nel tentativo, da parte degli utenti, di stampare un programma oggetto in forma binaria; di solito questo tentativo porta semplicemente a uno spreco di carta, ma si potrebbe impedire se il sistema operativo fosse informato del fatto che il file è un programma oggetto in forma binaria.

APPLICAZIONE DI LOCK NEL LINGUAGGIO JAVA

L'acquisizione del lock di un file tramite la API Java prevede in primo luogo l'acquisizione del `FileChannel` relativo al file; il metodo `lock()` del `FileChannel` permette poi di ottenere il lock. Il prototipo del metodo `lock()` è:

```
FileLock lock(long begin, long end, boolean shared)
```

dove `begin` ed `end` sono il punto iniziale e finale della parte da sottoporre a lock. Se `shared` vale `true`, si ottiene un lock condiviso; altrimenti, un lock esclusivo. Il lock si rilascia tramite il metodo `release()` invocato sull'oggetto `FileLock` restituito da `lock()`.

Il programma della Figura 11.2 illustra questa tecnica. Esso acquisisce due lock del file `file.txt`. La prima metà del file è soggetta a lock esclusivo, la seconda a lock condiviso.

```
import java.io.*;
import java.nio.channels.*;

public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;

    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;

        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");

            // acquisisce il canale per il file
            FileChannel ch = raf.getChannel();

            // acquisisce lock esclusivo per la prima metà del file
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

            /* Modifica i dati . . . */

            // rilascia il lock
            exclusiveLock.release();

            // acquisisce lock condiviso per la seconda metà del file
            sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);

            /* Legge i dati . . . */

            // rilascia il lock
            sharedLock.release();
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (exclusiveLock != null)
                exclusiveLock.release();
            if (sharedLock != null)
                sharedLock.release();
        }
    }
}
```

Figura 11.2 Esempio di applicazione di lock a un file in Java.

Tipi di file	Estensione usuale	Funzione
Eseguibile	exe, com, bin, o nessuna	Programma eseguibile, in linguaggio macchina
Oggetto	obj, o	Compilato, in linguaggio di macchina, non linkato
Codice sorgente	c, cc, java, perl, asm	Codice sorgente in vari linguaggi di programmazione
Batch	bat, sh	Comandi per l'interprete dei comandi
Markup	xml, html, tex	Dati testuali, documenti
Word processor	xml, rtf, docx	Vari formati di word processor
Libreria	lib, a, so, dll	Librerie di procedure per la programmazione
Stampa o visualizzazione	gif, pdf, jpg	File ASCII o binari in formato per la stampa o la visualizzazione
Archivio	rar, zip, tar	File contenenti più file tra loro correlati, talvolta compressi, per archiviazione o memorizzazione
Multimediali	mpeg, mov, mp3, mp4, avi	File binari contenenti informazioni audio o A/V

Figura 11.3 Comuni tipi di file.

Una tecnica comune per realizzare la gestione dei tipi di file consiste nell'includere il tipo nel nome del file. Il nome è suddiviso in due parti, un nome e un'estensione, di solito separate da un punto (Figura 11.3); in questo modo l'utente e il sistema operativo possono risalire al tipo del file semplicemente esaminandone il nome. La maggior parte dei sistemi operativi permette agli utenti di specificare i nomi dei file come sequenze di caratteri seguite da un punto e concluse da un'estensione formata da caratteri aggiuntivi. Esempi di nomi di file sono `resume.docx`, `server.c` e `ReaderThread.cpp`. Il sistema usa l'estensione per stabilire il *tipo* del file e le operazioni che si possono eseguire su tale file. Per esempio, solamente i file con estensione `.com`, `.exe` o `.sh` sono eseguibili; i file con estensione `.com` e `.exe` sono due formati di file eseguibili, mentre i file con estensione `.sh` sono script di shell contenenti una sequenza di comandi, scritti in formato ASCII, diretti al sistema operativo. Anche le applicazioni utilizzano estensioni per indicare i tipi di file di proprio interesse. Per esempio, i compilatori Java si aspettano sorgenti con un'estensione `.java` e l'elaboratore di testi Microsoft Word si aspetta file con estensione `.doc` o `.docx`.

Queste estensioni non sono sempre necessarie, ma la loro presenza consente a un utente di ridurre il numero dei caratteri specificando il nome del file senza estensione e lasciando all'applicazione il compito di cercare il file con il nome impostato e l'e-

stensione attesa. Poiché queste estensioni non sono gestite dal sistema operativo, si possono considerare un suggerimento rivolto alle applicazioni che operano su di loro.

Consideriamo, inoltre, il sistema operativo Mac OS X. In questo sistema ciascun file ha un tipo, per esempio .app per le applicazioni. Ciascun file possiede anche un attributo di creazione contenente il nome del programma che lo ha creato. Questo attributo è impostato dal sistema operativo durante la chiamata di sistema `create()`, quindi il suo utilizzo è forzato e supportato dal sistema operativo. Per esempio, un file prodotto da un elaboratore di testi avrà il nome dell'elaboratore di testi come attributo di creazione. Quando un utente apre il file, con un doppio clic del mouse sull'icona che lo rappresenta, si attiva automaticamente l'elaboratore di testi che apre il file, pronto per essere letto e modificato.

Il sistema operativo UNIX non fornisce una funzione di questo tipo, ma si limita a memorizzare un semplice codice (noto come **magic number**) all'inizio di alcuni tipi di file allo scopo di indicarne in modo generico il tipo: programma eseguibile, script di shell, file pdf, e così via. Non tutti i file possiedono tale codice, quindi il sistema non può affidarsi unicamente a questo tipo d'informazione; inoltre, UNIX non memorizza il nome del programma che ha creato il file. UNIX consente di sfruttare le estensioni come suggerimento del tipo di file; queste non vengono però imposte né vengono utilizzate dal sistema operativo; il loro compito consiste principalmente nell'aiutare gli utenti a riconoscere il tipo di contenuto del file. Un'applicazione può usare o ignorare le estensioni; dipende dalle scelte dei programmati.

11.1.4 Struttura dei file

I tipi di file si possono anche adoperare per indicare la struttura interna dei file. Come si è accennato nel Paragrafo 11.1.3, i file sorgente e i file oggetto hanno una struttura corrispondente a ciò che il programma che dovrà leggerli si attende. Inoltre alcuni file devono rispettare una determinata struttura comprensibile al sistema operativo. Per esempio, il sistema operativo richiede che un file eseguibile abbia una struttura specifica che consenta di determinare dove caricare il file in memoria e quale sia la locazione della prima istruzione. Alcuni sistemi operativi estendono questa idea a un insieme di strutture di file supportate dal sistema, con un insieme di operazioni specifiche per la manipolazione dei file con queste strutture.

L'analisi precedente ci porta a uno degli svantaggi dei sistemi operativi che gestiscono più strutture di file: la dimensione risultante del sistema operativo è rilevante. Se definisce cinque strutture di file differenti, il sistema operativo deve contenere il codice per gestirle tutte. Inoltre qualsiasi file potrebbe dover essere definito come uno dei tipi gestiti dal sistema operativo: ciò provoca notevoli problemi se nuove applicazioni richiedono una strutturazione dei propri dati in modi non previsti dal sistema operativo.

Per esempio, si supponga che un sistema operativo preveda due tipi di file: file di testo (composti da righe di caratteri ASCII separate da caratteri di ritorno del carrello e avanzamento di riga) e file binari eseguibili. Un utente che volesse definire un file

cifrato per proteggere i propri dati da letture non autorizzate potrebbe scoprire che nessuna delle due strutture si adatta al problema: non è un file di righe di testo ASCII, ma un insieme di bit (apparentemente casuali), e sebbene possa sembrare un file binario, non è eseguibile. Queste limitazioni impongono all'utente di bypassare o usare in modo scorretto il meccanismo dei tipi di file definito dal sistema operativo, oppure di abbandonare lo schema di codifica.

Alcuni sistemi operativi impongono (e supportano) un numero minimo di strutture di file. Questo orientamento è stato seguito da UNIX, Windows e altri. UNIX considera ciascun file come una sequenza di byte, senza alcuna interpretazione da parte del sistema operativo. Questo schema garantisce la massima flessibilità, ma il minimo supporto. Qualsiasi programma applicativo deve contenere il proprio codice per interpretare in modo appropriato la struttura di un file in ingresso. A ogni modo, per poter caricare ed eseguire i programmi, tutti i sistemi operativi devono prevedere almeno un tipo di struttura, quella dei file eseguibili.

11.1.5 Struttura interna dei file

Per il sistema operativo la localizzazione di un offset all'interno di un file può essere complicata. I dischi hanno una dimensione dei blocchi ben definita, determinata dalla dimensione di un settore. Tutti gli I/O su disco si eseguono in unità di un blocco (record fisico), e tutti i blocchi hanno la stessa dimensione. È improbabile che la dimensione del record fisico corrisponda esattamente alla lunghezza del record logico desiderato, che può anche essere variabile. Una soluzione diffusa per questo tipo di problema consiste nell'**impaccamento** di un certo numero di record logici in blocchi fisici.

Il sistema operativo UNIX, per esempio, definisce tutti i file semplicemente come un flusso di byte. A ciascun byte si può accedere in modo individuale tramite il suo offset a partire dall'inizio, o dalla fine, del file. In questo caso il record logico è un byte. Il file system *impacca* automaticamente i byte in blocchi fisici (per esempio 512 byte per blocco) com'è necessario.

La dimensione dei record logici, quella dei blocchi fisici e la tecnica d'impaccamento determinano il numero dei record logici all'interno di ogni blocco fisico. L'impaccamento può essere fatto dal programma applicativo dell'utente oppure dal sistema operativo.

In entrambi i casi il file si può considerare come una sequenza di blocchi. Tutte le funzioni di I/O di base operano in termini di blocchi. La conversione da record logici a blocchi fisici è un problema di programmazione relativamente semplice.

Poiché lo spazio del disco è sempre assegnato in blocchi, una parte dell'ultimo blocco di ogni file in genere è sprecata. Se ogni blocco è composto di 512 byte, a un file di 1949 byte si assegnano quattro blocchi (2048 byte); gli ultimi 99 byte sono sprecati. I byte sprecati, a causa della gestione in multipli di blocchi invece che di byte, costituiscono la **frammentazione interna**. Tutti i file system ne soffrono; maggiore è la dimensione dei blocchi, maggiore sarà la frammentazione interna.

11.2 Metodi d'accesso

I file memorizzano informazioni; al momento dell'uso è necessario accedere a queste informazioni e trasferirle in memoria. Esistono molti metodi per accedere alle informazioni dei file; alcuni sistemi consentono un solo metodo d'accesso ai file, mentre altri offrono diversi metodi d'accesso: in questo caso la scelta del metodo giusto per una particolare applicazione è un importante problema di progettazione.

11.2.1 Accesso sequenziale

Il più semplice metodo d'accesso è l'**accesso sequenziale**: le informazioni del file si elaborano ordinatamente, un record dopo l'altro; questo metodo d'accesso è di gran lunga il più comune, ed è usato, per esempio, dagli editor e dai compilatori.

Le più comuni operazioni che si compiono sui file sono le letture e le scritture: un'operazione di lettura – `read_next()` – legge la prossima porzione di file e fa avanzare automaticamente il puntatore del file che tiene traccia della locazione di I/O; analogamente, un'operazione di scrittura – `write_next()` – fa un'aggiunta in coda al file e avanza fino alla fine delle informazioni appena scritte, che costituisce la nuova fine del file. Un file siffatto si può reimpostare sull'inizio e, in alcuni sistemi, un programma può riuscire ad andare avanti o indietro di n record, con n intero (in alcuni casi solo per $n = 1$). L'accesso sequenziale è illustrato nella Figura 11.4. L'accesso sequenziale utilizza per il file il modello del nastro magnetico, e funziona nei dispositivi ad accesso sequenziale così come nei dispositivi ad accesso diretto.

11.2.2 Accesso diretto

Un altro metodo è l'**accesso diretto** (o **accesso relativo**). In questo caso, un file è formato da elementi logici (**record**) di lunghezza fissa; ciò consente ai programmi di leggere e scrivere rapidamente tali elementi senza un ordine particolare. Il metodo ad accesso diretto si fonda su un modello di file che si rifà al disco: i dischi permettono, infatti, l'accesso diretto a ogni blocco di file. Il file si considera come una sequenza numerata di blocchi o record: si può per esempio leggere il blocco 14, quindi il blocco 53 e poi scrivere il blocco 7. Non esistono restrizioni all'ordine di lettura o scrittura di un file ad accesso diretto.

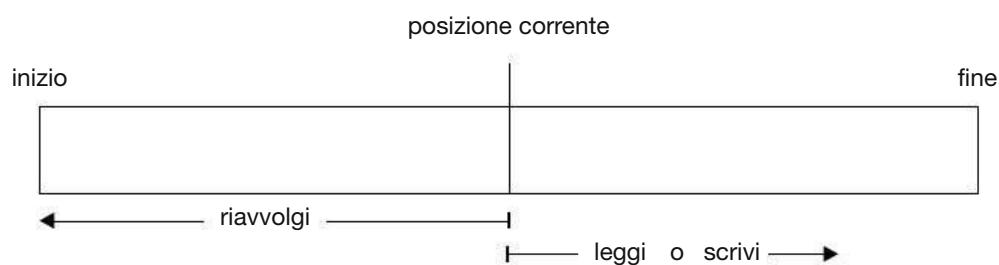


Figura 11.4 File ad accesso sequenziale.

I file ad accesso diretto sono molto utili quando è necessario accedere immediatamente a grandi quantità di informazioni. Spesso le basi di dati sono di questo tipo: quando si presenta un'interrogazione riguardante un oggetto particolare, occorre determinare quale blocco contiene la risposta alla richiesta e quindi leggere direttamente quel blocco, ottenendo così le informazioni richieste.

Per esempio, in un sistema di prenotazione dei voli, potremmo registrare tutte le informazioni su un particolare volo, per esempio il volo 713, nel blocco identificato da tale numero di volo. Quindi, il numero di posti disponibili per il volo 713 si memorizza nel blocco 713 del file di prenotazione. Per registrare informazioni riguardanti un gruppo più grande, per esempio una popolazione, si può eseguire la ricerca calcolando una funzione hash sui nomi delle persone, oppure accedere a un piccolo indice residente in memoria per determinare il blocco da leggere e scandire.

Per il metodo ad accesso diretto, si devono modificare le operazioni sui file per inserire il numero del blocco in forma di parametro. Quindi, si hanno `read(n)`, dove n è il numero del blocco, al posto di `read_next()`, e `write(n)`, invece che `write_next()`. Un metodo alternativo prevede di mantenere `read_next()` e `write_next()`, come nell'accesso sequenziale, e di aggiungere un'operazione `position_file(n)`, dove n è il numero del blocco. Quindi un'operazione `read(n)` corrisponde a una `position_file(n)` e una `read_next()`.

Il numero del blocco fornito dall'utente al sistema operativo è normalmente un **numero di blocco relativo**. Si tratta di un indice relativo all'inizio del file, quindi il primo blocco relativo del file è 0, il successivo è 1 e così via, anche se l'indirizzo assoluto nel disco del blocco può essere 14703 per il primo blocco e 3192 per il secondo. L'uso dei numeri di blocco relativi permette al sistema operativo di decidere dove posizionare il file (si tratta del *problema dell'allocazione* trattato nel Capitolo 12) e aiuta a impedire che l'utente acceda a porzioni del file system che possono non far parte del suo file. Alcuni sistemi iniziano la numerazione dei blocchi relativi da 0, altri da 1.

Come soddisfa il sistema operativo una richiesta di un record N in un file? Assumendo che la lunghezza del record logico sia l , una richiesta per il record N determina una richiesta di I/O per l byte a partire dalla locazione $l * N$ all'interno del file (assumendo che il primo record sia $N = 0$). Lettura, scrittura e cancellazione di un record sono rese semplici dalla sua dimensione fissa.

Non tutti i sistemi operativi gestiscono ambedue i tipi di accesso; alcuni permettono il solo accesso sequenziale, altri solo quello diretto. Alcuni sistemi richiedono che si definisca il tipo d'accesso al file al momento della sua creazione; a tale file si può accedere soltanto nel modo definito. Tuttavia si può facilmente simulare l'accesso sequenziale a un file ad accesso diretto mantenendo una variabile cp che, come illustra la Figura 11.5, definisce la nostra posizione corrente. D'altra parte è estremamente macchinoso e inefficiente simulare l'accesso diretto a un file che di per sé è ad accesso sequenziale.

Accesso sequenziale	Realizzazione nel caso di accesso diretto
reset	<code>cp = 0;</code>
read_next()	<code>read cp; cp = cp + 1;</code>
write_next()	<code>write cp; cp = cp + 1;</code>

Figura 11.5 Simulazione dell'accesso sequenziale a un file ad accesso diretto.

11.2.3 Altri metodi d'accesso

Sulla base di un metodo d'accesso diretto se ne possono costruire altri, che implicano generalmente la costruzione di un indice per il file. L'**indice** (*index*) contiene puntatori ai vari blocchi; per trovare un elemento del file occorre prima cercare nell'indice, e quindi usare il puntatore per accedere direttamente al file e trovare l'elemento desiderato.

Si consideri, per esempio, un file contenente prezzi al dettaglio, contenente una lista dei codici universali dei prodotti (*universal product codes*, UPC), a ciascuno dei quali è associato un prezzo. Dato un record di 16 byte, questo è composto da un codice UPC a 10 cifre e un prezzo a 6 cifre. Se il disco usato ha 1024 byte per blocco, in ogni blocco si possono memorizzare 64 record. Un file di 120.000 record occupa circa 2000 blocchi (2 milioni di byte). Ordinando il file secondo il codice UPC si può definire un indice composto dal primo codice UPC di ogni blocco. Tale indice è costituito di 2000 elementi di 10 cifre ciascuno (20.000 byte) e quindi può essere tenuto in memoria. Per trovare il prezzo di un oggetto specifico si può fare una ricerca binaria nell'indice, che permette di sapere esattamente quale blocco contiene l'elemento desiderato e quindi accedere a quel blocco. Questa struttura permette di compiere ricercate in file molto grandi limitando il numero di operazioni di I/O.

Nel caso di file molto lunghi, lo stesso file indice può diventare troppo lungo perché sia tenuto in memoria. Una soluzione a questo problema è data dalla creazione di un indice per il file indice. Il file indice principale contiene puntatori ai file indice secondari, che puntano agli effettivi elementi di dati.

Il metodo ad accesso sequenziale indicizzato di IBM (*indexed sequential access method*, ISAM), per esempio, usa un piccolo indice principale che punta ai blocchi del disco di un indice secondario, e i blocchi dell'indice secondario puntano ai blocchi del file effettivo. Il file è ordinato rispetto a una chiave definita. Per trovare un particolare elemento, si fa inizialmente una ricerca binaria nell'indice principale, che fornisce il numero del blocco dell'indice secondario. Questo blocco viene letto e sotto-posto a una seconda ricerca binaria che individui il blocco contenente l'elemento richiesto. Infine, si fa una ricerca sequenziale sul blocco. In questo modo si può localizzare ogni elemento tramite la sua chiave con al massimo due letture ad accesso diretto. La Figura 11.6 mostra uno schema simile, com'è realizzato nel VMS con indici e relativi file.

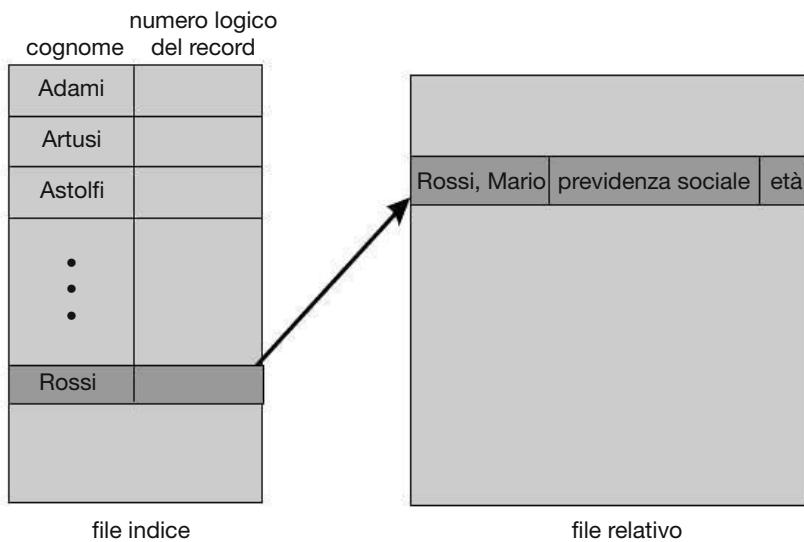


Figura 11.6 Esempio di indice e relativi file.

11.3 Struttura della directory e del disco

Si prenderanno ora in considerazione le modalità di memorizzazione dei file. Ovviamente, nessun computer a uso generale contiene un unico file. In un computer sono infatti memorizzati spesso migliaia, milioni o persino miliardi di file. I file vengono salvati in dispositivi di memorizzazione ad accesso casuale, come dischi fissi, dischi ottici e dischi a stato solido (basati sulla memoria).

Un dispositivo di memorizzazione può essere interamente utilizzato per un file system, ma può anche essere suddiviso per un controllo più raffinato. Un disco può per esempio essere **partizionato** e ogni partizione può contenere un file system. I dispositivi di memorizzazione possono essere raccolti in insiemi RAID che proteggono dal guasto di un singolo disco (come descritto al Paragrafo 10.7). Alcune volte, i dischi sono suddivisi e al contempo raccolti in insiemi RAID.

La suddivisione in partizioni è utile anche per limitare la dimensione dei singoli file system, per mettere sullo stesso dispositivo tipi diversi di file system, oppure per liberare per altri scopi una parte del dispositivo, come nel caso dello spazio di swap o dello spazio su disco non formattato (**raw**). Ogni entità contenente un file system è generalmente nota come **volume**. Il volume può essere un sottoinsieme di un dispositivo, un dispositivo intero o dispositivi multipli collegati in RAID. Ogni volume può essere pensato come un disco virtuale. I volumi possono inoltre contenere diversi sistemi operativi, permettendo al sistema di avviare ed eseguire più di un sistema operativo.

Ogni volume contenente un file system deve anche avere in sé le informazioni sui file presenti nel sistema. Tali informazioni risiedono in una **directory del dispositivo** o **indice del volume**. La directory del dispositivo (in breve **directory**) registra informazioni, quali nome, posizione, dimensioni e tipo, di tutti i file del volume. La Figura 11.7 mostra una tipica organizzazione dei file system.

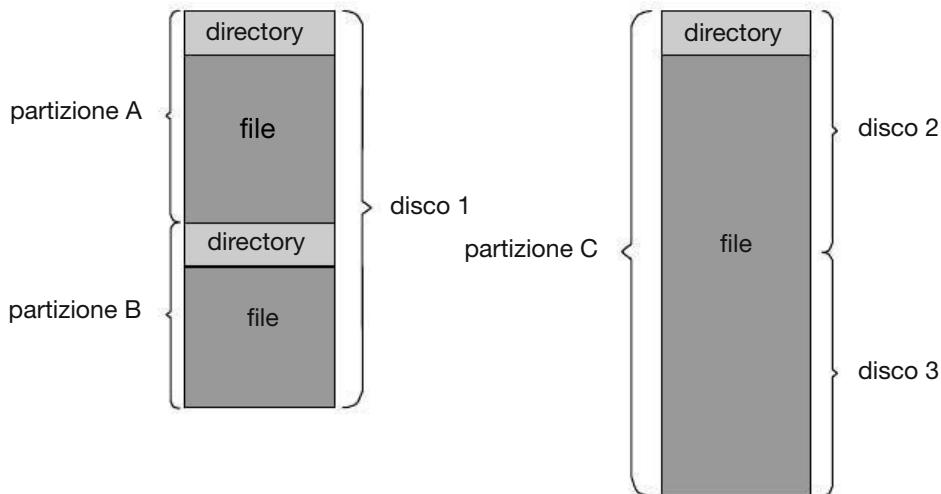


Figura 11.7 Tipica organizzazione di un file system.

11.3.1 Struttura della memorizzazione di massa

Come discusso, un sistema informatico a uso generale dispone di dispositivi di memorizzazione multipli, che possono essere ripartiti nei volumi che contengono i file system. Il numero di file system in un sistema informatico può variare da zero a molti, e possono essere di tipo diverso. Per esempio, un tipico sistema Solaris può avere molti file system di una dozzina di tipi diversi, come evidenziato nella Figura 11.8.

In questo libro prenderemo in considerazione solo file system di utilizzo generico. Vale la pena però di notare che esistono molti file system per scopi specifici. Consideriamo i tipi di file system dell'esempio menzionato precedentemente riguardante Solaris:

- **tmpfs** – un file system “temporaneo” creato nella memoria centrale volatile e i cui contenuti vengono cancellati se il sistema si riavvia o si blocca;
- **objfs** – un file system “virtuale” (essenzialmente un’interfaccia con il kernel che è simile a un file system) che permette agli strumenti che eseguono il debug di accedere ai simboli del kernel;
- **ctfs** – un file system virtuale che mantiene le informazioni “contrattuali” per gestire quali sono i processi che partono quando il sistema si avvia e quali devono continuare a essere eseguiti durante il funzionamento del sistema;
- **lofs** – un file system di tipo “loop back” che permette di accedere a un file system al posto di un altro;
- **procfs** – un file system virtuale che presenta le informazioni su tutti i processi presenti come se esse risiedessero su un file system;
- **ufs, zfs** – file system a scopo generico.

I file system dei computer possono quindi essere numerosi. Persino all’interno di un file system è utile dividere i file in gruppi da gestire e sui quali agire collettivamente. Un’organizzazione di questo tipo richiede l’utilizzo di directory, argomento che esamineremo nel resto di questo paragrafo.

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	procfs
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fdfs
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

Figura 11.8 File system in Solaris.

11.3.2 Aspetti generali delle directory

La directory si può considerare come una tabella di simboli che traduce i nomi dei file negli elementi in essa contenuti. Considerando questo punto di vista, si capisce che la stessa directory si può organizzare in molti modi diversi; l'organizzazione deve rendere possibile l'inserimento di nuovi elementi, la cancellazione di elementi esistenti, la ricerca di un elemento, e l'elenco di tutti gli elementi della directory. In questo paragrafo esaminiamo diversi schemi per definire la struttura logica del sistema di directory. Nel considerare una particolare struttura della directory si deve tenere presente l'insieme delle seguenti operazioni che si possono eseguire su una directory.

- **Ricerca di un file.** Deve esserci la possibilità di scorrere una directory per individuare l'elemento associato a un particolare file. Poiché i file possono avere nomi simbolici, e poiché nomi simili possono indicare relazioni tra file, deve esistere la possibilità di trovare tutti i file il cui nome soddisfi una particolare espressione.
- **Creazione di un file.** Deve essere possibile creare nuovi file e aggiungerli alla directory.
- **Cancellazione di un file.** Quando non serve più, si deve poter rimuovere un file dalla directory.

- **Elencazione di una directory.** Deve esistere la possibilità di elencare tutti i file di una directory, e il contenuto degli elementi della directory associati a ciascun file nell’elenco.
- **Ridenominazione di un file.** Poiché il nome di un file rappresenta per i suoi utenti il contenuto del file, questo nome deve poter essere modificato quando il contenuto o l’uso del file subiscono cambiamenti. La ridenominazione di un file potrebbe anche permettere la variazione della posizione del file nella directory.
- **Attraversamento del file system.** Si potrebbe voler accedere a ogni directory e a ciascun file contenuto in una directory. Per motivi di affidabilità, è opportuno salvare il contenuto e la struttura dell’intero file system a intervalli regolari. Questo salvataggio consiste nella copiatura di tutti i file in un nastro magnetico; tale tecnica consente di avere una copia di riserva (*backup*) che sarebbe utile nel caso in cui si dovesse verificare un guasto nel sistema. Inoltre, se un file non è più in uso, si può copiarlo su nastro e liberare lo spazio da esso occupato nel disco, rendendolo riutilizzabile per altri file.

Nei paragrafi seguenti sono descritti gli schemi più comuni per la definizione della struttura logica di una directory.

11.3.3 Directory a un livello

La struttura più semplice per una directory è quella a un livello. Tutti i file sono contenuti nella stessa directory, facilmente gestibile e comprensibile (Figura 11.9).

Una directory a un livello presenta però limiti notevoli quando aumenta il numero dei file in essa contenuti, oppure se il sistema è usato da più utenti. Poiché si trovano tutti nella stessa directory, i file devono avere nomi unici; se due utenti attribuiscono lo stesso nome al loro file di dati, per esempio `test.txt`, si viola la regola del nome unico. Come esempio, si consideri una classe di programmazione in cui 23 studenti chiamano il programma del secondo esercizio `prog2.c` e altri 11 chiamano lo stesso programma `compito2.c`. Fortunatamente, la maggior parte dei file system supporta nomi di file di 255 caratteri, per cui è abbastanza semplice assegnare ai file un nome univoco.

Anche per un solo utente, con una directory a un livello, diventa difficile ricordare i nomi dei file con l’aumentare del loro numero. Non è affatto raro che un utente abbia centinaia di file in un calcolatore e altrettanti file in un altro sistema. In un tale ambiente, sarebbe un compito arduo dover ricordare tanti nomi di file.

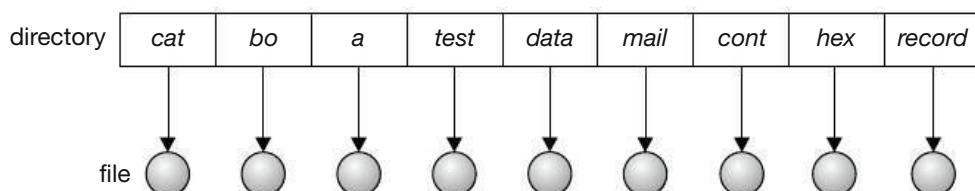


Figura 11.9 Directory a livello singolo.

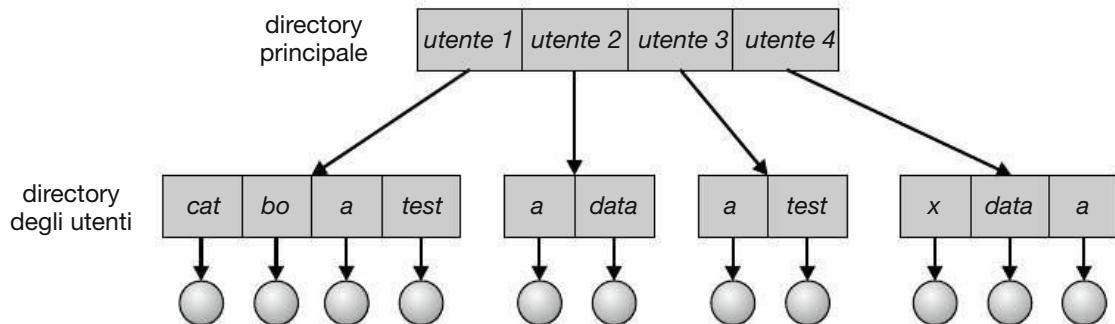


Figura 11.10 Struttura della directory a due livelli.

11.3.4 Directory a due livelli

Come abbiamo visto, una directory a un livello spesso causa la confusione dei nomi dei file tra diversi utenti. La soluzione più ovvia prevede la creazione di una directory separata per ogni utente.

Nella struttura a due livelli, ogni utente dispone della propria **directory utente** (*userfile directory*, UFD). Tutte le directory utente hanno una struttura simile, ma in ciascuna sono elencati solo i file del proprietario. Quando comincia l'elaborazione di un lavoro dell'utente, oppure un utente inizia una sessione di lavoro, si fa una ricerca nella **directory principale** (*master file directory*, MFD) del sistema. La directory principale viene indicizzata con il nome dell'utente o il numero di account, e ogni suo elemento punta alla relativa directory utente (Figura 11.10).

Quando un utente fa un riferimento a un file particolare, il sistema operativo esegue la ricerca solo nella directory di quell'utente. In questo modo utenti diversi possono avere file con lo stesso nome, purché tutti i nomi di file all'interno di ciascuna directory utente siano unici. Per creare un file per un utente, il sistema operativo controlla che non ci sia un altro file con lo stesso nome soltanto nella directory di tale utente. Per cancellare un file il sistema operativo limita la propria ricerca alla directory utente locale, quindi non può cancellare per errore un file con lo stesso nome che appartenga a un altro utente.

Le stesse directory utente devono essere create e cancellate quando è necessario; a tale scopo si esegue uno speciale programma di sistema con nome dell'utente e informazioni relative all'account. Il programma crea una nuova directory utente e aggiunge l'elemento a essa corrispondente nella directory principale. L'esecuzione di questo programma può essere limitata all'amministratore del sistema. L'allocatione dello spazio nei dischi per le directory utente può essere gestita con le tecniche descritte per i file nel Capitolo 12.

Sebbene risolva la questione delle collisioni dei nomi, la struttura della directory a due livelli presenta ancora dei problemi. In effetti, questa struttura isola un utente dagli altri. Questo isolamento può essere un vantaggio quando gli utenti sono completamente indipendenti, ma è uno svantaggio quando più utenti *vogliono* cooperare e accedere ai rispettivi file. Alcuni sistemi non permettono l'accesso a file di utenti locali da parte di altri utenti.

Se l’accesso è autorizzato, un utente deve avere la possibilità di riferirsi al nome di un file che si trova nella directory di un altro utente. Per attribuire un nome unico a un particolare file di una directory a due livelli, occorre indicare sia il nome dell’utente sia il nome del file. Una directory a due livelli si può pensare come un albero di altezza 2. La radice dell’albero è la directory principale, i suoi diretti discendenti sono le directory utente, da cui discendono i file che sono le foglie dell’albero. Specificando un nome utente e un nome di file si definisce un percorso che parte dalla radice (la directory principale) e arriva a una specifica foglia (il file specificato). Quindi, un nome utente e un nome di file definiscono un *nome di percorso* (*path name*). Ogni file del sistema ha un nome di percorso. Per attribuire un nome unico a un file, un utente deve conoscere il nome di percorso del file desiderato.

Se, per esempio, l’utente *A* desidera accedere al proprio file chiamato *prova.txt*, è sufficiente che faccia riferimento a *prova.txt*. Invece, per accedere al file denominato *prova.txt* dell’utente *B*, con nome di elemento della directory *utenteB*, l’utente *A* deve fare riferimento a */utenteB/prova.txt*. Ogni sistema ha la propria sintassi per riferirsi ai file delle directory diverse da quella dell’utente.

Per specificare il volume cui appartiene un file occorrono ulteriori regole sintattiche. In Windows, per esempio, il volume è indicato da una lettera seguita dai due punti. Quindi l’indicazione di un file potrebbe essere del tipo *C:\utenteB\prova*. Alcuni sistemi vanno oltre e separano: volume, nome della directory, e nome del file. Nel VMS, per esempio, il file *login.com* potrebbe essere indicato come *u:[sst.jdeck]login.com;1*, dove *u* è il nome del volume, *sst* è il nome della directory, *jdeck* è il nome della sottodirectory e *1* è il numero della versione. Altri sistemi, come UNIX e Linux, trattano il nome del volume semplicemente come parte del nome della directory. Il primo elemento è quello del volume, il resto è composto dalla directory e dal file. Per esempio, */u/pbg/prova* potrebbe indicare il volume *u*, la directory *pbg* e il file *prova*.

Un caso particolare di questa situazione riguarda i file di sistema. I programmi forniti come elementi integranti del sistema, come loader, assemblatori, compilatori, utilità di sistema, librerie e così via, sono infatti definiti come file. Quando si imparatiscono al sistema operativo i comandi appropriati, il caricatore legge questi file che poi vengono eseguiti. Molti interpreti di comandi operano semplicemente trattando questo comando come il nome di un file da caricare ed eseguire. Nel sistema di directory che abbiamo definito, questo nome di file viene cercato nella directory utente locale. Una soluzione potrebbe essere la copiatura dei file di sistema in ciascuna directory utente. Tuttavia, con la copiatura di tutti i file di sistema si spreca un’enorme quantità di spazio. Se i file di sistema occupano 5 MB, con 12 utenti si avrebbe un’occupazione di spazio pari a $5 \times 12 = 60$ MB, solo per le copie dei file di sistema.

La soluzione standard prevede una leggera complicazione della procedura di ricerca; si definisce una speciale directory utente contenente i file di sistema, per esempio la directory utente 0. Ogni volta che si indica un file da caricare, il sistema operativo lo cerca innanzitutto nella directory utente locale, e, se lo trova, lo usa; se non lo trova, il sistema cerca automaticamente nella speciale directory utente contenente

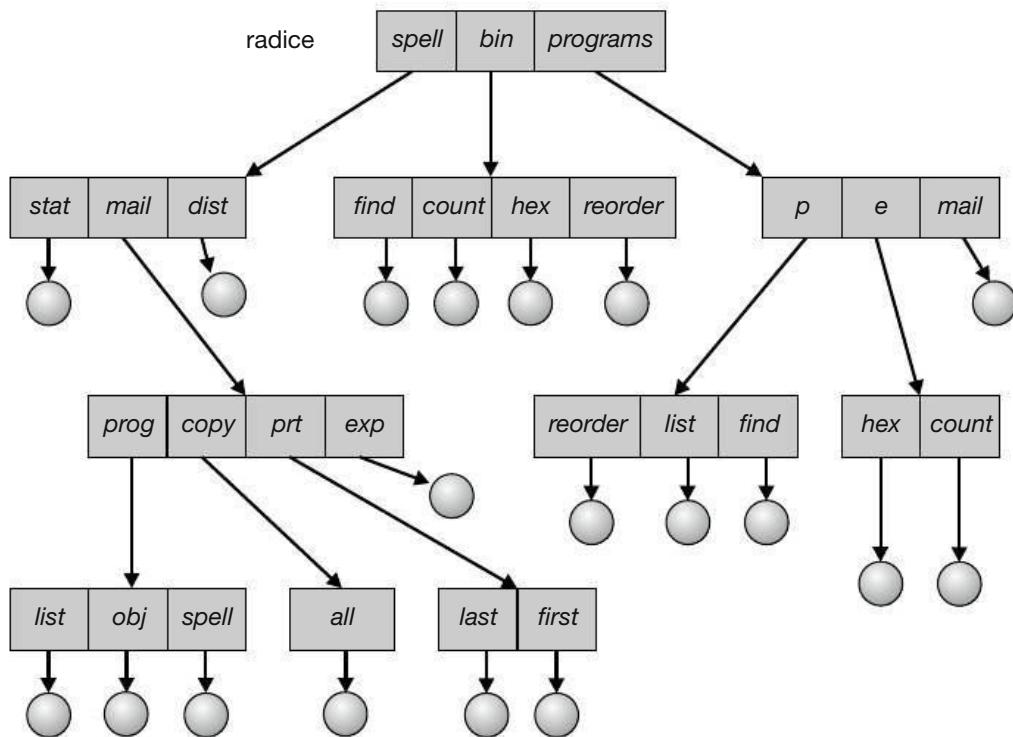


Figura 11.11 Struttura della directory ad albero.

i file di sistema. La sequenza delle directory in cui è stata fatta la ricerca avviata dal riferimento a un file è detta **percorso di ricerca** (*search path*). Tale percorso si può estendere in modo da contenere una lista illimitata di directory in cui fare le ricerche quando si dà il nome di un comando. Questo metodo è il più usato in UNIX e Windows. Alcuni sistemi prevedono anche che ogni utente disponga del proprio percorso di ricerca personale.

11.3.5 Directory con struttura ad albero

La corrispondenza strutturale tra directory a due livelli e albero a due livelli può essere facilmente generalizzata, estendendo la struttura della directory a un albero di altezza arbitraria (Figura 11.11). Questa generalizzazione permette agli utenti di creare proprie sottodirectory e di organizzare i file di conseguenza. Un albero è il più comune tipo di struttura delle directory. L'albero ha una directory radice (*root directory*), e ogni file del sistema ha un unico nome di percorso.

Una directory, o una sottodirectory, contiene un insieme di file o sottodirectory. Le directory sono semplicemente file, trattati però in modo speciale. Tutte le directory hanno lo stesso formato interno. La distinzione tra file e directory è data da un bit di ogni elemento della directory. Per creare e cancellare le directory si adoperano speciali chiamate di sistema.

Normalmente, ogni utente dispone di una directory corrente. La **directory corrente** dovrebbe contenere la maggior parte dei file di interesse corrente per il proces-

so. Quando si fa un riferimento a un file, si esegue una ricerca nella directory corrente; se il file non si trova in tale directory, l’utente deve specificare un nome di percorso oppure cambiare la directory corrente facendo diventare tale la directory contenente il file desiderato. Per cambiare directory corrente si fa uso di una chiamata di sistema che prende un nome di directory come parametro e lo usa per ridefinire la directory corrente. Quindi, l’utente può cambiare la propria directory corrente ogni volta che lo desidera. Da una chiamata di sistema `change_directory()` alla successiva, tutte le chiamate di sistema `open()` cercano i file specificati nella directory corrente. Si noti che il percorso di ricerca può contenere o meno uno speciale elemento che rappresenta “la directory corrente”.

La directory corrente iniziale di un utente è stabilita quando viene lanciato un job dell’utente, oppure quando questi inizia una sessione di lavoro; il sistema operativo cerca nel file di accounting, o in qualche altra locazione predefinita, l’elemento relativo a questo utente. Nel file di accounting è memorizzato un puntatore (oppure il nome) della directory iniziale dell’utente. Tale puntatore viene copiato in una variabile locale per l’utente che specifica la sua directory corrente iniziale. Dalla shell dell’utente si possono poi avviare altri processi: la loro directory corrente è solitamente la directory corrente del processo genitore al momento della creazione del figlio.

I nomi di percorso possono essere di due tipi: **nomi di percorso assoluti** e **nomi di percorso relativi**. Un *nome di percorso assoluto* comincia dalla radice dell’albero di directory e segue un percorso che lo porta fino al file specificato indicando i nomi delle directory lungo il percorso. Un *nome di percorso relativo* definisce un percorso che parte dalla directory corrente.

Per esempio, nel file system ad albero della Figura 11.11, se la directory corrente è `root/spell/mail`, il nome di percorso relativo `prt/first` si riferisce allo stesso file indicato dal percorso assoluto `root/spell/mail/prt/first`.

Se si permette all’utente di definire le proprie sottodirectory, gli si consente anche di dare una struttura ai suoi file. Questa struttura può presentare directory distinte per file associati a soggetti diversi; per esempio, si può creare una sottodirectory contenente il testo di questo libro, oppure diversi tipi di informazioni, per esempio la directory `programmi` può contenere programmi sorgente; la directory `bin` può contenere tutti i programmi eseguibili.

Una decisione importante relativa alla strutturazione ad albero delle directory riguarda il modo di gestire la cancellazione di una directory. Se una directory è vuota, è sufficiente cancellare l’elemento che la designa nella directory che la contiene. Tuttavia se la directory da cancellare non è vuota, ma contiene file oppure sottodirectory, è possibile procedere in due modi. Alcuni sistemi non cancellano una directory a meno che non sia vuota; per cancellarla l’utente deve prima cancellare i file in essa contenuti. Se esiste qualche sottodirectory, questa procedura si deve applicare anche alle sottodirectory. Questo metodo può richiedere una discreta quantità di lavoro. In alternativa, come nel comando `rm` di UNIX, si può avere un’opzione che, alla richiesta di cancellazione di una directory, cancelli anche tutti i file e tutte le sottodirectory in essa contenuti. Entrambi i criteri sono abbastanza facili da realizzare; si tratta soltanto

di stabilire la politica da seguire. Il secondo criterio è più comodo, anche se più pericoloso, poiché si può rimuovere un'intera struttura della directory con un solo comando. Se si eseguisse tale comando per sbaglio sarebbe necessario ripristinare un gran numero di file e directory dalle copie di riserva (ipotizzandone l'esistenza).

Con un sistema di directory strutturato ad albero anche l'accesso ai file di altri utenti è di facile realizzazione. Per esempio, l'utente *B* può accedere ai file dell'utente *A* specificando i nomi di percorso assoluti oppure relativi. In alternativa, l'utente *B* può far sì che la propria directory corrente sia quella dell'utente *A* e accedere ai file usando direttamente i loro nomi.

11.3.6 Directory con struttura a grafo aciclico

Si considerino due programmatore che lavorano a un progetto comune. I file associati a quel progetto si possono memorizzare in una sottodirectory, separandoli da altri progetti e file dei due programmatore, ma poiché entrambi i programmatore hanno le stesse responsabilità sul progetto, ciascuno vuole che la sottodirectory si trovi nelle proprie directory. In questa situazione la sottodirectory comune deve essere *condivisa*. Nel sistema esiste quindi una directory o un file condivisi, in due, o più, posizioni contemporaneamente.

La struttura ad albero non ammette la condivisione di file o directory. Un **grafo aciclico** (cioè senza cicli) permette alle directory di avere sottodirectory e file condivisi (Figura 11.12). Lo stesso file o la stessa sottodirectory possono essere in due directory diverse. Un grafo aciclico rappresenta la generalizzazione naturale dello schema delle directory con struttura ad albero.

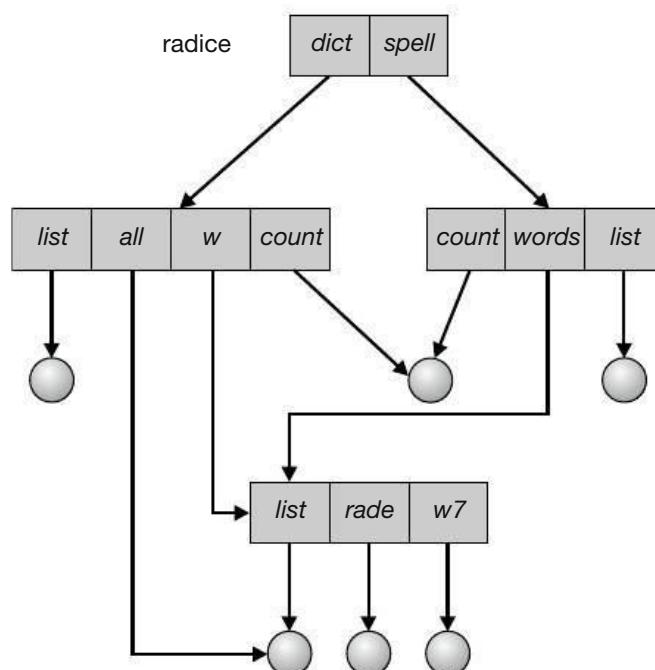


Figura 11.12 Struttura della directory a grafo aciclico.

Il fatto che un file sia condiviso, o che lo sia una directory, è diverso dall'avere due copie del file: con due copie ciascun programmatore potrebbe vedere la copia presente nella propria directory e non l'originale; se un programmatore modifica il file, le modifiche non appaiono nell'altra copia. Se invece il file è condiviso esiste *un* solo file effettivo, perciò tutte le modifiche sono immediatamente visibili. La condivisione è di particolare importanza se applicata alle sottodirectory; un nuovo file creato da un utente appare automaticamente in tutte le sottodirectory condivise.

Quando più persone lavorano insieme, tutti i file da condividere si possono inserire in una directory comune. La directory utente di ogni membro del gruppo contiene questa directory di file condivisi in forma di sottodirectory. Anche nel caso di un singolo utente, l'organizzazione dei file di tale utente può richiedere che alcuni file siano inseriti in più sottodirectory. Per esempio, un programma scritto per un progetto particolare deve trovarsi sia nella directory di tutti i programmi sia nella directory di quel progetto.

I file e le sottodirectory condivisi si possono realizzare in molti modi. Un metodo diffuso, esemplificato da molti tra i sistemi UNIX, prevede la creazione di un nuovo elemento di directory, chiamato collegamento. Un **collegamento** (*link*) è un puntatore a un altro file o un'altra directory. Per esempio, un collegamento si può realizzare come un nome di percorso assoluto o relativo. Quando si fa riferimento a un file, si compie una ricerca nella directory. Se l'elemento cercato è contrassegnato come collegamento, riporta il nome di percorso del file reale. Quindi si *risolve* il collegamento usando il nome di percorso per localizzare il file reale. I collegamenti si identificano facilmente tramite il loro formato nell'elemento della directory (o, nei sistemi che gestiscono i tipi, dal tipo speciale), e sono in pratica puntatori indiretti. Durante l'attraversamento degli alberi delle directory il sistema operativo ignora questi collegamenti, preservando così la struttura aciclica.

Un altro comune metodo per la realizzazione dei file condivisi prevede semplicemente la duplicazione di tutte le informazioni relative ai file in entrambe le directory che lo condividono: i due elementi delle directory sono identici. Si consideri la differenza fra i due approcci. Un collegamento è chiaramente diverso dall'elemento originale della directory. Duplicando gli elementi della directory, invece, la copia e l'originale sono resi indistinguibili: sorge allora il problema di mantenere la coerenza se il file viene modificato.

Una struttura di directory a grafo aciclico è più flessibile di una semplice struttura ad albero, ma anche più complessa. Si devono prendere in considerazione parecchi problemi. Un file può avere più nomi di percorso assoluti, quindi nomi diversi possono riferirsi allo stesso file. Questa situazione è simile al problema degli *alias* nei linguaggi di programmazione. Quando si percorre tutto il file system – per trovare un file, per raccogliere dati statistici su tutti i file o per fare le copie di backup dei file – il problema diviene importante poiché le strutture condivise non si devono attraversare più di una volta.

Un altro problema riguarda la cancellazione, poiché è necessario stabilire in quali casi è possibile allocare e riutilizzare lo spazio allocato a un file condiviso. Una pos-

sibilità prevede che a ogni operazione di cancellazione seguа l'immediata rimozione del file; quest'azione puо perа lasciare puntatori sospesi (*dangling*) a un file che ormai non esiste pi. Problema ancora pi grave, se i puntatori contengono indirizzi effettivi del disco e lo spazio viene poi riutilizzato per altri file, i puntatori potrebbero puntare nel mezzo di questi altri file.

In un sistema dove la condivisione е realizzata da collegamenti simbolici la gestione di questa situazione е relativamente semplice. La cancellazione di un collegamento non influisce sul file originale, poich si rimuove solo il collegamento. Se si cancella il file, si libera lo spazio corrispondente lasciando in sospeso il collegamento;  possibile ricercare tutti questi collegamenti e rimuoverli, ma se in ogni file non esiste una lista dei collegamenti associati al file stesso questa ricerca puо essere abbastanza onerosa. In alternativa, si possono lasciare i collegamenti finch non si tenta di usarli, quindi si scopre che il file con il nome dato dal collegamento non esiste e non si riesce a risolvere il collegamento; l'accesso е trattato proprio come qualsiasi altro nome di file errato. In questo caso, il progettista del sistema deve decidere attentamente cosa si debba fare quando si cancella un file e si crea un altro file con lo stesso nome, prima che sia stato usato un collegamento simbolico al file originario. In UNIX, quando si cancella un file, i collegamenti simbolici restano,  l'utente che deve rendersi conto che il file originale  scomparso o  stato sostituito. Nella famiglia di sistemi operativi Microsoft Windows si segue lo stesso criterio.

Un altro tipo di approccio alla cancellazione prevede la conservazione del file finch non siano stati cancellati tutti i riferimenti a esso. In questo caso  necessario disporre di un meccanismo che permetta di determinare che l'ultimo riferimento a quel file  stato cancellato;  possibile tenere una lista di tutti i riferimenti a un file (elementi di directory o collegamenti simbolici). Quando si crea un collegamento, oppure una copia dell'elemento della directory, si aggiunge un nuovo elemento alla lista dei riferimenti al file; quando si cancella un collegamento oppure un elemento della directory, si elimina dalla lista l'elemento corrispondente. Quando la sua lista di riferimenti  vuota, il file viene cancellato.

Questo metodo presenta, perа, un problema: la dimensione della lista dei riferimenti al file puо essere variabile e potenzialmente grande. Tuttavia, non  realmente necessario mantenere l'intera lista,  sufficiente un contatore del numero di riferimenti. Un nuovo collegamento o un nuovo elemento della directory incrementa il numero dei riferimenti; la cancellazione di un collegamento o di un elemento decrementa questo numero. Quando il contatore  uguale a 0 si puо cancellare il file, poich non ci sono pi riferimenti a tale file. Il sistema operativo UNIX usa questo metodo per i collegamenti non simbolici, o **collegamenti effettivi** (*hard link*); il contatore dei riferimenti  tenuto nel blocco di controllo del file o *inode*. Impedendo che si facciano pi riferimenti a una directory, si puо mantenere una struttura a grafo aciclico.

Per evitare questi problemi alcuni sistemi semplicemente non consentono la condivisione delle directory né i collegamenti.

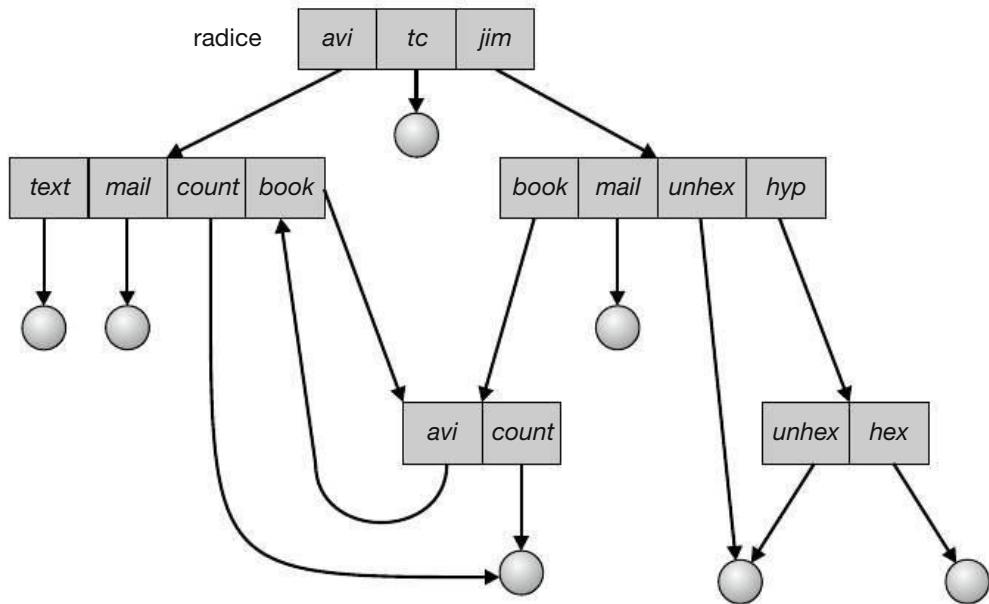


Figura 11.13 Directory a grafo generale.

11.3.7 Directory con struttura a grafo generale

Un serio problema connesso all’uso di una struttura a grafo aciclico consiste nell’assicurare che non vi siano cicli. Iniziando con una directory a due livelli e permettendo agli utenti di creare sottodirectory si crea una directory con struttura ad albero. È facile capire che aggiungendo nuovi file e nuove sottodirectory alla directory con struttura ad albero, la natura di quest’ultima persiste. Tuttavia, quando si aggiungono dei collegamenti a una directory con struttura ad albero, tale struttura si trasforma in una semplice struttura a grafo, come quella illustrata nella Figura 11.13.

Il vantaggio principale di un grafo aciclico è dato dalla semplicità degli algoritmi necessari per attraversarlo e per determinare quando non ci siano più riferimenti a un file. È preferibile evitare un duplice attraversamento di sezioni condivise di un grafo aciclico, soprattutto per motivi di prestazioni. Se un file particolare è stato appena cercato in una grande sottodirectory condivisa, ma non è stato trovato, è preferibile evitare una seconda ricerca nella stessa sottodirectory, che costituirebbe solo una perdita di tempo.

Se si permette che nella directory esistano cicli, è preferibile evitare una duplice ricerca di un elemento, per motivi di correttezza e di prestazioni. Un algoritmo mal progettato potrebbe causare un ciclo infinito di ricerca. Una soluzione è quella di limitare arbitrariamente il numero di directory cui accedere durante una ricerca.

Un problema analogo si presenta al momento di stabilire quando sia possibile cancellare un file. Come con le strutture delle directory a grafo aciclico, la presenza di uno 0 nel contatore dei riferimenti significa che non esistono più riferimenti al file o alla directory, e quindi il file può essere cancellato. Tuttavia, se esistono cicli, è possibile che il contatore dei riferimenti possa essere non nullo, anche se non è più possibile

far riferimento a una directory o a un file. Questa anomalia è dovuta alla possibilità di autoriferimento (ciclo) nella struttura delle directory. In questo caso è generalmente necessario usare un metodo di “ripulitura” (**garbage collection**) per stabilire quando sia stato cancellato l’ultimo riferimento e quando sia possibile riallocare lo spazio dei dischi. Tale metodo implica l’attraversamento del file system, durante il quale si contrassegna tutto ciò che è accessibile; in un secondo passaggio si raccoglie in un elenco di blocchi liberi tutto ciò che non è contrassegnato. Una procedura di marcatura analoga è utilizzabile per assicurare che un attraversamento o una ricerca coprano tutto quel che si trova nel file system una e una sola volta. La garbage collection di un file system basato su dischi richiede però molto tempo, perciò viene tentata solo di rado.

Inoltre, poiché è necessaria solo a causa della presenza dei cicli, è molto più conveniente lavorare con una struttura a grafo aciclico. La difficoltà consiste nell’evitare i cicli quando si aggiungono nuovi collegamenti alla struttura. Per sapere quando un nuovo collegamento ha completato un ciclo si possono impiegare gli algoritmi che permettono di individuare la presenza di cicli nei grafi. Dal punto di vista del calcolo, però, questi algoritmi sono onerosi, soprattutto quando il grafo si trova in memoria secondaria. Nel caso particolare di directory e collegamenti, un semplice algoritmo prevede di non percorrere i collegamenti durante l’attraversamento delle directory: si evitano così i cicli senza alcun carico ulteriore.

11.4 Montaggio di un file system

Così come si deve *aprire* un file per poterlo usare, per essere reso accessibile ai processi di un sistema, un file system deve essere *montato*. In particolare deve essere costruita la struttura della directory, composta di volumi, che devono essere montati affinché siano disponibili nello spazio dei nomi del file system.

La procedura di montaggio è molto semplice: si fornisce al sistema operativo il nome del dispositivo e la sua locazione (detta **punto di montaggio**) nella struttura di file e directory alla quale agganciare il file system. Alcuni sistemi operativi richiedono che venga specificato il tipo di file system, mentre altri ispezionano le strutture del dispositivo e determinano il tipo del file system presente. Di solito, un punto di montaggio è una directory vuota. Per esempio, in un sistema UNIX, un file system contenente le directory iniziali degli utenti si potrebbe montare come `/home`; quindi per avere accesso alla struttura della directory all’interno di quel file system, si premette `/home` ai nomi della directory (ad esempio `/home/jane`). Se lo stesso file system si montasse come `/users` il percorso per quella stessa directory sarebbe `/users/jane`.

Il passo successivo consiste nella verifica da parte del sistema operativo della validità del file system contenuto nel dispositivo. La verifica si compie chiedendo al driver del dispositivo di leggere la directory di dispositivo e controllando che tale directory abbia il formato previsto. Infine, il sistema operativo annota nella sua struttura della directory che un certo file system è montato al punto di montaggio specificato. Questo schema permette al sistema operativo di attraversare la sua struttura della directory, passando da un file system all’altro, anche di tipi diversi, secondo le necessità.

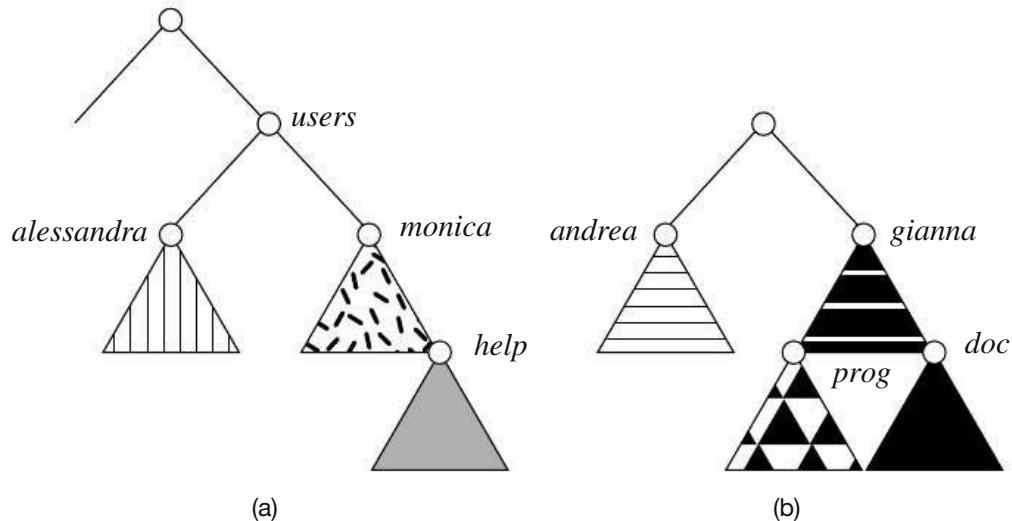


Figura 11.14 File system; (a) sistema esistente; (b) volume non montato.

Per illustrare l'operazione di montaggio, si consideri il file system rappresentato nella Figura 11.14, in cui i triangoli rappresentano i sottoalberi di directory di interesse. La Figura 11.14(a), mostra un file system esistente, mentre nella Figura 10.14(b) è raffigurato un volume non ancora montato che risiede in /device/dsk. A questo punto, si può accedere solo ai file del file system esistente. Nella Figura 11.15 si possono vedere gli effetti dell'operazione di montaggio del volume residente in /device/dsk al punto di montaggio /users. Se si smonta il volume, il file system ritorna alla situazione rappresentata nella Figura 11.14.

Per precisare le funzionalità, i sistemi operativi impongono regole semantiche a queste operazioni. Per esempio, un sistema potrebbe vietare il montaggio in una directory contenente file, o rendere disponibile il file system montato in tale directory e nascondere i file preesistenti nella directory finché non si *smonti* il file system, con-

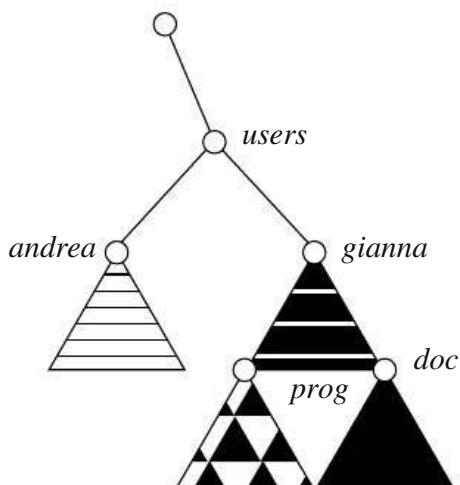


Figura 11.15 Punto di montaggio.

cludendone l'uso e permettendo l'accesso ai file originariamente presenti in tale directory. Come ulteriore esempio, un sistema potrebbe permettere il montaggio ripetuto dello stesso file system in diversi punti di montaggio, o potrebbe imporre una sola possibilità di montaggio per ciascun file system.

Si considerino le azioni del sistema operativo Mac OS X. Ogni volta che il sistema rileva per la prima volta un disco (sia nella fase d'avviamento che mentre il sistema è in esecuzione), il sistema operativo Mac OS X cerca un file system nel dispositivo. Se ne trova uno, esegue automaticamente il montaggio del filesystem nella directory `/Volumes`, aggiungendo un'icona di cartella sul desktop, etichettata con il nome del file system (secondo quel che è memorizzato nella directory di dispositivo). A questo punto l'utente può selezionare l'icona con il mouse e quindi vedere il contenuto del nuovo file system appena montato.

La famiglia di sistemi operativi Microsoft Windows mantiene una struttura della directory a due livelli estesa, con una lettera di unità associata a dispositivi e volumi. I volumi hanno una struttura della directory a grafo generale associata a una lettera di unità. Il percorso completo per uno specifico file è quindi della forma `lettera_di_unità:\percorso\file`. Le versioni più recenti di Windows permettono di montare un file system su qualunque punto dell'albero delle directory, esattamente come UNIX. I sistemi Windows rilevano automaticamente tutti i dispositivi e montano tutti i file system rilevati all'avvio della macchina. In altri sistemi, per esempio UNIX, i comandi di montaggio sono esplicativi. Un file di configurazione del sistema contiene una lista di dispositivi (e relativi punti di montaggio) da montare automaticamente all'avvio, ma altri montaggi possono essere eseguiti manualmente.

Le questioni riguardanti il montaggio di file system sono approfondite nel Paragrafo 12.2.2.

11.5 Condivisione di file

Nei paragrafi precedenti sono state presentate le motivazioni alla base della necessità di condivisione dei file, oltre ad alcune difficoltà che si incontrano nel permettere che diversi utenti possano condividere file. Tale possibilità è particolarmente utile per utenti che vogliono collaborare per ridurre le risorse richieste per raggiungere un certo obiettivo. Quindi, nonostante le difficoltà inerenti alla condivisione, i sistemi operativi orientati agli utenti devono soddisfare questa esigenza.

In questo paragrafo si considerano diversi aspetti della condivisione dei file, partendo dalle problematiche generali che nascono quando più utenti condividono dei file. Una volta che più utenti possono condividere file, l'obiettivo diventa estendere la condivisione a più file system, compresi i file system remoti. Infine, ci possono essere diverse interpretazioni delle azioni conflittuali intraprese su file condivisi. Per esempio, se più utenti stanno scrivendo nello stesso file, ci si può chiedere se il sistema dovrebbe permettere tutte le operazioni di scrittura, oppure dovrebbe proteggere le azioni di ciascun utente da quelle degli altri.

11.5.1 Utenti multipli

Se un sistema operativo permette l’uso del sistema da parte di più utenti, diventano particolarmente rilevanti i problemi relativi alla condivisione dei file, alla loro identificazione tramite nomi e alla loro protezione. Data una struttura della directory che permette la condivisione di file da parte degli utenti, il sistema deve intermediare questa condivisione. Il sistema può permettere a ogni utente di accedere ai file degli altri utenti per default, oppure richiedere che un utente debba esplicitamente concedere i permessi di accesso ai file. Questi aspetti sono alla base dei temi del controllo degli accessi e della protezione, trattati nel Paragrafo 11.6.

Per realizzare i meccanismi di condivisione e protezione, il sistema deve memorizzare e gestire più attributi per le directory e i file rispetto a un sistema che consente un singolo utente. Sebbene storicamente siano stati proposti molti metodi per la realizzazione di questi meccanismi, la maggior parte dei sistemi è arrivata ad adottare – per ciascun file o directory – i concetti di *proprietario* (owner) o *utente* e *gruppo*. Il proprietario è l’utente che può cambiare gli attributi, concedere l’accesso e che, in generale, ha il maggior controllo sul file. L’attributo di gruppo di un file si usa per definire il sottoinsieme di utenti autorizzati a condividere il file. Per esempio, il proprietario di un file in un sistema UNIX può fare qualsiasi operazione sul file, mentre i membri del gruppo possono compiere un sottoinsieme di queste operazioni e il resto degli utenti un altro sottoinsieme. Il proprietario del file può definire l’esatto insieme di operazioni che i membri del gruppo e gli altri utenti possono eseguire. Maggiori dettagli sugli attributi che regolano i permessi sono trattati nel paragrafo successivo.

Gli identificatori (ID) del gruppo e del proprietario di un certo file o directory sono memorizzati insieme con gli altri attributi del file. Quando un utente richiede di compiere un’operazione su un file, per verificare se l’utente richiedente è il proprietario del file si può confrontare l’ID utente con l’attributo che identifica il proprietario. Analogamente si confrontano gli ID di gruppo. Ne risultano i permessi applicabili, che il sistema considera per consentire o impedire l’operazione richiesta.

Molti sistemi operativi hanno più file system locali, inclusi volumi di un unico disco, o più volumi in diversi dischi connessi al sistema. In questi casi, la verifica degli ID e il confronto dei permessi si possono fare facilmente dopo l’operazione di montaggio dei file system.

11.5.2 File system remoti

L’avvento delle reti (Capitolo 17) ha permesso la comunicazione tra calcolatori remoti. Le reti permettono la condivisione di risorse sparse nell’area di un campus o addirittura in diversi luoghi del mondo. Un’ovvia risorsa da condividere sono i dati, nella forma di file.

I metodi con i quali i file si condividono in una rete sono cambiati molto, seguendo l’evoluzione della tecnologia delle reti e dei file. Il primo metodo utilizzato consiste nel trasferimento dei file richiesto in modo esplicito dagli utenti, attraverso programmi come l’`ftp`. Un secondo metodo è quello del **file system distribuito** (*distributed*

file system, DFS), che permette la visibilità nel calcolatore locale delle directory remote. Il terzo metodo, il **World Wide Web**, è, da un certo punto di vista, un ritorno al primo. Per accedere ai file remoti si usa un browser, e operazioni distinte – essenzialmente un involucro (*wrapper*) per l'`ftp` – per trasferirli. Un altro metodo, sempre più diffuso, per la condivisione di file è il **cloud computing** (Paragrafo 1.11.7).

L'`ftp` si usa sia per l'accesso anonimo sia per quello autenticato.

L'**accesso anonimo** permette di trasferire file senza avere un account nel sistema remoto. Il World Wide Web usa quasi esclusivamente lo scambio di file anonimo. Un DFS comporta un'integrazione molto più stretta tra il calcolatore che accede ai file remoti e il calcolatore che fornisce i file. Tale integrazione incrementa la complessità, com'è illustrato nel prossimo paragrafo.

11.5.2.1 Modello client-server

I file system remoti permettono il montaggio di uno o più file system di uno o più calcolatori remoti in un calcolatore locale. In questo caso, il calcolatore contenente i file si chiama **server**, mentre il calcolatore che richiede l'accesso ai file si chiama **client**. La relazione tra client e server è piuttosto comune tra i calcolatori di una rete. In generale, il server dichiara che determinate risorse sono disponibili ai client, specificando esattamente quali (in questo caso, quali file) ed esattamente a quali client. Un server può gestire richieste provenienti da più client, e un client può accedere a più server, secondo l'implementazione del particolare sistema client-server.

Il server in genere specifica i file disponibili su di un volume o livello di directory. L'identificazione dei client è più difficile; un client può essere identificato tramite i relativi nomi simbolici di rete, o tramite altri identificatori come un indirizzo IP, ma questi possono essere facilmente imitati (*spoofing*). Di conseguenza un client non autorizzato può accedere a un server. Tra le soluzioni più sicure ci sono quelle che prevedono l'autenticazione dei client tramite chiavi di cifratura. Sfortunatamente, l'introduzione di tecniche per la sicurezza introduce nuovi problemi, per esempio la necessità della compatibilità tra client e server (si devono impiegare gli stessi algoritmi di cifratura) e dello scambio sicuro delle chiavi (l'intercettazione delle chiavi può permettere accessi non autorizzati). Questi problemi sono sufficientemente difficili da far sì che nella maggioranza dei casi si usino metodi di autenticazione insicuri.

Nel caso di UNIX e del suo file system di rete (*network file system, NFS*), l'autenticazione avviene, per default, tramite le informazioni di rete relative al client. In questo schema, gli identificatori (ID) dell'utente devono coincidere nel client e nel server; diversamente, il server non può determinare i diritti d'accesso ai file. Si consideri per esempio un utente con un identificatore uguale a 1000 nel client e a 2000 nel server. Una richiesta per uno specifico file dal client al server non potrà essere gestita correttamente, perché il server cercherà di determinare se l'utente 1000 ha i permessi d'accesso al file, invece di usare il reale ID dell'utente che è 2000. L'accesso sarà concesso o negato secondo un'informazione di autenticazione sbagliata. Il server deve fidarsi del client e assumere che quest'ultimo gli presenti l'identificatore corretto.

Si noti che il protocollo NFS permette relazioni da molti a molti; cioè più server possono fornire file a più client. Infatti, un calcolatore può comportarsi sia da server per altri client NFS, sia da client di altri server NFS.

Una volta montato il file system remoto, le richieste delle operazioni su file sono inviate al server, attraverso la rete, per conto dell’utente, usando il protocollo DFS. Normalmente, una richiesta di apertura di file si invia insieme con l’ID dell’utente richiedente. Il server quindi applica i normali controlli d’accesso per determinare se l’utente ha le credenziali per accedere al file nel modo richiesto; se tali controlli hanno esito positivo, riporta un handle d’accesso al file all’applicazione client, che la usa per eseguire sul file operazioni di lettura, scrittura e altro. Il client chiude il file quando non deve più accedervi. Il sistema operativo può applicare una semantica simile a quella adottata per il montaggio di un file system locale, oppure una semantica diversa.

11.5.2.2 Sistemi di informazione distribuiti

Per semplificare la gestione dei servizi client-server, i **sistemi di informazione distribuiti**, noti anche come **servizi di naming distribuiti**, sono stati concepiti per fornire un accesso unificato alle informazioni necessarie per il calcolo remoto. Il **sistema dei nomi di dominio** (*domain name system, DNS*) fornisce le traduzioni dai nomi dei calcolatori agli indirizzi di rete per l’intera Internet. Prima che il DNS si diffondesse capillarmente nella rete, si scambiavano tra i calcolatori, per posta elettronica o *f tp*, file contenenti le stesse informazioni. Questo metodo, ovviamente, non poteva adattarsi dinamicamente all’aumento delle dimensioni della rete Internet. Il DNS è trattato ulteriormente nel Paragrafo 17.4.1.

Altri sistemi di informazione distribuiti forniscono uno spazio identificato da *nome utente/parola d’ordine/identificatore utente/identificatore di gruppo* per un servizio distribuito. I sistemi UNIX hanno adottato un’ampia varietà di metodi per l’informazione distribuita. Sun Microsystems (che ora è parte della Oracle Corporation) ha introdotto il sistema *yellow pages* (poi ribattezzato *network information service, NIS*), adottato da gran parte dell’industria. Questo servizio centralizza la memorizzazione dei nomi degli utenti e dei calcolatori, delle informazioni sulle stampanti e altro. Sfortunatamente, usa metodi di autenticazione insicuri, per esempio l’invio di parole d’ordine dell’utente non cifrate (*in chiaro*) e l’identificazione dei calcolatori attraverso gli indirizzi IP. Il sistema NIS+ di Sun era una versione del NIS più sicura, ma anche molto più complessa e non ha avuto una gran diffusione.

Nel caso del *common internet file system (CIFS)* di Microsoft, le informazioni di rete si usano insieme con gli elementi di autenticazione dell’utente (nome dell’utente e parola d’ordine) per creare un **login di rete** che il server usa per decidere se permettere o negare l’accesso a un file system richiesto. Affinché questa autenticazione sia valida, i nomi utente devono essere uguali nelle varie macchine (come per l’NFS). Per fornire un unico spazio di nomi per gli utenti, Microsoft usa l’**active directory**. Una volta impostata, la funzione di naming è usata per autenticare gli utenti da tutti i client e da tutti i server.

L’industria si sta orientando verso il protocollo **LDAP** (*lightweight directory-access protocol*) come meccanismo sicuro per il naming distribuito. Lo stesso *active direc-*

tory è basato sull'LDAP. Oracle Solaris e altri importanti sistemi operativi includono LDAP e permettono l'uso di questo protocollo per l'autenticazione degli utenti e per altri servizi di ricerca di informazioni a livello dell'intero sistema, per esempio la disponibilità delle stampanti. È pensabile che un'organizzazione possa usare una singola directory LDAP distribuita per memorizzare le informazioni su tutti gli utenti e le risorse di tutti i calcolatori dell'organizzazione stessa. Si avrebbe un **unico punto di autenticazione sicura** (*single sign-on*) per gli utenti, che inserirebbero una sola volta le proprie informazioni di autenticazione per avere accesso a tutti i calcolatori dell'organizzazione. Questa soluzione semplificherebbe anche i compiti degli amministratori di sistema, concentrando in un unico punto informazioni ora sparse in vari file in ciascun sistema o in diversi servizi di informazione distribuiti.

11.5.2.3 Tipi di malfunzionamento

I file system locali possono presentare malfunzionamenti per varie cause: problemi dei dischi che li contengono, alterazione dei dati relativi alle strutture delle directory o a informazioni necessarie alla gestione dei dischi (chiamate collettivamente **metadati**), malfunzionamenti dei controllori dei dischi, problemi ai cavi di connessione o agli adattatori. Anche un errore di un utente o dell'amministratore di sistema può causare la perdita di file, d'intero directory o addirittura la cancellazione di volumi. Molti di questi malfunzionamenti portano alla caduta del sistema (*crash*), all'emissione di una condizione d'errore e alla necessità di un intervento umano per risolvere il problema.

L'uso di file system remoti implica ulteriori tipi di malfunzionamento; a causa della complessità dei sistemi di rete e della necessità di interazioni tra calcolatori remoti, i problemi che possono interferire con il corretto funzionamento dei file system remoti sono infatti molto più numerosi. Nel caso delle reti, si possano verificare interruzioni del collegamento tra due calcolatori, dovute a guasti hardware o a improprie configurazioni dell'architettura, oppure a problemi di implementazione della rete. Sebbene alcune reti includano meccanismi di tolleranza ai guasti, compresi cammini multipli tra ogni coppia di calcolatori, altre non li prevedono. Qualsiasi malfunzionamento potrebbe interrompere il flusso dei comandi del DFS.

Si consideri un client mentre usa un file system remoto. Il client ha qualche file remoto aperto; tra le varie attività potrebbe percorrere le directory remote per aprire aprire file, svolgere operazioni di lettura e scrittura e chiudere i file. Si consideri ora un malfunzionamento della rete, una caduta del server remoto, oppure anche uno spegnimento programmato di quel server: improvvisamente il file system remoto è inaccessibile. Questo scenario è piuttosto comune, quindi il client non dovrebbe comportarsi come nel caso di una perdita del file system locale. Piuttosto, il sistema dovrebbe terminare tutte le operazioni sul server non più raggiungibile, oppure posticiparle finché il server sarà nuovamente disponibile. Questa semantica di trattamento dei malfunzionamenti si definisce e si realizza come parte del protocollo di file system remoto. La terminazione di tutte le operazioni può portare alla perdita di dati (e della pazienza) da parte degli utenti. Quindi la maggior parte dei protocoli DFS impone o

permette la posticipazione delle operazioni sul file system remoto, con la speranza che il calcolatore remoto diventi nuovamente disponibile. Per realizzare questo tipo di recupero dai malfunzionamenti è necessario mantenere alcune **informazioni di stato** sia sui client sia sui server. Se sia i client sia i server tengono traccia delle loro attività correnti e dei loro file aperti, entrambi possono riprendersi senza problemi da un malfunzionamento. Nel caso in cui il server “cada” ma debba tracciare il montaggio remoto di file system e i file aperti, l’NFS segue un criterio semplice realizzando un DFS **senza stato**. Sostanzialmente, assume che una richiesta di un client per la lettura o scrittura di un file non sarebbe avvenuta, a meno che il file system non sia stato montato in modo remoto e il file in questione aperto prima della richiesta. Il protocollo NFS trasporta tutte le informazioni necessarie per localizzare il file appropriato e per svolgere l’operazione richiesta sul file. Allo stesso modo, non tiene traccia di quali client abbiano montato i propri volumi esportati, assumendo anche in questo caso che, se perviene una richiesta, debba essere legittima. Sebbene questo metodo senza stato renda l’NFS tollerante ai guasti e piuttosto facile da realizzare, lo rende insicuro. Ad esempio un server NFS potrebbe permettere richieste contraffatte di lettura o scrittura. Questioni del genere sono regolamentate dallo standard industriale NFS versione 4, in cui NFS è dotato di stato per migliorare le proprie funzionalità, prestazioni e sicurezza.

11.5.3 Semantica della coerenza

La **semantica della coerenza** è un importante criterio per la valutazione di qualsiasi file system che consenta la condivisione dei file. Questa semantica specifica il modo in cui più utenti devono accedere contemporaneamente a un file condiviso. In particolare, questa semantica deve specificare quando le modifiche ai dati apportate da un utente possano essere osservate da altri utenti. La semantica è tipicamente realizzata come parte del codice del file system.

La semantica della coerenza è direttamente correlata agli algoritmi di sincronizzazione dei processi del Capitolo 5. Tuttavia, a causa delle lunghe latenze e delle basse velocità di trasferimento dei dischi e delle reti, i complessi algoritmi descritti in tale capitolo di solito non s’impiegano per l’I/O su file. Per esempio, l’esecuzione di una transazione atomica su dischi remoti può coinvolgere molte comunicazioni di rete e molte letture e scritture nei dischi. I sistemi che tentano una così completa serie di funzioni tendono ad avere scarse prestazioni. Una realizzazione riuscita di una complessa semantica della condivisione si trova nel file system Andrew.

Nella trattazione seguente si suppone che una serie d’accessi, cioè letture e scritture, tentati da un utente allo stesso file sia sempre compresa tra una coppia di operazioni `open()` e `close()`. Tale serie d’accessi si chiama **sessione di file**. Per illustrare questo concetto si descrivono sommariamente qui di seguito alcuni importanti esempi di semantica della coerenza.

11.5.3.1 Semantica UNIX

Il file system di UNIX, descritto nel Capitolo 17, usa la seguente semantica della coerenza.

- Le scritture in un file aperto da parte di un utente sono immediatamente visibili ad altri utenti che hanno lo stesso file contemporaneamente aperto.
- Un metodo di condivisione permette agli utenti di condividere il puntatore alla locazione corrente nel file. Quindi l'avanzamento del puntatore da parte di un utente influenza su tutti gli utenti che condividono il file. In questo caso il file ha una singola immagine e tutti gli accessi si alternano (intercalandosi) a prescindere dalla loro origine.

Nella semantica UNIX un file è associato a una singola immagine fisica, accessibile come una risorsa esclusiva. La contesa su quest'immagine singola determina ritardi nei processi utente.

11.5.3.2 Semantica delle sessioni

Il file system Andrew (OpenAFS) usa la seguente semantica della coerenza:

- le scritture in un file aperto da un utente non sono visibili immediatamente ad altri utenti che hanno lo stesso file contemporaneamente aperto;
- una volta chiuso il file, le modifiche apportate sono visibili solo nelle sessioni che iniziano successivamente. Le istanze del file già aperte non riportano queste modifiche.

Secondo questa semantica, un file può essere contemporaneamente associato a parecchie immagini, probabilmente diverse. Di conseguenza, più utenti possono eseguire accessi concorrenti di lettura o scrittura sulla rispettiva immagine del file senza subire ritardi. Non si impone quasi alcun vincolo nella gestione degli accessi.

11.5.3.3 Semantica dei file condivisi immutabili

Un approccio particolare è quello dei **file condivisi immutabili**; una volta che un file è stato dichiarato *condiviso* dal suo creatore, non può essere modificato. Un file immutabile presenta due caratteristiche chiave: il suo nome non si può riutilizzare e il suo contenuto non può essere modificato. Quindi il nome di un file immutabile indica che i contenuti di quel file sono fissi. Come si descrive nel Capitolo 17, la realizzazione di questa semantica in un sistema distribuito è semplice; infatti la condivisione è molto disciplinata poiché consente la sola lettura.

11.6 Protezione

Le informazioni contenute in un sistema elaborativo devono essere protette dai danni fisici (la questione della *affidabilità*) e da accessi impropri (la questione della *protezione*).

Generalmente l'affidabilità è assicurata da più copie dei file. Molti calcolatori hanno programmi di sistema che copiano i file dai dischi ai nastri a intervalli regolari,

per esempio una volta al giorno, alla settimana o al mese; quest'operazione di copiatura può essere automatica, o controllata dall'intervento di un operatore. Lo scopo è quello di conservare copie di riserva utili nei casi in cui il file system andasse accidentalmente distrutto. I danni possono essere causati da problemi hardware (di lettura o scrittura), sovraccarichi o cadute della tensione elettrica, roture delle testine, sporcizia, temperature estreme, atti vandalici, ecc. I file possono inoltre essere cancellati accidentalmente, e anche errori di programmazione possono causare la perdita del contenuto dei file. L'affidabilità è stata trattata con maggiori dettagli nel Capitolo 10.

La protezione si può ottenere in molti modi. Per un computer portatile monoutente, la protezione si può ottenere chiudendo il computer in un cassetto della scrivania oppure in un armadietto. In un sistema multiutente più grande sono necessari altri metodi.

11.6.1 Tipi d'accesso

La necessità di proteggere i file deriva direttamente dalla possibilità di accedervi. I sistemi che non permettono l'accesso ai file di altri utenti non richiedono protezione; quindi si può ottenere una completa protezione proibendo l'accesso. In alternativa si può permettere un accesso totalmente libero senza alcuna protezione. Questi orientamenti sono entrambi eccessivi, quindi non si possono applicare in generale; ciò che serve in realtà è un **accesso controllato**.

Il controllo offerto dai meccanismi di protezione si ottiene limitando i possibili tipi d'accesso. Gli accessi si permettono o si negano secondo diversi fattori, innanzitutto i tipi d'accesso richiesti. Si possono controllare diversi tipi di operazione.

- **Lettura.** Lettura da file.
- **Scrittura.** Scrittura o riscrittura di file.
- **Esecuzione.** Caricamento di file in memoria ed esecuzione.
- **Aggiunta.** Scrittura di nuove informazioni in coda ai file.
- **Cancellazione.** Cancellazione di file e liberazione del relativo spazio per un possibile riutilizzo.
- **Elencazione.** Elencazione del nome e degli attributi dei file.

Si possono controllare anche altre operazioni, come ridenominazione, copiatura o modifica dei file. Tuttavia, in molti sistemi queste funzioni di livello superiore si possono realizzare tramite un programma di sistema che compie alcune chiamate di sistema di livello inferiore, quindi la protezione viene garantita a livello inferiore. Per esempio, la copiatura di un file si può realizzare semplicemente con una sequenza di richieste di lettura; in questo caso un utente con accesso per la lettura di un file può richiederne la copiatura, la stampa o altro.

Sono stati proposti molti meccanismi di protezione. Come sempre, ogni meccanismo presenta vantaggi e svantaggi, e deve essere appropriato alla sua particolare applicazione. Un piccolo calcolatore usato soltanto da pochi membri di un gruppo di

ricerca non richiede la stessa protezione del sistema elaborativo di una grande società, usato per operazioni di ricerca, finanza e per la gestione del personale. Discutiamo alcuni approcci alla protezione nel seguito e trattiamo il problema più completamente nel Capitolo 14.

11.6.2 Controllo degli accessi

L'approccio più comune al problema della protezione è rendere l'accesso dipendente dall'identità dell'utente. Utenti differenti possono richiedere diversi tipi d'accesso a un file o a una directory. Lo schema più generale per realizzare gli accessi dipendenti dall'identità consiste nell'associare una **lista di controllo degli accessi** (*access-control list, ACL*) a ogni file e directory; in tale lista sono specificati i nomi degli utenti e i relativi tipi d'accesso consentiti. Quando un utente richiede un accesso a un particolare file il sistema operativo esamina la lista di controllo degli accessi associata a quel file; se tale utente è presente nella lista per quel tipo di accesso, viene autorizzato, altrimenti si verifica una violazione della protezione e si nega l'accesso al file.

Questo sistema ha il vantaggio di permettere complessi metodi d'accesso. Il problema maggiore delle liste di controllo degli accessi è la loro lunghezza: per permettere a tutti di leggere un file, la lista deve contenere tutti gli utenti con accesso per la lettura. Questa tecnica comporta due inconvenienti:

- la costruzione di una lista di questo tipo può essere un compito noioso e non gratificante, soprattutto se la lista degli utenti del sistema non è nota a priori;
- l'elemento della directory, precedentemente di dimensione fissa, deve essere di dimensione variabile, quindi anche la gestione dello spazio è più complicata.

Questi problemi si possono risolvere introducendo una versione condensata della lista di controllo degli accessi.

Per condensarne la lunghezza, molti sistemi raggruppano gli utenti di ogni file in tre classi distinte.

- **Proprietario.** È l'utente che ha creato il file.
- **Gruppo.** Si tratta di un insieme di utenti che condividono il file e hanno bisogno di tipi di accesso simili.
- **Universo.** Tutti gli altri utenti del sistema.

Il più comune orientamento recente prevede la combinazione delle liste di controllo degli accessi con lo schema di controllo degli accessi per proprietario, gruppo e universo (più facile da realizzare). Il sistema operativo Solaris, per esempio, impiega le tre categorie d'accesso per default ma, se si vuole una maggiore selettività del controllo degli accessi, permette l'attribuzione di liste di controllo degli accessi a specifici file e directory.

Si consideri, per esempio, una persona, Donatella, che sta scrivendo un nuovo libro. Donatella ha assunto tre studenti, Giulia, Paolo e Carlo, per aiutarla a lavorare al progetto. Il testo del libro è mantenuto in un file chiamato `libro.tex`.

La protezione associata a tale file è la seguente:

- Donatella può compiere tutte le operazioni sul file;
- Giulia, Paolo e Carlo possono solo leggere e scrivere il file ma non possono cancellarlo;
- tutti gli altri utenti possono leggere, ma non scrivere, il file (Donatella ha interesse che il libro sia letto dal maggior numero possibile di persone, in modo da ottenere dei pareri).

Per ottenere tale protezione si deve creare un nuovo gruppo composto da Giulia, Paolo e Carlo, e si deve associare il nome del gruppo, per esempio `testo`, al file `libro.tex` con i diritti d'accesso conformi alla politica ora descritta.

Si consideri un ospite cui Donatella vorrebbe concedere un accesso temporaneo al Capitolo 1. L'ospite non si può aggregare al gruppo `testo` poiché ciò gli darebbe accesso a tutti i capitoli, e considerato che i file possono essere in un solo gruppo, non si può associare un altro gruppo al Capitolo 1. L'aggiunta della funzione delle liste di controllo degli accessi permette di inserire l'ospite nella lista di controllo degli accessi del Capitolo 1.

Affinché questo schema funzioni correttamente, è necessario uno stretto controllo dei permessi e delle liste di controllo degli accessi, fattibile in diversi modi. Nel sistema UNIX per esempio solo un utente con compiti di gestione (o un *superuser*) può creare e modificare i gruppi, quindi questo controllo si ottiene con la partecipazione umana. Le liste di controllo degli accessi sono trattate anche nel Paragrafo 14.5.2.

Per definire la protezione, data questa più limitata classificazione, occorrono solo tre campi. Normalmente ogni campo è formato di un insieme di bit, ciascuno dei quali permette o impedisce l'accesso che gli è associato. Nel sistema UNIX, per esempio, sono definiti tre campi di tre bit ciascuno: `rwx`, dove `r` controlla l'accesso per la lettura, `w` quello per la scrittura e `x` per l'esecuzione. Tre campi separati sono riservati al proprietario del file, al gruppo proprietario e a tutti gli altri utenti. In questo schema, per registrare le informazioni di protezione sono necessari nove bit per file. Così, nell'esempio, i campi di protezione per il file `libro.tex` si impostano come segue: per Donatella, il proprietario, tutti e tre i bit sono settati; per il gruppo `testo` i bit `r` e `w`; per l'universo il solo bit `r`.

Nel combinare i due metodi si presenta una difficoltà nell'interfaccia utente; gli utenti devono poter indicare l'impostazione opzionale dei permessi mediante ACL per un file. Nell'esempio di Solaris, si aggiunge un segno “+” ai permessi d'accesso standard:

```
19 -rw-r-r-+ 1 alessandra gruppo 130 May 25 22:13 file1
```

Una specifica serie di comandi, `setfacl` e `getfacl`, si usa per gestire le liste di controllo degli accessi.

Gli utenti di Windows, in genere, gestiscono le liste di controllo degli accessi tramite un'interfaccia grafica. La Figura 11.16 mostra la finestra dei permessi di accesso a un file del file system NTFS di Windows 7. In questo esempio si vieta esplicitamente all'utente Guest l'accesso al file `ListPanel.java`.

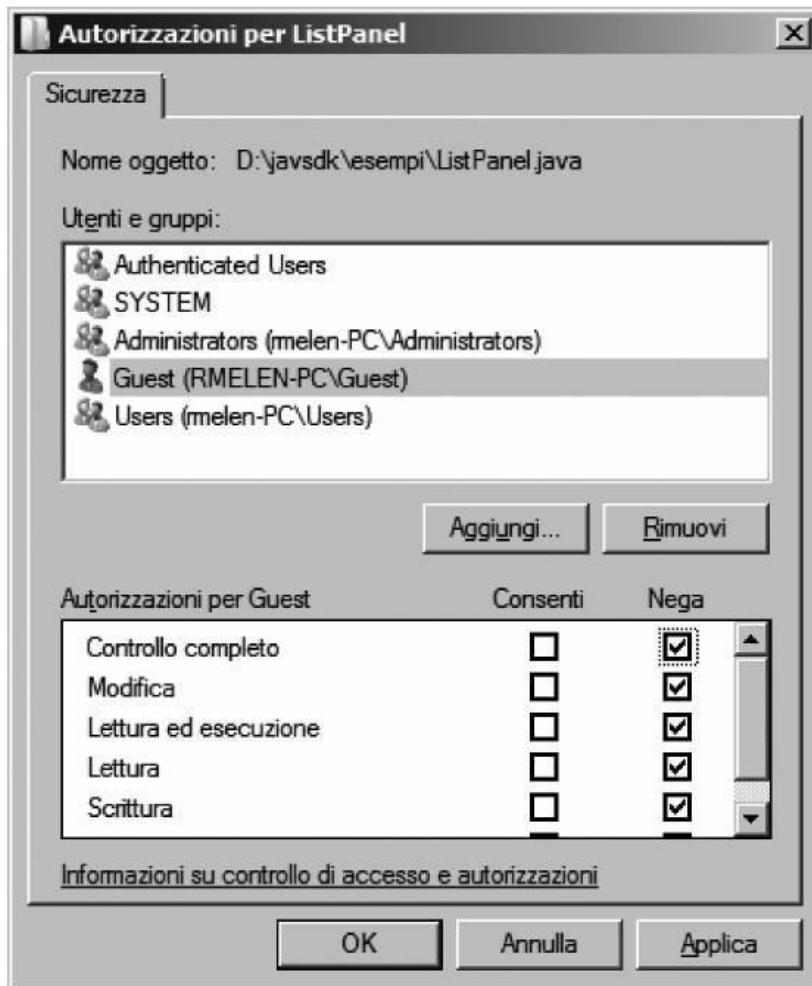


Figura 11.16 Gestione della lista di controllo degli accessi in Windows 7.

Un'altra difficoltà s'incontra nell'assegnazione delle precedenze quando ci sono conflitti tra i permessi e le ACL. Per esempio, se Andrea è in un gruppo di file, che ha il permesso di lettura, ma il file ha una lista di controllo degli accessi contenente i permessi di lettura e scrittura per Andrea, si pone il problema della concessione del permesso di scrittura. Nel sistema operativo Solaris hanno precedenza i permessi contenuti nelle liste di controllo degli accessi; sono più selettivi e non sono predefiniti. Si segue il principio generale che la maggiore specificità deve essere prioritaria.

11.6.3 Altri metodi di protezione

Un altro metodo di protezione consiste nell'associazione di una password a ciascun file. Proprio come l'accesso al sistema elaborativo è spesso controllato da una password, anche l'accesso a ogni file può avere lo stesso tipo di protezione. Se le password sono scelte a caso e si cambiano spesso, questo schema può essere efficace nel

○ PERMESSI IN UN SISTEMA UNIX

Nel sistema UNIX la protezione delle directory è gestita in modo simile alla protezione dei file. A ciascuna sottodirectory sono associati tre campi (proprietario, gruppo e universo), ciascuno composto dei tre bit rwx. Quindi, un utente può elencare il contenuto di una directory solamente se il bit r è inserito nel campo appropriato. Analogamente, un utente può cambiare la propria directory corrente in un'altra directory (per esempio foo) solo se il bit x associato alla directory foo, è settato nel campo appropriato.

Un esempio di elenco del contenuto di una directory nell'ambiente UNIX è illustrato qui di seguito:

```
-rw-rw-r--  1 pbg  staff      31200  set    3 08:30     intro.ps
drwx----- 5 pbg  staff       512   lug    8 09:33     privato/
drwxrwxr-x  2 pbg  staff      512   lug    8 09:35     doc/
drwxrwx---  2 pbg  studente   512   ago    3 14:13     studente-prog/
-rw-r--r--  1 pbg  staff      9423   feb   24 2003     program.c
-rwxr-xr-x  1 pbg  staff     20471   feb   24 2003     program
drwx--x--x  4 pbg  facoltà   512   lug   31 10:31     lib/
drwx----- 3 pbg  staff      1024   ago   29 06:52     mail/
drwxrwxrwx  3 pbg  staff      512   lug    8 09:35     test/
```

Il primo campo descrive le protezioni di file e directory, il carattere d presente all'inizio del campo contraddistingue le sottodirectory; inoltre, l'elenco contiene il numero di collegamenti relativi al file, il nome del proprietario e del gruppo, la dimensione del file in byte, la data dell'ultima modifica e infine il nome del file (con l'eventuale estensione).

limitare l'accesso ai file. Questo metodo presenta tuttavia diversi svantaggi: innanzitutto, il numero di parole d'ordine da ricordare può diventare molto alto, rendendo tale metodo impraticabile; secondariamente, se si impiega la stessa parola d'ordine per tutti i file, la sua scoperta li rende tutti accessibili. La protezione è basata sul principio del “o tutto o niente”. Per risolvere questo problema alcuni sistemi permettono a un utente di associare una parola d'ordine a una directory anziché a un singolo file.

In una struttura della directory a più livelli è necessario proteggere non solo i singoli file, ma anche gruppi di file contenuti in directory; quindi è necessario disporre di un meccanismo per la protezione delle directory. Le operazioni riguardanti le directory da proteggere sono piuttosto diverse dalle operazioni sui file. Esse sono la creazione e la cancellazione dei file in una directory; probabilmente va anche controllata la possibilità, per un utente, di determinare l'esistenza di un file in una directory. Talvolta la conoscenza dell'esistenza di un file e del suo nome può essere di per sé significativa, perciò l'elencazione del contenuto di una directory dev'essere un'operazione protetta. Se un nome di percorso fa riferimento a un file in una certa directory, all'utente deve essere consentito l'accesso sia al file sia alla directory. Nei sistemi dove i file possono avere numerosi nomi di percorso (come quelli con struttura a grafo aciclico o a grafo generale) un certo utente può avere diversi diritti d'accesso a un file a seconda del nome di percorso di cui fa uso.

11.7 Sommario

Un file è un tipo di dati astratto definito e realizzato dal sistema operativo. È una sequenza di elementi logici (o *record*), ciascuno dei quali può essere un byte, una riga di lunghezza fissa o variabile, oppure un elemento di dati più complesso. Il sistema operativo può gestire in modo specifico diversi tipi di elementi logici o può lasciare tale gestione al programma applicativo.

Il compito più importante del sistema operativo consiste nell'associare il concetto logico di file ai dispositivi fisici di memorizzazione, per esempio dischi o nastri magnetici. Poiché le dimensioni dei blocchi fisici dei dispositivi possono non coincidere con quelle dei record logici, può essere necessario riunire un certo numero di record logici per mapparli in un blocco fisico. Anche questo compito può essere gestito dal sistema operativo, oppure lasciato al programma applicativo.

Ciascun dispositivo di un file system conserva una tabella dei contenuti del volume o una directory di dispositivo che elenca la locazione dei file presenti nel dispositivo. Inoltre, è utile creare directory per permettere di organizzare i file. Una directory a un livello in un sistema multiutente causa problemi di naming, poiché ogni file deve avere un nome unico. Una directory a due livelli limita questo problema creando una directory distinta per i file di ciascun utente. Le directory contengono la lista dei file che le costituiscono, insieme con informazioni a essi associate: locazione nei dischi, lunghezza, tipo, proprietario, ora di creazione, ora dell'ultimo uso, e così via.

La naturale generalizzazione del concetto di directory a due livelli è la directory con struttura ad albero. Tale tipo di struttura permette a un utente di creare sottodirectory in cui organizzare i file. Le strutture delle directory a grafo aciclico permettono la condivisione di sottodirectory e file, ma complicano le funzioni di ricerca e cancellazione. Una struttura a grafo generale permette la massima flessibilità nella condivisione dei file e delle directory, ma talvolta richiede operazioni di “ripulitura” (*garbage collection*) per recuperare lo spazio inutilizzato nei dischi.

I dischi sono suddivisi in uno o più volumi, ciascuno contenente un file system o privo di struttura (*raw*). Per essere resi disponibili, i file system si possono montare nelle strutture di naming del sistema. Lo schema di naming varia da sistema a sistema. Una volta montati, i file all'interno del volume sono disponibili per l'uso. I file system si possono smontare per disabilitarne l'accesso o per attività di manutenzione.

La condivisione dei file dipende dalla semantica definita dal sistema. I file possono avere più lettori, più scrittori, o limiti alla condivisione. I file system distribuiti consentono ai sistemi client di montare volumi o directory da server, purchè siano accessibili tramite una rete. I file system remoti pongono delle sfide in termini di affidabilità, prestazioni e sicurezza. I sistemi di informazione distribuiti mantengono informazioni su utenti, calcolatori e accessi in modo che i client e i server condividano le informazioni di stato per la gestione dell'uso e degli accessi.

Poiché i file rappresentano il principale meccanismo di memorizzazione delle informazioni, è necessario che siano dotati di un sistema di protezione. Ogni tipo

d’accesso ai file si può controllare separatamente: lettura, scrittura, esecuzione, aggiunta, cancellazione, elencazione del contenuto di directory, e così via. La protezione dei file si può ottenere con password, liste d’accesso, o altre tecniche.

Esercizi di ripasso

- 11.1** Alcuni sistemi cancellano automaticamente tutti i file utente quando un utente si disconnette oppure quando un job termina, a meno che l’utente non richieda esplicitamente che questi vengano conservati; altri sistemi conservano tutti i file a meno che l’utente non li cancelli esplicitamente. Discutete i meriti di ciascun approccio.
- 11.2** Perché alcuni sistemi tengono traccia del tipo di un file, mentre altri lasciano questo compito all’utente e altri semplicemente non implementano tipi di file multipli? Quale sistema è “migliore”?
- 11.3** In modo simile, alcuni sistemi mettono a disposizione molte tipologie diverse di strutture per i dati di un file, mentre altri trattano solo un semplice flusso di byte. Quali sono i vantaggi e gli svantaggi di tali approcci?
- 11.4** È possibile simulare una struttura di directory multilivello con una struttura di directory a un livello nella quale possono essere usati nomi arbitrariamente lunghi? Se la risposta è affermativa, spiegate come ciò sia fattibile e confrontate questo schema con lo schema a directory multilivello. In caso di risposta negativa, spiegate che cosa impedisce il successo della simulazione. Come cambierebbe la vostra risposta se la lunghezza del nome dei file fosse limitata a sette caratteri?
- 11.5** Spiegate lo scopo delle operazioni `open()` e `close()`.
- 11.6** In alcuni sistemi, una sottodirectory può essere letta e scritta da un utente autorizzato esattamente come un file ordinario.
Descrivete i possibili problemi di protezione che ne derivano
Suggerite un metodo per risolvere ciascuno di questi problemi
- 11.7** Considerate un sistema che supporti 5.000 utenti. Supponete di voler permettere a 4.990 utenti di accedere a un dato file.
- Come specifichereste questo schema di protezione in UNIX?
 - Potete suggerire un altro schema di protezione utilizzabile a questo scopo più efficacemente dello schema fornito da UNIX?
- 11.8** Alcuni ricercatori hanno suggerito che, invece di associare una lista di accesso a ogni file (dove la lista specifica quali utenti possono accedere al file e come), dovremmo avere *una lista di controllo degli utenti* associata a ogni utente (dove la lista specifica a quali file un utente può accedere e come). Discutete i meriti relativi a questi due schemi.

Esercizi

- 11.9** Considerate un file system in cui si può cancellare un file e reclamare il suo spazio di memoria secondaria mentre esistono ancora collegamenti (*link*) a esso. Dite quale problema si può presentare se si crea un nuovo file nella stessa area di memoria o con lo stesso nome di percorso assoluto. Spiegate come tali problemi siano evitabili.
- 11.10** La tabella dei file aperti registra le informazioni riguardanti i file aperti in quel momento. Il sistema operativo dovrebbe mantenere una tabella separata per ciascun utente oppure mantenere una tabella unica per tutti gli utenti con le indicazioni sui file a cui accedono in un certo momento? Se due diversi programmi o utenti eseguono accessi al medesimo file, questi dovrebbero comparire come accessi separati nella tabella dei file aperti? Giustificate le vostre risposte.
- 11.11** Quali sono vantaggi e svantaggi di un sistema che fornisce lock obbligatori anziché lock consultivi, il cui utilizzo è rimesso al giudizio degli utenti?
- 11.12** Fornite esempi di applicazioni che tipicamente accedono ai file sequenzialmente e di applicazioni che accedono ai file in maniera casuale.
- 11.13** Alcuni sistemi aprono un file automaticamente quando ci si riferisce a esso per la prima volta, e lo chiudono al termine del job. Illustrate vantaggi e svantaggi di questo schema, confrontandolo con quello tradizionale, in cui l'utente deve aprire e chiudere il file esplicitamente.
- 11.14** Qualora il sistema operativo sapesse che una certa applicazione accederà ai dati di un file in modo sequenziale, come potrebbe sfruttare questa informazione per migliorare le prestazioni?
- 11.15** Illustrate un'applicazione che potrebbe trarre vantaggio da un sistema operativo che offre l'accesso casuale ai file indicizzati.
- 11.16** Analizzate vantaggi e svantaggi di permettere collegamenti a file che oltrepassano i punti di montaggio (vale a dire, collegamenti che rimandano a file memorizzati in un volume differente).
- 11.17** Alcuni sistemi consentono la condivisione dei file usando una singola copia di ogni file; altri sistemi impiegano più copie, una per ciascun utente che condivide il file. Discutete i vantaggi di ciascun metodo.
- 11.18** Considerate vantaggi e svantaggi insiti nell'associare a file system remoti (memorizzati su file server) una semantica del fallimento diversa da quella prevista nei file system locali.
- 11.19** Quali sono le implicazioni derivanti dall'abbinare a file residenti su file system remoti la semantica della coerenza di UNIX per l'accesso condiviso?

Note bibliografiche

I sistemi di gestione delle basi di dati e le loro strutture di file sono ampiamente descritti in [Silberschatz et al. 2010].

La struttura delle directory a più livelli è stata realizzata per la prima volta nel sistema MULTICS [Organick 1972]. Attualmente la maggior parte dei sistemi dispone di una struttura delle directory a più livelli, per esempio Linux [Love 2010], Mac OS X [Singh 2007], Solaris [McDougal e Mauro 2007] e tutte le versioni di Windows [Russinovich e Solomon 2005].

Il Network File System (NFS), progettato dalla Sun Microsystems, permette di distribuire le strutture delle directory sui calcolatori di una rete. La versione 4 di NFS è descritta in RFC3505 (<http://www.ietf.org/rfc/rfc3505.txt>). Una trattazione generale sui file system in Solaris si può trovare nella guida di Sun *System Administration Guide: Devices and File Systems* (<http://docs.oracle.com/cd/E19253-01/817-5093/>).

Inizialmente proposto da [Su 1982], DNS è passato attraverso diverse revisioni. Il protocollo LDAP, noto anche come X.509, è un sottoinsieme del protocollo di directory distribuita X.500. È stato definito da [Yeong et al. 1995] e incorporato in molti sistemi operativi.

Bibliografia

- [Love 2010] R. Love, *Linux Kernel Development*, 3° Ed., Developer’s Library, 2010.
- [McDougall e Mauro 2007] R. McDougall e J. Mauro, *Solaris Internals*, 2° Ed., Prentice Hall, 2007.
- [Organick 1972] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
- [Russinovich e Solomon 2005] M. E. Russinovich e D. A. Solomon, *Microsoft Windows Internals*, 6° Ed., Microsoft Press, 2005.
- [Silberschatz et al. 2010] A. Silberschatz, H. F. Korth e S. Sudarshan, *Database System Concepts*, 6° Ed., McGraw-Hill, 2010.
- [Singh 2007] A. Singh, *Mac OS X Internals: A Systems Approach*, Addison-Wesley, 2007.
- [Su 1982] Z. Su, “A Distributed System for Internet Name Service”, Network Working Group, Request for Comments: 830, 1982.
- [Yeong et al. 1995] W. Yeong, T. Howes e S. Kille, “Lightweight Directory Access Protocol”, Network Working Group, Request for Comments: 1777, 1995.

CAPITOLO

12

OBIETTIVI DEL CAPITOLO

- Descrizione dei dettagli realizzativi di file system locali e strutture di directory.
- Descrizione della realizzazione di file system remoti.
- Esame dell'allocazione dei blocchi e dei problemi contro degli algoritmi per la gestione dello spazio libero.

Realizzazione del file system

Come illustrato nel Capitolo 11, il file system fornisce il meccanismo per la memorizzazione e l'accesso al contenuto dei file, compresi dati e programmi. Il file system risiede permanentemente nella memoria secondaria, progettata per contenere in modo permanente grandi quantità di dati. Questo capitolo riguarda principalmente i problemi connessi alla memorizzazione e all'accesso ai file nel più comune mezzo di memoria secondaria, il disco. Si esaminano varie modalità d'uso dei file, l'allocazione dello spazio dei dischi, il recupero dello spazio liberato, la registrazione delle locazioni dei dati, e l'interfaccia di altri componenti del sistema operativo alla memoria secondaria. Nel corso della trattazione si considerano anche i problemi riguardanti le prestazioni.

12.1 Struttura del file system

I dischi costituiscono la maggior parte della memoria secondaria in cui si conservano i file system. Hanno due caratteristiche importanti che ne fanno un mezzo adatto a questo scopo:

1. si possono riscrivere localmente; si può leggere un blocco dal disco, modificarlo e quindi scriverlo nella stessa posizione;
2. è possibile accedere direttamente a qualsiasi blocco di informazioni del disco, quindi risulta semplice accedere a qualsiasi file, sia in modo sequenziale sia in modo diretto, e passare da un file all'altro spostando le testine di lettura e scrittura e attendendo la rotazione del disco.

La struttura dei dischi è analizzata in modo particolareggiato nel Capitolo 10.

Per migliorare l'efficienza dell'I/O, i trasferimenti tra memoria centrale e dischi si eseguono per **blocchi**. Ciascun blocco è composto da uno o più settori. A seconda dell'unità a disco, la dimensione dei settori è compresa tra 32 byte e 4096 byte; di solito è pari a 512 byte.

Per fornire un efficiente e conveniente accesso al disco, il sistema operativo fa uso di uno o più **file system** che consentono di memorizzare, individuare e recuperare facilmente i dati. Un file system presenta due problemi di progettazione molto diversi. Il primo riguarda la definizione dell'aspetto del file system agli occhi dell'utente. Questo compito implica la definizione di un file e dei suoi attributi, delle operazioni messe su un file e della struttura delle directory per l'organizzazione dei file. Il secondo riguarda la creazione di algoritmi e strutture dati che permettano di far corrispondere il file system logico ai dispositivi fisici di memoria secondaria.

Lo stesso file system è generalmente composto da molti livelli distinti. La struttura illustrata nella Figura 12.1 è un esempio di struttura stratificata. Ogni livello si serve delle funzioni dei livelli inferiori per creare nuove impostate dai livelli superiori.

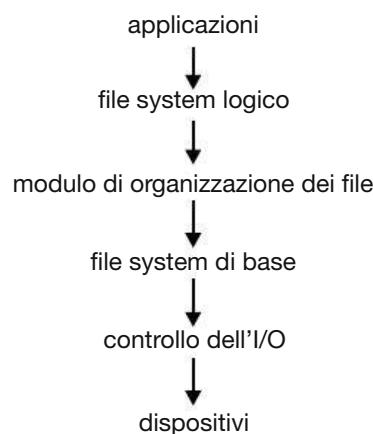


Figura 12.1 File system stratificato.

Il livello più basso, il **controllo dell'I/O**, costituito dai driver dei dispositivi e dai gestori dei segnali d'interruzione, si occupa del trasferimento delle informazioni tra memoria centrale e memoria secondaria. Un driver di dispositivo si può concepire come un traduttore che riceva comandi ad alto livello, come “recupera il blocco 123”, e che emette istruzioni di basso livello dipendenti dall'hardware, usate dal controllore che fa da interfaccia tra i dispositivi di I/O e il resto del sistema. Un driver di dispositivo di solito scrive specifiche configurazioni di bit in specifiche locazioni della memoria del controllore di I/O per indicare quali azioni il dispositivo di I/O debba compiere, e su quali locazioni. I dettagli dei driver dei dispositivi e le strutture per l'I/O sono trattati nel Capitolo 13.

Il **file system di base** deve soltanto inviare dei generici comandi all'appropriato driver di dispositivo per leggere e scrivere blocchi fisici nel disco. Ogni blocco fisico si identifica col suo indirizzo numerico nel disco, per esempio unità 1, cilindro 73, traccia 2, settore 10. Questo strato gestisce inoltre buffer di memoria e le cache che conservano vari blocchi del file system, delle directory e dei dati. Un blocco viene allocato nel buffer prima che possa verificarsi il trasferimento di un blocco del disco. Quando il buffer è pieno, il gestore del buffer deve recuperare più spazio di memoria per il buffer oppure deve liberare spazio nel buffer per permettere il completamento di un I/O richiesto. Le cache servono a conservare i metadati del file system usati frequentemente, in modo da migliorare le prestazioni. La gestione dei loro contenuti è quindi un punto critico per conseguire prestazioni ottimali del sistema.

Il **modulo di organizzazione dei file** è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi. Conoscendo il tipo di allocazione dei file usato e la locazione dei file, può tradurre gli indirizzi dei blocchi logici negli indirizzi dei blocchi fisici che il file system di base deve trasferire. I blocchi logici di ciascun file sono numerati da 0 (o 1) a n ; i numeri dei blocchi fisici contenenti i dati di solito non corrispondono ai numeri dei blocchi logici; per individuare ciascun blocco è quindi necessaria una traduzione. Il modulo di organizzazione dei file comprende anche il gestore dello spazio libero, che registra i blocchi non assegnati e li mette a disposizione del modulo di organizzazione dei file quando sono richiesti.

Infine, il **file system logico** gestisce i metadati; si tratta di tutte le strutture del file system, eccetto gli effettivi dati (il contenuto dei file). Il file system logico gestisce la struttura della directory per fornire al modulo di organizzazione dei file le informazioni di cui necessita, dato un nome simbolico di file. Mantiene le strutture di file tramite i **blocchi di controllo dei file** (*file control block*, FCB), detti **inode** nei file system UNIX, contenenti informazioni sui file, come la proprietà, i permessi, e la posizione del contenuto del file. Come si discute nel Capitolo 11 e Capitolo 14, il file system logico è responsabile anche della protezione e della sicurezza.

Nei file system stratificati la duplicazione di codice è ridotta al minimo. Il controllo dell'I/O e, talvolta, il codice di base del file system, possono essere utilizzati da più di un file system. Ogni file system ha poi i propri moduli che gestiscono il file system logico e l'organizzazione dei file. Sfortunatamente, la stratificazione può comportare un maggior overhead del sistema operativo, che può generare un conseguente

decadimento delle prestazioni. L'utilizzo della stratificazione e le scelte sul numero di strati da impiegare e sulle loro funzionalità rappresentano una grande sfida per la progettazione di nuovi sistemi.

Esistono svariati tipi di file system al giorno d'oggi, e non è raro che i sistemi operativi ne prevedano più d'uno. Molti CD-ROM, a esempio, sono scritti nel formato ISO 9660, uno standard concordato dai produttori di CD-ROM. Oltre ai file system dei supporti rimovibili, ciascun sistema operativo possiede un file system, o più di uno, basato sui dischi. UNIX adotta il **file system UNIX** (UFS), che si fonda a sua volta sul Berkeley Fast File System (FFS). Windows adotta i formati FAT, FAT32 e NTFS (o File System di Windows NT), così come i formati per CD-ROM e DVD. Sebbene Linux possa funzionare con più di quaranta file system diversi, quello standard è noto come **file system esteso**, le cui versioni maggiormente diffuse sono ext2 ed ext3. Esistono anche file system distribuiti, in cui un file system su server è montato da uno o più client in una rete.

La ricerca relativa ai file system continua a essere un'area attiva della progettazione e dell'implementazione dei sistemi operativi. Per soddisfare esigenze di memorizzazione e recupero dati specifiche dell'azienda, Google ha progettato un proprio file system che permette un accesso ad alte prestazioni a un gran numero di dischi da parte di numerosi client. Un altro progetto interessante è FUSE, che garantisce flessibilità nello sviluppo e nell'utilizzo del file system grazie all'implementazione e all'esecuzione di file system a livello utente invece che a livello del codice del kernel. Gli utenti di FUSE possono aggiungere nuovi file system a numerosi sistemi operativi e utilizzarli per gestire i propri file.

12.2 Realizzazione del file system

Come detto nel Paragrafo 11.1.2, per permettere ai processi di richiedere l'accesso al contenuto dei file, i sistemi operativi offrono le chiamate di sistema `open()` e `close()`. In questo paragrafo si approfondiscono le strutture dati e le operazioni usate per realizzare le operazioni del file system.

12.2.1 Introduzione

Per realizzare un file system si usano parecchie strutture dati, sia nei dischi sia in memoria. Queste strutture variano a seconda del sistema operativo e del file system, ma esistono dei principi generali. Nei dischi, il file system tiene informazioni su come eseguire l'avviamento di un sistema operativo memorizzato nei dischi stessi, il numero totale di blocchi, il numero e la locazione dei blocchi liberi, la struttura delle directory e i singoli file. Molte di loro sono analizzate in modo particolareggiato nel seguito di questo capitolo.

Fra le strutture presenti nei dischi ci sono le seguenti.

- Il **blocco di controllo dell'avviamento** (*boot control block*), per ogni volume, contenente le informazioni necessarie al sistema per l'avviamento di un siste-

ma operativo da quel volume; se il disco non contiene un sistema operativo, tale blocco può essere vuoto. Di solito è il primo blocco di un volume. Nell'UFS, si chiama **blocco d'avviamento** (*boot block*); nell'NTFS, **settore d'avviamento della partizione** (*partition boot sector*).

- Il **blocco di controllo del volume** (*volume control block*); ciascuno di essi contiene i dettagli riguardanti il relativo volume (o partizione), come il numero e la dimensione dei blocchi nella partizione, il contatore dei blocchi liberi e i relativi puntatori, il contatore degli FCB liberi e i relativi puntatori. Nell'UFS si chiama **superblocco**; nell'NTFS si chiama **tabella principale dei file** (*master file table, MFT*).
- La **struttura della directory** (una per file system) usata per organizzare i file. Nel caso dell'UFS comprende i nomi dei file e i numeri di **inode** associati. Nel caso dell'NTFS è memorizzata nella **tabella principale dei file** (*master file table*).
- Il **blocco di controllo del file** (FCB), contenenti molti dettagli del relativo file. Ha un identificatore unico per poterlo associare a una voce della directory. Nell'NTFS, queste informazioni sono memorizzate all'interno della tabella principale dei file, che si serve di una struttura di base di dati relazionale, con una riga per ciascun file.

Le informazioni tenute in memoria servono sia per la gestione del file system sia per migliorare le prestazioni attraverso l'uso di cache. I dati si caricano al momento del montaggio, si aggiornano mentre si opera sul file system e si eliminano allo smontaggio. Le strutture che vi possono essere incluse sono di diverso tipo:

- la **tabella di montaggio**, in memoria, che contiene informazioni relative a ciascun volume montato;
- una cache della struttura della directory, tenuta in memoria, contenente le informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente (per le directory che costituiscono dei punti di montaggio, può essere presente un puntatore alla tabella dei volumi);
- la **tabella di sistema dei file aperti**, contenente una copia dell'FCB per ciascun file aperto, insieme con altre informazioni;
- la **tabella dei file aperti per ciascun processo**, contenente un puntatore al corrispondente elemento della tabella generale dei file aperti, insieme con altre informazioni;
- i buffer che conservano blocchi del file system durante la loro lettura o scrittura sul disco.

Le applicazioni, per creare un nuovo file, eseguono una chiamata al file system logico, il quale conosce il formato della struttura della directory. Per creare un nuovo file, esso crea un nuovo FCB. (In alternativa, nel caso dei file system che creano tutti gli FCB al momento della loro installazione, esso alloca semplicemente un FCB libero.) Il si-

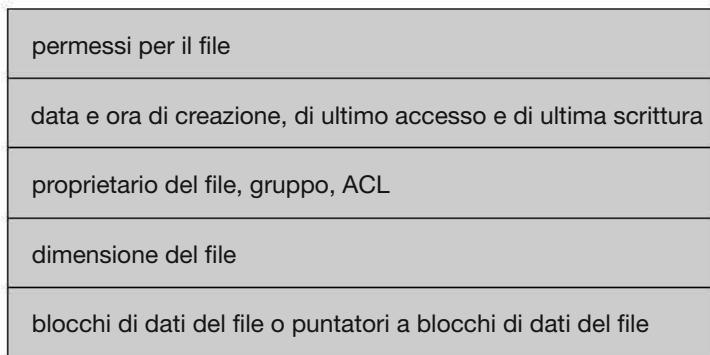


Figura 12.2 Tipica struttura di FCB (*file-control block*).

stema carica quindi la directory appropriata in memoria, la aggiorna con il nome del nuovo file e con l'FCB associato, e la scrive nuovamente sul disco. Una tipica struttura di FCB è illustrata nella Figura 12.2.

Alcuni sistemi operativi, compreso UNIX, trattano le directory esattamente come i file, distinguendole con un campo per il tipo che indica che si tratta di una directory. Altri, tra cui il sistema operativo Windows, dispongono di chiamate di sistema distinte per i file e le directory e trattano le directory come entità separate dai file. Indipendentemente da tali questioni strutturali, il file system logico può basarsi sul modulo che si occupa dell'organizzazione dei file per far corrispondere l'I/O su directory ai numeri di blocchi di disco, che poi si passano al file system di base e al sistema per il controllo dell'I/O.

Una volta creato un file, per essere usato per operazioni di I/O deve essere *aperto*. La chiamata di sistema `open()` passa un nome di file al file system logico. Per controllare se il file sia già in uso da parte di qualche processo, la chiamata `open()` dapprima esamina la tabella di sistema dei file aperti. In caso affermativo, aggiunge un elemento alla tabella dei file aperti del processo che punta alla tabella dei file aperti in tutto il sistema. Questo algoritmo può eliminare significativi overhead. Se il file non è già aperto, se ne ricerca il nome all'interno della directory. Alcune porzioni della struttura delle directory sono di solito tenute in memoria per accelerare le operazioni sulle directory. Una volta trovato il file, si copia l'FCB nella tabella di sistema dei file aperti, tenuta in memoria. Questa tabella non solo contiene l'FCB, ma tiene anche traccia del numero di processi che in quel momento hanno il file aperto.

Successivamente, si crea un elemento nella tabella dei file aperti del processo con un puntatore alla tabella di sistema e con alcuni altri campi. Questi altri campi possono comprendere un puntatore alla posizione corrente nel file (per successive operazioni `read()` o `write()`) e il tipo d'accesso specificato all'apertura del file. La `open()` riporta un puntatore all'elemento appropriato nella tabella dei file aperti del processo, sicché tutte le operazioni sul file si svolgeranno usando questo puntatore. Il nome del file potrebbe non essere contenuto nella tabella dei file aperti, visto che, una volta che il corrispondente FCB è stato individuato nei dischi, il sistema non ne ha bisogno. Tuttavia, potrebbe venir memorizzato in una cache per risparmiare tempo

sulle aperture successive dello stesso file. Il nome dato all'elemento della tabella è detto **descrittore di file** (*file descriptor*) in UNIX, e **handle del file** in Windows.

Quando un processo chiude il file, si cancella il relativo elemento nella tabella dei file aperti del processo e si decrementa il contatore associato al file nella tabella di sistema. Se tutti i processi che avevano aperto il file lo hanno chiuso, si riscrivono i metadati aggiornati nella struttura della directory nei dischi e si cancella il relativo elemento nella tabella di sistema dei file aperti.

Alcuni sistemi complicano ulteriormente lo schema descritto, usando il file system come interfaccia per altri aspetti del sistema, come la comunicazione in rete. Per esempio, nell'UFS, la tabella generale dei file aperti contiene gli *inode* e altre informazioni su file e directory, ma contiene anche informazioni simili per le connessioni di rete e i dispositivi. In questo modo si può usare un unico meccanismo per molteplici fini.

Le questioni concernenti l'uso delle cache per queste strutture non vanno però trascurate. La maggior parte dei sistemi mantiene in memoria tutta l'informazione su un file aperto, eccetto i suoi effettivi blocchi di dati. Il sistema UNIX BSD è noto per il suo uso di cache ovunque sia possibile risparmiare su operazioni di I/O nei dischi. La sua frequenza media di successi nella cache, pari all'85 per cento, dimostra l'utilità di queste tecniche.

La Figura 12.3 riassume le strutture dati che si usano nella realizzazione di un file system.

12.2.2 Partizioni e montaggio

Un disco si può strutturare in vari modi, a seconda del sistema operativo che lo gestisce. Si può suddividere in più partizioni, oppure un volume può comprendere più partizioni su molteplici dischi. Qui trattiamo il primo caso, mentre il secondo, che si può considerare più correttamente un caso particolare di organizzazione RAID, è trattato nel Paragrafo 10.7.

Ciascuna partizione può essere priva di struttura logica (*raw partition*), oppure può contenere un file system. Si usa un **disco privo di struttura logica** (*raw disk*) se nessun file system è appropriato all'utilizzo che se ne vuole fare. Il sistema operativo UNIX impiega una partizione priva di struttura per l'area d'avvicendamento dei processi; per questo scopo usa un formato specifico. Allo stesso modo alcuni sistemi di gestione di basi di dati usano dischi non strutturati e formattano i dati secondo le proprie necessità. Un disco privo di struttura logica può anche contenere informazioni necessarie per sistemi RAID di gestione dei dischi, per esempio le mappe di bit che indicano quali blocchi sono duplicati in altri dischi, e in quali sono state effettuate modifiche che si devono duplicare. Analogamente, può contenere una piccola base di dati di informazioni sulla configurazione RAID, per esempio, quali dischi appartengono a ciascun insieme RAID. Il Paragrafo 10.5.1 affronta altri aspetti concernenti l'uso dei raw disk.

Come descritto nel Paragrafo 10.5.2 le informazioni relative all'avviamento del sistema si possono registrare in un'apposita partizione, che anche in questo caso ha

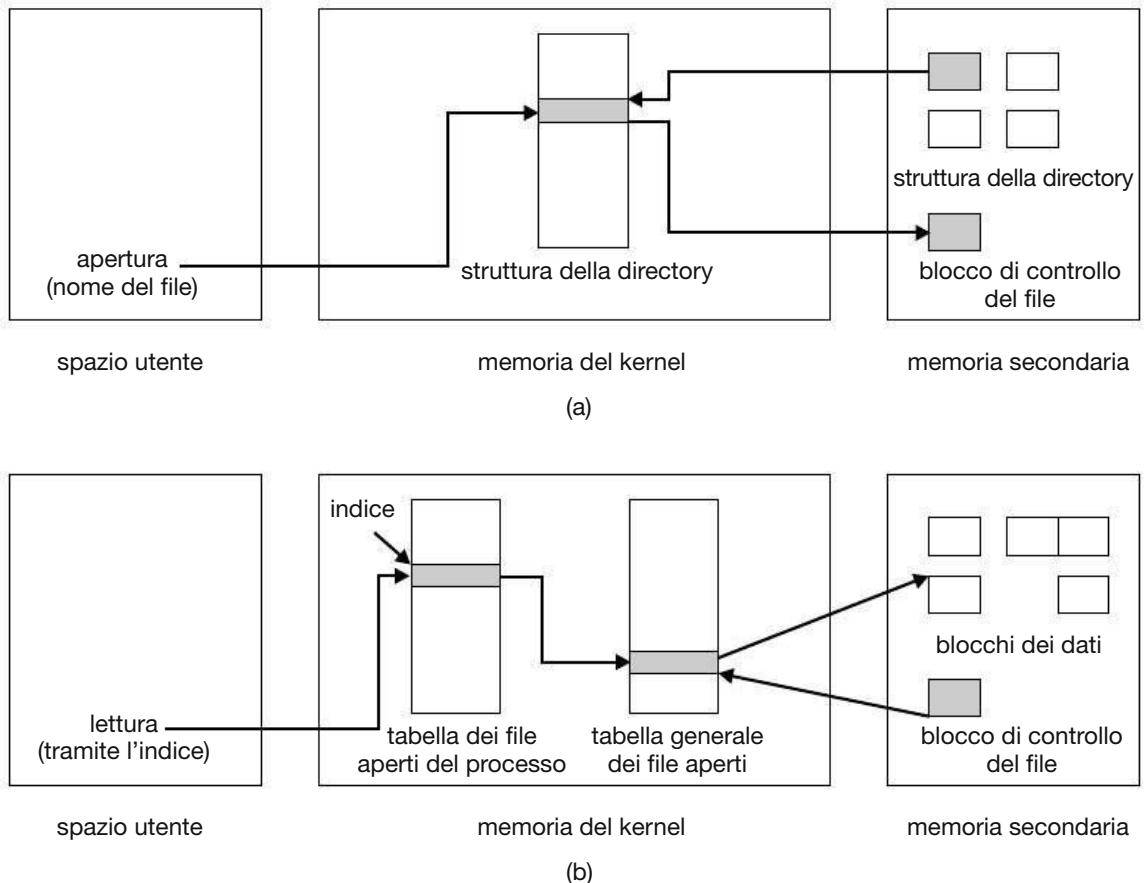


Figura 12.3 Strutture del file system che si mantengono nella memoria; (a) apertura di file; (b) lettura di file.

un proprio formato, poiché nella fase d'avviamento il sistema non ha ancora caricato il codice del file system e quindi non può interpretarne il formato. Questa partizione consiste piuttosto in una serie sequenziale di blocchi, che si carica in memoria come un'immagine. L'esecuzione dell'immagine comincia a una locazione prefissa, per esempio il primo byte. Questo **boot loader** ha invece abbastanza informazioni sul file system da essere in grado di caricare il kernel e avviare l'esecuzione. Esso può contenere più informazioni di quelle che servono per un singolo sistema operativo. Su molti sistemi, per esempio, è possibile l'installazione di più sistemi operativi. In questo caso l'area d'avviamento contiene un boot loader, capace di interpretare diversi file system e diversi sistemi operativi. Una volta caricato, può avviare uno dei sistemi operativi disponibili nei dischi. Il disco può avere più partizioni, ognuna contenente un diverso tipo di file system e un sistema operativo differente.

Nella fase di caricamento del sistema operativo, si esegue il montaggio della **partizione radice** (*root partition*), che contiene il kernel del sistema operativo e in alcuni casi altri file di sistema. A seconda del sistema operativo, il montaggio di altri volumi avviene automaticamente in questa fase oppure si può compiere successivamente in maniera manuale. Durante l'operazione di montaggio, il sistema verifica

che il dispositivo contenga un file system valido chiedendo al dispositivo di leggere la directory di dispositivo e verificando che la directory abbia il formato corretto. Se così non fosse, è necessaria una verifica della coerenza della partizione e una eventuale correzione, con o senza l'intervento dell'utente. Infine, il sistema annota nella **tabella di montaggio** in memoria che un file system è stato montato e il tipo di file system. I dettagli di questa funzione dipendono dal sistema operativo.

I sistemi Microsoft Windows eseguono il montaggio di ogni volume in uno spazio di nomi separato, identificato da una lettera seguita dai due punti (:). Per esempio, per memorizzare che un file system è stato montato in F:, il sistema operativo introduce un puntatore al file system in un campo della struttura del dispositivo corrispondente a F:. Quando un processo specifica la lettera dell'unità, il sistema operativo trova il puntatore al file system appropriato e attraversa le strutture della directory in quel dispositivo per trovare lo specifico file o directory. Le più recenti versioni di Windows permettono il montaggio di un file system in qualsiasi punto all'interno della struttura della directory esistente.

In UNIX il montaggio di un file system si può compiere in qualsiasi directory. Questa funzione si realizza impostando un flag nella copia in memoria dell'*inode* di quella directory, segnalando che la directory è un punto di montaggio. Un campo dell'*inode* punta a un elemento nella tabella di montaggio, che indica quale dispositivo è montato in quella posizione. L'elemento della tabella di montaggio contiene un puntatore al superblocco del file system in quel dispositivo. Questo schema permette al sistema operativo di attraversare facilmente la propria struttura della directory, attraversando in maniera trasparente file system di tipo diverso.

12.2.3 File system virtuali

Nel paragrafo precedente si sottolinea il fatto che i sistemi operativi moderni devono gestire contemporaneamente tipi di file system diversi. Per capire come si può realizzare questa funzione occorre tuttavia considerare il modo in cui un sistema operativo può consentire l'integrazione di diversi tipi di file system in un'unica struttura della directory, così da permettere agli utenti di spostarsi senza problemi da un tipo di file system all'altro, mentre percorrono tutta la struttura.

Un metodo ovvio ma non ottimale per realizzare più tipi di file system è scrivere procedure di gestione di file e directory differenti per ciascun tipo di file system. Al contrario, la maggior parte dei sistemi operativi, compreso UNIX, impiega tecniche orientate agli oggetti per semplificare, organizzare e rendere modulare la soluzione. L'uso di queste tecniche rende possibile l'implementazione, nella stessa struttura, di tipi di file system molto diversi tra loro, compresi i file system di rete, come l'NFS. Gli utenti possono accedere a file contenuti in diversi file system nei dischi locali, o anche in file system disponibili tramite la rete.

Per isolare le funzioni di base delle chiamate di sistema dai dettagli di implementazione si adoperano apposite strutture dati e procedure. In questo modo la realizzazione del file system si articola in tre strati principali, riportati in modo schematico

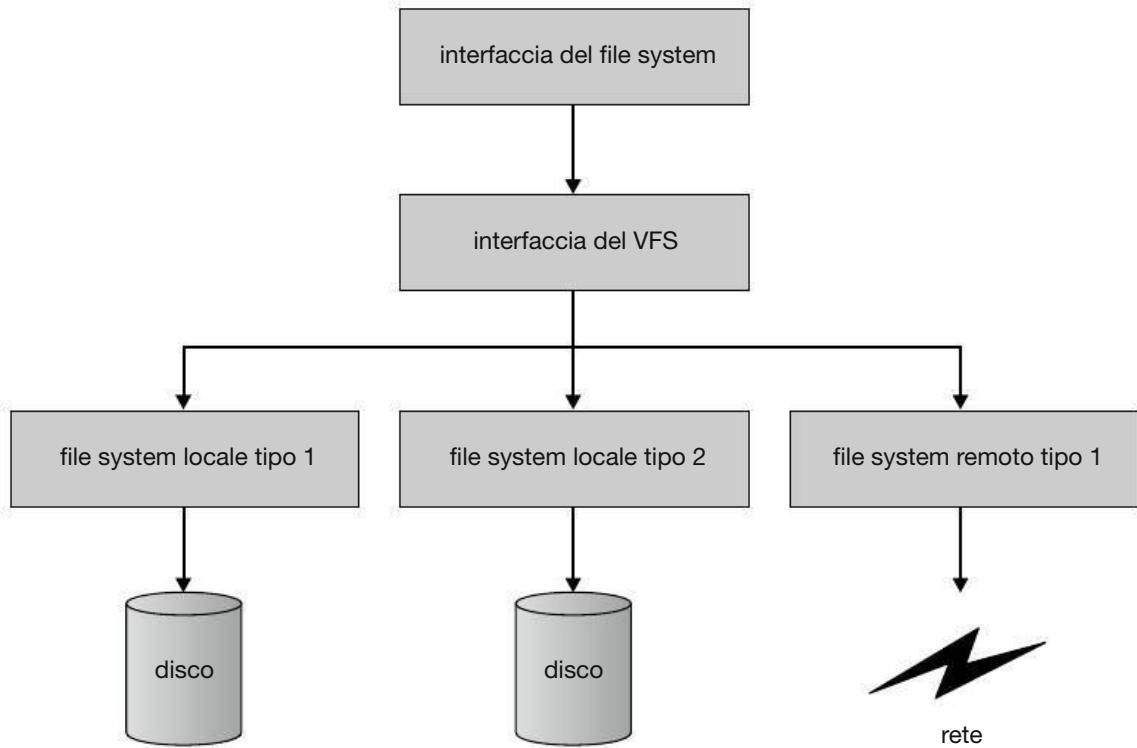


Figura 12.4 Schema di un file system virtuale.

nella Figura 12.4. Il primo strato è l'interfaccia del file system, basata sulle chiamate di sistema `open()`, `read()`, `write()` e `close()` e sui descrittori di file.

Il secondo strato si chiama strato del **file system virtuale** (*virtual file system*, VFS) e svolge due funzioni importanti.

1. Separa le operazioni generiche del file system dalla loro realizzazione definendo un'interfaccia VFS uniforme. Nello stesso calcolatore possono coesistere più implementazioni dell'interfaccia VFS, che permettono un accesso trasparente a diversi tipi di file system montati localmente.
2. Permette la rappresentazione univoca di un file su tutta una rete. Il VFS è basato su una struttura di rappresentazione dei file detta **vnode** che contiene un indicatore numerico unico per tutta la rete per ciascun file. (Gli *inode* di UNIX sono unici solo all'interno di un singolo file system.) Tale unicità per tutta la rete è richiesta per la gestione dei file system di rete. Il kernel contiene una struttura *vnode* per ciascun nodo attivo, sia che si tratti di un file sia che si tratti di una directory.

Quindi, il VFS distingue i file locali da quelli remoti, e distingue i file locali a seconda del relativo tipo di file system.

Il VFS, a seconda del tipo di file system, attiva le operazioni specifiche di un file system per gestire le richieste locali, e invoca le procedure del protocollo NFS per le richieste remote. Gli handle del file si costruiscono a partire dai *vnode* relativi e s'inviano a queste procedure come argomenti. Lo strato che realizza il codice di uno specifico file system o il protocollo di file system remoto è il terzo dell'architettura.

Esaminiamo succintamente l’architettura VFS di Linux. I quattro tipi più importanti di oggetti definiti nel VFS di Linux sono:

- l’**oggetto inode**, che rappresenta il singolo file;
- l’**oggetto file**, che rappresenta un file aperto;
- l’**oggetto superblock**, che rappresenta un intero file system;
- l’**oggetto dentry**, che rappresenta il singolo elemento della directory.

Per ognuno di questi tipi, VFS specifica un insieme di operazioni da implementare. Ciascun oggetto di uno di questi tipi contiene un puntatore a una tabella di funzioni; questa, alla sua volta, contiene gli indirizzi delle effettive funzioni che implementano le operazioni definite per quel particolare oggetto. Per esempio, una versione abbreviata della API di alcune delle operazioni dell’oggetto file comprende:

- `int open(...)` – apre il file.
- `int close(...)` – chiude un file aperto.
- `ssize_t read(...)` – legge dal file.
- `ssize_t write(...)` – scrive sul file.
- `int mmap(...)` – mappa il file in memoria.

Ogni implementazione dell’oggetto file per uno specifico tipo di file deve implementare tutte le funzioni specificate nella definizione dell’oggetto file. (La definizione completa dell’oggetto file si trova nella struttura `struct file_operations`, nel file `/usr/include/linux/fs.h`).

Questo schema permette allo strato software VFS di eseguire operazioni su uno degli oggetti in questione invocando l’appropriata funzione della tabella delle funzioni dell’oggetto, senza dover conoscere i dettagli dello specifico oggetto. Per esempio, VFS non sa, né vuol sapere, se un certo inode rappresenti un file su disco, un file di directory o un file remoto. L’implementazione appropriata dell’operazione `read()` per l’oggetto in questione è comunque reperibile sempre nel medesimo punto all’interno della tabella delle funzioni: invocando tale implementazione, VFS può disinteressarsi dei dettagli legati a come vengono effettivamente letti i dati.

12.3 Realizzazione delle directory

La selezione degli algoritmi di allocazione e degli algoritmi di gestione delle directory ha un grande effetto sull’efficienza, le prestazioni e l’affidabilità del file system. Nel seguito discuteremo i tradeoff legati alla scelta di questi algoritmi.

12.3.1 Lista lineare

Il più semplice metodo di realizzazione di una directory è basato sull’uso di una lista lineare contenente i nomi dei file con puntatori ai blocchi di dati. Questo metodo è di facile programmazione, ma la sua esecuzione è onerosa in termini di tempo. Per

creare un nuovo file occorre prima esaminare la directory per essere sicuri che non esista già un file con lo stesso nome, quindi aggiungere un nuovo elemento alla fine della directory. Per cancellare un file occorre cercare nella directory il file con quel nome, quindi rilasciare lo spazio che gli era assegnato. Esistono vari metodi per riutilizzare un elemento della directory: si può contrassegnare l'elemento come non usato (attribuendogli un nome speciale, come un nome blank, oppure includendo un bit d'uso in ogni elemento), oppure può essere aggiunto a una lista di elementi di directory liberi; una terza possibilità prevede la copiatura dell'ultimo elemento della directory in una locazione liberata e la diminuzione della lunghezza della directory. Per ridurre il tempo di cancellazione di un file si può usare anche una lista concatenata.

Il vero svantaggio dato da una lista lineare di elementi di directory è dato dalla ricerca lineare di un file. Le informazioni sulla directory vengono usate frequentemente, e gli utenti si accorgono se l'accesso a tali informazioni è lento. In effetti, molti sistemi operativi impiegano una cache software per memorizzare le informazioni di directory usate più recentemente. La presenza nella cache delle informazioni richieste ne evita la continua rilettura dai dischi. Una lista ordinata permette una ricerca binaria e riduce il tempo medio di ricerca, tuttavia il requisito dell'ordinamento può complicare la creazione e la cancellazione di file, poiché, per tenere ordinata la lista, può essere necessario spostare quantità notevoli di informazioni di directory. In questo caso, può essere d'aiuto una struttura dati più raffinata, come un albero bilanciato. Un vantaggio della lista ordinata è che consente di produrre l'elenco ordinato del contenuto della directory senza una fase d'ordinamento separata.

12.3.2 Tabella hash

Un'altra struttura dati che si usa per realizzare le directory è la **tabella hash**. In questo caso una lista lineare contiene gli elementi di directory, ma si usa anche una struttura dati hash. La tabella hash riceve un valore calcolato a partire dal nome del file e riporta un puntatore al nome del file nella lista lineare. Questa struttura dati può diminuire notevolmente il tempo di ricerca nella directory. L'inserimento e la cancellazione sono abbastanza semplici, anche se occorre prendere provvedimenti per gestire le **collisioni**, cioè situazioni in cui da due nomi di file si ottiene un riferimento alla stessa locazione.

Le maggiori difficoltà legate a una tabella hash sono la sua dimensione, che in genere è fissa, e la dipendenza della funzione hash da tale dimensione. Si supponga, per esempio, di realizzare una tabella hash di 64 elementi con gestione lineare delle collisioni; la funzione hash converte i nomi di file in interi da 0 a 63, per esempio, usando il resto di una divisione per 64. Per creare in un secondo tempo un sessantacinquesimo file occorre allungare la tabella hash della directory, per esempio fino a 128 elementi. Occorre quindi una nuova funzione hash per associare i nomi di file all'intervallo compreso tra 0 e 127, e gli elementi esistenti nella directory si devono riorganizzare in modo da riflettere i loro nuovi valori della funzione hash.

Alternativamente, ciascun elemento della tabella hash, anziché un singolo valore, può essere una lista concatenata; ciò consente di risolvere le collisioni aggiungendo il nuovo elemento alla lista concatenata. Le ricerche vengono alquanto rallentate, poiché la ricerca per nome può richiedere l’attraversamento di una lista concatenata degli elementi in collisione della tabella hash; tuttavia tale metodo è verosimilmente più veloce di una ricerca lineare nell’intera directory.

12.4 Metodi di allocazione

La natura ad accesso diretto dei dischi dà flessibilità nella realizzazione dei file. In quasi tutti i casi, molti file si memorizzano nello stesso disco. Il problema principale consiste dunque nell’allocare lo spazio per questi file in modo che lo spazio nel disco sia usato efficientemente e l’accesso ai file sia rapido. Esistono tre metodi principali per l’allocazione dello spazio di un disco; può essere contigua, concatenata o indirizzata. Ciascuno di questi metodi presenta vantaggi e svantaggi. Anche se alcuni sistemi dispongono di tutti e tre i metodi, più spesso un sistema usa un unico metodo per tutti i file all’interno di un certo tipo di file system.

12.4.1 Allocazione contigua

Per usare il metodo di **allocazione contigua**, ogni file deve occupare un insieme di blocchi contigui del disco. Gli indirizzi del disco definiscono un ordinamento lineare nel disco stesso. Con questo ordinamento, supponendo che un unico job stia accedendo al disco, l’accesso al blocco $b + 1$ dopo il blocco b non richiede normalmente alcuno spostamento della testina. Se la testina deve essere spostata (dall’ultimo settore di un cilindro al primo settore del cilindro successivo) lo spostamento è di una sola traccia. Quindi, il numero dei posizionamenti (*seek*) richiesti per accedere a file il cui spazio è allocato in modo contiguo è minimo, così com’è trascurabile il tempo di ricerca (*seek time*), quando quest’ultimo è necessario.

L’allocazione contigua dello spazio per un file è definita dall’indirizzo del primo blocco e dalla lunghezza (espressa in numero di blocchi). Se il file è lungo n blocchi e comincia dalla locazione b , allora occupa i blocchi $b, b + 1, b + 2, \dots, b + n - 1$. L’elemento di directory per ciascun file indica l’indirizzo del blocco d’inizio e la lunghezza dell’area assegnata per questo file (Figura 12.5).

Accedere a un file il cui spazio è assegnato in modo contiguo è facile. Quando si usa un accesso sequenziale, il file system memorizza l’indirizzo dell’ultimo blocco cui è stato fatto riferimento e, se è necessario, legge il blocco successivo. Nel caso di un accesso diretto al blocco i di un file che comincia al blocco b si può accedere immediatamente al blocco $b + i$. Quindi, sia l’accesso sequenziale sia quello diretto si possono gestire con l’allocazione contigua.

L’allocazione contigua presenta però alcuni problemi. Una difficoltà riguarda l’individuazione dello spazio per un nuovo file. La realizzazione del sistema di gestione dello spazio libero, illustrata nel Paragrafo 12.5, determina il modo in cui tale compito

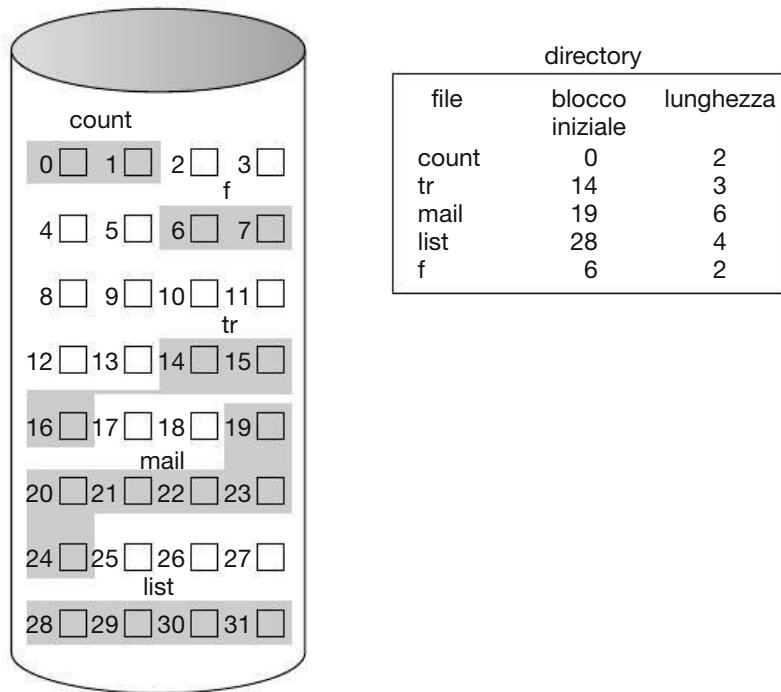


Figura 12.5 Allocazione contigua dello spazio dei dischi.

viene eseguito. Si può usare ogni sistema di gestione, anche se alcuni sono più lenti di altri.

Il problema dell’allocazione contigua dello spazio dei dischi si può considerare un’applicazione particolare del problema generale dell’**allocazione dinamica della memoria**, trattato nel Paragrafo 8.3; il problema generale è, infatti, quello di soddisfare una richiesta di dimensione n data una lista di buchi liberi. I più comuni criteri di scelta di un buco libero da un insieme di buchi disponibili sono *first-fit* e *best-fit*. Simulazioni hanno dimostrato che questi due criteri sono più efficienti del *worst-fit* sia in termini di tempo sia d’uso della memoria. Nessuno dei due è chiaramente migliore dell’altro rispetto all’uso della memoria, ma *first-fit* è generalmente più rapido.

Questi algoritmi soffrono della **frammentazione esterna**: assegnando e liberando lo spazio per i file, lo spazio libero dei dischi viene frammentato in tanti piccoli pezzi. La frammentazione esterna si ha ogniqualvolta lo spazio libero è suddiviso in pezzi, e diviene un problema quando il più grande di tali pezzi contigui non è sufficiente a soddisfare una richiesta; la memoria viene frammentata in tanti buchi, nessuno dei quali è abbastanza grande da contenere i dati. A seconda della capacità dei dischi e della dimensione media dei file, la frammentazione esterna può essere un problema più o meno grave.

Una strategia per prevenire la perdita di una quantità significativa di spazio sul disco a causa della frammentazione esterna consiste nel copiare un intero file system su un altro disco. Così si libera completamente il primo disco creando un ampio spazio libero contiguo; poi si copiano nuovamente i file nel primo disco, assegnando spa-

zio contiguo da questa ampia zona libera. Questo schema **compatta** efficacemente tutto lo spazio libero in uno spazio contiguo, risolvendo il problema della frammentazione. Il costo di questa compattazione è rappresentato dal tempo necessario, ed è particolarmente alto per i dischi di grande capacità; per essi, compattare lo spazio può richiedere ore e può essere necessario eseguire tale operazione settimanalmente. Alcuni sistemi richiedono l'esecuzione **non in linea** (*off-line*) di questa funzionalità, ossia con il file system non montato. Durante questo periodo di indisponibilità (*down time*) il normale funzionamento del sistema non è in genere possibile, quindi tale compattazione viene evitata a tutti i costi per i calcolatori operativi. La maggior parte dei sistemi moderni sono invece in grado di eseguire la deframmentazione **in linea** (*on-line*), ossia durante il loro normale funzionamento, a prezzo, però, di una notevole diminuzione delle prestazioni.

Un altro problema che riguarda l'allocazione contigua è la determinazione della quantità di spazio necessaria per un file. Quando si crea un file, occorre trovare e allocare lo spazio di cui necessita. Come può il programma o la persona che lo crea conoscere la dimensione del file da creare? In alcuni casi questa dimensione si può stabilire in modo abbastanza semplice, per esempio quando si copia un file esistente; in generale, tuttavia, non è facile stimare la dimensione di un file che deve contenere dati prodotti da un programma.

Se un file riceve poco spazio, può essere impossibile estenderlo: soprattutto nel caso in cui si adoperi il criterio di allocazione *best-fit*, lo spazio alle due estremità del file può essere già in uso, quindi non è possibile ampliare il file in modo contiguo. Esistono allora due possibilità. La prima è terminare il programma utente con un idoneo messaggio d'errore. L'utente deve allora allocare più spazio ed eseguire di nuovo il programma. Queste esecuzioni ripetute possono essere onerose; per prevenire tale circostanza, normalmente l'utente sovrastima la quantità di spazio necessaria, sprecandone parecchio. L'altra possibilità consiste nel trovare un buco più grande, copiare il contenuto del file nel nuovo spazio e rilasciare lo spazio precedente. Queste operazioni si possono ripetere finché esiste spazio, anche se ciò può far perdere tempo. In questo caso tuttavia non è necessario informare esplicitamente l'utente su che cosa stia succedendo; anche se sempre più lentamente, il sistema prosegue le attività nonostante il problema.

Anche se si conosce in anticipo la quantità di spazio necessaria per un file, l'allocazione preventiva può in ogni modo essere inefficiente. A un file che cresce lentamente in un periodo di tempo lungo (mesi o anni) si deve allocare spazio sufficiente per la sua dimensione finale, anche se molto di quello spazio può rimanere inutilizzato per parecchio tempo. Il file ha perciò una grande frammentazione interna.

Per ridurre al minimo questi inconvenienti, alcuni sistemi operativi fanno uso di uno schema di allocazione contigua modificato: inizialmente si assegna una porzione di spazio contiguo, e se questa non è abbastanza grande si aggiunge un'altra porzione di spazio contiguo, detta **estensione**. Allora la locazione dei blocchi dei file si registra come una locazione e un numero dei blocchi, insieme con l'indirizzo del primo blocco della prossima estensione. In alcuni sistemi il proprietario del file può impostare

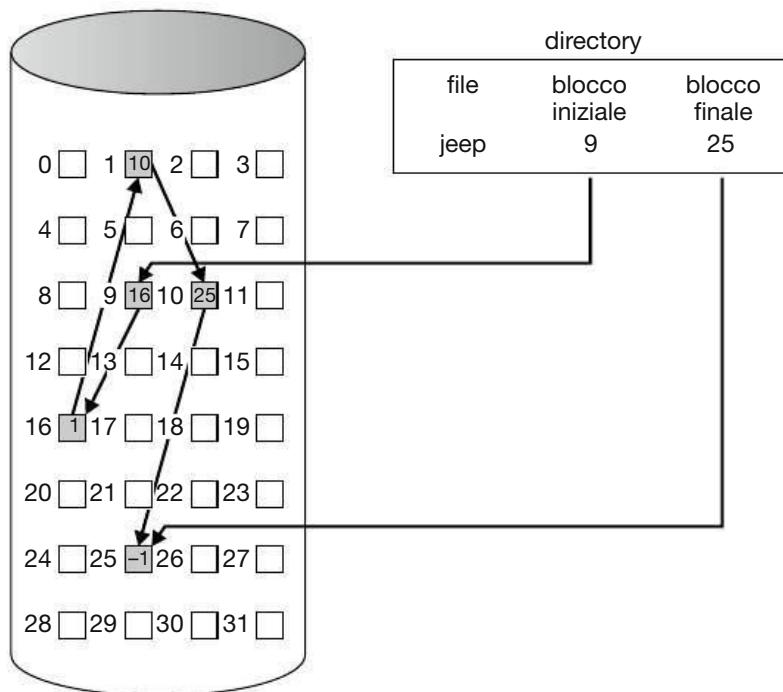


Figura 12.6 Allocazione concatenata dello spazio dei dischi.

la dimensione dell'estensione, ma tale possibilità, se l'indicazione è imprecisa, può causare inefficienze. La frammentazione interna può ancora essere un problema se le estensioni sono troppo grandi; si possono presentare problemi dovuti alla frammentazione esterna quando si assegnano e si rilasciano estensioni di dimensione variabile. Il file system commerciale Veritas impiega le estensioni per ottimizzare le prestazioni; Veritas è un sostituto ad alte prestazioni dell'ordinario UFS di UNIX.

12.4.2 Allocazione concatenata

L'**allocazione concatenata** risolve tutti i problemi dell'allocazione contigua. Con questo tipo di allocazione, infatti, ogni file è composto da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Per esempio, un file di cinque blocchi può cominciare dal blocco 9, continuare al blocco 16, quindi al blocco 1, al blocco 10 e infine terminare al blocco 25 (Figura 12.6). Ogni blocco contiene un puntatore al blocco successivo. Questi puntatori non sono disponibili all'utente, quindi se ogni blocco è formato di 512 byte e un indirizzo del disco (il puntatore) richiede 4 byte, l'utente vede blocchi di 508 byte.

Per creare un nuovo file si crea semplicemente un nuovo elemento nella directory. Con l'allocazione concatenata, ogni elemento della directory ha un puntatore al primo blocco del file. Questo puntatore s'inizializza a `null` (valore del puntatore di fine lista) a indicare un file vuoto; anche il campo della dimensione s'imposta a 0. Un'operazione di scrittura nel file determina la ricerca di un blocco libero attraverso il si-

stema di gestione dello spazio libero, la scrittura in tale blocco, e la concatenazione di tale blocco alla fine del file. Per leggere un file occorre semplicemente leggere i blocchi seguendo i puntatori da un blocco all'altro. Con l'allocazione concatenata non esiste frammentazione esterna e per soddisfare una richiesta si può usare qualsiasi blocco libero della lista. Inoltre non è necessario dichiarare la dimensione di un file al momento della sua creazione. Un file può continuare a crescere finché sono disponibili blocchi liberi, di conseguenza non è mai necessario compattare lo spazio del disco.

L'allocazione concatenata presenta comunque alcuni svantaggi. Il problema principale riguarda il fatto che può essere usata in modo efficiente solo per i file ad accesso sequenziale. Per trovare l' i -esimo blocco di un file occorre partire dall'inizio del file e seguire i puntatori finché non si raggiunge l' i -esimo blocco. Ogni accesso a un puntatore implica una lettura del disco, e talvolta un posizionamento della testina. Di conseguenza, per file il cui spazio è assegnato in modo concatenato, la funzione d'accesso diretto è inefficiente.

Un altro svantaggio dell'allocazione concatenata riguarda lo spazio richiesto per i puntatori. Se un puntatore richiede 4 byte di un blocco di 512 byte, allora lo 0,78 per cento del disco è usato per i puntatori anziché per le informazioni: ogni file richiede un po' più spazio di quanto ne richiederebbe altrimenti.

La soluzione più comune a questo problema consiste nel riunire un certo numero di blocchi contigui in **cluster** (gruppi di blocchi), e nell'allocare i cluster anziché i blocchi. Per esempio, il file system può definire cluster di 4 blocchi e operare nel disco soltanto per unità di cluster. Così i puntatori usano una percentuale molto più piccola dello spazio del disco. Questo metodo permette che la corrispondenza tra blocchi logici e blocchi fisici rimanga semplice, ma migliora il throughput del disco poiché si hanno meno posizionamenti della testina, inoltre diminuisce lo spazio necessario per l'allocazione dei blocchi e la gestione della lista dei blocchi liberi. Il costo di questo metodo è dato da un incremento della frammentazione interna, poiché se un cluster è parzialmente pieno si spreca più spazio di quanto se ne sprecerebbe con un solo blocco parzialmente pieno. I cluster si possono usare per ottimizzare l'accesso ai dischi in molti altri algoritmi, quindi s'impiegano nella maggior parte dei file system.

Un altro problema dell'allocazione concatenata riguarda l'affidabilità. Poiché i file sono tenuti insieme da puntatori sparsi per tutto il disco, s'immagini che cosa accadrebbe se un puntatore andasse perduto o danneggiato. Un errore di programmazione del sistema operativo oppure un errore hardware di un'unità a disco potrebbero causare la lettura di un puntatore errato. Questo errore, a sua volta, potrebbe causare il collegamento alla lista dei blocchi liberi oppure a un altro file. Una soluzione parziale a tale problema consiste nell'usare liste doppiamente concatenate oppure nel memorizzare il nome del file e il relativo numero di blocco in ogni blocco; questi schemi però richiedono un overhead ancora maggiore per ogni file.

Una variante importante del metodo di allocazione concatenata consiste nell'uso della **tavella di allocazione dei file** (*file allocation table*, FAT). Tale metodo di allo-

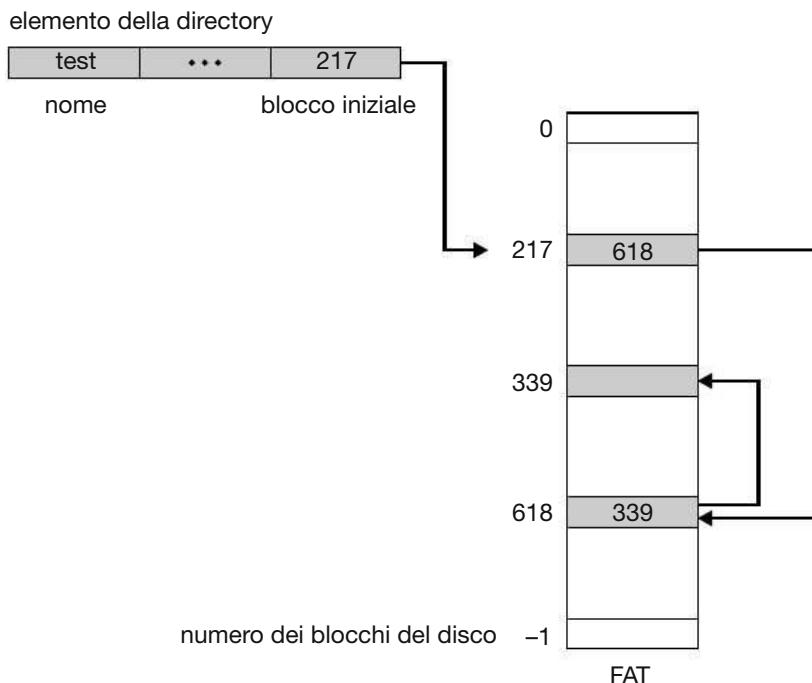


Figura 12.7 Tabella di allocazione dei file.

cazione dello spazio dei dischi, semplice ma efficiente, era usato nei sistemi operativi MS-DOS. Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascun volume; la FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; si usa essenzialmente come una lista concatenata. L'elemento di directory contiene il numero del primo blocco del file. L'elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero del blocco successivo del file. Questa catena continua fino all'ultimo blocco, che ha come elemento della tabella un valore speciale di fine del file. I blocchi inutilizzati sono indicati nella tabella da un valore 0. L'allocazione di un nuovo blocco a un file richiede semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine del file precedente con l'indirizzo del nuovo blocco; lo 0 è quindi sostituito con il valore di fine del file. Un esempio esplicativo di tale metodo è dato dalla struttura della FAT della Figura 12.7, per un file formato dai blocchi 217, 618 e 339.

Lo schema di allocazione basato sulla FAT, se non si usa una cache, può causare un significativo numero di posizionamenti della testina. La testina del disco deve spostarsi all'inizio del volume per leggere la FAT e trovare la locazione del blocco in questione, quindi raggiungere la locazione del blocco stesso; nel caso peggiore sono necessari ambedue i movimenti per ciascun blocco. Un vantaggio è dato dall'ottimizzazione del tempo d'accesso diretto, poiché la testina del disco può trovare la locazione di ogni blocco leggendo le informazioni contenute nella FAT.

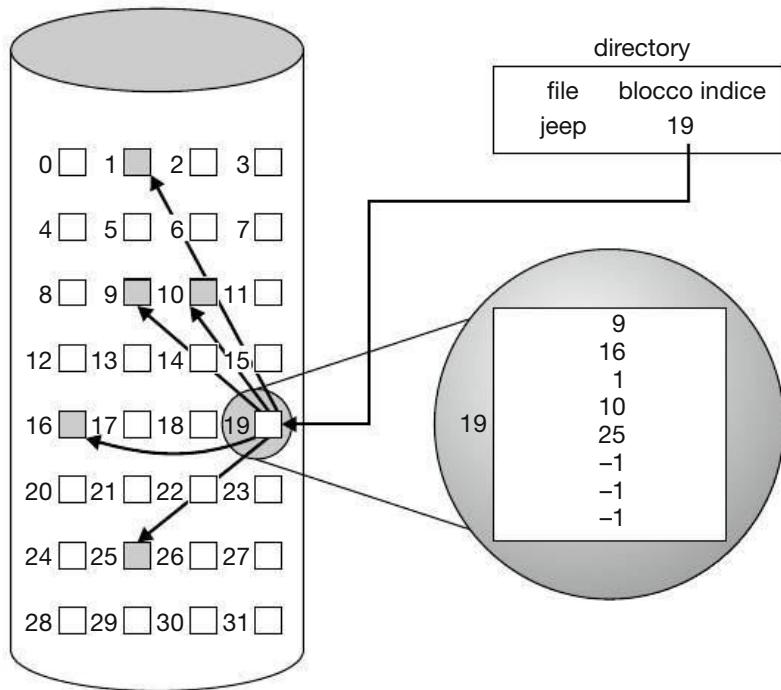


Figura 12.8 Allocazione indicizzata dello spazio dei dischi.

12.4.3 Allocazione indicizzata

L’allocazione concatenata risolve il problema della frammentazione esterna e quello della dichiarazione delle dimensioni dei file, presenti nell’allocazione contigua. Tuttavia, in mancanza di una FAT, l’allocazione concatenata non è in grado di sostenere un efficiente accesso diretto, poiché i puntatori ai blocchi sono sparsi, con i blocchi stessi, per tutto il disco e si devono recuperare in ordine. **L’allocazione indicizzata** risolve questo problema, raggruppando tutti i puntatori in una sola locazione: il **blocco indice**.

Ogni file ha il proprio blocco indice: si tratta di un array d’indirizzi di blocchi del disco. L’ i -esimo elemento del blocco indice punta all’ i -esimo blocco del file. La directory contiene l’indirizzo del blocco indice, com’è illustrato nella Figura 12.8. Per individuare e leggere l’ i -esimo blocco occorre usare il puntatore che si trova nell’ i -esimo elemento del blocco indice. Questo schema è simile a quello della paginazione descritto nel Paragrafo 8.5.

Una volta creato il file, tutti i puntatori del blocco indice sono impostati a null. Quando si scrive l’ i -esimo blocco per la prima volta, il gestore dei blocchi liberi fornisce un blocco; l’indirizzo di questo blocco viene inserito nell’ i -esimo elemento del blocco indice. Poiché ogni blocco libero del disco può soddisfare una richiesta di maggiore spazio, l’allocazione indicizzata consente l’accesso diretto senza soffrire di frammentazione esterna.

L’allocazione indicizzata soffre tuttavia di un overhead maggiore: lo spazio aggiuntivo richiesto dai puntatori del blocco indice è generalmente maggiore dello spa-

zio aggiuntivo necessario per l’allocazione concatenata. Si consideri il comune caso di un file con uno o due blocchi; con l’allocazione concatenata si perde il solo spazio di un puntatore per blocco, complessivamente uno o due puntatori; con l’allocazione indicizzata occorre allocare un intero blocco indice, anche se solo uno o due puntatori sono diversi da `null`.

Questo punto solleva la questione della dimensione del blocco indice. Ogni file deve avere un blocco indice, quindi è auspicabile che questo sia quanto più piccolo è possibile; ma se il blocco indice è troppo piccolo non può contenere un numero di puntatori sufficiente per un file di grandi dimensioni, quindi è necessario disporre di un meccanismo per gestire questa situazione. Fra i possibili meccanismi vi sono i seguenti.

- **Schema concatenato.** Un blocco indice è formato normalmente di un solo blocco di disco; perciò, ciascun blocco indice può essere letto e scritto esattamente con un’operazione. Per permettere la presenza di lunghi file è possibile collegare tra loro parecchi blocchi indice. Per esempio, un blocco indice può contenere una piccola intestazione in cui sono riportati il nome del file e l’insieme dei primi 100 indirizzi di blocchi del disco. L’indirizzo successivo, vale a dire l’ultima parola del blocco indice, è `null` (per un file piccolo) oppure è un puntatore a un altro blocco indice (per un file lungo).
- **Indice a più livelli.** Una variante della rappresentazione concatenata consiste nell’impiego di un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che, a loro volta, puntano ai blocchi dei file. Per accedere a un blocco, il sistema operativo usa l’indice di primo livello, con il quale individua il blocco indice di secondo livello, e con esso trova il blocco di dati richiesto. Questo metodo potrebbe continuare fino a un terzo o quarto livello, a seconda della massima dimensione desiderata del file. Con blocchi di 4096 byte si possono memorizzare 1024 puntatori di 4 byte in un blocco indice. Due livelli di indici consentono 1.048.576 blocchi di dati, che permettono di avere file sino a 4 GB.
- **Schema combinato.** Un’altra possibilità, utilizzata nei sistemi basati su UNIX, consistente nel tenere i primi 15 puntatori del blocco indice nell’*inode* del file. I primi 12 di questi 15 puntatori puntano a **blocchi diretti**, cioè contengono direttamente gli indirizzi di blocchi contenenti dati del file. Quindi, i dati per piccoli file (non più di 12 blocchi) non richiedono un blocco indice distinto. Se la dimensione dei blocchi è di 4 KB, è possibile accedere direttamente fino a 48 KB di dati. Gli altri tre puntatori puntano a **blocchi indiretti**. Il primo è un puntatore a un **blocco indiretto singolo**; si tratta di un blocco indice che non contiene dati, ma indirizzi di blocchi che contengono dati. Il secondo è un puntatore a un **blocco indiretto doppio** contenente l’indirizzo di un blocco che a sua volta contiene gli indirizzi di blocchi contenenti puntatori agli effettivi blocchi di dati. Il terzo è un puntatore a un **blocco indiretto triplo**. Un *inode* UNIX è mostrato nella Figura 12.9.

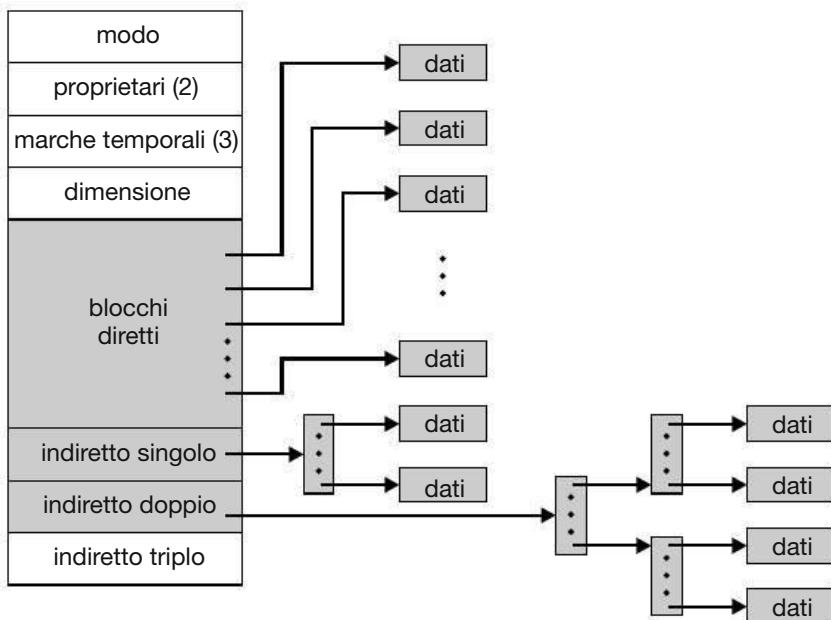


Figura 12.9 Inode di UNIX.

Con questo metodo il numero dei blocchi che si può allocare a un file supera la quantità di spazio che possono indirizzare i puntatori a file di 4 byte usati da molti sistemi operativi. Un puntatore a file di 32 bit consente di arrivare a soli 2^{32} byte, 4 GB. Molte versioni di UNIX e Linux ora gestiscono puntatori a file sino a 64 bit. Puntatori di questa dimensione permettono di avere file e file system di dimensioni dell'ordine degli exabyte (2^{60} byte). Il file system ZFS supporta puntatori a 128 bit.

Gli schemi d'allocazione indicizzata soffrono di alcuni dei problemi di prestazioni dell'allocazione concatenata. In particolare, i blocchi indice si possono caricare in memoria, ma i blocchi dei dati possono essere sparsi per un'intero volume.

12.4.4 Prestazioni

I metodi d'allocazione presentati hanno diversi livelli di efficienza di memorizzazione e differenti tempi d'accesso ai blocchi di dati; entrambi i fattori sono importanti nella scelta del metodo o dei metodi d'allocazione più adatti da impiegare in un sistema operativo.

Prima di scegliere un metodo di allocazione, è necessario determinare il modo in cui si usano i sistemi: un sistema con una prevalenza di accessi sequenziali farà uso di un metodo differente da quello di un sistema con una prevalenza di accessi diretti.

Per qualsiasi tipo d'accesso, l'allocazione contigua richiede un solo accesso per ottenere un blocco. Poiché è facile tenere l'indirizzo iniziale del file in memoria, si può calcolare immediatamente l'indirizzo del disco dell' i -esimo blocco, oppure del blocco successivo, e leggerlo direttamente.

Con l'allocazione concatenata si può tenere in memoria anche l'indirizzo del blocco successivo e leggerlo direttamente. Questo metodo è valido per l'accesso sequen-

ziale mentre, per quel che riguarda l’accesso diretto, un accesso all’ i -esimo blocco può richiedere i letture del disco. Questo spiega perché l’allocazione concatenata non si dovrebbe usare per un’applicazione che richiede accessi diretti.

Da tutto ciò segue che alcuni sistemi gestiscono i file ad accesso diretto usando l’allocazione contigua, e i file ad accesso sequenziale tramite l’allocazione concatenata. Per questi sistemi, il tipo d’accesso si deve dichiarare al momento della creazione del file. Un file creato per l’accesso sequenziale è un file concatenato e non si può usare per l’accesso diretto. Un file creato per l’accesso diretto è contiguo e consente entrambi i tipi d’accesso, ma bisogna dichiararne la lunghezza massima al momento della sua creazione. In questo caso, il sistema operativo deve avere strutture dati idonee e algoritmi capaci di gestire *entrambi* i metodi di allocazione. I file si possono convertire da un tipo all’altro creando un nuovo file del tipo desiderato, nel quale si copia il contenuto del vecchio file; si può quindi cancellare quest’ultimo e rinominare il nuovo file.

L’allocazione indicizzata è più complessa. Se il blocco indice è già in memoria, l’accesso può essere diretto. Tuttavia, per tenere il blocco indice in memoria occorre una quantità di spazio considerevole. Se questo spazio di memoria non è disponibile, occorre leggere prima il blocco indice e quindi il blocco di dati desiderato. Per un indice a due livelli possono essere necessarie due letture di blocco indice. Se un file è estremamente grande, per compiere l’accesso a un blocco che si trovi vicino alla fine del file, prima di leggere il blocco dei dati occorre leggere tutti i blocchi indice per seguire la catena dei puntatori. Quindi le prestazioni dell’allocazione indicizzata dipendono dalla struttura dell’indice, dalla dimensione del file e dalla posizione del blocco desiderato.

Alcuni sistemi combinano l’allocazione contigua con l’allocazione indicizzata, usando quella contigua per i file piccoli (fino a tre o quattro blocchi) e passando automaticamente a quella indicizzata per i file grandi. Poiché la maggior parte dei file sono piccoli, e in questo caso l’allocazione contigua è efficiente, le prestazioni medie possono risultare decisamente buone.

Sono possibili e si usano effettivamente anche molte altre ottimizzazioni. Data la disparità tra velocità della CPU e velocità dei dischi, non è irragionevole aggiungere al sistema operativo migliaia di istruzioni solo per risparmiare alcuni movimenti della testina. Con il passare del tempo tale disparità continua ad aumentare tanto che, per ottimizzare i movimenti della testina, si potranno ragionevolmente usare centinaia di migliaia di istruzioni.

12.5 Gestione dello spazio libero

Poiché la quantità di spazio dei dischi è limitata, è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere nuovi file, se possibile (i dischi ottici a una sola scrittura permettono una sola scrittura in qualsiasi settore e quindi il riutilizzo è fisicamente impossibile). Per tener traccia dello spazio libero in un disco, il sistema conserva una **lista dello spazio libero**; vi sono registrati tutti gli spazi *liberi*, cioè non allocati ad alcun file o directory. Per creare un file occorre cercare nella lista dello spazio

libero la quantità di spazio necessaria e assegnarla al nuovo file, quindi rimuovere questo spazio dalla lista. Quando si cancella un file, si aggiungono alla lista dello spazio libero i blocchi di disco a esso assegnati. A dispetto del suo nome, la lista dello spazio libero potrebbe non essere realizzata come una lista, come vedremo più avanti.

12.5.1 Vettore di bit

Spesso la lista dello spazio libero si realizza come una **mappa di bit**, o **vettore di bit**. Ogni blocco è rappresentato da un bit: se il blocco è libero, il bit è 1, se il blocco è assegnato il bit è 0.

Si consideri, per esempio, un disco dove i blocchi 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 sono liberi e gli altri sono allocati. La mappa di bit dello spazio libero è la seguente:

00111100111110001100000011100000...

I vantaggi principali che derivano da questo metodo sono la sua relativa semplicità ed efficienza nel trovare il primo blocco libero o n blocchi liberi consecutivi nel disco; in effetti, molti calcolatori forniscono istruzioni di manipolazione dei bit utilizzabili con efficacia a tale scopo. Una tecnica per individuare il primo blocco libero su un sistema che usa un vettore di bit per allocare spazio su disco è controllare in modo sequenziale ogni parola nella mappa di bit per verificare che il valore non sia 0, poiché una parola con valore 0 ha tutti i bit a 0 e rappresenta un insieme di blocchi assegnati. La prima parola non 0 viene scenduta alla ricerca del primo bit 1, che indica la locazione del primo blocco libero. Il numero del blocco è dato dalla seguente espressione:

$$(numero\ di\ bit\ per\ parola) \times (numero\ di\ parole\ di\ valore\ 0) + offset\ del\ primo\ bit\ 1.$$

Anche in questo caso le caratteristiche dell'hardware guidano le funzioni del sistema operativo. Sfortunatamente, i vettori di bit sono efficienti solo se tutto il vettore è mantenuto in memoria centrale, e viene scritto in memoria secondaria solo saltuariamente allo scopo di consentire eventuali operazioni di ripristino; è possibile tenere il vettore in memoria centrale solo se i dischi sono piccoli; tale soluzione non è applicabile ai dischi più grandi. Un disco di 1,3 GB con blocchi di 512 byte richiederebbe una mappa di bit di oltre 332 KB per tenere traccia dei suoi blocchi liberi, anche se il clustering a quattro blocchi riduce questo numero a 83 KB per disco. Un disco di 1 TB con blocchi di 4 KB richiede 256 MB per memorizzare la propria mappa di bit. Dato che la dimensione del disco è in costante crescita, i problemi legati ai vettori di bit continueranno ad aggravarsi.

12.5.2 Lista concatenata

Un altro metodo di gestione degli spazi liberi consiste nel collegarli tutti, tenere un puntatore al primo di questi in una speciale locazione del disco e caricarlo in memoria. Il primo blocco libero contiene un puntatore al successivo, e così via. Facciamo riferimento al nostro esempio precedente (Paragrafo 12.5.1) nel quale i blocchi 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 erano liberi, mentre i restanti blocchi era-

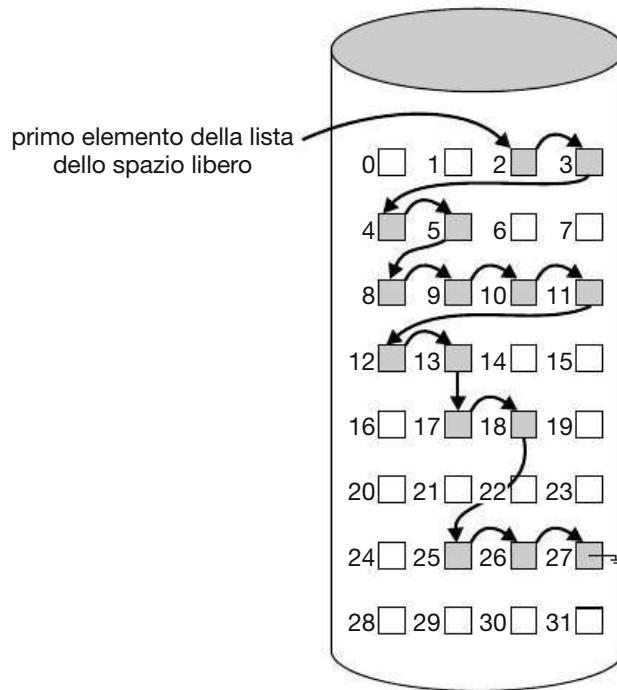


Figura 12.10 Lista concatenata degli spazi liberi su disco.

no allocati. In questa situazione dovremmo mantenere un puntatore al blocco 2, trattandosi del primo blocco libero. Il blocco 2 conterebbe un puntatore al blocco 3, che punterebbe al blocco 4, che punterebbe al blocco 5, il quale punterebbe a sua volta al blocco 8, e così via (Figura 12.10). Questo schema non è efficiente; per attraversare la lista è infatti necessario leggere ogni blocco, con un notevole tempo di I/O. Fortunatamente la necessità di attraversare la lista dello spazio libero non è frequente. Di solito il sistema operativo ha semplicemente bisogno di un blocco libero perché possa assegnarlo a un file, quindi si usa il primo blocco della lista. Il metodo che fa uso della FAT include la lista dei blocchi liberi nella struttura dati per l’allocazione; non è necessario un metodo separato.

12.5.3 Raggruppamento

Una possibile modifica del metodo della lista dello spazio libero prevede la memorizzazione degli indirizzi di n blocchi liberi nel primo di questi. I primi $n - 1$ di questi blocchi sono effettivamente liberi; l’ultimo blocco contiene gli indirizzi di altri n blocchi liberi, e così via. Con questo metodo, diversamente dall’ordinaria lista concatenata, è possibile trovare rapidamente gli indirizzi di un gran numero di blocchi liberi.

12.5.4 Conteggio

Un altro approccio sfrutta il fatto che, generalmente, più blocchi contigui si possono allocare o liberare contemporaneamente, soprattutto quando lo spazio viene allocato

usando l'algoritmo di allocazione contigua o attraverso l'uso di cluster. Quindi, anziché tenere una lista di n indirizzi liberi, è sufficiente tenere l'indirizzo del primo blocco libero e il numero n di blocchi liberi contigui che seguono il primo blocco. Ogni elemento della lista dello spazio libero è formato da un indirizzo del disco e un contatore. Anche se ogni elemento richiede più spazio di quanto ne richieda un semplice indirizzo del disco, se il contatore è generalmente maggiore di 1 la lista complessiva è più corta. Si noti che questo metodo, il tracciamento dello spazio libero, è simile al metodo delle estensioni per allocare i blocchi. Questi elementi possono essere memorizzati in un albero bilanciato invece che in una lista concatenata, in modo da permettere ricerche, inserzioni e cancellazioni efficienti.

12.5.5 Mappe di spazio

Il file system ZFS di Oracle (presente in Solaris e in altri sistemi operativi) è stato progettato per contenere un enorme numero di file, directory e persino di file system (in ZFS è possibile creare gerarchie di file system). Su queste scale, l'I/O dei metadati può avere un impatto notevole sulle prestazioni. Notate per esempio che, se la lista dello spazio libero è implementata come una mappa di bit, le mappe di bit devono essere modificate sia quando i blocchi vengono allocati sia quando vengono liberati. Liberare 1 GB di dati su un disco di 1 TB potrebbe comportare l'aggiornamento di migliaia di blocchi di mappe di bit, perché quei blocchi potrebbero essere sparpagliati sull'intero disco. Chiaramente, le strutture dati per un simile sistema potrebbero essere grandi e inefficienti.

Per la gestione dello spazio libero ZFS utilizza una combinazione di tecniche per controllare la dimensione delle strutture di dati e minimizzare l'I/O necessario a gestirle. Per prima cosa, ZFS crea **metalastre** (*metaslab*) per dividere lo spazio sul dispositivo in parti che abbiano una dimensione gestibile. Un dato volume potrebbe contenere centinaia di metalastre. A ogni metalastra è associata una mappa dello spazio. ZFS utilizza l'algoritmo di conteggio per memorizzare informazioni riguardanti i blocchi liberi. Piuttosto che scrivere sempre su disco i contatori, li registra utilizzando le tecniche dei file system con logging delle modifiche. La mappa dello spazio è un registro di tutte le attività relative ai blocchi (allocazione e liberazione), in ordine cronologico, in formato di conteggio. Quando ZFS decide di allocare o liberare spazio su una metalastra, carica la mappa dello spazio associata nella memoria, in una struttura ad albero bilanciato (per operazioni molto efficienti), indicizzato sull'base degli offset, e replica il log in tale struttura.

A quel punto la mappa dello spazio all'interno della memoria è un'accurata rappresentazione dello spazio allocato e libero nella metalastra. ZFS condensa la mappa il più possibile combinando blocchi liberi contigui in una singola voce. Infine la lista dello spazio libero viene aggiornata sul disco come parte delle operazioni orientate alle transazioni di ZFS. Durante la fase di raccolta e ordinamento possono verificarsi ancora richieste di blocchi, che vengono soddisfatte dal registro. In sostanza, il registro, insieme all'albero bilanciato, *costituiscono* la lista delle locazioni libere.

12.6 Efficienza e prestazioni

Dopo avere descritto le opzioni di allocazione dei blocchi e di gestione delle directory, è possibile considerare i loro effetti sulle prestazioni e l'efficienza d'uso dei dischi. I dischi tendono di solito a essere il principale collo di bottiglia per le prestazioni di un sistema, essendo i più lenti tra i componenti principali di un calcolatore. In questo paragrafo si considerano diverse tecniche utili per migliorare l'efficienza e le prestazioni della memoria secondaria.

12.6.1 Efficienza

L'uso efficiente di un disco dipende fortemente dagli algoritmi usati per l'allocatione del disco e la gestione delle directory. Per esempio, gli *inode* di UNIX sono preallocati su un volume. Quindi anche un disco “vuoto” impiega una certa percentuale del suo spazio per gli *inode*. D'altra parte, l'allocazione preventiva degli *inode* e la loro distribuzione nel volume migliorano le prestazioni del file system. Queste migliori prestazioni sono il risultato degli algoritmi di allocatione e di gestione dei blocchi liberi adottati da UNIX, i quali cercano di mantenere i blocchi di dati di un file vicini al blocco che ne contiene l'*inode* allo scopo di ridurre il tempo di ricerca.

Come ulteriore esempio, si consideri lo schema che fa uso dei cluster presentato nel Paragrafo 12.4, che migliora le prestazioni di ricerca e trasferimento per un file al costo di una maggiore frammentazione interna. Per ridurre questa frammentazione, il BSD UNIX varia la dimensione del cluster al crescere della dimensione del file. Cluster più grandi si usano dove possono essere riempiti, mentre cluster più piccoli si usano per file di piccole dimensioni e per l'ultimo cluster di un file.

Si devono tenere in considerazione anche il tipo di dati normalmente contenuti in un elemento della directory (o di un *inode*). Di solito si memorizza la *data dell'ultima scrittura* per fornire informazioni all'utente e per determinare se è necessario effettuare il backup del file. Alcuni sistemi mantengono anche la *data dell'ultimo accesso* per consentire all'utente di risalire all'ultima volta che un file è stato letto. Per mantenere questa informazione, ognqualvolta si legge un file, si deve aggiornare un campo della directory. Questa modifica richiede la lettura nella memoria del blocco, la modifica di una sua parte e la riscrittura del blocco nel disco, poiché sui dischi si può operare solamente per blocchi (o cluster). Quindi, ogni volta che si apre un file per la lettura, si deve leggere e scrivere anche l'elemento della directory a esso associato. Ciò può essere inefficiente per file cui si accede frequentemente, quindi nella fase della progettazione del file system è necessario confrontare i benefici con i costi in termini di prestazioni. In generale, è necessario considerare l'influenza sull'efficienza e sulle prestazioni di *ogni* informazione che si vuole associare a un file.

A titolo d'esempio, si consideri come l'efficienza sia influenzata dalle dimensioni dei puntatori usati per l'accesso ai dati. La maggior parte dei sistemi usa puntatori di 32 o 64 bit ovunque all'interno del sistema operativo. L'uso di puntatori di 32 bit limita la lunghezza di un file a 2^{32} byte (4 GB). I puntatori di 64 bit portano il limite a valori molto più grandi, ma i puntatori di 64 bit richiedono più spazio per la loro me-

morizzazione e di conseguenza fanno sì che i metodi di allocazione e di gestione dello spazio libero (liste concatenate, indici, e così via) impieghino più spazio.

Una delle difficoltà nella scelta della dimensione dei puntatori, o di qualsiasi altra dimensione di allocazione fissa all'interno di un sistema operativo, è la pianificazione degli effetti provocati dal cambiamento della tecnologia. Basti considerare che il primo IBM PC XT aveva un disco di 10 MB e che il file system dell'MS-DOS poteva gestire solamente 32 MB. (Ciascun elemento della FAT era di 12 bit e puntava a un cluster di 8 KB.) Con la crescita della capacità dei dischi, i dischi più grandi si dovevano suddividere in partizioni di 32 MB, poiché il file system non poteva tener traccia di blocchi disposti oltre i 32 MB. Quando divennero comuni dischi di capacità superiore ai 100 MB, si dovettero modificare le strutture dati e gli algoritmi usati dall'MS-DOS per gestire i dischi in modo da consentire file system più grandi. (La dimensione di ciascun elemento della FAT fu portata a 16 bit e più tardi a 32 bit.) La decisione iniziale fu presa per motivi di efficienza; tuttavia, con l'avvento della Versione 4 dell'MS-DOS milioni di utenti si trovarono a disagio quando dovettero passare ai nuovi, più grandi file system. Il file system ZFS di Solaris adotta puntatori di 128 bit, che in teoria non dovrebbero mai necessitare di un'estensione. (La minima massa di un dispositivo in grado di archiviare 2^{128} byte tramite memorizzazione al livello atomico è di circa 272 miliardi di tonnellate.)

Come altro esempio, si consideri l'evoluzione del sistema operativo Solaris. Originariamente molte strutture dati avevano lunghezza fissa ed erano assegnate all'avviamento del sistema. Queste strutture comprendevano la tabella dei processi e quella dei file aperti. Una volta riempite queste tabelle, non si potevano più creare nuovi processi o aprire nuovi file, sicché il sistema non riusciva nel proprio compito di fornire un servizio agli utenti. L'unico modo di aumentare le dimensioni di queste tabelle era la ricompilazione del kernel e il riavvio del sistema. Nelle versioni più recenti di Solaris quasi tutte le strutture dati del kernel si assegnano in modo dinamico, eliminando questi limiti artificiali alle prestazioni del sistema. Naturalmente, gli algoritmi che manipolano queste tabelle sono ora più complessi e il sistema operativo è un po' più lento dovendo allocare e rilasciare dinamicamente gli elementi delle tabelle, ma questo è il prezzo da pagare per la generalizzazione delle funzioni.

12.6.2 Prestazioni

Anche dopo aver scelto gli algoritmi fondamentali del file system le prestazioni possono essere migliorate in diversi modi. Come si osserverà nel Capitolo 13, alcuni controllori di unità a disco contengono una memoria locale sufficiente per la creazione di una **cache** interna al controllore abbastanza grande da memorizzare intere tracce del disco alla volta. Eseguito il posizionamento della testina, si legge la traccia nella cache del controllore del disco a partire dal settore sotto cui si viene a trovare la testina (riducendo il tempo di latenza). Il controllore trasferisce quindi al sistema operativo tutte le richieste di settori. Quando i blocchi sono trasferiti dal controllore del disco alla memoria centrale, il sistema operativo ha la possibilità di inserirli in una propria cache nella memoria centrale.

Alcuni sistemi riservano una sezione separata della memoria centrale come **buffer cache** (*cache del disco*), dove tenere i blocchi in previsione di un loro riutilizzo entro breve tempo. Altri sistemi impiegano una **cache delle pagine** per i file; si tratta di una soluzione che impiega tecniche di memoria virtuale per la gestione dei dati dei file come pagine anziché come blocchi del file system; l'uso degli indirizzi virtuali per effettuare il caching dei dati dei file è molto più efficiente dell'effettuare il caching dei blocchi fisici di disco, in quanto gli accessi avvengono attraverso la memoria virtuale e non attraverso il file system. Diversi sistemi, compreso Solaris, Linux e Windows usano le cache delle pagine, sia per le pagine relative ai processi sia per i dati dei file. Questo metodo è noto come **memoria virtuale unificata**.

Alcune versioni di UNIX e Linux prevedono la cosiddetta **buffer cache unificata**. Per illustrarne i vantaggi si considerino le due possibilità di aprire un file e accedervi: l'uso della mappatura dei file in memoria (Paragrafo 9.7) e l'uso delle ordinarie chiamate di sistema `read()` e `write()`. Senza una buffer cache unificata, si verifica una situazione simile a quella illustrata nella Figura 12.11. In questo caso, le chiamate di sistema `read()` e `write()` passano attraverso la buffer cache. La chiamata con I/O memory-mapped, tuttavia, richiede l'uso di due cache, la cache delle pagine e la buffer cache. Il memory-mapping prevede la lettura dei blocchi di disco dal file system e la loro memorizzazione nella buffer cache. Poiché il sistema di memoria virtuale non si interfaccia direttamente con la buffer cache, si deve copiare nella cache delle pagine il contenuto del file presente nella buffer cache. Questa situazione è nota come

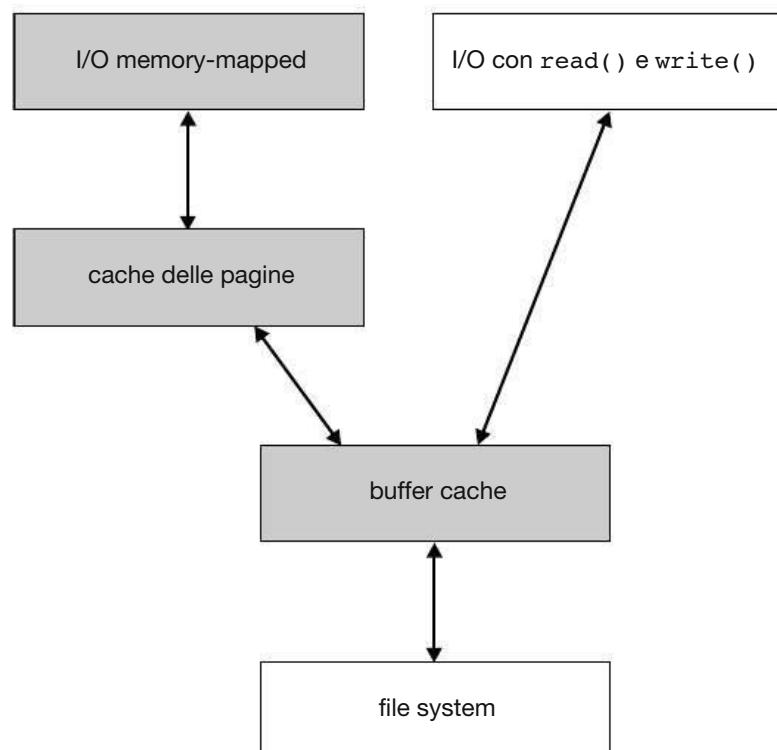


Figura 12.11 I/O senza una buffer cache unificata.

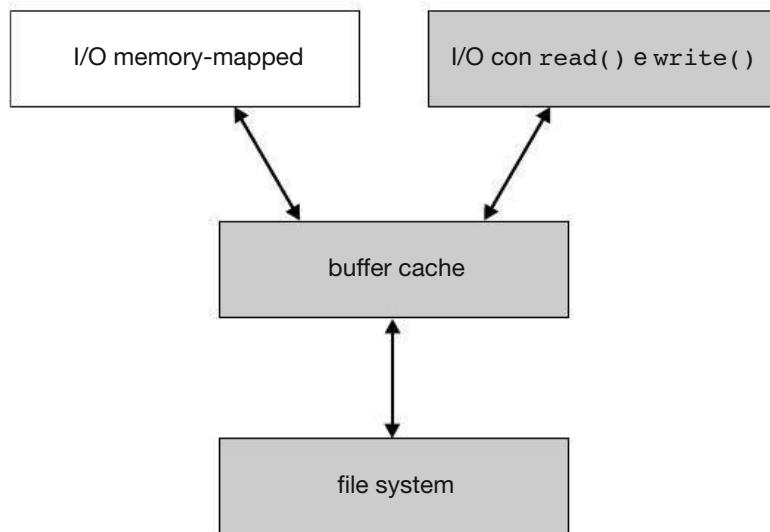


Figura 12.12 I/O con una buffer cache unificata.

double caching proprio perché i dati del file system richiedono un doppio passaggio di cache. Non solo ciò comporta uno spreco di memoria, ma anche uno spreco di cicli della CPU e di I/O dovuti a un ulteriore trasferimento di dati nella memoria del sistema. Inoltre, eventuali incoerenze tra le due cache possono generare errori nella memorizzazione dei dati nei file. Invece, con una buffer cache unificata, sia il memory-mapping sia le chiamate di sistema `read()` e `write()` usano la stessa cache delle pagine, con il vantaggio di evitare il double caching e di permettere al sistema di memoria virtuale di gestire dati del file system. La Figura 12.12 illustra l’uso della buffer cache unificata.

Indipendentemente dalla gestione delle cache per blocchi di disco oppure per pagine (o per entrambi), l’algoritmo LRU (Paragrafo 9.4.4) sembra essere un ragionevole algoritmo generale per la sostituzione dei blocchi o delle pagine. Tuttavia, l’evoluzione degli algoritmi di gestione delle cache delle pagine usati dal sistema operativo Solaris rivela le difficoltà nella scelta di un algoritmo ottimale. Tale sistema operativo permette ai processi e alla cache delle pagine di condividere la memoria inutilizzata; prima della versione 2.5.1, non si facevano distinzioni tra l’allocazione delle pagine a un processo o alla cache delle pagine, con la conseguenza che un sistema che eseguiva molte operazioni di I/O usava la maggior parte della memoria disponibile per la cache delle pagine. A causa dell’alta frequenza delle operazioni di I/O, quando la memoria libera diventa troppo esigua, il modulo di scansione delle pagine (Paragrafo 9.10.2) sottraeva pagine ai processi anziché alla cache delle pagine. In Solaris 2.6 e in Solaris 7 è stata realizzata in forma opzionale la tecnica di *paginazione con priorità*, secondo la quale il modulo di scansione delle pagine dà la priorità alle pagine dei processi rispetto a quelle della cache delle pagine. Il sistema operativo Solaris 8 ha applicato un limite prefissato alle pagine dei processi e alla cache delle pagine per il file system, impedendo a ciascun meccanismo di sottrarre totalmente la memoria

all’altro. Con Solaris 9 e 10 sono stati nuovamente modificati gli algoritmi per migliorare l’uso della memoria e contenere il fenomeno del thrashing.

Un altro aspetto che può influenzare le prestazioni di I/O è l’utilizzo di scritture sincrone o asincrone nel file system. Le **scritture sincrone** avvengono nell’ordine in cui le riceve il sottosistema per la gestione del disco e non subiscono la memorizzazione transitoria. Quindi la procedura chiamante prima di proseguire deve attendere che i dati raggiungano l’unità disco. Nelle **scritture asincrone** si memorizzano i dati nella cache e si restituisce immediatamente il controllo alla procedura chiamante. Le scritture sono per lo più asincrone, anche se le scritture dei metadati, tra le altre, possono essere sincrone. I sistemi operativi spesso includono un flag nella chiamata di sistema `open()` per permettere a un processo di richiedere che le operazioni di scrittura si eseguano in modo sincrono. I sistemi di gestione delle basi di dati per esempio usano questa funzione per realizzare le transazioni atomiche, in modo da assicurare che i dati raggiungano la memoria stabile nell’ordine richiesto.

Alcuni sistemi ottimizzano la cache delle pagine adottando differenti algoritmi di sostituzione a seconda del tipo d’accesso ai file. Le pagine relative a un file da leggere o scrivere in modo sequenziale non si dovrebbero sostituire nell’ordine LRU, infatti la pagina usata più di recente sarà usata nuovamente per ultima, o forse mai. Gli accessi sequenziali si potrebbero invece ottimizzare con tecniche note come rilascio all’indietro e lettura anticipata. Il **rilascio all’indietro** (*free-behind*) rimuove una pagina dal buffer non appena si verifica una richiesta della pagina successiva; le pagine precedenti con tutta probabilità non saranno più usate e quindi sprecano spazio nel buffer. Con la **lettura anticipata** (*read-ahead*) si leggono e si mettono nella cache la pagina richiesta e parecchie pagine successive: è probabile che queste pagine siano richieste una volta terminata l’elaborazione della pagina corrente. Il recupero di questi dati dal disco con un unico trasferimento e la memorizzazione nella cache consentono di risparmiare una quantità di tempo considerevole. Si noti che la presenza nel controllore di una cache per le tracce non elimina la necessità di adottare la tecnica di lettura anticipata in un sistema multiprogrammato, infatti, a causa dell’elevata latenza e dell’overhead determinato dai tanti piccoli trasferimenti dalla cache per le tracce alla memoria centrale, il ricorso alla *lettura anticipata* resta vantaggioso.

La cache delle pagine, il file system e i driver del disco interagiscono in modi interessanti. Quando i dati vengono scritti in un file su disco, le pagine sono memorizzate nella cache e il driver del disco ordina la propria coda di dati in base all’indirizzo sul disco. Queste due azioni consentono al driver del disco di minimizzare gli spostamenti della testina del disco e fanno sì che la scrittura dei dati rispetti i tempi ottimali per la rotazione del disco. A meno che le scritture richieste siano sincrone, un processo, per scrivere sul disco, scrive semplicemente nella cache e il sistema trasferisce i dati su disco, in maniera asincrona, quando lo ritiene opportuno. Dal punto di vista del processo utente, le scritture sembreranno estremamente rapide. Durante la lettura, il sistema di I/O per i blocchi su disco effettua qualche lettura anticipata; ma, in realtà, le scritture sono molto più asincrone delle letture. Per grandi quantità di dati, quindi, la scrittura su disco tramite il file system è spesso più veloce della lettura, contrariamente a ciò che suggerirebbe l’intuizione.

12.7 Ripristino

Poiché i file e le directory sono mantenuti sia in memoria centrale sia nei dischi, è necessario aver cura di assicurare che il verificarsi di un malfunzionamento nel sistema non comporti la perdita di dati o la loro incoerenza. In questo paragrafo tratteremo questo problema e vedremo anche come un sistema può essere ripristinato in seguito a un malfunzionamento.

Una caduta del sistema può causare incoerenze tra le strutture dati del file system su disco, come le strutture delle directory, i puntatori ai blocchi liberi e i puntatori agli FCB liberi. Molti file system applicano delle modifiche direttamente a queste strutture. Operazioni comuni come la creazione di un file possono comportare molti cambiamenti strutturali all'interno del file system di un disco. Le strutture delle directory vengono modificate, gli FCB e i blocchi di dati allocati e i contatori degli elementi liberi per tutti questi blocchi diminuiti. Quando queste modifiche sono interrotte da una caduta del sistema, ne possono derivare incoerenze tra le strutture. Per esempio, il contatore degli FCB liberi potrebbe indicare che un FCB è stato allocato, ma la struttura della directory potrebbe non avere un puntatore a quel FCB. L'utilizzo della cache che i sistemi operativi adottano per ottimizzare le prestazioni di I/O aggrava questo problema. Alcuni cambiamenti potrebbero andare direttamente sul disco, mentre altri possono essere memorizzati nella cache. Se i cambiamenti nella cache non raggiungono il disco prima che si verifichi una caduta, è possibile che la situazione peggiori ulteriormente.

Inoltre, anche i bachi nell'implementazione del file system, i controllori del disco, e persino le applicazioni utente possono indurre incoerenze nel file system. I file system hanno svariati metodi per affrontare queste circostanze, a seconda delle strutture dati e degli algoritmi che utilizzano. Ci occuperemo adesso di questi temi.

12.7.1 Verifica della coerenza

Quale che sia la causa degli errori, un file system deve prima scoprire i problemi e poi correggerli. Per scoprire gli errori vengono esaminati tutti i metadati su ogni file system per verificare la coerenza del sistema. Sfortunatamente, questo procedimento può richiedere diversi minuti, o addirittura delle ore, e dovrebbe avvenire tutte le volte che il sistema si avvia. In alternativa, un file system può registrare il suo stato all'interno dei metadati. All'inizio di ogni serie di modifiche dei metadati è impostato un bit di stato per indicare che i metadati sono in stato di modifica. Se tutti gli aggiornamenti dei metadati si completano con successo, il file system può azzerare quel bit. Se tuttavia il bit dello stato rimane settato, entra in funzione un verificatore della coerenza.

Il **verificatore della coerenza** – un programma di sistema come `fsck` in UNIX – confronta i dati delle directory con i blocchi dati dei dischi, tentando di correggere ogni incoerenza. Gli algoritmi di allocazione e di gestione dello spazio libero determinano il genere di problemi che questo programma può riconoscere e con quanto successo riuscirà a risolverli. Per esempio, se si adotta uno schema di allocazione concatenata con un puntatore da ciascun blocco al successivo, si può ricostruire l'in-

tero file e ricreare il corrispondente elemento nella directory analizzando i blocchi di dati. Al contrario, la perdita di un elemento di una directory in un sistema ad allocazione indicizzata potrebbe essere disastrosa, poiché ogni blocco di dati non contiene alcuna informazione sugli altri blocchi di dati. Per questo motivo, UNIX gestisce tramite cache gli elementi delle directory per le letture, mentre qualsiasi operazione di scrittura che produca l'allocazione di spazio, o altre modifiche dei metadati, è svolta in modo sincrono, prima della scrittura dei corrispondenti blocchi di dati. Naturalmente possono ancora insorgere dei problemi se una scrittura sincrona viene interrotta da una caduta del sistema.

12.7.2 File system con log delle modifiche

Spesso nell'informatica si adottano algoritmi e tecnologie anche in aree diverse da quelle per le quali sono stati progettati. È il caso degli algoritmi per il ripristino, utilizzati per le basi di dati, basati sulla registrazione (*log*) delle modifiche. Questi algoritmi sono stati applicati con successo al problema della verifica della coerenza, realizzando i **file system orientati alle transazioni** e basati sul **log delle modifiche** (*log-based transaction-oriented file system*), noti anche come **file system annotati** (*journaling file system*).

Si osservi che l'approccio alla verifica della coerenza esaminato precedentemente permette in sostanza alle strutture di esibire incoerenze successivamente corrette nel ripristino. Questa strategia comporta tuttavia vari problemi. Per esempio, l'incoerenza potrebbe rivelarsi irreparabile: il verificatore della coerenza potrebbe non essere in grado di ripristinare le strutture, con una conseguente perdita di file o addirittura di intere directory. Ancora, il verificatore della coerenza potrebbe richiedere l'intervento umano per risolvere i conflitti, il che causa inconvenienti: in mancanza di assistenza da parte di qualcuno, il sistema potrebbe rimanere inutilizzabile finché una persona non gli indichi come procedere. Infine, il verificatore della coerenza sottrae risorse al sistema: per controllare terabyte di dati possono essere necessarie molte ore.

La soluzione a questo problema consiste nell'applicare agli aggiornamenti dei metadati del file system metodi di ripristino basati sul log delle modifiche. Sia il file system NTFS sia il Veritas usano questo metodo, che è anche incluso nelle recenti versioni di UFS su Solaris. In realtà, sta diventando un metodo comune in molti sistemi operativi.

Fondamentalmente, tutte le modifiche dei metadati si registrano in modo sequenziale in un file di log. Ogni insieme di operazioni che esegue uno specifico compito si chiama **transazione**. Una volta che le modifiche sono riportate nel file di log, le operazioni si considerano portate a termine con successo (*committed*) e la chiamata di sistema può restituire il controllo al processo utente, permettendogli di proseguire la sua esecuzione. Nel frattempo, si applicano alle effettive strutture del file system le operazioni scritte nel log, e man mano che si eseguono si aggiorna un puntatore che indica quali azioni sono state completate e quali sono ancora incomplete. Quando un'intera transazione è stata completata, se ne rimuovono le annotazioni dal log, che è in realtà un buffer circolare. I **buffer circolari** scrivono fino alla fine dello spazio

disponibile, e poi ricominciano dall'inizio, sovrascrivendo i vecchi contenuti. Naturalmente, si devono prendere delle misure per evitare che dati non ancora salvati siano sovrascritti. Il log si può mantenere in una sezione separata del file system, o anche in un disco separato. È più efficiente, anche se è più complesso, averlo sotto testine di lettura e scrittura separate, poiché si riducono le situazioni di contesa della testina e i tempi di ricerca (*seek time*).

Se si verifica una caduta del sistema, nel log ci potranno essere zero o più transazioni. Le transazioni presenti non sono mai state ultimate nel file system, anche se il sistema operativo le definisce portate a termine con successo, e quindi si devono completare. Le transazioni si possono eseguire a partire dalla posizione corrente del puntatore fino al completamento, in modo che le strutture del file system rimangano coerenti. L'unico problema che si può presentare è il caso in cui una transazione sia fallita (*aborted*), cioè non sia stata dichiarata terminata con successo prima della caduta del sistema. In questo caso, si devono annullare tutti i cambiamenti che erano stati applicati al file system dalla transazione, mantenendo anche in questo caso la coerenza del file system. Questo ripristino è tutto ciò che è necessario fare dopo una caduta del sistema, eliminando tutti i problemi della verifica della coerenza.

Un vantaggio indiretto dell'uso del logging degli aggiornamenti dei metadati dei dischi è che gli aggiornamenti sono molto più rapidi di quando si applicano direttamente alle strutture dati nei dischi. La ragione di questo miglioramento sta nel vantaggio, dal punto di vista delle prestazioni, dell'I/O ad accesso sequenziale rispetto a quello ad accesso diretto. Le onerose operazioni di scrittura sincrona ad accesso diretto sui metadati vengono sostituite con molto meno gravose operazioni di scrittura sequenziali sincrone nell'area di log di un file system annotato. I cambiamenti determinati da quelle operazioni si riportano successivamente in modo asincrono nelle strutture appropriate nei dischi attraverso operazioni di scrittura ad accesso diretto. Il risultato complessivo è un significativo guadagno in termini di prestazioni per le operazioni orientate ai metadati, come la creazione e la cancellazione dei file.

12.7.3 Altre soluzioni

Un'altra alternativa alla verifica della coerenza è impiegata dal file system WAFL di Network Appliance e da ZFS di Solaris. Entrambi i sistemi non sovrascrivono mai i blocchi con nuovi dati; al contrario, una transazione scrive tutti i cambiamenti di dati e metadati su nuovi blocchi. Quando la transazione viene completata, le strutture dei metadati che puntavano alla vecchia versione di questi blocchi sono aggiornate in modo da puntare ai nuovi. Il file system può quindi rimuovere i vecchi puntatori e i vecchi blocchi per renderli nuovamente disponibili. Se i vecchi puntatori e i vecchi blocchi vengono mantenuti, viene creata una **snapshot** (*istantanea*), cioè un'immagine del file system prima dell'ultimo aggiornamento. Questa soluzione non dovrebbe richiedere una verifica della coerenza se l'aggiornamento del puntatore è eseguito atomicamente. Ciononostante, il WAFL possiede comunque un controllore della coerenza, perché alcuni tipi di malfunzionamento possono ancora causare un errore nei metadati (per dettagli sul sistema WAFL si veda il Paragrafo 12.9).

Lo ZFS di Sun adotta un approccio ancora più innovativo alla coerenza del disco. ZFS non sovrascrive mai i blocchi, come WAFL, ma è in grado di andare oltre fornendo il checksumming di tutti i metadati e i blocchi di dati. Questa soluzione (se combinata con RAID) assicura che i dati siano sempre corretti. ZFS non possiede quindi un controllore della coerenza. (Maggiori dettagli su ZFS sono presenti nel Paragrafo 10.7.6.)

12.7.4 Copie di riserva e recupero dei dati

I dischi magnetici sono soggetti a guasti, ed è necessario preoccuparsene e provvedere affinché in tal caso i dati non vadano persi definitivamente. A questo scopo si possono usare programmi di sistema che consentano di fare delle **copie di riserva** (*backup*) dei dati residenti nei dischi su altri dispositivi di memorizzazione, come nastri magnetici o hard disk supplementari. Il ripristino della situazione antecedente la perdita di un singolo file, o del contenuto di un intero disco, richiederà il **recupero** (*restore*) dei dati dalle copie di backup.

Al fine di ridurre al minimo la quantità di dati da copiare, è possibile sfruttare le informazioni contenute nell'elemento della directory associato a ogni file. Per esempio, se il programma di creazione delle copie di riserva sa quando è stata eseguita l'ultimo backup di un file, e se la data di ultima scrittura di quel file, registrata nella directory, indica che il file da quel momento non ha subito variazioni, non sarà necessario copiare nuovamente il file. Quella che segue è una tipica sequenza di gestione dei backup.

- **Giorno 1.** Copiatura nel supporto di backup di tutti i file contenuti nel disco; questo è detto **backup completo**.
- **Giorno 2.** Copiatura su un altro supporto di tutti i file modificati dal Giorno 1; questo è un **backup incrementale**.
- **Giorno 3.** Copiatura su un altro supporto di tutti i file modificati dal Giorno 2.
 -
 -
 -
- **Giorno *n*.** Copiatura su un altro supporto di tutti i file modificati dal Giorno *n* – 1. Ritorno al Giorno 1.

Il nuovo ciclo può scrivere le nuove copie riutilizzando il precedente insieme di supporti di backup, oppure in un nuovo insieme.

Utilizzando questo metodo si ha la possibilità di recuperare il contenuto dell'intero disco iniziando le operazioni di recupero dalla copia di backup completa e proseguendo con i backup incrementalni. Naturalmente, più grande è *n*, maggiore sarà il numero di nastri o dischi da leggere per un completo recupero. Un ulteriore vantaggio di questo ciclo di creazione di copie di riserva è la possibilità di recuperare qualsiasi file accidentalmente cancellato durante il ciclo, recuperandolo dalle copie del giorno precedente. La lunghezza del ciclo è un compromesso tra la quantità di supporti necessari per i backup e il numero di giorni coperti da un'operazione di recupero.

Una possibilità per diminuire la quantità di nastri che è necessario leggere per portare a termine il ripristino è di eseguire inizialmente un backup completo, per poi eseguire ogni giorno il backup di tutti i file modificati dal giorno del backup completo. In questo modo, il ripristino può fondarsi sull'ultimo backup incrementale, insieme all'ultimo backup completo, senza necessità di altri backup incrementalni. Qui il compromesso da tenere a mente è che il numero di file modificati aumenta giornalmente: ogni nuovo backup incrementale, quindi, richiede più spazio.

Un utente potrebbe accorgersi dopo molto tempo che si è perso o si è danneggiato un particolare file. Per questa ragione si è soliti pianificare di tanto in tanto un backup completo che sarà conservato in modo permanente. È opportuno conservare tali backup permanenti lontano dalle copie ordinarie per proteggerle dai vari pericoli, come un incendio che può distruggere il calcolatore e tutte le copie di backup. Se il ciclo di creazione delle copie di backup prevede il reimpiego dei mezzi che le contengono, è anche necessario aver cura di non usarli troppe volte: se dovessero danneggiarsi per l'uso, potrebbe essere impossibile ripristinare i dati in essi contenuti.

12.8 NFS

I file system di rete sono ampiamente diffusi; in genere si integrano con l'intera struttura della directory e l'interfaccia del sistema client. L'NFS è un buon esempio di file system di rete, client-server, ampiamente usato e ben realizzato. In questo paragrafo lo utilizzeremo per esplorare i dettagli di implementazione dei file system di rete.

L'NFS è sia una realizzazione sia una specifica di un sistema software per l'accesso a file remoti attraverso LAN, o anche WAN. Fa parte dell'ONC+, adottato dalla maggior parte dei fornitori di sistemi operativi UNIX e in alcuni sistemi operativi per PC. La versione qui descritta fa parte del sistema operativo Solaris, che è a sua volta una versione modificata dello UNIX SVR4. Esso usa il protocollo TCP/IP o UDP/IP (a seconda della rete di comunicazione). Nella nostra descrizione dell'NFS, specifica e implementazione si accavallano: per ogni descrizione dettagliata si fa riferimento alla versione realizzata per Solaris; mentre ogni descrizione generale vale anche per la sua specifica.

Ci sono numerose versioni di NFS, l'ultima delle quali è la Versione 4. Esamineremo qui la Versione 3, attualmente la più utilizzata.

12.8.1 Generalità

Nel contesto dell'NFS si considera un insieme di stazioni di lavoro interconnesse come un insieme di calcolatori indipendenti con file system indipendenti. Lo scopo è quello di permettere un certo grado di condivisione tra i file system, su richiesta esplicita, in modo trasparente. La condivisione è basata su una relazione client-server. Un calcolatore può essere, come spesso accade, sia un client sia un server. La condivisione è ammessa tra ogni coppia di calcolatori. Per assicurare l'indipendenza dei calcolatori, la condivisione di un file system remoto ha effetto esclusivamente sul calcolatore client.

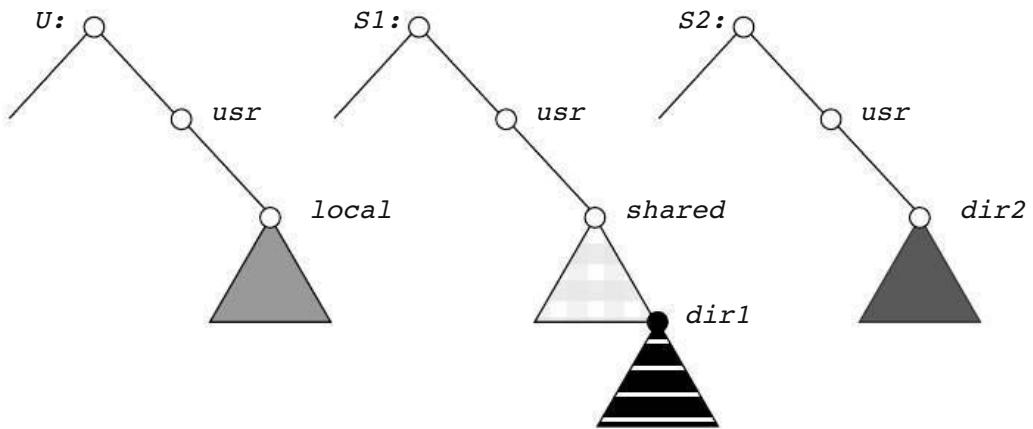


Figura 12.13 Tre file system indipendenti.

Affinché una directory remota sia accessibile in modo trasparente da un calcolatore particolare, per esempio C_1 , un client di quel calcolatore deve prima eseguire un’operazione di montaggio. La semantica dell’operazione richiede di montare una directory remota in corrispondenza di una directory di un file system locale. Una volta completata l’operazione di montaggio, la directory montata assume l’aspetto di un sottoalbero integrato nel file system locale, e sostituisce il sottoalbero che discende dalla directory locale; questa, a sua volta, rappresenta la radice della directory appena montata. La directory remota si specifica come argomento dell’operazione di montaggio in modo esplicito: si deve fornire la locazione (nome del calcolatore) della directory remota. Tuttavia, da questo momento in poi gli utenti del calcolatore C_1 possono accedere ai file della directory remota in modo del tutto trasparente.

Per illustrare l’operazione di montaggio, si consideri il file system rappresentato nella Figura 12.13, in cui i triangoli rappresentano i sottoalberi di directory di interesse. La figura illustra tre file system di calcolatori indipendenti chiamati U , S_1 e S_2 . A questo punto, in ogni calcolatore si può accedere solo a file locali. Nella Figura 12.14(a) mostra l’effetto del montaggio di $S_1:/usr/shared$ in $U:/usr/local$. In questa figura è illustrata la visione che gli utenti di U hanno del loro file system. Occorre osservare che, una volta completato il montaggio, essi possono accedere a qualsiasi file che si trovi nella directory **dir1**, usando il prefisso $/usr/local/dir1$. La directory originale $/usr/local$ di quel calcolatore non è più visibile.

Potenzialmente ogni file system o ogni directory in un file system, nel rispetto dei diritti d’accesso, si possono montare in modo remoto sopra qualsiasi directory locale. Le stazioni di lavoro prive di dischi possono persino montare i propri file system sulla radice prelevandoli dai server. In alcune versioni dell’NFS sono permessi anche i montaggi a cascata; ciò significa che un file system si può montare in corrispondenza di un altro file system non locale, ma già montato in modo remoto. Un calcolatore vede tuttavia gli effetti dei soli montaggi da esso richiesti. Montando un file system remoto, il client non acquisisce l’accesso ai file system che, eventualmente, erano montati sopra questo; così, il meccanismo di montaggio non ha la proprietà transitiva.

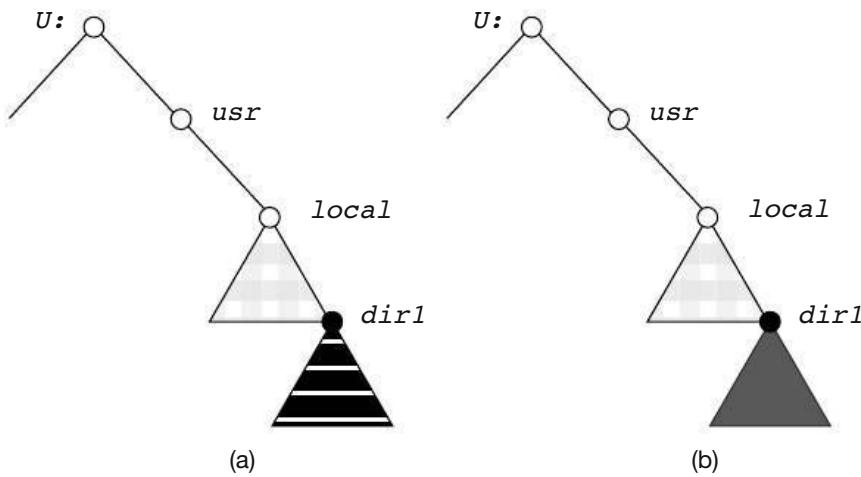


Figura 12.14 Montaggio nell’NFS; (a) montaggio; (b) montaggi a cascata.

Nella Figura 12.14(b) sono riportati i montaggi a cascata. Nella figura è riportato il risultato del montaggio di $S_2 : /usr/dir2$ in $U : /usr/local/dir1$, che era già stato montato in modo remoto da S_1 . Gli utenti di U possono accedere ai file di $dir2$ usando il prefisso $/usr/local/dir1$. Se si monta un file system condiviso in corrispondenza delle directory iniziali degli utenti di tutti i calcolatori di una rete, un utente può aprire una sessione in qualsiasi stazione di lavoro e prelevare il proprio ambiente di lavoro iniziale. Questa proprietà permette la **mobilità dell’utente**.

Uno degli scopi nella progettazione dell’NFS era quello di operare in un ambiente eterogeneo di calcolatori, sistemi operativi e architetture di rete. La definizione dell’NFS è indipendente da questi aspetti. Questa indipendenza si ottiene usando primitive RPC costruite su un protocollo di rappresentazione esterna dei dati (*external data representation*, XDR), usato tra due interfacce indipendenti dall’implementazione. Quindi, se il sistema è formato da calcolatori e file system eterogenei adeguatamente interfacciati all’NFS, si possono montare file system di diversi tipi, sia localmente sia in modo remoto.

La definizione dell’NFS distingue tra i servizi offerti da un meccanismo di montaggio e gli effettivi servizi d’accesso ai file remoti. Di conseguenza, per questi servizi si definiscono due protocolli distinti: un protocollo di montaggio e un protocollo per gli accessi ai file remoti, il **protocollo NFS**. I protocolli sono definiti come insiemi di RPC. Queste RPC sono gli elementi di base usati per realizzare, in modo trasparente, l’accesso remoto ai file.

12.8.2 Protocollo di montaggio

Il **protocollo di montaggio** stabilisce la connessione logica iniziale tra un server e un client. In Solaris ogni calcolatore ha un processo server, esterno al kernel, che esegue le funzioni del protocollo.

Un’operazione di montaggio comprende il nome della directory remota da montare e il nome del calcolatore server in cui tale directory è memorizzata. La richiesta

di montaggio si mappa sulla RPC corrispondente e s’invia al server di montaggio in esecuzione nello specifico calcolatore server. Il server conserva una **lista di esportazione** (la `/etc/dfs/dfstab` nel sistema Solaris, modificabile soltanto da un *superuser*) che specifica i file system locali esportati per il montaggio e i nomi dei calcolatori a cui tale operazione è permessa. La lista può comprendere anche i diritti d’accesso, come la sola scrittura. Per semplificare la manutenzione delle liste di esportazione e delle tabelle di montaggio, si può usare uno schema distribuito di naming per contenere queste informazioni e renderle disponibili agli appropriati client.

Occorre ricordare che qualsiasi directory all’interno di un file system esportato si può montare in modo remoto da un calcolatore accreditato. Quando il server riceve una richiesta di montaggio conforme alla propria lista di esportazione, riporta al client un **handle del file** da usare come chiave per ulteriori accessi ai file che si trovano all’interno del file system montato. L’handle del file contiene tutte le informazioni di cui ha bisogno il server per identificare un proprio file. Nei termini dell’ambiente UNIX, l’handle del file è composto da un identificatore di file system e da un numero di *inode* per identificare la specifica directory montata nell’ambito del file system esportato.

Il server contiene anche una lista dei calcolatori client e delle corrispondenti directory correntemente montate. Questa lista si usa soprattutto per scopi amministrativi, per esempio per informare i client che un server sta andando fuori servizio. L’aggiunta o la cancellazione di elementi da questa lista sono gli unici modi in cui il protocollo di montaggio può modificare lo stato del server.

Generalmente un sistema ha una configurazione di montaggio predefinita che si stabilisce nella fase d’avviamento (`/etc/vfstab` in Solaris); tale configurazione si può comunque modificare. Oltre alla procedura di montaggio effettiva, il protocollo di montaggio comprende numerose altre procedure, come lo smontaggio e la restituzione della lista d’esportazione.

12.8.3 Protocollo NFS

Il protocollo NFS offre un insieme di RPC per operazioni su file remoti che supportano le seguenti operazioni:

- ricerca di un file in una directory;
- lettura di un insieme di elementi di una directory;
- manipolazione di collegamenti e di directory;
- accesso ad attributi di file;
- lettura e scrittura di file.

Queste procedure si possono invocare soltanto dopo aver stabilito un handle del file per la directory montata in remoto.

L’omissione delle operazioni `open` e `close` è intenzionale. Una caratteristica importante dei server NFS è l’*assenza dell’informazione di stato*. I server *stateless* non conservano informazioni sui loro client da un accesso all’altro. Dal lato del server

NFS non esistono equivalenti della tabella dei file aperti o delle strutture di file di UNIX, quindi ogni richiesta deve fornire un insieme completo di argomenti, tra cui un identificatore unico di file e un offset assoluto all'interno del file per svolgere le operazioni appropriate. L'architettura che ne deriva è robusta; non si devono prendere misure speciali per ripristinare un server dopo un guasto. Per tale ragione, le operazioni sui file devono essere idempotenti (ripeterle più volte non modifica l'effetto già ottenuto con la prima esecuzione dell'operazione). Per ottenere l'idempotenza, ciascuna richiesta dell'NFS ha un numero di sequenza, che permette al server di determinare duplicazioni o perdite nella sequenza delle richieste.

La presenza della suddetta lista di client sembra violare la proprietà dell'assenza di informazione di stato nel server. Tuttavia, essa non è essenziale ai fini del corretto funzionamento del client o del server, quindi non è necessario ricostruire tale lista dopo la caduta di un server; tale lista può contenere anche dati incoerenti e viene trattata come un semplice suggerimento.

Un'ulteriore implicazione della filosofia dei server senza informazione di stato è un risultato del funzionamento sincrono di una RPC consiste nel fatto che i dati modificati, tra cui riferimenti indiretti e blocchi di stato, si devono scrivere nei dischi del server prima che i risultati siano riportati al client. Un client può cioè ricorrere a una cache per i blocchi di scrittura, ma quando li invia al server, assume che abbiano raggiunto i dischi del server. Un server deve scrivere tutti i dati dell'NFS in modo sincrono. In questo modo la caduta di un server e il successivo ripristino saranno invisibili al client; tutti i blocchi che il server gestisce per il client resteranno intatti. La conseguente perdita di prestazioni può essere rilevante poiché si perdono i vantaggi derivanti dall'impiego del caching. Le prestazioni si possono incrementare impiegando dispositivi di memoria secondaria con una propria cache non volatile (di solito si tratta di memorie alimentate da una batteria). Il controllore del disco riporta che la scrittura nel disco è avvenuta quando la scrittura è avvenuta nella cache non volatile. Essenzialmente, il calcolatore "vede" una scrittura sincrona molto rapida. Questi blocchi restano intatti anche dopo una caduta del sistema, e periodicamente vengono trasferiti da questa memoria stabile al disco.

Una singola procedura di scrittura dell'NFS è garantita essere l'atomicità e non è inframmezzata ad altre chiamate di scrittura nello stesso file. Tuttavia, il protocollo NFS non fornisce meccanismi di controllo della concorrenza. Poiché ogni chiamata di scrittura o lettura dell'NFS può contenere fino a 8 KB di dati e i pacchetti UDP sono limitati a 1500 byte, può essere necessario dividere una chiamata di sistema `write()` in diverse RPC di scrittura. Quindi, due utenti che scrivono nello stesso file remoto possono riscontrare interferenze nei loro dati. Poiché la gestione di meccanismi di lock è inherentemente basata su informazioni di stato, si richiede che un servizio esterno all'NFS fornisca i meccanismi di locking, come nel caso di Solaris. Gli utenti sanno che per coordinare l'accesso ai file condivisi devono usare meccanismi che sono al di fuori dell'ambito dell'NFS.

L'NFS è integrato nel sistema operativo tramite un VFS. Per illustrarne l'architettura si può descrivere il modo in cui si gestisce un'operazione su un file remoto già

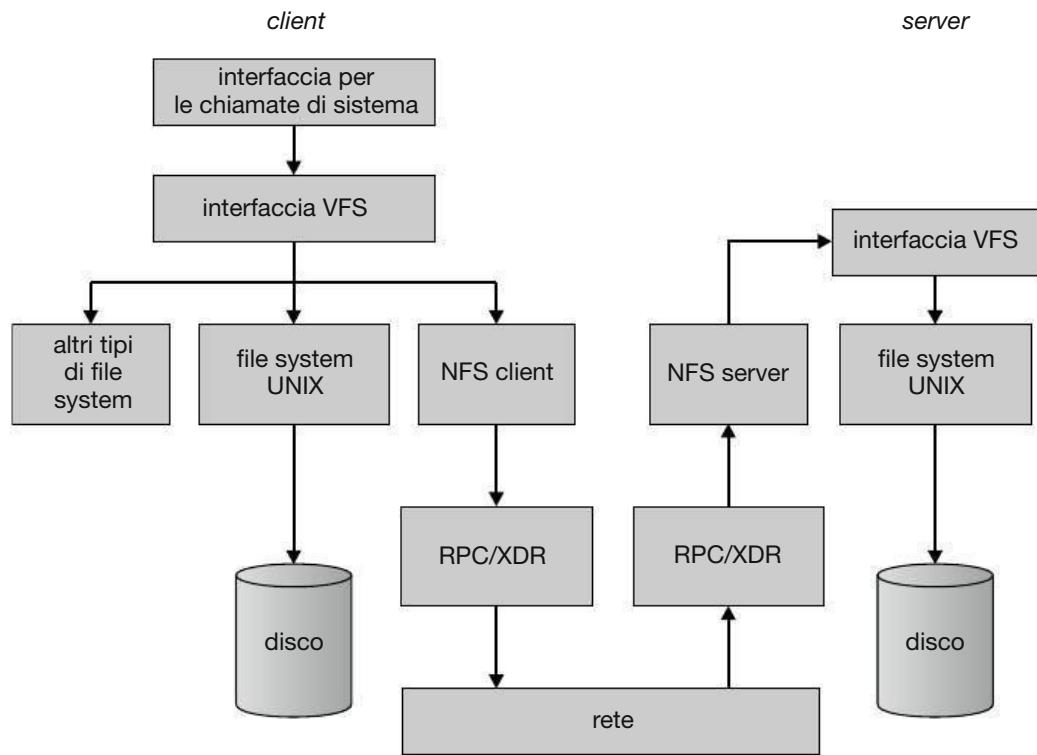


Figura 12.15 Schema dell'architettura dell'NFS.

aperto (Figura 12.15). Il client inizia l'operazione con un'ordinaria chiamata di sistema. Lo strato del sistema operativo mappa tale chiamata di sistema in un'operazione del VFS sull'opportuno *vnode*. Lo strato VFS identifica il file come remoto e invoca l'opportuna procedura dell'NFS. Avviene quindi una chiamata RPC allo strato del servizio NFS nel server remoto. Tale chiamata si reintroduce nello strato del VFS del sistema remoto, che riconosce essere locale e invoca l'appropriata operazione del file system. Questo cammino si ripercorre al contrario per restituire il risultato. Un vantaggio di tale architettura è che il client e il server sono identici; così un calcolatore può essere un client, un server o entrambi. L'effettivo servizio su ciascun server è eseguito da thread del kernel.

12.8.4 Traduzione dei nomi di percorso

La **traduzione dei nomi di percorso** (*path-name translation*) del protocollo NFS prevede l'analisi di un nome di percorso – per esempio `/usr/local/dir1/file.txt` – al fine di estrarre i nomi delle singole directory, o componenti – nell'esempio: (1) `usr`, (2) `local` e (3) `dir1`. La traduzione dei nomi di percorso si compie suddividendo il percorso stesso in nomi di componenti ed eseguendo una chiamata `lookup()` dell'NFS separata per ogni coppia formata da un nome di componente e un *vnode* del directory. Quando si attraversa un punto di montaggio, ogni `lookup` di un componente causa una RPC separata al server. Questo schema di attraversamento del nome di percorso è costoso ma necessario, poiché ogni client ha un'unica con-

gurazione del proprio spazio di nomi logico, determinata dai montaggi che ha eseguito. Sarebbe molto più efficiente consegnare un nome di percorso a un server e ricevere un *vnode* di destinazione una volta incontrato un punto di montaggio. Tuttavia dovunque nel percorso può essere presente un altro punto di montaggio per quel particolare client, sconosciuto al server stateless.

Una cache per la ricerca dei nomi delle directory, nel client, conserva i *vnode* per i nomi delle directory remote; in questo modo si accelerano i riferimenti ai file con lo stesso nome di percorso iniziale. Se gli attributi restituiti dal server non corrispondono agli attributi del *vnode* nella cache, si scarta il contenuto della cache della directory.

Occorre ricordare che alcune implementazioni dell’NFS permettono il montaggio di un file system remoto sopra un altro file system remoto già montato; si tratta del *montaggio a cascata*. Quando un client ha un montaggio a cascata, nel caso di un attraversamento di un nome di percorso può essere coinvolto più di un server. Tuttavia, quando un client fa una ricerca in una directory sulla quale il server ha montato un file system, il client vede la directory sottostante e non la directory montata.

12.8.5 Operazioni remote

Eccetto che per l’apertura e la chiusura dei file, tra le normali chiamate di sistema di UNIX per operazioni su file e le RPC del protocollo NFS esiste una corrispondenza quasi uno a uno. Quindi, un’operazione remota su un file si può tradurre direttamente nella RPC corrispondente. Dal punto di vista concettuale l’NFS aderisce al paradigma del servizio remoto, ma in pratica si usano tecniche di buffering e cache per migliorare le prestazioni. La corrispondenza tra un’operazione remota e una RPC non è diretta; invece le RPC prelevano blocchi e attributi del file e li memorizzano localmente nelle cache. Le successive operazioni remote usano i dati nella cache, purchè siano rispettati i vincoli di coerenza.

Esistono due cache: la cache degli attributi dei file (informazioni degli *inode*) e la cache dei blocchi del file. Quando un file viene aperto, il kernel fa un controllo col server remoto per stabilire se deve prelevare o riconvalidare gli attributi nella cache: i blocchi del file che sono in cache si usano solo se i corrispondenti attributi sono aggiornati nella loro cache. La cache degli attributi viene aggiornata ogni volta che arrivano nuovi attributi dal server. Gli attributi nella cache si scartano, per default, dopo 60 secondi. Tra il server e il client si usano le tecniche di lettura anticipata (*read-ahead*) e scrittura differita (*delayed-write*). Finché il server non ha confermato che i dati sono stati scritti nei dischi, i client non liberano i blocchi di scrittura differita. La scrittura differita viene mantenuta anche quando si apre un file concorrentemente, in modi conflittuali. Ne deriva che la semantica UNIX (Paragrafo 11.5.3.1) non si conserva.

Mettere a punto il sistema per migliorarne le prestazioni rende difficile caratterizzare la semantica della coerenza dell’NFS. File nuovi creati in un calcolatore possono non essere visibili in altri calcolatori per 30 secondi. Inoltre, le scritture eseguite in un file in un sito possono o meno essere visibili anche in altri siti che hanno aperto

lo stesso file per la lettura. Le nuove aperture di un file consentono di osservare solo le modifiche già inviate al server, quindi l’NFS non fornisce né una stretta emulazione della semantica UNIX, né la semantica delle sessioni di Andrew (Paragrafo 11.5.3.2). Nonostante questi inconvenienti, l’utilità e le alte prestazioni del meccanismo ne fanno il sistema distribuito multi-vendor più usato.

12.9 Esempio: il file system WAFL

L’I/O da e verso il disco si riflette significativamente sulle prestazioni del sistema. Di conseguenza, i progettisti devono esercitare grande cura sia nella progettazione sia nell’implementazione del file system. Alcuni file system sono concepiti per finalità generali, ossia sono in grado di offrire prestazioni accettabili e funzionalità adatte a file che differiscono per tipo e dimensione, e a carichi di I/O diversi. Altri sono ottimizzati per compiti specifici, nel tentativo di fornire, per alcune applicazioni dedicate, prestazioni migliori di quelle dei sistemi a carattere generale. Un’ottimizzazione di questo genere è rappresentata dal file system WAFL di Network Appliance. WAFL, acronimo di *write-anywhere file layout* (“modello di file per la scrittura ovunque”), è un file system potente ed elegante, ottimizzato per le scritture casuali.

È utilizzato in esclusiva dai file server di rete prodotti da Network Appliance, ed è pensato per essere utilizzato come file system distribuito. Benché sia stato originalmente progettato per i soli protocolli NFS e CFS, consente l’invio di file ai client tramite NFS, CIFS, `ftp` e `http`. Quando molti client contattano un file server attraverso questi protocolli, le richieste di letture casuali, e ancor di più quelle relative a scritture casuali, aumentano sensibilmente. I protocolli NFS e CIFS utilizzano il caching dei dati provenienti dalle operazioni in lettura: per i file server, dunque, è la scrittura che assume la massima importanza.

L’impiego del WAFL presuppone che i file server dispongano di una cache NVRAM per le scritture. I progettisti del WAFL hanno sfruttato il vantaggio di lavorare su un’architettura specifica, con una cache per la memorizzazione stabile dei dati, al fine di ottimizzare il file system per l’I/O ad accesso casuale. Uno dei principi che hanno ispirato il WAFL è la facilità d’uso. I suoi autori, inoltre, hanno aggiunto una nuova funzionalità di duplicazione istantanea (*snapshot*), per creare in più momenti, come vedremo, diverse copie a sola lettura del file system.

Il file system è simile al Berkeley Fast, con varie modifiche; è basato sui blocchi e usa gli inode per la descrizione dei file. Ciascun inode contiene 16 puntatori ad altrettanti blocchi (o blocchi indiretti), che appartengono al file descritto dall’inode. Ciascun file system possiede un inode radice. Come si vede nella Figura 12.16, tutti i metadati sono custoditi all’interno di file: un file per gli inode, un altro per la mappa dei blocchi liberi e un terzo per la mappa degli inode liberi. Poiché si tratta di file ordinari, i blocchi di dati non hanno un indirizzo predefinito, ma possono risiedere dovunque. Se un file system viene ampliato con l’aggiunta di dischi, esso aumenterà in modo automatico la lunghezza di questi file per i metadati.

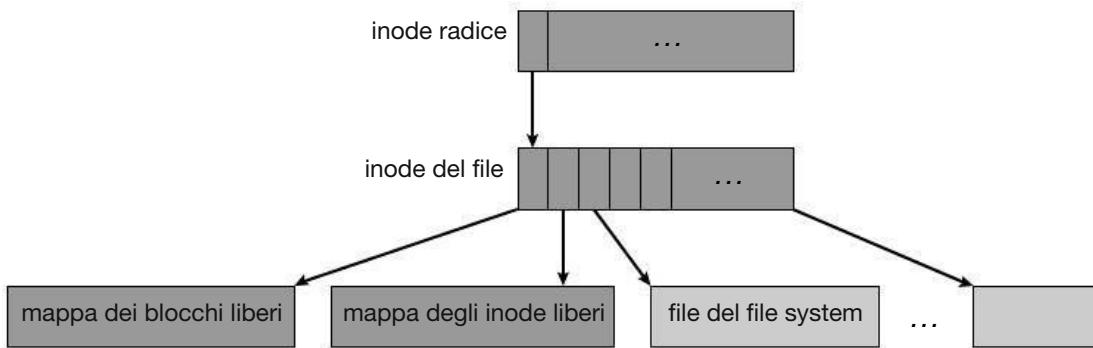


Figura 12.16 Organizzazione del file system WAFL.

Pertanto, un file system WAFL è un albero di blocchi che si dipartono dall'inode radice; per creare uno snapshot, WAFL crea una copia dell'inode radice. In seguito a ciò, ogni aggiornamento dei file o dei metadati occuperà blocchi nuovi, anziché sovrascrivere i relativi blocchi esistenti. Il nuovo inode radice punta ai metadati e ai dati nella loro versione aggiornata. Nello stesso tempo, il vecchio inode radice continua a puntare ai blocchi precedenti, non aggiornati, e in tal modo dà accesso a un'immagine del file system che ricalca esattamente il momento in cui era stato fotografato; lo spazio occupato sul disco per questa operazione è davvero esiguo. In sostanza, lo snapshot richiede un supplemento di spazio sul disco equivalente ai soli blocchi che siano stati modificati dal momento in cui si è scattata l'istantanea.

Una differenza di rilievo in confronto a file system più tradizionali è data dalla mappa dei blocchi liberi, che possiede più di un bit per blocco. Si tratta di una maschera di bit che prevede un bit impostato a uno a ogni istantanea che stia utilizzando il blocco. Quando tutte le istantanee che stavano utilizzando un blocco sono cancellate, la maschera di bit associata è formata da zeri, e il blocco può essere riutilizzato.

I blocchi usati non sono mai sovrascritti, di modo che le scritture avvengono a gran velocità, visto che le operazioni in scrittura possono sfruttare il blocco libero più vicino alla posizione corrente della testina. Vi sono, nel WAFL, molti altri meccanismi per ottimizzare le prestazioni.

Possono coesistere allo stesso tempo numerose istantanee: se ne può scattare una per ogni ora del giorno e ogni giorno del mese. Gli utenti abilitati ad accedere a queste istantanee, hanno accesso a ciascuno dei file per come era nei momenti in cui è stato fotografato. Questa funzionalità è anche utile per il backup, il testing, per gestire diverse versioni di un progetto, e altro ancora. L'implementazione degli snapshot in WAFL è molto efficiente, dato che non richiede neppure di duplicare con la copiatura su scrittura ciascun blocco di dati prima che sia modificato. Altri file system offrono la medesima funzionalità, ma spesso con minor efficienza. Le istantanee del WAFL sono descritte nella Figura 12.17.

Le versioni più recenti del WAFL permettono istantanee di lettura e scrittura, note come **cloni**. Come le istantanee, anche i cloni sono efficienti, in quanto utilizzano le stesse tecniche. In questo caso, uno snapshot di sola lettura cattura lo stato del file

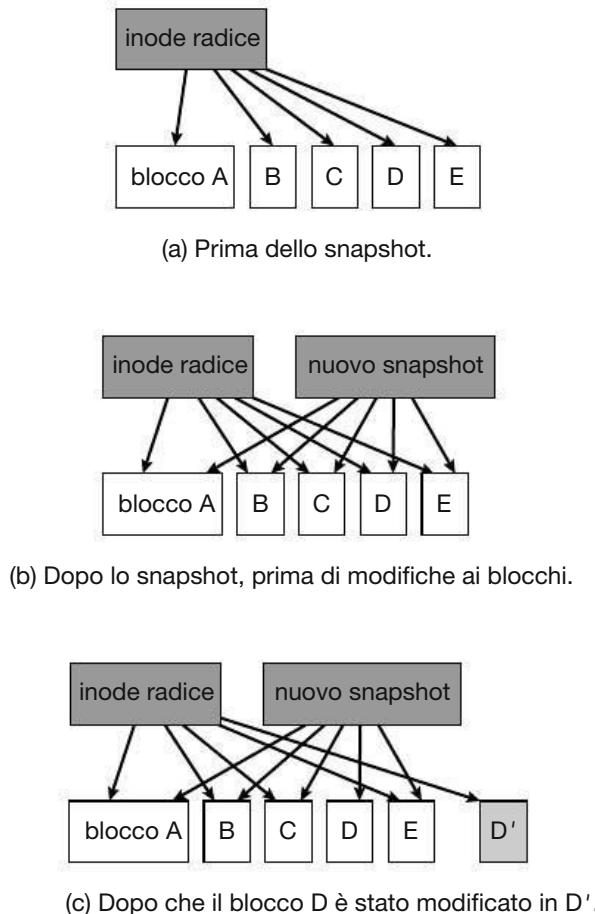


Figura 12.17 Snapshot (istantanee) di WAFL.

system e il clone fa riferimento a quello snapshot. Eventuali scritture sul clone sono memorizzate in nuovi blocchi e i puntatori del clone vengono aggiornati per fare riferimento ai nuovi blocchi. L'istantanea originale non è modificata, mantenendo così l'immagine del file system prima dell'aggiornamento del clone. I cloni possono essere “promossi” a sostituti del file system originale e ciò comporta l'eliminazione dei vecchi puntatori e di ogni vecchio blocco a essi associato. I cloni sono utili per il testing e gli aggiornamenti; la versione originale rimane infatti immutata e il clone viene cancellato quando il test è stato effettuato o quando l'aggiornamento fallisce.

Un'altra caratteristica che si ottiene naturalmente dall'implementazione del file system WAFL è la **replica**, ovvero la duplicazione e la sincronizzazione di un insieme di dati su un altro sistema attraverso una rete. Per prima cosa, lo snapshot di un file system WAFL viene duplicato su un altro sistema. Quando si genera un altro snapshot del sistema sorgente, per aggiornare il sistema remoto è sufficiente inviare tutti i blocchi contenuti nel nuovo snapshot. Questi blocchi sono quelli che hanno subito modifiche nell'intervallo di tempo tra lo scatto delle due istantanee. Il sistema remoto aggiunge questi blocchi al file system e aggiorna i propri puntatori. Il nuovo sistema è dunque un duplciato del sistema sorgente nel momento in cui è stata generato il secondo snapshot.

La ripetizione di questo processo fa sì che il sistema remoto sia una copia quasi aggiornata del primo sistema. Le repliche sono utili per il *disaster recovery*. Nel caso in cui il primo sistema sia vittima di una perdita totale di dati, la maggior parte dei suoi dati sarebbe comunque disponibile nella replica sul sistema remoto.

Infine è opportuno ricordare che il file system ZFS supporta snapshot, cloni e un sistema di repliche altrettanto efficienti.

12.10 Sommario

Il file system risiede permanentemente in memoria secondaria, progettata per contenere, permanentemente, una grande quantità di dati. Il più comune mezzo di memoria secondaria è il disco.

I dischi si possono segmentare in partizioni, allo scopo di controllarne l'uso e consentire più file system, anche di tipo diverso, per ogni disco. Questi file system si montano su una architettura di file system logico per renderli disponibili all'uso. I file system spesso si realizzano secondo una struttura stratificata o modulare: i livelli più bassi hanno a che fare con le caratteristiche fisiche dei dispositivi di memorizzazione; i livelli più alti hanno a che fare con i nomi simbolici e le caratteristiche logiche dei file; i livelli intermedi fanno corrispondere le caratteristiche logiche dei file alle caratteristiche fisiche dei dispositivi.

Ogni tipo di file system può avere diverse strutture e algoritmi. Uno strato VFS consente ai livelli superiori di aver a che fare con ciascun tipo di file system in modo uniforme. Nella struttura della directory del sistema si possono integrare anche file system remoti sui quali si agisce con le ordinarie chiamate di sistema tramite l'interfaccia del VFS.

Lo spazio dei dischi può essere allocato ai file in tre modi: allocazione contigua, concatenata e indicizzata. L'allocazione contigua può risentire di frammentazione esterna. L'accesso diretto ai file è molto inefficiente con l'allocazione concatenata. L'allocazione indicizzata, infine, può richiedere un notevole overhead per il proprio blocco indice. Questi algoritmi si possono ottimizzare in molti modi. Lo spazio contiguo si può allargare attraverso delle estensioni allo scopo di aumentare la flessibilità e ridurre la frammentazione esterna. L'allocazione indicizzata si può realizzare in cluster per incrementare il throughput e ridurre il numero di elementi dell'indice necessari. L'indicizzazione in cluster di grandi dimensioni è simile all'allocazione contigua con estensioni.

I metodi di allocazione dello spazio libero influenzano anche l'efficienza d'uso dello spazio dei dischi, le prestazioni del file system e l'affidabilità della memoria secondaria. I metodi usati comprendono i vettori di bit e le liste concatenate. Le ottimizzazioni comprendono il raggruppamento, il conteggio e la FAT, che colloca la lista concatenata in una singola area contigua.

Le procedure di gestione delle directory devono tener conto dell'efficienza, delle prestazioni e dell'affidabilità. La tabella hash è un metodo usato comunemente; è veloce ed efficiente. Sfortunatamente, il danneggiamento di una tabella o la caduta del

sistema possono causare incoerenze tra le informazioni contenute nelle directory e il contenuto del disco. Con un verificatore di coerenza si può porre rimedio ai danni. Gli strumenti del sistema operativo per la creazione di copie di backup consentono la copiatura nelle unità a nastro dei dati contenuti nei dischi allo scopo di poterli ripristinare in seguito a perdite dovute a malfunzionamenti dei dispositivi fisici, errori del sistema operativo, o a errori degli utenti.

I file system di rete, come l’NFS, usano un metodo client-server per permettere agli utenti di accedere a file e directory in calcolatori remoti come se fossero in file system locali. Le chiamate di sistema del client vengono tradotte nei protocolli di rete, per poi ritradurle in operazioni del file system nel server. La messa in rete e gli accessi di più client creano sfide nei campi della coerenza dei dati e delle prestazioni.

A causa del ruolo fondamentale che i file system hanno nel funzionamento di un sistema, le loro prestazioni e affidabilità sono fondamentali. Tecniche come quelle che impiegano le cache e i file di log migliorano le prestazioni, mentre i log e le tecniche RAID migliorano l’affidabilità. Il sistema WAFL è un esempio di ottimizzazione delle prestazioni per rispondere a uno specifico carico di I/O.

Esercizi di ripasso

12.1 Prendete in considerazione un file costituito da 100 blocchi. Assumete che il blocco di controllo del file (e il blocco dell’indice, in caso di allocazione indicizzata) sia già in memoria. Calcolate quante operazioni di I/O del disco sono necessarie con le strategie di allocazione contigua, concatenate e indicizzate (singolo livello), se, per un blocco, valgono le condizioni che seguono. Nel caso di allocazione contigua, assumete che non ci sia spazio di crescita all’inizio, ma solo alla fine. Assumete inoltre che il blocco di informazione da aggiungere sia salvato in memoria.

- a. Il blocco viene aggiunto all’inizio.
- b. Il blocco viene aggiunto al centro.
- c. Il blocco viene aggiunto alla fine.
- d. Il blocco viene rimosso dall’inizio.
- e. Il blocco viene rimosso dal centro.
- f. Il blocco viene rimosso dalla fine.

12.2 Quali problemi potrebbero verificarsi se un sistema permettesse di montare il file system simultaneamente in più di una posizione?

12.3 Perché la mappa di bit per l’allocazione dei file deve essere conservata nella memoria di massa, e non nella memoria principale?

12.4 Considerate un sistema che supporta le strategie di allocazione contigua, concatenata e indicizzata. Quali criteri dovrebbero essere impiegati per decidere quale strategia è migliore per un dato file?

- 12.5** Un problema dell’allocazione contigua consiste nel fatto che l’utente deve preallocare abbastanza spazio per ogni file. Se il file diventa più grande dello spazio che gli è stato allocato, devono essere intraprese delle azioni specifiche. Questo problema può essere risolto definendo una struttura di file che consiste in un’area inizialmente contigua (di una dimensione specificata.) Se l’area viene riempita, il sistema operativo definisce automaticamente un’area di overflow linkata all’area contigua iniziale. Se l’area di overflow viene riempita, si alloca una seconda area di overflow. Paragonate questa implementazione con le versioni standard della allocazione contigua e concatenata.
- 12.6** Come possono le cache contribuire a migliorare le prestazioni? Perché i sistemi non utilizzano un maggior numero di cache, oppure cache più grandi, se esse sono così utili?
- 12.7** Perché è vantaggioso per l’utente che il sistema operativo allochi dinamicamente le proprie tabelle interne? Quali sono le conseguenze negative in cui incorrono i sistemi operativi che si comportano in questo modo?
- 12.8** Spiegate come lo strato VSF permetta al sistema operativo di supportare facilmente diversi tipi di file system.

Esercizi

- 12.9** Considerate un file system che adopera un metodo di allocazione contigua modificato, comprensivo di estensioni. Un file contiene diverse estensioni, ognuna delle quali corrisponde a un insieme contiguo di blocchi. Un aspetto cruciale, per tali sistemi, è il grado di variabilità nella dimensione delle estensioni. Quali sono i vantaggi e gli svantaggi nel caso che:
- tutte le estensioni siano della stessa grandezza, che è predeterminata;
 - le estensioni possono essere di qualunque misura e sono allocate dinamicamente;
 - le estensioni variano tra poche misure fisse, che sono predeterminate.
- 12.10** Confrontate le prestazioni delle tre tecniche per l’allocazione dei blocchi di un disco (allocazione contigua, concatenata e indicizzata), sia per file ad accesso sequenziale sia per file ad accesso casuale.
- 12.11** Quali vantaggi offre la variante dell’allocazione concatenata che utilizza una FAT per collegare i blocchi di un file?
- 12.12** Considerate un sistema che tenga traccia dello spazio libero in una lista apposita.
- Supponete di perdere il puntatore alla lista. Il sistema è in grado di ricostruire la lista dello spazio libero? Argomentate la vostra risposta.

- b. Considerate un file system simile a quello con allocazione indicizzata impiegato da UNIX. Quante operazioni di I/O del disco potrebbero essere necessarie per leggere i contenuti di un piccolo file locale posto in /a/b/c? Si assuma che nessuno dei blocchi del disco sia stato memorizzato nella cache.
- c. Proponete una soluzione che impedisca la perdita del puntatore dovuta a un guasto della memoria.

12.13 In alcuni file system è possibile allocare lo spazio sul disco con diverse granularità. Un file system, per esempio, può allocare 4 KB di spazio sul disco scegliendo un blocco unico da 4 KB oppure otto blocchi da 512 byte. In quale modo si può trarre vantaggio da questa flessibilità per migliorare le prestazioni? Quali modifiche si dovrebbero introdurre nel modello di gestione dello spazio libero per includervi questa caratteristica?

12.14 Discutete di come i meccanismi di ottimizzazione delle prestazioni dei file system potrebbero generare difficoltà nel mantenimento della coerenza dei sistemi, nel caso di un crash di sistema.

12.15 Considerate un file system in un disco con dimensioni dei blocchi logici e fisici di 512 byte. Supponete che le informazioni su ciascun file siano già in memoria. Per ciascuno dei tre metodi di allocazione (contigua, concatenata e indicizzata) fornite le risposte alle seguenti domande:

- a. dite come in questo sistema si fanno corrispondere gli indirizzi logici agli indirizzi fisici (per l'allocazione indicizzata supponete che la lunghezza di un file sia sempre inferiore a 512 blocchi);
- b. se l'ultimo accesso è stato fatto al blocco 10 (posizione corrente), dite quanti blocchi fisici si devono leggere dal disco per accedere al blocco logico 4.

12.16 Considerate un file system che utilizza degli inode per rappresentare i file. I blocchi del disco hanno una dimensione di 8 KB e un puntatore a un blocco del disco richiede 4 byte. Questo file system ha 12 blocchi diretti sul disco, ma anche blocchi indiretti singoli, doppi e tripli. Qual è la dimensione massima di file che può essere memorizzata nel sistema?

12.17 In un dispositivo di memoria si può eliminare la frammentazione ricompattando le informazioni. I tipici dispositivi a disco non dispongono di registri di rilocazione o registri di base (come quelli usati per compattare la memoria); in questa situazione dite come si possono rilocare i file. Fornite tre motivi per i quali la ricompattazione e la rilocazione dei file vengono spesso evitate.

12.18 Considerate l'integrazione seguente a un protocollo per l'accesso remoto ai file. Ciascun client gestisce una cache per i nomi, in cui memorizza le traduzioni dei nomi dei file nei corrispondenti handle. Quali problemi dovete tenere in considerazione nel realizzare la cache per i nomi?

12.19 Spiegate perché l'annotazione degli aggiornamenti dei metadati assicura il ripristino di un file system dopo una caduta del sistema.

12.20 Considerate il seguente schema di creazione di copie di riserva.

Giorno 1. Copiatura nel supporto di backup delle copie di riserva di tutti i file contenuti nel disco.

Giorno 2. Copiatura in un altro supporto di tutti i file modificati dal giorno 1.

Giorno 3. Copiatura in un altro supporto di tutti i file modificati dal giorno 1.

Tale schema differisce dalla sequenza data nel Paragrafo 12.7.4 per il fatto che tutte le copiate riguardano tutti i file modificati dopo la prima copiatura completa. Dite quali sono i vantaggi e gli svantaggi di questo sistema rispetto a quello del Paragrafo 12.7.4 e se le operazioni di recupero sono più semplici o più difficili. Motivate le vostre risposte.

Problemi di programmazione

L'esercizio seguente esamina la relazione tra i file e gli inode su sistemi UNIX e Linux. Su questi sistemi i file sono rappresentati mediante inode, ovvero ogni inode è un file (e viceversa). È possibile completare questo esercizio sulla macchina virtuale Linux (*Linux Virtual Machine*) fornita con questo testo, oppure su qualsiasi sistema Linux, UNIX o Mac OS X, ma in questo caso è richiesta la creazione di due semplici file di testo denominati `file1.txt` e `file3.txt` contenenti ciascuno frasi distinte.

12.21 Aprite il file `file1.txt`, disponibile con il codice sorgente di questo testo, ed esaminatene il contenuto. Ottenete poi il numero di inode di questo file con il comando

```
ls -li file1.txt
```

Questo comando produrrà un output simile al seguente:

```
16980 -rw-r--r-- 2 os os 22 Sep 14 16:13 file1.txt
```

dove il numero di inode è in grassetto. Probabilmente il numero di inode di `file1.txt` sarà diverso sul vostro sistema.

Il comando UNIX `ln` crea un collegamento tra un file sorgente e un file destinazione e si utilizza nel seguente modo:

```
ln [-s] <file sorgente> <file destinazione>
```

UNIX fornisce due tipi di collegamento: (1) hard link e (2) soft link. Un hard link crea un file di destinazione che ha lo stesso inode del file sorgente. Digitate il seguente comando per creare un hard link tra `file1.txt` e `file2.txt`:

```
ln file1.txt file2.txt
```

Quali sono i valori di inode di `file1.txt` e `file2.txt`? Sono uguali o distinti? I due file hanno lo stesso contenuto o contenuti diversi?

Successivamente, modificate il contenuto di `file2.txt` e, dopo averlo fatto, esamineate il contenuto di `file1.txt`. Il contenuto di `file1.txt` e `file2.txt` è uguale o differente?

Immettete quindi il seguente comando che rimuove `file1.txt`:

```
rm file1.txt
```

`file2.txt` esiste ancora?

Ora esamineate le pagine del manuale (tramite il comando `man`) relative ai comandi `unlink` e `rm`. In seguito, rimuovete `file2.txt` con il comando

```
strace rm file2.txt
```

Il comando `strace` ripercorre le chiamate di sistema realizzate durante l'esecuzione del comando `rm file2.txt`. Quale chiamata di sistema viene utilizzata per eliminare `file2.txt`?

Un soft link (o link simbolico) crea un nuovo file che “punta” al nome del file collegato. Create un link simbolico al file `file3.txt`, disponibile nel codice sorgente di questo testo, immettendo il seguente comando:

```
ln -s file3.txt file4.txt
```

Ottenete poi i numeri di inode di `file3.txt` e `file4.txt` utilizzando il comando

```
ls -li file*.txt
```

Gli inode coincidono oppure no? Modificate ora il contenuto di `file4.txt`. Anche il contenuto di `file3.txt` è stato modificato? Infine, eliminate `file3.txt` e, dopo averlo fatto, spiegate cosa succede quando si tenta di modificare `file4.txt`.

Note bibliografiche

Il sistema FAT dell'MS-DOS è spiegato in [Norton e Wilton 1988]. L'organizzazione interna del sistema BSD UNIX è ampiamente trattata in [McKusick e Neville-Neil 2005]. I dettagli riguardanti i file system per Linux si possono trovare in [Love 2010]. Il file system di Google è descritto in [Ghemawat et al. 2003]. Si può trovare FUSE all'indirizzo <http://fuse.sourceforge.net/>.

I modelli di accesso ai file con logging per migliorare sia le prestazioni sia la coerenza sono stati presentati da [Rosenblum e Ousterhout 1991], [Seltzer et al. 1993], [Seltzer et al. 1995]. Gli algoritmi sugli alberi bilanciati sono discussi (insieme a molto altro materiale) in [Knuth 1998] e in [Cormen et al. 2009]. [Silvers 2000] discute l'implementazione della cache delle pagine in NetBSD. Il codice sorgente ZFS per le mappe di spazio è disponibile su:

http://src.opensolaris.org/source/xref/onnv/onnvgate/usr/src/uts/common/fs/zfs/space_map.c.

L'uso delle memorie cache nella gestione dei dischi è trattato da [McKeon 1985] e [Smith 1985]; l'uso della cache nel sistema operativo sperimentale Sprite, è descritto in [Nelson et al. 1988]. Analisi generali sulle tecnologie delle memorie di massa si trovano in [Chi 1982] e [Hoagland 1985]. L'opera di [Folk e Zoellick 1987] concerne le varie strutture di file. [Silvers 2000] analizza la realizzazione della cache delle pagine nel sistema operativo NetBSD.

[Callaghan 2000] trattano il file system di rete NFS. La versione 4 di NFS è uno standard descritto su <http://www.ietf.org/rfc3530.txt>. [Ousterhout 1991] discute il ruolo dello stato distribuito nei file system di rete. Architetture basate sulla registrazione per file system di rete sono proposte in [Hartman e Ousterhout 1995], e da [Thekkath et al. 1997]. [Vahalia 1996] e [Mauro e McDougall 2007] descrivono l'NFS e il file system del sistema operativo UNIX, l'UFS. [Solomon 1998] descrive il file system NTFS. Il file system *Ext3* del sistema operativo Linux è stato trattato da [Mauerer 2008], mentre [Hitz et al. 1995] si è occupato del file system WAFL. Per approfondimenti riguardanti ZFS, si consulti <http://www.opensolaris.org/os/community/ZFS/docs>.

Bibliografia

- [Callaghan 2000] B. Callaghan, *NFS Illustrated*, Addison-Wesley, 2000.
- [Cormen et al. 2009] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, *Introduction to Algorithms*, Third Edition, MIT Press, 2009.
- [Ghemawat et al. 2003] S. Ghemawat, H. Gobioff e S.-T. Leung, “The Google File System”, Proceedings of the ACM Symposium on Operating Systems Principles, 2003.
- [Hartman e Ousterhout 1995] J. H. Hartman e J. K. Ousterhout, “The Zebra Striped Network File System”, ACM Transactions on Computer Systems, Vol. 13, Num. 3, p. 274–310, 1995.
- [Hitz et al. 1995] D. Hitz, J. Lau e M. Malcolm, “File System Design for an NFS File Server Appliance”, Technical report, NetApp, 1995.
- [Knuth 1998] D. E. Knuth, *The Art of Computer Programming*, Volume 3: Sorting and Searching, 2° ed., Addison-Wesley, 1998.
- [Love 2010] R. Love, *Linux Kernel Development*, 3° ed., Developer's Library, 2010.
- [Mauerer 2008] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons, 2008.
- [Mauro e McDougall 2007] J. Mauro e R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall, 2007.
- [McKusick e Neville-Neil 2005] M. K. McKusick e G. V. Neville-Neil, *The Design and Implementation of the FreeBSD UNIX Operating System*, Addison Wesley, 2005.
- [Norton e Wilton 1988] P. Norton e R. Wilton, *The New Peter Norton Programmer's Guide to the IBM PC & PS/2*, Microsoft Press, 1988.

- [**Ousterhout 1991**] J. Ousterhout. “The Role of Distributed State”. In CMU Computer Science: a 25th Anniversary Commemorative, R. F. Rashid, Ed., Addison-Wesley, 1991.
- [**Rosenblum e Ousterhout 1991**] M. Rosenblum e J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 1–15, 1991.
- [**Seltzer et al. 1993**] M. I. Seltzer, K. Bostic, M. K. McKusick e C. Staelin, “An Implementation of a Log-Structured File System for UNIX”, USENIX Winter, p. 307–326, 1993.
- [**Seltzer et al. 1995**] M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains e V. N. Padmanabhan, File System Logging Versus Clustering: A Performance Comparison”, USENIX Winter, p. 249–264, 1995.
- [**Silvers 2000**] C. Silvers, “UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD”, USENIX Annual Technical Conference—FREENIX Track, 2000.
- [**Solomon 1998**] D. A. Solomon, *Inside Windows NT*, 2° ed., Microsoft Press, 1998.
- [**Thekkath et al. 1997**] C. A. Thekkath, T. Mann e E. K. Lee, “Frangipani: A Scalable Distributed File System”, Symposium on Operating Systems Principles, p. 224–237, 1997.
- [**Vahalia 1996**] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall, 1996.

CAPITOLO

13

OBIETTIVI DEL CAPITOLO

- Analisi della struttura del sottosistema di I/O di un sistema operativo.
- Elementi base dell'hardware di I/O e problematiche correlate.
- Aspetti relativi alle prestazioni dell'hardware e del software di I/O.

Sistemi di I/O

I due compiti principali di un calcolatore sono l'I/O e l'elaborazione. In molti casi il compito principale è costituito dall'I/O mentre l'elaborazione è semplicemente accessoria: quando si consulta una pagina web, o quando si modifica un file, ciò che più direttamente interessa l'utente è la lettura o l'immissione di informazioni, non l'elaborazione di qualche risposta.

Il ruolo di un sistema operativo nell'I/O di un calcolatore è quello di gestire e controllare le operazioni e i dispositivi di I/O. Sebbene in altri capitoli compaiano argomenti collegati, in questo capitolo sono raccolti tutti gli elementi utili alla composizione di un quadro d'insieme dell'I/O. Poiché il genere delle interfacce hardware stabilisce i requisiti che le funzioni interne del sistema operativo devono possedere, si descrivono innanzitutto i fondamenti dell'hardware di I/O. Quindi si discutono i servizi di I/O che il sistema operativo fornisce e come questi sono inclusi nell'interfaccia di I/O per le applicazioni; inoltre si spiega come il sistema operativo colmi il divario tra le interfacce hardware e quelle per le applicazioni. Si prende in esame anche il meccanismo STREAMS di UNIX System V, che consente a un'applicazione di comporre dinamicamente catene di codice di driver. Infine, si trattano gli aspetti riguardanti le prestazioni dell'I/O e i principi di progettazione dei sistemi operativi utili al miglioramento delle prestazioni dell'I/O.

13.1 Introduzione

Il controllo dei dispositivi connessi a un calcolatore è una delle questioni più importanti che riguardano i progettisti di sistemi operativi. Poiché i dispositivi di I/O sono così largamente diversi per funzioni e velocità (si considerino per esempio un mouse, un disco e un archivio di nastri), altrettanto diversi devono essere i metodi di controllo. Tali metodi costituiscono il *sottosistema di I/O* del kernel; questo sottosistema separa il resto del kernel dalla complessità di gestione dei dispositivi di I/O.

La tecnologia dei dispositivi di I/O mostra due tendenze tra loro in conflitto. Da una parte, si osserva la crescente uniformazione a standard delle interfacce fisiche e logiche, e ciò semplifica l'introduzione nei calcolatori e nei sistemi operativi già esistenti di più avanzate generazioni di dispositivi. D'altra parte, però, si assiste a una crescente varietà di dispositivi di I/O; alcuni di loro sono tanto diversi dai dispositivi precedenti da rendere molto difficile il compito di integrarli nei calcolatori e nei sistemi operativi esistenti. Questo problema si affronta con una combinazione di tecniche hardware e software. Gli elementi di base dell'hardware di I/O – porte, bus e controllori di dispositivi – si possono connettere con un'ampia varietà di dispositivi di I/O. Il kernel del sistema operativo è strutturato in moduli di driver dei dispositivi allo scopo di incapsulare i dettagli e le particolarità dei diversi dispositivi. I **driver dei dispositivi** offrono al sottosistema di I/O un'interfaccia uniforme per l'accesso ai dispositivi, così come le chiamate di sistema forniscono un'interfaccia uniforme tra le applicazioni e il sistema operativo.

13.2 Hardware di I/O

I calcolatori fanno funzionare un gran numero di tipi di dispositivi. La maggior parte rientra nella categoria dei dispositivi di memorizzazione (dischi, nastri), dispositivi di trasmissione (connessioni di rete, Bluetooth), interfacce uomo-macchina (schermi, tastiere, mouse, ingressi e uscite audio). Altri dispositivi sono più specializzati, come i dispositivi di pilotaggio di un caccia a reazione. In questi velivoli il pilota interagisce col calcolatore di bordo tramite la *cloche* e la pedaliera, il calcolatore invia comandi per l'attivazione dei motori che azionano timoni, *flap* e propulsori. Nonostante l'incredibile varietà dei dispositivi di I/O, bastano alcuni concetti per capire come siano connessi e come il sistema operativo li controlli.

Un dispositivo comunica con un sistema elaborativo inviando segnali attraverso un cavo o attraverso l'etere e comunica con il calcolatore tramite un punto di connessione (**porta**), per esempio una porta seriale. Se più dispositivi condividono un insieme di fili, la connessione è detta *bus*. Un **bus** è un insieme di fili e un protocollo rigorosamente definito che specifica l'insieme dei messaggi che si possono inviare attraverso i fili. In termini elettronici, i messaggi si inviano tramite configurazioni di livelli di tensione elettrica applicate ai fili con una definita scansione temporale. Quando un dispositivo *A* ha un cavo che si connette a un dispositivo *B* e il dispositivo *B* ha un cavo che si connette a un dispositivo *C* che a sua volta è collegato a una porta

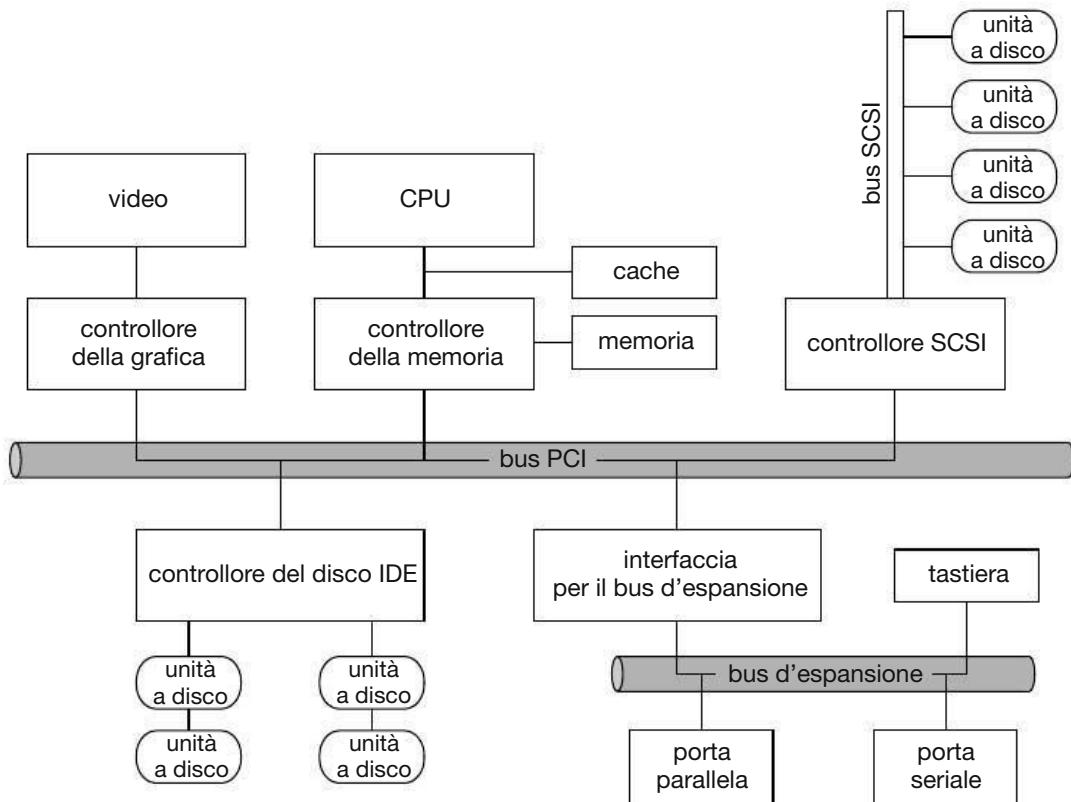


Figura 13.1 Tipica struttura del bus di un PC.

di un calcolatore, si ottiene il cosiddetto **collegamento in daisy chain**, che di solito funziona come un bus.

I bus sono ampiamente usati nell'architettura dei calcolatori e differiscono tra loro per formato dei segnali, velocità, throughput e metodo di connessione. La Figura 13.1 mostra una tipica struttura di bus di PC; si tratta di un **bus PCI** (il comune bus di sistema dei PC) che connette il sottosistema CPU-memoria ai dispositivi veloci, e di un **bus d'espansione** cui si connettono i dispositivi relativamente lenti come la tastiera e le porte seriali e USB. Nella parte superiore destra della figura quattro dischi sono collegati a un bus SCSI (*small computer system interface*) inserito nel relativo controllore. Altri tipi comuni di bus usati per connettere i principali componenti di un computer includono il **PCI Express** (PCIe), con throughput massimo di 16 GB al secondo e **HyperTransport**, con throughput che arriva fino a 25 GB al secondo.

Un **controllore** è un insieme di componenti elettronici che può far funzionare una porta, un bus o un dispositivo. Un controllore di porta seriale è un semplice controllore di dispositivo; si tratta di un singolo circuito integrato (o di una sua parte) nel calcolatore che controlla i segnali presenti nei fili della porta seriale. Per contro un controllore SCSI non è semplice; poiché il protocollo SCSI è complesso, il controllore del bus SCSI è spesso realizzato come una scheda hardware separata, un **adattatore**, che s'inserisce nel calcolatore. Esso contiene generalmente un'unità d'elaborazione, microcodice, e memoria privata che gli consentono di elaborare i messaggi del protocollo SCSI. Alcuni

dispositivi sono dotati di propri controllori incorporati. Osservando un’unità a disco si vede, da un lato, una scheda elettronica a essa agganciata; si tratta del controllore che attua la parte lato disco del protocollo di qualche tipo di connessione, per esempio SCSI o SATA (*serial advanced technology attachment*). Ha un’unità d’elaborazione e microcodice per l’esecuzione di molti compiti, come localizzazione dei settori difettosi, prelievo anticipato (*prefetching*), gestione del buffer e della cache.

L’unità d’elaborazione fornisce comandi e dati al controllore per portare a termine trasferimenti di I/O tramite uno o più registri per dati e segnali di controllo. La comunicazione con il controllore avviene attraverso la lettura e la scrittura, da parte dell’unità d’elaborazione, di configurazioni di bit in questi registri. Un modo in cui questa comunicazione può avvenire è tramite l’uso di speciali istruzioni di I/O che specificano il trasferimento di un byte o una parola a un indirizzo di porta di I/O. L’istruzione di I/O attiva le linee di bus per selezionare il giusto dispositivo e trasferire bit dentro o fuori dal registro di dispositivo. In alternativa, il controllore di dispositivo può supportare l’**I/O memory mapped** (*mappato in memoria*). In questo caso i registri di controllo del dispositivo sono mappati in un sottoinsieme dello spazio d’indirizzi della CPU, che esegue le richieste di I/O usando le ordinarie istruzioni di trasferimento di dati per leggere e scrivere i registri di controllo del dispositivo alle locazioni di memoria fisica in cui sono mappati.

Certi sistemi usano entrambe le tecniche. I PC per esempio usano istruzioni di I/O per controllare alcuni dispositivi e l’I/O memory mapped per controllarne altri. Nella Figura 13.2 sono riportati gli usuali indirizzi delle porte di I/O dei PC. Il controllore della grafica utilizza alcune porte di I/O per le operazioni di controllo di base, ma dispone di un’ampia regione mappata in memoria che serve a mantenere i contenuti dello schermo. Il processo scrive sullo schermo inserendo i dati nella regione mappata in memoria; il controllore genera l’immagine dello schermo sulla base del contenuto

indirizzi per l’I/O (in esadecimale)	dispositivo
000-00F	controllore DMA
020-021	controllore delle interruzioni
040-043	timer
200-20F	controllore dei giochi
2F8-2FF	porta seriale (secondaria)
320-32F	controllore del disco
378-37F	porta parallela
3D0-3DF	controllore della grafica
3F0-3F7	controllore dell’unità a dischetti
3F8-3FF	porta seriale (principale)

Figura 13.2 Indirizzi delle porte dei dispositivi di I/O nei PC (elenco parziale).

di questa regione di memoria. Questa tecnica è semplice da usare; inoltre la scrittura di milioni di byte nella memoria grafica è più veloce dell'invio di milioni di istruzioni di I/O. La facilità di scrittura in un controllore di I/O memory mapped è però controbilanciata da uno svantaggio: un comune errore di programmazione è la scrittura in una regione di memoria sbagliata causata da un puntatore errato. Ciò rende i registri dei dispositivi mappati in memoria vulnerabili a modifiche accidentali. Naturalmente, le tecniche di protezione della memoria aiutano a ridurre tale rischio.

Una porta di I/O consiste in genere di quattro registri: **status**, **control**, **data-in** e **data-out**.

- La CPU legge dal registro **data-in** per ricevere dati.
- La CPU scrive nel registro **data-out** per emettere dati.
- Il registro **status** contiene alcuni bit che possono essere letti dalla CPU e indicano lo stato della porta; per esempio indicano se è stata portata a termine l'esecuzione del comando corrente, se un byte è disponibile per essere letto dal registro **data-in**, se si è verificato un errore del dispositivo.
- Il registro **control** può essere scritto per attivare un comando o per cambiare il modo di funzionamento del dispositivo. Per esempio, un certo bit nel registro **control** della porta seriale determina il tipo di comunicazione tra *half-duplex* e *full-duplex*, un altro abilita il controllo di parità, un terzo imposta la lunghezza delle parole a 7 o 8 bit, altri selezionano una tra le velocità che la porta seriale può sostenere.

La tipica dimensione dei registri di dati varia tra 1 e 4 byte. Certi controllori hanno circuiti integrati FIFO che possono contenere parecchi byte per l'invio e la ricezione di dati, in modo da espandere la capacità del controllore oltre la dimensione del registro di dati. Un circuito integrato FIFO può contenere una piccola sequenza di dati finché il dispositivo o la CPU non sono in grado di riceverli.

13.2.1 Polling

Il protocollo completo per l'interazione fra la CPU e un controllore può essere intricato, ma la fondamentale nozione di **handshaking** (*negoziazione*) è semplice, ed è illustrata con un esempio. Si assuma l'uso di due bit per coordinare la relazione di tipo produttore-consamatore fra il controllore e la CPU. Il controllore specifica il suo stato per mezzo del bit **busy** del registro **status**; pone a 1 il bit **busy** quando è impegnato in un'operazione, e lo pone a 0 quando è pronto a eseguire il comando successivo. La CPU comunica le sue richieste tramite il bit **command-ready** nel registro **command**: pone questo bit a 1 quando il controllore deve eseguire un comando. In questo esempio, la CPU scrive in una porta coordinandosi con un controllore per mezzo del seguente handshaking.

1. La CPU legge ripetutamente il bit **busy** finché questo non vale 0.
2. La CPU pone a 1 il bit **write** del registro dei comandi e scrive un byte nel registro **data-out**.

3. La CPU pone a 1 il bit `command-ready`.
4. Quando il controllore si accorge che il bit `command-ready` è posto a 1, pone a 1 il bit `busy`.
5. Il controllore legge il registro dei comandi e trova il comando `write`; legge il registro `data-out` per ottenere il byte da scrivere, e compie l’operazione di I/O sul dispositivo.
6. Il controllore pone a 0 il bit `command-ready`, pone a 0 il bit `error` nel registro `status` per indicare che l’operazione di I/O ha avuto esito positivo, e pone a 0 il bit `busy` per indicare che l’operazione è terminata.

La sequenza appena descritta si ripete per ogni byte.

Durante l’esecuzione del passo 1, la CPU è in **attesa attiva** (*busy-waiting*) o in **interrogazione ciclica** (*polling*): itera la lettura del registro `status` finché il bit `busy` assume il valore 0. Se il controllore e il dispositivo sono veloci, questo metodo è ragionevole, ma se l’attesa rischia di prolungarsi, sarebbe probabilmente meglio se la CPU si dedicasse a un’altra operazione. In questo caso si pone il problema di come la CPU possa sapere quando il controllore è tornato libero. È necessario che la CPU serva certi tipi di dispositivi rapidamente, o si potrebbero perdere alcuni dati. Quando, per esempio, i dati affluiscono in una porta seriale o dalla tastiera, il piccolo buffer del controllore diverrà presto pieno, e se la CPU attende troppo a lungo prima di riprendere la lettura dei byte, si perderanno informazioni.

In molte architetture di calcolatori sono sufficienti tre istruzioni della CPU per effettuare il polling di un dispositivo: `read`, lettura di un registro del dispositivo; `logical-and`, usata per estrarre il valore di un bit di stato, e `branch`, salto a un altro punto del codice se l’argomento è diverso da zero. Chiaramente, il polling è in sé un’operazione efficiente; tale tecnica diviene però inefficiente se le ripetute interrogazioni trovano raramente un dispositivo pronto per il servizio, mentre altre utili elaborazioni attendono la CPU. In tali casi, anziché richiedere alla CPU di eseguire il polling, può essere più efficiente far sì che il controllore comunichi alla CPU che il dispositivo è pronto. Il meccanismo hardware che permette tale comunicazione si chiama **interruzione** (*interrupt*).

13.2.2 Interruzioni

Il meccanismo di base dell’interruzione funziona come segue. L’hardware della CPU ha un input, detto **linea di richiesta dell’interruzione**, del quale la CPU controlla lo stato dopo l’esecuzione di ogni istruzione. Quando rileva il segnale di un controllore sulla linea di richiesta dell’interruzione, la CPU salva lo stato corrente e salta alla **routine di gestione dell’interruzione** (*interrupt-handler routine*), che si trova a un indirizzo prefissato di memoria. Questa procedura determina le cause dell’interruzione, porta a termine l’elaborazione necessaria, ripristina lo stato ed esegue un’istruzione `return from interrupt` per far sì che la CPU ritorni nello stato in cui si trovava prima della sua interruzione. Il controllore del dispositivo *genera* un’interruzione della CPU sulla linea di richiesta delle interruzioni, che la CPU *rileva e recapita*

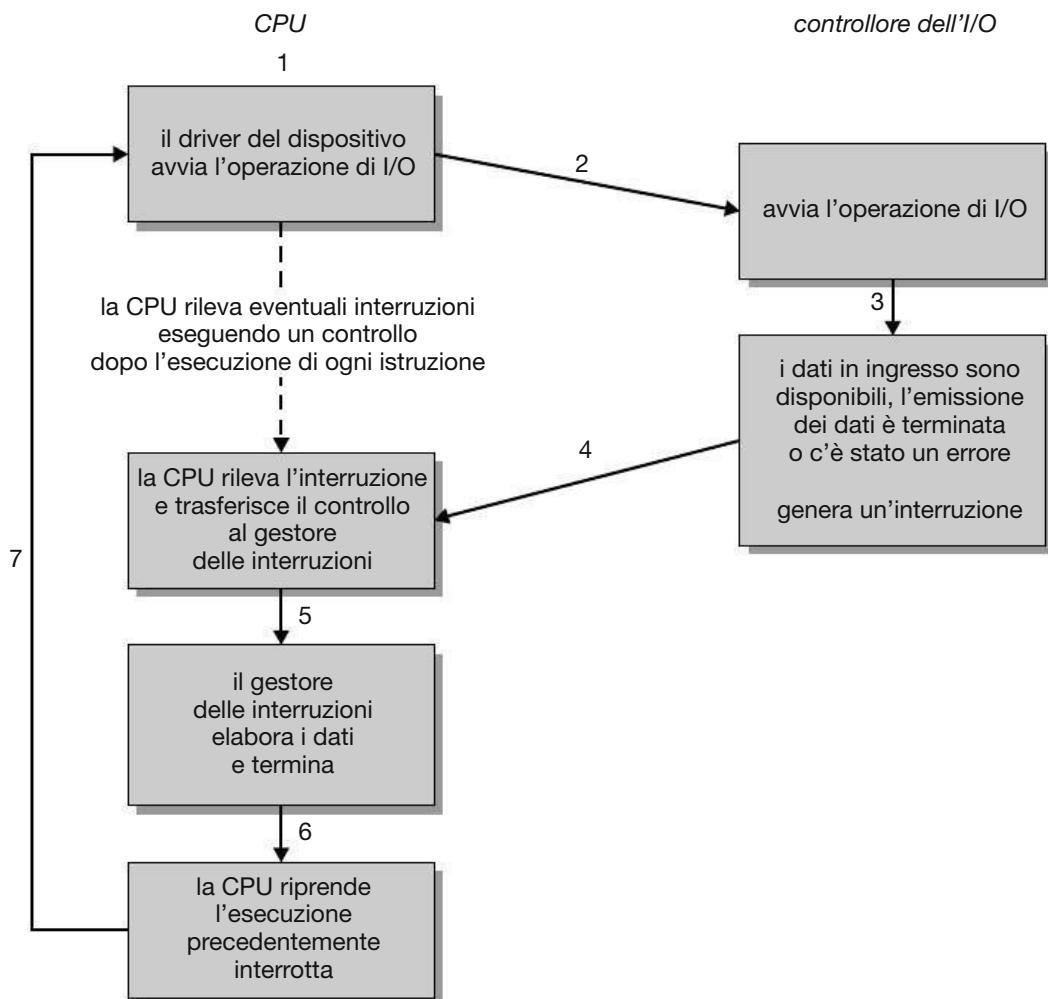


Figura 13.3 Ciclo di I/O basato sulle interruzioni.

al gestore delle interruzioni, che a sua volta *gestisce* l'interruzione corrispondente servendo il dispositivo. Nella Figura 13.3 è riassunto il ciclo di I/O causato da un'interruzione della CPU. In questo capitolo daremo molta importanza alla gestione delle interruzioni, perché persino un sistema a singola utenza ne gestisce centinaia al secondo, mentre un server ne può gestire centinaia di migliaia.

Il meccanismo di base delle interruzioni che abbiamo appena descritto permette alla CPU di rispondere a un evento asincrono, come quello di un controllore di un dispositivo che divenga pronto per essere servito. Nei sistemi operativi moderni sono necessarie capacità di gestione delle interruzioni più raffinate.

1. Si deve poter posporre la gestione delle interruzioni durante le fasi critiche dell'elaborazione.
2. Si deve disporre di un meccanismo efficiente per passare il controllo all'appropriato gestore delle interruzioni, senza dover esaminare ciclicamente tutti i dispositivi (*polling*) per determinare quale abbia generato l'interruzione.

3. Si deve disporre di più livelli d'interruzione, di modo che il sistema possa distinguere le interruzioni ad alta priorità da quelle a priorità inferiore, servendo le richieste con la celerità appropriata al caso.

In un calcolatore moderno queste tre caratteristiche sono fornite dalla CPU e dal **controllore hardware delle interruzioni**.

La maggior parte delle CPU ha due linee di richiesta delle interruzioni. Una è quella delle **interruzioni non mascherabili**, riservata a eventi quali gli errori di memoria irrecuperabili. La seconda linea è quella delle **interruzioni mascherabili**: può essere disattivata dalla CPU prima dell'esecuzione di una sequenza critica di istruzioni che non deve essere interrotta. L'interruzione mascherabile è usata dai controllori dei dispositivi per richiedere un servizio.

Il meccanismo delle interruzioni accetta un **indirizzo** – un numero che seleziona una specifica procedura di gestione delle interruzioni da un insieme ristretto. Nella maggior parte delle architetture questo indirizzo è uno scostamento relativo in una tabella detta **vettore delle interruzioni**, contenente gli indirizzi di memoria degli specifici gestori delle interruzioni. Lo scopo di un meccanismo vettorizzato di gestione delle interruzioni è di ridurre la necessità che un singolo gestore debba individuare tutte le possibili fonti d'interruzione per determinare quale di loro abbia richiesto un servizio. In pratica, tuttavia, i calcolatori hanno più dispositivi (e quindi, più gestori delle interruzioni) che elementi nel vettore delle interruzioni. Una maniera diffusa di risolvere questo problema consiste nel **concatenamento delle interruzioni** (*interrupt chaining*), in cui ogni elemento del vettore delle interruzioni punta alla testa di una lista di gestori delle interruzioni. Quando si verifica un'interruzione, si chiamano uno alla volta i gestori nella lista corrispondente finché non se ne trova uno che può soddisfare la richiesta. Questa struttura è un compromesso fra l'overhead di una tabella delle interruzioni enorme e l'inefficienza dell'uso di un solo gestore delle interruzioni.

Nella Figura 13.4 è descritto il vettore delle interruzioni della CPU Intel Pentium. Gli eventi da 0 a 31, non mascherabili, si usano per segnalare varie condizioni d'errore; quelli dal 32 al 255, mascherabili, si usano, per esempio, per le interruzioni generate dai dispositivi.

Il meccanismo delle interruzioni realizza anche un sistema di **livelli di priorità delle interruzioni**. Esso permette alla CPU di differire la gestione delle interruzioni di bassa priorità senza mascherare tutte le interruzioni, e permette a un'interruzione di priorità alta di sospendere l'esecuzione della procedura di servizio di un'interruzione di priorità bassa.

Un sistema operativo moderno interagisce con il meccanismo delle interruzioni in vari modi. All'accensione della macchina esamina i bus per determinare quali dispositivi siano presenti, e installa gli indirizzi dei corrispondenti gestori delle interruzioni nel vettore delle interruzioni. Durante l'I/O, i vari controllori di dispositivi generano le interruzioni della CPU quando sono pronti per un servizio. Queste interruzioni significano che è stato completato un output, o che sono disponibili dati in ingresso, o che un'operazione non è andata a buon fine. Il meccanismo delle interruzioni si usa anche per gestire un'ampia gamma di **eccezioni**, come la divisione

indice del vettore	descrizione
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19-31	(Intel reserved, do not use)
32-255	maskable interrupts

Figura 13.4 Vettore delle interruzioni della CPU Intel Pentium.

per 0, l'accesso a indirizzi di memoria protetti o inesistenti o il tentativo di eseguire un'istruzione privilegiata in modalità utente. Gli eventi che producono le interruzioni hanno una proprietà in comune: inducono il sistema operativo a eseguire urgentemente una procedura autonoma.

Un sistema operativo può fare altri usi proficui di un efficiente meccanismo hardware e software che memorizza una piccola quantità d'informazioni sullo stato della CPU e poi richiama una procedura del kernel. Per esempio, molti sistemi operativi usano il meccanismo delle interruzioni per la gestione della memoria virtuale. Un'eccezione di pagina mancante genera un'interruzione che sospende il processo corrente e trasferisce il controllo dell'esecuzione al relativo gestore nel kernel. Tale gestore memorizza le informazioni sullo stato del processo, lo sposta nella coda d'attesa, compie le necessarie operazioni di gestione della cache delle pagine, avvia un'operazione di I/O per prelevare la pagina giusta, schedula la ripresa dell'esecuzione di un altro processo e ritorna dall'interruzione.

Un altro esempio è dato dall'implementazione delle *chiamate di sistema*. Solitamente i programmi sfruttano routine di libreria per eseguire chiamate di sistema. La routine controlla i parametri passati dall'applicazione, li assembla in una struttura dati appropriata da passare al kernel, e infine esegue una particolare istruzione detta **interruzione software** o **trap**. Questa istruzione ha un operando che identifica il servizio del kernel desiderato. Quando un processo esegue l'istruzione di eccezione, l'hardware delle interruzioni salva lo stato del codice utente, passa al modo kernel e recapita l'interruzione alla procedura del kernel che realizza il servizio richiesto.

A una trap si assegna una priorità di interruzione relativamente bassa rispetto a quelle date alle interruzioni dei dispositivi – eseguire una chiamata di sistema per conto di un'applicazione è meno urgente di quanto non sia servire un controllore prima che la sua coda FIFO trabocchi causando la perdita di informazioni.

Le interruzioni si possono inoltre usare per gestire il flusso di controllo all'interno del kernel. Si consideri un esempio di elaborazione richiesta per completare una lettura da un disco. Un passo necessario è quello di copiare dati dallo spazio del kernel al buffer dell'utente. Questa azione richiede tempo, ma non è urgente, e non dovrebbe bloccare la gestione delle interruzioni con priorità più alta. Un altro passo è quello di avviare il successivo I/O in attesa relativo a quell'unità a disco. Questo passo ha priorità più alta: se le unità a disco si devono usare in modo efficiente, è necessario avviare l'evasione della successiva richiesta di I/O non appena la precedente sia stata soddisfatta. Di conseguenza una *coppia* di gestori delle interruzioni realizza il codice del kernel che compie le letture dai dischi. Il gestore ad alta priorità mantiene le informazioni sullo stato dell'I/O, risponde al segnale d'interruzione del dispositivo, avvia il prossimo I/O in attesa e genera un'interruzione a bassa priorità per completare il lavoro. Più tardi, in un momento in cui la CPU non è occupata in compiti ad alta priorità, si serve l'interruzione a bassa priorità. Il gestore corrispondente completa l'I/O a livello utente copiando i dati dal buffer del kernel a quello dell'applicazione, e richiamando poi lo scheduler per aggiungere l'applicazione alla coda dei processi pronti.

Un'architettura del kernel basata su thread è adatta alla realizzazione di più livelli di priorità delle interruzioni, e a dare la precedenza alla gestione delle interruzioni rispetto alle elaborazioni in background delle procedure del kernel e delle applicazioni. Questo concetto si può esemplificare considerando il kernel del sistema operativo Solaris. In questo sistema i gestori delle interruzioni si eseguono come thread del kernel cui si riservano valori elevati di priorità. Tali priorità garantiscono la precedenza dei gestori delle interruzioni rispetto al codice delle applicazioni e al lavoro ordinario del kernel, e inoltre realizzano le necessarie relazioni di priorità fra i diversi gestori delle interruzioni. Le priorità fanno sì che lo scheduler dei thread di Solaris sospenda i gestori delle interruzioni di bassa priorità a vantaggio di quelli di priorità più alta, e la realizzazione basata su thread permette alle architetture multiprocessore di eseguire parallelamente diversi gestori delle interruzioni. L'architettura delle interruzioni dei sistemi Windows è descritta nel Capitolo 19, presente sulla pagina web del volume.

Riassumendo, le interruzioni sono usate diffusamente dai sistemi operativi moderni per gestire eventi asincroni e per eseguire procedure in modalità supervisore nel kernel. Per far sì che i compiti più urgenti siano portati a termine per primi, i calcolatori moderni usano un sistema di priorità delle interruzioni. I controllori dei dispositivi, i guasti hardware e le chiamate di sistema generano interruzioni al fine di innescare l'esecuzione di procedure del kernel. Poiché le interruzioni sono usate in modo massiccio per affrontare situazioni in cui il tempo è un fattore critico, è necessario avere un'efficiente gestione delle interruzioni per ottenere buone prestazioni del sistema.

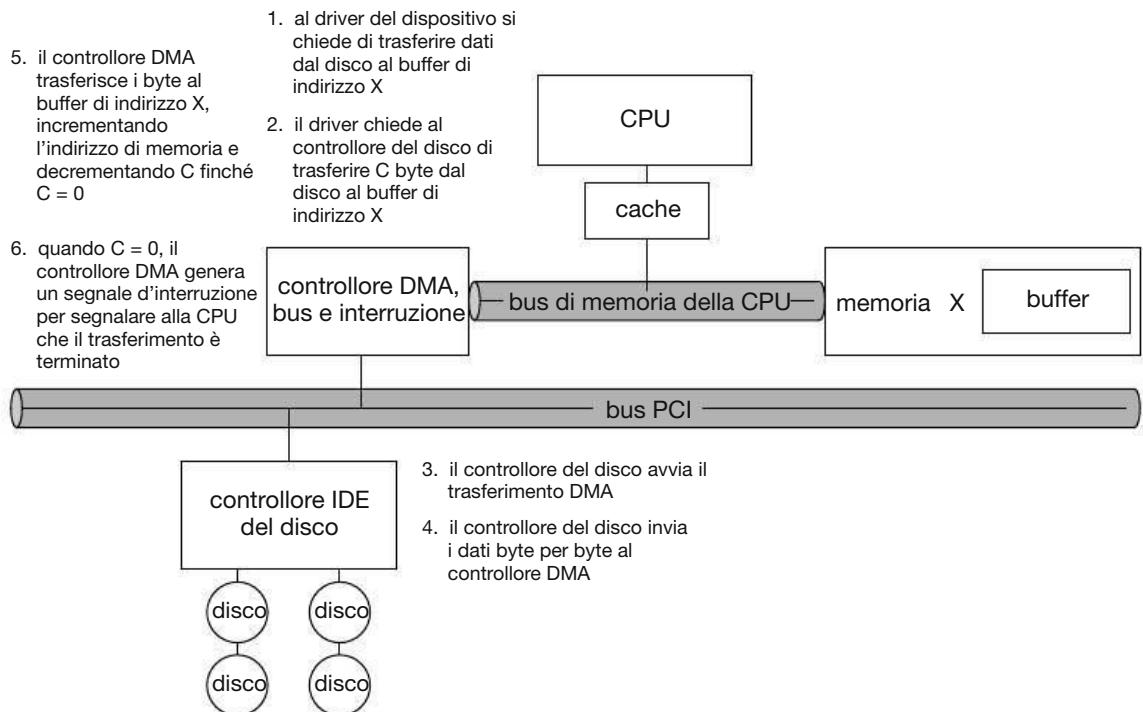
13.2.3 Accesso diretto alla memoria (DMA)

Quando un dispositivo compie trasferimenti di grandi quantità di dati, come nel caso di un'unità a disco, l'uso di una costosa CPU per il controllo dei bit di stato e per la scrittura di dati nel registro del controllore un byte alla volta, detto **I/O programmato** (*programmed I/O, PIO*), sembra essere uno spreco. In molti calcolatori si evita di sovraccaricare la CPU assegnando una parte di questi compiti a un processore specializzato, detto controllore dell'**accesso diretto alla memoria** (*direct memory-access, DMA*). Per dar avvio a un trasferimento DMA, la CPU scrive in memoria un blocco di comando per il DMA. Esso contiene un puntatore alla locazione dei dati da trasferire, un altro puntatore alla destinazione dei dati, e il numero dei byte da trasferire. La CPU scrive l'indirizzo di questo blocco di comando nel controllore del DMA, e prosegue con altre attività. Il controllore DMA agisce quindi direttamente sul bus della memoria, presentando al bus gli indirizzi di memoria necessari per eseguire il trasferimento senza l'aiuto della CPU. Un semplice controllore DMA è un componente standard in tutti i sistemi moderni, dagli smartphone ai mainframe.

L'handshaking tra il controllore del DMA e il controllore del dispositivo si svolge grazie a una coppia di fili detti **DMA-request** e **DMA-acknowledge**. Il controllore del dispositivo manda un segnale sulla linea **DMA-request** quando una word di dati è disponibile per il trasferimento. Questo segnale fa sì che il controllore DMA prenda possesso del bus di memoria, presenti l'indirizzo desiderato ai fili d'indirizzamento della memoria e mandi un segnale lungo la linea **DMA-acknowledge**. Quando il controllore del dispositivo riceve questo segnale, trasferisce in memoria la word di dati e rimuove il segnale dalla linea **DMA-request**.

Quando l'intero trasferimento termina, il controllore del DMA interrompe la CPU. Nella Figura 13.5 è rappresentato questo processo. Quando il controllore del DMA prende possesso del bus di memoria, la CPU è temporaneamente impossibilitata ad accedere alla memoria centrale, sebbene abbia accesso ai dati contenuti nella sua cache primaria e secondaria. Questo fenomeno, noto come **sottrazione di cicli**, può rallentare le computazioni della CPU; ciononostante l'assegnamento del lavoro di trasferimento di dati a un controllore DMA migliora in generale le prestazioni complessive del sistema. In alcune architetture per realizzare la tecnica DMA si usano gli indirizzi della memoria fisica, mentre in altre s'impiega l'**accesso diretto alla memoria virtuale** (*direct virtual-memory access, DVMA*): in questo caso si usano indirizzi virtuali che poi si traducono in indirizzi fisici. La tecnica DVMA permette di compiere i trasferimenti di dati tra due dispositivi che eseguono I/O memory mapped senza far intervenire la CPU o accedere alla memoria centrale.

Nei kernel che operano in modalità protetta, in genere il sistema operativo non permette ai processi di impartire direttamente comandi ai dispositivi. Ciò protegge i dati dalle violazioni dei controlli d'accesso e il sistema da un eventuale uso scorretto dei controllori dei dispositivi che potrebbe portare a una caduta del sistema stesso. Il sistema operativo invece esporta delle funzioni di I/O che possono essere eseguite da processi sufficientemente privilegiati per effettuare operazioni di basso livello sull'hardware sottostante. Quando invece il kernel non può garantire la protezione della

**Figura 13.5** Passi di un trasferimento DMA.

memoria, i processi hanno accesso diretto ai controllori dei dispositivi. Questo accesso diretto si può utilizzare in modo da ottenere buone prestazioni, perché evita la comunicazione col kernel, i cambi di contesto e l’interazione fra diversi livelli del kernel. Purtroppo interferisce con la stabilità e la sicurezza del sistema. La tendenza comune per i sistemi operativi d’uso generale è quella di proteggere la memoria e i dispositivi in modo da salvaguardarli da applicazioni accidentalmente o volutamente dannose.

13.2.4 Concetti principali dell’hardware di I/O

Sebbene gli aspetti dell’I/O che riguardano i dispositivi siano complessi se si analizzano tanto dettagliatamente quanto farebbe un progettista elettronico, i concetti appena descritti sono sufficienti per comprendere molti aspetti dell’I/O per ciò che concerne i sistemi operativi. Ecco un sommario dei concetti principali:

- bus;
- controllore;
- porta di I/O e suoi registri;
- procedura di handshaking tra la CPU e il controllore di un dispositivo;
- esecuzione dell’handshaking per mezzo del polling o delle interruzioni;
- delega dell’I/O a un controllore DMA nel caso di trasferimenti di grandi quantità di dati.

Precedentemente in questo paragrafo è stato fornito un esempio dell'*handshaking* che avviene tra un controllore di dispositivo e un host. In realtà, la grande varietà di dispositivi esistenti pone un problema a chi voglia realizzare concretamente un sistema operativo. Ogni tipo di dispositivo ha proprie funzionalità, proprie definizioni dei bit di controllo, e un proprio protocollo per l'interazione con la macchina – e tutto ciò varia da dispositivo a dispositivo. Come deve essere progettato un sistema operativo affinché sia possibile collegare al calcolatore nuovi dispositivi senza che sia necessario riscrivere il sistema operativo stesso? E inoltre, vista la grande varietà di dispositivi, come può il sistema operativo fornire alle applicazioni un'interfaccia per l'I/O uniforme ed efficace? Discuteremo questi aspetti nel seguito.

13.3 Interfaccia di I/O per le applicazioni

In questo paragrafo si discutono le tecniche e le interfacce di un sistema operativo che permettono un trattamento standardizzato e uniforme dei dispositivi di I/O. Si spiega, per esempio, come un'applicazione possa aprire un file residente in un disco senza sapere di che tipo di disco si tratti, e come si possano aggiungere al calcolatore nuove unità a disco e altri dispositivi senza che si debba modificare il sistema operativo.

I metodi qui esposti coinvolgono l'astrazione, l'incapsulamento e la stratificazione del software. In particolare, si può effettuare un'astrazione rispetto ai dettagli delle differenze tra i dispositivi per l'I/O identificandone alcuni tipi generali. A ognuno di questi tipi si accede per mezzo di un insieme standardizzato di funzioni – un'**interfaccia**. Le differenze sono incapsulate in moduli del kernel detti driver dei dispositivi, specializzati internamente per gli specifici dispositivi, ma che comunicano con l'esterno per mezzo delle interfacce uniformi. Nella Figura 13.6 è illustrata la divisione in strati software di quelle parti del kernel che riguardano la gestione dell'I/O.

Lo scopo dello strato dei driver dei dispositivi è di nascondere al sottosistema di I/O del kernel le differenze fra i controllori dei dispositivi, in modo simile a quello con cui le chiamate di sistema di I/O incapsulano il comportamento dei dispositivi in alcune classi generiche che nascondono le differenze hardware alle applicazioni. Il fatto che così il sottosistema di I/O sia reso indipendente dall'hardware semplifica il lavoro di chi sviluppa il sistema operativo, e va inoltre a vantaggio dei costruttori dei dispositivi. Questi, infatti, o progettano i nuovi dispositivi in modo tale che siano compatibili con un'interfaccia host-controllore già esistente (per esempio SATA), oppure scrivono driver che permettano ai nuovi dispositivi di essere gestiti dai sistemi operativi più diffusi. In questo modo, nuovi dispositivi sono utilizzabili da un calcolatore senza che occorra attendere lo sviluppo del codice di supporto da parte del produttore del sistema operativo.

Sfortunatamente per i produttori di dispositivi, ogni tipo di sistema operativo ha le sue convenzioni riguardanti l'interfaccia dei driver dei dispositivi. Così, un dato dispositivo potrà essere venduto con molti driver diversi – per esempio, driver per Windows, Linux, AIX e Mac OS X. I dispositivi (Figura 13.7) possono differire in molti aspetti.

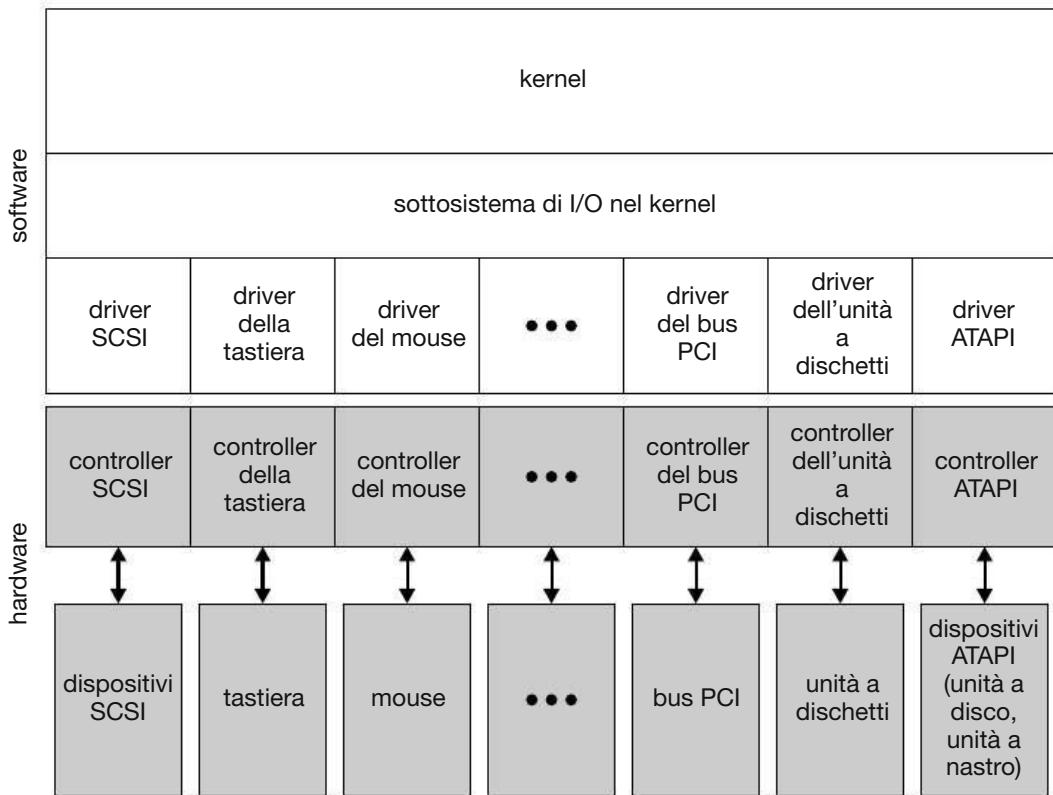


Figura 13.6 Struttura relativa all'I/O nel kernel.

- **Trasferimento a flusso di caratteri o a blocchi.** Un dispositivo a flusso di caratteri trasferisce dati un byte alla volta, mentre uno a blocchi trasferisce un blocco di byte in un'unica soluzione.
- **Sequenziale o accesso diretto.** Un dispositivo sequenziale trasferisce dati secondo un ordine fisso dipendente dal dispositivo, mentre l'utente di un dispositivo ad accesso diretto può richiedere l'accesso a una qualunque delle possibili locazioni di memorizzazione.
- **Dispositivi sincroni o asincroni.** Un dispositivo sincrono trasferisce dati con un tempo di risposta prevedibile, in maniera coordinata rispetto al resto del sistema. Un dispositivo asincrono ha tempi di risposta irregolari o non prevedibili, non coordinati con gli altri eventi del computer.
- **Condivisibili o dedicati.** Un dispositivo condivisibile può essere usato in modo concorrente da diversi processi o thread, mentre ciò è impossibile se il dispositivo è dedicato.
- **Velocità di funzionamento.** Può variare da alcuni byte al secondo fino a qualche gigabyte al secondo.
- **Lettura e scrittura, sola lettura o sola scrittura.** Alcuni dispositivi possono emettere e ricevere dati, ma altri possono trasferire dati in una sola direzione.

aspetto	variazione	esempio
modalità di trasferimento dei dati	a caratteri a blocchi	terminale unità a disco
modalità d'accesso	sequenziale casuale	modem lettore di CD-ROM
prevedibilità dell'I/O	sincrono asincrono	unità a nastro tastiera
condivisione	dedicato condiviso	unità a nastro tastiera
velocità	latenza tempo di ricerca velocità di trasferimento attesa fra le operazioni	
direzione dell'I/O	solo lettura solo scrittura lettura e scrittura	lettore di CD-ROM controllore della grafica unità a disco

Figura 13.7 Caratteristiche dei dispositivi per l'I/O.

Per ciò che riguarda l'accesso delle applicazioni ai dispositivi, molte di queste differenze sono nascoste dal sistema operativo, e i dispositivi sono raggruppati in poche classi convenzionali. Si è riscontrato che i modi d'accesso ai dispositivi che ne risultano sono utili e largamente applicabili. Anche se la forma precisa delle chiamate di sistema può variare nei diversi sistemi operativi, le classi di dispositivi sono abbastanza regolari. Le convenzioni d'accesso principali includono l'I/O a blocchi, l'I/O a flusso di caratteri, l'accesso ai file mappati in memoria, e le socket di rete. I sistemi operativi forniscono anche chiamate di sistema speciali per l'accesso a qualche dispositivo aggiuntivo, per esempio un orologio o un timer. Qualche sistema operativo mette a disposizione un insieme di chiamate di sistema per display grafici, video e audio.

La maggior parte dei sistemi ha anche una “via di fuga” (*escape*) o *back door* che permette il passaggio trasparente di comandi arbitrari da un'applicazione a un driver di dispositivo. In UNIX tale funzione è svolta dalla chiamata di sistema `ioctl()` (che sta per *I/O control*) e consente a un'applicazione di impiegare qualsiasi funzionalità fornita da qualsiasi driver di dispositivo, senza che per questo sia necessario creare nuove chiamate di sistema. Gli argomenti di `ioctl()` sono tre: il primo è un descrittore di file che collega l'applicazione al driver desiderato facendo riferimento a un dispositivo gestito da quel driver; il secondo è un numero intero che seleziona uno dei comandi forniti dal driver; il terzo argomento, infine, è un puntatore a un'arbitraria struttura dati in memoria, tramite la quale l'applicazione e il driver si scambiano le informazioni di controllo o i dati necessari.

13.3.1 Dispositivi con trasferimento a blocchi e a caratteri

L’interfaccia per i **dispositivi a blocchi** sintetizza tutti gli aspetti necessari per accedere alle unità a disco e ad altri dispositivi basati sul trasferimento di blocchi di dati. Ci si aspetta che il dispositivo comprenda istruzioni come `read()` e `write()` e, nel caso sia ad accesso casuale, anche un comando `seek()` per specificare il blocco successivo da trasferire. Di solito le applicazioni comunicano con questi dispositivi tramite un’interfaccia di file system. Si vede che `read()`, `write()` e `seek()` catturano l’essenza del comportamento dei dispositivi con trasferimento a blocchi, in modo che le applicazioni non vedano le differenze di basso livello fra questi dispositivi.

Il sistema operativo e certe applicazioni particolari come quelle per la gestione delle basi di dati possono trovare più conveniente trattare questi dispositivi come una semplice sequenza lineare di blocchi. In questo caso si parla di **I/O a basso livello** (*raw I/O*). L’uso del file system da parte delle applicazioni che gestiscono già in proprio un buffer per l’I/O comporta l’inutile intervento di un buffer aggiuntivo. Analogamente, l’uso dei lock da parte del sistema operativo nei confronti di applicazioni che già implementano meccanismi per la mutua esclusione relativi ai blocchi dei file risulta ridondante, o peggio contraddittorio. Per superare tali potenziali conflitti, l’I/O a basso livello passa il controllo del dispositivo direttamente all’applicazione, togliendo così di mezzo il sistema operativo. Purtroppo, ciò significa anche che non risulta disponibile sul dispositivo in questione alcun servizio del sistema operativo. Un compromesso che si sta diffondendo sempre più è fornire una modalità d’accesso ai file che disabiliti il buffer e i meccanismi di gestione dei lock; nel mondo UNIX si parla di **I/O diretto**.

L’accesso ai file mappato in memoria può costituire uno strato software sopra i driver dei dispositivi a blocchi. Piuttosto che offrire funzioni di lettura e scrittura, un’interfaccia di mappaggio in memoria fornisce la possibilità di accedere a un’unità a disco tramite un vettore di byte della memoria centrale. La chiamata di sistema che associa un file a una regione di memoria restituisce l’indirizzo di memoria virtuale di una copia del file. Gli effettivi trasferimenti di dati sono eseguiti solo quando necessari per soddisfare una richiesta d’accesso all’immagine in memoria. Poiché i trasferimenti si trattano nello stesso modo in cui si gestisce l’accesso su richiesta a una pagina di memoria virtuale, l’I/O memory mapped è efficiente. Esso è inoltre conveniente per i programmati perché l’accesso a un file memory mapped è semplice tanto quanto la lettura e la scrittura in memoria. I sistemi operativi che gestiscono la memoria virtuale utilizzano comunemente l’interfaccia di mappaggio in memoria per i servizi del kernel. Per esempio, quando il sistema operativo deve eseguire un programma, mappa l’eseguibile in memoria, quindi trasferisce il controllo all’indirizzo iniziale. Questo tipo d’interfaccia è spesso usato anche per l’accesso del kernel all’area di swapping nei dischi.

La tastiera è un esempio di dispositivo al quale si accede tramite un’interfaccia a **flusso di caratteri**. Le chiamate di sistema fondamentali per le interfacce di questo tipo permettono a un’applicazione di acquisire (`get()`) o inviare (`put()`) un carat-

tere. Basandosi su quest’interfaccia è possibile costruire librerie che offrono l’accesso riga per riga, con buffering ed editing (per esempio, quando l’utente preme il tasto backspace, si rimuove il carattere precedente dalla sequenza di caratteri da inserire). Questo tipo d’accesso è conveniente per dispositivi come tastiere, mouse, modem, che producono dati “spontaneamente”, cioè in momenti che non possono essere sempre previsti dalle applicazioni. Lo stesso tipo d’accesso è adatto anche ai dispositivi che emettono dati organizzati in modo naturale come sequenza lineare di byte, per esempio le stampanti o le schede audio.

13.3.2 Dispositivi di rete

Poiché i modi di indirizzamento e le prestazioni tipiche dell’I/O di rete sono notevolmente differenti da quelli dell’I/O delle unità a disco, la maggior parte dei sistemi operativi fornisce un’interfaccia per l’I/O di rete diversa da quella caratterizzata dalle operazioni `read()`, `write()` e `seek()` usata per i dischi. Un’interfaccia disponibile in molti sistemi operativi, tra i quali UNIX e Windows NT, è l’interfaccia di rete **socket** (*presa di corrente*).

Si pensi a una presa di corrente elettrica a muro: vi si può collegare qualunque apparecchiatura elettrica; per analogia, le chiamate di sistema di un’interfaccia socket permettono a un’applicazione di creare una socket, collegare una socket locale all’indirizzo di un altro punto della rete (ciò ha l’effetto di collegare questa applicazione alla socket creata da un’altra applicazione), controllare se un’applicazione si inserisce nella socket locale, e inviare o ricevere pacchetti di dati lungo la connessione. Per supportare lo sviluppo di server, l’interfaccia socket fornisce anche una funzione chiamata `select()` che gestisce un insieme di socket. Essa restituisce informazioni sulle socket per le quali sono presenti pacchetti che attendono d’essere ricevuti, e su quelle che hanno spazio per accettare un pacchetto da inviare. L’uso della funzione `select()` elimina il polling e il busy waiting altrimenti necessari per l’I/O di rete. Queste funzioni incapsulano il comportamento essenziale delle reti, facilitando notevolmente la creazione di applicazioni distribuite che possano sfruttare qualsiasi hardware di rete e stack di protocolli.

Sono stati sviluppati molti altri metodi per affrontare il problema della comunicazione di rete e fra i processi. Il sistema operativo Windows, per esempio, fornisce un’interfaccia per la scheda di rete e un’altra per i protocolli di rete. Il sistema operativo UNIX, banco di prova storico delle tecnologie di rete, offre pipe *half-duplex*, code FIFO *full-duplex*, STREAMS *full-duplex*, code di messaggi e socket.

13.3.3 Orologi e timer

La maggior parte dei calcolatori ha timer e orologi hardware che forniscono tre funzioni essenziali:

- segnare l’ora corrente;
- segnalare il tempo trascorso;
- regolare un timer in modo da avviare l’operazione *x* al tempo *t*.

Queste funzioni sono spesso usate sia dal sistema operativo sia da applicazioni per cui il tempo è un fattore importante. Purtroppo, le chiamate di sistema che realizzano queste funzioni non sono standardizzate da un sistema operativo all’altro.

Il dispositivo che misura la durata di un lasso di tempo e che può avviare un’operazione si chiama **timer programmabile**; si può regolare in modo da attendere un certo tempo e poi generare un’interruzione, e può anche ripetere questo processo continuamente, generando così interruzioni periodiche. Lo scheduler usa questo meccanismo per generare un’interruzione che sospende un processo quando il suo quanto di tempo è scaduto. Il sottosistema dell’I/O delle unità a disco lo usa per riversare periodicamente nei dischi il contenuto della *buffer cache*, e il sottosistema di rete lo usa per annullare operazioni che procedono troppo lentamente a causa di congestioni di rete o guasti. Il sistema operativo può inoltre fornire un’interfaccia per permettere ai processi utenti di usare i timer. Simulando orologi virtuali, il sistema operativo può anche gestire un numero di richieste d’uso dei timer maggiore del numero dei timer fisici. Per far ciò il kernel (o il driver del timer) mantiene una lista ordinata cronologicamente delle interruzioni richieste dagli utenti e dalle proprie procedure, e imposta il timer per la prima scadenza. Quando il timer genera l’interruzione, il kernel manda un segnale al richiedente, e reimposta il timer per la scadenza successiva.

In molti calcolatori, la frequenza delle interruzioni generate dall’orologio è fra le 18 e le 60 al secondo. Ciò costituisce un grado di precisione non sufficientemente fine per un calcolatore moderno che può eseguire centinaia di milioni di istruzioni al secondo. La precisione degli impulsi d’attivazione è limitata dalla bassa frequenza del timer, e dall’overhead aggiuntivo dato dal mantenimento di orologi virtuali. Inoltre, se lo stesso timer si usa per fornire l’ora corrente del sistema, questa potrà derivare. Nella maggior parte dei calcolatori, l’orologio hardware è costruito sulla base di un contatore ad alta frequenza. In alcuni casi, è possibile leggere da un registro del dispositivo il valore di questo contatore, cosicché esso può essere visto come un orologio ad alta precisione. Sebbene non sia in grado di generare interruzioni, offre una misura accurata degli intervalli di tempo.

13.3.4 I/O non bloccante e asincrono

Un altro aspetto dell’interfaccia delle chiamate di sistema è la scelta fra I/O bloccante e non bloccante. Quando un’applicazione impiega una chiamata di sistema **bloccante** si sospende l’esecuzione dell’applicazione, che passa dalla coda dei processi pronti per l’esecuzione a una coda d’attesa del sistema. Quando la chiamata di sistema termina, l’applicazione è posta nuovamente nella coda dei processi pronti in modo che possa riprendere l’esecuzione; Quando riprenderà l’esecuzione, riceverà i valori riportati dalla chiamata di sistema. Le operazioni fisiche compiute dai dispositivi di I/O sono in genere asincrone – richiedono un tempo variabile o non prevedibile. Cionondimeno, la maggior parte dei sistemi operativi impiega chiamate di sistema bloccanti come interfaccia per le applicazioni, perché in questo caso il codice delle applicazioni è più facilmente comprensibile del corrispondente codice non bloccante.

Alcuni processi a livello utente necessitano di una forma **non bloccante** di I/O. Un esempio è quello di un'interfaccia utente con cui s'interagisce col mouse e la tastiera mentre elabora dati e li mostra sullo schermo. Un altro esempio è un'applicazione video che legge frame da un file su disco e simultaneamente li decomprime e li mostra sullo schermo.

Uno dei modi in cui chi progetta un'applicazione può sovrapporre elaborazione e I/O è scrivere un'applicazione a più thread. Alcuni di loro eseguono chiamate di sistema bloccanti, mentre altri continuano l'elaborazione. Alcuni sistemi operativi forniscono chiamate di sistema non bloccanti per l'I/O. Una chiamata di questo tipo non arresta l'esecuzione dell'applicazione per un tempo significativo. Al contrario, essa restituisce rapidamente il controllo all'applicazione, fornendo un parametro che indica quanti byte di dati sono stati trasferiti.

Una possibile alternativa alle chiamate di sistema non bloccanti è costituita dalle chiamate di sistema asincrone. Esse restituiscono immediatamente il controllo al chiamante, senza attendere che l'I/O sia stato completato. L'applicazione continua a essere eseguita, e il completamento dell'I/O è successivamente comunicato all'applicazione per mezzo dell'impostazione del valore di una variabile nello spazio d'indirizzi dell'applicazione oppure tramite la generazione di un segnale o interrupt software, o ancora tramite una procedura di richiamo (*callback*) eseguita fuori del normale flusso lineare d'elaborazione dell'applicazione. La differenza fra chiamate di sistema non bloccanti e asincrone è che una `read()` non bloccante restituisce immediatamente il controllo, fornendo i dati che è stato possibile leggere (l'intero numero di byte richiesti, una parte, o anche nessun dato). Una chiamata `read()` asincrona richiede un trasferimento di cui il sistema garantisce il completamento, ma solo in un momento successivo e non prevedibile. Entrambi i metodi sono illustrati dalla Figura 13.8.

In tutti i moderni sistemi operativi si verificano attività asincrone. Spesso queste attività non sono gestite da utenti o applicazioni, ma fanno parte del funzionamento del sistema operativo. Due validi esempi sono l'I/O del disco e della rete. Di norma,

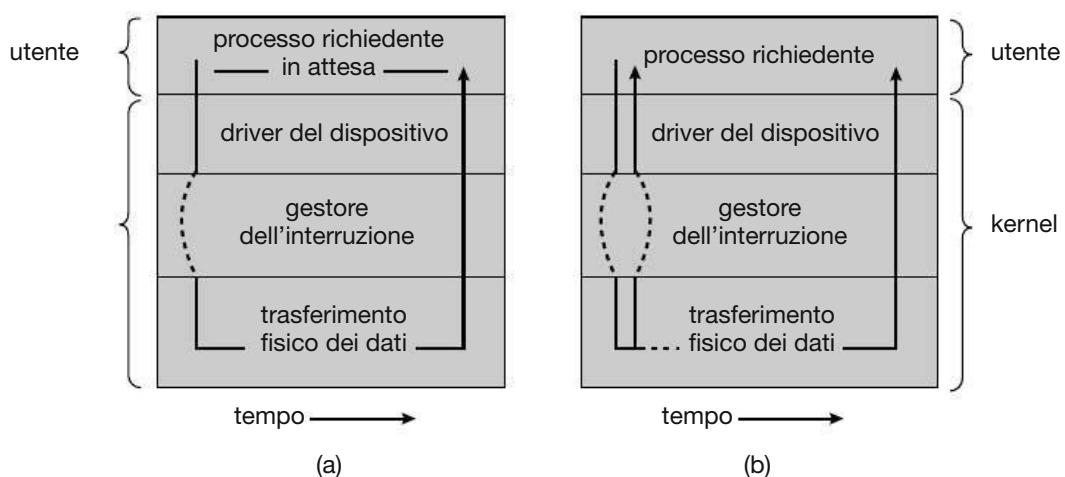


Figura 13.8 Due metodi per l'I/O; (a) sincrono e (b) asincrono.

quando un'applicazione emette una richiesta di invio sulla rete o una richiesta di scrittura su disco, il sistema operativo rileva la richiesta, inserisce l'I/O in un buffer e restituisce il controllo all'applicazione. Appena possibile, per ottimizzare le prestazioni complessive, il sistema operativo completa la richiesta. Se nel frattempo si verifica un errore di sistema, l'applicazione perderà tutte le richieste in atto. I sistemi operativi impongono pertanto, di solito, un limite sul tempo massimo per rispondere a una richiesta. Per esempio, alcune versioni di UNIX ripuliscono i propri buffer del disco ogni 30 secondi; in altre parole ogni richiesta deve essere evasa entro 30 secondi. La coerenza dei dati delle applicazioni viene mantenuta dal kernel, che legge i dati dai suoi buffer prima di inviare richieste di I/O ai dispositivi, assicurando che i dati non ancora scritti siano comunque restituiti al richiedente. Si noti che più thread che eseguono I/O sullo stesso file potrebbero non ricevere dati coerenti, a seconda di come il kernel implementa il suo I/O. In questa situazione, i thread potrebbero dover utilizzare protocolli di locking. Alcune richieste di I/O devono essere eseguite immediatamente, per cui le chiamate di sistema di I/O forniscono di solito un modo per indicare che una determinata richiesta, o l'I/O verso un particolare dispositivo, dovrà essere eseguita in maniera sincrona.

Un buon esempio di comportamento non bloccante è la chiamata di sistema `select()` per le socket di rete. Essa include un argomento che specifica il tempo d'attesa massimo. Se questo valore è 0, l'applicazione può rilevare attività di rete senza arrestarsi. Tuttavia, l'uso della `select()` introduce un overhead aggiuntivo, perché essa può stabilire soltanto se sia possibile compiere dell'I/O: per un effettivo trasferimento di dati, la `select()` deve essere seguita da istruzioni come `read()` o `write()`. Una variante di questo metodo, adottata per esempio dal sistema Mach, è l'impiego di una chiamata di sistema bloccante per la lettura multipla. Essa specifica con una singola chiamata di sistema le richieste di lettura desiderate per diversi dispositivi, e termina non appena una di loro sia stata soddisfatta.

13.3.5 I/O vettorizzato

Alcuni sistemi operativi forniscono un'altra importante variante di I/O tramite le loro interfacce delle applicazioni. L'I/O vettorizzato permette a una chiamata di sistema di eseguire più operazioni di I/O che coinvolgono più locazioni. Per esempio, la chiamata di sistema UNIX `readv` accetta un vettore di buffer e permette di inserire nel vettore i dati letti da una sorgente oppure di scrivere da quel vettore verso una destinazione. Lo stesso trasferimento potrebbe essere realizzato per mezzo di diverse singole invocazioni di chiamate di sistema, ma questo metodo, detto **scatter-gather**, risulta utile per una serie di motivi.

Il trasferimento del contenuto di più buffer distinti tramite un'unica chiamata di sistema permette di evitare cambi di contesto e l'overhead delle chiamate di sistema. In assenza di I/O vettorizzato i dati dovrebbero prima essere trasferiti in un unico buffer più grande, nel giusto ordine, e poi trasmessi. Quest'ultimo metodo risulta piuttosto inefficiente. Inoltre, alcune versioni di scatter-gather forniscono l'atomicità, assicurando così che tutti gli I/O vengano eseguiti senza interruzioni (evitando dunque

la corruzione di dati nel caso in cui altri thread stiano effettuando I/O verso gli stessi buffer). Quando possibile, i programmatori sfruttano le potenzialità dell’I/O scatter-gather per aumentare la produttività e diminuire l’overhead del sistema.

13.4 Sottosistema di I/O del kernel

Il kernel fornisce molti servizi riguardanti l’I/O; vari servizi – scheduling, gestione del buffer, delle cache, delle code di spooling, riservazione dei dispositivi e gestione degli errori – sono offerti dal sottosistema di I/O del kernel, e sono realizzati a partire dai dispositivi e dai relativi driver. Il sottosistema di I/O è responsabile anche della propria salvaguardia contro processi malfunzionanti e utenti malintenzionati.

13.4.1 Scheduling dell’I/O

Fare lo scheduling di un insieme di richieste di I/O significa stabilirne un ordine d’esecuzione efficace; l’ordine in cui si verificano le chiamate di sistema da parte delle applicazioni è raramente la scelta migliore. Lo scheduling può migliorare le prestazioni complessive del sistema, distribuire equamente gli accessi dei processi ai dispositivi e ridurre il tempo d’attesa medio per il completamento di un’operazione di I/O. Ecco un semplice esempio che illustra queste potenzialità. Si supponga che la testina di lettura di un’unità a disco sia vicina alla parte iniziale del disco, e che tre applicazioni impartiscano comandi di lettura bloccanti per quest’unità. L’applicazione 1 richiede la lettura di un blocco che si trova vicino alla parte finale del disco, l’applicazione 2 quella di un blocco vicino alla parte iniziale e l’applicazione 3 quella di un blocco situato nella zona centrale. Il sistema operativo può ridurre la distanza percorsa dalla testina del disco servendo le richieste nell’ordine 2, 3, 1. Simili riordinamenti delle sequenze di servizio delle richieste sono l’essenza dello scheduling dell’I/O.

I progettisti di sistemi operativi realizzano lo scheduling mantenendo una coda di richieste per ogni dispositivo. Quando un’applicazione richiede l’esecuzione di una chiamata di sistema di I/O bloccante, si aggiunge la richiesta alla coda relativa al dispositivo. Lo scheduler dell’I/O riorganizza l’ordine della coda per migliorare l’efficienza globale del sistema e il tempo medio d’attesa cui sono sottoposte le applicazioni. Il sistema operativo può anche tentare di essere equo, in modo che nessuna applicazione riceva un servizio carente, o può dare priorità alle richieste sensibili al ritardo. Per esempio, le richieste del sottosistema per la memoria virtuale potrebbero avere priorità su quelle delle applicazioni. Parecchi algoritmi di scheduling per l’I/O delle unità a disco sono descritti dettagliatamente nel Paragrafo 10.4.

I kernel che mettono a disposizione l’I/O asincrono devono essere in grado di tener traccia di più richieste di I/O contemporaneamente. A questo fine, alcuni sistemi associano una **tabella dello stato dei dispositivi** alla coda dei processi in attesa. Gli elementi della tabella – uno per ogni dispositivo di I/O – indicano il tipo, l’indirizzo e lo stato del dispositivo: non funzionante, inattivo o occupato. Se il dispositivo è im-

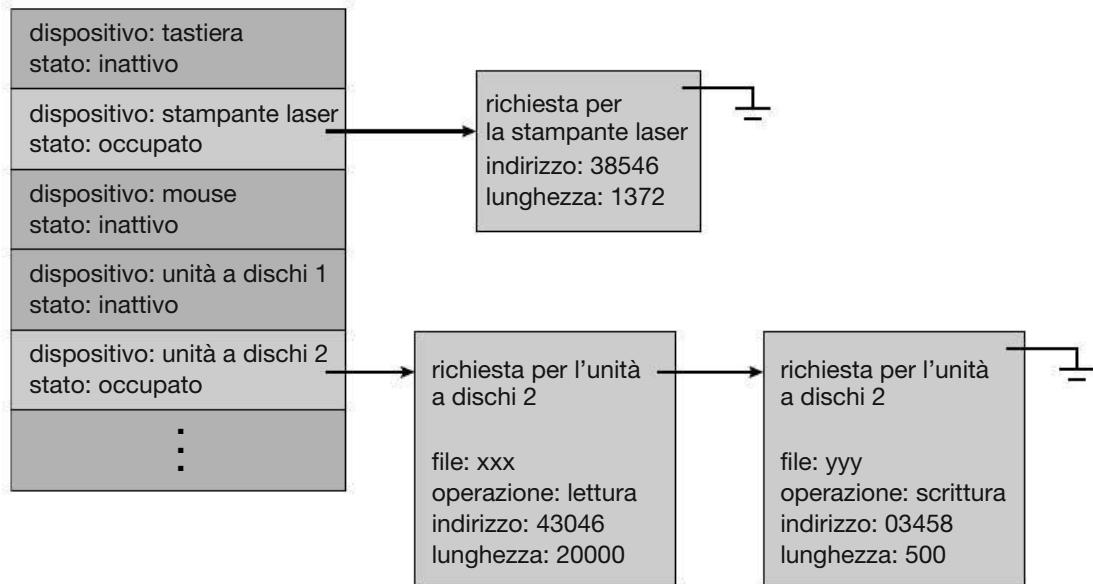


Figura 13.9 Tabella dello stato dei dispositivi.

pegnato nel servire una richiesta, il corrispondente elemento della tabella riporterà il tipo della richiesta e altri parametri a essa relativi. Si veda la Figura 13.9.

Lo scheduling dell’I/O è uno dei modi in cui il sottosistema di I/O migliora l’efficienza di un calcolatore; un altro è l’uso di spazio di memorizzazione nella memoria centrale o nei dischi, per tecniche di buffering, caching e spooling.

13.4.2 Gestione dei buffer

Un **buffer** è un’area di memoria che contiene dati durante il trasferimento fra due dispositivi o tra un’applicazione e un dispositivo. Si ricorre ai buffer per tre ragioni. La prima è la necessità di gestire la differenza di velocità fra il produttore e il consumatore di un flusso di dati. Si supponga, per esempio, di ricevere un file attraverso un modem e di volerlo memorizzare in un’unità a disco: il modem è circa mille volte più lento del disco, perciò conviene creare un buffer nella memoria principale per accumulare i byte che giungono dal modem. Quando tale buffer è pieno, si trasferisce il suo contenuto nel disco con un’unica operazione. Poiché quest’operazione di scrittura non è istantanea e il modem ha bisogno di ulteriore spazio per memorizzare i dati in arrivo, è necessario impiegare due buffer di questo tipo: quando il primo è pieno, si richiede la scrittura nel disco del suo contenuto e il modem comincia a scrivere nel secondo buffer mentre il primo viene scritto su disco. La scrittura nel disco dovrebbe terminare prima che il modem possa riempirlo, cosicché il modem potrà ricominciare a usare il primo buffer, mentre si trasferisce nel disco il contenuto del secondo. Questa **doppia bufferizzazione** svincola il produttore dal consumatore, rendendo così meno critico il problema della loro sincronizzazione. La necessità di questo disaccoppiamento è illustrata dalla Figura 13.10, che mostra le enormi differenze di velocità tra i dispositivi tipici di un calcolatore.

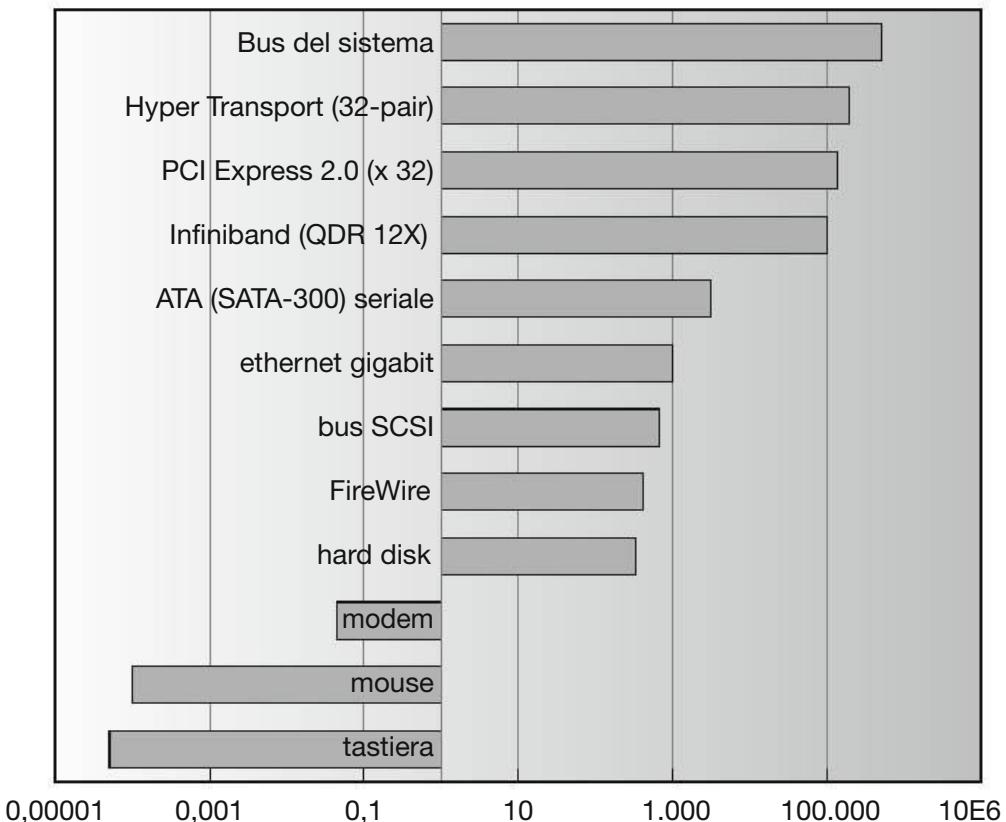


Figura 13.10 Velocità di trasferimento dei dispositivi di un Sun Enterprise 6000 (scala logaritmica).

Un secondo uso della bufferizzazione riguarda la gestione dei dispositivi che trasferiscono dati in blocchi di dimensioni diverse. Queste disparità sono particolarmente comuni nelle reti di calcolatori, dove spesso i buffer sono usati per frammentare e ricomporre messaggi. Quando un mittente invia un messaggio molto lungo, esso è spezzato in piccoli pacchetti che si spediscono attraverso la rete; il sistema destinatario provvede a ricomporre in un apposito buffer l'intero messaggio originario.

Il terzo modo in cui si può impiegare un buffer è per la realizzazione della *semantica della copia* nell'ambito dell'I/O delle applicazioni. Un esempio chiarirà il significato di semantica della copia. Si supponga che un'applicazione disponga di un buffer contenente dati da trasferire in un disco: esso richiederà l'esecuzione della chiamata di sistema `write()`, fornendo un puntatore al buffer e un numero intero che specifica il numero di byte da trasferire. Ci si può chiedere che cosa succede se, dopo che la chiamata di sistema restituisce il controllo all'applicazione, quest'ultima modifica il contenuto del buffer. Ebbene, la **semantica della copia** garantisce che la versione dei dati scritta nel disco sia conforme a quella contenuta nel buffer al momento della chiamata di sistema, indipendentemente da ogni successiva modifica. Una semplice maniera di realizzare questa semantica consiste nel far sì che la chiamata di sistema `write()` copi i dati forniti dall'applicazione in un buffer del kernel prima di restituire il controllo all'applicazione stessa. La scrittura nel disco si compie dalla memoria del

kernel, cosicché ogni successivo cambiamento nel buffer dell'applicazione non avrà effetti. In molti sistemi operativi si usa il metodo appena descritto: nonostante implichi una diminuzione dell'efficienza di certe operazioni di I/O, la sua semantica è chiara. Lo stesso effetto, tuttavia, si può ottenere più efficientemente tramite un uso intelligente della memoria virtuale e della protezione data dal copy-on-write delle pagine.

13.4.3 Cache

Una **cache** è una regione di memoria veloce che serve per mantenere copie di dati: l'accesso a queste copie è più rapido dell'accesso agli originali. Per esempio, le istruzioni di un processo correntemente in esecuzione sono memorizzate in un disco, copiate nella memoria fisica (che assume il ruolo di cache rispetto al disco) e copiate ulteriormente nelle cache primaria e secondaria della CPU. La differenza fra un buffer e una cache consiste nel fatto che il primo può contenere dati di cui non esiste altra copia, mentre una cache, per definizione, mantiene su un mezzo più efficiente una copia di informazioni memorizzate altrove.

L'uso delle cache e l'uso dei buffer sono due funzioni distinte, anche se a volte una stessa regione di memoria si può usare per entrambi gli scopi. Per esempio, per realizzare la semantica della copia e permettere uno scheduling efficiente dell'I/O su disco, il sistema operativo impiega dei buffer in memoria centrale per i dati dei dischi. Questi buffer sono anche usati come cache per migliorare l'efficienza delle operazioni di I/O che coinvolgono file condivisi da più applicazioni o file per i quali gli accessi per lettura e scrittura si susseguano rapidamente. Quando riceve una richiesta di I/O relativa a un file, il kernel controlla se la parte interessata del file è già presente nella cache: in questo caso è possibile evitare o differire l'accesso fisico al disco. Inoltre, i dati da scrivere nel disco sono depositati nella cache per diversi secondi, cosicché si accumulano grandi quantità di dati da trasferire: ciò permette uno scheduling efficiente. La strategia consistente nel differire le scritture per migliorare l'efficienza dell'I/O è illustrata nel Paragrafo 17.9.2, nel contesto dell'accesso ai file remoti.

13.4.4 Coda di spooling e riservazione dei dispositivi

Una **coda di spooling** è un buffer contenente dati da inviare ad un dispositivo che non può accettare flussi di dati intercalati, per esempio una stampante. Sebbene una stampante possa servire una sola richiesta alla volta, diverse applicazioni devono poter richiedere simultaneamente la stampa di dati, senza che le stampe si mischino. Il sistema operativo risolve questo problema filtrando tutti i dati per la stampante: i dati da stampare provenienti da ogni singola applicazione si registrano in uno specifico spool file su disco; quando un'applicazione termina di stampare, il sistema di spooling aggiunge tale file alla coda di stampa; quest'ultima viene copiata sulla stampante, un file per volta. In certi sistemi operativi questa funzione viene gestita da un processo di sistema specializzato (demone), in altri da un thread del kernel. In entrambi i casi il sistema operativo fornisce un'interfaccia di controllo che permette agli utenti e agli amministratori del sistema di esaminare la coda, eliminare elementi della coda prima che siano stampati, sospendere il servizio di stampa per attività di manutenzione, e così via.

Alcuni dispositivi, come le unità a nastro e le stampanti, non possono alternare più richieste concorrenti di I/O da parte di diverse applicazioni. Lo spooling è uno dei modi in cui il sistema operativo può coordinare output concorrenti; un altro è quello di fornire esplicite funzioni di coordinamento. Alcuni sistemi operativi, fra i quali il VMS, permettono di accedere a un dispositivo in modo esclusivo: un processo può accedere a un dispositivo che non utilizzato, riservandosene l'uso, e restituirlo al sistema quando non ne ha più bisogno. Altri sistemi operativi impediscono l'apertura di più di un handle di file per un dato dispositivo. Molti sistemi operativi forniscono funzioni che permettono ai processi stessi di coordinare l'uso esclusivo dei dispositivi: il sistema Windows, per esempio, mette a disposizione chiamate di sistema che permettono a un'applicazione di aspettare finché un certo dispositivo si liberi. Inoltre la sua chiamata di sistema `OpenFile()` accetta un parametro che specifica il tipo d'accesso concesso ad altri thread concorrenti. In questi sistemi le applicazioni hanno la responsabilità di evitare le situazioni di stallo.

13.4.5 Gestione degli errori

Un sistema operativo che usa la protezione della memoria può difendersi da molti tipi di errori dovuti all'hardware o alle applicazioni, cosicché il blocco completo del sistema non è la necessaria conseguenza di ogni piccolo malfunzionamento. I dispositivi e i trasferimenti di I/O possono essere soggetti ad errori in molti modi, sia per motivi contingenti, come il sovraccarico di una rete di comunicazione, sia per ragioni "permanenti", come nel caso in cui il controllore dell'unità a disco si guasti. I sistemi operativi sono spesso capaci di compensare efficacemente le conseguenze negative dovute a errori temporanei: se per esempio una chiamata di sistema `read()` non ha successo, il sistema ritenterà la lettura; se una chiamata `send()` provoca un errore, il protocollo di rete può richiedere una `resend()`. Purtroppo, però, è difficile che il sistema operativo riesca a compensare gli effetti di errori dovuti a guasti permanenti di qualche componente importante.

Di norma, una chiamata di sistema di I/O riporta un bit d'informazione sullo stato d'esecuzione della chiamata, che indica la riuscita o l'insuccesso dell'operazione richiesta. Il sistema operativo UNIX usa inoltre una variabile intera detta `errno` per codificare piuttosto genericamente il tipo d'errore avvenuto; i valori possibili sono un centinaio e denotano errori dovuti per esempio a puntatori non validi, file non aperti o argomenti oltre i limiti ammessi. Per contro, alcuni tipi di dispositivi possono fornire informazioni assai dettagliate sugli errori, sebbene molti sistemi operativi attuali non siano progettati per passare questi dati alle applicazioni. Per esempio, il malfunzionamento di un dispositivo SCSI è riportato dal protocollo SCSI a tre livelli di dettaglio: usando un **codice di rilevazione** (*sense key*) che identifica la natura generale dell'errore (per esempio: errore hardware, o richiesta illegale); un **codice di rilevazione addizionale** che descrive la categoria cui appartiene il malfunzionamento (parametro del comando errato, o insuccesso del self-test del dispositivo); e infine un **qualificatore del codice di rilevazione addizionale** che fornisce informazioni ancora più dettagliate (quale parametro è errato, o quale componente del dispositivo non ha

superato il test). Inoltre, molti dispositivi SCSI mantengono internamente pagine di log degli errori avvenuti; queste pagine possono essere richieste dalla macchina, ma ciò accade raramente.

13.4.6 Protezione dell'I/O

Gli errori sono strettamente connessi alla tematica della protezione. Un processo utente che, intenzionalmente o accidentalmente, cerchi di impartire istruzioni di I/O illegali può danneggiare il funzionamento normale di un sistema. Per impedire che tali danneggiamenti abbiano luogo, si possono opporre diverse contromisure.

Onde evitare che gli utenti effettuino operazioni di I/O illegali, si definiscono come privilegiate tutte le istruzioni relative all'I/O. Ne consegue che gli utenti non potranno impartire in modo diretto alcuna istruzione, ma dovranno farlo attraverso il sistema operativo. Un programma utente, per eseguire l'I/O, invoca una chiamata di sistema per chiedere al sistema operativo di svolgere una data operazione nel suo interesse (Figura 13.11). Il sistema, passando alla modalità privilegiata, verifica che la richiesta sia valida e, in tal caso, esegue l'operazione; esso trasferisce quindi il controllo all'utente.

Inoltre, il sistema di protezione della memoria deve tutelare dall'accesso degli utenti tutti gli indirizzi mappati in memoria e gli indirizzi delle porte di I/O. Il kernel, tuttavia, non può semplicemente negare qualunque tentativo di accesso da parte degli utenti: quasi tutti i videogiochi, nonché i programmi per il montaggio e la riproduzione di video, per esempio, per ottimizzare le prestazioni della grafica necessitano

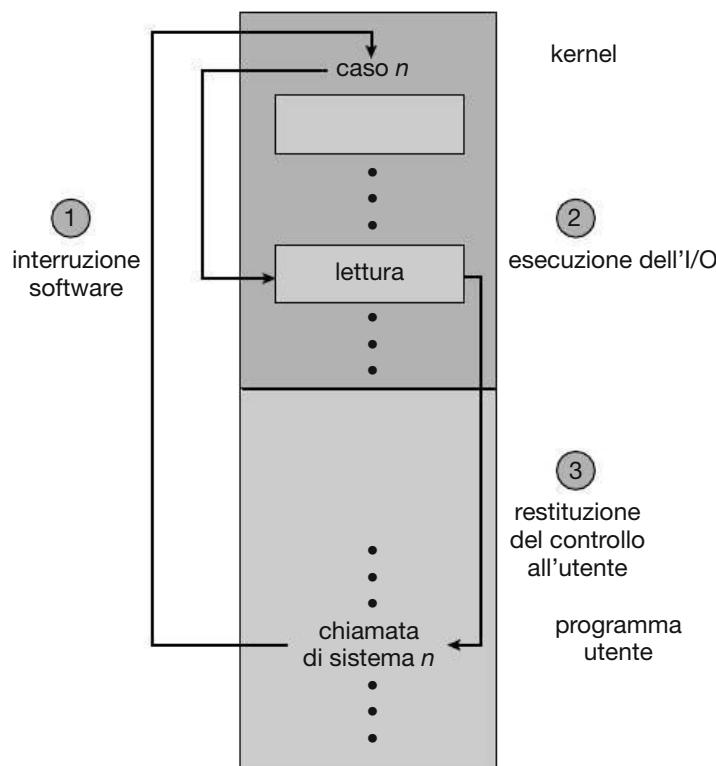


Figura 13.11 Uso delle chiamate di sistema per eseguire I/O.

dell'accesso diretto alla memoria del controllore della grafica, che è gestita in modalità memory mapped. In questi casi, il kernel potrebbe applicare dei lock per assegnare a un solo processo per volta la porzione della memoria grafica che rappresenta una finestra sullo schermo.

13.4.7 Strutture dati del kernel

Il kernel ha bisogno di mantenere informazioni sullo stato dei componenti di I/O, e usa a questo fine diverse strutture dati interne, un esempio delle quali è la tabella dei file aperti descritta nel Paragrafo 12.1. Il kernel usa molte strutture di questo tipo per tener traccia dei collegamenti di rete, delle comunicazioni con i dispositivi a caratteri e di altre attività di I/O.

Il sistema operativo UNIX permette l'accesso, con le modalità tipiche del file system, a diversi oggetti: file degli utenti, dispositivi, spazio d'indirizzi dei processi, e altri ancora. Sebbene ognuno di questi oggetti supporti una chiamata `read()`, le semantiche sono diverse secondo i casi. Quando il kernel, per esempio, deve leggere un file utente, ha bisogno di controllare la *buffer cache* prima di decidere l'effettiva esecuzione di un'operazione di I/O su un disco. Per leggere un disco privo di struttura logica (*raw disk*), il kernel deve accertarsi del fatto che la dimensione dell'insieme dei dati di cui è stato richiesto il trasferimento sia un multiplo della dimensione dei settori del disco e sia allineato con il settore interessato. Per leggere l'immagine di un processo, tutto ciò che occorre è copiare dati dalla memoria. UNIX incapsula queste differenze in una struttura uniforme usando una tecnica orientata agli oggetti. Il record di un file aperto, mostrato nella Figura 13.12, contiene una tabella di puntatori alle procedure appropriate secondo il tipo di file in questione.

Alcuni sistemi operativi applicano metodi orientati agli oggetti in misura più rilevante: il sistema Windows, per esempio, usa per l'I/O un sistema basato sullo scambio di messaggi. Una richiesta di I/O si converte in un messaggio che s'invia tramite il kernel al sottosistema per la gestione dell'I/O, quindi al driver del dispositivo; i contenuti del messaggio possono essere modificati a ogni passaggio intermedio. Quando l'operazione richiesta è di output, il messaggio contiene i dati da scrivere; quando invece l'operazione richiesta è di input, il messaggio contiene un buffer che si usa per ricevere i dati. Questo metodo può comportare una minore efficienza rispetto alle tecniche procedurali basate sulla condivisione delle strutture dati, ma semplifica la progettazione e la struttura del sistema di I/O e permette una maggiore flessibilità.

13.4.8 Concetti principali del sottosistema di I/O del kernel

Riassumendo, il sistema per l'I/O coordina un'ampia raccolta di servizi disponibili per le applicazioni e per altre parti del kernel; in generale sovrintende alle seguenti funzioni:

- gestione dello spazio dei nomi per file e dispositivi;
- controllo dell'accesso ai file e ai dispositivi;
- controllo delle operazioni (per esempio, un modem non può effettuare un `seek()`);

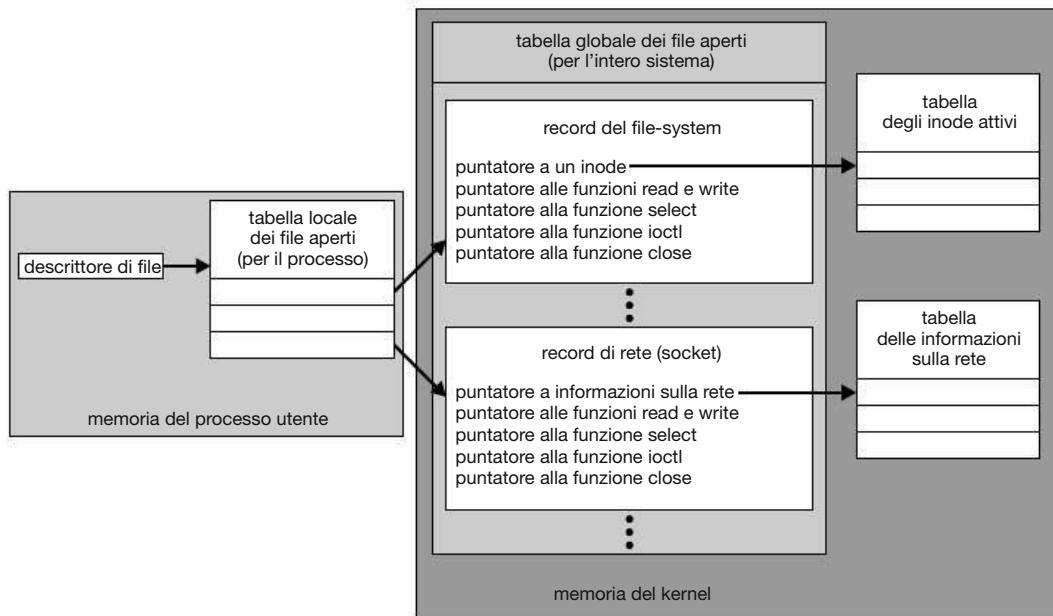


Figura 13.12 Struttura dell'I/O nel kernel di UNIX.

- allocazione dello spazio per il file system;
- allocazione dei dispositivi;
- gestione dei buffer, delle cache e delle code di spooling;
- scheduling dell'I/O;
- controllo dello stato dei dispositivi, gestione degli errori e procedure di ripristino;
- configurazione e inizializzazione dei driver dei dispositivi.

I livelli superiori del sottosistema per la gestione dell'I/O accedono ai dispositivi per mezzo dell'interfaccia uniforme fornita dai driver.

13.5 Trasformazione delle richieste di I/O in operazioni hardware

Il meccanismo di handshaking tra un driver e un controllore di dispositivo è già stato illustrato; tuttavia, non si è ancora spiegato come il sistema operativo associa alla richiesta di un'applicazione un insieme di fili di rete o uno specifico settore di disco. Si consideri per esempio la lettura di un file da un'unità a disco. L'applicazione fa riferimento ai dati per mezzo del nome del file: è compito del file system fornire il modo di giungere, attraverso la struttura delle directory, alla regione del disco appropriata, cioè quella dove i dati del file sono fisicamente residenti. Nell'MS-DOS, per esempio, il nome del file è associato a un numero che individua un elemento della tabella d'accesso ai file; tale elemento identifica i blocchi del disco assegnati al file. In UNIX il nome è associato a un numero di *inode*; l'*inode* corrispondente contiene le

informazioni necessarie per individuare lo spazio allocato. Ma come viene effettuato il collegamento fra il nome del file e il controller del disco (indirizzo hardware della porta o registri memory mapped del controller)?

Un metodo è quello utilizzato da un sistema relativamente semplice come l'MS-DOS. La prima parte di un nome di file dell'MS-DOS, precisamente la parte che precede i due punti, identifica uno specifico dispositivo. Per esempio, **C:** è la parte iniziale di ogni nome di file residente nell'unità a disco principale. Questa convenzione è codificata all'interno del sistema operativo: **C:** è associato a uno specifico indirizzo di porta per mezzo di una tabella dei dispositivi. Grazie all'uso dei due punti come separatore, lo spazio dei nomi dei dispositivi è distinto dallo spazio dei nomi del file system, ciò semplifica al sistema operativo l'associazione di funzioni aggiuntive ai dispositivi. Per esempio, è facile attivare lo spooling per i file di cui è stata richiesta la stampa.

Se, invece, i nomi dei dispositivi sono inclusi nell'ordinario spazio dei nomi del file system, come in UNIX, sono automaticamente disponibili i servizi legati ai nomi dei file. Per esempio, se il file system associa dei possessori ai nomi dei file e fornisce il controllo degli accessi a ogni nome di file, si potrà controllare anche l'accesso ai dispositivi, ed essi avranno un possessore. Visto che i file risiedono nei dispositivi, una tale interfaccia fornisce due livelli d'accesso al sistema d'I/O: i nomi si possono usare per accedere ai dispositivi stessi o ai file in essi contenuti.

UNIX rappresenta i nomi dei dispositivi all'interno dell'ordinario spazio dei nomi del file system. A differenza di un nome di file dell'MS-DOS, che include i due punti come separatore, in un nome di percorso di UNIX il nome del dispositivo non è esplicitamente separato. In effetti, nessuna parte del nome di percorso di un file è il nome di un dispositivo; UNIX impiega una **tabella di montaggio** (*mount table*), per associare i prefissi dei nomi di percorso ai corrispondenti nomi di dispositivi. Quando deve risolvere un nome di percorso, il sistema esamina la tabella per trovare il più lungo prefisso corrispondente: questo elemento della tabella indica il nome del dispositivo voluto. Anche questo nome è rappresentato come un oggetto del file system: tuttavia, quando UNIX cerca questo nome nelle strutture delle directory del file system, non trova il numero di un *inode*, ma una coppia di numeri **<principale, secondario>** (**<major, minor>**) che identifica un dispositivo. Il numero principale individua il driver che si deve usare per gestire l'I/O nel dispositivo in questione, mentre il numero secondario deve essere passato a questo driver affinché esso possa determinare, per mezzo di un'altra tabella, l'indirizzo della porta o l'indirizzo memory mapped del controllore del dispositivo interessato. I moderni sistemi operativi riescono a ottenere un notevole grado di flessibilità grazie all'uso di tabelle di lookup a vari livelli durante il processo che porta da una richiesta al controllore del dispositivo; questo processo, inoltre, è del tutto generale, cosicché non è necessario ricompilare il kernel ogni volta che si aggiungono al calcolatore nuovi dispositivi e nuovi driver. In effetti, alcuni sistemi operativi hanno la capacità di caricare driver di dispositivi su richiesta: all'avviamento, il sistema sonda i bus per determinare quali dispositivi siano presenti; quindi carica i necessari driver, operazione che può anche essere rinviata fino alla prima richiesta di I/O.

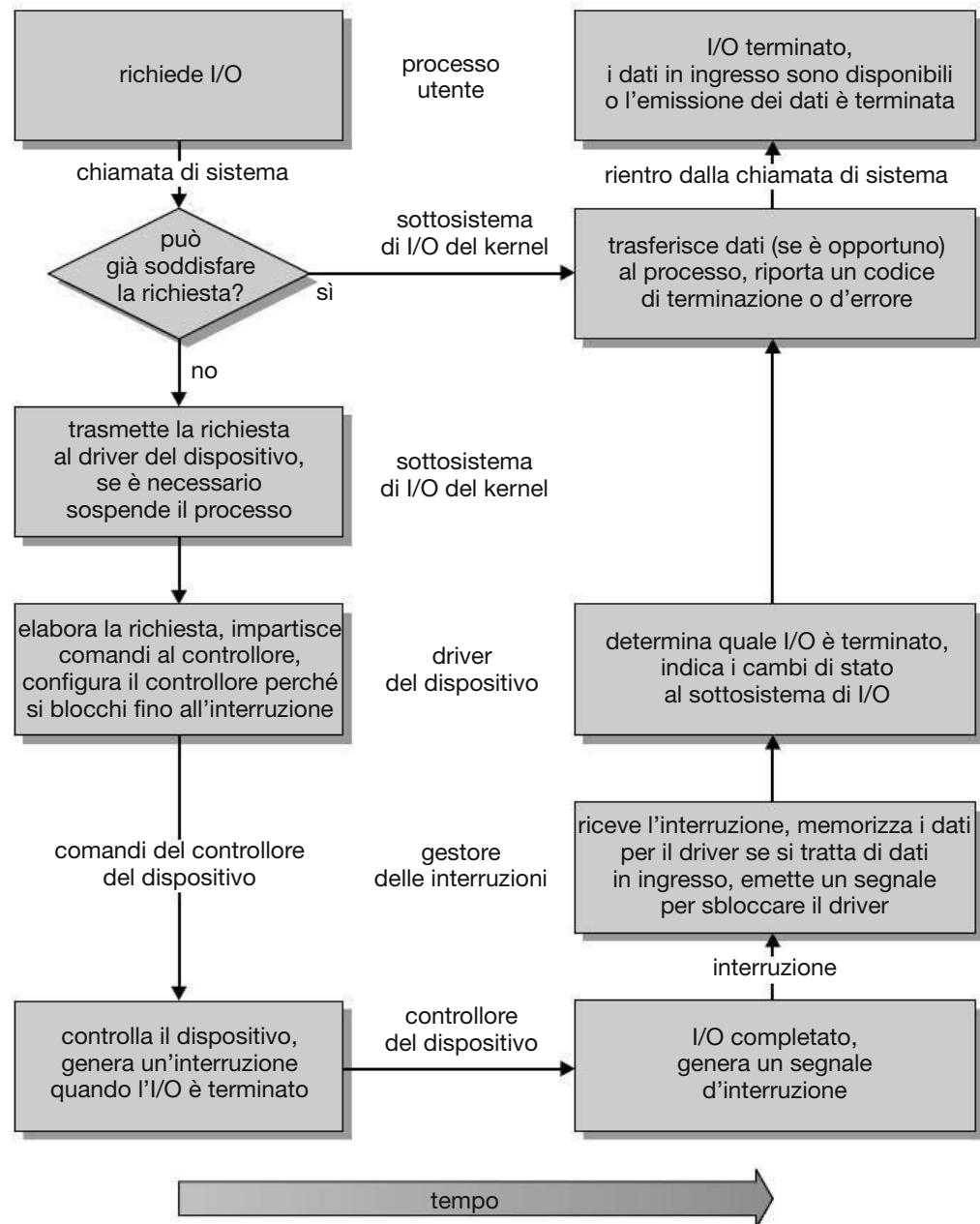


Figura 13.13 Schema d'esecuzione di una richiesta di I/O.

La seguente descrizione del tipico svolgimento di una richiesta di lettura bloccante (Figura 13.13) indica che l'esecuzione di un'operazione di I/O richiede una gran quantità di passi; ciò implica l'uso di un enorme numero di cicli di CPU.

1. Un processo esegue una chiamata di sistema `read()` bloccante relativa a un descrittore di file di un file precedentemente aperto.
2. Il codice della chiamata di sistema all'interno del kernel controlla la correttezza dei parametri. Nel caso di input, se i dati sono già presenti nella *buffer cache*, si passano al processo chiamante e l'operazione è conclusa.

3. Altrimenti, è necessario eseguire un'operazione di I/O fisico. Si rimuove il processo dalla coda dei processi pronti per l'esecuzione per inserirlo nella coda d'attesa relativa al dispositivo interessato e si effettua lo scheduling della richiesta di I/O. Infine il sottosistema di I/O invia la richiesta al driver del dispositivo; a seconda del sistema operativo, ciò avviene tramite la chiamata di una procedura, o per mezzo dell'invio di un messaggio interno al kernel.
4. Il driver del dispositivo assegna un buffer nello spazio d'indirizzi del kernel che serve per ricevere i dati immessi, ed esegue lo scheduling dell'I/O. Infine il driver impedisce comandi al controllore del dispositivo scrivendo nei suoi registri.
5. Il controllore aziona il dispositivo hardware per compiere il trasferimento dei dati.
6. Il driver può operare in polling, o può aver predisposto un trasferimento DMA nella memoria del kernel. Supponiamo che il trasferimento sia gestito dal controllore DMA, il quale genera un'interruzione al termine dell'operazione.
7. Tramite il vettore delle interruzioni, si attiva l'appropriato gestore dell'interruzione che, dopo aver memorizzato i dati necessari, avverte con un segnale il driver del dispositivo ed effettua il ritorno dall'interruzione.
8. Il driver riceve il segnale, individua la richiesta di I/O che è stata completata, si accerta della riuscita o del fallimento dell'operazione e segnala al sottosistema di I/O del kernel il completamento dell'operazione.
9. Il kernel trasferisce dati e/o codici di stato nello spazio degli indirizzi del processo chiamante, e sposta tale processo dalla coda d'attesa alla coda dei processi pronti per l'esecuzione.
10. Nel momento in cui è posto nella coda dei processi pronti per l'esecuzione, il processo non è più bloccato: quando lo scheduler gli assegnerà la CPU, esso riprenderà l'elaborazione. L'esecuzione della chiamata di sistema è completata.

13.6 STREAMS

Il sistema operativo UNIX System V offre un interessante meccanismo, chiamato **STREAMS**, che permette a un'applicazione di comporre dinamicamente catene di codice di driver. Uno *stream* è una connessione *full-duplex* fra un driver di dispositivo e un processo utente, e consiste di un elemento di interfaccia per il processo utente (*stream head*), un elemento che controlla il dispositivo (*driver end*), ed eventualmente un certo numero di moduli intermedi fra questi due elementi (*stream modules*). Tutti questi componenti contengono una coppia di code, una di lettura e una di scrittura; per il trasferimento dei dati tra le due code s'impiega uno schema a scambio di messaggi. La Figura 13.14 mostra la struttura del meccanismo di STREAMS.

Le funzioni d'elaborazione di STREAMS sono fornite da moduli che si inseriscono nello stream attraverso la chiamata di sistema `ioctl()`. Un processo può, per esempio, aprire tramite uno stream una porta seriale, e può inserire un modulo per editare

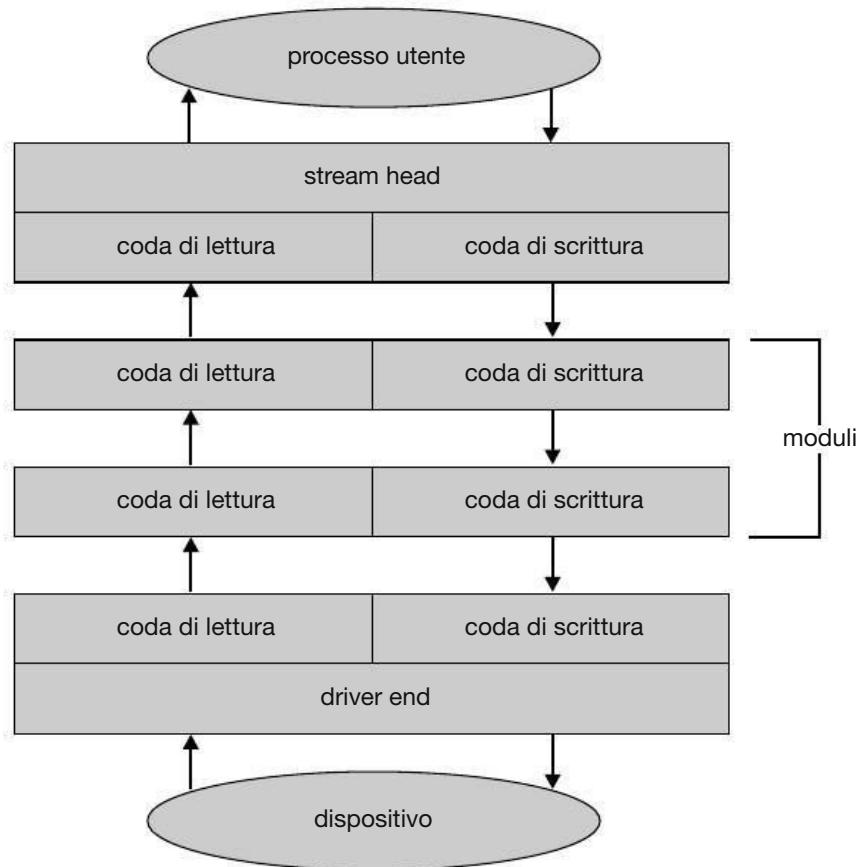


Figura 13.14 Struttura di STREAMS.

i dati immessi. Poiché i messaggi sono scambiati tra code di moduli adiacenti, la coda di un modulo potrebbe mandare in overflow la coda di un modulo adiacente. Per prevenire questo problema, una coda può disporre di un meccanismo di **controllo di flusso**. Senza controllo di flusso, una coda accetta tutti i messaggi e li trasferisce immediatamente alla coda del modulo adiacente, senza buffering. Una coda che invece impiega il controllo di flusso memorizza i messaggi in un buffer; se lo spazio disponibile in esso non è sufficiente, non accetta messaggi. Questo meccanismo richiede scambi di messaggi di controllo tra code in moduli adiacenti.

Un processo utente usa le chiamate di sistema `write()` o `putmsg()` per scrivere dati in un dispositivo; la chiamata di sistema `write()` scrive semplicemente dati non strutturati nello *stream*, mentre `putmsg()` permette al processo utente di specificare un messaggio. Indipendentemente dalla chiamata di sistema adoperata dal processo utente, lo stream head copia i dati in un messaggio e li recapita alla coda del modulo successivo. Questa copiatura dei messaggi continua finché il messaggio non giunge al driver end e quindi al dispositivo. Analogamente, il processo utente legge i dati dallo stream head usando la chiamata di sistema `read()` oppure `getmsg()`. Se si usa la `read()` lo stream head preleva un messaggio dalla coda del modulo adiacente e riporta al processo dati ordinari (una sequenza non strutturata di byte). Se si usa la `getmsg()` viene inviato un messaggio al processo utente.

L'I/O per mezzo di STREAMS è asincrono (o non bloccante), con l'eccezione di quando il processo utente comunica con lo stream head. Mentre scrive nello *stream*, il processo utente si blocca, se la coda successiva impiega il controllo di flusso, finché non ci sia spazio sufficiente per copiarvi il messaggio. Analogamente, il processo utente si blocca durante la lettura dallo *stream* finché non ci sono dati disponibili.

Come si è detto, il driver end, come lo stream head e i moduli intermedi, ha una coda di lettura e una di scrittura. Tuttavia deve poter rispondere a interruzioni come quelle generate quando un pacchetto di dati è pronto per essere letto da una rete. A differenza dello stream head che si può bloccare se non è possibile copiare un messaggio nella coda del modulo successivo, il driver end deve gestire tutti i dati in arrivo. I driver devono anche supportare il controllo di flusso. Tuttavia, se il buffer di un dispositivo è pieno, di solito ignora i messaggi in entrata. Si consideri una scheda di rete il cui buffer d'ingresso sia pieno; la scheda di rete deve semplicemente ignorare gli ulteriori messaggi fintantoché non vi sia sufficiente spazio per memorizzare i messaggi in arrivo.

Il vantaggio nell'utilizzo di STREAMS consiste nel disporre di un ambiente che permette uno sviluppo modulare e incrementale di driver di dispositivi e protocolli di rete. I moduli possono essere usati da diversi stream e quindi da diversi dispositivi. Per esempio, un modulo di rete potrebbe essere adoperato sia da una scheda di rete Ethernet sia da una scheda di rete wireless 802.11. Inoltre, invece di trattare l'I/O di dispositivi a caratteri come una sequenza non strutturata di byte, STREAMS permette la gestione dei limiti dei messaggi e delle informazioni di controllo tra i diversi moduli. L'impiego di STREAMS è assai diffuso nella maggior parte dei sistemi UNIX, ed è il metodo più usato per la scrittura di protocolli e driver di dispositivi. In UNIX System V e in Solaris, per esempio, il meccanismo delle socket è realizzato tramite STREAMS.

13.7 Prestazioni

L'I/O è uno tra i principali fattori che influiscono sulle prestazioni di un sistema: richiede un notevole impegno della CPU per l'esecuzione del codice dei driver e per uno scheduling equo ed efficiente dei processi quando essi sono bloccati e sbloccati. I risultanti cambi di contesto sfruttano fino in fondo la CPU e le sue memorie cache. L'I/O, inoltre, rivela le eventuali inefficienze dei meccanismi del kernel per la gestione delle interruzioni, e impiega il bus della memoria durante i trasferimenti di dati tra i controllori dei dispositivi e la memoria fisica, e ancora tra i buffer del kernel e lo spazio d'indirizzi delle applicazioni. Soddisfare in modo elegante tutte queste esigenze è una delle principali questioni che riguardano i progettisti di un calcolatore.

Sebbene i calcolatori moderni siano capaci di gestire molte migliaia di interruzioni al secondo, la loro gestione è un processo relativamente oneroso. Ogni interruzione fa sì che il sistema cambi stato, esegua il gestore delle interruzioni, e infine ripristini lo stato originario. Se il numero di cicli impiegato nel busy waiting non è eccessivo,

l’I/O programmato può essere più efficiente di quello basato sulle interruzioni. Il completamento di un’operazione di I/O in genere implica lo sblocco di un processo, comportando così l’overhead dovuto a un cambio di contesto.

Anche il traffico di una rete può portare a un alto numero di cambi di contesto; si consideri, per esempio, il login remoto da un calcolatore ad un’altro. Ciascun carattere inserito in un calcolatore deve essere comunicato all’altro: quando si inserisce un carattere nel primo calcolatore, la tastiera produce un segnale d’interruzione; il carattere arriva tramite il gestore delle interruzioni al driver del dispositivo, da questo al kernel, e quindi al processo utente. Il processo utente esegue una chiamata di sistema di I/O di rete al fine di inviare il carattere al secondo calcolatore. All’interno del kernel del primo calcolatore, il carattere attraversa gli strati di protocollo necessari per la costruzione di un pacchetto di rete e giunge al driver di rete, il quale trasferisce il pacchetto al controllore della interfaccia di rete. Quest’ultimo invia il carattere e genera un’interruzione che segnala al kernel il completamento della chiamata di sistema di I/O di rete.

A questo punto, il dispositivo di rete del secondo calcolatore riceve il pacchetto, e genera un’interruzione. I protocolli di rete estraggono il carattere dal pacchetto e lo consegnano all’appropriato demone di rete. Il demone di rete individua a quale sessione di login remoto appartiene il carattere e lo passa al sottodemone che gestisce la specifica sessione. Durante questo processo avvengono cambi di contesto e di stato (Figura 13.15). Di solito il destinatario rispedisce al mittente, sotto forma di eco, una copia del carattere originario, e ciò raddoppia il lavoro necessario.

Per eliminare i cambi di contesto implicati dal trasferimento di ciascun carattere dal demone al kernel, gli sviluppatori di Solaris hanno implementato nuovamente il demone `telnet` tramite thread interni al kernel. Secondo stime della Sun, queste migliorie hanno portato il massimo numero di connessioni contemporanee sostenibili da un grande server da qualche centinaio a qualche migliaio.

Altri sistemi usano unità d’elaborazione specifiche per la gestione dell’I/O dei terminali, riducendo così il carico delle gestione delle interruzioni gravante sulla CPU. Per esempio, i **concentratori di terminali** convogliano il traffico proveniente da centinaia di terminali su un’unica porta di un grande calcolatore. I **canali di I/O** sono unità d’elaborazione specializzate presenti nei mainframe e altri sistemi di alto profilo; il loro compito è sollevare la CPU di una parte del peso della gestione dell’I/O. L’idea di base è che i canali di I/O mantengano il flusso dei dati costante e uniforme, mentre la CPU rimane libera di elaborare i dati acquisiti. Come i controller dei dispositivi e i controllori DMA che si trovano nei calcolatori di minori dimensioni, un canale può eseguire programmi più raffinati e generali, e può quindi essere tarato per specifici carichi di lavoro.

Per migliorare l’efficienza dell’I/O si possono applicare diversi principi.

- Ridurre il numero dei cambi di contesto.
- Ridurre il numero di copiature dei dati in memoria durante i trasferimenti fra dispositivi e applicazioni.

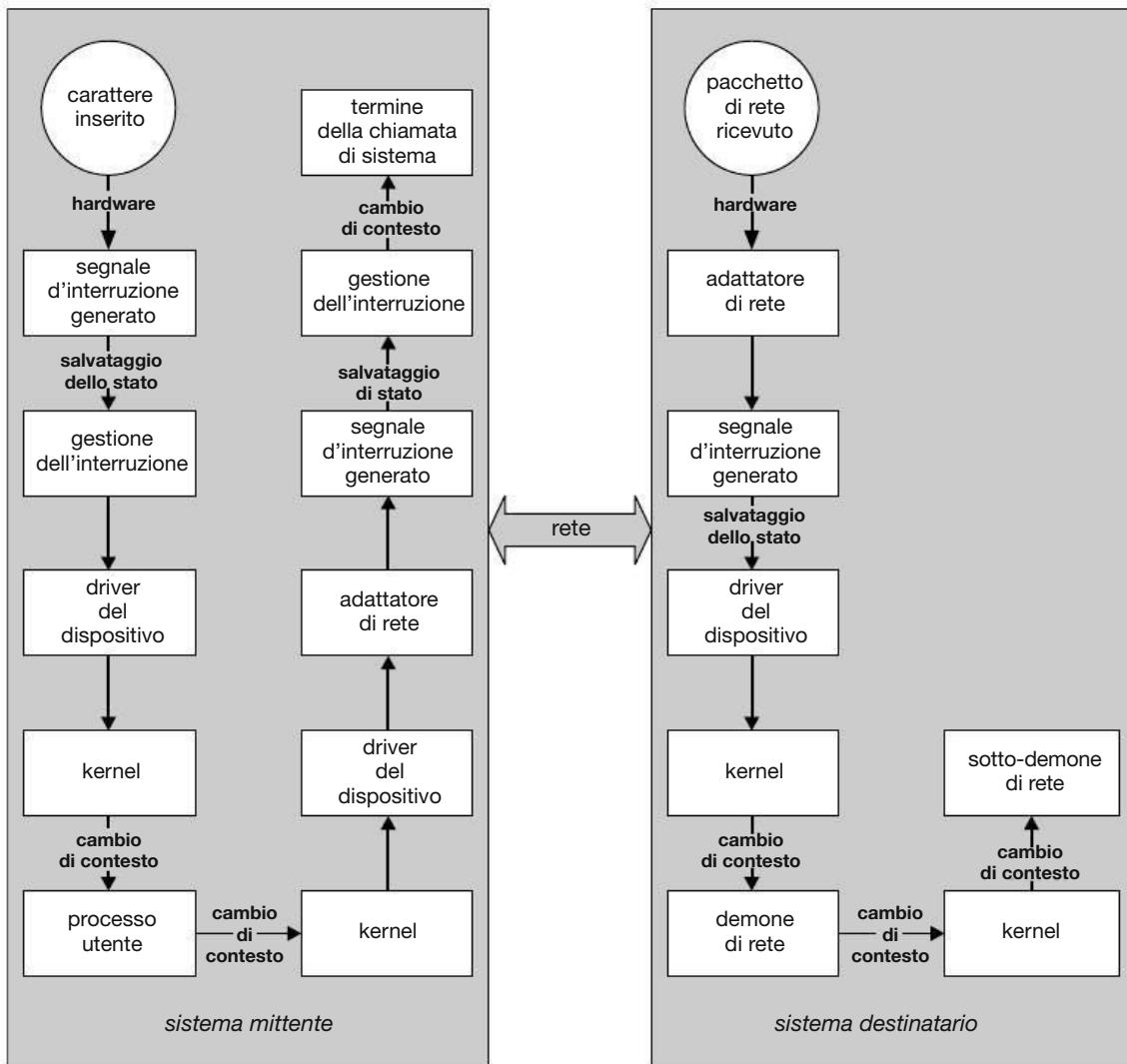


Figura 13.15 Comunicazione tra calcolatori.

- Ridurre la frequenza delle interruzioni utilizzando il trasferimento in blocco di grandi quantità di dati, controllori intelligenti e il polling (nel caso in cui si possa minimizzare il busy waiting).
- Aumentare il tasso di concorrenza usando controllori DMA intelligenti o canali di I/O per sollevare la CPU dalle semplici operazioni di copiatura di dati.
- Realizzare le primitive di elaborazione direttamente nell'hardware, così da permettere che la loro esecuzione sia simultanea alle operazioni di bus e di CPU.
- Equilibrare le prestazioni della CPU, del sottosistema di memoria, del bus e dell'I/O, giacché il sovraccarico di uno qualunque di questi settori provoca l'inutilizzo degli altri.

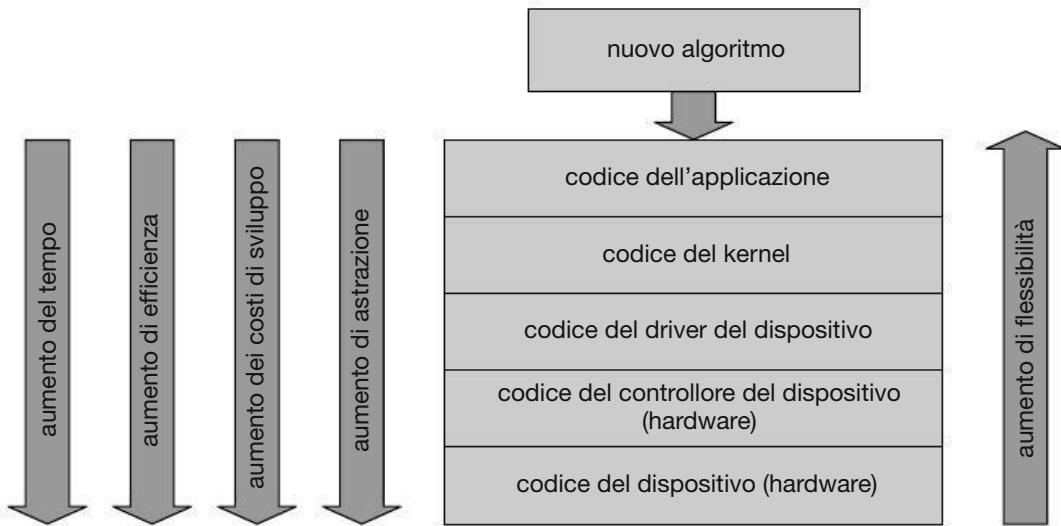


Figura 13.16 Successione delle funzionalità dei servizi di I/O.

La complessità dei dispositivi è assai variabile: un mouse per esempio è piuttosto semplice; i suoi movimenti e la pressione sui pulsanti sono convertiti in valori numerici, passati attraverso il driver del mouse, all'applicazione. Per contro, i servizi forniti dal driver delle unità a disco del sistema operativo Windows sono assai complessi: non solo il driver gestisce singole unità a disco, ma costruisce anche array RAID (si veda il Paragrafo 10.7) convertendo le richieste di lettura o scrittura di un'applicazione in un insieme coordinato di operazioni di I/O sui dischi. Il driver, inoltre, esegue sofisticati algoritmi di gestione degli errori e di recupero dei dati, e svolge diverse funzioni di ottimizzazione delle prestazioni dei dischi.

Ci si può chiedere se i servizi di I/O si debbano implementare nei dispositivi hardware, nei loro driver, o nelle applicazioni. Talvolta si può osservare (Figura 13.16) la seguente successione.

- Inizialmente, gli algoritmi sperimentali per l'I/O si codificano a livello dell'applicazione, dato che il codice dell'applicazione è flessibile ed è difficile che errori di programmazione causino l'arresto completo del sistema. In questo modo si evita inoltre di dover riavviare o ricaricare i driver dei dispositivi ogni volta che si modifica il codice degli algoritmi. Tuttavia, questi algoritmi sono spesso inefficienti a causa dell'overhead dovuto ai cambi di contesto necessari e dell'impossibilità di sfruttare le strutture dati del kernel e i suoi meccanismi interni (per esempio, la gestione dei messaggi, l'uso dei thread, i lock).
- Quando uno di questi algoritmi è stato messo a punto, è possibile ricodificarlo all'interno del kernel. Ciò può portare a un miglioramento delle prestazioni, ma la stesura del codice è più impegnativa, perché il kernel è un ambiente vasto e complesso. È inoltre necessario verificare accuratamente la correttezza del codice al fine di evitare alterazioni dei dati e l'arresto del sistema.

- Le prestazioni migliori si ottengono con l'integrazione delle funzioni di tali algoritmi in hardware, nei dispositivi o nei controllori. Gli svantaggi di questa tecnica comprendono la difficoltà e il costo di successive migliorie o dell'eliminazione di eventuali errori, il maggior tempo richiesto per portarne a termine la realizzazione (mesi invece di giorni), e la minore flessibilità. Per esempio, un controllore RAID può essere privo di funzioni che permettono al kernel di modificare la locazione o l'ordine di lettura o scrittura di singoli blocchi, anche se il kernel potrebbe possedere informazioni particolari sul carico di lavoro che gli permetterebbero di migliorare le prestazioni dell'I/O.

13.8 Sommario

I principali elementi hardware di un calcolatore coinvolti nell'esecuzione dell'I/O sono i bus, i controllori dei dispositivi e i dispositivi stessi. I trasferimenti dei dati tra i dispositivi e la memoria centrale sono controllati dalla CPU nel caso di I/O programmato, altrimenti, sono demandati al controllore DMA. Un modulo del kernel che controlla un dispositivo è detto driver. L'interfaccia fornita alle applicazioni, costituita dalle chiamate di sistema, è progettata per gestire diverse categorie fondamentali di dispositivi hardware, come i dispositivi a blocchi e a caratteri, i timer programmabili, i file mappati in memoria, le socket di rete. Di solito le chiamate di sistema bloccano il processo che le ha invocate; il kernel e le applicazioni che non devono attendere il completamento di un'operazione di I/O impiegano chiamate di sistema non bloccanti o asincrone.

Il sottosistema di I/O del kernel fornisce numerosi servizi: fra gli altri, lo scheduling dell'I/O, il buffering, il caching, le code di spooling, la gestione degli errori e la riservazione dei dispositivi. Un altro servizio è la traduzione dei nomi, che permette di associare ai nomi simbolici di file usati dalle applicazioni i dispositivi hardware corrispondenti. Si tratta di un processo che passa attraverso diversi stadi: dalla sequenza di caratteri che rappresenta il nome a un driver specifico e all'indirizzo di un dispositivo, e da qui all'indirizzo fisico delle porte di I/O o dei controllori di bus. Tale interpretazione può avvenire nell'ambito dello spazio dei nomi del file system (come in UNIX), o in uno specifico spazio di nomi dei dispositivi (come nell'MS-DOS).

STREAMS è una realizzazione e una metodologia che permettono di sviluppare in modo modulare ed incrementale i driver e i protocolli di rete. Utilizzando gli stream, i driver possono essere organizzati in una catena, attraverso cui passano i dati in maniera sequenziale e bidirezionale per l'elaborazione.

A causa dei molti strati di software presenti fra un dispositivo fisico e l'applicazione, le chiamate di sistema per l'I/O sono onerose in termini di utilizzazione della CPU. Questa struttura a strati comporta diversi overhead: per realizzare i cambi di contesto, gestire le interruzioni e i segnali usati per la comunicazione con i dispositivi, e copiare dati fra le aree di memoria per l'I/O del kernel e lo spazio d'indirizzi delle applicazioni.

Esercizi di ripasso

- 13.1** Individuate tre vantaggi che si ottengono dal collocare funzionalità all'interno del controllore di un dispositivo piuttosto che nel kernel. Individuate poi tre svantaggi.
- 13.2** L'esempio di handshaking del Paragrafo 13.2 utilizza 2 bit: un bit busy e un bit command-ready. È possibile implementare la stessa negoziazione con un solo bit? Se è possibile, descrivete il protocollo. Se non lo è, spiegate perché un bit non è sufficiente.
- 13.3** Perché un sistema potrebbe utilizzare I/O guidato dalle interruzioni per gestire una porta seriale singola e il polling per gestire un processore di front-end come un terminal concentrator?
- 13.4** Il polling per il completamento di I/O può sprecare un gran numero di cicli di CPU se il processore itera un ciclo busy-waiting molte volte prima che l'I/O sia terminato. D'altro canto, se il dispositivo di I/O è pronto per il servizio, l'interrogazione ciclica può essere molto più efficiente rispetto a rilevare e gestire un'interruzione. Descrivete una strategia ibrida per un dispositivo di I/O che combini polling, sleeping e interruzioni. Per ognuna di queste tre strategie (polling puro, interruzioni pure e ibrida) descrivete un contesto in cui quella strategia sia più efficiente delle altre.
- 13.5** In che modo il DMA aumenta la concorrenza? In che senso complica il progetto dell'hardware?
- 13.6** Perché è importante aumentare la velocità del bus di sistema e delle periferiche all'aumentare della velocità della CPU?
- 13.7** Fate una distinzione tra un driver STREAMS e un modulo STREAMS.

Esercizi

- 13.8** Nel caso di interruzioni multiple, inviate da dispositivi diversi approssimativamente nello stesso istante, si può applicare un criterio di priorità per stabilire l'ordine di precedenza da dare alle interruzioni. Valutate gli elementi che è necessario considerare per assegnare priorità alle interruzioni.
- 13.9** Valutate gli aspetti positivi e negativi causati della mappatura in memoria dei registri di controllo dei dispositivi per l'I/O.
- 13.10** Considerate le seguenti situazioni riguardanti l'I/O in un PC monoutente:
- un mouse usato insieme con un'interfaccia utente grafica;
 - un lettore di nastri in un sistema operativo multitasking (assumete l'impossibilità di preassegnare i dispositivi);
 - un'unità a disco contenente i file dell'utente;

- d. una scheda grafica con connessione diretta tramite bus, accessibile per mezzo di I/O memory mapped.

Per ognuna di queste situazioni, dite se è opportuno progettare il sistema operativo in modo che possa impiegare la gestione del buffer, lo spooling, il caching, o una loro combinazione. Dite inoltre se è opportuno usare il polling o le interruzioni. Argomentate le risposte.

- 13.11** In molti sistemi multiprogrammati, i programmi utenti accedono alla memoria per mezzo degli indirizzi virtuali, mentre il sistema operativo utilizza direttamente gli indirizzi fisici. Come si riflette questo fatto sulle operazioni di I/O, nella fase di avvio da parte del programma utente e, in seguito, sulla loro esecuzione da parte del sistema?
- 13.12** Quali sono le diverse forme di overhead di gestione causate dal servire un'interruzione?
- 13.13** Descrivete tre circostanze in cui sarebbe opportuno usare l'I/O bloccante, e altre tre in cui dovrebbe essere usato l'I/O non bloccante. Riflettete sulla possibilità di realizzare semplicemente l'I/O non bloccante e lasciare che i processi interroghino ciclicamente i dispositivi richiesti finché essi sono pronti.
- 13.14** Di norma, quando un dispositivo completa il proprio I/O, si genera una singola interruzione, gestita dal processore nel modo appropriato. In alcuni frangenti, tuttavia, il codice da eseguire al termine del lavoro di I/O può essere suddiviso in due segmenti: il primo, eseguito subito dopo che l'I/O è completato, pianifica anche una seconda interruzione per innescare l'esecuzione del secondo segmento di codice, in un momento successivo. A quale scopo si adotta questa strategia nel progettare i gestori delle interruzioni?
- 13.15** Alcuni controllori DMA forniscono l'accesso diretto alla memoria virtuale. In questo caso, i destinatari delle operazioni di I/O sono indirizzi virtuali, tradotti poi in indirizzi fisici durante l'accesso diretto alla memoria. Per quali versi tale funzionalità complica la progettazione del controllore DMA? Quali sono i suoi vantaggi?
- 13.16** Il sistema UNIX coordina le attività dei componenti per l'I/O del kernel manipolando le strutture dati condivise interne al kernel; invece il sistema Windows utilizza lo scambio di messaggi tra i componenti del kernel, con tecniche orientate agli oggetti. Valutate tre elementi a favore e tre a sfavore di ciascuna strategia.
- 13.17** Scrivete in uno pseudocodice una procedura che realizzi un orologio virtuale che comprenda l'accodamento e la gestione delle richieste del timer per il kernel e le applicazioni. Assumete che l'hardware fornisca tre canali di temporizzazione.
- 13.18** Considerate vantaggi e svantaggi che derivano dal garantire un trasferimento dei dati affidabile tra i moduli di un'applicazione STREAMS.

Note bibliografiche

[Vahalia 1996] dà una buona visione d’insieme dell’I/O e della comunicazione di rete nel sistema operativo UNIX. [McKusick e Neville-Neil] fornisce dettagli sulle strutture e sui metodi per l’I/O in FreeBSD. L’uso e la programmazione dei vari protocolli per la comunicazione di rete e fra processi sono trattati in [Stevens 1992].

[Hart 2005] si occupa di programmazione Windows.

[Intel 2011] costituisce una buona fonte sui processori Intel. [Rago 1993] offre una buona trattazione del meccanismo STREAMS. [Hennessy e Patterson 2012] descrive i sistemi multiprocessore e i problemi di coerenza della cache.

Bibliografia

- [Hart 2005] J. M. Hart, *Windows System Programming*, 3° ed., Addison-Wesley, 2005.
- [Hennessy e Patterson 2012] J. Hennessy e D. Patterson, *Computer Architecture: A Quantitative Approach*, 5° ed., Morgan Kaufmann, 2012.
- [Intel 2011] *Intel 64 and IA-32 Architectures Software Developer’s Manual*, Vol. 1, 2A, 2B, 3A, 3B. Intel Corporation, 2011.
- [McKusick e Neville-Neil 2005] M. K. McKusick e G. V. Neville-Neil, *The Design and Implementation of the FreeBSD UNIX Operating System*, Addison-Wesley, 2005.
- [Rago 1993] S. Rago, *UNIX System V Network Programming*, Addison-Wesley, 1993.
- [Stevens 1992] R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.
- [Vahalia 1996] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall, 1996.

Protezione e sicurezza

I meccanismi di protezione offrono un accesso controllato limitando i tipi d'accesso ai file permessi agli utenti. La protezione deve anche assicurare che solo i processi che hanno ottenuto l'autorizzazione dal sistema operativo possano usare i segmenti di memoria, la CPU e altre risorse.

La protezione è assicurata da un meccanismo che controlla l'accesso di programmi, processi o utenti alle risorse di un sistema elaborativo. Questo meccanismo deve offrire uno strumento per specificare quali siano i controlli da impostare, e un sistema che permetta di farli rispettare.

La sicurezza garantisce l'autenticazione degli utenti del sistema per proteggere l'integrità delle informazioni (dati e codice) in esso memorizzate e delle risorse fisiche di cui il sistema elaborativo dispone. Il sistema di sicurezza impedisce gli accessi non autorizzati al sistema, i tentativi dolosi di distruzione o alterazione dei dati, e l'introduzione accidentale d'incoerenze.

CAPITOLO

14

OBIETTIVI DEL CAPITOLO

- Obiettivi e principi relativi alla protezione in un moderno sistema elaborativo.
- Analisi dell'utilizzo di domini di protezione combinati con matrici d'accesso per la specifica di risorse accessibili da un processo.
- Valutazione di sistemi di protezione basati su abilitazione e sul linguaggio.

Protezione

I processi di un sistema operativo devono essere protetti dalle attività altrui. Molteplici meccanismi provvedono al raggiungimento di tale scopo, in modo che file, segmenti di memoria, CPU e altre risorse possano essere adoperati solo dai processi che hanno ottenuto l'autorizzazione dal sistema operativo.

La protezione riguarda il meccanismo per il controllo dell'accesso alle risorse definite da un sistema elaborativo da parte di programmi, processi o utenti. Questo meccanismo deve fornire un metodo per specificare i controlli da imporre e alcuni mezzi per garantirli. È necessario fare una distinzione tra protezione e sicurezza; quest'ultima è una misura della fiducia sul mantenimento dell'integrità di un sistema e dei suoi dati. In questo capitolo tratteremo la protezione. La garanzia della sicurezza è un argomento più generico rispetto alla protezione, ed è trattato nel Capitolo 15.

14.1 Scopi della protezione

I calcolatori sono divenuti più complessi e le loro applicazioni sono cresciute a dismisura; di conseguenza è aumentata anche la necessità di proteggere la loro integrità. La protezione era originariamente concepita come un elemento aggiuntivo dei sistemi operativi con multiprogrammazione tale che utenti non fidati potessero condividere tranquillamente uno spazio di nomi logici comuni, come una directory di file, oppure uno spazio di nomi fisici comuni, come la memoria. I moderni concetti di protezione si sono evoluti in modo da aumentare l'affidabilità di qualsiasi sistema complesso che usi risorse condivise.

È necessario disporre di sistemi di protezione per diversi motivi. Tra i più ovvi c'è la necessità di prevenire la violazione intenzionale e dannosa di un vincolo d'accesso da parte di un utente. Tuttavia, ha un'importanza più generale la necessità di assicurare che ogni componente del programma attivo in un sistema impieghi le risorse del sistema in modo coerente con i criteri (*policies*) stabiliti; per un sistema affidabile, questo requisito è assolutamente necessario.

La protezione può migliorare l'affidabilità rilevando errori latenti nelle interfacce tra sottosistemi componenti. Un'individuazione precoce di errori d'interfaccia riesce spesso a impedire la contaminazione di un sottosistema intatto da parte di un sottosistema malfunzionante. Una risorsa non protetta non può difendersi contro l'uso, o l'abuso, da parte di un utente non autorizzato o incompetente. Un sistema orientato alla protezione offre i mezzi per distinguere tra uso autorizzato e non autorizzato.

Il ruolo della protezione è quello di offrire un meccanismo d'imposizione di criteri che controllino l'uso delle risorse. Questi criteri si possono stabilire in vari modi: alcuni sono stati fissati nella fase di progettazione del sistema, altri sono determinati dalla gestione di un sistema; altri ancora sono definiti dai singoli utenti per proteggere i loro file e programmi. Un sistema di protezione deve avere una flessibilità tale da consentire d'imporre vari tipi di criteri.

I criteri d'uso di una risorsa possono variare secondo l'applicazione e possono cambiare nel tempo. Per queste ragioni la protezione non è più una questione riguardante solamente i progettisti di un sistema operativo; anche i programmatore di applicazioni hanno bisogno di meccanismi per proteggere dagli abusi le risorse create e gestite dai sottosistemi applicativi. In questo capitolo si descrivono i meccanismi di protezione che il sistema operativo dovrebbe fornire, e che anche i progettisti di applicazioni possono impiegarli nella progettazione del proprio sistema di protezione.

I criteri vanno distinti dai *meccanismi*. I meccanismi determinano *come* qualcosa si debba eseguire; i criteri decidono *che cosa* si debba fare. La distinzione tra criteri e meccanismi è importante ai fini della flessibilità. I criteri sono soggetti a cambiamenti in dipendenza dall'ambiente e dal tempo. Nel peggior dei casi qualsiasi cambiamento di criterio richiederebbe un cambiamento anche nel meccanismo sottostante. Utilizzando meccanismi generali si possono evitare tali situazioni.

14.2 Princìpi di protezione

Spesso un principio guida può essere utilizzato attraverso un intero progetto, quale per esempio la progettazione di un sistema operativo. L'osservanza di questo principio semplifica le decisioni relative al progetto, garantendo che il sistema sia coerente e facile da capire. Un principio guida, che nel tempo ha confermato la sua importanza per la protezione, è il **principio del privilegio minimo**. Esso sancisce che programmi, utenti, e finanche i sistemi, devono ricevere solo le risorse minime necessarie all'esecuzione dei rispettivi compiti.

Si consideri l'analogia di una guardia di vigilanza in possesso di un passe-partout. Se, con questa chiave, la guardia può accedere solamente alle aree pubbliche che controlla, l'uso improprio della chiave provocherà un danno minimo. Se però la chiave è per tutte le aree, qualora venga smarrita, rubata, usata impropriamente, duplicata o in alcun modo alterata, la rilevanza del danno a cui essa espone sarà molto maggiore.

Un sistema operativo fondato sul principio del privilegio minimo implementa le proprie funzionalità, programmi, chiamate di sistema e strutture dati in modo da contenere al minimo il danno derivante dal guasto o dal funzionamento indebito di un componente. L'overflow del buffer di un demone di sistema, per esempio, potrebbe causare un malfunzionamento del demone, ma non dovrebbe permettere l'esecuzione di codice dallo stack del relativo processo, permettendo che un utente remoto guadagni i massimi privilegi e possa accedere all'intero sistema (come troppo spesso accade al giorno d'oggi).

Un tale sistema operativo, inoltre, deve realizzare le chiamate di sistema e i servizi in modo da consentire alle applicazioni il controllo degli accessi con elevata granularità. Il sistema deve fornire meccanismi per attivare i privilegi quando sono necessari e per disattivarli quando non lo sono. Altrettanto utile è la creazione di *tracciature di verifica (audit trails)* relative all'accesso a tutte le funzioni privilegiate. Un audit trail permette al programmatore, all'amministratore del sistema, o al rappresentante dell'autorità giudiziaria, di ricostruire tutte le attività di protezione e di sicurezza compiute nel sistema.

Applicare il principio del privilegio minimo agli utenti comporta la creazione di un account per ciascun utente, ognuno dei quali prevede i minimi privilegi necessari. Un operatore che debba montare nastri e creare copie di backup sul sistema avrà accesso solo a quei comandi e a quei file che gli sono necessari per ultimare il lavoro. Alcuni sistemi adottano il controllo dell'accesso basato sui ruoli (RBAC) per fornire questa funzionalità.

Quando un centro di calcolo rispetta il principio del privilegio minimo, gli elaboratori che ne fanno parte possono offrire ciascuno l'esecuzione di specifici servizi, o l'accesso a macchine remote tramite servizi prefissati e in orari predeterminati. Queste restrizioni sono normalmente messe in atto grazie all'attivazione o disattivazione di ciascun servizio e mediante liste di controllo degli accessi, come spiegato nei Paragrafi 11.6.2 e 14.6.

Il principio del privilegio minimo può contribuire a creare un ambiente elaborativo più sicuro; spesso, però, non è così. Il sistema Windows 2000, per esempio, malgrado

abbia alla base una complessa struttura di protezione, rivela numerose falle nella sicurezza. In confronto, Solaris è considerato relativamente sicuro, anche se deriva da UNIX che, agli inizi, non ha fatto della protezione una sua priorità. La spiegazione di questa differenza potrebbe risiedere nel fatto che Windows 2000 ha una maggior quantità di servizi e di linee di codice rispetto a Solaris, e quindi più risorse da proteggere. Può anche darsi, però, che lo schema di protezione di Windows 2000 sia incompleto, o protegga aspetti irrilevanti del sistema, lasciandone altri vulnerabili.

14.3 Domini di protezione

Un sistema elaborativo è un insieme di processi e oggetti. Con il nome di *oggetti* si designano sia gli **oggetti fisici**, come CPU, segmenti di memoria, stampanti, dischi e unità a nastri, sia gli **oggetti logici**, come file, programmi e semafori. Ogni oggetto ha un nome unico che lo distingue da tutti gli altri oggetti del sistema; è inoltre possibile accedere a ciascuno di loro solo tramite operazioni ben definite e significative. Gli oggetti sono fondamentalmente tipi di dati astratti.

Le operazioni possibili dipendono dall'oggetto: per esempio, una CPU può compiere solo elaborazioni; i segmenti di memoria si possono leggere e scrivere, mentre da un lettore di CD-ROM o DVD-ROM si può soltanto leggere; le unità a nastri si possono leggere, scrivere e riavvolgere; i file di dati si possono creare, aprire, leggere, scrivere, chiudere e cancellare; i file di programmi si possono leggere, scrivere, eseguire e cancellare.

A un processo si deve permettere l'accesso alle sole risorse su cui ha l'autorizzazione. Inoltre, a un dato istante, un processo deve poter accedere solamente alle risorse di cui ha bisogno per eseguire il proprio compito. Questo secondo requisito, noto con il nome di **principio della necessità di sapere** (*need-to-know-principle*), è utile per limitare i danni che possono essere causati al sistema da un processo difettoso. Se, per esempio, il processo *P* invoca la procedura *A()*, alla procedura si deve permettere l'accesso solo alle proprie variabili e ai parametri che le vengono passati; non deve poter accedere a tutte le variabili del processo *P*. Analogamente, si consideri il caso in cui il processo *P* richieda la compilazione di un particolare file. Al compilatore non si deve permettere l'accesso a qualsiasi file, ma solo al ben definito sottinsieme di file associato al file da compilare. Viceversa, il compilatore può avere dei file privati, che impiega per scopi di accounting o di ottimizzazione, ai quali il processo *P* non deve aver accesso. Il *principio della necessità di sapere* somiglia al *principio del privilegio minimo* illustrato nel Paragrafo 14.2, in quanto gli scopi della protezione consistono nel minimizzare i rischi di possibili violazioni alla sicurezza.

14.3.1 Struttura dei domini di protezione

Per facilitare lo schema appena descritto, un processo opera all'interno di un **dominio di protezione**, che specifica le risorse accessibili dal processo. Ogni dominio definisce un insieme di oggetti e i tipi di operazioni che si possono compiere su ciascun

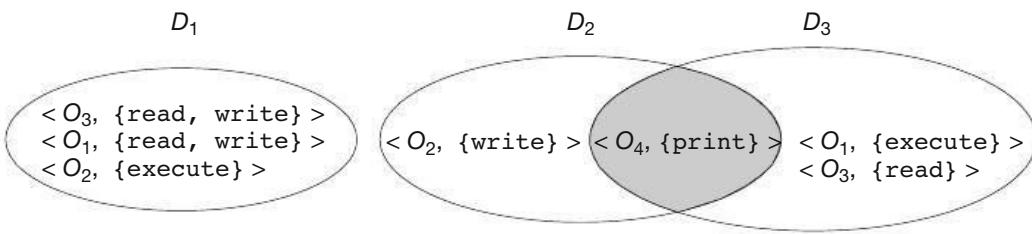


Figura 14.1 Sistema con tre domini di protezione.

oggetto. La possibilità di eseguire un’operazione su un oggetto è detta **diritto d’accesso**. Un **dominio** è dunque un insieme di diritti d’accesso, ciascuno dei quali è composto di una coppia ordinata $\langle \text{nome oggetto, insieme di diritti} \rangle$. Per esempio, se il dominio D ha il diritto d’accesso $\langle \text{file } F, \{\text{read, write}\} \rangle$, un processo in esecuzione nel dominio D può leggere e scrivere il file F , ma non può eseguire altre operazioni su quello stesso oggetto.

I domini possono condividere diritti d’accesso. Nella Figura 14.1, per esempio, sono illustrati tre domini: D_1 , D_2 e D_3 . Il diritto d’accesso $\langle O_4, \{\text{print}\} \rangle$ è condiviso da D_2 e D_3 , implicando che un processo in esecuzione su uno dei due domini può stampare l’oggetto O_4 . Occorre notare che un processo, per leggere e scrivere l’oggetto O_1 , deve essere in esecuzione nel dominio D_1 . D’altra parte, soltanto i processi che si trovano nel dominio D_3 possono eseguire l’oggetto O_1 .

L’associazione tra un processo e un dominio può essere **statica**, se l’insieme delle risorse disponibili per un processo è fisso per tutta la durata del processo, o **dinamica**. Com’è prevedibile, la determinazione dei domini di protezione dinamici è più complicata della determinazione dei domini di protezione statici.

Se l’associazione tra processi e domini è fissa, per aderire al principio del privilegio minimo è necessario disporre di un meccanismo che permetta di modificare il contenuto di un dominio. Ciò deriva dal fatto che l’esecuzione di un processo si può dividere in più fasi e, ad esempio, il processo può richiedere l’accesso in lettura in una fase e l’accesso in scrittura in un’altra. Se un dominio è statico, occorre definire un dominio che contenga sia l’accesso per la lettura sia quello per la scrittura. Tuttavia questa disposizione fornisce più diritti di quanti siano necessari in ciascuna delle due fasi, poiché è disponibile il diritto d’accesso per la lettura nella fase in cui è necessario il solo accesso per la scrittura e viceversa; quindi il principio del privilegio minimo è violato. È necessario permettere che il contenuto di un dominio sia modificato, in modo che il dominio rifletta sempre i minimi diritti d’accesso necessari.

Se l’associazione è dinamica, deve essere disponibile un meccanismo per permettere a un processo di passare da un dominio all’altro (*domain switching*). Si può anche voler modificare al contenuto di un dominio. Se non è possibile modificare il contenuto di un dominio, si può ottenere lo stesso effetto creando un nuovo dominio con il contenuto modificato e passando al nuovo dominio quando sia necessario modificare il contenuto del dominio originario.

Si noti che un dominio si può realizzare in diversi modi.

- Ogni *utente* può essere un dominio. In questo caso l’insieme d’oggetti cui si può accedere dipende dall’identità dell’utente. Il cambio di dominio avviene quando cambia l’utente, generalmente quando un utente chiude una sessione di lavoro e un altro la comincia.
- Ogni *processo* può essere un dominio. In questo caso l’insieme d’oggetti cui si può accedere dipende dall’identità del processo. Il cambio di dominio avviene quando un processo invia un messaggio a un altro processo e quindi attende una risposta.
- Ogni *procedura* può essere un dominio. In questo caso l’insieme d’oggetti cui si può accedere corrisponde alle variabili locali definite all’interno della procedura. Il cambio di dominio avviene quando s’invoca una procedura.

Il cambio di dominio è approfondito nel Paragrafo 14.4.

Si consideri l’ordinaria duplice modalità di funzionamento (di sistema e utente) dei sistemi operativi. Quando si esegue un processo in modalità di sistema, esso può impiegare istruzioni privilegiate e quindi acquisire il controllo completo del calcolatore. D’altra parte, se è eseguito in modalità utente, il processo può impiegare solo istruzioni non privilegiate; di conseguenza, può essere eseguito solo all’interno del proprio spazio di memoria predefinito. Questi due modi proteggono il sistema operativo, che è eseguito nel dominio di sistema, dai processi utenti, che si eseguono nel dominio dell’utente. In un sistema operativo con multiprogrammazione due domini di protezione sono insufficienti, poiché gli utenti richiedono anche la protezione reciproca. Serve quindi uno schema più elaborato: lo illustreremo esaminando due influenti sistemi operativi, UNIX e MULTICS, e osservando come realizzano questi concetti.

14.3.2 Un esempio: UNIX

Nel sistema operativo UNIX un dominio è associato all’utente. Il cambio di dominio corrisponde al cambio temporaneo dell’identificatore dell’utente. Questo cambio si compie tramite il file system. A ogni file sono associati un identificatore di proprietario e un bit di dominio (noto con il nome di **setuid bit**). Quando il *setuid bit* è **on**, e un utente esegue quel file, lo *userID* è impostato a quello del proprietario del file; quando il *setuid bit* è **off** lo *userID* non cambia. Ad esempio, quando un utente, con *userID* uguale ad *A*, avvia l’esecuzione di un file posseduto da *B*, il cui *setuid bit* è **off**, l’identificatore del processo viene impostato ad *A*. Quando il *setuid bit* è **on**, lo *userID* viene impostato a quello del proprietario del file, in questo caso *B*. Quando il processo termina, cessa anche quest’impostazione temporanea dello *userID*.

Per cambiare domini nei sistemi operativi in cui i domini sono definiti dagli identificatori degli utenti s’impiegano anche altri metodi. Quasi tutti i sistemi hanno bisogno di un tale meccanismo, che si usa per rendere disponibile a tutti gli utenti una funzione il cui utilizzo è normalmente privilegiato. Per esempio, è auspicabile che tutti gli utenti possano ottenere il permesso di accedere a una rete senza che ciascuno debba

scrivere il proprio programma di interfacciamento con la rete. In tal caso in un sistema UNIX si attiva il bit *setuid* di un programma di rete con la conseguenza che l'identificatore dell'utente cambia quando il programma è in esecuzione: l'identificatore di un utente con il privilegio d'accesso alla rete (come `root`, l'utente più importante) sostituisce il corrente identificatore dell'utente. Un problema che si pone con questo metodo è che se un utente riesce a creare un file con identificatore `root` e con il *setuid* bit posto a *on*, può assumere l'identità `root` e fare qualsiasi operazione sul sistema.

Un metodo alternativo, usato in altri sistemi operativi, prevede l'inserimento dei programmi privilegiati in una directory speciale. In questo caso il sistema operativo è progettato in modo da cambiare al momento dell'esecuzione l'identificatore dell'utente di ogni programma residente in questa directory, rendendolo equivalente a `root` o all'identificatore dell'utente proprietario della directory. Ciò elimina il problema di sicurezza che si verifica quando intrusi creano programmi per manipolare la funzione *setuid* e nascondere i programmi presenti nel sistema per usi futuri. Questo metodo è comunque meno flessibile di quello usato nello UNIX.

Ancora più restrittivi, e quindi più protettivi, sono i sistemi che semplicemente non permettono di modificare l'identificatore dell'utente. In questi casi è necessario ricorrere a speciali tecniche per permettere agli utenti di accedere a funzioni privilegiate. Per esempio, si può attivare un **processo demone** nella fase d'avviamento del sistema ed eseguirlo con uno speciale `user ID`. Gli utenti quindi eseguono un programma distinto che invia richieste a questo processo ogni volta che hanno bisogno di usare la relativa funzione. Questo metodo è impiegato dal sistema operativo TOPS-20.

In ciascuno di questi sistemi la scrittura dei programmi privilegiati si deve realizzare con molta cura: qualsiasi svista può causare una totale mancanza di protezione del sistema. Di solito questi programmi sono i primi a essere attaccati dalle persone che tentano di penetrare in un sistema; sfortunatamente tali attacchi hanno spesso successo; per esempio, la sicurezza è stata infranta in molti sistemi UNIX attraverso la funzionalità del *setuid*. La sicurezza è trattata nel Capitolo 15.

14.3.3 Un esempio: MULTICS

Nel sistema MULTICS i domini di protezione sono organizzati gerarchicamente in una struttura ad anelli, numerati da 0 a 7. Ciascun anello corrisponde a un dominio (Figura 14.2). Si supponga che D_i e D_j siano due anelli di dominio. Se $j < i$, D_i è un sottoinsieme di D_j ; ciò significa che un processo in esecuzione nel dominio D_j ha più privilegi di quanti ne abbia uno in esecuzione nel dominio D_i . Un processo in esecuzione nel dominio D_0 ha più privilegi di tutti. Se ci sono due soli anelli, questo schema corrisponde alla modalità d'esecuzione di sistema e utente, dove la modalità di sistema corrisponde a D_0 e la modalità utente a D_1 .

MULTICS ha uno spazio d'indirizzi segmentato; ogni segmento è un file ed è associato a uno degli anelli. Un descrittore del segmento include un elemento che identifica il numero dell'anello. Inoltre contiene tre bit d'accesso per il controllo di lettura, scrittura ed esecuzione. L'associazione tra segmenti e anelli è una scelta che riguarda le policy e che non esamineremo.

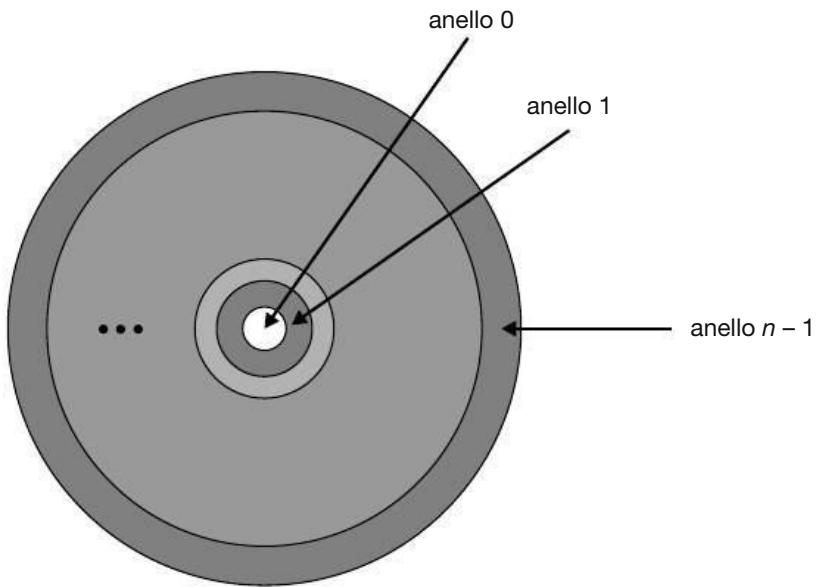


Figura 14.2 Struttura ad anelli del sistema MULTICS.

A ogni processo è associato un contatore del *numero corrente d'anello*, che identifica l'anello in cui il processo è attualmente in esecuzione. Quando un processo è in esecuzione nell'anello i , non può accedere a un segmento associato all'anello j , tale che $j < i$, ma può accedere a un segmento associato all'anello k , tale che $k \geq i$. Comunque, il tipo d'accesso è limitato dai bit d'accesso associati a quel segmento.

In MULTICS il cambio dei domini avviene quando un processo passa da un anello all'altro invocando una procedura in un anello diverso. Naturalmente questo cambio si deve compiere in modo controllato, altrimenti un processo potrebbe iniziare l'esecuzione nell'anello 0, senza che sia garantita alcuna protezione. Per permettere un cambio di dominio controllato, occorre modificare il campo concernente l'anello del descrittore del segmento inserendovi le seguenti informazioni.

- **Un intervallo d'accesso.** Una coppia d'interi b_1 e b_2 tali che $b_1 \leq b_2$.
- **Un limite.** Un intero b_3 tale che $b_3 > b_2$.
- **Una lista degli ingressi (o gates).** Identifica i punti d'accesso (ingressi) da cui si possono invocare i segmenti.

Se un processo in esecuzione nell'anello i invoca una procedura (segmento) con intervallo d'accesso (b_1, b_2) , la chiamata è ammessa se $b_1 \leq i \leq b_2$ e il numero corrente d'anello del processo rimane i . Altrimenti s'invia un segnale di eccezione al sistema operativo e la situazione viene gestita come segue.

- Se $i < b_1$, si può eseguire la chiamata, poiché si ha un trasferimento a un anello (dominio) con meno privilegi. Tuttavia, se si passano parametri che si riferiscono a segmenti in un anello inferiore (vale a dire segmenti che non sono accessibili alla procedura invocata), questi segmenti devono essere copiati in un'area alla quale può accedere anche la procedura invocata.

- Se $i > b_2$, si può eseguire la chiamata solo se b_3 è maggiore o uguale a i e la chiamata è indirizzata a uno dei punti d'accesso stabiliti nella lista. Questo schema permette a processi con diritti d'accesso limitati di invocare, ma solo in modo attentamente controllato, procedure che si trovano in anelli inferiori e che hanno più diritti d'accesso.

Lo svantaggio maggiore legato alla struttura (gerarchica) ad anelli consiste nel fatto che non consente l'applicazione del principio della necessità di sapere. In particolare, se un oggetto deve essere accessibile nel dominio D_j , ma non nel dominio D_i , allora deve essere $j < i$, ma ciò significa che ogni segmento accessibile in D_i è accessibile anche in D_j .

Il sistema di protezione di MULTICS è in generale più complesso e meno efficiente di quello adoperato nei correnti sistemi operativi. Se la protezione interferisce con la comodità d'uso del sistema, o ne riduce significativamente le prestazioni, il suo impiego deve essere attentamente valutato rispetto agli scopi del sistema. Per esempio, si potrebbe volere un complesso sistema di protezione in un calcolatore usato in un'università per gestire i voti degli studenti, e usato anche dagli studenti per i loro compiti. Un simile sistema di protezione non sarebbe adatto a un calcolatore che s'impiega per elaborazioni numeriche, in cui le prestazioni sono della massima importanza. Sarebbe quindi preferibile separare i meccanismi di protezione dai criteri di protezione, permettendo allo stesso sistema di avere una protezione complessa o semplice secondo le necessità dei propri utenti. Per separare i meccanismi dai criteri è necessario un modello di protezione più generale.

14.4 Matrice d'accesso

Il modello generale di protezione qui descritto si può considerare in modo astratto come una matrice, chiamata **matrice d'accesso**. Le righe della matrice rappresentano i domini, e le colonne gli oggetti. Ciascun elemento della matrice consiste di un insieme di diritti d'accesso. Poiché le colonne definiscono esplicitamente gli oggetti, si possono omettere i nomi degli oggetti dai diritti d'accesso. L'elemento $\text{access}(i, j)$ definisce l'insieme di operazioni che un processo in esecuzione nel dominio D_i può richiamare sull'oggetto O_j .

Per spiegare questi concetti si consideri la matrice d'accesso riportata nella Figura 14.3, in cui sono presenti quattro domini e quattro oggetti: tre file (F_1, F_2, F_3) e una stampante. Quando viene eseguito nel dominio D_1 , un processo può leggere i file F_1 ed F_3 . Un processo in esecuzione nel dominio D_4 ha gli stessi privilegi di un processo in esecuzione nel dominio D_1 , ma in più può scrivere anche sui file F_1 e F_3 . Solo un processo in esecuzione nel dominio D_2 può accedere alla stampante.

Lo schema della matrice d'accesso offre un meccanismo che permette di specificare diversi criteri. Il meccanismo consiste nella realizzazione della matrice d'accesso e nella garanzia che le proprietà semantiche sottolineate siano sempre valide. Più specificamente, occorre assicurare che un processo in esecuzione nel dominio D_i possa

dominio \ oggetto	F_1	F_2	F_3	stampante
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figura 14.3 Matrice d’accesso.

accedere solo agli oggetti specificati nella riga i e solo nel modo indicato dagli elementi della matrice d’accesso.

Con la matrice d’accesso si possono attuare i criteri riguardanti la protezione. Tali criteri implicano la scelta dei diritti da inserire nell’ (i, j) -esimo elemento. Occorre anche stabilire il dominio in cui avviene l’esecuzione di ogni processo. Quest’ultimo criterio è generalmente stabilito dal sistema operativo.

Normalmente sono gli utenti a decidere il contenuto degli elementi della matrice d’accesso. Quando un utente crea un nuovo oggetto O_j , si aggiunge la colonna O_j alla matrice d’accesso con gli elementi di inizializzazione stabiliti dal creatore. L’utente può decidere di inserire alcuni diritti in determinati elementi della colonna j e altri diritti in altri elementi, secondo le necessità.

La matrice d’accesso fornisce un meccanismo adeguato alla definizione e realizzazione di uno stretto controllo sia per l’associazione statica sia per l’associazione dinamica tra processi e domini. Quando un processo passa da un dominio all’altro, si esegue un’operazione (*switch*) su un oggetto (il dominio). Il passaggio da un dominio all’altro si può controllare inserendo i domini tra gli oggetti della matrice d’accesso. Analogamente, quando si modifica il contenuto della matrice d’accesso, si esegue un’operazione su un oggetto: la matrice d’accesso. Anche in questo caso si possono controllare le modifiche considerando la stessa matrice d’accesso come un oggetto. In effetti, poiché si può modificare singolarmente, ogni elemento della matrice d’accesso si deve considerare come un oggetto da proteggere. A questo punto si devono considerare solo le operazioni possibili su questi nuovi oggetti, domini e matrice d’accesso, e occorre decidere come debbano essere eseguite dai processi.

I processi devono poter passare da un dominio all’altro. Il passaggio dal dominio D_i al dominio D_j è permesso se e solo se l’operazione $\text{switch} \in \text{access}(i, j)$. Quindi, com’è illustrato nella Figura 14.4, un processo in esecuzione nel dominio D_2 può passare al dominio D_3 oppure al dominio D_4 . Un processo del dominio D_4 può passare al dominio D_1 , e uno del dominio D_1 può passare al dominio D_2 .

oggetto dominio \	F_1	F_2	F_3	stampante	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figura 14.4 Matrice d'accesso della Figura 14.3 con domini come oggetti.

Permettere la modifica controllata del contenuto degli elementi della matrice d'accesso richiede altre tre operazioni: `copy`, `owner` e `control`. Esamineremo queste operazioni nel seguito.

La possibilità di copiare un diritto d'accesso da un dominio (o riga) a un altro della matrice d'accesso è indicata da un asterisco (*) apposto sul diritto d'accesso. Il diritto `copy` permette di copiare il diritto d'accesso solo all'interno della colonna (cioè per l'oggetto) per la quale il diritto stesso è definito. Per esempio, nella Figura 14.5 (a), un processo in esecuzione nel dominio D_2 può copiare l'operazione `read` in un elemento qualsiasi associato al file F_2 . Quindi, la matrice d'accesso della Figura 14.5(a)

oggetto dominio \	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

oggetto dominio \	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figura 14.5 Matrice d'accesso con diritti copy.

si può modificare nella matrice d’accesso illustrata nella Figura 14.5(b). Questo schema ha due ulteriori varianti.

1. Un diritto viene copiato da $\text{access}(i, j)$ ad $\text{access}(k, j)$ e viene successivamente rimosso da $\text{access}(i, j)$; quest’azione è un *trasferimento* di un diritto piuttosto che una copiatura.
2. La propagazione del diritto **copy** può essere limitata. Ciò significa che quando si copia il diritto R^* da $\text{access}(i, j)$ ad $\text{access}(k, j)$, si crea solo il diritto R e non R^* . Un processo in esecuzione nel dominio D_k non può copiare ulteriormente il diritto R .

Un sistema può scegliere uno tra questi tre diritti **copy**, oppure può fornirli tutti e tre identificandoli come diritti separati: **copy**, **transfer** e **limited copy**.

Occorre anche un meccanismo che permetta di aggiungere nuovi diritti e rimuoverne altri; queste operazioni sono controllate dal diritto **owner**. Se $\text{access}(i, j)$ contiene il diritto **owner**, un processo in esecuzione nel dominio D_i può aggiungere e rimuovere qualsiasi diritto di qualsiasi elemento della colonna j . Per esempio, nella Figura 14.6(a) il dominio D_1 è il proprietario di F_1 e quindi può aggiungere e cancellare qualsiasi diritto valido nella colonna di F_1 . Analogamente, il dominio D_2 è proprietario di F_2 e F_3 , quindi può aggiungere e rimuovere qualsiasi diritto valido che si trovi all’interno di queste due colonne. Così, la matrice d’accesso della Figura 14.6(a) si può modificare nella matrice d’accesso illustrata nella Figura 14.6(b).

I diritti **copy** e **owner** permettono a un processo di modificare gli elementi di una colonna. Occorre anche un meccanismo per modificare gli elementi di una riga. Il diritto **control** si può applicare solo a oggetti di dominio. Se $\text{access}(i, j)$ contiene il diritto **control**, allora un processo in esecuzione nel dominio D_i può rimuovere qualsiasi diritto d’accesso dalla riga j . Si supponga, prendendo come esempio la Figura 14.4, di inserire il diritto **control** in $\text{access}(D_2, D_4)$. Quindi un processo in esecuzione nel dominio D_2 può modificare il dominio D_4 , com’è illustrato nella Figura 14.7.

I diritti **copy** e **owner** forniscono un meccanismo per limitare la propagazione dei diritti d’accesso, però non forniscono strumenti adeguati per impedire la propagazione delle informazioni. Il problema di garantire che nessuna informazione, tenuta inizialmente in un oggetto, possa migrare all’esterno del proprio ambiente d’esecuzione si chiama **problema della reclusione** (*confinement problem*). Tale problema è in generale insolubile (si vedano le Note bibliografiche in proposito).

Queste operazioni sui domini e sulla matrice d’accesso non sono di per sé importanti, ma spiegano come il modello della matrice d’accesso consenta la realizzazione e il controllo dei requisiti di protezione dinamica. Nuovi oggetti e nuovi domini si possono creare dinamicamente e inserire nel modello della matrice d’accesso. Tuttavia, in questo paragrafo è spiegato soltanto il meccanismo di base; chi progetta e usa il sistema deve scegliere i criteri riguardanti quali domini, e in che modo, abbiano accesso a determinati oggetti.

oggetto dominio \	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

oggetto dominio \	F_1	F_2	F_3
D_1	owner execute		
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Figura 14.6 Matrice d'accesso con diritti owner.

14.5 Realizzazione della matrice d'accesso

È necessario implementare efficacemente la matrice d'accesso. Tale matrice generalmente è sparsa, ossia la maggior parte dei suoi elementi è vuota. Tuttavia, a causa del modo in cui si usa la funzione di protezione, le tecniche standard di strutturazione dei dati per la rappresentazione delle matrici sparse non sono particolarmente utili

oggetto dominio \	F_1	F_2	F_3	stampante	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figura 14.7 Matrice d'accesso della Figura 14.4 modificata.

per quest'applicazione. Nel seguito descriveremo e confronteremo vari metodi per implementare la matrice d'accesso.

14.5.1 Tabella globale

La realizzazione più semplice della matrice d'accesso è una tabella globale costituita di un insieme di triple ordinate $\langle \text{dominio}, \text{oggetto}, \text{insieme dei diritti} \rangle$. Ogni volta che si esegue un'operazione M su un oggetto O_j del dominio D_i , si cerca una tripla $\langle D_i, O_j, R_k \rangle$ nella tabella globale, tale che $M \in R_k$. Se si trova questa tripla, l'operazione può continuare, altrimenti si verifica una condizione di eccezione (errore).

Questa tipo di realizzazione ha, però, parecchi svantaggi. Generalmente la tabella è grande e non può essere conservata nella memoria centrale, perciò sono richieste ulteriori operazioni di I/O. Per gestire questa tabella spesso si usano tecniche di memoria virtuale; inoltre è difficile anche trarre vantaggio da speciali raggruppamenti di oggetti o domini. Per esempio, se chiunque può leggere un particolare oggetto, esso deve avere un elemento distinto in ogni dominio.

14.5.2 Liste d'accesso per oggetti

Ogni colonna della matrice d'accesso si può realizzare come una lista d'accesso per un oggetto (Paragrafo 11.6.2). Naturalmente gli elementi vuoti si possono scartare. Per ogni oggetto la lista risultante è composta di coppie ordinate $\langle \text{dominio}, \text{insieme dei diritti} \rangle$ che definiscono tutti i domini il cui insieme di diritti d'accesso per quell'oggetto risulta non vuoto.

Questo metodo si può estendere facilmente definendo una lista più un insieme di *default* di diritti d'accesso. Quando nel dominio D_i si tenta un'operazione M su un oggetto O_j , si cerca un elemento $\langle D_i, R_k \rangle$, con $M \in R_k$, nella lista d'accesso relativa all'oggetto O_j . Se si trova quest'elemento, l'operazione è permessa; altrimenti, si controlla l'insieme di default. Se M si trova in questo insieme, l'accesso è permesso, altrimenti l'accesso viene negato e si verifica una condizione di eccezione. Per motivi di efficienza, conviene controllare prima l'insieme di default e poi cercare nella lista d'accesso.

14.5.3 Liste delle abilitazioni per domini

Anziché associare le colonne della matrice d'accesso agli oggetti formando liste d'accesso, è possibile associare ogni riga della matrice al suo dominio. La **lista delle abilitazioni** (*capability list*) per un dominio è una lista di oggetti insieme con le operazioni ammesse su quegli oggetti. Un oggetto è spesso rappresentato dal suo nome fisico o indirizzo, detto **abilitazione** (*capability*). Per eseguire l'operazione M sull'oggetto O_j , il processo esegue l'operazione M , specificando l'abilitazione (puntatore) per l'oggetto O_j come parametro. Il semplice **possesso** dell'abilitazione indica che l'accesso è permesso.

La lista delle abilitazioni è associata a un dominio, ma non è mai direttamente accessibile a un processo che si trova in esecuzione in quel dominio: è un oggetto protetto, mantenuto dal sistema operativo e al quale gli utenti possono accedere solo indirettamente. La protezione basata sulle abilitazioni si fonda sul presupposto che non è mai permessa la migrazione delle abilitazioni in qualsiasi spazio di indirizzi direttamente accessibile a un processo utente, dove queste potrebbero essere modificate. Se tutte le abilitazioni sono sicure, è sicuro anche l'oggetto da esse protetto contro accessi non autorizzati.

Le abilitazioni sono state originariamente proposte come un tipo di puntatore sicuro, per soddisfare la necessità di protezione delle risorse dovuta alla diffusione dei calcolatori multiprogrammati. L'idea di un puntatore intrinsecamente protetto (dal punto di vista dell'utente di un sistema) fornisce la base per una protezione che si può estendere fino a livello delle applicazioni.

Per fornire una protezione intrinseca occorre distinguere le abilitazioni dagli altri oggetti, e interpretarle attraverso una macchina astratta su cui si eseguono programmi di livello superiore. Generalmente le abilitazioni si distinguono dagli altri dati in uno dei due modi seguenti.

- Ogni oggetto ha un'**etichetta** (*tag*) che ne indica il tipo: abilitazione o dati accessibili. Le etichette non devono essere direttamente accessibili ai programmi applicativi. Per imporre tale limite si può ricorrere al supporto dell'hardware o del firmware. Anche se per distinguere tra abilitazioni e altri oggetti è sufficiente un bit, spesso se ne adoperano di più. Questa estensione permette all'hardware di applicare a tutti gli oggetti etichette indicanti i rispettivi tipi. Quindi l'hardware può distinguere interi, numeri in virgola mobile, puntatori, valori booleani, caratteri, istruzioni, abilitazioni e valori non inizializzati grazie alle rispettive etichette.
- Come alternativa, lo spazio d'indirizzi associato a un programma si può dividere in due parti: una contenente i dati e le istruzioni del programma, accessibile al programma; l'altra, contenente la lista delle abilitazioni, accessibile solo al sistema operativo. Lo spazio di memoria segmentato è un utile supporto di questo metodo (Paragrafo 8.4).

Sono stati sviluppati parecchi sistemi di protezione basati sulle abilitazioni, brevemente descritti nel Paragrafo 14.8. Anche il sistema operativo Mach fa uso di un tipo di protezione basata sulle abilitazioni.

14.5.4 Meccanismo chiave-serratura

Lo **schema chiave-serratura** (*lock-key scheme*) rappresenta un compromesso tra le liste d'accesso e le liste di abilitazioni. Ogni oggetto ha una lista di sequenze di bit uniche, chiamate **serrature** (*lock*); analogamente, ogni dominio ha una lista di sequenze di bit uniche, chiamate **chiavi** (*key*). Un processo in esecuzione in un dominio può accedere a un oggetto solo se quel dominio ha una chiave che corrisponde a una delle serrature dell'oggetto.

Come le liste delle abilitazioni, anche la lista delle chiavi per un dominio deve essere gestita dal sistema operativo per conto del dominio. Gli utenti non possono esaminare o modificare direttamente la lista delle chiavi o delle serrature.

14.5.5 Confronto

Come è ovvio, la scelta di una tecnica di implementazione della matrice degli accessi comporta vari trade-off. L'uso di una tabella globale è relativamente semplice; tuttavia, tale tabella può essere piuttosto grande e spesso non può avvantaggiarsi dalla esistenza di gruppi speciali di oggetti o domini. Le liste d'accesso corrispondono direttamente alle necessità degli utenti. Quando un utente crea un oggetto, può specificare quali domini abbiano accesso a quell'oggetto e quali operazioni siano permesse. Tuttavia, poiché le informazioni sui diritti d'accesso per un particolare dominio non sono localizzate, è difficile stabilire l'insieme dei diritti d'accesso per ogni dominio. Inoltre ogni accesso all'oggetto deve essere controllato, il che richiede una ricerca nella lista d'accesso. In un grande sistema con lunghe liste d'accesso, questa ricerca può causare un notevole spreco di tempo.

Le liste delle abilitazioni non corrispondono direttamente alle necessità degli utenti, ma sono utili per localizzare le informazioni per un dato processo. Un processo che tenti un accesso deve presentare la relativa abilitazione, quindi il sistema di protezione deve verificare soltanto che l'abilitazione sia valida. Tuttavia la revoca delle abilitazioni può essere inefficiente; questo problema è trattato nel Paragrafo 14.7.

Il meccanismo chiave-serratura rappresenta un compromesso tra questi due schemi. Il meccanismo può essere efficace e flessibile, a seconda della lunghezza delle chiavi, che possono essere passate liberamente da dominio a dominio. Inoltre i privilegi d'accesso si possono revocare in modo semplice ed efficace modificando alcune chiavi associate all'oggetto; anche questo problema è trattato nel Paragrafo 14.7.

La maggior parte dei sistemi adopera una combinazione di liste d'accesso e liste di abilitazioni. Quando un processo tenta per la prima volta di accedere a un oggetto, si fa una ricerca nella lista d'accesso. Se l'accesso viene negato, si verifica una condizione di eccezione, altrimenti si crea un'abilitazione che si associa al processo. I riferimenti successivi si servono dell'abilitazione per dimostrare rapidamente che l'accesso è consentito. Dopo l'ultimo accesso, l'abilitazione viene distrutta. Questa strategia è usata nel sistema MULTICS e nel sistema CAL.

Come esempio di come opera questa strategia, si consideri un file system in cui a ogni file è associata una lista d'accesso. Quando un processo apre un file, si fa una ricerca nella directory per trovare il file, si controlla il permesso d'accesso e si assegnano i buffer per l'I/O. Tutte queste informazioni si registrano in un nuovo elemento della tabella dei file associata al processo. L'operazione riporta un indice in questa tabella per il file appena aperto, tramite il quale si compiono tutte le operazioni sul file. Quindi l'elemento della tabella dei file punta al file e ai suoi buffer. Quando il file viene chiuso, si cancella l'elemento della tabella dei file. Poiché la tabella dei file è mantenuta dal sistema operativo, gli utenti non possono alterarla accidentalmente, quindi gli utenti possono accedere ai soli file che sono stati aperti. Poiché l'accesso

viene controllato al momento dell'apertura del file, la protezione è assicurata. Nel sistema operativo UNIX si adopera questa strategia.

Il diritto d'accesso si deve ancora controllare per ogni accesso e l'elemento della tabella dei file contiene l'abilitazione per le sole operazioni ammesse. Se si apre un file per la lettura, nell'elemento della tabella dei file viene inserita un'abilitazione d'accesso alla lettura. Se si tenta di scrivere in quel file, il sistema rileva questa violazione della protezione confrontando l'operazione richiesta con l'abilitazione presente nell'elemento della tabella dei file.

14.6 Controllo dell'accesso

Nel Paragrafo 11.6.2 abbiamo descritto l'uso dei controlli dell'accesso ai file in un file system. A tutti i file e le directory è assegnato un proprietario, un gruppo o, in alcuni casi, una lista di utenti, e a ciascuna di tali entità sono assegnate informazioni di controllo dell'accesso. Una funzionalità simile può essere realizzata per altri aspetti del sistema; Solaris 10 ne è un buon esempio.

Solaris 10 migliora la protezione offerta dal sistema operativo applicando il principio del minor privilegio tramite il **controllo dell'accesso basato sui ruoli** (*role-based access control*, RBAC). Questa funzionalità si basa sui privilegi. Si dice privilegio il diritto di eseguire una chiamata di sistema o di sfruttare un'opzione di tale chiamata (come l'apertura di un file con accesso in scrittura). I privilegi possono essere assegnati ai processi, limitandoli al minimo indispensabile per svolgere il compito cui sono preposti. Inoltre, privilegi e programmi possono essere assegnati sulla base di **ruoli**. Un utente può avere un ruolo assegnato o può assumere un ruolo tramite l'uso di una password. In questo modo, un utente può assumere un ruolo che gli attribuisce un certo privilegio; ciò permette all'utente di eseguire un programma per portare a termine un compito specifico, come rappresentato nella Figura 14.8. Questa organizzazione attenua i rischi per la sicurezza del sistema, in particolare quelli legati ai superuser e ai programmi e setuid.

Si noti l'analogia di questa tecnica con la matrice d'accesso, trattata nel Paragrafo 14.4. Questa relazione sarà esplorata più compiutamente negli esercizi che concludono il capitolo.

14.7 Revoca dei diritti d'accesso

In un sistema di protezione dinamica talvolta può essere necessario revocare i diritti d'accesso a oggetti condivisi da diversi utenti. A proposito della revoca si possono presentare diverse questioni.

- **Immediata o ritardata.** Occorre stabilire se la revoca ha effetto immediatamente o con ritardo. Se la revoca è ritardata, occorre stabilire quando avverrà.
- **Selettiva o generale.** Quando si revoca un diritto d'accesso a un oggetto, occorre stabilire se la revoca vale per *tutti* gli utenti che hanno un diritto d'accesso a

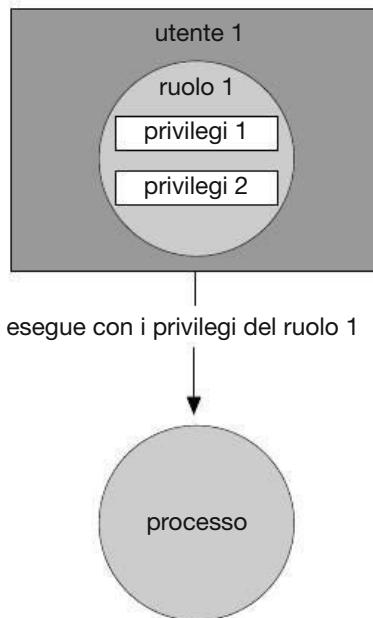


Figura 14.8 Controllo dell'accesso basato sui ruoli in Solaris 10.

quell'oggetto, oppure se si può specificare un gruppo di utenti a cui si debbano revocare i diritti d'accesso.

- **Parziale o totale.** Occorre stabilire se si può revocare un sottoinsieme di diritti associati a un oggetto, oppure se si devono revocare tutti i diritti d'accesso a quest'oggetto.
- **Temporanea o permanente.** Occorre stabilire se l'accesso si può revocare in modo permanente, cioè il diritto d'accesso non sarà più disponibile, oppure se può essere nuovamente ottenuto.

Disponendo di un sistema basato su liste d'accesso, effettuare una revoca è facile. Si cercano i diritti d'accesso da revocare nella lista d'accesso, e li si cancellano dalla lista. La revoca è immediata e può essere generale o selettiva, totale o parziale, permanente o temporanea.

La revoca delle abilitazioni, invece, è molto più difficile. Poiché le abilitazioni sono distribuite su tutto il sistema, occorre prima trovarle e poi revocarle. Tra gli schemi che realizzano la revoca delle abilitazioni ci sono i seguenti.

- **Riacquisizione.** Le abilitazioni vengono cancellate periodicamente da ogni dominio. Se un processo intende usare un'abilitazione, può scoprire che quell'abilitazione è stata cancellata. Il processo allora può tentare di riacquistarla. Se l'accesso è stato revocato, il processo non è più in grado di riacquisire l'abilitazione.
- **Puntatori all'indietro.** Insieme a ciascun oggetto si conserva una lista di puntatori a tutte le abilitazioni a esso associate. Quando si richiede una revoca, si possono seguire questi puntatori, modificando le abilitazioni secondo le necessità. Questo

schema era adottato nel sistema MULTICS, ed è abbastanza generale, anche se la sua realizzazione è onerosa.

- **Riferimento indiretto.** Le abilitazioni non puntano direttamente agli oggetti. Ogni abilitazione punta all'elemento di una tabella globale che a sua volta punta all'oggetto. La revoca si realizza cercando nella tabella globale l'elemento desiderato e cancellandolo. Quando si tenta l'accesso, si riscontra che l'abilitazione punta a un elemento illegale della tabella. Gli elementi della tabella si possono riutilizzare senza difficoltà per altre abilitazioni, poiché sia l'abilitazione sia l'elemento della tabella contengono il nome unico dell'oggetto. L'oggetto dell'abilitazione e dell'elemento nella tabella devono corrispondere. Questo schema, adottato nel sistema CAL, non permette la revoca selettiva.
- **Chiavi.** Una chiave è una sequenza unica di bit associabile a ogni abilitazione. Tale chiave viene definita al momento della creazione dell'abilitazione e non può essere modificata né esaminata dal processo proprietario dell'abilitazione stessa. Esiste una **chiave principale** (*master key*) associata a ogni oggetto, che si può definire o sostituire con l'operazione `set-key`. Quando si crea un'abilitazione, a questa si associa il valore corrente della chiave principale. Quando l'abilitazione viene esercitata, si confronta la sua chiave con la chiave principale. Se le due chiavi coincidono, l'operazione può continuare, altrimenti si verifica una condizione di eccezione. La revoca sostituisce la chiave principale con un valore nuovo tramite l'operazione `set-key`, che invalida tutte le abilitazioni precedenti per quest'oggetto. Questo schema non permette la revoca selettiva, poiché a ogni oggetto è associata solo una chiave principale. Se a ogni oggetto si associa una lista di chiavi, si può realizzare la revoca selettiva. Infine, tutte le chiavi si possono raggruppare in una tabella globale di chiavi. Un'abilitazione è valida solo se la sua chiave coincide con una delle chiavi della tabella globale. La revoca si realizza rimuovendo dalla tabella la chiave coincidente. In questo schema una chiave si può associare a più oggetti, e più chiavi si possono associare a ciascun oggetto, offrendo la massima flessibilità. Negli schemi basati sulle chiavi, le operazioni di definizione, inserimento in liste e cancellazione dalle liste delle chiavi stesse non devono essere a disposizione di tutti gli utenti. In particolare, è ragionevole permettere soltanto al proprietario di un oggetto di impostare le chiavi per quell'oggetto. Questa scelta, in ogni modo, riguarda un criterio (*policy*), che il sistema di protezione può realizzare ma non deve definire.

14.8 Sistemi basati su abilitazioni

In questo paragrafo si esaminano due sistemi di protezione basati su abilitazioni. Questi sistemi sono diversi nel grado di complessità e nel tipo di criteri realizzabili su di essi. Nessuno dei due è molto usato, ma entrambi sono interessanti banchi di prova delle teorie sulla protezione.

14.8.1 Un esempio: Hydra

Il sistema Hydra è un sistema di protezione basato sulle abilitazioni, caratterizzato da una notevole flessibilità. Il sistema fornisce un prefissato insieme di possibili diritti d'accesso tra cui sono presenti le principali forme d'accesso come il diritto per lettura, scrittura o esecuzione di un segmento di memoria. Inoltre il sistema offre agli utenti (del sistema di protezione) gli strumenti necessari per dichiarare altri diritti. I diritti definiti dagli utenti sono interpretati esclusivamente dai programmi dell'utente, ma il sistema fornisce la protezione degli accessi nell'uso di questi diritti e dei diritti definiti dal sistema. Queste funzioni costituiscono un notevole passo avanti nella tecnologia della protezione.

Le operazioni sugli oggetti sono definite in modo procedurale, e le procedure che realizzano tali operazioni sono a loro volta oggetti, accessibili indirettamente attraverso abilitazioni. I nomi delle procedure definite dagli utenti si devono comunicare al sistema di protezione, se deve gestire oggetti del tipo definito dagli utenti. Quando si comunica la definizione di un oggetto al sistema Hydra, i nomi delle operazioni sul tipo diventano **diritti ausiliari**. Questi diritti ausiliari si possono descrivere in un'abilitazione per un'istanza del tipo. Affinché un processo possa eseguire un'operazione su un oggetto tipizzato, l'abilitazione per quell'oggetto deve contenere tra i diritti ausiliari il nome dell'operazione invocata. Questo limite permette di distinguere i diritti d'accesso istanza per istanza e processo per processo.

Il sistema Hydra offre anche l'**amplificazione dei diritti**. Questo schema permette di certificare che una procedura è *fidata* per agire su un parametro formale di un tipo specificato, per conto di qualsiasi processo che abbia un diritto d'esecuzione della procedura. I diritti posseduti da una procedura fidata sono indipendenti dai diritti posseduti dal processo chiamante e possono anche superarli. Tuttavia tale procedura non deve essere considerata universalmente fidata (per esempio, alla procedura non è permesso di agire su altri tipi), e l'affidabilità non deve essere estesa a qualsiasi altra procedura o altro segmento di programma che può essere eseguito da un processo.

L'amplificazione permette alle procedure di accedere alle variabili che rappresentano il tipo di dato astratto. Se, per esempio, un processo possiede un'abilitazione a un oggetto tipizzato *A*, quest'abilitazione può comprendere un diritto ausiliario a richiamare una generica operazione *P*, ma non può comprendere nessuno dei cosiddetti diritti kernel, come i diritti per lettura, scrittura o esecuzione, sul segmento che rappresenta *A*. Tale abilitazione offre a un processo un mezzo per accedere indirettamente, tramite l'operazione *P*, alla rappresentazione di *A*, ma solo per scopi specifici.

D'altra parte, quando un processo invoca l'operazione *P* su un oggetto *A*, l'abilitazione d'accesso ad *A* può essere amplificata quando il controllo passa al corpo del codice di *P*. Quest'amplificazione può essere necessaria per conferire a *P* il diritto d'accesso al segmento di memoria che rappresenta *A*, così da realizzare l'operazione che *P* definisce sul tipo di dato astratto. Si può permettere al corpo del codice di *P* di leggere o scrivere direttamente nel segmento di *A*, anche se il processo chiamante non può farlo. Al termine di *P*, si riporta l'abilitazione per *A* al suo stato originale non amplificato. Questo è un tipico caso in cui i diritti posseduti da un processo per accedere

a un segmento protetto devono cambiare dinamicamente, secondo il compito da svolgere. La regolazione dinamica dei diritti si esegue per garantire la coerenza delle astrazioni definite dai programmatori. Nel sistema operativo Hydra l'amplificazione dei diritti si può stabilire in modo esplicito nelle dichiarazioni dei tipi di dato astratti.

Quando un utente passa un oggetto come argomento a una procedura, può essere necessario assicurare che la procedura non possa modificare l'oggetto. Questa limitazione si può realizzare facilmente passando un diritto d'accesso senza diritto di modifica (scrittura). Tuttavia, se l'amplificazione è possibile, il diritto di modifica può essere ripristinato, e il requisito di protezione dell'utente può quindi essere aggirato. In generale, naturalmente, un utente può supporre che una procedura esegua correttamente il proprio compito. Tale ipotesi, però, non sempre è corretta, poiché possono verificarsi errori fisici o logici. Il sistema Hydra risolve questo problema limitando le amplificazioni.

Il meccanismo di chiamata di procedura del sistema Hydra è stato progettato come una diretta soluzione al *problema dei sottosistemi mutuamente sospetti*. Questo problema è definito come segue. Si supponga che per un programma possa essere invocato come servizio da diversi utenti (per esempio, una procedura di ordinamento, un compilatore, un videogioco). Quando gli utenti invocano questo programma di servizio, corrono il rischio che il programma non funzioni bene e possa danneggiare i dati forniti, o trattenere, senza averne l'autorità, qualche diritto d'accesso ai dati, da adoperare in seguito. Analogamente, il programma di servizio può avere alcuni file privati, per esempio file di accounting, cui il programma utente chiamante non deve accedere direttamente. Il sistema Hydra fornisce meccanismi per affrontare questo problema in modo diretto.

Un sottosistema di Hydra è costruito sopra un kernel di protezione e può richiedere la protezione dei propri componenti. Un sottosistema interagisce con il kernel tramite un insieme di primitive stabilite dal kernel, che definiscono i diritti d'accesso alle risorse definite dal sottosistema. I criteri relativi all'uso di queste risorse da parte dei processi utenti possono essere definiti da chi progetta il sottosistema, ma si implementano tramite l'uso della protezione degli accessi standard offerta dal sistema di abilitazioni.

Un programmatore può servirsi direttamente del sistema di protezione, dopo aver studiato le sue caratteristiche. Il sistema Hydra offre un'ampia libreria di procedure definite dal sistema che i programmi utenti possono impiegare. Un utente del sistema Hydra può incorporare esplicitamente le chiamate dirette a queste procedure di sistema nel codice del proprio programma, oppure servirsi di un traduttore di programmi interfacciato ad Hydra.

14.8.2 Un esempio: sistema Cambridge CAP

Un diverso orientamento alla protezione basata sulle abilitazioni è stato seguito nella progettazione del sistema Cambridge CAP. Il sistema di abilitazioni di CAP è più semplice e a prima vista meno potente di quello di Hydra. Tuttavia, con un esame più attento, è possibile capire che anche questo sistema si può usare per offrire una prote-

zione sicura agli oggetti definiti dagli utenti. CAP ha due tipi di abilitazioni. Il tipo ordinario si chiama **abilitazione dati**, e si può impiegare per fornire l’accesso agli oggetti, ma gli unici diritti forniti sono quelli ordinari di lettura, scrittura o esecuzione dei singoli segmenti di memoria associati all’oggetto. Le abilitazioni dati sono interpretate dal microcodice della macchina CAP.

Il secondo tipo è la cosiddetta **abilitazione software**, che è protetta, ma non interpretata, dal microcodice di CAP. L’interpretazione spetta a una procedura *protetta* (cioè privilegiata) che può essere scritta da un programmatore di applicazioni come parte di un sottosistema. A una procedura protetta è associato un particolare tipo di amplificazione di diritti. Quando si esegue il corpo del codice di tale procedura, un processo acquisisce temporaneamente i diritti di lettura o scrittura del contenuto dell’abilitazione software stessa. Questo particolare tipo di amplificazione di diritti corrisponde a una realizzazione delle primitive `seal` e `unseal` sulle abilitazioni. Naturalmente questo privilegio rimane soggetto alla verifica del tipo per assicurare che solo le abilitazioni software per uno specifico tipo astratto possano passare a una particolare procedura. Nessun codice gode di totale fiducia, tranne il microcodice della macchina CAP. Per avere riferimenti in merito si vedano le Note bibliografiche alla fine del capitolo.

L’interpretazione di un’abilitazione software è lasciata esclusivamente al sottosistema, che la esegue per mezzo delle proprie procedure protette. Questo schema permette di realizzare un gran numero di criteri di protezione. Benché un programmatore possa definire le proprie procedure protette, la sicurezza del sistema nel suo complesso non può essere compromessa (anche se tali procedure contengono errori). Il sistema di protezione di base non permetterebbe a una procedura protetta non verificata, definita dall’utente, di accedere a qualsiasi segmento di memoria, o abilitazione, che non appartenga all’ambiente di protezione in cui la procedura stessa risiede. La conseguenza più grave dovuta a una procedura protetta non sicura è la violazione della protezione del sottosistema del quale quella procedura è responsabile.

I progettisti del sistema CAP hanno notato che l’uso delle abilitazioni software permette di fare notevoli economie nella formulazione e nella realizzazione di criteri di protezione adeguati ai requisiti delle risorse astratte. Tuttavia un progettista di sottosistemi che voglia usare questa funzione non può semplicemente studiare un manuale, come nel caso del sistema Hydra, ma deve apprenderne i principi e le tecniche di protezione, poiché il sistema non offre alcuna libreria di procedure.

14.9 Protezione basata sul linguaggio

Il grado di protezione fornito negli attuali sistemi elaborativi è di solito ottenuto attraverso il kernel del sistema operativo, che si occupa di controllare e convalidare ogni tentativo d’accesso a una risorsa protetta. Poiché una completa convalida degli accessi è potenzialmente una fonte di notevole sovraccarico, o si dispone del supporto dell’hardware per ridurre il costo di ogni convalida, o si deve accettare un compromesso rispetto agli obiettivi della protezione. Se la flessibilità di realizzazione dei cri-

teri di protezione è limitata dai meccanismi di supporto disponibili, o se gli ambienti di protezione sono resi più grandi di quanto è necessario per assicurare una maggiore efficienza, soddisfare tutti questi obiettivi è difficile.

Gli scopi della protezione sono stati perfezionati con l'aumentare della complessità dei sistemi operativi e soprattutto con il tentativo di fornire interfacce utente di livello superiore. I progettisti dei sistemi di protezione si sono basati in modo determinante su idee nate nell'ambito dei linguaggi di programmazione e, soprattutto, sui concetti di tipo di dati astratti e di oggetto. Attualmente i sistemi di protezione non riguardano solo l'identità di una risorsa a cui si tenta di accedere, ma anche la natura funzionale di quell'accesso. Nei sistemi di protezione più recenti la azione di controllo sulle funzioni da invocare va oltre un insieme di funzioni definite dal sistema, come gli ordinari metodi per l'accesso ai file, per comprendere anche funzioni definibili dagli utenti.

Anche i criteri concernenti l'uso delle risorse variano secondo l'applicazione e, con il passare del tempo, possono essere soggetti a cambiamenti. Per questi motivi la protezione non può più essere considerata come un esclusivo interesse del progettista di un sistema operativo; dovrebbe essere uno strumento disponibile al progettista di applicazioni per proteggere le risorse di un sottosistema d'applicazione da manomissioni o effetti dovuti a errori.

14.9.1 Controllo realizzato dal compilatore

A questo punto entrano in gioco i linguaggi di programmazione. Specificare il controllo degli accessi desiderato per una risorsa condivisa significa fare un'asserzione dichiarativa su tale risorsa. Questo tipo di dichiarazione si può integrare in un linguaggio di programmazione estendendone la funzione di tipizzazione. Quando insieme alla tipizzazione dei dati si dichiara anche la protezione, i progettisti dei sottosistemi possono specificare i propri requisiti di protezione, come anche la necessità di utilizzare altre risorse del sistema. Tali specificazioni si dovrebbero fornire direttamente nella fase di stesura dei programmi, e nello stesso linguaggio in cui si scrivono i programmi. Questo metodo presenta importanti vantaggi:

1. le necessità di protezione si devono semplicemente dichiarare e non programmare come una sequenza di chiamate di procedure di un sistema operativo;
2. i requisiti di protezione si possono stabilire indipendentemente dalle funzioni fornite da uno specifico sistema operativo;
3. i progettisti dei sottosistemi non devono fornire i meccanismi di controllo dei criteri;
4. la notazione dichiarativa è naturale, perché i privilegi d'accesso sono strettamente connessi al concetto linguistico di tipo di dati.

L'implementazione di un linguaggio di programmazione può fornire diverse tecniche per imporre protezioni, ma ciascuna di loro deve dipendere in qualche misura dalle funzioni di supporto offerte dalla macchina sottostante e dal suo sistema operativo.

Si supponga, per esempio, che un linguaggio sia impiegato per generare codice da eseguire sul sistema Cambridge CAP. In questo sistema ogni riferimento alla memoria effettuato sull'hardware sottostante avviene indirettamente tramite abilitazioni. Questo limite impedisce a un processo qualsiasi di accedere a una risorsa esterna al proprio ambiente di protezione. Tuttavia un programma può imporre limitazioni arbitrarie su come una risorsa può essere usata durante l'esecuzione di un particolare segmento di codice. Tali limiti si possono realizzare in modo pressoché immediato ricorrendo alle abilitazioni software offerte da CAP. L'implementazione di un linguaggio di programmazione potrebbe fornire procedure protette standard per interpretare le abilitazioni software che realizzano i criteri di protezione specificabili nel linguaggio stesso. Questo schema permette ai programmatore di specificare i criteri di protezione, senza costringerli a occuparsi dei dettagli riguardanti i relativi meccanismi di supporto.

Anche se un sistema non offre un kernel di protezione potente come quelli di Hydra o CAP, esistono meccanismi che permettono la realizzazione delle specifiche di protezione presenti in un linguaggio di programmazione. La differenza principale è dovuta al fatto che la *sicurezza* di questa protezione non è grande quanto quella gestita da un kernel di protezione, poiché il meccanismo si deve basare su un maggior numero di assunzioni sullo stato operativo del sistema. Un compilatore può separare i riferimenti per i quali non è possibile avere violazioni di protezione da quelli per i quali la violazione può avvenire, e trattarli in modi diversi. La sicurezza offerta da questo tipo di protezione si fonda sul presupposto che il codice generato dal compilatore non venga modificato prima o durante la sua esecuzione.

Di seguito sono elencati i vantaggi dei meccanismi basati esclusivamente su un kernel, rispetto a quelli che si hanno con i meccanismi forniti da un compilatore.

- **Sicurezza.** Il controllo effettuato da un kernel offre un grado di sicurezza del sistema di protezione maggiore di quello offerto dalla generazione, da parte di un compilatore, di codice per il controllo della protezione. In uno schema supportato dal compilatore, la sicurezza è basata sulla correttezza del traduttore, su qualche meccanismo di gestione della memoria che protegge i segmenti dai quali si esegue il codice compilato e, in ultima analisi, sulla sicurezza dei file dai quali si carica il programma. Alcune di queste considerazioni valgono anche per un kernel di protezione supportato esclusivamente dal software, ma in questo caso in misura minore poiché il kernel può risiedere in segmenti fissati di memoria fisica e può essere caricato solo da un file designato. In un sistema di abilitazioni con etichette, in cui tutto il calcolo degli indirizzi è eseguito dall'hardware o da un microprogramma fisso, si può avere una sicurezza ancora maggiore. La protezione basata sull'hardware è anche relativamente immune dalle violazioni della protezione verificabili a causa di malfunzionamenti sia fisici sia logici.
- **Flessibilità.** La flessibilità di un kernel di protezione ha alcuni limiti per quel che riguarda la realizzazione di criteri di protezione definiti dagli utenti, anche se può fornire al sistema le funzioni necessarie per l'imposizione dei propri criteri. Con un linguaggio di programmazione i criteri di protezione si possono dichiarare, così come si può fornire il controllo richiesto tramite l'implementazione del linguag-

gio. Se un linguaggio non offre una sufficiente flessibilità, si può estendere o sostituire, interferendo con il sistema in servizio meno di quanto non accadrebbe modificando il kernel del sistema operativo.

- **Efficienza.** La maggior efficienza si ha quando la protezione è gestita direttamente dall'hardware (o dal microcodice). Il controllo basato sul linguaggio ha il vantaggio di poter verificare il controllo degli accessi in maniera statica nella fase di compilazione. Inoltre, poiché un compilatore intelligente può adattare il meccanismo di controllo alla specifica necessità, spesso si può evitare l'overhead fisso delle chiamate del kernel.

Riepilogando, la specificazione della protezione in un linguaggio di programmazione permette la descrizione ad alto livello di criteri di allocazione e uso di risorse. L'implementazione di un linguaggio di programmazione può fornire gli strumenti per la realizzazione della protezione quando non è disponibile il controllo automatico gestito dall'hardware; inoltre può interpretare le indicazioni di protezione per generare chiamate a qualsiasi sistema di protezione sia fornito dall'hardware e dal sistema operativo.

Un modo per rendere disponibile la protezione ai programmi applicativi prevede l'uso delle abilitazioni software come oggetti di calcolo. Questo concetto è basato sull'idea che alcuni componenti di programmi possano avere il privilegio di creare o esaminare queste abilitazioni. Un programma che crea un'abilitazione può eseguire un'operazione primitiva (`seal`) che sigilla una struttura dati, rendendo il contenuto di quest'ultima inaccessibile a qualsiasi componente di programma che non possieda i privilegi `seal` o `unseal`. Questi componenti potrebbero copiare la struttura dati, o passarne l'indirizzo ad altri componenti del programma, ma non possono ottenere l'accesso al suo contenuto. Tali abilitazioni software si introducono per portare un meccanismo di protezione all'interno del linguaggio di programmazione. L'unico problema che s'incontra seguendo tale metodo riguarda l'uso delle operazioni `seal` e `unseal`; tale uso richiede infatti un orientamento procedurale alla specifica della protezione. Per rendere disponibile l'ambiente di protezione ai programmatore di applicazioni sembra preferibile una notazione non procedurale o dichiarativa.

È necessario un meccanismo dinamico sicuro di controllo degli accessi, per poter distribuire tra i processi utenti le abilitazioni alle risorse del sistema. Per contribuire all'affidabilità generale di un sistema, il meccanismo per il controllo dell'accesso deve essere utilizzabile in modo sicuro, e per essere utile nella pratica deve essere anche ragionevolmente efficiente. Questo requisito ha portato allo sviluppo di un certo numero di costrutti di linguaggio che consentono ai programmatore di dichiarare vari limiti riguardanti l'uso di specifiche risorse (si vedano le Note bibliografiche per gli appropriati riferimenti).

Questi costrutti forniscono meccanismi per tre funzioni:

1. distribuire, in modo sicuro ed efficiente, le abilitazioni tra i processi clienti: in particolare i meccanismi assicurano che un processo utente si servirà di una risorsa solo se gli è stata concessa una specifica abilitazione;

2. specificare il tipo di operazioni che un processo particolare può compiere su una risorsa assegnata (per esempio, a un lettore di file si deve permettere soltanto di leggere i file, mentre a uno scrittore si possono concedere sia lettura sia scrittura): non dovrebbe essere necessario concedere lo stesso insieme di diritti a ogni processo utente e un processo non dovrebbe avere la possibilità di ampliare il proprio insieme di diritti d'accesso, tranne che se abbia ricevuto l'autorizzazione da parte del meccanismo di controllo degli accessi;
3. specificare l'ordine in cui un processo particolare può compiere le diverse operazioni su una risorsa (per esempio, un file deve essere aperto prima di essere letto): deve essere possibile dare a due processi limitazioni diverse sull'ordine in cui possono compiere le operazioni sulla risorsa assegnata.

L'incorporazione dei concetti di protezione nei linguaggi di programmazione, intesa come strumento pratico per la progettazione dei sistemi, è in una fase iniziale. La protezione probabilmente acquisterà un crescente interesse da parte dei progettisti di nuovi sistemi con architetture distribuite, e requisiti sempre più severi sulla sicurezza dei dati; sarà quindi riconosciuta più diffusamente anche l'importanza d'idonee notazioni di linguaggio in cui esprimere i requisiti di protezione.

14.9.2 Protezione nel linguaggio Java

Poiché Java è stato pensato per l'esecuzione in ambiente distribuito, la macchina virtuale Java, o JVM (*Java virtual machine*), è dotata di molti meccanismi di protezione. I programmi Java sono composti da **classi**, ognuna delle quali è un insieme di campi di dati e di funzioni (chiamate **metodi**) che operano su quei campi. La JVM carica una classe come risposta a una richiesta di creazione di istanze (o oggetti) di quella classe. Una tra le caratteristiche più originali e utili del linguaggio Java è la gestione del caricamento dinamico di classi non fidate da una rete e dell'esecuzione all'interno della stessa JVM di classi che si ritengono mutuamente sospette.

Proprio per queste caratteristiche, il problema della protezione è di vitale importanza. Le classi eseguite dalla stessa JVM possono provenire da diverse fonti e possono avere diversi gradi di affidabilità. Quindi, è insufficiente imporre la protezione a livello del processo della JVM. Intuitivamente, il fatto che una richiesta d'apertura di file sia consentita dipenderà generalmente da quale classe ha fatto la richiesta d'apertura. Il sistema operativo non ha queste informazioni.

Dunque questo tipo di decisioni di protezione sono gestite all'interno della JVM. Quando la JVM carica una classe, la assegna a un dominio di protezione che fornisce i permessi per quella classe. Il dominio di protezione al quale si assegna una classe dipende dall'URL da cui la classe è stata caricata e da eventuali firme digitali sul file della classe (le firme digitali sono trattate nel Paragrafo 15.4.1.3). Un file che permette la configurazione dei criteri di protezione determina i permessi che si dànno al dominio (e alle sue classi). Per esempio, le classi prelevate da un server fidato si potrebbero mettere in un dominio di protezione che permette a tali classi di accedere ai file

nelle directory iniziali degli utenti, mentre le classi prelevate da un server ritenuto non fidato potrebbero non avere alcun permesso d'accesso ai file.

Per una JVM può essere difficile stabilire quale sia la classe responsabile di una richiesta d'accesso a una risorsa protetta. Gli accessi avvengono spesso in modo indiretto, per mezzo di librerie di sistema o altre classi. Si consideri per esempio una classe cui non sia permesso aprire connessioni di rete. Potrebbe chiamare una libreria di sistema per richiedere il caricamento dei contenuti di un URL. La JVM deve decidere se aprire a no una connessione di rete per questa richiesta. Ma quale classe si dovrebbe considerare per determinare se si debba concedere o no la connessione, l'applicazione o la libreria di sistema?

L'orientamento seguito nel linguaggio Java è quello di richiedere alla classe di libreria di permettere esplicitamente una connessione di rete. Più in generale, per poter accedere a una risorsa protetta, uno dei metodi nella sequenza delle chiamate che ha portato alla richiesta deve esplicitamente asserire il privilegio di accedere alla risorsa. In questo modo, il metodo si *assume la responsabilità* della richiesta e, presumibilmente, eseguirà anche tutti i controlli necessari ad assicurare la sicurezza della richiesta stessa. Chiaramente, non tutti i metodi sono abilitati ad asserire privilegi; lo possono fare solo se la classe d'appartenenza è in un dominio di protezione che ha esso stesso il permesso di esercitare quel privilegio.

Questo metodo di realizzazione si chiama **ispezione dello stack**. Ogni thread nella JVM ha uno stack associato per le sue attuali invocazioni di metodi. Quando un chiamante può non essere fidato, un metodo esegue una richiesta d'accesso all'interno di un blocco `doPrivileged()`, per accedere direttamente o indirettamente a una risorsa protetta. `doPrivileged()` è un metodo statico della classe `AccessController` al quale si passa una classe con un metodo `run()` da invocare. Quando l'esecuzione entra nel blocco `doPrivileged()`, si annota l'elemento dello stack per questo metodo per indicare questo fatto, e successivamente si eseguono le istruzioni nel blocco. Quando, in seguito, si richiede l'accesso a una risorsa protetta, o da parte di questo metodo o da parte di un metodo da esso chiamato, s'invoca `checkPermissions()` per richiedere l'ispezione dello stack, allo scopo di determinare se l'accesso richiesto debba essere concesso. L'ispezione consiste nell'esame degli elementi presenti nello stack del thread chiamante, partendo da quello inserito più recentemente e procedendo verso il meno recente. Se prima si trova un elemento che ha l'annotazione `doPrivileged()`, `checkPermissions()` permette immediatamente l'accesso. Se invece si trova prima un elemento per il quale l'accesso non è consentito nell'ambito del dominio di protezione della classe del metodo, `checkPermissions()` genera un'eccezione `AccessControlException`. Se l'ispezione esaurisce lo stack senza trovare né un tipo d'elemento né l'altro, allora la concessione dell'accesso dipende dalla implementazione (alcune implementazioni della JVM permettono l'accesso, altre lo negano).

La procedura d'ispezione dello stack è illustrata nella Figura 14.9. In quest'esempio, il metodo `gui()` di una classe nel dominio di protezione *untrusted applet* compie due operazioni: prima una `get()` e poi una `open()`. La prima corrisponde all'invocazione del metodo `get()` di una classe nel dominio di protezione *URL loader*, che

dominio di protezione:	<i>applet</i> non fidata	caricatore di URL	interconnessione
permesso della socket:	nessuno	*.lucent.com:80, connect	qualsiasi
classe:	gui: ... get(url); open(addr);	get(URL u): ... doPrivileged { open('proxy.lucent.com:80'); } <request u from proxy>	open(Addr a): ... checkPermission(a, connect); connect (a);

Figura 14.9 Ispezione dello stack.

ha i permessi per aprire sessioni nei siti nel dominio `lucent.com`, e in particolare per il server `proxy.lucent.com` per individuare gli URL. Per questa ragione, l'invocazione di `get()` dall'*applet* non fidata andrà a buon fine: la chiamata a `checkPermissions()` nella libreria di rete troverà l'elemento dello stack relativo al metodo `get()` che ha eseguito la sua `open()` in un blocco `doPrivileged()`. Tuttavia, l'invocazione della `open()` da parte dell'*applet* non fidata risulta invece in un'eccezione, poiché la chiamata a `checkPermissions()` non trova alcuna annotazione `doPrivileged()` prima di raggiungere l'elemento dello stack relativo al metodo `gui()`.

Ovviamente, affinché l'ispezione dello stack possa funzionare, un programma non deve poter modificare le annotazioni sul suo stesso elemento dello stack, né compiere altre manipolazioni che interferiscano con l'ispezione dello stack. Questa è una delle differenze più importanti tra il linguaggio Java e molti altri linguaggi (compreso il C++). Un programma scritto in Java non può accedere direttamente alla memoria. Piuttosto, può manipolare solo oggetti per i quali ha un riferimento. I riferimenti non si possono contraffare e le manipolazioni si compiono per mezzo d'interfacce ben definite. La conformità a questi criteri è imposta da un raffinato insieme di controlli della fase di caricamento e della fase d'esecuzione. Ne segue che un oggetto non può manipolare il proprio stack, poiché non può ottenere un riferimento né a essa né ad altri componenti del sistema di protezione.

Più in generale, i controlli del linguaggio Java nella fase di caricamento e nella fase d'esecuzione impongono la **sicurezza dei tipi** (*type safety*) delle classi. La sicurezza dei tipi garantisce che le classi non possano trattare gli interi come puntatori, scrivere oltre la fine di un array, accedere alla memoria in modi arbitrari. Un programma può accedere a un oggetto soltanto attraverso i metodi definiti per quell'oggetto dalla sua classe. Su ciò si fonda il sistema di protezione del linguaggio Java, poiché permette a una classe di *incapsulare* efficacemente i propri dati e metodi e di proteggerli da altre classi caricate nella stessa JVM. Per esempio, una variabile si può definire `private`, in modo che solo la classe che la contiene possa accedervi, oppure si può definire `protected`, in modo che possano accedervi solo la classe che la contiene, le sue sottoclassi e le classi dello stesso package. La sicurezza dei tipi garantisce l'imposizione di queste limitazioni.

14.10 Sommario

I sistemi elaborativi contengono molti oggetti, che devono essere protetti contro i possibili abusi. Gli oggetti possono essere hardware (come la memoria, il tempo di CPU o i dispositivi di I/O) o software (come i file, i programmi e i semafori). Un diritto d'accesso è un permesso per eseguire un'operazione su un oggetto. Un dominio è un insieme di diritti d'accesso. I processi vengono eseguiti in domini e possono usare tutti i diritti d'accesso del dominio per accedere agli oggetti e manipolarli. Durante il suo ciclo di vita un processo può essere vincolato a un dominio di protezione o può essergli consentito di passare da un dominio a un altro.

La matrice d'accesso è un modello generale di protezione; fornisce un meccanismo per la protezione senza imporre uno specifico criterio di protezione al sistema o ai suoi utenti. La separazione tra criteri e meccanismi è un'importante caratteristica di progettazione.

La matrice d'accesso è sparsa e normalmente si realizza per mezzo di liste d'accesso associate a ciascun oggetto, oppure per mezzo di liste di abilitazioni associate a ciascun dominio. Si può inserire la protezione dinamica nel modello della matrice d'accesso considerando i domini e la stessa matrice d'accesso come oggetti. La revoca dei diritti d'accesso in un modello di protezione dinamico è di solito più facile da realizzare con lo schema delle liste d'accesso che con le liste di abilitazioni.

I sistemi reali sono molto più limitati e tendono a fornire protezioni solo per i file. UNIX è un caso tipico, poiché per ogni file fornisce le protezioni per lettura, scrittura ed esecuzione, distinte per proprietario, gruppo e per chiunque. Il sistema MULTICS, oltre alla protezione dei file, impiega una struttura dei domini ad anelli. Hydra, il sistema Cambridge CAP e Mach sono sistemi con abilitazioni che estendono la protezione agli oggetti software definiti dagli utenti. Solaris 10 realizza il principio del privilegio minimo attraverso il controllo dell'accesso basato sul ruolo, una forma di matrice d'accesso.

La protezione basata sul linguaggio offre un controllo delle richieste e dei privilegi più selettivo di quello ottenibile con il sistema operativo. Per esempio, una singola JVM può eseguire molti thread, ognuno in un diverso dominio di protezione. La JVM controlla le richieste di risorse attraverso un raffinato meccanismo di ispezione dello stack e attraverso la sicurezza dei tipi offerta dal linguaggio.

Esercizi di ripasso

- 14.1** Quali sono le principali differenze tra liste delle abilitazioni e liste d'accesso?
- 14.2** Un file nel Borroughs B7000/B6000 MCP può essere contrassegnato come dato sensibile. Quando un tale file viene cancellato, l'area in cui era memorizzato viene sovrascritta da bit casuali. Per quali scopi un tale schema può essere utile?
- 14.3** In un sistema di protezione ad anello il livello 0 ha l'accesso più ampio agli oggetti, e il livello n (con $n > 0$) ha diritti di accesso più bassi. I diritti di ac-

cesso di un programma a un dato livello nell’anello sono considerati un insieme di abilitazioni. Che relazione c’è tra le abilitazioni a un oggetto per un dominio a livello j e un dominio a livello i (per $j > i$)?

- 14.4** Il sistema RC 4000, come altri sistemi, ha definito un albero dei processi tale che tutti i discendenti di un processo possono ottenere risorse (oggetti) e diritti di accesso solo dai loro antenati. Un discendente non potrà quindi mai fare niente di quello che gli antenati non possono fare. La radice dell’albero è il sistema operativo, che ha la capacità di fare ogni cosa. Assumete che l’insieme dei diritti di accesso sia rappresentato tramite una matrice di accesso A . $A(x,y)$ definisce i diritti di accesso del processo x a un oggetto y . Se x è discendente di z , che relazione c’è tra $A(x,y)$ e $A(z,y)$, per un oggetto arbitrario y ?
- 14.5** Quali problemi di protezione possono nascere se viene utilizzata uno stack condiviso per il passaggio di parametri?
- 14.6** Considerate un ambiente elaborativo in cui a ogni processo e a ogni oggetto del sistema sia associato un numero univoco. Supponete che un processo con numero n possa accedere a oggetti con numero m solo se $n > m$. Quale tipo di struttura di protezione avremmo?
- 14.7** Considerare un ambiente elaborativo in cui un processo ottiene il privilegio di accedere a un oggetto soltanto n volte. Suggerite uno schema per implementare una tale politica.
- 14.8** Se tutti i diritti di accesso di un oggetto vengono cancellati, l’oggetto non è più accessibile. A questo punto, lo stesso oggetto dovrebbe essere cancellato, e lo spazio da lui occupato dovrebbe essere restituito al sistema. Suggerite un’efficiente implementazione di questo schema.
- 14.9** Quali difficoltà si incontrano nella protezione di un sistema in cui gli utenti hanno il permesso di eseguire le loro operazioni di I/O?
- 14.10** La lista delle abilitazioni è solitamente mantenuta nello spazio di indirizzi dell’utente. In che modo il sistema assicura che l’utente non possa modificare il contenuto della lista?

Esercizi

- 14.11** Considerate la struttura di protezione ad anelli di MULTICS. Se dovessimo realizzare le chiamate di sistema di un tipico sistema operativo, memorizzandole in un segmento associato all’anello 0, quali sarebbero i valori da memorizzare nel campo concernente l’anello del descrittore del segmento? Che cosa succede durante una chiamata di sistema, quando un processo in esecuzione in un anello superiore invoca una procedura dell’anello 0?

- 14.12** Grazie alla matrice per il controllo dell'accesso si può determinare se un processo sia abilitato a passare, per esempio, dal dominio A al dominio B, e se possa godere dei privilegi d'accesso del dominio B. Valutate se questa soluzione sia da considerarsi equivalente all'inclusione dei privilegi d'accesso del dominio B in quelli del dominio A.
- 14.13** Considerate un sistema in cui i videogiochi possano essere usati dagli studenti solo tra le 22 e le 6, dai membri della facoltà tra le 17 e le 8 e dal personale del centro di calcolo a tutte le ore. Suggerite uno schema per realizzare efficacemente questo criterio.
- 14.14** Dite quali caratteristiche dell'hardware di un calcolatore siano necessarie per ottenere un'efficiente manipolazione delle abilitazioni. Dite se queste caratteristiche si possano adoperare per la protezione della memoria.
- 14.15** Evidenziate i punti di forza e le vulnerabilità di una matrice d'accesso realizzata tramite le liste d'accesso associate agli oggetti.
- 14.16** Evidenziate i punti di forza e le vulnerabilità di una matrice d'accesso realizzata tramite le abilitazioni associate ai domini.
- 14.17** Spiegate perché i sistemi basati sull'abilitazione, come Hydra, siano più flessibili nell'applicare i parametri di protezione rispetto alla struttura di protezione ad anelli.
- 14.18** Esaminate la necessità dell'amplificazione dei diritti nel sistema Hydra. Quali tratti specifici contraddistinguono questa pratica da quella delle chiamate attraverso più anelli in una struttura ad anelli?
- 14.19** Dite che cos'è il principio della necessità di sapere, e perché è importante che un sistema di protezione aderisca a questo principio.
- 14.20** Chiarite quale tra i seguenti sistemi consente ai progettisti di moduli di rendere effettivo il principio della necessità di sapere.
- a. La struttura di protezione ad anelli di MULTICS.
 - b. Le funzionalità del sistema Hydra.
 - c. Lo schema di ispezione dello stack della JVM.
- 14.21** Descrivete in che modo il modello di protezione del linguaggio Java verrebbe meno se si permettesse a un programma scritto in Java di alterare direttamente le annotazioni nel suo elemento dello stack.
- 14.22** In che cosa si assomigliano la funzionalità della matrice d'accesso e il controllo dell'accesso basato sui ruoli? In che cosa differiscono?
- 14.23** In che modo il principio del privilegio minimo aiuta a creare sistemi di protezione?
- 14.24** Come possono persistere, nei sistemi che adottano il principio del privilegio minimo, carenze di protezione tali da causare violazioni alla sicurezza?

Note bibliografiche

Il modello di protezione della matrice d’accesso tra domìni e oggetti è sviluppato in [Lampson 1969] e [Lampson 1971]. [Popek 1974] e [Saltzer e Schroeder 1975] presentano eccellenti rassegne sull’argomento della protezione. [Harrison et al. 1976] si serve di una versione formale della matrice di accesso per dimostrare matematicamente alcune proprietà di un sistema di protezione.

Il concetto di abilitazione è un’evoluzione di quello di *codeword*, di Iliffe e Jodeit, impiegato nel calcolatore della Rice University, [Iliffe e Jodeit 1962]. Il termine *capability* (*abilitazione*) è stato introdotto da [Dennis e Horn 1966].

Il sistema Hydra è descritto in [Wulf et al. 1981]. Il sistema CAP è descritto in [Needham e Walker 1977]. [Organick 1972] discute il sistema di protezione ad anelli del sistema MULTICS.

La revoca è trattata in [Redell e Fabry 1974], [Cohen e Jefferson 1975] e [Ekanadham e Bernstein 1979]. Il principio di separazione tra criteri e meccanismi di protezione è stato sostenuto dal progettista del sistema Hydra, [Levin et al. 1975]. Il problema della reclusione è stato trattato per la prima volta in [Lampson 1973] ed esaminato successivamente in [Lipner 1975].

L’uso di linguaggi di livello più alto per specificare il controllo degli accessi è stato suggerito per la prima volta da [Morris 1973], che ha proposto l’uso delle operazioni `seal` e `unseal`, descritte nel Paragrafo 14.9. [Kieburz e Silberschatz 1978], [Kieburz e Silberschatz 1983] e [McGraw e Andrews 1979] hanno proposto diversi costrutti di linguaggio per la gestione degli schemi dinamici generali per la gestione delle risorse. [Jones e Liskov 1978] considera il problema dell’incorporazione di uno schema statico di controllo degli accessi in un linguaggio di programmazione che prevede i tipi di dati astratti. Un coinvolgimento minimo del sistema operativo nell’attuare la protezione è stato perorato dal Progetto Exokernel ([Ganger et al. 2002], [Kaashoek et al. 1997]). L’estensibilità del codice di sistema per mezzo di meccanismi di protezione basati sul linguaggio è stata discussa in [Bershad et al. 1995b]. Altre tecniche di attuazione della protezione comprendono il cosiddetto *sandboxing* [Goldberg et al. 1996] e l’isolamento dei malfunzionamenti software [Wahbe et al. 1993]. Le tematiche dell’abbassamento dei costi di gestione legati alla protezione e della concessione dell’accesso a livello utente ai dispositivi di rete sono state trattate in [McCanne e Jacobson 1993], e in [Basu et al. 1995].

Un’analisi più dettagliata dell’ispezione dello stack, che comprende il confronto con altri approcci alla sicurezza dell’ambiente Java, si trova in [Wallach et al. 1997] e [Gong et. al. 1997].

Bibliografia

[Basu et al. 1995] A. Basu, V. Buch, W. Vogels e T. von Eicken, “U-Net: A User-Level Network Interface for Parallel and Distributed Computing”, Proceedings of the ACM Symposium on Operating Systems Principles, 1995.

- [Bershad et al. 1995]** B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers e C. Chambers, “Extensibility, Safety and Performance in the SPIN Operating System”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 267–284, 1995.
- [Cohen e Jefferson 1975]** E. S. Cohen e D. Jefferson, “Protection in the Hydra Operating System”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 141–160, 1975.
- [Dennis e Horn 1966]** J. B. Dennis e E. C.V. Horn, “Programming Semantics for Multiprogrammed Computations”, Communications of the ACM, Vol. 9, Num. 3, p. 143–155, 1966.
- [Ekanadham e Bernstein 1979]** K. Ekanadham e A. J. Bernstein, “Conditional Capabilities”, IEEE Transactions on Software Engineering, Vol. SE-5, Num. 5, p. 458–464, 1979.
- [Ganger et al. 2002]** G. R. Ganger, D. R. Engler, M. F. Kaashoek, H.M. Briceno, R. Hunt e T. Pinckney, “Fast and Flexible Application-Level Networking on Exokernel Systems”, ACM Transactions on Computer Systems, Vol. 20, Num. 1, p. 49–83, 2002.
- [Goldberg et al. 1996]** I. Goldberg, D. Wagner, R. Thomas e E. A. Brewer, “A Secure Environment for Untrusted Helper Applications”, Proceedings of the 6th Usenix Security Symposium, 1996.
- [Gong et al. 1997]** L. Gong, M. Mueller, H. Prafullchandra e R. Schemers, “Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2”, Proceedings of the USENIX Symposium on Internet Technologies and Systems, 1997.
- [Harrison et al. 1976]** M. A. Harrison, W. L. Ruzzo e J. D. Ullman, “Protection in Operating Systems”, Communications of the ACM, Vol. 19, Num. 8, p. 461–471, 1976.
- [Iliffe e Jodeit 1962]** J. K. Iliffe e J. G. Jodeit, “A Dynamic Storage Allocation System”, Computer Journal, Vol. 5, Num. 3, p. 200–209, 1962.
- [Jones e Liskov 1978]** A. K. Jones e B. H. Liskov, “A Language Extension for Expressing Constraints onDataAccess”, Communications of the ACM, Vol. 21, Num. 5, p. 358–367, 1978.
- [Kaashoek et al. 1997]** M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti e K. Mackenzie, “Application Performance and Flexibility on Exokernel Systems”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 52–65, 1997.
- [Kieburstz e Silberschatz 1978]** R. B. Kieburstz e A. Silberschatz, “Capability Managers”, IEEE Transactions on Software Engineering, Vol. SE-4, Num. 6, p. 467–477, 1978.
- [Kieburstz e Silberschatz 1983]** R. B. Kieburstz e A. Silberschatz, “Access Right Expressions”, ACM Transactions on Programming Languages and Systems, Vol. 5, Num. 1, p. 78–96, 1983.

- [**Lampson 1969**] B.W. Lampson, “Dynamic Protection Structures”, Proceedings of the AFIPS Fall Joint Computer Conference, p. 27–38, 1969.
- [**Lampson 1971**] B. W. Lampson, “Protection”, Proceedings of the Fifth Annual Princeton Conference on Information Systems Science, p. 437–443, 1971.
- [**Lampson 1973**] B. W. Lampson, “A Note on the Confinement Problem”, Communications of the ACM, Vol. 10, Num. 16, p. 613–615, 1973.
- [**Levin et al. 1975**] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack e W. A. Wulf, “Policy/Mechanism Separation in Hydra”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 132–140, 1975.
- [**Lipner 1975**] S. Lipner, “A Comment on the Confinement Problem”, Operating System Review, Vol. 9, Num. 5, p. 192–196, 1975.
- [**McCanne e Jacobson 1993**] S. McCanne e V. Jacobson, “The BSD Packet Filter: A New Architecture for User-level Packet Capture”, USENIX Winter, p. 259–270, 1993.
- [**McGraw e Andrews 1979**] J. R. McGraw e G. R. Andrews, “Access Control in Parallel Programs”, IEEE Transactions on Software Engineering, Vol. SE-5, Num. 1, p. 1–9, 1979.
- [**Morris 1973**] J. H. Morris, “Protection in Programming Languages”, Communications of the ACM, Vol. 16, Num. 1, p. 15–21, 1973.
- [**Needham e Walker 1977**] R. M. Needham e R. D. H. Walker, “The Cambridge CAP Computer and Its Protection System”, Proceedings of the Sixth Symposium on Operating System Principles, p. 1–10, 1977.
- [**Organick 1972**] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
- [**Popek 1974**] G. J. Popek, “Protection Structures”, Computer, Vol. 7, Num. 6, p. 22–33, 1974.
- [**Redell e Fabry 1974**] D. D. Redell e R. S. Fabry, “Selective Revocation of Capabilities”, Proceedings of the IRIA International Workshop on Protection in Operating Systems, p. 197–210, 1974.
- [**Saltzer e Schroeder 1975**] J. H. Saltzer e M. D. Schroeder, “The Protection of Information in Computer Systems”, Proceedings of the IEEE, p. 1278–1308, 1975.
- [**Wahbe et al. 1993**] R. Wahbe, S. Lucco, T. E. Anderson e S. L. Graham, “Efficient Software-Based Fault Isolation”, ACM SIGOPS Operating Systems Review, Vol. 27, Num. 5, p. 203–216, 1993.
- [**Wallach et al. 1997**] D. S. Wallach, D. Balfanz, D. Dean e E. W. Felten, “Extensible Security Architectures for Java”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 116–128, 1997.
- [**Wulf et al. 1981**] W. A. Wulf, R. Levin e S. P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, 1981.

CAPITOLO

15

OBIETTIVI DEL CAPITOLO

- Analisi di minacce e attacchi contro la sicurezza.
- Princìpi di cifratura, autenticazione e funzioni hash.
- Applicazioni della crittografia nei sistemi elaborativi.
- Descrizione delle possibili contromisure per neutralizzare gli attacchi alla sicurezza.

Sicurezza

La protezione, così com’è stata esaminata nel Capitolo 14, è un problema strettamente *interno* e riguarda la realizzazione di un accesso controllato a programmi e dati memorizzati in un calcolatore. La **sicurezza**, d’altra parte, non richiede solo un adeguato sistema di protezione, ma anche la considerazione dell’ambiente *esterno* in cui opera il sistema. Un sistema di protezione è inefficace se l’autenticazione degli utenti è compromessa o se un programma viene eseguito da un utente non autorizzato.

Le risorse del computer vanno preservate da accessi non autorizzati, distruzione o alterazione dolosa e da involontaria introduzione di elementi di incoerenza. Queste risorse includono l’informazione memorizzata nel sistema sotto forma di dati e programmi, così come CPU, memoria, dischi, nastri e connessioni di rete che l’elaboratore gestisce. In questo capitolo partiamo esaminando le modalità con cui può verificarsi l’uso improprio delle risorse, sia esso intenzionale o fortuito. Prenderemo quindi in considerazione un fondamentale meccanismo di abilitazione della sicurezza: la crittografia. In conclusione studieremo i meccanismi in grado di riconoscere e neutralizzare gli attacchi.

15.1 Il problema della sicurezza

Per molte applicazioni la garanzia della sicurezza nei sistemi elaborativi merita un impegno di notevole portata. I grandi sistemi commerciali, contenenti le retribuzioni aziendali o altri documenti finanziari, sono obiettivi invitanti per i ladri. Sistemi che custodiscono i dati inerenti all'operatività di un'azienda possono suscitare l'interesse di concorrenti senza scrupoli. A ciò si aggiunga che la perdita involontaria di tali dati, o la loro sottrazione dolosa, può pregiudicare gravemente il funzionamento di un'azienda.

Nel Capitolo 14 sono trattati alcuni meccanismi offerti dai sistemi operativi (con l'ausilio di appropriate caratteristiche dell'hardware) per consentire agli utenti di proteggere le loro risorse (di solito programmi e dati). Questi meccanismi funzionano bene fintanto che gli utenti si conformano alle modalità d'uso e d'accesso previste per tali risorse. Si dice che un sistema è **sicuro** se, in ogni circostanza, vi si accede e si utilizzano le risorse solo secondo le modalità previste. Sfortunatamente non è possibile ottenere una sicurezza totale; ciononostante si deve poter disporre di meccanismi che rendano l'elusione della sicurezza un caso raro e non la norma.

Le violazioni della sicurezza del sistema si possono classificare come intenzionali (dolose) o accidentali. È più semplice proteggere un sistema contro le violazioni accidentali che contro quelle dolose. I meccanismi di protezione sono, nella maggior parte dei casi, la base per la difesa dalle violazioni di sicurezza. Nell'elenco che segue sono comprese sia le intrusioni accidentali sia le violazioni dolose. È bene notare che indicheremo con le espressioni *intruso*, *aggressore* e *pirata informatico* coloro che tentino di attuare infrazioni della sicurezza; inoltre chiameremo *minaccia* ogni potenziale pericolo per la sicurezza (quale, per esempio, la scoperta di una vulnerabilità), mentre il termine *attacco* denoterà il tentativo di infrangere le misure di sicurezza.

- **Violazione della riservatezza.** Questo tipo di violazione consiste nella lettura non autorizzata dei dati (o nel furto di informazioni), una finalità tipica degli intrusi. Con la sottrazione di dati segreti da un sistema, o l'intercettazione i dati in transito, un intruso può ricavare denaro da informazioni, quali i numeri delle carte di credito o dalle informazioni sull'identità dell'utente.
- **Compromissione dell'integrità.** Questa violazione si realizza con la modifica non autorizzata dei dati. Attacchi simili possono, per esempio, causare il trasferimento di una responsabilità a un soggetto innocente o la modifica del codice sorgente di un'importante applicazione commerciale.
- **Violazione della disponibilità.** Questo abuso consiste nella distruzione non autorizzata di dati. Alcuni pirati informatici (*cracker*), pur di vedere accresciuta la propria “reputazione”, preferiscono infliggere danni devastanti ai sistemi piuttosto che guadagnare denaro. Il sabotaggio dei siti web è un classico esempio di questo tipo di infrazione.

- **Appropriazione del servizio.** Questa violazione è relativa all'uso non autorizzato delle risorse. A titolo di esempio, un intruso (o un programma di intrusione) potrebbe installare un demone che funga da file server all'interno di un sistema.
- **Rifiuto del servizio.** Questa violazione impedisce l'utilizzo legittimo del sistema. Gli attacchi di rifiuto del servizio, o DOS (*Denial-Of-Service*), sono talvolta accidentali. Il primo worm di Internet si trasformò in un attacco DOS, quando un baco causò la sua rapida proliferazione. Ritorneremo sugli attacchi DOS più avanti, nel Paragrafo 15.3.3.

Gli **aggressori** ricorrono a numerosi metodi standard per tentare di infrangere la sicurezza. Il più comune è l'**attacco mimetico** (*masquerading*), che si realizza quando, in una comunicazione, un partecipante finge di essere qualcun'altro (un'altra persona o un'altra macchina). Gli aggressori usano l'attacco mimetico per violare l'**autenticazione** (*authentication*), cioè la correttezza dell'identificazione; in questo modo possono ottenere permessi d'accesso che normalmente gli sarebbero negati, oppure scalare privilegi, appropriandosi di diritti per i quali non avrebbero titolo. La riproduzione di informazioni catturate nell'ambito di uno scambio di dati rappresenta un'altra modalità di attacco. Un **attacco replay** (*replay attack*) è basato sulla ripetizione dolosa o fraudolenta di informazioni valide già trasmesse. Talvolta l'attacco si esaurisce nella semplice riproduzione, con la reiterazione, per esempio, di una richiesta di trasferimento di denaro. Spesso invece è accompagnato da una **modifica del messaggio** (*message modification*) che mira, ancora una volta, ad accaparrarsi privilegi non dovuti. Si consideri il danno che potrebbe sorgere se, per una richiesta di autenticazione, si sostituissero alle vere informazioni su un utente quelle di un utente non autorizzato. Ancora diversa è la tipologia di **attacco di interposizione** (*man-in-the-middle attack*), che avviene quando, nel flusso di dati di una comunicazione, si intromette un attaccante, che imita il trasmettitore nei confronti del ricevitore, e viceversa. In una comunicazione di rete, un attacco man-in-the-middle può essere preceduto da un **dirottamento della sessione** (*session hijacking*), con cui è intercettata una sessione di comunicazione attiva. La Figura 15.1 illustra varie modalità di attacco.

Come si è detto, una protezione totale del sistema dalle violazioni non è possibile, ma le si può rendere di complessità tale da scoraggiare la maggior parte degli intrusi. In alcuni casi, quali gli attacchi DOS, anche se è preferibile evitare l'attacco, è comunque sufficiente che esso sia rilevato per adottare contromisure appropriate.

Per proteggere il sistema è necessario prendere misure di sicurezza a quattro livelli.

1. **Fisico.** I siti che ospitano i sistemi elaborativi devono essere protetti fisicamente contro gli accessi armati o furtivi da parte d'intrusi. Vanno protetti sia i luoghi che ospitano le macchine sia le stazioni di lavoro o i terminali che vi hanno accesso.
2. **Umano.** L'autorizzazione degli utenti richiede cautela, per garantire che accedano al sistema solo gli utenti che ne hanno diritto. Persino gli utenti autorizzati, tuttavia, potrebbero essere "incoraggiati" a cedere le loro credenziali ad altri (in cambio di una bustarella, per esempio). Inoltre, possono essere spinti con l'inganno a

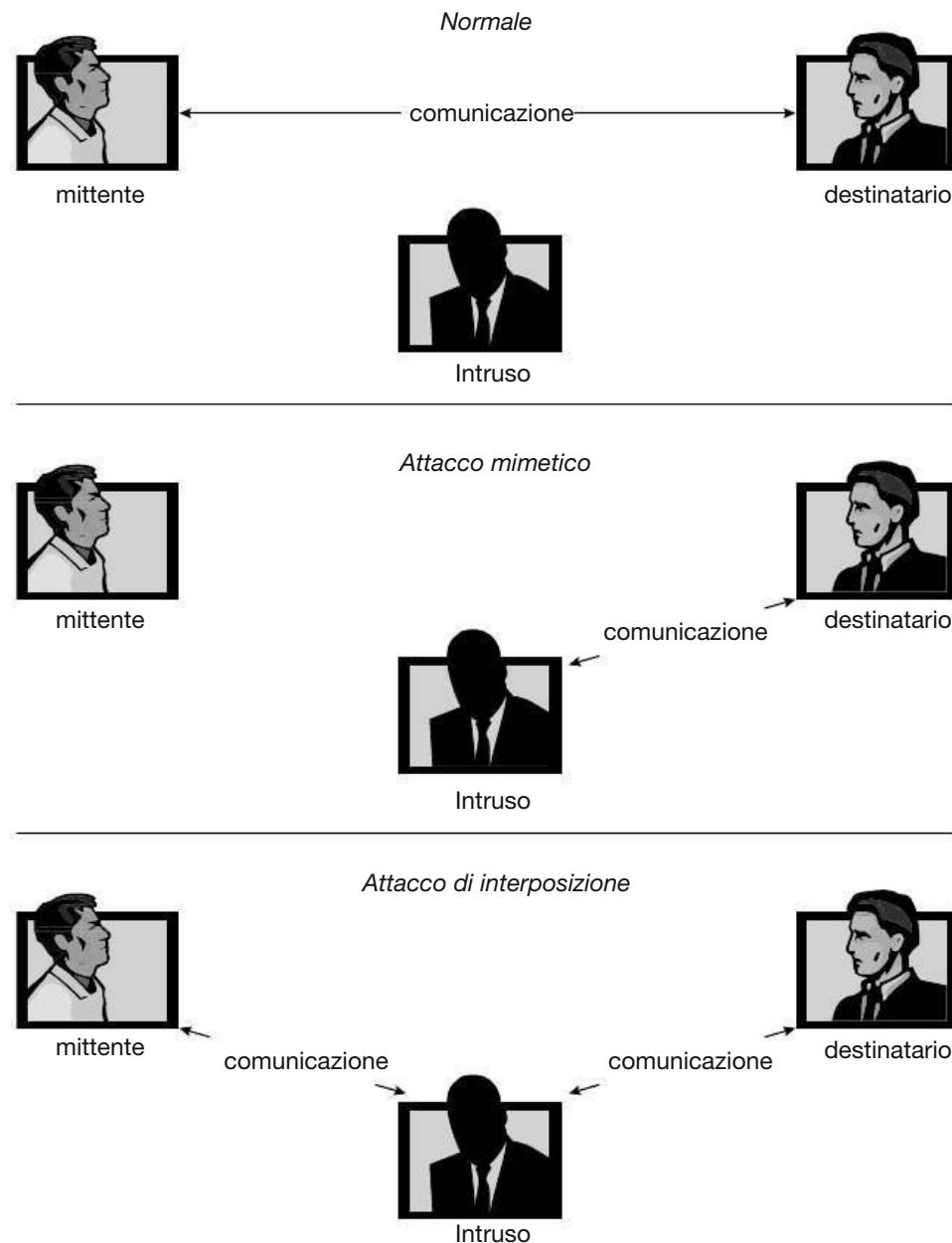


Figura 15.1 Attacchi comuni alla sicurezza.

tale condotta, mediante tecniche di **ingegneria sociale** (*social engineering*). Una tipologia di attacco che si serve dell’ingegneria sociale è il **phishing**¹. Questa tecnica consiste nel contraffare e-mail o pagine web, rendendole in tutto simili a quelle autentiche, per spingere gli utenti tratti in inganno a comunicare informazioni

¹ Il termine *phishing* si pronuncia come *fish*, ossia “pescare”, suggerendo che i pirati “pescano” le informazioni fornite dagli utenti che “abboccano” (N.d.R.).

confidenziali. Un'altra pratica è nota come **frugare nella spazzatura** (*dumpster diving*), un'espressione che rimanda al tentativo di raccogliere informazioni, onde procurarsi abusivamente l'accesso a un calcolatore (per esempio andando alla ricerca di rubriche telefoniche o di foglietti contenenti password fra i rifiuti). Questioni simili non competono ai sistemi operativi, ma sono piuttosto problemi di management e gestione del personale.

3. **Sistema operativo.** Il sistema deve proteggere se stesso dalle violazioni della sicurezza intenzionali o accidentali. Un processo fuori controllo può dar luogo a un attacco DOS accidentale. L'interrogazione di un servizio potrebbe rendere pubbliche alcune password. Da un attacco basato su uno stack overflow può avere origine il lancio di un processo non autorizzato. L'elenco delle possibili violazioni è quasi interminabile.
4. **Rete.** Molti dati informatici nei sistemi moderni viaggiano in linee di comunicazione private, linee condivise (come quelle della rete Internet), connessioni wireless o accessi dial-up. L'intercettazione di questi dati può essere tanto dannosa quanto l'intrusione in un calcolatore. L'interruzione di queste comunicazioni potrebbe costituire un attacco DOS remoto, che riduce le possibilità d'uso da parte degli utenti e l'affidamento che si può fare sul sistema.

Per garantire la sicurezza del sistema operativo è necessario mantenere la sicurezza ai primi due livelli: un punto debole ad alto livello della sicurezza (fisico o umano) consente di eludere le misure di sicurezza più rigorose a un livello più basso (sistema operativo). Pertanto il vecchio adagio, secondo il quale la catena si spezza nel suo anello più debole, trova conferma nel caso della sicurezza dei sistemi. Dovendo garantire la sicurezza non può essere sottovalutato alcun aspetto.

Inoltre l'architettura del sistema deve offrire caratteristiche di protezione (Capitolo 14) al fine di consentire la realizzazione di funzioni di sicurezza. Senza la facoltà di autorizzare utenti e processi, di controllarne l'accesso, e di registrare le loro attività, sarebbe impossibile, per un sistema operativo, un'esecuzione sicura o l'implementazione delle procedure di sicurezza. Le funzionalità di protezione fornite dall'hardware sono indispensabili per mettere in atto una complessiva strategia di protezione. Un sistema privo di protezione della memoria, per esempio, non può essere sicuro. Le più recenti funzionalità dell'hardware, come vedremo, favoriscono una migliore tutela dei sistemi.

Sfortunatamente la sicurezza ha ben poco di prevedibile. Contromisure di sicurezza vengono elaborate e messe in atto dopo che gli intrusi hanno sfruttato i punti deboli di un sistema; ciò spinge gli incursori a raffinare le modalità di attacco. Tra gli episodi recenti di violazione si può citare, per esempio, l'utilizzo di programmi *spyware*, che sfruttano sistemi ignari per inviare posta elettronica indesiderata, detta *spam* (illustreremo questa prassi nel Paragrafo 15.2). È probabile che questa rincorsa continuerà, con la necessità di incrementare gli strumenti per bloccare tecniche e attività perfezionate dagli incursori.

Nel seguito del capitolo tratteremo la sicurezza a livello del sistema operativo; la sicurezza ai livelli umano e fisico, benché importante, va oltre gli scopi di questo li-

bro. La sicurezza nei sistemi operativi e tra sistemi operativi si realizza in diversi modi: dalle password per l'autenticazione alla vigilanza contro i virus, fino alla scoperta delle intrusioni. Per cominciare, tratteremo le minacce alla sicurezza.

15.2 Minacce legate ai programmi

I processi, insieme al kernel, costituiscono per un elaboratore gli unici strumenti per portare a termine del lavoro. Di conseguenza, scrivere un programma che crei una falla nella sicurezza o introdurre anomalie nell'esecuzione di un processo per ottenere il medesimo risultato, è un normale obiettivo dei pirati informatici. In realtà, molte infrazioni alla sicurezza indipendenti dai programmi hanno spesso l'obiettivo di generare minacce ai programmi stessi. Benché sia utile, per esempio, accedere a un sistema senza autorizzazione, è di gran lunga più vantaggioso lasciarsi alle spalle un **demone di back-door** (letteralmente: *porta sul retro*), in grado di fornire informazioni o procurare facilmente l'accesso, anche se l'attacco principale è stato bloccato. In questo paragrafo analizziamo i metodi più comuni con cui i programmi possono generare violazioni della sicurezza. Poiché vi è notevole discrepanza nelle definizioni convenzionali delle falliche di sicurezza, useremo i termini più comuni o più descrittivi.

15.2.1 Cavalli di Troia

Molti sistemi dispongono di meccanismi che permette agli utenti di usare programmi scritti da altri. Se questi programmi si eseguono in un dominio che fornisce i diritti d'accesso dell'utente che esegue il programma, gli altri utenti possono abusare di questi diritti. Un text editor, per esempio, può includere il codice per la ricerca di certe parole chiave; se le parole vengono trovate, tutto il file può essere copiato in un'area speciale accessibile al creatore del text editor. Un segmento di codice che abusi del suo ambiente è detto **cavallo di Troia**. I lunghi percorsi di ricerca, come quelli diffusi nei sistemi UNIX, aggravano il problema dei cavalli di Troia. Il percorso di ricerca elenca l'insieme delle directory in cui compiere la ricerca quando si sottopone un nome di programma ambiguo. Se si trova un file con il nome corrispondente lungo il percorso, questo viene eseguito. Tutte le directory che si trovano nel percorso di ricerca devono essere sicure, altrimenti un cavallo di Troia può insinuarsi nel percorso dell'utente ed essere eseguito senza intenzione.

Si consideri, per esempio, l'uso del carattere “.” in un percorso di ricerca; tale carattere indica all'interprete dei comandi di includere la directory corrente nella ricerca. Quindi, se un utente ha il carattere “.” nel suo percorso di ricerca, ha impostato la sua directory corrente alla directory di un amico e inserisce il nome di un normale comando di sistema, il comando potrebbe essere eseguito dalla directory dell'amico. Il programma sarebbe eseguito all'interno del dominio dell'utente, consentendo al programma stesso di far tutto ciò che è consentito fare all'utente, compresa, per esempio, la cancellazione dei suoi file.

Una variante del cavallo di Troia è un programma che emula una procedura di login: l'ignaro utente, nella fase d'accesso a un terminale, crede di aver scritto erroneamente la propria password; prova ancora e, questa volta, ha successo. Ciò che realmente accade in un caso come questo è che un emulatore della procedura d'accesso alla sessione di lavoro sottrae il nome utente e la password dell'utente. L'emulatore copia la password e mostra un messaggio d'errore nell'inserimento dei dati, quindi interrompe la propria esecuzione e lascia l'utente di fronte alla vera procedura d'accesso. Questo tipo d'attacco può essere respinto dal sistema operativo mostrando un messaggio informativo alla fine di ogni sessione di lavoro interattiva o attraverso una sequenza di caratteri non "catturabile" inviata dalla tastiera, come la sequenza **Ctrl-Alt-Canc** impiegata dai moderni sistemi operativi Windows.

Un'altra variante del cavallo di Troia è il cosiddetto **spyware**. Lo spyware talvolta accompagna un programma che l'utente ha scelto di installare. Nel caso più frequente, è annesso ai programmi per uso gratuito (*freeware*) o privi di licenza commerciale (*shareware*), ma può essere incluso anche in quelli commerciali. La finalità di un programma spyware è di visualizzare annunci pubblicitari sullo schermo dell'utente, creare finestre a comparsa nel browser quando si visitano alcuni siti, o prelevare informazioni dal sistema dell'utente per trasmetterle a un sito di raccolta. Quest'ultima modalità rientra nella categoria di attacchi noti in genere come **canali coperti** (*covert channel*), in cui si stabiliscono comunicazioni nascoste. Per esempio, un programma in apparenza inoffensivo, installabile su un sistema Windows, potrebbe portare al caricamento di un demone spyware. Lo spyware potrebbe contattare un sito centrale, ricevere da esso un messaggio con una lista di indirizzi dei destinatari, e recapitare spam a tali utenti dalla macchina Windows. Questo meccanismo si ripete fintanto che l'utente scopra lo spyware: spesso, però, non si giunge neppure a scoprirllo. Nel 2010 si calcola che il 90 per cento dello spam sia stato inviato in questa maniera. Nella maggior parte dei paesi questa appropriazione del servizio non è neanche considerata reato!

Lo spyware è un caso particolare di un problema più esteso: la violazione del principio del privilegio minimo. L'utente di un sistema operativo, salvo rare eccezioni, non ha bisogno di installare un demone di rete. Demoni simili sono installati in seguito a due errori. In primo luogo, all'utente possono essere attribuiti privilegi in eccesso (per esempio, i privilegi che spettano all'amministratore), di modo che i programmi da lui eseguiti abbiano maggiore accesso al sistema di quanto sia necessario. Questo è un caso di errore umano – una debolezza comune per i sistemi di sicurezza. Oppure un sistema operativo può concedere agli utenti, per default, maggiori privilegi di quelli necessari. Questo caso mette in luce l'inadeguatezza delle scelte compiute in sede di progettazione del sistema. Un sistema operativo (e in generale il software) dovrebbe prevedere, per la sicurezza e il controllo dell'accesso, un elevato livello di granularità; il sistema, però, deve anche risultare semplice da comprendere e da gestire. Misure di sicurezza scomode o inadeguate sono destinate a essere eluse, determinando, nel complesso, l'affievolirsi di quella stessa sicurezza che avrebbero dovuto garantire.

15.2.2 Trabocchetti

Il progettista di un programma o di un sistema può lasciare nel programma un “buco” segreto che lui solo è in grado di utilizzare. Questo tipo di violazione della sicurezza, detto **trabocchetto** (*trap door*), è stato mostrato nel film *War Games*. Il codice può per esempio cercare uno specifico userID, o una determinata password, e può quindi aggirare le normali procedure di sicurezza. Alcuni programmatori sono stati arrestati per aver truffato banche inserendo errori d’arrotondamento nel loro codice per ottenere l’accredito nei propri conti dei risultanti mezzi centesimi di dollaro. Considerato il numero di transazioni eseguite da una grande banca, con tali accrediti si possono raggiungere delle cospicue quantità di denaro.

Un abile trabocchetto si potrebbe inserire in un compilatore, che in questo caso potrebbe generare sia un codice oggetto normale sia un codice oggetto contenente un trabocchetto, a prescindere dal codice sorgente da compilare. Si tratta di un’attività particolarmente nefasta, poiché un’ispezione del codice sorgente del programma non rivelerebbe alcun problema. L’informazione è contenuta solo nel codice sorgente del compilatore.

I trabocchetti costituiscono un problema difficile: per individuarli è necessario analizzare tutto il codice sorgente dei componenti di un sistema. Poiché i sistemi possono essere composti da milioni di righe di codice, queste analisi non si fanno spesso, e spesso non si fanno mai.

15.2.3 Bomba logica

Un programma che minacci la sicurezza solo al verificarsi di precise condizioni è difficile da neutralizzare, perché durante le normali operazioni non darebbe alcun problema di sicurezza. Tuttavia, quando dovesse verificarsi un certo insieme di condizioni stabilite a priori, il programma violerebbe la sicurezza. Questa situazione è nota come **bomba logica** (*logic bomb*). Per esempio un programmatore potrebbe scrivere alcune righe di codice che appurano se egli è ancora impiegato dall’azienda per cui lavora. In caso di riscontro negativo, il programma installa un demone per consentire l’accesso remoto o creare ed eseguire nuovi programmi intesi a danneggiare l’azienda.

15.2.4 Stack e buffer overflow

L’attacco basato sulla generazione di overflow dello stack o di un buffer è il modo più diffuso con cui un utente esterno a un sistema può ottenere un accesso non autorizzato, attraverso una rete. Un utente autorizzato potrebbe servirsi dello stesso metodo per ottenere privilegi d’accesso che in realtà non gli spettano (*privilege escalation*).

Sostanzialmente questi attacchi sfruttano un errore in un programma. L’errore può essere dovuto semplicemente alla bassa qualità della programmazione, per esempio alla mancanza di controllo che si rispettino i limiti di dimensioni di un input. In questo caso l’aggressore invia più dati di quelli che il programma si aspetta. Con una serie di tentativi, oppure esaminando il codice sorgente del programma da attaccare, se

questo è disponibile, l'aggressore determina i punti vulnerabili e scrive un programma che permette di compiere le seguenti azioni:

1. superare (*overflow*) i limiti di un campo di input, o di un argomento della riga di comando, o di un buffer che riceve dati in ingresso – per esempio in un demone di rete – fino a scrivere sullo stack;
2. sovrascrivere il corrente indirizzo di rientro annotato nello stack con l'indirizzo del segmento di codice che compie l'attacco;
3. scrivere il semplice segmento di codice, per l'area successiva nello stack, che include i comandi che l'aggressore desidera eseguire, per esempio l'attivazione di un interprete di comandi.

Il risultato dell'esecuzione di questo programma d'attacco sarà la disponibilità di un'interprete dei comandi con privilegi root o l'esecuzione di un altro comando privilegiato.

Per esempio, se un modulo in una pagina web prevede che il nome utente sia inserito in un apposito campo, l'aggressore potrebbe immettere il nome utente, un insieme di caratteri aggiuntivi con l'obiettivo di superare uno dei limiti del buffer che li riceve e raggiungere lo stack, un nuovo indirizzo di rientro da caricare nello stack, e il codice che l'aggressore vuole eseguire. Quando la funzione per la lettura dei dati contenuti nel buffer termina la sua esecuzione, l'indirizzo di rientro è quello del codice dell'aggressore, che quindi viene eseguito.

Analizziamo più da vicino un tentativo di sfruttare il buffer overflow. Si consideri il semplice programma in C riportato nella Figura 15.2. Esso crea un array di caratteri, la cui grandezza è pari a `BUFFER_SIZE`, e copia i contenuti del parametro passato dalla riga di comando `argv[1]`. Finché la grandezza di questo parametro resta inferiore a `BUFFER_SIZE` (ci serve un byte per memorizzare il carattere di terminazione

```
#include <stdio.h>
#define BUFFER_SIZE 256

int main(int argc, char *argv[])
{
    char buffer[BUFFER_SIZE];

    if (argc < 2)
        return -1;
    else {
        strcpy(buffer, argv[1]);
        return 0;
    }
}
```

Figura 15.2 Programma C che esemplifica il buffer overflow.

della stringa), il programma funziona correttamente. Vediamo tuttavia che cosa succede se il parametro è più lungo di `BUFFER_SIZE`. In questa eventualità, la funzione `strcpy()` inizierà a copiare da `argv[1]`, finché incontra il carattere di terminazione (\0) o il programma ha un crash. Questo programma, dunque, soffre di un potenziale problema di trabocco del buffer, in cui i dati copiati eccedono la capienza dell'array `buffer`.

Si noti che un programmatore attento avrebbe incluso nel codice controlli sulla dimensione di `argv[1]` usando la funzione `strncpy()` invece di `strcpy()`: la riga `strcpy(buffer, argv[1])` sarebbe allora diventata `strncpy(buffer, argv[1], sizeof(buffer)-1)`. Purtroppo, però, le buone pratiche di controllo sugli indici degli array sono l'eccezione, non la regola.

L'omissione di tali controlli, inoltre, non è l'unica causa di comportamenti analoghi a quello del programma illustrato dalla figura. È infatti possibile scrivere programmi accuratamente progettati per compromettere l'integrità del sistema sfruttando il trabocco dei buffer, come ora vedremo.

All'invocazione di una funzione, la gran parte dei sistemi operativi memorizza sullo stack le variabili locali della funzione (dette a volte **variabili automatiche**), i parametri passati alla funzione, e l'indirizzo da cui riprendere l'esecuzione al termine della funzione. La struttura di una tipica stack frame è mostrata nella Figura 15.3: dal basso verso l'alto, si possono osservare i parametri passati alla funzione, seguiti dalle variabili automatiche dichiarate dalla funzione (se presenti). L'elemento successivo è il **puntatore al frame**, ossia l'indirizzo del punto d'inizio dello stack frame. Infine, è presente l'indirizzo di rientro, che specifica dove restituire il controllo al termine della funzione. Il puntatore al frame deve essere salvato sullo stack, perché il valore del puntatore allo stack può variare durante l'esecuzione: il puntatore al frame permette l'accesso ai parametri e alle variabili automatiche tramite riferimenti relativi.

Sulla base di questo schema standard, i pirati informatici possono metter in atto un attacco tramite buffer overflow. Lo scopo è di sostituire l'indirizzo di rientro nello

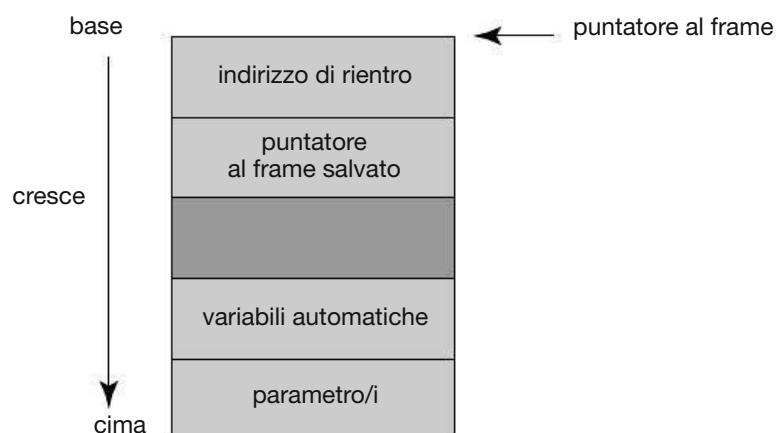


Figura 15.3 Forma di un tipico stack frame.

stack in modo da farlo puntare al segmento di codice che contiene il programma intruso.

Per prima cosa, il programmatore scrive un breve segmento di codice, come:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    execvp("\bin\sh","\bin \sh", NULL);
    return 0;
}
```

Tramite la chiamata di sistema `execvp()`, il programma lancia un processo shell (cioè, una console). Se il programma sotto attacco esegue con permessi d'accesso a tutto il sistema, la nuova shell avrà accesso a tutto il sistema; ovviamente, il segmento di codice potrà fare tutto ciò che gli è permesso dai privilegi del programma sotto attacco. Il codice sorgente è quindi compilato, e il risultante codice assembly è modificato per ridurne al minimo la dimensione, perché esso deve poter risiedere all'interno dello stack frame. Si ottiene così un frammento di codice binario sul quale si baserà l'attacco.

Si consideri nuovamente il programma della Figura 15.2. Supponiamo che, al momento della chiamata al `main()`, lo stack sia come nella Figura 15.4(a). Usando un debugger, il programmatore trova l'indirizzo di `buffer[0]` nello stack: questo è l'indirizzo del codice che l'intruso intende eseguire. Di conseguenza, l'intruso aggiunge

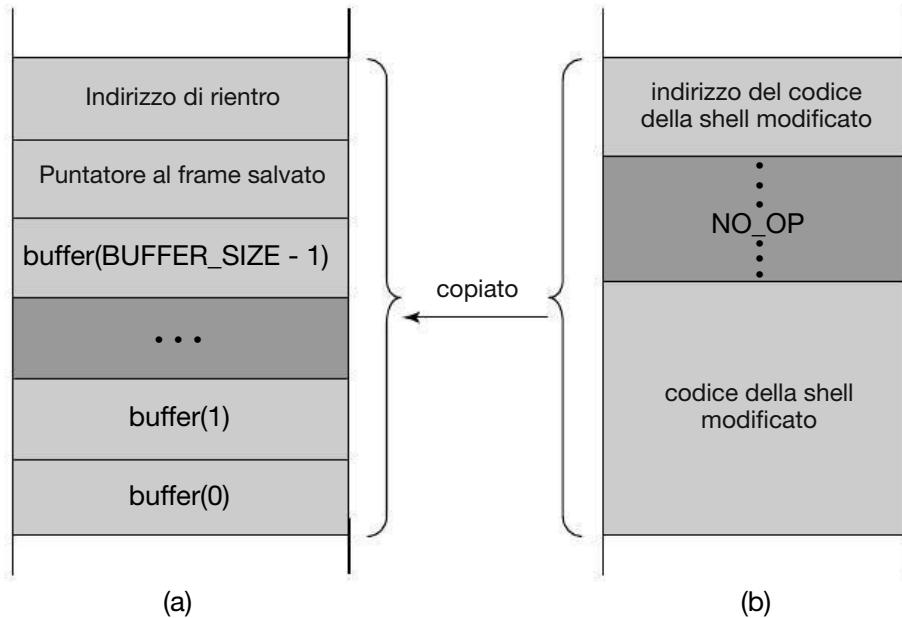


Figura 15.4 Ipotetico stack frame relativo al codice della Figura 15.2; (a) prima e (b) dopo.

al codice binario precedentemente costruito le istruzioni NO_OP (per NO-Operation, ossia nessuna operazione) necessarie per riempire lo stack fino alla locazione dell’indirizzo di rientro; infine, l’intruso aggiunge la locazione di `buffer[0]`, il nuovo indirizzo di rientro. L’attacco prende il via quando l’intruso fornisce il codice binario così costruito in ingresso al processo, che lo copia da `argv[1]` alla posizione `buffer[0]` dello stack. Quando però il `main()` termina, il controllo è passato alla shell modificata, e non al punto di rientro originale; la shell è dunque eseguita con i diritti d’accesso del processo attaccato! La Figura 15.4(b) mostra la struttura del codice della shell modificata usata per l’attacco.

Vi sono molti modi per sfruttare i potenziali problemi di buffer overflow. Nel nostro esempio abbiamo ipotizzato che il programma sotto attacco (Figura 15.2) godesse di diritto d’accesso a tutto il sistema. Ma anche in assenza di tale ipotesi il codice eseguito a causa della modifica dell’indirizzo di rientro può arrecare danni di ogni tipo, dalla cancellazione di file all’apertura di porte di rete per successivi usi dolosi, e così via.

L’esempio indica che sono richieste notevoli conoscenze e abilità di programmazione per riconoscere e sfruttare ai fini di un attacco le vulnerabilità del codice. Purtroppo, però, non bisogna essere grandi programmati per sferrare un attacco alla sicurezza: basta che un pirata informatico esperto scriva il codice da usare nell’attacco affinché poi chiunque abbia rudimentali nozioni informatiche – i cosiddetti *script kiddie* – possa sfruttarlo a tal fine.

Gli attacchi basati sul buffer overflow sono particolarmente pericolosi in quanto possono essere eseguiti da un sistema verso un altro, viaggiando attraverso i canali di comunicazione legali. Essi possono essere lanciati sfruttando i normali protocolli per la comunicazione fra due macchine, risultando difficili da individuare e prevenire. Questi attacchi possono persino scavalcare i firewall (Paragrafo 15.7).

Una soluzione al problema consiste nel far sì che la CPU proibisca l’esecuzione di codice all’interno dei segmenti della memoria adibiti a stack. Le versioni recenti del processore SPARC della Sun hanno questa proprietà, e le versioni recenti di Solaris la sfruttano. L’indirizzo di rientro della routine soggetta a trabocco può ancora essere modificato, ma se esso indica un punto dello stack e il codice ivi contenuto sta per essere eseguito, il sistema solleva un’eccezione che conduce alla terminazione (con segnalazione d’errore) del programma.

Per prevenire tali attacchi, le ultime versioni dei processori AMD e Intel x86 mettono a disposizione la funzionalità NX; essa è sfruttata da molti sistemi operativi x86, compresi Linux e Windows. L’implementazione hardware contempla l’uso di un bit aggiuntivo nella tabella delle pagine della CPU che indica se la pagina associata sia eseguibile o meno; nel caso non lo sia, le eventuali istruzioni che essa contiene non possono essere lette ed eseguite. Con il diffondersi di tali funzionalità si dovrebbe assistere a una notevole diminuzione degli attacchi basati sul buffer overflow.

15.2.5 Virus

Un'altra minaccia basata su programmi è costituita dai **virus**. Un virus è un frammento di codice inserito in un programma legittimo. I virus si autoriproducono e sono concepiti in modo da “contagiare” altri programmi; possono recare danni enormi a un sistema, modificandone o distruggendone i file, oltre a causare difetti di funzionamento nei programmi e crash del sistema. Come molti attacchi invasivi, i virus sfruttano le peculiarità di una singola architettura, sistema operativo o applicazione: per gli utenti dei PC rappresentano un problema temibile. UNIX e gli altri sistemi operativi multiutente, in genere, non sono soggetti ai virus, poiché i loro programmi eseguibili sono protetti dalla scrittura. Se anche un virus infetta tali programmi ha, in linea di massima, una capacità offensiva limitata, perché altri aspetti del sistema sono protetti.

Normalmente i virus si trasmettono per posta elettronica: la posta indesiderata (*spam*) è il veicolo più comune. Possono diffondersi, inoltre, attraverso lo scambio di dischi infetti tra utenti, o quando si scaricano programmi contaminati dai servizi di condivisione dei file su Internet.

Un altro veicolo di propagazione dei virus utilizza i file di Microsoft Office, per esempio i documenti creati con Microsoft Word. Questi documenti possono contenere delle cosiddette *macro* (cioè, programmi in Visual Basic) che vengono eseguite automaticamente dai programmi del pacchetto Office (Word, PowerPoint ed Excel). Dal momento che tali programmi sono eseguiti sotto l’account dell’utente, le macro possono agire quasi senza controllo (e possono cancellare in maniera indiscriminata i file dell’utente, per esempio). Di norma il virus si autotrasmette, con l’invio di messaggi di posta elettronica agli indirizzi contenuti nella rubrica dell’utente. Dall’esempio di codice seguente si può constatare come sia facile scrivere una macro in Visual Basic, con la quale un virus potrebbe formattare il disco rigido di un calcolatore Windows, non appena il file contenente la macro fosse aperto:

```
Sub AutoOpen()
    Dim oFS
    Set oFS = CreateObject("Scripting.FileSystemObject")
    vs = Shell("c:command.com /k format c:", vbHide)
End Sub
```

Come funzionano i virus? Dopo che un virus raggiunge la macchina presa di mira, un programma chiamato **portatore di virus** (*virus dropper*) inserisce il virus nel sistema. Il portatore di virus è solitamente un cavallo di Troia che, sebbene apparentemente eseguito per altre ragioni, ha per vero obiettivo l’installazione del virus. Una volta installato, il virus può fare una serie di cose. Esistono migliaia di virus, che tuttavia possono essere ricondotti ad alcune categorie principali. Molti di loro appartengono a più di una categoria.

- **File.** Un virus di questo tipo infetta il sistema aggiungendosi in coda a un file. Esso modifica le istruzioni iniziali del programma in modo da far eseguire il pro-

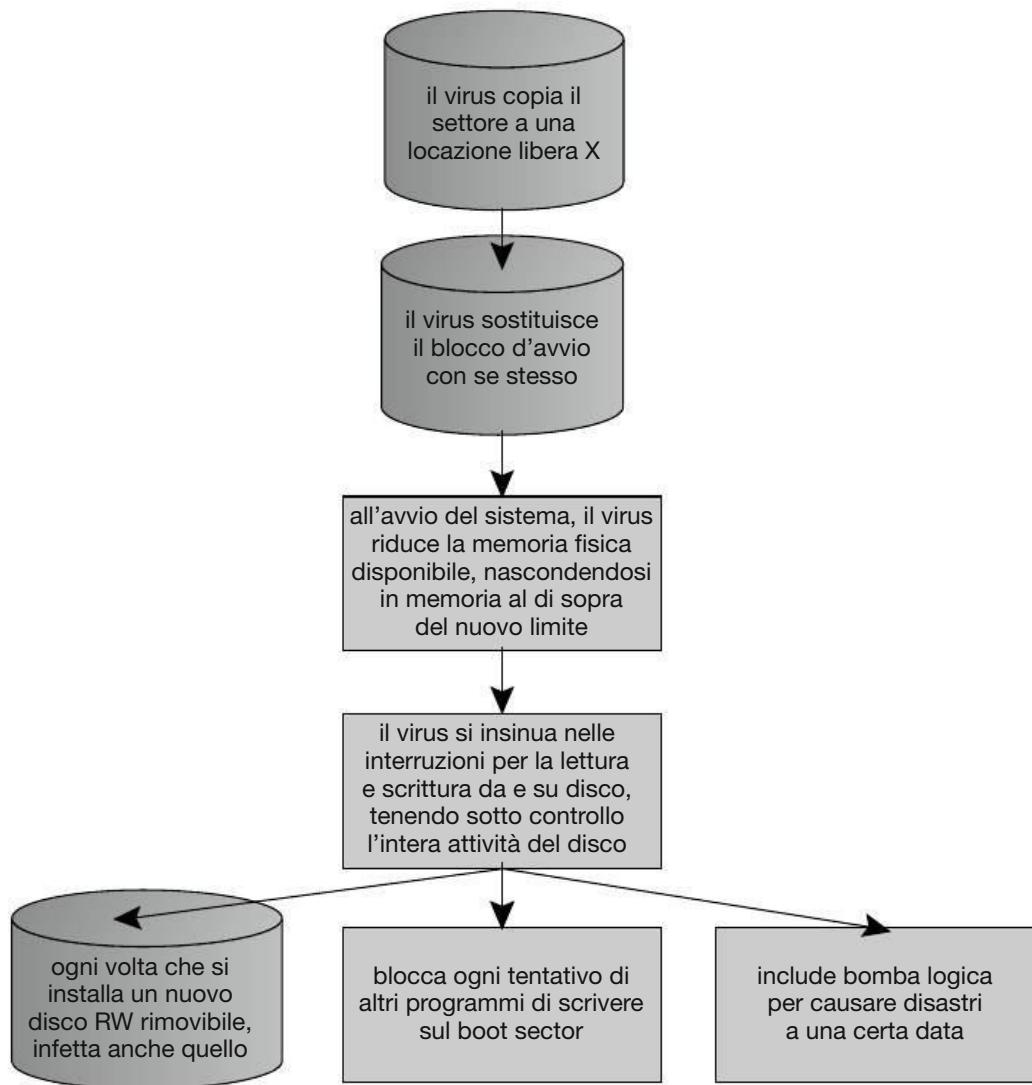


Figura 15.5 Virus del boot sector di un computer.

prio codice. Dopo essere stato eseguito restituisce il controllo al programma, in modo da non essere notato. I virus di file sono talora denominati virus parassiti, perché non si lasciano alle spalle alcun file completo, preservando la funzionalità del programma ospitante.

- **Avvio.** Un virus di avvio contagia la partizione d'avvio del sistema: va in esecuzione ogni volta che si avvia il sistema e prima che il sistema operativo sia caricato. Ricerca quindi altri dispositivi dotati di boot sector e li infetta. Questi virus sono noti anche come virus della memoria, dato che non compaiono nel file system. La Figura 15.5 illustra il funzionamento di un virus di avvio.
- **Macro.** La maggior parte dei virus è scritta in un linguaggio a basso livello, quale l'assembly o il C. I virus delle macro sono invece creati con un linguaggio ad alto livello, per esempio Visual Basic. Essi sono innescati all'avvio di un programma

in grado di eseguire le macro. Un virus delle macro, per esempio, potrebbe essere contenuto in un foglio di calcolo.

- **Codice sorgente.** Questo tipo di virus tenta di insediarsi all'interno di codice sorgente, e vi apporta modifiche che ne favoriscono la diffusione.
- **Polimorfico.** Onde evitare l'azione di rilevamento dei programmi antivirus, questo virus cambia aspetto ogni volta che viene installato. Le modifiche non alterano le sue caratteristiche, ma la sua firma. La **firma del virus** è una sequenza di informazioni da cui può essere identificato; generalmente si tratta di una serie di byte nel codice del virus.
- **Cifrato.** I virus cifrati comprendono il codice di decifrazione e il codice cifrato del virus vero e proprio; lo scopo è ancora quello di evitare la rilevazione. Le istruzioni contenute nel virus sono prima decifrate, quindi eseguite dal virus stesso.
- **Clandestino (stealth).** Questo virus ingannevole tenta di sottrarsi al rilevamento modificando le parti del sistema che potrebbero essere usate per individuarlo. Per esempio, potrebbe modificare la chiamata di sistema `read` in modo da ottenere, leggendo il file che ha alterato, il codice nella versione originale, anziché il codice infetto.
- **Tunnel.** Questo virus tenta di sfuggire alla sorveglianza dei programmi antivirus, installandosi nella catena di gestione delle interruzioni. Virus di natura analoga si installano nei driver dei dispositivi.
- **Composito.** Un virus di questo tipo è in grado di infettare numerose parti di un sistema, tra cui i settori di avviamento, la memoria e i file. Questa circostanza lo rende difficile da scoprire e arginare.
- **Corazzato.** Un virus corazzato ha un codice particolarmente ostico da capire per i ricercatori che devono scrivere un antivirus. Per evitare di essere rilevato e neutralizzato, può anche presentarsi in forma compressa. Inoltre, i portatori di virus e gli altri file coinvolti nell'attacco virale, sono spesso nascosti tramite l'impostazione degli attributi dei file o la scelta di nomi di file non visibili.

Tale ampia varietà di virus sembra destinata a crescere. Per esempio nel 2004 fu scoperto un virus nuovo e assai esteso, il cui modo di operare sfruttava tre bachi differenti. Questo virus inizialmente infestò centinaia di server (tra cui molti siti ritenuti sicuri), in cui era installato il Microsoft Internet Information Server (IIS). Ogni browser Microsoft Explorer vulnerabile era a rischio: non appena uno di loro visitava tali siti scaricava un virus per il browser. Il virus del browser installava poi numerosi programmi back-door, tra i quali un **keystroke logger**, in grado di registrare tutto ciò che è digitato sulla tastiera (inclusi numeri delle carte di credito e password). Esso, inoltre, installava un primo demone, per consentire agli intrusi l'accesso remoto senza limiti, e un secondo demone, grazie al quale gli intrusi potevano smistare messaggi abusivi di posta elettronica, inviandoli dal calcolatore infetto.

Generalmente i virus sono la tipologia più nociva di attacchi alla sicurezza e, data la loro efficacia, continueranno a essere inventati e diffusi. Una delle tematiche rela-

tive alla sicurezza dibattute nella comunità informatica riguarda l'esistenza di una **monocultura**, in cui moltissime macchine adottano lo stesso hardware e medesimi sistemi operativi e programmi applicativi. Questa monocultura sarebbe costituita dai prodotti Microsoft. Il primo interrogativo che questo dibattito pone è se ancora oggi esista una monocultura; un'altra questione è se essa favorisce o meno il diffondersi delle minacce e dei danni causati dai virus e da altre violazioni di sicurezza.

15.3 Minacce relative al sistema e alla rete

Le minacce ai programmi sfruttano i malfunzionamenti nei meccanismi di protezione di un sistema per attaccare i programmi. Le minacce relative ai sistemi e alle reti, invece, riguardano l'abuso dei servizi e delle connessioni di rete. Queste minacce creano una situazione in cui le risorse del sistema operativo e i file degli utenti subiscono un uso scorretto. Può accadere che un attacco di questo tipo costituisca la via per sferzare un attacco ai programmi, e viceversa.

Più un sistema è aperto, ossia più servizi e funzioni rende disponibili, più è probabile che si possa trovare un baco da sfruttare. I sistemi operativi cercano sempre di più di essere **sicuri per default**: per esempio Solaris 10 è passato da un modello in cui molti servizi (fra cui FTP e telnet) erano automaticamente abilitati all'installazione del sistema, a un modello in cui quasi tutti i servizi sono inizialmente disabilitati e devono essere abilitati esplicitamente dall'amministratore di sistema. Questi cambiamenti riducono la **superficie di attacco**, ossia l'insieme dei modi in cui un attaccante può provare a violare il sistema.

Analizziamo nel corso di questo paragrafo alcuni esempi di minacce di sistema e di rete, tra cui i worm, la scansione delle porte e gli attacchi denial-of-service. È importante notare come gli attacchi mimetici e replay vengano normalmente lanciati tramite le reti di collegamento fra i sistemi. In realtà, quanti più sistemi sono collegati, tanto maggiore sarà l'efficacia di questi attacchi, e la loro capacità di eludere le strategie difensive. All'interno di un calcolatore, per esempio, il sistema operativo può identificare il mittente e il destinatario di un messaggio. Se pure il mittente cambiasse il proprio ID, potrebbe lasciare traccia di tale cambiamento di ID. Quando, invece, sono coinvolti diversi sistemi, in particolar modo sistemi controllati dagli incursori, il compito di rintracciare l'origine degli attacchi risulta molto più arduo.

In genere, la condivisione di dati segreti (per provare l'identità e come chiave crittografica) è necessaria per l'autenticazione e per la crittografia, e ciò risulta più agevole negli ambienti (quale un sistema operativo singolo) che dispongono di metodi sicuri per la condivisione. Tra questi metodi figurano la memoria condivisa e lo scambio di messaggi tra i processi. La comunicazione e l'autenticazione sicure sono tratte nei Paragrafi 15.4 e 15.5.

15.3.1 Worm

Un **worm** è un processo che utilizza un meccanismo di **proliferazione** (*spawn*) per generare continuamente copie di se stesso abusando delle risorse del sistema, talvolta fino a renderlo inutilizzabile da tutti gli altri processi. I worm diventano particolarmente efficaci nelle reti di computer, poiché hanno la possibilità di riprodursi in diversi sistemi collegati in rete e quindi di far crollare l'intera rete. Una situazione di questo tipo si è verificata nel 1988 tra sistemi UNIX connessi alla rete Internet, causando milioni di dollari di perdita in tempo di calcolo e in lavoro degli amministratori di sistema.

Al termine della giornata di lavoro del 2 novembre 1988, Robert Tappan Morris Jr., uno studente della Cornell University, inserì un programma worm in uno o più calcolatori connessi alla rete Internet. Questo programma aveva come obiettivo le stazioni di lavoro Sun 3 di Sun Microsystems e i calcolatori VAX che eseguivano varianti della versione 4 del BSD UNIX. Il programma si propagò rapidamente per grandi distanze, ed entro poche ore dalla sua immissione consumò le risorse dei sistemi infetti fino a causarne il crollo.

Sebbene Robert Morris avesse progettato il programma affinché si riproducesse e distribuisse rapidamente, furono alcune caratteristiche dell'ambiente di rete di UNIX che fornirono al programma i mezzi per propagarsi per tutto il sistema. Probabilmente Morris scelse per l'attivazione iniziale un calcolatore connesso alla rete Internet accessibile dagli utenti esterni. Da qui il worm sfruttò le lacune nelle procedure di sicurezza di UNIX e sfruttò le funzioni che semplificano la condivisione delle risorse in una rete locale per ottenere accessi non autorizzati a migliaia di altri siti connessi. Il metodo di attacco del Morris worm è descritto nel seguito.

Il worm era composto da due programmi, un **programma arpione** (*grappling hook*, detto anche **avviamento** o **vettore**) e un programma principale. L'arpione, di nome `11.c`, era costituito da 99 righe di codice in linguaggio C che venivano compilate ed eseguite in ogni calcolatore raggiunto. Una volta stabilitosi nel calcolatore sotto attacco, il programma arpione si connetteva al calcolatore nel quale era stato generato e caricava una copia del programma principale nel sistema *agganciato* (Figura 15.6). Il programma principale iniziava quindi la ricerca di altri calcolatori cui il sistema appena infettato poteva connettersi con facilità. Per questa operazione, Morris sfruttò il comando `rsh`; si tratta di un comando di rete del sistema operativo UNIX che permette di lanciare facilmente le esecuzioni remote. Mediante l'impostazione di file speciali contenenti una lista di coppie di nomi *<calcolatore, nome utente>* gli utenti possono omettere l'inserimento della password a ogni accesso ai calcolatori remoti presenti nell'elenco di coppie. Il worm analizzava questi file speciali alla ricerca dei nomi dei siti che avrebbero consentito l'esecuzione remota senza chiedere l'immissione della password, attivava in questi sistemi una shell remota, quindi vi caricava il programma principale e ricominciava l'esecuzione.

L'attacco per mezzo dell'accesso remoto era solo uno dei tre metodi d'infezione impiegati. Gli altri due sfruttavano bachi del sistema operativo presenti nei programmi `finger` e `sendmail` del sistema operativo UNIX.

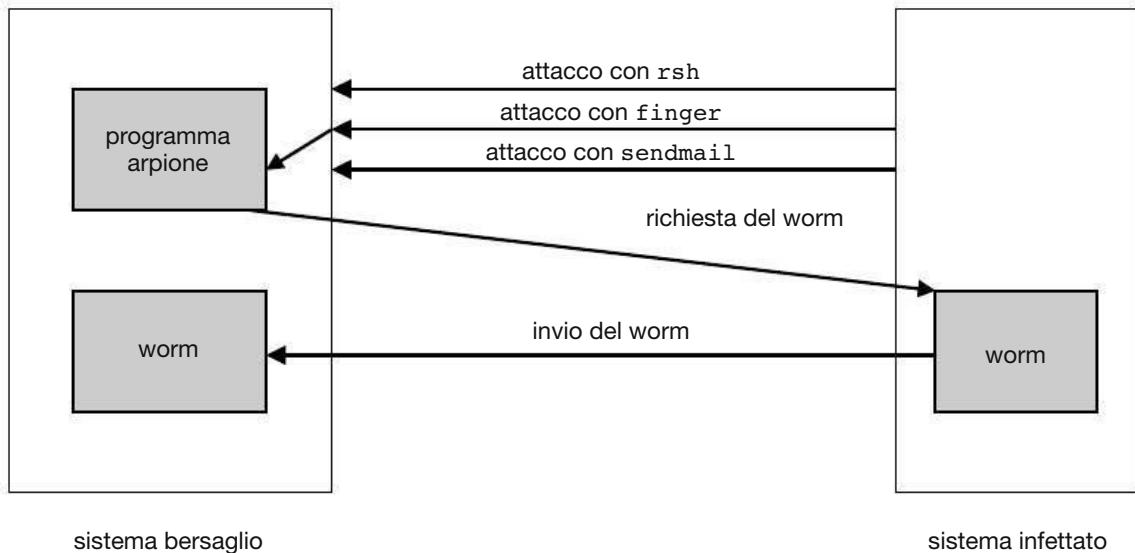


Figura 15.6 L'*Internet worm* di Robert Morris.

Il programma `finger` funziona come una guida telefonica elettronica; il comando

`finger nome-utente@nome_del_calcolatore`

riporta il nome reale e per l'accesso al sistema di una persona, e altre informazioni eventualmente specificate dall'utente, come gli indirizzi e i numeri telefonici di casa e ufficio, l'ambito di lavoro, o interessanti citazioni. Il comando `finger` viene eseguito come processo in background (o demone) in ciascun sito BSD e risponde alle richieste inviategli da qualsiasi sito della rete Internet. Il worm effettuava un attacco a `finger` basato sul buffer overflow, presentandogli una stringa di 536 byte costruita per superare la dimensione del buffer allocato per l'input, e per poi sovrascrivere lo stack. Anziché restituire il controllo alla funzione `main` in cui il programma si trovava prima della chiamata di Morris, il controllo del demone `finger` veniva deviato a una procedura contenuta nella sequenza di 536 byte ora residente nello stack. Questa procedura eseguiva il comando `/bin/sh`, che, nel caso di successo, dava al worm un interprete dei comandi remoto nella macchina presa di mira.

Anche il baco sfruttato in `sendmail` comportava lo sfruttamento di un demone per un accesso doloso. Il programma `sendmail` invia, riceve e instrada i messaggi di posta elettronica. Il codice di debugging contenuto nel programma consente di controllare e visualizzare lo stato del sistema di gestione della posta. Questa funzione di debugging è utile agli amministratori di un sistema ed era spesso lasciata in attività come processo in background. Morris incluse nel proprio arsenale di attacco una chiamata al comando `debug`, la quale, invece di specificare un indirizzo utente come nei normali casi di verifica, inviava un insieme di comandi che spedivano ed eseguivano una copia del programma arpione.

Una volta giunto sul posto, il programma principale del worm intraprendeva una serie di tentativi sistematici per scoprire le password degli utenti. Inizialmente cercava

utenti privi di password o con password costituite da combinazioni tra i nomi di account e di utente, quindi provava con le parole contenute in un dizionario interno di 432 parole comunemente usate come password, e infine provava come possibile parola d'ordine ogni parola del dizionario in linea di UNIX. Questo raffinato quanto efficiente algoritmo di ricerca in tre fasi delle password consentiva al worm di accedere a ulteriori account sul sistema infettato. Poi il worm cercava in ogni utenza violata nuovi file con i dati di `rsh` e li utilizzava come si è descritto precedentemente per ottenere l'accesso a utenze in sistemi remoti.

A ogni nuovo accesso il worm cercava copie di se stesso già attive. Se ne trovava una, la nuova copia terminava la propria esecuzione, a esclusione di ogni settima istanza. Se fosse stato progettato in modo da terminare ogniqualvolta si propagava su una macchina già infettata, probabilmente non sarebbe mai stato scoperto. Consentendo a ogni settima copia di proseguire (probabilmente per disorientare i tentativi di arrestarne la diffusione attraverso l'adescamento con worm *fasulli*) portò alla totale infestazione di sistemi Sun e VAX sulla rete Internet.

Le stesse caratteristiche dell'ambiente di rete del sistema operativo UNIX che favorirono la propagazione del worm contribuirono anche al suo contenimento. La semplicità nelle comunicazioni elettroniche, i meccanismi per la copiatura di file sorgenti e binari in calcolatori remoti e la possibilità di accedere sia al codice sorgente sia all'esperienza degli altri amministratori consentirono, grazie a sforzi congiunti, il rapido sviluppo di una soluzione. Già dalla sera del giorno successivo, il 3 novembre, cominciarono a circolare via Internet, tra gli amministratori dei vari sistemi, metodi per arrestare il programma invasore. Nel giro di pochi giorni erano disponibili specifiche patch per la correzione dei difetti nella sicurezza sfruttati dal worm.

Ci si può chiedere: perché Morris diffuse il suo *Internet worm*? L'azione è stata considerata sia come bravata, in seguito degenerata, sia come seria azione criminale. Vista la complessità del metodo d'attacco, è improbabile che l'immissione del worm o la dimensione della sua diffusione non fossero intenzionali. Il programma compiva operazioni elaborate per nascondere le proprie tracce e per respingere ogni tentativo di arrestarne la diffusione. D'altra parte il programma non conteneva codice rivolto al danneggiamento o alla distruzione dei sistemi in cui veniva eseguito. L'autore chiaramente possedeva l'esperienza necessaria a includere questo genere di comandi; in realtà nel codice d'avviamento erano contenute alcune strutture dati utilizzabili per trasferire un cavallo di Troia o un virus. Il comportamento del programma potrebbe condurre a interessanti osservazioni, anche se non fornisce fondati elementi per dedurne le motivazioni. Di definitivo resta comunque il risvolto legale dell'azione: una corte federale ha dichiarato Robert Morris colpevole, condannandolo a tre anni di libertà vigilata, 400 ore di servizio alla comunità e un'ammenda di 10.000 dollari; le sue spese legali superarono probabilmente i 100.000 dollari.

Gli esperti di sicurezza proseguono nell'elaborazione di metodi per diminuire o eliminare i worm. Un episodio tra i più recenti, tuttavia, rivela in modo inconfondibile la loro presenza su Internet, dimostrando inoltre che, di pari passo con lo sviluppo di Internet, aumenta anche il danno che worm "innocui" possono produrre. Il caso in

questione ebbe luogo nel mese di agosto 2003. La quinta versione del worm “Sobig”, noto più propriamente come “W32.Sobig.F@mm”, è stata diffusa da persone tuttora ignote. È stato il worm che, al momento, registrò la maggior rapidità di proliferazione: al culmine della sua attività, riuscì a infettare centinaia di migliaia di elaboratori, e uno su diciassette messaggi e-mail inviati tramite Internet; intasava caselle di posta elettronica, rallentava le reti, e la sua rimozione costò un numero enorme di ore di lavoro.

Sobig.F è stato diffuso, mediante invio a un newsgroup sulla pornografia, da parte di qualcuno che aveva creato un profilo con una carta di credito rubata: il worm era stato camuffato da fotografia. L’obiettivo prescelto era la famiglia di sistemi Microsoft Windows. Il virus sfruttava il proprio motore SMTP per trasmettersi a tutti gli indirizzi di posta elettronica trovati nel sistema contagiato. Disponeva di vari subject per rendere difficile da identificare il messaggio, tra i quali “Grazie!” “Le sue specifiche” e “Re: Approvato”. Utilizzava inoltre un indirizzo casuale come mittente, rendendo difficile risalire dal messaggio alla fonte infetta, cioè alla macchina da cui era partito. Sobig.F presentava un allegato, anch’esso sotto vari nomi, con l’intento di farlo aprire al destinatario del messaggio. Qualora ciò fosse avvenuto, un programma chiamato WINPPR32.EXE sarebbe stato memorizzato nella directory predefinita di Windows, insieme a un file di testo, e il registry di Windows sarebbe stato modificato.

Il codice contenuto nell’allegato, inoltre, era programmato per tentare di connettersi a uno tra venti server a rotazione, da cui scaricare ed eseguire un programma. Fortunatamente, i server erano stati disabilitati prima che il codice potesse essere scaricato. Non si conosce tuttora il contenuto del programma ospitato dai server. Qualora il codice fosse stato nocivo, avrebbe potuto causare danni incalcolabili a un gran numero di macchine.

15.3.2 Scansione delle porte

La scansione delle porte non è un attacco, ma piuttosto un mezzo impiegato dai pirati informatici per sondare le vulnerabilità di un sistema, in vista di un attacco. La scansione delle porte funziona in automatico, grazie a un dispositivo che tenta di stabilire una connessione TCP/IP con una porta specifica (o con una serie di porte). Ipotizziamo, per esempio, che vi sia una vulnerabilità nota (o un baco) in `sendmail`. Un pirata informatico potrebbe azionare un dispositivo di scansione delle porte per tentare di connettersi, per esempio, alla porta 25 di un certo sistema o di un gruppo di sistemi. Una volta stabilita la connessione, il pirata informatico (ovvero il dispositivo) potrebbe tentare di comunicare con il servizio installato su quella porta per accettare che sia effettivamente `sendmail` e, in tal caso, che sia proprio la versione bacata.

Immaginate ora un dispositivo capace di ricercare ogni baco presente in qualunque servizio di qualsiasi sistema operativo. Il dispositivo potrebbe tentare di connettersi a ciascuna porta di uno o più sistemi. Per ogni servizio che rispondesse, potrebbe tentare di sfruttarne tutti i bachi conosciuti. Spesso, il baco è il trabocco del buffer, che consente di creare una shell di comando privilegiata sul sistema. Una volta creata la

shell, naturalmente, un pirata informatico potrebbe installare cavalli di Troia, backdoors e così via.

Tale dispositivo non esiste, e tuttavia esistono strumenti che possono realizzare in parte le operazioni descritte. Per esempio, l'applicazione open-source nmap (si veda <http://www.insecure.org/nmap/>), permette di esplorare le reti e tracciare rapporti sulla sicurezza con grande versatilità. Una volta puntata su un obiettivo, essa rivela quali servizi sono in esecuzione, compresi i nomi delle applicazioni e le versioni, e può rivelare quale sistema operativo sia in uso. L'applicazione permette, inoltre, di riconoscere le tecniche di difesa adottate: per esempio, quali firewall siano posti a difesa del bersaglio. Non sfrutta alcun baco noto.

Poiché i tentativi di scansione delle porte possono essere scoperti (si veda il Paragrafo 15.6.3), sono spesso lanciati da **sistemi zombie**. Sono così definiti i sistemi indipendenti che, dopo essere stati compromessi all'insaputa dei proprietari, vengono utilizzati per scopi scorretti, tra cui gli attacchi denial-of-service e la diffusione di posta indesiderata. La presenza di uno zombie è di notevole intralcio all'azione di contrasto dei pirati informatici, perché rende difficile accettare la provenienza dell'attacco e la persona che lo ha messo in atto. Questa è una delle numerose ragioni per cui occorre proteggere anche i sistemi "irrilevanti", e non solo quelli che contengono informazioni o servizi "preziosi".

15.3.3 Attacchi denial-of-service

Come accennato in precedenza, gli attacchi DOS non mirano a ottenere informazioni o a sottrarre risorse, bensì a impedire l'uso corretto di un sistema o di una funzionalità. La maggior parte degli attacchi basati sul rifiuto del servizio è diretta a sistemi che l'aggressore non è riuscito a violare. Infatti, non di rado risulta più facile sferrare un attacco che alteri l'uso corretto di una macchina, anziché penetrarvi all'interno.

Gli attacchi denial-of-service hanno origine, solitamente, dalla rete. Appartengono a due categorie. Nel primo caso l'aggressore occupa un numero così alto di risorse di un servizio da bloccarne completamente la funzionalità. Un click su un sito web, per esempio, potrebbe comportare lo scaricamento di una applet Java, che potrebbe poi consumare tutto il tempo di CPU disponibile, o creare finestre all'infinito. Il secondo caso riguarda il sabotaggio di una rete che ospita un servizio. Numerosi attacchi denial-of-service sono andati a segno con queste modalità, ai danni di grossi siti web. Essi derivano dall'abuso di alcune funzioni fondamentali del protocollo TCP/IP. Se l'aggressore, per esempio, invia la parte del protocollo che afferma "voglio stabilire una connessione TCP", ma a questa non fa mai seguire il consueto messaggio: "la connessione è ora completa", ne potrebbero risultare varie sessioni TCP parzialmente avviate. Se ripetuto per molte sessioni, questo meccanismo può erodere le risorse di rete del sistema, impedendo l'instaurazione di una connessione TCP legittima. A causa di attacchi simili, che possono durare ore o giorni, la funzionalità colpita può rimanere parzialmente o totalmente indisponibile. Gli attacchi di questo genere sono bloccati, di solito, intervenendo a livello della rete, almeno finché i sistemi operativi non siano aggiornati e resi meno vulnerabili.

Generalmente è impossibile impedire gli attacchi denial-of-service, poiché essi sfruttano gli stessi meccanismi del funzionamento normale. Perfino più difficili da prevenire e controbattere sono gli **attacchi denial-of-service distribuiti** (*distributed denial-of-service*, DDOS). Questi attacchi sono scagliati verso un bersaglio comune simultaneamente da numerosi siti, che spesso sono siti zombie. Gli attacchi DDOS sono diventati più comuni e spesso sono associati a tentativi di ricatto: un sito viene attaccato, e l'attaccante richiede denaro per fermare l'attacco.

Talvolta l'attacco riesce persino a passare inosservato; può essere difficile stabilire se il rallentamento di un sistema sia dovuto al suo uso intensivo oppure a un attacco. Basti pensare a come una campagna pubblicitaria incisiva che aumenti il traffico verso un sito possa essere facilmente scambiata per un attacco DDOS.

Vi sono altri aspetti interessanti in merito agli attacchi DOS; tra le altre cose, è necessario, per i programmati e i gestori di sistemi, comprendere in profondità gli algoritmi e le tecnologie che utilizzano. Se un algoritmo di autenticazione, dopo vari tentativi falliti, impedisce l'accesso a un utente per un periodo di tempo, un aggressore potrebbe servirsi di questo meccanismo per bloccare i processi di autenticazione di tutti gli utenti. Analogamente, un firewall che blocchi automaticamente alcune tipologie di traffico può essere indotto a bloccare quel traffico quando invece dovrebbe lasciarlo transitare. Infine, le esercitazioni all'interno dei corsi universitari di informatica sono vere e proprie fucine di attacchi DOS involontari. Si pensi alle prime esercitazioni di programmazione in cui gli studenti imparano a creare processi figli o thread. Un baco tra i più comuni riguarda la proliferazione a oltranza dei processi figli: per la memoria libera del sistema e la CPU non c'è via di scampo.

15.4 Crittografia come strumento per la sicurezza

Le difese contro gli attacchi informatici sono varie, e spaziano da quelle metodologiche alle più tecnologiche. La crittografia è lo strumento più generale di cui progettisti di sistema e utenti possono disporre. In questo paragrafo discuteremo della crittografia e le sue applicazioni nella sicurezza dei calcolatori. Si noti che la nostra trattazione della crittografia è semplificata per scopi didattici. Evitate di usare direttamente gli schemi qui descritti in situazioni pratiche. Esistono ottime librerie per la crittografia e queste librerie, facilmente reperibili, costituiscono una buona base per il progetto di applicazioni.

In un calcolatore isolato il sistema operativo può determinare con sicurezza chi è il mittente e chi il destinatario di qualsiasi comunicazione tra processi, poiché esso controlla tutti i canali di comunicazione all'interno del calcolatore. In una rete di calcolatori la situazione è completamente diversa. Un calcolatore in una rete riceve bit dai *cavi*, senza aver alcun modo immediato e affidabile per determinare quale calcolatore o applicazione abbia inviato quei bit. In modo analogo, un calcolatore invia bit nella rete senza sapere con certezza chi effettivamente li riceverà. Inoltre, sia in fase di invio che in fase di ricezione, il sistema non ha modo di sapere se qualcuno sta ascoltando di nascosto la comunicazione.

Di solito, per risalire ai potenziali mittenti e destinatari dei messaggi si usano gli indirizzi di rete. I pacchetti di rete arrivano con un indirizzo di sorgente, come un indirizzo IP, e quando un calcolatore invia un messaggio specifica esplicitamente il destinatario nell'indirizzo di destinazione. Tuttavia, per le applicazioni per le quali la sicurezza è importante, non è possibile fidarsi del fatto che gli indirizzi di sorgente e destinazione di un pacchetto identifichino con certezza chi lo ha inviato o chi lo riceve. Un calcolatore in mano a un pirata informatico potrebbe inviare un messaggio con un indirizzo sorgente falsificato, e molti altri calcolatori, oltre a quello specificato nell'indirizzo di destinazione, potrebbero ricevere il pacchetto (cosa che di solito effettivamente avviene). Per esempio, tutti i router nel cammino tra sorgente e destinazione riceveranno il pacchetto. Dunque, come può il sistema operativo decidere se soddisfare una richiesta se non può considerare fidata l'identità del richiedente? E come può fornire la protezione per una richiesta o per dei dati se non è in grado di determinare con esattezza chi riceverà la risposta o il contenuto del messaggio che invia nella rete?

In generale, si considera impossibile costruire una rete di qualsiasi dimensione in cui gli indirizzi di sorgente e di destinazione dei pacchetti si possano considerare *fidati* in questo senso. Quindi, l'unica possibilità è cercare di eliminare in qualche modo la necessità di considerare come fidata la rete. Per questo è essenziale il ruolo della crittografia. Da un punto di vista astratto, la **crittografia** si usa per definire i potenziali mittenti e destinatari di un messaggio. La crittografia moderna si fonda su codici segreti, chiamati **chiavi**, che si distribuiscono selettivamente ai calcolatori di una rete e si usano per elaborare i messaggi. La crittografia permette al destinatario di un messaggio di verificare che il messaggio sia stato creato da un calcolatore che possiede una certa chiave. In modo analogo, un processo mittente può codificare il suo messaggio in modo tale che solo un calcolatore in possesso di una certa chiave possa decifrare il messaggio. Tuttavia, a differenza degli indirizzi di rete, le chiavi sono progettate in modo che sia computazionalmente impossibile derivarle a partire dai messaggi generati tramite esse e da qualsiasi altra informazione pubblica. Dunque, questo meccanismo costituisce un mezzo molto più fidato per definire i mittenti e i destinatari dei messaggi. Si noti che la crittografia è un ambito di studio a sé stante, caratterizzato da grandi e piccole complessità e sottigliezze. Ciò che esploriamo qui sono gli aspetti principali di quelle parti della crittografia che riguardano i sistemi operativi.

15.4.1 Cifratura

Poiché risolve un'ampia varietà di problemi di sicurezza delle comunicazioni, la **cifratura** (*encryption*) è frequentemente utilizzata in molti ambiti della moderna computazione. La cifratura è utilizzata per inviare messaggi sulla rete in maniera sicura e per proteggere da letture non autorizzate i dati contenuti in un database, in un file o in un intero disco. Un algoritmo di cifratura permette al mittente di un messaggio di imporre che solo un calcolatore che possiede una certa chiave possa leggere il messaggio. Un algoritmo di cifratura permette anche di assicurare che un dato possa

essere letto solo da chi l’ha scritto. La cifratura dei messaggi, come si sa, è una pratica antica; alcuni algoritmi di cifratura risalgono all’antichità. In questo paragrafo ci occuperemo degli algoritmi e dei principi fondamentali dell’era moderna.

Un algoritmo di cifratura comprende i seguenti componenti.

- Un insieme K di chiavi.
- Un insieme M di messaggi.
- Un insieme C di testi cifrati.
- Una funzione di cifratura $E : K \rightarrow (M \rightarrow C)$. Cioè, per ogni $k \in K$, E_k è una funzione che genera testi cifrati a partire dai messaggi. Sia E sia E_k devono essere funzioni calcolabili in modo efficiente per ogni k . In genere, E_k mappa in maniera randomizzata i messaggi in testi cifrati.
- Una funzione di decifratura $D : K \rightarrow (C \rightarrow M)$. Cioè, per ogni $k \in K$, D_k è una funzione che genera messaggi a partire dai testi cifrati. Sia D sia D_k devono essere funzioni calcolabili in modo efficiente per ogni k .

La proprietà fondamentale che un algoritmo di autenticazione deve possedere è la seguente: dato un testo cifrato $c \in C$, un calcolatore può calcolare m tale che $E_k(m) = c$ solo se è in possesso di k . Quindi, un calcolatore che ha k può decifrare i testi cifrati ottenendo i testi in chiaro corrispondenti. Tuttavia, un calcolatore che non possiede k non può decifrarli. Poiché i testi cifrati sono di solito visibili (per esempio, s’inviano in una rete), è essenziale che sia infattibile derivare k dai testi cifrati.

Ci sono due tipi principali di algoritmi di cifratura: simmetrica e asimmetrica. Nei paragrafi successivi ci occuperemo sia dell’una sia dell’altra.

15.4.1.1 Cifratura simmetrica

In un **algoritmo di cifratura simmetrica** la stessa chiave è usata per cifrare e decifrare; ciò significa che la segretezza di k deve essere protetta. La Figura 15.7 mostra un esempio di due utenti che comunicano in maniera sicura su un canale insicuro per mezzo di una cifratura simmetrica. Si noti che lo scambio della chiave può avvenire direttamente tra le due parti oppure per mezzo di una terza parte fidata (un’autorità di certificazione), come illustrato nel Paragrafo 15.4.1.4.

Negli ultimi decenni, l’algoritmo di cifratura simmetrica più usato negli Stati Uniti d’America per scopi civili è stato il **DES** (*data encryption standard*), adottato dal National Institute of Standards and Technology (NIST). Il DES funziona prendendo blocchi in chiaro di 64 bit e una chiave, della lunghezza di 56 bit, ed effettuando una serie di trasformazioni basate su sostituzioni e permutazioni. Poiché agisce su un blocco di bit per volta, l’algoritmo DES è un **codice a blocchi** e le sue trasformazioni sono quelle tipiche della cifratura a blocchi. Nella cifratura a blocchi, l’utilizzo di una sola chiave per cifrare una quantità eccessiva di dati presterebbe il fianco a eventuali attacchi.

Il DES è ormai considerato insicuro per molte applicazioni, perché le sue chiavi possono essere provate esaustivamente anche con risorse di calcolo modeste. (Si noti

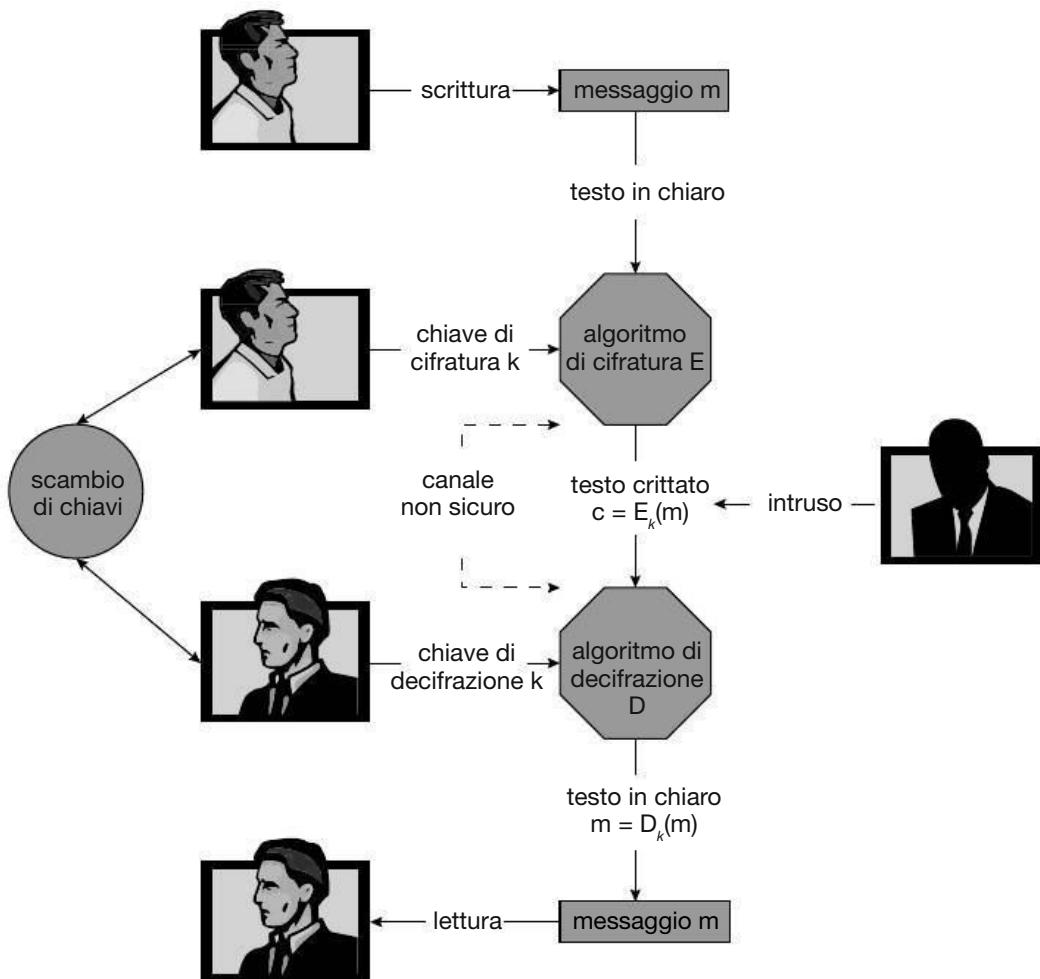


Figura 15.7 Comunicazione sicura su un canale non sicuro.

che, nonostante ciò, DES è ancora molto utilizzato). Anziché abbandonarlo, tuttavia, il NIST ha elaborato una variante del DES, chiamata **DES triplo**, in cui l'algoritmo originale è ripetuto per tre volte (due cifrature e una decifrazione) sul medesimo testo in chiaro, usando due o tre chiavi – per esempio, $c = E_{k_3}(D_{k_2}(E_{k_1}(m)))$. Quando si usano tre chiavi, la lunghezza effettiva della chiave è di 168 bit. Il DES triplo è attualmente molto usato.

Per sostituire il DES, nel 2001 il NIST ha adottato un nuovo algoritmo di cifratura, detto **AES** (*advanced encryption standard*, ossia “standard di cifratura avanzata”). L’AES è un altro algoritmo per la cifratura simmetrica a blocchi; utilizza chiavi con lunghezza pari a 128, 192 o 256 bit e opera su blocchi di 128 bit. In genere, è un algoritmo compatto ed efficiente.

La cifratura a blocchi non è di per sé uno schema di cifratura sicuro. In particolare, essa non gestisce direttamente messaggi che superano le dimensioni di blocco richieste. Vi sono tuttavia molti metodi di cifratura basati su codici stream che possono essere utilizzati per crittografare messaggi più lunghi in maniera sicura.

L’algoritmo **RC4** è, forse, il più comune codice stream. I **codici stream** sono metodi di cifratura volti a crittografare e decrittografare non già un blocco, ma una sequenza di byte o di bit; ciò è utile quando i messaggi in chiaro sono talmente lunghi da rendere la cifratura a blocchi troppo lenta. La chiave è immessa in un generatore di bit pseudo-casuali, ossia un algoritmo che tenta di produrre sequenze casuali di bit. Ciò che restituisce il generatore, quando riceve una chiave, è un insieme infinito di bit, detto **keystream**, che può essere utilizzato per la codifica di un testo semplicemente calcolando lo XOR (lo XOR, o OR esclusivo, è un’operazione che confronta due bit in ingresso e restituisce 0 se i bit sono uguali, 1 se sono diversi) tra il testo e il keystream. Lo RC4 è impiegato per crittografare flussi di dati, per esempio nel WEP, un protocollo per le reti LAN senza fili. Purtroppo, nella versione adottata da WEP, nello standard IEEE 802.11, l’algoritmo ha dato prova di poter essere violato in tempi contenuti. E in effetti, RC4 soffre di vulnerabilità intrinseche.

15.4.1.2 Cifratura asimmetrica

In un **algoritmo di cifratura asimmetrica** vi sono chiavi diverse per la cifratura e la decifrazione. Un’entità che desidera ricevere una comunicazione cifrata crea due chiavi e ne mette una (chiamata chiave pubblica) a disposizione di chiunque lo voglia. Ciascun mittente può utilizzare la chiave per cifrare la comunicazione, ma solo il creatore della chiave è in grado di decifrarla. Questo schema, noto come **crittografia a chiave pubblica**, ha costituito una svolta nella crittografia. Non è più necessario che la chiave venga tenuta segreta e consegnata in modo sicuro, ma chiunque può inviare un messaggio cifrato al soggetto ricevente e non importa chi altro è in ascolto, perché solo il ricevente può decifrare il messaggio.

Come esempio di crittografia a chiave pubblica descriviamo l’algoritmo noto come **RSA**, dal nome dei suoi inventori (Rivest, Shamir e Adleman). RSA è l’algoritmo asimmetrico più ampiamente usato. Negli ultimi tempi, tuttavia, registrano crescente diffusione gli algoritmi asimmetrici basati sulle curve ellittiche, che mantengono lo stesso livello di efficacia crittografica con chiavi più brevi.

In RSA, k_e è la **chiave pubblica** e k_d è la **chiave privata**. N è il prodotto di due numeri primi grandi, scelti casualmente, p e q (p e q per esempio misurano ognuno 512 bit). Deve essere computazionalmente impossibile ricavare $k_{d,N}$ da $k_{e,N}$, affinché k_e non debba essere tenuto segreto e possa essere divulgato. L’algoritmo di cifratura è $E_{k_e,N}(m) = m^{k_e} \text{ mod } N$, dove k_e soddisfa $k_e k_d \text{ mod } (p-1)(q-1) = 1$. L’algoritmo di decifrazione è allora $D_{k_d,N}(c) = c^{k_d} \text{ mod } N$.

La Figura 15.8 rappresenta un esempio che fa uso di valori piccoli. Si suppone $p = 7$ e $q = 13$, e si calcolano $N = 7*13 = 91$ e $(p-1)(q-1) = 72$. Si sceglie quindi un k_e strettamente minore di 72 tale che 72 e k_e siano relativamente primi, per esempio, $k_e = 5$. Si calcola poi k_d tale che valga $k_e k_d \text{ mod } 72 = 1$, ottenendo $k_d = 29$. Si ottiene così la chiave pubblica, $k_{e,N} = 5,91$, e la chiave privata, $k_{d,N} = 29,91$. La cifratura del messaggio 69 con la chiave pubblica genera il messaggio 62, che viene quindi decodificato dal destinatario grazie alla chiave privata.

L’uso della cifratura asimmetrica ha inizio con la pubblicazione della chiave pubblica del destinatario. Se la comunicazione è a due vie, anche il mittente deve pub-

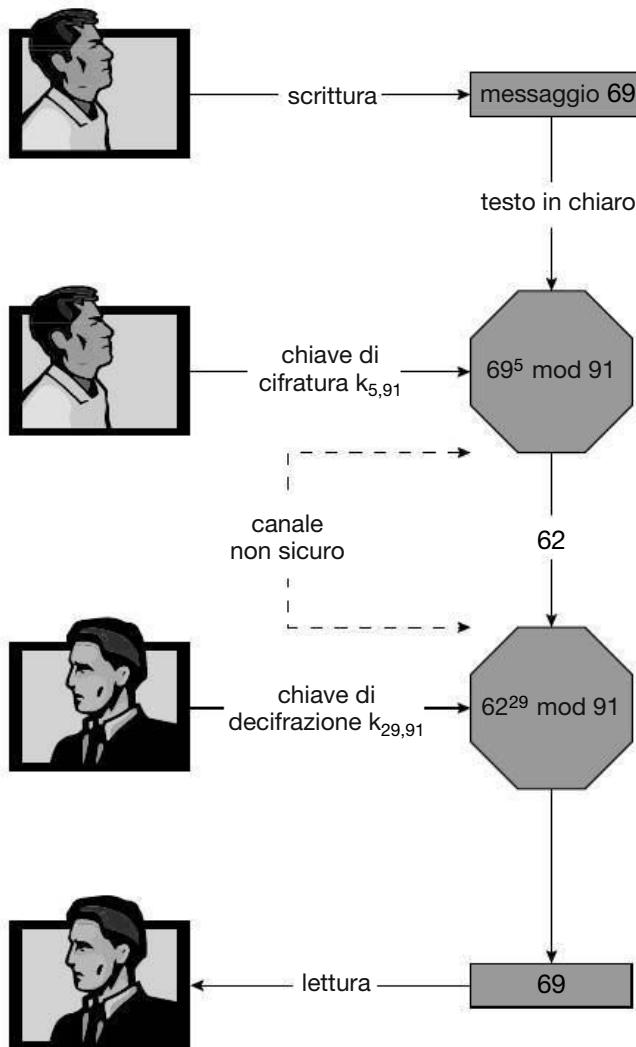


Figura 15.8 Cifratura e decifrazione per mezzo della crittografia asimmetrica RSA.

blicare la propria chiave pubblica. Per “pubblicazione” si può intendere la semplice trasmissione di una copia elettronica della chiave, oppure qualcosa di più complesso. La chiave privata (detta anche “segreta”) deve essere custodita con estrema cura, dato che chiunque ne entri in possesso può decrittografare ogni messaggio creato con la chiave pubblica corrispondente.

La differenza nell’uso delle chiavi, tra crittografia asimmetrica e simmetrica, a prima vista trascurabile, è in realtà piuttosto netta. La crittografia asimmetrica esige risorse computazionali di gran lunga superiori per la propria esecuzione. Un elaboratore può codificare e decodificare un testo molto più velocemente mediante gli usuali algoritmi simmetrici, che non usando un algoritmo asimmetrico. Perché, dunque, usare un algoritmo asimmetrico? In realtà, questi algoritmi non sono affatto applicati alla cifratura di grandi quantità di dati per scopi generali. Essi sono invece usati sia per la cifratura di piccole quantità d’informazione, sia per l’autenticazione, la sicurezza e la distribuzione delle chiavi, come vedremo nei paragrafi seguenti.

15.4.1.3 Autenticazione

Abbiamo visto che la cifratura permette di limitare l'insieme dei potenziali destinatari di un messaggio. Garantire l'insieme dei potenziali mittenti di un messaggio è un processo che si chiama anche **autenticazione**. L'autenticazione è quindi complementare alla cifratura. L'autenticazione è anche utile per attestare che un messaggio non sia stato modificato. In questo paragrafo illustriamo l'autenticazione come filtro dei possibili mittenti dei messaggi. Si tratta di una nozione simile, ma non identica, all'autenticazione degli utenti trattata nel Paragrafo 15.5.

Un algoritmo di autenticazione che utilizza chiavi simmetriche comprende i seguenti componenti.

- Un insieme K di chiavi.
- Un insieme M di messaggi.
- Un insieme A di autenticatori.
- Una funzione $S : K \rightarrow (M \rightarrow A)$. Cioè, per ogni $k \in K$, S_k è una funzione che genera autenticatori a partire da messaggi. Sia S sia S_k devono essere funzioni calcolabili in modo efficiente per ogni k .
- Una funzione $V : K \rightarrow (M \times A \rightarrow \{\text{true}, \text{false}\})$. Cioè, per ogni $k \in K$, V_k è una funzione che verifica gli autenticatori sui messaggi. Sia V che V_k devono essere funzioni calcolabili in modo efficiente per ogni k .

La proprietà fondamentale che un algoritmo di autenticazione deve possedere è la seguente: dato un messaggio m , un calcolatore può generare un autenticatore $a \in A$ tale che $V_k(m, a) = \text{true}$ soltanto se esso possiede k . Quindi, un calcolatore che possiede k può generare autenticatori su messaggi in modo che tutti gli altri calcolatori che possiedono k li possano verificare. Tuttavia, un calcolatore che non ha k non può generare autenticatori su messaggi che possono essere verificati tramite V_k . Poiché gli autenticatori sono generalmente visibili (per esempio, s'inviano nella rete con i messaggi stessi), k non si deve poter derivare a partire dagli autenticatori. In pratica, se $V_k(m, a) = \text{true}$, allora sappiamo che m non è stato modificato e che il mittente del messaggio possiede k . Se condividiamo k con una sola entità, allora sappiamo che il messaggio arriva da k .

Così come esistono due tipi di algoritmi di cifratura, vi sono due varietà principali di algoritmi di autenticazione. Il primo passo per capire tali algoritmi è la comprensione delle funzioni hash. La funzione hash $H(m)$ crea un piccolo blocco di dati di dimensioni prefissate, noto come **valore hash** o **digest del messaggio**, da un messaggio m . La maniera in cui procedono le funzioni hash è di dividere il messaggio in segmenti, ed elaborare i segmenti in modo da produrre un valore hash di n bit. La funzione hash H deve essere a prova di collisione: cioè, dato m , il calcolo di un $m' \neq m$ tale che $H(m) = H(m')$ deve risultare impraticabile. Da ciò segue che $H(m) = H(m')$ comporta che $m = m'$, ossia, che il messaggio non è stato modificato. Alcune delle funzioni hash più comuni sono **MD5**, oggi considerata insicura, che produce valori hash da 128 bit, e **SHA-1**, che fornisce valori da 160 bit. Le funzioni hash sono utili per rilevare modifiche dei messaggi, ma non bastano ad autenticarli: $H(m)$ può certo

essere spedito insieme al messaggio, ma se H è una funzione nota, chiunque può modificare il messaggio m in m' , computare $H(m')$, e produrre un messaggio alterato non rilevabile. È per questo che occorre autenticare $H(m)$.

Il primo tipo di algoritmo d'autenticazione sfrutta la cifratura simmetrica. Nei **codici per l'autenticazione dei messaggi** (*message-authentication code*, MAC), una somma di controllo crittografica è generata dal messaggio per mezzo della chiave segreta. Un codice MAC fornisce un modo per autenticare in modo sicuro piccoli valori. Se lo usiamo per autenticare $H(m)$, per un H a prova di collisione, allora si ottiene un modo per autenticare in modo sicuro messaggi lunghi utilizzando la funzione hash.

Il secondo tipo di algoritmo di autenticazione è un algoritmo basato sulla **firma digitale** e gli autenticatori prodotti in questo caso sono chiamati *firme digitali*. Le firme digitali sono molto utili, perché permettono a *chiunque* di verificare l'autenticità del messaggio. In un algoritmo basato sulla firma digitale è praticamente impossibile dal punto di vista computazionale derivare k_s da k_v . Per questa ragione k_v è la **chiave pubblica** e k_s è la **chiave privata**.

Si consideri per esempio l'algoritmo RSA per la firma digitale. È simile all'algoritmo crittografico RSA, ma l'uso delle chiavi è invertito. La firma digitale di un messaggio si ottiene calcolando $S_{k_s}(m) = H(m)^{k_s} \bmod N$. La chiave k_s è di nuovo una coppia (d, N) , dove N è il prodotto di due primi grandi p e q scelti a caso. L'algoritmo di verifica è allora $V_{k_v}(m, a) \stackrel{?}{=} (a^{k_v} \bmod N = H(m))$, dove k_v soddisfa $k_v k_s \bmod (p-1)(q-1) = 1$.

Notate che crittografia e autenticazione possono essere usate insieme o separatamente. A volte ci serve autenticazione ma non riservatezza. Per esempio, un'azienda potrebbe fornire una patch software e "firmare" quella patch per dimostrare che arriva da quell'azienda e che non è stata modificata.

L'autenticazione è parte integrante di molti aspetti della sicurezza. La firma digitale, per esempio, è uno strumento per realizzare il **non ripudio** (*nonrepudiation*), ossia per dimostrare che un soggetto abbia compiuto una certa azione. Un esempio tipico si ha quando il soggetto compila dei moduli elettronici invece di firmare contratti cartacei: il non ripudio impedisce che il soggetto neghi di aver compilato i moduli.

15.4.1.4 Distribuzione delle chiavi

Senza dubbio una buona parte della battaglia tra crittografi (coloro che inventano i codici) e i crittoanalisti (coloro che tentano di decifrarli) è incentrata sulla questione delle chiavi. Con gli algoritmi simmetrici, entrambi i soggetti coinvolti nella comunicazione devono possedere la chiave, ma essa non deve essere nota ad alcuno. La consegna delle chiavi simmetriche comporta una sfida impegnativa. Talvolta si ricorre a una comunicazione **non in linea** (*out-of-band*), tramite documenti cartacei o una conversazione. Questi metodi, tuttavia, non si prestano a essere generalizzati. Anche la questione di come gestire le chiavi va presa in considerazione. Si supponga che un utente desideri comunicare in privato con N altri utenti; questo utente dovrebbe avere N chiavi e, per maggior sicurezza, si vedrebbe costretto a cambiarle spesso.

L'analisi di tali situazioni è proprio ciò che ha dato impulso alla creazione degli algoritmi a chiave asimmetrica. Ogni utente non solo può così scambiarsi le chiavi

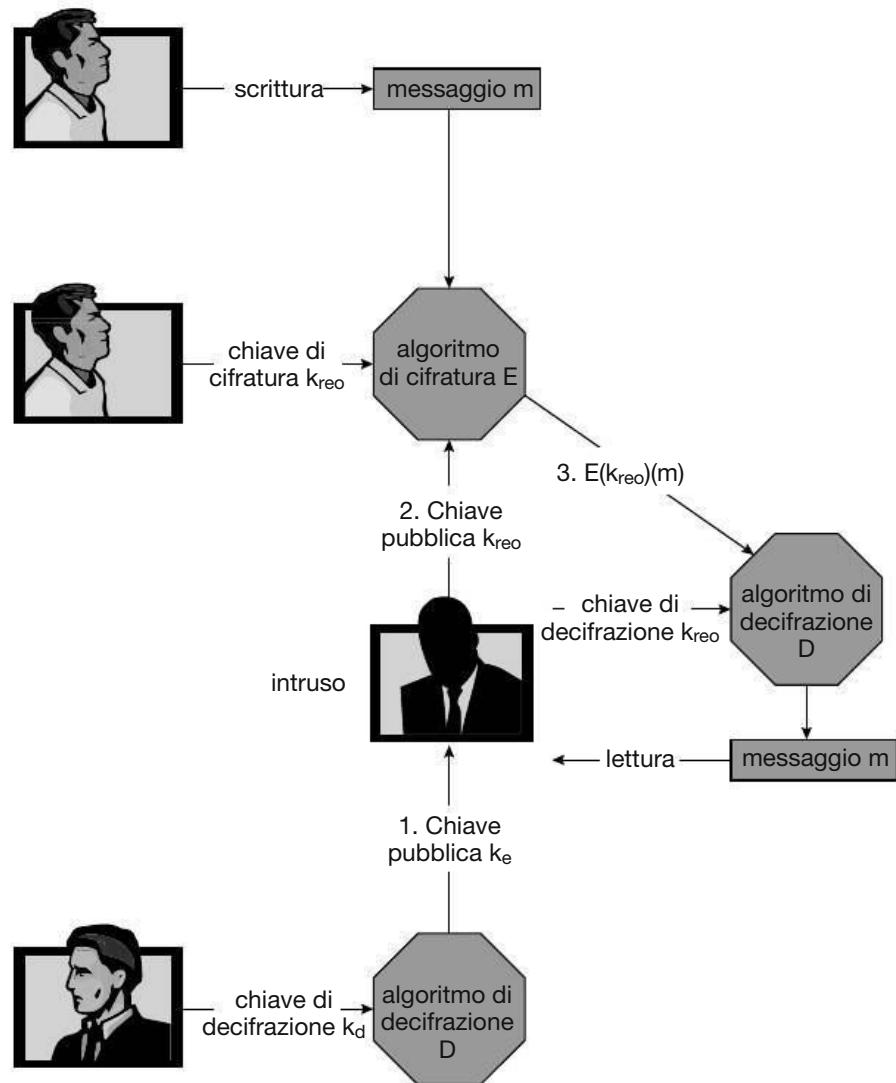


Figura 15.9 Attacco di interposizione alla cifratura asimmetrica.

in pubblico, ma ha bisogno di possedere una sola chiave privata, indipendentemente dal numero di persone coinvolte nella comunicazione. Rimane il problema della gestione di tante chiavi pubbliche quanti sono i destinatari da contattare; ma, poiché non è necessario mantenere segrete le chiavi pubbliche, si può adoperare la memorizzazione ordinaria per ciascun **mazzo di chiavi** (*key ring*).

Purtroppo, anche la distribuzione delle chiavi pubbliche richiede determinate accortezze. Si osservi l'attacco di interposizione illustrato nella Figura 15.9. In questo caso, la persona che intende ricevere un messaggio codificato distribuisce la sua chiave pubblica, ma l'intruso, a sua volta, trasmette la propria chiave pubblica illegittima (che fa coppia con la sua chiave privata). La persona che vuole inviare il messaggio in codice non ha il minimo sospetto sulla provenienza della chiave, dunque la utilizza per criptare il messaggio; l'intruso, a questo punto, può decriptografarlo tranquillamente.

Il problema esposto concerne l'autenticazione: è necessaria una prova su chi (o che cosa) detenga una chiave pubblica. Una soluzione applicabile prevede l'uso dei certificati digitali. Un **certificato digitale** è una chiave pubblica firmata digitalmente da un soggetto fidato. Questi riceve un attestato d'identità da una terza parte e certifica che la chiave pubblica appartiene alla parte. Ma da quali garanzie è sorretta la credibilità del soggetto certificatore? Le **autorità di certificazione** includono le proprie chiavi pubbliche nei browser web (e in altre applicazioni che utilizzano i certificati) prima della loro distribuzione. Le medesime autorità, quindi, possono attestare che altre autorità sono affidabili (firmando digitalmente le chiavi pubbliche di queste ultime); a loro volta, le autorità certificate proseguono in questo processo, creando una rete di fiducia. I certificati possono essere distribuiti nel formato digitale standard X.509, che il calcolatore è in grado di analizzare. Questo schema è usato per la comunicazione sicura sul Web, come si vedrà nel Paragrafo 15.4.3.

15.4.2 Implementazione della crittografia

I protocolli di rete sono di solito organizzati in **strati**, come una cipolla, in cui ciascuno strato agisce come un client rispetto allo strato sottostante. Cioè, quando un protocollo genera un messaggio per il protocollo di pari livello nel calcolatore di destinazione, consegna il suo messaggio al protocollo sottostante nello stack dei protocolli di rete in modo che questo lo consegni al rispettivo protocollo in quella macchina. Per esempio, in una rete IP, il TCP (un protocollo dello *strato di trasporto*) agisce come un client dell'IP (un protocollo dello *strato di rete*): i pacchetti del TCP sono passati all'IP, lo strato sottostante, affinché siano consegnati all'entità IP all'altro estremo della connessione. IP incapsula il pacchetto TCP in un pacchetto IP che, in modo analogo, li passa allo *strato di collegamento dei dati* sottostante affinché siano trasmessi attraverso la rete al corrispondente livello IP nel calcolatore di destinazione. Questo livello IP a sua volta consegnerà i pacchetti TCP a livello TCP su quella macchina.

La crittografia si può includere quasi in ogni livello del modello OSI. Il protocollo SSL (Paragrafo 15.4.3) per esempio fornisce sicurezza a livello di trasporto. La sicurezza a livello di rete, definita nello standard **IPSec**, specifica formati dei pacchetti IP che permettono l'inserimento di autenticatori e la cifratura del contenuto dei pacchetti. IPSec utilizza la cifratura simmetrica e sfrutta il protocollo Internet Key Exchange (IKE) per lo scambio di chiavi. IKE si basa sulla crittografia a chiave pubblica. L'IPSec si sta diffondendo come base per la realizzazione di **reti private virtuali** (*virtual private network, VPN*), nelle quali tutto il traffico fra due punti terminali IPsec è codificato in modo da rendere privata una rete che sarebbe altrimenti pubblicamente accessibile. Sono stati anche sviluppati numerosi protocolli utilizzabili dalle applicazioni (per esempio PGP, per la cifratura di email), ma poi le stesse applicazioni devono realizzare il codice per implementare la sicurezza.

Ci si può chiedere quale sia la miglior collocazione della protezione crittografica all'interno dello stack dei protocolli. Non vi è una risposta univoca a questa domanda. Da una parte, se si colloca la protezione negli strati bassi dello stack, può beneficiarne un maggior numero di protocolli. Per esempio, poiché i pacchetti IP incapsulano i

pacchetti TCP, la cifratura dei pacchetti IP (per esempio con l’IPSec) nasconde anche il contenuto del pacchetto TCP incapsulato. Analogamente, gli autenticatori su pacchetti IP rilevano le modifiche delle informazioni nelle intestazioni TCP contenute.

D’altra parte, la protezione collocata negli strati bassi dello stack dei protocolli potrebbe non essere sufficiente ai protocolli dei livelli più alti. Per esempio, un server applicativo che accetta connessioni cifrate con IPSec potrebbe autenticare i calcolatori client dai quali riceve le richieste. Tuttavia, per autenticare un utente in uno specifico calcolatore, si deve poter usare un protocollo del livello applicativo (che include, per esempio, l’inserimento della password da parte dell’utente). Si consideri inoltre il problema della posta elettronica: prima che i messaggi spediti tramite il protocollo standard SMTP siano recapitati al destinatario, sono memorizzati e inoltrati da macchine intermedie, spesso più volte. Ognuno di questi passaggi può coinvolgere reti sicure o non sicure. Affinché la posta elettronica sia resa sicura, quindi, i messaggi devono essere crittografati in modo da renderli sicuri indipendentemente dal mezzo che li trasporta.

15.4.3 Un esempio: SSL

L’SSL 3.0 è un protocollo crittografico che permette a due calcolatori di comunicare in modo sicuro, cioè, in modo che ognuno dei due possa limitare il mittente e il destinatario dei messaggi esclusivamente all’altro interlocutore. Si tratta forse del protocollo crittografico oggi più comunemente usato nella rete Internet, poiché è il protocollo standard per mezzo del quale i browser Web possono comunicare in modo sicuro con i server. Per amore di completezza, occorre osservare che SSL, originariamente progettato da Netscape, si è evoluto nello standard commerciale TLS; la nostra descrizione intenderà con SSL sia SSL che TLS.

L’SSL è un protocollo complesso con molte opzioni e in questo testo si presenta una sola delle sue varianti, e solo in una forma astratta e molto semplificata, in modo tale da focalizzarci principalmente sull’uso delle sue primitive crittografiche. Ciò che ci apprestiamo a studiare è un’interazione complessa fra client e server, in cui la crittografia asimmetrica serve a produrre una **chiave della sessione** utilizzabile per crittografare simmetricamente la comunicazione fra i due attori della sessione – il tutto evitando gli attacchi replay e di interposizione. Per migliorare l’efficacia della crittografia, la chiave della sessione è eliminata al termine della sessione cui si applica: ulteriori comunicazioni fra client e server richiedono la generazione di una nuova chiave.

Il protocollo SSL è avviato da un **client** allo scopo di comunicare in modo sicuro con un **server**. Prima dell’avvio del protocollo, s’ipotizza che il server s abbia ottenuto un **certificato**, denotato con cert_s , da un’autorità di certificazione **CA**. Questo certificato è una struttura dati che contiene i seguenti elementi:

- vari attributi (**attrs**) del server, come il *nome unico che lo contraddistingue* e il suo *nome comune* (DNS);
- l’identità di un algoritmo asimmetrico di cifratura $E()$ per il server;

- la chiave pubblica k_e di questo server;
- un intervallo di validità (interval) durante il quale il certificato si può considerare valido;
- una firma digitale a sulle informazioni di cui sopra effettuata da parte della CA, cioè, $a = S_{kCA}(\langle \text{attrs}, E_{ke}, \text{interval} \rangle)$.

Prima dell'attivazione del protocollo, si presume che il client abbia ottenuto l'algoritmo pubblico di verifica V_{kCA} per la specifica CA. Nel caso del Web, il browser impiegato dall'utente viene consegnato dal relativo produttore insieme con gli algoritmi di verifica e le chiavi pubbliche di alcune autorità di certificazione. L'utente può aggiungere o cancellare a proprio piacimento queste informazioni.

Quando un client si connette a un server, gli invia un valore casuale n_c di 28 byte. Il server s risponde inviando al client un proprio valore casuale n_s oltre al suo certificato cert_s . Il client verifica che $V_{kCA}(\langle \text{attrs}, E_{ke}, \text{interval} \rangle, a) = \text{true}$ e che il momento attuale sia compreso nell'intervallo di validità interval del certificato. Se entrambe le condizioni sono verificate, il server ha attestato la propria identità. Il client può quindi generare un **premaster secret** pms di 46 byte, e trasmettere $\text{cpms} = E_{ks}(\text{pms})$ al server. Esso ottiene in chiaro $\text{pms} = D_{kd}(\text{cpms})$. Entrambe le parti, adesso, sono in possesso di n_c , n_s e pms, e possono computare il **master secret** condiviso di 48 byte tramite $\text{ms} = H(n_c, n_s, \text{pms})$. Solo il client e il server possono calcolare ms, giacché solo loro conoscono pms. La dipendenza di ms da n_c e n_s , inoltre, garantisce che esso sia un nuovo valore, non usato per una precedente comunicazione. Il client e il server, a questo punto, computano entrambi le seguenti chiavi a partire da ms:

- una chiave di cifratura simmetrica k_{cs}^{crypt} per cifrare i messaggi dal client al server;
- una chiave di cifratura simmetrica k_{sc}^{crypt} per cifrare i messaggi dal server al client;
- una chiave di generazione di MAC k_{cs}^{mac} per generare autenticatori sui messaggi dal client al server;
- una chiave di generazione di MAC k_{sc}^{mac} per generare autenticatori sui messaggi dal server al client.

Per inviare un messaggio m al server, il client invia

$$c = E_{k_{cs}^{\text{crypt}}}(\langle m, S_{k_{cs}^{\text{mac}}}(m) \rangle).$$

Al ricevimento di c , il server ricostruisce

$$\langle m, a \rangle = D_{k_{cs}^{\text{crypt}}}(c)$$

e accetta m se $V_{k_{cs}^{\text{mac}}}(m, a) = \text{true}$. Analogamente, per inviare un messaggio m al client, il server invia

$$c = E_{k_{sc}^{\text{crypt}}}(\langle m, S_{k_{sc}^{\text{mac}}}(m) \rangle).$$

e il client ricostruisce

$$\langle m, a \rangle = D_{k_{sc}^{\text{crypt}}}(c)$$

e accetta m se $V_{k_{sc}^{\text{mac}}}(m, a) = \text{true}$.

Questo protocollo permette al server di limitare i riceventi dei suoi messaggi al client che ha generato pms, e i mittenti dei messaggi che esso accetta a quello stesso client. Analogamente, il client può limitare i destinatari dei messaggi che invia e il mittente dei messaggi che accetta alla parte che conosce k_d (cioè, la parte capace di decifrare cpms). In molte applicazioni, come le transazioni che si svolgono nel Web, il client deve verificare l’identità della parte che conosce k_d . Questo è uno degli scopi del certificato cert_s ; in particolare, il campo **attrs** contiene informazioni che il client può usare per determinare l’identità (per esempio, il nome del dominio) del server con il quale sta comunicando. Per applicazioni nelle quali anche il server deve avere informazioni sul client, l’SSL ha un’opzione tramite la quale un client può inviare un certificato al server.

Il protocollo SSL trova vasta applicazione anche al di fuori di Internet: le reti SSL VPN per esempio, sono oggi un concorrente di IPsec VPN. Mentre quest’ultimo è competitivo per la cifratura del traffico punto a punto, per esempio tra due divisioni di una società, le reti SSL VPN sono più flessibili ma meno efficienti. Per questo motivo, potrebbero essere scelte per la comunicazione fra un singolo impiegato e la sede della sua azienda.

15.5 Autenticazione degli utenti

Fin qui il tema dell’autenticazione ha riguardato messaggi e sessioni di lavoro, ma non gli utenti. Se un sistema non può autenticare un utente, che valore ha l’autenticazione del messaggio proveniente da quell’utente? Per i sistemi operativi un fondamentale problema di sicurezza riguarda quindi l’**autenticazione dell’utente**. Il sistema di protezione dipende dalla capacità d’identificare i programmi e i processi correntemente in esecuzione, che a sua volta è basata sulla capacità di identificare ogni utente del sistema. Normalmente un utente identifica se stesso, quindi si tratta di stabilire se l’identità di un utente è autentica. Generalmente l’autenticazione è basata su uno o più dei seguenti tre elementi: oggetti posseduti dall’utente (una chiave o una scheda); conoscenze dell’utente (un identificatore utente e una password); un attributo fisico dell’utente (impronta digitale, impronta della retina o firma).

15.5.1 Password

Il metodo più diffuso per identificare un utente è quello che prevede l’uso di **password**. Quando un utente s’identifica attraverso il proprio user ID o account name, riceve la richiesta d’immissione di una password. Se la password immessa dall’utente corrisponde a quella memorizzata nel sistema, il sistema presume che l’utente sia legittimo.

Spesso le password si usano per proteggere oggetti del calcolatore quando non esistono schemi di protezione più completi. Le password si possono considerare un caso particolare di chiavi o di abilitazioni. Per esempio, si potrebbe associare una password a ogni risorsa, come un file; in questo modo ogni volta che si chiede l’uso della

risorsa, si deve fornire la relativa password; se questa è giusta, si permette l'accesso. A diritti d'accesso diversi si possono associare password diverse. Si possono per esempio impiegare password diverse per ciascuna delle seguenti operazioni su file: lettura, aggiunta di dati e aggiornamento.

In pratica quasi tutti i sistemi concedono pieni diritti agli utenti sulla base di una sola password. Più password, in teoria, renderebbero il sistema più sicuro, ma anche meno comodo da usare, motivo per cui sistemi del genere sono rari. È un esempio di un principio generale: se la sicurezza rende qualcosa scomodo da usare, si preferisce spesso rinunciare alle misure di sicurezza, o aggirarle in qualche modo.

15.5.2 Vulnerabilità delle password

Le password sono assai comuni poiché sono facili da capire e da usare. Sfortunatamente si possono indovinare, rendere accidentalmente visibili, sottrarre o trasferire illegalmente da un utente autorizzato a uno privo d'autorizzazione.

Ci sono due comuni metodi per indovinare una password. Uno si fonda sulla conoscenza dell'utente (o d'informazioni che lo coinvolgono) da parte di intrusi (sia umani sia programmi): troppo spesso le persone usano per le password informazioni ovvie (come il nome del loro gatto o del loro consorte). L'altro modo è l'uso della *forza bruta*: enumerare tutte le possibili combinazioni di lettere, numeri e segni d'interpunkzione finché si trova la password cercata. Le password brevi sono quindi le più vulnerabili; una password di quattro cifre decimali permette solo 10.000 combinazioni. In media occorrono solo 5000 tentativi per indovinare quella giusta. Un programma che può fare un tentativo ogni millisecondo impiegherebbe solo cinque secondi a indovinare una password di quattro cifre. L'enumerazione non è però adatta all'individuazione delle password in sistemi che consentono password lunghe, che distinguono le lettere maiuscole dalle minuscole, e che permettono l'uso dei numeri e dei caratteri di punteggiatura. Naturalmente gli utenti devono approfittare delle maggiori possibilità di scelta e non devono usare, per esempio, soltanto le lettere minuscole.

Oltre a essere indovinate, le password possono anche essere individuate da una sorveglianza visiva o elettronica: un intruso può sbirciare sopra la spalla (**shoulder surfing**) di un utente mentre questo inizia una sessione di lavoro e carpire facilmente la sua password osservando la tastiera. Alternativamente, chiunque abbia accesso alla rete in cui risiede un calcolatore può inserire un monitor di rete che gli consenta di osservare tutti i dati trasferiti nella rete (**sniffing**), compresi gli identificatori degli utenti e le password. La cifratura dei flussi di dati contenenti le password risolve questo problema. Tuttavia, persino un sistema di questo tipo può essere vittima di furti. Per esempio, se per conservare le password si usa un file dedicato allo scopo, questo potrebbe essere copiato su altri sistemi per analizzarlo. Si pensi anche a un cavallo di Troia installato sul sistema in grado di rilevare ogni singolo carattere battuto sulla tastiera prima che questo sia inviato alla applicazione.

L'involontaria visibilità della password diventa un problema particolarmente grave se questa viene annotata dove può essere letta o persa. Alcuni sistemi obbligano gli

utenti a scegliere password lunghe o difficili da ricordare oppure a cambiare password con una certa frequenza; questo metodo può spingere qualche utente ad annotare per iscritto la propria password o a riutilizzarla, determinando una sicurezza assai minore di quella di sistemi che consentono password semplici.

L'ultimo metodo di compromissione delle password, il trasferimento illecito, deriva dalla natura umana. In molti centri di calcolo vige una regola che vieta la condivisione degli account; tale regola talvolta si stabilisce per ragioni amministrative, ma spesso è impiegata per migliorare la sicurezza. Per esempio, se uno userID è condiviso da più utenti e si è verificata con esso una violazione della sicurezza, è impossibile sapere chi l'ha usato al momento della violazione, o anche se era uno degli utenti autorizzati; con uno userID diverso per ciascun utente, ogni utente può essere direttamente reso responsabile del suo uso. Inoltre l'utente potrebbe notare qualcosa di insolito relativamente al proprio account, e scoprire la violazione. Talvolta gli utenti violano le regole di condivisione per aiutare loro amici o per aggirare la contabilizzazione; tale comportamento può causare l'accesso al sistema di utenti non autorizzati, e magari pericolosi.

Le password possono essere generate dal sistema o scelte dall'utente. Quelle generate dal sistema possono essere difficili da ricordare e quindi gli utenti potrebbero trascriverle. Quelle scelte dagli utenti sono invece più facili da indovinare, un utente potrebbe per esempio usare il proprio nome o il nome della sua automobile preferita. Gli amministratori possono controllare periodicamente le password degli utenti e informarli se sono troppo corte o troppo facili da indovinare. Prima di accettarla, alcuni sistemi verificano l'efficacia della password, cioè controllano che non sia facilmente intuibile. Alcuni sistemi prevedono l'*invecchiamento* delle password, costringendo gli utenti a modificarle a intervalli regolari, per esempio ogni tre mesi. Questo metodo non è del tutto sicuro, poiché gli utenti possono alternare sempre le stesse due password. Questo problema si risolve, come accade in alcuni sistemi, registrando la storia delle password utilizzate da ciascun utente allo scopo di impedirne il riutilizzo.

Lo schema delle password può avere parecchie varianti. Per esempio, la password può essere cambiata spesso, anche a ogni sessione di lavoro. Alla fine di *ogni* sessione di lavoro, il sistema o l'utente sceglie una nuova password che si dovrà usare per la sessione di lavoro successiva. In questo caso, anche se una password viene usata abusivamente, può essere usata una sola volta. Quando l'utente legittimo, all'apertura della sessione di lavoro successiva, prova a usare la password, trova che non più valida e scopre la violazione della sicurezza. A questo punto si può reagire con azioni che rimedino alla violazione della sicurezza.

15.5.3 Password cifrate

Comune a tutti questi metodi è la difficoltà di mantenere segrete le password all'interno del calcolatore. Come può il sistema memorizzare le password in modo sicuro, ma permetterne l'uso quando un utente presenta la propria password? Il sistema UNIX utilizza funzioni hash per evitare di mantenere segreta la propria lista di password.

Visto che la lista è trattata con funzioni hash e non crittografata, il sistema non può decifrarne i valori per determinare le password originali.

Vediamo come funziona questo meccanismo. Il sistema contiene una funzione estremamente difficile da invertire – i progettisti sperano che sia impossibile – ma semplice da calcolare. Cioè, dato un valore x , è facile calcolare il valore della funzione $f(x)$; ma, dato un valore $f(x)$, è “impossibile” calcolare x . Questa funzione si usa per cifrare tutte le password, che si memorizzano solo in questa forma. Quando un utente immette una password, questa viene cifrata e confrontata con quella già cifrata e memorizzata. Anche se la password cifrata viene vista, non potendo essere decifrata, è impossibile stabilire la password originale. Quindi non è necessario tenere segreto il file che contiene le password.

La pecca di questo metodo consiste nel fatto che il sistema operativo non ha più il controllo delle password. Anche se cifrate, chiunque disponga di una copia del file che le contiene può servirsi di efficienti procedure di hashing contro tale file, per esempio effettuando l’hash di ciascuna parola di un dizionario e confrontando il risultato con le password cifrate contenute nel file; se l’utente ha scelto come password una parola contenuta nel dizionario, la password viene scoperta. Se s’impiegano calcolatori sufficientemente veloci, o anche cluster di calcolatori lenti, un simile confronto può richiedere solo poche ore. Inoltre, poiché i sistemi UNIX fanno uso di un ben noto algoritmo di cifratura, un pirata informatico potrebbe conservare un insieme di password già violate. Per questo motivo i sistemi includono nell’algoritmo di cifratura anche un numero casuale, detto *salt*, che viene aggiunto alla password per far sì che, nel caso in cui due password coincidano, le loro versioni cifrate siano diverse. Il valore salt rende inoltre inefficace la cifratura di un dizionario, perché ogni termine del dizionario dovrebbe essere combinato con ogni valore salt per poter essere confrontato con le password memorizzate. Inoltre le nuove versioni di UNIX registrano le password cifrate in un file che può essere letto solo dall’amministratore del sistema (**superuser**). I programmi che confrontano le password inserite con quelle registrate eseguono `setuid` sull’utente *root* in modo che possano leggere questo file, ma gli altri utenti non possano farlo.

Un ulteriore punto debole dei sistemi di password di UNIX è determinato dal fatto che molti sistemi UNIX considerano significativi soltanto i primi otto caratteri; è quindi estremamente importante che gli utenti sfruttino appieno lo spazio delle password disponibile. A complicare ancor di più le cose, alcuni sistemi non consentono l’uso di parole del dizionario come password. Un buon metodo è quello di generare le proprie password usando la prima lettera di ciascuna parola di una frase facile da ricordare, impiegando sia lettere minuscole sia maiuscole, intercalate da un adeguato numero di segni d’interpunzione. Per esempio, la frase “Il nome di mia madre è Teresa.” può produrre la password “IndmmèT.!”; che è difficile da scoprire, ma, per l’utente, facile da ricordare. Un sistema più sicuro potrebbe permettere password formate da più caratteri, o anche permettere di includere nelle password il carattere di spazio, in modo che un utente possa utilizzare un’intera frase (**passphrase**).

15.5.4 Password monouso

Uno dei modi in cui un sistema può evitare i tentativi di sottrazione delle password è l'uso di un insieme di **password accoppiate**. All'inizio di una sessione di lavoro, il sistema sceglie a caso una coppia di password dall'insieme delle password accoppiate e presenta all'utente un elemento della coppia selezionata; l'utente deve fornire l'altro elemento. In questo tipo di sistema l'utente viene **sfidato** (*challenge*) a fornire la **risposta** (*response*) corretta.

Questo metodo si può generalizzare impiegando un algoritmo come password.

Le password algoritmiche non sono soggette a riutilizzo. Un utente può cioè immettere una password e nessuna entità che intercetta tale password potrà riutilizzarla. In questo schema, l'utente e il sistema condividono una password simmetrica pw che non è mai trasmessa in modi che rischino di comprometterne la riservatezza, ma viene piuttosto utilizzata come argomento per una funzione insieme a un valore ch (*challenge*) presentato dal sistema. Il valore restituito dalla funzione $H(pw, ch)$ viene comunicato al calcolatore per l'autenticazione. Poiché il calcolatore è a conoscenza sia di pw sia di ch , può anch'esso calcolare il valore della funzione. Se i due risultati coincidono, l'identità dell'utente è autenticata. All'accesso successivo il sistema genera un altro valore ch e si ripete il procedimento. Questa volta l'autenticatore calcolato sarà diverso. Le password monouso sono uno fra i pochissimi metodi capaci d'impedire l'errata autenticazione di un utente dovuta all'esposizione della password.

Prodotti commerciali che realizzano le password monouso impiegano specifiche calcolatrici elettroniche dotate di un display e talvolta di un tastierino numerico; molte di loro hanno la forma delle carte di credito, di portachiavi, o delle periferiche USB. Il software in esecuzione sul computer o sullo smartphone fornisce all'utente la funzione $H(pw, ch)$; pw può essere immesso dall'utente o generato dalla calcolatrice in sincronizzazione con il computer. A volte pw è solo un numero di identificazione personale (PIN). L'output di uno qualsiasi di questi sistemi consiste nella password monouso. Un generatore di password monouso che richiede l'input da parte dell'utente comporta un'autenticazione a due fattori (*two-factor authentication*). In questo caso sono richiesti due diversi tipi di componenti, per esempio un generatore di password monouso che genera la risposta corretta solo se il PIN è valido. L'autenticazione a due fattori offre una protezione decisamente migliore rispetto all'autenticazione a fattore singolo, poiché richiede sia "qualcosa che si ha", sia "qualcosa che si sa".

Una variante di questi sistemi usa un **libro dei codici** (*code book*), o **taccuino monouso** (*one-time pad*), cioè una lista di password usa e getta. In questo caso ogni password della lista si usa, nell'ordine, una sola volta, poi si cancella dalla lista. Il diffuso sistema S/Key impiega come fonte delle password monouso uno specifico programma di calcolo o un libro di codici compilato in base a questi calcoli. Naturalmente, l'utente deve proteggere il proprio libro dei codici. È utile, a tal fine, che il libro non identifichi il sistema su cui i codici sono utilizzati.

15.5.5 Tecniche biometriche

Un'altra variante d'uso delle password per l'autenticazione coinvolge l'applicazione di tecniche biometriche. Per garantire l'accesso sicuro ad ambienti fisici protetti, come centri d'elaborazione dati, spesso si usano strumenti per la rilevazione dell'impronta del palmo o dell'intera mano. Questi lettori confrontano i parametri memorizzati con quelli che rileva la tavoletta per la lettura dei parametri della mano. Tali parametri possono comprendere una mappa della temperatura, la lunghezza delle dita, la loro larghezza e il disegno delle linee. Questi dispositivi sono però ancora troppo ingombranti e costosi per essere usati per la normale autenticazione da parte di un calcolatore.

I lettori d'impronte digitali sono diventati accurati e sufficientemente economici e dovrebbero diffondersi maggiormente in futuro. Questi dispositivi leggono il disegno formato dalle increspature della pelle sulle dita e lo convertono in una sequenza di numeri. Di solito memorizzano un insieme di sequenze, per adattarsi alla posizione del dito sulla tavoletta di lettura e ad altri fattori. Uno specifico programma può a questo punto eseguire la scansione del dito e confrontare i dati ottenuti con quelli memorizzati, per vedere se corrispondono. Chiaramente, si possono mantenere profili di molti utenti e il dispositivo di scansione consente di distinguerli. Si può ottenere uno schema di autenticazione a due fattori molto accurato richiedendo sia un nome utente e la relativa password sia la scansione dell'impronta digitale. Se per il trasferimento si cifrano queste informazioni, il sistema può essere molto resistente alle tecniche d'imitazione delle identità e di replay.

L'autenticazione **multifattoriale** dà risultati ancora migliori. Si pensi alle garanzie offerte da un sistema d'autenticazione che preveda il collegamento di un dispositivo USB al sistema cui si intende accedere, l'inserimento di un PIN, e la lettura delle impronte digitali. A parte il fatto che l'utente dovrà collegare il dispositivo USB e poggiarvi sopra il dito, si tratta di un metodo d'autenticazione non meno comodo delle usuali password. Si ricordi, però, che un buon metodo di autenticazione non può di per sé garantire l'identità dell'utente: le sessioni autenticate possono sempre essere dirottate da intrusi, se non opportunamente crittate.

15.6 Realizzazione delle misure di sicurezza

Alla miriade di minacce che mettono a rischio le reti e i sistemi si possono contrapporre numerose strategie di sicurezza. Le strategie difensive spaziano dal miglioramento della consapevolezza degli utenti, alla tecnologia, fino allo scrivere programmi privi di bachi. Molti esperti nel campo della sicurezza sostengono la teoria della **difesa in profondità**, che afferma la necessità di moltiplicare gli strati di protezione per ottenere una difesa soddisfacente; tale teoria, ovviamente, si applica a qualsiasi contesto di sicurezza: si pensi alla differenza tra un'abitazione senza porta blindata, con una porta blindata, o con porta blindata e allarme. Nei seguenti paragrafi esamineremo i principali metodi, strumenti e tecniche utilizzabili al fine di migliorare la resistenza alle minacce.

15.6.1 Politiche di sicurezza

Il primo passo per consolidare la sicurezza dei calcolatori in ogni suo aspetto è adottare una **politica della sicurezza** (*security policy*). Tra le politiche di sicurezza vi può essere notevole differenza ma, solitamente, essi definiscono esattamente ciò che deve essere protetto. Una politica, per esempio, potrebbe dichiarare che tutte le applicazioni accessibili dall'esterno prevedano, prima del loro utilizzo, una revisione del codice, oppure vietare la condivisione delle password da parte degli utenti o ancora, se si tratta di una società, stabilire l'obbligo di eseguire la scansione delle porte, ogni sei mesi, per tutte le interconnessioni con l'esterno. Senza la definizione di una politica, sarebbe impossibile per utenti e amministratori distinguere tra ciò che è ammисibile, ciò che è necessario e ciò che non è permesso. Le politiche delineano un percorso operativo verso la sicurezza – indispensabile, per esempio, per definire le azioni da intraprendere per i siti che vogliono conseguire un livello di protezione più alto.

Una volta che le norme di sicurezza stabilite dalla politica siano operative, i soggetti a cui sono rivolte dovrebbero conoscerle bene e farne la propria guida. Il complesso di queste norme dovrebbe costituire un **documento vivo** (*living document*) da aggiornare e rivedere periodicamente, per garantirne pertinenza e validità nel corso del tempo.

15.6.2 Verifica delle vulnerabilità

Come si può accettare se una politica di sicurezza sia stata correttamente attuata? Il modo migliore consiste nell'eseguire una verifica delle vulnerabilità. Il raggio d'azione di queste verifiche può spaziare dall'ingegneria sociale, alla valutazione del rischio, alla scansione delle porte. La **valutazione del rischio** (*risk assessment*), per esempio, è il tentativo di stimare il valore di un certo oggetto (un programma, un settore dirigenziale, un sistema o un servizio), e la probabilità che attacchi alla sicurezza ne mettano a repentaglio il valore. Solo sulla base di tali stime si può quantificare il valore economico del mettere in sicurezza l'oggetto.

Lo strumento principale delle verifiche di vulnerabilità è il **test di penetrazione** (*penetration test*) che scandaglia l'oggetto in esame alla ricerca di vulnerabilità note. Poiché questo libro concerne i sistemi operativi e i programmi che vi sono eseguiti, la nostra trattazione si concentrerà su questi aspetti.

Le scansioni delle vulnerabilità si effettuano di preferenza quando l'elaboratore è scarsamente utilizzato, per ridurne al minimo l'impatto. Se opportuno, vengono sperimentate tramite simulazioni su sistemi di test, e non su sistemi reali, dato che possono provocare effetti indesiderati sul funzionamento del sistema o dei dispositivi di rete.

L'azione di scandaglio all'interno di un singolo sistema può individuare vari aspetti:

- password brevi o facili da indovinare;
- programmi privilegiati non autorizzati, come i programmi di *setuid*;
- programmi non autorizzati nelle directory di sistema;

- processi dall'esecuzione inaspettatamente lunga;
- improprie protezioni delle directory sia degli utenti sia di sistema;
- improprie protezioni dei file di dati del sistema, come il file delle password, i driver dei dispositivi, o anche dello stesso kernel del sistema operativo;
- elementi pericolosi nel percorso di ricerca dei programmi (per esempio, cavalli di Troia – Paragrafo 15.2.1);
- modifiche ai programmi di sistema, individuate con somme di controllo;
- demoni di rete inattesi o nascosti.

Ogni problema individuato dalla scansione di sicurezza può essere risolto automaticamente oppure notificato al gestore del sistema.

I calcolatori collegati in rete sono più soggetti ad attacchi contro la sicurezza di quanto lo siano i sistemi indipendenti. In questo caso gli attacchi arrivano da un insieme sconosciuto e vasto di punti d'accesso invece che da un insieme noto di punti d'accesso, come i terminali direttamente connessi. Anche se in misura minore, anche i sistemi collegati tramite modem alle linee telefoniche sono più esposti dei sistemi indipendenti.

Il governo degli USA, per esempio, considera i sistemi sicuri tanto quanto è sicuro il loro collegamento più esteso. A un sistema di massima segretezza è possibile accedere solo dall'interno di un edificio considerato di massima segretezza. Il sistema perde il proprio grado di massima segretezza se dall'esterno di tale ambiente può avvenire un qualsiasi tipo di comunicazione. Alcuni servizi governativi prendono misure di sicurezza estreme; quando un terminale non è in uso i connettori utilizzati per collegarlo con un calcolatore sicuro vengono chiusi in una cassaforte nell'ufficio. Ecco un esempio di autenticazione multifattoriale: se un individuo vuole entrare nell'edificio, e nell'ufficio, deve possedere appositi documenti di riconoscimento, conoscere la combinazione con cui aprire la serratura, e disporre dei dati di autenticazione che gli consentono l'accesso tramite il terminale.

Sfortunatamente per gli amministratori di sistemi e per i professionisti della sicurezza dei calcolatori, è spesso impossibile confinare una macchina in una stanza e disabilitare ogni accesso remoto. La rete Internet, per esempio, connette milioni di calcolatori e sta diventando una risorsa indispensabile per il lavoro di molte società e persone. Se si considera Internet come una comunità, come accade in ogni comunità formata di milioni di membri, moltissimi sono onesti e alcuni sono disonesti. Gli utenti disonesti di Internet dispongono di molti strumenti che consentono loro di tentare l'accesso ai calcolatori collegati, proprio come fece Robert Morris col proprio *Internet worm*.

La scansione delle vulnerabilità può essere applicata alle reti per risolvere alcuni problemi connessi alla sicurezza in rete. Ogni scansione ricerca porte che rispondano alle richieste. Se sono attivi servizi che non dovrebbero essere disponibili, il loro accesso può essere bloccato, oppure i servizi possono essere disabilitati. Le scansioni quindi analizzano i dettagli delle applicazioni che sono in ascolto su una data porta e

cercano eventuali vulnerabilità note; l'esame di queste vulnerabilità rivela se il sistema è configurato erroneamente o se è privo delle necessarie patch di sicurezza.

Infine è opportuno sottolineare come i dispositivi di scansione delle porte possano essere utilizzati da pirati informatici alla ricerca di vulnerabilità nei sistemi da attaccare, invece che da addetti alla sicurezza. (Fortunatamente, è possibile tener sotto controllo l'attività di scansione delle porte attraverso il rilevamento delle anomalie, come vedremo tra breve.) La sfida costante alla sicurezza nasce da un'ambivalenza: i medesimi strumenti, infatti, possono servire a fin di bene o per fare danni. Infatti alcuni hanno sostenuto il concetto di **sicurezza tramite segretezza**, basato sull'idea di non sviluppare alcuno strumento per il riconoscimento delle falte di sicurezza, perché potrebbe essere utilizzato per attaccare i sistemi. Altri contestano la validità di tale soluzione obiettando, per esempio, che i pirati informatici sono in grado di scrivere le proprie applicazioni da soli. Sembra ragionevole ritenere la sicurezza tramite segretezza solo uno degli strati della sicurezza, a patto che non resti l'unico. Per esempio, un'azienda può rendere nota al pubblico l'intera struttura della propria rete; se, invece, decidesse di tener segrete queste informazioni, gli intrusi avrebbero maggiori difficoltà nella scelta degli obiettivi da attaccare, e nello stabilire quali azioni possano essere identificate. Persino in questo caso, tuttavia, un'azienda darebbe prova di incerto ottimismo se considerasse scontata la segretezza a lungo termine di tali dati.

15.6.3 Rilevamento delle intrusioni

Il **rilevamento delle intrusioni**, come suggerisce il nome, è quell'attività volta a scoprire tentativi di intrusione o intrusioni già avvenute nei sistemi elaborativi e ad attivare azioni appropriate in risposta alle intrusioni stesse. Per il rilevamento delle intrusioni s'impiegano molte tecniche che si differenziano secondo vari fattori, fra cui i seguenti.

- La fase in cui avviene il rilevamento dell'intrusione: in tempo reale, ossia mentre si sta verificando, o solo successivamente.
- Le informazioni esaminate per scoprire l'attività intrusiva. Queste potrebbero comprendere comandi impartiti tramite shell, chiamate di sistema da parte dei processi, oltre a intestazioni e contenuto dei pacchetti di rete. Alcune forme d'intrusione si possono rilevare solo attraverso una correlazione delle informazioni acquisite da più di una di queste sorgenti.
- La varietà delle opzioni di risposta. Alcune semplici forme di risposta consistono nell'informare l'amministratore del sistema della potenziale intrusione oppure nel bloccare in qualche modo la potenziale attività intrusiva, per esempio arrestando un processo impegnato in tale attività. Con una forma di risposta più raffinata, un sistema potrebbe sviare, in modo trasparente, l'attività dell'intruso, portandolo verso un **honeypot**, una risorsa fittizia esposta agli attacchi. La risorsa appare reale agli intrusi, e permette di acquisire informazioni su di loro tenendone sotto controllo l'attività.

Questi gradi di libertà hanno portato a un'ampia gamma di soluzioni che vanno sotto il nome di **sistemi di rilevamento delle intrusioni (IDS)** e **sistemi di rilevamento e prevenzione delle intrusioni (IDPS)**. I sistemi IDS lanciano l'allarme quando rilevano un'intrusione, mentre quelli IDPS fungono da router, facendo passare il traffico fino a quando rilevano un'intrusione (e bloccando il traffico a quel punto).

Ma che cosa si intende per intrusione? È difficile definirne con esattezza il significato, quindi i sistemi IDS e IDPS attuali di solito seguono uno tra due metodi meno ambiziosi. Secondo il primo di questi, chiamato **rilevamento basato sulle tracce (signature-based detection)**, si esaminano i dati d'ingresso al sistema o il traffico della rete alla ricerca di particolari schemi o sequenze di azioni, chiamati **tracce**, che sono notoriamente indicazioni di attacchi. Un semplice esempio di rilevamento basato su tracce è la ricerca di pacchetti di rete che contengano la sequenza `/etc/passwd/` indirizzati a un sistema UNIX. Un altro esempio ancora è offerto da un programma antivirus, che analizza i file binari o pacchetti di rete alla ricerca di virus conosciuti.

Il secondo metodo è in genere conosciuto come **rilevamento di anomalie (anomaly detection)**, cerca di identificare, con varie tecniche, i comportamenti anomali in un sistema. Ovviamente, non tutte le attività di sistema anomale indicano un'intrusione, ma l'ipotesi che si fa è che le intrusioni generino spesso un comportamento anomalo. Un esempio di rilevamento di anomalie è il controllo delle chiamate di sistema usate da un demone, allo scopo di identificare se l'uso di queste chiamate differisce significativamente da quello usuale, indicando in questo caso la possibilità che il demone sia stato manipolato da un attacco basato su buffer overflow. Un altro esempio è il controllo dei comandi impartiti all'interprete per individuare comandi anomali da parte di un certo utente, oppure l'individuazione di un accesso che si svolge in un orario anomalo per un certo utente, eventi che potrebbero indicare che un aggressore è riuscito a violare l'account di quell'utente.

Il rilevamento basato su tracce e il rilevamento di anomalie si possono considerare come due facce della stessa medaglia: il rilevamento basato su tracce cerca di caratterizzare comportamenti pericolosi e li identifica quando questi si presentano nel sistema, mentre il rilevamento di anomalie cerca di caratterizzare i comportamenti normali (o non pericoloso) e identifica qualsiasi comportamento che differisce da questi.

Questi differenti metodi tuttavia portano a IDS e a IDPS con caratteristiche molto diverse tra loro. In particolare, il rilevamento di anomalie può riuscire a scoprire metodi d'intrusione che in precedenza erano sconosciuti (i cosiddetti **attacchi giorno zero, zero-day attacks**). Il rilevamento basato su tracce identifica invece solo gli attacchi conosciuti e che si possono codificare in uno specifico schema di azioni riconoscibili. Per questo motivo, gli attacchi che non erano stati considerati al momento della generazione delle tracce non potranno essere identificati. Il problema è ben noto alle aziende che commercializzano programmi antivirus, costrette a rilasciare frequentemente gli aggiornamenti delle *signature* a mano a mano che nuovi virus vengono creati e identificati con procedure manuali.

Il rilevamento di anomalie non è necessariamente superiore a quello basato su tracce. Infatti analizzare e caratterizzare accuratamente quello che è il comportamento

“normale” di un sistema è un problema considerevole per i sistemi che si basano sul rilevamento di anomalie. Se l’intrusione è già avvenuta quando il sistema viene analizzato, l’attività intrusiva potrebbe essere inclusa nel comportamento “normale”. Anche se il sistema viene analizzato correttamente, senza l’influenza di comportamenti intrusivi, l’analisi deve fornire un’immagine sufficientemente completa del comportamento normale, altrimenti il numero di falsi positivi (falsi allarmi) o peggio falsi negativi (intrusioni non individuate) sarebbe eccessivo.

Per illustrare l’impatto che può avere una frequenza di falsi allarmi anche solo moderatamente alta, si consideri un’installazione composta da un centinaio di stazioni di lavoro con sistema operativo UNIX dalle quali si registrano dati relativi a eventi legati alla sicurezza, allo scopo di rilevare le intrusioni. Un’installazione di piccole dimensioni come questa può facilmente generare un milione di record di verifica (*audit*) al giorno. Tuttavia, solo una o due di queste potrebbero essere tali da meritare un’analisi più approfondita da parte dell’amministratore. Se si suppone, in modo ottimistico, che ogni attacco si rifletta in dieci annotazioni di verifica, si può calcolare con una certa approssimazione la frequenza di annotazioni di verifica che riflettono le vere attività intrusive per mezzo della formula:

$$\frac{2 \frac{\text{intrusioni}}{\text{giorno}} \cdot 10 \frac{\text{annotazioni}}{\text{intrusione}}}{10^6 \frac{\text{annotazioni}}{\text{giorno}}} = 0,00002$$

Interpretando il valore che si ottiene come una “probabilità di occorrenza di record d’intrusione”, la si denota con $P(I)$; cioè, l’evento I rappresenta l’occorrenza di un record che riflette un reale comportamento intrusivo. Poiché $P(I) = 0,00002$, si trova che $P(\neg I) = 1 - P(I) = 0,99998$. Si supponga ora che A denoti l’attivazione di un allarme da parte dell’IDS. Un IDS accurato dovrebbe massimizzare sia $P(I|A)$ sia $P(\neg I|\neg A)$, cioè, sia la probabilità che un allarme indichi un’intrusione, sia la probabilità che l’assenza di un allarme indichi che non vi sono intrusioni. Considerando per il momento $P(I|A)$, si può calcolare tale valore usando la **formula di Bayes**:

$$P(I|A) = \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)} = \frac{0,00002 \cdot P(A|I)}{0,00002 \cdot P(A|I) + 0,99998 \cdot P(A|\neg I)}$$

Si consideri ora l’impatto della frequenza di falsi allarmi $P(A|\neg I)$ su $P(I|A)$. Anche nel caso di una frequenza di veri allarmi molto buona, con $P(A|I) = 0,8$, da una frequenza di falsi allarmi apparentemente buona, $P(A|\neg I) = 0,0001$, si ottiene $P(I|A) \approx 0,14$. Questo significa che meno di uno su sette allarmi indica una vera intrusione. Nei sistemi in cui un amministratore della sicurezza investiga su ogni allarme, questa frequenza di falsi allarmi porterebbe a un enorme spreco di tempo e convincerebbe molto presto l’amministratore a ignorare tutti gli allarmi.

Quest’esempio mostra un principio generale per i sistemi IDS E IDPS: per essere utilizzabili proficuamente, devono offrire una frequenza di falsi allarmi estremamente

bassa. Per i sistemi di rilevamento di anomalie, il raggiungimento di una frequenza di falsi allarmi sufficientemente bassa è un problema particolarmente serio, proprio per le difficoltà che si hanno nell’analizzare e caratterizzare adeguatamente il comportamento normale dei sistemi. Tuttavia, la ricerca nel campo del rilevamento di anomalie registra continui miglioramenti. Il software per la rilevazione delle intrusioni si evolve in modo da integrare tracce di attacchi, algoritmi per le anomalie, e altri algoritmi ancora, così da conseguire una più accurata frequenza di rilevazione delle anomalie.

15.6.4 Protezione dai virus

I virus, come sappiamo, possono arrecare ingenti danni ai sistemi, e sono dunque una questione importante per la sicurezza. Per proteggersi si ricorre di frequente all’opera di programmi antivirus. Alcuni di questi programmi hanno efficacia solo verso particolari virus, che sono già noti: essi operano cercando, in tutti i programmi di un sistema, la specifica sequenza di istruzioni che compone il virus. Quando individuano la sequenza incriminata, eliminano le istruzioni per **disinfettare** il programma. I programmi antivirus possono avere elenchi con migliaia di definizioni dei virus ricercati.

Sia i virus sia i programmi antivirus diventano sempre più sofisticati. Alcuni virus si modificano mentre infettano i programmi, allo scopo di evitare la ricerca per sequenze note, che è il metodo base dei programmi antivirus. Questi, a loro volta, si evolvono cercando famiglie di sequenze di istruzioni, piuttosto che sequenze singole; alcuni programmi antivirus implementano vari algoritmi di ricerca differenti, e possono anche decomprimere i virus compressi prima di cercarne la firma. Alcuni di loro hanno anche una funzionalità per rilevare le anomalie dei processi: un processo che apre un file eseguibile per la scrittura, per esempio, è sospetto, a meno che sia un compilatore. Un’altra tecnica diffusa prevede l’esecuzione dei programmi in una **scatola di sabbia** (*sandbox*), che è un sottoambiente emulato o controllato, non comunicante con il resto del sistema. Il programma antivirus osserva l’operato del codice nella scatola di sabbia, prima di permettere che sia eseguito senza controllo. Alcuni antivirus, inoltre, anziché limitarsi all’analisi dei file nel file system, innalzano un intero scudo a protezione del sistema. Questi programmi setacciano le partizioni di avviamento, la memoria, la posta elettronica in entrata e in uscita, i file mentre vengono scaricati, i file ospitati da dischi e memorie rimovibili, e così via.

La migliore protezione contro i virus informatici è la prevenzione (o *safe computing*). Acquistare programmi sigillati dai rivenditori autorizzati, ed evitare le copie abusive provenienti da fonti pubbliche o da scambi di dischi, rappresentano il comportamento più sicuro per tenersi alla larga dalle infezioni. D’altra parte, persino i programmi legittimi non sono immuni dalle infezioni virali: ci sono stati casi di impiegati scontenti che, per causare danni alla società in cui lavoravano, hanno infettato le copie originali dei programmi messi in commercio dalla società stessa. Per quanto riguarda i virus delle macro, quando si scambiano documenti Microsoft Word, una misura di prevenzione consiste nell’utilizzo del formato di file **rich text format** (RTF).

A differenza del formato originale Word, l'RTF non è predisposto per l'inclusione delle macro.

Un'ulteriore cautela è evitare l'apertura di qualunque allegato di posta elettronica, se il messaggio proviene da un indirizzo sconosciuto. È assodato, purtroppo, che mentre si pone riparo alle insidie trasmesse per posta elettronica, se ne scoprono costantemente di nuove. Nel 2000, per esempio, il virus *love bug* si diffondeva ampiamente sotto la falsa apparenza di un messaggio d'amore inviato da un amico del ricevente. Una volta aperto, lo script allegato in Visual Basic si propagava autotrasmettendosi ai primi indirizzi presenti in rubrica. Fortunatamente, al di là dell'intasamento dei sistemi di posta elettronica e delle cartelle della posta in arrivo degli utenti, era un virus piuttosto innocuo. Dimostrava, tuttavia, come si potesse vanificare la misura difensiva consistente nell'aprire solo gli allegati di cui si conosceva il mittente. Un metodo di difesa più efficace consiste nell'evitare l'apertura di tutti gli allegati che contengono codice eseguibile. Alcune aziende impongono tale metodo come criterio di sicurezza generale, rimuovendo tutti gli allegati dai messaggi in arrivo.

Un'altra precauzione, ancorché non eviti le infezioni, ne consente una tempestiva individuazione. Si comincia riformattando completamente i dischi del sistema, i settori d'avviamento, spesso soggetti ad attacchi da parte dei virus. Si caricano solamente programmi sicuri e si calcola una firma per ogni programma tramite il calcolo dei valori di una funzione hash. È necessario conservare la lista con i nomi dei file e le firme associate, in modo da impedire l'accesso non autorizzato. A intervalli periodici, oppure ogni volta che viene eseguito un programma, il sistema operativo ricalcolerà le firme e le confronterà con quelle dell'elenco originale; ogni differenza funge da allarme per possibili infezioni.

Questa tecnica può essere combinata con altre. Si può effettuare, per esempio, un esame antivirus molto dispendioso in termini di risorse computazionali, quale quello della scatola di sabbia, creando una firma da associare ai programmi che superano la prova. Se, al nuovo avvio del programma, le firme corrispondono a quelle dell'avvio precedente, si può fare a meno di sottoporre il programma all'antivirus.

15.6.5 Verifica, accounting e log

Verifica (*auditing*), accounting e log possono diminuire le prestazioni del sistema, ma sono utili in diverse aree; una di queste è la sicurezza. L'uso di log (*logging*) può essere generale o specifico. Tutte le chiamate di sistema possono essere registrate per monitorare il comportamento dei programmi e scoprirne le anomalie. Più normalmente, sono registrati in un log gli eventi di natura sospetta. I casi di autenticazione e di autorizzazione fallita possono rivelare molto sui tentativi di violazione.

L'accounting può aggiungersi alla dotazione di strumenti degli amministratori della sicurezza; può essere utile a rivelare cambiamenti nelle prestazioni, che a loro volta potrebbero celare problemi di sicurezza. Uno tra i primi episodi di violazione ai calcolatori UNIX fu scoperto da Cliff Stoll, che notò un'anomalia mentre esaminava i log di accounting.

FILE SYSTEM TRIPWIRE

Un esempio di semplice strumento per il rilevamento di anomalie è il **Tripwire**, uno strumento per il controllo dell'integrità del file system del sistema operativo UNIX, sviluppato alla Purdue University. Il Tripwire opera seguendo l'ipotesi che una vasta classe d'intrusioni generi modifiche nei file e nelle directory di sistema. Per esempio, un intruso potrebbe modificare alcuni programmi di sistema, magari inserendo delle copie con cavalli di Troia, o inserire nuovi programmi nelle directory che si trovano di solito nei percorsi di ricerca degli interpreti di comandi degli utenti. Oppure, un intruso potrebbe rimuovere file di log del sistema per nascondere le proprie tracce. Il Tripwire è uno strumento che serve a tenere sotto controllo il file system rispetto alle operazioni di aggiunta, cancellazione e modifica dei file e per avvertire gli amministratori di queste modifiche.

Il comportamento del Tripwire è controllato da un file di configurazione, `tw.config`, che elenca le directory e i file per i quali si vogliono tenere sotto controllo le modifiche, le cancellazioni o le aggiunte. Ogni elemento di questo file di configurazione comprende una maschera di selezione che specifica quali attributi del file (attributi dell'*inode*) si devono controllare. Per esempio, la maschera di selezione potrebbe specificare che si rilevano i cambiamenti relativi ai permessi di accesso al file, ma si ignori la sequenza temporale degli accessi. Inoltre la maschera potrebbe specificare il controllo dei cambiamenti del contenuto del file. Tenere sotto controllo il valore hash di un file per rilevarne cambiamenti ha lo stesso effetto di controllare il file stesso, ma la memorizzazione del valore hash richiede molto meno spazio.

La prima volta che si attiva, il Tripwire prende in ingresso il file `tw.config` e calcola una firma (*signature*) per ciascun file o directory, contenente gli attributi che si devono controllare (attributi di *inode* e valori hash). Queste tracce si memorizzano in una base di dati. Le volte successive, il Tripwire considera come ingresso sia `tw.config` sia la base di dati precedentemente memorizzata, ricalcola la traccia per ciascun file o directory elencato in `tw.config` e confronta questa traccia con quella (se c'è) memorizzata nella base di dati. Gli eventi che si segnalano all'amministratore comprendono una diversa firma di un file o directory rispetto a quella memorizzata precedentemente nella base di dati (un file modificato), ogni file o directory per i quali non esisteva una traccia nella base di dati (un file aggiunto) e qualsiasi traccia nella base di dati per la quale non esiste più un file o una directory (un file eliminato).

Sebbene funzioni egregiamente per un'ampia classe di attacchi, il Tripwire ha alcuni limiti. Forse il più ovvio è la necessità di proteggere da modifiche non autorizzate i file di programma dello stesso Tripwire e i file di dati associati, inclusa la base di dati. Per questa ragione, il Tripwire e i file associati dovrebbero risiedere in qualche mezzo a prova di manomissione, come un disco protetto dalle scritture oppure un server sicuro dove le connessioni al sistema sono controllate rigorosamente. Sfortunatamente questo rende più scomodo l'aggiornamento della base di dati dopo modifiche autorizzate alle directory e ai file tenuti sotto controllo. Una seconda limitazione è dovuta al fatto che alcuni file rilevanti per la sicurezza, per esempio i file di log di sistema, devono cambiare nel tempo e il Tripwire non fornisce un modo per distinguere tra modifiche autorizzate e non autorizzate. Quindi, per esempio, un attacco che modifica (senza cancellarlo) un file di log di sistema, che sarebbe stato comunque modificato dalla normale attività di sistema, sfuggirebbe alla capacità di rilevamento del Tripwire. Il meglio che potrebbe fare in questo caso è il rilevamento di determinate banali incoerenze (per esempio, se il file di log si riduce di dimensioni). Di questo strumento esistono sia versioni commerciali sia libere, disponibili su <http://tripwire.org> e <http://tripwire.com>.

15.7 Firewall a protezione di sistemi e reti

Torniamo ora al problema di come i calcolatori fidati si possano collegare in modo sicuro a una rete non fidata. Una soluzione è data dall'uso di un **firewall** (*barriera di sicurezza*), che separa i sistemi fidati da quelli non fidati; si tratta di un calcolatore, di un router o di un dispositivo specializzato situato tra questi due gruppi di sistemi. Un firewall di rete limita l'accesso di rete tra i due **domini di sicurezza** e controlla e registra tutte le connessioni. Può anche bloccare le connessioni a seconda degli indirizzi di sorgente o di destinazione, delle porte di sorgente o di destinazione, o della direzione delle connessioni. I server web, per esempio, impiegano il protocollo HTTP per comunicare con i browser; in tal caso il firewall potrebbe consentire l'accesso al server, da parte dei sistemi esterni, solo col protocollo HTTP ma, per esempio, impedire l'accesso tramite il protocollo *finger*, usato dall'*Internet worm* di Morris per inserirsi nei calcolatori.

Un firewall può così separare una rete in più domini. Uno schema comune considera la rete Internet come dominio non fidato; prevede una rete parzialmente fidata, la cosiddetta **zona smilitarizzata** (*demilitarized zone*, DMZ), come secondo dominio; e un terzo dominio che comprende i calcolatori aziendali (Figura 15.10). Le connessioni sono consentite da Internet ai calcolatori della DMZ e dai calcolatori aziendali alla rete Internet, ma non da Internet né dalla DMZ ai calcolatori aziendali. Opzionalmente, può essere consentita una comunicazione controllata fra uno o più calcolatori aziendali e la DMZ. Per esempio un web server nella DMZ può effettuare una query su un database nella rete aziendale. In questo modo ogni accesso è controllato, e qualsiasi sistema della DMZ, anche se compromesso, non può in ogni caso accedere ai calcolatori aziendali.

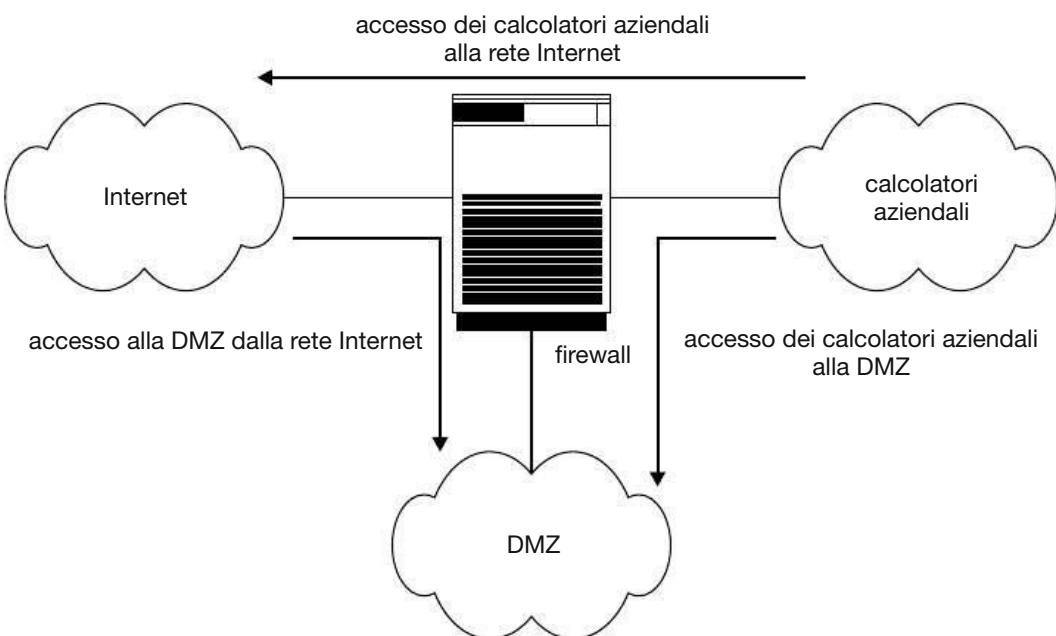


Figura 15.10 Sicurezza di rete con separazione in domini tramite firewall.

Ovviamente, lo stesso sistema che costituisce il firewall deve essere sicuro e resistente agli attacchi, altrimenti le sue capacità di fornire connessioni sicure possono essere compromesse. I firewall non riescono a prevenire gli attacchi che utilizzano un **tunnel**, ossia si trasmettono all'interno di protocolli e connessioni che gli stessi firewall permettono. Un attacco basato sul trabocco del buffer diretto a un server web non viene bloccato da un firewall perché le connessioni HTTP sono permesse, ed è proprio il contenuto della connessione HTTP che ospita l'attacco. Allo stesso modo gli attacchi per rifiuto del servizio possono colpire i firewall proprio come ogni altra macchina. Un altro punto vulnerabile dei firewall è la contraffazione delle identità (**spoofing**), tramite la quale un sistema non autorizzato finge di esserlo, riuscendo a soddisfare alcuni criteri di autorizzazione. Per esempio, se una regola del firewall permette una connessione da uno specifico sistema e lo identifica con il suo indirizzo IP, un altro sistema potrebbe inviare pacchetti servendosi proprio di quell'indirizzo e quindi ottenere il permesso d'accesso.

Oltre ai comuni firewall di rete, ve ne sono altri tipi più recenti, ognuno con i suoi vantaggi e svantaggi. Con **firewall personale** si intende un livello di software incluso nel sistema operativo, o aggiuntovi come applicazione. Esso non limita la comunicazione tra domini della sicurezza, ma tra il sistema e un dato host. Un utente, per esempio, può aggiungere un firewall personale al proprio PC in modo da negare a un cavallo di Troia l'accesso alla rete cui il PC è connesso. Un **firewall proxy per le applicazioni** comprende il protocollo seguito da certe applicazioni sulla rete. SMTP, per esempio, è usato per il trasferimento di posta elettronica; un suo proxy accetta connessioni esattamente come se fosse un server, per poi connettersi al vero server SMTP destinatario. Può quindi tenere sotto controllo il traffico in transito, tentando di rilevare e disabilitando comandi illegali, scoprendo i tentativi di sfruttamento dei bachi, e così via. Alcuni firewall di questo tipo sono progettati espressamente per un unico protocollo: un **firewall XML**, per esempio, analizza solo i contenuti XML, bloccando il codice XML non permesso o malformato. I **firewall delle chiamate di sistema** risiedono tra le applicazioni e il kernel, e monitorano l'esecuzione delle chiamate. In Solaris 10, per esempio, l'opzione "privilegio minimo" implementa una lista di più di cinquanta chiamate che i processi possono o non possono invocare. I processi che non hanno bisogno di generare figli, per esempio, possono vedersi vietata la possibilità di eseguire chiamate di sistema con quello scopo.

15.8 Classificazione della sicurezza dei calcolatori

Nel documento *Department of Defence Trusted Computer System Evaluation Criteria*, del Dipartimento della Difesa degli USA, sono specificate quattro categorie di sicurezza: *A*, *B*, *C* e *D*. Questa classificazione che ora esaminiamo è ampiamente usata per la verifica della sicurezza di un sistema e per classificare le soluzioni di sicurezza. La categoria di livello più basso è la *D*, e corrisponde al minimo di protezione. Non ha sottocategorie, e si assegna ai sistemi che non soddisfano i criteri di nessuna delle categorie rimanenti. A essa appartengono, per esempio, i sistemi operativi MS-DOS e Windows 3.1.

Il livello successivo, la categoria *C*, offre forme di protezione discrezionale insieme con la possibilità d’identificare e controllare gli utenti e le loro azioni per mezzo di strumenti di verifica. La categoria *C* è suddivisa nelle due classi *C1* e *C2*. Un sistema appartenente alla prima di esse incorpora qualche forma di controllo che permette agli utenti di proteggere le loro informazioni private e che impedisce ad altri di leggerle accidentalmente o di distruggerle. La classe *C1* è un ambiente in cui più utenti che collaborano accedono ai dati allo stesso livello di riservatezza. La maggior parte delle versioni del sistema operativo UNIX appartiene a questa classe.

L’insieme dei sistemi di protezione di un sistema elaborativo (hardware, software, firmware) che assicura l’imposizione dei criteri di sicurezza è noto come **piattaforma di calcolo fidata** (*trusted computer base*, TCB). La TCB di un sistema di classe *C1* controlla l’accesso degli utenti ai file permettendo agli utenti di regolamentare la condivisione dei loro oggetti con individui identificati o gruppi definiti. Inoltre, la TCB richiede che ogni utente dichiari la propria identità prima che egli inizi una qualunque attività mediata dalla TCB stessa. Il processo d’autenticazione si basa su un meccanismo di protezione o sulle password; la TCB protegge i dati d’autenticazione rendendoli inaccessibili agli utenti non autorizzati.

Un sistema appartenente alla classe *C2* unisce alle caratteristiche di un sistema di classe *C1* la capacità di controllare gli accessi a livello individuale. Per esempio, i diritti d’accesso a un file si possono specificare fino a livello del singolo individuo. Inoltre, l’amministratore del sistema può verificare le azioni di uno o più utenti, specificamente individuati. La TCB protegge anche il proprio codice e le proprie strutture dati da eventuali tentativi di modifica. Inoltre, nessuna informazione precedentemente prodotta da un utente è disponibile a un altro utente che acceda a un oggetto di memoria che è stato liberato. Alcune versioni di UNIX particolarmente sicure appartengono alla classe *C2*.

I sistemi a protezione forzata della categoria *B* hanno tutte le proprietà dei sistemi di classe *C2* e applicano a ogni oggetto un’etichetta che indica il livello di riservatezza. Il TCB della classe *B1* mantiene tali etichette, che vengono usate per prendere decisioni riguardanti il controllo forzato degli accessi. Per esempio, un utente classificato a livello “confidenziale” non può avere accesso a un file che sia classificato al più riservato livello “segreto”. La TCB specifica anche il livello di riservatezza all’inizio e alla fine di ogni pagina prodotta in forma leggibile. Oltre le normali informazioni di autenticazione (nome utente e password), la TCB gestisce le autorizzazioni e i permessi dei singoli utenti e incorpora almeno due livelli di sicurezza. Questi livelli sono gerarchici, cosicché un utente può aver accesso a un qualunque oggetto cui è assegnata un’etichetta di riservatezza di livello pari o inferiore alla sua autorizzazione. Per esempio, un utente di livello “segreto”, in assenza di prescrizioni più dettagliate, può accedere a un file di livello “confidenziale”. I processi sono inoltre isolati l’uno dall’altro tramite l’uso di spazi d’indirizzi distinti.

Un sistema di classe *B2* estende l’uso di etichette di riservatezza a ogni risorsa del sistema. Ai dispositivi fisici sono assegnati livelli minimi e massimi di sicurezza che il sistema usa per garantire il rispetto dei vincoli imposti dall’ambiente fisico nel quale

tali dispositivi sono situati. Inoltre, un sistema di classe *B2* permette la protezione dei canali di comunicazione e la verifica degli eventi che potrebbero portare allo sfruttamento illegittimo di un canale protetto.

Un sistema di classe *B3* permette la creazione di liste di controllo degli accessi che identificano utenti o gruppi ai quali *non* sia consentito l'accesso a un oggetto specificato. La TCB contiene anche un meccanismo di controllo di quegli eventi che potrebbero indicare una violazione del protocollo di sicurezza. Il meccanismo informa l'amministratore del sistema e, se è necessario, forza la conclusione degli eventi nella maniera meno dannosa possibile.

La categoria *A* è il grado più alto della classificazione. Un sistema di classe *A1* è dal punto di vista dell'architettura equivalente a un sistema di classe *B3*, ma si deve realizzare impiegando metodi formali di definizione e verifica, garantendo con un'elevato livello di sicurezza che la TCB sia stata correttamente realizzata. Un sistema appartiene a una classe superiore ad *A1* se, per esempio, è stato progettato e realizzato in un impianto di produzione affidabile da personale affidabile.

La TCB assicura solo che il sistema possa garantire il rispetto dei vincoli previsti da una politica di sicurezza, ma non specifica il contenuto della politica. In genere, per un dato sistema informatico si sviluppa la politica di sicurezza per la **certificazione**, che viene poi vagliata e **accreditata** da un'agenzia specializzata, per esempio l'NCSC (*national computer security center*). Per alcuni sistemi informatici si potrebbe aver bisogno di altre certificazioni, come quella rilasciate dal TEMPEST, che garantisce la protezione dall'intercettazione elettronica. Per esempio, i terminali di un sistema certificato dal TEMPEST sono schermati in modo da evitare la propagazione esterna del campo elettromagnetico. Questa schermatura assicura che un'attrezzatura posta fuori della stanza o dell'edificio in cui i terminali sono situati non possa rilevare alcuna informazione visualizzata sul terminale.

15.9 Un esempio: Windows 7

Microsoft Windows 7 è un sistema operativo a uso generale progettato per disporre di una varietà di funzioni e metodi di sicurezza. In questo paragrafo si esaminano gli strumenti impiegati da questo sistema operativo per garantire le funzionalità di sicurezza; per maggiori informazioni su Windows 7 si veda il Capitolo 19 (presente sul sito web del volume).

Il modello di sicurezza si basa sulla nozione di **utente accreditato del sistema** (*user account*). Il sistema operativo Windows 7 permette la creazione di un numero arbitrario di user account, che si possono raggruppare in un qualunque modo. L'accesso agli oggetti del sistema si può poi consentire o negare come si vuole. Il sistema identifica gli utenti per mezzo di un identificatore di sicurezza *unico*. Quando un utente accede al sistema, si crea un **contrassegno d'accesso di sicurezza** (*security access token*) che include l'identificatore di sicurezza dell'utente, gli identificatori di sicurezza per tutti i gruppi dei quali l'utente è membro, e una lista di tutti i privilegi speciali di cui l'utente gode. Alcuni esempi di privilegi speciali sono la possibilità di fare

copie di backup di file e directory, di spegnere il calcolatore, di accedere al sistema in modo interattivo e di regolare l’orologio del sistema. Ogni processo che il sistema esegue per conto di un utente riceve una copia del security token. Ogniqualvolta l’utente – direttamente o per mezzo di un processo – tenti di accedere a oggetti del sistema, l’accesso è negato o concesso secondo gli identificatori di sicurezza contenuti nel token. L’autenticazione è di solito effettuata tramite nome utente e password, anche se la struttura modulare del sistema Windows 7 permette lo sviluppo di metodi di autenticazione specifici. Per esempio, si potrebbe usare un analizzatore elettronico della retina per verificare la reale identità dell’utente.

Il sistema Windows 7 usa l’idea del soggetto per far sì che i programmi eseguiti per conto di un utente non ottengano permessi d’accesso al sistema più ampi di quelli dell’utente stesso. Un **soggetto** (*subject*) si usa per identificare e gestire i permessi relativi a ogni programma eseguito dall’utente. È composto dall’access token e dal programma che agisce per conto dell’utente. Poiché il sistema Windows 7 si basa su un modello client-server, per controllare gli accessi si usano due classi di soggetti: soggetti semplici e soggetti server. Un esempio di **soggetto semplice** è il tipico programma applicativo che un utente esegue dopo aver ottenuto l’accesso al sistema. A un soggetto semplice si assegna un **contesto di sicurezza** definito secondo l’access token dell’utente. Un **soggetto server** è un processo realizzato come un server protetto che usa il contesto di sicurezza del client quando agisce per suo conto.

Come si è detto nel Paragrafo 15.7, l’auditing è un utile strumento per migliorare la sicurezza. Il sistema Windows 7 è dotato di un sistema di verifica integrato che permette il controllo di molte comuni minacce per la sicurezza del sistema. Alcuni esempi di casi in cui la verifica è utile per localizzare minacce alla sicurezza sono la registrazione degli accessi non riusciti per individuare i tentativi d’accesso con password casuali e quella degli accessi riusciti per scoprire attività all’interno del sistema in orari insoliti; inoltre, al fine di prevenire il diffondersi di un virus, è utile tener traccia dei tentativi di scrittura – riusciti e falliti – su file eseguibili; infine, la registrazione dei tentativi d’accesso a un file permette d’individuare gli accessi ai file riservati.

Windows ha aggiunto un controllo di integrità obbligatorio, che funziona assegnando un’etichetta di integrità a ogni oggetto e soggetto da proteggere. Per accedere a un oggetto, un dato soggetto deve avere l’accesso richiesto nell’elenco di controllo di accesso discrezionale e la sua etichetta di integrità deve essere uguale o superiore a quella dell’oggetto protetto (per l’operazione in questione). Le etichette di integrità in Windows 7 sono (in ordine crescente): non attendibile (untrusted), bassa (low), media (medium), alta (high) e di sistema (system). Inoltre, tre bit della maschera di accesso sono utilizzati per le etichette di integrità: NoReadUp, NoWriteUp e NoExecuteUp. NoWriteUp viene applicata automaticamente, di modo che un soggetto di integrità inferiore non possa eseguire un’operazione di scrittura su un oggetto di integrità superiore. Tuttavia, se non esplicitamente vietato dal descrittore di protezione, il soggetto può eseguire operazioni di lettura ed esecuzione.

Agli oggetti protetti privi di un’esplicita etichetta di integrità, viene assegnata l’etichetta “media”, predefinita. L’etichetta per un dato soggetto viene assegnata durante

l'accesso. Per esempio, un utente che non è amministratore riceverà un'etichetta di integrità media. Oltre alle etichette di integrità, Windows 7 utilizza il meccanismo UAC (User Account Control), che rappresenta un account amministratore (diverso da quello predefinito) con due token separati: uno, per un utilizzo normale, vede disabilitato il gruppo Administrators predefinito e ha un'etichetta di integrità media; l'altro, per un utilizzo di livello più alto, vede il gruppo Administrators abilitato e ha un'etichetta di integrità alta.

Gli attributi di sicurezza di un oggetto del sistema Windows 7 sono descritti da un **descrittore di sicurezza**, contenente l'identificatore di sicurezza del proprietario dell'oggetto (che può cambiare i permessi d'accesso, un identificatore di sicurezza di gruppo usato solo dal sottosistema POSIX; una lista di controllo discrezionale degli accessi che stabilisce quali utenti o gruppi abbiano possibilità d'accesso (o abbiano esplicitamente negata questa possibilità) e una lista di sistema di controllo degli accessi che controlla quali messaggi di verifica saranno generati. Opzionalmente la lista di sistema di controllo degli accessi può impostare l'integrità di un oggetto e indicare le operazioni da vietare ai soggetti con integrità più bassa: lettura, scrittura (sempre vietata) o esecuzione. Per esempio, il descrittore di sicurezza del file `foo.bar` potrebbe essere di proprietà di `avi` e avere la seguente lista di controllo discrezionale degli accessi:

- `avi` – consentito ogni accesso;
- gruppo `cs` – consentiti accessi per lettura e scrittura;
- utente `cliff` – nessun accesso consentito.

Inoltre, il descrittore potrebbe includere una lista di sistema di controllo degli accessi che richieda di verificare le scritture di ogni utente, oltre a un'etichetta di integrità media per vietare scrittura, lettura ed esecuzione a soggetti di integrità più bassa.

Una lista di controllo degli accessi contiene elementi composti dell'identificatore di sicurezza dell'individuo e della maschera d'accesso che definisce tutte le azioni permesse sull'oggetto, che può assumere il valore *accesso consentito* o *accesso negato* per ogni azione. I file di Windows 7 possono avere i seguenti tipi d'accesso: `ReadData`, `WriteData`, `AppendData`, `Execute`, `ReadExtendedAttribute`, `WriteExtendedAttribute`, `ReadAttributes` e `WriteAttributes`. È chiaro che ciò permette un preciso grado di controllo delle modalità d'accesso agli oggetti.

Il sistema Windows 7 classifica gli oggetti come contenitori e non contenitori. Gli **oggetti contenitori**, per esempio le directory, possono contenere, in senso logico, altri oggetti. Di norma, quando si crea un oggetto all'interno di un oggetto contenitore, il nuovo oggetto eredita i permessi dell'oggetto genitore; così pure, se l'utente copia un file da una directory a un'altra, il file eredita i permessi della directory di destinazione. Gli **oggetti non contenitori** non ereditano nessun altro permesso. Tuttavia, se si cambiano i permessi di una directory, i nuovi permessi non si applicano automaticamente alle directory e ai file in essa già contenuti; l'utente, se lo desidera, può richiedere esplicitamente che siano applicati i nuovi permessi.

L'amministratore del sistema può anche inibire l'uso di una delle stampanti del sistema per un intero giorno o per parte di esso, e può avvalersi del Monitor delle Prestazioni per individuare problemi emergenti. In generale, un punto di forza del sistema Windows 7 è la capacità di mettere a disposizione strumenti che aiutano a fornire un ambiente informatico sicuro. Molti di questi strumenti però di solito non sono abilitati per default, il che probabilmente costituisce una delle spiegazioni alla miriade di violazioni di sicurezza che si verificano ai danni dei sistemi Windows 7. Un'altra causa è il gran numero di servizi installati da questo sistema all'avvio, e il numero di applicazioni generalmente installate. In un reale ambiente multiutente, l'amministratore del sistema dovrebbe formulare e implementare una politica della sicurezza, sia con gli strumenti propri di Windows 7 sia tramite altri mezzi.

15.10 Sommario

La protezione è un problema interno. La sicurezza invece deve prendere in considerazione sia il sistema elaborativo sia l'ambiente – persone, edifici, affari, oggetti di valore, minacce – all'interno del quale si usa il sistema.

I dati memorizzati in un sistema elaborativo devono essere protetti da accessi non autorizzati, distruzioni o alterazioni dolose, e dall'introduzione accidentale d'incoerenze. È più semplice proteggere un sistema dalla perdita accidentale di coerenza dei dati che da accessi dolosi ai dati. Un'assoluta protezione dalle azioni dolose sulle informazioni memorizzate in un sistema elaborativo non è possibile, ma il costo di tali azioni può essere reso sufficientemente alto da scoraggiare quasi tutti, se non tutti, i tentativi d'accesso non autorizzati alle informazioni contenute nel sistema.

Vari tipi di attacco possono essere sferrati contro specifici programmi e calcolatori, o anche agire in maniera indiscriminata. Le tecniche di alterazione dello stack e di buffer overflow procurano agli intrusi maggiori privilegi di accesso al sistema. I virus e i worm sono in grado di autopropagarsi e riescono talvolta a infettare migliaia di elaboratori. Gli attacchi basati sul rifiuto del servizio impediscono il corretto utilizzo dei sistemi colpiti.

La cifratura delimita l'insieme di coloro i quali ricevono informazioni, mentre l'autenticazione circoscrive il dominio di chi le trasmette. La cifratura è il metodo utilizzato per garantire la riservatezza delle informazioni, all'atto di memorizzarle o trasferirle. La cifratura simmetrica richiede una chiave condivisa, mentre la cifratura asimmetrica è effettuata con una chiave pubblica e una chiave privata. L'uso combinato dell'autenticazione e delle funzioni hash permette di verificare che i dati non abbiano subito modifiche.

I metodi di autenticazione degli utenti sono volti a identificare gli utenti legittimi di un sistema. Ai tradizionali metodi di protezione basati su nome-utente e password, se ne possono affiancare altri. Le password monouso, per esempio, cambiano da sessione a sessione per evitare attacchi replay. L'autenticazione a due fattori richiede due elementi di autenticazione, per esempio un dispositivo hardware insieme a un PIN di

attivazione. L'autenticazione multifattoriale impiega tre o più elementi. I metodi citati riducono fortemente le possibilità di falsificare l'autenticazione.

I metodi per prevenire o rilevare le violazioni alla sicurezza comprendono i sistemi di rilevamento delle intrusioni, i programmi antivirus, l'auditing e il log delle attività, il monitoraggio delle modifiche ai programmi del sistema, il monitoraggio delle chiamate di sistema e i firewall.

Esercizi di ripasso

- 15.1** Gli attacchi per buffer overflow sono evitabili adottando una migliore metodologia di programmazione o con l'ausilio di hardware particolare. Considerate queste soluzioni.
- 15.2** Una password può divenire nota ad altri utenti in diversi modi. Dite se esiste un metodo semplice per individuare un evento di questo tipo e motivate la risposta.
- 15.3** A quale scopo si usa un valore *salt* insieme alla password fornita dall'utente? Dove andrebbe memorizzato questo valore, e come andrebbe usato?
- 15.4** La lista di tutte le password è conservata all'interno del sistema operativo. Quindi, se un utente riesce a leggerla, la protezione delle password non è più garantita. Proponete uno schema per impedire il verificarsi del problema. (Suggerimento: si usino diverse rappresentazioni interne ed esterne.)
- 15.5** Un'estensione sperimentale del sistema operativo UNIX permette a un utente di associare un programma “guardiano” (*watchdog*) a un file in modo che tale programma sia attivato ogni volta che un programma richieda l'accesso al file. Il guardiano allora permette o nega l'accesso al file. Analizzate due pro e due contro di questo metodo ai fini della sicurezza.
- 15.6** Il programma COPS del sistema UNIX scandisce un dato sistema alla ricerca di eventuali buchi nella sicurezza e avverte l'utente della presenza di possibili problemi. Elencate due rischi potenziali nell'uso di tale sistema di sicurezza. Dite in che modo tali problemi si possono limitare o eliminare.
- 15.7** Discutete un modo in cui gli amministratori di sistemi connessi a Internet potrebbero progettare il proprio sistema in modo da limitare o eliminare il danno causato dai worm. Quali sono gli inconvenienti dell'effettuazione dei cambiamenti che proponete?
- 15.8** Esprimetevi pro o contro la condanna contro Robert Morris Jr. per aver prodotto ed eseguito l'*Internet worm*, trattato nel Paragrafo 15.3.1.
- 15.9** Elencate sei elementi riguardanti la sicurezza di un sistema elaborativo per una banca. Per ogni elemento nell'elenco specificate se riguarda la sicurezza fisica, umana, o del sistema operativo.

- 15.10** Esponete due vantaggi offerti dalla cifratura dei dati memorizzati in un sistema elaborativo.
- 15.11** Quali programmi di uso comune sono vulnerabili al pericolo di attacchi di interposizione? Esaminate le soluzioni adatte a evitare questa forma di attacco.
- 15.12** Ponete a confronto i modelli di cifratura simmetrica e asimmetrica, considerando in quali circostanze sia opportuno utilizzare l'uno o l'altro per un sistema distribuito.
- 15.13** Per quale ragione $D_{kd,N}(E_{ke,N}(m))$ non garantisce l'autenticazione del mittente? A quali scopi può servire un'operazione di cifratura simile?
- 15.14** Spiegate come si può utilizzare l'algoritmo di cifratura asimmetrica per raggiungere i seguenti obiettivi.
- Autenticazione: il destinatario sa che può essere stato solo il mittente a generare il messaggio.
 - Segretezza: soltanto il destinatario può decrittografare il messaggio.
 - Autenticazione e segretezza: soltanto il destinatario può decrittografare il messaggio, e il destinatario sa che solo il mittente può aver generato il messaggio.
- 15.15** Si consideri un sistema che generi 10 milioni di record di monitoraggio al giorno. Si supponga, inoltre, che questo sistema riceva in media 10 attacchi al giorno e che ciascuno di questi attacchi dia luogo a 20 record di monitoraggio. Se il sistema di rilevamento delle intrusioni prevede una frequenza di veri allarmi pari a 0,6, e una frequenza di falsi allarmi pari a 0,0005, qual è la percentuale di allarmi nel sistema che corrisponde alle intrusioni reali?

Note bibliografiche

Una trattazione generale della sicurezza si trova in [Denning 1982], [Pfleeger e Pfleeger 2006] e [Tanenbaum 2010]. Delle reti di calcolatori discutono [Kurose e Ross 2013].

Argomenti relativi alla progettazione e alla verifica dei sistemi di sicurezza sono trattati in [Rushby 1981] e [Silverman 1983]. Un kernel di sicurezza per un microcalcolatore con più unità d'elaborazione è descritto in [Schell 1983]. Un sistema distribuito sicuro è descritto in [Rushby e Randell 1983].

La sicurezza delle password è trattata in [Morris e Thompson 1979]. I metodi per combattere i ladri di password sono illustrati in [Morshedian 1986]. L'autenticazione di password con comunicazione non sicura è esaminata in [Lamport 1981]. La questione della violazione delle password è trattata in [Seely 1989]. Il problema delle intrusioni nei calcolatori è trattato in [Lehmann 1987] e [Reid 1987]. Problemi connessi all'affidabilità dei programmi del calcolatore sono presenti in [Thompson 1984].

Analisi della sicurezza del sistema operativo UNIX si trovano in [Grampp e Morris 1984], [Wood e Kochan 1985], [Farrow 1986], [Filipski e Hanko 1986], [Hecht et al. 1988], [Kramer 1988], [Garfinkel et al. 2003]. [Bershad e Pinkerton 1988] presentano l'estensione dei programmi guardiani di BSD UNIX.

[Spafford 1989] presenta una dettagliata analisi tecnica dell'*Internet worm*; l'articolo di Spafford è apparso, insieme con altri tre articoli sull'argomento, in una sezione speciale dedicata all'*Internet worm* di *Communications of the ACM*, Volume 32, Numero 6, giugno 1989.

I problemi di sicurezza legati alla suite di protocolli TCP/IP sono esaminati da [Bel-lovin 1989]; si può consultare [Cheswick et al. 2003] per le tecniche di difesa più comuni dai relativi attacchi. Un'altra soluzione contro gli attacchi dall'interno che minacciano le reti consiste nel rendere sicura la topologia o la configurazione dei nodi di instradamento. [Kent et al. 2000], [Hu et al. 2002], [Zapata e Asokan 2002], nonché [Hu e Perrig 2004] presentano soluzioni per l'instradamento sicuro. [Savage et al. 2000] si occupano dell'attacco distribuito basato sul rifiuto del servizio, proponendo, in merito a questo problema, soluzioni che consentono di risalire agli indirizzi IP. [Perlman 1988] suggerisce un metodo per identificare i guasti quando su una rete vi siano router sospetti.

Informazioni circa i virus e i worm sono reperibili all'indirizzo <http://www.securelist.com>, così come in [Ludwig 1998] e [Ludwig 2002].

Un altro sito contenente informazioni aggiornate sulla sicurezza è <http://www.eeye.com/resources/security-center/research>.

All'indirizzo <http://cryptome.org/cyberinsecurity.htm> è disponibile un articolo sui pericoli della monocultura.

[Diffie e Hellman 1976] e [Diffie e Hellman 1979] sono stati i primi ricercatori a proporre l'uso dello schema di cifratura a chiave pubblica. L'algoritmo presentato nel Paragrafo 15.4.1, basato su tale schema, è stato sviluppato da [Rivest et al. 1978]. [C. Kaufmann 2002] e [Stallings 2011] indagano l'uso della crittografia nei sistemi elaborativi. Analisi della protezione delle firme digitali sono presentate in [Akl 1983], [Davies 1983], [Denning 1983] e [Denning 1984]. Informazioni dettagliate sulla crittografia sono disponibili in [Schneier 1996] e [Katz e Lindell 2008].

L'algoritmo RSA è presentato in [Rivest et al. 1978]. Nel sito <http://www.nist.gov/aes/> si trovano informazioni sulle attività del NIS riguardanti l'AES; nello stesso sito si trovano anche informazioni su altri standard di cifratura per gli USA. Nel 1999 l'SSL 3.0 è stato leggermente modificato e presentato col nome TLS in una RFC dell'IETF.

L'esempio del Paragrafo 15.6.3, che illustra l'impatto del tasso di falsi allarmi sull'efficacia degli IDS è basato su [Axelsson 1999]. La descrizione del Tripwire del Paragrafo 15.6.5 si basa su [Kim e Spafford 1993]. La tematica del rilevamento delle anomalie basate sulle chiamate di sistema è approfondita da [Forrest et al. 1996].

Il governo federale degli USA è ovviamente interessato alla sicurezza, e ha pubblicato il [*Department of Defense Trusted Computer System Evaluation Criteria* 1985], noto anche come *Orange Book*, in cui sono descritti una serie di livelli di si-

curezza e le caratteristiche che un sistema operativo deve avere per essere qualificato a ciascun grado di sicurezza. La sua lettura è un buon punto di partenza per la comprensione degli aspetti riguardanti la sicurezza dei sistemi elaborativi. Il testo *Microsoft Windows NT Workstation Resource Kit* Microsoft [1996] descrive le caratteristiche e l'uso del modello di sicurezza del sistema operativo Windows NT.

Bibliografia

- [**Akl 1983**] S. G. Akl, “Digital Signatures: A Tutorial Survey”, Computer, Vol. 16, N. 2, p. 15–24, 1983.
- [**Axelsson 1999**] S. Axelsson, “The Base-Rate Fallacy and Its Implications for Intrusion Detection”, Proceedings of the ACM Conference on Computer and Communications Security, p. 1–7, 1999.
- [**Bellovin 1989**] S. M. Bellovin, “Security Problems in the TCP/IP Protocol Suite”, Computer Communications Review, Vol. 19:2, p. 32–48, 1989.
- [**Bershad e Pinkerton 1988**] B. N. Bershad e C. B. Pinkerton, “Watchdogs: Extending the Unix File System”, Proceedings of the Winter USENIX Conference, 1988.
- [**C. Kaufman 2002**] M. S. C. Kaufman, R. Perlman, *Network Security: Private Communication in a Public World*, 2° Ed., Prentice Hall, 2002.
- [**Cheswick et al. 2003**] W. Cheswick, S. Bellovin e A. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*, 2° Ed., Addison-Wesley, 2003.
- [**Davies 1983**] D.W. Davies, “Applying the RSA Digital Signature to Electronic Mail”, Computer, Vol. 16, N. 2, p. 55–62, 1983.
- [**Denning 1982**] D. E. Denning, *Cryptography and Data Security*, Addison-Wesley, 1982.
- [**Denning 1983**] D. E. Denning, “Protecting Public Keys and Signature Keys”, Computer, Vol. 16, N. 2, p. 27–35, 1983.
- [**Denning 1984**] D. E. Denning, “Digital Signatures with RSA and Other Public-Key Cryptosystems”, Communications of the ACM, Vol. 27, N. 4, p. 388–392, 1984.
- [**Diffie e Hellman 1976**] W. Diffie e M. E. Hellman, “New Directions in Cryptography”, IEEE Transactions on Information Theory, Vol. 22, N. 6, p. 644–654, 1976.
- [**Diffie e Hellman 1979**] W. Diffie e M. E. Hellman, “Privacy and Authentication”, Proceedings of the IEEE, p. 397–427, 1979.
- [**DoD 1985**] *Trusted Computer System Evaluation Criteria*. Department of Defense, 1985.
- [**Farrow 1986**] R. Farrow, “Security Issues and Strategies for Users”, UNIX World April, p. 65–71, 1986.
- [**Filipski and Hanko 1986**] A. Filipski and J. Hanko, “Making UNIX Secure”, Byte April, p. 113–128, 1986.
- [**Forrest et al. 1996**] S. Forrest, S. A. Hofmeyr e T. A. Longstaff, “A Sense of Self for UNIX Processes”, Proceedings of the IEEE Symposium on Security and Privacy, p. 120–128, 1996.

- [Garfinkel et al. 2003]** S. Garfinkel, G. Spafford e A. Schwartz, *Practical UNIX & Internet Security*, O'Reilly & Associates, 2003.
- [Grampp e Morris 1984]** F. T. Grampp e R. H. Morris, “UNIX Operating-System Security”, AT&T Bell Laboratories Technical Journal, Vol. 63, N. 8, p. 1649–1672, 1984.
- [Hecht et al. 1988]** M. S. Hecht, A. Johri, R. Aditham e T. J. Wei, “Experience Adding C2 Security Features to UNIX”, Proceedings of the Summer USENIX Conference, p. 133–146, 1988.
- [Hu e Perrig 2004]** Y.C. Hu e A. Perrig, “SPV: A Secure Path Vector Routing Scheme for Securing BGP”, Proceedings of ACM SIGCOMM Conference on Data Communication, 2004.
- [Hu et al. 2002]** Y.-C. Hu, A. Perrig e D. Johnson, “Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks”, Proceedings of the Annual International Conference on Mobile Computing and Networking, 2002.
- [Katz e Lindell 2008]** J. Katz e Y. Lindell, *Introduction to Modern Cryptography*, Chapman & Hall/CRC Press, 2008.
- [Kent et al. 2000]** S. Kent, C. Lynn e K. Seo, “Secure Border Gateway Protocol (Secure-BGP)”, IEEE Journal on Selected Areas in Communications, Vol. 18, N. 4, p. 582–592, 2000.
- [Kim e Spafford 1993]** G. H. Kim e E. H. Spafford, “The Design and Implementation of Tripwire: A File System Integrity Checker”, Technical report, Purdue University, 1993.
- [Kramer 1988]** S. M. Kramer, “Retaining SUID Programs in a Secure UNIX”, Proceedings of the Summer USENIX Conference, p. 107–118, 1988.
- [Kurose e Ross 2013]** J. Kurose e K. Ross, *Computer Networking—A Top-Down Approach*, 6° Ed., Addison-Wesley, 2013.
- [Lamport 1981]** L. Lamport, “Password Authentication with Insecure Communications”, Communications of the ACM, Vol. 24, N. 11, p. 770–772, 1981.
- [Lehmann 1987]** F. Lehmann, “Computer Break-Ins”, Communications of the ACM, Vol. 30, N. 7, p. 584–585, 1987.
- [Ludwig 1998]** M. Ludwig, *The Giant Black Book of Computer Viruses*, 2° Ed., American Eagle Publications, 1998.
- [Ludwig 2002]** M. Ludwig, *The Little Black Book of Email Viruses*, American Eagle Publications, 2002.
- [Microsoft 1996]** Microsoft Windows NT Workstation Resource Kit. Microsoft Press, 1996.
- [Morris e Thompson 1979]** R. Morris e K. Thompson, “Password Security: A Case History”, Communications of the ACM, Vol. 22, N. 11, p. 594–597, 1979.
- [Morshedian 1986]** D. Morshedian, “How to Fight Password Pirates”, Computer, Vol. 19, N. 1, 1986.
- [Perlman 1988]** R. Perlman, *Network Layer Protocols with Byzantine Robustness*. PhD thesis, Massachusetts Institute of Technology, 1988.
- [Pfleeger e Pfleeger 2006]** C. Pfleeger e S. Pfleeger, *Security in Computing*, 4° Ed. Prentice Hall, 2006.

- [Reid 1987]** B. Reid, “Reflections on Some Recent Widespread Computer Break-Ins”, Communications of the ACM, Vol. 30, N. 2, p. 103–105, 1987.
- [Rivest et al. 1978]** R. L. Rivest, A. Shamir e L. Adleman, “On Digital Signatures and Public Key Cryptosystems”, Communications of the ACM, Vol. 21, N. 2, p. 120–126, 1978.
- [Rushby 1981]** J. M. Rushby, “Design and Verification of Secure Systems”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 12–21, 1981.
- [Rushby e Randell 1983]** J. Rushby e B. Randell, “A Distributed Secure System”, Computer, Vol. 16, N. 7, p. 55–67, 1983.
- [Savage et al. 2000]** S. Savage, D. Wetherall, A. R. Karlin e T. Anderson, “Practical Network Support for IP Traceback”, Proceedings of ACM SIGCOMM Conference on Data Communication, p. 295–306, 2000.
- [Schell 1983]** R. R. Schell, “A Security Kernel for a Multiprocessor Microcomputer”, Computer, p. 47–53, 1983.
- [Schneier 1996]** B. Schneier, *Applied Cryptography*, 2° Ed., John Wiley and Sons, 1996.
- [Seely 1989]** D. Seely, “Password Cracking: A Game of Wits”, Communications of the ACM, Vol. 32, N. 6, p. 700–704, 1989.
- [Silverman 1983]** J. M. Silverman, “Reflections on the Verification of the Security of an Operating System Kernel”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 143–154, 1983.
- [Spafford 1989]** E. H. Spafford, “The Internet Worm: Crisis and Aftermath”, Communications of the ACM, Vol. 32, N. 6, p. 678–687, 1989.
- [Stallings 2011]** W. Stallings, *Operating Systems*, 7° Ed., Prentice Hall, 2011.
- [Tanenbaum 2010]** A. S. Tanenbaum, *Computer Networks*, 5° Ed., Prentice Hall, 2010.
- [Thompson 1984]** K. Thompson, “Reflections on Trusting Trust”, Communications of ACM, Vol. 27, N. 8, p. 761–763, 1984.
- [Wood e Kochan 1985]** P. Wood e S. Kochan, *UNIX System Security*, Hayden, 1985.
- [Zapata e Asokan 2002]** M. Zapata e N. Asokan, “Securing Ad Hoc Routing Protocols”, Proc. 2002 ACM Workshop on Wireless Security, p. 1–10, 2002.

Argomenti avanzati

La virtualizzazione permea tutti gli aspetti della computazione. Le macchine virtuali sono un esempio di questa tendenza. In genere per mezzo di una macchina virtuale i sistemi operativi e le applicazioni ospitati vengono eseguiti in un ambiente che appare loro come l'hardware nativo. Questo ambiente si comporta verso di loro come farebbe un hardware nativo, ma inoltre li protegge, li gestisce e li limita.

Un sistema distribuito è un insieme di processori che non condividono la memoria o il clock. Ogni processore ha la propria memoria locale e i processori comunicano tra loro attraverso linee di comunicazione come LAN o WAN. I sistemi distribuiti offrono diversi vantaggi: danno agli utenti l'accesso a più risorse rispetto a quelle gestite dal sistema, aumentano la velocità di calcolo e migliorano la disponibilità e l'affidabilità dei dati.

CAPITOLO

16

OBIETTIVI DEL CAPITOLO

- Presentazione della storia e dei vantaggi delle macchine virtuali.
- Analisi delle varie tecnologie di macchine virtuali.
- Descrizione dei metodi utilizzati per implementare la virtualizzazione.
- Introduzione delle caratteristiche hardware più comuni che supportano la virtualizzazione; spiegazione di come queste caratteristiche vengono utilizzate dai moduli del sistema operativo.

Macchine virtuali

Il termine **virtualizzazione** ha molti significati e gli aspetti della virtualizzazione permeano il mondo dei calcolatori. Le macchine virtuali sono un esempio di questa tendenza. In genere per mezzo di una macchina virtuale i sistemi operativi e le applicazioni ospiti vengono eseguiti in un ambiente che appare loro come l'hardware nativo, ma che inoltre li protegge, li gestisce e li limita.

Questo capitolo approfondisce gli usi, le funzioni e l'implementazione delle macchine virtuali. Le macchine virtuali possono essere implementate in vari modi e questo capitolo descrive queste varianti. Una possibilità è quella di aggiungere il supporto per la macchina virtuale al kernel. Dato che questo metodo di implementazione è il più pertinente a questo libro lo analizzeremo in maniera completa. Inoltre, dato che le caratteristiche hardware fornite dalla CPU e dai dispositivi di I/O possono supportare l'implementazione della macchina virtuale, vedremo come queste caratteristiche vengono utilizzate dai moduli del kernel appropriati.

INDIREZIONE

“In informatica qualsiasi problema può essere risolto da un altro livello di indirezione” – David Wheeler “... a parte quello di troppi livelli di indirezione.” – Kevin Henney.

16.1 Introduzione

L’idea fondamentale che sta dietro a una macchina virtuale è quella di astrarre l’hardware di un singolo computer (CPU, memoria, dischi, schede di rete e così via) in diversi ambienti di esecuzione differenti, creando così l’illusione che ogni ambiente distinto sia in esecuzione su un proprio computer privato. Questo concetto può sembrare simile al metodo stratificato di implementazione del sistema operativo (si veda il Paragrafo 2.7.2) e per certi versi lo è. Nel caso della virtualizzazione, vi è uno strato che crea un sistema virtuale in cui è possibile eseguire sistemi operativi o applicazioni.

L’implementazione della macchina virtuale coinvolge diverse componenti. Alla base vi è l’host, il sistema hardware sottostante che gestisce le macchine virtuali. Il **gestore della macchina virtuale** (*virtual machine manager*, VMM), noto anche come **hypervisor**, crea e gestisce le macchine virtuali fornendo un’interfaccia che è identica all’host (eccetto nel caso della paravirtualizzazione, discussa più avanti). A ogni processo guest (*ospite*) viene fornita una copia virtuale dell’host (Figura 16.1). Di solito il processo guest è un sistema operativo. Una singola macchina fisica può quindi eseguire più sistemi operativi simultaneamente, ciascuno nella propria macchina virtuale.

Vale la pena di prendersi un momento per riflettere su come, con la virtualizzazione, la definizione di “sistema operativo” risulta ulteriormente sfumata. Considerate, per esempio, un software VMM come VMware ESX. Questo software di virtualizzazione è installato sull’hardware, viene eseguito all’avvio e fornisce servizi alle applicazioni.

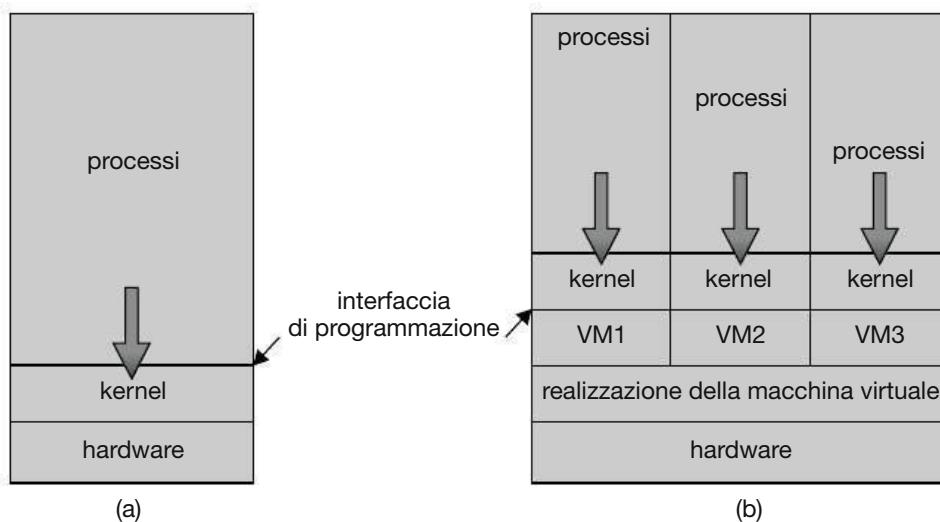


Figura 16.1 Modelli di sistema: (a) semplice; (b) macchina virtuale.

I servizi includono quelli tradizionali, come la pianificazione e la gestione della memoria, insieme a nuove tipologie di servizi, come la migrazione delle applicazioni tra sistemi. Inoltre, le applicazioni sono di fatto sistemi operativi guest. Possiamo considerare il VMM VMware ESX come un sistema operativo che, a sua volta, esegue altri sistemi operativi? Certamente, perché si comporta come un sistema operativo. Per chiarezza, però, noi chiamiamo VMM il componente che offre ambienti virtuali.

L'implementazione di un VMM è molto varia. Tra le opzioni disponibili vi sono le seguenti.

- Soluzioni basate su hardware che forniscono il supporto per la creazione e la gestione della macchina virtuale tramite firmware. Questi VMM, che si trovano comunemente nei mainframe e nei server di medie/grandi dimensioni, sono generalmente conosciuti come **hypervisor di tipo 0**. IBM LPAR e Oracle LDoms ne sono esempi.
- Software assimilabili a sistemi operativi costruiti per fornire la virtualizzazione. Tra questi vi sono VMware ESX (citato in precedenza), Joyent SmartOS e Citrix XenServer. Questi VMM sono noti come **hypervisor di tipo 1**.
- Sistemi operativi general-purpose che forniscono funzioni standard e funzioni di VMM, tra cui Microsoft Windows Server con HyperV e Linux RedHat con la funzionalità KVM. Poiché tali sistemi hanno un insieme di funzionalità simile a quello degli hypervisor di tipo 1, anch'essi sono classificati come di tipo 1.
- Applicazioni che si eseguono su sistemi operativi standard, ma forniscono funzionalità di VMM per sistemi operativi guest. Queste applicazioni, che includono VMware Workstation e Fusion, Parallels Desktop e Oracle Virtual-Box, sono **hypervisor di tipo 2**.
- **Paravirtualizzazione**, una tecnica in cui il sistema operativo guest viene modificato per lavorare in collaborazione con il VMM al fine di ottimizzare le prestazioni.
- **Virtualizzazione dell'ambiente di programmazione**, in cui i VMM non virtualizzano l'hardware vero e proprio, ma creano piuttosto un sistema virtuale ottimizzato. Questa tecnica è utilizzata da Oracle Java e Microsoft.Net.
- **Emulatori** che consentono ad applicazioni scritte per un certo ambiente hardware di funzionare su un ambiente hardware molto diverso, per esempio su un diverso tipo di CPU.
- **Contenitori di applicazioni**, che non forniscono una completa virtualizzazione, ma offrono alcune funzionalità di virtualizzazione separando le applicazioni dal sistema operativo. Oracle Solaris Zones, BSD Jails e IBM AIX WPARs “contengono” le applicazioni rendendole più sicure e gestibili.

La varietà delle tecniche di virtualizzazione in uso oggi è una testimonianza dell'ampiezza, della profondità e dell'importanza della virtualizzazione nell'informatica moderna. La virtualizzazione è di fondamentale importanza per i data-center, per lo sviluppo di applicazioni efficienti e per il test del software, solo per fare alcuni esempi.

16.2 Storia

Le macchine virtuali hanno fatto la loro comparsa sul mercato nel 1972, con i mainframe IBM. La virtualizzazione era fornita dal sistema operativo IBM VM, che con il tempo si è evoluto ed è disponibile ancora oggi. Molti dei concetti originali di IBM VM si trovano in altri sistemi, ragion per cui vale davvero la pena di analizzare questo sistema operativo.

IBM VM370 divideva un mainframe in più macchine virtuali, ciascuna con il proprio sistema operativo. Una delle maggiori difficoltà riguardava i sistemi di dischi. Supponiamo che la macchina fisica avesse tre dischi, ma si volessero supportare sette macchine virtuali. Chiaramente non era possibile allocare un disco a ogni macchina virtuale: la soluzione fu quella di fornire dischi virtuali, chiamati **minidisk** nel sistema operativo VM di IBM. I minidisk sono identici ai dischi rigidi del sistema in tutti gli aspetti eccetto la dimensione. Il sistema implementava ogni minidisk allocando le tracce sui dischi fisici di cui il minidisk aveva bisogno.

Una volta che le macchine virtuali erano state create, gli utenti potevano eseguire uno dei sistemi operativi o pacchetti software che erano disponibili sulla macchina sottostante. Nel caso del sistema IBM VM, un utente eseguiva di norma CMS – un sistema operativo interattivo monoutente.

Per molti anni dopo che IBM ha introdotto questa tecnologia la virtualizzazione è rimasta confinata in quell'ambito; la maggior parte dei sistemi non la supportava. Tuttavia, una definizione formale di virtualizzazione ha contribuito a stabilire i requisiti di sistema e gli obiettivi per le funzionalità. I requisiti di virtualizzazione erano:

1. un VMM fornisce un ambiente per i programmi essenzialmente identico alla macchina originale;
2. i programmi in esecuzione all'interno di tale ambiente soffrono di una perdita di prestazioni modesta;
3. il VMM è in completo controllo delle risorse di sistema.

Questi requisiti di fedeltà, prestazioni e sicurezza guidano ancora oggi lo sviluppo della virtualizzazione.

Dalla fine degli anni '90 le CPU Intel 80x86 diventarono piuttosto diffuse, veloci e ricche di funzionalità. Come conseguenza gli sviluppatori iniziarono a dedicare i loro sforzi all'implementazione della virtualizzazione su tale piattaforma. Sia **Xen** che **VMware** hanno creato tecnologie, ancora in uso oggi, per permettere di eseguire sistemi operativi guest su 80x86. Da quel momento la virtualizzazione è cresciuta in modo da coinvolgere tutte le CPU più diffuse, molti strumenti commerciali e open-source e molti sistemi operativi. Per esempio, il progetto open-source VirtualBox (<http://www.virtualbox.org>), fornisce un programma che può essere eseguito su CPU Intel x86 e AMD64 e su sistemi operativi host Windows, Linux, Mac OS X e Solaris. Tra i possibili sistemi operativi guest vi sono diverse versioni di Windows, Linux, Solaris e BSD, oltre a MS-DOS e IBM OS/2.

16.3 Vantaggi e caratteristiche

Diversi vantaggi rendono la virtualizzazione attraente. La maggior parte di questi è essenzialmente legata alla capacità di condividere lo stesso hardware, ma eseguire diversi ambienti differenti (cioè sistemi operativi diversi) contemporaneamente.

Un importante vantaggio della virtualizzazione è che il sistema host viene protetto dalle macchine virtuali, proprio come le macchine virtuali sono protette l'una dall'altra. Un virus all'interno di un sistema operativo guest potrebbe danneggiare il sistema operativo, ma è improbabile che possa influenzare l'host o gli altri guest. Poiché ogni macchina virtuale è pressoché completamente isolata da tutte le altre, non esistono quasi problemi di protezione.

Un potenziale svantaggio di questo isolamento è che esso può impedire la condivisione delle risorse. Sono stati seguiti due diversi approcci per fornire la condivisione. In primo luogo, è possibile condividere un volume del file system e quindi condividere i file. In secondo luogo, è possibile definire una rete di macchine virtuali, ciascuna delle quali può inviare informazioni attraverso la rete di comunicazione virtuale. La rete modella reti fisiche di comunicazione, ma è implementata via software. Naturalmente il VMM è libero di consentire a qualunque ospite di utilizzare le risorse fisiche, come una connessione di rete fisica (con condivisione fornita dal VMM), nel qual caso gli ospiti accettati possono comunicare tra loro attraverso la rete fisica.

Una caratteristica comune alla maggior parte delle implementazioni della virtualizzazione è la capacità di congelare, o *sospendere*, una macchina virtuale in esecuzione. Molti sistemi operativi forniscono una tale caratteristica, fondamentale per i processi, ma i VMM fanno un passo in più e permettono di effettuare copie e **istantanee** del guest. La copia può essere usata per creare una nuova VM o per spostare una VM da una macchina all'altra mantenendo inalterato il suo stato. Il guest può quindi *riprendere* dal punto in cui si trovava, come se fosse ancora sulla sua macchina originale, formando così un **clone**. L'istantanea memorizza la situazione in un dato istante di tempo in modo che il guest, se necessario, possa essere riportato a quelllo stato (per esempio quando è stato fatto un cambiamento, ma questo non è più desiderato). Spesso un VMM consente di scattare diverse istantanee. Per esempio, è possibile registrare lo stato di un guest con un'istantanea ogni giorno per un mese, rendendo possibile il ripristino di uno qualsiasi degli stati registrati. Queste funzionalità sono utilizzate con grande vantaggio in ambienti virtuali.

Un sistema virtuale è uno strumento perfetto per ricerca e sviluppo su sistemi operativi. Normalmente la modifica di un sistema operativo è un'operazione complicata, perché i sistemi operativi sono programmi grandi e complessi e un cambiamento in una loro parte può causare la comparsa di misteriosi errori in altre parti. La potenza del sistema operativo rende le modifiche particolarmente pericolose. Poiché il sistema operativo viene eseguito in modalità kernel, una modifica errata a un puntatore potrebbe causare un errore in grado di distruggere l'intero file system. È pertanto necessario testare con attenzione tutte le modifiche al sistema operativo.

Inoltre il sistema operativo viene eseguito su una macchina di cui ha il completo controllo, che deve dunque essere arrestata e restare inutilizzata mentre le modifiche

vengono apportate e testate. Questo intervallo di tempo viene comunemente chiamato **periodo di sviluppo del sistema** e sui sistemi condivisi, dal momento che rende il sistema indisponibile agli utenti, è spesso pianificato a tarda notte o nei fine settimana, quando il carico del sistema è basso.

Un sistema virtuale può eliminare in gran parte quest'ultimo problema. I programmatore di sistema sono dotati di una macchina virtuale privata e lo sviluppo avviene sulla macchina virtuale invece che su una macchina fisica. Il normale funzionamento del sistema viene interrotto solo quando una modifica è completata, testata e pronta per essere messa in produzione.

Un altro vantaggio offerto agli sviluppatori dalle macchine virtuali è la possibilità di eseguire più sistemi operativi contemporaneamente sulla propria workstation. Questa workstation virtualizzata consente di trasferire e testare rapidamente i programmi in ambienti diversi. È inoltre possibile eseguire più versioni di uno stesso programma, ciascuna nel suo sistema operativo isolato, su un unico sistema. Allo stesso modo, gli ingegneri del controllo qualità possono testare le applicazioni in diversi ambienti senza il bisogno di acquistare, alimentare e mantenere un computer per ogni ambiente.

Un vantaggio importante nell'utilizzo di macchine virtuali nei data-center di produzione è il consolidamento del sistema, che consiste nel prendere due o più sistemi fisici distinti ed eseguirli, all'interno di macchine virtuali, su un unico sistema. Le conversioni da fisico a virtuale permettono un'ottimizzazione delle risorse, poiché molti sistemi scarsamente utilizzati possono essere combinati per creare un unico sistema con una utilizzazione maggiore.

Va considerato, inoltre, che gli strumenti di gestione che fanno parte di un VMM permettono agli amministratori di sistema di gestire molti più sistemi di quanto potrebbero fare altrimenti. Un ambiente virtuale potrebbe includere 100 server fisici con 20 server virtuali in esecuzione su ogni server fisico. Senza virtualizzazione, la presenza di 2.000 server richiederebbe diversi amministratori di sistema. Con la virtualizzazione e i suoi strumenti, lo stesso lavoro può essere gestito da uno o due amministratori. Uno degli strumenti utilizzati in questi casi è il **templating**, in cui un'immagine standard della macchina virtuale, che include un sistema operativo ospite installato e configurato e le sue applicazioni, viene salvata e utilizzata come sorgente per più macchine virtuali in esecuzione. Tra le altre caratteristiche menzioniamo la gestione degli aggiornamenti di tutti i guest, il loro backup, il loro ripristino e il controllo del loro utilizzo delle risorse.

La virtualizzazione, oltre a migliorare l'utilizzo delle risorse, ne migliora anche la gestione. Alcuni VMM includono una funzionalità di migrazione in tempo reale (**live migration**) che permette di spostare un guest in esecuzione da un server fisico a un altro senza interrompere il suo funzionamento o le connessioni di rete attive. Se un server è sovraccarico, la migrazione in tempo reale può liberare risorse sull'host senza interrompere il guest. Analogamente, quando l'hardware di un host deve essere riparato o aggiornato, i guest possono essere spostati su altri server, per poi effettuare la manutenzione dell'host e infine riportare i guest sull'host originale. Questa operazione avviene senza tempi morti e senza interruzioni per gli utenti.

Pensiamo ora ai possibili vantaggi della virtualizzazione nella distribuzione delle applicazioni. Se un sistema può facilmente aggiungere, rimuovere o spostare una macchina virtuale, allora perché installare le applicazioni direttamente sul sistema? L'applicazione potrebbe piuttosto essere preinstallata su un sistema operativo messo a punto e personalizzato in una macchina virtuale. Questo metodo offrirebbe diversi vantaggi agli sviluppatori di applicazioni: la gestione dell'applicazione diventerebbe più semplice, sarebbe richiesto meno tuning e il supporto tecnico sarebbe più immediato. Anche gli amministratori di sistema dovrebbero trovare l'ambiente più facile da gestire: l'installazione sarebbe semplice e ridistribuire l'applicazione su altri sistemi sarebbe molto più facile rispetto alle solite fasi di disinstallazione e reinstallazione. Affinché questa metodologia venga adottata in maniera diffusa, però, il formato delle macchine virtuali deve essere standardizzato in modo che ogni macchina virtuale possa essere eseguita su qualsiasi piattaforma di virtualizzazione. *L'Open Virtual Machine Format* è un tentativo di fornire una tale standardizzazione e potrebbe riuscire a unificare i formati delle macchine virtuali.

La virtualizzazione ha gettato le basi per molti altri progressi nella realizzazione, nella gestione e nel monitoraggio di sistemi informatici. Il **cloud computing**, per esempio, è reso possibile da una virtualizzazione in cui risorse come la CPU, la memoria e l'I/O vengono offerti come servizi ai clienti, utilizzando tecnologie Internet. Utilizzando le API un programma può chiedere a una struttura cloud di creare migliaia di macchine virtuali, su ognuna delle quali è in esecuzione uno specifico sistema operativo guest e un'applicazione, a cui altri possono accedere via Internet. Molti giochi multiutente, siti di condivisione di fotografie e altri servizi web sfruttano queste caratteristiche.

Nell'ambito del mondo desktop, la virtualizzazione sta permettendo agli utenti di connettersi alle macchine virtuali situate in data-center remoti e accedere alle loro applicazioni come se fossero locali. Questa pratica può aumentare la sicurezza, perché non ci sono dati memorizzati sui dischi locali dell'utente, e il costo delle risorse di calcolo dell'utente può diminuire. L'utente deve disporre di rete, CPU e memoria, ma tutto ciò che queste componenti del sistema devono fare è visualizzare l'immagine del guest in esecuzione in remoto (attraverso un protocollo come RDP). Non c'è pertanto bisogno che queste componenti siano costose e ad alte prestazioni. Emergeranno sicuramente in futuro nuovi impieghi della virtualizzazione, grazie all'aumento della sua diffusione e al miglioramento del supporto hardware.

16.4 Blocchi costituenti

Il concetto di macchina virtuale è utile, ma è difficile da implementare. È necessario molto lavoro per fornire un duplicato esatto della macchina sottostante e ciò è vero in particolare su sistemi dual-mode, dove la macchina sottostante dispone solo di modalità utente e modalità kernel. In questo paragrafo esamineremo i componenti necessari per la costruzione di una virtualizzazione efficiente. Si noti che questi componenti non sono richiesti dall'hypervisor di tipo 0, come discusso nel Paragrafo 16.5.2.

La possibilità di virtualizzare dipende dalle caratteristiche offerte dalla CPU. Se le caratteristiche sono sufficienti, allora è possibile scrivere un VMM in grado di fornire un ambiente guest; in caso contrario la virtualizzazione è impossibile. Un VMM utilizza diverse tecniche per implementare la virtualizzazione, tra cui trap-and-emulate e la traduzione binaria. In questo paragrafo analizziamo ciascuna di queste tecniche, oltre al supporto hardware necessario per supportare la virtualizzazione.

Un importante concetto presente nella maggior parte delle varianti della virtualizzazione è l'implementazione di una **CPU virtuale** (VCPU). La VCPU non esegue il codice, ma rappresenta lo stato in cui la macchina guest ritiene che si trovi la CPU. Il VMM mantiene per ogni guest una VCPU che rappresenta lo stato corrente della CPU secondo il corrispondente guest. Quando il VMM assegna una CPU a un guest vengono utilizzate le informazioni della VCPU per caricare il giusto contesto, proprio come un sistema operativo general-purpose utilizzerebbe il PCB.

16.4.1 Trap-and-Emulate

In un tipico sistema dual-mode il guest della macchina virtuale può essere eseguito solo in modalità utente (a meno che non sia fornito un supporto hardware aggiuntivo). Il kernel, naturalmente, viene eseguito in modalità kernel e non è sicuro consentire al codice a livello utente l'esecuzione in modalità kernel. Tuttavia, la macchina virtuale deve avere due modalità, proprio come la macchina fisica ha due modalità. Occorre avere di conseguenza una modalità utente virtuale e una modalità kernel virtuale, entrambe in esecuzione in modalità fisica utente. Le azioni che causano il passaggio dalla modalità utente alla modalità kernel su una macchina reale (per esempio una chiamata di sistema, un interrupt o un tentativo di eseguire un'istruzione privilegiata) devono causare nella macchina virtuale il passaggio dalla modalità utente virtuale alla modalità kernel virtuale.

Come può essere realizzato questo cambio di modalità? Quando il kernel del guest tenta di eseguire un'istruzione privilegiata si ha un errore (perché il sistema è in modalità utente) e ciò provoca una trap al VMM nella macchina reale. A questo punto il VMM ottiene il controllo ed esegue (o “emula”) l'azione che è stata tentata dal kernel guest per conto del guest stesso. Il VMM ritorna poi il controllo alla macchina virtuale. Questo metodo è chiamato **trap-and-emulate** ed è illustrato nella Figura 16.2. La maggior parte dei prodotti di virtualizzazione utilizza questo metodo.

Quando si opera con le istruzioni privilegiate il tempo può diventare un problema. Tutte le istruzioni non privilegiate vengono eseguite nativamente sull'hardware, fornendo ai guest le stesse prestazioni delle applicazioni native. Le istruzioni privilegiate, tuttavia, creano overhead aggiuntivo e dunque il guest lavora più lentamente di quanto avverrebbe in maniera nativa. Inoltre, il fatto che la CPU venga multiprogrammata tra molte macchine virtuali può ulteriormente rallentare le macchine virtuali in modo imprevedibile.

Questo problema è stato affrontato in vari modi. IBM VM, per esempio, permette alle istruzioni normali delle macchine virtuali di essere eseguite direttamente sull'hardware. Solo le istruzioni privilegiate (necessarie soprattutto per l'I/O) devono es-

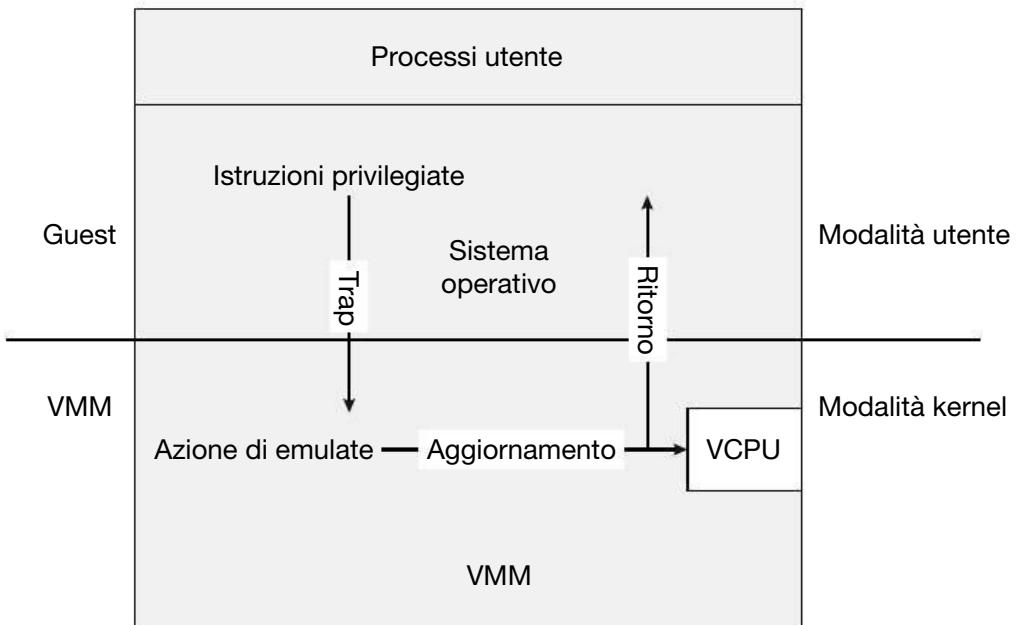


Figura 16.2 Implementazione della virtualizzazione mediante trap-and-emulate.

sere emulate e sono quindi eseguite più lentamente. In generale, con l’evoluzione dell’hardware, le prestazioni del meccanismo di trap-and-emulate sono state migliorate e sono stati ridotti i casi in cui è necessario utilizzare tale meccanismo. Molte CPU, per esempio, hanno ora modalità extra aggiuntive rispetto al loro funzionamento standard a due modalità. La VCPU non deve tenere traccia della modalità in cui si trova il sistema operativo guest, perché questa funzione è svolta dalla CPU fisica. Alcune CPU offrono ai guest la gestione dello stato della CPU via hardware, in modo che il VMM non abbia bisogno di fornire tale funzionalità ed eliminando così l’overhead.

16.4.2 Traduzione binaria

Alcune CPU non hanno una netta separazione tra istruzioni privilegiate e istruzioni non privilegiate. Sfortunatamente per gli implementatori di virtualizzazione, la linea di CPU Intel x86 fa parte di queste. Nel progetto dell’x86 non si è pensato alla virtualizzazione: in effetti la prima CPU della famiglia, l’Intel 4004, uscita nel 1971, è stata progettata per essere una calcolatrice. Il chip ha mantenuto la compatibilità con le versioni precedenti per tutta la sua esistenza, impedendo, per molte generazioni, le modifiche che avrebbero facilitato la virtualizzazione. Consideriamo un esempio del problema. Il comando `popf` carica il registro flag dal contenuto dello stack. Se la CPU è in modalità privilegiata tutti i flag vengono sostituiti, mentre se la CPU è in modalità utente ne vengono sostituiti solo alcuni e gli altri vengono ignorati. Poiché non vengono generate trap se `popf` viene eseguito in modalità utente, la procedura di trap-and-emulate diventa inutile. Altre istruzioni x86 causano problemi simili. Ai fini di questa discussione, chiameremo questo insieme di istruzioni *istruzioni speciali*. Fino al 1998 l’utilizzo di trap-and-emulate per implementare la virtualizzazione su x86 era considerato impossibile a causa di queste istruzioni speciali.

Questo problema che sembrava insormontabile è stato risolto con l'applicazione della tecnica di traduzione binaria. La **traduzione binaria** è abbastanza semplice concettualmente, ma è complessa nella realizzazione. I passi fondamentali sono i seguenti:

1. se la VCPU guest è in modalità utente, il guest può eseguire le istruzioni in modo nativo su una CPU fisica;
 2. se la VCPU guest è in modalità kernel, il guest crede di essere in esecuzione in modalità kernel. Il VMM esamina ogni istruzione eseguita dal guest in modalità kernel virtuale leggendo le prossime istruzioni che il guest sta per eseguire, basandosi sul program counter del guest. Le istruzioni diverse dalle istruzioni speciali vengono eseguite in modo nativo. Le istruzioni speciali vengono tradotte in un nuovo insieme di istruzioni che eseguono lo stesso task, per esempio la modifica dei flag nella VCPU.

La traduzione binaria, illustrata nella Figura 16.3, è implementata per mezzo di codice per la traduzione interno al VMM. Il codice legge dinamicamente dal guest le istruzioni binarie native, su richiesta, e genera codice nativo binario che viene eseguito al posto del codice originale.

Il metodo di base per la traduzione binaria appena descritto verrebbe eseguito correttamente, ma avrebbe uno scarso rendimento. Per fortuna la stragrande maggioranza delle istruzioni vengono eseguite in modo nativo. Ma come potrebbero essere migliorate le prestazioni per le altre istruzioni? Una risposta ci è data da una specifica implementazione della traduzione binaria: il metodo di VMware. La soluzione di VMware si basa sull'uso della cache. Il codice di sostituzione di ogni istruzione che

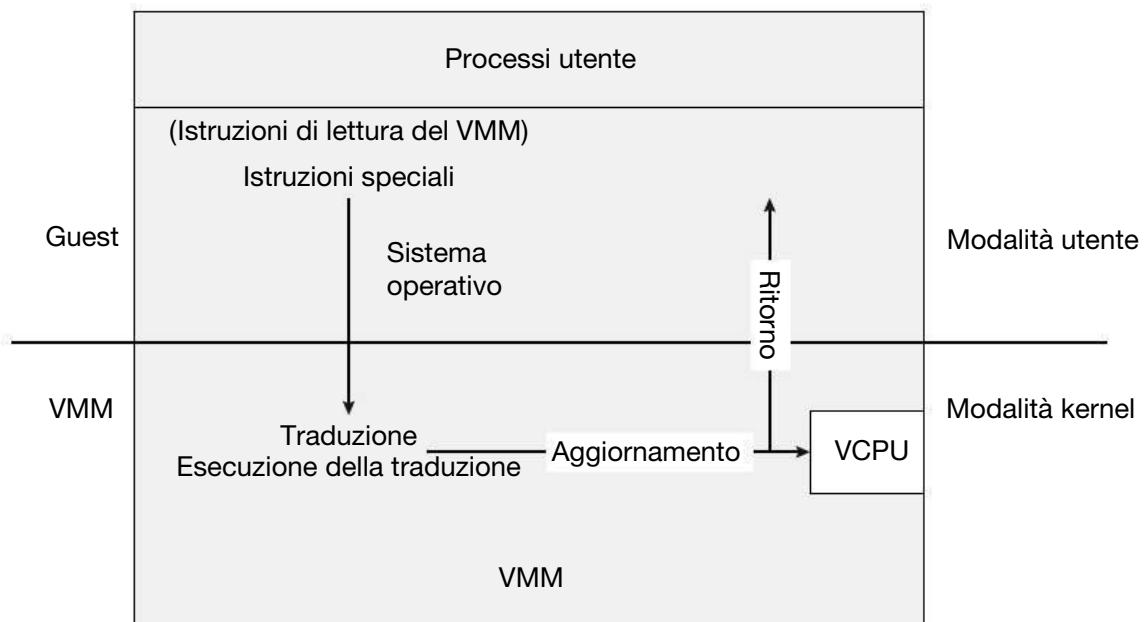


Figura 16.3 Implementazione della virtualizzazione mediante traduzione binaria.

deve essere tradotta è memorizzato in una cache. Tutte le successive esecuzioni di tali istruzioni sono eseguite dalla cache di traduzione senza bisogno di tradurle nuovamente. Se la cache è abbastanza grande, questo metodo può migliorare notevolmente le prestazioni.

Consideriamo un'altra problematica della virtualizzazione: la gestione della memoria, in particolare delle tabelle delle pagine. Come può il VMM mantenere lo stato delle tabelle delle pagine sia per i guest, che credono di gestire direttamente la tabella, che per il VMM stesso? Un metodo comune, utilizzato sia con trap-and-emulate che con la traduzione binaria, è quello di utilizzare le tabelle delle pagine nidificate (*nested page table*, NPT). Ogni sistema operativo guest mantiene una o più tabelle delle pagine da tradurre dal virtuale alla memoria fisica. Il VMM mantiene tabelle NPT per rappresentare lo stato delle tabelle delle pagine dei guest, proprio come crea una VCPU per rappresentare lo stato della CPU dei guest. Il VMM sa quando il guest tenta di modificare la sua tabella delle pagine ed effettua la variazione equivalente nella NPT. Quando l'ospite è sulla CPU, il VMM mette il puntatore alla NPT appropriata nel registro della CPU opportuno per rendere tale tabella la tabella delle pagine attiva. Se i guest ha bisogno di modificare la tabella delle pagine (per esempio in caso di errore di pagina), questa operazione deve essere intercettata dal VMM e devono essere apportate opportune modifiche alle tabelle nidificate e alle tabelle delle pagine di sistema. Sfortunatamente l'uso delle NPT può causare un aumento dei miss di TLB. Molte altre complicazioni devono essere affrontate per ottenere prestazioni ragionevoli.

Anche se a prima vista il metodo di traduzione binaria crea grandi quantità di overhead, tale metodo si comporta abbastanza bene da aver promosso la nascita di un nuovo settore tecnologico dedicato a virtualizzare sistemi basati su Intel x86. VMware ha testato l'impatto della traduzione binaria sulle prestazioni avviando uno di questi sistemi, Windows XP, e arrestandolo immediatamente, monitorando nel frattempo il tempo trascorso e il numero di traduzioni prodotte dal metodo di traduzione. Il risultato è stato di 950.000 traduzioni, ognuna della durata di 3 microsecondi, per un incremento totale di 3 secondi (circa il 5%) rispetto all'esecuzione nativa di Windows XP. Per ottenere questo risultato gli sviluppatori hanno fatto ricorso a diversi miglioramenti delle prestazioni che qui non saranno discussi. Per ulteriori informazioni consultate le note bibliografiche alla fine di questo capitolo.

16.4.3 Assistenza hardware

Senza un certo livello di supporto hardware la virtualizzazione sarebbe impossibile. Quanto maggiore è il supporto hardware disponibile all'interno di un sistema, tanto più stabili e ricche di funzionalità possono essere le macchine virtuali e tanto migliori possono risultare le prestazioni. A partire dal 2005 e per le generazioni successive Intel ha aggiunto alla famiglia di CPU Intel x86 il nuovo supporto della virtualizzazione (le istruzioni VT-x). Ora dunque non è più necessaria la traduzione binaria.

In pratica, tutte le principali CPU general-purpose stanno fornendo quantità sempre maggiori di supporto hardware per la virtualizzazione. Per esempio, la tecnologia di

virtualizzazione AMD (**AMD-V**) è apparsa in numerosi processori AMD a partire dal 2006. Questa tecnologia definisce due nuove modalità di funzionamento, host e guest: si passa da un processore a due modalità a un processore multimodale. Il VMM può attivare la modalità host, definire le caratteristiche di ogni macchina virtuale guest e quindi impostare il sistema in modalità guest, passando il controllo del sistema a un sistema operativo guest in esecuzione sulla macchina virtuale. In modalità guest il sistema operativo virtualizzato pensa di essere in esecuzione su hardware nativo e vede tutti i dispositivi inclusi nella definizione del guest fornita dall'host. Se il guest tenta di accedere a una risorsa virtualizzata il controllo viene passato al VMM per la gestione di tale interazione. Intel VT-x è simile a AMD-V, in quanto fornisce le modalità root e nonroot, equivalenti alle modalità host e guest. Entrambe forniscono alla VCPU strutture di dati per il caricamento e la memorizzazione automatici dello stato della CPU durante i cambi di contesto del guest. Sono inoltre fornite le **strutture di controllo della macchina virtuale (VMCS)**, per la gestione degli stati di host e guest e per i vari controlli di esecuzione del guest, i controlli di uscita e le informazioni sul perché i guest ritornano il controllo all'host. In quest'ultimo caso, per esempio, una violazione di tabella delle pagine nidificata causata da un tentativo di accedere a memoria indisponibile può provocare l'uscita del guest.

AMD e Intel, nel loro ambiente virtuale, hanno anche indirizzato il problema della gestione della memoria. Con i miglioramenti nella gestione della memoria portati da RVI di AMD e EPT di Intel, il VMM non ha più necessità di implementare le tabelle NPT via software. In sostanza, queste CPU implementano tabelle delle pagine nidificate nell'hardware per consentire al VMM di controllare completamente la paginazione mentre le CPU accelerano la traduzione di indirizzi virtuali in indirizzi fisici. Le NPT aggiungono un nuovo livello che rappresenta il punto di vista dei guest sulla traduzione da logico a fisico degli indirizzi. La funzione di walk sulle tabelle offerta dalla CPU include questo nuovo livello, permettendo di scorrere dalla tabella guest alla tabella VMM alla ricerca dell'indirizzo fisico desiderato. Un insuccesso della TLB porta a un degrado delle prestazioni, perché devono essere attraversate più tabelle (le tabelle delle pagine dell'host e del guest) per completare la ricerca. La Figura 16.4 mostra il lavoro di traduzione supplementare svolto dall'hardware per tradurre un indirizzo virtuale dell'ospite nel corrispondente indirizzo fisico.

L'I/O è un altro settore che ha subito miglioramenti grazie all'assistenza dell'hardware. Si consideri che il controller standard della memoria ad accesso diretto (DMA) accetta un indirizzo di memoria di destinazione e un dispositivo sorgente di I/O e trasferisce i dati tra i due senza l'intervento del sistema operativo. Senza assistenza hardware, un guest potrebbe provare a impostare un trasferimento DMA che interessa anche la memoria del VMM o di altri guest. In CPU che forniscono DMA assistito dall'hardware (come le CPU Intel con VT-d), anche DMA ha un livello di indirizzazione. In primo luogo, il VMM impone **domini di protezione** per segnalare alla CPU quale parte della memoria fisica appartiene a ogni ospite. Successivamente, assegna i dispositivi di I/O ai domini di protezione, permettendo loro l'accesso diretto a quelle regioni di memoria (e solo a quelle). L'hardware trasforma poi l'indirizzo

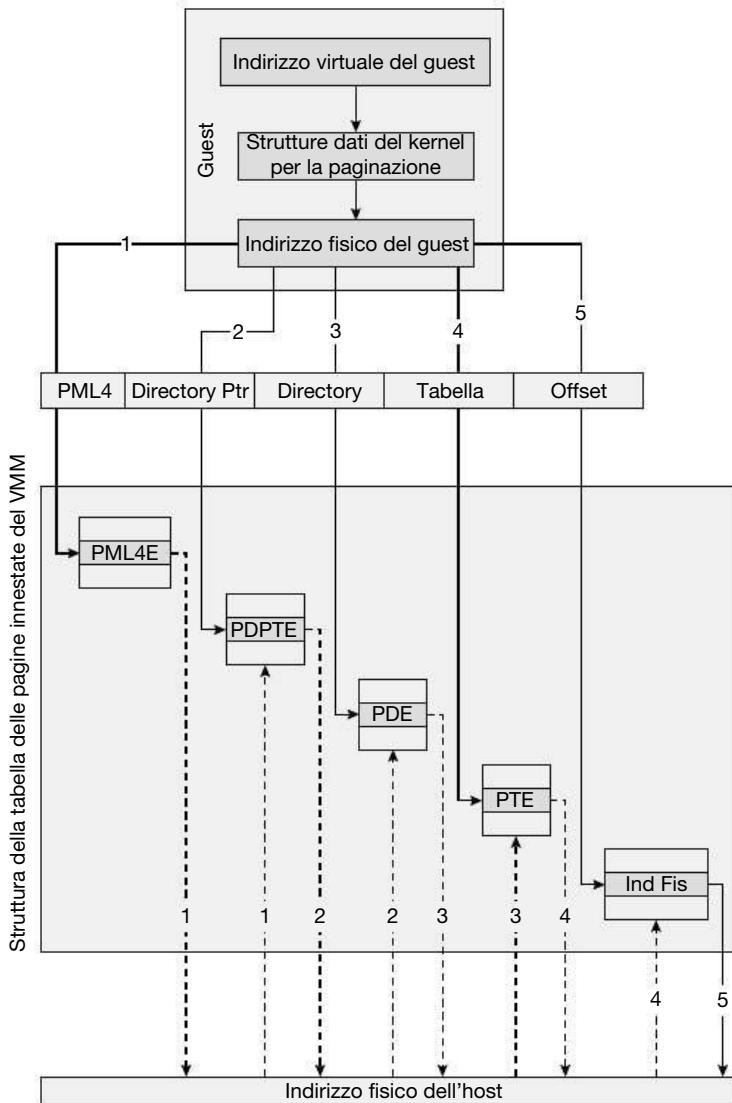


Figura 16.4 Tabelle delle pagine innestate.

presente in una richiesta DMA emessa da un dispositivo di I/O nell'indirizzo di memoria fisica dell'host associata all'I/O. In questo modo i trasferimenti DMA vengono passati da un guest a un dispositivo senza l'interferenza del VMM.

Allo stesso modo, gli interrupt devono essere consegnati al guest appropriato e non devono essere visibili ad altri guest. Per mezzo una funzione di rimappatura degli interrupt le CPU con assistenza hardware alla virtualizzazione inviano automaticamente un interrupt per un guest a un core dove è in esecuzione un thread di quel guest. In questo modo il guest riceve gli interrupt senza bisogno di un intervento del VMM. Senza la rimappatura degli interrupt i guest malevoli potrebbero generare interrupt allo scopo di ottenere il controllo del sistema host. Si vedano le Note bibliografiche alla fine di questo capitolo per maggiori dettagli.

16.5 Tipologie di macchine virtuali e loro implementazioni

Abbiamo fin qui analizzato alcune delle tecniche utilizzate per implementare la virtualizzazione. Prendiamo ora in esame i principali tipi di macchine virtuali, la loro implementazione, le loro funzionalità e come vengono utilizzati i blocchi costituenti appena descritti per creare un ambiente virtuale. Naturalmente l'hardware su cui le macchine virtuali sono in esecuzione è causa di notevole variabilità tra i metodi di implementazione. Discuteremo qui le implementazioni in generale, lasciando sottointeso il fatto che il VMM può approfittare di assistenza hardware, ove disponibile.

16.5.1 Il ciclo di vita della macchina virtuale

Cominciamo la nostra trattazione con il ciclo di vita della macchina virtuale. Qua-lunque sia il tipo di hypervisor, al momento della creazione di una macchina virtuale vengono forniti determinati parametri al VMM. Questi parametri includono di solito il numero di CPU, la quantità di memoria, i dettagli della rete e dello storage. Di questi parametri il VMM terrà conto durante la creazione del guest. Un utente, per esempio, potrebbe voler creare un nuovo guest con due CPU virtuali, 4 GB di memoria, 10 GB di spazio su disco, una interfaccia di rete che ottiene il suo indirizzo IP via DHCP e l'accesso al drive DVD.

Il VMM crea la macchina virtuale secondo questi parametri. Nel caso di un hypervisor di tipo 0 le risorse sono generalmente dedicate. In questa situazione, se non esistono due CPU virtuali disponibili e non assegnate la richiesta di creazione fallisce. In altri casi, a seconda delle tipologie di hypervisor, le risorse possono essere dedicate o virtuali. Certamente, un indirizzo IP non può essere condiviso, ma le CPU virtuali sono di solito **multiplate** (*multiplexed*) sulle CPU fisiche, come discusso nel Paragrafo 16.6.1. Allo stesso modo la gestione della memoria comporta solitamente l'allocazione ai guest di più memoria di quanta in realtà ne esista fisicamente. Questo meccanismo è più complicato, ed è descritto nel Paragrafo 16.6.2.

Infine, quando la macchina virtuale non è più necessaria può essere eliminata. Quando ciò accade, il VMM libera lo spazio disco utilizzato e poi rimuove la configurazione associata alla macchina virtuale, in sostanza dimenticando l'esistenza di questa macchina.

Questi passaggi sono abbastanza semplici in confronto alla costruzione, alla configurazione, all'esecuzione e alla rimozione di macchine fisiche. La creazione di una macchina virtuale a partire da una macchina virtuale esistente può consistere semplicemente nel fare clic sul pulsante “clone” e nel fornire un nuovo nome e un nuovo indirizzo IP. Questa semplicità nella creazione può portare a una *crescita disordinata delle macchine virtuali*, ovvero alla presenza su un sistema di così tante macchine virtuali che il loro utilizzo, la loro storia e il loro stato risultano confusi e difficili da monitorare.

16.5.2 Hypervisor di tipo 0

Gli hypervisor di tipo 0 sono esistiti per molti anni con diverse denominazioni, tra cui “partizioni” e “domini”. Si tratta di strumenti hardware e per questo vi sono lati positivi e negativi. I sistemi operativi non hanno bisogno di fare nulla di speciale per sfruttare le loro caratteristiche. Il VMM è codificato nel firmware, viene caricato al momento dell'avvio e, a sua volta, carica le immagini del guest da eseguire in ogni partizione. Le funzionalità di un hypervisor di tipo 0 tendono a essere meno di quelle degli altri tipi, perché è implementato in hardware. Per esempio, un sistema può essere suddiviso in quattro sistemi virtuali, ciascuno con CPU, memoria e dispositivi di I/O dedicati. Ogni guest ritiene di avere hardware dedicato proprio perché è così nella realtà. Ciò semplifica molti dettagli di implementazione.

L'I/O presenta qualche difficoltà, perché non è facile dedicare dispositivi di I/O ai guest se non ve ne sono a sufficienza. Che cosa succede se un sistema è dotato di due porte Ethernet e di più di due guest, per esempio? I casi sono due: o tutti gli ospiti devono avere propri dispositivi di I/O, o il sistema deve permettere la condivisione dei dispositivi. A seconda dei casi, l'hypervisor gestisce l'accesso condiviso oppure attribuisce i permessi di accesso di tutti i dispositivi a una **partizione di controllo**. Nella partizione di controllo un sistema operativo guest fornisce servizi (come il networking) ad altri guest per mezzo di demoni e l'hypervisor indirizza le richieste di I/O in modo appropriato. Alcuni hypervisor di tipo 0 sono ancora più sofisticati e possono muovere le CPU fisiche e la memoria tra i guest in esecuzione. In questi casi i guest sono paravirtualizzati, sono cioè consapevoli della virtualizzazione e contribuiscono alla sua esecuzione. Un guest, per esempio, deve monitorare i segnali provenienti dall'hardware o dal VMM che indicano un cambiamento hardware, sondare i propri dispositivi per rilevare il cambiamento e aggiungere o sottrarre CPU o memoria dalle proprie risorse disponibili.

Poiché la virtualizzazione di tipo 0 è molto vicina all'implementazione hardware cruda, andrebbe considerata separatamente dagli altri metodi discussi qui. Un hypervisor di tipo 0 può eseguire più sistemi operativi guest (uno in ogni partizione hardware). Tutti questi guest, poiché sono in esecuzione direttamente sull'hardware, possono essere a loro volta VMM. In sostanza, i sistemi operativi guest in un hypervisor di tipo 0 sono sistemi operativi nativi con un sottoinsieme di hardware a loro disposizione. Grazie a ciò, ciascuno può avere i propri sistemi operativi guest (Figura 16.5). Altri tipi di hypervisor di solito non sono in grado di fornire questa funzionalità di virtualizzazione nella virtualizzazione.

16.5.3 Hypervisor di tipo 1

Gli hypervisor di tipo 1 si trovano comunemente nei data center aziendali e stanno diventando in un certo senso “i sistemi operativi dei data center”. Si tratta di sistemi operativi per usi speciali che girano nativamente sull'hardware, ma piuttosto che fornire chiamate di sistema e altre interfacce per il funzionamento dei programmi, creano, eseguono e gestiscono sistemi operativi guest. Oltre a essere in esecuzione su hardware standard, essi possono essere eseguiti su hypervisor di tipo 0, ma non su

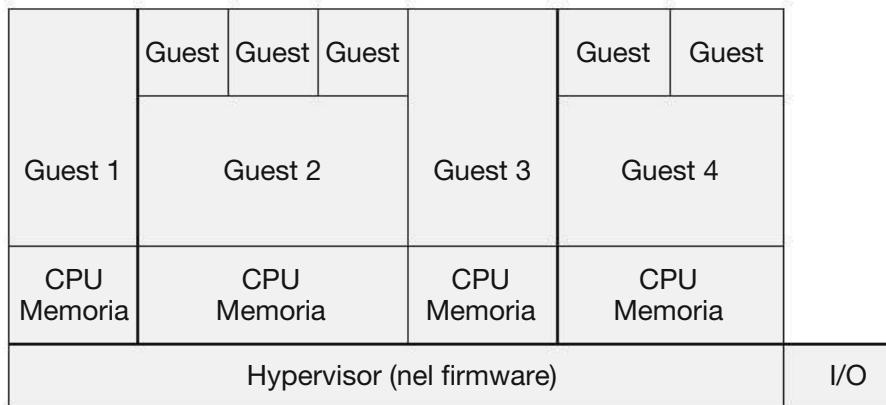


Figura 16.5 Hypervisor di tipo 0.

hypervisor di tipo 1. Qualunque sia la piattaforma, in genere i guest non sanno di essere in esecuzione su qualcosa di diverso dall'hardware nativo.

Gli hypervisor di tipo 1 sono in esecuzione in modalità kernel e traggono vantaggio dalla protezione hardware. Quando la CPU host lo consente, usano molteplici modalità per permettere ai sistemi operativi guest il proprio controllo e per migliorare le prestazioni. Gli hypervisor di tipo 1 implementano i driver di periferica per l'hardware su cui girano, perché nessun altro componente potrebbe farlo. Visto che sono sistemi operativi, devono anche fornire scheduling della CPU, gestione della memoria, gestione dell'I/O, protezione e anche sicurezza. Spesso forniscono API a supporto di applicazioni dei guest o applicazioni esterne che forniscono funzionalità come backup, monitoraggio e sicurezza. Molti hypervisor di tipo 1 sono prodotti commerciali “closed source”, come per esempio VMware ESX, mentre altri sono open source o ibridi, come per esempio Citrix XenServer e la sua controparte open source Xen.

Utilizzando hypervisor di tipo 1 i gestori dei data center possono controllare e gestire i sistemi operativi e le applicazioni in modi nuovi e sofisticati. Un vantaggio importante è la capacità di consolidare più sistemi operativi e applicazioni su un minor numero di sistemi. Per esempio, piuttosto che avere dieci sistemi utilizzati al 10% delle loro potenzialità, un data center potrebbe avere un unico server per la gestione dell'intero carico. Se cresce l'utilizzo, i guest e le loro applicazioni possono essere spostati su sistemi meno carichi senza interruzione del servizio. Con l'utilizzo di istantanee e clonazione il sistema può salvare gli stati dei guest e duplicarli, un compito molto più facile rispetto al ripristino da backup o all'installazione manuale o tramite script o altri strumenti. Il prezzo di questa maggiore gestibilità è il costo del VMM (se si tratta di un prodotto commerciale), la necessità di imparare nuovi strumenti e metodi di gestione e la maggiore complessità.

Un'altra tipologia di hypervisor di tipo 1 comprende vari sistemi operativi general-purpose dotati di funzionalità VMM. In questo caso, un sistema operativo come Linux RedHat Enterprise, Windows, Solaris o Oracle svolge le sue normali funzioni, oltre a fornire un VMM per permettere ad altri sistemi operativi di essere eseguiti come guest. A causa delle loro funzioni supplementari, questi hypervisor in genere forni-

scono meno funzionalità di virtualizzazione di altri hypervisor di tipo 1. Essi trattano un sistema operativo guest come un normale processo, che gode inoltre di una gestione speciale nel caso in cui il guest tenti di eseguire istruzioni speciali.

16.5.4 Hypervisor di tipo 2

Gli hypervisor di tipo 2 sono meno interessanti per noi che siamo interessati ai sistemi operativi, perché c'è molto poco coinvolgimento del sistema operativo in questi gestori di macchine virtuali a livello applicativo. Questo tipo di VMM è semplicemente un processo eseguito e gestito dall'host e nemmeno l'host sa che è in atto una virtualizzazione all'interno del VMM.

Gli hypervisor di tipo 2 hanno dei limiti non riscontrabili in altri tipi. Per esempio, un utente ha bisogno di privilegi di amministratore per accedere a molte delle funzioni assistite dall'hardware nelle CPU moderne. Se il VMM viene eseguito da un utente standard senza privilegi aggiuntivi non può sfruttare queste caratteristiche. A causa di questa limitazione e del sovraccarico dato dall'esecuzione di un sistema operativo general-purpose insieme ad altri sistemi operativi guest, gli hypervisor di tipo 2 tendono ad avere prestazioni complessive più scarse rispetto agli hypervisor di tipo 0 o 1.

Come spesso accade, i limiti degli hypervisor di tipo 2 sono in parte compensati da alcuni vantaggi. Essi infatti possono essere eseguiti su una varietà di sistemi operativi general-purpose e la loro esecuzione non richiede nessuna modifica al sistema operativo host. Uno studente può utilizzare un hypervisor di tipo 2, per esempio, per testare un sistema operativo non nativo senza sostituire il sistema operativo nativo. In questo modo, uno studente potrebbe disporre su un portatile Apple di versioni di Windows, Linux, Unix e sistemi operativi meno comuni, tutti disponibili per l'apprendimento e la sperimentazione.

16.5.5 Paravirtualizzazione

Come abbiamo visto, la paravirtualizzazione segue una strada diversa rispetto agli altri tipi di virtualizzazione. Piuttosto che cercare di indurre un sistema operativo guest a credere di avere un sistema per sé, la paravirtualizzazione presenta all'ospite un sistema che è simile, ma non identico, al suo sistema preferenziale. Il guest deve essere modificato per funzionare sull'hardware virtuale paravirtualizzato. Il lavoro addizionale richiesto da queste modifiche offre in cambio un guadagno in termini di uso più efficiente delle risorse e snellezza del livello paravirtuale.

Il VMM Xen, leader nella paravirtualizzazione, ha implementato diverse tecniche per ottimizzare le prestazioni dei guest, così come del sistema host. Per esempio, come abbiamo già visto, alcuni VMM presentano agli ospiti dispositivi virtuali che sembrano essere dispositivi reali. Invece di seguire tale approccio, Xen presenta un'astrazione pulita e semplice dei dispositivi che consente un I/O efficiente e una buona comunicazione tra il guest e il VMM per ciò che concerne il dispositivo di I/O. Per ogni dispositivo utilizzato da ogni guest c'è un buffer circolare condiviso tra il guest e il VMM tramite memoria condivisa. I dati di lettura e scrittura vengono inseriti in questo buffer, come mostrato nella Figura 16.6.

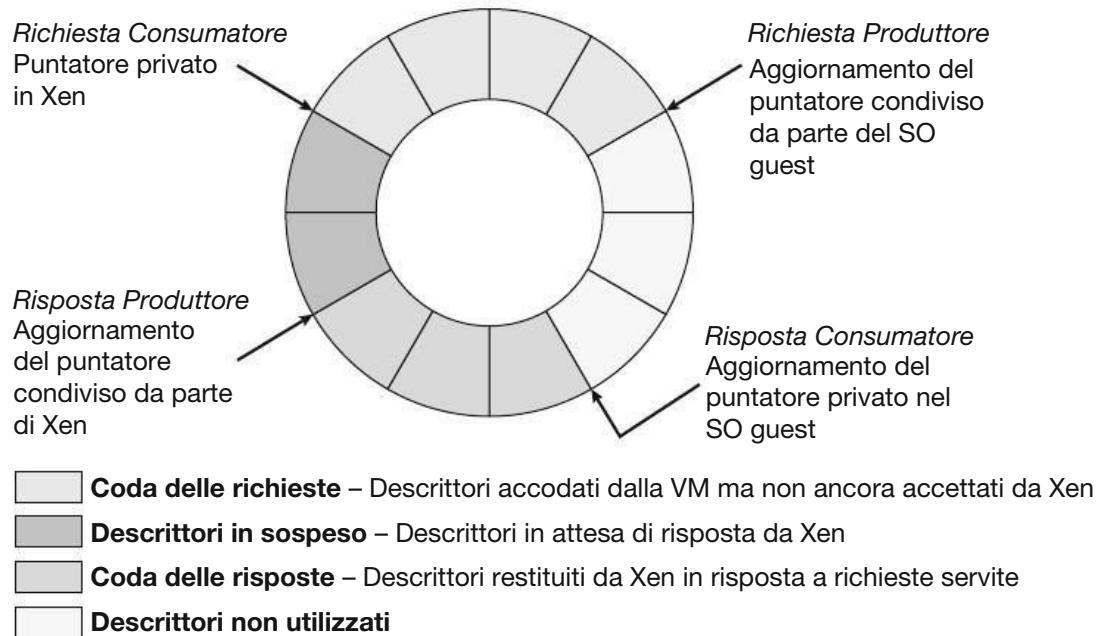


Figura 16.6 I/O di Xen mediante buffer circolare condiviso.

Per la gestione della memoria Xen non implementa tabelle delle pagine nidificate, ma ogni guest ha il proprio insieme di tabelle delle pagine, impostate in sola lettura. Quando è necessario un cambio di pagina della tabella, Xen richiede agli ospiti di usare un meccanismo specifico, una **hypercall** da parte del guest verso il VMM hypervisor. Il codice del kernel del sistema operativo guest deve quindi essere modificato per supportare questi metodi specifici di Xen. Per ottimizzare le prestazioni Xen permette ai guest di accodare modifiche multiple a una tabella delle pagine in modo asincrono tramite hypercall e quindi effettua controlli per assicurare che le modifiche siano completate prima di continuare le operazioni.

Xen permette la virtualizzazione delle CPU x86 senza l'uso di una traduzione binaria, richiedendo però alcune modifiche nei sistemi operativi guest, come descritto in precedenza. Nel corso del tempo, Xen ha saputo sfruttare le caratteristiche hardware a supporto della virtualizzazione. Come risultato, esso non richiede più la modifica dei guest ed essenzialmente non necessita più del meccanismo di paravirtualizzazione. La paravirtualizzazione è comunque ancora usata in altre soluzioni, per esempio negli hypervisor di tipo 0.

16.5.6 Virtualizzazione dell'ambiente di programmazione

Un altro tipo di virtualizzazione, basato su un modello di esecuzione diverso, è la virtualizzazione degli ambienti di programmazione. In questo caso un linguaggio di programmazione viene progettato per andare in esecuzione su un ambiente virtualizzato costruito ad hoc. Per esempio, Java di Oracle ha molte caratteristiche che dipendono dalla sua esecuzione nella macchina virtuale Java (JVM), tra cui i metodi specifici per la gestione della sicurezza e della memoria.

Se per virtualizzazione intendiamo esclusivamente la duplicazione di hardware, allora questo metodo non può essere realmente considerato come virtualizzazione. Non limitandoci a tale restrizione, possiamo definire un ambiente virtuale, basato su API, che fornisce un insieme di caratteristiche messe a disposizione di un particolare linguaggio e a programmi scritti in quel linguaggio. I programmi Java sono eseguiti all'interno dell'ambiente JVM e la JVM è compilata come un programma nativo sui sistemi su cui viene eseguita. Ciò significa che i programmi Java vengono scritti una sola volta e possono poi essere eseguiti su qualsiasi sistema (compresi tutti i principali sistemi operativi) su cui è disponibile una JVM. Lo stesso vale per i linguaggi interpretati, eseguiti all'interno di programmi che leggono ogni istruzione e la interpretano in operazioni native.

16.5.7 Emulazione

La virtualizzazione è probabilmente il metodo più comune per eseguire applicazioni progettate per un sistema operativo su un sistema operativo diverso, ma sulla stessa CPU. Questo metodo è abbastanza efficiente, perché le applicazioni sono state compilate per lo stesso insieme di istruzioni utilizzato dal sistema di destinazione.

Che cosa succede quando si ha bisogno di eseguire un'applicazione o un sistema operativo su una CPU diversa? In questo caso è necessario convertire tutte le istruzioni della CPU sorgente in istruzioni equivalenti della CPU destinazione. Un tale ambiente non è più virtualizzato, ma è piuttosto completamente emulato.

L'**emulazione** è utile quando il sistema host ha un'architettura di sistema che è diversa da quella per cui è stato compilato il sistema guest. Si supponga, per esempio, che una società abbia sostituito il suo sistema informatico obsoleto con un nuovo sistema, ma voglia continuare a eseguire alcuni importanti programmi che sono stati compilati per il vecchio sistema. I programmi possono essere eseguiti all'interno di un emulatore che converte tutte le istruzioni del sistema obsoleto nelle istruzioni native del nuovo sistema. L'emulazione può prolungare la vita dei programmi e ci permette di esplorare le vecchie architetture senza bisogno di possedere realmente una macchina con tale architettura.

Come ci si può aspettare, il grande problema dell'emulazione sono le prestazioni. L'emulazione di un insieme di istruzioni viene eseguita molto più lentamente rispetto alle istruzioni native, perché possono essere necessarie anche dieci istruzioni sul nuovo sistema per leggere, analizzare e simulare un'istruzione dal vecchio sistema. Così, a meno che la nuova macchina sia dieci volte più veloce rispetto alla vecchia, il programma in esecuzione sulla nuova macchina sarà eseguito più lentamente di quanto avveniva sul suo hardware nativo. Un altro ostacolo per i programmatori di emulatori sta nella difficoltà di creare un emulatore corretto, perché, in sostanza, il loro lavoro comporta la scrittura di un'intera CPU via software.

Nonostante queste difficoltà l'emulazione è molto popolare, soprattutto nell'ambito dei videogiochi. Molti famosi videogiochi sono state scritte per piattaforme che ora non sono più in produzione. Gli utenti che desiderano eseguire quei giochi possono spesso trovare un emulatore della piattaforma interessata ed eseguire il gioco

all'interno dell'emulatore senza doverlo modificare. I sistemi moderni sono così veloci rispetto alle vecchie console da gioco che esiste un emulatore di giochi persino per iPhone e vi sono giochi disponibili da eseguire al suo interno.

16.5.8 Contenitori di applicazioni

L'obiettivo della virtualizzazione è in alcuni casi quello di fornire un metodo per separare le applicazioni, controllare le loro prestazioni e il loro utilizzo delle risorse e creare un modo semplice per avviarle, arrestarle, spostarle e gestirle. In questi casi non è probabilmente necessaria una vera e propria virtualizzazione, in particolare se le applicazioni sono tutte compilate per lo stesso sistema operativo. Possiamo in alternativa utilizzare dei contenitori di applicazioni.

Consideriamo un esempio di contenitore di applicazioni. A partire dalla versione 10, Oracle Solaris ha incluso contenitori (container), o zone, che creano uno strato virtuale tra il sistema operativo e le applicazioni. In questo sistema è installato un unico kernel e l'hardware non è virtualizzato. A essere virtualizzato è il sistema operativo con i suoi dispositivi, in modo da offrire ai processi una zona in cui ognuno abbia l'impressione di essere l'unico processo in esecuzione sul sistema. Possono essere create una o più zone, ognuna con le proprie applicazioni, il proprio indirizzo di rete, le proprie porte, i propri account utente, e così via. Le risorse di CPU e memoria possono essere ripartite tra le zone e i processi di sistema. Ogni zona è in grado di eseguire il proprio scheduler per ottimizzare le prestazioni delle applicazioni sulle risorse assegnate. La Figura 16.7 mostra un sistema Solaris 10 con due zone e lo spazio utente standard, chiamato “globale”.

16.6 Virtualizzazione e componenti dei sistemi operativi

Abbiamo finora esplorato i blocchi costituenti necessari per la virtualizzazione e i vari tipi di virtualizzazione. In questo paragrafo ci immergeremo negli aspetti della virtualizzazione che riguardano il sistema operativo. Tra questi vedremo come il VMM fornisce funzioni di base del sistema operativo, come lo scheduling e la gestione di I/O e memoria. Risponderemo in particolare a domande come le seguenti. Come effettua lo scheduling della CPU il VMM, quando i sistemi operativi guest credono di avere CPU dedicate? Come può funzionare la gestione della memoria quando molti guest richiedono grandi quantità di memoria?

16.6.1 Scheduling della CPU

Un sistema con la virtualizzazione, anche un sistema a singola CPU, si comporta spesso come un sistema multiprocessore. Il software di virtualizzazione presenta una o più CPU virtuali a ciascuna delle macchine virtuali in esecuzione sul sistema e poi pianifica l'utilizzo delle CPU fisiche tra le macchine virtuali.

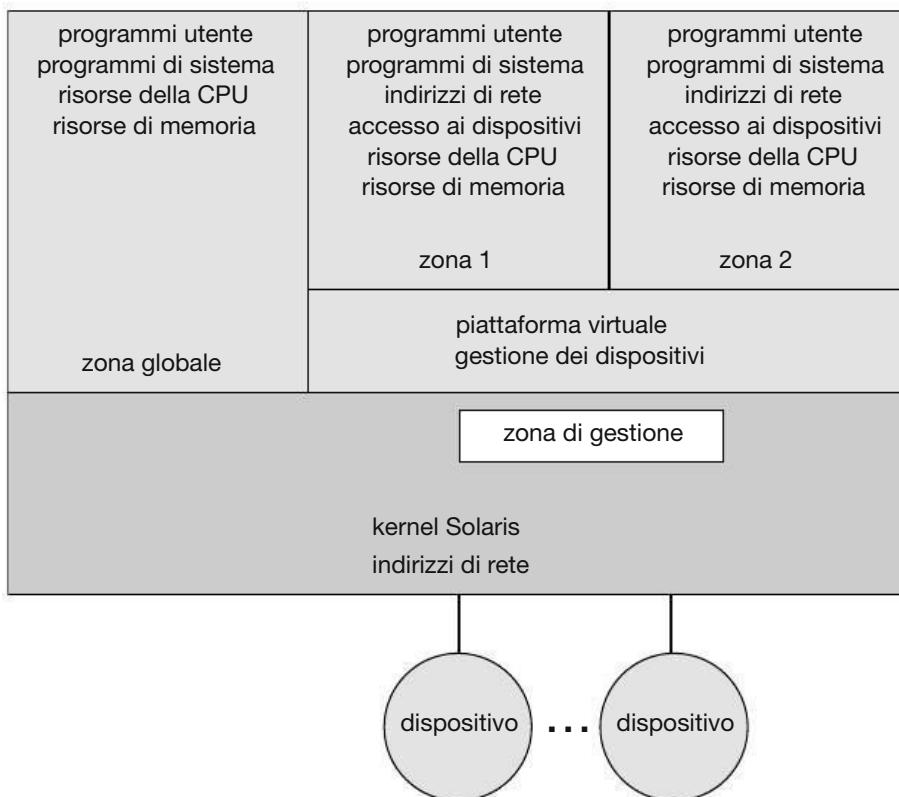


Figura 16.7 Solaris 10 con due zone.

Le differenze significative tra le tecnologie di virtualizzazione rendono difficile riassumere l'effetto della virtualizzazione sullo scheduling. Consideriamo in primo luogo il caso generale dello scheduling del VMM. Il VMM ha un certo numero di CPU fisiche disponibili e un certo numero di thread da eseguire su tali CPU. I thread possono essere del VMM o dei guest. I guest sono configurati con un certo numero di CPU virtuali al momento della creazione, ma questo numero può essere regolato nel corso dell'esistenza della VM. Quando ci sono abbastanza CPU per assegnare il numero richiesto da ogni guest, il VMM può trattare la CPU come dedicata e pianificare su di essa solo i thread del guest che la possiede. In questa situazione i guest si comportano proprio come sistemi operativi nativi in esecuzione su CPU native.

Naturalmente, in altre situazioni, il numero di CPU potrebbe non essere sufficiente. Lo stesso VMM ha bisogno di alcuni cicli di CPU per la gestione dei guest e dell'I/O e può rubare cicli ai guest pianificando i suoi thread su qualsiasi CPU del sistema, ma l'impatto di questa azione è relativamente basso. Più difficile da trattare è il caso di sovra-assegnazione (*overcommitment*), in cui i guest sono configurati per usare più CPU di quante ne esistano nel sistema. In questo caso un VMM può utilizzare algoritmi standard di scheduling per far progredire ogni thread, ma può anche aggiungere elementi di equità a tali algoritmi. Per esempio, se ci fossero sei CPU hardware e 12 CPU assegnate ai guest, il VMM potrebbe allocare le risorse della CPU in modo proporzionale, dando a ciascun guest la metà delle risorse delle CPU che crede di avere. Il VMM

può comunque presentare ai guest tutte e 12 le CPU virtuali, ma nella loro mappatura sulle CPU fisiche il VMM può utilizzare il proprio scheduler per condividere le CPU in modo appropriato.

Anche quando si utilizza uno scheduler che fornisce equità, qualsiasi algoritmo di schedulazione del sistema operativo guest che assume di ottenere una certa quantità di progresso nella elaborazione in un determinato periodo di tempo sarà influenzato negativamente dalla virtualizzazione. Si consideri un sistema operativo time-sharing che tenta di assegnare 100 millisecondi per ogni intervallo di tempo per garantire agli utenti un tempo di risposta ragionevole. All'interno di una macchina virtuale questo sistema operativo è in balia del sistema di virtualizzazione per quanto riguarda le risorse della CPU che effettivamente riceve. Un determinato intervallo di tempo di 100 millisecondi può richiedere molto di più di 100 millisecondi di tempo della CPU virtuale. A seconda di quanto il sistema è impegnato, l'intervallo di tempo può arrivare a un secondo o più, con tempi di risposta molto scarsi per gli utenti registrati su quella macchina virtuale. L'effetto su un sistema operativo real-time può essere ancora peggiore.

L'effetto netto di questo scheduling stratificato è che i singoli sistemi operativi virtualizzati ricevono solo una parte dei cicli di CPU disponibili, anche se credono di poter ricevere tutti i cicli e, anzi, credono di essere responsabili della pianificazione di tutti quei cicli. Capita di frequente che gli orologi time-of-day delle macchine virtuali non siano corretti, perché i timer impiegano più tempo a scattare rispetto a quanto avverrebbe su CPU dedicate. La virtualizzazione può quindi vanificare gli sforzi per una buona realizzazione degli algoritmi di scheduling nei sistemi operativi all'interno di macchine virtuali.

Per correggere questi problemi un VMM avrà, per ogni tipo di sistema operativo che gli amministratori intendono installare come guest, un'applicazione in grado di prevenire i ritardi dell'orologio di sistema, dotata anche di altre funzioni come la gestione virtuale delle periferiche.

16.6.2 Gestione della memoria

L'uso efficiente della memoria nei sistemi operativi general-purpose è una delle chiavi principali per ottenere prestazioni migliori. In ambienti virtualizzati la memoria ha più utenti (i guest e le loro applicazioni, nonché il VMM) e si induce quindi una maggior pressione sul suo utilizzo. Un ulteriore fattore di pressione è il fatto che il VMM tipicamente sovrasserà (overcommit) la memoria, in modo che la memoria totale secondo le configurazioni dei guest supera la quantità di memoria fisicamente presente nel sistema. Il maggior bisogno di un utilizzo efficiente della memoria non viene dimenticato dagli implementatori di VMM, che prendono importanti misure atte a garantirne un utilizzo ottimale.

VMware ESX, per esempio, utilizza almeno tre metodi di gestione della memoria. Prima di occuparsi dell'ottimizzazione della memoria, il VMM deve stabilire quanta memoria reale deve utilizzare ogni guest. Per fare ciò, il VMM valuta prima la quantità massima di memoria di ogni guest secondo la sua configurazione. Dato che i sistemi

operativi general-purpose non si aspettano variazioni nella quantità di memoria del sistema, i VMM devono mantenere l'illusione che il guest abbia a disposizione quella quantità di memoria. Successivamente, il VMM calcola un obiettivo per l'allocazione di memoria reale a ogni guest in base alla memoria configurata per quel guest e ad altri fattori, quali la sovra-assegnazione di memoria e il carico del sistema. Il VMM utilizza quindi i tre meccanismi di basso livello descritti qui sotto per recuperare la memoria dai guest. L'effetto complessivo è quello di consentire ai guest di comportarsi come se avessero l'intera quantità di memoria richiesta anche se in realtà ne hanno meno.

1. Ricordate che un ospite crede di controllare l'allocazione della memoria tramite la gestione della sua tabella delle pagine, mentre in realtà il VMM mantiene una tabella delle pagine nidificata che ritraduce la tabella delle pagine del guest nella tabella delle pagine reale. Il VMM può utilizzare questo ulteriore livello per ottimizzare l'utilizzo della memoria da parte dei guest senza la conoscenza o l'aiuto da parte dei guest. Un approccio è quello di fornire una doppia paginazione, in cui il VMM ha i propri algoritmi di sostituzione delle pagine e utilizza pagine sul backing-store facendo credere al guest che queste si trovino nella memoria fisica. Naturalmente, il VMM ha meno conoscenza dei modelli di accesso alla memoria del guest rispetto al guest stesso, dunque la sua paginazione è meno efficiente e crea problemi di prestazioni. I VMM utilizzano questo metodo quando non ce ne sono altri disponibili, o quando gli altri metodi non permettono di avere sufficiente memoria libera. Tuttavia, questo non è l'approccio preferito.
2. Una soluzione comune è la seguente. Il VMM installa in ogni guest uno **pseudo-driver di periferica** o un modulo kernel su cui ha il controllo. (Uno pseudo-driver di periferica utilizza interfacce di driver di periferica e appare al kernel come un vero driver, ma in realtà non controlla nessun dispositivo. Si tratta piuttosto di un modo semplice per aggiungere codice in modalità kernel senza modificare direttamente il kernel). Questo gestore di memoria, chiamato **balloon**, comunica con il VMM e riceve comandi di allocazione o deallocazione di memoria. Quando il comando è di allocazione, il balloon aloca la memoria e chiede al sistema operativo di fissare (*pinning*) nella memoria fisica le pagine allocate (ricordiamo che il pinning blocca una pagina nella memoria fisica in modo che non possa essere spostata o tolta dalla memoria). Il guest avverte la pressione dovuta alla presenza di queste pagine appuntate e reagisce, in sostanza, diminuendo la quantità di memoria fisica che ha a disposizione. Il guest potrà quindi essere indotto a liberare memoria fisica per essere sicuro di avere una quantità sufficiente di memoria libera. Nel frattempo il VMM, sapendo che le pagine bloccate non saranno mai utilizzate dal processo balloon, toglie queste pagine fisiche dal guest e le assegna a un altro guest. Allo stesso tempo il guest utilizza i propri algoritmi di gestione della memoria e paginazione per gestire la memoria disponibile in maniera più efficiente. Se l'utilizzo della memoria nell'intero sistema diminuisce il VMM dirà al processo balloon all'interno dell'ospite di sbloccare e liberare in tutto o in parte la memoria, concedendo al guest un maggior numero di pagine da utilizzare.

3. Un altro metodo utilizzato comunemente per ridurre la pressione sulla memoria consiste nel determinare se la stessa pagina è stata caricata più di una volta. In tal caso il VMM elimina tutte le copie della pagina tranne una e ridirige tutti gli utenti della pagina all'unico esemplare rimasto. VMware, per esempio, campiona casualmente la memoria di un guest e crea un hash per ogni pagina campionata. Questo valore hash è un'identificazione (*thumbprint*) della pagina. L'hash di ogni pagina esaminata viene confrontato con gli altri hash già memorizzati in una tabella hash. Se c'è una corrispondenza, le pagine vengono confrontate byte per byte per vedere se sono davvero identiche e se lo sono una pagina viene liberata e il suo indirizzo logico viene mappato sull'altro indirizzo fisico. Questa tecnica potrebbe sembrare, a prima vista, inefficace, ma occorre considerare il fatto che i guest eseguono sistemi operativi. Quando più guest eseguono lo stesso sistema operativo è sufficiente avere in memoria solo una sola copia delle pagine attive del sistema operativo. Analogamente, lo stesso insieme di applicazioni potrebbe essere in esecuzione su diversi guest, portando anche in questo caso a una possibile fonte di condivisione di memoria.

16.6.3 I/O

Per quanto concerne l'I/O, gli hypervisor hanno un certo margine di manovra e possono essere meno preoccupati di rappresentare esattamente l'hardware sottostante ai loro guest. A causa di tutte le variazioni nei dispositivi di I/O, i sistemi operativi sono abituati a servirsi di meccanismi di I/O vari e flessibili. I sistemi operativi sono dotati per esempio di un meccanismo per fornire un'interfaccia uniforme a qualunque dispositivo di I/O sottostante. Le interfacce dei driver di I/O sono progettate per consentire ai produttori di hardware di terze parti di fornire i driver di periferica che collegano i loro dispositivi al sistema operativo. Di solito i driver di periferica possono essere caricati e scaricati dinamicamente. La virtualizzazione si avvale di questa flessibilità integrata, fornendo specifici dispositivi virtualizzati per sistemi operativi guest.

Come descritto nel Paragrafo 16.5, i VMM differiscono notevolmente nel modo di fornire I/O ai loro guest. I dispositivi di I/O possono essere dedicata ai guest, per esempio, oppure il VMM può disporre di driver di periferica su cui mappare l'I/O dei guest. Il VMM può anche fornire ai guest dei driver di periferica idealizzati, permettendo così una facile erogazione e gestione dei servizi di I/O dei guest. In questo caso, l'ospite vede un dispositivo facile da controllare, ma in realtà questo semplice driver di dispositivo comunica con il VMM che invia le richieste a un dispositivo reale più complesso attraverso un driver reale più complesso. L'I/O in ambienti virtuali è complicato e richiede un'attenta progettazione e implementazione del VMM.

Consideriamo il caso di un hypervisor e di una combinazione hardware che permette ai dispositivi di essere dedicati a un guest e consente ai guest di accedere direttamente a tali dispositivi. Un dispositivo dedicato a un guest non è ovviamente a disposizione di altri guest, ma questo accesso diretto può comunque essere utile in alcune circostanze. La ragione per consentire l'accesso diretto è di migliorare le prestazioni di I/O: meno deve fare l'hypervisor per permettere l'I/O ai suoi guest, più ve-

loce può essere l'I/O. Nel caso degli hypervisor di tipo 0 che forniscono accesso diretto al dispositivo, i guest possono spesso ottenere le stesse velocità dei sistemi operativi nativi. Quando invece un hypervisor di tipo 0 fornisce dispositivi condivisi, le prestazioni, in confronto, possono risentirne.

Utilizzando l'accesso diretto al dispositivo in hypervisor di tipo 1 e 2 le prestazioni possono essere simili a quelle dei sistemi operativi nativi se si è in presenza di un supporto hardware. L'hardware deve fornire DMA pass-through, con servizi come VT-d, oltre alla consegna diretta di interrupt a specifici guest. Data la frequenza con cui si verificano gli interrupt, non dovrebbe sorprendere che su un hardware senza queste caratteristiche i guest hanno prestazioni peggiori di quelle che avrebbero se fossero in esecuzione nativamente.

Oltre all'accesso diretto, i VMM forniscono l'accesso condiviso ai dispositivi. Si consideri un disco a cui accedono più guest. Nel condividere il dispositivo il VMM deve fornire una certa protezione, assicurando che un guest possa accedere solo ai blocchi specificati nella sua configurazione. In queste situazioni, il VMM deve prendere parte in ogni I/O, verificando la correttezza e instradando i dati nella comunicazione tra i guest e gli opportuni dispositivi.

I VMM devono intervenire anche nella gestione della rete. I sistemi operativi general-purpose hanno in genere un solo indirizzo IP, anche se a volte ne hanno più d'uno, per esempio per connettersi a una rete di gestione, di backup e di produzione. Con la virtualizzazione, ogni ospite ha bisogno di almeno un indirizzo IP, perché questa è la modalità principale di comunicazione del guest. Un server che esegue un VMM può avere quindi decine di indirizzi. Il VMM agisce come uno switch virtuale che instrada i pacchetti di rete verso il guest destinatario.

I guest possono essere “direttamente” collegati alla rete mediante un indirizzo IP visibile dall'esterno (**bridging**) o, in alternativa, il VMM può fornire un **indirizzo NAT** (*network address translation*). L'indirizzo NAT è locale al server su cui il guest è in esecuzione e il VMM fornisce l'instradamento tra la rete esterna e il guest. Il VMM fornisce inoltre servizi di firewalling, mediando le connessioni tra i guest all'interno del sistema e tra i guest e i sistemi esterni.

16.6.4 Gestione dello storage

Una questione importante nel determinare come funziona la virtualizzazione è questa: se sono stati installati più sistemi operativi, cos'è e dov'è il disco di avvio? Gli ambienti virtualizzati devono chiaramente avere un approccio diverso nella gestione dello storage rispetto ai sistemi operativi nativi. Anche il metodo multiboot, che prevede il partizionamento del disco di root, l'installazione di un boot manager in una partizione e l'installazione di ogni altro sistema operativo in altre partizioni non è sufficiente, perché il partizionamento ha dei limiti che impedirebbero di lavorare con decine o centinaia di macchine virtuali.

La soluzione a questo problema dipende ancora una volta dal tipo di hypervisor. Gli hypervisor di tipo 0 tendono a consentire il partizionamento del disco di root, in parte perché questi sistemi tendono a eseguire un minor numero di guest rispetto ad

altri sistemi. In alternativa, essi possono avere un gestore del disco come parte della partizione di controllo. Il gestore del disco fornisce spazio disco (compresi i dischi di avvio) alle altre partizioni.

Gli hypervisor di tipo 1 memorizzano il disco di root del guest (insieme alle informazioni di configurazione) in uno o più file all'interno dei file system forniti dal VMM. Gli hypervisor di tipo 2 memorizzano le stesse informazioni nel file system del sistema operativo host. In sostanza, un'immagine del disco con tutto il contenuto del disco principale del guest è memorizzata in un file del VMM. A parte per i potenziali problemi di prestazioni che ciò comporta, si tratta di una soluzione intelligente, perché semplifica la copia e lo spostamento dei guest. Se l'amministratore vuole un duplicato di un guest (per esempio per effettuare un test), deve semplicemente copiare l'immagine del guest e segnalare al VMM la presenza della nuova copia. L'avvio della nuova VM permette di avere a disposizione un guest identico. Spostare una macchina virtuale da un sistema a un altro che esegue lo stesso VMM è semplice quanto arrestare il guest, copiare l'immagine nell'altro sistema e avviare il guest nel nuovo ambiente.

I guest hanno a volte bisogno di più spazio su disco rispetto a quanto ve ne sia disponibile nella loro immagine. Per esempio, un server di database non virtualizzato potrebbe utilizzare diversi file system distribuiti su più dischi per memorizzare varie parti del database. La virtualizzazione di una simile banca dati di solito comporta la creazione di diversi file che il VMM presenta ai guest come dischi. Il guest viene eseguito normalmente e il VMM traduce le richieste di I/O provenienti dal guest in comandi di I/O per i file corretti.

Spesso i VMM forniscono un meccanismo per catturare un sistema fisico con la sua attuale configurazione e convertirlo in un guest che sono in grado di gestire ed eseguire. Sulla base di quanto descritto in precedenza, dovrebbe essere chiaro che questa conversione **da fisico a virtuale** (*P-to-V, Physical-to-Virtual*) legge i blocchi dei dischi del sistema fisico e li memorizza all'interno dei file sul sistema del VMM oppure su storage condiviso a cui il VMM può accedere. Non è forse così evidente la necessità di una procedura **da virtuale a fisico** (*V-to-P, Virtual-to-Physical*) per convertire un guest in un sistema fisico. Questa operazione è talvolta necessaria per il debugging: un problema potrebbe essere causato dal VMM o dai componenti a esso associati e l'amministratore può tentare di risolvere il problema rimuovendo la virtualizzazione dalle possibili cause. La conversione V-to-P può prendere i file contenenti tutti i dati del guest e generare blocchi sul disco di sistema, ricreando il guest come un sistema operativo nativo con le sue applicazioni. Una volta che il test si è concluso, il sistema nativo può essere riutilizzato per altri scopi quando la macchina virtuale ritorna in servizio oppure può essere eliminata la macchina virtuale lasciando che il sistema nativo continui a funzionare.

16.6.5 Migrazione in tempo reale

Una caratteristica che non si trova in sistemi operativi general-purpose ma è presente negli hypervisor di tipo 0 e di tipo 1 è la migrazione in tempo reale (Live Migration) da un sistema all'altro di un guest in esecuzione. Abbiamo accennato a questa fun-

zionalità in precedenza. Vediamo ora in dettaglio come funziona la migrazione in tempo reale e perché i VMM riescono a realizzarla in maniera relativamente semplice, mentre i sistemi operativi general-purpose, nonostante le ricerche e alcuni tentativi, non ne sono in grado.

Consideriamo innanzitutto il funzionamento della migrazione in tempo reale. Un guest in esecuzione su un sistema viene copiato su un altro sistema che esegue lo stesso VMM. La copia avviene con una così breve interruzione del servizio che gli utenti connessi al guest e le connessioni di rete continuano senza subire un significativo impatto negativo. Questa capacità piuttosto sorprendente è molto efficace nella gestione delle risorse e nell'amministrazione dell'hardware. Dopo tutto, basta fare un confronto con i passi necessari in assenza di virtualizzazione: avvisare gli utenti, arrestare i processi, spostare eventualmente i file binari e riavviare i processi sul nuovo sistema in modo che gli utenti, solo a questo punto, siano in grado di utilizzare nuovamente i servizi. Con la migrazione in tempo reale, un sistema sovraccarico può essere alleggerito in tempo reale senza nessuna interruzione percettibile. Allo stesso modo, in un sistema che necessita di modifiche hardware o di sistema (per esempio, un aggiornamento del firmware o l'aggiunta, la rimozione o la riparazione di componenti hardware) è possibile migrare i guest, svolgere il lavoro di manutenzione e riportare di nuovo i guest sul sistema senza un impatto significativo sugli utenti o sulle connessioni remote.

La migrazione in tempo reale è resa possibile dalle interfacce ben definite tra guest e VMM e dalle limitate informazioni di stato che il VMM mantiene per il guest. Il VMM migra un ospite implementando le seguenti fasi.

1. Il VMM sorgente stabilisce una connessione con il VMM destinazione e riceve l'autorizzazione ad inviare un guest.
2. La destinazione crea un nuovo guest, mediante la creazione, tra l'altro, di una nuova VCPU e di una nuova tabella delle pagine annidata.
3. La sorgente invia tutte le pagine di memoria di sola lettura alla destinazione.
4. La sorgente invia tutte le pagine di lettura-scrittura alla destinazione, contrassegnandole come "pulite".
5. La sorgente ripete il passo 4, perché in quella fase alcune pagine sono state probabilmente modificate dal guest e sono ora "sporche". Queste pagine devono essere inviate nuovamente e contrassegnate di nuovo come "pulite".
6. Quando il ciclo delle fasi 4 e 5 diventa molto breve, il VMM sorgente blocca il guest, invia lo stato finale della VCPU e altri dettagli sullo stato, invia le ultime pagine "sporche" e dice alla destinazione di avviare l'esecuzione del guest. Una volta che la destinazione segnala alla sorgente che il guest è in esecuzione, la sorgente può terminarlo.

Tale sequenza è mostrata nella Figura 16.8.

Concludiamo questa discussione con alcuni dettagli interessanti, evidenziando anche alcuni limiti della migrazione in tempo reale. Innanzitutto, affinché le connessioni di rete possano continuare senza interruzioni l'infrastruttura di rete deve accettare il

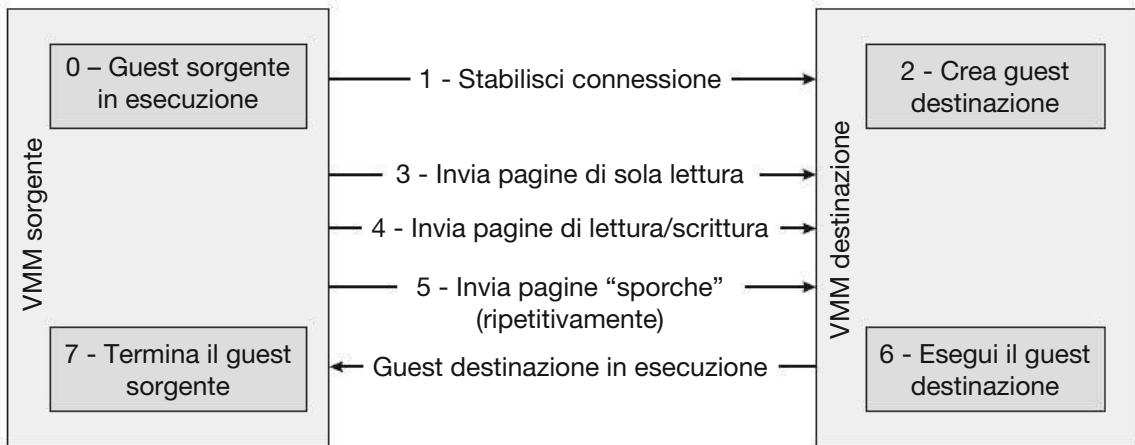


Figura 16.8 Migrazione in tempo reale di un guest tra due server.

fatto che un indirizzo MAC – l’indirizzo dell’hardware di rete – può muoversi tra i sistemi. Prima della virtualizzazione questo non accadeva, perché l’indirizzo MAC era legato all’hardware fisico. Con la virtualizzazione il MAC deve essere mobile per permettere alle connessioni di rete esistenti di continuare senza necessità di un ripristino. Gli switch di rete moderni sono in grado di gestire questa mobilità e instradare il traffico ovunque si trovi l’indirizzo MAC, seguendo dunque gli spostamenti.

Un limite della migrazione in tempo reale sta nel fatto che nessuno stato del disco viene trasferito. Uno tra i fattori che rendono possibile la migrazione in tempo reale è che la maggior parte dello stato del guest viene mantenuto all’interno del guest stesso – per esempio, la tabella dei file aperti, lo stato delle chiamate di sistema, lo stato del kernel, e così via. Siccome l’I/O su disco è molto più lento rispetto all’accesso alla memoria e lo spazio utilizzato su disco è di solito molto più grande rispetto alla memoria utilizzata, i dischi associati a un guest non possono essere spostati in una migrazione in tempo reale. Il disco del guest viene mantenuto in remoto e vi si accede attraverso la rete. In questo caso, lo stato di accesso al disco è mantenuto all’interno del guest e l’unica cosa che conta per il VMM sono le connessioni di rete. Visto che le connessioni di rete sono mantenute durante la migrazione, l’accesso al disco remoto può continuare. Per memorizzare le immagini della macchina virtuale e per realizzare tutte le altre memorizzazioni di cui il guest ha bisogno si utilizzano generalmente NFS, CIFS o iSCSI. Gli accessi a questi storage possono continuare in modo semplice se le connessioni di rete non vengono interrotte durante la migrazione del guest.

La migrazione in tempo reale rende possibili modalità interamente nuove nella gestione dei data center. Per esempio, gli strumenti di gestione della virtualizzazione sono in grado di monitorare tutti i VMM presenti in un ambiente e bilanciare automaticamente l’uso delle risorse spostando i guest tra i VMM. È inoltre possibile ottimizzare il consumo di energia elettrica e la gestione del raffreddamento migrando tutti i guest da alcuni server selezionati su altri server in grado di gestirne il carico, arrestando poi completamente i primi server. Se il carico aumenta, questi strumenti di gestione sono in grado di riaccendere i server e migrare di nuovo gli ospiti.

16.7 Esempi

Nonostante i vantaggi offerti dalle macchine virtuali, dopo il loro primo sviluppo esse sono state oggetto di poche attenzioni per diversi anni. Ciononostante, oggi le macchine virtuali sono diventate di moda come strumento per risolvere problemi di compatibilità di sistemi. In questo paragrafo analizziamo due popolari macchine virtuali: VMware workstation e la Java virtual machine. Come si vedrà, queste macchine virtuali possono generalmente essere eseguite su sistemi operativi di qualsiasi tipologia, descritti nei capitoli precedenti. I metodi di progettazione del sistema operativo (struttura semplice, microkernel, moduli e macchine virtuali) non sono pertanto mutuamente esclusivi.

16.7.1 VMware

VMware Workstation è una nota applicazione commerciale che astrae hardware Intel x86 e compatibile in macchine virtuali distinte. VMware è un ottimo esempio di hypervisor di tipo 2 e funge da applicazione in un sistema operativo host, quale Windows o Linux, che può così eseguire diversi sistemi operativi guest contemporaneamente come macchine virtuali indipendenti.

Si può osservare l'architettura di un sistema siffatto nella Figura 16.9. In questo esempio, Linux è il sistema operativo host, mentre FreeBSD, Windows NT e Windows XP sono i sistemi guest. Lo strato incaricato della virtualizzazione è il fulcro di VMware, poiché grazie a esso l'hardware viene astratto in macchine virtuali a sé stanti che eseguono sistemi operativi guest. Ciascuna macchina virtuale, oltre a possedere una CPU virtuale, può contare su elementi virtuali propri quali la memoria, i dischi, le interfacce di rete e via di seguito.

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

16.7.2 Java virtual machine

Il linguaggio di programmazione Java, introdotto dalla Sun Microsystems alla fine del 1995, è un linguaggio orientato agli oggetti molto diffuso. Java fornisce, oltre alla specifica del linguaggio e a una vasta libreria API, anche la definizione della macchina virtuale Java (*Java virtual machine*, JVM). Java è dunque un esempio della virtualizzazione dell'ambiente di programmazione trattata nel Paragrafo 16.5.6.

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

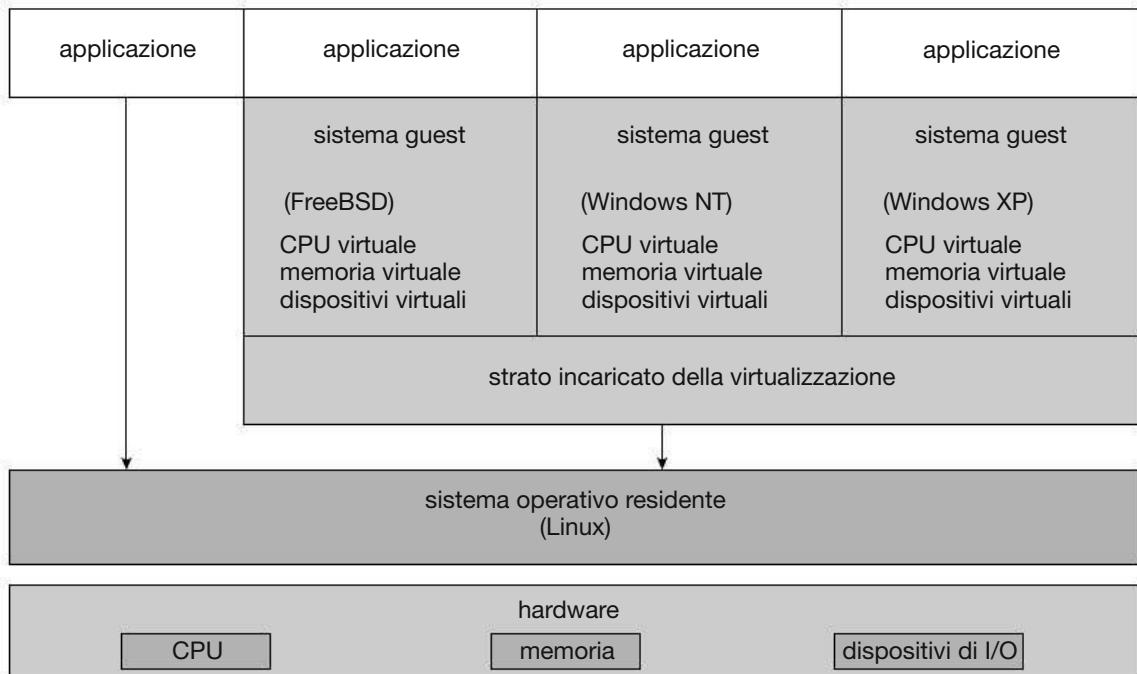


Figura 16.9 Architettura di VMware Workstation.

La JVM è un calcolatore astratto che consiste di un caricatore delle classi e di un interprete del linguaggio che esegue il bytecode (Figura 16.10). Il caricatore delle classi carica i file `.class`, sia del programma scritto in Java sia dalla libreria API, affinché l'interprete possa eseguirli. Dopo che una classe è stata caricata, il verificatore delle classi controlla la correttezza sintattica del bytecode, che il codice non produca accessi oltre i limiti dello stack e che non esegua operazioni aritmetiche sui puntatori, che potrebbero generare accessi illegali alla memoria. Se il controllo ha un esito positivo, la classe viene eseguita dall'interprete. La JVM gestisce la memoria in modo automatico procedendo alla sua “ripulitura” (*garbage collection*) che consiste nel recupero delle aree della memoria assegnate a oggetti non più in uso per restituirla al sistema. Al fine di incrementare le prestazioni dei programmi eseguiti dalla macchina virtuale una notevole attività di ricerca è focalizzata allo studio degli algoritmi di ripulitura della memoria.

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

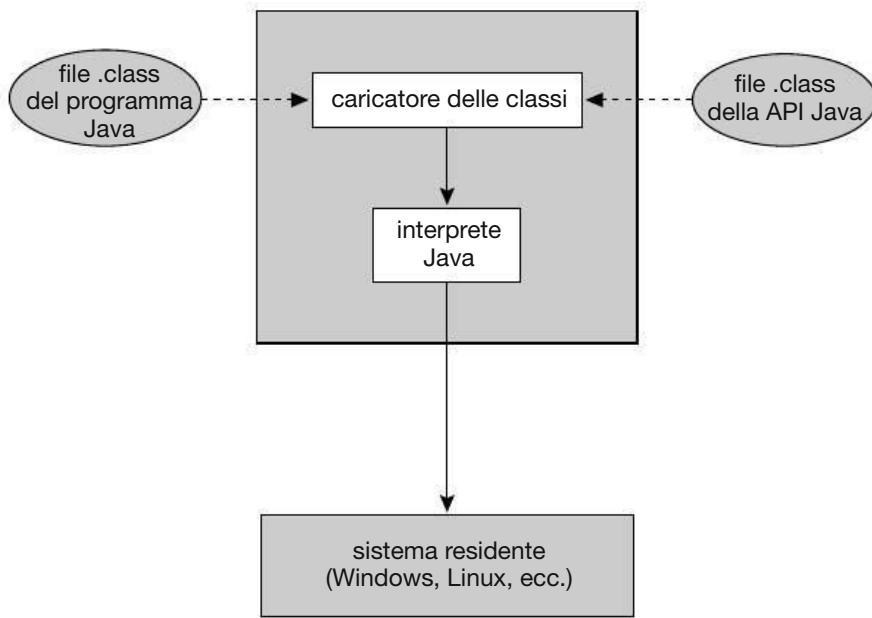


Figura 16.10 Macchina virtuale Java.

mente ancora più veloce è implementare la JVM in un circuito integrato che esegua le istruzioni bytecode come codice macchina nativo, eliminando del tutto la necessità di interpreti e compilatori.

16.8 Sommario

La virtualizzazione è un metodo per fornire a un guest un duplicato dell'hardware di un sistema. Diversi guest possono essere in funzione su un dato sistema e ciascuno crede di essere il sistema operativo nativo, con pieno controllo sul sistema. La virtualizzazione è nata come un metodo per consentire a IBM di isolare gli utenti e fornire loro i propri ambienti di esecuzione su mainframe IBM. Da allora, grazie al miglioramento dei sistemi e delle prestazioni delle CPU e attraverso tecniche software innovative, la virtualizzazione è diventata una caratteristica diffusa nei data center e anche sui personal computer. In seguito alla diffusione della virtualizzazione, i progettisti di CPU hanno aggiunto funzionalità per supportarla. Questo effetto "palla di neve" è destinato a continuare e farà crescere l'utilizzo della virtualizzazione e il supporto hardware alla stessa.

La virtualizzazione di tipo 0 è implementata nell'hardware e richiede modifiche al sistema operativo per garantire un corretto funzionamento. Queste modifiche offrono un esempio di paravirtualizzazione, in cui il sistema operativo non è inconsapevole delle virtualizzazioni, ma ha invece caratteristiche aggiuntive e algoritmi modificati per migliorarne le potenzialità e le prestazioni. Nella virtualizzazione di tipo 1, un gestore di macchina virtuale host (VMM) fornisce l'ambiente e le caratteristiche necessarie per creare, eseguire e distruggere le macchine virtuali guest. Ogni guest include

tutto il software tipicamente associato a un sistema nativo completo, compresi il sistema operativo, i driver dei dispositivi, le applicazioni, gli account utente, e così via.

Gli hypervisor di tipo 2 sono semplicemente applicazioni che girano su altri sistemi operativi ignari del fatto che sta avvenendo una virtualizzazione. Questi hypervisor non godono del supporto dell'hardware o di un host e devono quindi eseguire tutte le attività di virtualizzazione nell'ambito di un processo.

Sono diffuse anche altre tipologie di servizi che sono simili alla virtualizzazione, ma non soddisfano completamente la richiesta di replicare con esattezza l'hardware. La virtualizzazione dell'ambiente di programmazione fa parte del progetto di un linguaggio di programmazione. Il linguaggio specifica un'applicazione contenitore in cui eseguire i programmi e questa applicazione fornisce servizi ai programmi. L'emulazione viene utilizzata quando un sistema host ha un'architettura diversa da quella in cui è stato compilato il sistema guest. Ogni istruzione che il guest intende eseguire deve essere tradotta dal suo insieme di istruzioni nell'insieme di istruzioni dell'hardware nativo. Anche se questo metodo porta a un calo delle prestazioni, la penalizzazione è bilanciata dalla possibilità di eseguire vecchi programmi su un nuovo hardware incompatibile o di far girare videogiochi progettati per le vecchie console su un hardware moderno.

Implementare la virtualizzazione è impegnativo, soprattutto quando il supporto hardware è scarso. Per la virtualizzazione è richiesto un minimo supporto hardware, ma in presenza di un numero maggiore di funzionalità offerte dal sistema la virtualizzazione diventa più facile da implementare e le prestazioni dei guest migliorano. I VMM sfruttano qualsiasi supporto hardware a loro disposizione per ottimizzare lo scheduling della CPU, la gestione della memoria e i moduli di I/O, al fine di fornire ai guest un uso ottimale delle risorse, proteggendo contemporaneamente il VMM dai guest e i guest l'uno dall'altro .

Esercizi di ripasso

16.1 Descrivete i tre tipi tradizionali di virtualizzazione.

16.2 Descrivete i quattro ambienti di esecuzione che somigliano ad ambienti virtuali, ma non costituiscono una vera e propria virtualizzazione. Specificate come e perché non si tratta di una “vera” virtualizzazione.

16.3 Descrivete quattro vantaggi della virtualizzazione.

16.4 Perché su alcune CPU un VMM non può implementare una virtualizzazione basata su trap-and-emulate? Senza la possibilità di utilizzare trap-and-emulate, quale metodo può essere utilizzato per implementare la virtualizzazione?

16.5 Quale assistenza hardware per la virtualizzazione può essere fornita dalle moderne CPU?

16.6 Perché la migrazione in tempo reale è possibile in ambienti virtuali, ma lo è molto meno per un sistema operativo nativo?

Note bibliografiche

Il sistema IBM VM originale è stato descritto in [Meyer e Seawright 1970]. [Popek e Goldberg 1974] stabilisce le caratteristiche che aiutano a definire un VMM. I metodi di implementazione delle macchine virtuali sono trattati in [Agesen et al. 2010].

La virtualizzazione è un'area di ricerca attiva da molti anni. Uno dei primi tentativi di utilizzare la virtualizzazione per forzare un isolamento logico e per rendere scalabili i sistemi multicore è stato Disco ([Bugnion et al . 1997]). Basandosi su questo e su altri lavori, Quest-V ha usato la virtualizzazione per creare un intero sistema operativo distribuito all'interno di un sistema multicore ([Li et al. 2011]).

Il supporto hardware alla virtualizzazione di Intel x86 è descritto in [Neiger et al. 2006], mentre il supporto di AMD è descritto in un libro bianco pubblicato all'indirizzo <http://developer.amd.com/assets/npt-WP-1%201-final-TM.pdf>.

KVM è descritto in [Kivity et al. 2007] e Xen è descritto in [Barham et al. 2003] . [Poul-Henning Kamp 2000] descrive la somiglianza tra i contenitori Oracle Solaris e i jail BSD.

[Agesen et al. 2010] analizza le prestazioni della traduzione binaria. La gestione della memoria in VMware è trattata in [Waldspurger 2002]. Il problema dell'overhead nell'I/O in ambienti virtualizzati ha una soluzione proposta in [Gordon et al. 2012]. Alcune questione relative alla protezione e agli attacchi in ambienti virtuali sono analizzate in [Wojtczuk e Ruthkowska (2011)] .

La migrazione in tempo reale dei processi fu oggetto di ricerca negli anni '80. Il primo lavoro in cui venne trattato il problema è [Powell e Miller 1983]. I problemi individuati in quella ricerca lasciarono la migrazione in uno stato funzionalmente limitato, come descritto in [Milojicic et al. 2000]. VMware si rese conto che la virtualizzazione poteva consentire una migrazione in tempo reale ([Chandra et al. 2002]) e rilasciò vMotion, una funzione di migrazione in tempo reale, come parte di VMware vCenter, come descritto nel manuale d'uso VMware VirtualCenter Version 1.0: (http://www.vmware.com/pdf/VirtualCenter_Users_Manual.pdf).

I dettagli della realizzazione di una funzionalità simile nel VMM Xen si trovano in [Clark et al. 2005].

[Wojtczuk e Ruthkowska 2011] affrontano una ricerca atta a mostrare che senza rimappatura degli interrupt i guest malevoli potrebbero generare interruzioni che permettono di ottenere il controllo del sistema host.

Bibliografia

[Agesen et al. 2010] O. Agesen, A. Garthwaite, J. Sheldon e P. Subrahmanyam, “The Evolution of an x86 Virtual Machine Monitor”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 3–18, 2010.

[Barham et al. 2003] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt e A. Warfield, “Xen and the Art of Virtualization”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 164–177, 2003.

- [Bugnion et al. 1997]** E. Bugnion, S. Devine e M. Rosenblum, “Disco: Running Commodity Operating Systems on Scalable Multiprocessors”, Proceedings of the ACM Symposium on Operating Systems Principles, p. 143–156, 1997.
- [Chandra et al. 2002]** R. Chandra, B. Pfaff, J. Chow, M. Lam e M. Rosenblum, “Optimizing the Migration of Virtual Computers”, p. 377–390, 2002.
- [Clark et al. 2005]** C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt e A. Warfield, “Live Migration of Virtual Machines”, Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, p. 273–286, 2005.
- [Gordon et al. 2012]** A. Gordon, N. Amit, N. Har’El, M. Ben-Yehuda, A. Landau, A. Schuster e D. Tsafrir, “ELI: Bare-metal Performance for I/O Virtualization”, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, p. 411–422, 2012.
- [Kivity et al. 2007]** A. Kivity, Y. Kamay, D. Laor, U. Lublin e A. Liguori, “kvm: the Linux Virtual Machine Monitor”, Proceedings of the Linux Symposium, 2007.
- [Li et al. 2011]** Y. Li, M. Danish e R. West, “Quest-V: A Virtualized Multikernel for High-Confidence Systems”, Technical report, Boston University, 2011.
- [Meyer and Seawright 1970]** R. A. Meyer e L. H. Seawright, “A Virtual Machine Time-Sharing System”, IBM Systems Journal, Vol. 9, Num. 3, p. 199–218, 1970.
- [Milojicic et al. 2000]** D. S. Milojevic, F. Douglis, Y. Paindaveine, R. Wheeler e S. Zhou, “ProcessMigration”, ACM Computing Surveys, Vol. 32, Num. 3, p. 241–299, 2000.
- [Neiger et al. 2006]** G. Neiger, A. Santoni, F. Leung, D. Rodgers e R. Uhlig, “Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization”, Intel Technology Journal, Vol. 10, 2006.
- [Popek and Goldberg 1974]** G. J. Popek e R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures”, Communications of the ACM, Vol. 17, Num. 7, p. 412–421, 1974.
- [Poul-henning Kamp 2000]** R. N. M.W. Poul-henning Kamp, “Jails: Confining the Omnipotent Root”, Proceedings of the 2nd International System Administration and Networking Conference, 2000.
- [Powell and Miller 1983]** M. Powell e B. Miller, “Process Migration in DEMOS/MP”, Proceedings of the ACM Symposium on Operating Systems Principles, 1983.
- [Waldspurger 2002]** C. Waldspurger, “Memory Resource Management in VMware ESX Server”, Operating Systems Review, Vol. 36, Num. 4, p. 181–194, 2002.
- [Wojtczuk and Ruthkowska 2011]** R. Wojtczuk e J. Ruthkowska, “Following the White Rabbit: Software Attacks Against Intel VT-d Technology”, The Invisible Things Lab’s blog, 2011.

CAPITOLO

17

OBIETTIVI DEL CAPITOLO

- Panoramica di alto livello dei sistemi distribuiti e delle reti che li collegano.
- Descrizione della struttura generale dei sistemi operativi distribuiti.
- Descrizione generale della struttura e dei protocolli di comunicazione.
- Analisi delle problematiche nel progetto di sistemi distribuiti.

Sistemi distribuiti

Un sistema distribuito è un insieme di processori che non condividono la memoria o un clock. Ogni nodo ha la propria memoria locale e la comunicazione avviene tramite diversi tipi di reti, come bus ad alta velocità o Internet. In questo capitolo si tratta la struttura generale dei sistemi distribuiti e delle reti che li interconnettono, e si evidenziano le principali differenze di progettazione fra tali sistemi e i sistemi centralizzati.

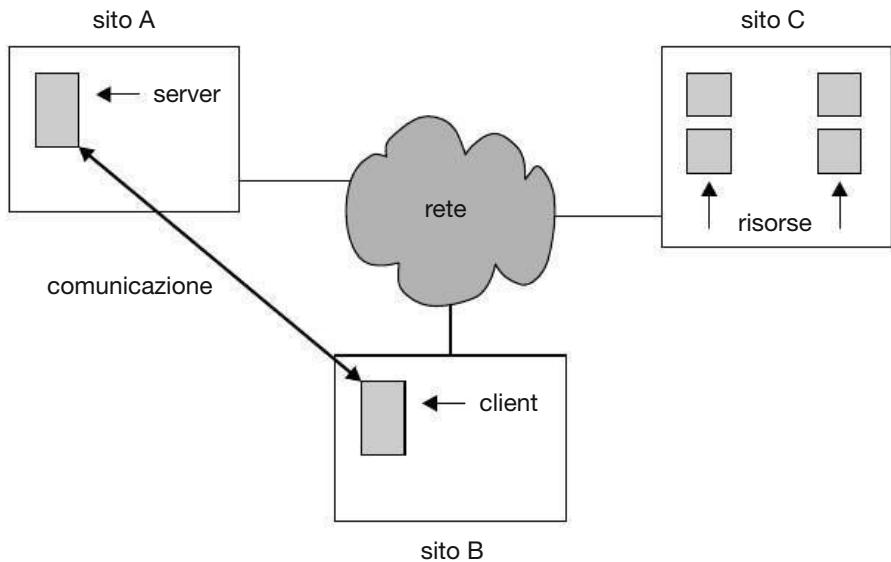


Figura 17.1 Sistema distribuito.

17.1 Vantaggi dei sistemi distribuiti

Un **sistema distribuito** è un insieme di nodi lasciamente connessi tramite una rete di comunicazione. Ogni nodo di un sistema distribuito considera remoti gli altri nodi del sistema e le rispettive risorse, mentre considera locali le proprie.

I nodi presenti in un sistema distribuito possono variare per dimensioni e funzioni; possono comprendere piccole unità d'elaborazione, personal computer e grandi calcolatori d'uso generale. I nodi sono chiamati in molti modi: processori, *siti*, *macchine* o *host*, secondo il contesto in cui sono citati. In questo testo si usa principalmente il termine *sito* per indicare una locazione fisica delle macchine, e *nodo* per riferirsi a uno specifico sistema in un sito. In genere un nodo in un sito, il *server*, dispone di una risorsa che un altro nodo in un altro sito, il *client* (o utente), vuole usare. Un sistema distribuito è illustrato nella Figura 17.1. I quattro motivi principali per costruire sistemi distribuiti sono: *condivisione delle risorse*, *velocizzazione delle elaborazioni*, *affidabilità* e *comunicazione*. Li discuteremo nel seguito.

17.1.1 Condivisione delle risorse

Se siti diversi, con risorse diverse, sono collegati tra loro, l'utente di un sito può avere la possibilità di usare le risorse disponibili in un altro sito. Per esempio, un utente del sito A può usare una stampante laser presente nel sito B. Nel frattempo, un utente del sito B può accedere a un file che risiede nel sito A. In generale, la **condivisione delle risorse** di un sistema distribuito offre meccanismi per la condivisione dei file in siti remoti, per l'elaborazione delle informazioni in una base di dati distribuita, per la stampa dei file in siti remoti, per l'uso remoto di hardware specializzato (come un supercomputer), e per eseguire altre operazioni.

17.1.2 Velocizzazione delle elaborazioni

Se una particolare computazione si può dividere in componenti eseguibili in modo concorrente, un sistema distribuito permette di ripartire queste attività elaborative tra i diversi siti; i componenti si possono eseguire in modo concorrente determinando una **velocizzazione dell'elaborazione**. Inoltre, se un particolare sito è attualmente sovraccarico, alcuni job si possono spostare in altri siti con carico inferiore. Lo spostamento dei lavori d'elaborazione si chiama **condivisione del carico** (*load sharing*) o **migrazione dei job**. Una condivisione del carico automatica, in cui il sistema operativo distribuito sposta i lavori autonomamente, non è ancora comune nei sistemi commerciali.

17.1.3 Affidabilità

Se cade un sito di un sistema distribuito, i siti restanti possono continuare a funzionare, offrendo al sistema una maggiore affidabilità. Se il sistema distribuito è formato da più grandi sistemi elaborativi autonomi, vale a dire calcolatori d'uso generale, il guasto di uno di loro non influisce sugli altri sistemi elaborativi. Se invece il sistema è formato da piccoli calcolatori, ciascuno dei quali è responsabile di una funzione cruciale per il sistema (come il server web oppure il file system) un singolo guasto può fermare il funzionamento di tutto il sistema. In generale, con un opportuno livello di ridondanza (sia dei dispositivi sia dei dati) il sistema può continuare a funzionare anche se qualcuno dei siti si guasta.

Il guasto di un sito deve essere rilevato dal sistema, e per superare tale situazione si devono intraprendere azioni adeguate. Il sistema non deve più utilizzare i servizi di quel sito; inoltre, se la funzione del sito guasto può essere rilevata da un altro sito, il sistema deve assicurare che il trasferimento della funzione avvenga correttamente. Infine, quando il sito guasto viene ripristinato o riparato devono essere disponibili meccanismi che lo reintegrino facilmente nel sistema.

17.1.4 Comunicazione

Se più siti sono collegati per mezzo di una rete di comunicazione, gli utenti dei diversi siti hanno la possibilità di scambiarsi informazioni. A un livello basso, tra i sistemi si scambiano **messaggi** in modo simile a quello dei messaggi in un singolo calcolatore (Paragrafo 3.4). Disponendo dello scambio di messaggi, tutti i servizi di livello superiore disponibili nei sistemi centralizzati si possono estendere in modo da adattarsi al sistema distribuito. Tali funzioni comprendono il trasferimento di file, il login, la posta elettronica e le chiamate di procedure remote (RPC).

Il vantaggio dato da un sistema distribuito consiste nel fatto che queste funzioni si possono eseguire su grandi distanze. Due persone che lavorano in siti geograficamente diversi possono per esempio collaborare a un progetto. Trasferendo i file del progetto, entrando nei rispettivi sistemi remoti per eseguire programmi e comunicando via mail per coordinare il lavoro, gli utenti riducono al minimo i limiti impliciti in un lavoro a lunga distanza. Questo libro è stato scritto collaborando in tal modo.

I vantaggi dei sistemi distribuiti hanno determinato una tendenza, tra le industrie, verso la *riduzione delle dimensioni dei calcolatori* (**downsizing**): molte aziende sostituiscono i loro mainframe con reti di stazioni di lavoro o PC. Le aziende ottengono un miglior rapporto tra prezzo e prestazioni, una maggiore flessibilità nella dislocazione delle risorse, migliori interfacce utente e una manutenzione più semplice.

17.2 Tipi di sistemi operativi distribuiti

In questo paragrafo si descrivono due categorie generali di sistemi operativi orientati alle reti: i sistemi operativi di rete e i sistemi operativi distribuiti. I sistemi operativi di rete sono più semplici da realizzare, ma spesso sono più difficili da usare dei sistemi operativi distribuiti, che offrono più servizi.

17.2.1 Sistemi operativi di rete

Un **sistema operativo di rete** offre un ambiente nel quale gli utenti, che sanno della presenza di più calcolatori, possono accedere alle risorse remote iniziando una sessione di lavoro sugli appropriati calcolatori remoti oppure trasferendo dati dai calcolatori remoti al proprio. Al giorno d'oggi tutti i sistemi operativi general-purpose e persino i sistemi operativi mobili come Android e iOS sono sistemi operativi di rete.

17.2.1.1 Login remoto

Una funzione importante dei sistemi operativi di rete è consentire agli utenti di effettuare un login remoto. A tale scopo la rete Internet fornisce la funzione `ssh`. Per illustrare questa funzione si supponga che un utente, al Westminster College, voglia compiere alcune elaborazioni servendosi di `cs.utexas.edu`, un calcolatore che si trova alla Yale University. Perché ciò sia possibile, l'utente deve avere un account valido sul calcolatore remoto. Per effettuare il login remoto l'utente usa il comando

```
ssh cs.yale.edu
```

Questo comando fa sì che si realizzi una connessione socket criptata tra il calcolatore locale del Westminster College e il calcolatore `cs.yale.edu`. Stabilita tale connessione, il software di rete crea un collegamento bidirezionale, trasparente, tale che tutti i caratteri inseriti dall'utente siano inviati a un processo in `cs.yale.edu`, e tutti i dati emessi da questo processo siano rispediti all'utente. Il processo nel calcolatore remoto chiede all'utente il suo nome utente e la relativa password; ricevute le corrette informazioni, il processo agisce per conto dell'utente, che può compiere le proprie elaborazioni servendosi del calcolatore remoto come ogni utente locale.

17.2.1.2 Trasferimento di file remoti

Un'altra tra le funzioni principali di un sistema operativo di rete consiste nel fornire un meccanismo per il **trasferimento di file remoti** tra calcolatori. In un ambiente di questo tipo ogni calcolatore mantiene il proprio file system locale. Se un utente in un sito (per esempio, `cs.uvm.edu`) vuole accedere a un file che si trova in un altro cal-

colatore (per esempio, cs.yale.edu), deve copiare esplicitamente il file dal calcolatore della Yale University al calcolatore dell'University of Vermont.

La rete Internet offre meccanismi, il programma FTP (*file transfer protocol*) e il più sicuro SFTP (*secure file transfer protocol*), che permettono di compiere tale trasferimento. Si supponga che un utente di cs.uvm.edu voglia copiare il programma Java Server .java che si trova in cs.yale.edu; l'utente deve prima invocare il programma **sftp**:

```
sftp cs.yale.edu
```

Quindi il programma richiede il nome utente e la relativa password; una volta che il programma ha ricevuto le corrette informazioni, l'utente deve connettersi alla directory in cui si trova il file **Server.java** e copiare il file con il comando

```
get Server.java
```

In questo schema la locazione del file non è trasparente per l'utente; gli utenti devono sapere esattamente dove si trova ogni file. Inoltre non esiste una reale condivisione di file, poiché un utente può solo *copiare* un file da un sito all'altro; possono quindi esistere più copie dello stesso file con conseguente spreco di spazio e, nel caso di differenti modifiche, di incoerenza tra le copie.

In questo esempio l'utente dell'University of Vermont deve avere il permesso d'accesso a cs.yale.edu. Comunque l'FTP offre un'opzione che consente anche a un utente non accreditato nel calcolatore della Yale di copiare file in modo remoto; questa possibilità si ottiene col metodo detto **FTP anonimo** che funziona come segue. Il file da copiare, cioè **Server.java**, viene messo in una specifica directory, per esempio **ftp**, con la protezione impostata in modo da consentire a chiunque la lettura del file. Un utente che voglia copiare il file usa il comando **ftp**, fornendo il nome *anonymous* alla richiesta del nome utente e una password qualunque.

Una volta che l'accesso anonimo è stato effettuato, il sistema deve prendere precauzioni che assicurino che tale utente, parzialmente autorizzato, possa accedere soltanto ai file appropriati. Generalmente si consente l'accesso ai soli file presenti nell'albero di directory dell'utente “*anonymous*”; tutti i file in esso presenti sono accessibili a ogni utente anonimo, sottoposto all'usuale schema di protezione impiegato nel calcolatore. Gli utenti anonimi non possono però accedere ai file presenti fuori da tale albero di directory.

Il meccanismo FTP è realizzato in modo simile a **ssh**. Un demone nel sito remoto verifica le richieste di connessione alla porta FTP del sistema; si convalida l'accesso e si consente all'utente di richiedere a distanza l'esecuzione dei comandi di trasferimento. Diversamente dal demone **ssh**, che esegue per l'utente ogni comando, il demone FTP risponde soltanto a un insieme predefinito di comandi riguardanti i file, tra i quali i seguenti:

- **get** trasferisce un file da un calcolatore remoto a uno locale;
- **put** trasferisce un file da un calcolatore locale a uno remoto;
- **ls** o **dir** elencano i file presenti nella directory corrente nel calcolatore remoto;
- **cd** cambia la directory corrente nel calcolatore remoto.

Inoltre vari comandi consentono di cambiare i modi di trasferimento (per file binari o ASCII) e per determinare lo stato della connessione.

Sia `ssh` sia l'`FTP` richiedono un cambio di paradigma da parte dell'utente. `FTP` richiede che l'utente conosca un insieme di comandi completamente diverso dai normali comandi del sistema operativo. Nel caso di `ssh` l'utente deve conoscere i comandi appropriati sul sistema remoto. Per esempio, un utente su una macchina Windows che si connette in remoto a una macchina UNIX durante la sessione `ssh` deve passare comandi UNIX (in rete, una **sessione** è una completa attività di comunicazione, che inizia spesso con un login per l'autenticazione e termina con una disconnessione che interrompe la comunicazione). Ovviamente gli utenti troverebbero più conveniente non essere obbligati a utilizzare un diverso insieme di comandi. I sistemi operativi distribuiti sono progettati per migliorare questa situazione.

17.2.2 Sistemi operativi distribuiti

In un sistema operativo distribuito gli utenti accedono alle risorse remote nello stesso modo in cui accedono alle risorse locali. I trasferimenti di dati e processi tra siti sono sotto il controllo del sistema operativo distribuito.

17.2.2.1 Migrazione dei dati

Si supponga che un utente del sito *A* voglia accedere a dati (per esempio, contenuti in un file) che risiedono nel sito *B*. Il sistema può trasferire i dati impiegando uno fra due metodi di base. Un approccio alla **migrazione dei dati** consiste nel trasferire tutto il file al sito *A*. Da questo punto in poi l'intero accesso al file è di tipo locale. Quando l'utente non ha più necessità di accedere al file, se il file era stato modificato, si ritrasmette una sua copia al sito *B*. Si devono trasferire tutti i dati anche se il file ha subito solo una lieve modifica. Questo metodo, che si può considerare come un sistema `FTP` automatico, era usato nel file system Andrew, ma è stato ritenuto troppo inefficiente.

L'altro approccio consiste nel trasferire al sito *A* solo le parti del file immediatamente necessarie. Se in seguito è richiesta un'altra sua parte, si esegue un nuovo trasferimento. Quando l'utente non avrà più bisogno di accedere al file, si ritrasmetterà al sito *B* qualsiasi sua parte sia stata modificata (si noti l'analogia con la paginazione su richiesta). Questo metodo è adoperato dal protocollo `NFS` (*network file system*) di Sun Microsystems (Paragrafo 12.8) e dalle più recenti versioni di Andrew. Anche il protocollo `SMB` di Microsoft (conosciuto anche come `CIFS`, *common internet file system*) consente la condivisione dei file in una rete. `SMB` è descritto nel Paragrafo 19.6.2.1 (presente sul sito web del volume).

Per accedere a una piccola parte di un file di grandi dimensioni, ovviamente conviene servirsi del secondo metodo; quando è necessario accedere a parti estese del file, invece, conviene copiarlo tutto. In entrambi i metodi, la migrazione dei dati comprende assai più del mero trasferimento dei dati da un sito all'altro. Se i due siti coinvolti nel trasferimento non fossero direttamente compatibili (per esempio, se impiegassero codici dei caratteri diversi o rappresentassero i numeri interi con un diverso numero o ordinamento dei bit), il sistema dovrebbe eseguire anche diverse traduzioni dei dati.

17.2.2.2 Migrazione delle computazioni

In alcune circostanze si può voler trasferire le computazioni anziché i dati; questo metodo si chiama **migrazione delle computazioni**. Si consideri, per esempio, un job che necessiti di accedere a diversi file di grandi dimensioni che risiedono in diversi siti per ottenere un sommario di tutti quei file. È più conveniente accedere ai file nei siti in cui risiedono e riportare i risultati al sito che ha iniziato il calcolo. In generale s'impiega il comando remoto se il tempo di trasferimento dei dati è maggiore del tempo d'esecuzione del comando remoto.

Una computazione di questo genere si può eseguire in diversi modi. Si supponga che il processo P voglia accedere a un file presente nel sito A . L'accesso al file si esegue nel sito A e può essere avviato da una RPC. Una RPC usa **protocolli di rete** per far eseguire una procedura in un sistema remoto (Paragrafo 3.6.2). Il processo P invoca una procedura predefinita nel sito A ; tale procedura esegue adeguatamente il proprio compito e riporta i risultati a P .

In alternativa il processo P può inviare un *messaggio* al sito A ; il sistema operativo di A crea un nuovo processo Q la cui funzione è quella di eseguire il compito designato; quando il processo Q termina l'esecuzione, invia il risultato richiesto a P per mezzo del sistema di messaggistica. In questo schema il processo P si può eseguire in modo concorrente al processo Q e, in effetti, più processi si possono eseguire in modo concorrente in diversi siti.

Entrambi i metodi si possono usare per accedere a più file residenti in vari siti. Una RPC può causare l'invocazione di un'altra RPC, o addirittura il trasferimento di messaggi a un altro sito. Analogamente il processo Q , durante la propria esecuzione, può inviare un messaggio a un altro sito, che a sua volta crea un altro processo. Questo processo può inviare un messaggio di risposta a Q , oppure ripetere il ciclo.

17.2.2.3 Migrazione dei processi

Una logica estensione della migrazione delle computazioni è la **migrazione dei processi**. Quando si esegue un processo, non sempre l'esecuzione si svolge nel sito nel quale è stata avviata. L'intero processo, o parti di esso, si possono eseguire in siti diversi. Questo schema può essere utilizzato per diversi scopi.

- **Bilanciamento del carico.** I processi (o sottoprocessi) si possono distribuire tramite la rete per equilibrare il carico di lavoro.
- **Velocizzazione dell'elaborazione.** Se un singolo processo si può dividere in un certo numero di sottoprocessi che si possono eseguire in modo concorrente in siti diversi, è possibile ridurre il tempo di completamento totale.
- **Preferenze hardware.** Il processo può avere caratteristiche che lo rendono più adatto a un'esecuzione in un'unità d'elaborazione specializzata (per esempio, un'inversione di matrice in un processore vettoriale anziché in un'ordinaria CPU).
- **Preferenze software.** Il processo può richiedere un programma disponibile solo in un particolare sito, ma o il programma non può essere spostato, o è meno costoso spostare il processo.

- **Accesso ai dati.** Come nella migrazione delle computazioni, se i dati da usare nella computazione sono numerosi, l'esecuzione remota del processo può essere più efficiente del trasferimento dei dati.

Per spostare i processi in una rete di calcolatori si possono usare due metodi complementari. Col primo, il sistema può cercare di nascondere al client la migrazione del processo. Il client non ha quindi bisogno di codificare il suo programma esplicitamente per realizzare la migrazione. Questo schema si usa normalmente per ottenere un bilanciamento del carico e una velocizzazione delle elaborazioni tra sistemi omogenei, poiché essi non richiedono un'interazione con l'utente nell'esecuzione remota dei programmi.

L'altro metodo consiste nel permettere (o richiedere) all'utente di specificare esplicitamente come il processo debba migrare. Questo metodo si usa normalmente nelle situazioni in cui si deve trasferire un processo per soddisfare una preferenza hardware o software.

Il lettore si è probabilmente accorto del fatto che il World Wide Web ha molte caratteristiche di un ambiente d'elaborazione distribuito. Di certo offre la migrazione dei dati (tra un server web e un client) così come la migrazione delle computazioni: un client può per esempio attivare un'operazione su una base di dati di un server web. Infine, con Java, Javascript e linguaggi simili, fornisce una forma di migrazione dei processi: le *Java applet* e gli script Javascript sono inviati dal server al client dove vengono eseguiti. Un sistema operativo di rete offre la maggior parte di queste funzioni, ma un sistema operativo distribuito le integra in maniera uniforme e le rende più facilmente accessibili. Il risultato è uno strumento potente e facile da usare, una delle ragioni dell'enorme crescita del Web.

17.3 Tipi di reti

I due tipi di rete fondamentali sono le reti locali e le reti geografiche. La principale differenza tra le due consiste nella loro distribuzione geografica. Le **reti locali** (*local-area network*, LAN) comprendono un certo numero di host distribuiti su piccole aree, come un unico edificio o alcuni edifici adiacenti. Le **reti geografiche** (*wide-area network*, WAN), invece, comprendono sistemi distribuiti su vaste aree geografiche (per esempio, gli Stati Uniti). Queste differenze implicano notevoli variazioni nella velocità e nell'affidabilità delle reti di comunicazione, e si riflettono nella progettazione dei sistemi operativi distribuiti.

17.3.1 Reti locali

Le reti locali (LAN) sono nate nei primi anni '70 con lo scopo di sostituire i sistemi basati su mainframe. Molte imprese trovano più economico avere tanti piccoli calcolatori, ciascuno con le proprie applicazioni autonome, anziché un unico sistema di grandi dimensioni. Poiché è probabile che ogni piccolo calcolatore necessiti di una serie completa di dispositivi periferici, come dischi e stampanti, e poiché è probabile

che esista qualche forma di condivisione di dati in un'unica società, fu naturale collegare questi piccoli sistemi in una rete.

Come si è detto, generalmente le LAN sono progettate per coprire piccole aree, e di solito sono usate in ambienti di uffici. In questi sistemi tutti i siti sono vicini fra loro, e di conseguenza i collegamenti tendono ad avere una maggior velocità e una minor frequenza di errori rispetto alle reti geografiche.

Nelle reti locali i collegamenti sono in genere realizzati con doppini (*twisted pair*) o con fibre ottiche. Le configurazioni più diffuse sono le reti a stella. In una rete a stella i nodi sono connessi a uno o più switch e gli switch sono collegati tra loro, permettendo così a ogni coppia di nodi di comunicare. La velocità di comunicazione varia da 1 **megabit** al secondo per reti come Appletalk, reti a raggi infrarossi e Bluetooth fino ai 40 **gigabit** al secondo per le reti Ethernet più veloci. **Ethernet 10BaseT** ha una velocità di 10 megabit al secondo, mentre **Ethernet 100BaseT** e **Ethernet 1000BaseT** consentono velocità di 100 megabit e un gigabit al secondo, rispettivamente, su doppino in rame. L'utilizzo di cablaggi in fibra ottica è in crescita e permette di raggiungere velocità di comunicazione più alte e coprire distanze maggiori rispetto al rame.

Una LAN tipica è composta da diversi calcolatori, dai mainframe ai laptop e altri dispositivi portatili, vari dispositivi periferici condivisi (come stampanti laser o array di storage), e uno o più **router** (unità d'elaborazione specializzate per le reti) che danno accesso ad altre reti (Figura 17.2). Per costruire le LAN comunemente si usa la tecnologia Ethernet. In una rete Ethernet non c'è un controllore centrale poiché è una rete con bus multiaccesso, quindi si possono aggiungere facilmente nuovi calcolatori alla rete. Il protocollo Ethernet è definito dallo standard IEEE 802.3.

Il segnale wireless è sempre più utilizzato per la progettazione di reti locali. La tecnologia wireless (o WiFi) permette di costruire una rete utilizzando soltanto un

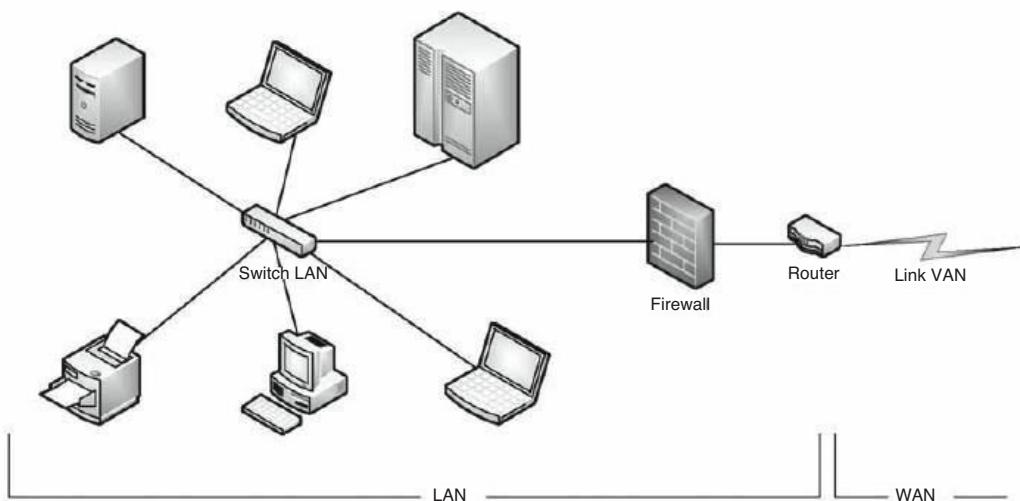


Figura 17.2 Rete locale (LAN).

router wireless per trasmettere segnali tra gli host. Ogni host ha un trasmettitore e un ricevitore wireless che utilizza per partecipare alla rete. Uno svantaggio delle reti wireless è loro velocità. Mentre la velocità di una rete Ethernet arriva spesso a 1 gigabit al secondo, le reti WiFi offrono in genere una velocità più bassa. Ci sono diversi standard IEEE per le reti wireless: 802.11g può teoricamente funzionare a 54 megabit al secondo, anche se in pratica la velocità è spesso meno della metà di quella teorica; il recente standard 802.11n fornisce teoricamente una velocità di molto superiore a quella dell'802.11g, ma nella pratica le reti 802.11n non superano la velocità di 75 megabit al secondo. La velocità di trasferimento su reti wireless è fortemente influenzata dalla distanza tra il router wireless e l'host e dalle interferenze subite dal segnale. Guardando agli aspetti positivi, le reti wireless hanno un vantaggio fisico sulle reti Ethernet cablate, perché non richiedono alcun cablaggio per la connessione di host comunitanti. Ciò rende le reti wireless molto diffuse in abitazioni e aziende, nonché in aree pubbliche quali biblioteche, Internet café, campi sportivi, mezzi pubblici e aerei.

17.3.2 Reti geografiche

Le reti geografiche (WAN) sono nate alla fine degli anni '60 come progetto di ricerca accademica per fornire un sistema di comunicazione efficiente tra siti, che permettesse a un'ampia comunità di utenti di condividere in modo conveniente ed economico le risorse elaborative. La prima WAN progettata e sviluppata è stata l'*Arpanet*, nata nel 1968. Partita come rete sperimentale a quattro siti, è cresciuta fino a diventare una “rete di reti” mondiale, l'Internet, che comprende, fra l'altro, milioni di sistemi elaborativi.

Poiché i siti di una WAN sono fisicamente distribuiti su una vasta zona geografica, i collegamenti di comunicazione sono relativamente lenti e inaffidabili. Questi collegamenti possono essere linee telefoniche, collegamenti a microonde, linee dedicate in affitto, cavi ottici, onde radio e canali via satellite. Questi collegamenti sono controllati da speciali **unità d'elaborazione di comunicazione** o **communication processor** (Figura 17.3), comunemente chiamati **gateway** o, più spesso, **router**, che definiscono l'interfaccia attraverso la quale i siti comunicano sulla rete e gestiscono il trasferimento delle informazioni tra i diversi siti.

Si consideri, per esempio, la WAN Internet, che offre ai calcolatori in siti geograficamente separati la possibilità di comunicare tra loro. Generalmente, i calcolatori differiscono per velocità, tipo di CPU, sistema operativo e così via. I calcolatori normalmente fanno parte di LAN, che a loro volta sono collegate alla rete Internet per mezzo di reti regionali. Le reti regionali sono interconnesse tramite router, descritti nel Paragrafo 17.4.2, per formare la rete mondiale. Negli Stati Uniti i collegamenti tra le reti impiegano talvolta un servizio telefonico, chiamato T1, che fornisce una velocità di trasferimento di 1,544 megabit al secondo su una linea noleggiata. Per siti che richiedono un accesso più veloce, i T1 sono raccolti in unità di più T1 che lavorano in parallelo per fornire una maggiore produttività. Per esempio, un T3 è composto da 28 connessioni T1 e ha una velocità di trasferimento di 45 megabit al secondo.

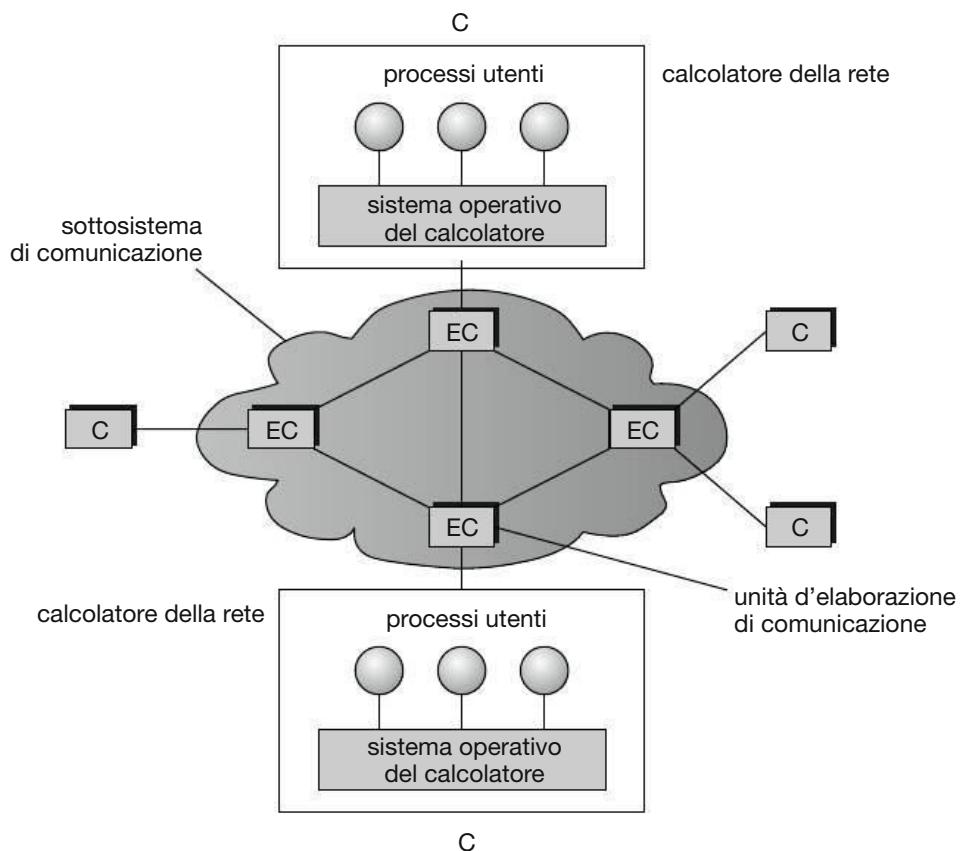


Figura 17.3 Processori di comunicazione in una rete geografica.

Connessioni quali OC-12 sono piuttosto comuni e forniscono 622 megabit al secondo. Le aree residenziali possono connettersi a Internet via telefono, via cavo o tramite un Internet service provider specializzato che installa dei router per collegare le residenze ai servizi centrali. Ci sono ovviamente altre WAN oltre a Internet. Una società può creare la sua WAN privata per aumentare la sicurezza, le prestazioni o l'affidabilità.

Come accennato, le WAN sono generalmente più lente delle reti LAN, anche se le connessioni backbone delle WAN che collegano le principali città possono avere velocità di trasferimento di oltre 40 gigabit al secondo. Spesso le WAN e le LAN sono interconnesse ed è difficile dire dove finisce l'una e comincia l'altra. Si consideri la rete dati cellulare. I telefoni cellulari sono utilizzati sia per comunicazioni vocali sia per connessioni dati e, in una determinata zona, si collegano tramite onde radio a una cella che contiene ricevitori e trasmettitori. Questa parte della rete è simile a una LAN tranne per il fatto che i telefoni cellulari non comunicano tra loro, a meno che due persone in comunicazione tra loro siano collegate alla stessa cella. Più spesso, le celle sono collegate ad altre celle e a hub che collegano le celle di comunicazione alle linee terrestri o ad altri mezzi e instradano i pacchetti verso le loro destinazioni. Questa parte della rete è più simile a una WAN. Una volta che la cella appropriata riceve i pacchetti, si utilizzano i trasmettitori per inviarli al destinatario corretto.

17.4 Struttura della comunicazione

Dopo aver trattato gli aspetti fisici della comunicazione tramite reti si può considerarne il funzionamento interno. Il progettista di una rete di comunicazione deve considerare quattro problemi fondamentali.

- **Naming e risoluzione dei nomi.** Come vengono individuati i processi ai fini della comunicazione?
- **Strategie d'instradamento.** Attraverso quali vie si trasmettono i messaggi attraverso la rete?
- **Strategie riguardanti l'invio dei pacchetti.** I pacchetti vengono inviati individualmente o in sequenza?
- **Strategie di connessione.** Con quali regole due processi si scambiano una sequenza di messaggi?

Questi argomenti sono sviluppati nei paragrafi seguenti.

17.4.1 Naming e risoluzione dei nomi

Il primo problema nelle comunicazioni tramite rete è il modo di indicare i nomi (*naming*) dei sistemi nella rete. Affinché due processi in esecuzione in due siti differenti *A* e *B* possano scambiarsi informazioni devono poter fare riferimento l'uno all'altro. All'interno di un calcolatore ciascun processo ha un identificatore di processo, quindi i messaggi si possono indirizzare con l'identificatore del processo. Poiché i sistemi presenti nella rete non condividono memoria, inizialmente un host non ha alcuna informazione riguardante processi in esecuzione su altri sistemi.

Per risolvere questo problema i processi di un sistema remoto sono generalmente identificati dalla coppia *<nome del calcolatore, identificatore>*, dove *nome del calcolatore* (host) è di solito un nome unico all'interno della rete, e *identificatore* può essere un identificatore di processo o un altro numero unico all'interno del calcolatore. Un *nome del calcolatore* è di solito un identificatore alfanumerico, anziché un numero, per renderne più semplice l'utilizzo da parte degli utenti. Per esempio, il sito *A* potrebbe contenere calcolatori con i nomi *homer, marge, bart* e *lisa*; è sicuramente più facile ricordare il nome *bart* rispetto al numero 12814831100.

Sebbene i nomi siano comodi per gli esseri umani, nei calcolatori è preferibile usare i numeri, sia per motivi di velocità sia di semplicità. Per questa ragione ci deve essere un meccanismo per la **risoluzione** del nome del calcolatore nel corrispondente identificatore, che descrive il sistema di destinazione ai dispositivi della rete. Questo meccanismo di risoluzione è simile all'associazione (*binding*) tra nomi e indirizzi definita durante compilazione, linking, caricamento ed esecuzione di un programma (Capitolo 8). Nel caso dei nomi di calcolatori esistono due possibilità; nella prima ciascun calcolatore può avere un file contenente i nomi e gli indirizzi di tutti i calcolatori raggiungibili tramite la rete (situazione simile al binding nella fase di compilazione). Il problema di questo modello è che l'aggiunta o la rimozione di un calco-

latore dalla rete richiede l'aggiornamento in tutti i calcolatori dei suddetti file. La seconda possibilità consiste nel distribuire le informazioni tra i sistemi connessi in rete. La rete deve quindi adoperare un protocollo per distribuire e recuperare queste informazioni. Questo schema corrisponde all'associazione tra nomi e indirizzi nella fase d'esecuzione. Il primo metodo era originariamente adottato nella rete Internet; col crescere della rete, si rese necessario adottare il secondo metodo chiamato **DNS** (*domain name system*).

Il DNS specifica la struttura di naming dei calcolatori e la risoluzione dei nomi in indirizzi. I calcolatori della rete Internet si indirizzano logicamente con un nome composto da più parti, detto **nome di dominio**. Le parti di un nome di dominio progrediscono dalla parte più specifica a quella più generale dell'indirizzo, con i diversi campi separati da un punto: bob.cs.brown.edu si riferisce al calcolatore bob del *Department of Computer Science* della Brown University nel dominio di primo livello *edu*. (Altri domini del primo livello comprendono *com*, per usi commerciali, e *org*, per le organizzazioni, oltre ai domini indicanti i Paesi connessi alla rete, per i sistemi identificati a seconda del Paese di appartenenza e non dalle loro finalità.). In generale, il sistema risolve l'indirizzo esaminando gli elementi del nome del calcolatore nell'ordine inverso. Per ciascun elemento c'è un **server dei nomi** – semplicemente un processo in un sistema – che accetta un nome e risponde con l'indirizzo del server dei nomi responsabile della risoluzione di quel nome. Come passo finale si interroga il server dei nomi associato al calcolatore in questione, che provvederà a riportare l'indirizzo del calcolatore. Per esempio, una richiesta di connessione a *bob.cs.brown.edu* effettuata da un processo sul sistema A vedrà eseguiti i seguenti passi.

1. La libreria di sistema o il kernel del sistema A inoltra una richiesta al server dei nomi del dominio *edu*, richiedendo l'indirizzo del server dei nomi di *brown.edu*. Per poter essere interrogato, il server dei nomi di *edu* deve avere un indirizzo noto a priori.
2. Il server dei nomi di *edu* riporta l'indirizzo del calcolatore in cui risiede il server dei nomi di *brown.edu*.
3. Il sistema A interroga il server dei nomi situato a questo indirizzo riguardo al dominio *cs.brown.edu*.
4. Si ottiene in risposta un indirizzo. Si invia una richiesta a tale indirizzo per risolvere *bob.cs.brown.edu* e si ottiene finalmente come risposta l'identificatore del calcolatore sotto forma di **indirizzo Internet** di quel sistema (per esempio, *128.148.31.100*).

Questo protocollo può sembrare inefficiente, ma ciascun server dei nomi mantiene cache locali per accelerare il processo, memorizzando gli indirizzi IP che sono già stati risolti. Naturalmente il contenuto di queste cache deve essere aggiornato col passare del tempo, poiché sia il server dei nomi sia il suo indirizzo potrebbero essere cambiati. In pratica questo servizio è così importante da indurre l'introduzione di numerose ottimizzazioni e molte protezioni nel protocollo. Si consideri che cosa accadrebbe nel caso di una caduta del server dei nomi primario *edu*: si potrebbe perdere

la capacità di risolvere gli indirizzi dei calcolatori del dominio `edu`, che diventerebbero tutti irraggiungibili. La soluzione consiste nell’impiego di server dei nomi di backup, nei quali duplicare il contenuto dei server dei nomi primari.

Prima dell’introduzione dei server dei nomi, tutti i calcolatori connessi alla rete Internet avevano bisogno di copie di un file contenente i nomi e gli indirizzi di tutti i calcolatori della rete. Tutte le modifiche a questo file dovevano essere registrate presso un particolare sito, il sistema SRI-NIC, e periodicamente tutti i calcolatori dovevano prelevarvi la versione aggiornata di questo file per poter contattare nuovi sistemi o trovare i calcolatori che avevano cambiato indirizzo. Con il DNS, il server dei nomi di ciascun sito è responsabile dell’aggiornamento delle informazioni riguardanti il proprio dominio. Per esempio, il server dei nomi di `brown.edu` è responsabile di tutte le modifiche riguardanti i calcolatori della Brown University, e tali modifiche non devono essere notificate ad alcuno. Le ricerche del DNS recupereranno automaticamente le informazioni aggiornate, poiché contatteranno direttamente `brown.edu`. All’interno del dominio possono esistere sottodomini autonomi allo scopo di distribuire ulteriormente la responsabilità circa le variazioni dei nomi e degli identificatori dei calcolatori.

Java mette a disposizione la API necessaria per progettare programmi che associno nomi di dominio a indirizzi IP. Il programma della Figura 17.4 accetta in ingresso dalla riga di comando un nome di dominio (per esempio, “`bob.cs.brown.edu`”) e fornisce in uscita l’indirizzo IP della macchina, o un messaggio che indichi l’impossibilità di tradurre il nome. La classe Java `InetAddress` rappresenta un nome o un indirizzo IP; il suo metodo statico `getByName()` accetta come parametro un nome di dominio sotto forma di stringa, e restituisce il corrispondente `InetAddress`.

```
/**
 * Uso: java DNSLookUp <nome IP>
 * Esempio: java DNSLookUp hpe.pearsoned.it
 */
public class DNSLookUp {
    public static void main(String[] args) {
        InetAddress hostAddress;

        try {
            hostAddress = InetAddress.getByName(args[0]);
            System.out.println(hostAddress.getHostAddress());
        }
        catch (UnknownHostException uhe) {
            System.err.println("Unknown host: " + args[0]);
        }
    }
}
```

Figura 17.4 Programma Java che illustra l’uso di DNS.

Il programma invoca quindi il metodo `getHostAddress()`, la cui implementazione usa DNS per cercare l'indirizzo della macchina indicata.

In generale, il sistema operativo è responsabile di accettare dai propri processi messaggi destinati a *<nome del calcolatore, identificatore>* e di trasferire il messaggio al calcolatore appropriato. Il kernel del calcolatore di destinazione è quindi responsabile del trasferimento del messaggio al processo specificato dall'identificatore. Questo scambio di informazioni è tutt'altro che elementare, ed è descritto nel Paragrafo 17.4.4.

17.4.2 Strategie d'instradamento

In questo paragrafo si descrive il modo in cui si sceglie la via per trasmettere un messaggio di processo del sito *A* che vuole comunicare con un processo del sito *B*. Se tra *A* e *B* esiste un solo percorso fisico, il messaggio deve seguire necessariamente quel percorso, ma se i percorsi fisici da *A* a *B* sono più di uno esistono diverse possibilità d'instradamento. Ogni sito ha una **tabella d'instradamento** (*routing table*) che indica i percorsi che si possono seguire nel recapitare un messaggio ad altri siti. La tabella può contenere informazioni sulla velocità e sul costo dei diversi percorsi di comunicazione e, quando è necessario, può anche essere aggiornata manualmente oppure tramite programmi che scambiano informazioni d'instradamento. I tre schemi d'instradamento più diffusi quello **fisso**, **virtuale** e **dinamico**.

- **Instradamento fisso.** Un percorso da *A* a *B* viene determinato in anticipo e non cambia, se non per un guasto fisico che interrompe il percorso stesso. Generalmente si sceglie il percorso più breve, in modo da ridurre al minimo i costi della comunicazione.
- **Instradamento virtuale.** Si fissa un percorso da *A* a *B* per la durata di una **sessione**. Sessioni diverse con messaggi che vanno da *A* a *B* possono avere percorsi diversi. Una sessione può essere breve, come il trasferimento di un file, o lunga, come la durata di un login remoto.
- **Instradamento dinamico.** Il percorso da impiegare per inviare un messaggio dal sito *A* al sito *B* si sceglie al momento dell'invio del messaggio. Poiché la decisione viene presa dinamicamente, a messaggi distinti si possono assegnare percorsi diversi. Il sito *A* decide di inviare il messaggio al sito *C*, che a sua volta decide di inviare il messaggio al sito *D* e così via. Alla fine, un sito invia il messaggio a *B*. Generalmente, un sito invia un messaggio al sito connesso tramite il collegamento che in quel momento è meno usato.

Questi tre schemi hanno diversi pro e contro. L'instradamento fisso non si adatta a malfunzionamenti nelle linee di comunicazione o a cambiamenti del carico; in altre parole, se è stato fissato un percorso tra *A* e *B*, si devono inviare i messaggi lungo questo percorso anche se è interrotto o è più fortemente usato di altri possibili percorsi. Questo problema si può risolvere parzialmente impiegando l'instradamento virtuale, e si può evitare impiegando l'instradamento dinamico. L'uso dell'instradamento

fisso o dell’instradamento virtuale assicura che i messaggi da *A* a *B* arrivino nell’ordine in cui sono stati trasmessi. Con l’instradamento dinamico i messaggi possono arrivare in un altro ordine; questo problema si può risolvere assegnando a ogni messaggio un numero di sequenza.

L’instradamento dinamico è molto più complicato da impostare ed eseguire, ma è il modo migliore per gestire l’instradamento in ambienti complessi. Il sistema operativo UNIX offre sia l’instradamento fisso, da usare in calcolatori all’interno di reti semplici, sia l’instradamento dinamico per ambienti di rete complessi; è possibile anche una combinazione dei due metodi. Nell’ambito di un sito i calcolatori devono soltanto sapere come raggiungere il sistema che connette la rete locale ad altre reti (come reti aziendali o l’Internet); tale nodo è noto come **gateway**. Questi singoli calcolatori hanno un instradamento statico verso il gateway, che invece utilizza l’instradamento dinamico per raggiungere ogni calcolatore nel resto della rete.

Il router è processore di comunicazione che è responsabile dell’instradamento dei messaggi nella rete di calcolatori. Può essere un calcolatore dotato di programmi d’instradamento o un dispositivo specifico. In entrambi i casi deve avere almeno due connessioni di rete, altrimenti non avrebbe nessun posto dove instradare i messaggi. Un router decide se ogni dato messaggio deve essere passato dalla rete da cui viene ricevuto a un’altra rete a esso connessa. Tale decisione si prende esaminando l’indirizzo Internet di destinazione del messaggio: il router esamina le proprie tabelle per determinare la locazione del calcolatore di destinazione, o almeno della rete cui inviare il messaggio verso il calcolatore di destinazione. Nel caso dell’instradamento statico questa tabella viene modificata solo da aggiornamenti eseguiti manualmente (si carica un nuovo file nel router). Con l’instradamento dinamico, si usa un **protocollo d’instradamento** tra i router per informare gli stessi delle modifiche alla rete e consentire l’aggiornamento automatico delle rispettive tabelle d’instradamento.

Gateway e router sono di solito dispositivi hardware dedicati che eseguono codice firmware. Di recente l’instradamento è stato gestito da software in grado di dirigere diversi dispositivi di rete in maniera più intelligente rispetto a quanto un singolo router sia in grado di fare. Il software è indipendente dal dispositivo e permette dunque ai dispositivi di rete di fornitori diversi di collaborare più facilmente. Per esempio, lo standard OpenFlow consente agli sviluppatori di introdurre nuove funzionalità e migliorie nel funzionamento della rete disaccoppiando le decisioni di instradamento dei dati dai dispositivi di rete sottostanti.

17.4.3 Strategie riguardanti l’invio dei pacchetti

I messaggi sono di lunghezza molto variabile. Per semplificare la progettazione del sistema, di solito si realizza la comunicazione usando messaggi di lunghezza limitata detti **pacchetti, frame o datagrammi**. I messaggi contenuti in un singolo pacchetto si possono inviare al destinatario tramite una comunicazione **priva di connessione**. Essa può essere **inaffidabile**, in tal caso il mittente non ha alcuna garanzia che il pacchetto giunga a destinazione e non dispone di alcun modo per verificare se un pacchetto spedito sia andato perduto. In alternativa, il pacchetto può essere **affidabile**, in tal

caso il destinatario risponde al mittente con un altro pacchetto (di acknowledgment) come conferma dell'arrivo del pacchetto originale. (Naturalmente il pacchetto di conferma potrebbe andar perso e non giungere al mittente.) Se un messaggio è troppo lungo per essere codificato in un solo pacchetto, o se si devono ripetutamente scambiare pacchetti fra due interlocutori, si stabilisce una connessione al fine di permettere una comunicazione affidabile di più pacchetti.

17.4.4 Strategie di connessione

Una volta che i messaggi sono in grado di raggiungere le loro destinazioni, i processi possono instaurare **sessioni di comunicazione** per lo scambio di informazioni. Le coppie di processi che intendono comunicare attraverso la rete si possono connettere in diversi modi. I tre schemi più diffusi sono: **commutazione di circuito** (*circuit switching*), **di messaggio** (*message switching*) e **di pacchetto** (*packet switching*).

- **Commutazione di circuito.** Se due processi vogliono comunicare, si stabilisce tra loro un collegamento fisico permanente. Questo collegamento resta assegnato per tutta la durata della comunicazione e nessun altro processo può usarlo per tutto questo periodo, anche se esistono intervalli di tempo in cui i due processi non comunicano attivamente. Questo schema è simile a quello adoperato nel sistema telefonico. Una volta aperta una linea di comunicazione tra due parti, vale a dire che la parte A chiama la parte B, nessun altro può usare questo circuito finché non si termina esplicitamente la comunicazione (per esempio, quando una delle due parti riappende).
- **Commutazione di messaggio.** Se due processi vogliono comunicare, si stabilisce un collegamento temporaneo per la durata del trasferimento di un messaggio. I collegamenti fisici si assegnano dinamicamente secondo le necessità e tale assegnazione ha breve durata. Ogni messaggio è composto di un blocco di dati e di informazioni di sistema, come origine, destinazione, codici per la correzione degli errori, che permettono alla rete di comunicazione di recapitare messaggi correttamente. Questo schema è simile a quello postale: ogni lettera è considerata come un messaggio che contiene sia l'indirizzo di destinazione sia quello d'origine. Sullo stesso collegamento si possono inviare i messaggi di utenti diversi.
- **Commutazione di pacchetto.** Un messaggio logico si può dividere in un dato numero di pacchetti, ciascuno dei quali si può inviare alla sua destinazione separatamente, perciò deve contenere, oltre ai dati, un indirizzo di provenienza e uno di destinazione; ogni pacchetto può seguire un percorso diverso attraverso la rete. Quando arrivano a destinazione, i pacchetti devono essere ricomposti nei messaggi originari. Si noti che suddividere i dati in pacchetti, magari instradati separatamente e poi rimessi insieme al momento della destinazione, non causa effetti indesiderati sui dati, mentre spezzettare un segnale audio (una comunicazione telefonica, per esempio) può generare grande confusione se non si usano i necessari espedienti tecnici.

Anche questi schemi hanno diversi pro e contro. La commutazione di circuito richiede un sostanziale tempo d'impostazione e può sprecare banda della rete, ma esige un minor overhead per la spedizione di ogni messaggio. La commutazione di messaggi e pacchetti, invece, richiede un tempo di impostazione minore, ma un overhead maggiore per ogni messaggio. Inoltre, la commutazione di pacchetto implica la divisione di ogni messaggio in pacchetti e la successiva ricomposizione. La commutazione di pacchetto è il metodo più comunemente usato nelle reti di comunicazione dei dati, poiché fa il miglior uso della banda della rete.

17.5 Protocolli di comunicazione

Nel progettare una rete di comunicazione, è necessario considerare la complessità di coordinare operazioni asincrone che comunicano in un ambiente potenzialmente lento e soggetto a errori. Inoltre i sistemi nella rete devono accordarsi su un protocollo o su un insieme di protocolli per determinare i nomi dei calcolatori, individuare i calcolatori nella rete, stabilire le connessioni, e così via. Il problema della progettazione e della relativa realizzazione si può semplificare mediante una suddivisione in strati. Ciascuno strato in un sistema comunica con lo strato corrispondente negli altri sistemi. In genere ogni strato ha i propri protocolli e la comunicazione fra strati corrispondenti segue uno specifico protocollo. I protocolli si possono realizzare in hardware o possono essere programmi. La Figura 17.5 mostra, per esempio, uno schema delle comunicazioni logiche tra due calcolatori, con i tre strati di livello più basso realizzati

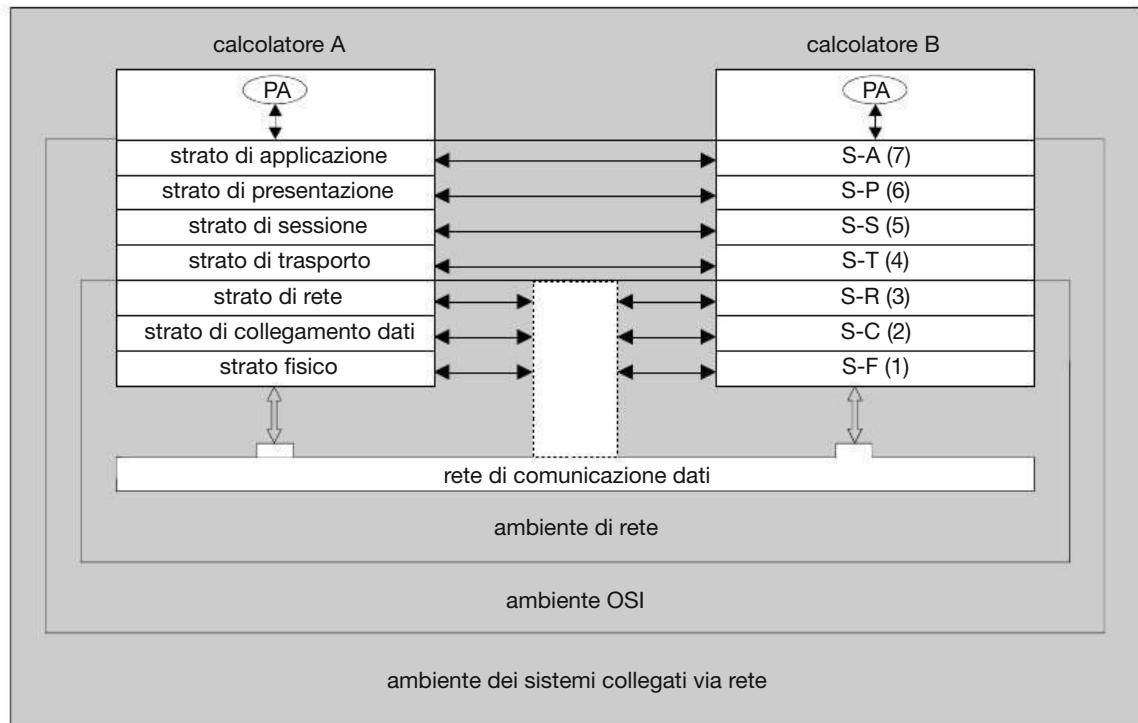


Figura 17.5 Due calcolatori comunicanti attraverso il modello di rete OSI.

in hardware. L'International Standards Organization ha creato il modello OSI per descrivere gli strati di una rete di comunicazione. Anche se questi strati non sono implementati nella pratica, sono utili per capire come lavora la rete a livello logico. Tali strati sono definiti come segue.

1. Strato 1: **Strato fisico.** È responsabile della gestione degli aspetti meccanici ed elettrici della trasmissione fisica di un flusso di bit. Nello strato fisico i sistemi di comunicazione devono accordarsi sulla rappresentazione elettrica delle cifre binarie 0 e 1, in modo che quando s'inviano i dati come flusso di segnali elettrici il ricevitore possa interpretare i dati correttamente come dati binari. Questo strato si realizza nell'hardware dei dispositivi di rete ed è il responsabile dell'invio dei bit.
2. Strato 2: **Strato di collegamento dati.** È responsabile delle funzioni di gestione dei *frame*, o di parti di pacchetti, comprese l'individuazione e la correzione degli errori che si sono verificati nello strato fisico. Spedisce frame fra siti collegati direttamente.
3. Strato 3: **Strato di rete.** È responsabile della fornitura dei collegamenti e dell'instradamento dei pacchetti nella rete di comunicazione, comprese la gestione degli indirizzi dei pacchetti in uscita, la decodifica degli indirizzi dei pacchetti in arrivo e la gestione delle informazioni d'instradamento per rispondere adeguatamente ai cambiamenti dei livelli del carico. I router operano in questo strato.
4. Strato 4: **Strato di trasporto.** È responsabile del trasferimento dei messaggi tra gli host, comprese la suddivisione dei messaggi in pacchetti, l'ordinamento dei pacchetti, il controllo del flusso per evitare la congestione.
5. Strato 5: **Strato di sessione.** È responsabile della realizzazione di sessioni, ossia di protocolli di comunicazione da processo a processo.
6. Strato 6: **Strato di presentazione.** È responsabile della risoluzione delle differenze di formato che si possono presentare tra diversi siti della rete, fra cui la conversione di caratteri e le modalità *half duplex-full duplex* (echo dei caratteri).
7. Strato 7: **Strato di applicazione.** È responsabile dell'interazione diretta con gli utenti. Questo strato tratta il trasferimento di file, i protocolli per la gestione dei login remoti, la posta elettronica, e i protocolli per le basi di dati distribuite.

La Figura 17.6 riassume la **pila di protocolli OSI** – un insieme di protocolli cooperanti – e illustra il flusso fisico dei dati. Come si è detto, dal punto di vista logico ciascuno strato di una pila di protocolli comunica con lo strato corrispondente negli altri sistemi. Fisicamente, però, un messaggio parte dallo strato di applicazione, o da sopra di esso, e passa attraverso ciascun livello inferiore. Ciascuno strato può modificare il messaggio e includere dati d'intestazione del messaggio per lo strato corrispondente del ricevente. Infine il messaggio raggiunge lo strato della rete di comunicazione e viene trasferito sotto forma di uno o più pacchetti (Figura 17.7). Lo strato di collegamento dati del sistema di destinazione riceve questi dati e il messaggio risale attraverso la pila di protocolli; viene analizzato, modificato e privato delle intestazioni; infine raggiunge lo strato di applicazione e può essere usato dal processo ricevente.

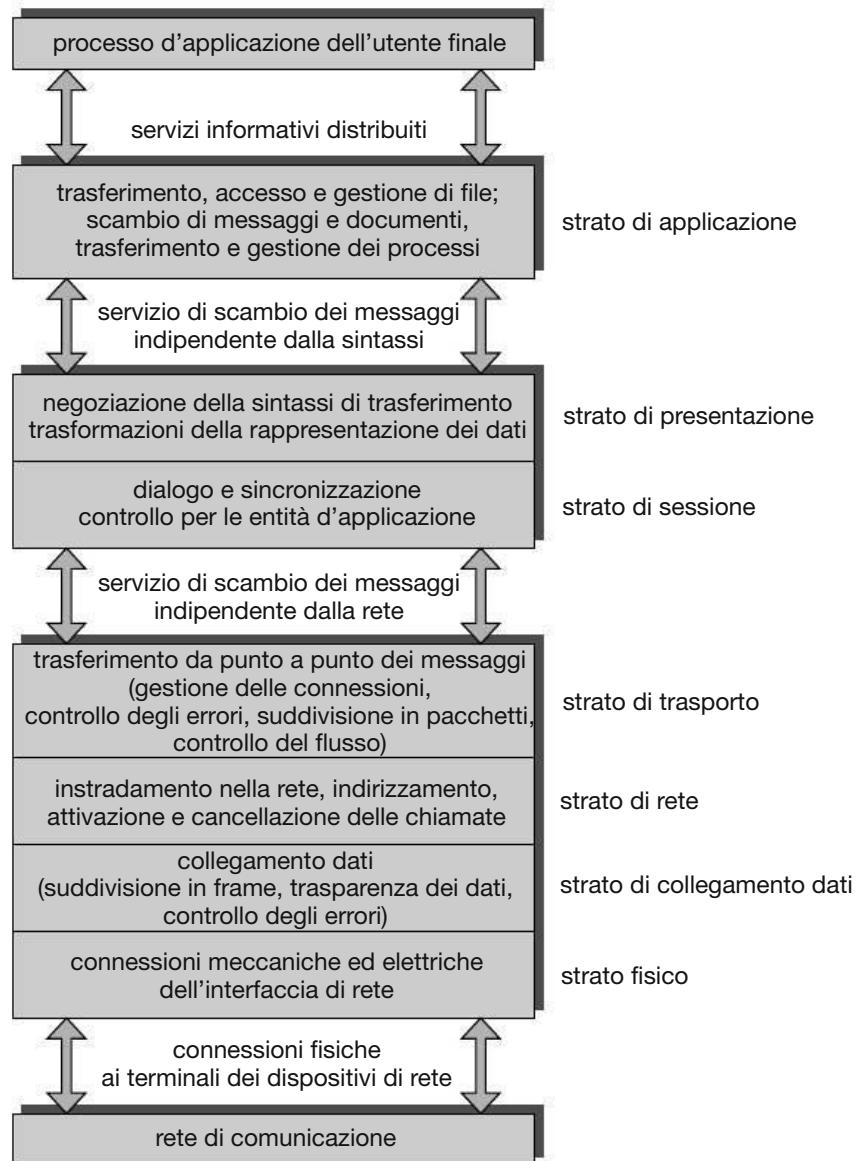


Figura 17.6 Pila di protocolli OSI.

Il modello OSI formalizza alcuni precedenti lavori svolti sui protocolli di rete; era stato sviluppato alla fine degli anni '70 e non è attualmente utilizzato; la pila di protocolli più ampiamente utilizzata è il modello TCP/IP, adottata da tutti i siti Internet. La pila di protocolli TCP/IP ha meno strati di quelli stabiliti dal modello OSI, e poiché combina diverse funzioni in ciascuno strato è, teoricamente, più difficile da realizzare ma più efficiente delle reti OSI. La relazione tra i modelli OSI e TCP/IP è illustrata dalla Figura 17.8.

Lo strato applicativo TCP/IP comprende diversi protocolli molto diffusi su Internet, come HTTP, FTP, Telnet, ssh, DNS e SMTP. Lo strato di trasporto comprende i protocolli **UDP** (*user datagram protocol*), inaffidabile e privo di connessione, e **TCP** (*transmission control protocol*), affidabile e orientato alla connessione. Il protocollo



Figura 17.7 Messaggio di rete OSI.

Internet (IP) è responsabile dell'instradamento dei datagrammi IP lungo Internet. Il modello TCP/IP non specifica formalmente lo strato di collegamento dati e fisico, permettendo al traffico TCP/IP di fluire su qualunque rete fisica. Nel Paragrafo 17.6 si considera il modello TCP/IP implementato su una rete Ethernet.

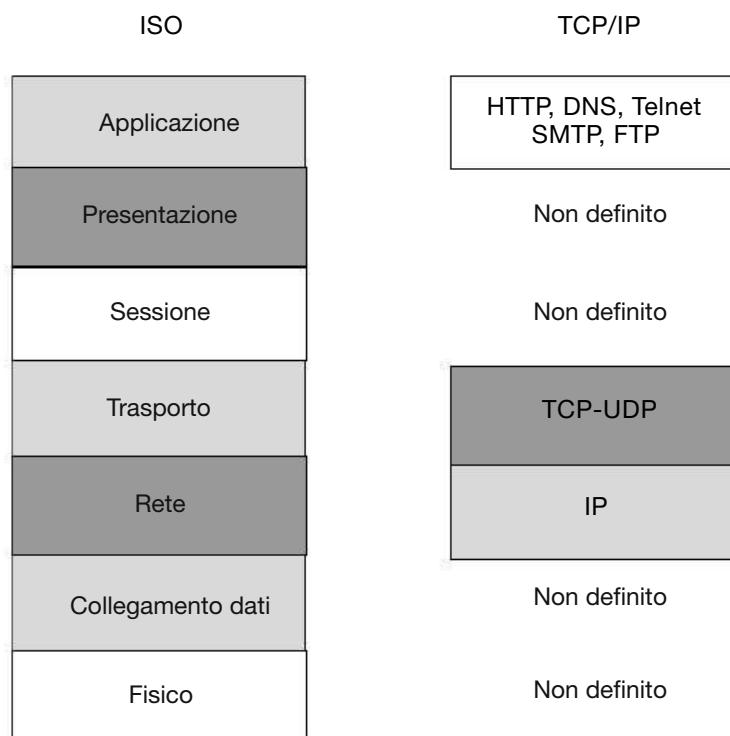


Figura 17.8 Pile di protocolli OSI e TCP/IP.

Nel disegno e nell’implementazione di un moderno protocollo di comunicazione la preoccupazione per la sicurezza assume una notevole importanza. Per una comunicazione sicura sono necessari sia un’autenticazione forte sia la crittografia. L’autenticazione forte assicura che il mittente e il destinatario di una comunicazione siano quelli desiderati. La crittografia protegge i contenuti della comunicazione da eventuali intercettazioni. L’autenticazione debole e la comunicazione in chiaro sono comunque, per diverse ragioni, ancora molto utilizzate. Quando la maggior parte dei protocolli oggi comunemente utilizzati fu progettata, la sicurezza era spesso meno importante rispetto alle prestazioni, alla semplicità e all’efficienza. L’eredità lasciata da queste scelte è tangibile ancora oggi: aggiungere sicurezza a infrastrutture esistenti è un’operazione difficile e complessa.

L’autenticazione forte richiede un protocollo di negoziazione a più passi o un dispositivo di autenticazione, rendendo così il protocollo più complesso. Le CPU moderne possono eseguire la crittografia in maniera efficiente, spesso offrendo istruzioni per velocizzarla, e i sistemi demandano spesso questa attività a processori dedicati, così che le prestazioni non siano compromesse. Le comunicazioni a lunga distanza possono essere rese sicure autenticando gli estremi della connessione e crittografando il flusso di pacchetti in una rete privata virtuale, come discusso nel Paragrafo 15.4.2. Le comunicazioni in una LAN restano in chiaro nella maggior parte dei casi, ma protocolli come NFS nella sua versione 4, che include un’autenticazione forte incorporata e la crittografia dei dati, possono aiutare a migliorare anche la sicurezza di una LAN.

17.6 Un esempio: TCP/IP

A questo punto è utile tornare al problema della risoluzione dei nomi affrontato nel Paragrafo 17.4.1 per esaminarne il funzionamento nella pila di protocolli TCP/IP nella rete Internet. Consideriamo poi le elaborazioni necessarie al trasferimento di un pacchetto tra due calcolatori appartenenti a due diverse reti Ethernet. La nostra descrizione è basata sui protocolli IPV4 che sono i più utilizzati al giorno d’oggi.

In una rete TCP/IP, ciascun calcolatore possiede un nome e un indirizzo Internet a 32 bit a esso associato. Entrambi i valori devono essere unici e sono suddivisi per facilitare la gestione dello spazio dei nomi. Il nome è gerarchico (come si spiega nel Paragrafo 17.4.1) e specifica sia il nome del calcolatore sia le organizzazioni a cui il calcolatore è associato. L’indirizzo IP del calcolatore è suddiviso in un numero di rete e un numero di macchina. Le proporzioni della suddivisione variano secondo la dimensione della rete. Una volta che gli amministratori di Internet hanno assegnato il numero di rete, nel sito con quel numero si è liberi di assegnare gli identificatori dei calcolatori come meglio si crede.

Il sistema mittente ricerca all’interno delle proprie tabelle d’instradamento un router cui spedire il pacchetto. I router sfruttano la parte di rete dell’identificatore del calcolatore per trasferire il pacchetto dalla rete di partenza a quella di destinazione. Il sistema destinatario riceve quindi il pacchetto, che può essere il messaggio completo o sempli-

cemente una sua parte. In quest'ultimo caso, prima di ricostruire il messaggio e passarlo al processo destinatario, è necessario attendere l'arrivo degli altri pacchetti.

A questo punto si sa come un pacchetto passa dalla rete di partenza alla sua destinazione, ma ci si può chiedere come fa un pacchetto, all'interno di una rete, a passare dal mittente (calcolatore o router) al ricevente. Ogni dispositivo Ethernet ha un numero unico che lo individua detto **indirizzo MAC** (*medium access control address*). Due dispositivi di una rete locale comunicano tra loro servendosi unicamente di questo numero. Se un sistema necessita di inviare dati a un altro, il software di rete genera un pacchetto ARP (*address resolution protocol*) contenente l'indirizzo IP del sistema di destinazione. Questo pacchetto viene **diffuso** (*broadcast*) a tutti gli altri sistemi della rete Ethernet.

Il broadcast adopera uno speciale indirizzo di rete (di solito l'indirizzo massimo) per segnalare a tutti i calcolatori connessi di ricevere ed elaborare il pacchetto. I pacchetti trasmessi per diffusione non sono ritrasmessi dai gateway, perciò sono ricevuti solamente dai sistemi connessi alla rete locale. Solo il sistema il cui indirizzo IP corrisponde all'indirizzo IP dell'ARP risponde e invia il proprio indirizzo MAC al sistema che ha fatto l'interrogazione. Per motivi di efficienza, il calcolatore copia in una tabella cache interna la coppia *<indirizzo IP, indirizzo MAC>*. Gli elementi della cache sono fatti “invecchiare” (*aged*) fino a essere rimossi qualora in un certo tempo non sia richiesto un accesso a quel sistema destinatario. In questo modo i calcolatori rimossi dalla rete vengono *dimenticati*. Per aumentare le prestazioni, si possono appuntare nella cache ARP gli elementi relativi ai calcolatori più usati.

Dopo che un dispositivo Ethernet ha annunciato il suo identificatore del calcolatore e il suo indirizzo, la comunicazione può cominciare. Un processo può specificare il nome del calcolatore con cui intende comunicare, il software di rete prende questo nome e determina l'indirizzo IP della destinazione attraverso un'interrogazione al DNS. Il messaggio passa dallo strato di applicazione all'interfaccia hardware, attraversando tutti gli strati di programmi e protocolli. Al livello hardware, il pacchetto (o i pacchetti) contiene l'indirizzo Ethernet al suo inizio e una coda, che indica la fine del pacchetto e contiene checksum che serve a rilevare eventuali errori nel pacchetto stesso (Figura 17.9). Il pacchetto viene posto nella rete dal dispositivo Ethernet. La sezione dei dati del pacchetto può contenere tutto il messaggio o solamente una parte dei suoi dati, così come può contenere alcune delle intestazioni di livello superiore che compongono il messaggio. In altre parole, tutti gli elementi del messaggio originale si devono inviare dall'indirizzo di provenienza alla destinazione e tutte le intestazioni poste sopra lo strato 802.3 (lo strato di collegamento dati) sono incluse nel pacchetto Ethernet sotto forma di dati.

Se la destinazione si trova nella stessa rete locale dell'indirizzo di provenienza, il sistema può analizzare la propria cache di ARP, trovare l'indirizzo Ethernet del calcolatore e immettere il pacchetto sulla linea. Il dispositivo Ethernet della destinazione riconosce il proprio indirizzo e legge il pacchetto, passandolo ai livelli superiori della pila di protocolli.

Se il sistema di destinazione si trova in una rete diversa da quella di provenienza, il sistema di partenza cerca il router appropriato sulla propria rete e gli spedisce il

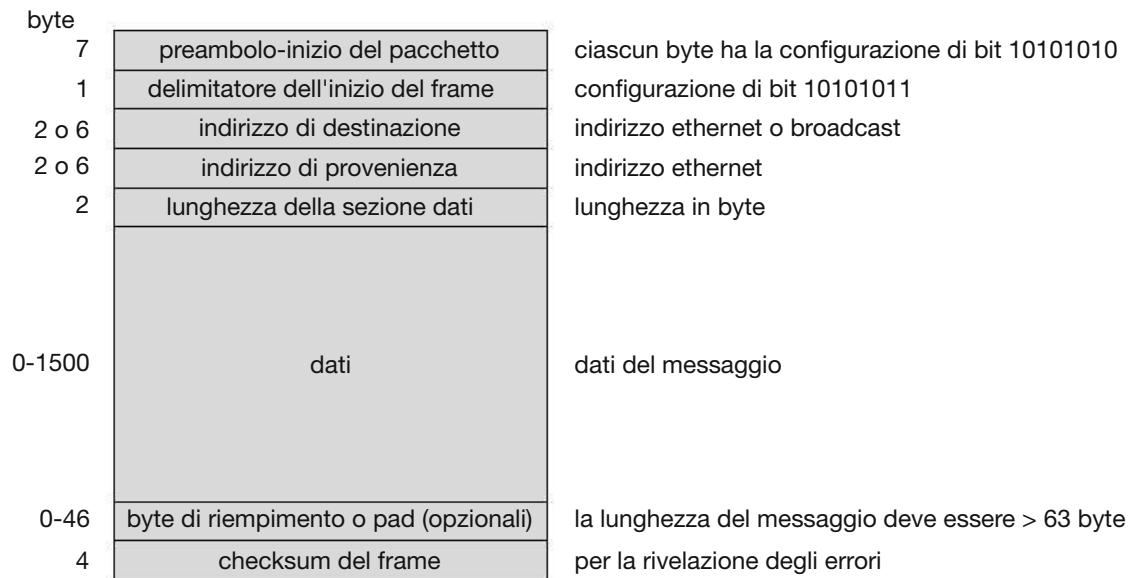


Figura 17.9 Pacchetto Ethernet.

pacchetto. I router trasmettono quindi il pacchetto lungo la WAN finché esso raggiunge la sua rete di destinazione. Il router che connette la rete di destinazione esamina la propria cache di ARP, cerca il numero Ethernet della destinazione e spedisce il pacchetto al calcolatore. Attraverso tutti questi trasferimenti, l'intestazione dello strato di collegamento dati può cambiare man mano viene usato l'indirizzo Ethernet del successivo router della catena, mentre le altre intestazioni del pacchetto restano invariate finché il pacchetto non venga ricevuto ed elaborato dalla pila di protocolli e quindi passato dal kernel al processo ricevente.

17.7 Robustezza

Un sistema distribuito può soffrire di guasti fisici di diverso tipo. Il guasto di un collegamento, di un sito e la perdita di un messaggio sono quelli più frequenti. Per assicurare la robustezza del sistema occorre **individuare** tutti questi guasti, **riconfigurare** il sistema per poter continuare la computazione ed effettuare il **ripristino** una volta riparato il collegamento o il sito.

17.7.1 Rilevamento dei guasti

In un ambiente senza memoria condivisa non si è generalmente capaci di distinguere tra il guasto di un collegamento, il guasto di un sito e la perdita di un messaggio. Normalmente si può stabilire solo che si è verificato uno di questi guasti, ma non è possibile identificarne il tipo. Una volta rilevato un guasto è necessario intraprendere azioni appropriate, che dipendono dalla particolare applicazione.

Per rilevare un guasto in un collegamento o in un sito si usa una procedura detta di **heartbeat**. Si supponga che i siti *A* e *B* abbiano tra loro un collegamento fisico di-

retto. A intervalli fissi entrambi i siti s'inviano un messaggio *I-am-up*. Se il sito *A* non riceve questo messaggio entro un dato periodo di tempo, può supporre che il sito *B* sia guasto, che sia guasto il collegamento tra *A* e *B*, oppure che sia andato perduto il messaggio proveniente da *B*. A questo punto il sito *A* ha due possibilità: può attendere un altro periodo di tempo per ricevere un messaggio *I-am-up* da *B*, oppure può inviare un messaggio *Are-you-up?* a *B*.

Se il sito *A* non riceve un messaggio *I-am-up* oppure una risposta alla propria domanda, si può ripetere la procedura. L'unica conclusione che il sito *A* può trarre con sicurezza è che si è verificato un guasto.

Il sito *A* può tentare di distinguere tra la possibilità di un guasto al collegamento e quella di un guasto al sito inviando a *B* il messaggio *Are-you-up?* per un altro percorso, se questo esiste. Se e quando riceve questo messaggio, *B* risponde subito positivamente. Questa risposta positiva indica ad *A* che *B* è attivo, e che il guasto si trova nel collegamento diretto. Poiché non si può sapere a priori quanto tempo impieghi il messaggio per andare da *A* a *B* e tornare indietro, è necessario usare uno **schema d'attesa** (*time-out*). Quando *A* invia il messaggio *Are-you-up?*, specifica un intervallo di tempo entro il quale è disposto ad attendere la risposta di *B*. Se *A* riceve il messaggio di risposta entro quell'intervallo di tempo, può concludere con sicurezza che *B* è attivo; altrimenti *A* può concludere che si è verificata una (o più) delle seguenti situazioni:

- il sito *B* è fuori servizio;
- il collegamento diretto, se esiste, tra *A* e *B* è fuori servizio;
- il percorso alternativo da *A* a *B* è fuori servizio;
- il messaggio è andato perduto.

Il sito *A* non può in ogni caso determinare quale di questi eventi si sia verificato.

17.7.2 Riconfigurazione

Si supponga che il sito *A* abbia scoperto, per mezzo del meccanismo appena descritto, che si è verificato un guasto. Deve allora cominciare una procedura che permetta al sistema di riconfigurarsi e continuare a funzionare normalmente.

- Se si è guastato un collegamento diretto tra *A* e *B*, questa informazione deve essere diffusa a ogni sito del sistema, in modo da poter aggiornare adeguatamente le tabelle di routing.
- Se *A* ritiene che il sito *B* sia fuori servizio, poiché tale sito non è più raggiungibile, si deve informare della situazione ciascun sito del sistema, in modo che non tenti più di usare i servizi del sito guasto. Il guasto di un sito usato come coordinatore centrale di qualche attività, come il rilevamento delle situazioni di stallo, implica l'elezione di un nuovo coordinatore. Analogamente, se il sito guasto fa parte di un anello logico, è necessario costruire un nuovo anello logico. Occorre notare che, se il sito non si è guastato (vale a dire che è funzionante ma non rag-

giungibile), è possibile incorrere nella situazione spiacevole in cui due siti agiscono da coordinatore. Se la rete è partizionata, i due coordinatori, uno per ogni partizione, possono condurre ad azioni conflittuali. Se i coordinatori sono responsabili, per esempio, della realizzazione della mutua esclusione, si può avere una situazione in cui due processi sono in esecuzione contemporaneamente nelle rispettive sezioni critiche.

17.7.3 Ripristino dopo un guasto

Quando un collegamento o un sito guasto è stato riparato, deve essere reintegrato nel sistema in modo semplice e lineare.

- Si supponga che si sia guastato un collegamento tra *A* e *B*. Una volta riparato il guasto, è necessario informare della riparazione sia *A* sia *B*. Quest’informazione può essere data ripetendo continuamente la procedura di heartbeat descritta nel Paragrafo 17.7.1.
- Si supponga che si sia guastato il sito *B*. Questo, una volta ripristinato, deve informare tutti gli altri siti che è di nuovo attivo. Il sito *B* può quindi ricevere informazioni da altri siti per aggiornare le sue tabelle locali; per esempio, possono servirgli le informazioni contenute nelle tabella di routing, un elenco di siti fuori servizio, messaggi e posta non consegnati, oppure un log delle transazioni contenente le transazioni non eseguite. Se il sito non era guasto, ma era semplicemente irraggiungibile, queste informazioni devono comunque essergli inviate.

17.7.4 Tolleranza ai guasti

Un sistema distribuito deve tollerare guasti di una certa entità e continuare a funzionare normalmente quando incontra diversi tipi di guasto. Il progetto di installazioni tolleranti ai guasti ha inizio a livello di protocollo, come descritto precedentemente, ma coinvolge poi tutti gli aspetti del sistema. Il termine *tolleranza ai guasti* si usa in senso lato; gli errori di comunicazione, alcuni guasti della macchina, la rottura dei dispositivi di memoria e il deterioramento degli strumenti di memorizzazione dovrebbero essere tutti in una certa misura tollerabili. Un **sistema tollerante ai guasti** (*fault-tolerant*) deve continuare a funzionare, eventualmente con efficacia ridotta, anche quando si presentano tali guasti. Le conseguenze negative possono riguardare le prestazioni, il funzionamento, o entrambi gli aspetti; ma in ogni caso, il peggioramento dovrebbe essere proporzionale ai guasti che l’hanno causato. Un sistema che si ferma quando si guasta uno solo dei suoi componenti non si può certo considerare tollerante ai guasti.

Sfortunatamente la tolleranza ai guasti è costosa e difficile da realizzare. A livello di rete, per evitare interruzioni alla comunicazione sono necessari diversi percorsi di comunicazione ridondanti e dispositivi di rete come switch e router. Un guasto nei dispositivi di memoria può causare la perdita del sistema operativo, delle applicazioni o dei dati. Le unità di memorizzazione possono utilizzare componenti hardware ri-

dondanti in grado di sostituirsi a vicenda in maniera automatica in caso di guasti. Inoltre i sistemi RAID sono anche in grado di garantire un accesso continuo ai dati in caso di guasto di uno o più dischi (Paragrafo 10.7).

Un guasto del sistema in mancanza di ridondanza può causare il blocco di un'applicazione o di un intero servizio di sistema. Il caso più semplice di guasto è quello che coinvolge un sistema che esegue solo applicazioni prive di stato. Tali applicazioni possono essere riavviate senza compromettere l'operatività; in questo modo, a patto che l'applicazione sia in esecuzione su più di un computer (nodo), si mantiene la continuità operativa. Tale organizzazione è solitamente nota come **cluster di computazione**, perché mette al centro la computazione.

Per contro, i sistemi data-centrati utilizzano applicazioni che accedono a dati condivisi e li modificano. Ne risulta che è più difficile rendere tolleranti ai guasti i servizi di sistemi data-centrati. Tali sistemi richiedono un software di monitoraggio dei guasti e una infrastruttura speciale. Per esempio, i **cluster ad alta disponibilità** sono formati da due o più computer e da un insieme di dischi condivisi. Ogni applicazione può essere memorizzata sui computer oppure sui dischi condivisi, mentre i dati devono essere memorizzati sui dischi condivisi. Un nodo avente un'applicazione in esecuzione ha accesso esclusivo ai dati dell'applicazione su disco. L'applicazione viene monitorata dal software del cluster e in caso di guasto viene automaticamente riavviata. Se non è possibile eseguire un riavvio, o se un guasto coinvolge l'intero computer, l'accesso esclusivo del nodo ai dati dell'applicazione viene interrotto e assegnato a un altro nodo del cluster; l'applicazione viene quindi riavviata sul nuovo nodo. L'applicazione perde tutte le informazioni di stato memorizzate sul sistema guasto, ma può continuare sulla base delle informazioni che ha scritto per ultime sul disco condiviso. Dal punto di vista dell'utente un servizio viene terminato e riavviato, eventualmente con la perdita di alcuni dati.

Alcune applicazioni specifiche possono migliorare questa funzionalità implementando la gestione dei lock insieme al clustering. Grazie alla gestione dei lock l'applicazione può essere in esecuzione su più nodi e utilizzare gli stessi dati sui dischi condivisi in maniera concorrente. I cluster di basi di dati implementano spesso questa funzionalità. Se un nodo è vittima di un guasto le transazioni possono continuare sugli altri nodi e gli utenti non rilevano alcuna interruzione del servizio, a condizione che il client sia in grado di localizzare automaticamente gli altri nodi del cluster. Ogni transazione non completata sul nodo guasto viene persa ma, ancora una volta, le applicazioni sul client possono essere progettate per rieseguire le transazioni non complete nel caso in cui venga rilevato un guasto sul nodo della base dati.

17.8 Problemi di progettazione

Rendere **trasparente** agli utenti la molteplicità delle unità d'elaborazione e dei dispositivi di memorizzazione è stato lo scopo di molti progettisti. Idealmente un sistema distribuito dovrebbe apparire ai suoi utenti come un sistema centralizzato convenzionale. L'interfaccia utente di un sistema distribuito trasparente non deve fare

distinzioni tra risorse locali e risorse remote. Ciò significa che gli utenti dovrebbero poter accedere a sistemi distribuiti remoti come se fossero locali, e spetterebbe al sistema distribuito localizzare le risorse e predisporre le corrette interazioni.

Un altro aspetto della trasparenza riguarda la mobilità degli utenti; è conveniente permettere agli utenti di aprire sessioni in qualsiasi calcolatore del sistema, anziché obbligarli a usare un calcolatore specifico. Un sistema distribuito trasparente facilita la mobilità degli utenti trasportando l’ambiente dell’utente, per esempio la directory iniziale, ovunque l’utente apra sessioni. I protocolli come LDAP forniscono un sistema di autenticazione per utenti locali, remoti o mobili. Una volta che l’utente si è autenticato, funzionalità come la virtualizzazione del desktop gli consentono di operare sulle proprie sessioni di lavoro da remoto.

La capacità di un sistema di adattarsi a un maggior carico di servizio è la cosiddetta **scalabilità**. I sistemi hanno risorse limitate e, alla presenza di un incremento del carico, possono giungere alla completa saturazione. Per esempio, per quel che riguarda un file system, la saturazione avviene quando la CPU di un server funziona a un alto tasso d’utilizzo, oppure quando le richieste di accesso ai dischi saturano il sistema di I/O. La scalabilità è una caratteristica relativa, ma che si può misurare con precisione. Un sistema scalabile reagisce più gradualmente a un carico maggiore di quanto non lo faccia un sistema non scalabile. Innanzitutto le sue prestazioni degradano più moderatamente; in secondo luogo, le sue risorse raggiungono la saturazione più tardi. Anche un sistema perfettamente progettato non può sopportare un carico sempre crescente. L’aggiunta di nuove risorse può risolvere il problema, ma può anche generare un ulteriore carico indiretto su altre risorse (per esempio, l’aggiunta di calcolatori a un sistema distribuito potrebbe produrre un intasamento della rete e aumentare i carichi di servizio). O peggio, l’espansione del sistema potrebbe richiedere costose modifiche strutturali. Un sistema scalabile deve avere la potenzialità di crescere evitando questi problemi. In un sistema distribuito una conveniente scalabilità ha un’importanza particolare, poiché è una pratica diffusa espandere la rete aggiungendo calcolatori o connettendo due reti. In breve, un progetto scalabile deve resistere a un alto carico di servizio, far fronte alla crescita della comunità di utenti e permettere una facile integrazione di ulteriori risorse.

La scalabilità è legata alla tolleranza ai guasti, discussa in precedenza. Un componente estremamente carico può paralizzarsi e comportarsi come un componente guasto, quindi lo spostamento del carico da un componente guasto a quello di riserva può saturare anche quest’ultimo. In generale, disporre di risorse di riserva è essenziale sia ai fini dell’affidabilità sia per poter gestire in modo conveniente i picchi di carico. Un sistema distribuito ha il vantaggio intrinseco della potenziale tolleranza ai guasti e la potenziale scalabilità date dalla molteplicità delle risorse. Tuttavia una progettazione non appropriata può occultare queste potenzialità. Le considerazioni di tolleranza ai guasti e di scalabilità richiedono una progettazione caratterizzata dalla distribuzione del controllo e dei dati.

Soluzioni come il file system distribuito Hadoop sono state create tenendo in considerazione questo problema. Hadoop è basato sui progetti Google MapReduce e

Google File System, che hanno creato una struttura per monitorare ogni pagina web su Internet. Hadoop è un framework di programmazione open source che supporta l’elaborazione di grandi insiemi di dati in ambienti elaborativi distribuiti. I sistemi tradizionali con database tradizionali non possono raggiungere le capacità e le prestazioni necessarie per progetti “Big Data” (o almeno non a prezzi ragionevoli). Un esempio di progetto che lavora su grandi dataset consiste nell’interrogazione di Twitter alla ricerca di informazioni pertinenti a una società. Un altro esempio è l’analisi dei dati finanziari per derivare le tendenze dei prezzi delle azioni. Con Hadoop e gli strumenti correlati, migliaia di sistemi sono in grado di lavorare insieme per gestire un database distribuito contenente petabyte di informazioni.

17.9 File system distribuiti

Sebbene il World Wide Web sia il sistema distribuito decisamente più diffuso, non è l’unico. Un altro importante esempio elaborazione distribuita sono i **file system distribuiti**, o DFS. In questo paragrafo analizzeremo i file system distribuiti attraverso due esempi: OpenAFS, un file system distribuito open-source, e NFS, il più comune DFS basato su UNIX. Esistono diverse versioni di NFS, ma in questo paragrafo, se non diversamente indicato, faremo riferimento a NFS versione 3.

Per descrivere la struttura di un DFS occorre definire innanzitutto i termini servizio, server e client nel contesto dei file system distribuiti. Il **servizio** è il software in esecuzione in una o più macchine che fornisce un tipo particolare di funzione a client sconosciuti a priori; il **server** è il programma di servizio in esecuzione in una singola macchina; il client è un processo che può richiedere un servizio servendosi di una serie di operazioni che costituiscono la sua interfaccia, detta interfaccia del client. Talvolta si definisce un’interfaccia di livello inferiore per l’effettiva interazione tra le macchine; si tratta dell’**interfaccia intermacchina**.

Secondo questa terminologia, diciamo che un file system fornisce servizi relativi ai file ai suoi client. Un’interfaccia del client per un servizio relativo ai file è composta da un insieme di operazioni primitive su file, come creazione di un file, cancellazione, lettura e scrittura. Il principale elemento fisico controllato da un file server è composto da un insieme di dispositivi locali di memoria secondaria, generalmente dischi magnetici, nei quali si memorizzano i file e dai quali si recuperano gli stessi secondo le richieste dei client.

Un DFS è un file system i cui client, server e dispositivi di memorizzazione sono sparsi tra le macchine di un sistema distribuito. Di conseguenza l’attività di servizio si deve eseguire attraverso la rete e, anziché un unico sito di memorizzazione dei dati centralizzato, il sistema ha più dispositivi di memorizzazione indipendenti. La configurazione e la realizzazione concrete di un DFS possono essere di vario tipo. In alcune configurazioni i server sono eseguiti su macchine specifiche, in altre una macchina può essere sia un server sia un client. Un DFS si può realizzare come parte di un sistema operativo distribuito o, in alternativa, con uno strato software il cui compito consiste nella gestione della comunicazione tra sistemi operativi convenzionali e file system.

Le caratteristiche che contraddistinguono un DFS sono la molteplicità e l'autonomia dei client e dei server del sistema. Tuttavia un DFS deve, idealmente, apparire ai client come un file system centralizzato convenzionale. Ciò significa che l'interfaccia del client di un DFS non deve distinguere tra file locali e file remoti. Spetta al DFS localizzare i file e predisporre il trasporto dei dati. Un DFS trasparente, come i sistemi distribuiti trasparenti menzionati in precedenza, facilita la mobilità dell'utente portando il suo ambiente, cioè la directory iniziale, ovunque egli apra una sessione.

La misura delle prestazioni più importante per un DFS è rappresentata dal tempo necessario a soddisfare le richieste di servizio. Nei sistemi convenzionali questo tempo è dato dal tempo d'accesso al disco e da una piccola quantità di tempo di CPU. In un DFS, invece, un accesso remoto risente anche di un ulteriore carico dovuto alla struttura distribuita. In questo overhead rientrano il tempo necessario per inviare la richiesta a un server e il tempo necessario per ricevere la risposta. Pertanto per ogni direzione occorre considerare, oltre al trasferimento dell'informazione, anche il carico della CPU dovuto all'esecuzione del protocollo di comunicazione. Le prestazioni di un DFS si possono considerare come un'altra misura della sua trasparenza. Ciò significa che le prestazioni di un DFS ideale devono essere paragonabili a quelle di un file system convenzionale.

Il fatto che un DFS gestisca un insieme di dispositivi di memorizzazione sparsi è la caratteristica che contraddistingue il DFS stesso. Lo spazio di memorizzazione totale gestito da un DFS è composto da diversi spazi di memorizzazione più piccoli, localizzati in posizioni remote. Generalmente questi spazi di memorizzazione costituenti corrispondono a insiemi di file. Un'unità componente è il più piccolo insieme di file che si può memorizzare in una singola macchina, indipendentemente da altre unità. Tutti i file che appartengono alla stessa unità componente devono risiedere nella stessa locazione.

17.9.1 Naming e trasparenza

Il naming è una funzione di associazione tra oggetti logici e oggetti fisici. Per esempio, gli utenti trattano oggetti logici contenenti dati, rappresentati dai nomi dei file, mentre il sistema manipola blocchi fisici di dati, memorizzati sulle tracce di un disco. Generalmente l'utente fa riferimento a un file attraverso un nome testuale. A quest'ultimo si fa corrispondere un identificatore numerico di livello inferiore che a sua volta si fa corrispondere ai blocchi dei dischi. Quest'associazione a più livelli fornisce agli utenti un'astrazione del file che nasconde i particolari concernenti la sua memorizzazione.

In un DFS trasparente, all'astrazione si aggiunge un nuovo aspetto: nascondere la posizione all'interno della rete in cui è posizionato il file. In un file system convenzionale il risultato della funzione di naming è un indirizzo in un disco. In un DFS si estende l'insieme dei valori risultanti in modo da comprendere anche la specifica macchina nel cui disco è contenuto il file. Un passo successivo nel trattare i file come oggetti astratti, porta alla possibilità di **replicazione dei file**. Dato il nome di un file, il sistema di associazione riporta una serie di locazioni delle repliche di questo file. In quest'astrazione sono nascoste sia l'esistenza di copie multiple sia la loro locazione.

17.9.1.1 Strutture di naming

Occorre distinguere due nozioni correlate riguardo al sistema di associazione dei nomi di un DFS.

1. **Trasparenza rispetto alla locazione.** Il nome di un file non rivela alcun indizio sulla sua locazione fisica.
2. **Indipendenza dalla locazione.** Non si deve modificare il nome di un file se cambia la sua locazione di memoria fisica.

Entrambe le definizioni si riferiscono al livello di naming esaminato precedentemente, poiché i file hanno nomi diversi a livelli diversi: a livello utente si usano nomi testuali, mentre a livello del sistema si usano gli identificatori numerici. Uno schema di naming indipendente dalla locazione è un'associazione dinamica poiché può far corrispondere locazioni diverse in due momenti diversi allo stesso nome di file. Perciò l'indipendenza dalla locazione è una caratteristica più forte di quanto non lo sia la trasparenza rispetto alla locazione.

In pratica la maggior parte degli attuali DFS offre un'associazione statica con trasparenza di locazione per i nomi a livello utente. Alcuni DFS supportano la **migrazione dei file**, cioè il cambio automatico di posizione di un file, garantendo così l'indipendenza dalla locazione. OpenAFS, per esempio, supporta l'indipendenza dalla locazione e la mobilità dei file. Il **file system distribuito Hadoop** (HDFS), uno speciale file system progettato recentemente per il framework Hadoop, supporta la migrazione dei file, ma lo fa senza seguire gli standard POSIX, fornendo una maggiore flessibilità nell'implementazione e nell'interfaccia. HDFS tiene traccia della posizione dei dati, ma nasconde questa informazione ai client. Questa strategia permette al meccanismo sottostante un tuning automatico. Un altro esempio è l'infrastruttura di **cloud storage** di Amazon, che fornisce blocchi di storage on demand per mezzo di API, mettendo lo storage dove lo ritiene opportuno e spostando i dati secondo le necessità, al fine di migliorare le prestazioni, l'affidabilità e le esigenze di capacità.

L'indipendenza dalla locazione e la trasparenza di locazione statica si differenziano ancora per altri aspetti.

- La separazione dei dati dalla locazione, come nel caso dell'indipendenza dalla locazione, offre una migliore astrazione per i file. Il nome di un file deve indicare gli attributi più significativi del file stesso, legati al suo contenuto, piuttosto che la sua locazione. I file indipendenti dalla locazione si possono considerare come contenitori di dati logici che non si assegnano a una specifica locazione di memoria secondaria. Se è prevista soltanto la trasparenza di locazione statica, il nome del file continua a indicare un gruppo specifico, anche se nascosto, di blocchi fisici nei dischi.
- La trasparenza di locazione statica fornisce agli utenti un modo per condividere convenientemente i dati. Gli utenti possono condividere file remoti semplicemente assegnandogli nomi in maniera trasparente rispetto alla locazione, esattamente come se questi fossero locali. Dropbox e altre soluzioni di cloud storage lavorano in questo modo. L'indipendenza dalla locazione consente la condivisione dello spa-

zio di memorizzazione stesso e degli oggetti di dati. Quando i file si possono rendere mobili, lo spazio di memorizzazione dell'intero sistema assume l'aspetto di una singola risorsa virtuale. Uno dei vantaggi che ne conseguono è la possibilità di bilanciare l'utilizzo dei dischi all'interno del sistema.

- L'indipendenza dalla locazione separa la gerarchia di naming dalla gerarchia dei dispositivi di memorizzazione e dalla suddivisione tra i calcolatori. Al contrario, usando la trasparenza di locazione statica, si può facilmente evidenziare la corrispondenza tra unità componenti e macchine, anche se i nomi sono trasparenti. Le macchine sono configurate impiegando un modello simile alla struttura di naming; questa configurazione può forzare l'architettura del sistema a rispettare vincoli non necessari, ed è anche in conflitto con altre considerazioni. Un server che gestisca una directory radice è un esempio di struttura dettata dalla gerarchia di naming e contraddice le direttive di decentramento.

Una volta completata la separazione tra nome e locazione, i client possono accedere a file che risiedono in sistemi server remoti. Infatti questi client possono essere privi di dischi e adoperare un server per accedere a tutti i file, compreso il kernel del sistema operativo. Per la sequenza d'avviamento sono però necessari protocolli speciali; si consideri il problema di caricare il kernel su una stazione di lavoro priva di dischi. Poiché tale stazione non ha il kernel, non può usare il codice del DFS per recuperarlo. Invece s'impiega uno speciale protocollo d'avviamento, contenuto in memoria di sola lettura (ROM) del client, che abilita la connessione alla rete e recupera solo un file speciale, il kernel o il codice d'avviamento da una locazione fissa. Una volta che il kernel è stato copiato attraverso la rete e caricato, il suo DFS rende disponibili tutti gli altri file del sistema operativo. I vantaggi dati da client privi di dischi sono molti, tra i quali il minor costo (le macchine sono appunto prive di dischi) e una maggior facilità di gestione (quando si aggiorna un sistema operativo, occorre modificare solo il server anziché tutti i client). Lo svantaggio invece è dato dalla maggiore complessità dei protocolli d'avviamento e dal calo delle prestazioni dovuto all'uso di una rete anziché di un disco locale.

17.9.1.2 Schemi di naming

In un DFS sono possibili tre metodi principali per gli schemi di naming. Il metodo più semplice prevede di nominare i file per mezzo di una combinazione del loro nome di macchina e nome locale; ciò garantisce un unico nome su tutto il sistema. Nel sistema Ibis, per esempio, un file è identificato unicamente dal nome *macchina:nome-locale*, dove nome-locale indica un percorso di tipo UNIX. Anche gli URL di Internet seguono questo approccio. Questo schema di naming non gode della trasparenza di locazione e tanto meno dell'indipendenza dalla locazione. La struttura del DFS è rappresentata da un insieme di unità componenti isolate costituite da interi file system convenzionali. Le unità componenti rimangono isolate, anche se sono disponibili mezzi per fare riferimento ai file remoti. Non proseguiremo oltre la descrizione di questo schema.

Il secondo metodo si è diffuso con il network file system di Sun (NFS). NFS si trova in molti sistemi, tra cui UNIX e Linux e offre i mezzi per unire le directory remote

alle directory locali, dando all’utente l’impressione di un albero di directory coerente. Nelle prime versioni era possibile accedere in modo trasparente solo a directory remote montate in precedenza. Con l’avvento della funzione di **automontaggio** i montaggi si eseguono su richiesta secondo una tabella di punti di montaggio e nomi di strutture di file. I componenti sono integrati per gestire la condivisione trasparente, sebbene tale integrazione sia limitata e non uniforme, poiché ogni macchina può aggiungere diverse directory remote al proprio albero. La struttura risultante è flessibile.

L’integrazione totale dei componenti del file system si ottiene per mezzo del terzo metodo. Una sola struttura globale di nomi si estende a tutti i file del sistema. Idealmente, la struttura composta del file system è isomorfa alla struttura di un file system convenzionale. In pratica, tuttavia, esistono molti file speciali (per esempio i file che rappresentano i dispositivi e le directory dei file eseguibili della specifica macchina) che rendono difficile il conseguimento di questo scopo.

Per valutare le strutture di mappaggio ne consideriamo la complessità amministrativa. La struttura più complessa e più difficile da mantenere è la struttura dell’NFS. Poiché ogni directory remota può essere montata in qualsiasi punto dell’albero di una directory locale, la gerarchia risultante può essere quasi priva di struttura. Se un server diventa non disponibile, diventa non disponibile anche un insieme arbitrario di directory di diverse macchine. Inoltre, un meccanismo separato di accreditamento controlla quale macchina può aggiungere una determinata directory al suo albero. Così un utente può ottenere l’accesso a un albero di directory remoto in un client, ma vederlo negare in un altro.

17.9.1.3 Realizzazione

La realizzazione del naming trasparente richiede un meccanismo per far corrispondere i nomi alle locazioni dei file. Per mantenere gestibile tale corrispondenza occorre aggregare insiemi di file in unità componenti e fornire una corrispondenza secondo l’unità componente anziché secondo il singolo file. Questa aggregazione è utile anche per scopi amministrativi. I sistemi del tipo UNIX impiegano l’albero gerarchico delle directory per fornire la corrispondenza tra nomi e locazioni e per aggregare i file nelle directory in modo ricorsivo.

Per aumentare la disponibilità delle informazioni fondamentali su tali associazioni si possono usare metodi come replicazione, uso di cache locali, o entrambi. Come si è detto, l’indipendenza dalla locazione implica che l’associazione cambi con il tempo; quindi, replicando l’associazione, diventa impossibile ottenere un aggiornamento semplice e coerente di queste informazioni. Per superare quest’ostacolo si può adoperare una tecnica che introduce **identificatori di file di basso livello indipendenti dalla locazione** (OpenAFS utilizza questo approccio). Ai nomi dei file si fanno corrispondere identificatori di file di basso livello, che indicano a quale unità componente appartiene il file; tali identificatori sono ancora indipendenti dalla locazione. Si possono replicare liberamente e copiare in una cache, senza che siano invalidati dalla migrazione delle unità componenti. L’inevitabile prezzo di tale tecnica è un secondo livello di associazione che metta in corrispondenza le unità componenti con le locazioni e richiede un meccanismo di aggiornamento semplice e coerente. Realizzare

alberi di directory di tipo UNIX impiegando questi identificatori di basso livello indipendenti dalla locazione, rende invariante l'intera gerarchia nel caso della migrazione di unità componenti. L'unica variazione riguarda l'associazione delle locazioni delle unità componenti.

Un metodo diffuso per realizzare questi identificatori di basso livello consiste nell'uso di nomi strutturati. Questi nomi sono composti da sequenze di bit formate normalmente da due parti: la prima identifica l'unità componente a cui appartiene il file; la seconda identifica il file all'interno dell'unità. Sono possibili varianti con più parti. L'aspetto invariante dei nomi strutturati, tuttavia, è che ciascuna parte del nome è unica in ogni momento soltanto nel contesto delle parti restanti. L'unicità in ogni momento si può ottenere avendo cura di non riutilizzare un nome già usato, oppure aggiungendo ulteriori bit (metodo utilizzato in OpenAFS) o impiegando una marca temporale (timestamp) come se fosse una parte del nome (come accadeva nell'Apollo Domain). Un altro modo di vedere questo processo è dire che si prende un sistema con trasparenza di locazione, come l'Ibis, e vi si aggiunge un altro livello di astrazione per produrre uno schema di naming con indipendenza dalla locazione.

17.9.2 Accesso ai file remoti

Si consideri un utente che richieda l'accesso a un file remoto. Supponendo che il server che contiene il file sia stato localizzato dallo schema di naming, ora è necessario compiere l'effettivo trasferimento dei dati.

Tale trasferimento si può ottenere con il **meccanismo del servizio remoto**, in cui le richieste d'accesso sono inviate al server che esegue gli accessi e ritrasmette i risultati all'utente. Uno dei modi più diffusi per realizzare il servizio remoto è il paradigma della chiamata di procedura remota (RPC), descritto nel Capitolo 3. Esiste un'analogia diretta tra il metodo d'accesso al disco nei file system convenzionali e il metodo del servizio remoto in un DFS: l'uso del servizio remoto è analogo al compiere un accesso al disco per ciascuna richiesta d'accesso.

Per assicurare prestazioni ragionevoli del meccanismo di servizio remoto si usa un meccanismo di caching. Nei file system convenzionali lo scopo delle cache consiste nel ridurre gli accessi ai dischi, aumentando così le prestazioni; nei DFS lo scopo è quello di ridurre sia il traffico nella rete sia gli accessi ai dischi. In seguito si considera la gestione delle cache in un DFS, a confronto del paradigma del servizio remoto di base.

17.9.2.1 Caching di base

Il concetto di caching è semplice; se i dati necessari a soddisfare la richiesta d'accesso non sono già stati copiati nella cache, si trasferisce una copia di quei dati dal server al sistema client. Gli accessi richiesti si eseguono sulla copia in cache. L'idea è di tenere nella cache blocchi di disco a cui si è fatto riferimento di recente, gestendo localmente gli accessi ripetuti alle stesse informazioni, e riducendo il traffico di rete. Un criterio di sostituzione (per esempio LRU) limita la dimensione della cache. Non esiste una corrispondenza diretta fra gli accessi e il traffico verso il server: i file sono

ancora identificati con una copia principale che risiede nella macchina server, ma altre copie del file o di parti di esso sono sparse in diverse cache. Se si modifica una copia contenuta in una cache, le modifiche vanno riportate anche sulla copia principale, in modo da conservare la coerenza dei dati. Il problema di mantenere le copie coerenti con il file principale è detto **problema di coerenza della cache** (Paragrafo 17.9.2.4). L'impiego delle cache in un DFS si può chiamare semplicemente **memoria virtuale di rete**: infatti funziona in modo analogo alla memoria virtuale con paginazione su richiesta, con la differenza che in questo caso la memoria ausiliaria non è un disco locale, ma un server remoto. NFS permette il montaggio a distanza dell'area di swap; può quindi effettivamente implementare la memoria virtuale sulla rete, anche se a scapito di una calo nelle prestazioni.

Nel caso di un DFS, la dimensione dei dati da copiare nelle cache può variare dai blocchi di file ai file interi. In genere si copiano più dati di quanti non siano necessari a soddisfare un singolo accesso, in modo da aumentare le probabilità di soddisfare più accessi a tali dati. Questo procedimento è molto simile alla lettura anticipata dei dati dai dischi (Paragrafo 12.6.2). OpenAFS copia i file in grandi sezioni (64 KB); gli altri sistemi che abbiamo citato copiano blocchi singoli in modo controllato dalle richieste dei client. Aumentando la quantità unitaria di dati copiati, aumenta il tasso di successi, ma aumenta anche il costo di ogni insuccesso nella ricerca nella cache, poiché ogni insuccesso richiede il trasferimento di una maggiore quantità di dati. Inoltre aumentano le possibilità dei problemi di coerenza. Nella scelta della quantità di dati unitaria da copiare nelle cache occorre considerare parametri come l'unità di trasferimento della rete e l'unità di servizio del protocollo delle RPC (nel caso in cui sia usato un protocollo RPC). L'unità di trasferimento della rete (in Ethernet è un pacchetto) è di circa 1,5 KB, perciò le unità di dati più grandi devono essere scomposte prima di essere inviate e quindi ricomposte al momento della ricezione.

Naturalmente la dimensione dei blocchi e la dimensione totale della memoria cache sono importanti per gli schemi di gestione delle cache a blocchi. Nei sistemi di tipo UNIX le dimensioni normali dei blocchi sono di 4 KB o 8 KB. Per cache di grandi dimensioni (oltre 1 MB) convengono blocchi più grandi (oltre 8 KB); per cache più piccole, i blocchi di grandi dimensioni sono meno utili, poiché implicano un numero inferiore di blocchi contenuto nelle cache, e un tasso di successi inferiore.

17.9.2.2 Locazione delle cache

Ci si può chiedere se i dati della cache vadano memorizzati su disco o in memoria centrale. Le cache su dischi hanno un evidente vantaggio rispetto alle cache in memoria centrale: sono affidabili. Se la cache è nella memoria volatile, le modifiche ai dati in essa contenuti si perdono nel caso di una caduta del sistema. Inoltre, se la cache è mantenuta in memoria secondaria, i dati continuano a trovarvisi anche durante il ripristino, e non è necessario prelevarli di nuovo. D'altra parte le cache in memoria centrale hanno parecchi specifici vantaggi.

- Le cache in memoria centrale permettono l'uso di stazioni di lavoro prive di dischi.

- È possibile accedere più rapidamente ai dati in una cache in memoria centrale, che ai dati in una cache su disco.
- Le attuali tecnologie tendono a produrre memorie più grandi e meno costose. Si prevede che l'incremento delle prestazioni supererà in importanza i vantaggi offerti dalle cache su dischi.
- Le cache dei server, impiegate per accelerare l'I/O dei dischi, si trovano in memoria centrale a prescindere dalla locazione delle cache degli utenti; se si usano le cache in memoria centrale anche per le macchine degli utenti, si può realizzare un unico meccanismo di gestione utilizzabile sia dai server sia dagli utenti.

Molti sistemi d'accesso remoto si possono considerare come un ibrido fra l'uso di cache e servizio remoto. Nell'NFS, per esempio, l'implementazione è basata sul servizio remoto, ma è affiancata dall'uso di cache in memoria sia da parte del client sia da parte del server per incrementare le prestazioni. Analogamente il sistema Sprite si fonda sull'uso delle cache, ma in alcuni casi adotta il metodo del servizio remoto. Così nel valutare i due metodi si considera fino a che punto sia enfatizzato ciascun metodo.

Il protocollo NFS e la maggior parte delle sue realizzazioni non offrono la gestione di cache su dischi.

17.9.2.3 Criteri di aggiornamento delle cache

La politica che si usa per riscrivere blocchi di dati modificati nella copia principale sul server ha un effetto critico sulle prestazioni e sull'affidabilità del sistema. Il criterio più semplice consiste nello scrivere direttamente i dati nei dischi non appena questi vengono modificati in una cache qualsiasi. Il vantaggio della scrittura immediata (**write-through policy**) è l'affidabilità; se un sistema client si guasta, si perdono solo poche informazioni. Tuttavia questo criterio richiede che ogni accesso per scrittura attenda l'invio delle informazioni al server, il che implica scarse prestazioni delle operazioni di scrittura. L'uso di cache write-through corrisponde all'uso del servizio remoto per le scritture e allo sfruttamento delle cache per le sole letture.

Un'alternativa è il criterio di scrittura differita (o **write-back caching**), che consiste nel differire gli aggiornamenti della copia principale. Le modifiche si scrivono nella cache e più tardi nel server. Questo criterio ha due vantaggi rispetto alla scrittura diretta: innanzitutto, poiché le scritture sono effettuate sulla cache, gli accessi per scrittura sono molto più rapidi; in secondo luogo, si possono sovrascrivere i dati prima che siano riportati al server, in tal caso è necessario riportare solo l'ultimo aggiornamento. Sfortunatamente gli schemi di scrittura differita causano problemi di affidabilità, poiché se una macchina utente subisce un guasto si perdono i dati non scritti.

Le varianti del criterio della scrittura differita si differenziano secondo il momento in cui i blocchi di dati modificati sono inviati al server. Una possibilità è quella che prevede l'invio di un blocco quando questo sta per essere espulso dalla cache del client. Ciò può avere esiti positivi sulle prestazioni, ma può accadere che alcuni blocchi risiedano per un lungo periodo nella cache del client prima di essere riscritti nel server. Un compromesso tra questo metodo e la scrittura immediata consiste nell'esaminare la cache a intervalli regolari e inviare al server solo i blocchi modificati dopo

l'ultima verifica, esattamente come in UNIX si scandisce la cache locale. Nel sistema Sprite s'impiega questo criterio con un intervallo di verifica di 30 secondi. L'NFS adopera questo criterio per i dati nei file, ma una volta che una scrittura è stata inviata al server durante uno svuotamento della cache, per essere considerata completa, deve giungere al disco del server. L'NFS tratta i metadati (i dati nelle directory e i dati sugli attributi dei file) in modo differente: tutti i cambiamenti dei metadati s'inviano in modo sincrono ai relativi server. In questo modo si evitano le perdite di file e le alterazioni delle strutture delle directory in seguito a un crollo di un client o di un server.

Un'altra variante della scrittura differita consiste nello scrivere i dati nel server quando si chiude un file. Questo criterio, detto di scrittura su chiusura (**write-on-close policy**), è adottato dal sistema OpenAFS. Nel caso in cui si aprano i file per periodi brevi oppure si modifichino solo di rado, questo criterio non riduce significativamente il traffico presente in rete. Inoltre il criterio di scrittura su chiusura richiede che il processo in chiusura sia ritardato per permettere la scrittura diretta del file; ciò riduce i vantaggi della scrittura differita. Le migliori prestazioni di questo criterio, rispetto a quelli della scrittura differita con un più frequente invio al server, sono evidenti quando i file rimangono aperti per lunghi periodi e vengono modificati spesso.

17.9.2.4 Coerenza della cache

Una macchina client deve talvolta affrontare il problema di decidere se una copia in una cache sia o no coerente con la copia principale, e quindi se possa essere usata. Se la macchina client stabilisce che i suoi dati nella cache non sono aggiornati, occorre ottenere una copia aggiornata dei dati e inserirli nella cache prima di consentire ulteriori accessi. Per compiere tale verifica si possono seguire due metodi.

1. **Verifica iniziata dal client.** Il client inizia un controllo di validità mettendosi in contatto con il server per controllare se i dati locali siano coerenti con la copia principale. La frequenza del controllo di validità è il fulcro di questo metodo e determina la conseguente semantica della coerenza. Si passa da un controllo prima di ogni accesso, fino a scendere a un solo controllo durante il primo accesso a un file (normalmente quando il file viene aperto). Ogni accesso unito a un controllo di validità è ritardato rispetto a un accesso servito immediatamente dalla memoria cache. In alternativa si può iniziare un controllo a intervalli di tempo prefissati. A seconda della frequenza d'esecuzione il controllo di validità può caricare sia la rete sia il server.
2. **Verifica iniziata dal server.** Il server registra, per ogni client, i file (o le parti di file) che copia nella cache e interviene se individua una potenziale incoerenza. Una situazione di questo tipo si verifica quando due diversi client copiano un file in una cache in modi conflittuali. Se viene implementata la semantica UNIX (Paragrafo 11.5.3), si possono risolvere le potenziali incoerenze facendo in modo che il server svolga un ruolo attivo. Il server deve essere informato ogni volta che un file viene aperto, e per ogni apertura deve essere indicata la modalità (lettura o scrittura). Il server può intervenire quando individua un file che viene aperto contemporaneamente in modalità conflittuali, disabilitando la possibilità di copiare quel file nelle cache. Tale disabilitazione significa un passaggio a una modalità di funzionamento di servizio remoto.

I file system distribuiti sono oggi comunemente utilizzati e forniscono la condivisione di file all'interno di reti LAN e WAN. La complessità di implementazione di questi sistemi non deve essere sottovalutata. Va soprattutto considerato che devono essere indipendenti dal sistema operativo, per permetterne un utilizzo diffuso, e devono fornire disponibilità e buone prestazioni anche in presenza di notevoli distanze e connessioni non completamente affidabili.

17.10 Sommario

Un sistema distribuito è un insieme di unità d'elaborazione che non condividono memoria o un clock. Al contrario, ciascuna unità d'elaborazione ha la propria memoria locale e la comunicazione avviene attraverso varie linee di comunicazione, come bus ad alta velocità o Internet. Le dimensioni e le funzioni delle unità d'elaborazione di un sistema distribuito sono diverse. Il sistema può comprendere piccole unità d'elaborazione, personal computer e grandi calcolatori d'uso generale. Le unità d'elaborazione del sistema sono collegate attraverso una rete di comunicazione.

Un sistema distribuito fornisce all'utente l'accesso a tutte le risorse offerte dal sistema stesso. L'accesso a una risorsa condivisa può essere fornito tramite la migrazione dei dati, la migrazione delle computazioni o la migrazione dei processi. L'accesso può essere specificato dall'utente o implicitamente fornito dal sistema operativo e dalle applicazioni.

Le comunicazioni all'interno di un sistema distribuito possono essere realizzate tramite commutazione di circuito, messaggio, o pacchetto. La commutazione di pacchetto è il metodo più comunemente utilizzato su reti dati. Attraverso questi metodi i nodi del sistema possono scambiarsi messaggi.

Le pile di protocolli, com'è specificato dai modelli di stratificazione delle reti, aggiungono informazioni ai messaggi per assicurare che essi raggiungano la loro destinazione. Per tradurre il nome di un calcolatore nel corrispondente indirizzo di rete si deve usare un sistema di naming come il DNS; un altro protocollo, come l'ARP, può servire a tradurre l'indirizzo di rete in un indirizzo fisico del dispositivo, per esempio un indirizzo Ethernet. Se i sistemi risiedono su reti diverse sono necessari dei router per trasferire i pacchetti dalla rete di provenienza a quella di destinazione.

Vi sono diversi ostacoli da superare per far funzionare correttamente un sistema distribuito. Tra le varie problematiche da affrontare vi sono il naming dei nodi e dei processi nel sistema, la tolleranza ai guasti, il ripristino dopo un guasto e la scalabilità.

Un DFS è un servizio di file system i cui client, server e dispositivi di memorizzazione sono sparsi tra i siti di un sistema distribuito. Di conseguenza, l'attività di servizio si deve eseguire per mezzo della rete e invece di un unico sito di memorizzazione dei dati centralizzato vi sono più dispositivi di memorizzazione indipendenti. In teoria, un DFS dovrebbe apparire ai propri client come un file system centralizzato convenzionale. La molteplicità e la dispersione dei suoi server e dispositivi di memorizzazione dovrebbe essere resa trasparente. Un DFS trasparente facilita la mobilità

del client trasportando l'ambiente di quest'ultimo al sito nel quale un client apre le proprie sessioni.

Esistono parecchi metodi per il naming in un DFS. In quello più semplice i file si nominano attraverso una combinazione del loro nome di macchina e del loro nome locale, il che garantisce un nome unico per tutto il sistema. Un altro metodo, reso comune dall'NFS, fornisce mezzi per unire directory remote a directory locali, dando così l'impressione di un albero di directory coerente.

Le richieste d'accesso a un file remoto sono generalmente gestite con due metodi complementari. Con il servizio remoto, le richieste d'accesso sono consegnate al server; la macchina server esegue gli accessi e i loro risultati sono riportati al client. Con l'uso di cache, se i dati necessari per soddisfare la richiesta d'accesso non sono ancora stati copiati nella cache, una copia di quei dati viene portata dal server al client. Gli accessi si eseguono sulla copia presente nella cache. Il problema di mantenere le copie nelle cache coerenti con il file principale costituisce il problema della coerenza della cache.

Esercizi di ripasso

- 17.1** Perché il passaggio di pacchetti broadcast tra le reti sarebbe una pessima scelta per i gateway? Quali sarebbero i vantaggi di tale comportamento?
- 17.2** Discutete vantaggi e svantaggi del salvataggio in una cache di traduzioni di nomi per computer localizzati in domini remoti.
- 17.3** Quali sono i vantaggi e gli svantaggi nell'utilizzo della commutazione di circuito? Per quali tipologie di applicazioni la commutazione di circuito è una strategia applicabile?
- 17.4** Quali sono due difficili problemi che i progettisti devono risolvere nell'implementazione di un sistema di rete avente la qualità di essere trasparente?
- 17.5** La migrazione di processi su una rete eterogenea è di solito impossibile a causa delle differenze tra le architetture e tra i sistemi operativi. Descrivete un metodo per la migrazione di processi tra architetture diverse che eseguano:
 - a. lo stesso sistema operativo;
 - b. diversi sistemi operativi.
- 17.6** Per costruire un sistema distribuito robusto occorre conoscere quali tipi di guasti possono verificarsi.
 - a. Elencate tre tipi possibili di guasto in un sistema distribuito.
 - b. Specificate quali di questi guasti si possono verificare anche su sistemi centralizzati.
- 17.7** È sempre di cruciale importanza sapere che un messaggio inviato è correttamente giunto a destinazione? Se la vostra risposta è sì, motivatela. Se la vostra risposta è no, fornite un esempio.

17.8 Considerate un sistema distribuito con due siti A e B. Valutate se il sito A può distinguere fra i seguenti problemi

- a. Cade B.
- b. Cade il collegamento tra A e B.
- c. B è estremamente sovraccarico e il suo tempo di risposta è di 100 volte superiore al normale.

Quali implicazioni ha la vostra risposta sul ripristino in sistemi distribuiti?

Esercizi

17.9 Qual è la differenza fra la migrazione della computazione e dei processi? Quale delle due è più facile da implementare e perché?

17.10 Anche se il modello di rete OSI specifica sette strati di funzioni, nella maggior parte dei sistemi elaborativi si usano meno strati per realizzare una rete; spiegate perché e dite quali problemi può causare l'uso di un minor numero di strati.

17.11 Spiegate perché il raddoppio della velocità di un sistema in un segmento Ethernet può causare una diminuzione delle prestazioni della rete. Dite quali modifiche potrebbero migliorare la situazione.

17.12 Dite quali vantaggi porta l'uso di dispositivi specifici per le funzioni di router e gateway. Individuate gli svantaggi dell'impiego di calcolatori d'uso generale.

17.13 Dite in quali situazioni l'uso di un server dei nomi è più vantaggioso dell'uso di tabelle statiche nel calcolatore. Dite quali sono i problemi e le complicazioni connessi all'uso dei server dei nomi. Dite quali metodi si possono usare per ridurre il traffico generato dai server dei nomi per soddisfare le richieste di traduzione.

17.14 I server dei nomi sono organizzati in maniera gerarchica; a che scopo?

17.15 Gli strati inferiori del modello OSI prevedono un servizio di datagrammi, senza garanzie di consegna dei messaggi. Lo strato di trasporto di protocolli come TCP/IP serve ad assicurare affidabilità. Discutete vantaggi e svantaggi del mettere a disposizione la consegna affidabile al più basso strato possibile.

17.16 Quali conseguenze ha l'instradamento dinamico sul comportamento delle applicazioni? Quali applicazioni sono avvantaggiate se si sostituisce l'instradamento dinamico con quello virtuale?

17.17 Eseguite il programma della Figura 17.4 per determinare l'indirizzo IP dei seguenti nomi simbolici:

- hpe.pearsoned.it
- www.cs.yale.edu
- www.unimib.it
- www.westmistercollege.edu
- www.ietf.org

- 17.18** L'originario protocollo HTTP impiegava il TCP/IP come protocollo sottostante: per ciascuna pagina, grafico o applet si costruiva, utilizzava, quindi chiudeva una distinta sessione TCP. Con questo metodo, a causa del sovraccarico dovuto alla costruzione e chiusura delle connessioni TCP/IP, si verificavano problemi di prestazioni. Dite se l'uso dell'UDP al posto del TCP costituirebbe una buona alternativa e se è possibile fare altre modifiche che migliorino le prestazioni dell'HTTP.
- 17.19** Elencate i vantaggi e gli svantaggi della trasparenza di una rete di calcolatori per l'utente.
- 17.20** Elencate i vantaggi di un DFS rispetto al file system di un sistema centralizzato.
- 17.21** Dite quale tra i DFS presentati in questo capitolo gestirebbe nel modo più efficiente una grande applicazione di base di dati con più client. Spiegate la risposta.
- 17.22** Dite, motivando la risposta, se OpenAFS e NFS garantiscano: (a) trasparenza di locazione (b) indipendenza dalla locazione.
- 17.23** Dite in quali circostanze un client preferisce un DFS con trasparenza di locazione e in quali circostanze preferisce un DFS con indipendenza di locazione. Discutete i motivi di queste preferenze.
- 17.24** Dite quali aspetti di un sistema distribuito scegliereste per un sistema in esecuzione su una rete totalmente affidabile.
- 17.25** Considerate il file system distribuito dotato di stato (*stateful*) OpenAFS. Quali azioni occorre intraprendere a seguito della caduta di un server per preservare la coerenza garantita dal sistema?
- 17.26** Confrontate le tecniche di copiatura in una cache locale dei blocchi dei dischi nel sistema client con le tecniche di caching sul server.
- 17.27** OpenAFS è progettato per reggere un gran numero di client. Considerate tre tecniche adottate da OpenAFS che lo rendono scalabile.
- 17.28** Elencate vantaggi e svantaggi ottenuti col mapping degli oggetti nella memoria virtuale (come nel caso di Apollo Domain).
- 17.29** Descrivete alcune differenze fondamentali tra OpenAFS e NFS (Capitolo 12).

Note bibliografiche

[Tanenbaum 2010] e [Kurose e Ross 2013] offrono una trattazione generale delle reti di calcolatori. Internet e i suoi protocolli sono descritti in [Comer 1999] e [Comer 2000]. Il TCP/IP è trattato in [Fall e Stevens 2011 e [Stevens 1995]. [Stevens et al.] e [Stevens 1998] descrivono in dettaglio la programmazione di rete in UNIX.

Discussioni riguardanti il bilanciamento e la condivisione del carico sono presentate da [Harchol-Balter e Downey 1997] e [Vee e Hsu 2000]. [Harish e Owens 1999]

discute il bilanciamento del carico dei server DNS. Il Network File System di Sun (NFS) è presentato in [Callaghan 2000] e [Sandberg et al. 1985]. Il sistema OpenAFS è discusso in [Morris et al. 1986], [Howard et al. 1988] e [Satyanarayanan 1990]. All’indirizzo <http://www.openafs.org> sono disponibili informazioni su OpenAFS. Il file system Andrew è analizzato in [Howard et al. 1988]. Il metodo MapReduce di Google è descritto alla pagina <http://research.google.com/archive/mapreduce.html>.

Bibliografia

- [Callaghan 2000] B. Callaghan, *NFS Illustrated*, Addison-Wesley, 2000.
- [Comer 1999] D. Comer, *Internetworking with TCP/IP*, Vol. II, 3° Ed., Prentice Hall, 1999.
- [Comer 2000] D. Comer, *Internetworking with TCP/IP*, Vol. I, 4° Ed., Prentice Hall, 2000.
- [Fall e Stevens 2011] K. Fall e R. Stevens, *TCP/IP Illustrated*, Volume 1: The Protocols, 2° Ed., John Wiley and Sons, 2011.
- [Harchol-Balter e Downey 1997] M. Harchol-Balter e A. B. Downey, “Exploiting Process Lifetime Distributions for Dynamic Load Balancing”, *ACM Transactions on Computer Systems*, Vol. 15, Num. 3, p. 253–285, 1997.
- [Harish e Owens 1999] V. C. Harish e B. Owens, “Dynamic Load Balancing DNS”, *Linux Journal*, Vol. 1999, Num. 64, 1999.
- [Howard et al. 1988] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan e R. N. Sidebotham, “Scale and Performance in a Distributed File System”, *ACM Transactions on Computer Systems*, Vol. 6, Num. 1, p. 55–81, 1988.
- [Kurose e Ross 2013] J. Kurose e K. Ross, *Computer Networking—A Top–Down Approach*, 6° Ed., Addison-Wesley, 2013.
- [Morris et al. 1986] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S.H. Rosenthal e F.D. Smith, “Andrew: A Distributed Personal Computing Environment”, *Communications of the ACM*, Vol. 29, Num. 3, p. 184–201, 1986.
- [Sandberg et al. 1985] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh e B. Lyon, “Design and Implementation of the Sun Network Filesystem”, *Proceedings of the Summer USENIX Conference*, p. 119–130, 1985.
- [Satyanarayanan 1990] M. Satyanarayanan, “Scalable, Secure and Highly Available Distributed File Access”, *Computer*, Vol. 23, Num. 5, p. 9–21 1990.
- [Steven et al.] R. Steven, B. Fenner, and A. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API*, 3° Ed., Wiley, 2003.
- [Stevens 1995] R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1995.
- [Stevens 1998] W. R. Stevens, *UNIX Network Programming—Volume II*, Prentice Hall, 1998.
- [Tanenbaum 2010] A. S. Tanenbaum, *Computer Networks*, 5° Ed., Prentice Hall, 2010.
- [Vee e Hsu 2000] V. Vee e W. Hsu, “Locality-Preserving Load-Balancing Mechanisms for Synchronous Simulations on Shared-Memory Multiprocessors”, *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, p. 131–138, 2000.

Indice analitico

A

Address extension (PAE), 422
Advanced encryption standard (AES), 749
Alberi, 37, 39, 55
Alberi red-black, 39
Albero binario, 37
Albero binario di ricerca bilanciato, 38
Algoritmo a orologio (clock), 458
Algoritmo di cifratura simmetrica, 748
Algoritmo di controllo dell'ammissione, 312
Ambienti di programmazione
 cloud computing, 47
 mobile computing, 41
 sistemi distribuiti, 42
 sistemi embedded real-time, 48
 virtualizzazione, 45
AMD, tecnologia di virtualizzazione (AMD-V), 798
AMD64, architettura, 423, 790
Amdahl, legge di, 183
Android, sistema operativo, 41, 95, 194
API (*application program interface*), 70
Apple iPad, 41, 66-67, 94
Applicazioni di realtà aumentata, 41
Aqua, interfaccia, 66, 93
ARM, architettura, 424
Array, 35
ASID, identificatori dello spazio d'indirizzi (*address-space identifiers*), 408
Assembly, linguaggio, 85
Attributi estesi dei file, 555

Avvicendamento

nei sistemi mobili, 391
standard, 389

Autenticazione, 727

multifattoriale, 763

B

Background, processi in, 78, 82, 128
Bitmap, 39
Bridging, 811
Bug, 73

C

Chiamata di procedura locale avanzata (ALPC), 149
Chrome, 137
Clone, 791
Cloud computing, 47, 793
Cloud ibrido, 47
Cloud privato, 47
Cloud pubblico, 47
Cluster asimmetrici, 19
Cluster simmetrici, 19
Cocoa Touch, 93, 94
Code, 35
Computing
 mobile, 41
Concorrenza, 183
Concurrency Runtime (ConcRT), 323
Contenitori (zone), 806
Contenitori di applicazioni, 789, 806
Crittografia a chiave pubblica, 750

D

Dalvik, macchina virtuale, 95
 Dati specifici dei thread, 205
 Demoni, 8
 Difesa in profondità, 763
 Digital Equipment Corporation (DEC), 413
Disk
 a stato solido, 513
 Dispatcher, 295, 310, 320
 Dispositivi mobili
 avvicendamento nei, 391-392
 multitasking nei, 128
 DMA, controllore, 652, 659
 Domini di protezione, 798

E

Earliest-deadline-first (EDF), scheduling, 314
 EC2 (*elastic compute cloud*), 47
 EDF (*earliest-deadline-first*), scheduling, 314
 Emulatore, 731
 Emulazione, 45, 805
 Encryption (cifratura), 747
 chiave pubblica, 750
 Erlang, linguaggio, 248, 264
 exit() chiamata di sistema, 131, 133

F

FIFO, 37
 Figli, 37
 Figlio destro, 37
 Figlio sinistro, 37
 Finestra di informazioni (Mac OS X), 555
 Foreground, processi, 128, 298
 Fork-join, strategia, 190

G

GCD (*Grand Central Dispatch*), 200
 Gesti sullo schermo (gestures), 66

Gestore della macchina virtuale, (VMM), 25, 46, 788
 GNOME, desktop, 66
 GPS (*global positioning system*), 41
 Grand Central Dispatch (GCD), 200
 GUI (*graphical user interfaces*), 62

H

Hadoop, 848-849
 Hadoop distributed file system (HDFS), 851
 Hardware
 machine virtuali, 788
 Hash, collisioni, 608
 Hash, funzioni, 38, 752
 Hash, funzioni e mappe, 38
 HDFS (*hadoop distributed file system*), 851
 Hypercall, 804
 Hypervisor di tipo 0, 801
 Hypervisor di tipo 1, 801-803
 Hypervisor di tipo 2, 803

I

IA-32, architettura, 419
 paginazione in, 421
 segmentazione in, 420
 IA-64 architettura, 423
 IaaS (*infrastructure as a service*), 47
 Ibridi, sistemi, 93
 Android, 94
 iOS, 94
 Mac OS X, 93
 IDSs (*intrusion-detection systems*), 767
 Infrastructure as a service (IaaS), 47
 Intel, processori
 IA-32, architettura, 419
 IA-64 architettura, 423
 Interfacce
 scelta delle, 67
 Internet Key Exchange (IKE), 755
 Interpretazione, 45
 I/O (input/output)
 macchine virtuali, 810

I/O vettorizzato, 668
 iOS, sistema operativo, 93, 94
 IP (Internet Protocol), 755
 iPad, *vedi* Apple iPad

K

Kernel, moduli
 Linux, 107

L

Latenza
 nei sistemi real-time, 308-310
 target, 319
 Latenza di dispatch, 287
 Latenza relativa all'evento, 309
 Legge di Moore, 6
 LIFO, 36
 Linguaggi imperativi (o procedurali), 263
 Linguaggi interpretati, 805
 Linguaggi procedurali, 263
 Linux
 moduli del kernel, 107
 Linux, sistemi
 ottenere la dimensione della pagina su, 403
 Lista circolare, 36
 Lista doppiamente concatenata, 36
 Liste, 36
 Liste semplicemente concatenate, 36
 Livelli di privilegio, 25
 Lock mutex, 231
 Love bug, virus, 770

M

Mac OS X, sistema operativo, 93
 Memoria
 gestione della con le macchine virtuali, 808
 perdite di, 111
 transazionale, 260
 Memory-management unit (MMU), 386-387

Micro TLB, 425
 Migrazione
 con le macchine virtuali, 812
 Migrazione in tempo reale (macchine virtuali), 792
 Mobile computing, 41
 Moore, legge di, 6
 Mutex lock, 231

N

NAT (*network address translation*), 811
 Nested page tables (NPTs), 797
 Network address translation (NAT), 811
 NPT (*nested page tables*), 797

O

OpenMP, 198, 262
 OpenSolaris, 52
 Oracle SPARC Solaris, 418
 OSI modello, 840
 Overcommitment (sovraassegnazione), 807

P

PaaS (*platform as a service*), 47
 Paginazione
 e Oracle SPARC Solaris, 418
 Parallelismo, 183, 185
 Parallelismo dei dati, 185
 Parallelismo delle attività, 185
 Paravirtualizzazione, 803
 Partizione di controllo, 801
 Periodo di sviluppo del sistema, 792
 Personalità del sistema operativo, 93
 PFF (*page-fault-frequency*), 470
 Platform as a service (PaaS), 47
 Pop, 37
 POSIX
 scheduling real-time, 316

Problema della sezione critica
e mutex locks, 231

Processi
in background, 298
in foreground, 298
periodici, 311

Protocolli di scoperta, 44

Pseudo-driver di periferica, 809

Pthreads
punto di cancellazione, 204

Punto di ingresso del modulo, 108

Punto di uscita del modulo, 108

Push, 36

R

RC4, algoritmo, 750

RDP, 797

Regolazione automatica del working
set, 489

Replicazione dei file, 850

ROM (*read-only memory*), 8, 103, 526

Router, 836

RR, algoritmo di scheduling circolare,
295

S

SaaS (*software as a service*), 47

Scala, linguaggio, 264

Scheduling con priorità, 293

Scheduling con priorità proporzionale
alla frequenza, 313
algoritmo di, 313-314

Scheduling della CPU

a quote proporzionali, 315
basato sulla priorità, 310
con priorità proporzionale alla
frequenza, 312

EDF (*earliest-deadline-first
scheduling*), 314

machine virtuali, 787

minimizzazione della latenza, 308

POSIX real-time scheduling, 316

real-time, 308

Scheduling real-time della CPU, 308

Servizio, sistema operativo, 128

Sincronizzazione dei processi

approcci alternativi, 260

Sistemi debolmente accoppiati, 18

Sistemi multicore, 15, 18, 182

Sistemi multiprocessore, 186

Sistemi operativi
ibridi, 93

SJF, algoritmo di scheduling, 330

Skype, 45

SLAB, allocatore, 481

SLOB, allocatore, 481

SLUB, allocatore, 481

Software as a service (SaaS), 47

Solid-state disk (SSD), 12, 513, 524

Sospendere una macchina virtuale, 791

Sottosistemi, 149

SPARC, 418

SSTF algoritmo di scheduling, 520

Stack, 35

Storage

thread-local, 205

Storage, gestione, 811

con macchine virtuali, 812

Strutture di controllo della macchina
virtuale (VMCSs), 798

Superuser, 761

Swapping standard, 389

SYSGEN, 101

T

Tabella dei file aperti, 557

Terminal, applicazione, 107

Thin client (computer network), 40

Thread

threading implicito, 196

Thread pool, 197

Threading asincrono, 189

Threading implicito, 196

Time sharing (multitasking), 118, 122

Touch screen (touchscreen computing),
5, 22, 66

Traduzione binaria, 795
 Transazioni
 atomiche, 626
 Transmission Control Protocol/Internet
 Protocol (TCP/IP), 840
 Trap-and-emulate, metodo, 794

U

UAC (*user account control*), 777
 USB (*universal serial buses*), 487
 User Account Control (UAC), 777

V

VAX minicomputer, 413
 VCPU (virtual CPU), 794
 Virtual CPU (VCPU), 794
 Virtualizzazione, 45
 dell'ambiente di programmazione,
 804-805
 e componenti dei sistemi operativi, 806
 e contenitori di applicazioni, 806
 ed emulazione, 805
 gestione della memoria, 808
 gestione dello storage, 811
 I/O, 810
 migrazione in tempo reale, 812-814
 scheduling della CPU, 806
 vantaggi e caratteristiche, 791
 VMCSs (*virtual machine control
 structures*), 798
 VMware, 788, 799, 790

W

wait(), chiamata di sistema, 130
 Windows 7
 contesto di sicurezza, 777
 descrittore di sicurezza, 777
 scheduling in, 323
 Windows, Task Manager, 97
 Workstation (VMWare), 789, 815

X

x86-64, architettura, 419
 Xen, 790

Z

Zona, 806

Casi di studio

A questo punto si possono riunire i concetti illustrati in questo libro descrivendo alcuni sistemi operativi reali. Due sono trattati in dettaglio: Linux e Windows 7. Il sistema Linux è stato scelto per diverse ragioni: è diffuso, è liberamente disponibile e rappresenta un sistema UNIX completo nelle sue funzioni. Ciò offre a chi studia i sistemi operativi l'opportunità di esaminare – e modificare – il codice sorgente di un vero sistema operativo.

Anche il sistema Windows 7 è analizzato nei particolari. Questo recente sistema operativo prodotto da Microsoft sta diffondendosi non solo sul mercato dei PC, ma anche su quello dei server per gruppi di lavoro. Lo si è scelto perché offre la possibilità di studiare un sistema operativo moderno che differisce radicalmente da UNIX sia nel progetto sia nell'implementazione.

Sono poi brevemente presentati altri sistemi operativi molto influenti. Infine sono presi in esame altri due sistemi operativi. Il sistema operativo FreeBSD è un altro sistema UNIX ma, mentre Linux combina le caratteristiche di diversi tipi di sistemi UNIX, FreeBSD si basa sul modello del BSD. Il codice sorgente di FreeBSD, come quello di Linux, è liberamente disponibile. Mach è un moderno sistema operativo compatibile con BSD UNIX.

CAPITOLO

18

OBIETTIVI DEL CAPITOLO

- Evoluzione storica del sistema operativo UNIX – da cui Linux deriva – e presentazione dei principi alla base del progetto di Linux.
- Esame del modello di processo in Linux, dello scheduling e della comunicazione tra processi.
- Presentazione della gestione della memoria in Linux.
- Implementazione dei file system e gestione dei dispositivi di I/O.

Linux

Aggiornamento a cura di Robert Love

Questo capitolo presenta un’analisi approfondita del sistema operativo Linux. Esaminando un sistema reale nel dettaglio potremo vedere come i concetti che abbiamo discusso siano in relazione tra loro e vengono applicati praticamente.

Linux è una versione di UNIX che negli ultimi decenni ha avuto una crescente diffusione e viene ora utilizzato sia su piccoli dispositivi come i telefoni cellulari che su grandi supercomputer. In questo capitolo è trattata la storia e lo sviluppo di Linux, e sono esaminate le sue interfacce per il programmatore e l’utente, interfacce che devono molto alla tradizione di UNIX. Si analizzano anche i metodi adottati per il progetto e l’implementazione di queste interfacce.

Linux è un sistema operativo in rapida evoluzione: il kernel descritto in questo capitolo è la versione 3.2, rilasciata alla fine del 2012.

18.1 Storia di Linux

Il sistema operativo Linux assomiglia molto a un qualunque altro sistema UNIX, e in effetti la compatibilità con UNIX è stata uno degli obiettivi principali nella sua progettazione; tuttavia, il sistema Linux è molto più recente della maggior parte dei sistemi UNIX. Il suo sviluppo ha avuto inizio nel 1991 quando Linus Torvalds, uno studente finlandese, cominciò a sviluppare un kernel piccolo ma autosufficiente per la CPU 80386, la prima vera CPU a 32 bit della gamma Intel per i PC-compatibili.

Già fin dagli inizi del suo sviluppo il codice sorgente di Linux fu reso gratuitamente disponibile tramite la rete Internet, senza costi e con restrizioni minime sulle possibilità di distribuirlo: ne è risultata una storia ricca di contributi da parte di sviluppatori di tutto il mondo che comunicavano quasi esclusivamente tramite Internet. Da un kernel iniziale che realizzava parzialmente un ridotto sottoinsieme dei servizi di sistema di UNIX, il sistema Linux è cresciuto fino a includere tutte le funzionalità che ci si aspetta di trovare in un moderno sistema UNIX.

I primi stadi di sviluppo di Linux avevano a che fare in gran parte con il kernel del sistema operativo, il programma privilegiato che interagisce direttamente con l'hardware del calcolatore e gestisce tutte le risorse del sistema; chiaramente per ottenere un sistema operativo completo occorre molto più di questo kernel. È utile distinguere il kernel di Linux da un sistema Linux completo. Il **kernel** di Linux è un programma completamente originale sviluppato interamente dalla comunità Linux; il **sistema** Linux, nella sua forma odierna, incorpora una moltitudine di componenti, alcuni scritti *ex novo*, altri presi in prestito da diversi progetti di sviluppo e altri ancora creati in collaborazione con altri gruppi di programmatore.

Il sistema Linux di base è un ambiente standard per le applicazioni e per il codice scritto dagli utenti, ma non impone convenzioni rigide sulla gestione complessiva delle funzionalità. Un ulteriore livello organizzativo si è reso necessario con la progressiva maturazione di Linux: tale necessità è stata soddisfatta da varie distribuzioni. Una **distribuzione Linux** include tutti i componenti ordinari del sistema operativo più una serie di strumenti di gestione che semplifica l'installazione iniziale, gli aggiornamenti successivi, e l'installazione o la rimozione di altri pacchetti. Un moderno pacchetto di distribuzione, inoltre, fornisce di solito strumenti per la gestione dei file system, la creazione e gestione degli account utente, l'amministrazione dei servizi di rete, browser Web, word processor e così via.

18.1.1 Kernel del sistema Linux

La prima versione del kernel di Linux resa disponibile al pubblico fu la 0.01, datata 14 maggio 1991; non forniva alcun servizio di rete, funzionava solo su PC con CPU compatibile con Intel 80386, e la sua gestione dei driver dei dispositivi era estremamente limitata. Anche il sottosistema per la memoria virtuale era abbastanza elementare, e non supportava i file memory-mapped; tuttavia, anche questa primissima versione permetteva la condivisione delle pagine con copiatura su scrittura (copy-on-write) e spazi degli

indirizzi protetti. Il solo file system disponibile era quello di Minix, perché i primi kernel Linux furono sviluppati su piattaforme Minix.

La successiva versione principale, Linux 1.0, fu rilasciata il 14 marzo 1994; rappresentava il culmine di tre anni di rapido sviluppo del kernel. Forse la più importante nuova caratteristica riguardava la gestione dei servizi di rete: incorporava l'implementazione standard UNIX dei protocolli TCP/IP, e un'interfaccia socket compatibile con BSD UNIX per la programmazione di rete; grazie ai nuovi driver era anche possibile utilizzare il protocollo IP su una rete Ethernet o su linee seriali e modem usando i protocolli PPP o SLIP.

Il kernel 1.0 includeva anche un nuovo file system molto migliorato che non soffriva delle limitazioni del file system originario Minix; inoltre, supportava diversi tipi di controller SCSI per l'accesso ai dischi ad alte prestazioni. Anche il sottosistema per la memoria virtuale era stato migliorato, e dava ora la possibilità della paginazione per effettuare lo swapping dei file e di effettuare il memory mapping di file arbitrari (anche se solo per operazioni di lettura).

Questa versione supportava inoltre una serie di ulteriori dispositivi che, seppur ancora limitati alla piattaforma Intel-PC, comprendevano floppy disk e CD-ROM, schede audio, vari tipi di mouse e tastiere internazionali. Il kernel era anche in grado di simulare le operazioni in virgola mobile per i calcolatori basati sulla CPU 80386 che non erano dotate del coprocessore matematico 80387, e realizzava inoltre la **comunicazione tra processi** (IPC) nello stile di UNIX System V, compresa la memoria condivisa, i semafori e le code di messaggi.

A questo punto si cominciò a sviluppare la versione 1.1 del kernel, ma fu necessario distribuire in seguito numerose patch per la versione 1.0, nella quale erano stati riscontrati errori. Come convenzione di numerazione per le versioni del kernel fu adottato lo schema seguente: le versioni con un numero dispari dopo il punto, come la 1.1 o la 2.5, sono **kernel di sviluppo**; quelle con numero pari dopo il punto sono invece **kernel di produzione**. Gli aggiornamenti rispetto alle versioni stabili sono da intendersi solo come correzioni, mentre i kernel di sviluppo potrebbero includere nuovi servizi relativamente poco collaudati. Come vedremo, questo modello è rimasto in vigore fino alla versione 3.

Il kernel 1.2 fu rilasciato nel marzo 1995; non rappresentava un miglioramento paragonabile a quello della versione 1.0, ma in ogni modo permetteva la gestione di una ben più ampia varietà di hardware, compreso il nuovo bus PCI. Gli sviluppatori inclusero anche la possibilità di usare la modalità virtuale 8086 della CPU 80386 – un'altra funzione specificamente dedicata ai PC – che permetteva di emulare il sistema operativo DOS per PC. Fu inoltre aggiornata l'implementazione del protocollo IP, con l'aggiunta del supporto per accounting e firewall, e venne incluso un semplice supporto per i moduli del kernel caricabili e scaricabili dinamicamente.

Il kernel 1.2 fu anche l'ultima versione dedicata ai soli PC: i file sorgenti di Linux 1.2 comprendevano già un supporto parziale per le CPU SPARC, Alpha e MIPS, ma la piena integrazione di queste altre architetture ebbe inizio solo dopo la diffusione della versione stabile 1.2.

Linux 1.2 si concentrava sul problema di gestire un maggiore varietà di hardware e di fornire implementazioni più complete delle funzioni esistenti. Molti nuovi servizi erano a quel tempo in via di sviluppo, ma l'integrazione del nuovo codice all'interno del kernel fu rimandata a dopo la distribuzione della versione stabile del kernel 1.2; di conseguenza, la versione 1.3 portò con sé una gran quantità di nuove funzionalità.

Questo materiale fu infine distribuito come Linux 2.0 nel giugno 1996. L'incremento del numero di versione principale era giustificato da due nuove caratteristiche di basilare importanza: il supporto di diverse architetture, tra cui il supporto nativo a 64 bit per Alpha, e il supporto al multiprocessing simmetrico (SMP). Inoltre, il codice di gestione della memoria fu notevolmente migliorato per fornire una cache unificata per i dati del file system indipendente dalla cache dei dispositivi a blocchi. Grazie a questa modifica, il kernel poté offrire prestazioni notevolmente migliorate per quel che riguarda il file system e la memoria virtuale. Per la prima volta i meccanismi di caching del file system furono estesi ai file system di rete e vennero supportate le regioni scrivibili mappate in memoria. Tra le altre importanti novità citiamo i thread interni al kernel, la gestione delle dipendenze fra i moduli caricabili e il caricamento automatico dei moduli richiesti, il meccanismo delle quote per il file system e l'adozione delle classi di scheduling dei processi per l'elaborazione in tempo reale, compatibili con lo standard POSIX.

Nel 1999 fu rilasciato Linux 2.2 con ulteriori miglioramenti. È stato introdotto un porting su sistemi UltraSPARC. I servizi di rete sono stati affinati con un servizio di firewall più flessibile, una migliore gestione del traffico e dell'instradamento, e l'uso di finestre TCP ampie e di acknowledgement selettivi. Era diventato possibile leggere i dischi Acorn, Apple ed NTFS, ed era stato stato migliorato l'NFS con l'aggiunta di un demone, eseguito in modalità kernel. La gestione dei segnali, delle interruzioni, e di alcuni aspetti dell'I/O era eseguita impiegando un sistema di lock più selettivo per migliorare le prestazioni dell'SMP.

Fra i miglioramenti delle versioni 2.4 e 2.6, citiamo i progressi al supporto dei sistemi SMP, i journaling file system, oltre a migliorie alla gestione della memoria e dell'I/O a blocchi. Nella versione 2.6 lo scheduler dei processi è stato modificato in modo da usare un efficiente algoritmo di scheduling che esegue in tempo $O(1)$. Inoltre il kernel di Linux 2.6 applica la prelazione anche ai processi eseguiti in modalità kernel.

La versione 3.0 del kernel Linux è stata rilasciata nel luglio 2011. Il passaggio a un nuovo numero di versione principale è stato effettuato per ricordare il ventesimo anniversario di Linux. Le nuove caratteristiche includono un migliorato supporto della virtualizzazione, un nuovo meccanismo di write-back delle pagine, miglioramenti al sistema di gestione della memoria e un'ulteriore novità per quanto riguarda lo scheduler: il Completely Fair Scheduler (CFS). Nel resto di questo capitolo ci concentriamo su questo nuovo kernel.

18.1.2 Il sistema Linux

Come abbiamo già osservato, il kernel rappresenta per molti aspetti la parte centrale del progetto Linux, ma altri componenti concorrono a costituire un sistema operativo Linux completo. Mentre il kernel è composto di codice scritto *ex novo* specificamente per il progetto Linux, molti programmi aggiuntivi che completano il sistema non sono stati espressamente concepiti per il sistema Linux, ma sono condivisi da un certo numero di sistemi operativi ispirati a UNIX. In particolare il sistema Linux adotta molti strumenti sviluppati come parti del sistema operativo BSD di Berkeley, del sistema X Window del MIT, e del progetto GNU della Free Software Foundation.

Questa condivisione ha funzionato in entrambe le direzioni: lo sviluppo delle principali librerie di sistema di Linux ha avuto origine dal progetto GNU, ma la comunità Linux le ha poi notevolmente migliorate affrontandone le carenze e inefficienze, ed eliminando gli errori. Altri componenti, per esempio il compilatore per il linguaggio C del progetto GNU, **GCC (GNU C compiler)**, erano già di qualità sufficientemente alta per essere direttamente usati in Linux. Gli strumenti di gestione dei servizi di rete sono stati derivati da codice originariamente scritto per BSD 4.3, ma discendenti più recenti del BSD come FreeBSD hanno preso a loro volta in prestito codice dal sistema Linux. Tra gli esempi di questa condivisione vi sono la libreria di simulazione delle operazioni in virgola mobile per le CPU Intel e i driver dei dispositivi audio per PC.

La manutenzione complessiva del sistema Linux è curata da una rete alquanto lasca di sviluppatori che collaborano tramite la rete Internet, e la responsabilità dell'integrità di alcune specifiche componenti è assegnata a piccoli gruppi o individui. Un limitato numero di archivi ftp pubblici su Internet agiscono *de facto* come repository standard di questi componenti. Anche il documento **File System Hierarchy Standard** è curato dalla comunità Linux al fine di preservare la compatibilità fra le varie parti del sistema: esso specifica la struttura generale di un file system Linux, stabilendo sotto quali nomi di directory i file di configurazione, le librerie, i file eseguibili di sistema e i file di dati dovrebbero essere archiviati.

18.1.3 Distribuzioni Linux

In teoria chiunque potrebbe installare un sistema Linux compilando le ultime versioni del codice sorgente ottenibili dai siti ftp; una volta questo era esattamente ciò che ogni utente Linux doveva fare. A mano a mano che il sistema maturava, però, diversi individui o gruppi hanno cercato di rendere quest'inconvenienza meno pesante fornendo pacchetti precompilati standard di facile installazione.

Queste raccolte, o distribuzioni, comprendono molto di più del sistema Linux di base: generalmente forniscono strumenti aggiuntivi per l'installazione e la gestione del sistema, e anche pacchetti precompilati e pronti per l'installazione che forniscono molti comuni strumenti del sistema UNIX, per esempio server di news, Web browser, text editor e persino giochi.

Le prime distribuzioni fornivano semplicemente un modo per estrarre e collocare i file nelle posizioni appropriate. Uno dei contributi importanti delle moderne versioni

è la gestione avanzata dei pacchetti: esse comprendono un base dati di tracciatura dei pacchetti che permette di installare, aggiornare e rimuovere senza difficoltà i pacchetti desiderati.

Nei primi anni di vita di Linux la versione SLS fu la prima raccolta di pacchetti Linux classificabile come una vera e propria distribuzione; tuttavia, anche se poteva essere installata in un'unica soluzione, non forniva gli strumenti di gestione dei pacchetti che ci si può aspettare oggi. La versione **Slackware** rappresentò complessivamente un gran salto di qualità, nonostante la sua gestione dei pacchetti fosse piuttosto scadente, essa è ancora oggi assai diffusa tra la comunità Linux.

Dal rilascio di Slackware, sono divenute disponibili molte nuove distribuzioni commerciali e non commerciali. Due versioni di grande diffusione sono **Red Hat** e **Debian**, la prima prodotta da un'impresa commerciale, la seconda fornita dalla comunità Linux. Altre distribuzioni commerciali sono **Canonical** e **SUSE**. Ci sono molte altre versioni di Linux, ma sono troppe per poterle elencare tutte. Questa gran varietà non impedisce la compatibilità fra distribuzioni diverse. Il formato RPM per i file contenenti i pacchetti è adottato o almeno compreso dalla maggior parte delle distribuzioni, e le applicazioni commerciali distribuite in questo formato si possono installare ed eseguire con qualunque versione di Linux che accetti i file RPM.

18.1.4 Licenze Linux

Il kernel di Linux è distribuito in accordo alla versione 2.0 della GNU General Public License (GPL), i cui termini sono stati stabiliti dalla Free Software Foundation. Il sistema Linux non è software **public domain**: questa locuzione implica la rinuncia ai diritti d'autore, mentre i diritti sul codice di Linux sono tuttora detenuti dai vari autori. Tuttavia, il sistema Linux è *libero* nel senso che si può copiare, modificare e usare in qualunque modo si desideri, e si può far circolare (o vendere) le proprie copie.

La principale conseguenza dei termini della licenza di Linux è che chiunque lo usi o crei un prodotto da esso derivato (un'attività legittima) non può distribuire il suo prodotto senza il codice sorgente. I programmi distribuiti in accordo con la GPL non si possono ridistribuire in forma puramente binaria. Se rilasciate software che comprende una qualsiasi componente coperta da GPL dovete rendere il codice sorgente disponibile sotto GPL insieme a qualsiasi distribuzione del codice binario (si noti che questa limitazione non impedisce la realizzazione o persino la vendita di versioni puramente binarie, purché chiunque riceva i file eseguibili abbia la possibilità di ottenere il codice sorgente a un ragionevole prezzo).

18.2 Princìpi di progettazione

Dal punto di vista del progetto complessivo il sistema Linux assomiglia a qualunque altro sistema UNIX tradizionale non basato su microkernel. È un sistema multiutente multitasking con prelazione dotato di una completa serie di strumenti compatibili con UNIX; il suo file system aderisce alla tradizionale semantica UNIX, e anche il modello

standard di comunicazione in rete di UNIX è pienamente realizzato. I dettagli interni della progettazione di Linux sono stati assai influenzati dalla storia del suo sviluppo.

Sebbene oggi il sistema Linux possa essere eseguito su un'ampia gamma di piattaforme, esso fu inizialmente concepito esclusivamente per l'architettura dei PC, e gran parte di questa prima fase di sviluppo fu portata a termine da appassionati e non da professionisti dotati di infrastrutture di ricerca e sviluppo ben finanziate: il risultato fu che, fin dall'inizio, Linux ottenne da risorse limitate il massimo possibile di capacità funzionali. Anche se oggi funziona perfettamente su calcolatori con più unità d'elaborazione con gigabyte di memoria centrale e terabyte di spazio di dischi, Linux può ancora funzionare con meno di 16 MB di RAM.

A mano a mano che i PC divenivano più potenti e la memoria e i dischi più economici, l'originario kernel “minimalista” di Linux s'arricchiva di nuovi servizi tipici del sistema UNIX; la velocità e l'efficienza sono ancora oggi obiettivi importanti, ma molta ricerca recente e attualmente in corso si concentra su un terzo importante obiettivo: la standardizzazione. Uno dei prezzi che occorre pagare per la diversità delle versioni di UNIX disponibili è il fatto che non sia garantita la possibilità di trasferire codice sorgente fra due sue versioni: anche quando le chiamate di sistema sono le medesime nei due sistemi, non è detto che il loro comportamento sia identico. Gli standard POSIX comprendono un insieme di specifiche riguardanti diversi aspetti del comportamento di un sistema operativo, ed esistono documenti POSIX relativi ai servizi più comuni dei sistemi operativi e a estensioni quali i thread dei processi e le operazioni in tempo reale. Il sistema Linux è concepito al fine di conformarsi ai documenti POSIX, e almeno due sue distribuzioni hanno ottenuto la certificazione POSIX.

Presentando interfacce standard sia al programmatore sia all'utente, il sistema operativo Linux riserva poche sorprese a chiunque conosca già il sistema UNIX, queste interfacce non saranno trattate qui nei dettagli. Il contenuto dei paragrafi sull'interfaccia per il programmatore e per l'utente del sistema BSD vale anche per il sistema Linux. Si noti solo che ordinariamente l'interfaccia per il programmatore di Linux segue la semantica di UNIX SVR4 e non quella del BSD; per ottenere la semantica del BSD, nei casi in cui vi sono differenze significative, è disponibile una raccolta di librerie.

Esistono molti altri standard nel mondo UNIX, ma la certificazione di Linux in conformità a essi è a volte rallentata dal fatto che di solito queste certificazioni sono disponibili solo a pagamento, e la certificazione di un sistema operativo rispetto alla maggior parte degli standard è un processo molto costoso. Ciononostante, poiché la capacità di eseguire un'ampia gamma di applicazioni è importante per qualunque sistema operativo, l'implementazione di standard è un obiettivo fondamentale nello sviluppo di Linux anche in assenza di certificazioni formali. Oltre allo standard di base POSIX, Linux attualmente comprende le estensioni POSIX per il threading (Pthreads) e un sottoinsieme delle estensioni POSIX per il controllo dei processi in tempo reale.

18.2.1 Componenti del sistema

Il sistema Linux è composto da tre porzioni principali di codice, in linea con le tradizionali versioni del sistema UNIX.

1. **Kernel.** Il kernel è responsabile di tutte le astrazioni importanti messe in atto dal sistema operativo, fra cui la memoria virtuale e i processi.
2. **Librerie di sistema.** Le librerie di sistema definiscono un insieme standard di funzioni grazie alle quali le applicazioni interagiscono col kernel, e realizzano la maggior parte dei servizi forniti dal sistema operativo che non necessitano dei pieni privilegi del kernel. La libreria di sistema più importante è la libreria C, conosciuta come libc. Oltre a fornire la libreria C standard, libc implementa la parte in modalità utente dell’interfaccia delle chiamate di sistema Linux e altre interfacce critiche a livello di sistema.
3. **Utilità di sistema.** Le utilità di sistema sono programmi che eseguono singoli compiti specializzati di gestione. Alcune di loro vengono chiamate solo una volta per inizializzare e configurare qualche aspetto del sistema; altre – note come **demoni** nella terminologia UNIX – rimangono permanentemente in esecuzione per eseguire compiti come la gestione delle richieste di nuove connessioni, la gestione delle richieste d’accesso dai terminali o l’aggiornamento dei file di log.

Nella Figura 18.1 sono illustrati i vari elementi che costituiscono un sistema Linux completo. La distinzione più importante in questo contesto è quella fra il kernel e tutto il resto. Tutto il codice del kernel è eseguito dalla CPU in modo privilegiato e con piena possibilità d’accesso alle risorse fisiche del calcolatore: nel contesto del sistema Linux ci si riferisce a questa modalità privilegiata come alla **modalità kernel**. Il kernel di Linux non contiene codice utente: tutto il codice di supporto al sistema operativo che non ha bisogno di essere eseguito in modalità kernel si trova nelle librerie di sistema e viene eseguito in modalità utente. A differenza della modalità kernel, la modalità utente ha accesso soltanto a un sottoinsieme controllato delle risorse di sistema.

Anche se parecchi sistemi operativi moderni hanno adottato per i loro kernel un’architettura basata sullo scambio di messaggi, Linux si conforma al modello storico dello UNIX: il kernel è un unico blocco monolitico di codice binario. La ragione

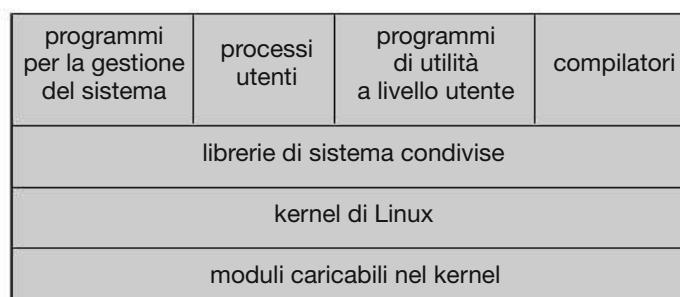


Figura 18.1 Componenti del sistema Linux.

principale di ciò sono le prestazioni. Poiché tutto il codice e le strutture dati del kernel sono contenute in un unico spazio d'indirizzi, non vi è necessità d'alcun cambio di contesto quando un processo invoca una funzione del sistema operativo o nel caso della gestione di un interrupt hardware. Inoltre il kernel può fare richieste e scambiare dati tra diversi sottosistemi invocando funzioni C relativamente economiche, rispetto alla più complicata comunicazione tra processi (IPC). Non è solo il codice di base relativo allo scheduling e alla gestione della memoria virtuale a occupare l'unico spazio d'indirizzi; *tutto* il codice del kernel, inclusi i driver dei dispositivi, il file system e il codice relativo al networking, si trova nello stesso spazio d'indirizzi.

Il fatto che tutto il codice del kernel sia messo nello stesso calderone non significa che non vi sia spazio per la modularità: così come le applicazioni possono caricare librerie condivise durante l'esecuzione per ottenere le parti di codice aggiuntivo di cui hanno bisogno, il kernel può caricare (o scaricare) dinamicamente moduli durante l'esecuzione. Non è necessario che il kernel sappia in anticipo quali moduli si dovranno caricare: i moduli sono componenti caricabili in modo veramente indipendente.

Il kernel costituisce la parte centrale del sistema operativo Linux: fornisce tutti i servizi necessari per eseguire i processi e gestire l'accesso alle risorse fisiche in modo controllato e protetto. In effetti, fa tutto ciò che un sistema operativo deve essere in grado di fare. Preso isolatamente, però, il sistema operativo fornito dal kernel non è un sistema UNIX completo: mancano molte delle caratteristiche di UNIX e anche quelle fornite non hanno il formato che le applicazioni si aspettano in ambiente UNIX. L'interfaccia del sistema operativo vista dalle applicazioni in esecuzione non fa propriamente parte del kernel: le applicazioni chiamano le librerie di sistema, che si servono a loro volta degli opportuni servizi del kernel.

Le librerie di sistema forniscono molti tipi di funzioni: a livello più semplice permettono alle applicazioni di effettuare chiamate di sistema al kernel. L'esecuzione di una chiamata di sistema richiede il passaggio del controllo dalla modalità utente a quella privilegiata del kernel; i dettagli di questo passaggio variano secondo l'architettura del sistema. Le librerie si occupano di raccogliere gli argomenti della chiamata di sistema e, se è necessario, organizzano gli argomenti nella forma richiesta dall'esecuzione della chiamata di sistema.

Le librerie possono anche fornire versioni più complesse delle chiamate di sistema di base: per esempio, le funzioni del linguaggio C di gestione dei file con buffer (*buffering*) sono tutte implementate nelle librerie di sistema; esse forniscono un più efficace controllo dell'I/O su file rispetto alle chiamate di sistema di base del kernel. Le librerie forniscono anche funzioni che non corrispondono per niente a chiamate di sistema, per esempio algoritmi di ordinamento, funzioni matematiche e procedure per la manipolazione di stringhe di caratteri. Tutte le funzioni necessarie per l'esecuzione di applicazioni UNIX o POSIX sono codificate nelle librerie di sistema.

Il sistema Linux include un'ampia varietà di programmi eseguibili in modalità utente: si tratta di utilità di sistema o di utente. Le utilità di sistema includono tutti i programmi necessari per inizializzare e poi amministrare il sistema, come quelli per configurare le interfacce di rete o per aggiungere e rimuovere utenti dal sistema.

Anche gli strumenti per gli utenti sono necessari per il funzionamento di base del sistema, ma non sono richiesti privilegi elevati per eseguirli. Includono semplici programmi per la gestione dei file, come le utilità per copiare file, creare directory e modificare file di testo. Uno degli strumenti più importanti è la shell, l'interfaccia standard a riga di comando su sistemi UNIX. Linux supporta diverse shell: la più comune è la shell Bourne-Again (bash).

18.3 Moduli del kernel

Il kernel di Linux è in grado di caricare e scaricare su richiesta arbitrarie parti di codice: questi moduli caricabili sono eseguiti in modo privilegiato, e hanno quindi pieno accesso alle risorse fisiche del calcolatore. In teoria non c'è limite a ciò che un modulo del kernel può fare; può, per esempio, implementare un driver di dispositivo, un file system o un protocollo di comunicazione.

I moduli del kernel sono convenienti per diverse ragioni. Il codice sorgente di Linux è gratuito, e quindi chiunque volesse scrivere codice del kernel potrebbe compilare il kernel modificato e riavviare il sistema per caricare la nuova funzione, ma la ricompilazione, il collegamento e il ricaricamento dell'intero kernel è un processo laborioso che si dovrebbe ripetere molte volte durante la stesura di un driver. Usando i moduli del kernel non è necessario costituire un nuovo kernel per provare un nuovo driver: il driver si può compilare separatamente e può essere caricato dal kernel già in uso. Naturalmente, dopo che un nuovo driver è stato scritto, si può distribuire come modulo cosicché anche altri utenti potranno trarne beneficio senza dover ricompilare integralmente il kernel.

Quest'ultimo punto un'altra implicazione: essendo soggetto a licenza GPL, il kernel di Linux non può essere distribuito insieme con componenti proprietari, a meno che anch'essi non siano soggetti a licenza GPL e il loro codice sorgente non sia disponibile su richiesta. L'interfaccia a moduli del kernel permette a terzi di scrivere e distribuire alle loro condizioni driver di dispositivi o file system che non si potrebbero distribuire secondo le norme GPL.

I moduli del kernel permettono a un sistema Linux di essere attivato con un kernel minimale standard privo di driver aggiuntivi incorporati: i driver dei dispositivi necessari all'utente si possono caricare in modo esplicito nella fase d'avviamento del sistema, o possono essere caricati automaticamente dal sistema su richiesta e scaricati dopo l'uso. Un driver per un mouse, per esempio, può essere caricato nel momento in cui si inserisce la presa USB del mouse e scaricato quando la presa viene disinserita.

Gli elementi principali del supporto ai moduli di Linux sono i seguenti.

1. Il sistema di **gestione dei moduli** permette il caricamento dei moduli in memoria e la loro comunicazione con il resto del kernel.
2. Il modulo di caricamento e scaricamento, che è un'utilità in modalità utente, lavora con il sistema di gestione dei moduli e carica moduli in memoria.

3. Il sistema di **registrazione dei driver** permette di comunicare al resto del kernel la disponibilità di un nuovo driver.
4. La **risoluzione dei conflitti** è un meccanismo che permette a un driver di riservarsi l'uso di certe risorse fisiche e di proteggerle da un uso accidentale da parte di un altro driver.

18.3.1 Gestione dei moduli

Il caricamento di un modulo è un processo più complesso del semplice trasferimento in memoria del codice binario che lo costituisce: il sistema deve anche accertarsi del fatto che ogni riferimento fatto dal modulo a simboli o punti d'ingresso del kernel sia aggiornato per puntare alle corrette locazioni di memoria all'interno dello spazio d'indirizzi del kernel. Il sistema Linux affronta questo problema dividendo il caricamento in due parti: la gestione di sezioni di codice del modulo nella memoria del kernel, e quella dei simboli cui i moduli possono fare riferimento.

Il sistema Linux mantiene una tabella interna dei simboli nel kernel: essa non contiene l'intero insieme di simboli definiti durante la compilazione del kernel, ma solo i simboli esplicitamente esportati dal kernel. L'insieme di simboli esportati costituisce una ben definita interfaccia tramite la quale un modulo può interagire con il kernel.

Sebbene l'esportazione di un simbolo da una funzione del kernel presupponga la richiesta esplicita del programmatore, nessuno sforzo aggiuntivo è necessario per importare questi simboli in un modulo; chi scrive il modulo usa semplicemente il meccanismo standard di linking esterno del linguaggio C: tutti i simboli esterni cui il modulo fa riferimento ma che non sono dichiarati dal modulo stesso sono contrassegnati come irrisolti nel codice binario finale prodotto dal compilatore. Quando un modulo deve essere caricato in memoria, una utilità di sistema lo esamina per rilevare questi riferimenti irrisolti: il valore d'ogni simbolo esterno irrisolto viene cercato nella tabella dei simboli del kernel, e vengono sostituiti nel codice del modulo gli indirizzi corretti per quei simboli relativi al kernel attualmente in esecuzione. Solo a questo punto il modulo può essere caricato dal kernel: se l'utilità di sistema non riesce a risolvere un riferimento cercandone l'indirizzo nella tabella dei simboli del kernel, il modulo non è accettato.

Il caricamento di un modulo avviene in due fasi. In primo luogo, l'utilità di caricamento richiede al kernel di riservare per il modulo una regione continua di memoria virtuale di kernel; il kernel restituisce l'indirizzo dell'area di memoria allocata, e il caricatore può usare quest'indirizzo per rilocare il codice macchina del modulo rispetto all'indirizzo di caricamento corretto. Una seconda chiamata di sistema passa al kernel il nuovo modulo, insieme con l'eventuale tabella di simboli esterni che il modulo chiede di esportare: il modulo è copiato direttamente nello spazio di memoria precedentemente riservato, e si aggiorna la tabella dei simboli del kernel in base ai nuovi simboli per un eventuale uso da parte d'altri moduli non ancora caricati.

L'ultimo componente della gestione dei moduli è il meccanismo di richiesta dei moduli. Il kernel definisce un'interfaccia di comunicazione alla quale il programma

di gestione dei moduli si può collegare; grazie a ciò il kernel potrà comunicare al programma di gestione se un processo richiede l'uso del driver di un dispositivo, di un file system o di un servizio di rete i cui moduli non sono ancora stati caricati, dando così al gestore la possibilità di caricare il modulo appropriato; le richieste di servizio saranno soddisfatte dopo il completamento di quest'operazione. Il programma di gestione interroga il kernel a intervalli regolari per appurare quali moduli dinamicamente caricati non sono più in uso, scaricando i moduli che non sono necessari al momento.

18.3.2 Registrazione dei driver

Un modulo che è stato caricato rimane una regione isolata della memoria fino a quando non comunica al kernel quali nuove funzionalità è in grado di fornire. Il kernel mantiene tabelle dinamiche di tutti i driver noti, e fornisce un insieme di procedure che permettono di aggiungere o rimuovere un driver da queste tabelle in qualunque momento. Il kernel assicura l'avviamento della procedura d'inizializzazione d'ogni nuovo modulo caricato, e l'attivazione della procedura di chiusura prima che esso sia scaricato: queste procedure sono responsabili della registrazione delle funzionalità forniti dal modulo.

Un modulo può registrare molti tipi di funzionalità, senza limitarsi a un solo tipo. Un certo driver, per esempio, potrebbe voler registrare due meccanismi distinti per l'accesso a un dispositivo. Le tabelle di registrazione includono, tra gli altri, i seguenti elementi.

- **Driver dei dispositivi.** Questi driver comprendono i dispositivi a caratteri (come stampanti, terminali e mouse), i dispositivi a blocchi (fra i quali tutte le unità a disco), e i dispositivi d'interfacciamento alla rete.
- **File system.** Un file system è una qualsiasi entità che implementa le chiamate di sistema del file system virtuale di Linux: può trattarsi della implementazione di un formato per la scrittura di file nei dischi, oppure di un file system di rete come l'NFS, o ancora di un file system virtuale i cui contenuti sono generati su richiesta, come il file system `/proc`.
- **Protocolli di rete.** Un modulo può essere la implementazione di un intero protocollo di rete, come per esempio TCP, o più semplicemente un insieme di regole per il filtraggio dei pacchetti di dati per un firewall di rete.
- **Formato binario.** Questo formato specifica un modo per riconoscere, caricare ed eseguire un nuovo tipo di file eseguibile.

Inoltre, un modulo può registrare un nuovo insieme di elementi nelle tabelle `systcl` e `/proc`, in modo che lo stesso modulo si possa configurare dinamicamente (Paragrafo 18.7.4).

18.3.3 Risoluzione dei conflitti

Le versioni commerciali del sistema operativo UNIX sono di solito concepite per essere eseguite sui calcolatori di un singolo produttore: un vantaggio di questa situazione è che chi sviluppa i programmi ha un'idea piuttosto precisa delle possibili configurazioni dell'hardware. L'hardware di un PC, d'altro canto, è disponibile in un gran numero di configurazioni diverse, e con molti possibili driver per dispositivi come schede di rete e schede grafiche. Il problema di trattare le differenti configurazioni hardware si aggrava quando si adotta un criterio modulare di gestione dei driver dei dispositivi, perché l'insieme dei dispositivi attivi diviene variabile dinamicamente.

Il sistema Linux fornisce un meccanismo centrale di risoluzione dei conflitti che aiuta a regolare l'accesso a certe risorse hardware. I suoi obiettivi sono i seguenti:

- impedire che moduli diversi entrino in conflitto per l'accesso alle risorse hardware;
- impedire che una **procedura di autoverifica** (*autoprobos*) di un certo driver – cioè una procedura del driver che determina automaticamente la configurazione del dispositivo – interferisca con driver già presenti;
- risolvere i conflitti che possono nascere fra diversi driver che tentino di accedere alle stesse risorse fisiche; per esempio, sia il driver per la porta parallela di una stampante sia il driver di rete IP relativo a una porta parallela (*parallel-line IP*, PLIP) potrebbero cercare di comunicare con la porta parallela.

Per raggiungere questi scopi il kernel mantiene una lista delle risorse hardware assegnate. Il PC ha un numero limitato di possibili porte di I/O (indirizzi nel suo spazio d'indirizzi per i dispositivi di I/O), linee per le interruzioni e canali DMA; da un driver di un dispositivo che voglia accedere a una di queste risorse ci si aspetta che prima di tutto prenoti la risorsa presso il database del kernel. Questo prerequisito, fra l'altro, permette all'amministratore del sistema di determinare esattamente quali risorse siano state riservate da quale driver a ogni dato istante.

Ci si attende che un modulo usi questo meccanismo per prenotare ogni risorsa fisica che intenda impiegare. Se la richiesta di prenotazione non è accettata, per esempio perché la risorsa non esiste o è già in uso, sta al modulo decidere quale tipo d'azione intraprendere: potrebbe dichiarare fallita la sua procedura d'inizializzazione e richiedere di essere scaricato, oppure continuare tentando di usare risorse hardware alternative.

18.4 Gestione dei processi

Un processo è il contesto fondamentale all'interno del quale tutte le attività richieste dagli utenti sono servite dal sistema operativo. Per essere compatibile con altri sistemi UNIX, il sistema operativo Linux deve adottare un modello dei processi simile a quello di altre versioni del sistema UNIX: tuttavia, Linux si comporta in modo diverso in alcuni punti chiave. In questo paragrafo si richiama il tradizionale modello UNIX dei processi e si presenta il modello di threading di Linux.

18.4.1 Modello `fork()` ed `exec()` dei processi

Il principio fondamentale della gestione dei processi in UNIX è di separare due operazioni che sono solitamente combinate: la creazione di un nuovo processo e l'esecuzione di un nuovo programma. La chiamata di sistema `fork()` assolve il primo compito, mentre l'esecuzione di un nuovo programma è avviata dalla chiamata di sistema `exec()`: queste due funzioni sono nettamente differenti: si può creare un nuovo processo tramite la `fork()` senza avviare l'esecuzione di un nuovo programma – il processo figlio continua a eseguire esattamente lo stesso programma, a partire dalla stessa posizione, del processo genitore. Similmente, l'esecuzione di un nuovo programma non richiede la creazione di un nuovo processo: un qualunque processo può eseguire la chiamata di sistema `exec()` in qualunque momento, nel qual caso un nuovo oggetto binario viene caricato nello spazio degli indirizzi del processo e il nuovo programma è eseguito nel contesto del processo esistente.

Il vantaggio di questo modello è la sua gran semplicità: non si deve specificare ogni dettaglio riguardante l'ambiente di un nuovo programma tramite la chiamata di sistema che ne avvia l'esecuzione: i nuovi programmi sono semplicemente eseguiti nell'ambiente esistente. Se un processo genitore desidera modificare l'ambiente d'esecuzione di un nuovo programma, può eseguire una `fork()` e poi, continuando a eseguire il programma originale nell'ambito del processo figlio, eseguire tutte le chiamate di sistema necessarie per modificare quel processo figlio prima di avviare infine il nuovo programma.

Nel sistema UNIX, quindi, un processo racchiude tutte le informazioni di cui il sistema operativo necessita per gestire il contesto di una singola esecuzione di un singolo programma; in Linux, questo contesto può essere suddiviso in un certo numero di sezioni specifiche. A grandi linee, le proprietà di un processo si dividono in tre gruppi: identità, ambiente e contesto del processo.

18.4.1.1 Identità dei processi

L'identità di un processo consiste principalmente dei seguenti elementi.

- **Identificatore del processo** (*process identifier, PID*). A ogni processo è associato un identificatore unico. Il PID si usa per specificare un certo processo al sistema operativo quando un'applicazione invoca una chiamata di sistema riferita a quel processo. Ulteriori identificatori associano un processo a un gruppo di processi (tipicamente un albero di processi generati da `fork()` a partire da un unico comando utente) e a una sessione di login.
- **Credenziali.** Ogni processo deve avere associati un identificatore utente e uno o più identificatori di gruppi di utenti che determinano i diritti d'accesso alle risorse del sistema e ai file (i gruppi di utenti sono discussi nel Paragrafo 11.6.2).
- **Personalità.** Il concetto di personalità di un processo non fa parte della tradizione di UNIX, ma nel sistema Linux ogni processo ha un identificatore di personalità associato che può modificare lievemente la semantica di certe chiamate di sistema: le personalità sono principalmente usate da librerie di emulazione per ottenere la compatibilità di certe chiamate di sistema con certe specifiche versioni di UNIX.

- Spazio dei nomi. Ogni processo è associato a una visione specifica della gerarchia del file system, chiamata il suo **spazio dei nomi** (namespace). La maggior parte dei processi condivide un namespace comune e opera quindi su una gerarchia di file system condivisa. I processi e i loro figli possono tuttavia avere namespace distinti, ognuno con un'unica gerarchia del file system, con la propria root directory e con il proprio insieme di file system montati.

Un processo ha un controllo limitato dei suoi stessi identificatori: gli identificatori del gruppo e della sessione di login possono essere cambiati se il processo desidera avviare un nuovo gruppo o una nuova sessione; le credenziali possono essere cambiate subordinatamente a certi controlli di sicurezza; il PID di un processo, invece, non è modificabile, e lo identifica in modo univoco fino alla sua terminazione.

18.4.1.2 Ambiente di un processo

L'ambiente di un processo è ereditato dal suo genitore, ed è composto di due vettori: il vettore degli argomenti e il vettore d'ambiente. Il **vettore degli argomenti** elenca semplicemente gli argomenti della riga di comando usata per invocare il programma in esecuzione, e comincia per convenzione con il nome del programma stesso. Il **vettore d'ambiente** è una lista di coppie della forma “NOME=VALORE” che associano ai nomi delle variabili d'ambiente valori testuali arbitrari. La descrizione dell'ambiente non è contenuta in memoria del kernel ma nello spazio d'indirizzi in modalità utente del processo, ed è il primo dato sullo stack del processo.

Questi due vettori non sono alterati a seguito della creazione di un nuovo processo: il processo figlio eredita l'ambiente del processo genitore. L'avvio di un nuovo programma, invece, implica la costituzione di un ambiente interamente nuovo: un processo che esegue la chiamata `exec()` deve anche specificare l'ambiente del nuovo programma tramite variabili d'ambiente che sono passate dal kernel al nuovo programma, e sostituiscono l'ambiente attuale del processo. A parte questa circostanza, i vettori d'ambiente e degli argomenti sono ignorati dal kernel; la loro interpretazione è lasciata interamente alle librerie in modalità utente e alle applicazioni.

Il passaggio delle variabili d'ambiente da un processo all'altro e il meccanismo di ereditarietà di queste variabili costituiscono un modo flessibile di scambiare informazioni ai componenti del sistema eseguiti in modalità utente. Alcune importanti variabili d'ambiente hanno un significato convenzionale correlato a certe parti del sistema: i valori della variabile `TERM`, per esempio, denotano il tipo di terminale usato durante una sessione di login, e molti programmi sfruttano questo fatto per stabilire come eseguire certe operazioni che hanno effetti sullo schermo dell'utente (spostamento del cursore, scorrimento di parti di testo, e così via). I programmi in grado di supportare più lingue usano la variabile `LANG` per stabilire in quale lingua mostrare i messaggi.

Il meccanismo delle variabili d'ambiente personalizza il sistema operativo, processo per processo: ogni utente può scegliere una lingua o un editor particolare indipendentemente dalle scelte altri.

18.4.1.3 Contesto di un processo

L'identità di un processo e le proprietà dell'ambiente sono di solito stabilite al momento della creazione di un processo, e non mutano fino alla sua terminazione, anche se un processo potrebbe decidere di cambiare il proprio ambiente o alcuni aspetti della propria identità. Il contesto di un processo, invece, è lo stato del programma in esecuzione in ogni istante: esso cambia continuamente. Il contesto di un processo include le parti seguenti.

- **Contesto di scheduling.** La parte più importante del contesto di un processo è il suo contesto di scheduling: le informazioni di cui lo scheduler necessita per sospendere o riavviare il processo, comprese le copie di tutti i suoi registri. I registri in virgola mobile sono memorizzati separatamente, e ripristinati solo quando è necessario, per rendere più efficiente il salvataggio di stato dei processi che non usano l'aritmetica in virgola mobile. Il contesto di scheduling comprende anche informazioni sulla priorità di scheduling e su ogni segnale che attende d'essere recapitato al processo. Una parte cruciale del contesto di scheduling è lo stack di kernel del processo: un'area di memoria nello spazio d'indirizzi del kernel riservata esclusivamente all'uso di codice eseguito in modalità kernel; sia le chiamate di sistema sia la gestione delle interruzioni occorrenti durante l'esecuzione del processo usano questo stack.
- **Accounting.** Il kernel mantiene informazioni sulle risorse correntemente impiegate da ciascun processo e anche sulla quantità totale di risorse impiegate durante la sua intera esistenza.
- **Tabella dei file.** È un array di puntatori alle strutture dati del kernel relative ai file aperti. Quando un processo esegue una chiamata di sistema di I/O relativa a un file, lo identifica tramite un intero, il suo descrittore di file (fd), che il kernel utilizza come indice in questa tabella.
- **Contesto del file-system.** Mentre la tabella dei file elenca i file aperti esistenti, il contesto del file system viene utilizzato nelle richieste di apertura di nuovi file. Il contesto del file system include la root directory del processo, la directory di lavoro corrente e lo spazio dei nomi.
- **Tabella dei gestori dei segnali.** I sistemi UNIX possono recapitare a un processo segnali asincroni in risposta a un evento esterno. Questa tabella definisce quali azioni si devono intraprendere quando si riceve un certo segnale. Tra le possibili azioni vi sono: ignorare il segnale, terminare il processo, invocare una procedura nello spazio degli indirizzi del processo.
- **Contesto della memoria virtuale.** Il contesto della memoria virtuale descrive completamente i contenuti dello spazio d'indirizzi di un processo, ed è trattato nel Paragrafo 18.6.

18.4.2 Processi e thread

Linux possiede la chiamata di sistema `fork()` per duplicare un processo senza caricare una nuova immagine eseguibile; esso offre anche un'altra opportunità per gene-

rare i thread, ovvero la chiamata di sistema `clone()`. Linux, però, non distingue tra processi e thread, impiegando generalmente il termine *task* – invece di *processo* o *thread* – in riferimento a un flusso di controllo nell’ambito di un programma. La chiamata `clone()` si comporta come la `fork()`, a eccezione del fatto che accetta come parametro un insieme di flag che stabiliscono quante e quali risorse del processo genitore debbano essere condivise dal processo figlio, mentre un processo creato mediante `fork()` non condivide risorse con il genitore. Alcuni di questi flag sono mostrati dallo schema seguente.

flag	significato
<code>CLONE_FS</code>	Le informazioni sul file system sono condivise
<code>CLONE_VM</code>	Condivisione dello stesso spazio di memoria
<code>CLONE_SIGHAND</code>	Condivisione dei gestori dei segnali
<code>CLONE_FILES</code>	Condivisione dell’insieme dei file aperti

Per esempio, qualora `clone()` riceva i flag `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, il processo genitore e il processo figlio condivideranno le medesime informazioni sul file system (come la directory di lavoro corrente), lo stesso spazio di memoria, gli stessi gestori dei segnali e lo stesso insieme di file aperti. Utilizzare `clone()` in questo modo equivale alla creazione di un thread in altri sistemi, dal momento che il processo genitore condivide la maggior parte delle proprie risorse con il processo figlio. Se invece nessuno dei flag è impostato, non si ha alcuna condivisione, e la funzionalità risultante è simile a quella fornita dalla chiamata di sistema `fork()`.

La mancanza di distinzione tra processi e thread è possibile perché Linux non memorizza l’intero contesto di un processo nella struttura dati principale del processo: esso è infatti distribuito in sottocontesti indipendenti. Pertanto, il contesto del file system di un processo, la tabella dei descrittori dei file, la tabella del gestore dei segnali e il contesto della memoria virtuale risiedono in strutture dati distinte. La struttura dati di un processo contiene semplicemente un puntatore per ogni struttura sopra citata. In tal modo, un qualsiasi numero di processi può facilmente condividere un sottocontesto impostando il puntatore e incrementando un contatore di riferimenti al sottocontesto.

La chiamata di sistema `clone()` riceve argomenti che specificano quali sottocontesti copiare e quali condividere. Al nuovo processo è sempre attribuita una nuova identità e un nuovo contesto di scheduling (ciò è essenziale per un processo Linux); a seconda degli argomenti che riceve, tuttavia, il kernel può sia creare nuove strutture dati del sottocontesto, inizializzate come copie del processo genitore, sia stabilire che il nuovo processo utilizzi le stesse strutture dati del sottocontesto, usate dal processo genitore. La chiamata di sistema `fork()`, che copia tutti i sottocontesti senza condividerne alcuno, non è altro che un caso particolare di `clone()`.

18.5 Scheduling

Lo scheduling consiste nell'allocazione del tempo di CPU ai diversi task all'interno di un sistema operativo. Linux, come tutti i sistemi UNIX, supporta il multitasking con prelazione. In un tale sistema lo scheduler decide quale processo viene eseguito e quando. Prendere queste decisioni in modo da bilanciare equità e prestazioni sotto carichi di lavoro molto diversi è una delle sfide più complicate dei moderni sistemi operativi.

Di norma si pensa allo scheduling come all'esecuzione e sospensione dei processi utente, ma un altro aspetto dello scheduling importante per il sistema Linux è l'esecuzione dei vari task del kernel, che include sia i task richiesti da un processo in esecuzione, sia i task interni eseguiti per conto del kernel stesso, come i task generati dal sottosistema di I/O di Linux.

18.5.1 Scheduling dei processi

Il sistema Linux adotta due distinti algoritmi di scheduling: uno è un algoritmo time sharing con prelazione per l'equa condivisione del tempo di CPU da parte di più processi, e l'altro è concepito per elaborazioni in tempo reale dove le priorità assolute sono più importanti dell'equità.

L'algoritmo di scheduling adoperato per i task in time sharing ha beneficiato di forti innovazioni con il passaggio alla versione 2.6 del kernel. In precedenza, il kernel di Linux impiegava una variante del tradizionale algoritmo di scheduling di UNIX. Questo algoritmo non offre adeguato supporto ai sistemi SMP, non scala bene al crescere del numero di task nel sistema e non gestisce equamente i task interattivi, in particolari su sistemi desktop e mobili. Lo scheduler dei processi è stato revisionato con la versione 2.5 del kernel, con l'implementazione di un algoritmo di scheduling che seleziona il task da eseguire in tempo costante, ossia O(1), indipendentemente dal numero di task e processori nel sistema. Il nuovo scheduler fornisce un maggiore supporto per SMP, includendo la predilezione del processore e il bilanciamento del carico. Questi cambiamenti, pur migliorando la scalabilità, non hanno migliorato le prestazioni per quanto riguarda i processi interattivi e l'equità, anzi, di fatto, hanno reso più problematiche certe situazioni sotto particolari carichi di lavoro. Lo scheduler dei processi è stato dunque revisionato per una seconda volta e con il kernel Linux versione 2.6 è stato inaugurato il Completely Fair Scheduler (CFS).

Lo scheduler di Linux è un algoritmo con prelazione basato su priorità, con due intervalli distinti di priorità: un intervallo **real-time**, con valori da 0 a 99, e un **nice value** con valori da -20 a 19. Valori di nice value minori indicano priorità più alta. Aumentando il nice value, si sta diminuendo la priorità e ci si sta comportando in modo "nice" (gentile) verso il resto del sistema.

CFS costituisce un allontanamento significativo dal tradizionale scheduler di UNIX, in cui le variabili fondamentali dell'algoritmo di scheduling sono priorità e porzioni di tempo (time slice). Il time slice è il tempo del processore che viene assegnato a un processo. I sistemi UNIX tradizionali danno ai processi un intervallo di tempo fissato,

talvolta con una aggiunta o una penalità rispettivamente per i processi a priorità più alta e più bassa. Un processo può restare in esecuzione per tutta la durata della sua porzione di tempo e i processi con priorità maggiore vengono eseguiti prima dei processi a bassa priorità. Si tratta di un semplice algoritmo utilizzato da molti sistemi diversi da UNIX. Questo meccanismo molto semplice ha funzionato bene per i primi sistemi time-sharing, ma si è dimostrato incapace di fornire buone prestazioni in termini di equità e gestione di processi interattivi nei moderni sistemi desktop e nei dispositivi mobili.

CFS ha introdotto un nuovo algoritmo di scheduling, chiamato fair scheduling, che elimina i time slice intesi in senso tradizionale. Al posto delle porzioni di tempo, a tutti i processi viene assegnata una percentuale del tempo del processore. CFS calcola per quanto tempo un processo deve essere eseguito in funzione del numero totale di processi eseguibili. Per iniziare, CFS stabilisce che se ci sono N processi eseguibili a ognuno deve essere accordata una frazione $1/N$ del tempo del processore. A questo punto CFS calibra questa ripartizione ponderando l'assegnazione a ogni processo a seconda del suo nice value. Ai processi con nice value di default viene dato un peso pari a 1 (la loro priorità resta invariata), ai processi con un nice value minore (priorità superiore) viene dato un peso maggiore, mentre i processi con un nice value maggiore (priorità inferiore) ricevono un peso inferiore. CFS esegue poi ogni processo per un intervallo di tempo proporzionale al peso del processo diviso per il peso totale di tutti i processi eseguibili.

Per calcolare il periodo di tempo per cui un processo viene effettivamente eseguito, CFS si basa su una variabile configurabile chiamata **latenza obiettivo** (*target latency*) che rappresenta l'intervallo di tempo nel quale ogni task eseguibile dovrebbe andare in esecuzione almeno una volta. Si supponga, per esempio, che la latenza obiettivo sia di 10 millisecondi e si supponga di avere due processi eseguibili con la stessa priorità. Questi processi hanno lo stesso peso e quindi ricevono una uguale percentuale del tempo del processore. In questo caso, con una latenza obiettivo di 10 millisecondi, il primo processo viene eseguito per 5 millisecondi, quindi l'altro processo viene eseguito per 5 millisecondi, quindi il primo processo viene eseguito di nuovo per 5 millisecondi, e così via. Se avessimo 10 processi eseguibili, CFS manderebbe in esecuzione ogni processo per un millesimo di secondo prima di ripetere il ciclo.

Che cosa succederebbe se avessimo, per esempio, 10.000 processi? Secondo il procedimento descritto ogni processo resterebbe in esecuzione per 1 microsecondo e, a causa dei costi per lo switch di contesto, la pianificazione per una durata così breve risulterebbe inefficiente. Per risolvere questa questione CFS si basa su una seconda variabile configurabile, la granularità minima, che rappresenta il periodo minimo di tempo del processore che può essere assegnato a ogni processo. Tutti i processi, indipendentemente dalla latenza obiettivo, resteranno in esecuzione per un tempo pari almeno alla granularità minima. In questo modo CFS assicura che i costi dell'avvicendamento dei processi non crescano in maniera inaccettabile quando il numero di processi eseguibili diventa troppo grande, anche se, nel fare ciò, viola i suoi vincoli di equità. Visto che, tipicamente, il numero di processi eseguibili rimane ragionevole, sia i costi di avvicendamento che l'equità ne risultano massimizzati.

Con il passaggio al fair scheduling, il comportamento di CFS si differenzia da quello dello scheduler tradizionale UNIX sotto diversi aspetti. In particolare, come abbiamo visto, CFS elimina il concetto di time slice fisso. I processi ricevono invece una percentuale del tempo del processore e l'ammontare di questo tempo dipende da quanti altri processi eseguibili ci sono. Questo approccio risolve diversi problemi relativi all'assegnazione delle priorità negli algoritmi di scheduling con prelazione basati sulle priorità. È possibile, naturalmente, risolvere questi problemi in altri modi senza abbandonare il tradizionale scheduler UNIX. Tuttavia CFS è in grado di risolvere i problemi con un semplice algoritmo che si comporta bene su sistemi interattivi come i telefoni cellulari senza compromettere le prestazioni sui server di grandi dimensioni.

18.5.2 Scheduling real-time

L'algoritmo di scheduling real-time di Linux è molto più semplice rispetto all'algoritmo fair scheduling utilizzato per i processi time-sharing standard. Linux implementa le due classi di scheduling in tempo reale richieste da POSIX.1b: FCFS e RR (Paragrafo 6.3.1 e Paragrafo 6.3.4, rispettivamente). In entrambi i casi, ogni processo ha una priorità in aggiunta alla sua classe di scheduling. Lo scheduler esegue sempre il processo con la priorità più alta, oppure, a parità di priorità, il processo che attende da più tempo. L'unica differenza fra lo scheduling FCFS e quello RR è che un processo FCFS continua l'esecuzione fino alla terminazione o finché non si blocca, mentre un processo RR può essere sospeso allo scadere del suo quanto di tempo, nel qual caso è posto alla fine della coda di scheduling; la conseguenza di ciò è che fra i processi RR della stessa priorità si effettua il time sharing.

Lo scheduling real-time di Linux è in tempo reale debole (*soft*), non in tempo reale stretto (*hard*); lo scheduler offre rigide garanzie sulle priorità relative dei processi in tempo reale, ma il kernel non fornisce alcuna garanzia sulla rapidità con cui un processo in tempo reale pronto per l'esecuzione sarà effettivamente eseguito. Nei sistemi in tempo reale hard, invece, viene garantito un ritardo massimo entro il quale un processo pronto per l'esecuzione deve essere eseguito.

18.5.3 Sincronizzazione del kernel

Il modo in cui il kernel esegue lo scheduling delle sue stesse operazioni è essenzialmente diverso dal modo in cui esegue lo scheduling dei processi. La richiesta d'esecuzione di codice in modalità kernel può avvenire in due modi: un programma in esecuzione può richiedere un servizio del sistema operativo esplicitamente, tramite una chiamata di sistema, o implicitamente, per esempio nel caso di un'eccezione di pagina mancante; oppure, il driver di un dispositivo può generare un segnale d'interruzione che induce l'esecuzione di un gestore di tale interruzione presente nel kernel.

Il problema che il kernel si trova a dover affrontare è che questi task potrebbero tentare di accedere alle stesse strutture dati interne: se una procedura di gestione di un segnale d'interruzione interrompe un task del kernel che sta usando certe strutture dati, la procedura in questione non può accedere a quelle stesse strutture dati se non a rischio di creare incoerenze. Quest'osservazione è connessa all'idea delle sezioni

critiche, cioè parti di codice che condividono certe strutture dati e che non si possono eseguire in modo concorrente. Quindi la sincronizzazione del kernel richiede più del semplice scheduling dei processi: è necessario un meccanismo che permette l'esecuzione dei task del kernel senza rischiare i violare l'integrità dei dati condivisi.

Prima della versione 2.6, Linux adoperava un kernel senza prelazione: non era possibile applicare la prelazione ai processi eseguiti in modalità kernel, neppure nel caso che un processo con priorità più alta fosse pronto per l'esecuzione. Con il passaggio alla versione 2.6, il kernel ha adottato la prelazione: ogni task attivo nel kernel può ora essere sottoposto a prelazione.

Il kernel di Linux si serve di semafori convenzionali e spinlock (nonché della variante lettore-scrittore di questi due meccanismi) per implementare il locking nel kernel. Sulle macchine SMP, il lock fondamentale è lo spinlock; il kernel è realizzato in modo da applicare gli spinlock solo per brevi periodi di tempo. Gli spinlock sono inadatti alle macchine monoprocesso, per cui si abilita e disabilita il diritto di prelazione del kernel. Su tali macchine, in pratica, anziché applicare uno spinlock, il kernel disabilita la prelazione, e anziché rimuovere lo spinlock, esso abilita la prelazione. Lo schema è il seguente.

monoprocesso	multiprocessore
disabilita la prelazione del kernel	applica spinlock
abilita la prelazione del kernel	rimuove spinlock

Il modo impiegato da Linux per attivare e disattivare il diritto di prelazione nel kernel è di particolare interesse; si basa su due semplici interfacce, `preempt_disable()` e `preempt_enable()`. Inoltre, se un task attivo in modalità kernel possiede un lock non è possibile ricorrere alla prelazione nel kernel. Per implementare questa regola, ogni task del sistema possiede una struttura `thread-info`, il cui campo `preempt_count` indica il numero dei lock posseduti dal task. Quando il task applica un lock, `preempt_count` è incrementato; quando ne rimuove uno, il contatore è decrementato. Se per il task in esecuzione `preempt_count` ha valore strettamente maggiore di zero, la prelazione a livello del kernel non è sicura, perché il task sta usando un lock. Se il valore è uguale a zero, invece, il kernel può essere interrotto, purché non vi siano chiamate in corso a `preempt_disable()`.

Gli spinlock e il meccanismo di abilitazione e inibizione della prelazione sono usati dal kernel solo quando il lock deve essere applicato per un breve periodo. Quando vi sia necessità di tenere in funzione più a lungo un lock, si usano i semafori.

La seconda tecnica di protezione che Linux usa si applica alle sezioni critiche incluse nelle procedure di gestione degli interrupt. Lo strumento fondamentale in questo caso è l'hardware di controllo delle interruzioni della CPU: disabilitando le interruzioni durante l'esecuzione di una sezione critica, il kernel si assicura di poter procedere senza rischiare accessi concorrenti a strutture dati condivise.

Disabilitare le interruzioni comporta tuttavia una penalizzazione: le istruzioni di abilitazione e disabilitazione delle interruzioni sono onerose nella maggior parte delle

architetture hardware; inoltre ogni operazione di I/O è sospesa finché le interruzioni rimangono disabilitate, e tutti i dispositivi che attendono di essere serviti dovranno aspettare la riabilitazione delle interruzioni, cosa che ha effetti negativi sulle prestazioni. Per risolvere il problema il kernel di Linux usa un'architettura di sincronizzazione che permette l'esecuzione completa di lunghe regioni critiche senza richiedere la disabilitazione delle interruzioni. Questa possibilità torna particolarmente utile per il codice di rete: un interrupt relativo al driver di un dispositivo di rete può segnalare l'arrivo di un intero pacchetto di dati, e quindi implicare l'esecuzione di una gran quantità di codice per decodificare, instradare e inoltrare il pacchetto nell'ambito della procedura di gestione dell'interruzione.

Il sistema Linux realizza in pratica quest'architettura dividendo le procedure di gestione delle interruzioni in due parti: la parte superiore e quella inferiore. La **parte superiore** è un'ordinaria procedura di gestione delle interruzioni, ed è eseguita solo dopo la disabilitazione condizionale delle interruzioni (segnali d'interruzione dello stesso tipo o sulla stessa linea sono disabilitati, mentre altri possono essere serviti). La **parte inferiore** della procedura di gestione, invece, è eseguita, con tutte le interruzioni abilitate, da un minischeduler in grado di assicurare che le parti inferiori non s'interrompano mai fra loro; il minischeduler è automaticamente chiamato ogni volta che termina una procedura di gestione di interrupt.

Questa divisione permette al kernel di completare l'esecuzione di complesse elaborazioni come risposta a un'interruzione senza che il kernel debba preoccuparsi di essere interrotto. Se un'interruzione si verifica durante l'esecuzione di una parte inferiore, anche se l'interruzione può richiedere l'esecuzione della stessa parte inferiore, essa sarà differita fino alla terminazione di quella correntemente in esecuzione. L'esecuzione di una parte inferiore può essere interrotta da una parte superiore, ma non può mai essere interrotta da una parte inferiore simile.

L'architettura a parti è completata da un meccanismo di disabilitazione selettiva delle parti inferiori durante l'esecuzione di normale codice di kernel. Usando questo sistema il kernel può codificare facilmente le sezioni critiche. Gli interrupt handler pongono le proprie sezioni critiche nelle parti inferiori, e quando il codice di kernel ordinario vuole eseguire una sua sezione critica, può disabilitare ogni parte inferiore rilevante per evitare di essere interrotto da altre sezioni critiche. Alla fine della sezione critica il kernel riabilita le parti inferiori ed esegue quelle che, nel corso della sezione critica, sono state accodate a causa dell'esecuzione delle rispettive parti superiori.

Nella Figura 18.2 sono riassunti i vari livelli di protezione delle interruzioni all'interno del kernel; ogni livello può essere interrotto da codice eseguito a un livello

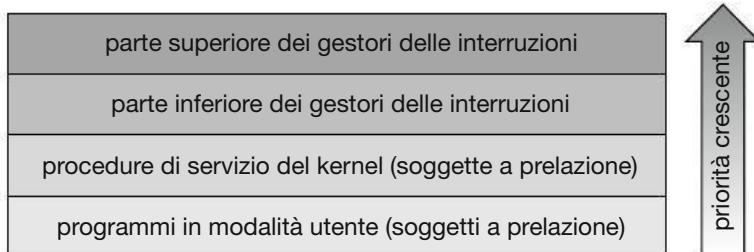


Figura 18.2 Livelli di protezione delle interruzioni.

superiore, ma non sarà mai interrotto da codice eseguito allo stesso livello o a livelli inferiori. Fa eccezione il codice utente: un processo utente può sempre essere sospeso da un altro processo utente a causa dello scadere del quanto di tempo.

18.5.4 Multielaborazione simmetrica

La versione del *Kernel 2.0* fu il primo kernel stabile di Linux in grado di gestire hardware di **multiprocessing simmetrico (SMP)**: processi distinti si possono eseguire in parallelo su unità d'elaborazione distinte; la prima realizzazione dell'SMP in Linux stabiliva che una sola unità d'elaborazione alla volta potesse eseguire codice in modalità kernel.

Nella versione 2.2 del kernel venne introdotto un solo spinlock (talvolta denominato **BKL**, da *big kernel lock*), che permetteva a più processi, eseguiti da processori diversi, di essere attivi concorrentemente nel kernel. Il BKL, però, non permetteva che un controllo poco granulare della mutua esclusione, con la conseguenza di una scarsa scalabilità su macchine con molti processori e processi. Nelle ultime versioni del kernel si è resa più scalabile l'implementazione della SMP, sostituendo l'unico spinlock del kernel con lock multipli, ognuno dei quali protegge solo una piccola parte delle strutture dati del kernel. Gli spinlock in questione sono descritti nel Paragrafo 18.5.3. Il kernel 3.0 ha apportato ulteriori migliorie alla SMP, tra cui meccanismi di lock più fini, l'affinità per il processore e gli algoritmi di bilanciamento del carico.

18.6 Gestione della memoria

Il sistema di gestione della memoria del sistema Linux ha due componenti: il primo si occupa dell'allocazione e del rilascio della memoria fisica: pagine, gruppi di pagine e piccoli blocchi di RAM; il secondo si occupa della gestione della memoria virtuale, la memoria indirizzabile dai processi in esecuzione. In questo paragrafo si descrivono entrambi questi meccanismi, e si esamina poi il modo in cui i componenti caricabili di un nuovo programma sono portati all'interno della memoria virtuale di un processo in conseguenza di una chiamata di sistema `exec()`.

18.6.1 Gestione della memoria fisica

A causa di specifiche caratteristiche dell'hardware, Linux divide la memoria fisica in quattro **zone**, che rappresentano altrettante regioni della memoria. Le zone sono note come:

- `ZONE_DMA`
- `ZONE_DMA32`
- `ZONE_NORMAL`
- `ZONE_HIGHMEM`

Queste zone sono specifiche dell'architettura. Per esempio, sull'architettura Intel 80x86 alcuni dispositivi ISA (*industry standard architecture*) possono accedere soltanto ai 16 MB inferiori della memoria fisica, tramite DMA. Su tali sistemi, i primi 16 MB della memoria fisica comprendono ZONE_DMA. Su altri sistemi, alcuni dispositivi possono accedere solo ai primi 4 GB di memoria fisica, nonostante il supporto a indirizzi di 64 bit. In tali sistemi, i primi 4 GB di memoria fisica comprendono ZONE_DMA32.

ZONE_HIGHMEM (che sta per “memoria alta”) si riferisce alla memoria fisica che non è associata allo spazio degli indirizzi del kernel. Nell'architettura Intel a 32 bit (con uno spazio degli indirizzi, quindi, da 4 GB = 2^{32} bit), il kernel è mappato nei primi 896 MB dello spazio degli indirizzi; la memoria restante, che assume appunto il nome di **memoria alta**, è allocata da ZONE_HIGHMEM. Infine, ZONE_NORMAL comprende tutto il resto (le normali pagine, regolarmente mappate). La presenza di una data zona in un'architettura dipende dai suoi vincoli. Una moderna architettura a 64 bit come Intel x86-64 ha una piccola ZONE_DMA di 16 MB (per i dispositivi legacy) e tutto il resto della memoria è in ZONE_NORMAL, senza “memoria alta”.

La relazione tra zone e indirizzi fisici sull'architettura Intel x86-32 è mostrata dalla Figura 18.3. Il kernel mantiene una lista delle pagine libere per ciascuna zona, e soddisfa la richiesta di memoria fisica usando la zona appropriata.

Il principale strumento di gestione della memoria fisica nel kernel di Linux è l'**allocatore delle pagine**: ciascuna zona ha un suo proprio allocatore, responsabile di allocazione e rilascio delle pagine fisiche, ed è in grado di allocare su richiesta gruppi di pagine fisicamente contigue. L'allocatore usa un **sistema buddy** (Paragrafo 9.8.1) per tener traccia delle pagine fisiche disponibili; un allocatore di questo tipo associa coppie di unità adiacenti di memoria assegnabile: ogni regione di memoria assegnabile ha una gemella adiacente (*buddy*, appunto), e ogni volta che due regioni gemelle allocate si liberano entrambe, vengono combinate per formare una regione più ampia, un *buddy-heap*. Anche questa più ampia regione ha una gemella con la quale potenzialmente potrà formare una regione allocabile ancora più ampia. D'altro canto, se una richiesta di una piccola regione di memoria non può essere soddisfatta assegnando una regione libera esistente, una regione libera più ampia sarà suddivisa in due regioni gemelle al fine di soddisfare la richiesta. Per tener traccia delle regioni di memoria libere di ogni specifica dimensione permessa si usano distinte liste concatenate; nel sistema Linux la più piccola dimensione assegnabile tramite questo meccanismo è di 4 KB.

zona	memoria fisica
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 ... 896 MB
ZONE_HIGHMEM	> 896 MB

Figura 18.3 Relazione fra le zone e gli indirizzi fisici nell'architettura Intel x86-32.

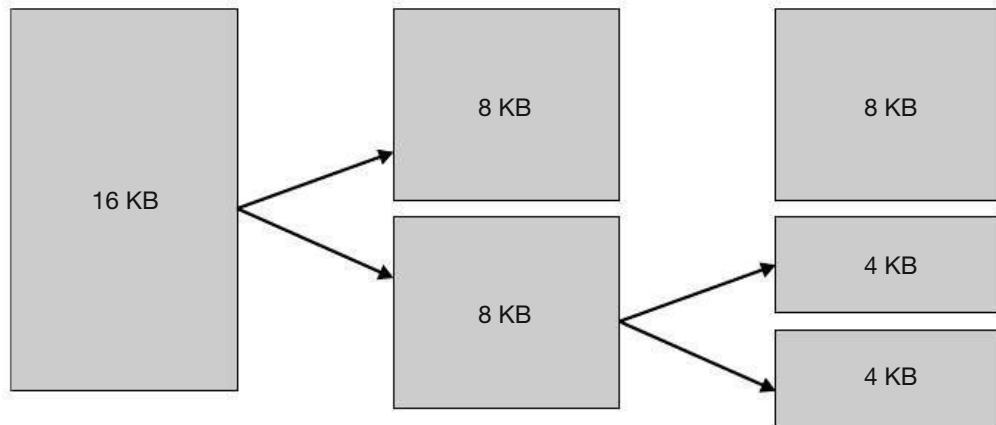


Figura 18.4 Divisione delle regioni di memoria secondo l'algoritmo *buddy-heap*.

canismo è di una singola pagina fisica. Nella Figura 18.4 è mostrato un esempio di allocazione *buddy-heap*: si vuole allocare una regione di 4 KB, ma la più piccola regione disponibile è di 16 KB. La regione è allora ricorsivamente divisa fino a ottenere la dimensione richiesta.

In ultima analisi tutte le operazioni di allocazione di memoria nel kernel di Linux avvengono o staticamente, da driver che riservano aree di memoria contigua durante l'avvio del sistema, o dinamicamente tramite l'allocatore delle pagine; tuttavia le funzioni del kernel non devono necessariamente usare direttamente l'allocatore per riservare memoria: esistono molti sottosistemi specializzati di gestione della memoria che si appoggiano sull'allocatore delle pagine per gestire le loro proprie risorse di memoria. I più importanti sono quello della memoria virtuale, descritto nel Paragrafo 18.6.2, l'allocatore a lunghezza variabile `kmalloc()`, l'allocatore delle lastre, che alloca la memoria necessaria per le strutture dati del kernel, e la cache delle pagine, che funge da cache per le pagine appartenenti ai file.

Molti componenti del sistema operativo Linux hanno bisogno di allocare su richiesta intere pagine, ma spesso c'è anche la necessità di blocchi di memoria più piccoli: il kernel fornisce un allocatore aggiuntivo per le richieste di dimensione arbitraria, in cui la dimensione di una richiesta non è nota in anticipo e potrebbe anche essere solo di qualche byte. Questo servizio è offerto dalla funzione `kmalloc()`, analoga alla `malloc()` del linguaggio C, la quale assegna su richiesta intere pagine fisiche, ma le divide poi in blocchi più piccoli. Il kernel mantiene diverse liste delle pagine usate dalla funzione `kmalloc()`, e tutte le pagine su una data lista vengono suddivise in blocchi di una fissata dimensione. L'allocazione della memoria implica in questo caso l'individuazione della lista appropriata e il successivo uso del primo elemento libero nella lista, o l'allocazione di una nuova pagina e la sua suddivisione. Le regioni di memoria richieste attraverso `kmalloc()` sono allocate in modo permanente, finché non sono esplicitamente liberate con una chiamata `kfree()`; per sopperire a carenze di memoria `kmalloc()` non può riallocare queste regioni, né revocarne l'uso.

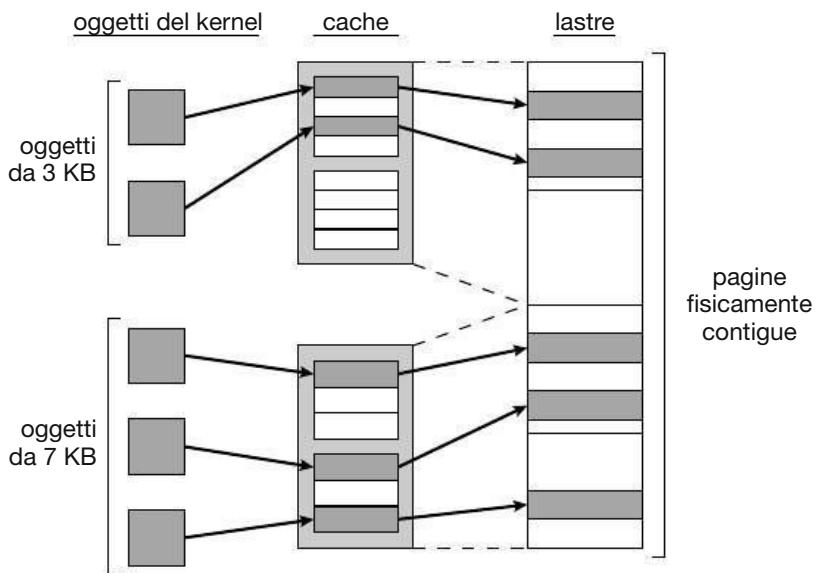


Figura 18.5 Allocatore a lastre di Linux.

Un altro metodo adottato da Linux per la memoria del kernel è noto come *allocazione a lastre* (*slab allocation*). Una **lastra** consiste di una o più pagine fisicamente contigue, usate per contenere le strutture dati del kernel. Una **cache** consiste di una o più lastre. Vi è una sola cache per ciascun tipo di struttura dati del kernel – per esempio, una cache per la struttura dati che rappresenta i descrittori dei processi, una cache dedicata agli oggetti che rappresentano file, un’altra per gli inode, e via di seguito. Ogni cache è popolata da **oggetti**, cioè istanze della struttura dati del kernel che la cache rappresenta. La cache che rappresenta gli inode, per esempio, contiene istanze di strutture inode, quella che contiene i descrittori dei processi memorizza istanze di strutture per descrittore di processo, e così via. La relazione fra lastre, cache e oggetti è illustrata nella Figura 18.5, che mostra oggetti del kernel da 3 e 7 KB. Questi oggetti sono memorizzati nelle rispettive cache, contenenti ciascuna oggetti da 3 KB e 7 KB.

L’algoritmo di allocazione delle lastre utilizza le cache per memorizzare oggetti del kernel. Quando si crea una cache, alcuni oggetti, inizialmente dichiarati **liberi**, sono assegnati alla cache. Il numero di questi oggetti dipende dalla grandezza della lastra corrispondente. Per esempio, una lastra che misura 12 KB (formata da tre pagine contigue di 4 KB) potrebbe contenere sei oggetti da 2 KB. All’inizio, tutti gli oggetti nella cache sono dichiarati liberi. Quando una struttura dati del kernel ha bisogno di un nuovo oggetto, l’allocatore può selezionare dalla cache qualunque oggetto libero e assegnarlo alla richiesta, marcandolo come **usato**.

Consideriamo una situazione in cui il kernel richieda all’allocatore delle lastre la memoria per un oggetto che rappresenta un descrittore di processo. Nei sistemi Linux, il descrittore ha tipo `struct task_struct`, che richiede circa 1,7 KB di memoria. Quando il kernel di Linux genera un nuovo task, richiede alla cache relativa la me-

moria necessaria per l'oggetto `struct task_struct`. La cache risponderà alla richiesta impiegando un oggetto di questo tipo che sia già stato allocato in una lastra e che risulti libero.

In Linux, ogni lastra può presentare uno degli stati seguenti.

1. **Piena.** Tutti gli oggetti della lastra sono dichiarati usati.
2. **Vuota.** Tutti gli oggetti della lastra sono dichiarati liberi.
3. **Parziale.** La lastra contiene sia oggetti usati, sia oggetti liberi.

L'allocatore delle lastre, per soddisfare una richiesta, tenta in primo luogo di utilizzare un oggetto libero in una lastra parziale. Se non ne esistono, assegna un oggetto libero da una lastra vuota. In mancanza di lastre vuote disponibili, una nuova lastra viene creata da pagine fisiche contigue e assegnata a una cache; da tale lastra si attinge la memoria che deve essere allocata all'oggetto.

Gli altri due sottosistemi principali di Linux che attuano una gestione autonoma delle pagine fisiche sono strettamente collegati fra loro: la **cache delle pagine** e il sistema per la memoria virtuale. La cache delle pagine è la cache principale del kernel per i file e costituisce il meccanismo primario attraverso cui effettuare le operazioni di I/O relative ai dispositivi a blocchi (Paragrafo 18.8.1). Qualunque tipologia di file system, inclusi i file system nativi di Linux, basati su unità a disco, e i file system di rete NFS, impiegano la cache delle pagine. Essa memorizza pagine intere di dati contenuti nei file e il suo uso non è limitato ai dispositivi a blocchi: può anche essere usata per memorizzare dati a cui si accede dalla rete. Il sistema per la memoria virtuale gestisce i contenuti dello spazio d'indirizzi virtuali di ciascun processo. Questi due sistemi interagiscono strettamente tra loro: il caricamento di una pagina di dati nella cache delle pagine richiede la mappatura delle pagine nella cache, tramite il sistema di memoria virtuale. I paragrafi successivi illustrano nei particolari il sistema di memoria virtuale.

18.6.2 Memoria virtuale

Il sistema di memoria virtuale di Linux si occupa dello spazio d'indirizzi accessibile a ogni processo. Esso crea pagine di memoria virtuale su richiesta e gestisce il loro caricamento da disco o il loro trasferimento nell'area d'avvicendamento su disco quando è richiesto. In Linux il gestore della memoria virtuale considera lo spazio d'indirizzi di un processo da due punti di vista diversi: come insieme di regioni distinte e come insieme di pagine.

Il primo punto di vista è di natura logica, e riflette le istruzioni che il sistema per la memoria virtuale ha ricevuto riguardo all'organizzazione dello spazio d'indirizzi; quest'ultimo è visto come un insieme di regioni non sovrapposte, e ogni regione rappresenta un sottoinsieme continuo e allineato alle pagine dello spazio d'indirizzi. Ogni regione è descritta internamente da un'unica struttura dati `vm_area_struct` che ne definisce le caratteristiche, compresi i permessi di lettura, scrittura ed esecuzione del processo, e le informazioni riguardanti eventuali file a essa associati. Le re-

gioni riguardanti un dato spazio d'indirizzi sono organizzate in una struttura ad albero binario bilanciato che permette una rapida ricerca della regione corrispondente a un indirizzo virtuale.

Il kernel utilizza anche un secondo punto di vista riguardante lo spazio d'indirizzi, questa volta di natura fisica, rappresentata nella page table hardware legata al processo. Gli elementi della page table determinano l'esatta posizione corrente di ogni pagina della memoria virtuale, sia che essa si trovi in un disco sia che risieda nella memoria fisica; la gestione della memoria dal punto di vista fisico è realizzata da un insieme di procedure invocate dai gestori dei segnali di eccezione del kernel, ognqualvolta un processo tenta di accedere a una pagina che non è in quel momento presente nelle page table. Ogni struttura `vm_area_struct` nella descrizione dello spazio d'indirizzi contiene un campo che punta a una tabella di funzioni che realizzano i servizi chiave di gestione delle pagine per ogni data regione di memoria virtuale. Tutte le richieste di operazioni relative a una pagina non disponibile sono recapitate alla fine all'appropriato gestore nella tabella di funzioni della struttura `vm_area_struct`, cosicché le procedure di gestione della memoria centrale non devono essere a conoscenza dei dettagli riguardanti la gestione di ogni possibile tipo di regione di memoria.

18.6.2.1 Regioni di memoria virtuale

Il sistema Linux opera con diversi tipi di regioni di memoria virtuale. Una prima proprietà caratterizzante un tipo di memoria virtuale è la sua memoria ausiliaria, cioè la descrizione dell'origine delle pagine; nella maggior parte dei casi, si tratta di un file o non è presente. Una regione priva di memoria ausiliaria è il tipo più semplice di memoria virtuale: essa rappresenta **memoria a valori nulli**, nel senso che quando un processo tenta di leggere una pagina di questa regione ottiene come risposta semplicemente una pagina di memoria riempita di zeri.

Una regione la cui memoria ausiliaria sia un file agisce come una finestra sui contenuti di quel file: quando un processo tenta di accedere a una pagina della regione, nella page table è scritto l'indirizzo di una pagina della cache delle pagine del kernel corrispondente all'appropriato offset all'interno del file. La stessa pagina di memoria fisica è usata sia dalla cache delle pagine sia dalle page table del processo, cosicché ogni cambiamento apportato al file dal file system è immediatamente visibile per ogni processo che abbia quel file mappato nel proprio spazio d'indirizzi. Il numero di processi che possono mappare una stessa regione dello stesso file è arbitrario, e a questo scopo ciascuno di essi userà la stessa pagina di memoria fisica.

Un'altra caratteristica di una regione di memoria virtuale è la sua reazione alle operazioni di scrittura; il mapping di una regione di memoria dallo spazio d'indirizzi di un processo può infatti essere *privato* o *condiviso*. Nel primo caso, se il processo scrive in quella regione, la procedura di paginazione rileva la necessità di una copia-tura su scrittura per garantire l'effetto locale dei cambiamenti; nel secondo caso, d'altra parte, l'operazione comporta l'aggiornamento dell'oggetto associato a quella regione, in modo che i cambiamenti siano immediatamente visibili a ogni altro processo che mappa lo stesso oggetto.

18.6.2.2 Durata di uno spazio d'indirizzi virtuale

Il kernel crea un nuovo spazio d'indirizzi virtuale in due situazioni: l'avviamento di un nuovo programma tramite la chiamata di sistema `exec()`; la creazione di un nuovo processo per mezzo della chiamata di sistema `fork()`. Nel primo caso l'operazione è semplice: si assegna al nuovo processo un nuovo spazio d'indirizzi completamente vuoto; è compito delle procedure che caricano il programma preparare lo spazio d'indirizzi con regioni di memoria virtuale.

Nel secondo caso, la creazione di un nuovo processo tramite la `fork()` comporta la creazione di una copia completa dello spazio d'indirizzi virtuali esistente del processo genitore. Il kernel copia i descrittori `vm_area_struct` del processo genitore, e crea poi un nuovo insieme di tabelle delle pagine per il figlio; le tabelle del genitore sono copiate direttamente in quelle del figlio, incrementando solo il conteggio dei riferimenti a ogni pagina coinvolta: ne segue che dopo la `fork()` i processi genitore e figlio condividono le stesse pagine di memoria fisica nei loro spazi d'indirizzi.

Un caso particolare si ha quando durante l'operazione di copiatura s'incontra una regione di memoria virtuale privata; tutte le pagine appartenenti a questa regione sulle quali il processo ha scritto qualcosa sono private, e gli eventuali successivi cambiamenti apportati dal genitore o dal figlio non devono ripercuotersi sulla pagina corrispondente nello spazio d'indirizzi dell'altro processo. Durante la copiatura delle tabelle si stabilisce che le pagine in questione siano soltanto leggibili e le si contrassegna per la copiatura su scrittura: fino a che nessuno dei due processi modifica queste pagine, i due processi condividono le stesse pagine di memoria fisica, ma quando un processo tenta di scrivere in una di esse, si controlla il conteggio dei riferimenti alla pagina, e se essa è ancora condivisa il processo ne copia i contenuti in una nuova pagina di memoria fisica, usando poi questa copia in luogo dell'originale. Si tratta di un meccanismo che assicura il più a lungo possibile la condivisione tra processi di pagine di dati privati; le copie si creano solo se è assolutamente necessario.

18.6.2.3 Paginazione e avvicendamento dei processi

Uno dei compiti importanti che il sistema di memoria virtuale deve assolvere è spostare pagine di memoria dalla memoria fisica al disco quando la memoria fisica in questione è richiesta per altri scopi. I primi sistemi UNIX raggiungevano lo scopo spostando nei dischi i contenuti di interi processi in un'unica soluzione, ma le versioni moderne sfruttano maggiormente la paginazione, cioè il trasferimento di singole pagine di memoria virtuale dalla memoria fisica al disco e viceversa. Il sistema Linux non usa l'avvicendamento d'interi processi, ma adotta esclusivamente i più recenti meccanismi di paginazione.

Il sistema di paginazione si può dividere in due parti: l'**algoritmo di scelta** (*policy algorithm*) decide quali pagine trasferire su disco, e quando farlo; il **meccanismo di paginazione** compie il trasferimento su disco ed esegue anche l'operazione inversa non appena ve ne sia bisogno.

L'algoritmo di scelta di Linux è una versione modificata dell'algoritmo dell'orologio (con seconda chance), descritto nel Paragrafo 9.4.5.2. Il sistema Linux adotta

una scansione in più passi e ogni pagina ha un’*età* ritoccata a ogni passo. L’età, per la precisione, è una misura della “giovinezza” di una pagina, cioè di quanto la pagina è stata usata di recente: le pagine per le quali è stato frequentemente richiesto l’accesso avranno un valore maggiore, mentre il valore delle pagine meno richieste diminuirà a ogni passo. L’algoritmo di paginazione sceglie per il trasferimento le pagine meno frequentemente usate (*least frequently used*, LFU).

Il meccanismo di paginazione è in grado di paginare sia in specifici dispositivi e partizioni di swapping, sia in normali file, anche se quest’ultima operazione è assai più lenta a causa dei rallentamenti indotti dal file system. L’allocazione di blocchi che risiedono in dispositivi di swapping è eseguita attraverso una bitmap dei blocchi usati che si trova sempre nella memoria fisica. L’allocatore adotta un algoritmo *next-fit* che tenta di scrivere le pagine in successioni ininterrotte di blocchi del disco al fine di migliorare le prestazioni. L’allocatore registra che una pagina è stata trasferita nel disco usando una caratteristica delle page table delle moderne CPU: il bit di pagina assente dell’elemento della page table è posto a uno, il che permette al resto dell’elemento della tabella di contenere un indice che identifica il luogo in cui la pagina è stata trasferita.

18.6.2.4 Memoria virtuale del kernel

Il sistema Linux riserva per il proprio uso una regione costante e dipendente dall’architettura dello spazio d’indirizzi virtuali di ogni processo. Gli elementi della tabella delle pagine che si riferiscono a queste pagine del kernel sono contrassegnati come protetti, cosicché le pagine non sono né visibili né modificabili quando la CPU è in esecuzione in modalità utente. Quest’area di memoria virtuale del kernel è costituita da due regioni. La prima è statica e contiene riferimenti a ogni pagina di memoria fisica disponibile nel sistema, fornendo un semplice modo di tradurre indirizzi fisici in indirizzi virtuali durante l’esecuzione di codice di kernel. La parte centrale del kernel insieme con tutte le pagine assegnate dall’ordinario allocatore delle pagine risiedono in questa regione.

Il resto della parte riservata al kernel dello spazio d’indirizzi non è dedicato ad alcuno scopo specifico: gli elementi della tabella delle pagine che si riferiscono a questa regione possono essere modificati dal kernel in modo da puntare a qualunque altra area di memoria desiderata. Il kernel fornisce due funzioni che permettono al codice kernel di usare questa memoria virtuale: `vmalloc()` assegna un numero arbitrario di pagine di memoria fisica, non necessariamente contigue, e le associa a un’unica regione contigua della memoria virtuale del kernel. La funzione `vremap()` associa una sequenza d’indirizzi virtuali a un’area di memoria usata da un driver di dispositivo per compiere l’I/O memory mapped.

18.6.3 Caricamento ed esecuzione dei programmi utente

L’esecuzione dei programmi utente da parte del kernel di Linux si attiva per mezzo della chiamata di sistema `exec()`, la quale chiede al kernel di eseguire un nuovo programma all’interno del processo corrente in modo tale da sovrascrivere integralmente

il contesto d'esecuzione attuale con il contesto iniziale del nuovo programma. Il primo compito di questo servizio di sistema è verificare che il processo chiamante abbia diritto d'esecuzione sul file interessato; risolta tale questione, il kernel invoca una procedura di caricamento per avviare l'esecuzione del programma. Il caricatore non trasferisce necessariamente i contenuti del file da eseguire nella memoria fisica, ma per lo meno mappa il programma in memoria virtuale.

In Linux non c'è una singola procedura per il caricamento dei programmi: il sistema operativo mantiene invece una tabella dei possibili caricatori, e dà a ognuno di essi l'opportunità di tentare di caricare il file in questione al momento dell'esecuzione di una `exec()`. Il motivo originario dell'esistenza di una tale tabella era che nel passaggio dalla versione 1.0 alla versione 1.2 del kernel il formato dei file binari di Linux è stato modificato: i primi kernel adottavano il formato a.out, che è relativamente semplice e comune ai vecchi sistemi UNIX; i più recenti sistemi Linux usano il più moderno formato ELF, adottato adesso dalla maggior parte delle attuali versioni dello UNIX. Il formato ELF offre un certo numero di vantaggi rispetto ad a.out, fra i quali si segnalano la flessibilità e l'estendibilità: si possono aggiungere nuove sezioni a un file binario ELF (per esempio per fornire ulteriori informazioni utili al debugging), senza che le procedure di caricamento vadano incontro a problemi. Grazie alla registrazione di più procedure di caricamento il sistema Linux gestisce facilmente i formati binari ELF e a.out all'interno di un unico sistema.

Nei Paragrafi 18.6.3.1 e 18.6.3.2 sono trattati esclusivamente il caricamento e l'esecuzione di programmi in formato ELF: le procedure di caricamento concernenti il formato a.out sono simili anche se più semplici.

18.6.3.1 Mappatura dei programmi in memoria

Il caricamento di un file binario nella memoria fisica non è eseguito dal caricatore binario; le pagine del file binario sono invece associate a regioni della memoria virtuale: solo quando il programma tenterà di accedere a una data pagina, la conseguente eccezione di page fault causerà il caricamento di quella pagina nella memoria fisica per mezzo della paginazione su richiesta.

L'iniziale associazione del file a regioni di memoria virtuale è compito del caricatore binario del kernel. Un file in formato binario ELF consiste in un'intestazione seguita da diverse sezioni allineate alle pagine: il caricatore ELF lavora leggendo l'intestazione e mappando le sezioni del file in regioni distinte della memoria virtuale.

Nella Figura 18.6 è mostrata l'organizzazione tipica delle regioni di memoria preparate dal caricatore ELF. Il kernel si trova nella regione riservata a un estremo dello spazio d'indirizzi; si tratta di una regione privilegiata di memoria virtuale inaccessibile agli ordinari programmi eseguiti in modalità utente. Il resto della memoria virtuale è disponibile per le applicazioni che possono usare le funzioni del kernel che creano regioni associate a porzioni di un file o disponibili per i dati delle applicazioni.

Il compito del caricatore è di instaurare le associazioni iniziali per permettere l'avvio dell'esecuzione del programma: fra le regioni da inizializzare ci sono lo stack e i segmenti di testo e di dati del programma.

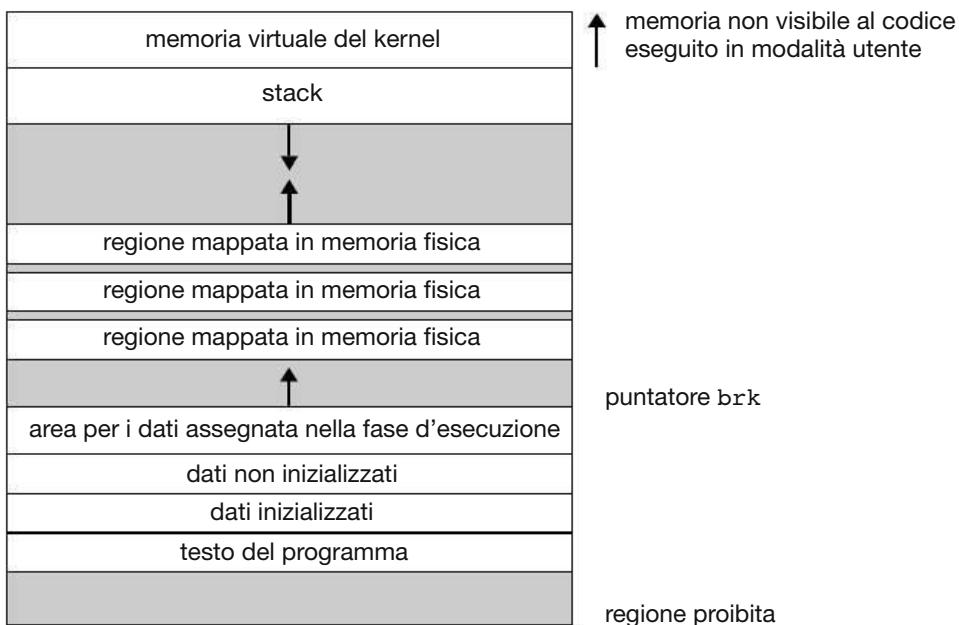


Figura 18.6 Organizzazione della memoria per i programmi ELF.

Lo stack è posto in cima alla memoria virtuale utente, e cresce verso il basso impiegando indirizzi via via inferiori. Comprende copie delle variabili d'ambiente che costituiscono gli argomenti passati al programma dalla chiamata di sistema `exec()`. Le altre regioni sono poste vicino all'estremo inferiore della memoria virtuale; le sezioni del file binario che contengono il testo del programma o dati solamente leggibili sono assegnate a regioni protette dalla scrittura. Sono poi mappati nella memoria virtuale i dati scrivibili inizializzati e infine i dati non inizializzati, che sono mappati in regioni private a valori nulli.

Appena sopra queste regioni di dimensione fissa si trova una regione a dimensione variabile che i programmi possono espandere secondo le necessità al fine di memorizzare dati allocati nella fase d'esecuzione. Ogni processo ha un puntatore `brk` che punta all'estensione attuale di questa regione di dati, e un processo può ampliare o contrarre la sua regione relativa a `brk` con una singola chiamata di sistema `sbrk()`.

Una volta completate queste operazioni, il caricatore inizializza il contatore di programma del processo secondo il punto d'inizio registrato nell'intestazione ELF, e il processo è pronto per lo scheduling.

18.6.3.2 Collegamento statico e dinamico

Una volta che il programma è stato caricato e avviato, tutti i dati necessari contenuti nel file binario sono stati trasferiti nello spazio d'indirizzi virtuale del processo; tuttavia, la maggior parte dei programmi deve eseguire funzioni delle librerie di sistema, e anche queste devono essere caricate. Nel caso più semplice le funzioni di libreria necessarie sono direttamente incorporate nell'eseguibile binario del programma: si dice in questo caso che il programma è collegato staticamente alle librerie, e programmi di questo tipo possono cominciare l'esecuzione non appena caricati.

Lo svantaggio principale del collegamento statico è che ogni programma deve contenere le proprie copie delle stesse funzioni di libreria: sia in termini di memoria fisica sia di spazio di disco è più efficiente caricare in memoria le librerie di sistema una sola volta. Il collegamento dinamico permette di operare in questo modo.

Il sistema Linux realizza il collegamento dinamico in modalità utente grazie a una speciale libreria di collegamento. Ogni programma che utilizza il collegamento dinamico contiene una piccola funzione collegata staticamente che è chiamata all'avviamento del programma. Questa funzione non fa altro che mappare la libreria di collegamento nella memoria virtuale ed eseguire il codice contenuto nella funzione; la libreria di collegamento determina l'elenco delle librerie dinamiche richieste dal programma e i nomi delle variabili e delle funzioni di libreria che il programma richiede, leggendo alcune sezioni del file binario ELF. La libreria di collegamento mappa poi le librerie richieste nella memoria virtuale, e risolve i riferimenti ai simboli contenuti in queste librerie. Non è importante l'esatto punto della memoria dove risiedono queste librerie condivise: si tratta di codice compilato in modo d'essere **codice indipendente dalla posizione** (*position independent code*, PIC), che si può eseguire a partire da qualunque indirizzo di memoria.

18.7 File system

Linux adotta il modello standard di file system di UNIX. In UNIX un file non è necessariamente un oggetto memorizzato in un disco o prelevato da un file server di rete: un file è qualunque elemento sia in grado di gestire l'immissione o l'emissione di un flusso di dati. I driver dei dispositivi possono apparire come file agli occhi dell'utente, e così pure i canali di comunicazione fra processi o le connessioni di rete.

Il kernel di Linux gestisce tutti questi tipi di file nascondendone i dettagli relativi alla struttura interna sotto uno strato software: il file system virtuale (VFS). Di seguito ne diamo una presentazione, e in seguito analizziamo file system standard di Linux: *ext3*.

18.7.1 File system virtuale

Il VFS di Linux è concepito secondo principi di progettazione orientata agli oggetti. È costituito da due componenti: un insieme di definizioni che stabiliscono che cosa può essere un oggetto di tipo file, e uno strato software che serve a manipolare questi oggetti. Il VFS definisce quattro tipi principali di oggetti.

- L'**oggetto inode**, che rappresenta un file singolo.
- L'**oggetto file**, che rappresenta un file aperto.
- L'**oggetto superblocco**, che rappresenta un intero file system.
- L'**oggetto dentry**, che rappresenta una singola voce di una directory.

Per ciascuno dei quattro tipi di oggetti menzionati il VFS definisce un insieme di operazioni: ogni oggetto contiene un puntatore a una tabella di funzioni, e questa elenca

gli indirizzi delle funzioni che realizzano le operazioni richieste per l’oggetto in questione. Per esempio, una API abbreviata per alcune operazioni relative agli oggetti file prevede.

- `int open(...)` – Apre un file.
- `ssize_t read(...)` – Legge da un file.
- `ssize_t write(...)` – Scrive su un file.
- `int mmap(...)` – Mappa un file in memoria.

La definizione completa dell’oggetto file è specificata nella `struct file_operations`, che si trova nel file `/usr/include/linux/fs.h`. Un’implementazione dell’oggetto file (relativa a un tipo di file specifico) deve realizzare ciascuna funzione specificata dalla definizione dell’oggetto file.

Lo strato di programmi del VFS può quindi eseguire un’operazione su uno di questi oggetti chiamando una funzione appropriata scelta dalla tabella delle funzioni dell’oggetto, senza dover sapere in anticipo esattamente con che tipo d’oggetto ha a che fare: il VFS non sa né ha bisogno di sapere se un *inode* rappresenta un file di rete, un file in un disco, una directory o una socket di rete; in ogni caso, la funzione che realizza l’operazione `read()` per quell’oggetto si trova in un punto ben determinato e invariabile della tabella delle funzioni dell’oggetto, e lo strato software del VFS richiamerà la funzione in questione senza interessarsi del modo in cui i dati saranno effettivamente letti.

Gli oggetti *inode* e gli oggetti file costituiscono il meccanismo d’accesso ai file. Un oggetto *inode* è una struttura dati contenente puntatori ai blocchi del disco dove si trova l’effettivo contenuto del file; un oggetto file rappresenta un punto d’accesso ai dati in un file aperto. Un processo non può quindi avere accesso ai dati contenuti in un file relativo a un *inode* se non ottiene prima un oggetto file corrispondente a tale *inode*. L’oggetto file tiene traccia del punto del file nel quale il processo sta correntemente leggendo o scrivendo, cosa che permette di gestire l’I/O sequenziale sui file; esso registra inoltre i permessi (ad esempio di lettura e scrittura) richiesti al momento dell’apertura del file e tiene traccia dell’attività del processo quando ciò è necessario per eseguire letture anticipate adattative (cioè il trasferimento anticipato di dati in memoria al fine di migliorare le prestazioni).

Di solito gli oggetti file appartengono a un solo processo, mentre gli oggetti *inode* sono condivisi. Esiste un oggetto file per ogni istanza di un file aperto, ma l’oggetto *inode* è sempre unico. Anche quando un certo file non è più usato da nessun processo il suo *inode* può essere mantenuto nella cache dal VFS per ottenere migliori prestazioni nell’ipotesi di un prossimo accesso al file relativo. Tutti i dati del file mantenuti nella cache sono organizzati in una lista all’interno dell’oggetto *inode* del file; l’*inode* registra anche alcune informazioni standard sul file, per esempio il proprietario, la dimensione e la data dell’ultima modifica.

Le directory sono trattate in modo leggermente diverso dagli altri file; l’interfaccia di programmazione UNIX definisce un certo numero di operazioni eseguibili sulle directory, come la creazione, rimozione o il cambiamento di nome di un file contenuto

in una directory: al contrario delle operazioni di lettura e scrittura, che necessitano della preliminare apertura del file, le chiamate di sistema relative a queste operazioni non richiedono l'apertura da parte dell'utente dei file coinvolti. Il VFS perciò definisce queste operazioni sulle directory all'interno dell'oggetto *inode*, e non nell'oggetto file.

L'oggetto superblocco rappresenta un insieme di file correlati, che formano un file system completo e autosufficiente. Il kernel del sistema operativo mantiene un singolo oggetto superblocco per ogni unità a disco montata come file system e per ciascun file system di rete attualmente connesso. Il compito primario dell'oggetto superblocco è di fornire accesso agli *inode*. Il VFS identifica ciascun *inode* tramite una coppia univoca (file system/numero di *inode*), e rileva l'*inode* corrispondente a un dato numero di *inode* richiedendo all'oggetto superblocco la restituzione dell'*inode* associato a tale numero.

Infine, un oggetto *dentry* rappresenta una voce di directory che può includere il nome della directory nel percorso di un file (come `/usr`) oppure il file medesimo (per esempio `stdio.h`). Il file `/usr/include/stdio.h`, per esempio, contiene le voci di directory (1) `/`, (2) `usr`, (3) `include` e (4) `stdio.h`. Ciascuno di questi elementi è rappresentato da un oggetto *dentry* distinto.

Si consideri l'esempio seguente. Un processo intende aprire il file il cui percorso è `/usr/include/stdio.h`, servendosi di un editor. Poiché Linux tratta i nomi di directory alla stregua di file, la traduzione di questo percorso richiede che si ottenga, in via preliminare, l'*inode* associato alla radice, `/`. Il sistema operativo deve quindi leggere il file corrispondente per ottenere l'*inode* relativo al file `usr`, e deve ripetere tale processo fino a ottenere l'*inode* per il file `stdio.h`. Dal momento che la traduzione dei percorsi può richiedere molto tempo, Linux mantiene una cache per gli oggetti *dentry*, da consultarsi durante la traduzione dei path name. Estrarre un *inode* dalla cache degli oggetti *dentry* è un processo decisamente più veloce della lettura di file su disco.

18.7.2 File system ext3

Il file system residente su dischi ordinariamente usato dal Linux si chiama *ext3* per ragioni storiche. Originariamente Linux adottava un file system compatibile con quello del sistema Minix allo scopo di facilitare lo scambio di dati con la piattaforma di sviluppo Minix; quel file system, tuttavia, soffriva di gravi limitazioni quali la massima lunghezza dei nomi dei file, pari a 14 caratteri, e la massima dimensione del file system, pari a 64 MB. Il file system del Minix fu sostituito da un nuovo file system chiamato *extfs* (*extended file system*); successive modifiche apportate al fine di migliorare le prestazioni e la scalabilità e aggiungere qualche servizio mancante portarono all'*ext2* (*second extended file system*). Ulteriori sviluppi hanno aggiunto la funzionalità di journaling e il sistema è stato ribattezzato come *ext3* (*third extended file system*). Gli sviluppatori del kernel Linux stanno lavorando su un'evoluzione di *ext3* con l'aggiunta di moderne caratteristiche, come le aree di memorizzazione contigue chiamate *extent*. Questo nuovo file system si chiama *ext4* (*fourth extended file*

system). Nel resto di questo paragrafo verrà trattato ext3, perché è il file system di Linux più diffuso. La maggior parte delle nostre considerazioni vale anche per ext4.

Il file system *ext3* ha molto in comune con l'*ffs* (*fast file system*) del sistema BSD; adotta un meccanismo simile per individuare i blocchi di dati appartenenti a un certo file, memorizzando puntatori a blocchi di dati in blocchi indiretti sparsi per il file system, impiegando fino a tre livelli di indirezione. Come nell'*ffs*, le directory sono memorizzate nei dischi esattamente come ogni altro file, anche se i loro contenuti sono interpretati diversamente: ogni blocco di una directory consiste di una lista concatenata d'elementi ognuno contenente la lunghezza dell'elemento stesso, il nome di un file e il numero dell'*inode* cui l'elemento si riferisce.

Le differenze principali tra l'*ext3* e l'*ffs* riguardano le strategie d'allocazione dello spazio dei dischi: nell'*ffs* lo spazio nei dischi si assegna ai file in blocchi di 8 KB, con blocchi che possono essere ulteriormente suddivisi in frammenti di 1 KB al fine di memorizzare piccoli file o blocchi riempiti solo parzialmente alla fine di un file; per contro, l'*ext3* non usa affatto i frammenti ma esegue tutte le assegnazioni secondo unità più piccole. La dimensione di default di un blocco nell'*ext3* varia in funzione della dimensione complessiva del file system. Le dimensioni supportate sono 1, 2, 4 e 8 KB.

Per ottenere buone prestazioni il sistema operativo deve tentare di eseguire trasferimenti di I/O in unità di grandi dimensioni, ognualvolta ciò sia possibile, raggruppando richieste di I/O fisicamente adiacenti. Il raggruppamento riduce l'overhead per ciascuna richiesta indotto dai driver dei dispositivi, dai dischi e dal controller del disco. Trasferimenti di I/O delle dimensioni di un blocco sono troppo piccoli per garantire buone prestazioni, quindi l'*ext3* adotta strategie d'allocazione concepite per collocare blocchi di un file logicamente adiacenti in blocchi fisicamente adiacenti nel disco, rendendo così possibile l'accorpamento di molte richieste relative a blocchi differenti in un'unica operazione di I/O.

La politica di allocazione di ext3 funziona come segue. Come nell'*ffs*, un file system *ext3* è suddiviso in segmenti chiamati **gruppi di blocchi**; l'*ffs* usa il simile concetto di **gruppi di cilindri**, dove ogni gruppo corrisponde a un singolo cilindro del disco fisico. Si noti che la tecnologia delle moderne unità a disco distribuisce i settori nei dischi con diverse densità, e quindi anche con diverse dimensioni dei cilindri, secondo la distanza dal centro del disco, perciò gruppi di cilindri di dimensione costante non corrispondono necessariamente alla geometria del disco.

Prima di allocare lo spazio a un file, l'*ext3* deve scegliere un gruppo di blocchi per quel file. Per quel che riguarda i blocchi di dati tenta di scegliere lo stesso gruppo di blocchi nel quale è stato allocato l'*inode* del file; per allocare un *inode* che non sia relativo a una directory, sceglie lo stesso gruppo di blocchi della directory cui il file appartiene. Le directory, invece di essere raggruppate, sono distribuite su tutti i blocchi disponibili. Queste strategie hanno l'obiettivo di mantenere informazioni correlate all'interno di uno stesso gruppo di blocchi, ma anche di distribuire il carico d'informazioni su tutti i gruppi di blocchi del disco, così da ridurre la frammentazione d'ogni area del disco stesso.

All'interno di un gruppo di blocchi, l'*ext3* tenta di compiere le assegnazioni in modo fisicamente contiguo se è possibile, riducendo quindi la frammentazione. Esso mantiene una bitmap di tutti i blocchi liberi di un gruppo di blocchi: durante l'allocazione dei primi blocchi relativi a un nuovo file, l'*ext3* comincia la ricerca di un blocco libero dall'inizio del gruppo di blocchi; quando invece si tratta di ampliare un file, continua la ricerca dall'ultimo blocco assegnato al file. La ricerca si svolge in due fasi: innanzitutto si cerca un intero byte libero nella mappa di bit; se ciò non riesce, si cerca un bit libero. La ricerca di interi byte liberi ha lo scopo di tentare di allocare lo spazio nei dischi in porzioni di almeno 8 blocchi.

Una volta che si sia identificato un blocco libero, si estende la ricerca all'indietro fino a incontrare un blocco assegnato. Quando si trova un byte libero nella mappa di bit, quest'estensione a ritroso impedisce che l'*ext3* lasci un buco fra l'ultimo blocco assegnato nel precedente byte non nullo e il byte libero trovato; una volta individuato il blocco da allocare tramite una ricerca per byte o per bit, l'*ext3* estende l'allocazione in avanti fino a un massimo di otto blocchi **preassegnando** questo spazio supplementare al file. La preallocazione concorre a diminuire il grado di frammentazione nel corso di scritture intercalate in file diversi, e riduce anche il costo dell'allocazione in termini d'impegno della CPU grazie all'allocazione simultanea di più blocchi; i blocchi preassegnati sono nuovamente riportati sulla bitmap dei blocchi liberi al momento della chiusura del file.

Nella Figura 18.7 sono illustrati i criteri d'allocazione. Ogni riga rappresenta una sequenza di bit posti a uno o a zero nella bitmap di allocazione; questi bit contrassegnano i blocchi liberi e i blocchi in uso del disco. La prima possibilità è che si trovino blocchi libri sufficientemente vicini al punto d'inizio della ricerca, in tal caso essi

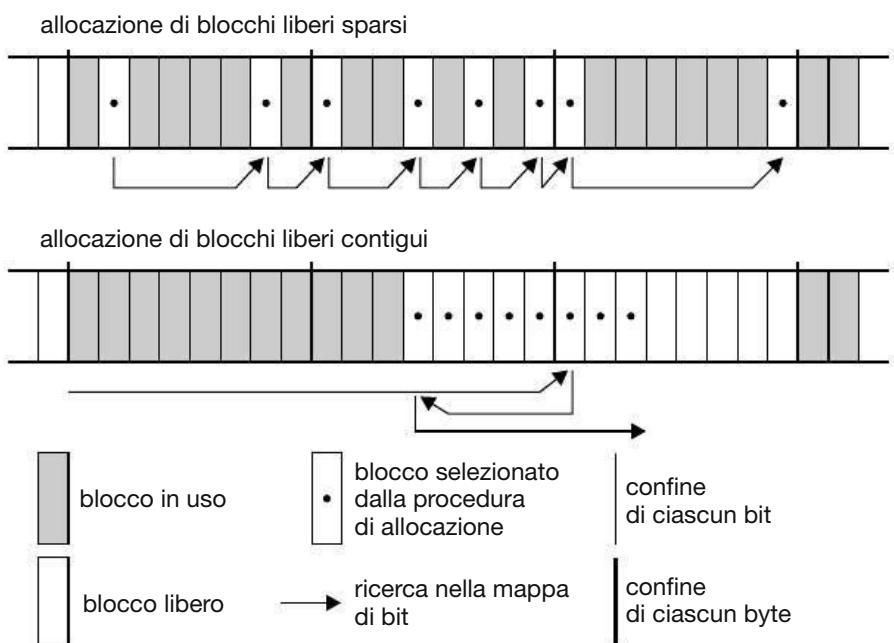


Figura 18.7 Criteri di allocazione dei blocchi dell'*ext3*.

sono assegnati indipendentemente dal livello di frammentazione; infatti, dato che i blocchi sono vicini fra loro, è probabile che possano essere letti senza dover eseguire operazioni di seek nel disco, cosa che compensa parzialmente la frammentazione. Inoltre, allocare tutti questi blocchi a un solo file è una scelta che a lungo termine si rivela migliore dell'allocazione di blocchi isolati a file distinti, quando a lungo andare le regioni ampie di spazio libero su disco divengono rare. La seconda possibilità è che non si sia trovato un blocco libero nelle vicinanze del punto d'inizio della ricerca, quindi si procede nella ricerca di un intero byte libero nella bitmap; se si assegnasse questo byte senza prendere alcuna precauzione si creerebbe un frammento di spazio libero prima di esso, quindi è necessario tornare indietro per colmare lo spazio rimasto fra l'ultimo blocco assegnato precedentemente e il blocco in questione. Infine, per rispettare la dimensione predefinita di un'allocazione, che è di otto blocchi, si estende l'allocazione in avanti fin quando è necessario.

18.7.3 Annotazione delle modifiche (journaling)

Il file system ext3 supporta l'**annotazione delle modifiche** (*journaling*), ovvero le modifiche effettuate nel file system sono trascritte, in modo sequenziale, in un log (detto *journal*). L'insieme di operazioni che esegue un compito specifico è detto **transazione**. Una volta trascritte le modifiche nel log, la transazione relativa si considera completata (*committed*). Una chiamata di sistema che modifica il file system (per esempio, `write()`) può dunque restituire il controllo al processo utente, permettendogli di proseguire con l'esecuzione. Nel frattempo, le voci del journal inerenti alla transazione sono replicate nelle strutture del file system reale: mentre sono eseguite queste modifiche, un puntatore viene aggiornato per indicare quali azioni sono complete e quali ancora da completare. Quando un'intera transazione è completata, viene rimossa dal journal. Quest'ultimo, che consiste in un buffer circolare, può risiedere in una sezione separata del file system, o anche in un disco dedicato. È più efficiente, ma più complesso, collocarlo sotto testine di lettura e di scrittura dedicate, diminuendo così le possibilità di contesa delle testine e il tempo di seek.

Se il sistema subisce un crash, nel journal rimarranno alcune transazioni. Le transazioni presenti non sono mai state completate nel file system; ma, poiché il sistema operativo le considera compiute con successo, esse devono essere completate. Le transazioni possono essere eseguite a partire dalla posizione corrente del puntatore sino al termine del lavoro, e le strutture del file system rimarranno coerenti. L'unico problema sorge nell'eventualità che una transazione fallisca: in altri termini, quando essa non è completata con successo prima del crash del sistema. Tutte le modifiche introdotte dalle transazioni fallite devono essere annullate, onde preservare la coerenza del sistema. Questo ripristino è tutto ciò che bisogna fare, in seguito a un crash, e ciò elimina i problemi di verifica della coerenza.

I file system con annotazioni delle modifiche possono anche essere più veloci di quelli che ne sono privi, visto che le modifiche risultano molto più veloci quando sono applicate al journal in memoria anziché direttamente alle strutture dati su disco. La ragione di questo miglioramento risiede nella maggior efficienza che l'esecuzione

di I/O ad accesso sequenziale permette, rispetto agli accessi casuali. Le onerose operazioni di scrittura sincrone ad accesso casuale sul file system, sono convertite in operazioni molto meno dispendiose di scrittura sincrona sequenziale nel journal. I cambiamenti determinati da tali operazioni si riportano successivamente in modo asincrono nelle strutture appropriate, attraverso operazioni di scrittura ad accesso casuale. Il risultato complessivo è un significativo guadagno in termini di prestazioni per le operazioni orientate ai metadati, come la creazione e la cancellazione dei file. Per sfruttare questo miglioramento delle prestazioni, ext3 può essere configurato per annotare solo i metadati e non i file di dati.

18.7.4 Il process file system di Linux

Il file system VFS è sufficientemente flessibile da permettere l'implementazione di un file system che non archivia in modo permanente alcun dato, ma semplicemente funge da interfaccia per qualche altro servizio. Il **process file system** di Linux, noto come `/proc`, è un esempio di file system i cui contenuti non sono in realtà memorizzati in alcun luogo, ma sono calcolati on demand in seguito alle richieste di I/O degli utenti.

Un file system `/proc` non è una caratteristica esclusiva di Linux: UNIX SVR4 adottava un file system di questo tipo come interfaccia efficiente per le funzioni di debugging dei processi del kernel; ogni sottodirectory del file system corrispondeva non a una directory in qualche disco, ma piuttosto a un processo attivo del sistema, cosicché un elenco dei contenuti del file system presentava una directory per processo, essendo il nome della directory la rappresentazione decimale, in formato ASCII, dell'identificatore unico del processo (PID).

Il sistema Linux implementa un file system di questo tipo, ma lo amplia notevolmente aggiungendo un certo numero di ulteriori directory e file di testo sotto la directory radice del file system. Questi elementi corrispondono a diverse statistiche relative al kernel e ai driver caricati; il file system `/proc` fornisce ai programmi la possibilità di accedere a queste informazioni leggendo semplici file di testo che nell'ambiente utente standard di UNIX è possibile trattare con strumenti particolarmente efficaci. Per fare un esempio, il classico comando `ps` di UNIX, che elenca gli stati di tutti i processi in esecuzione, è stato in passato realizzato come un processo privilegiato che leggeva gli stati dei processi direttamente dalla memoria virtuale del kernel; in Linux, questo comando è realizzato da un programma del tutto ordinario che semplicemente esamina e compone in forma maggiormente leggibile le informazioni contenute nel `/proc`.

Il file system `/proc` deve implementare una struttura di directory e i contenuti dei file in esso residenti. Giacché un file system UNIX è per definizione un insieme di *inode* relativi a file e directory identificati dal loro numero di *inode*, il file system `/proc` deve definire in modo unico un numero di *inode* permanente per ogni directory e per i file in essa contenuti. Una volta instaurata questa corrispondenza si può usare il numero di *inode* per identificare esattamente l'operazione richiesta da un utente che tenti di leggere un certo file o di esaminare una certa directory. Quando si

leggono dati da uno di questi file, il file system `/proc` estrae l'informazione appropriata, la compone in forma testuale e la colloca nel buffer di lettura del processo richiedente.

La corrispondenza fra un numero di *inode* e il tipo d'informazione divide il numero di *inode* in due parti: nel sistema Linux un PID è lungo 16 bit, ma un numero di *inode* è lungo 32 bit; i primi 16 bit del numero di *inode* sono interpretati come un PID, e i rimanenti definiscono il tipo d'informazione richiesta in merito al processo.

Un PID pari a zero non è ammesso, quindi un campo PID pari a zero nel numero di *inode* significa che questo *inode* contiene informazioni globali e non relative a uno specifico processo: `/proc` contiene file globali distinti che forniscono informazioni come la versione del kernel, la memoria libera, statistiche sulle prestazioni, e i driver attualmente attivi.

Non tutti i numeri di *inode* di questo tipo sono riservati; il kernel può allocare nuovi *inode* del `/proc` dinamicamente, mantenendo una mappa dei numeri di *inode* assegnati. Esso mantiene anche una struttura dati ad albero degli elementi globali del file system `/proc` che sono stati registrati: ogni elemento contiene il numero di *inode* del file, il nome del file, i permessi d'accesso e le funzioni speciali usate per generare i contenuti del file; i driver possono registrare o rimuovere elementi da quest'albero in qualunque momento, e una sezione speciale dell'albero che appare nella directory `/proc/sys` è riservata alle variabili del kernel. I file contenuti in quest'albero sono trattati per mezzo di un insieme di procedure di gestione comuni che permettono sia la lettura sia la scrittura di queste variabili, cosicché un amministratore del sistema può regolare i valori dei parametri del kernel semplicemente scrivendoli in ASCII nel file appropriato.

Per permettere alle applicazioni di accedere efficientemente a queste variabili, il sottoalbero `/proc/sys` è reso disponibile tramite una speciale chiamata di sistema, `sysctl()`, che legge e scrive le stesse variabili in formato binario anziché sotto forma di testo, evitando i ritardi indotti dal file system; questa chiamata di sistema non fornisce una funzione aggiuntiva ma legge semplicemente l'albero dinamico degli elementi di `/proc` per decidere a quali variabili l'applicazione si riferisce.

18.8 Input e output

Per l'utente, il sistema per l'I/O di Linux appare molto simile a quello di un qualunque sistema UNIX: tutti i driver dei dispositivi, nei limiti del possibile, sono rappresentati come file ordinari. L'utente apre un canale d'accesso a un dispositivo nello stesso modo in cui apre un qualunque file: i dispositivi compaiono come oggetti del file system. L'amministratore del sistema può creare file speciali all'interno del file system che contengono riferimenti a uno specifico driver di dispositivo, e un utente che apra un tale file potrà leggere e scrivere nel dispositivo associato. Grazie all'ordinario sistema di protezione dei file, che stabilisce chi può accedere a quali file, l'amministratore può controllare l'accesso ai dispositivi.

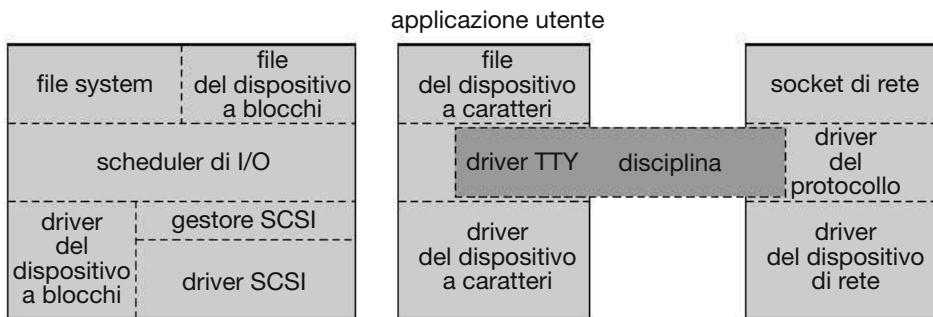


Figura 18.8 Struttura del sistema dei driver dei dispositivi.

Il sistema Linux suddivide i dispositivi in tre classi: dispositivi a blocchi, dispositivi a caratteri, e dispositivi di rete. Nella Figura 18.8 è illustrata la struttura globale del sistema dei driver dei dispositivi.

I **dispositivi a blocchi** comprendono tutti i dispositivi capaci di fornire l'accesso casuale a blocchi di dati di dimensione fissa completamente indipendenti; fra questi vi sono i dischi magnetici, i floppy disk, i CD-ROM, i Blu-ray e le memorie flash. I dispositivi a blocchi sono generalmente usati per contenere interi file system, ma c'è anche la possibilità di accedere direttamente a questi dispositivi per permettere a programmi specializzati di creare e riparare i file system in essi contenuti. Anche le applicazioni possono accedere direttamente a questi dispositivi se lo desiderano; una base di dati, per esempio, potrebbe voler mappare i dati nei dischi in modo appropriato ai propri scopi specifici, invece di usare il file system generale.

I **dispositivi a caratteri** comprendono la maggior parte degli altri dispositivi, come mouse e tastiere. La differenza fondamentale tra i dispositivi a blocchi e quelli a caratteri è data dall'accesso casuale: ai primi, infatti, si può accedere in modo casuale, mentre i secondi sono ad accesso seriale. La ricerca di una certa posizione all'interno di un file, per esempio, potrebbe essere prevista per un DVD, ma non avrebbe senso nel caso di un mouse.

I **dispositivi di rete** sono trattati diversamente dai dispositivi delle altre due classi: gli utenti non possono trasferire direttamente dati ai dispositivi di rete, ma devono comunicare indirettamente tramite il sottosistema di rete del kernel. L'interfaccia ai dispositivi di rete è illustrata separatamente nel Paragrafo 18.10.

18.8.1 Dispositivi a blocchi

I dispositivi a blocchi costituiscono l'interfaccia principale a tutte le unità a disco del sistema. Le prestazioni sono una questione di particolare rilevanza per i dischi, e il sistema per i dispositivi a blocchi deve quindi fornire servizi che permettano di accedere ai dischi il più rapidamente possibile. Questa funzionalità si ottiene attraverso lo scheduling delle operazioni di I/O.

Per i dispositivi a blocchi, un **blocco** rappresenta l'unità sulla base della quale il kernel esegue le operazioni di I/O. Quando un blocco è caricato in memoria, è me-

morizzato in un **buffer**. Il **gestore delle richieste** è lo strato di software incaricato di gestire lettura e scrittura dei contenuti del buffer, da e verso il driver del dispositivo a blocchi.

Una lista distinta di richieste è mantenuta per ogni driver di un dispositivo a blocchi. Tradizionalmente lo scheduling di queste richieste avviene mediante l'algoritmo dell'ascensore (C-SCAN) che sfrutta l'ordine nel quale le richieste sono inserite o rimosse dalle liste. Le liste di richieste sono mantenute in ordine crescente di settore iniziale. Quando una richiesta è selezionata per essere elaborata da un driver, non è rimossa dalla lista; la rimozione avviene solo quando l'operazione di I/O è terminata: a questo punto il driver prosegue con la successiva richiesta nella lista anche se nuove richieste sono state inserite nella lista prima della richiesta attiva.

Il kernel Linux versione 2.6 ha introdotto un nuovo algoritmo di scheduling dell'I/O. Anche se rimane disponibile un semplice algoritmo dell'ascensore, lo scheduler dell'I/O di default è ora il Completely Fair Queuing scheduler (CFQ). Lo scheduler CFQ è fondamentalmente diverso dagli algoritmi dell'ascensore, perché invece di ordinare le richieste in una lista mantiene un insieme di liste, per default una per ogni processo. Le richieste provenienti da un processo finiscono nella lista di quel processo. Per esempio, se due processi stanno emettendo richieste di I/O, CFQ manterrà due liste separate di richieste, una per ogni processo. Le liste sono mantenute secondo l'algoritmo C-SCAN.

CFQ serve le liste in maniera differenziata. Mentre un algoritmo tradizionale C-SCAN non è sensibile allo specifico processo, CFQ serve la lista di ogni processo secondo una politica round-robin. CFQ estrae un numero configurabile di richieste (4 per default) da ciascuna lista prima di passare alla lista successiva. Questo metodo garantisce equità a livello dei processi: ogni processo riceve una uguale frazione della banda del disco. Il risultato è particolarmente vantaggioso in presenza di carichi di lavoro interattivi, dove la latenza di I/O è importante. Nella pratica CFQ si comporta bene con la maggior parte dei carichi di lavoro.

18.8.2 Dispositivi a caratteri

Un driver di un dispositivo a caratteri può essere qualsiasi driver che non offre l'accesso diretto a blocchi di dati di dimensioni fisse. Ogni driver di un dispositivo a caratteri registrato dal kernel di Linux deve anche registrare un insieme di funzioni che realizzino le operazioni di I/O su file gestite dal driver. Il kernel non esegue quasi alcuna elaborazione preliminare su una richiesta di lettura o scrittura su file relativa a un dispositivo a caratteri, ma passa semplicemente la richiesta al dispositivo in questione lasciandogli il compito di servirla.

L'eccezione principale a questa regola è costituita dai driver dei dispositivi a caratteri relativi ai terminali: il kernel mantiene un'interfaccia uniforme a questi driver per mezzo di un insieme di strutture `tty_struct`: ognuna di esse fornisce il controllo del flusso e la bufferizzazione dei dati provenienti dal terminale e passa questi dati a un **interprete** (*line discipline*).

Una **disciplina** è un interprete delle informazioni provenienti dal terminale. La più comune è la `tty`, che incolla il flusso dei dati del terminale ai flussi di I/O standard dei processi utenti in esecuzione, permettendo a questi processi di comunicare direttamente con il terminale dell'utente. Questo compito è complicato dal fatto che ci può essere più di un processo di questo tipo contemporaneamente in esecuzione, e la disciplina `tty` è responsabile del collegamento e del distacco dell'I/O del terminale dai vari processi a esso collegati a mano a mano che questi processi sono sospesi o riattivati dall'utente.

Il sistema Linux dispone di altre discipline che non hanno nulla a che fare con l'I/O di un processo utente: i protocolli di rete PPP e SLIP rappresentano un modo di codificare un collegamento di rete su un dispositivo terminale come una linea seriale. Questi protocolli sono realizzati nel sistema Linux come driver che da un lato appaiono al sistema della gestione dei terminali come interpreti con una determinata disciplina, e dall'altro appaiono al sistema per la gestione della rete come driver di dispositivi di rete: dopo che una di queste discipline è stata attivata su un dispositivo terminale, ogni dato che appare su quel terminale sarà direttamente instradato all'appropriato driver del dispositivo di rete.

18.9 Comunicazione fra processi

Il sistema UNIX fornisce un ricco ambiente per la comunicazione fra processi. La comunicazione può limitarsi alla notifica del verificarsi di un evento, da parte di un processo a un altro processo, ma può anche coinvolgere il trasferimento di dati fra processi.

18.9.1 Sincronizzazione e segnali

Il meccanismo standard di UNIX usato per comunicare a un processo che un evento si è verificato è il **segnale**. I segnali si possono inviare da un processo a ogni altro processo, con alcune limitazioni che riguardano i segnali inviati a processi di proprietà di un altro utente. Tuttavia è disponibile solo un limitato numero di segnali, che per di più non veicolano al processo destinatario altra informazione se non il mero fatto che un segnale è stato generato. Non è necessario che un segnale sia generato da un altro processo: anche il kernel genera internamente dei segnali; per esempio, può inviare un segnale a un processo server quando arrivano dati in un canale di rete, a un processo genitore quando un figlio termina, o a un processo in attesa quando è trascorso un intervallo di tempo impostato in un timer.

Internamente il kernel di Linux non usa segnali per comunicare con i processi eseguiti in modalità kernel: se uno di essi attende il verificarsi di un evento, non userà i segnali per ricevere la comunicazione dell'evento. Infatti, le comunicazioni nel kernel riguardanti eventi asincroni sono realizzate per mezzo degli stati di scheduling e delle strutture `wait_queue`. Questo meccanismo permette ai processi eseguiti in modalità kernel di informarsi vicendevolmente sugli eventi rilevanti, e dà la possibilità ai driver

e al sistema di networking di generare eventi. Ogniqualvolta un processo desidera attendere la terminazione di un evento, si sposta alla struttura `wait_queue` associata a quell'evento e comunica allo scheduler di non poter essere posto in esecuzione; l'evento in questione riattiva tutti i processi nella `wait_queue` al momento della sua terminazione. Si tratta di una procedura che permette a più processi di attendere il verificarsi di un unico evento: per esempio, se diversi processi stanno tentando di leggere un file da un disco, essi saranno tutti riattivati quando i dati sono stati trasferiti correttamente in memoria.

Anche se i segnali sono sempre stati il principale meccanismo di comunicazione asincrona fra processi, il sistema Linux dispone anche del meccanismo basato sui semafori dello UNIX System V: un processo si pone in attesa a un semaforo con la stessa facilità con la quale aspetterebbe un segnale, ma i semafori hanno due vantaggi: poter essere condivisi in gran numero da diversi processi indipendenti, e l'esecuzione atomica delle operazioni su più semafori. Internamente il meccanismo standard delle `wait queue` di Linux sincronizza i processi che comunicano tramite semafori.

18.9.2 Passaggio di dati fra processi

Il sistema Linux mette a disposizione diversi metodi per il passaggio di dati fra processi. Il meccanismo standard delle *pipe* UNIX permette a un processo figlio di ereditare dal genitore un canale di comunicazione; i dati scritti a un'estremità di tale canale di comunicazione possono essere letti all'altra estremità. Nel sistema Linux le pipe sono semplicemente un tipo particolare di *inode* del file system virtuale, dotato di una coppia di strutture `wait_queue` per sincronizzare il lettore e lo scrittore. Il sistema UNIX definisce anche un insieme di servizi di rete che possono inviare flussi di dati a processi locali o remoti. I servizi di rete sono trattati nel Paragrafo 18.10.

Un altro metodo di comunicazione è la memoria condivisa che offre un modo estremamente rapido di comunicare quantità arbitrarie di dati: qualsiasi informazione scritta da un processo in una regione di memoria condivisa può essere immediatamente letta da ogni altro processo che abbia quella regione associata al suo spazio d'indirizzi. Il principale svantaggio della memoria condivisa è che di per sé non offre alcun metodo di sincronizzazione: un processo non può chiedere al sistema operativo se qualche dato sia stato scritto in una regione di memoria condivisa, né può sospendere l'esecuzione fino a che una tale operazione di scrittura si sia verificata. La memoria condivisa diviene uno strumento particolarmente potente quando è usata insieme con un altro meccanismo di comunicazione fra processi che fornisca la sincronizzazione mancante.

In Linux una regione di memoria condivisa è un oggetto permanente che può essere creato o cancellato dai processi; un tale oggetto è trattato come se fosse un piccolo spazio d'indirizzi indipendente: gli algoritmi di paginazione possono scegliere di trasferire nel disco alcune pagine di memoria condivisa allo stesso modo in cui possono scegliere di trasferire una pagina di dati di un processo. L'oggetto che rappresenta la memoria condivisa agisce come memoria ausiliaria della regione di memoria condivisa proprio come un file può essere la memoria ausiliaria di una regione

di memoria virtuale. Quando un file è associato a una regione di memoria virtuale, il verificarsi di un page fault causa la mappatura in memoria virtuale dell'appropriata pagina del file; similmente le associazioni relative alla memoria condivisa fanno sì che dalle eccezioni di pagina mancante risulti il caricamento in memoria di pagine prelevate da un oggetto permanente di memoria condivisa. Sempre in analogia con i file, gli oggetti che rappresentano la memoria condivisa conservano i loro contenuti anche se al momento non sono associati alla memoria virtuale di alcun processo.

18.10 Struttura di rete

Il networking è uno dei punti di forza di Linux: questo sistema operativo non solo gestisce i protocolli standard Internet per la comunicazione fra sistemi UNIX, ma realizza anche altri protocolli originariamente sviluppati per sistemi operativi diversi da UNIX. Essendo stato originariamente concepito per i PC e non per grandi stazioni di lavoro o server, gestisce molti protocolli usati nelle reti di PC, per esempio l'Apple-Talk e l'IPX.

Internamente il kernel di Linux realizza i servizi di rete per mezzo di tre strati software:

1. l'interfaccia socket;
2. i driver dei protocolli;
3. i driver dei dispositivi di rete.

Le applicazioni degli utenti richiedono tutti i servizi di rete tramite l'interfaccia socket; essa è progettata per somigliare allo strato di socket del 4.3BSD, cosicché i programmi scritti per far uso delle socket di Berkeley potranno essere eseguiti dal sistema Linux senza che sia necessario apportare modifiche al codice sorgente. L'interfaccia socket BSD è sufficientemente generale da poter rappresentare gli indirizzi di rete di un'ampia gamma di protocolli; il sistema Linux usa questa sola interfaccia per accedere a tutti i protocolli supportati, e non solo quelli dei sistemi BSD standard.

Lo strato software seguente è quello dei protocolli, organizzato in maniera simile all'analogo livello del BSD. Si richiede che tutti i dati che arrivino a questo livello siano etichettati da un identificatore che specifichi a quale protocollo appartengono; ciò vale sia per i dati provenienti dalla socket di un'applicazione, sia per quelli provenienti dal driver di un dispositivo di rete. I protocolli possono eventualmente comunicare fra di loro: all'interno dell'insieme dei protocolli Internet, per esempio, protocolli distinti gestiscono l'instradamento, la comunicazione degli errori e la ritrasmissione affidabile dei dati persi.

Lo strato dei protocolli può riscrivere i pacchetti, crearne di nuovi, dividerli in frammenti o riassemblarli da frammenti, o anche semplicemente scartare i dati in arrivo; alla fine, quando ha terminato l'elaborazione di un gruppo di pacchetti, li passa all'interfaccia socket se la destinazione dei dati è locale, oppure a un driver di un dispositivo di rete, se i pacchetti devono essere inviati lungo la rete. Lo strato di protocolli decide a quale socket o dispositivo inviare il pacchetto.

Tutta la comunicazione che avviene fra gli strati che realizzano i servizi di rete è eseguita passando singole strutture dette `skbuff` (socket buffer); esse contengono un insieme di puntatori a un'unica regione contigua di memoria che funge da buffer all'interno del quale i pacchetti di rete possono essere assemblati. I dati validi puntati da una struttura `skbuff` non devono necessariamente trovarsi all'inizio del buffer, e non devono neanche estendersi fino alla sua fine: il codice di networking può aggiungere o togliere dati agli estremi del pacchetto, a patto che il risultato non superi le dimensioni del buffer. Questa possibilità è particolarmente importante per le moderne CPU, i cui miglioramenti in velocità hanno abbondantemente superato le prestazioni della memoria centrale: l'architettura basata sulle strutture `skbuff` rende possibile gestire le intestazioni e dei codici per il controllo degli errori dei pacchetti evitando duplicazioni non necessarie dei dati.

L'insieme di protocolli più importante in Linux è la suite di protocolli IP, composta di un certo numero di protocolli distinti. Il protocollo IP realizza l'instradamento da un calcolatore all'altro ovunque nella rete; su questo protocollo si appoggiano i protocolli UDP, TCP e ICMP. Il protocollo UDP trasferisce singoli datagrammi fra i calcolatori; il protocollo TCP instaura connessioni affidabili fra sistemi con consegna garantita nell'ordine originario, e ritrasmissione automatica dei dati persi; il protocollo ICMP si usa per la trasmissione di messaggi di stato e di vari tipi di messaggi d'errore.

Si assume che i pacchetti (`skbuff`) che raggiungano il software dei protocolli siano già etichettati da un identificatore interno che indica a quale protocollo è attinente il pacchetto. Differenti driver dei dispositivi di rete codificano il tipo di protocollo in modo differente, quindi l'identificazione del protocollo va eseguita all'interno dei driver stessi. I driver usano a questo scopo una tabella hash degli identificatori di protocollo noti, e passano poi il pacchetto al protocollo appropriato; è possibile aggiungere nuovi protocolli alla tabella hash sfruttando il meccanismo di caricamento dei moduli del kernel.

I pacchetti IP che giungono al sistema sono consegnati al driver IP, il cui compito è quello di eseguire l'instradamento: esso determina la destinazione del pacchetto e lo inoltra all'appropriato driver di protocollo interno per la consegna locale, oppure lo reinserirà nella coda del driver di un dispositivo di rete se deve essere inoltrato a un altro calcolatore. La decisione riguardante l'instradamento viene presa sulla base di due tabelle: la base permanente di informazioni sull'instradamento (*forwarding information base*, FIB), e una cache delle più recenti decisioni di instradamento. La FIB contiene informazioni sulle configurazioni d'instradamento e può specificare percorsi basati su singoli indirizzi di destinazione o su raggruppamenti di destinazioni; è organizzata come un insieme di tabelle hash indicizzate dagli indirizzi di destinazione, e la ricerca parte sempre dalle tabelle che contengono i percorsi più specifici. Quando una ricerca di questo tipo ha successo, il percorso individuato è posto nella cache dei percorsi, la quale contiene solo instradamenti per destinazioni specifiche e non per gruppi, in modo che le ricerche siano più rapide. Un elemento della cache dei percorsi viene eliminato dopo un certo tempo d'inutilizzo.

In diverse fasi il protocollo IP passa i pacchetti a una sezione distinta di codice per la **gestione del firewall**, cioè il filtraggio selettivo dei pacchetti secondo criteri arbi-

trari ma di solito relativi alle strategie di sicurezza. Il gestore del firewall mantiene un certo numero di **catene di firewall** distinte, e permette il confronto di una struttura `skbuff` con una qualunque di esse; catene distinte assolvono funzioni distinte: una si usa per inoltrare i pacchetti, una per la ricezione dei pacchetti, e una per i dati generati localmente. Ogni catena è costituita da un insieme ordinato di regole, ciascuna delle quali specifica una fra varie funzione di filtro, oltre a pattern di dati che devono trovare riscontro nei pacchetti.

Due altre funzioni eseguite dal driver IP sono la scomposizione e il riassemblaggio dei pacchetti di grandi dimensioni: un pacchetto da spedire troppo grande per essere posto nella coda di un dispositivo viene semplicemente diviso in **frammenti** più piccoli che possono essere posti in coda; il calcolatore ricevente si occuperà di riassemblare i frammenti. Il driver IP mantiene un oggetto `ipfrag` per ogni frammento che attende il riassemblaggio, e un oggetto `ipq` per ogni datagramma in corso d'assemblaggio. I frammenti in arrivo sono confrontati con ogni `ipq`, e nel caso di riscontro positivo il frammento è aggiunto all'oggetto; altrimenti, si crea un nuovo `ipq`. Quando l'ultimo frammento di un `ipq` è arrivato si costruisce una struttura `skbuff` interamente nuova per alloggiare il pacchetto, e si passa di nuovo il pacchetto al driver IP.

I pacchetti che l'IP identifica come destinati al calcolatore locale sono passati a uno degli altri driver di protocollo. I protocolli TCP e UDP adottano lo stesso metodo per associare ogni pacchetto alle relative socket mittenti e destinarie: ogni coppia di socket collegate è identificata in modo unico dagli indirizzi del mittente e del destinatario e dai corrispondenti numeri di porta. Gli elenchi delle socket sono riuniti in una tabella hash la cui chiave è costituita da questi quattro valori di indirizzo e porta e che può essere usata per individuare le socket relative ai pacchetti in arrivo. Il protocollo TCP deve anche occuparsi delle connessioni non affidabili, e a tal fine mantenere liste ordinate dei pacchetti trasmessi, ma non riscontrati, che saranno ritrasmessi dopo un certo tempo, e liste dei pacchetti ricevuti in modo disordinato, che saranno presentati alla socket una volta ricevuti i dati mancanti.

18.11 Sicurezza

Il modello di sicurezza di Linux è strettamente correlato ai modelli di sicurezza tipici di UNIX. I problemi relativi alla sicurezza sono classificabili in due gruppi.

1. **Autenticazione.** Assicurare che nessuno possa accedere al sistema senza prima dimostrare di averne diritto.
2. **Controllo dell'accesso.** Fornire un meccanismo che permetta di controllare se un utente abbia diritto d'accesso a un certo oggetto, e che impedisca l'accesso se l'esito del controllo è negativo.

18.11.1 Autenticazione

L'usuale meccanismo di autenticazione in UNIX è sempre stato basato su un file di password leggibile da tutti: la password di un utente è combinata con un valore casuale, e il risultato è codificato tramite una funzione di codifica non invertibile e infine registrato nel file delle password. L'uso di una funzione di codifica non invertibile significa che la password originale non si può ricavare dal file delle password se non per tentativi. Quando un utente presenta una password al sistema essa viene ricombinata con il valore casuale memorizzato nel file delle password; al risultato si applica la funzione di codifica non invertibile, e se ciò che si ottiene coincide con il contenuto del file delle password l'utente è ammesso al sistema.

Storicamente le implementazioni di UNIX di questo meccanismo hanno incontrato diversi problemi: le password erano spesso limitate alla lunghezza di otto caratteri, e il numero dei possibili valori casuali era così basso che si potevano facilmente combinare le password più comuni con ogni possibile valore casuale e avere buone possibilità d'individuare una o più password contenute nel file, ottenendo così l'accesso non autorizzato alle utenze corrispondenti. Sono state introdotte estensioni del meccanismo delle password al fine di mantenere segrete le password codificate memorizzandole in un file non accessibile al pubblico; altre estensioni permettono l'uso di password più lunghe o adottano metodi di codifica più sicuri. Sono stati introdotti altri meccanismi di autenticazione che limitano il tempo di collegamento al sistema. Esistono anche meccanismi per trasmettere informazioni di autenticazione a sistemi collegati tramite la rete.

Per affrontare i problemi di autenticazione un certo numero di produttori di sistemi UNIX ha sviluppato un nuovo meccanismo di sicurezza: il sistema **PAM** (*pluggable authentication modules*), basato su una libreria condivisa che può essere usata da ogni componente del sistema che abbia bisogno di autenticare utenti. Una implementazione di questo sistema è disponibile per il sistema Linux. Il sistema PAM permette di caricare su richiesta moduli di autenticazione secondo le indicazioni contenute in un file di configurazione valido per tutto il sistema. Un nuovo meccanismo di autenticazione aggiunto in seguito si può registrare nel file di configurazione in modo che tutti i componenti del sistema possano immediatamente usufruirne. I moduli PAM sono in grado di specificare metodi di autenticazione, limitazioni relative agli account, funzioni di avvio di una sessione di lavoro o funzioni di cambio di password. In quest'ultimo caso, quando un utente cambia la propria password, tutti i necessari meccanismi di autenticazione possono essere aggiornati in un'unica soluzione.

18.11.2 Controllo degli accessi

Nei sistemi UNIX, e così anche nel sistema Linux, il controllo degli accessi è realizzato per mezzo d'identificatori numerici unici. Un identificatore utente (*UID*) individua un singolo utente o un singolo insieme di diritti d'accesso, un identificatore di gruppo (*GID*) è un identificatore aggiuntivo che si può usare per determinare i diritti di più di un utente.

Il controllo degli accessi si applica a vari oggetti del sistema: ogni file disponibile nel sistema è protetto dal meccanismo standard del controllo degli accessi, e ciò vale anche per altri oggetti condivisi come le regioni di memoria condivise e i semafori.

Ogni oggetto in un sistema UNIX sottoposto al controllo degli accessi di singoli utenti o gruppi dispone di un unico UID e un unico GID associati; anche i processi utenti hanno un unico UID, ma possono avere più di un GID. Se l'UID di un processo corrisponde all'UID di un oggetto, quel processo gode dei **diritti utente** o dei **diritti di proprietà** di quell'oggetto; se gli UID non corrispondono ma uno dei GID di un processo corrisponde al GID di un oggetto, il processo gode dei **diritti di gruppo** su quell'oggetto; altrimenti, il processo ha i **diritti generici** (*world rights*) sull'oggetto.

Il sistema Linux esegue il controllo degli accessi assegnando agli oggetti una **maschera di protezione** che specifica quali modi d'accesso – lettura, scrittura o esecuzione – si devono concedere ai processi con diritti d'accesso proprietari, di gruppo o generici. Il proprietario di un oggetto, per esempio, potrebbe avere accesso a un file per la sua lettura, scrittura ed esecuzione; gli utenti di un certo gruppo potrebbero avere accesso per la lettura ma non per la scrittura; e chiunque altro potrebbe non avere alcun diritto d'accesso.

L'unica eccezione a quanto detto è costituita dallo UID privilegiato **root**: un processo dotato di questo speciale UID gode automaticamente dei diritti d'accesso a qualunque oggetto del sistema, scavalcando i normali controlli; processi di questo tipo hanno anche il permesso di eseguire operazioni privilegiate come la lettura di qualunque regione della memoria fisica o aprire socket di rete riservate. Questo meccanismo permette al kernel di impedire agli utenti ordinari di accedere a determinate risorse del sistema: la maggior parte delle risorse chiave interne del kernel sono implicitamente di proprietà dell'UID *root*.

Il sistema Linux adotta il meccanismo standard **setuid** dello UNIX; esso permette a un programma di godere, durante una certa esecuzione, di privilegi diversi da quelli dell'utente che esegue il programma: per esempio, il programma *lpr* che pone un file in coda di stampa ha accesso alle code di stampa del sistema, anche se l'utente che ne richiede l'esecuzione non lo ha. L'implementazione in UNIX di **setuid** distingue fra lo UID *reale* e lo UID *effettivo* di un processo: il primo è quello dell'utente che richiede l'esecuzione del programma; il secondo è quello del proprietario del file.

In Linux questo meccanismo è esteso in due direzioni. In primo luogo implementa il meccanismo **saved user-id** secondo le specifiche POSIX; si tratta di permettere a un processo di perdere e riacquisire ripetutamente il suo UID effettivo. Per ragioni di sicurezza, infatti, un programma potrebbe voler eseguire la maggior parte delle sue operazioni in un modo sicuro, rinunciando ai privilegi conferitigli dal suo **setuid** status, ma potrebbe voler usufruire di tutti i suoi privilegi durante le esecuzioni di alcune operazioni. Le implementazioni standard di UNIX riescono a fornire questo servizio solo scambiando gli UID reali ed effettivi; una volta che ciò è avvenuto, lo UID effettivo precedente viene ricordato, ma lo UID reale del programma non sempre corrisponde allo UID dell'utente che ne richiede l'esecuzione. Il meccanismo **saved user-id** permette a un processo di rendere il suo UID effettivo uguale al suo UID

reale e di ritornare poi al valore precedente del suo UID effettivo senza dover mai modificare il valore dello UID reale.

Il secondo miglioramento apportato dal sistema Linux è l'aggiunta di una caratteristica dei processi che permette di usufruire di un sottoinsieme dei diritti conferiti dallo UID effettivo: le proprietà **fsuid** e **fsgid** di un processo sono usate quando è consentito l'accesso a un file, e sono attive ogniqualsiasi lo UID effettivo o il GID lo sono; tuttavia, fsuid e fsgid possono essere attivate indipendentemente dagli identificatori effettivi, cosa che permette a un processo di accedere ai file per conto di un altro utente senza dover assumere per altri aspetti l'identità di quell'utente. In particolare, i processi server possono impiegare questo meccanismo per fornire file a un certo utente senza divenire suscettibili di terminazione o sospensione da parte di quell'utente.

Linux offre un metodo per il passaggio flessibile dei diritti da un programma a un altro, divenuto comune nelle moderne versioni dello UNIX. Una volta che un collegamento fra due processi del sistema sia stato istituito per mezzo di una socket locale, ognuno dei due processi può inviare all'altro un descrittore di file relativo a uno dei suoi file aperti; l'altro processo riceve quindi un descrittore duplicato dello stesso file: ciò permette a un client di offrire a un processo server l'accesso selettivo a un solo file senza conferirgli alcun altro privilegio. Per esempio, non è più necessario che un server di stampa sia in grado di leggere tutti i file di un utente che chieda una stampa; l'utente, e cioè il client, potrebbe semplicemente comunicare al server i descrittori dei file dei quali richiede la stampa, negando al server la possibilità di accedere agli altri file di sua proprietà.

18.12 Sommario

Linux è un moderno sistema operativo gratuito basato sugli standard UNIX. È stato progettato per essere eseguito efficientemente e in modo affidabile sui comuni PC, ma può anche essere eseguito su diverse altre piattaforme, come i telefoni mobili. Fornisce un'interfaccia per il programmatore e un'interfaccia per l'utente compatibili con i sistemi UNIX standard, e può eseguire moltissime applicazioni UNIX, compreso un crescente numero di applicazioni commerciali.

Il sistema Linux non si è sviluppato dal nulla: nel suo complesso comprende molti componenti originariamente sviluppati in maniera indipendente. La parte centrale del kernel del sistema operativo è interamente originale, ma permette l'esecuzione di molti programmi gratuiti esistenti per il sistema UNIX; ciò rende il tutto un completo sistema operativo compatibile con UNIX e non soggetto a limitazioni legali imposte dagli autori del codice.

Per ragioni legate alle prestazioni il kernel di Linux è realizzato secondo schemi tradizionali come un blocco monolitico di codice, ma è sufficientemente modulare da permettere alla maggior parte dei driver di essere caricati o rimossi dinamicamente.

Il sistema Linux è un sistema multiutente che fornisce meccanismi di protezione dei processi ed è in grado di eseguire più processi in time sharing. Un nuovo processo può condividere selettivamente parti del suo ambiente d'esecuzione con il processo

genitore, permettendo di realizzare la programmazione multithread. La comunicazione fra processi sfrutta sia i meccanismi del System V – code di messaggi, semafori e memoria condivisa – sia l’interfaccia a socket del BSD. È possibile usufruire simultaneamente di più protocolli di rete differenti per mezzo dell’interfaccia a socket.

Il sistema per la gestione della memoria usa la condivisione delle pagine e la copiatura su scrittura per minimizzare la duplicazione dei dati condivisi da processi diversi. Le pagine sono caricate quando si genera un riferimento a esse, e sono trasferite di nuovo in memoria ausiliaria secondo un algoritmo LFU quando è necessario riappropriarsi di regioni della memoria fisica.

Dal punto di vista dell’utente, il file system è un albero di directory conforme alla semantica UNIX; internamente Linux usa uno strato d’astrazione per gestire molti file system differenti. Gestisce file system virtuali, di rete e orientati ai dispositivi. I file system orientati ai dispositivi accedono ai dischi attraverso una cache delle pagine unificata con il sistema per la memoria virtuale.

Esercizi di ripasso

- 18.1** I moduli del kernel caricabili dinamicamente offrono flessibilità quando si aggiungono driver al sistema. Hanno anche qualche svantaggio? In quali circostanze conviene compilare il kernel in un singolo file binario e in quali conviene invece mantenerlo suddiviso in moduli? Spiegate la vostra risposta.
- 18.2** Il multithreading è una tecnica di programmazione comune. Descrivete tre modi diversi per implementare i thread e confrontate questi tre metodi con il meccanismo `clone()` di Linux. Quando un meccanismo alternativo può essere preferibile all’impiego di cloni? Quando può valere il viceversa?
- 18.3** Il kernel di Linux non permette la paginazione della memoria del kernel. Come influisce questa restrizione sul progetto del kernel? Elencate due vantaggi e due svantaggi di questa scelta di progetto.
- 18.4** Discutete tre vantaggi del collegamento dinamico (condiviso) di librerie rispetto al collegamento statico. Descrivete due casi in cui sia preferibile il collegamento statico.
- 18.5** Confrontate l’utilizzo di socket di rete con l’utilizzo di memoria condivisa come meccanismi di comunicazione di dati tra processi in uno stesso computer. Quali sono i vantaggi di ciascun metodo? Quando uno è preferibile rispetto all’altro?
- 18.6** Una volta i sistemi UNIX utilizzavano ottimizzazioni dell’organizzazione del disco basate sulla posizione dei dati rispetto alla rotazione del disco. Le moderne implementazioni, incluso Linux, effettuano invece un’ottimizzazione mirata semplicemente all’accesso sequenziale. Perché è stata fatta questa scelta? Da quali caratteristiche hardware trae vantaggio l’accesso sequenziale? Perché l’ottimizzazione basata sulla rotazione non è più così utile?

Esercizi

- 18.7** Quali vantaggi e svantaggi presenta la scrittura di un sistema operativo con un linguaggio ad alto livello, quale il C?
- 18.8** In quali circostanze la successione di chiamate di sistema `fork()` ed `exec()` è da ritenersi la più appropriata? Quando è preferibile `vfork()`?
- 18.9** Quale tipo di socket dovrebbe essere usato per realizzare un programma per il trasferimento di file tra calcolatori? Quale tipo si dovrebbe scegliere per un programma che effettui un monitoraggio periodico al fine di verificare la presenza di altri elaboratori sulla rete? Motivate le vostre risposte.
- 18.10** Il sistema operativo Linux è eseguibile su diverse piattaforme. Dite che cosa devono fare gli sviluppatori di Linux per assicurare l'adattabilità del sistema a diverse CPU e diverse architetture per la gestione della memoria, e per minimizzare la quantità richiesta di codice del kernel specifico per le singole architetture.
- 18.11** Supponete di restringere l'accesso, da parte dei moduli caricabili del kernel, soltanto ad alcuni dei simboli definiti all'interno del kernel stesso. Quali i vantaggi e gli svantaggi di questa scelta?
- 18.12** Quali obiettivi fondamentali ha il meccanismo di risoluzione dei conflitti, utilizzato dal kernel di Linux per il caricamento dei moduli del kernel?
- 18.13** Discutete come l'operazione `clone()` di Linux sia usata per ottenere comportamenti equivalenti sia a quelli dei processi sia a quelli dei thread.
- 18.14** Attribuireste ai thread di Linux la qualifica di thread a livello utente, o piuttosto quella di thread a livello kernel? Fornite le opportune argomentazioni a sostegno della vostra risposta.
- 18.15** Dite quali sono i costi aggiuntivi implicati dalla creazione e dallo scheduling di un processo rispetto al costo di un thread ottenuto col meccanismo `clone`.
- 18.16** Come può il Completely Fair Scheduler (CFS) di Linux fornire una maggiore equità rispetto a uno scheduler tradizionale UNIX? Quando è garantita questa equità?
- 18.17** Dite quali sono le due variabili configurabili del Completely Fair Scheduler (CFS). Descrivete i pro e i contro di impostare ciascuna di esse a valori molto piccoli e molto grandi.
- 18.18** Lo scheduler di Linux implementa lo scheduling in tempo reale debole (*soft real-time scheduling*); dite quali funzioni, necessarie per certi compiti di programmazione in tempo reale, mancano? Come si potrebbero includere nel kernel e a che costo?
- 18.19** In quali circostanze un processo utente richiederebbe un'operazione il cui esito è l'allocazione di una regione di memoria a valori nulli?

- 18.20** Quali circostanze fanno sì che una pagina di memoria sia mappata nello spazio degli indirizzi di un programma utente con l'opzione di copiatura su scrittura attivata?
- 18.21** Nel sistema Linux le librerie condivise eseguono molte operazioni fondamentali per il sistema operativo. Dite qual è il vantaggio di scorporare dal kernel il codice relativo a questi servizi, ed elencate anche gli eventuali svantaggi. Motivate le risposte.
- 18.22** Quali sono i vantaggi di un file system con journaling come ext3 di Linux? Quali sono i costi? Perché ext3 fornisce la possibilità di effettuare il journaling solo per i metadati?
- 18.23** La struttura della directory nel sistema Linux potrebbe ricomprendere file appartenenti, in senso stretto, a file system diversi: uno di loro è il file system /proc. Quali conseguenze comporta la necessità di ospitare file system di tipo diverso nell'architettura del kernel di Linux?
- 18.24** Per quali aspetti la funzionalità `setuid` di Linux differisce dalla medesima funzionalità dei sistemi SVR4?
- 18.25** Il codice sorgente di Linux è gratuitamente disponibile tramite la rete Internet o presso i rivenditori di CD-ROM. Individuate tre possibili conseguenze riguardanti la sicurezza del sistema Linux.

Note bibliografiche

Il sistema Linux è un prodotto della rete Internet; di conseguenza, la maggior parte della documentazione disponibile su Linux si trova in qualche forma in rete. I seguenti siti chiave rimandano alla maggior parte delle informazioni utili esistenti.

- La *Linux Cross Reference Page* (LXR) all'indirizzo <http://lxr.linux.no/> contiene il codice sorgente aggiornato del kernel di Linux, consultabile tramite il Web e con riferimenti incrociati esaustivi.
- La Kernel Hackers' Guide fornisce un'utile panoramica del kernel Linux. Si trova all'indirizzo <http://tldp.org/LDP/tlk.html>.
- Il Linux Weekly News (LWN) (<http://lwn.net>) fornisce notizie su Linux aggiornate settimanalmente e include una sottosezione molto ben documentata sul kernel Linux.

Inoltre esistono molte mailing list dedicate al sistema Linux; le più importanti sono tenute da un gestore raggiungibile all'indirizzo di posta elettronica majordomo@vger.rutgers.edu. Per ottenere informazioni su come iscriversi a una delle mailing list inviate a tale indirizzo un messaggio contenente la sola parola "help".

Infine, lo stesso sistema Linux si può ottenere tramite la rete Internet. I pacchetti di distribuzione completi si possono ricavare dai siti delle aziende produttrici; la co-

munità Linux mantiene archivi di componenti del sistema nella rete Internet; l'archivio più importante è

- <ftp://linux.kernel.org/pub/linux/>

Oltre a esplorare le risorse in rete il lettore può consultare [Mauerer 2008] e [Love 2010] sul funzionamento interno di Linux.

Bibliografia

[Love 2010] R. Love, *Linux Kernel Development*, 3° Ed., Developer's Library, 2010.

[Mauerer 2008] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons, 2008.

CAPITOLO

19

OBIETTIVI DEL CAPITOLO

- Esplorare i principi alla base della progettazione di Windows 7 e dei suoi componenti.
- Fornire un resoconto dettagliato del file system di Windows 7.
- Illustrare i protocolli di rete supportati da Windows 7.
- Descrivere l'interfaccia disponibile per i programmatore di sistema e di applicazioni.
- Descrivere gli algoritmi principali implementati per Windows 7.

Windows 7

Aggiornamento a cura di Dave Probert

Microsoft Windows 7 è un sistema operativo client multitasking a 32 o 64 bit con prelazione sviluppato per i microprocessori che implementano i set di istruzioni delle architetture Intel IA-32 e AMD64. Il corrispondente sistema operativo server di Microsoft, Windows Server 2008 R2, si basa sullo stesso codice di Windows 7, ma supporta solo le architetture a 64 bit AMD64 e IA64 (Itanium). Windows 7 è l'ultimo di una serie di sistemi operativi Microsoft basati sul codice NT, che ha sostituito i precedenti sistemi basati su Windows 95/98. In questo capitolo sono trattati gli obiettivi chiave del sistema, la sua architettura a strati che lo rende così facile da usare, il file system, le caratteristiche di networking e l'interfaccia di programmazione.

19.1 Storia

Nella metà degli anni '80 Microsoft e IBM cooperarono per sviluppare il sistema operativo OS/2, scritto in linguaggio assembly per sistemi a singola CPU Intel 80286. Nel 1988 Microsoft decise di abbandonare la collaborazione con IBM e sviluppare un proprio sistema operativo facilmente adattabile dalla "nuova tecnologia" (NT), che includeva le interfacce per la programmazione delle applicazioni OS/2 e POSIX. Nell'ottobre del 1988 Dave Cutler, progettista del sistema operativo DEC VAX/VMS, fu assunto e incaricato della progettazione e realizzazione di questo nuovo sistema operativo.

Originariamente per il sistema Windows NT si sarebbe dovuta adottare l'API del sistema operativo OS/2 come suo ambiente naturale, ma durante lo sviluppo si scelse di adottare una nuova API a 32 bit (Win32), basata sulla diffusa API a 16 bit di Windows 3.0. Le prime versioni del sistema Windows NT furono Windows NT 3.1 e Windows NT 3.1 Advanced Server (in quel periodo, la più recente versione dell'ambiente Windows a 16 bit era la 3.1). Nella versione 4.0 il sistema Windows NT adottò l'interfaccia utente di Windows 95 e incorporò il software del server Web e il browser; inoltre le procedure dell'interfaccia utente e il codice per la grafica furono spostati all'interno del kernel al fine di migliorare le prestazioni, con l'effetto collaterale di ridurre l'affidabilità del sistema.

Nonostante le prime versioni del sistema NT fossero state adattate per altre architetture, Windows 2000 (uscito nel febbraio 2000) ha limitato la portabilità ai soli processori Intel e compatibili, per ragioni di mercato. Rispetto a Windows NT, Windows 2000 vantava considerevoli innovazioni, quali: aggiunta della Active Directory (un servizio di directory basato sullo standard X.500), migliore gestione dei servizi di rete, supporto ai dispositivi *plug-and-play*, un file system distribuito, la possibilità di aumentare le unità di elaborazione ed espandere la memoria.

Nell'ottobre 2001 l'uscita di Windows XP ha costituito l'evoluzione del sistema operativo desktop Windows 2000 e, nel contempo, come sostituzione di Windows 95/98. Nel 2002 venivano licenziate le versioni server di Windows XP, con il nome di Windows .NET Server. Windows XP ha dotato l'interfaccia utente (GUI) di un aspetto grafico rinnovato, a cui si aggiungono *nuove funzionalità di uso intuitivo*. Numerose funzionalità sono state introdotte per risolvere in modo automatico i problemi delle applicazioni, e del sistema operativo stesso. Windows XP può contare su una migliore gestione delle connessioni di rete e dei dispositivi (fra cui configurazione automatica delle connessioni wireless, un servizio di messaggistica istantanea, supporto multimediale e applicazioni digitali fotografiche e video), su un notevolissimo aumento delle prestazioni, sia per i PC desktop sia per sistemi multiprocessore più potenti, e su accresciute garanzie di affidabilità e sicurezza.

L'aggiornamento tanto atteso di Windows XP, chiamato Windows Vista, è stato rilasciato nel novembre 2006, ma non è stato ben accolto. Anche se Windows Vista includeva molti miglioramenti che più tardi sono stati riproposti in Windows 7, questi sono stati offuscati dai problemi di lentezza e di compatibilità percepiti nell'utilizzo

di Windows Vista. Microsoft ha risposto alle critiche al sistema Vista migliorando i propri processi di ingegnerizzazione e lavorando a stretto contatto con i produttori di hardware e applicazioni per Windows. Il risultato è stato **Windows 7**, rilasciato nell'ottobre del 2009 insieme alle corrispondenti versioni server di Windows. Tra le modifiche ingegneristiche più significative vi è l'aumento dell'uso della **tracciatura di esecuzione** (*execution tracing*) per l'analisi del comportamento del sistema, al posto dell'utilizzo di contatori o profiling. Il tracing permane in esecuzione nel sistema, monitorando centinaia di scenari di esecuzione. Quando uno di questi scenari fallisce, o quando ha successo ma non ottiene buone prestazioni, è possibile analizzare i dati raccolti per determinare la causa del problema.

Windows 7 si serve di un'architettura client-server (come quella di Mach) per implementare due personalità del sistema operativo, Win32 e POSIX, tramite processi a livello utente chiamati sottosistemi. Un tempo Windows supportava anche un sottosistema OS/2, ma il supporto è stato rimosso in Windows XP a causa della scomparsa di OS/2. L'architettura del sottosistema permette di apportare miglioramenti a una singola personalità del sistema operativo senza compromettere la compatibilità con le applicazioni dell'altra. Anche se in Windows 7 continua a essere disponibile il sottosistema POSIX, l'API Win32 ha visto una grande diffusione e le API POSIX vengono utilizzate solo in pochi casi. Lo studio dell'approccio a sottosistemi continua a essere interessante dal punto di vista del sistema operativo, ma le tecnologie di virtualizzazione stanno diventando il metodo dominante per eseguire più sistemi operativi su una singola macchina.

Windows 7 è un sistema operativo multiutente, che permette accessi simultanei tramite servizi distribuiti, o tramite istanze multiple dell'interfaccia grafica (GUI) gestite dal terminal server del sistema. Le versioni server di Windows 7 permettono più sessioni contemporanee di connessione al server da parte dei sistemi desktop Windows. Le versioni desktop, invece, implementano sessioni virtuali per ogni utente collegato, applicando a tastiera, mouse e monitor la tecnica del multiplex. Questa caratteristica, chiamata *commutazione veloce dell'utente*, permette agli utenti di gestire a turno il controllo del PC senza dover uscire e rientrare nel sistema.

Abbiamo sottolineato in precedenza che qualche applicazione GUI in Windows NT 4.0 è stata spostata in modalità kernel. Il ritorno alla modalità utente è iniziato con Windows Vista, che includeva il **DWM (desktop window manager)** come processo in modalità utente. DWM implementa la composizione del desktop di Windows, fornendo l'aspetto dell'interfaccia Windows **Aero**, e si colloca al di sopra del software grafico DirectX. DirectX di Windows è ancora eseguito in modalità kernel, così come il codice che implementa la precedente interfaccia e i precedenti modelli grafici di Windows (Win32k e GDI). Windows 7 ha apportato modifiche sostanziali al DWM, riducendo in modo significativo la sua occupazione di memoria e migliorandone le prestazioni.

Windows XP è stata la prima versione di Windows a 64 bit (dal 2001 per IA64 e dal 2005 per AMD64). Il file system nativo di NT (NTFS) e molte delle API Win32 hanno sempre usato numeri interi a 64 bit, quando necessario; pertanto, l'estensione a

64 bit di Windows XP serviva principalmente per il supporto di indirizzi virtuali estesi. Le edizioni a 64 bit di Windows, tuttavia, supportano anche memorie fisiche molto più grandi. Quando Windows 7 è stato rilasciato l'ISA AMD64 era diventato disponibile su quasi tutte le CPU, sia di Intel sia di AMD. Inoltre, allo stesso tempo, le memorie fisiche sui sistemi client iniziarono a superare spesso il limite di 4 GB imposto dall'architettura IA-32. Per queste ragioni la versione a 64 bit di Windows 7 è ormai comunemente installata sui sistemi client di buon livello. Poiché l'architettura AMD64 è fedelmente compatibile con IA-32 a livello dei singoli processi, in un unico sistema è possibile utilizzare liberamente applicazioni a 32 e 64 bit.

Nel resto della nostra descrizione di Windows 7 non faremo distinzione tra le versioni client e le corrispondenti versioni server. Le diverse versioni si basano sugli stessi componenti di base ed eseguono gli stessi file binari per il kernel e la maggior parte dei driver. Analogamente, anche se Microsoft distribuisce diverse edizioni di ogni release per essere presente sul mercato con diverse fasce di prezzo, sono poche le differenze che si riflettono sulla parte centrale del sistema. In questo capitolo ci concentreremo principalmente sulle componenti fondamentali di Windows 7.

19.2 Princìpi di progettazione

Gli obiettivi di progettazione dichiarati da Microsoft per il sistema operativo Windows includono sicurezza, affidabilità, compatibilità con le applicazioni Windows e POSIX, alte prestazioni e adattamento a livello internazionale. Alcuni ulteriori obiettivi quali l'efficienza energetica e il supporto dinamico dei dispositivi sono stati recentemente aggiunti a questa lista. Nel seguito discutiamo ciascuno di questi obiettivi e vediamo come essi vengano raggiunti in Windows 7.

19.2.1 Sicurezza

Gli obiettivi di **sicurezza** di Windows 7 richiedevano qualcosa in più della semplice uniformità agli standard progettuali che hanno permesso a Windows NT 4.0 di ricevere la classificazione di sicurezza C2 dal governo degli Stati Uniti d'America (la classificazione C2 indica un moderato livello di protezione nei confronti di programmi difettosi e di attacchi malevoli; Queste classificazioni sono state definite dal Dipartimento della Difesa nel “Trusted Computer System Evaluation Criteria”, noto anche come **Orange Book**, come descritto nel Capitolo 15, Paragrafo 15.8.) Un'approfondita revisione del codice e attività di testing sono state combinate con strumenti di analisi automatizzati per identificare e studiare potenziali difetti, dietro i quali potrebbero annidarsi falle di sicurezza.

Windows basa la sicurezza sul controllo discrezionale degli accessi. Gli oggetti di sistema, inclusi i file, le impostazioni del registro di sistema e gli oggetti del kernel, sono protetti da liste di controllo degli accessi (ACL) (si veda il Capitolo 11, Paragrafo 11.6.2). Le ACL sono però vulnerabili a errori di utenti e programmatore e ai più comuni attacchi sui sistemi consumer, in cui l'utente viene indotto in maniera inganne-

vole a eseguire codice, spesso durante la navigazione sul Web. Windows 7 utilizza i livelli di integrità come elementare sistema di capability per il controllo degli accessi. Agli oggetti e ai processi viene assegnata un'integrità bassa, media o alta e Windows non consente a un processo di modificare un oggetto con un livello di integrità più alto, indipendentemente dalle informazioni contenute nella ACL.

Tra le altre misure di sicurezza vi sono l'**ASLR** (*address-space layout randomization*), lo stack e l'heap non eseguibili e i servizi di crittografia e firma digitale. ASLR ostacola molte forme di attacco impedendo a piccole quantità di codice di essere facilmente iniettate nel codice già caricato in un processo durante la normale esecuzione. Questo meccanismo di protezione fa sì che un sistema sotto attacco sarà probabilmente soggetto a errori o crash, ma non lascerà che il codice malevolo prenda il controllo.

I chip recenti di Intel e AMD sono basati sull'architettura AMD64 che consente di contrassegnare pagine di memoria in modo che non possano contenere codice eseguibile. Windows tenta di contrassegnare stack e heap di memoria in modo che non possano essere utilizzati per eseguire codice, impedendo attacchi in cui un bug di un programma permette un buffer overflow ed è indotto poi a eseguire il contenuto del buffer. Questa tecnica non può essere applicata a tutti i programmi, perché alcuni si basano sulla modifica dei dati e la loro esecuzione. Una colonna denominata "Data Execution Prevention" nel task manager di Windows mostra quali processi sono contrassegnati per la prevenzione da questi attacchi.

Windows utilizza la crittografia all'interno dei comuni protocolli, per esempio dei protocolli utilizzati per comunicare in modo sicuro con i siti web. La crittografia è usata anche per proteggere da occhi indiscreti i file degli utenti memorizzati su disco. Windows 7 consente agli utenti di crittografare facilmente anche un intero disco, oltre ai dispositivi di memorizzazione rimovibili come le unità USB di memoria flash, per mezzo di una funzione chiamata **BitLocker**. Se viene rubato un computer con un disco criptato, i ladri avranno bisogno di utilizzare tecnologie molto sofisticate (come un microscopio elettronico) per ottenere l'accesso a qualsiasi file. Windows utilizza firme digitali sui file binari del sistema operativo in modo da poter verificare che i file siano stati prodotti da Microsoft o da un'altra società nota. In alcune edizioni di Windows viene attivato all'avvio un modulo di integrità del codice per garantire che tutti i moduli caricati nel kernel abbiano firme valide, assicurando che questi non siano stati alterati da attacchi off-line.

19.2.2 Affidabilità

Il sistema operativo Windows è cresciuto molto nei suoi primi dieci anni e questa maturazione ha portato al sistema Windows 2000. Allo stesso tempo è aumentata la sua affidabilità, grazie a fattori quali la maturità del codice sorgente, i numerosi test di funzionamento sotto stress, il miglioramento delle architetture delle CPU e il rilevamento automatico di molti gravi errori nei driver di Microsoft e di terze parti. Windows ha successivamente esteso gli strumenti per ottenere maggiore affidabilità includendo l'analisi automatica del codice sorgente per la rilevazione di errori, i test

in grado di fornire parametri di ingresso non validi o imprevisti (noti come **fuzzing**) utili per rilevare gli errori di validazione dell'input e una versione applicativa del verificatore di driver che applica il controllo dinamico su una vasta gamma di errori comuni di programmazione in modalità utente. Ulteriori miglioramenti in termini di affidabilità sono dovuti allo spostamento di una quantità maggiore di codice dal kernel ai servizi in modalità utente. Windows fornisce un ampio supporto per lo sviluppo di driver in modalità utente e diversi servizi di sistema che, una volta, erano nel kernel sono ora eseguiti in modalità utente: tra questi vi sono il DWM e gran parte del software per l'audio.

Uno dei miglioramenti più significativi introdotti grazie all'esperienza nei sistemi Windows consisteva nell'aggiunta dell'opzione di diagnostica della memoria all'avvio, particolarmente preziosa perché ben pochi PC consumer hanno una memoria con correzione d'errore. Quando una RAM fallata inizia a perdere bit il risultato è un frustrante comportamento irregolare nel sistema. La disponibilità della diagnostica della memoria ha notevolmente ridotto i livelli di stress degli utenti in possesso di RAM scadenti.

Windows 7 ha introdotto una memoria heap con tolleranza agli errori. L'heap apprende dai crash di un'applicazione e inserisce in maniera automatica alcune forme di mitigazione nelle sue future esecuzioni, rendendo così l'applicazione più affidabile anche se contiene bug comuni come l'utilizzo di memoria già liberata o l'accesso alla memoria oltre il limite dell'allocazione.

Il raggiungimento di un'elevata affidabilità in Windows è particolarmente impegnativo, perché quasi un miliardo di computer esegue questo sistema operativo. Persino i problemi di affidabilità che interessano solo una piccola percentuale di utenti possono impattare su un numero altissimo di persone. La complessità dell'ecosistema Windows aggiunge ulteriori ostacoli. Milioni di applicazioni diverse, driver e altri software sono costantemente scaricati ed eseguiti su sistemi Windows e naturalmente c'è anche un flusso costante di attacchi malware. Dato che Windows è diventato più difficile da attaccare direttamente, gli attacchi prendono sempre più di mira le applicazioni più diffuse.

Per far fronte a queste problematiche, Microsoft fa sempre più affidamento sulle comunicazioni provenienti dalle macchine dei clienti per raccogliere grandi quantità di dati provenienti da tutto l'ecosistema. I dati possono essere campionati per avere informazioni sulle prestazioni che le macchine ottengono, su quali software sono in esecuzione e sui problemi che si incontrano. I clienti possono inviare dati a Microsoft in caso di crash o blocchi delle applicazioni o del sistema. Questo flusso costante di dati proveniente dalle macchine viene raccolto con molta attenzione, con il consenso degli utenti e senza invadere la privacy. Di conseguenza Microsoft va costruendosi un'immagine sempre più chiara di quello che succede nell'ecosistema di Windows e ciò consente continui miglioramenti tramite aggiornamenti software e fornisce le informazioni che guidano lo sviluppo delle versioni future di Windows.

19.2.3 Compatibilità tra applicazioni Windows e POSIX

Come sottolineato in precedenza, Windows XP non è stato soltanto un aggiornamento di Windows 2000: è il sistema che sostituisce Windows 95/98. Windows 2000 era studiato principalmente per la compatibilità con le applicazioni della clientela professionale, mentre Windows XP prevede una compatibilità molto più elevata con le applicazioni di largo consumo eseguibili in Windows 95/98. La **compatibilità delle applicazioni** è difficile da ottenere, in quanto molte applicazioni verificano preliminarmente una specifica versione di Windows, risentono delle peculiarità nelle implementazioni delle API, possono nascondere bachi latenti che non si manifestavano nel sistema precedente, e così via. Le applicazioni possono inoltre essere state compilate per un set di istruzioni differente. Windows 7 implementa diverse strategie per eseguire le applicazioni nonostante l'incompatibilità.

Come Windows XP, Windows 7 ha uno strato di compatibilità che si trova fra le applicazioni e le API di Win32: grazie a tale strato, Windows XP appare compatibile con le precedenti versioni di Windows, addirittura quasi baco per baco (*bug-for-bug*). Windows XP, come il suo predecessore NT, mantiene il supporto per molte applicazioni a 16 bit usando uno strato di conversione (o *thunking*) che traduce le chiamate delle API a 16 bit in chiamate equivalenti a 32 bit. Analogamente, la versione a 64 bit di Windows XP fornisce uno strato che converte le chiamate delle API a 32 bit in chiamate native a 64 bit.

Il modello a sottosistemi di Windows consente il supporto di personalità multiple del sistema operativo. Come osservato in precedenza, anche se l'API più comunemente usata in Windows è l'API Win32, alcune edizioni di Windows 7 supportano un sottosistema POSIX. POSIX è una specifica standard per UNIX che permette alla maggior parte del software compatibile con UNIX di essere compilato ed eseguito senza alcuna modifica.

Come ulteriore strumento di compatibilità, varie edizioni di Windows 7 forniscono una macchina virtuale che esegue Windows XP all'interno di Windows 7, in modo da consentire alle applicazioni di ottenere una compatibilità con Windows XP a livello di bug.

19.2.4 Elevate prestazioni

Windows è stato progettato per fornire elevate **prestazioni** nei sistemi desktop (fortemente condizionati dalle prestazioni dell'I/O), nei sistemi server (dove la CPU è spesso il collo di bottiglia) e nei grandi ambienti multithread e multiprocessore (in cui la gestione dei lock e delle cache è uno snodo cruciale per la scalabilità). Per mantenere alto il rendimento delle prestazioni, NT utilizzava svariate tecniche, quali esecuzione asincrona dell'I/O, protocolli ottimizzati per le reti, grafica basata sul kernel e sofisticate tecniche di gestione della cache del file system. La progettazione degli algoritmi di gestione della memoria e di sincronizzazione è concepita tenendo presenti le prestazioni legate alle cache e ai multiprocessori.

Windows NT è stato progettato per il multiprocessing simmetrico (SMP). In un computer multiprocessore, più thread possono contemporaneamente essere in esecu-

zione, anche nel kernel. Windows NT utilizza su ogni CPU uno scheduling dei thread basato su priorità con prelazione. A eccezione dei thread in esecuzione nel dispatcher del kernel o a livello di interrupt, i thread di qualsiasi processo in esecuzione su Windows possono subire la prelazione da thread a più alta priorità. Il sistema, quindi, è in grado di fornire risposte rapide (si veda il Capitolo 6).

I sottosistemi che costituiscono Windows NT comunicano tra loro in modo efficiente mediante le **chiamate di procedura locali** (*local procedure call*, LPC), che garantiscono lo scambio di messaggi ad alte prestazioni. Quando un thread richiede un servizio sincrono da un altro processo attraverso una LPC, il thread che offre il servizio viene contrassegnato come pronto (ready) e la sua priorità viene temporaneamente aumentata per evitare i ritardi di scheduling che si verificherebbero se dovesse attendere thread già in coda.

Windows XP ha ulteriormente migliorato le prestazioni riducendo la lunghezza del codice nelle funzioni critiche, adottando algoritmi migliori e strutture dati dedicate al singolo processore, usando la “colorazione” della memoria per **macchine NUMA** (*non-uniform memory access*, “accesso non uniforme alla memoria”) e implementando protocolli di gestione dei lock scalabili, come le code di spinlock. I nuovi protocolli di locking contribuiscono a ridurre i cicli di bus del sistema; comprendono liste e code prive di lock, l’uso di operazioni atomiche read-modify-write (come l’incremento interallacciato – *interlocked increment* –) e altre tecniche avanzate di sincronizzazione.

Quando Windows 7 è stato sviluppato sono state approntate diverse modifiche sul versante della computazione. La computazione client/server aveva nel frattempo assunto maggiore importanza ed è stata dunque introdotta una chiamata di procedura locale avanzata (ALPC) per fornire prestazioni più elevate e maggiore affidabilità rispetto alla LPC. Il numero di CPU e la quantità di memoria fisica disponibile sui sistemi più grandi erano aumentati notevolmente: numerosi sforzi sono stati dunque rivolti al miglioramento della scalabilità del sistema operativo.

L’implementazione di SMP in Windows NT utilizzava bitmask per rappresentare collezioni di processori e per individuare, per esempio, su quale insieme di processori poteva essere eseguito un particolare thread. Le maschere erano state progettate per occupare una singola parola di memoria, limitando a 64 il numero di processori supportati all’interno di un sistema. Windows 7 ha aggiunto il concetto di **gruppi di processori** per rappresentare un numero arbitrario di CPU, in modo da poterne ospitare una quantità superiore. Il numero di CPU all’interno dei singoli sistemi ha continuato ad aumentare non solo grazie alla presenza di più core, ma anche per il fatto che i core supportano più di un thread logico in esecuzione alla volta.

Tutte queste CPU aggiuntive hanno portato a una maggior contesa per i lock utilizzati per lo scheduling di CPU e memoria. Windows 7 ha differenziato questi lock. Per esempio, prima di Windows 7 lo scheduler di Windows utilizzava un singolo lock per sincronizzare l’accesso alle code contenenti thread in attesa di eventi. In Windows 7 ogni oggetto ha un proprio lock, permettendo un accesso concorrente alle code. Inoltre, molti percorsi di esecuzione dello scheduler sono stati riscritti e resi privi di

lock. Questo cambiamento ha portato a buone prestazioni in termini di scalabilità anche su sistemi con 256 thread hardware.

Altre modifiche sono dovute alla crescente importanza del supporto al calcolo parallelo. Per anni l'industria informatica è stata dominata dalla legge di Moore che ha portato a una maggiore densità dei transistor che a loro volta consentono frequenze di clock più alte per ogni CPU. La legge di Moore continua a valere, ma ormai sono stati raggiunti i limiti che impediscono alla frequenza di clock della CPU di crescere ulteriormente. I transistor vengono ora utilizzati per costruire sempre più CPU all'interno di ogni chip. Nuovi modelli di programmazione per realizzare il parallelismo, come Concurrency RunTime di Microsoft (ConcRT) e Threading Building Blocks di Intel (TBB), vengono utilizzati per implementare il parallelismo nei programmi C++. Anche se la legge di Moore ha governato l'informatica per 40 anni, sembra ora che la legge di Amdahl, che disciplina il calcolo parallelo, sia destinata a dominare il futuro.

Per supportare il parallelismo a livello di task, Windows 7 fornisce una nuova forma di **scheduling in modalità utente** (UMS) che permette ai programmi di essere scomposti in task poi pianificati sulle CPU disponibili da uno scheduler che opera in modalità utente anziché nel kernel.

L'utilizzo di più CPU sui piccoli computer è solo una parte del cambiamento che sta avvenendo nel calcolo parallelo. Le **unità di elaborazione grafica** (GPU) accelerano gli algoritmi di calcolo necessari per la grafica utilizzando architetture SIMD per eseguire una singola istruzione su più dati allo stesso tempo. Tale potenzialità ha condotto all'utilizzo delle GPU per calcoli generici e non solo per la grafica. Il supporto del sistema operativo a software come OpenCL e CUDA sta permettendo ai programmi di sfruttare le GPU. Windows supporta l'uso delle GPU attraverso il software presente nel suo supporto grafico DirectX. Questo software, chiamato DirectCompute, permette ai programmi di specificare nuclei computazionali utilizzando lo stesso modello di programmazione **HLSL** (*high-level shader language*) utilizzato per programmare l'hardware SIMD per **shader grafici**. I nuclei computazionali vengono eseguiti molto velocemente sulla GPU e restituiscono i loro risultati alla computazione principale in esecuzione sulla CPU.

19.2.5 Estendibilità

L'**estendibilità** si riferisce alla capacità di un sistema operativo di tenere il passo con gli sviluppi della tecnologia informatica. Il sistema Windows ha una struttura stratificata che facilita eventuali cambiamenti nel corso del tempo. Il suo modulo executive, che esegue in modalità kernel, fornisce i servizi di sistema fondamentali e le astrazioni a supporto di un uso condiviso del sistema; sopra l'executive operano molti sottosistemi server eseguiti in modalità utente, fra i quali i **sottosistemi d'ambiente** che simulano differenti sistemi operativi: in questo modo, un programma scritto per Win32 o POSIX può essere eseguito dal sistema operativo nell'ambiente appropriato. Grazie alla struttura modulare è possibile aggiungere nuovi sottosistemi d'ambiente senza che ciò abbia ripercussioni sull'executive; inoltre, il sistema Windows all'interno del si-

stema di I/O usa driver caricabili, cosicché si possano aggiungere nuovi file system e nuovi tipi di dispositivi di I/O o di rete mentre il sistema è attivo. Il sistema Windows, come il Mach, adotta un modello client-server e gestisce l'elaborazione distribuita per mezzo di chiamate di procedure remote (RPC) secondo le specifiche dell'Open Software Foundation.

19.2.6 Portabilità

Un sistema operativo è **portabile** (*portable*) se per poter essere eseguito in un'altra architettura richiede un numero di modifiche relativamente piccolo; Windows è progettato per essere portabile. Così come per il sistema operativo UNIX, la maggior parte del sistema Windows è scritta in linguaggio C o in C++.

Il codice sorgente specifico per l'architettura è relativamente piccolo e si fa molto poco uso di codice assembly. Il porting di Windows su una nuova architettura riguarda soprattutto il kernel, dato che il codice in modalità utente viene scritto quasi esclusivamente per essere indipendente dall'architettura. Per il porting di Windows deve essere effettuato il porting del codice del kernel specifico per l'architettura e, a causa dei cambiamenti nelle principali strutture dati, come il formato della tabella delle pagine, è talvolta necessaria una compilazione condizionale in altre parti del kernel. L'intero sistema Windows deve poi essere ricompilato per il nuovo set di istruzioni della CPU.

I sistemi operativi sono sensibili non solo all'architettura della CPU, ma anche ai chip di supporto alla CPU e ai programmi di boot. L'insieme formato da CPU e chip di supporto è noto complessivamente come **chipset**. I chipset e il codice di avvio a essi associato determinano la modalità di emissione degli interrupt, descrivono le caratteristiche fisiche di ogni sistema e forniscono interfacce per gli aspetti più profondi dell'architettura della CPU, come la gestione degli errori e dell'alimentazione. Sarebbe davvero oneroso effettuare il porting di Windows su ciascun tipo di chip di supporto e su ogni architettura di CPU. Per ovviare a questo problema, Windows isola la maggior parte del codice dipendente dal chipset in una libreria dinamica (DLL) che costituisce lo **strato di astrazione dell'hardware (HAL)** e che viene caricata con il kernel. Il kernel di Windows dipende dalle interfacce HAL piuttosto che dai dettagli del chipset sottostante. Ciò permette di utilizzare il singolo insieme di file binari del kernel e dei driver per una particolare CPU con diversi chipset semplicemente caricando una versione diversa dell'HAL.

Nel corso degli anni Windows è stato portato su una serie di diverse architetture di CPU: CPU a 32 bit compatibili con IA-32 di Intel, CPU a 64 bit compatibili con IA64 o AMD64, DEC Alpha, MIPS e PowerPC. La maggior parte di queste architetture è scomparsa dal mercato e quando Windows 7 è stato rilasciato solo le architetture IA-32 e AMD64 sono state supportate sui computer client e le architetture AMD64 e IA64 sui server.

19.2.7 Funzionalità di adattamento internazionale

Windows è anche stato concepito per un'**utenza internazionale**. Si adatta alle lingue locali grazie all'API per la **gestione delle lingue nazionali** (*national language support, NLS*), che fornisce procedure specializzate per trattare date, orari e valute in accordo agli usi dei diversi paesi. I confronti fra sequenze di caratteri tengono conto dei diversi alfabeti. Il codice dei caratteri proprio di Windows è UNICODE, i caratteri ANSI sono convertiti in UNICODE (da 8 a 16 bit) prima della manipolazione. Le stringhe contenenti informazioni di sistema si trovano in file sostituibili per adattare il sistema a lingue diverse. È anche possibile impiegare contemporaneamente versioni per lingue differenti, il che è importante per gli utenti e le aziende multilingue.

19.2.8 Efficienza energetica

Aumentare l'efficienza energetica dei computer fa sì che le batterie di computer portatili e netbook durino più a lungo, permette di risparmiare notevolmente sui costi di esercizio e di raffreddamento dei data center e contribuisce a iniziative ecologiche volte a ridurre il consumo di energia da parte delle imprese e dei consumatori. Da qualche tempo Windows ha messo in atto diverse strategie per ridurre il consumo di energia. Quando possibile, le CPU vengono impostate in uno stato di potenza inferiore, per esempio abbassando la frequenza di clock. Inoltre, quando un computer non viene attivamente utilizzato Windows può mettere l'intero sistema in uno stato di basso consumo (sleep) o persino salvare tutta la memoria su disco e spegnere il computer (sospensione). Al ritorno dell'utente, il computer si accende e continua dal suo stato precedente, senza necessità di riavviare sistema e applicazioni.

Windows 7 ha aggiunto alcune nuove strategie per il risparmio energetico, che cresce proporzionalmente al tempo in cui la CPU resta inutilizzata: poiché i computer sono molto più veloci degli esseri umani, può essere risparmiata una notevole quantità di energia mentre gli esseri umani sono fermi a pensare. Il problema è che diversi programmi interrogano costantemente il sistema (polling) per vedere che cosa stia succedendo e tali interrogazioni periodiche impediscono alla CPU di restare inattiva abbastanza a lungo da permettere un risparmio effettivo. Windows 7 è in grado di aumentare il tempo di inattività della CPU ignorando gli impulsi di clock, accorpendo i timer dei software in un numero inferiore di eventi e “parcheggiando” intere CPU quando il sistema non è troppo carico.

19.2.9 Supporto dinamico dei dispositivi

All'inizio della storia dell'industria dei PC le configurazioni di un computer erano abbastanza statiche; al massimo potevano essere occasionalmente collegati dei nuovi dispositivi alle porte seriali o alle porte per stampante o di gioco presenti sul retro del PC. I primi passi verso la configurazione dinamica dei PC furono i dock per computer portatili e le schede PCMCIA: un PC poteva improvvisamente essere collegato o scollegato da tutta una serie di periferiche. In un PC moderno la situazione è completamente diversa. I PC sono progettati per consentire agli utenti di collegare e scollegare

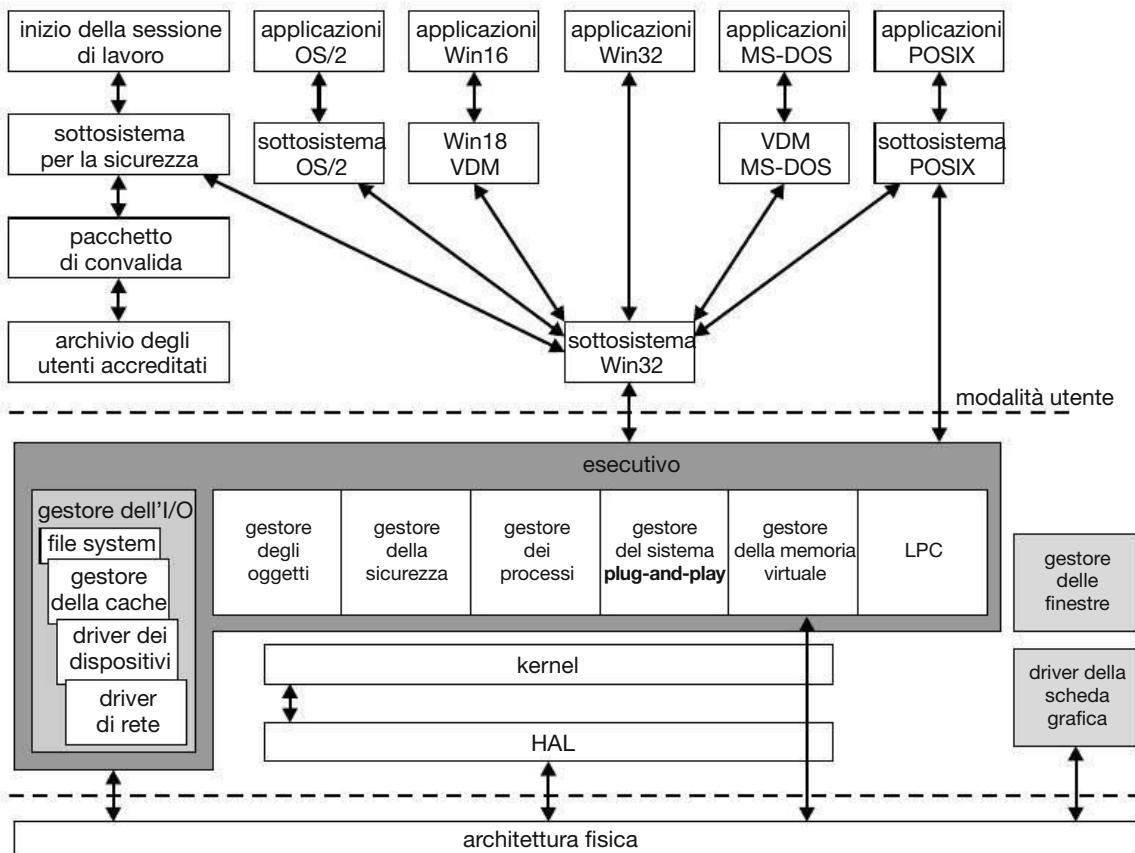


Figura 19.1 Schema a blocchi del sistema operativo Windows.

una vasta gamma di periferiche in qualsiasi momento. Dischi esterni, chiavette USB, fotocamere e dispositivi simili vengono continuamente inseriti e disinseriti.

Nel sistema Windows il supporto per la configurazione dinamica dei dispositivi è in continua evoluzione. Il sistema è in grado di riconoscere automaticamente i dispositivi quando sono collegati e può trovare, installare e caricare i driver appropriati spesso senza l'intervento dell'utente. Quando i dispositivi vengono scollegati i driver sono automaticamente scaricati e l'esecuzione del sistema continua senza interrompere altri software.

19.3 Componenti del sistema

La struttura del sistema operativo Windows è un sistema stratificato di moduli (Figura 19.1). Gli strati principali sono l'HAL, il kernel e l'executive, operanti in modo kernel, e un'ampia raccolta di sottosistemi eseguiti in modalità utente; questi ultimi si dividono in due categorie: i sottosistemi d'ambiente, che simulano sistemi operativi diversi, e i **sottosistemi di protezione**, che forniscono servizi di sicurezza. Uno dei vantaggi principali di questo tipo d'organizzazione è che le interazioni fra i moduli sono semplificate. Il resto di questo paragrafo descrive i diversi strati e sottosistemi.

19.3.1 Strato di astrazione dell'hardware

Lo strato di astrazione dell'hardware (HAL) è uno strato software che nasconde agli strati superiori le differenze presenti nel chipset, contribuendo alla portabilità del sistema operativo. Lo HAL realizza una macchina virtuale usata come interfaccia dal dispatcher del kernel, dall'executive e dai driver dei dispositivi; è sufficiente una sola versione di ogni driver per ogni architettura della CPU, indipendentemente dai chip di supporto utilizzati. I driver mappano i dispositivi e vi accedono direttamente, ma i dettagli specifici del chipset legati alla mappatura in memoria, alla configurazione dei bus per l'I/O, al DMA, alla gestione di funzionalità specifiche della scheda madre, sono tutti curati dalle interfacce HAL.

19.3.2 Kernel

Il livello kernel di Windows ha quattro compiti principali: lo scheduling dei thread, la gestione delle interruzioni e delle eccezioni, la sincronizzazione a basso livello della CPU e il passaggio tra la modalità utente e la modalità kernel. Il kernel è implementato in linguaggio C, con l'utilizzo del linguaggio assembly solo nei casi in cui è assolutamente necessario per interfacciarsi con i livelli più bassi dell'architettura.

Il kernel è orientato agli oggetti: un **tipo di oggetto** è un tipo di dato definito dal sistema che ha un insieme di attributi (valori dei dati) e possiede un insieme di metodi (cioè, funzioni o operazioni). Un **oggetto** è semplicemente un'istanza di uno specifico tipo d'oggetto. Il kernel svolge i suoi compiti usando un insieme d'oggetti i cui attributi memorizzano i dati necessari al kernel, e i cui metodi eseguono le attività del kernel.

19.3.2.1 Dispatcher del kernel

Il dispatcher del kernel costituisce il fondamento per l'executive e i sottosistemi. La gran parte del dispatcher non è mai rimossa dalla memoria centrale, e la sua esecuzione non è mai soggetta a prelazione. I suoi compiti principali sono lo scheduling dei thread, il cambio di contesto, l'implementazione delle primitive di sincronizzazione, la gestione del timer, delle interruzioni software (chiamate di procedura asincrone e differite), e la consegna (*dispatching*) delle eccezioni.

19.3.2.2 Thread e scheduling

Come molti altri moderni sistemi operativi, il sistema Windows associa le nozioni di processo e thread al codice eseguibile. Ogni processo ha uno o più thread, cioè unità d'esecuzione elaborate dal kernel; ogni thread possiede un suo stato, che comprende una priorità, dei gradi di affinità relativi alle unità d'elaborazione, e informazioni sull'uso della CPU.

I sei possibili stati di un thread sono: pronto, in standby, in esecuzione, in attesa, in transizione, e terminato. Un thread è **pronto** quando attende d'essere eseguito. Il thread con priorità più alta è posto **in standby** e sarà il thread successivo a essere eseguito; in un sistema multiprocessore c'è un thread in standby per ogni unità d'elaborazione. Un thread è **in esecuzione** quando un'unità d'elaborazione lo sta eseguendo;

rimane in questo stato finché non viene interrotto da un thread con priorità più alta, o termina, o il suo quanto di tempo scade, o si blocca in attesa su un oggetto dispatcher – per esempio, un evento che segnala la terminazione di un’operazione di I/O. Un thread è **in attesa** se attende la segnalazione di un oggetto dispatcher. Un thread è in stato di **transizione** quando attende le risorse necessarie all’esecuzione (per esempio, potrebbe essere in attesa che il suo stack venga caricato dal disco alla memoria). Un thread è **terminato** quando ha completato l’esecuzione.

Il dispatcher adotta uno schema con 32 livelli di priorità per determinare l’ordine d’esecuzione dei thread; le priorità si dividono in due classi: la classe a priorità variabile, contenente i thread le cui priorità sono comprese fra 1 e 15, e la classe per l’elaborazione in tempo reale, contenente i thread le cui priorità vanno da 16 a 31. Il dispatcher usa una coda per ogni livello di priorità, e percorre l’insieme delle code partendo dalla priorità più alta finché non incontra un thread pronto per l’esecuzione. Se un thread ha una particolare affinità con un certo processore non disponibile in quel momento, il dispatcher lo ignora e continua a cercare un thread pronto per l’esecuzione; se nessun thread è pronto, il dispatcher esegue un thread speciale detto *idle thread*. La classe di priorità 0 è riservata per l’idle thread.

Quando scade il suo quanto di tempo, l’interruzione sollevata dal clock accoda al processore una **chiamata di procedura differita** (DPC) relativa al quanto scaduto. L’accodamento della DPC provoca un interrupt software quando il processore torna alla priorità d’interruzione normale. L’interrupt fa sì che il dispatcher riprogrammi il processore affinché esegua il successivo thread disponibile al livello di priorità del thread che subisce la prelazione.

La priorità di quest’ultimo thread può essere modificata prima che sia rimesso nelle code del dispatcher. Se il thread che subisce prelazione ha priorità di classe variabile, la sua priorità viene ridotta, anche se mai sotto la priorità di base. Il fatto di ridurre la priorità tende a limitare l’impiego di tempo di CPU da parte dei thread con prevalenza d’elaborazione rispetto a quelli con prevalenza di I/O. Quando un thread a priorità variabile abbandona lo stato d’attesa, il dispatcher aumenta la priorità in funzione dell’evento atteso dal thread: un thread che attende I/O dalla tastiera otterrà un notevole aumento di priorità, mentre un thread che attende un’operazione di I/O da un disco otterrà un aumento più moderato. Questa strategia tende a garantire brevi tempi di risposta ai thread interattivi che usano mouse e finestre, e permette ai thread con prevalenza di I/O di tenere occupati i dispositivi di I/O mentre i thread con prevalenza d’elaborazione possono utilizzare in background i cicli di CPU disponibili. Inoltre, la finestra con cui l’utente sta interagendo riceverà un aumento di priorità in modo da migliorare i tempi di risposta.

Lo scheduling può avvenire quando un thread è pronto o è posto nello stato d’attesa, quando termina, o quando un’applicazione ne cambia la priorità o il grado di affinità con un’unità d’elaborazione. Se un thread a più alta priorità diviene pronto durante l’esecuzione di un thread a priorità più bassa, quest’ultimo sarà interrotto; ciò fornisce al thread a priorità maggiore un accesso preferenziale alla CPU. Il sistema Windows, tuttavia, non è un sistema operativo per l’elaborazione in tempo reale stret-

to (*hard*) perché non garantisce che un thread real time sia eseguito entro limiti di tempo fissati; i thread sono bloccati a tempo indeterminato mentre le DPC e le routine di interrupt di servizio (ISR) sono in esecuzione (come verrà spiegato più avanti).

Tradizionalmente gli scheduler del sistema operativo si servivano del campionamento per misurare l'utilizzo della CPU da parte dei thread. Il timer di sistema emetteva un segnale periodico e il gestore degli interrupt del timer prendeva nota di quali thread erano in esecuzione al momento dell'interrupt e di quali tra questi erano eseguiti in modalità kernel o utente. Questa tecnica di campionamento era necessaria perché la CPU non aveva un clock ad alta risoluzione, oppure perché l'accesso al clock era troppo costoso o troppo inaffidabile per poter essere effettuato frequentemente. Anche se efficiente, il campionamento era impreciso e portava ad anomalie come il conteggio del tempo di gestione di un interrupt come tempo del thread e la conseguente interruzione di thread che avevano utilizzato solo una frazione del loro quanto di tempo. A partire da Windows Vista l'uso del tempo di CPU è stato monitorato utilizzando il **contatore timestamp hardware** (TSC), presente nei processori recenti. L'utilizzo del TSC conferisce una maggiore accuratezza ai calcoli relativi all'utilizzo della CPU e lo scheduler non interrompe più i thread prima che questi abbiano utilizzato completamente il loro quanto di tempo.

19.3.2.3 Realizzazione delle primitive di sincronizzazione

Le principali strutture dati del sistema operativo sono gestite come oggetti che usano funzionalità comuni per l'allocazione, il conteggio dei riferimenti e la sicurezza. Gli **oggetti dispatcher** controllano, nel sistema, il dispatching e la sincronizzazione.

- L'**oggetto evento** è usato per registrare il verificarsi di un evento e per sincronizzarlo con una qualche azione. Gli eventi di notifica segnalano tutti i thread in attesa, mentre gli eventi di sincronizzazione segnalano un singolo thread in attesa.
- L'**oggetto mutante** fornisce la mutua esclusione in modalità kernel o utente, associata con la nozione di proprietà.
- Il **mutex**, disponibile unicamente in modalità kernel, garantisce mutua esclusione senza pericolo di stallo.
- Un **oggetto semaforo** opera come un contatore, per controllare il numero di thread che accede alla risorsa.
- L'**oggetto thread** è l'entità schedulata dal dispatcher del kernel ed è associata con un **oggetto processo**, che incapsula uno spazio di indirizzamento virtuale. L'oggetto thread riceve un segnale all'uscita del thread e l'oggetto processo all'uscita del processo.
- Gli **oggetti timer** sono usati per tenere traccia del tempo e per segnalare timeout quando le operazioni impiegano troppo tempo e devono essere interrotte, oppure quando deve essere schedulata un'attività ripetitiva.

A molti degli oggetti del dispatcher si può accedere in modalità utente tramite un'operazione `open`, che restituisce un handle (o riferimento); il codice in modalità utente

controlla periodicamente e/o attende in maniera bloccante gli handle per sincronizzarsi con gli altri thread, così come con il sistema operativo (si veda il Paragrafo 19.7.1).

19.3.2.4 Interruzioni software: chiamate di procedura asincrona e differita

Il dispatcher implementa due tipi di interruzioni software: la **chiamata di procedura asincrona** (*asynchronous procedure call*, APC) e la **chiamata di procedura differita** (*deferred procedure call*, DPC). La chiamata di procedura asincrona interrompe un thread in esecuzione e richiama una procedura. Le APC servono per iniziare l'esecuzione di un nuovo thread, per sospendere o riattivare un thread esistente, per terminare i thread e i processi, per consegnare la notifica relativa a un'operazione di I/O asincrona che è stata completata e per estrarre il contenuto dei registri della CPU da un thread in esecuzione. Le APC sono accodate a specifici thread e permettono al sistema di eseguire sia il codice di sistema sia quello utente entro il contesto del processo. L'esecuzione di una APC in modalità utente non può avvenire in qualsiasi momento, ma solamente quando il thread è in attesa nel kernel ed è contrassegnato come *alertable* (allertabile).

Le chiamate di procedura differite sono finalizzate a posporre l'elaborazione delle interruzioni. Dopo aver gestito tutti i processi urgenti bloccati da interruzioni dei dispositivi, la **procedura di servizio delle interruzioni** (*interrupt service routine*, ISR) pianifica i lavori di elaborazione rimasti in sospeso accodando una DPC. Il dispatcher pianifica le interruzioni software con una priorità più bassa di quella delle interruzioni dei dispositivi, in modo che le DPC non blocchino altre ISR, ma più alta della priorità con cui il thread è in esecuzione. Oltre che per rinviare l'elaborazione delle interruzioni dei dispositivi, il dispatcher usa le DPC per elaborare le interruzioni generate dal timer e per interrompere l'esecuzione dei thread il cui quanto di tempo è scaduto.

L'esecuzione delle DPC impedisce ai thread di essere schedulati nel processore corrente, e alle APC di segnalare il completamento di un'operazione di I/O; ciò affinché le procedure DPC non impieghino quantità di tempo eccessive per essere complete. In alternativa, il dispatcher mantiene un gruppo di "thread di lavoro". Le ISR e le DPC possono accodare i lavori da svolgere ai thread di lavoro e questi saranno eseguiti utilizzando il normale scheduling dei thread. Le DPC sono limitate in modo che non possano incorrere in page fault, richiamare servizi del sistema o compiere qualsiasi altra azione che potrebbe sfociare nel tentativo di sospendere l'esecuzione in attesa di un oggetto dispatcher. A differenza delle APC, le routine DPC non fanno ipotesi in merito a quale contesto di processo il processore stia eseguendo.

19.3.2.5 Eccezioni e interruzioni

Il dispatcher del kernel fornisce anche la gestione delle eccezioni e delle interruzioni; Windows definisce diverse eccezioni indipendenti dall'architettura, tra cui:

- accesso illegale alla memoria;
- overflow in operazioni su interi;

- overflow e underflow in operazioni in virgola mobile;
- divisione intera per zero;
- divisione in virgola mobile per zero;
- istruzione illegale;
- dati non allineati;
- istruzione privilegiata;
- errore di lettura della pagina;
- violazione d'accesso;
- quota di paginazione ecceduta;
- punto d'arresto (*breakpoint*) del debugger;
- singolo passo del debugger.

Le eccezioni semplici possono essere trattate dai trap handler; le altre sono gestite dal **dispatcher delle eccezioni** del kernel, il quale annota la causa dell'eccezione per individuarne l'appropriato gestore.

Quando si verifica un'eccezione in modalità kernel, il dispatcher delle eccezioni invoca semplicemente una procedura che ne individua il gestore; nel caso in cui non se ne trovi alcuno avviene un errore di sistema fatale che lascia l'utente davanti al famigerato “schermo blu della morte”, il quale sta a significare l'arresto totale del sistema.

La gestione delle eccezioni è più complessa nel caso dei processi eseguiti in modalità utente, perché un sottosistema d'ambiente (per esempio il POSIX) può specificare una porta di debugging e una porta per le eccezioni per ogni processo da esso creato (per maggiori dettagli sulle porte si veda il Paragrafo 19.3.3.4). Se è registrata una porta di debugging, il dispatcher delle eccezioni trasmette l'eccezione a tale porta; se non si trova una porta di debugging o se essa non è in grado di gestire l'eccezione, il dispatcher tenta di individuare un appropriato gestore dell'eccezione; se non riesce a trovarlo, richiama di nuovo il debugger in modo che esso possa rilevare l'errore. Se tale programma non è in esecuzione il dispatcher manda un messaggio alla porta per le eccezioni del processo per dare al sottosistema d'ambiente la possibilità di tradurre l'eccezione: per esempio l'ambiente POSIX traduce messaggi d'eccezione in segnali POSIX prima di mandarli al thread che ha causato l'eccezione. In ultima analisi, se tutto il resto non ha funzionato il kernel termina forzatamente il processo contenente il thread che ha causato l'eccezione.

Quando Windows non riesce a gestire un'eccezione, può costruire una descrizione dell'errore che si è verificato e richiedere l'autorizzazione da parte dell'utente per inviare le informazioni a Microsoft per ulteriori analisi. In alcuni casi, l'analisi automatica di Microsoft è in grado di riconoscere immediatamente l'errore e suggerire una correzione o una soluzione alternativa.

Il dispatcher delle interruzioni del kernel gestisce le interruzioni richiamando una procedura di servizio delle interruzioni (*interrupt service routine*, ISR) o una proce-

Livelli di interruzione	Tipi di interruzione
31	controllo di macchina o errore sul bus
30	calo di tensione
29	comunicazione fra unità d'elaborazione (richiesta d'azione a un'altra unità d'elaborazione, ad esempio, avvio di un processo o aggiornamento delle TLB)
28	orologio
27	profile
3-26	ordinarie interruzioni IRQ dei PC
2	dispatch e chiamata di procedura differita (DPC) (kernel)
1	chiamata di procedura asincrona (APC)
0	passiva

Figura 19.2 Segnali d'interruzione del sistema Windows.

dura interna del kernel. Un segnale d'interruzione è rappresentato da un **oggetto interruzione** contenente tutte le informazioni necessarie per gestire l'interruzione, il che rende facile associare procedure di servizio a un segnale d'interruzione senza dover accedere direttamente al dispositivo in questione.

Diverse architetture hanno tipi e numeri d'interruzione diversi; per garantire la portabilità il dispatcher delle interruzioni associa le interruzioni a un insieme convenzionale. Ai segnali d'interruzione è assegnata una priorità e sono serviti secondo questa priorità in ordine decrescente; nel sistema Windows ci sono 32 livelli d'interruzione (**IRQL**): otto sono riservati all'uso del kernel; gli altri ventiquattro rappresentano segnali d'interruzione provenienti dai dispositivi tramite la mediazione dello HAL (sebbene la maggior parte dei sistemi IA-32 ne usi solo 16). L'elenco delle interruzioni di Windows è riportato nella Figura 19.2.

Il kernel usa una **tavella di dispatch delle interruzioni** (IDT) per associare a ogni livello delle interruzioni una procedura di servizio; nel caso di un multiprocessore, il kernel di Windows mantiene IDT separate per ogni processore, e l'IRQL di ogni processore si può regolare in modo indipendente per mascherare determinate interruzioni. Tutte le interruzioni che occorrono a un livello pari o inferiore dell'IRQL di un processore sono bloccate finché l'IRQL non si sia abbassato da un thread a livello del kernel o dal completamento di esecuzione di una ISR. Il sistema Windows sfrutta questa caratteristica impiegando le eccezioni per eseguire funzioni di sistema: per esempio, il kernel usa eccezioni per avviare il dispatch di un thread, per gestire i timer e sincronizzare i thread con il completamento degli I/O.

19.3.2.6 Commutazione tra thread in modalità utente e thread in modalità kernel

Quando un programmatore Windows pensa a un thread in realtà sta pensando a due thread: un **thread in modalità utente (UT)** e un **thread in modalità kernel (KT)**. Ognuno di questi thread ha il proprio stack, i propri valori dei registri e il proprio contesto di esecuzione. Un UT richiede un servizio di sistema eseguendo un'istruzione che provoca una trap alla modalità kernel. Lo strato kernel esegue quindi un gestore

di trap che passa dal thread UT al corrispondente thread KT. Quando un KT ha completato l'esecuzione nel kernel ed è pronto a tornare al corrispondente UT, viene invocato lo strato kernel per effettuare il passaggio all'UT, che continua la sua esecuzione in modalità utente.

Windows 7 modifica il comportamento dello strato kernel per supportare lo scheduling in modalità utente dei thread UT. Lo scheduler in modalità utente di Windows 7 supporta lo scheduling cooperativo. Da un UT si può passare esplicitamente a un altro UT chiamando lo scheduler in modalità utente, senza intervento del kernel. Lo scheduling in modalità utente è descritto in dettaglio nel Paragrafo 19.7.3.7.

19.3.3 Executive

L'executive del sistema Windows fornisce una serie di servizi utilizzabili da tutti i sottosistemi d'ambiente. I servizi sono raggruppati come segue: gestore degli oggetti, gestore della memoria virtuale, gestore dei processi, servizi relativi alle chiamate di procedura locali avanzate, gestore dell'I/O, della cache, monitor della sicurezza dei riferimenti, gestori del *plug-and-play* e dell'alimentazione, registri (*registry*) e avvio del sistema.

19.3.3.1 Gestore degli oggetti

Per la gestione di entità in modalità kernel, Windows usa un insieme di interfacce manipolate dai programmi in modalità utente. In Windows queste entità sono chiamate **oggetti**, mentre il componente di codice eseguibile che li gestisce è detto **gestore degli oggetti** (*object manager*). Esempi di oggetti sono: semafori, mutex, eventi, processi e thread. Gli oggetti citati sono tutti *oggetti dispatcher*. I thread possono bloccarsi nel dispatcher del kernel, nell'attesa che uno di questi oggetti sia segnalato. Processi, thread e API della memoria virtuale usano gli handle per identificare il processo o il thread su cui operare. Altri esempi di oggetti includono file, sezioni, porte e vari oggetti interni di I/O. Gli oggetti file sono usati per mantenere lo stato aperto dei file e dei dispositivi. Le sezioni sono usate per mappare i file. I punti terminali locali delle comunicazione sono implementati come oggetti porta.

Il codice in modalità utente accede a questi oggetti utilizzando un valore opaco chiamato **handle**, che viene restituito da molte API. Ogni processo ha una tabella degli handle contenente le voci che tengono traccia degli oggetti utilizzati dal processo. Il **processo di sistema**, che contiene il kernel, ha la sua tabella degli handle protetta dal codice utente. Le tabelle degli handle sono rappresentate in Windows da una struttura ad albero che può contenere da 1.024 fino a oltre 16 milioni di handle. Il codice in modalità kernel può accedere a un oggetto utilizzando un handle o un **puntatore di riferimento** (*referenced pointer*).

Un processo ottiene un handle creando un oggetto, aprendo un oggetto esistente, ricevendo un handle duplicato da un altro processo o ereditando un handle dal processo padre. Quando un processo termina tutti i suoi handle ancora aperti vengono implicitamente chiusi. Dal momento che il gestore degli oggetti è l'unica entità che genera handle di oggetti, si tratta del luogo naturale per controllare la sicurezza. Il

gestore degli oggetti verifica se un processo ha il diritto di accedere a un oggetto quando tenta di aprirlo. Il gestore degli oggetti applica inoltre delle quote, come la quantità massima di memoria che un processo può utilizzare, tenendo in conto della memoria occupata da tutti gli oggetti referenziati dal processo e rifiutando di allocare ulteriore memoria quando la quantità di memoria accumulata eccede la quota assegnata al processo.

Il gestore degli oggetti tiene traccia di due conteggi per ciascun oggetto: il numero di handle e il numero di puntatori. Il primo è il numero di handle che fanno riferimento all'oggetto nelle tabelle degli handle di tutti i processi, compreso il processo di sistema che contiene il kernel. Il conteggio dei puntatori che fanno riferimento all'oggetto viene incrementato ogni volta che il kernel ha bisogno di un nuovo puntatore e decrementato quando il kernel ha finito di utilizzare un puntatore. Lo scopo del conteggio di questi riferimenti è quello di garantire che un oggetto non venga liberato mentre vi fa ancora riferimento un handle o un puntatore interno al kernel.

Il gestore degli oggetti mantiene lo spazio dei nomi interno di Windows. A differenza di UNIX, che radica lo spazio dei nomi del sistema nel file system, Windows utilizza uno spazio dei nomi astratto e collega i file system come dispositivi. L'assegnazione di un nome a un oggetto di Windows dipende da chi ha creato l'oggetto. Processi e thread vengono creati senza nome e vi si fa riferimento sia mediante handle sia tramite un identificatore numerico separato. Gli eventi di sincronizzazione hanno di solito dei nomi, in modo che possano essere aperti da processi indipendenti. Un nome può essere permanente o temporaneo: un nome permanente rappresenta un'entità, come un disco rigido, che rimane anche se nessun processo vi sta accedendo; un nome temporaneo esiste solo fino a quando un processo contiene un handle all'oggetto. Il gestore degli oggetti supporta le directory e i link simbolici nello spazio dei nomi. A titolo di esempio, le lettere delle unità disco in MS-DOS vengono implementate usando collegamenti simbolici: `\Global??\C:` è un collegamento simbolico all'oggetto dispositivo `\Device\HarddiskVolume2` che rappresenta un volume del file-system montato nella directory `\Device`.

Ogni oggetto, come accennato in precedenza, è un'istanza di un tipo di oggetto. Il tipo di oggetto specifica come devono essere allocate le istanze, come devono essere definiti i campi di dati e come deve essere implementato il set standard di funzioni virtuali utilizzate per tutti gli oggetti. Le funzioni standard implementano operazioni come la mappatura dei nomi agli oggetti, la chiusura, l'eliminazione e l'applicazione di controlli di sicurezza. Le funzioni specifiche per un particolare tipo di oggetto sono implementate dai servizi di sistema progettati per funzionare su quel particolare tipo di oggetto, non da metodi specificati nel tipo di oggetto.

La funzione `parse()` è la più interessante tra le funzioni standard e consente l'implementazione di un oggetto. I file system, l'archivio di configurazione del registro di sistema e gli oggetti GUI utilizzano le funzioni di parse per estendere lo spazio dei nomi di Windows, e ne rappresentano i principali utilizzatori.

Tornando al nostro esempio di assegnazione dei nomi in Windows, gli oggetti dispositivo utilizzati per rappresentare i volumi del file system forniscono una funzione

di parse per permettere a un nome come \Global??\C:\foo\bar.doc di essere interpretato come il file \foo\bar.doc del volume rappresentato dall'oggetto dispositivo HarddiskVolume2. Possiamo osservare come l'assegnazione dei nomi, le funzioni di parse, gli oggetti e gli handle lavorano insieme illustrando i passi che permettono di aprire un file in Windows.

1. Un'applicazione richiede l'apertura di un file denominato C:\foo\bar.doc.
2. Il gestore degli oggetti trova l'oggetto dispositivo HarddiskVolume2, cerca la procedura di parse IopParseDevice dal tipo di oggetto e la richiama con il nome del file relativo alla radice del file system.
3. IopParseDevice() alloca un oggetto file e lo passa al file system che fornisce i dettagli su come accedere a C:\foo\bar.doc sul volume.
4. Quando il file system ha terminato, IopParseDevice() assegna una voce nella tabella degli handle per l'oggetto file per il processo corrente e restituisce l'handle all'applicazione.

Se il file non può essere aperto correttamente, IopParseDevice() cancella l'oggetto file che ha allocato e restituisce all'applicazione una segnalazione di errore.

19.3.3.2 Gestore della memoria virtuale

Il componente dell'executive che gestisce lo spazio degli indirizzi virtuali, l'allocazione della memoria fisica e la paginazione è il **gestore della memoria virtuale** (*virtual memory manager*, VM). Il gestore della VM è concepito assumendo che l'architettura sottostante sia in grado di associare indirizzi virtuali a indirizzi fisici, possieda un meccanismo di paginazione, realizzzi trasparentemente la coerenza della cache nei sistemi multiprocessore, e permetta l'associazione di più elementi della tabella delle pagine alla stessa pagina fisica.

Il gestore della VM di Windows adotta uno schema di gestione basato su pagine della dimensione di 4 KB e 2 MB su processori AMD64 e IA-32 compatibili, e di 8 KB su IA64. Le pagine di dati allocate a un processo, ma non residenti nella memoria fisica, sono memorizzate nel **file di paginazione** in un disco o mappate direttamente su un file ordinario di un file system locale o remoto. Le pagine possono anche essere marcate come *zero-fill-on-demand*, il che ha l'effetto di inizializzarle a zero prima dell'allocazione, cancellandone i contenuti precedenti.

Sui processori IA-32, i processi hanno 4 GB di memoria virtuale a testa. I 2 GB superiori sono quasi identici per tutti i processi, e servono a Windows per accedere in modalità kernel alle strutture dati e al codice del sistema operativo. Nel caso dell'architettura AMD64 Windows fornisce 8 TB di spazio di indirizzamento virtuale per la modalità utente nell'ambito dei 16 EB supportati dall'hardware per ogni processo.

Alcune aree cruciali della regione di memoria del kernel che variano da processo a processo sono la **mappa della page table**, l'**iperspazio** e lo **spazio della sessione**. L'hardware denota le tabelle delle pagine di un processo per mezzo dei numeri dei frame fisici e la mappa (*self-map*) della page table rende i contenuti della page table dei processi accessibili tramite indirizzi virtuali. L'iperspazio mappa le informazioni

relative all’insieme di lavoro corrente del processo nello spazio degli indirizzi accessibile in modalità kernel. Lo spazio della sessione serve per condividere un’istanza di Win32 e altri driver legati alla specifica sessione fra tutti i processi della stessa sessione del terminal server (TS). Sessioni TS distinte condividono istanze differenti di questi driver, anche se questi sono mappati agli stessi indirizzi virtuali. La regione, più bassa, della modalità utente dello spazio degli indirizzi virtuali è specifica per ogni processo e accessibile sia dai thread utente che dai thread del kernel.

Per assegnare memoria virtuale il gestore della VM procede in due passi: innanzitutto *riserva* una o più pagine di indirizzi virtuali dello spazio d’indirizzi virtuali del processo; quindi *impegna* l’assegnazione dello spazio per la memoria virtuale (spazio in memoria fisica o nel file di paginazione). Il sistema operativo Windows può limitare lo spazio del file di paginazione impiegato da un processo imponendo una quota massima d’assegnazione; un processo può rilasciare parti di memoria non più in uso a beneficio di altri processi. Le API che riservano gli indirizzi virtuali e impegnano la memoria virtuale ricevono come parametro l’handle di un oggetto processo. In questo modo, un dato processo è in grado di controllare la memoria virtuale di un altro processo. I sottosistemi d’ambiente gestiscono la memoria dei loro processi client in questo modo.

Windows implementa la memoria condivisa tramite gli **oggetti sezione**. Un processo, dopo aver ottenuto un handle dell’oggetto sezione, mappa la porzione di memoria della sezione su un intervallo di indirizzi, chiamato **vista** (*view*). Un processo può stabilire una vista dell’intera sezione o solo della parte di sezione di cui ha bisogno. Windows consente alle sezioni di essere mappate non solo nel processo corrente, ma in un qualunque processo per cui il chiamante possegga un handle.

Le sezioni sono utilizzabili in molti modi. Una sezione può utilizzare come memoria ausiliaria il file di paginazione di sistema o un file ordinario (un **file mappato in memoria**). Una sezione può essere *basata* (*based*), nel senso che appare allo stesso indirizzo virtuale a ogni processo che vi accede. Le sezioni possono anche rappresentare la memoria fisica, consentendo così a un processo a 32 bit di accedere a una maggior quantità di memoria fisica di quanta ne possa essere contenuta nel suo spazio di indirizzamento virtuale. Infine la protezione delle pagine della sezione si può impostare per la sola lettura, lettura e scrittura, sola esecuzione, non accessibile, e copiatura su scrittura.

Le ultime due impostazioni di protezione richiedono alcune spiegazioni.

- Una *pagina non accessibile* solleva un’eccezione qualora si tenti di accedervi; l’eccezione si può usare, fra l’altro, per controllare se un programma difettoso esegue iterazioni che vanno oltre la dimensione di un array o semplicemente per rilevare se un programma tenta di accedere a indirizzi virtuali non presenti in memoria. Gli stack in modalità utente e kernel utilizzano le pagine non accessibili come **sentinelle** (*guard page*) per rilevare eventuali stack overflow. Un altro possibile uso è il controllo del superamento delle dimensioni del buffer (buffer overrun) nello heap. Sia l’allocatore della memoria utente sia l’allocatore del kernel usato dal verificatore del dispositivo possono essere configurati per map-

pare ogni allocazione richiesta fino alla fine di una pagina seguita da una pagina non accessibile; ciò permette di rilevare errori di programmazione che causano accessi oltre il limite dell'allocazione.

- Il meccanismo di copiatura su scrittura permette al gestore della VM di risparmiare memoria: quando due processi vogliono copie indipendenti dei dati dello stesso oggetto, il gestore pone solo una copia condivisa nella memoria fisica e attribuisce a quella sezione la proprietà di copiatura su scrittura; se uno dei processi tenta di scrivere dati in una delle pagine caratterizzate da tale proprietà, il gestore fornisce al processo una copia privata della pagina che può poi essere modificata.

La traduzione degli indirizzi virtuali in Windows sfrutta una tabella delle pagine a più livelli. Nel caso di processori IA-32 e AMD64, ogni processo ha una **directory delle pagine** contenente 512 **elementi della directory delle pagine** (*page-directory entries*, PDE), da 8 byte l'uno; ognuno di loro punta a una tabella delle pagine, che a sua volta contiene 512 elementi della tabella delle pagine (*page-table entries*, PTE) di 8 byte, che infine puntano a **frame** da 4 KB nella memoria fisica. Per una serie di motivi, l'hardware richiede che le directory delle pagine o le tabelle dei PTE a ogni livello di una tabella delle pagine multilivello occupino una sola pagina. In tal modo, il numero di PDE o PTE contenuti in una pagina determina quanti indirizzi virtuali sono tradotti da quella pagina. Per uno schema di questa struttura si veda la Figura 19.3.

La struttura fin qui descritta può essere utilizzata per rappresentare la traduzione di solo 1 GB di indirizzi virtuali. Nel caso di IA-32 è necessario un secondo livello di directory delle pagine contenente solo quattro voci, come mostrato nel diagramma. Su processori a 64 bit sono necessari più livelli. Per AMD64 Windows utilizza in totale

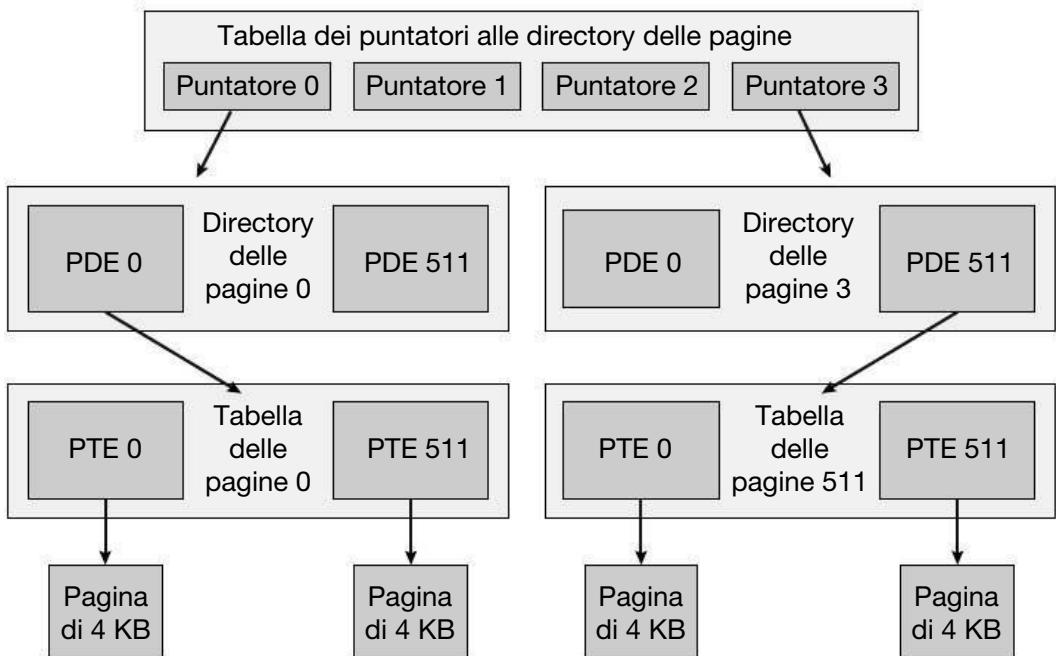


Figura 19.3 Struttura della tabella delle pagine.



Figura 19.4 Composizione di un indirizzo di memoria virtuale di un'architettura IA32.

quattro livelli completi. La dimensione totale di tutte le pagine della tabella delle pagine necessarie per rappresentare pienamente anche solo uno spazio di indirizzamento virtuale a 32 bit per un processo è di 8 MB. Il gestore della VM alloca pagine di PDE e PTE secondo le necessità e sposta le pagine della tabella su disco quando non sono utilizzate. Le pagine della tabella sono riportate in memoria quando vi si fa riferimento.

Descriviamo ora come gli indirizzi virtuali siano tradotti in indirizzi fisici su processori IA-32. Un valore di 2 bit può rappresentare i valori 0, 1, 2, 3, un valore di 9 bit può rappresentare valori nell'intervallo 0-511 e un valore di 12 bit valori da 0 a 4095. Un valore di 12 bit può pertanto selezionare qualsiasi byte all'interno di una pagina di 4 KB di memoria. Un valore di 9 bit può rappresentare uno qualsiasi dei 512 PDE o PTE in una directory delle pagine o in una pagina della tabella dei PTE. Come mostrato nella Figura 19.4, la traduzione di un puntatore di indirizzo virtuale nell'indirizzo di un byte nella memoria fisica si effettua suddividendo il puntatore a 32 bit in quattro valori. A partire dai bit più significativi.

- Due bit vengono utilizzati per indicizzare i quattro PDE al livello superiore della tabella delle pagine. Il PDE selezionato conterrà il numero di pagina fisica per ciascuna delle quattro pagine della directory delle pagine che mappano 1 GB di spazio di indirizzi.
- Nove bit vengono utilizzati per selezionare un altro PDE, questa volta da una directory delle pagine di secondo livello. Questo PDE conterrà i numeri di pagina fisica di un massimo di 512 pagine della tabella dei PTE.
- Nove bit vengono utilizzati per selezionare uno dei 512 PTE dalla pagina selezionata della tabella dei PTE. Il PTE selezionato conterrà il numero di pagina fisica per il byte a cui stiamo accedendo.
- Dodici bit sono usati come offset all'interno della pagina. L'indirizzo fisico del byte a cui stiamo accedendo è costruito aggiungendo i 12 bit più bassi dell'indirizzo virtuale alla fine del numero di pagina fisica che abbiamo trovato nel PTE selezionato.

Il numero di bit in un indirizzo fisico può essere diverso dal numero di bit in un indirizzo virtuale. Nell'architettura IA-32 originale, il PTE e il PDE erano strutture a 32 bit che facevano spazio a 20 bit di numero di pagina fisica: la dimensione dell'indirizzo fisico e la dimensione degli indirizzi virtuali erano quindi le stesse e tali sistemi

potevano indirizzare solo 4 GB di memoria fisica. Più tardi, la IA-32 fu estesa con l'introduzione dei PTE a 64 bit utilizzati ancora oggi e l'hardware iniziò a supportare numeri di pagina fisica a 24 bit. Questi sistemi sono in grado di gestire 64 GB di memoria ed erano utilizzati su sistemi server. Oggi tutti i server Windows si basano su AMD64 o IA64 e supportano indirizzi fisici molto grandi, molto più grandi di quanto sia possibile utilizzare: ricordate il fatto che una volta 4 GB di memoria fisica sembravano ottimisticamente troppi.

Per migliorare le prestazioni il gestore della VM mappa la directory delle pagine e le pagine della tabella dei PTE nella stessa regione contigua di indirizzi virtuali in ogni processo. Questa auto-mappa (self-map) permette al gestore di utilizzare lo stesso puntatore per accedere al PDE o al PTE corrente, corrispondente a un particolare indirizzo virtuale, indipendentemente dal processo che è in esecuzione. In IA-32 l'auto-mappa utilizza una regione contigua da 8 MB dello spazio di indirizzamento virtuale del kernel, mentre in AMD64 la mappa occupa 512 GB. Nonostante il significativo spazio di indirizzamento occupato, questa mappa non richiede alcuna pagina di memoria virtuale aggiuntiva; inoltre, consente alle pagine della tabella delle pagine di essere automaticamente spostate dentro e fuori dalla memoria fisica.

Nella creazione di un'auto-mappa, uno dei PDE della directory delle pagine di livello superiore fa riferimento alla pagina stessa, formando un “ciclo” nella traduzione della tabella delle pagine. Quando non si dà percorre il ciclo si effettua un accesso alle pagine virtuali, quando il ciclo viene percorso una volta si accede alle pagine della tabella dei PTE, quando il ciclo viene percorso due volte si accede alle pagine della directory delle pagine di livello più basso, e così via.

I livelli aggiuntivi delle directory delle pagine utilizzati per la memoria virtuale a 64 bit sono tradotti nello stesso modo, eccetto per il fatto che il puntatore all'indirizzo virtuale viene suddiviso in ancor più valori. Nel caso di AMD64, Windows utilizza quattro livelli pieni, ognuno dei quali mappa 512 pagine, o $9+9+9+9+12 = 48$ bit di indirizzamento virtuale.

Per evitare i rallentamenti dovuti alla traduzione degli indirizzi virtuali tramite la ricerca dei PDE e PTE appropriati, i processori adottano uno speciale hardware detto **TLB** (*translation look-aside buffer*), contenente una memoria cache associativa per mappare pagine virtuali sui PTE. La TLB fa parte dell'unità di gestione della memoria (MMU) all'interno di ciascun processore. La MMU ha bisogno di percorrere la tabella delle pagine (le sue strutture dati) in memoria solo quando una traduzione è necessaria, perché manca nella TLB.

I PDE e i PTE non contengono solo i numeri di pagina fisica, ma hanno anche alcuni bit riservati per l'uso del sistema operativo e alcuni bit per controllare come l'hardware utilizzi la memoria, per esempio per controllare se la cache hardware debba essere utilizzata per ogni pagina. Inoltre le voci specificano il tipo di accesso consentito per le modalità kernel e utente.

Un PDE può essere contrassegnato per segnalare che dovrebbe funzionare come PTE piuttosto che come PDE. Su IA-32 i primi 11 bit del puntatore di indirizzo virtuale selezionano un PDE nei primi due livelli di traduzione. Se il PDE selezionato è con-

trassegnato per fungere da PTE i rimanenti 21 bit del puntatore sono usati come offset del byte. Ciò porta a una dimensione di pagina di 2 MB. Mischiare e far corrispondere pagine di 4 KB e 2 MB all'interno della tabella delle pagine è facile per il sistema operativo e può migliorare significativamente le prestazioni di alcuni programmi, riducendo la frequenza con cui la MMU ha bisogno di ricaricare le voci nella TLB, visto che un PDE di 2 MB sostituisce 512 PTE di 4 KB.

Gestire la memoria fisica in modo che le pagine di 2 MB siano disponibili al bisogno è tuttavia difficile, in quanto le pagine possono continuamente essere suddivise in pagine di 4 KB, provocando frammentazione esterna della memoria. Inoltre, le pagine grandi possono causare una frammentazione interna piuttosto significativa. A causa di questi problemi, sono in genere esclusivamente lo stesso Windows e le applicazioni per server di grandi dimensioni a utilizzare pagine grandi per migliorare le prestazioni della TLB. Essi sono più adatti a farlo, perché il sistema operativo e le applicazioni server iniziano l'esecuzione all'avvio del sistema, prima che la memoria sia diventata frammentata.

Windows gestisce la memoria fisica associando ciascuna pagina fisica a uno dei sette stati seguenti: libera, azzerata, modificata, in attesa, guasta, in transizione o valida.

- Una pagina libera non ha un contenuto particolare.
- Una pagina azzerata è libera, i suoi contenuti sono stati inizializzati a zero ed è disponibile per soddisfare richieste di pagine *zero-on-demand*.
- Una pagina modificata è stata modificata (tramite scrittura) da un processo e deve essere trasferita su disco prima di essere allocata a un altro processo.
- Una pagina in attesa è una copia d'informazioni già presenti su disco. Si può trattare di una pagina non ancora modificata, di una pagina già scritta su disco o di una pagina caricata anticipatamente in memoria perché ci si aspetta di utilizzarla al più presto.
- Una pagina guasta è inutilizzabile a causa di un errore hardware.
- Una pagina in transizione è in corso di trasferimento dal disco a un frame allocato nella memoria fisica.
- Una pagina valida è parte del working set di uno o più processi ed è contenuta all'interno delle tabelle delle pagine di questi processi.

Mentre le pagine valide sono contenute nelle tabelle delle pagine dei processi, le pagine in altri stati sono mantenute in liste distinte in base al tipo di stato. Le liste sono costruite associando le voci corrispondenti nel database **PFN** (**page frame number**, *numero di frame di pagina*), che contiene una voce per ogni pagina di memoria fisica. Le voci PFN includono anche informazioni come il numero di riferimenti, i lock e le informazioni NUMA. Si noti che il database PFN rappresenta pagine di memoria fisica, mentre il PTE rappresenta pagine di memoria virtuale.

Se il bit di validità di un PTE è a zero, l'hardware ignora gli altri bit e il gestore della VM può definirne il formato per il proprio uso. Le pagine non valide si possono



Figura 19.5 Struttura dei PTE. Il bit di validità è zero.

trovare in un certo numero di stati rappresentati da bit nel PTE. Le pagine dei file di paginazione mai caricati in seguito a un fault sono marcate *zero-on-demand*; le pagine mappate tramite oggetti sezione codificano un puntatore all'oggetto in questione; le pagine scritte sul file di paginazione contengono informazioni sufficienti per reperire la pagina sul disco, e così via. La struttura dei PTE è mostrata nella Figura 19.5. Per questo tipo di PTE i bit T, P e V sono tutti a zero. Il PTE contiene 5 bit dedicati alla protezione della pagina, 32 bit per l'offset all'interno del file di paginazione e 4 bit per selezionare il file di paginazione. Ci sono anche 20 bit riservati per ulteriori operazioni.

Windows utilizza una politica di sostituzione LRU sui working set per prelevare le pagine dai processi in maniera appropriata. Quando si avvia un processo, gli viene assegnata una dimensione minima predefinita del working set. Il working set di ogni processo può crescere fino a quando la quantità residua di memoria fisica comincia a scarseggiare; a quel punto il gestore della VM inizia a tracciare l'età delle pagine in ogni working set. Infine, se la memoria disponibile è quasi esaurita, il gestore della VM taglia il working set rimuovendo le pagine più vecchie.

L'età di una pagina non dipende da quanto tempo è stata in memoria, ma dall'ultimo riferimento alla pagina, e viene determinata scandendo periodicamente il working set di ogni processo e incrementando l'età delle pagine che dall'ultima passata non sono state evidenziate nel PTE come referenziate. Quando diventa necessario tagliare il working set, il gestore della VM utilizza euristiche per decidere quanto tagliare da ogni processo e quindi rimuove le pagine a partire dalle meno recenti.

È possibile che il working set di un processo venga tagliato anche quando resta disponibile molta memoria. Ciò avviene se è stato impostato un limite rigido sulla quantità di memoria fisica che il processo può usare. In Windows 7 il gestore della VM è in grado di tagliare anche i processi che stanno crescendo rapidamente, anche se la memoria è abbondante. Questa politica migliora significativamente la capacità di risposta del sistema sugli altri processi.

Windows tiene traccia dei working set non solo per i processi in modalità utente, ma anche per il processo di sistema, che comprende tutte le strutture dati paginabili e il codice che viene eseguito in modalità kernel. Windows 7 crea working set supplementari per il processo di sistema e li associa a particolari categorie di memoria del kernel. La cache dei file, l'heap del kernel e il codice del kernel hanno in Windows 7 i loro working set. La presenza di working set distinti consente al gestore della VM di utilizzare diverse politiche per tagliare le diverse categorie di memoria del kernel.

A seguito di un errore per page fault, il gestore della VM non carica solo la pagina immediatamente necessaria, perché vi è evidenza a supporto della tesi che i riferimenti in memoria seguano un principio di **località**: se una pagina è usata è probabile che lo siano anche le pagine adiacenti nel prossimo futuro. (Si pensi alle iterazioni per scandire un array, o al reperimento sequenziale delle istruzioni che costituiscono il codice di un thread). Per questi motivi, il gestore carica in memoria la pagina richiesta, insieme però a un certo numero di pagine adiacenti; ciò tende a ridurre il numero totale di page fault. Inoltre, le operazioni di lettura sono accorpate per migliorare le prestazioni.

Oltre a occuparsi della memoria impegnata, il gestore della VM si prende anche cura della memoria riservata da ogni processo, ossia del suo spazio degli indirizzi virtuali. Ciascun processo ha un albero associato che specifica la gamma di indirizzi virtuali usati, e la loro finalità. Su questa base, il gestore della VM può caricare le pagine mancanti man mano che servono: se il PTE non è inizializzato, il gestore cerca l'indirizzo interessato all'interno dell'**albero dei descrittori di indirizzi virtuali** (*virtual address descriptor*, VAD), e usa l'informazione così reperita per creare il PTE mancante e individuare la pagina. In qualche caso potrebbe non esistere la pagina della tabella dei PTE, quindi il gestore dovrà allocarla e inizializzarla in maniera trasparente. In altri casi, la pagina potrebbe essere condivisa come parte di un oggetto sezione e il VAD conterrà un puntatore a quell'oggetto sezione. L'oggetto sezione contiene informazioni su come trovare la pagina virtuale condivisa in modo che il PTE possa essere inizializzato per puntare direttamente a questa.

19.3.3.3 Gestore dei processi

Il gestore dei processi del sistema Windows fornisce i servizi necessari a creazione, eliminazione e uso dei thread e dei processi. Esso non possiede alcuna informazione sulle relazioni parentali o sulle gerarchie fra i processi; questi dettagli sono lasciati al particolare sottosistema d'ambiente relativo al processo. Il gestore dei processi non è coinvolto neppure nello scheduling dei processi, fuorché per determinare le priorità e le affinità dei processi e dei thread, quando essi sono creati. Lo scheduling dei thread ha luogo nel dispatcher del kernel.

Ciascun processo contiene uno o più thread. I processi medesimi possono essere raccolti in grosse unità, chiamate **oggetti job**; il ricorso a oggetti job permette di impostare limiti all'utilizzo della CPU e alla dimensione dei working set e di definire affinità per i processori che controllano più processi in una volta. Gli oggetti job sono usati per gestire le potenti macchine dei centri di elaborazione dati.

I seguenti passi fungono da esempio per la creazione di un processo in Win32.

1. Un'applicazione di Win32 richiama `CreateProcess()`.
2. Viene inviato un messaggio al sottosistema Win32 per notificare la creazione del processo.
3. `CreateProcess()`, nel processo originale, richiama una API nel gestore dei processi dell'executive per creare effettivamente il processo.
4. Il gestore dei processi richiama il gestore degli oggetti per creare un oggetto processo, e restituisce l'handle a Win32;
5. Win32 richiama di nuovo il gestore del processo al fine di costituire un thread per il processo e restituisce gli handle relativi al processo e al thread.

Le API di Windows che manipolano la memoria virtuale e i thread, e che duplicano gli handle, accettano come parametro in ingresso un handle del processo, in modo che i sottosistemi possano eseguire operazioni per conto di un nuovo processo senza dover eseguire direttamente nel contesto del nuovo processo. Non appena è creato un nuovo processo, si genera il thread iniziale; una chiamata di procedura asincrona è recapitata al thread per avviare l'esecuzione del caricatore dell'immagine in modalità utente. Il caricatore è nella `ntdll.dll`, una libreria di collegamento dinamico mappata automaticamente su ogni processo creato. Per la generazione dei processi, Windows fornisce altresì una chiamata `fork()`, di derivazione UNIX, allo scopo di supportare il sottosistema d'ambiente POSIX. Sebbene l'ambiente Win32 contatti il gestore dei processi dal processo client, la natura inter-processo delle API di Windows consente a POSIX la creazione del nuovo processo, direttamente dal processo del sottosistema in cui agisce.

Il gestore dei processi si basa sulle chiamate di procedura asincrone (APC) implementate dal livello kernel. Le APC vengono utilizzate per iniziare l'esecuzione dei thread, per sospendere e riprendere thread, per accedere ai registri dei thread, per terminare thread e processi e per il supporto dei debugger.

Il supporto del debugger da parte del gestore dei processi include la possibilità di sospendere e riavviare i thread, e di creare thread che iniziano l'esecuzione sospendendosi. Vi sono anche API del gestore dei processi in grado di leggere e impostare il contesto del registro di un thread e accedere alla memoria virtuale di un altro processo. I thread possono essere creati nel processo corrente, ma possono anche essere iniettati in un processo diverso. Il debugger fa uso di quest'ultimo meccanismo per eseguire codice all'interno di un processo in fase di debug.

Nell'ambito dell'executive, i thread esistenti possono aggregarsi temporaneamente a un altro processo: questo metodo è usato dai thread di lavoro che devono operare nel contesto del processo da cui proviene una richiesta di lavoro. Per esempio, il gestore di MV potrebbe utilizzare un thread aggregato quando è necessario l'accesso al working set o alla tabella delle pagine di un processo e il gestore di I/O potrebbe utilizzarlo per aggiornare la variabile di stato in un processo per le operazioni di I/O asincrono.

Il gestore prevede anche la **impersonazione**. A ogni thread è associato un contrassegno di sicurezza (*security token*). Quando il processo di login autentica un utente il security token viene collegato al processo dell'utente ed ereditato dai suoi processi figlio. Il token contiene il **SID** (*security identity*, identità di sicurezza) dell'utente, i SID dei gruppi cui appartiene l'utente, i privilegi dell'utente e il livello di integrità del processo. Per impostazione predefinita tutti i thread all'interno di un processo condividono un token comune che rappresenta l'utente e l'applicazione che ha avviato il processo. Tuttavia un thread in esecuzione in un processo con un security token appartenente a un utente può attivare un token specifico appartenente a un altro utente per impersonare tale utente.

La personificazione è fondamentale per il modello RPC client-server, dove i servizi devono agire per conto di una varietà di client con diversi ID di sicurezza. Il diritto di rappresentare un utente è spesso fornito come parte di una connessione RPC da un processo client a un processo server. La impersonazione consente al server di accedere ai servizi di sistema come se fosse il client in modo da poter accedere a oggetti e file o crearli per conto del client. Il processo server deve essere affidabile e deve essere scritto con molta attenzione per poter essere robusto contro gli attacchi. In caso contrario un client potrebbe subentrare a un processo server e impersonare quindi qualsiasi utente che in seguito effettuerà una richiesta.

19.3.3.4 Servizi per la computazione client-server

L'implementazione di Windows utilizza un modello client-server. I sottosistemi d'ambiente sono server che implementano specifiche personalità del sistema operativo. Oltre a ciò, il medesimo modello è applicato alla realizzazione di tutta una gamma di servizi del sistema: l'autenticazione degli utenti, i servizi di rete, la gestione delle code di stampa, i servizi web, i file system di rete, il *plug-and-play*. Per moderare la memoria necessaria, molti servizi sono spesso accorpati all'interno di pochi processi che eseguono il programma `svchost.exe`. Ogni servizio viene caricato come una libreria dinamica (DLL) che realizza il servizio sfruttando le potenzialità dei pool di thread in modalità utente per condividere thread e ricevere messaggi (si veda il Paragrafo 19.3.3.3).

Il paradigma normale per realizzare una computazione client-server consiste nell'utilizzo di RPC per comunicare le richieste. L'API Win32 supporta un protocollo RPC standard, come descritto nel Paragrafo 19.6.2.7. RPC utilizza diversi meccanismi di trasporto (per esempio, NamedPipes e TCP/IP) e può essere utilizzato per implementare RPC tra sistemi diversi. Quando una RPC viene sempre eseguita tra un client e un server sullo stesso sistema locale, per il trasporto può essere utilizzata la chiamata di procedura locale avanzata (ALPC). Al livello più basso del sistema, nell'implementazione dei sistemi di ambiente, e per i servizi che devono essere disponibili nelle fasi di avvio, RPC non è disponibile. I servizi nativi di Windows utilizzano in questi casi direttamente le ALPC.

Una ALPC è un meccanismo per lo scambio dei messaggi. Il processo server pubblica un oggetto globalmente visibile che rappresenta una porta di connessione: quan-

do un client necessita di un servizio da un sottosistema, apre un handle per l'oggetto porta di connessione del sottosistema e trasmette una richiesta di connessione alla porta in questione; il server crea un canale e restituisce un handle al client. Il canale consiste in una coppia di porte di comunicazione private: una per i messaggi dal client al server, l'altra per quelli dal server al client. I canali di comunicazione forniscono un meccanismo di richiamata che permette al client e al server di accettare richieste anche quando attendono una risposta.

Al momento della creazione di un canale ALPC è necessario specificare una fra tre possibili tecniche di scambio dei messaggi.

1. La prima tecnica è adatta a messaggi brevi o di lunghezza media (fino a 63 KB): in questo caso, si usa la coda dei messaggi della porta come mezzo di memorizzazione intermedio, e si copiano i messaggi da un processo all'altro.
2. La seconda tecnica è adatta a messaggi più lunghi: in questo caso, si crea un oggetto sezione di memoria condivisa per il canale; i messaggi trasmessi attraverso la coda dei messaggi della porta contengono un puntatore alla sezione, oltre a informazioni sulla dimensione dell'oggetto sezione. In questo modo si evita la necessità di copiare lunghi messaggi: il mittente inserisce dati nella sezione condivisa e il ricevente può vederli immediatamente.
3. La terza tecnica sfrutta le API che leggono e scrivono direttamente nello spazio degli indirizzi di un processo. Il meccanismo ALPC mette a disposizione dei server funzioni e strumenti di sincronizzazione che permettono di accedere ai dati del client. Questa tecnica è solitamente utilizzata dalle RPC per ottenere prestazioni migliori in alcuni scenari specifici.

Il gestore delle finestre di Win32 gestisce la trasmissione dei messaggi in modo diverso e indipendente dall'ALPC. Se un client richiede una connessione con trasmissione di messaggi basata sul gestore delle finestre, il server predispone tre oggetti: (1) un thread del server dedicato alla gestione delle richieste, (2) un oggetto sezione da 64 KB, e (3) un cosiddetto *oggetto coppia di eventi*. Quest'ultimo è un oggetto per la sincronizzazione che il sottosistema Win32 usa per notificare che il thread client ha copiato un messaggio verso il server Win32, o viceversa. L'oggetto sezione viene utilizzato per passare i messaggi, e l'oggetto coppia di eventi esegue la sincronizzazione.

Questo meccanismo per lo scambio di messaggi del gestore delle finestre offre molti vantaggi.

- L'oggetto sezione elimina la necessità di copiare messaggi, poiché rappresenta una regione di memoria condivisa.
- L'oggetto coppia di eventi elimina i rallentamenti legati all'uso di oggetti porta per passare i messaggi contenenti puntatori e lunghezze.
- Il thread server dedicato elimina i rallentamenti legati all'identificazione del thread client che richiede il servizio, visto che vi è un thread server per ciascun thread client.

- Il kernel avvantaggia lo scheduling di questi thread server dedicati, per migliorare le prestazioni del sistema.

19.3.3.5 Gestore dell'I/O

Il **gestore dell'I/O** è responsabile della gestione dei file system, dei driver dei dispositivi e dei driver di rete: tiene traccia dei driver e dei file system caricati e gestisce i buffer per le richieste di I/O; collabora con il gestore della VM per eseguire l'I/O dei file mappati in memoria, e controlla il gestore della cache di Windows che, a sua volta, gestisce i servizi di caching per l'intero sistema di I/O. Il gestore di I/O è sostanzialmente asincrono; l'I/O sincrono è realizzato tramite l'attesa esplicita del completamento dell'operazione. Il gestore di I/O fornisce numerosi modelli di completamento asincrono dell'I/O, compresa l'impostazione di eventi, l'aggiornamento di una variabile di stato nel processo chiamante, l'invio di APC al thread che ha avviato l'operazione, e le porte di completamento di I/O, che permettono a un singolo thread di gestire il completamento dell'I/O di molti thread.

I driver dei dispositivi sono organizzati, per ciascun dispositivo, in una lista detta stack di I/O del dispositivo. Un driver viene rappresentato nel sistema come oggetto driver. Visto che un singolo driver può operare su più dispositivi, i driver vengono rappresentati sullo stack dell'I/O mediante un oggetto dispositivo che contiene un link all'oggetto driver. Il gestore di I/O converte le richieste ricevute in una forma standard, detta **pacchetto di richiesta di I/O** (*I/O request packet*, IRP), che inoltra al primo driver nello stack affinché sia elaborato. Dopo che un driver ha elaborato l'IRP, richiama il gestore di I/O per inoltrare il pacchetto al driver successivo nello stack o, se l'elaborazione è terminata, per completare l'operazione sull'IRP.

Il completamento può avvenire in un contesto differente dalla richiesta originale di I/O. Per esempio, se un driver sta eseguendo la propria parte di un'operazione di I/O ed è obbligato a bloccarsi per lungo tempo, può accodare l'IRP a un thread di lavoro per proseguire l'elaborazione nel contesto del sistema. Al thread originale il driver restituisce un indicatore di stato per comunicare che la richiesta di I/O è in corso, in modo che possa continuare l'esecuzione in parallelo con l'operazione di I/O. I pacchetti IRP possono anche essere elaborati da procedure di servizio delle interruzioni; l'elaborazione può terminare in un contesto di processo arbitrario. Poiché una qualche elaborazione finale potrebbe essere necessaria nel contesto che ha iniziato l'I/O, il gestore di I/O usa una APC per eseguire l'elaborazione finale, a completamento dell'I/O, nel contesto del thread che ha dato l'avvio all'operazione.

Questo modello a stack è molto flessibile. Quando si costruisce uno stack, vari driver hanno l'opportunità di inserirsi nello stack come **driver filtro**. I driver filtro sono in grado di esaminare e potenzialmente modificare ciascuna operazione di I/O. La gestione del montaggio, delle partizioni, le operazioni di scomposizione in sezioni e mirroring del disco sono altrettanti esempi di funzionalità implementate usando i driver filtro che operano nello stack, al di sotto del file system. I driver filtro del file system agiscono al di sopra del file system e sono stati usati per realizzare funzionalità quali la gestione della memorizzazione gerarchica, la creazione di singole istanze

di file per l'avvio remoto e la conversione dinamica dei formati. Terze parti usano anche i driver filtro del file system per implementare il rilevamento dei virus.

I driver dei dispositivi di Windows sono scritti in conformità alle specifiche del “modello di driver di Windows” (*Windows driver model*, WDM). Esso stabilisce i requisiti del driver del dispositivo, incluse le modalità di stratificazione dei driver filtro, la condivisione del codice comune per gestire l'alimentazione e le richieste *plug-and-play*, la costruzione della corretta logica di cancellazione, e così via.

A causa della ricchezza del modello, scrivere un driver WDM completo per ogni periferica aggiunta può rivelarsi un compito improbo. Fortunatamente ciò non è necessario, grazie al cosiddetto modello porta/miniporta. Ciascuna istanza di un dispositivo che rientri in una classe di dispositivi simili, quali i driver audio, i dispositivi SATA e i controllori Ethernet, condivide un driver comune per quella classe, chiamato **driver della porta**. Il driver della porta implementa le operazioni standard per la classe di appartenenza e richiama poi le procedure specifiche del dispositivo nel **driver della miniporta** di quel dispositivo, al fine di realizzare le proprie funzionalità peculiari. Lo stack TCP/IP è implementato in questo modo, con il driver `ndis.sys` che implementa molte funzionalità dei driver di rete e chiama i driver delle miniporte di rete per lo specifico hardware.

Le versioni recenti di Windows, tra cui Windows 7, forniscono ulteriori semplificazioni per la scrittura di driver per dispositivi hardware. I driver in modalità kernel possono essere scritti utilizzando l'ambiente KMDF (*kernel-mode driver framework*) che fornisce un modello di programmazione semplificato per i driver su WDM. Un'altra opzione è l'ambiente UMDF (*user-mode driver framework*). Molti driver non hanno bisogno di operare in modalità kernel ed è più facile sviluppare e distribuire i driver in modalità utente. L'utilizzo di driver in modalità utente rende inoltre il sistema più affidabile, in quanto un errore di un driver in modalità utente non causa un crash in modalità kernel.

19.3.3.6 Gestore della cache

In molti sistemi operativi la funzionalità di cache è di pertinenza del file system. Windows, invece, offre un servizio di cache centralizzata in cui il **gestore della cache** opera in stretta collaborazione con il gestore della VM, per fornire servizi di cache a tutti i componenti del sistema sotto il controllo del gestore di I/O. Il caching in Windows è basata sui file, e non sui blocchi di basso livello. La dimensione della cache cambia dinamicamente, in base alla quantità di memoria libera disponibile nel sistema. Il gestore della cache mantiene un proprio working set piuttosto che condividere quello del processo di sistema. Il gestore della cache mappa i file nella memoria del kernel e quindi utilizza interfacce speciali verso il gestore della VM per aggiungere o tagliare pagine da questo working set privato.

La cache è divisa in blocchi da 256 KB, ognuno dei quali può contenere una vista di un file, cioè una parte del file mappata in memoria. Ciascun blocco della cache è descritto da un **blocco di controllo dell'indirizzo virtuale** (VACB), che memorizza l'indirizzo virtuale e l'offset della vista nel file, così come il numero di processi che

stanno usando la vista. I VACB risiedono in un singolo array mantenuto dal gestore della cache.

Quando il gestore dell'I/O riceve una richiesta di lettura di un file a livello utente invia un IRP allo stack di I/O per il volume su cui risiede il file. Per i file contrassegnati come memorizzabili nella cache, il file system chiama il gestore della cache per cercare i dati richiesti nelle sue viste dei file nella cache. Il gestore della cache calcola quale elemento dell'array VACB di quel file corrisponda all'offset in byte codificato nella richiesta. O l'elemento in questione punta alla vista interessata nella cache, oppure è nullo; in tal caso, il gestore della cache aloca un blocco della cache, insieme all'elemento corrispondente dell'array VACB, e mappa la vista nel nuovo blocco della cache. Esso tenta poi di copiare i dati dal file mappato al buffer del chiamante; se la copia riesce, l'operazione è completata.

Se la copia fallisce, ciò avviene a causa di una pagina mancante, che spinge il gestore della VM a inviare al gestore di I/O una richiesta di lettura con esplicita esclusione della cache; questi trasmette un'ulteriore richiesta allo stack dei driver del dispositivo per ottenere, questa volta, un'operazione di *paginazione*, che esclude il gestore della cache e legge i dati direttamente dal file nella pagina allocata per il gestore della cache: completata l'operazione, il VACB è impostato in modo da puntare a quella pagina. I dati, ora nella cache, sono copiati nel buffer del chiamante, cosa che completa l'originaria richiesta di I/O. La Figura 19.6 riassume graficamente queste operazioni.

Un'operazione di lettura a livello del kernel si svolge in modo simile, a eccezione del fatto che si può accedere ai dati direttamente dalla cache, anziché copiarli in un buffer nello spazio dell'utente. Per leggere i metadati del file system – le strutture dati che descrivono il file system – il kernel si serve dell'interfaccia del gestore della cache

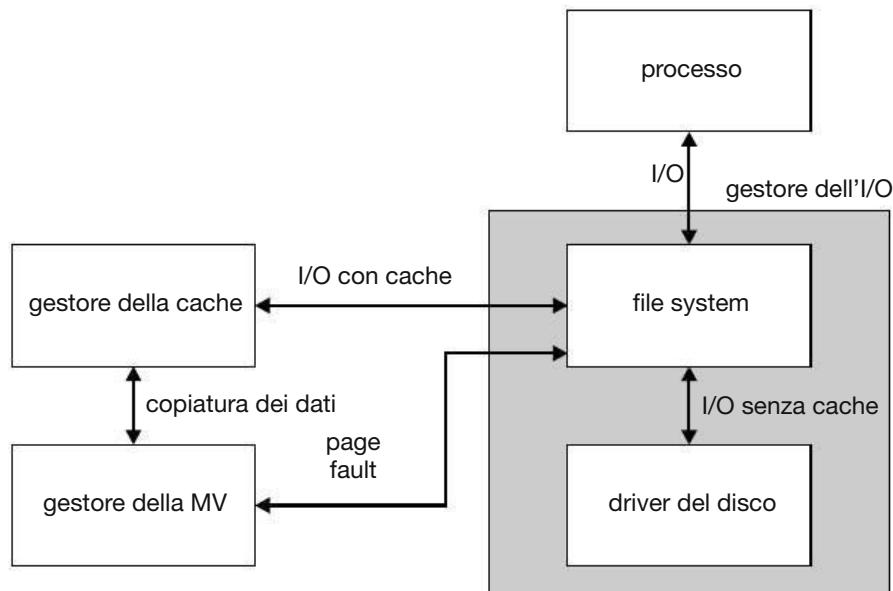


Figura 19.6 I/O con file

dedicata alla mappatura; per modificarli, usa invece l’interfaccia del gestore della cache per ancorare le pagine. Il **pinning** (cioè l’*ancoraggio*) di una pagina significa associarla a un frame della memoria fisica, in modo che il gestore della VM non possa spostarla o scaricarla. Dopo l’aggiornamento dei metadati, il file system richiede al gestore della cache di liberare la pagina ancorata. Una pagina modificata viene etichettata come *dirty*, sicché il gestore della VM ricopia la pagina su disco.

Per migliorare le prestazioni, il gestore della cache mantiene un piccolo archivio delle richieste di lettura, con cui cerca di predire le richieste successive. Se il gestore rileva uno schema predicibile nelle tre precedenti richieste, per esempio un accesso sequenziale in avanti o all’indietro, trasferisce i dati nella cache in anticipo, prima che giunga la successiva richiesta da parte dell’applicazione. In questo modo l’applicazione può trovare i propri dati già nella cache senza dover aspettare l’I/O del disco.

Altro compito del gestore della cache è di ordinare al gestore della VM lo svuotamento del contenuto della cache. Per default, il gestore della cache opera secondo la tecnica della scrittura differita (*write-back caching*): accumula scritture per 4-5 secondi e poi risveglia il thread di trasferimento su disco. Quando l’operazione deve invece seguire la tecnica della scrittura diretta (*write-through policy*), un processo può segnalarlo, impostando un flag all’apertura del file, o può richiamare un’esplicita funzione di svuotamento della cache.

Un processo di scrittura veloce potrebbe arrivare a riempire tutte le pagine libere della cache prima che il thread di trasferimento abbia la possibilità di attivarsi per scaricare le pagine su disco; il thread previene tale eventualità nel modo seguente. Quando la quantità di memoria libera nella cache si riduce, il gestore della cache sospende temporaneamente i processi che tentano di scrivere dati e risveglia il thread di trasferimento per scaricare le pagine su disco. Se il processo di scrittura implementa in realtà un reindirizzamento per un file system di rete, congelarlo troppo a lungo potrebbe far scadere i trasferimenti di rete e indurre la ritrasmissione dei dati; pertanto, si sprecherebbe banda di rete. Per impedire tale spreco, i reindirizzatori di rete possono chiedere al gestore della cache di limitare le scritture accumulate nella cache.

Poiché un file system di rete necessita di trasferire i dati fra un disco e l’interfaccia di rete, il gestore della cache fornisce anche un’interfaccia DMA per spostare i dati direttamente. Il trasferimento diretto evita la duplicazione dei dati attraverso un buffer intermedio.

19.3.3.7 Monitor della sicurezza dei riferimenti

Grazie alla centralizzazione della gestione delle entità del sistema nel gestore degli oggetti, Windows può realizzare un meccanismo uniforme per la validazione runtime degli accessi e per le verifiche di sicurezza per ogni elemento del sistema accessibile dagli utenti. Ognqualvolta un processo apre un oggetto, il **monitor della sicurezza dei riferimenti** (*security reference monitor*, SRM) controlla il token di sicurezza del processo e la lista di controllo degli accessi dell’oggetto al fine di determinare se il processo ha i diritti necessari.

L’SRM è anche responsabile della manipolazione dei privilegi nei token di sicurezza. Gli utenti che debbano eseguire backup o operazioni di ripristino del file

system, che debbano eseguire il debug di processi, e così via, necessitano di privilegi speciali. È anche possibile impostare un token di modo che neghi l'accesso a elementi solitamente disponibili per la gran parte degli utenti; ciò è utile soprattutto per limitare i danni che l'esecuzione di codice di provenienza incerta potrebbe causare.

Il livello di integrità del codice in esecuzione in un processo è rappresentato anche da un token. I livelli di integrità sono un meccanismo per gestire le capability di un processo, come accennato in precedenza. Un processo non può modificare un oggetto con un livello di integrità superiore a quello del codice in esecuzione nel processo, indipendentemente dalle altre autorizzazioni concesse. I livelli di integrità sono stati introdotti per rendere più difficile l'acquisizione del controllo sul sistema da parte di codice che attacca con successo il software in grado di collegarsi verso l'esterno, come Internet Explorer.

Un altro compito dell'SRM è la registrazione su log degli eventi di verifica della sicurezza. Il **Common Criteria** del Dipartimento della Difesa americano (il successore dell'Orange Book, pubblicato nel 2005) richiede che un sistema sicuro debba poter rilevare e registrare tutti i tentativi d'accesso alle proprie risorse, al fine di facilitare l'identificazione dei tentativi d'accesso non autorizzato. Poiché l'SRM è responsabile dei controlli d'accesso, esso genera la gran parte delle informazioni di verifica nel log della sicurezza.

19.3.3.8 Gestore del plug-and-play

Il sistema operativo usa il **gestore plug-and-play (PnP)** per riconoscere eventuali cambiamenti nella configurazione hardware del calcolatore e adattarvi opportunamente il sistema. I dispositivi PnP utilizzano protocolli standard per farsi identificare dal sistema. Il gestore PnP riconosce automaticamente i dispositivi installati e mentre il sistema è funzionante rileva eventuali cambiamenti nei dispositivi. Il gestore tiene anche traccia delle risorse impiegate da ciascun dispositivo, insieme con le risorse che potenzialmente potrebbero essere utilizzate e si occupa del caricamento dei driver necessari. Questa gestione delle risorse fisiche (che sono principalmente le interruzioni e gli intervalli di memoria per l'I/O) ha l'obiettivo d'identificare una configurazione hardware che permetta a tutti i dispositivi di funzionare correttamente.

Il gestore PnP opera la riconfigurazione dinamica come segue. Innanzitutto ottiene una lista di dispositivi da ciascun driver di bus (per esempio, PCI, USB). Successivamente carica il driver installato (se è necessario, dopo averne trovato uno) e invia un comando `add-device` al driver appropriato per ciascun dispositivo. Il gestore PnP trova poi le assegnazioni ottimali delle risorse e invia un comando `start-device` a ciascun driver per specificare l'assegnamento delle risorse per quel dispositivo. Se un dispositivo deve essere riconfigurato il gestore PnP invia un comando `query-stop` che richiede al driver se il dispositivo può essere temporaneamente disabilitato. Se il driver può disabilitare il dispositivo, allora tutte le operazioni sospese vengono completate e non si può avviare alcuna nuova operazione. Successivamente, il gestore PnP invia un comando `stop`, dopodiché può riconfigurare il dispositivo con un altro comando `start-device`.

Il gestore PnP prevede altri comandi, come `query-remove`, che si usa quando l’utente sta per rimuovere un dispositivo removibile, come una memoria USB, e che opera in modo simile a `query-stop`. Il comando `surprise-remove` si usa quando c’è un malfunzionamento di dispositivo, oppure, più comunemente, quando un utente estrae un dispositivo senza prima comunicarlo al sistema. Il comando `remove`, infine, richiede che il driver smetta di usare il dispositivo in maniere permanente.

Molti programmi presenti nel sistema sono interessati all’aggiunta o alla rimozione dei dispositivi e per questa ragione il gestore PnP offre funzionalità di notifica. Per esempio, una notifica fornisce ai menu dei file della GUI le informazioni di cui hanno bisogno per aggiornare l’elenco dei volumi del disco quando un nuovo dispositivo di archiviazione viene collegato o rimosso. L’installazione di periferiche si traduce spesso in aggiunta di nuovi servizi ai processi `svchost.exe` nel sistema. In molti casi questi servizi vengono eseguiti ogni volta che il sistema si avvia e continuano a funzionare anche se il dispositivo originale non viene mai inserito nel sistema. Windows 7 ha introdotto un **meccanismo di attivazione dei servizi** nel gestore SCM (*service control manager*), che gestisce i servizi di sistema. Mediante questo meccanismo i servizi possono essere configurati per essere eseguiti solo quando SCM riceve una notifica dal gestore PnP che segnala che il dispositivo di interesse è stato aggiunto al sistema.

19.3.3.9 Gestore dell’alimentazione

Windows collabora con l’hardware per implementare sofisticate strategie per l’efficienza energetica, come descritto nel Paragrafo 19.2.8. Le politiche che guidano queste strategie sono attuate dal **gestore dell’alimentazione** (*power manager*). Il gestore dell’alimentazione rileva le condizioni del sistema, come il carico della CPU o dei dispositivi di I/O, e migliora l’efficienza energetica riducendo le prestazioni e la reattività del sistema quando le richieste sono basse. Il gestore dell’alimentazione può anche mettere l’intero sistema nella modalità *sleep*, molto efficiente, o anche scrivere tutti i contenuti della memoria su disco e spegnere l’alimentazione per consentire al sistema di passare in uno stato di *sospensione*, o *ibernazione* (*hyibernation*).

Il vantaggio principale dello *sleep* è la rapidità con cui è possibile passare in questo stato: spesso sono sufficienti pochi secondi dopo aver chiuso il coperchio di un computer portatile. Anche il ripristino dallo *sleep* è abbastanza veloce. Nello stato di *sleep* viene tolta l’alimentazione a CPU e dispositivi di I/O, ma la memoria continua a essere alimentata a sufficienza per non perderne il contenuto.

La sospensione richiede molto più tempo, perché l’intero contenuto della memoria deve essere trasferito sul disco prima che il sistema venga spento. Tuttavia, il fatto che il sistema sia realmente spento costituisce un vantaggio significativo. Se l’alimentazione del sistema viene persa, per esempio quando viene sostituita la batteria su un computer portatile o quando un sistema desktop viene scollegato dalla rete elettrica, i dati memorizzati non verranno persi. A differenza dello spegnimento, la sospensione salva il sistema attualmente in uso, in modo che un utente possa riprendere da dove aveva lasciato. Poiché la sospensione non richiede alimentazione, un sistema è in grado di rimanere in questo stato per un tempo indefinito.

Come il gestore PnP, il gestore dell'alimentazione fornisce notifiche al resto del sistema sui cambiamenti nello stato dell'alimentazione. Alcune applicazioni vogliono sapere quando il sistema è in procinto di essere chiuso in modo che possano iniziare a memorizzare su disco il proprio stato.

19.3.3.10 Registro di sistema

Windows mantiene molte delle sue informazioni di configurazione nei database interni, chiamati hive, che sono gestiti dal gestore della configurazione di Windows, comunemente noto come il registro di sistema (*registry*). Vi sono hive separati per le informazioni di sistema, le preferenze predefinite dell'utente, l'installazione del software, la sicurezza e le opzioni di avvio. Le informazioni nell'hive **di sistema** servono all'avvio: per questo motivo, il gestore del registry è implementato come componente dell'executive.

Il registry rappresenta lo stato di configurazione in ogni hive come uno spazio dei nomi gerarchico di chiavi (directory), ognuna delle quali può contenere un insieme di valori tipizzati, come una stringa UNICODE, una stringa ANSI, un intero o dati binari non tipizzati. In teoria, le nuove chiavi e i valori vengono creati e inizializzati all'installazione di un nuovo software e vengono poi modificati per riflettere le modifiche nella configurazione di tale software. In pratica, il registry viene spesso usato come un database generico, come un meccanismo di comunicazione tra processi e per molti altri tali scopi.

Riavviare le applicazioni, o addirittura l'intero sistema, ogni volta che viene apportata una modifica nella configurazione sarebbe una seccatura. I programmi si basano allora su vari tipi di notifiche, come quelli forniti dal gestore PnP e dal gestore dell'alimentazione, per venire a conoscenza dei cambiamenti nella configurazione del sistema. Anche il registry fornisce notifiche che permettono ai thread di essere avvisati quando vengono apportate modifiche a una parte del registro di sistema. I thread possono così rilevare i cambiamenti di configurazione apportati al registry e adattarvisi.

Qualora vengano apportate modifiche significative al sistema, per esempio quando vengono installati aggiornamenti del sistema operativo o dei driver, vi è il pericolo che i dati di configurazione siano danneggiati (per esempio, se un driver funzionante viene sostituito da un driver non funzionante o se l'installazione di un'applicazione non riesce correttamente e lascia informazioni parziali nel registry). Prima di apportare questo genere di modifiche Windows crea un punto di ripristino. Il punto di ripristino contiene una copia degli hive prima del cambiamento e può essere utilizzato per ritornare alla vecchia versione degli hive permettendo a un sistema danneggiato di funzionare di nuovo.

Per migliorare la stabilità della configurazione del registry Windows ha aggiunto, a partire da Windows Vista, un meccanismo di transazione che può essere utilizzato per prevenire che il registry venga aggiornato parzialmente con un insieme di modifiche di configurazione correlate. Le transazioni del registry possono essere parte di transazioni più generali amministrate dal **gestore delle transazioni del kernel** (*kernel*

transaction manager, KTM), che può anche includere transazioni del file system. Le transazioni KTM non hanno la semantica completa delle transazioni nei normali database e non hanno soppiantato il ripristino del sistema nel recupero da danni alla configurazione del registry causati dall'installazione del software.

19.3.3.11 Avvio

L'avvio di un PC Windows inizia quando si fornisce alimentazione all'hardware e il firmware contenuto nella ROM è eseguito. In macchine più vecchie questo firmware era conosciuto come il BIOS, ma i sistemi più moderni utilizzano UEFI (Unified Extensible Firmware Interface), più veloce e più generale e consente un migliore utilizzo delle caratteristiche dei processori moderni. Il firmware esegue la diagnostica **POST** (*power-on self-test*), identifica molti dei dispositivi collegati al sistema e li inizializza a uno stato di accensione e poi costruisce la descrizione utilizzata dalla interfaccia avanzata di configurazione e alimentazione (ACPI). Successivamente, il firmware trova il disco di sistema, carica il programma `bootmgr` di Windows e ne inizia l'esecuzione.

Su una macchina che è stata sospesa viene successivamente caricato il programma `winresume` che ripristina dal disco il sistema in esecuzione, in modo che il sistema possa continuare dal punto in cui era prima della sospensione. Su una macchina che è stata arrestata, il `bootmgr` esegue un'ulteriore inizializzazione del sistema e quindi carica `WinLoad`, che a sua volta carica `hal.dll`, il kernel (`ntoskrnl.exe`), tutti i driver necessari all'avvio, e l'hive di sistema. `WinLoad` trasferisce poi l'esecuzione al kernel.

Il kernel si inizializza e crea due processi: il **processo di sistema** contiene tutti i thread di lavoro interni al kernel e non esegue mai in modalità utente; SMSS (*session manager subsystem*, ossia “sottosistema per la gestione della sessione”), il primo processo utente creato, è simile al processo INIT (da *inizializzazione*) di UNIX. Il processo SMSS prosegue l'inizializzazione del sistema, tra l'altro instaurando i file di pagina-zione, caricando altri driver dei dispositivi e gestendo le sessioni di Windows. Ogni sessione è utilizzata per rappresentare un utente connesso a eccezione della sessione 0, utilizzata per eseguire alcuni servizi di sistema in background, come LSASS e SERVICES. Una sessione viene ancorata da un'istanza del processo CSRSS. Ogni sessione diversa dalla sessione 0 esegue inizialmente il processo WINLOGON, che regista un utente e poi avvia il processo EXPLORER, che implementa la GUI di Windows. Il seguente elenco riporta alcuni degli aspetti della fase di avvio.

- SMSS completa l'inizializzazione del sistema e quindi avvia la sessione 0 e la prima sessione di login.
- WININIT viene eseguito nella sessione 0 per inizializzare la modalità utente e avviare LSASS, SERVICES e LSM, il gestore della sessione locale.
- LSASS, il sottosistema di sicurezza, implementa servizi quali l'autenticazione degli utenti.

- SERVICES contiene SCM, che supervisiona tutte le attività in background del sistema, compresi i servizi in modalità utente. Una serie di servizi saranno registrati per essere eseguiti all'avvio del sistema, mentre altri saranno avviati solo su richiesta o quando innescati da un evento come l'inserimento di un dispositivo.
- CSRSS è il processo del sottosistema dell'ambiente Win32. Viene avviato in ogni sessione, a differenza del sottosistema POSIX che viene avviato solo su richiesta quando viene creato un processo POSIX.
- WINLOGON viene eseguito in ogni sessione di Windows diversa dalla sessione 0 per registrare l'accesso di un utente.

Il sistema ottimizza le procedure d'avvio caricando preventivamente da disco alcuni file sulla base delle sessioni precedenti. La storia degli accessi al disco durante l'avvio serve anche per collocare i file di sistema su disco in modo da ridurre le operazioni di I/O richieste. Il numero di processi necessari all'avvio è ridotto raggruppando più servizi in un numero minore di processi. Tutti questi accorgimenti conferiscono un notevole contenimento dei tempi d'avvio del sistema, ma va anche sottolineato che i tempi d'avvio sono oggi meno importanti a causa delle funzionalità di sospensione e di sleep di Windows.

19.4 Terminal services e cambio rapido utente

Windows fornisce una console con interfaccia grafica che si interfaccia con l'utente tramite tastiera, mouse e monitor. La maggior parte dei sistemi supporta anche l'audio e il video. L'ingresso audio è utilizzato dal software di riconoscimento vocale di Windows, che rende il sistema più comodo e agevola la sua accessibilità per gli utenti disabili. Windows 7 ha aggiunto il supporto per l'hardware **multi-touch**, consentendo agli utenti di inserire dati toccando lo schermo e mediante i movimenti di uno o più dita. Infine, l'ingresso video attualmente utilizzato per applicazioni di comunicazione è suscettibile di utilizzo per l'interpretazione dei movimenti del viso, come Microsoft ha dimostrato con il suo prodotto Kinect per Xbox 360. Altre esperienze di input possono nascere in futuro dall'evoluzione del **surface computer** di Microsoft. Il surface computer, spesso installato in luoghi pubblici come alberghi e centri congressi, è una superficie piana dotata di speciali telecamere ed è in grado di monitorare le azioni di più utenti contemporaneamente e riconoscere gli oggetti che vengono posti sulla parte superiore.

Il PC è stato, ovviamente, pensato come un computer a uso personale, una macchina intrinsecamente a singolo utente. Le moderne versioni di Windows, tuttavia, supportano la condivisione di un PC tra più utenti. Ogni utente che è connesso mediante una GUI possiede una **sessione** creata per rappresentare l'ambiente GUI che verrà utilizzato e per contenere tutti i processi creati per eseguire le applicazioni. Windows consente la presenza di sessioni multiple allo stesso tempo su una singola macchina, ma supporta solo una console, costituita da tutti i monitor, le tastiere e i mouse collegati al PC. Una sola sessione alla volta può essere collegata alla console.

Dalla schermata di accesso visualizzata sulla console gli utenti possono creare nuove sessioni o connettersi a una sessione esistente creata in precedenza. Ciò consente a più utenti di condividere un singolo PC senza doversi disconnettere e riconnettere a ogni cambio di utente. Microsoft chiama questo uso delle sessioni **cambio rapido utente** (*fast user switching*).

Gli utenti possono anche creare nuove sessioni o connettersi a sessioni esistenti su un PC da una sessione in esecuzione su un altro PC Windows. Il terminal server (TS) collega una delle finestre GUI nella sessione locale di un utente alla sessione nuova o esistente, chiamata **desktop remoto**, sul computer remoto. L'uso più comune del desktop remoto è la connessione a una sessione sul proprio PC di lavoro dal proprio PC di casa.

Molte società utilizzano sistemi terminal server aziendali mantenuti in data center per eseguire tutte le sessioni utente che accedono alle risorse aziendali, piuttosto che permettere agli utenti di accedere a tali risorse dai PC presenti nell'ufficio di ciascun utente. Ogni computer server può gestire molte decine di sessioni di desktop remoto. In questi casi si parla di elaborazione **thin-client**, in cui i singoli computer si basano su un server per svolgere molte funzioni. Affidandosi ai terminal server dei data center si migliora l'affidabilità, la gestibilità e la sicurezza delle risorse informatiche aziendali.

Il TS è anche utilizzato da Windows per mettere in atto l'**assistenza remota**. Un utente remoto può essere invitato a condividere una sessione con l'utente connesso alla sessione sulla console. L'utente remoto può osservare le azioni dell'utente e ottenere il controllo del desktop per aiutarlo a risolvere eventuali problemi.

19.5 File system

Il file system nativo di Windows è NTFS, usato per tutti i volumi locali. Tuttavia, le memorie USB, le memorie flash delle fotocamere e i dischi esterni possono essere formattati con il file system FAT a 32 bit per questioni di portabilità. FAT è un formato di file system molto più vecchio ed è riconosciuto da molti sistemi diversi da Windows, come il software in esecuzione sulle macchine fotografiche. Uno degli svantaggi è che il file system FAT non limita l'accesso ai file agli utenti autorizzati. L'unica soluzione per la protezione dei dati con FAT consiste nell'eseguire un programma per crittografare i dati prima di memorizzarli nel file system.

Al contrario, NTFS utilizza ACL per controllare l'accesso ai singoli file e supporta la crittografia implicita di singoli file o di interi volumi (utilizzando la funzionalità BitLocker di Windows). NTFS implementa molte altre caratteristiche, tra cui il recupero dei dati, la tolleranza ai guasti, i file e i file system di grandi dimensioni, i flussi multipli di dati, i nomi Unicode, i file sparsi, il journaling, le copie ombra dei volumi e la compressione dei file.

19.5.1 Struttura interna di NTFS

L'entità fondamentale dell'NTFS è il volume: si crea con l'utilità di amministrazione dei dischi del sistema operativo ed è basato su una partizione logica del disco. Un volume può occupare parte di un disco, un intero disco, o anche più dischi.

L'NTFS non tratta direttamente i singoli settori dei dischi, ma usa **cluster** (*unità di assegnazione*) costituiti da un numero di settori pari a una potenza di 2. La dimensione di un cluster si stabilisce nella fase di formattazione di un file system. La dimensione di default di un cluster si basa sulla dimensione del volume ed è pari a 4 KB per volumi più grandi di 2 GB. Vista la dimensione degli attuali dischi, ha senso utilizzare una dimensione di cluster maggiore di quella di default per raggiungere prestazioni migliori, anche se il vantaggio che si ottiene va a scapito di una maggior frammentazione interna.

Come indirizzi relativi al disco l'NTFS usa **numeri di cluster logici** (*logical cluster number, LCN*), che assegna numerando i cluster dall'inizio alla fine del disco. Il sistema può quindi calcolare un offset (espresso in byte) relativo al disco fisico moltiplicando l'LCN per la dimensione del cluster.

Un file dell'NTFS non è una semplice sequenza di byte come in UNIX; è invece un oggetto strutturato costituito da **attributi**. Ciascun attributo di un file è una sequenza indipendente di byte che può essere creata, eliminata, letta e scritta. Alcuni attributi sono comuni a tutti i file, per esempio il nome (o i nomi, se il file ha degli alias come ad esempio un nome MS-DOS), l'ora di creazione, e il descrittore di sicurezza che stabilisce il controllo degli accessi. I dati dell'utente sono memorizzati in *attributi di dati*.

Molti dei tradizionali file di dati posseggono attributi di dati privi di nome contenenti tutti i dati del file. Si possono però creare altri flussi di dati con nomi espliciti. Per esempio, nei file Macintosh memorizzati sui server Windows, la *resource fork* è un flusso di dati con nome. Le interfacce IProp del modello COM impiegano un flusso di dati con nome per memorizzare proprietà su file ordinari, comprese le anteprime delle immagini. In generale, si possono aggiungere attributi secondo necessità; vi si accede seguendo la sintassi *nomefile:attributo*. NTFS restituisce solo la dimensione dell'attributo privo di nome in risposta a richieste di informazioni sui file, per esempio quando si esegue il comando `dir`.

Ogni file dell'NTFS è descritto da uno o più elementi contenuti in un array memorizzato in un file speciale detto **tabella principale dei file** (*master file table, MFT*); la dimensione di uno di questi elementi è determinata al momento della creazione del file system, e varia da 1 KB a 4 KB. Gli attributi di piccole dimensioni sono memorizzati nella MFT stessa, e sono detti **attributi residenti**; gli attributi più grandi, per esempio la massa anonima dei dati, sono invece **attributi non residenti** e sono memorizzati in una o più **estensioni contigue** (*extent*) nei dischi, e un puntatore a ogni estensione è memorizzato nell'MFT. Se un file è molto piccolo anche l'attributo dei dati si può memorizzare nell'elemento dell'MFT, mentre se un file possiede molti attributi o se è molto frammentato, e richiede quindi molti puntatori per individuare tutti i frammenti, un solo elemento nella MFT potrebbe essere insufficiente. In questo

caso il file è descritto da un **elemento di base del file** che punta a elementi aggiuntivi contenenti altri puntatori e attributi.

Ogni file in un volume NTFS possiede un unico identificatore detto **riferimento al file** (*file reference*); si tratta di 64 bit che consistono di un numero del file di 48 bit e di un numero di sequenza di 16 bit. Il numero del file è il numero dell'elemento (cioè l'indice dell'array) nella MFT che descrive il file; il numero di sequenza viene incrementato ogni volta che si riutilizza un elemento della MFT. Ciò permette all'NTFS di eseguire controlli interni di coerenza, per esempio per rilevare un riferimento a un file cancellato dopo che l'elemento della MFT è stato riusato per un nuovo file.

19.5.1.1 Albero B+ di NTFS

Lo spazio dei nomi dell'NTFS, come in UNIX, è organizzato come una gerarchia di directory. Ogni directory usa una struttura dati detta **albero B+** per memorizzare un indice dei nomi dei file contenuti nella directory; in un albero B+ la lunghezza di ogni percorso dalla radice a una foglia è la stessa e viene eliminato il costo della riorganizzazione dell'albero. La **radice indice** di una directory contiene il livello più alto di un albero B+; nel caso di una directory di grandi dimensioni, questo livello contiene i puntatori alle estensioni del disco nelle quali è memorizzato il resto dell'albero. Ogni elemento della directory contiene il nome del file e il suo riferimento, così come una copia dell'ora dell'ultimo aggiornamento e della dimensione del file presa dagli attributi residenti nella MFT; il motivo per cui le copie di queste informazioni sono memorizzate nella directory è che in questo modo è più efficiente elencarne i contenuti: tutti i nomi, le dimensioni e gli orari sono disponibili all'interno della directory stessa, cosicché non c'è bisogno di recuperare questi attributi dagli elementi della MFT per ognuno dei file.

19.5.1.2 Metadati di NTFS

I metadati di ogni volume dell'NTFS sono tutti memorizzati in alcuni file, il primo dei quali è proprio la MFT. Il secondo, usato durante la procedura di recupero dei dati a seguito del danneggiamento della MFT, contiene una copia dei primi 16 elementi della MFT. Anche un certo numero di file successivi è dedicato a compiti specifici, come descritto di seguito.

- Il **file di log** registra tutte le modifiche ai metadati del sistema.
- Il **file del volume** contiene il nome del volume, la versione di NTFS con cui il volume è stato formattato, e un bit che indica se il volume richiede un controllo di coerenza, con l'utilizzo del programma `chkdsk`, a seguito di un possibile guasto.
- La **tavella di definizione degli attributi** indica i tipi di attributi usati nel volume, e le operazioni da eseguire per ognuno di loro.
- La **directory radice** è la directory di livello più alto nella gerarchia del file system.
- Il **file bitmap** tiene traccia dei cluster allocati ai file e dei cluster liberi.

- Il **file d'avvio** contiene il codice d'avvio di Windows e deve risiedere a uno specifico indirizzo del disco in modo che sia facilmente reperibile da un semplice caricatore d'avvio collocato su ROM. Esso contiene inoltre l'indirizzo fisico della MFT.
- Il **file dei cluster guasti** tiene traccia delle aree del volume non funzionanti; NTFS sfrutta queste informazioni per il ripristino a seguito di errori.

Il mantenimento di tutti i metadati NTFS in veri e propri file offre una proprietà utile. Come discusso nel Paragrafo 19.3.3.6, il gestore della cache memorizza nella cache i dati contenuti nei file. Poiché tutti i metadati NTFS risiedono in file, questi dati possono essere memorizzati nella cache utilizzando gli stessi meccanismi che si impiegano per i dati comuni.

19.5.2 Ripristino

In molti semplici file system un calo di tensione al momento sbagliato può danneggiare le strutture dati del file system così gravemente da compromettere un intero volume. Molti file system di UNIX, incluso UFS, ma non ZFS, memorizzano nei dischi metadati ridondanti e tentano il ripristino dei dati a seguito di un crash del sistema usando il programma `fsck` per controllare tutte le strutture dati del file system e ripristinarne forzatamente uno stato coerente; si tratta di una procedura che spesso comporta l'eliminazione dei file danneggiati e il rilascio di cluster di dati su cui erano stati scritti dati degli utenti, ma che non erano stati correttamente registrati nelle strutture di metadati del file system: è un processo che può essere lento e può portare a significative perdite di dati.

Per conferire robustezza al file system l'NTFS adotta un orientamento differente: tutti gli aggiornamenti delle strutture dati del file system sono eseguiti all'interno di transazioni. Prima che una struttura dati sia modificata, la transazione scrive nel log le informazioni necessarie per ripetere o annullare l'operazione; dopo che la struttura dati è stata cambiata, la transazione scrive un'annotazione di conferma nel log per indicare il successo della transazione.

A seguito di un crash, il sistema è in grado di riportare le strutture dati del file system a uno stato coerente elaborando le informazioni contenute nel log, prima ripetendo le operazioni confermate, poi annullando le operazioni che non erano state confermate prima del crash. Periodicamente (di solito ogni 5 secondi) si scrive un elemento di controllo nel log: il sistema non fa uso degli elementi che precedono l'elemento di controllo al fine di ripristinare i dati dopo un crash; tali elementi si possono quindi eliminare in modo che il file contenente il log non cresca eccessivamente. La prima volta che si accede a un volume dopo l'avviamento del sistema, l'NTFS esegue automaticamente la procedura di ripristino.

Questa strategia non garantisce l'integrità di tutti i contenuti dei file degli utenti dopo un crash; assicura solo che le strutture dati del file system (cioè i file di metadati) siano integre e riflettano uno stato coerente precedente al crash. Sarebbe possibile estendere il metodo delle transazioni ai file degli utenti e Microsoft ha mosso i primi passi in questa direzione con Windows Vista.

Il log è memorizzato nel terzo file di metadati all'inizio del volume; viene creato nella fase di formattazione del file system e ha una dimensione massima fissata. È costituito da due parti: l'**area per il log**, che è una coda circolare di elementi, e l'**area di riavvio**, contenente informazioni contestuali quali il punto dell'area per il log dal quale l'NTFS dovrebbe cominciare la lettura durante una procedura di ripristino. In effetti l'area di riavvio contiene due copie di queste informazioni, in modo che il ripristino sia possibile se una copia è stata danneggiata durante il crash.

La funzione di logging è fornita dal *servizio di gestione del file di log*; oltre a scrivere gli elementi nel file di log e a eseguire operazioni di ripristino, questo servizio tiene traccia dello spazio libero nel file di log: quando esso si riduce eccessivamente il servizio di gestione del file di log accoda le transazioni inevase, e l'NTFS sospende tutte le nuove operazioni di I/O. Una volta che tutte le operazioni in corso sono state completate, l'NTFS attiva il gestore della cache per trasferire tutti i dati nei dischi, reimposta il file di log e, infine, esegue le transazioni in coda.

19.5.3 Sicurezza

La sicurezza di un volume dell'NTFS deriva dal modello a oggetti del sistema Windows. Ogni file NTFS fa riferimento a un descrittore di sicurezza, che specifica il proprietario del file, e a una lista di controllo degli accessi, che contiene le autorizzazioni di accesso concesse o negate a ciascun utente o gruppo elencato. Le prime versioni di NTFS utilizzavano un attributo descrittore di sicurezza separato per ciascun file. A partire da Windows 2000, l'attributo descrittore di sicurezza punta a una copia condivisa, con un notevole risparmio di spazio su disco e sulla cache, poiché numerosissimi file hanno descrittori di sicurezza identici.

Nel suo normale funzionamento, NTFS non fa rispettare i permessi nell'attraversamento delle directory dei nomi di percorso del file. Tuttavia, per garantire compatibilità con POSIX, questi controlli possono essere abilitati. I controlli sull'attraversamento sono intrinsecamente più onerosi, visto che la moderna analisi dei nomi di percorso dei file è basata sul matching dei prefissi anziché sull'analisi, directory per directory, dei nomi di percorso. Il matching dei prefissi è un algoritmo che cerca le stringhe in una cache e trova la voce con la corrispondenza più lunga. Per esempio, la voce `\foo\bar\dir` è una corrispondenza per `\foo\bar\dir2\dir3\myfile`. La cache di matching dei prefissi permette di iniziare l'attraversamento della struttura da una posizione più profonda, risparmiando diversi passaggi. Il rispetto dei controlli di attraversamento richiede che l'accesso dell'utente sia controllato a ogni livello di directory. Per esempio, un utente potrebbe non avere il permesso di attraversare `\foo\bar`, quindi iniziare dall'accesso a `\foo\bar\dir` sarebbe un errore.

19.5.4 Gestione dei volumi e tolleranza ai guasti

Il driver di disco fault tolerant del sistema Windows si chiama `FtDisk`: una volta installato mette a disposizione molti modi di combinare diverse unità a disco in un unico volume logico, in modo da migliorare prestazioni, capacità di memorizzazione e affidabilità.

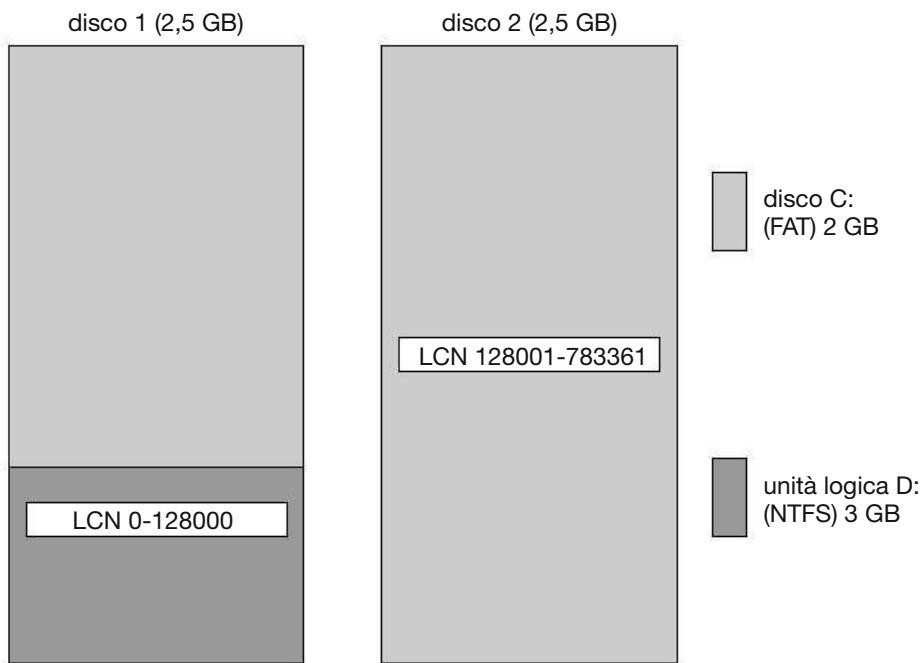


Figura 19.7 Insieme di volumi in due dischi.

19.5.4.1 Insiemi di volumi e insiemi RAID

Uno dei possibili modi di combinare dischi diversi è di concatenarli logicamente in modo da formare un solo volume logico, com'è illustrato dalla Figura 19.7. In Windows questo volume logico si chiama **insieme di volumi** (*volume set*) e consiste al massimo di 32 partizioni fisiche. Un insieme di volumi che contenga un volume dell'NTFS può essere esteso senza modificare i dati già memorizzati all'interno del file system: i metadati del file della bitmap del volume dell'NTFS sono semplicemente modificati al fine di includere lo spazio che si desidera aggiungere. L'NTFS continua a usare lo stesso meccanismo LCN che usa per un unico disco fisico, e il driver `FtDisk` fornisce le conversioni degli offset riferiti a un volume logico in offset riferiti a uno specifico disco.

Un altro modo di combinare più partizioni fisiche è di intercalare i loro blocchi per formare un cosiddetto **insieme di sezioni** (*stripe set*). Si tratta di un metodo noto anche come RAID di livello 0 o **sezionamento del disco** (per maggiori informazioni sul RAID si veda il Paragrafo 10.7 del Capitolo 10). Il driver `FtDisk` usa sezioni di 64 KB: i primi 64 KB del volume logico sono memorizzati nella prima partizione fisica, i secondi 64 KB nella seconda partizione, e così via finché ogni partizione non abbia contribuito con 64 KB di spazio; a questo punto il processo di allocazione riprende dal primo disco, che contribuisce con un secondo blocco di 64 KB. Un insieme di sezioni costituisce un solo ampio volume logico, ma l'effettiva configurazione fisica può migliorare l'ampiezza di banda dell'I/O, perché per trasferimenti ingenti tutti i dischi possono trasferire dati in parallelo. Windows supporta anche il RAID di livello 5, gli insiemi di sezioni con parità e il RAID di livello 1 (copia speculare o mirroring).

19.5.4.2 Accantonamento dei settori e rimappatura dei cluster

Per gestire il malfunzionamento dei settori F_tDisk si serve di una funzione hardware detta **accantonamento di settori**, e l'NTFS utilizza una tecnica software detta *rimappatura dei cluster*. L'accantonamento di settori è una funzionalità hardware fornita da molte unità disco: quando si formatta un disco, l'unità crea una corrispondenza fra i numeri dei blocchi logici e i settori fisici funzionanti del disco; essa accanta alcuni settori lasciandoli volutamente inutilizzati, in modo che se un settore diviene difettoso, F_tDisk può richiedere all'unità di sostituirlo con uno di loro. La **rimappatura dei cluster** è una procedura messa in atto dal file system: se un blocco del disco diventa difettoso, l'NTFS lo sostituisce con un blocco libero differente cambiando i puntatori interessati nell'MFT; l'NTFS inoltre annota che il blocco malfunzionante non dovrà più essere allocato.

L'usuale conseguenza del malfunzionamento di un blocco è la perdita di dati; la rimappatura dei cluster e l'accantonamento di settori, però, si possono combinare con i volumi tolleranti i guasti per mascherare il guasto di un blocco. Se una lettura non riesce, il sistema ricostruisce i dati mancanti leggendo l'immagine speculare o calcolando la parità nel caso di un insieme di sezioni con parità; i dati così ricostruiti si memorizzano in una nuova locazione del disco ottenuta grazie alla rimappatura dei cluster o all'accantonamento di settori.

19.5.5 Compressione

L'NTFS è in grado di eseguire la compressione dei dati di un singolo file o di tutti i file di dati di un'intera directory. Per comprimere un file, divide i dati in **unità di compressione**, cioè blocchi di 16 cluster contigui. Al momento della scrittura di ciascuna unità di compressione si applica un algoritmo di compressione dei dati; se il risultato occupa meno di 16 cluster, si scrive nei dischi la versione compressa. Durante la lettura l'NTFS è in grado di determinare se i dati sono stati compressi; in questo caso la lunghezza dell'unità di compressione memorizzata è inferiore ai 16 cluster; per migliorare le prestazioni durante la lettura di unità di compressione contigue, l'NTFS legge e decomprime in anticipo rispetto alle richieste dell'applicazione.

Nel caso di file sparsi o file contenenti prevalentemente zeri, l'NTFS adotta un'altra tecnica per risparmiare spazio: i cluster che contengono solo zeri non sono realmente assegnati o memorizzati nei dischi, ma si lasciano dei buchi (*gap*) nella sequenza dei numeri virtuali dei cluster memorizzati nell'elemento dell'MFT relativo al file. Se nella lettura di un file ritrova un buco nella sequenza in questione, l'NTFS riempie semplicemente di zeri la parte interessata del buffer per l'I/O del chiamante. Questa tecnica è adottata anche nel sistema operativo UNIX.

19.5.6 Punti di montaggio, collegamenti simbolici e collegamenti fisici

I punti di montaggio sono una forma di collegamento simbolico peculiare delle directory di NTFS, introdotti con Windows 2000. Essi offrono agli amministratori la possibilità di organizzare i volumi del disco in maniera più flessibile, rispetto all'uti-

lizzo di nomi globali (quali le lettere delle unità). I punti di montaggio sono realizzati come collegamento simbolico a dati associati contenenti il vero nome del volume. È presumibile che i punti di montaggio soppianteranno completamente le lettere delle unità, ma solo in seguito a una lunga transizione dovuta alla dipendenza di molte applicazioni dalle lettere delle unità.

Windows Vista ha introdotto il supporto a una forma più generale di collegamenti simbolici, simili a quelli che si trovano in UNIX. I collegamenti possono essere assoluti o relativi, possono puntare a oggetti che non esistono e possono puntare a file e directory, anche attraversando i volumi. NTFS supporta anche i collegamenti fisici (hard link), in cui un singolo file ha una voce in più di una directory sullo stesso volume.

19.5.7 Giornale delle modifiche

L'NTFS mantiene un giornale in cui descrive tutti i cambiamenti avvenuti nel file system. I servizi in modalità utente possono ricevere notifica delle modifiche intervenute e individuare in un secondo momento i file modificati, leggendo dal giornale. Il servizio di indicizzazione del contenuto usa il giornale delle modifiche per individuare i file che devono essere nuovamente indicizzati, mentre il servizio di replica dei file lo utilizza per identificare i file che devono essere replicati in rete.

19.5.8 Copie ombra dei volumi

Windows ha la capacità di portare un volume in uno stato conosciuto per poi creare una copia ombra (*shadow*), utilizzabile per creare copie di backup coerenti del volume. La stessa tecnica viene chiamata **istantanea** (*snapshot*) in altri file system. Si tratta di una variante della copiatura su scrittura: i blocchi modificati dopo la creazione di una copia ombra sono mantenuti nella forma originale su quest'ultima. Affinché la copia raggiunga uno stato coerente con il volume è richiesta la cooperazione delle applicazioni, dal momento che il sistema non può sapere quando i dati usati dall'applicazione siano in uno stato stabile a partire dal quale l'applicazione stessa può essere riavviata in modo sicuro.

La versione server di Windows usa copie ombra per rendere prontamente disponibili le vecchie versioni dei file memorizzate sui file server. Gli utenti possono così usufruire di documenti conformi alla versione originale memorizzata sui file server. Gli utenti possono avvalersi di questa caratteristica per recuperare file cancellati per errore, o semplicemente per consultare una versione precedente dei file, il tutto senza la necessità di un nastro contenente copie di riserva.

19.6 Servizi di rete

I servizi di rete di Windows operano sia secondo il modello client-server sia secondo il modello peer-to-peer; il sistema operativo fornisce anche strumenti per la gestione della rete. I componenti del sottosistema di networking permettono la trasmissione dei dati, la comunicazione fra processi, la condivisione dei file attraverso la rete e la possibilità di stampare impiegando stampanti remote.

19.6.1 Interfacce di rete

Nella descrizione dei servizi di rete di Windows si fa riferimento a due interfacce di rete interne, dette **NDIS** (*network device interface specification*) e **TDI** (*transport driver interface*). L’interfaccia NDIS fu sviluppata nel 1989 da Microsoft e da 3Com al fine di separare gli adattatori di rete dai protocolli di trasporto, in modo che gli uni potessero essere modificati senza che ciò avesse effetto sugli altri. L’NDIS costituisce l’interfaccia fra lo strato di collegamento dei dati e lo strato di rete nel modello OSI e permette a molti protocolli di funzionare alla presenza di diversi adattatori di rete. Analogamente, la TDI è l’interfaccia fra lo strato di trasporto (strato 4) e lo strato di sessione (strato 5) del modello OSI: essa permette a ogni componente dello strato di sessione di adoperare ogni meccanismo di trasporto disponibile. (Necessità simili hanno portato al cosiddetto meccanismo *stream* di UNIX.) L’interfaccia TDI può gestire sia il trasporto privo di connessione sia quello basato sulla connessione, ed è dotata di funzioni per la trasmissione di tutti i tipi di dati.

19.6.2 Protocolli

Nel sistema Windows i protocolli di trasporto sono realizzati come driver che si possono caricare e scaricare dinamicamente, anche se in pratica il sistema deve di solito essere riavviato a seguito di una di queste modifiche. Il sistema Windows dispone di diversi protocolli di networking. Di seguito se ne illustrano alcuni.

19.6.2.1 Protocollo SMB

Il protocollo **SMB** (*server message-block*) fu introdotto per la prima volta con l’MS-DOS 3.1 e si usa per trasmettere richieste di I/O sulla rete. Esso tratta quattro tipi di messaggi. I messaggi **Session control** sono comandi per l’instaurazione e la chiusura della connessione di un oggetto di ridirezione (*redirector*) con una risorsa condivisa del server. I messaggi **File** vengono usati da un redirector per accedere a file del server. I messaggi **Printer** sono usati per trasmettere dati a una coda di stampa remota e per ricevere informazioni sullo stato della stampa, mentre i messaggi **Message** si usano per comunicare con un altro sistema in rete. Una versione del protocollo SMB è stato pubblicato come Common Internet File System (CIFS) ed è disponibile su una serie di sistemi operativi.

19.6.2.2 Protocollo TCP/IP

La pila di protocolli TCP/IP che si usa nella rete Internet è diventata lo standard *de facto* per la comunicazione in rete. Il sistema operativo Windows usa i protocolli TCP/IP per mettere in comunicazione un’ampia gamma di sistemi operativi e di piattaforme. Il pacchetto TCP/IP di Windows comprende un protocollo per la gestione di rete SNMP (*simple network-management protocol*), il protocollo DHCP (*dynamic host-configuration protocol*) per la configurazione dinamica dei calcolatori che si connettono alla rete e il vecchio servizio WINS (*Windows Internet name service*) di risoluzione dei nomi. Windows Vista ha introdotto una nuova implementazione di TCP/IP che supporta sia IPv4 sia IPv6 nello stesso stack di rete. Questa nuova implemen-

tazione supporta anche l’offloading dello stack di rete su hardware avanzato, per ottenere prestazioni molto elevate sui server.

Windows fornisce un firewall software che limita le porte TCP utilizzabili dai programmi per la comunicazione di rete. I firewall di rete sono comunemente implementati nei router e costituiscono una misura di sicurezza molto importante. Avere un firewall integrato nel sistema operativo rende un router hardware inutile e fornisce una gestione più integrata e una maggior facilità di utilizzo.

19.6.2.3 Protocollo PPTP

Il protocollo **PPTP** (*point-to-point tunneling protocol*) è un protocollo messo a disposizione dal sistema Windows per la comunicazione fra moduli server di accesso remoto residenti in calcolatori server Windows e altri sistemi client connessi alla rete Internet; i server possono cifrare i dati trasmessi, e gestiscono **reti private virtuali** (*virtual private network*, VPN) multiprotocollo nella rete Internet.

19.6.2.4 Protocollo HTTP

Il protocollo HTTP è utilizzato per avere (`get`) o fornire (`put`) informazioni utilizzando il World Wide Web. Windows implementa HTTP utilizzando un driver in modalità kernel, quindi i server web sono in grado di funzionare con un basso overhead di connessione allo stack di rete. HTTP è un protocollo abbastanza generale che Windows rende disponibile come opzione di trasporto per l’implementazione di RPC.

19.6.2.5 Protocollo per la realizzazione e lo sviluppo distribuiti di contenuti

WebDAV (*web distributed authoring and versioning*) è un protocollo, basato su HTTP, per la realizzazione e lo sviluppo coordinato di contenuti in rete. Windows installa all’interno del file system un oggetto di ridirezione WebDAV; ciò permette al protocollo in questione di cooperare con altre funzionalità del file system, quali la crittografia. I file personali possono in tal modo essere memorizzati con sicurezza in un luogo pubblico. Visto che WebDAV utilizza HTTP, che è un protocollo `get/put`, Windows deve memorizzare localmente sulla cache i file in modo che i programmi possano usare operazioni di `read` e `write` su parti di essi.

19.6.2.6 Pipe con nome

Le **pipe con nome** sono un meccanismo di trasmissione dei messaggi. Un processo può utilizzare le pipe con nome per comunicare con altri processi sulla stessa macchina. Dal momento che le pipe con nome sono accessibili tramite l’interfaccia del file system, i meccanismi di sicurezza utilizzati per oggetti file si applicano anche alle pipe con nome. Il protocollo SMB supporta le pipe con nome, che possono dunque essere utilizzate anche per la comunicazione tra processi su sistemi diversi.

Il nome di una pipe con nome segue una convenzione detta **UNC** (*uniform naming convention*). Un nome UNC si presenta in modo simile a un tipico nome di file remoto.

Il suo formato è \\nome_del_server\name_della_condivisione\x\y\z, dove nome_del_server identifica un server della rete; nome_della_condivisione specifica qualsiasi risorsa resa disponibili agli utenti della rete, per esempio directory, file, pipe con nome e stampanti, e la parte terminale \x\y\z è un ordinario nome di percorso di un file.

19.6.2.7 RPC

Una RPC (*remote procedure call*) è un meccanismo di comunicazione client-server che permette a un'applicazione residente in una certa macchina di eseguire una chiamata di procedura relativa a codice residente in un'altra macchina. Quando il client invoca una procedura remota, il sistema delle RPC richiama una procedura locale detta **stub** che impacca i suoi argomenti in un messaggio e li invia tramite la rete a un determinato processo server, dopo di che si blocca. Nel frattempo, il server estrae gli argomenti dal messaggio, richiama la procedura, impacca i risultati in un messaggio, e li spedisce allo stub del client; quest'ultimo riprende l'elaborazione, riceve il messaggio, estrae i risultati della RPC e li restituisce al chiamante. Impaccamento ed estrazione degli argomenti sono a volte chiamati **marshaling**. Il codice dello stub e i descrittori necessari per impacchettare e spacchettare gli argomenti per una RPC sono compilati da una specifica scritta in MIDL (Microsoft Interface Definition Language).

La funzione RPC di Windows aderisce al diffuso standard RPC Distributed Computing Environment, cosicché i programmi che usano la RPC di questo sistema sono facilmente adattabili ad altri sistemi. Lo standard RPC è dettagliato: nasconde molte differenze esistenti fra le architetture dei calcolatori, per esempio la dimensione dei numeri binari e l'ordine dei bit e dei byte nelle parole, specificando formati convenzionali dei dati per i messaggi RPC.

19.6.2.8 Modello COM

Il **modello COM** (*component object model*) è un meccanismo per la comunicazione fra processi originariamente sviluppato per Windows. Gli oggetti COM mettono a disposizione una precisa interfaccia per la manipolazione dei loro dati. A titolo d'esempio, COM funge da infrastruttura per la realizzazione della tecnologia Microsoft **OLE** (*object linking and embedding*, ossia “collegamento e integrazione degli oggetti”), grazie alla quale è possibile inserire fogli di calcolo nei documenti Word. Diversi servizi Windows forniscono interfacce COM. Windows è dotato di un'estensione distribuita del modello COM nota come **DCOM**, che permette lo sviluppo trasparente di applicazioni distribuite su una rete grazie all'uso di RPC.

19.6.3 Redirector e server

Nel sistema operativo Windows un'applicazione può usare l'API di I/O per accedere ai file di un altro calcolatore come se essi fossero locali, ammesso che sul computer remoto giri un server CIFS come quelli forniti da Windows. Un **oggetto di ridirezione** (*redirector*) è un oggetto lato client che trasmette richieste di I/O per file remoti, poi soddisfatte da un server. Per motivi legati alle prestazioni e alla sicurezza, il redirector e i server sono eseguiti in modalità kernel.

Più in dettaglio, l'accesso a un file remoto avviene come segue:

1. l'applicazione richiama il gestore dell'I/O per richiedere l'apertura di un file, fornendo un nome nel formato standard UNC;
2. il gestore dell'I/O costruisce un pacchetto di richiesta di I/O com'è descritto nel Paragrafo 19.3.3.5;
3. il gestore dell'I/O rileva che la richiesta si riferisce a un file remoto, e chiama un driver detto **MUP** (*multiple universal-naming-convention provider*);
4. il MUP invia in modo asincrono l'IRP a tutti i redirector registrati;
5. un redirector capace di soddisfare la richiesta risponde al MUP; per evitare di porre in futuro la stessa domanda a tutti i redirector, il MUP usa una cache per annotare l'identità del redirector capace di trattare questo file;
6. il redirector trasmette la richiesta sulla rete al sistema remoto;
7. i driver di rete del sistema remoto ricevono la richiesta e la passano al driver server;
8. il driver server passa la richiesta al driver appropriato del file system locale;
9. l'appropriato driver del dispositivo è chiamato per accedere ai dati;
10. i risultati sono restituiti al driver server, che li rispedisce al redirector richiedente, il quale li restituisce all'applicazione chiamante tramite il gestore dell'I/O.

Un processo simile avviene nel caso delle applicazioni che usano l'API di rete Win32 anziché i servizi UNC, in questo caso invece di un MUP si usa un modulo detto instradatore multiplo.

Per motivi legati alla portabilità il redirector e i server usano le API TDI per il trasporto di rete; le richieste stesse sono espresse per mezzo di un protocollo di più alto livello, che di solito è il protocollo SMB menzionato nel Paragrafo 19.6.2. L'elenco dei redirector è contenuto nell'hive di sistema del registro.

19.6.3.1 File system distribuito

I nomi UNC, che fanno esplicito riferimento al nome del server, non si dimostrano sempre appropriati, poiché vi può essere disponibilità di numerosi file server per gli stessi contenuti, e i nomi UNC includono esplicitamente il nome del server. Windows fornisce un protocollo di **file system distribuito** (*distributed file system, DFS*), grazie al quale un amministratore di rete può smistare i file da più server utilizzando un solo spazio dei nomi distribuito.

19.6.3.2 Reindirizzamento delle cartelle e caching lato client

Al fine di migliorare l'esperienza degli utenti che operano spesso su PC diversi per motivi professionali, Windows permette agli amministratori di assegnare agli utenti un **profilo mobile**, che mantiene su server le preferenze dell'utente e altre impostazioni. Per memorizzare automaticamente i documenti e gli altri file dell'utente su un server, si applica quindi il reindirizzamento delle cartelle.

Questa tecnica funziona a dovere finché uno dei calcolatori non è più collegato alla rete, come nel caso di un portatile su un aereo. Per fornire agli utenti un accesso non in linea ai loro file reindirizzati, Windows utilizza il **caching lato client** (*client-side caching, CSC*). Questo meccanismo è anche impiegato quando il computer è in linea per mantenere copie dei file del server sulla macchina locale, al fine di ottenere prestazioni migliori. I file sono inviati al server non appena cambiano e, se il calcolatore si disconnette, i file sono ancora disponibili; l'aggiornamento del server è rinviato alla volta successiva in cui il calcolatore tornerà in linea.

19.6.4 Domìni

Per molti ambienti di rete esistono gruppi naturali di utenti, per esempio gli studenti di un laboratorio scolastico o gli impiegati di un'azienda. Molto spesso si vuole che tutti i membri di un gruppo possano accedere a risorse condivise messe a disposizione dai calcolatori del gruppo. Per gestire i diritti d'accesso globale all'interno di un tale gruppo, il sistema Windows fa uso del concetto di dominio. In precedenza, questi domìni non avevano alcuna relazione con il DNS (*domain name system*) che trasformava nomi di calcolatori collegati alla rete Internet in indirizzi IP; ora invece sono strettamente correlati.

In particolare, un dominio Windows è un gruppo di stazioni di lavoro e server che condivide le politiche di sicurezza e ha una comune database degli utenti. Poiché il sistema Windows impiega per l'autenticazione e la fidatezza il protocollo Kerberos, un dominio Windows è quello che nel Kerberos si chiama *realm* (reame). Nel sistema Windows si segue un approccio gerarchico per stabilire relazioni di fiducia tra i domini collegati. Le relazioni di fiducia si basano sul DNS e sono relazioni transitive che si possono trasmettere in entrambe le direzioni nella gerarchia. Questo metodo riduce il numero di relazioni di fiducia richieste da $n * (n - 1)$ a $O(n)$. Le stazioni di lavoro nel dominio si fidano del controllore di dominio affinché dia informazioni corrette sui diritti d'accesso di ciascun utente (caricati nel token di accesso dell'utente da LSASS). Ogni utente si riserva comunque la possibilità di limitare l'accesso alle proprie stazioni di lavoro, indipendentemente dalle disposizioni del controllore il dominio.

19.6.5 Active Directory

Active Directory è lo strumento con cui Windows implementa i servizi del protocollo **LDAP** (*lightweight directory-access protocol*). Active Directory memorizza le informazioni di topologia relative ai domìni, gestisce i profili individuali e di gruppo degli utenti di un dominio, con le relative parole d'ordine e offre una memorizzazione nel dominio per funzionalità Windows che ne hanno bisogno, come i **criteri di gruppo** (*Windows group policy*).

Gli amministratori impiegano i criteri di gruppo per stabilire standard uniformi per le preferenze e il software dei desktop. Per molte aziende commerciali che si occupano di tecnologia dell'informazione, l'uniformità consente di ridurre drasticamente il costo dell'elaborazione.

19.7 Interfaccia di programmazione

L’API Win32 è l’interfaccia fondamentale ai servizi del sistema operativo Windows. Questo paragrafo descrive cinque aspetti chiave dell’API Win32: l’accesso agli oggetti del kernel, la condivisione degli oggetti fra i processi, la gestione dei processi, la comunicazione fra i processi e la gestione della memoria.

19.7.1 Accesso agli oggetti del kernel

Il kernel del sistema operativo Windows mette a disposizione dei programmi applicativi molti servizi: i programmi usufruiscono di questi servizi manipolando oggetti del kernel. Un processo accede a un oggetto del kernel di nome `xxx` chiamando la funzione `CreateXXX()` per ottenere un handle di un’istanza di `xxx`; questo handle è unico per il processo. Se la funzione `Create` non va a buon fine riporta il valore 0 o una costante specifica detta `INVALID_HANDLE_VALUE`, secondo l’oggetto che si sta aprendo. Un processo può chiudere un handle richiamando la funzione `CloseHandle()`, e il sistema può eliminare l’oggetto se il contatore che totalizza il numero degli handle che fanno riferimento all’oggetto in tutti i processi arriva a zero.

19.7.2 Condivisione degli oggetti tra processi

Il sistema Windows fornisce ai processi tre modi di condividere un oggetto. Il primo consiste nel fatto che un processo figlio può ereditare un handle dell’oggetto: quando invoca la funzione `CreateXXX()`, il genitore fornisce una struttura `SECURITIES_ATTRIBUTES` il cui campo `bInheritHandle` ha valore `TRUE`; questo campo crea un handle ereditabile. A questo punto si può creare il processo figlio, passando il valore `TRUE` all’argomento `bInheritHandle` della funzione `CreateProcess()`. Nella Figura 19.8 è mostrato un esempio di codice che crea l’handle di un semaforo ereditato dal processo figlio.

Nell’ipotesi che il processo figlio sappia quali handle sono condivisi, la condivisione degli oggetti permette di realizzare la comunicazione fra genitore e figlio; nel-

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
HANDLE a_semaphore = CreateSemaphore(&sa, 1, 1, NULL);
char command_line[132];
ostringstream ostring(command_line, sizeof(command_line));
ostringstream << a_semaphore << ends;
CreateProcess("another_process.exe", command_line,
NULL, NULL, TRUE, ...);
```

Figura 19.8 Codice che permette a un figlio di condividere un oggetto ereditandone l’handle.

l'esempio della Figura 19.8 il processo figlio otterrebbe il valore dell'handle dal primo argomento della riga di comando e potrebbe quindi condividere il semaforo con il processo genitore.

Il secondo metodo di condivisione degli oggetti presuppone che l'oggetto abbia ricevuto un nome dal processo che lo ha creato, cosicché un altro processo possa richiedere l'apertura dell'oggetto riferendosi al suo nome. Questa tecnica ha due svantaggi: il primo è che il sistema operativo Windows non fornisce un modo di controllare se un oggetto con un certo nome già esiste; il secondo è che lo spazio dei nomi degli oggetti è globale, senza alcuna distinzione relativa ai tipi di oggetti. Due applicazioni potrebbero per esempio creare un oggetto denominato *foo*, mentre ciò che si voleva erano due oggetti distinti e forse anche di diverso tipo.

Il vantaggio degli oggetti con nome è che processi non correlati possono facilmente condividerli: un primo processo richiamerebbe una delle funzioni `CreateXXX()`, e fornirebbe un nome come parametro; un secondo processo potrebbe ottenere un handle di questo oggetto chiamando `OpenXXX()` (o `CreateXXX()`) con lo stesso nome, com'è mostrato nella Figura 19.9.

Il terzo modo di condividere oggetti si basa sulla funzione `DuplicateHandle()`; esso richiede altri metodi di comunicazione fra processi per trasmettere l'handle duplicato. Se un processo possiede un handle con un certo valore, un altro processo può ottenerne uno dello stesso oggetto, e pertanto condividerlo; un esempio di questo metodo è mostrato nella Figura 19.10.

19.7.3 Gestione dei processi

Nel sistema Windows un **processo** è un'istanza caricata di un'applicazione e un **thread** è un'unità eseguibile di codice che può essere sottoposta a scheduling dal dispatcher del kernel: un processo contiene uno o più thread. Un processo viene creato quando un thread in qualche altro processo invoca l'API `CreateProcess()`, che carica le librerie dinamiche usate dal processo e crea un thread iniziale nel processo; tramite la funzione `CreateThread()` si possono creare thread aggiuntivi: ogni thread possiede il suo stack, la cui dimensione predefinita è di 1 MB se non è altrettanto specificato da un argomento di `CreateThread()`.

```
// processo A
...
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, "MySEM1");
...
// processo B
...
HANDLE b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
                                    FALSE, "MySEM1");
...
```

Figura 19.9 Codice per la condivisione di un oggetto per ricerca del nome.

```

// il processo A vuole fornire al processo B
// l'accesso a un semaforo
// processo A
HANDLE a_semaphore = CreateSemaphore(NULL,1,1,NULL);
// invia il valore del semaforo al processo B
// usando un messaggio o la condivisione della memoria
...
// processo B
HANDLE process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
                                 process_id_of_A);
HANDLE b_semaphore;
DuplicateHandle(process_a,a_semaphore,
                GetCurrentProcess(),&b_semaphore,0, FALSE,
                DUPLICATE_SAME_ACCESS);
// usa b_semaphore per accedere al semaforo

```

Figura 19.10 Codice per la condivisione di un oggetto tramite il passaggio di un handle.

19.7.3.1 Regola di scheduling

Le priorità nell'ambiente Win32 sono basate sul modello di scheduling del kernel nativo (NT), ma non tutti i valori di priorità possono essere scelti. L'ambiente Win32 usa quattro classi di priorità:

1. IDLE_PRIORITY_CLASS (livello di priorità 4)
2. NORMAL_PRIORITY_CLASS (livello di priorità 8)
3. HIGH_PRIORITY_CLASS (livello di priorità 13)
4. REALTIME_PRIORITY_CLASS (livello 24).

Ordinariamente i processi appartengono alla classe NORMAL_PRIORITY_CLASS, sempre che il genitore del processo non sia di classe IDLE_PRIORITY_CLASS, oppure non sia stata specificata un'altra classe al momento della chiamata alla funzione `CreateProcess()`. La classe di priorità di un processo diventa la classe predefinita per tutti i thread eseguiti nel processo. La classe di priorità si può modificare tramite la funzione `SetPriorityClass()`, o passando un argomento al comando `START`. Si noti che solo gli utenti che godono del privilegio che consente d'incrementare le priorità di scheduling possono assegnare un processo alla classe REALTIME_PRIORITY_CLASS; gli amministratori e gli appartenenti al gruppo *Power users* godono automaticamente di questo privilegio.

Quando un utente esegue un programma interattivo il sistema deve schedulare i thread del processo per fornire buone prestazioni: è per questo motivo che Windows ha una regola di scheduling speciale per i processi di classe NORMAL_PRIORITY_CLASS. Windows distingue tra il processo associato alla finestra in primo piano e gli altri processi, in background. Quando un processo si sposta in primo piano il sistema operativo

incrementa il quanto di tempo di tutti i suoi thread di un fattore 3. Questo aumento ha l'effetto di concedere ai thread con prevalenza d'elaborazione (CPU-bound) in primo piano un tempo d'esecuzione tre volte più lungo rispetto ai thread simili dei processi in background.

19.7.3.2 Priorità dei thread

La priorità iniziale di un thread è determinata dalla sua classe, ma si può modificare tramite la funzione `SetThreadPriority()`; essa accetta un argomento che specifica una priorità relativa alla priorità di base della classe del thread:

- `THREAD_PRIORITY_LOWEST`: base – 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base – 1
- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

Per regolare la priorità si usano altri due valori convenzionali. Il kernel ha due classi di priorità (Paragrafo 19.3.2.2): la classe per le elaborazioni real time (livelli 16-31) e la classe a priorità variabile (livelli 1-15); `THREAD_PRIORITY_IDLE` rappresenta il livello di priorità 16 per i thread d'elaborazione real time, e il livello di priorità 1 per i thread a priorità variabile. `THREAD_PRIORITY_TIME_CRITICAL` rappresenta il livello di priorità 31 per i thread d'elaborazione real time, e il livello 15 per i thread a priorità variabile.

Come si descrive nel Paragrafo 19.3.2.2, il kernel regola dinamicamente la priorità di un thread in funzione del fatto che si tratti di un thread con prevalenza di I/O o con prevalenza d'elaborazione; l'API Win32 fornisce un modo di disattivare questo processo di regolazione tramite le funzioni `SetProcessPriorityBoost()` e `SetThreadPriorityBoost()`.

19.7.3.3 Sospensione e ripristino dei thread

Un thread può essere creato in uno stato sospeso o può essere collocato in uno stato sospeso in seguito utilizzando la funzione `SuspendThread()`. Prima che un thread sospeso possa essere schedulato dal dispatcher del kernel deve essere ripristinato dallo stato sospeso con l'uso della funzione `ResumeThread()`. Entrambe le funzioni impostano un contatore in modo che se un thread viene sospeso per due volte, deve essere ripristinato per due volte prima che possa essere eseguito.

19.7.3.4 Sincronizzazione dei thread

Per sincronizzare l'accesso concorrente agli oggetti condivisi dai thread il kernel fornisce oggetti di sincronizzazione come semafori e mutex, che sono oggetti dispatcher, come descritto nel Paragrafo 19.3.2.2. I thread possono anche essere sincronizzati con servizi del kernel che operano su oggetti kernel, come thread, processi e file, perché anche questi sono oggetti dispatcher. La sincronizzazione con oggetti dispatcher del kernel può essere ottenuta mediante l'uso delle funzioni `WaitForSingleObject()`

e `WaitForMultipleObjects()`, che aspettano la segnalazione di uno o più oggetti dispatcher.

Un altro metodo di sincronizzazione è disponibile per i thread all'interno dello stesso processo che vogliono eseguire codice esclusivo. L'**oggetto sezione critica** Win32 è un oggetto mutex in modalità utente che può spesso essere acquisito e rilasciato senza entrare nel kernel. In un sistema multiprocessore, una sezione critica Win32 tenterà di ciclare continuamente in attesa del rilascio di una sezione critica detenuta da un altro thread. Se l'attesa dura troppo, il thread che vuole effettuare l'acquisizione allocherà un mutex kernel e cederà la sua CPU. Le sezioni critiche sono particolarmente efficienti perché il mutex kernel viene allocato solo quando c'è contesa e quindi utilizzato solo dopo aver tentato il continuo ciclare. La maggior parte dei mutex nei programmi non è mai effettivamente contesa: il risparmio è quindi notevole.

Prima di utilizzare una sezione critica, alcuni thread nel processo devono chiamare `InitializeCriticalSection()`. Ogni thread che voglia acquisire il mutex chiama `EnterCriticalSection()` e poi chiama `LeaveCriticalSection()` per rilasciare il mutex. Esiste anche una funzione `TryEnterCriticalSection()` che tenta di acquisire il mutex senza che il thread si blocchi.

Per i programmi che utilizzano lock di lettura-scrittura in modalità utente piuttosto che i mutex, Win32 supporta i **lock SWR** (*slim reader-writer*). I lock SWR hanno API simili a quelle per le sezioni critiche, come `InitializeSRWLock`, `AcquireSRWLockXXX` e `ReleaseSRWLockXXX`, dove XXX ha valore `Exclusive` o `Shared`, a seconda se il thread desidera l'accesso in scrittura o se è sufficiente l'accesso in lettura all'oggetto protetto da lock. L'API Win32 supporta anche le variabili condizionali, che possono essere utilizzate sia con le sezioni critiche sia con i lock SRW.

19.7.3.5 Pool di thread

La frequente creazione e cancellazione di thread può imporre un alto costo alle applicazioni e ai servizi che usano ciascuno di loro per svolgere piccole quantità di lavoro. I pool di thread di Win32 offrono tre servizi ai programmi in modalità utente: una coda a cui si possono delegare le richieste di lavoro (tramite la funzione `SubmitThreadpoolWork()`), una API (`RegisterWaitForSingleObject()`) utilizzabile per legare le chiamate di ritorno agli handle in attesa e alcune API per lavorare con i timer (`CreateThreadpoolTimer()` e `WaitForThreadpoolTimerCallbacks()`) e per creare un legame con le chiamate di ritorno alle code di completamento di I/O (`BindIoCompletionCallback()`).

L'obiettivo dei pool di thread è migliorare le prestazioni e ridurre l'impatto della memoria: i thread sono relativamente costosi e un processore non può che eseguire un solo thread per volta, indipendentemente dal numero di thread disponibili. I pool di thread tentano di ridurre il numero di thread eseguibili con un lieve differimento delle richieste di lavoro (riutilizzando ciascun thread per più richieste), fornendo al contempo thread a sufficienza per sfruttare efficacemente le CPU della macchina. Le API per la chiamata di ritorno associata ad attese o timer permettono al pool di thread

di ridurre il numero di thread in un processo, usandone molto meno di quanti ne sarebbero necessari se un processo dovesse dedicare thread separati per servire l'attesa di un handle, di un timer o di una porta di completamento.

19.7.3.6 Fibre

Una **fibra** (*fiber*) è codice utente sottoposto a un regime di scheduling definito dall'utente. Le fibre lavorano interamente in modalità utente e il kernel non è a conoscenza della loro esistenza. Il meccanismo della fibra utilizza i thread di Windows come se fossero delle CPU per l'esecuzione delle fibre. Le fibre sono pianificate in maniera cooperativa, nel senso che non subiscono mai prelazione, ma devono cedere esplicitamente il thread su cui sono in esecuzione. Quando una fibra cede un thread, un'altra fibra può essere pianificata per l'esecuzione su quel thread dal sistema runtime (il codice run-time del linguaggio di programmazione).

Il sistema crea una fibra richiamando una tra le due funzioni `ConvertThreadToFiber()` o `CreateFiber()`; la differenza principale fra queste due funzioni è che la seconda non avvia l'esecuzione della fibra appena creata: per cominciare l'esecuzione l'applicazione deve richiamare la funzione `SwitchToFiber()`; per terminare l'esecuzione di una fibra l'applicazione deve richiamare la funzione `DeleteFiber()`.

Le fibre non sono raccomandate per i thread che utilizzano le API Win32 invece delle funzioni della libreria C standard a causa di potenziali incompatibilità. I thread Win32 in modalità utente possiedono un **TEB** (*thread-environment block*) che contiene numerosi campi specifici del thread utilizzati dalle API Win32. Le fibre devono condividere il TEB del thread su cui sono in esecuzione. Ciò può portare alcuni problemi quando un'interfaccia Win32 mette le informazioni di stato nel TEB per una data fibra e in seguito le informazioni vengono sovrascritte da una fibra differente. Le fibre sono incluse nell'API Win32 per facilitare il porting di applicazioni legacy UNIX che sono state scritte per un modello di thread in modalità utente come Pthreads.

19.7.3.7 UMS e ConcRT

Un nuovo meccanismo di Windows 7, lo scheduling in modalità utente (*User-Mode Scheduling*, UMS), risolve diversi limiti delle fibre. Si ricordi, prima di tutto, che le fibre sono inaffidabili per l'esecuzione di API Win32 perché non hanno propri TEB. Quando un thread che esegue una fibra si blocca nel kernel, lo scheduler utente perde per un certo tempo il controllo della CPU, mentre il dispatcher del kernel si fa carico dello scheduling. I problemi possono verificarsi quando le fibre cambiano lo stato nel kernel di un thread, come i token di priorità o di personalizzazione, o quando iniziano un I/O asincrono.

UMS fornisce un modello alternativo riconoscendo che ogni thread di Windows è in realtà formato da due thread: un thread del kernel (KT) e un thread utente (UT). Ogni tipo di thread ha un proprio stack e un proprio insieme di registri salvati. KT e UT appaiono come un singolo thread al programmatore perché i thread UT non si possono mai bloccare, ma devono sempre entrare nel kernel dove avviene un passaggio隐式 al corrispondente KT. UMS utilizza i TEB di ogni UT per identificarli univo-

camente. Quando un UT entra nel kernel viene effettuato un passaggio esplicito al KT corrispondente all'UT identificato dal TEB corrente. La ragione per cui il kernel non sa quale UT è in esecuzione è che i thread UT possono invocare uno scheduler in modalità utente, come fanno le fibre. In UMS, però, lo scheduler scambia i thread UT, oltre a scambiare i TEB.

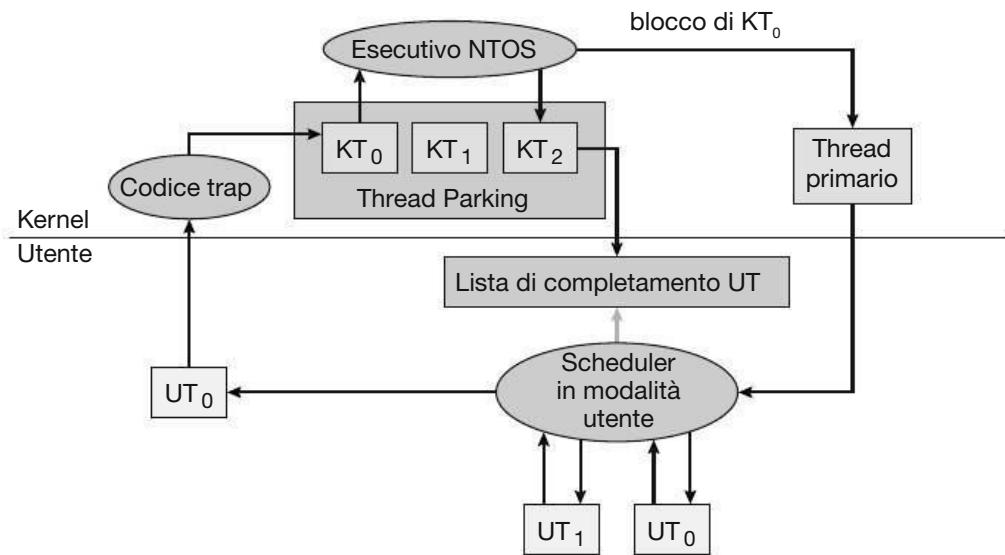
Quando un UT entra nel kernel, il suo KT può essere bloccato. Quando ciò accade, il kernel passa a un thread di scheduling, che UMS chiama thread primario, e usa questo thread per rientrare nello scheduler in modalità utente in modo da poter scegliere un altro UT da eseguire. Alla fine, un KT bloccato completerà le sue operazioni e sarà pronto per tornare alla modalità utente. Poiché UMS ha già riattivato lo scheduler in modalità utente per eseguire un UT diverso, accoda l'UT corrispondente al KT completato a una lista di completamento in modalità utente. Quando lo scheduler in modalità utente deve scegliere un nuovo UT a cui passare, può esaminare la lista di completamento e trattare qualsiasi UT sulla lista come candidato per lo scheduling.

A differenza delle fibre, UMS non è destinato all'utilizzo diretto da parte del programmatore. I dettagli della scrittura di uno scheduler in modalità utente possono essere molto impegnativi e UMS non include un tale scheduler. Gli scheduler provengono invece dalle librerie di un linguaggio di programmazione costruite su UMS. Microsoft Visual Studio 2010 dispone di Concurrency Runtime (ConcRT), un ambiente di programmazione concorrente per C++. ConcRT fornisce uno scheduler in modalità utente insieme a funzionalità per la scomposizione dei programmi in task che possono essere poi pianificati sulle CPU disponibili. ConcRT fornisce il supporto per gli stili `par_for` dei costrutti, oltre a una gestione rudimentale delle risorse e a primitive di sincronizzazione dei task. Le caratteristiche principali di UMS sono descritte nella Figura 19.11.

19.7.3.8 Winsock

Winsock è l'API di Windows per i socket. Winsock è un'interfaccia a livello di sessione in gran parte compatibile con i socket UNIX, ma dotata di alcune estensioni aggiuntive di Windows. Winsock fornisce un'interfaccia standardizzata a molti protocolli di trasporto che possono avere diversi schemi di indirizzamento, in modo che qualsiasi applicazione Winsock possa funzionare su qualsiasi pila di protocolli compatibile con Winsock. Winsock ha subito un importante aggiornamento in Windows Vista con l'aggiunta del tracing, del supporto IPv6, della impersonazione, di nuove API di sicurezza e di molte altre caratteristiche.

Winsock segue il modello Windows Open System Architecture (WOSA), che fornisce un'interfaccia del provider di servizi (SPI) standard tra le applicazioni e i protocolli di rete. Le applicazioni possono caricare e scaricare i protocolli stratificati che costruiscono funzionalità addizionali, come una sicurezza aggiuntiva, sopra agli strati dei protocolli di trasporto. Winsock supporta operazioni asincrone e notifiche, multicasting affidabile, socket sicuri e socket in modalità kernel. C'è anche il supporto per modelli di utilizzo semplificati, come la funzione `WSAConnectByName()` che accetta la destinazione come una stringa che specifica il nome o l'indirizzo IP del server e il servizio o il numero di porta di destinazione.



Solo il thread primario è eseguito in modalità utente
 Il codice della trap passa ai KT nel Thread Parking
 KT si blocca => il thread primario ritorna alla modalità utente
 KT si sblocca e entra nel Thread Parking => il completamento di UT viene accodato

Figura 19.11 Scheduling in modalità utente.

19.7.4 Comunicazione fra processi mediante messaggi di Windows

Le applicazioni Win32 gestiscono la comunicazione tra processi in molti modi. Uno di essi è basato sulla condivisione degli oggetti del kernel; un altro metodo, particolarmente diffuso per le applicazioni con interfaccia utente grafica Win32, è basato sullo scambio di messaggi di Windows. Un thread può spedire un messaggio a un altro thread o a una finestra richiamando una fra le funzioni `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()` e `SendMessageCallback()`. La differenza fra le funzioni *Post* e quelle *Send* è che le prime sono asincrone: restituiscono immediatamente il controllo, quindi il thread chiamante non sa esattamente quando il messaggio giungerà a destinazione; le funzioni *Send*, invece, sono sincrone e bloccano quindi il chiamante finché il messaggio non sia stato recapitato.

Un thread può trasmettere dati insieme con un messaggio; in questo caso i dati devono essere copiati, perché processi diversi hanno spazi d'indirizzi distinti. Il sistema copia i dati richiamando `SendMessage()` per trasmettere un messaggio di tipo `WM_COPYDATA` con una struttura dati `COPYDATASTRUCT` contenente la lunghezza e l'indirizzo dei dati da trasferire; quando si trasmette il messaggio, il sistema operativo copia i dati in una nuova regione della memoria, e ne fornisce l'indirizzo virtuale al processo ricevente.

Ogni thread Win32 ha la propria coda d'ingresso dalla quale riceve messaggi. Si tratta di un'architettura più affidabile rispetto alla condivisione della coda d'ingresso dell'ambiente Windows a 16 bit, perché con le code distinte un'applicazione bloccata non impedisce la ricezione dei dati per le altre applicazioni. Se un'applicazione Win32 non impiega la funzione `GetMessage()` per trattare gli eventi nella sua coda d'ingresso, la coda si riempirà, e dopo circa 5 secondi il sistema contrasseggerà l'applicazione come "Non rispondente".

19.7.5 Gestione della memoria

L'API Win32 mette a disposizione delle applicazioni molti modi per utilizzare la memoria: la memoria virtuale, i file associati allo spazio d'indirizzi di memoria, gli *heap* e la memoria locale dei thread.

19.7.5.1 Memoria virtuale

Per prenotare o eseguire l'assegnazione di una regione della memoria virtuale, un'applicazione invoca la funzione `VirtualAlloc()`, e per eseguire le operazioni inverse invoca `VirtualFree()`. Queste funzioni permettono all'applicazione di specificare l'indirizzo virtuale a partire dal quale si deve assegnare la memoria; esse operano con multipli della dimensione della pagina di memoria. Alcuni esempi di queste funzioni sono riportati dalla Figura 19.12.

Un processo può fissare nella memoria fisica alcune sue pagine richiamando la funzione `VirtualLock()`; il massimo numero fissabile di pagine per processo è 30, sempre che il processo non richiami prima la funzione `SetProcessWorkingSetSize()` per aumentare la dimensione minima del working set.

19.7.5.2 Mappatura di file in memoria

Un'applicazione può anche far uso della memoria associando un file al proprio spazio d'indirizzi; si tratta anche di un metodo conveniente per la condivisione di memoria fra due processi: entrambi i processi associano lo stesso file alla loro memoria vir-

```
// riserva 16 MB all'estremo superiore
// del nostro spazio di indirizzi
void *buf = VirtualAlloc(0,0x1000000,MEM_RESERVE |
    PAGE_READWRITE);
// effettua l'assegnazione degli 8 MB superiori
// dello spazio riservato
VirtualAlloc(buf + 0x800000,0x800000 MEM_COMMIT,PAGE_READWRITE);
// fa qualcosa
// annulla l'assegnazione
VirtualFree(buf + 0x800000,0x800000,Mem_DECOMMIT);
// rilascia tutto lo spazio assegnato
VirtualFree(buf,0,Mem_RELEASE);
```

Figura 19.12 Frammenti di codice per l'assegnazione della memoria virtuale.

```

// apre il file o lo crea nel caso in cui non esista
HANDLE hfile = CreateFile("somefile", GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);
// crea uno spazio di 8 MB per la mappatura del file
HANDLE hmap = CreateFileMapping(hfile, PAGE_READWRITE, SEC_COMMIT,
    0, 0x800000, "SHM_1");
// ottiene una vista dello spazio mappato
void *buf = MapViewOfFile(hmap, FILE_MAP_ALL_ACCESS, 0, 0, 0x800000);
// fa qualcosa
// annulla la mappatura del file
UnMapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);

```

Figura 19.13 Frammenti di codice per la mappatura in memoria di un file.

tuale. La mappatura di un file è un procedimento a più fasi, come dimostra l'esempio della Figura 19.13.

Se un processo desidera mappare uno spazio d'indirizzi al solo scopo di condividere memoria con un altro processo, la presenza di un file è superflua: il processo può invocare `CreateFileMapping()` specificando come handle del file il valore `0xffffffff`, oltre a una certa dimensione; l'oggetto risultante si può condividere per ereditarietà, per nome o per duplicazione dell'handle.

19.7.5.3 Heap

Il terzo modo in cui un'applicazione può usare la memoria, analogo all'uso di `malloc()` e `free()` nel C standard, è rappresentato dallo heap. Uno *heap* nell'ambiente Win32 è semplicemente una regione riservata di uno spazio d'indirizzi. Un processo Win32 è dotato al momento della sua creazione di uno **heap predefinito** (*default heap*); poiché molte applicazioni Win32 sono multithread è necessario sincronizzare gli accessi allo heap per evitare che le sue strutture dati siano danneggiate da accessi concorrenti di thread diversi.

Win32 mette a disposizione le seguenti funzioni per la gestione e l'assegnazione di uno heap: `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()` e `HeapDestroy()`; l'API Win32 fornisce anche le funzioni `HeapLock()` e `HeapUnlock()` che garantiscono a un thread l'accesso esclusivo a uno heap; a differenza di `VirtualLock()`, queste funzioni eseguono solo la sincronizzazione, e non fissano pagine nella memoria fisica.

Lo heap originale Win32 era ottimizzato per un uso efficiente dello spazio. Ciò ha portato a notevoli problemi con la frammentazione dello spazio degli indirizzi nei programmi dei server di grandi dimensioni in esecuzione per lunghi periodi di tempo. Un nuovo progetto di **heap a bassa frammentazione** (LFH), introdotto in Windows XP,

```
// riserva spazio per una variabile
DWORD var_index = TlsAlloc();
// gli assegna il valore 10
TlsSetValue(var_index,10);
// lo rilegge
int var = TlsGetValue(var_index);
// rilascia l'indice
TlsFree(var_index);
```

Figura 19.14 Codice per l'assegnazione dinamica di memoria locale a un thread.

ha notevolmente ridotto il problema della frammentazione. Il gestore di heap di Windows 7 è in grado di attivare automaticamente LFH quando conviene.

19.7.5.4 Memoria locale dei thread

Un quarto modo in cui un'applicazione può usare la memoria è rappresentato dal **meccanismo di memorizzazione locale dei thread** (TLS). Di solito le funzioni che fanno uso di dati statici o globali non operano correttamente in un ambiente multithread: la funzione run-time `strtok()` del linguaggio C, per esempio, impiega una variabile statica per tener traccia della posizione corrente durante la scansione di una sequenza di caratteri; affinché due thread concorrenti possano eseguire correttamente `strtok()` devono tenere distinte le variabili che mantengono la posizione corrente. TLS fornisce un meccanismo per mantenere le istanze delle variabili globali per la funzione che viene eseguita, ma non condivise con alcun altro thread.

Il meccanismo TLS può essere messo in atto sia tramite metodi dinamici sia tramite metodi statici. Il metodo dinamico è illustrato nella Figura 19.14. Esso alloca spazio dello heap globale e lo collega al blocco di ambiente del thread che Windows assegna a ogni thread in modalità utente. Il TEB è facilmente accessibile da ogni thread e viene utilizzato non solo per TLS, ma per tutte le informazioni di stato in modalità utente specifiche dei thread.

Per usare in un thread una variabile statica locale, in modo da assicurare che ogni thread ne possieda una copia privata, un'applicazione dovrebbe dichiarare la variabile come segue:

```
--declspec(thread) DWORD cur_pos = 0;
```

19.8 Sommario

Il sistema operativo Windows è stato concepito da Microsoft come sistema operativo portatile ed estendibile, capace di trarre vantaggio dalle innovazioni nelle tecniche e nell'hardware; Windows gestisce l'elaborazione parallela simmetrica e supporta diversi ambienti operativi, tra cui processori sia a 32 sia a 64 bit e i calcolatori NUMA. L'uso degli oggetti del kernel per fornire i servizi fondamentali e la gestione del mo-

dello di elaborazione client-server permette a Windows di offrire un'ampia gamma di ambienti d'applicazione. Windows fornisce la gestione della memoria virtuale, servizi integrati di caching e scheduling con prelazione. Adotta un modello di sicurezza elaborato e comprende caratteristiche di internazionalizzazione. Windows è eseguibile su un'ampia varietà di calcolatori; in questo modo gli utenti possono scegliere e aggiornare i propri calcolatori, e i dispositivi che li compongono, in funzione delle loro disponibilità finanziarie e delle prestazioni richieste senza dover modificare le applicazioni eseguite.

Esercizi di ripasso

- 19.1** Che tipo di sistema operativo è Windows? Descrivete due delle sue caratteristiche principali.
- 19.2** Elencate gli obiettivi di progetto di Windows. Descrivetene due in dettaglio.
- 19.3** Descrivete il processo di avvio di un sistema Windows.
- 19.4** Descrivete i tre livelli principali dell'architettura del kernel di Windows.
- 19.5** Che lavoro svolge il gestore degli oggetti?
- 19.6** Che tipi di servizio offre il gestore dei processi?
- 19.7** Che cos'è una chiamata di procedura locale?
- 19.8** Quali sono le responsabilità del gestore dell'I/O?
- 19.9** Quali tipologie di networking sono supportate da Windows? Come sono implementati i protocolli di trasporto da Windows? Descrivete due protocolli di rete.
- 19.10** Descrivete l'organizzazione dello spazio dei nomi di NTFS.
- 19.11** Come tratta le strutture dati NTFS? Come viene ripristinato NTFS dopo un crash di sistema? Che cosa viene garantito dopo che è avvenuto un ripristino?
- 19.12** Come alloca la memoria utente Windows?
- 19.13** Descrivete alcuni dei modi in cui un'applicazione utilizza la memoria per mezzo della API Win32.

Esercizi

- 19.14** Quando e perché si dovrebbe usare la funzionalità della procedura di chiamata differita in Windows?
- 19.15** Che cos'è un handle, e come fa un processo a ottenerne uno?

- 19.16** Descrivete il funzionamento del gestore della memoria virtuale. In che modo il gestore della VM può migliorare le prestazioni?
- 19.17** Descrivete un'applicazione utile della funzionalità che vieta l'accesso alle pagine fornita da Windows.
- 19.18** Descrivete le tre tecniche previste per trasmettere i dati in una procedura di chiamata locale. Quali differenti contesti portano all'applicazione dell'una o dell'altra tecnica di scambio dei messaggi?
- 19.19** Da quale entità è gestita la cache di Windows? E con quali modalità?
- 19.20** Per quali aspetti la struttura della directory NTFS differisce da quella adottata dai sistemi operativi UNIX?
- 19.21** Che cos'è un processo, e in che modo il sistema Windows lo gestisce?
- 19.22** Che cosa sono le “fibre” disponibili in Windows? In che cosa differiscono dai thread?
- 19.23** In che cosa lo scheduling in modalità utente (UMS) di Windows 7 differisce dalle fibre? Individuate alcuni compromessi tra fibre e UMS.
- 19.24** UMS ritiene che il thread sia composto da due parti, un UT e un KT. In che cosa potrebbe essere utile consentire agli UT di continuare l'esecuzione in parallelo con i loro KT?
- 19.25** Quali sono i vantaggi e gli svantaggi in termini di prestazioni nel permettere ai KT e agli UT di essere eseguiti su diversi processori?
- 19.26** Perché l'auto-mappa occupa grandi quantità di spazio degli indirizzi virtuali, ma non occupa memoria virtuale aggiuntiva?
- 19.27** In che modo l'auto-mappa facilita al gestore della VM lo spostamento delle pagine della tabella delle pagine da e verso il disco? Dove sono conservate le pagine della tabella delle pagine sul disco?
- 19.28** Quando un sistema Windows si sospende, il sistema viene spento. Supponiamo di sostituire la CPU o aumentare la quantità di RAM su un sistema che è stato sospeso. Il sistema potrebbe funzionare? Motivate la risposta.
- 19.29** Fornite un esempio che mostra come l'uso di un conteggio di sospensione è utile nella sospensione e nella ripresa dei thread in Windows.

Note bibliografiche

[Russinovich e Solomon 2009] forniscono una panoramica di Windows 7 e una considerevole serie di dettagli tecnici su caratteristiche e componenti del sistema.

[Brown 2000] presenta i dettagli dell'architettura di sicurezza di Windows.

La Microsoft Developer Network Library (<http://msdn.microsoft.com>) costituisce una ricca fonte di informazioni su Windows e altri prodotti Microsoft e comprende la documentazione di tutte le API pubblicate.

[Iseminger 2000] è un buon riferimento su Windows Active Directory. Una trattazione dettagliata della stesura di programmi che utilizzano l'API Win32 appare in [Richter 1997]. [Silberschatz et al. 2010] fornisce una buona descrizione degli alberi B+.

Il codice sorgente per una versione WRK del 2005 del kernel di Windows, insieme a una collezione di slide e ad altro materiale, è disponibile per l'utilizzo da parte delle università all'indirizzo www.microsoft.com/WindowsAcademic.

Bibliografia

- [Brown 2000] K. Brown, *Programming Windows Security*, Addison-Wesley, 2000.
- [Iseminger 2000] D. Iseminger, *Active Directory Services for Microsoft Windows 2000*. Technical Reference, Microsoft Press, 2000.
- [Richter 1997] J. Richter, *Advanced Windows*, Microsoft Press, 1997.
- [Russinovich and Solomon 2009] M. E. Russinovich e D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*, 5° Ed., Microsoft Press, 2009.
- [Silberschatz et al. 2010] A. Silberschatz, H. F. Korth e S. Sudarshan, *Database System Concepts*, 6° Ed., McGraw-Hill, 2010.

CAPITOLO

20

OBIETTIVI DEL CAPITOLO

- Illustrare come le caratteristiche dei sistemi operativi migrino nel tempo da grandi sistemi elaborativi a piccoli calcolatori.
- Discutere le caratteristiche di diversi sistemi operativi di importanza storica.

Prospettiva storica

Ora che siamo in grado di comprendere i concetti alla base dei sistemi operativi (scheduling della CPU, gestione della memoria, processi e così via) è possibile esaminare come questi stessi concetti siano stati applicati in più sistemi operativi molto influenti del passato. Alcuni di essi, come il sistema XDS-940 o il sistema THE, sono stati pezzi unici, altri, come l'OS/360, hanno avuto una grande diffusione. L'ordine di presentazione è stato scelto in modo da evidenziare analogie e differenze tra i sistemi, quindi non è strettamente cronologico o ordinato per importanza. Chiunque si occupi di sistemi operativi dovrebbe avere confidenza con tutti questi sistemi.

Nelle note bibliografiche alla fine del capitolo sono inclusi riferimenti a ulteriori letture. I documenti scritti dai progettisti dei sistemi sono importanti sia per il contenuto tecnico sia per lo stile e il gusto.

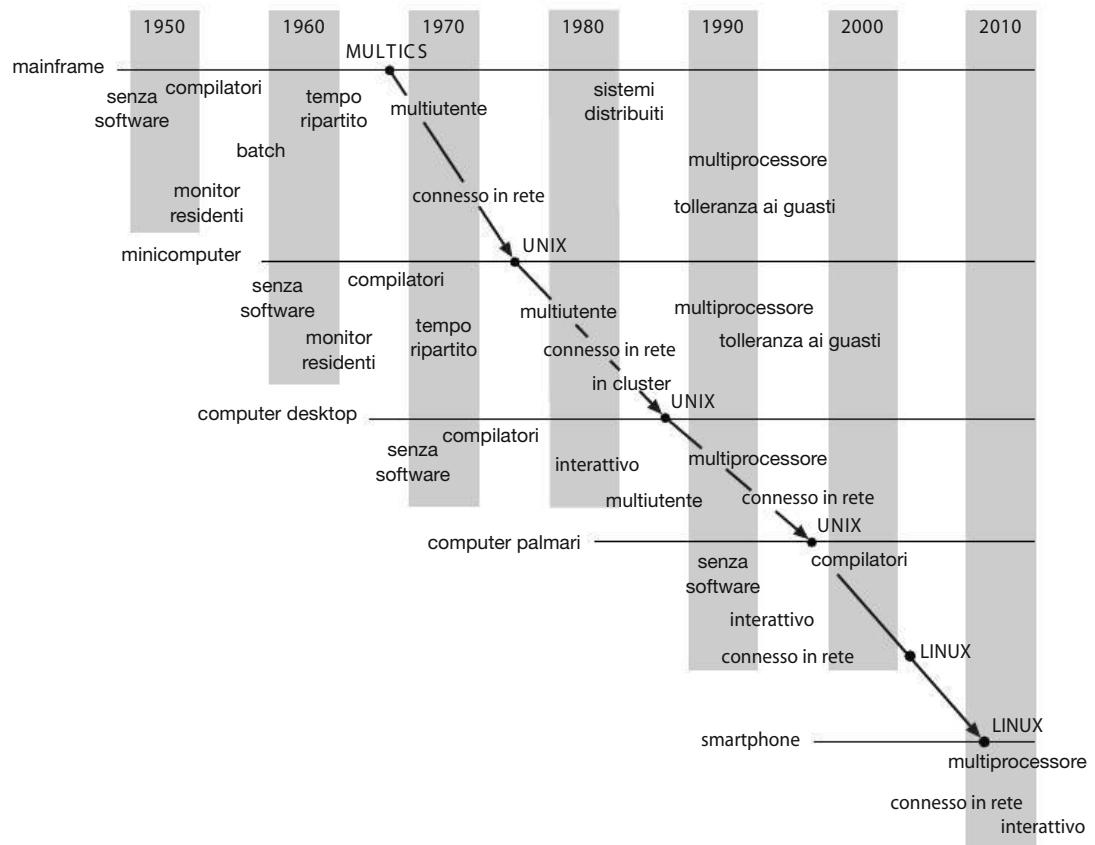


Figura 20.1 Migrazione di concetti e caratteristiche dei sistemi operativi.

20.1 Migrazione delle caratteristiche

Una ragione per studiare le prime architetture e i primi sistemi operativi risiede nel fatto che le funzionalità che una volta erano disponibili solamente su grandi sistemi sono ora riscontrabili su sistemi molto piccoli. In effetti, uno studio dei sistemi operativi per mainframe e per microcomputer mostra che molte funzionalità una volta disponibili soltanto su mainframe sono state adottate dai microcomputer. Gli stessi concetti di un sistema operativo sono quindi adatti per diverse classi di calcolatori: mainframe, minicomputer, microcomputer e dispositivi palmari. Per capire i moderni sistemi operativi è allora necessario conoscere il tema della migrazione delle caratteristiche e la lunga storia di molte funzionalità di un sistema operativo, come illustrato dalla Figura 20.1.

Un buon esempio di migrazione delle caratteristiche inizia dal sistema operativo **MULTICS** (*multiplexed information and computing services*). MULTICS fu sviluppato tra il 1965 e il 1970 dal MIT come una **utilità** per il calcolo e veniva eseguito su un mainframe (il GE 645) grande e complesso. Molte delle idee sviluppate per MULTICS vennero in seguito utilizzate dai Bell Laboratories (uno dei partner iniziali nello sviluppo di MULTICS) per il progetto di UNIX. Il sistema operativo UNIX fu progettato in-

torno al 1970 per un minicomputer PDP-11. Intorno al 1980 le caratteristiche di UNIX diventarono le basi per i sistemi operativi UNIX-like utilizzati sui microcomputer. Le stesse funzionalità sono presenti ora in diversi sistemi operativi recenti per microcomputer, come Microsoft Windows, Windows XP, e Mac OS X. Linux include alcune di queste stesse funzionalità, che possono oggi essere impiegati da dispositivi mobili.

20.2 Primi sistemi

Rivolgiamo ora la nostra attenzione a una rassegna storica dei primi sistemi elaborativi. Va notato che la storia della computazione inizia molto prima dei “computer”, con i telai e le macchine calcolatrici. Ciononostante, inizieremo la nostra analisi dai computer del ventesimo secolo.

Prima degli anni '40, i sistemi elaborativi erano progettati e implementati per eseguire compiti specifici e prefissati. Modificare uno di questi compiti richiedeva un grande sforzo e del lavoro manuale. Tutto questo cambiò negli anni '40, quando Alan Turing, John von Neumann e altri colleghi lavorarono, sia insieme che separatamente, all'idea di un **computer a programma memorizzato** (*stored program computer*) di uso più generale. Una tale macchina doveva memorizzare sia un programma sia i dati da elaborare e il programma memorizzato doveva fornire istruzioni per il trattamento dei dati.

Questa fondamentale idea permise la rapida nascita di diversi computer general purpose, ma molta della storia di queste macchine è circondata dal mistero e dai segreti del loro sviluppo durante la seconda guerra mondiale. È probabile che il primo computer general purpose a programma memorizzato funzionante fu il Manchester Mark 1, sperimentato con successo nel 1949. Il primo computer commerciale fu il suo discendente Ferranti Mark 1, messo in vendita nel 1951.

I primi calcolatori erano macchine molto grandi che si gestivano da una console dalla quale il programmatore, che era anche l'operatore del sistema elaborativo, poteva scrivere e gestire un programma. Il programma veniva dapprima caricato manualmente nella memoria impiegando gli interruttori del pannello frontale, un'istruzione alla volta, oppure inserendo un nastro perforato o un pacco di schede perforate; quindi si premevano i pulsanti appropriati per impostare l'indirizzo iniziale e per avviare l'esecuzione. Durante l'esecuzione del programma, il programmatore/operatore poteva controllarne l'esecuzione per mezzo di spie situate sulla console. Se si riscontravano errori, il programmatore poteva fermare il programma, esaminare il contenuto della memoria e dei registri e mettere a punto il programma direttamente dalla console. L'esito veniva stampato, oppure trascritto perforando appositi nastri o schede di carta per essere stampato successivamente.

20.2.1 Sistemi di elaborazione dedicati

Con il passare del tempo furono sviluppati altri programmi, e altro hardware. I lettori di schede, le stampanti e i nastri magnetici divennero d'uso comune. Per facilitare la programmazione furono progettati assemblatori, caricatori e linker, e create librerie

di funzioni comuni che si potevano copiare in un programma senza dover essere riscritte, consentendo il riutilizzo di codice già scritto.

Le procedure che eseguivano I/O assunsero un'importanza particolare. Ogni nuovo dispositivo di I/O aveva le sue caratteristiche che richiedevano un'attenta programmazione. Per ogni dispositivo di I/O si scriveva una procedura speciale, chiamata driver di dispositivo, capace di interagire con buffer, flag, registri, bit di controllo e bit di stato di ogni specifico dispositivo. Un compito semplice, come la lettura di un carattere da un lettore di nastri, poteva implicare complesse sequenze di operazioni specifiche del dispositivo. Invece di dover scrivere ogni volta il codice necessario, s'impiegava semplicemente il driver di dispositivo della libreria.

In seguito apparvero i compilatori per i linguaggi FORTRAN, COBOL e altri linguaggi, che resero molto più semplice la programmazione, ma più complesso il funzionamento del calcolatore. Per preparare l'esecuzione di un programma scritto in FORTRAN, il programmatore doveva prima caricare nel calcolatore il compilatore del FORTRAN. Il compilatore normalmente si teneva su un nastro magnetico, che quindi doveva essere montato nell'unità a nastri. Il programma veniva letto dal lettore di schede e scritto in un altro nastro. Il compilatore del FORTRAN produceva in uscita un programma in linguaggio assembler che, a sua volta, doveva essere assemblato; quest'operazione richiedeva il montaggio di un altro nastro in cui si trovava l'assemblatore. Il risultato prodotto dall'assemblatore si doveva linkare alle procedure della libreria. Infine, si poteva leggere ed eseguire la forma binaria del programma. Si effettuava il caricamento nella memoria, la correzione e la messa a punto dalla console e infine l'esecuzione.

Il tempo di preparazione necessario per eseguire un job poteva essere molto elevato; ciascun job consisteva di molti passi distinti:

1. il caricamento del nastro con il compilatore del linguaggio FORTRAN;
2. l'esecuzione del compilatore;
3. la rimozione del nastro con il compilatore;
4. il caricamento del nastro dell'assemblatore;
5. l'esecuzione dell'assemblatore;
6. la rimozione del nastro dell'assemblatore;
7. il caricamento del programma oggetto;
8. l'esecuzione del programma oggetto.

Se durante una di queste fasi si riscontrava un errore, il programmatore/operatorere doveva ricominciare tutto da capo. Ogni fase poteva implicare il caricamento e la rimozione di nastri magnetici, nastri di carta e schede perforate.

Il tempo necessario per la preparazione di un job era un vero problema: durante il montaggio dei nastri e l'attività del programmatore alla console la CPU restava inattiva. Si ricordi che i primi calcolatori disponibili erano molto pochi e quei pochi erano molto costosi; un calcolatore poteva costare milioni di dollari, anche senza conside-

rare i costi del lavoro dei programmatore, della corrente elettrica, del raffreddamento e così via; il tempo d'uso di un calcolatore assumeva quindi un valore molto elevato e per ammortizzare i costi d'investimento era necessaria una **utilizzazione** elevata.

20.2.2 Sistemi di elaborazione condivisi

Per ottimizzare la produttività di un calcolatore si trovò una soluzione su due fronti. Innanzitutto si ricorreva a un operatore professionista, e il programmatore quindi non doveva più lavorare alla macchina. Di conseguenza, non appena si terminava un job, l'operatore poteva iniziare quello successivo; poiché l'operatore aveva più esperienza nel montaggio dei nastri di quanta ne avesse un programmatore, anche il tempo di preparazione si ridusse. Il programmatore doveva fornire le schede o i nastri necessari insieme con una breve descrizione del modo d'esecuzione del programma. Naturalmente, l'operatore non poteva effettuare il debug di un programma non corretto dalla console, giacché che non era in grado di capire il programma stesso. Perciò, quando si presentavano errori nel programma, si stampava un'immagine dell'intero contenuto della memoria e dei registri e il programmatore doveva fare le correzioni basandosi sulle informazioni così ottenute. In questo modo l'operatore poteva continuare immediatamente con il lavoro successivo, ma al programmatore restava un compito di correzione più difficile.

Un secondo aspetto era che i job che avevano requisiti simili venivano raggruppati in lotti (*batch*) ed eseguiti nel calcolatore come un unico gruppo per ridurre il tempo di preparazione. Si supponga per esempio che l'operatore ricevesse un job scritto in FORTRAN, uno scritto in COBOL e uno scritto ancora in FORTRAN. Per eseguirli in quell'ordine avrebbe dovuto predisporre la macchina prima per il FORTRAN, quindi per il COBOL e poi di nuovo per il FORTRAN, tutto ciò richiedeva il caricamento dei nastri compilatori, e così via. Se, invece, eseguiva i due job scritti in FORTRAN come un unico batch, poteva predisporre il calcolatore una volta sola per il linguaggio FORTRAN, risparmiando tempo.

Ma c'erano ancora alcuni problemi. Per esempio, l'operatore doveva stabilire, osservando la console, quando un programma era terminato e *perché*, ossia se era terminato in maniera normale o anomala, quindi, se era necessario, doveva stampare l'immagine del contenuto della memoria, caricare il job successivo nel lettore di schede o nel lettore del nastro e infine riavviare il calcolatore. Durante il passaggio da un job a un altro, la CPU restava inattiva. Per eliminare anche questo tempo morto, fu sviluppato il cosiddetto **sequenzializzatore automatico dei job**; questa tecnica permise la creazione dei primi rudimentali sistemi operativi. Fu creato un piccolo programma, chiamato **monitor residente** per il trasferimento automatico del controllo dell'elaborazione da un job a quello successivo (Figura 20.2). Questo monitor si trovava sempre, cioè *risiedeva*, nella memoria.

Al momento dell'accensione del calcolatore, s'invocava il monitor residente che trasferiva il controllo a un programma. Terminato il programma, il controllo tornava al monitor residente che lo passava al programma successivo; in questo modo il monitor residente sequenzializzava automaticamente i programmi e i job.

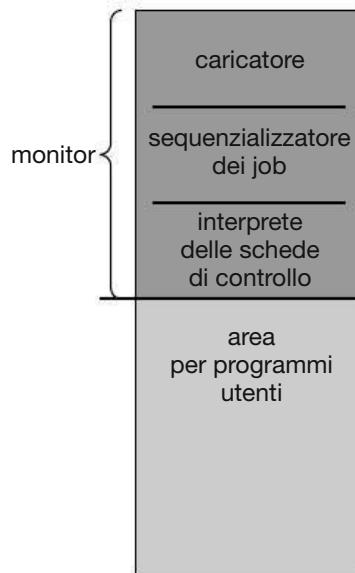


Figura 20.2 Configurazione della memoria per un monitor residente.

Ci si può domandare come facesse il monitor residente a sapere quale programma doveva eseguire. In precedenza all'operatore si consegnava una breve descrizione dei programmi da eseguire e dei relativi dati. Per passare queste informazioni direttamente al monitor furono adottate le **schede di controllo**. L'idea era semplice. Oltre al programma o ai dati, il programmatore inseriva le schede di controllo, che contenevano le direttive che indicavano al monitor residente quale programma doveva eseguire. Per esempio, un normale programma utente avrebbe potuto richiedere l'esecuzione di uno di questi tre programmi: compilatore del linguaggio FORTRAN (FTN), assemblatore (ASM), oppure programma utente (RUN). Per ciascuno di questi programmi si può impiegare una scheda di controllo diversa:

\$FTN — esegui il compilatore del linguaggio FORTRAN;

\$ASM — esegui l'assemblatore;

\$RUN — esegui il programma utente.

Queste schede indicavano al monitor residente quali erano i programmi da eseguire. Per definire i limiti di ogni job si potevano adoperare altre due schede di controllo:

\$JOB — prima scheda di un job;

\$END — ultima scheda di un job.

Queste due schede avrebbero potuto essere utili per contabilizzare le risorse di macchina impiegate dal programmatore.

Per definire il nome del job, il codice d'addebito e così via si potevano usare opportuni parametri. Per altre funzioni, come la richiesta di caricamento o rimozione di un nastro da parte dell'operatore, si potevano definire altre schede di controllo.

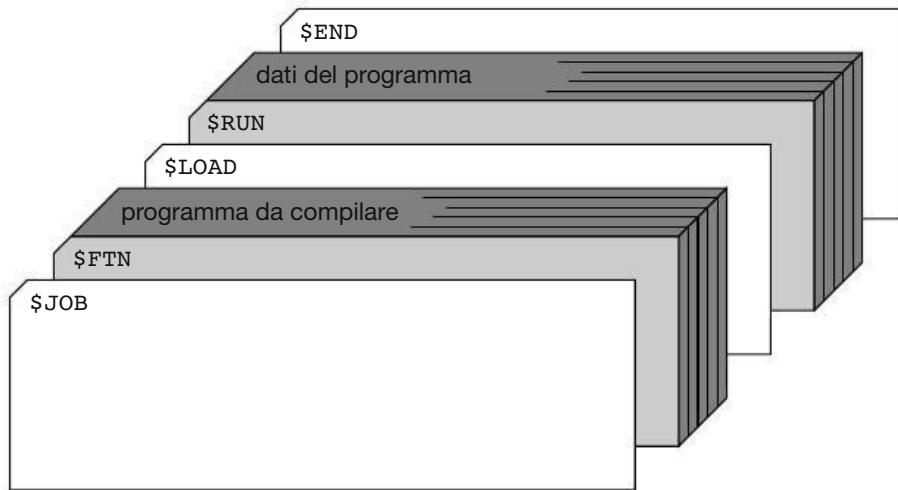


Figura 20.3 Pacco di schede per un sistema batch.

Le schede di controllo dovevano essere distinte dalle schede dei dati o del programma. Tale distinzione si otteneva per mezzo di un carattere o di un simbolo speciale riportato sulla scheda stessa. Parecchi sistemi identificavano una scheda di controllo riportando nella prima colonna il carattere \$. Altri usano un simbolo diverso: il Job Control Language (JCL) della IBM, per esempio, usava due barre (//) inserite nelle prime due colonne. Nella Figura 20.3 è riportato un esempio di preparazione del pacco di schede per un sistema batch.

Un monitor residente era quindi composto da diversi elementi chiaramente identificabili.

- **L'interprete delle schede di controllo**, responsabile della lettura e dell'esecuzione delle istruzioni contenute sulle schede stesse.
- **Il caricatore**, invocato dall'interprete delle schede di controllo per caricare nella memoria opportuni programmi di sistema e programmi applicativi.
- **I driver dei dispositivi**, usati sia dall'interprete delle schede di controllo sia dal caricatore per l'esecuzione delle operazioni di I/O. Spesso i programmi di sistema e quelli applicativi si servivano di questi stessi driver di dispositivi, che garantivano un funzionamento corretto e un risparmio di spazio nella memoria e di tempo di programmazione.

Questi sistemi batch lavoravano piuttosto bene. Il monitor residente garantiva la sequenzializzazione automatica dei lavori, così com'era indicata dalle schede di controllo. Quando una scheda di controllo indicava che si doveva eseguire un programma, il monitor caricava nella memoria il programma e gli trasferiva il controllo. Terminato il programma, il controllo tornava al monitor, il quale leggeva la scheda di controllo successiva, caricava il nuovo programma e così via. Si ripeteva il ciclo fino a che tutte le schede di controllo del job in corso erano state interpretate. Quindi, il monitor continuava automaticamente a lavorare, passando al job successivo.

Il passaggio ai sistemi batch con sequenzializzatore automatico dei lavori si è verificato per migliorare le prestazioni. Il problema, abbastanza semplice, consisteva nella lentezza dell'intervento umano, che veniva pertanto sostituito da programmi di un sistema operativo: il sequenzializzatore automatico dei lavori, infatti, eliminava la perdita di tempo derivante dalla preparazione e dalla sequenzializzazione dei lavori da parte dell'operatore.

Com'è stato evidenziato, anche con questo sistema, tuttavia, la CPU spesso rimaneva inattiva. Il problema a questo punto riguardava i dispositivi meccanici di I/O, intrinsecamente più lenti dei dispositivi elettronici. I tempi caratteristici di una CPU, anche se lenta, sono misurabili in microsecondi. Quindi, mentre questa eseguiva migliaia di istruzioni al secondo, un lettore di schede veloce, invece, poteva leggere 1200 schede al minuto, vale a dire 20 al secondo. Di conseguenza, la differenza di velocità tra la CPU e i suoi dispositivi di I/O poteva essere di tre o più ordini di grandezza. Con il tempo, i perfezionamenti tecnologici hanno portato allo sviluppo di dispositivi di I/O più veloci, ma contemporaneamente era aumentata anche la velocità delle CPU, quindi il problema non solo non si era risolto, ma si era aggravato.

20.2.3 I/O sovrapposto

Una soluzione diffusa al problema dell'I/O prevedeva la sostituzione dei lettori di schede e delle stampanti con unità a nastro magnetico. Alla fine degli anni Cinquanta e all'inizio degli anni Sessanta la maggior parte dei sistemi elaborativi era costituita da sistemi batch che utilizzavano come dispositivi di lettura i lettori di schede e come dispositivi di scrittura le stampanti o i perforatori di schede. Tuttavia, anziché far leggere le schede direttamente dalla CPU, si eseguiva un primo passaggio di copiatura delle schede in nastro magnetico. Quindi, quando un nastro era sufficientemente pieno, lo si rimuoveva e lo si portava al calcolatore. Quando una scheda veniva richiesta da un programma, si leggeva dal nastro il settore corrispondente. Analogamente, i risultati dell'elaborazione venivano inviati al nastro e il contenuto del nastro stesso veniva passato solo in una fase successiva alla stampante. I lettori di schede e le stampanti anziché dal calcolatore centrale erano gestiti fuori linea (*off-line*) (Figura 20.4).

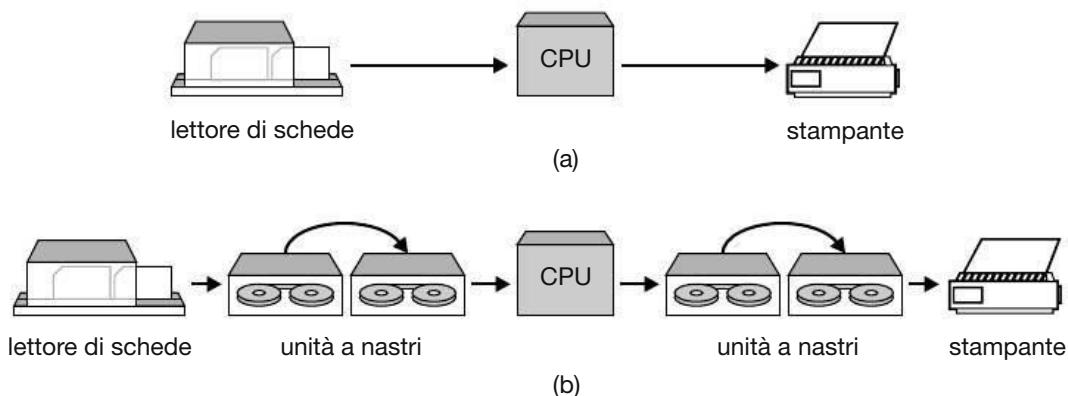


Figura 20.4 Funzionamento dei dispositivi di I/O: (a) in linea; (b) fuori linea.

Il vantaggio principale dato da questo metodo consisteva nel fatto che il calcolatore principale non dipendeva più dalla velocità dei lettori di schede e delle stampanti, ma da quella delle unità a nastro magnetico, che, comunque, erano molto più veloci dei dispositivi di I/O precedenti. L'uso del nastro magnetico come I/O poteva essere applicato con qualsiasi dispositivo, si trattasse di lettori di schede, perforatori, plotter, nastri perforati o stampanti.

Il guadagno effettivo ottenibile derivava dalla possibilità di impiegare per una stessa CPU più sistemi lettore-nastro e nastro-stampante. Se la CPU è in grado di elaborare i dati in ingresso con una velocità doppia rispetto a quella con cui il lettore legge le schede, allora due lettori contemporaneamente attivi possono produrre una quantità di nastro sufficiente per tenere costantemente occupata la CPU. D'altra parte c'è uno svantaggio: ora si riscontra un ritardo maggiore nell'esecuzione di un particolare job d'elaborazione; prima occorre eseguirne la lettura nel nastro, poi bisogna aspettare che il nastro sia riempito completamente con altri job. Quando il nastro è pieno deve essere riavvolto, rimosso, portato manualmente al calcolatore e infine montato su un'unità a nastri libera. Naturalmente, tutto questo è ragionevole per i sistemi batch. Molti job simili si possono raggruppare in batch in un nastro, prima di portare quest'ultimo al calcolatore.

Per qualche tempo i job continuarono a essere preparati con le procedure off-line, ma ben presto nella maggior parte dei sistemi queste procedure furono sostituite; la maggior diffusione di sistemi con unità disco permise di perfezionare le operazioni off-line. Il problema nelle unità a nastro era dovuto al fatto che il lettore di schede non poteva scrivere a un'estremità del nastro mentre la CPU leggeva dall'altra estremità. Prima di poter essere riavvolto e letto, il nastro doveva essere completamente scritto, giacché si trattava di un **dispositivo ad accesso sequenziale**. I sistemi a disco hanno eliminato questo problema, essendo **dispositivi ad accesso diretto**. La testina si sposta da una zona all'altra del disco, e può passare rapidamente dalla zona usata dal lettore per memorizzare il contenuto di nuove schede alla posizione richiesta dalla CPU per leggere la scheda successiva.

In un sistema a dischi le schede vengono trasferite direttamente dal lettore al disco, e la posizione delle schede viene registrata in una tabella del sistema operativo. Durante l'esecuzione di un job d'elaborazione, il sistema operativo soddisfa le richieste di lettura delle schede leggendo dal disco i dati corrispondenti alle schede richieste. Analogamente, quando il job richiede alla stampante di stampare una riga, quella riga viene copiata nella memoria del sistema e scritta nel disco. I risultati vengono effettivamente stampati quando il job è completato. Questo tipo di gestione asincrona dell'elaborazione e dell'I/O (Figura 20.5) – il cosiddetto **spooling** (*simultaneous peripheral operation on line*) – adopera fondamentalmente il disco come un enorme mezzo di bufferizzazione, per leggere in anticipo il maggior numero possibile di dati dai dispositivi d'ingresso e per memorizzare i dati in uscita fino a che i dispositivi di emissione siano in grado di accettarli.

Lo *spooling* si usa anche per elaborare dati in postazioni remote. La CPU invia i dati a una stampante remota su linee di comunicazione, oppure accetta in ingresso un

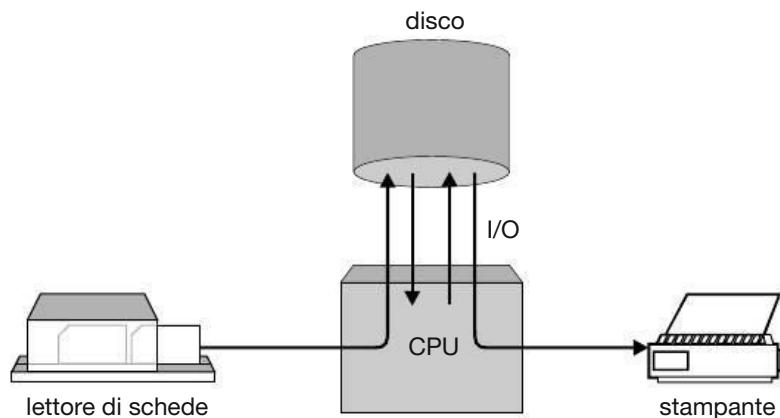


Figura 20.5 Gestione asincrona dell'elaborazione e dell'I/O (spooling).

intero job proveniente da un lettore di schede remoto. Le operazioni remote si compiono alla loro propria velocità, senza l'intervento della CPU; la CPU deve semplicemente essere informata del completamento delle operazioni in modo da intervenire sul batch successivo di dati.

Con lo spooling si sovrappone l'I/O di un job con la computazione di altri job; anche in un sistema semplice è infatti possibile leggere i dati in ingresso di un job mentre si stampano i risultati di un altro, contemporaneamente si può eseguirne un terzo (o più), leggendo le sue “schede” dal disco e “stampando” i risultati ancora nel disco.

Lo spooling ha un effetto positivo sulle prestazioni del sistema: impiegando poco spazio del disco e alcune tabelle, la computazione di un job può essere sovrapposta all'I/O di altri job; quindi permette alla CPU e ai dispositivi di I/O di lavorare a velocità maggiori. Lo spooling conduce in modo naturale alla multiprogrammazione, che è il fondamento di tutti i moderni sistemi operativi.

20.3 Atlas

Il sistema operativo Atlas è stato progettato alla University of Manchester in Inghilterra fra la fine degli anni Cinquanta e l'inizio degli anni Sessanta. Molte sue caratteristiche di base, che a quel tempo erano originali, sono diventate componenti standard dei sistemi operativi moderni. I driver dei dispositivi costituivano la parte principale del sistema. Inoltre, le chiamate di sistema erano state aggiunte attraverso un insieme di istruzioni speciali, chiamate *extra codes*.

L'Atlas era un sistema operativo batch con spooling. Ciò permetteva al sistema di eseguire lo scheduling dei job d'elaborazione secondo la disponibilità dei dispositivi periferici, come unità a nastri magnetici, lettori di nastri perforati, perforatori di nastri, stampanti, lettori di schede o perforatori di schede.

La caratteristica più notevole di questo sistema operativo era la sua gestione della memoria. A quel tempo la **memoria a nuclei magnetici** era ancora nuova e costosa. In molti calcolatori, come l'IBM 650, si usava un tamburo per la memoria centrale.

Anche nel sistema Atlas si usava un tamburo per la memoria centrale, ma come cache per il tamburo si adoperava una piccola quantità di memoria a nuclei magnetici. Per trasferire automaticamente le informazioni tra la memoria a nuclei magnetici e il tamburo si usava la paginazione su richiesta.

Il sistema operativo Atlas era eseguito su un calcolatore con parole di 48 bit. Gli indirizzi erano di 24 bit, ma erano codificati in decimali, il che permetteva di accedere a un milione di parole. A quel tempo era uno spazio d'indirizzi estremamente grande. La memoria fisica era costituita di un tamburo di 98 K parole e di 16 K parole di memoria a nuclei. La memoria era divisa in pagine di 512 parole, quindi forniva 32 pagine fisiche di memoria. Una memoria associativa di 32 registri realizzava l'associazione dagli indirizzi virtuali agli indirizzi fisici.

Se si verificava un page fault, si richiamava un algoritmo di sostituzione delle pagine. Si teneva sempre vuota una pagina fisica della memoria, perciò un trasferimento dal tamburo si poteva avviare immediatamente. L'algoritmo di sostituzione delle pagine tentava di predire il comportamento futuro dell'accesso alla memoria, basandosi sul comportamento passato. S'impostava un bit di riferimento per ciascuna pagina fisica a ogni accesso a tale pagina. I bit di riferimento venivano letti nella memoria ogni 1024 istruzioni. Gli ultimi 32 valori di questi bit si usavano per definire il tempo trascorso dal più recente riferimento (t_1) e l'intervallo di tempo trascorso tra gli ultimi due riferimenti (t_2). Si sceglievano le pagine per la sostituzione nel seguente ordine:

1. qualsiasi pagina con $t_1 > t_2 + 1$ si ritiene non sia più in uso e viene rimpiazzata;
2. se $t_1 \leq t_2$ per tutte le pagine, allora si sostituisce la pagina dove la differenza $t_2 - t_1$ è maggiore.

L'algoritmo di sostituzione delle pagine presuppone che i programmi accedano alla memoria in modo ciclico. Se il tempo trascorso tra gli ultimi due riferimenti è t_2 , il riferimento successivo è atteso dopo t_2 unità di tempo. Se non si verifica alcun riferimento ($t_1 > t_2$), si suppone che la pagina non sia più utilizzata e quindi viene sostituita. Se tutte le pagine sono ancora in uso, si sostituisce la pagina che non sarà più richiesta per il periodo di tempo più lungo. Si suppone che il riferimento successivo avvenga entro $t_2 - t_1$ unità di tempo.

20.4 XDS-940

Il sistema operativo XDS-940 è stato progettato alla University of California di Berkeley all'inizio degli anni Sessanta. Come il sistema Atlas, impiegava la paginazione per la gestione della memoria; ma diversamente dall'Atlas l'XDS-940 era un sistema time sharing.

La paginazione si usava solo per la rilocazione e non per la paginazione su richiesta. La memoria virtuale di ogni processo utente era di 16 K parole, mentre la memoria fisica conteneva 64 K parole, e le pagine erano ciascuna di 2 K parole. La tabella delle pagine era conservata in registri. Poiché la memoria fisica era più grande della memoria virtuale di un processo utente, parecchi processi potevano essere con-

temporaneamente nella memoria. Il numero degli utenti poteva essere aumentato condividendo le pagine, quando queste ultime contenevano codice rientrante di sola lettura. I processi erano conservati in un tamburo e venivano caricati nella memoria e da essa scaricati secondo le necessità.

Il sistema XDS-940 fu costruito a partire da un XDS-930 modificato. Le modifiche erano rappresentative di quelle apportate a un calcolatore semplice per permettere di scrivere adeguatamente un sistema operativo: fu aggiunto il duplice modo di funzionamento (di sistema e utente); alcune istruzioni, come quelle di I/O e d'arresto furono definite come privilegiate. Il tentativo di eseguire un'istruzione privilegiata in modalità utente avrebbe causato una trap per il sistema operativo.

All'insieme di istruzioni da eseguire in modalità utente fu aggiunta un'istruzione di chiamata di sistema. Quest'istruzione si usava per creare nuove risorse, come file, permettendo al sistema operativo di gestire le risorse fisiche. I file, per esempio, si registravano nel tamburo in blocchi di 256 parole. Per gestire i blocchi di tamburo liberi si usava una mappa di bit. Ogni file aveva un blocco indice con puntatore ai blocchi di dati effettivi. I blocchi indice erano concatenati tra loro.

Il sistema XDS-940 forniva anche chiamate di sistema per permettere ai processi di creare, avviare, sospendere ed eliminare sottoprocessi. Un programmatore poteva costruire un sistema di processi. Processi separati potevano condividere la memoria per scopi di comunicazione e sincronizzazione. La creazione di processi definiva una struttura ad albero, dove un processo costituiva la radice e i suoi sottoprocessi i nodi sottostanti nell'albero. Ognuno dei sottoprocessi poteva, a sua volta, creare altri sottoprocessi.

20.5 THE

Il sistema operativo THE è stato progettato alla Technische Hogeschool di Eindhoven in Olanda a metà degli anni Sessanta. Si tratta di un sistema batch che funzionava su un calcolatore olandese, l'EL X8, con 32 K parole di 27 bit. Il sistema si fece notare soprattutto per il suo progetto pulito, in particolare per la sua struttura a strati e il suo uso di un insieme di processi concorrenti sincronizzati tramite semafori.

Tuttavia, diversamente dal sistema XDS-940, l'insieme dei processi del sistema THE era statico. Il sistema operativo stesso era stato progettato come un insieme di processi cooperanti. Inoltre, erano stati creati cinque processi utenti, i quali servivano come agenti attivi per la compilazione, l'esecuzione e la stampa di programmi utenti. Una volta terminato un job, il processo ritornava alla coda d'ingresso per selezionarne un altro.

Si usava un algoritmo di scheduling della CPU con priorità. Le priorità si ricalcolavano ogni 2 secondi ed erano inversamente proporzionali al tempo di CPU usato recentemente, vale a dire negli ultimi 8-10 secondi. Questo schema offriva una priorità più alta ai processi con prevalenza di I/O e ai processi nuovi.

La gestione della memoria era limitata dall'assenza di adeguato supporto nell'hardware sottostante. Tuttavia, poiché il sistema era limitato e i programmi utenti

si potevano scrivere solo in Algol, s'impiegava uno schema di paginazione gestito dal sistema operativo. Il compilatore Algol generava automaticamente chiamate alle procedure di sistema, le quali assicuravano che le informazioni richieste si trovassero nella memoria, effettuando lo swapping se era necessario. La memoria ausiliaria era un tamburo di 512 K parole. Si adoperavano pagine di 512 parole, con un criterio di sostituzione delle pagine LRU.

Un altro punto importante del sistema THE era il controllo delle situazioni di stallo. Per evitare le situazioni di stallo si ricorreva all'algoritmo del banchiere.

Il sistema Venus è strettamente correlato al sistema THE. Anche il progetto del sistema Venus era basato su una struttura a strati, che impiegava semafori per sincronizzare i processi. I livelli inferiori del progetto, però, erano realizzati in microcodice, sicché il sistema era molto più veloce. Per la gestione della memoria si usava uno schema di memoria paginata e segmentata; inoltre il sistema è stato progettato come sistema time sharing, anziché come sistema batch.

20.6 RC 4000

Il sistema RC 4000, come il sistema THE, si fece notare soprattutto per i concetti applicati nella progettazione. È stato progettato alla fine degli anni Sessanta per il calcolatore danese RC 4000 dalla Regnecentralen, e in particolare da P. Brinch Hansen. L'obiettivo non era quello di progettare un sistema batch, oppure un sistema time sharing, o qualsiasi altro sistema specifico, ma quello di creare il nucleo, o kernel, di un sistema operativo, sul quale fosse possibile costruire un sistema operativo completo. Così, la struttura del sistema fu concepita a strati, e furono forniti solo i livelli inferiori, cioè il kernel.

Il kernel gestiva un insieme di processi concorrenti. Lo scheduler della CPU seguiva un criterio RR. Anche se i processi potevano condividere la memoria, il meccanismo principale di comunicazione e sincronizzazione era il **sistema di messaggi** fornito dal kernel. I processi potevano comunicare tra loro scambiandosi messaggi di dimensione fissa, 8 parole di lunghezza. Tutti i messaggi si memorizzavano in apposite sezioni della memoria (*buffer*) che facevano parte di un gruppo comune di sezioni. Quando uno di questi buffer per i messaggi non era più necessario, veniva restituita al gruppo comune.

A ogni processo era associata una **coda di messaggi**, che conteneva tutti i messaggi inviati a quel processo che non erano ancora stati ricevuti. I messaggi venivano rimossi dalla coda in ordine FIFO. Il sistema disponeva di quattro operazioni primitive, che erano eseguite in modo atomico:

- `send-message (in receiver, in message, out buffer);`
- `wait-message (out sender, out message, out buffer);`
- `send-answer (out result, in message, in buffer);`
- `wait-answer (out result, out message, in buffer).`

Le ultime due operazioni permettevano ai processi di scambiarsi più messaggi alla volta.

Queste primitive richiedevano che un processo servisse la sua coda di messaggi in ordine FIFO, e che si bloccasse mentre altri processi stavano gestendo i suoi messaggi. Per rimuovere tali limitazioni, i progettisti fornirono altre due primitive di comunicazione, che permettevano a un processo di attendere l'arrivo del messaggio successivo oppure di rispondere e servire la sua coda in qualunque ordine:

- `wait-event (in previous-buffer, out next-buffer, out result);`
- `get-event (out buffer).`

Anche i dispositivi di I/O erano trattati come processi. I driver dei dispositivi erano composti di codice che convertiva i segnali d'interruzione e i registri dei dispositivi in messaggi; così un processo scriveva in un terminale inviandogli un messaggio. Il driver del dispositivo riceveva il messaggio ed emetteva il carattere al terminale. L'immissione di un carattere interrompeva il sistema e determinava l'attivazione del driver del dispositivo, il quale a sua volta creava un messaggio contenente il carattere immesso e lo inviava al processo in attesa.

20.7 CTSS

Il sistema CTSS (*compatible time-sharing system*) è stato progettato al MIT come sistema sperimentale time sharing, e realizzato su un calcolatore IBM 7090. CTSS fece la sua apparizione nel 1961. Gestiva fino a 32 utenti interattivi. Gli utenti disponevano di un insieme di comandi interattivi, che permetteva loro di manipolare i file e di compilare ed eseguire programmi interagendo con il sistema tramite un terminale.

L'IBM 7090 aveva una memoria di 32 K parole di 36 bit. Il monitor impiegava 5 K parole, lasciando le altre 27 K parole agli utenti. Le immagini della memoria utente s'avvicendavano tra la memoria e un tamburo rapido. Per lo scheduling della CPU s'impiegava un algoritmo a code multiple con retroazione. Il quanto di tempo per il livello i era di $2 * i$ unità di tempo. Se un programma non terminava la propria sequenza di operazioni di CPU entro un quanto di tempo, veniva abbassato al livello successivo della coda, e otteneva un quanto di tempo doppio. Il programma che si trovava al livello massimo, con il quanto più corto, veniva eseguito per primo. Il livello iniziale di un programma era stabilito dalla sua dimensione in modo che il quanto di tempo fosse lungo almeno come il tempo di swap.

Il CTSS ha avuto un notevole successo ed era in uso fino al 1972. Benché limitato, esso è riuscito a dimostrare che il time sharing era un modo d'elaborazione conveniente e pratico. Un risultato ottenuto dal CTSS è stato quello di favorire lo sviluppo dei sistemi time sharing, tra i quali quello del MULTICS.

20.8 MULTICS

Il sistema operativo MULTICS è stato progettato al MIT tra il 1965 e il 1970 come un'estensione naturale del CTSS. Il CTSS e altri tra i primi sistemi time sharing ebbero un tale successo che si sentì subito il desiderio di procedere rapidamente verso sistemi più grandi e perfezionati. Quando divennero disponibili calcolatori più grandi, i progettisti del CTSS avviarono lo sviluppo di una **utilità** di elaborazione time sharing: il servizio elaborativo sarebbe stato fornito come l'energia elettrica; si sarebbero potuti collegare, attraverso linee telefoniche, grandi sistemi elaborativi a terminali di uffici e case di tutta una città. Il sistema operativo era un sistema time sharing continuamente in funzione con un vasto file system di programmi e dati condivisi.

Il MULTICS è stato progettato da un gruppo del MIT, il GE (che più tardi ha venduto il proprio dipartimento di informatica alla Honeywell), e dai Bell Laboratories (che uscirono dal progetto nel 1969). L'architettura del calcolatore GE 635 di base fu modificata in quella di un nuovo calcolatore chiamato GE 645, soprattutto per consentire la segmentazione paginata della memoria.

Un indirizzo virtuale era composto di un numero di segmento di 18 bit e di un offset di parola di 16 bit. I segmenti venivano quindi suddivisi in pagine di 1 K parole, per le quali si usava l'algoritmo di sostituzione con seconda chance.

Lo spazio d'indirizzi virtuali segmentato è stato unito al file system; ogni segmento era un file e ai segmenti si faceva riferimento tramite il nome del file corrispondente. Lo stesso file system era una struttura ad albero a più livelli, che permetteva agli utenti di creare le proprie strutture delle directory.

Come il CTSS, il MULTICS impiegava uno scheduling della CPU a code multiple con retroazione. La protezione era garantita da una lista di controllo degli accessi associata a ogni file e da un insieme di anelli di protezione per i processi in esecuzione. Il sistema, che è stato scritto quasi completamente in PL/1, comprendeva circa 300.000 righe di codice. È stato esteso a un sistema multiprocessore che permetteva di sospendere dal servizio una CPU per motivi di manutenzione mentre il sistema continuava la sua esecuzione.

20.9 OS/360 di IBM

La più lunga linea di sviluppo dei sistemi operativi è indubbiamente quella dei calcolatori IBM. I primi calcolatori IBM, come l'IBM 7090 e l'IBM 7094, sono i primi esempi dello sviluppo delle procedure di I/O comuni, seguite da un monitor residente, istruzioni privilegiate, protezione della memoria e semplice elaborazione batch. Questi sistemi sono stati sviluppati separatamente, spesso da siti indipendenti. Il risultato è stato che l'IBM si è trovata di fronte a molti calcolatori diversi, con linguaggi diversi e programmi di sistema diversi.

L'IBM/360, che fece la sua comparsa nella metà degli anni Sessanta, fu progettato per modificare tale situazione. Il suo progetto ([Mealy et al. 1966]) prevedeva una famiglia di calcolatori che si estendeva su una gamma completa che andava da piccole

macchine commerciali fino a grandi macchine scientifiche. Per questi sistemi sarebbe stato necessario un solo insieme di programmi; tutti questi sistemi impiegavano lo stesso sistema operativo: l'OS/360. Quest'orientamento doveva ridurre i problemi di manutenzione dell'IBM, e permettere agli utenti di spostare liberamente programmi e applicazioni tra i sistemi IBM.

Sfortunatamente con l'OS/360 s'è tentato di fare tutto per tutti, con il risultato di non riuscire a svolgere particolarmente bene alcun compito. Il file system comprendeva un campo di tipi che definiva il tipo di ciascun file, ed erano definiti diversi tipi di file con elementi (*record*) a lunghezza fissa o a lunghezza variabile, così come file a blocchi o non a blocchi. Si usava l'allocazione contigua, quindi l'utente doveva pre-dire la dimensione di ogni file impiegato per contenere gli output delle elaborazioni. Il linguaggio Job Control Language (JCL) prevedeva parametri per ogni possibile opzione, diventando incomprensibile all'utente medio.

Le procedure di gestione della memoria erano ostacolate dall'architettura. Sebbene si adoperasse un modo d'indirizzamento con registro di base, il programma poteva accedere al registro base e modificarlo, perciò la CPU generava indirizzi assoluti. Questa situazione ha impedito la rilocazione dinamica: l'associazione degli indirizzi del programma agli indirizzi della memoria fisica si eseguiva nella fase di caricamento. Di questo sistema operativo sono state prodotte due versioni: l'OS/MFT impiegava regioni fisse, mentre l'OS/MVT impiegava regioni variabili.

Il sistema fu scritto in linguaggio assembler da migliaia di programmatori, quindi le righe del codice risultante erano milioni. Lo stesso sistema operativo richiedeva grandi quantità di memoria per il proprio codice e le proprie tabelle. L'overhead del sistema operativo spesso richiedeva metà dei cicli totali della CPU. Con gli anni sono uscite nuove versioni mediante le quali sono state aggiunte nuove caratteristiche e sono stati corretti errori. Tuttavia, la correzione di un errore spesso ne causava un altro in una parte remota del sistema, perciò il numero degli errori noti del sistema era pressoché costante.

Con il passaggio all'architettura IBM 370, all'OS/360 è stata aggiunta la memoria virtuale. L'architettura sottostante forniva una memoria virtuale con segmentazione paginata. Le nuove versioni del sistema operativo impiegavano quest'architettura in modi diversi. L'OS/VS1 creava un grande spazio d'indirizzi virtuali e in quella memoria virtuale eseguiva l'OS/MFT. Quindi lo stesso sistema operativo era paginato, così come i programmi utenti. L'OS/VS2 Release 1 eseguiva l'OS/MVT nella memoria virtuale. Infine, l'OS/VS2 Release 2, che ora si chiama MVS, forniva a ogni utente la sua memoria virtuale.

L'MVS è ancora fondamentalmente un sistema operativo batch. Il sistema CTSS veniva eseguito su un IBM 7094, ma il MIT decise che lo spazio d'indirizzi del 360, il successore IBM del 7094, era troppo piccolo per il MULTICS, perciò cambiò fornitore. L'IBM decise allora di creare il proprio sistema time sharing, il TSS/360. Come il MULTICS, il TSS/360 era considerato come una utilità d'elaborazione time sharing. L'architettura di base del 360 fu modificata nel modello 67, per fornire la memoria virtuale. Parecchi enti acquistarono il 360/67 in vista del TSS/360.

Il TSS/360 subì però alcuni ritardi, perciò furono sviluppati altri sistemi time sharing che servivano come sistemi temporanei finché non fosse stato disponibile il TSS/360. All'OS/360 è stata aggiunta un'opzione per disporre della opzione time sharing (TSO). Il Cambridge Scientific Center dell'IBM sviluppò il CMS come sistema per utente singolo e il CP/67 per fornire una macchina virtuale su cui farlo funzionare.

Quando finalmente fu disponibile il TSS/360, si dimostrò un fallimento. Era troppo grande e troppo lento. Nessun ente cambiò il proprio sistema temporaneo con il TSS/360. Oggi, il time sharing su sistemi IBM si trova diffusamente con l'opzione TSO sul sistema MVS o anche con il CMS sul CP/67 (ribattezzato VM).

Dato il mancato successo commerciale, è lecito domandarsi quali fossero i problemi principali del TSS/360 e del MULTICS. Una parte dei problemi consisteva nel fatto che, pur trattandosi di sistemi progrediti, erano troppo grandi e troppo complessi per essere capiti. Un altro problema derivava dal presupposto che i servizi di calcolo sarebbero stati forniti da un calcolatore grande e remoto. Apparvero poi i minicomputer che diminuirono la necessità di grandi sistemi monolitici. Essi furono seguiti dalle workstation e poi dai personal computer, che portarono la potenza di calcolo sempre più vicina all'utente finale.

20.10 TOPS-20

Nella sua storia DEC ha creato diversi importanti sistemi elaborativi. Probabilmente il più famoso sistema operativo associato a DEC è VMS, un popolare sistema orientato al business tuttora utilizzato nella versione OpenVMS, un prodotto della Hewlett-Packard. Il più influente sistema operativo della DEC è però stato TOPS-20.

TOPS-20 è iniziato come un progetto di ricerca alla Bolt, Beranek e Newmann (BBN) intorno al 1970. BBN prese il computer per applicazioni business DEC PDP-10, che eseguiva TOPS-10, aggiunse un sistema hardware per la paginazione della memoria al fine di implementare la memoria virtuale e scrisse un nuovo sistema operativo per la macchina in modo da trarre vantaggio dalle nuove caratteristiche hardware. Il risultato fu TENEX, un sistema time sharing general purpose. DEC comprò quindi i diritti su TENEX e creò un nuovo computer con paginatore hardware incorporato. Ne risultarono il sistema DECSYSTEM-20 e il sistema operativo TOPS-20.

TOPS-20 disponeva di un interprete avanzato delle riga di comando in grado di offrire un aiuto agli utenti in caso di necessità. Insieme alla potenza del computer e al suo prezzo ragionevole, ciò fece del DECSYSTEM-20 il più popolare sistema time sharing del suo tempo. Nel 1984 DEC smise di lavorare sulla linea dei PDP-10 a 36 bit per concentrarsi sui sistemi VAX a 32 bit con sistema operativo VMS.

20.11 CP/M e MS/DOS

I primi computer per hobbisti venivano in genere assemblati a partire da kit, ed eseguivano un solo programma alla volta. I sistemi, con il miglioramento dei componenti dei computer, progredirono verso stadi più evoluti. Uno dei primi sistemi operativi “standard” per tali computer fu CP/M (abbreviazione di Control Program/Monitor), un sistema degli anni Settanta scritto da Gary Kindall della Digital Research. CP/M era prevalentemente installato sul primo “personal computer”, che montava una CPU a 8 bit Intel 8080. In origine CP/M supportava solo 64 KB di memoria ed era in grado di eseguire un solo programma alla volta. Ovviamente era un sistema testuale, dotato di un interprete dei comandi. L’interprete dei comandi era simile a quello di altri sistemi operativi di quei tempi, come il TOPS-10 della DEC.

Quando IBM entrò sul mercato dei personal computer decise di commissionare a Bill Gates e alla sua società il progetto di un nuovo sistema operativo per la CPU a 16 bit che era stata scelta, la Intel 8086. Questo sistema operativo, MS-DOS, era simile a CP/M, ma con un insieme più ricco di comandi incorporati, ed era ancora una volta un sistema ispirato a TOPS-10. MS-DOS divenne il sistema operativo per personal computer più popolare dei suoi tempi e dal 1981 fino al 2000 ha continuato la sua evoluzione. MS-DOS era in grado di gestire 640 KB di memoria, ma aveva la capacità di indirizzare memoria “estesa” ed “espansa” per andare oltre questo limite. Tuttavia, mancavano alcune funzionalità oggi fondamentali nei sistemi operativi, specialmente la memoria protetta.

20.12 Macintosh OS e Windows

Con l’avvento delle CPU a 16 bit i sistemi operativi diventarono più avanzati, ricchi di funzioni e usabili. Il computer **Apple Macintosh** è stato probabilmente il primo computer disegnato per gli utenti individuali dotato di una interfaccia grafica. A partire dal suo lancio nel 1984, per un certo periodo è stato sicuramente il sistema di maggior successo. Apple Macintosh utilizzava un mouse per il puntamento e la selezione sullo schermo e veniva venduto con molti programmi di utilità che traevano vantaggio da questa nuova interfaccia. I dischi fissi erano abbastanza cari nel 1984; per questa ragione Apple Macintosh veniva offerto con solo una unità floppy da 400 KB.

L’originale Mac OS veniva eseguito soltanto sui computer Apple e fu lentamente oscurato da Microsoft Windows (a partire dalla versione 1.0, distribuita nel 1985), che poteva essere installato su molti computer differenti tra loro e prodotti da diverse aziende. Quando le CPU divennero a 32 bit e incorporarono nuove funzionalità, come la memoria protetta e il cambiamento di contesto, questi sistemi operativi videro l’aggiunta di nuove caratteristiche che erano state fino ad allora appannaggio dei mainframe e dei minicomputer.

Con il passare del tempo, i personal computer divennero tanto potenti quanto quei sistemi, e più utili per diversi scopi. I minicomputer scomparvero e vennero rimpiazzati da “server” general purpose o a uso specifico. Anche se i personal computer con-

tinuano a migliorare le proprie capacità e prestazioni, i server tendono a restare sempre davanti a loro in fatto di quantità di memoria, capienza del disco, numero e velocità delle CPU disponibili. Al giorno d'oggi i server vengono in genere installati in centri di elaborazione dati e sale macchine, mentre i personal computer sono sulle scrivanie e comunicano tra loro e con i server attraverso la rete.

La rivalità tra Apple e Microsoft per il mercato dei PC continua ancora oggi con le nuove versioni di Windows e Mac OS concorrenti tra loro per funzioni, usabilità e funzionalità delle applicazioni. Altri sistemi operativi, come AmigaOS e OS/2, sono apparsi negli anni, ma nel lungo periodo non si sono dimostrati avversari temibili dei due sistemi operativi che dominano il mercato dei PC. Nel frattempo, il sistema Linux nelle sue molteplici forme continua a guadagnare popolarità tra gli utenti più esperti tecnicamente – e anche tra i non esperti, come nel caso della rete di computer per bambini **One Laptop per Child (OLPC)** (<http://laptop.org/>).

20.13 Mach

Le origini del sistema Mach risalgono al sistema operativo Accent sviluppato alla Carnegie Mellon University (CMU). Il sistema e la filosofia di comunicazione del Mach sono derivati dal sistema Accent, sebbene molte altre parti significative del sistema (per esempio, il sistema di memoria virtuale, la gestione di task e thread) furono sviluppate *ex novo*.

Il progetto di Mach è iniziato intorno alla metà degli anni Ottanta. Il sistema operativo Mach è stato progettato tenendo presenti i seguenti tre obiettivi critici:

1. emulare lo UNIX 4.3 BSD in modo che i file eseguibili per il sistema UNIX si potevessero eseguire anche nel sistema Mach;
2. essere un sistema operativo moderno che gestisse molti modelli di memoria, e l'elaborazione parallela e distribuita;
3. avere un kernel che fosse più semplice e più facile da modificare di quello del 4.3 BSD.

Lo sviluppo del sistema Mach ha seguito un percorso che parte dal sistema BSD UNIX. Il codice fu sviluppato inizialmente all'interno del kernel del 4.2BSD, sostituendo i componenti BSD con i componenti Mach appena questi venivano completati; i componenti BSD furono aggiornati alla versione 4.3BSD non appena questa divenne disponibile. Nel 1986 i sottosistemi di memoria virtuale e di comunicazione erano funzionanti per la famiglia di calcolatori DEC VAX, comprendente versioni multiprocessore. Dopo breve tempo seguirono le versioni per le workstation IBM RT/PC e SUN 3. Nel 1987 furono completate sia le versioni multiprocessore Encore Multimax e Sequent Balance, comprendenti la gestione di task e thread, sia le prime versioni ufficiali, Versione 0 e Versione 1.

Dalla Versione 2 il sistema Mach divenne compatibile con i corrispondenti sistemi BSD, poiché comprendeva nel kernel gran parte dello stesso codice BSD. Le nuove

caratteristiche e funzioni del sistema Mach fanno sì che il kernel di queste versioni sia più grande dei corrispondenti kernel BSD. Il Mach 3 trasferisce il codice BSD all'esterno del kernel, lasciando un microkernel assai più piccolo. Questo sistema incorpora nel kernel unicamente le funzioni essenziali; tutto il codice specifico dello UNIX è stato estratto per essere eseguito come server in modalità utente. L'esclusione dal kernel del codice specifico dello UNIX consente di sostituire il BSD con un altro sistema operativo o di eseguire simultaneamente sopra il microkernel più interfacce di sistemi operativi. Tali interfacce, oltre che per il BSD sono state sviluppate per l'MSDOS, per il sistema operativo Macintosh e per l'OSF/1. Questo metodo ha delle similitudini con il concetto di macchina virtuale, anche se in questo caso la macchina virtuale è definita da un livello software (l'interfaccia del kernel del Mach), anziché da una macchina fisica. Per quel che riguarda la versione 3.0, il sistema Mach divenne disponibile per molti sistemi diversi, tra cui i calcolatori con singola CPU della SUN Microsystems, Intel, IBM e DEC e i sistemi multiprocessore DEC, Sequent e Encore.

Il sistema Mach venne spinto all'attenzione dell'industria quando, nel 1989, la Open Software Foundation (OSF) annunciò che intendeva impiegarlo come base per il suo nuovo sistema operativo, l'OSF/1. L'edizione iniziale dell'OSF/1 apparve un anno dopo per competere con lo UNIX System V, versione 4, il sistema operativo scelto dai membri della UNIX International (UI). L'OSF vantava tra i suoi membri società tecnologicamente importanti come la IBM, DEC e HP. L'OSF ha nel frattempo cambiato orientamento e solo lo UNIX DEC è basato sul kernel Mach.

Diversamente da UNIX, che era stato sviluppato senza considerare la multielaborazione, il sistema operativo Mach offre un completa gestione della multielaborazione. Si tratta di una gestione molto flessibile, che varia dai sistemi con memoria condivisa tra le unità d'elaborazione ai sistemi senza memoria condivisa. Il sistema Mach usa i processi leggeri, nella forma di thread d'esecuzione multipli all'interno di un task (o spazio d'indirizzi), per gestire la multielaborazione e l'elaborazione parallela. Il suo esteso uso dei messaggi come unico metodo di comunicazione garantisce che i meccanismi di protezione siano completi ed efficienti. Integrando i messaggi con il sistema della memoria virtuale, il sistema Mach assicura che i messaggi siano gestiti in maniera efficiente. Infine, poiché il sistema di gestione della memoria virtuale impiega i messaggi per comunicare con i processi che gestiscono la memoria ausiliaria, il sistema operativo Mach consente una grande flessibilità nella progettazione e nella realizzazione dei task di gestione degli oggetti di memoria. Offrendo chiamate di sistema di basso livello, o primitive, con le quali si possono costruire funzioni complesse, il sistema Mach riduce le dimensioni del kernel permettendo l'emulazione dei sistemi operativi al livello utente in modo molto simile ai sistemi a macchine virtuali della IBM.

20.14 Altri sistemi

Naturalmente esistono altri sistemi operativi, la maggior parte dei quali ha caratteristiche interessanti. Il sistema operativo MCP, per la famiglia di calcolatori Burroughs è stato il primo a essere scritto in un linguaggio di programmazione di sistema. Impiegava anche la segmentazione e più unità d'elaborazione. Anche il sistema operativo SCOPE per il CDC 6600 era un sistema multiprocessore. Il coordinamento e la sincronizzazione dei processi multipli erano stati progettati sorprendentemente bene.

La storia è disseminata di sistemi operativi che hanno servito uno scopo per un periodo di tempo (lungo o breve che fosse) e poi, una volta scomparsi, sono stati rimpiazzati da sistemi più ricchi di funzioni, in grado di gestire hardware più moderno, e più facili da usare, o anche solamente meglio commercializzati. Siamo sicuri che questa tendenza continuerà in futuro.

Esercizi

- 20.1** Discutete le considerazioni sulla base delle quali gli operatori decidevano in quale ordine eseguire i programmi sui primi calcolatori gestiti manualmente.
- 20.2** Quali ottimizzazioni si impiegavano nei primi calcolatori per minimizzare la discrepanza di velocità fra la CPU e l'I/O?
- 20.3** Per quali aspetti l'algoritmo di sostituzione delle pagine impiegato da Atlas differisce dall'algoritmo a orologio del Paragrafo 9.4.5.2?
- 20.4** Considerate le code multiple con retroazione impiegate da CTSS e MULTICS. Supponete che un programma usi regolarmente sette unità di tempo alla volta, prima di eseguire un'operazione di I/O che lo sospenda. Quante unità di tempo sono allocate al programma, nei diversi momenti in cui la sua esecuzione è pianificata?
- 20.5** Quali sono le conseguenze della funzionalità BSD offerta da server in modalità utente all'interno del sistema operativo Mach?
- 20.6** Quali conclusioni si possono trarre circa l'evoluzione dei sistemi operativi? Quali fattori hanno portato alcuni sistemi a diventare sempre più popolari e altri a sparire?

Note bibliografiche

La storia dei calcolatori è raccontata in [Frah 2001] e mostrata graficamente in [Frauenfelder 2005].

Il Manchester Mark 1 è descritto in [Rojas e Hashagen 2000] e il suo discendente Ferranti Mark 1 in [Ceruzzi 1998].

[Kilburn et al. 1961] e [Howarth et al. 1961] presentano il sistema operativo Atlas.

Il sistema operativo XDS-940 è descritto in [Lichtenberger e Pirtle 1965].

Il sistema operativo THE è analizzato da [Dijkstra 1968] e [McKeag e Wilson 1976].

Il sistema Venus è descritto in [Liskov 1972]. [Brinch-Hansen 1970] e [Brinch-Hansen 1973] analizzano il sistema RC 4000.

[Corbato et al. 1962] presenta il sistema CTSS.

Il sistema operativo MULTICS è descritto in [Corbato e Vyssotsky 1965] e [Organick 1972].

[Mealy et al. 1966] presenta il sistema IBM/360, mentre [Lett e Konigsford 1968] presenta il TSS/360.

CP/67 è descritto in [Meyer e Seawright 1970] e [Parmelee et al. 1972].

DEC VMS è trattato in [Kenah et al. 1988] e TENEX in [Bobrow et al. 1972].

Una descrizione di Apple Macintosh appare in [Apple 1987]. Per ulteriori informazioni su questi sistemi operativi e la loro storia si veda [Freiberger e Swaine 2000].

Il sistema operativo Mach e il suo antenato Accent sono descritti da [Rashid e Robertson 1981]. Il sistema di comunicazione di Mach è trattato in [Rashid 1986], [Tevanian et al. 1989] e [Accetta et al. 1986]. Lo scheduler Mach è descritto in dettaglio in [Tevanian et al. 1987a] e [Black 1990]. Una prima versione del sistema di memoria condivisa di Mach è presentata in [Tevanian et al. 1987b].

Una buona descrizione del progetto Mach è disponibile all'indirizzo: <http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html>.

[McKeag e Wilson 1976] analizza il sistema operativo MCP per la famiglia di computer Burroughs e il sistema operativo SCOPE per il CDC 6600.

Bibliografia

- [Accetta et al. 1986] M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian e M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the Summer USENIX Conference*, p. 93–112, 1986.
- [Apple 1987] *Apple Technical Introduction to the Macintosh Family*. Addison-Wesley, 1987.
- [Black 1990] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, Vol. 23, Num. 5, p. 35–43, 1990.
- [Bobrow et al. 1972] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy e R.S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", *Communications of the ACM*, Vol. 15, Num. 3, 1972.
- [Brinch-Hansen 1970] P. Brinch-Hansen, "The Nucleus of a Multiprogramming System", *Communications of the ACM*, Vol. 13, Num. 4 , p. 238–241 e 250, 1970.
- [Brinch-Hansen 1973] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall, 1973.
- [Ceruzzi 1998] P. E. Ceruzzi, *A History of Modern Computing*, MIT Press, 1998.

- [Corbato e Vyssotsky 1965] F. J. Corbato e V. A. Vyssotsky, “Introduction and Overview of the MULTICS System”, *Proceedings of the AFIPS Fall Joint Computer Conference*, p. 185–196, 1965.
- [Corbato et al. 1962] F. J. Corbato, M. Merwin-Daggett e R. C. Daley, “An Experimental Time-Sharing System”, *Proceedings of the AFIPS Fall Joint Computer Conference*, p. 335–344, 1962.
- [Dijkstra 1968] E. W. Dijkstra, “The Structure of the THE Multiprogramming System”, *Communications of the ACM*, Vol. 11, Num. 5, p. 341–346, 1968.
- [Frah 2001] G. Frah, *The Universal History of Computing*, John Wiley and Sons, 2001.
- [Frauenfelder 2005] M. Frauenfelder, *The Computer—An Illustrated History*, Carlton Books, 2005.
- [Freiberger e Swaine 2000] P. Freiberger e M. Swaine, *Fire in the Valley—The Making of the Personal Computer*, McGraw-Hill, 2000.
- [Howarth et al. 1961] D. J. Howarth, R. B. Payne e F. H. Sumner, “The Manchester University Atlas Operating System, Part II: User’s Description”, *Computer Journal*, Vol. 4, Num. 3, p. 226–229, 1961.
- [Kenah et al. 1988] L. J. Kenah, R. E. Goldenberg e S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press, 1988.
- [Kilburn et al. 1961] T. Kilburn, D. J. Howarth, R. B. Payne e F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, *Computer Journal*, Vol. 4, Num. 3, p. 222–225, 1961.
- [Lett e Konigsford 1968] A. L. Lett e W. L. Konigsford, “TSS/360: A Time-Shared Operating System”, *Proceedings of the AFIPS Fall Joint Computer Conference*, p. 15–28, 1968.
- [Lichtenberger e Pirtle 1965] W. W. Lichtenberger e M. W. Pirtle, “A Facility for Experimentation in Man-Machine Interaction”, *Proceedings of the AFIPS Fall Joint Computer Conference*, p. 589–598, 1965.
- [Liskov 1972] B. H. Liskov, “The Design of the Venus Operating System”, *Communications of the ACM*, Vol. 15, Num. 3, p. 144–149, 1972.
- [McKeag e Wilson 1976] R. M. McKeag e R. Wilson, *Studies in Operating Systems*, Academic Press, 1976.
- [Mealy et al. 1966] G. H. Mealy, B. I. Witt e W. A. Clark, “The Functional Structure of OS/360”, *IBM Systems Journal*, Vol. 5, Num. 1, p. 3–11, 1966.
- [Meyer e Seawright 1970] R. A. Meyer e L. H. Seawright, “A Virtual Machine Time-Sharing System”, *IBM Systems Journal*, Vol. 9, Num. 3, p. 199–218, 1970.
- [Organick 1972] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
- [Parmelee et al. 1972] R. P. Parmelee, T. I. Peterson, C. C. Tillman e D. Hatfield, “Virtual Storage and Virtual Machine Concepts”, *IBM Systems Journal*, Vol. 11, Num. 2, p. 99–130, 1972.
- [Rashid 1986] R. F. Rashid, “From RIG to Accent to Mach: The Evolution of a Network Operating System”, *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference*, p. 1128–1137, 1986.

- [Rashid e Robertson 1981] R. Rashid e G. Robertson, “Accent: A Communication-Oriented Network Operating System Kernel”, *Proceedings of the ACM Symposium on Operating System Principles*, p. 64–75, 1981.
- [Rojas e Hashagen 2000] R. Rojas e U. Hashagen, *The First Computers—History and Architectures*, MIT Press, 2000.
- [Tevanian et al. 1987a] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper e M. W. Young, “Mach Threads and the Unix Kernel: The Battle for Control”, *Proceedings of the Summer USENIX Conference*, 1987.
- [Tevanian et al. 1987b] A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky e R. Sanzi, “A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach”, *Technical report*, Carnegie-Mellon University, 1987.
- [Tevanian et al. 1989] A. Tevanian, Jr. e B. Smith, “Mach: The Model for Future Unix”, Byte, 1989.

BSD UNIX



In Chapter 21, we presented an in-depth examination of the Linux operating system. In this chapter, we examine another popular UNIX version—UnixBSD. We start by presenting a brief history of the UNIX operating system. We then present the system’s user and programmer interfaces. Finally, we discuss the internal data structures and algorithms used by the FreeBSD kernel to support the user–programmer interface.

A.1 UNIX History

The first version of UNIX was developed in 1969 by Ken Thompson of the Research Group at Bell Laboratories to use an otherwise idle PDP-7. Thompson was soon joined by Dennis Ritchie and they, with other members of the Research Group, produced the early versions of UNIX.

Ritchie had previously worked on the MULTICS project, and MULTICS had a strong influence on the newer operating system. Even the name *UNIX* is a pun on *MULTICS*. The basic organization of the file system, the idea of the command interpreter (or the shell) as a user process, the use of a separate process for each command, the original line-editing characters (# to erase the last character and @ to erase the entire line), and numerous other features came directly from MULTICS. Ideas from other operating systems, such as MIT’s CTSS and the XDS-940 system, were also used.

Ritchie and Thompson worked quietly on UNIX for many years. They moved it to a PDP-11/20 for a second version; for a third version, they rewrote most of the operating system in the systems-programming language C, instead of the previously used assembly language. C was developed at Bell Laboratories to support UNIX. UNIX was also moved to larger PDP-11 models, such as the 11/45 and 11/70. Multiprogramming and other enhancements were added when it was rewritten in C and moved to systems (such as the 11/45) that had hardware support for multiprogramming.

As UNIX developed, it became widely used within Bell Laboratories and gradually spread to a few universities. The first version widely available outside Bell Laboratories was Version 6, released in 1976. (The version number for early UNIX systems corresponds to the edition number of the *UNIX*

Appendix A BSD UNIX

Programmer's Manual that was current when the distribution was made; the code and the manual were revised independently.)

In 1978, Version 7 was distributed. This UNIX system ran on the PDP-11/70 and the Interdata 8/32 and is the ancestor of most modern UNIX systems. In particular, it was soon ported to other PDP-11 models and to the VAX computer line. The version available on the VAX was known as 32V. Research has continued since then.

A.1.1 UNIX Support Group

After the distribution of Version 7 in 1978, the UNIX Support Group (USG) assumed administrative control and responsibility from the Research Group for distributions of UNIX within AT&T, the parent organization for Bell Laboratories. UNIX was becoming a product, rather than simply a research tool. The Research Group continued to develop their own versions of UNIX, however, to support their internal computing. Version 8 included a facility called the [stream I/O system](#), which allows flexible configuration of kernel IPC modules. It also contained RFS, a remote file system similar to Sun's NFS. The current version is Version 10, released in 1989 and available only within Bell Laboratories.

USG mainly provided support for UNIX within AT&T. The first external distribution from USG was System III, in 1982. System III incorporated features of Version 7 and 32V, as well as features of several UNIX systems developed by groups other than Research. For example, features of UNIX/RT, a real-time UNIX system, and numerous portions of the Programmer's Work Bench (PWB) software tools package were included in System III.

USG released System V in 1983; it is largely derived from System III. The divestiture of the various Bell operating companies from AT&T left AT&T in a position to market System V aggressively. USG was restructured as the UNIX System Development Laboratory (USDL), which released UNIX System V Release 2 (V.2) in 1984. UNIX System V Release 2, Version 4 (V.2.4) added a new implementation of virtual memory with copy-on-write paging and shared memory. USDL was in turn replaced by AT&T Information Systems (ATTIS), which distributed System V Release 3 (V.3) in 1987. V.3 adapts the V8 implementation of the stream I/O system and makes it available as STREAMS. It also includes RFS, the NFS-like remote file system mentioned earlier.

A.1.2 Berkeley Begins Development

The small size, modularity, and clean design of early UNIX systems led to UNIX-based work at numerous other computer-science organizations, such as Rand, BBN, the University of Illinois, Harvard, Purdue, and DEC. The most influential of the non-Bell Laboratories and non-AT&T UNIX development groups, however, has been the University of California at Berkeley.

Bill Joy and Ozalp Babaoglu did the first Berkeley VAX UNIX work in 1978; they added virtual memory, demand paging, and page replacement to 32V to produce 3BSD UNIX. This version was the first to implement any of these facilities on a UNIX system. The large virtual-memory space of 3BSD allowed the development of very large programs, such as Berkeley's own Franz LISP. The memory-management work convinced the Defense Advanced Research Projects Agency (DARPA) to fund Berkeley for the development of a standard UNIX system for government use; 4 BSD UNIX was the result.

The 4 BSD work for DARPA was guided by a steering committee that included many notable people from the UNIX and networking communities. One of the goals of this project was to provide support for the DARPA Internet networking protocols (TCP/IP). This support was provided in a general manner. It is possible in 4.2 BSD to communicate uniformly among diverse network facilities, including local-area networks (such as Ethernets and token rings) and wide-area networks (such as NSFNET). This implementation was the most important reason for the current popularity of these protocols. It was used as the basis for the implementations of many vendors of UNIX computer systems, and even other operating systems. It permitted the Internet to grow from 60 connected networks in 1984 to more than 8,000 networks and an estimated 10 million users in 1993.

In addition, Berkeley adapted many features from contemporary operating systems to improve the design and implementation of UNIX. Many of the terminal line-editing functions of the TENEX (TOPS-20) operating system were provided by a new terminal driver. A new user interface (the C Shell), a new text editor (ex/vi), compilers for Pascal and LISP, and many new systems programs were written at Berkeley. For 4.2 BSD, certain efficiency improvements were inspired by the VMS operating system.

UNIX software from Berkeley is released in **Berkeley Software Distributions (BSD)**. It is convenient to refer to the Berkeley VAX UNIX systems following 3BSD as 4 BSD, but there were actually several specific releases, most notably 4.1 BSD and 4.2 BSD. The generic numbers BSD and 4 BSD are used for the PDP-11 and VAX distributions of Berkeley UNIX. 4.2 BSD, first distributed in 1983, was the culmination of the original Berkeley DARPA UNIX project. 2.9 BSD is the equivalent version for PDP-11 systems.

In 1986, 4.3 BSD was released. It was very similar to 4.2 BSD but included numerous internal changes, such as bug fixes and performance improvements. Some new facilities were also added, including support for the Xerox Network System protocols.

4.3 BSD Tahoe was the next version, released in 1988. It included improved networking congestion control and TCP/IP performance. Disk configurations were separated from the device drivers and read off the disks themselves. Expanded time-zone support was also included. 4.3 BSD Tahoe was actually developed on and for the CCI Tahoe system (Computer Console, Inc., Power 6 computer), rather than for the usual VAX base. The corresponding PDP-11 release is 2.10.1 BSD; it is distributed by the USENIX association, which also publishes the 4.3 BSD manuals. The 4.3 2 BSD Reno release saw the inclusion of an implementation of ISO/OSI networking.

The last Berkeley release, 4.4 BSD, was finalized in June of 1993. It includes new X.25 networking support and POSIX standard compliance. It also has a radically new file system organization, with a new virtual file system interface and support for *stackable* file systems, allowing file systems to be layered on top of each other for easy inclusion of new features. An implementation of NFS is included in the release (Chapter 17), as is a new log-based file system (see Chapter 12). The 4.4 BSD virtual memory system is derived from Mach (described in Section 23.13). Several other changes, such as enhanced security and improved kernel structure, are also included. With the release of version 4.4, Berkeley halted its research efforts.

A.1.3 The Spread of UNIX

4 BSD was the operating system of choice for the VAX from its initial release (in 1979) until the release of Ultrix, DEC's BSD implementation. 4 BSD is still the best choice for many research and networking installations. The current set of UNIX operating systems is not limited to those by Bell Laboratories (which is currently owned by Lucent Technology) and Berkeley, however. Sun Microsystems helped popularize the BSD flavor of UNIX by shipping it on Sun workstations. As UNIX grew in popularity, it was moved to many computers and computer systems. A wide variety of UNIX and UNIX-like operating systems have been created. DEC supports its UNIX (Ultrix) on its workstations and is replacing Ultrix with another UNIX-derived operating system, OSF/1; Microsoft rewrote UNIX for the Intel 8088 family and called it XENIX, and its new Windows NT operating system is heavily influenced by UNIX; IBM has UNIX (AIX) on its PCs, workstations, and mainframes. In fact, UNIX is available on almost all general-purpose computers; it runs on personal computers, workstations, minicomputers, mainframes, and supercomputers, from Apple Macintosh IIs to Cray IIs. Because of its wide availability, it is used in environments ranging from academic to military to manufacturing process control. Most of these systems are based on Version 7, System III, 4.2 BSD, or System V.

The wide popularity of UNIX with computer vendors has made UNIX the most portable of operating systems, and users can expect a UNIX environment independent of any specific computer manufacturer. But the large number of implementations of the system has led to remarkable variation in the programming and user interfaces distributed by the vendors. For true vendor independence, application-program developers need consistent interfaces. Such interfaces would allow all "UNIX" applications to run on all UNIX systems, which is certainly not the current situation. This issue has become important as UNIX has become the preferred program-development platform for applications ranging from databases to graphics and networking, and it has led to a strong market demand for UNIX standards.

Several standardization projects are underway, starting with the */usr/group 1984 Standard*, sponsored by the UniForum industry user's group. Since then, many official standards bodies have continued the effort, including IEEE and ISO (the POSIX standard). The X/Open Group international consortium completed XPG3, a Common Application Environment, which subsumes the IEEE interface standard. Unfortunately, XPG3 is based on a draft of the ANSI C standard, rather than the final specification, and therefore needed to be redone as XPG4. In 1989, the ANSI standards body standardized the C programming language, producing an ANSI C specification that vendors were quick to adopt.

As such projects continue, the flavors of UNIX will converge and lead to one programming interface to UNIX, allowing UNIX to become even more popular. In fact, two separate sets of powerful UNIX vendors are working on this problem: The AT&T-guided UNIX International (UI) and the Open Software Foundation (OSF) have both agreed to follow the POSIX standard. Recently, many of the vendors involved in those two groups have agreed on further standardization (the COSE agreement).

AT&T replaced its ATTIS group in 1989 with the UNIX Software Organization (USO), which shipped the first merged UNIX, System V Release 4. This system combines features from System V, 4.3 BSD, and Sun's SunOS, including long file

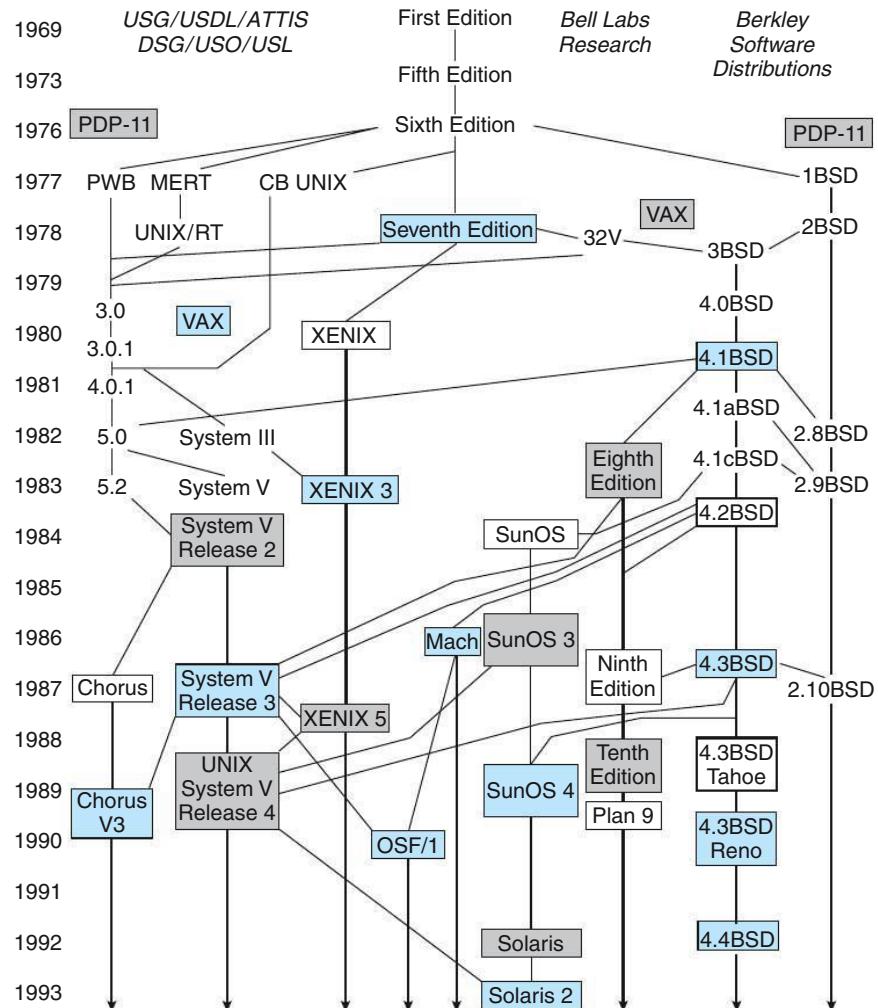


Figure A.1 History of UNIX versions up to 1993.

names, the Berkeley file system, virtual memory management, symbolic links, multiple access groups, job control, and reliable signals; it also conforms to the published POSIX standard, POSIX.1. After USO produced SVR4, it became an independent AT&T subsidiary named Unix System Laboratories (USL); in 1993, it was purchased by Novell, Inc. Figure A.1 summarizes the relationships among the various versions of UNIX.

The UNIX system has grown from a personal project of two Bell Laboratories employees to an operating system being defined by multinational standardization bodies. At the same time, UNIX is an excellent vehicle for academic study, and we believe it will remain an important part of operating-system theory and practice. For example, the Tunis operating system, the Xinu operating system,

and the Minix operating system are based on the concepts of UNIX but were developed explicitly for classroom study. There is a plethora of ongoing UNIX-related research systems, including Mach, Chorus, Comandos, and Roisin. The original developers, Ritchie and Thompson, were honored in 1983 by the Association for Computing Machinery Turing Award for their work on UNIX.

A.1.4 History of FreeBSD

The specific UNIX version used in this chapter is the Intel version of FreeBSD. This system implements many interesting operating-system concepts, such as demand paging with clustering, and networking. The FreeBSD project began in early 1993 to produce a snapshot of 386 BSD to solve problems that could not be resolved using the existing patch mechanism. 386 BSD was derived from 4.3 BSD-Lite (Net/2) and was released in June 1992 by William Jolitz. FreeBSD (coined by David Greenman) 1.0 was released in December 1993, and FreeBSD 1.1 was released in May 1994. Both versions were based on 4.3 BSD-Lite. Legal issues between UCB and Novell required that 4.3 BSD-Lite code no longer be used, so the final 4.3 BSD-Lite Release was made in July 1994 (FreeBSD 1.1.5.1).

FreeBSD was then reinvented based on 4.4BSD-Lite code, which was incomplete. FreeBSD 2.0 was released in November 1994. Later releases include 2.0.5 in June 1995, 2.1.5 in August 1996, 2.1.7.1 in February 1997, 2.2.1 in April 1997, 2.2.8 in November 1998, 3.0 in October 1998, 3.1 in February 1999, 3.2 in May 1999, 3.3 in September 1999, 3.4 in December 1999, 3.5 in June 2000, 4.0 in March 2000, 4.1 in July 2000, and 4.2 in November 2000.

The goal of the FreeBSD project is to provide software that can be used for any purpose with no strings attached. The idea is that the code will get the widest possible use and provide the most benefit. Fundamentally, FreeBSD is the same as described in McKusick et al. [1984] with the addition of a merged virtual memory and file-system buffer cache, kernel queues, and soft file-system updates. At present, it runs primarily on Intel platforms, although Alpha platforms are supported. Work is underway to port to other processor platforms as well.

A.2 Design Principles

UNIX was designed to be a time-sharing system. The standard user interface (the shell) is simple and can be replaced by another, if desired. The file system is a multilevel tree, which allows users to create their own subdirectories. Each user data file is simply a sequence of bytes.

Disk files and I/O devices are treated as similarly as possible. Thus, device dependencies and peculiarities are kept in the kernel as much as possible; even in the kernel, most of them are confined to the device drivers.

UNIX supports multiple processes. A process can easily create new processes. CPU scheduling is a simple priority algorithm. FreeBSD uses demand paging as a mechanism to support memory-management and CPU-scheduling decisions. Swapping is used if a system is suffering from excess paging.

Because UNIX was originated by Thompson and Ritchie as a system for their own convenience, it was small enough to understand. Most of the algorithms were selected for *simplicity*, not for speed or sophistication. The intent was to

have the kernel and libraries provide a small set of facilities that was sufficiently powerful to allow a person to build a more complex system if needed. UNIX's clean design has resulted in many imitations and modifications.

Although the designers of UNIX had a significant amount of knowledge about other operating systems, UNIX had no elaborate design spelled out before its implementation. This flexibility appears to have been one of the key factors in the development of the system. Some design principles were involved, however, even though they were not made explicit at the outset.

The UNIX system was designed by programmers for programmers. Thus, it has always been interactive, and facilities for program development have always been a high priority. Such facilities include the program *make* (which can be used to check which of a collection of source files for a program need to be compiled and then to do the compiling) and the *Source Code Control System (SCCS)* (which is used to keep successive versions of files available without having to store the entire contents of each step). The primary version-control system used by UNIX is the *Concurrent Versions System (CVS)* due to the large number of developers operating on and using the code.

The operating system is written mostly in C, which was developed to support UNIX, since neither Thompson nor Ritchie enjoyed programming in assembly language. The avoidance of assembly language was also necessary because of the uncertainty about the machines on which UNIX would be run. It has greatly simplified the problems of moving UNIX from one hardware system to another.

From the beginning, UNIX development systems have had all the UNIX sources available online, and the developers have used the systems under development as their primary systems. This pattern of development has greatly facilitated the discovery of deficiencies and their fixes, as well as of new possibilities and their implementations. It has also encouraged the plethora of UNIX variants existing today, but the benefits have outweighed the disadvantages: If something is broken, it can be fixed at a local site; there is no need to wait for the next release of the system. Such fixes, as well as new facilities, may be incorporated into later distributions.

The size constraints of the PDP-11 (and earlier computers used for UNIX) have forced a certain elegance. Where other systems have elaborate algorithms for dealing with pathological conditions, UNIX just does a controlled crash called *panic*. Instead of attempting to cure such conditions, UNIX tries to prevent them. Where other systems would use brute force or macro-expansion, UNIX mostly has had to develop more subtle, or at least simpler, approaches.

These early strengths of UNIX produced much of its popularity, which in turn produced new demands that challenged those strengths. UNIX was used for tasks such as networking, graphics, and real-time operation, which did not always fit into its original text-oriented model. Thus, changes were made to certain internal facilities, and new programming interfaces were added. Supporting these new facilities and others—particularly window interfaces—required large amounts of code, radically increasing the size of the system. For instance, networking and windowing both doubled the size of the system. This pattern in turn pointed out the continued strength of UNIX—whenever a new development occurred in the industry, UNIX could usually absorb it but remain UNIX.

A.3 Programmer Interface

Like most operating systems, UNIX consists of two separable parts: the kernel and the systems programs. We can view the UNIX operating system as being layered, as shown in Figure A.2. Everything below the system-call interface and above the physical hardware is the *kernel*. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Systems programs use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

System calls define the *programmer interface* to UNIX; the set of systems programs commonly available defines the *user interface*. The programmer and user interface define the context that the kernel must support.

Most systems programs are written in C, and the *UNIX Programmer's Manual* presents all system calls as C functions. A system program written in C for FreeBSD on the Pentium can generally be moved to an Alpha FreeBSD system and simply recompiled, even though the two systems are quite different. The details of system calls are known only to the compiler. This feature is a major reason for the portability of UNIX programs.

System calls for UNIX can be roughly grouped into three categories: file manipulation, process control, and information manipulation. In Chapter 2, we listed a fourth category, device manipulation, but since devices in UNIX are treated as (special) files, the same system calls support both files and devices (although there is an extra system call for setting device parameters).

A.3.1 File Manipulation

A *file* in UNIX is a sequence of bytes. Different programs expect various levels of structure, but the kernel does not impose a structure on files. For instance, the convention for text files is lines of ASCII characters separated by a single newline character (which is the linefeed character in ASCII), but the kernel knows nothing of this convention.

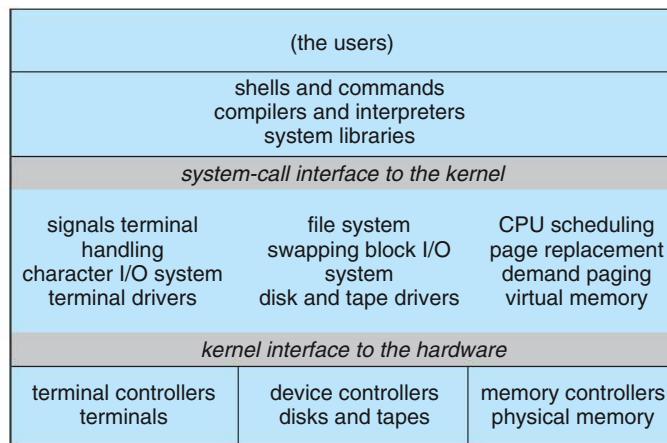


Figure A.2 4.4BSD layer structure.

Files are organized in tree-structured *directories*. Directories are themselves files that contain information on how to find other files. A *path name* to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically, it consists of individual file-name elements separated by the slash character. For example, in */usr/local/font*, the first slash indicates the root of the directory tree, called the *root* directory. The next element, *usr*, is a subdirectory of the root, *local* is a subdirectory of *usr*, and *font* is a file or directory in the directory *local*. Whether *font* is an ordinary file or a directory cannot be determined from the path-name syntax.

The UNIX file system has both *absolute path names* and *relative path names*. Absolute path names start at the root of the file system and are distinguished by a slash at the beginning of the path name; */usr/local/font* is an absolute path name. Relative path names start at the *current directory*, which is an attribute of the process accessing the path name. Thus, *local/font* indicates a file or directory named *font* in the directory *local* in the current directory, which might or might not be */usr*.

A file may be known by more than one name in one or more directories. Such multiple names are known as *links*, and all links are treated equally by the operating system. FreeBSD also supports *symbolic links*, which are files containing the path name of another file. The two kinds of links are also known as *hard links* and *soft links*. Soft (symbolic) links, unlike hard links, may point to directories and may cross file-system boundaries.

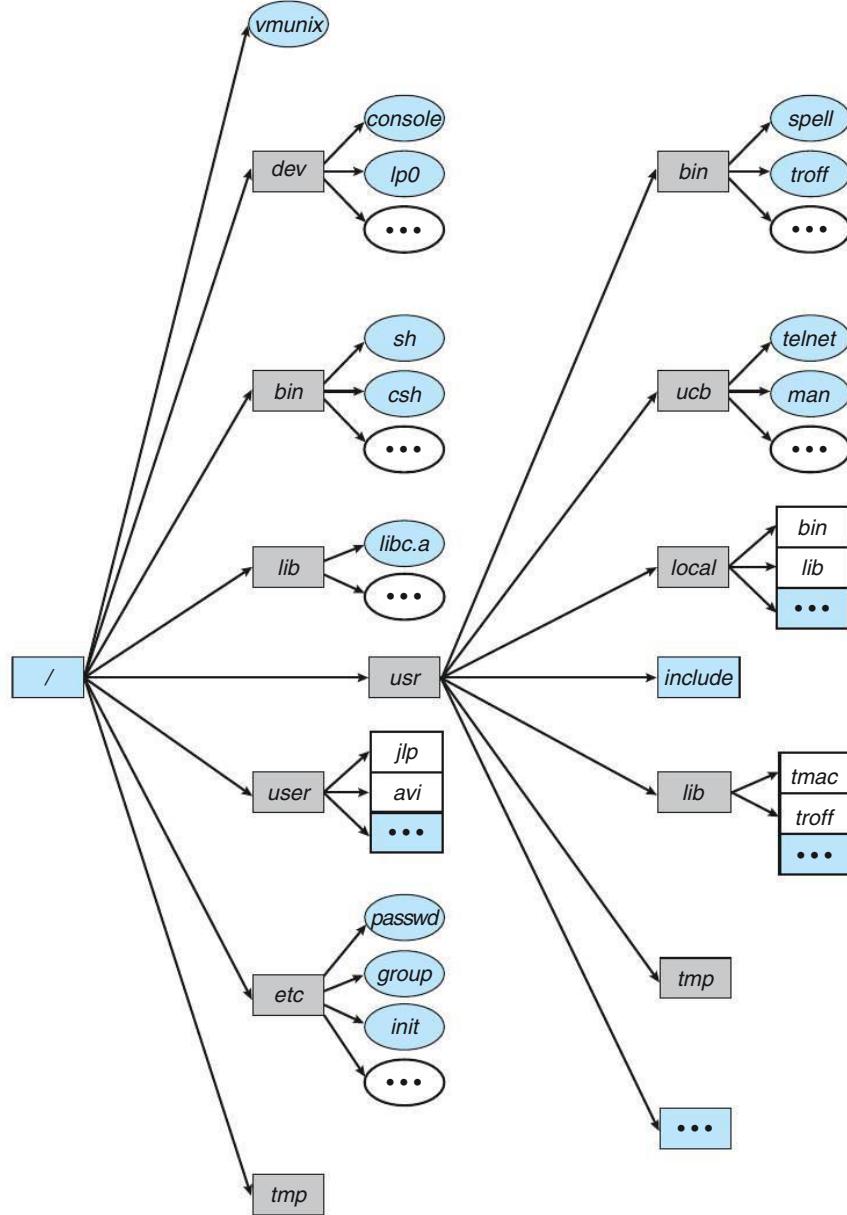
The file name “.” in a directory is a hard link to the directory itself. The file name “..” is a hard link to the parent directory. Thus, if the current directory is */user/jlp/programs*, then *../bin/wdf* refers to */user/jlp/bin/wdf*.

Hardware devices have names in the file system. These *device special files* or *special files* are known to the kernel as device interfaces, but they are nonetheless accessed by the user by much the same system calls as are other files.

Figure A.3 shows a typical UNIX file system. The root (/) normally contains a small number of directories as well as */kernel*, the binary boot image of the operating system; */dev* contains the device special files, such as */dev/console*, */dev/lp0*, */dev/mt0*, and so on; and */bin* contains the binaries of the essential UNIX systems programs. Other binaries may be in */usr/bin* (for applications systems programs, such as text formatters), */usr/compat* (for programs from other operating systems, such as Linux), or */usr/local/bin* (for systems programs written at the local site). Library files—such as the C, Pascal, and FORTRAN subroutine libraries—are kept in */lib* (or */usr/lib* or */usr/local/lib*).

The files of users themselves are stored in a separate directory for each user, typically in */usr*. Thus, the user directory for *carol* would normally be in */usr/carol*. For a large system, these directories may be further grouped to ease administration, creating a file structure with */usr/prof/avi* and */usr/staff/carol*. Administrative files and programs, such as the password file, are kept in */etc*. Temporary files can be put in */tmp*, which is normally erased during system boot, or in */usr/tmp*.

Each of these directories may have considerably more structure. For example, the font-description tables for the troff formatter for the Mergenthaler 202 typesetter are kept in */usr/lib/troff/dev202*. All the conventions concerning the location of specific files and directories have been defined by programmers and their programs; the operating-system kernel needs only */etc/init*, which is used to initialize terminal processes, to be operable.

**Figure A.3** Typical UNIX directory structure.

System calls for basic file manipulation are **creat**, **open**, **read**, **write**, **close**, **unlink**, and **trunc**. The **creat** system call, given a path name, creates an (empty) file (or truncates an existing one). An existing file is opened by the **open** system call, which takes a path name and a mode (such as read, write, or read-write)

and returns a small integer, called a *file descriptor*. A file descriptor may then be passed to a `read` or `write` system call (along with a buffer address and the number of bytes to transfer) to perform data transfers to or from the file. A file is closed when its file descriptor is passed to the `close` system call. The `trunc` call reduces the length of a file to 0.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files.

Each `read` or `write` updates the current offset into the file, which is associated with the file-table entry and is used to determine the position in the file for the next `read` or `write`. The `lseek` system call allows the position to be reset explicitly. It also allows the creation of sparse files (files with “holes” in them). The `dup` and `dup2` system calls can be used to produce a new file descriptor that is a copy of an existing one. The `fcntl` system call can also do that and in addition can examine or set various parameters of an open file. For example, it can make each succeeding `write` to an open file append to the end of that file. There is an additional system call, `ioctl`, for manipulating device parameters. It can set the baud rate of a serial port or rewind a tape, for instance.

Information about the file (such as its size, protection modes, owner, and so on) can be obtained by the `stat` system call. Several system calls allow some of this information to be changed: `rename` (change file name), `chmod` (change the protection mode), and `chown` (change the owner and group). Many of these system calls have variants that apply to file descriptors instead of file names. The `link` system call makes a hard link for an existing file, creating a new name for an existing file. A link is removed by the `unlink` system call; if it is the last link, the file is deleted. The `symlink` system call makes a symbolic link.

Directories are made by the `mkdir` system call and are deleted by `rmdir`. The current directory is changed by `cd`.

Although the standard file calls (`open` and others) can be used on directories, it is inadvisable to do so, since directories have an internal structure that must be preserved. Instead, another set of system calls is provided to open a directory, to step through each file entry within the directory, to close the directory, and to perform other functions; these are `opendir`, `readdir`, `closedir`, and others.

A.3.2 Process Control

A *process* is a program in execution. Processes are identified by their *process identifier*, which is an integer. A new process is created by the `fork` system call. The new process consists of a copy of the address space of the original process (the same program and the same variables with the same values). Both processes (the parent and the child) continue execution at the instruction after the `fork` with one difference: The return code for the `fork` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the `execve` system call is used after a fork by one of the two processes to replace that process’s virtual memory space with a new program. The `execve` system call loads a binary file into memory (destroying the

memory image of the program containing the `execve` system call) and starts its execution.

A process may terminate by using the `exit` system call, and its parent process may wait for that event by using the `wait` system call. If the child process crashes, the system simulates the `exit` call. The `wait` system call provides the process ID of a terminated child so that the parent can tell which of possibly many children terminated. A second system call, `wait3`, is similar to `wait` but also allows the parent to collect performance statistics about the child. Between the time the child exits and the time the parent completes one of the `wait` system calls, the child is *defunct*. A defunct process can do nothing but exists merely so that the parent can collect its status information. If the parent process of a defunct process exits before a child, the defunct process is inherited by the *init* process (which in turn `waits` on it) and becomes a *zombie* process. A typical use of these facilities is shown in Figure A.4.

The simplest form of communication between processes is by *pipes*, which may be created before the `fork` and whose endpoints are then set up between the `fork` and the `execve`. A pipe is essentially a queue of bytes between two processes. The pipe is accessed by a file descriptor, like an ordinary file. One process writes into the pipe, and the other reads from the pipe. The size of the original pipe system was fixed by the system. With FreeBSD pipes are implemented on top of the socket system, which has variable-sized buffers. Reading from an empty pipe or writing into a full pipe causes the process to be blocked until the state of the pipe changes. Special arrangements are needed for a pipe to be placed between a parent and child (so only one is reading and one is writing).

All user processes are descendants of one original process, called *init* (which has process identifier 1). Each terminal port available for interactive use has a *getty* process forked for it by *init*. The *getty* process initializes terminal line parameters and waits for a user's *login name*, which it passes through an `execve` as an argument to a *login* process. The *login* process collects the user's password, encrypts it, and compares the result to an encrypted string taken from the file */etc/passwd*. If the comparison is successful, the user is allowed to log in. The *login* process executes a *shell*, or command interpreter, after setting the numeric *user identifier* of the process to that of the user logging in. (The shell and the user identifier are found in */etc/passwd* by the user's login name.) It is with this shell that the user ordinarily communicates for the rest of the login session; the shell itself forks subprocesses for the commands the user tells it to execute.

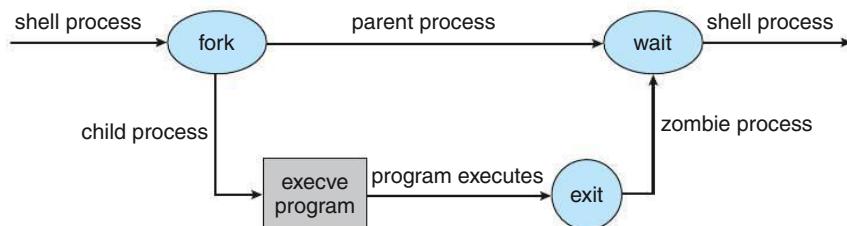


Figure A.4 A shell forks a subprocess to execute a program.

The user identifier is used by the kernel to determine the user's permissions for certain system calls, especially those involving file accesses. There is also a *group identifier*, which is used to provide similar privileges to a collection of users. In FreeBSD a process may be in several groups simultaneously. The *login* process puts the shell in all the groups permitted to the user by the files */etc/passwd* and */etc/group*.

Two user identifiers are used by the kernel: the effective user identifier and the real user identifier. The *effective user identifier* is used to determine file access permissions. If the file of a program being loaded by an `execve` has the `setuid` bit set in its inode, the effective user identifier of the process is set to the user identifier of the owner of the file, whereas the *real user identifier* is left as it was. This scheme allows certain processes to have more than ordinary privileges while still being executable by ordinary users. The `setuid` idea was patented by Dennis Ritchie (U.S. Patent 4,135,240) and is one of the distinctive features of UNIX. A similar `setgid` bit exists for groups. A process may determine its real and effective user identifier with the `getuid` and `geteuid` calls, respectively. The `getgid` and `getegid` calls determine the process's real and effective group identifier, respectively. The rest of a process's groups may be found with the `getgroups` system call.

A.3.3 Signals

Signals are a facility for handling exceptional conditions similar to software interrupts. There are 20 different signals, each corresponding to a distinct condition. A signal may be generated by a keyboard interrupt, by an error in a process (such as a bad memory reference), or by a number of asynchronous events (such as timers or job-control signals from the shell). Almost any signal may also be generated by the `kill` system call.

The *interrupt* signal, SIGINT, is used to stop a command before that command completes. It is usually produced by the ^C character (ASCII 3). As of 4.2 BSD, the important keyboard characters are defined by a table for each terminal and can be redefined easily. The *quit* signal, SIGQUIT, is usually produced by the ^bs character (ASCII 28). The *quit* signal both stops the currently executing program and dumps its current memory image to a file named *core* in the current directory. The *core* file can be used by debuggers. SIGILL is produced by an illegal instruction and SIGSEGV by an attempt to address memory outside of the legal virtual-memory space of a process.

Arrangements can be made either for most signals to be ignored (to have no effect) or for a routine in the user process (a signal handler) to be called. A signal handler may safely do one of two things before returning from catching a signal: call the `exit` system call or modify a global variable. One signal (the *kill* signal, number 9, SIGKILL) cannot be ignored or caught by a signal handler. SIGKILL is used, for example, to kill a runaway process that is ignoring other signals such as SIGINT or SIGQUIT.

Signals can be lost. If another signal of the same kind is sent before a previous signal has been accepted by the process to which it is directed, the first signal will be overwritten and only the last signal will be seen by the process. In other words, a call to the signal handler tells a process that there has been at least one occurrence of the signal. Also, there is no relative priority

among UNIX signals. If two different signals are sent to the same process at the same time, we cannot know which one the process will receive first.

Signals were originally intended to deal with exceptional events. As is true of the use of most UNIX features, however, signal use has steadily expanded. 4.1BSD introduced job control, which uses signals to start and stop subprocesses on demand. This facility allows one shell to control multiple processes—starting, stopping, and backgrounding them as the user wishes. The SIGWINCH signal, invented by Sun Microsystems, is used for informing a process that the window in which output is being displayed has changed size. Signals are also used to deliver urgent data from network connections.

Users also wanted more reliable signals and a bug fix in an inherent race condition in the old signals implementation. Thus, 4.2BSD brought with it a race-free, reliable, separately implemented signal capability. It allows individual signals to be blocked during critical sections, and it has a new system call to let a process sleep until interrupted. It is similar to hardware-interrupt functionality. This capability is now part of the POSIX standard.

A.3.4 Process Groups

Groups of related processes frequently cooperate to accomplish a common task. For instance, processes may create, and communicate over, pipes. Such a set of processes is termed a *process group*, or a *job*. Signals may be sent to all processes in a group. A process usually inherits its process group from its parent, but the `setpgrp` system call allows a process to change its group.

Process groups are used by the C shell to control the operation of multiple jobs. Only one process group may use a terminal device for I/O at any time. This *foreground* job has the attention of the user on that terminal while all other nonattached jobs (*background* jobs) perform their functions without user interaction. Access to the terminal is controlled by process group signals. Each job has a *controlling terminal* (again, inherited from its parent). If the process group of the controlling terminal matches the group of a process, that process is in the foreground and is allowed to perform I/O. If a nonmatching (*background*) process attempts the same, a SIGTTIN or SIGTTOU signal is sent to its process group. This signal usually causes the process group to freeze until it is foregrounded by the user, at which point it receives a SIGCONT signal, indicating that the process can perform the I/O. Similarly, a SIGSTOP may be sent to the foreground process group to freeze it.

A.3.5 Information Manipulation

System calls exist to set and return both an interval timer (`getitimer`/`setitimer`) and the current time (`gettimeofday`/`settimeofday`) in microseconds. In addition, processes can ask for their process identifier (`getpid`), their group identifier (`getgid`), the name of the machine on which they are executing (`gethostname`), and many other values.

A.3.6 Library Routines

The system-call interface to UNIX is supported and augmented by a large collection of library routines and header files. The header files provide the

definition of complex data structures used in system calls. In addition, a large library of functions provides additional program support.

For example, the UNIX I/O system calls provide for the reading and writing of blocks of bytes. Some applications may want to read and write only 1 byte at a time. Although possible, that would require a system call for each byte—a very high overhead. Instead, a set of standard library routines (the standard I/O package accessed through the header file `<stdio.h>`) provides another interface, which reads and writes several thousand bytes at a time using local buffers and transfers between these buffers (in user memory) when I/O is desired. Formatted I/O is also supported by the standard I/O package.

Additional library support is provided for mathematical functions, network access, data conversion, and so on. The FreeBSD kernel supports over 300 system calls; the C program library has over 300 library functions. The library functions eventually result in system calls where necessary (for example, the `getchar` library routine will result in a `read` system call if the file buffer is empty). However, the programmer generally does not need to distinguish between the basic set of kernel system calls and the additional functions provided by library functions.

A.4 User Interface

Both the programmer and the user of a UNIX system deal mainly with the set of systems programs that have been written and are available for execution. These programs make the necessary system calls to support their function, but the system calls themselves are contained within the program and do not need to be obvious to the user.

The common systems programs can be grouped into several categories; most of them are file or directory oriented. For example, the systems programs to manipulate directories are `mkdir` to create a new directory, `rmdir` to remove a directory, `cd` to change the current directory to another, and `pwd` to print the absolute path name of the current (working) directory.

The `ls` program lists the names of the files in the current directory. Any of 28 options can ask that properties of the files be displayed also. For example, the `-l` option asks for a long listing, showing the file name, owner, protection, date and time of creation, and size. The `cp` program creates a new file that is a copy of an existing file. The `mv` program moves a file from one place to another in the directory tree. In most cases, this move simply requires a renaming of the file; if necessary, however, the file is copied to the new location and the old copy is deleted. A file is deleted by the `rm` program (which makes an `unlink` system call).

To display a file on the terminal, a user can run `cat`. The `cat` program takes a list of files and concatenates them, copying the result to the standard output, commonly the terminal. On a high-speed cathode-ray tube (CRT) display, of course, the file may speed by too fast to be read. The `more` program displays the file one screen at a time, pausing until the user types a character to continue to the next screen. The `head` program displays just the first few lines of a file; `tail` shows the last few lines.

These are the basic systems programs widely used in UNIX. In addition, there are a number of editors (`ed`, `sed`, `emacs`, `vi`, and so on), compilers (C, Pascal,

FORTRAN, and so on), and text formatters (troff, TEX, scribe, and so on). There are also programs for sorting (*sort*) and comparing files (*cmp*, *diff*), looking for patterns (*grep*, *awk*), sending mail to other users (*mail*), and many other activities.

A.4.1 Shells and Commands

Both user-written and systems programs are normally executed by a command interpreter. The command interpreter in UNIX is a user process like any other. It is called a *shell*, as it surrounds the kernel of the operating system. Users can write their own shell, and, in fact, several shells are in general use. The *Bourne shell*, written by Steve Bourne, is probably the most widely used—or, at least, the most widely available. The *C shell*, mostly the work of Bill Joy, a founder of Sun Microsystems, is the most popular on BSD systems. The Korn shell, by Dave Korn, has become popular because it combines the features of the Bourne shell and the C shell.

The common shells share much of their command-language syntax. UNIX is normally an interactive system. The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line. For instance, in the line

```
% ls -l
```

the percent sign is the usual C shell prompt, and the *ls -l* (typed by the user) is the (long) list-directory command. Commands can take arguments, which the user types after the command name on the same line, separated by white space (spaces or tabs).

Although a few commands are built into the shells (such as *cd*), a typical command is an executable binary object file. A list of several directories, the *search path*, is kept by the shell. For each command, each of the directories in the search path is searched, in order, for a file of the same name. If a file is found, it is loaded and executed. The search path can be set by the user. The directories */bin* and */usr/bin* are almost always in the search path, and a typical search path on a FreeBSD system might be

```
( ./usr/avi/bin /usr/local/bin /bin /usr/bin )
```

The *ls* command's object file is */bin/ls*, and the shell itself is */bin/sh* (the Bourne shell) or */bin/csh* (the C shell).

Execution of a command is done by a **fork** system call followed by an **execve** of the object file. The shell usually then does a **wait** to suspend its own execution until the command completes (Figure A.4). There is a simple syntax (an ampersand [&] at the end of the command line) to indicate that the shell should *not* wait for the completion of the command. A command left running in this manner while the shell continues to interpret further commands is said to be a *background* command, or to be running in the background. Processes for which the shell *does* wait are said to run in the *foreground*.

The C shell in FreeBSD systems provides a facility called *job control* (partially implemented in the kernel), as mentioned previously. Job control allows processes to be moved between the foreground and the background. The

command	meaning of command
% <i>ls</i> > <i>filea</i>	direct output of <i>ls</i> to file <i>filea</i>
% <i>pr</i> < <i>filea</i> > <i>fileb</i>	input from <i>filea</i> and output to <i>fileb</i>
% <i>lpr</i> < <i>fileb</i>	input from <i>fileb</i>
% % make program > & errs	save both standard output and standard error in a file

Figure A.5 Standard I/O redirection.

processes can be stopped and restarted on various conditions, such as a background job wanting input from the user's terminal. This scheme allows most of the control of processes provided by windowing or layering interfaces but requires no special hardware. Job control is also useful in window systems, such as the X Window System developed at MIT. Each window is treated as a terminal, allowing multiple processes to be in the foreground (one per window) at any one time. Of course, background processes may exist on any of the windows. The Korn shell also supports job control, and job control (and process groups) will likely be standard in future versions of UNIX.

A.4.2 Standard I/O

Processes can open files as they like, but most processes expect three file descriptors (numbers 0, 1, and 2) to be open when they start. These file descriptors are inherited across the [fork](#) (and possibly the [execve](#)) that created the process. They are known as *standard input* (0), *standard output* (1), and *standard error* (2). All three are frequently open to the user's terminal. Thus, the program can read what the user types by reading standard input, and the program can send output to the user's screen by writing to standard output. The standard-error file descriptor is also open for writing and is used for error output; standard output is used for ordinary output. Most programs can also accept a file (rather than a terminal) for standard input and standard output. The program does not care where its input is coming from and where its output is going. This is one of the elegant design features of UNIX.

The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process. Changing a standard file is called *I/O redirection*. The syntax for I/O redirection is shown in Figure A.5. In this example, the *ls* command produces a listing of the names of files in the current directory, the *pr* command formats that list into pages suitable for a printer, and the *lpr* command spools the formatted output to a printer, such as */dev/lp0*. The subsequent command forces all output and all error messages to be redirected to a file. Without the ampersand, error messages appear on the terminal.

A.4.3 Pipelines, Filters, and Shell Scripts

The first three commands of Figure A.5 could have been coalesced into the one command

```
% ls | pr | lpr
```

Each vertical bar tells the shell to arrange for the output of the preceding command to be passed as input to the following command. A *pipe* is used to carry the data from one process to the other. One process writes into one end of the pipe, and another process reads from the other end. In the example, the write end of one pipe would be set up by the shell to be the standard output of *ls*, and the read end of the pipe would be the standard input of *pr*; another pipe would be between *pr* and *lpr*.

A command such as *pr* that passes its standard input to its standard output, performing some processing on it, is called a *filter*. Many UNIX commands can be used as filters. Complicated functions can be pieced together as pipelines of common commands. Also, common functions, such as output formatting, do not need to be built into numerous commands, because the output of almost any program can be piped through *pr* (or some other appropriate filter).

Both of the common UNIX shells are also programming languages, with shell variables and the usual higher-level programming-language control constructs (loops, conditionals). The execution of a command is analogous to a subroutine call. A file of shell commands, a *shell script*, can be executed like any other command, with the appropriate shell being invoked automatically to read it. *Shell programming* thus can be used to combine ordinary programs conveniently for sophisticated applications without the need for any programming in conventional languages.

This external user view is commonly thought of as the definition of UNIX, yet it is the most easily changed definition. Writing a new shell with a quite different syntax and semantics would greatly change the user view while not changing the kernel or even the programmer interface. Several menu-driven and iconic interfaces for UNIX now exist, and the X Window System is rapidly becoming a standard. The heart of UNIX is, of course, the kernel. This kernel is much more difficult to change than is the user interface, because all programs depend on the system calls that it provides to remain consistent. Of course, new system calls can be added to increase functionality, but programs must then be modified to use the new calls.

A.5 Process Management

A major design problem for operating systems is the representation of processes. One substantial difference between UNIX and many other systems is the ease with which multiple processes can be created and manipulated. These processes are represented in UNIX by various control blocks. No system control blocks are accessible in the virtual address space of a user process; control blocks associated with a process are stored in the kernel. The kernel uses the information in these control blocks for process control and CPU scheduling.

A.5.1 Process Control Blocks

The most basic data structure associated with processes is the *process structure*. A process structure contains everything that the system needs to know about a process when the process is swapped out, such as its unique process identifier, scheduling information (for example, the priority of the process), and pointers

to other control blocks. There is an array of process structures whose length is defined at system-linking time. The process structures of ready processes are kept linked together by the scheduler in a doubly linked list (the ready queue), and there are pointers from each process structure to the process's parent, to its youngest living child, and to various other relatives of interest, such as a list of processes sharing the same program code (text).

The *virtual address space* of a user process is divided into text (program code), data, and stack segments. The data and stack segments are always in the same address space, but they may grow separately, and usually in opposite directions; most frequently, the stack grows down as the data grow up toward it. The text segment is sometimes (as on an Intel 8086 with separate instruction and data space) in an address space different from the data and stack, and it is usually read-only. The debugger puts a text segment in read–write mode to allow insertion of breakpoints.

Every process with sharable text (almost all, under FreeBSD) has a pointer from its process structure to a *text structure*. The text structure records how many processes are using the text segment, including a pointer into a list of their process structures, and where the page table for the text segment can be found on disk when it is swapped. The text structure itself is always resident in main memory; an array of such structures is allocated at system link time. The text, data, and stack segments for the processes may be swapped. When the segments are swapped in, they are paged.

The *page tables* record information on the mapping from the process's virtual memory to physical memory. The process structure contains pointers to the page table, for use when the process is resident in main memory, or the address of the process on the swap device, when the process is swapped. There is no special separate page table for a shared text segment; every process sharing the text segment has entries for its pages in the process's page table.

Information about the process needed only when the process is resident (that is, not swapped out) is kept in the *user structure* (or *u structure*), rather than in the process structure. The *u structure* is mapped read-only into user virtual address space, so user processes can read its contents. It is writable by the kernel. The *u structure* contains a copy of the process control block, or PCB, which is kept here for saving the process's general registers, stack pointer, program counter, and page-table base registers when the process is not running. There is space to keep system-call parameters and return values. All user and group identifiers associated with the process (not just the effective user identifier kept in the process structure) are kept here. Signals, timers, and quotas have data structures here. Of more obvious relevance to the ordinary user, the current directory and the table of open files are maintained in the user structure.

Every process has both a user and a system mode. Most ordinary work is done in *user mode*, but when a system call is made, it is performed in *system mode*. The system and user phases of a process never execute simultaneously. When a process is executing in system mode, a *kernel stack* for that process is used, rather than the user stack belonging to that process. The kernel stack for the process immediately follows the user structure: The kernel stack and the user structure together compose the *system data segment* for the process. The kernel has its own stack for use when it is not doing work on behalf of a process (for instance, for interrupt handling).

Figure A.6 illustrates how the process structure is used to find the various parts of a process.

The `fork` system call allocates a new process structure (with a new process identifier) for the child process and copies the user structure. There is ordinarily no need for a new text structure, as the processes share their text; the appropriate counters and lists are merely updated. A new page table is constructed, and new main memory is allocated for the data and stack segments of the child process. The copying of the user structure preserves open file descriptors, user and group identifiers, signal handling, and most similar properties of a process.

The `vfork` system call does *not* copy the data and stack to the new process; rather, the new process simply shares the page table of the old one. A new user structure and a new process structure are still created. A common use of this system call occurs when a shell executes a command and waits for its completion. The parent process uses `vfork` to produce the child process. Because the child process wishes to use an `execve` immediately to change its virtual address space completely, there is no need for a complete copy of the parent process. Such data structures as are necessary for manipulating pipes may be kept in registers between the `vfork` and the `execve`. Files may be closed in one process without affecting the other process, since the kernel data structures involved depend on the user structure, which is not shared. The parent is suspended when it calls `vfork` until the child either calls `execve` or terminates, so that the parent will not change memory that the child needs.

When the parent process is large, `vfork` can produce substantial savings in system CPU time. However, it is a fairly dangerous system call, since any memory change occurs in both processes until the `execve` occurs. An alternative is to share all pages by duplicating the page table but to mark the entries of both page tables as *copy-on-write*. The hardware protection bits are set to trap

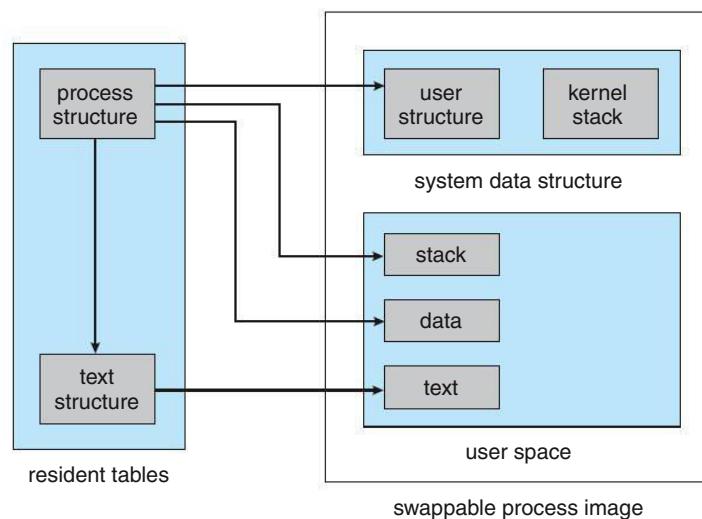


Figure A.6 Finding parts of a process using the process structure.

any attempt to write in these shared pages. If such a trap occurs, a new frame is allocated, and the shared page is copied to the new frame. The page tables are adjusted to show that this page is no longer shared (and therefore no longer needs to be write-protected), and execution can resume.

An `execve` system call creates no new process or user structure; rather, the text and data of the process are replaced. Open files are preserved (although there is a way to specify that certain file descriptors are to be closed on an `execve`). Most signal-handling properties are preserved, but arrangements to call a specific user routine on a signal are canceled, for obvious reasons. The process identifier and most other properties of the process are unchanged.

A.5.2 CPU Scheduling

CPU scheduling in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

Every process has a *scheduling priority* associated with it; larger numbers indicate lower priority. Processes doing disk I/O or other important tasks have priorities less than “pzero” and cannot be killed by signals. Ordinary user processes have positive priorities and thus are less likely to be run than is any system process, although user processes can set precedence over one another through the `nice` command.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa. This negative feedback in CPU scheduling makes it difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a 1-second quantum for the round-robin scheduling. FreeBSD reschedules processes every 0.1 second and recomputes priorities every second. The round-robin scheduling is accomplished by the *timeout* mechanism, which tells the clock interrupt driver to call a kernel subroutine after a specified interval; the subroutine to be called in this case causes the rescheduling and then resubmits a *timeout* to call itself again. The priority recomputation is also timed by a subroutine that resubmits a *timeout* for itself.

There is no preemption of one process by another in the kernel. A process may relinquish the CPU because it is waiting on I/O or because its time slice has expired. When a process chooses to relinquish the CPU, it goes to sleep on an *event*. The kernel primitive used for this purpose is called *sleep* (not to be confused with the user-level library routine of the same name). It takes an argument, which is by convention the address of a kernel data structure related to an *event* that the process wants to occur before that process is awakened. When the event occurs, the system process that knows about it calls *wakeup* with the address corresponding to the event, and *all* processes that had done a *sleep* on the same address are put in the ready queue to be run.

For example, a process waiting for disk I/O to complete will *sleep* on the address of the buffer header corresponding to the data being transferred. When the interrupt routine for the disk driver notes that the transfer is complete, it calls *wakeup* on the buffer header. The interrupt uses the kernel stack for whatever process happened to be running at the time, and the *wakeup* is done from that system process.

The process that actually does run is chosen by the scheduler. *Sleep* takes a second argument, which is the scheduling priority to be used for this purpose. This priority argument, if less than “pzero,” also prevents the process from being awakened prematurely by some exceptional event, such as a *signal*.

When a signal is generated, it is left pending until the system half of the affected process next runs. This event usually happens soon, since the signal normally causes the process to be awakened if the process has been waiting for some other condition.

No memory is associated with events. The caller of the routine that does a *sleep* on an event must be prepared to deal with a premature return, including the possibility that the reason for waiting has vanished.

Race conditions are involved in the event mechanism. If a process decides (because of checking a flag in memory, for instance) to sleep on an event, and the event occurs before the process can execute the primitive that does the actual sleep on the event, the process sleeping may then sleep forever. We prevent this situation by raising the hardware processor priority during the critical section so that no interrupts can occur, and thus only the process desiring the event can run until it is sleeping. Hardware processor priority is used in this manner to protect critical regions throughout the kernel and is the greatest obstacle to porting UNIX to multiple-processor machines. However, this problem has not prevented such porting from being done repeatedly.

Many processes, such as text editors, are I/O bound and usually will be scheduled mainly on the basis of waiting for I/O. Experience suggests that the UNIX scheduler performs best with I/O-bound jobs, as can be observed when several CPU-bound jobs, such as text formatters or language interpreters, are running.

What has been referred to here as *CPU scheduling* corresponds closely to the *short-term scheduling* of Chapter 3. However, the negative-feedback property of the priority scheme provides some long-term scheduling in that it largely determines the long-term *job mix*. Medium-term scheduling is done by the swapping mechanism described in Section A.6.

A.6 **Memory Management**

Much of UNIX’s early development was done on a PDP-11. The PDP-11 has only eight segments in its virtual address space, and the size of each is at most 8,192 bytes. The larger machines, such as the PDP-11/70, allow separate instruction and address spaces, effectively doubling the address space and number of segments, but this address space is still relatively small. In addition, the kernel was even more severely constrained due to dedication of one data segment to interrupt vectors, another to point at the per-process system data segment, and yet another for the UNIBUS (system I/O bus) registers. Further, on the smaller PDP-11s, total physical memory was limited to 256 KB. The total memory resources were insufficient to justify or support complex memory-management algorithms. Thus, UNIX swapped entire process memory images.

Berkeley introduced paging to UNIX with 3 BSD. VAX 4.2 BSD is a demand-paged virtual-memory system. Paging eliminates external fragmentation of memory. (Internal fragmentation still occurs, but it is negligible with a reasonably small page size.) Because paging allows execution with only parts of

each process in memory, more jobs can be kept in main memory, and swapping can be kept to a minimum. *Demand paging* is done in a straightforward manner. When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and the proper disk page is read into the frame.

There are a few optimizations. If the page needed is still in the page table for the process but has been marked invalid by the page-replacement process, it can be marked valid and used without any I/O transfer. Pages can similarly be retrieved from the list of free frames. When most processes are started, many of their pages are prepaged and are put on the free list for recovery by this mechanism. Arrangements can also be made for a process to have no prepaging on startup; but that is seldom done, as it results in more page-fault overhead, being closer to pure demand paging. FreeBSD implements page coloring with paging queues. The queues are arranged according to the size of the processor's L1 and L2 caches; and when a new page needs to be allocated, FreeBSD tries to get one that is optimally aligned for the cache.

If the page has to be fetched from disk, it must be locked in memory for the duration of the transfer. This locking ensures that the page will not be selected for page replacement. Once the page is fetched and mapped properly, it must remain locked if raw physical I/O is being done on it.

The *page-replacement* algorithm is more interesting. 4.2 BSD uses a modification of the *second-chance* (clock) algorithm described in Section 9.4.5. The map of all nonkernel main memory (the *core map* or *cmap*) is swept linearly and repeatedly by a software *clock hand*. When the clock hand reaches a given frame, if the frame is marked as being in use by some software condition (for example, if physical I/O is in progress using it) or if the frame is already free, the frame is left untouched, and the clock hand sweeps to the next frame. Otherwise, the corresponding text or process page-table entry for this frame is located. If the entry is already invalid, the frame is added to the free list; otherwise, the page-table entry is made invalid but reclaimable (that is, if it has not been paged out by the next time it is wanted, it can just be made valid again).

BSD Tahoe added support for systems that implement the reference bit. On such systems, one pass of the clock hand turns the reference bit off, and a second pass places those pages whose reference bits remain off onto the free list for replacement. Of course, if the page is dirty, it must first be written to disk before being added to the free list. Pageouts are done in clusters to improve performance.

There are checks to make sure that the number of valid data pages for a process does not fall too low and to keep the paging device from being flooded with requests. There is also a mechanism by which a process can limit the amount of main memory it uses.

The LRU clock-hand scheme is implemented in the *pagedaemon*, which is process 2. (Remember that the *swapper* is process 0, and *init* is process 1.) This process spends most of its time sleeping, but a check is done several times per second (scheduled by a *timeout*) to see if action is necessary; if it is, process 2 is awakened. Whenever the number of free frames falls below a threshold, *lotsfree*, the *pagedaemon* is awakened; thus, if there is always a large amount of free memory, the *pagedaemon* imposes no load on the system, because it never runs.

The sweep of the clock hand each time the *pagedaemon* process is awakened (that is, the number of frames scanned, which is usually more than the number paged out) is determined both by the number of frames lacking to reach *lotsfree* and by the number of frames that the *scheduler* has determined are needed for various reasons (the more frames needed, the longer the sweep). If the number of frames free rises to *lotsfree* before the expected sweep is completed, the hand stops, and the *pagedaemon* process sleeps. The parameters that determine the range of the clock-hand sweep are determined at system startup according to the amount of main memory, such that *pagedaemon* does not use more than 10 percent of all CPU time.

If the *scheduler* decides that the paging system is overloaded, processes will be swapped out whole until the overload is relieved. This swapping usually happens only if several conditions are met: Load average is high; free memory has fallen below a low limit, *minfree*; and the average memory available over recent time is less than a desirable amount, *desfree*, where $\text{lotsfree} > \text{desfree} > \text{minfree}$. In other words, only a chronic shortage of memory with several processes trying to run will cause swapping, and even then free memory has to be extremely low at the moment. (An excessive paging rate or a need for memory by the kernel itself may also enter into the calculations, in rare cases.) Processes may be swapped by the *scheduler*, of course, for other reasons (such as simply because they have not run for a long time).

The parameter *lotsfree* is usually one-quarter of the memory in the map that the clock hand sweeps, and *desfree* and *minfree* are usually the same across different systems but are limited to fractions of available memory. FreeBSD dynamically adjusts its paging queues so these virtual memory parameters will rarely need to be adjusted. *Minfree* pages must be kept free in order to supply any pages that might be needed at interrupt time.

Every process's text segment is by default shared and read-only. This scheme is practical with paging, because there is no external fragmentation, and the swap space gained by sharing more than offsets the negligible amount of overhead involved, as the kernel virtual space is large.

CPU scheduling, memory swapping, and paging interact: The lower the priority of a process, the more likely that its pages will be paged out and the more likely that it will be swapped in its entirety. The age preferences in choosing processes to swap guard against thrashing, but paging does so more effectively. Ideally, processes will not be swapped out unless they are idle, because each process will need only a small working set of pages in main memory at any one time, and the *pagedaemon* will reclaim unused pages for use by other processes.

The amount of memory the process will need is some fraction of that process's total virtual size—up to one-half if that process has been swapped out for a long time.

A.7 **File System**

The UNIX file system supports two main objects: files and directories. Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

A.7.1 Blocks and Fragments

Most of the file system is taken up by *data blocks*, which contain whatever the users have put in their files. Let us consider how these data blocks are stored on the disk.

The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for speed. However, because UNIX file systems usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1BSD file system was limited to a 1,024-byte (1-KB) block. The 4.2 BSD solution is to use *two* block sizes for files that have no indirect blocks. All the blocks of a file are of a large *block* size (such as 8 KB), except the last. The last block is an appropriate multiple of a smaller *fragment* size (for example, 1,024 KB) to fill out the file. Thus, a file of size 18,000 bytes would have two 8-KB blocks and one 2-KB fragment (which would not be filled completely).

The *block* and *fragment* sizes are set during file-system creation according to the intended use of the file system: If many small files are expected, the fragment size should be small; if repeated transfers of large files are expected, the basic block size should be large. Implementation details force a maximum block-to-fragment ratio of 8:1 and a minimum block size of 4 KB, so typical choices are 4,096:512 for the former case and 8,192:1,024 for the latter.

Suppose data are written to a file in transfer sizes of 1-KB bytes, and the block and fragment sizes of the file system are 4 KB and 512 bytes. The file system will allocate a 1-KB fragment to contain the data from the first transfer. The next transfer will cause a new 2-KB fragment to be allocated. The data from the original fragment must be copied into this new fragment, followed by the second 1-KB transfer. The allocation routines do attempt to find the required space on the disk immediately following the existing fragment so that no copying is necessary, but if they cannot do so, up to seven copies may be required before the fragment becomes a block. Provisions have been made for programs to discover the block size for a file so that transfers of that size can be made, to avoid fragment recopying.

A.7.2 Inodes

A file is represented by an *inode* (Figure 11.9). An inode is a record that stores most of the information about a specific file on the disk. The name *inode* (pronounced EYE node) is derived from “index node” and was originally spelled “i-node”; the hyphen fell out of use over the years. The term is sometimes spelled “I node.”

The inode contains the user and group identifiers of the file, the times of the last file modification and access, a count of the number of hard links (directory entries) to the file, and the type of the file (plain file, directory, symbolic link, character device, block device, or socket). In addition, the inode contains 15 pointers to the disk blocks containing the data contents of the file. The first 12 of these pointers point to *direct blocks*; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (no more than 12 blocks) can be referenced immediately, because a copy of the inode is kept in main memory while a file is open. If the block size is 4 KB, then up to 48 KB of data can be accessed directly from the inode.

The next three pointers in the inode point to *indirect blocks*. If the file is large enough to use indirect blocks, each of the indirect blocks is of the major block size; the fragment size applies only to data blocks. The first indirect block pointer is the address of a *single indirect block*. The single indirect block is an index block containing not data but the addresses of blocks that do contain data. Then, there is a *double-indirect-block pointer*, the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer would contain the address of a *triple indirect block*; however, there is no need for it.

The minimum block size for a file system in 4.2 BSD is 4 KB, so files with as many as 2^{32} bytes will use only double, not triple, indirection. That is, as each block pointer takes 4 bytes, we have 49,152 bytes accessible in direct blocks, 4,194,304 bytes accessible by a single indirection, and 4,294,967,296 bytes reachable through double indirection, for a total of 4,299,210,752 bytes, which is larger than 2^{32} bytes. The number 2^{32} is significant because the file offset in the file structure in main memory is kept in a 32-bit word. Files therefore cannot be larger than 2^{32} bytes. Since file pointers are signed integers (for seeking backward and forward in a file), the actual maximum file size is 2^{32-1} bytes. Two gigabytes is large enough for most purposes.

A.7.3 Directories

Plain files are not distinguished from directories at this level of implementation; directory contents are kept in data blocks, and directories are represented by an inode in the same way as plain files. Only the inode type field distinguishes between plain files and directories. Plain files are not assumed to have a structure, whereas directories have a specific structure. In Version 7, file names were limited to 14 characters, so directories were a list of 16-byte entries: 2 bytes for an inode number and 14 bytes for a file name.

In FreeBSD file names are of variable length, up to 255 bytes, so directory entries are also of variable length. Each entry contains first the length of the entry, then the file name and the inode number. This variable-length entry makes the directory management and search routines more complex, but it allows users to choose much more meaningful names for their files and directories. The first two names in every directory are “.” and “..”. New directory entries are added to the directory in the first space available, generally after the existing files. A linear search is used.

The user refers to a file by a path name, whereas the file system uses the inode as its definition of a file. Thus, the kernel has to map the supplied user path name to an inode. The directories are used for this mapping.

First, a starting directory is determined. If the first character of the path name is “/”, the starting directory is the root directory. If the path name starts with any character other than a slash, the starting directory is the current directory of the current process. The starting directory is checked for proper file type and access permissions, and an error is returned if necessary. The inode of the starting directory is always available.

The next element of the path name, up to the next “/” or to the end of the path name, is a file name. The starting directory is searched for this name, and an error is returned if the name is not found. If the path name has yet another element, the current inode must refer to a directory; an error is returned if it

does not or if access is denied. This directory is searched as was the previous one. This process continues until the end of the path name is reached and the desired inode is returned. This step-by-step process is needed because at any directory a mount point (or symbolic link, as discussed below) may be encountered, causing the translation to move to a different directory structure for continuation.

Hard links are simply directory entries like any other. We handle symbolic links for the most part by starting the search over with the path name taken from the contents of the symbolic link. We prevent infinite loops by counting the number of symbolic links encountered during a path-name search and returning an error when a limit (eight) is exceeded.

Nondisk files (such as devices) do not have data blocks allocated on the disk. The kernel notices these file types (as indicated in the inode) and calls appropriate drivers to handle I/O for them.

Once the inode is found by, for instance, the `open` system call, a *file structure* is allocated to point to the inode. The file descriptor given to the user refers to this file structure. FreeBSD has a *directory name cache* to hold recent directory-to-inode translations, which greatly increases file-system performance.

A.7.4 Mapping a File Descriptor to an Inode

System calls that refer to open files indicate the file by passing a file descriptor as an argument. The file descriptor is used by the kernel to index a table of open files for the current process. Each entry of the table contains a pointer to a file structure. This file structure in turn points to the inode; see Figure A.7. The open file table has a fixed length, which is only settable at boot time. Therefore, there is a fixed limit on the number of concurrently open files in a system.

The `read` and `write` system calls do not take a position in the file as an argument. Rather, the kernel keeps a *file offset*, which is updated by an appropriate amount after each `read` or `write` according to the number of data actually transferred. The offset can be set directly by the `Iseek` system call. If the file descriptor indexed an array of inode pointers instead of file pointers, this offset would have to be kept in the inode. Because more than one process may open the same file, and each such process needs its own offset for the file,

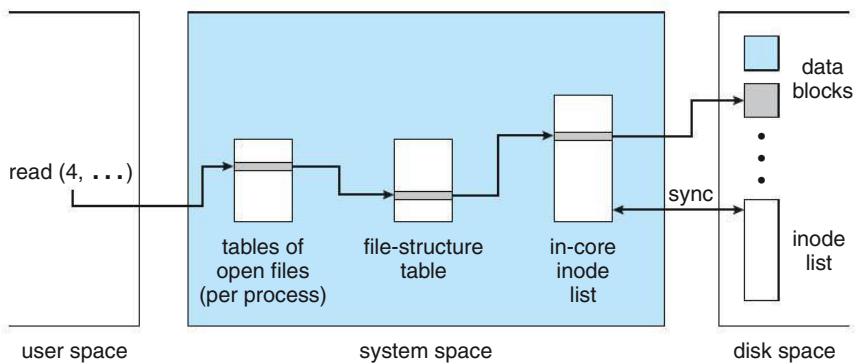


Figure A.7 File-system control blocks.

Appendix A BSD UNIX

keeping the offset in the inode is inappropriate. Thus, the file structure is used to contain the offset. File structures are inherited by the child process after a [fork](#), so several processes may share the *same* offset location for a file.

The *inode structure* pointed to by the file structure is an in-core copy of the inode on the disk. The in-core inode has a few extra fields, such as a reference count of how many file structures are pointing at it, and the file structure has a similar reference count for how many file descriptors refer to it. When a count becomes zero, the entry is no longer needed and may be reclaimed and reused.

A.7.5 Disk Structures

The file system that the user sees is supported by data on a mass storage device—usually, a disk. The user ordinarily knows of only one file system, but this one logical file system may actually consist of several *physical* file systems, each on a different device. Because device characteristics differ, each separate hardware device defines its own physical file system. In fact, we generally want to partition large physical devices, such as disks, into multiple *logical* devices. Each logical device defines a physical file system. Figure A.8 illustrates how a directory structure is partitioned into file systems, which are mapped onto logical devices, which are partitions of physical devices. The sizes and locations

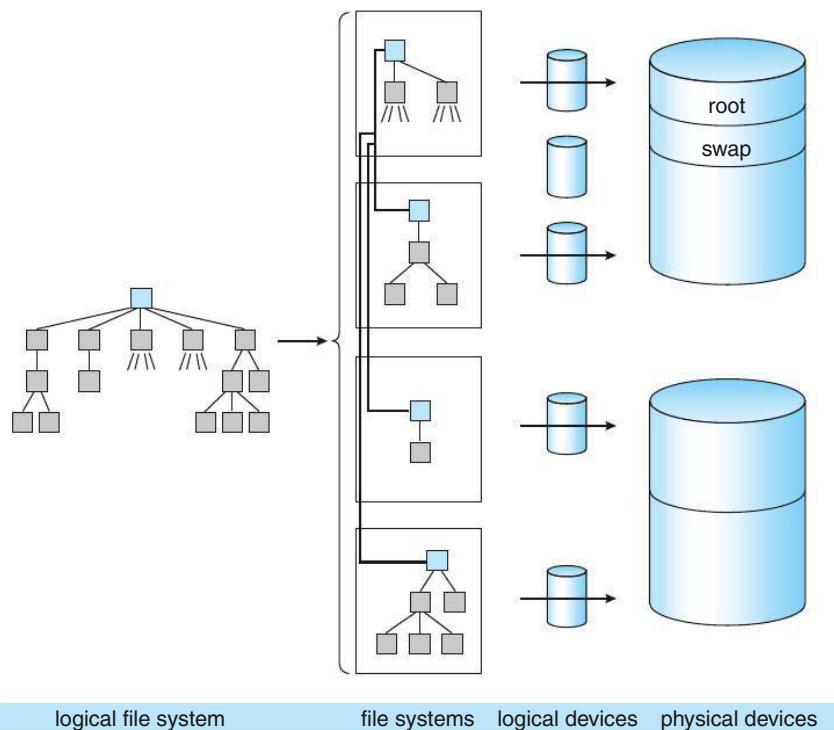


Figure A.8 Mapping of a logical file system to physical devices.

of these partitions were coded into device drivers in earlier systems, but they are maintained on the disk by FreeBSD.

Partitioning a physical device into multiple file systems has several benefits. Different file systems can support different uses. Although most partitions will be used by the file system, at least one will be needed as a swap area for the virtual-memory software. Reliability is improved, because software damage is generally limited to only one file system. We can improve efficiency by varying the file-system parameters (such as the block and fragment sizes) for each partition. Also, separate file systems prevent one program from using all available space for a large file, because files cannot be split across file systems. Finally, disk backups are done per partition, and it is faster to search a backup tape for a file if the partition is smaller. Restoring the full partition from tape is also faster.

The number of file systems on a drive varies according to the size of the disk and the purpose of the computer system as a whole. One file system, the *root file system*, is always available. Other file systems may be *mounted*—that is, integrated into the directory hierarchy of the root file system.

A bit in the inode structure indicates that the inode has a file system mounted on it. A reference to this file causes the *mount table* to be searched to find the device number of the mounted device. The device number is used to find the inode of the root directory of the mounted file system, and that inode is used. Conversely, if a path-name element is “..” and the directory being searched is the root directory of a file system that is mounted, the mount table is searched to find the inode it is mounted on, and that inode is used.

Each file system is a separate system resource and represents a set of files. The first sector on the logical device is the *boot block*, possibly containing a primary bootstrap program, which may be used to call a secondary bootstrap program residing in the next 7.5 KB. A system needs only one partition containing boot-block data, but the systems manager may install duplicates via privileged programs, to allow booting when the primary copy is damaged. The *superblock* contains static parameters of the file system. These parameters include the total size of the file system, the block and fragment sizes of the data blocks, and assorted parameters that affect allocation policies.

A.7.6 Implementations

The user interface to the file system is simple and well defined, allowing the implementation of the file system itself to be changed without significant effect on the user. The file system was changed between Version 6 and Version 7, and again between Version 7 and 4BSD. For Version 7, the size of inodes doubled, the maximum file and file-system sizes increased, and the details of free-list handling and superblock information changed. At that time also, *seek* (with a 16-bit offset) became *lseek* (with a 32-bit offset), to allow specification of offsets in larger files; but few other changes were visible outside the kernel.

In 4.0 BSD, the size of blocks used in the file system was increased from 512 bytes to 1,024 bytes. Although this increased size produced increased internal fragmentation on the disk, it doubled throughput, due mainly to the greater number of data accessed on each disk transfer. This idea was later adopted by System V, along with a number of other ideas, device drivers, and programs.

4.2 BSD added the Berkeley Fast File System, which increased speed and was accompanied by new features. Symbolic links required new system calls. Long file names necessitated new directory system calls to traverse the now-complex internal directory structure. Finally, `truncate` calls were added. The Fast File System was a success and is now found in most implementations of UNIX. Its performance is made possible by its layout and allocation policies, which we discuss next. In Section 11.4.4, we discussed changes made in SunOS to increase disk throughput further.

A.7.7 Layout and Allocation Policies

The kernel uses a *<logical device number, inode number>* pair to identify a file. The logical device number defines the file system involved. The inodes in the file system are numbered in sequence. In the Version 7 file system, all inodes are in an array immediately following a single superblock at the beginning of the logical device, with the data blocks following the inodes. The *inode number* is effectively just an index into this array.

With the Version 7 file system, a block of a file can be anywhere on the disk between the end of the inode array and the end of the file system. Free blocks are kept in a linked list in the superblock. Blocks are pushed onto the front of the free list and are removed from the front as needed to serve new files or to extend existing files. Thus, the blocks of a file may be arbitrarily far from both the inode and one another. Furthermore, the more a file system of this kind is used, the more disorganized the blocks in a file become. We can reverse this process only by reinitializing and restoring the entire file system, which is not a convenient task to perform. This process was described in Section 11.7.4.

Another difficulty is that the reliability of the file system is suspect. For speed, the superblock of each mounted file system is kept in memory. Keeping the superblock in memory allows the kernel to access a superblock quickly, especially for using the free list. Every 30 seconds, the superblock is written to the disk, to keep the in-core and disk copies synchronized (by the update program, using the `sync` system call). However, system bugs or hardware failures may cause a system crash, which destroys the in-core superblock between updates to the disk. Then, the free list on disk does not reflect accurately the state of the disk; to reconstruct it, we must perform a lengthy examination of all blocks in the file system. (This problem remains in the new file system.)

The 4.2 BSD file-system implementation is radically different from that of Version 7. This reimplementation was done primarily to improve efficiency and robustness, and most such changes are invisible outside the kernel. Other changes introduced at the same time are visible at both the system-call and the user levels; examples include symbolic links and long file names (up to 255 characters). Most of the changes required for these features were not in the kernel, however, but in the programs that use them.

Space allocation is especially different. The major new concept in FreeBSD is the *cylinder group*. The cylinder group was introduced to allow localization of the blocks in a file. Each cylinder group occupies one or more consecutive cylinders of the disk, so that disk accesses within the cylinder group require minimal disk head movement. Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks (Figure A.9).



Figure A.9 4.3 BSD cylinder group.

The superblocks in all cylinder groups are identical, so that a superblock can be recovered from any one of them in the event of disk corruption. The *cylinder block* contains dynamic parameters of the particular cylinder group. These include a bit map of free data blocks and fragments and a bit map of free inodes. Statistics on recent progress of the allocation strategies are also kept here.

The header information in a cylinder group (the superblock, the cylinder block, and the inodes) is not always at the beginning of the group. If it were, the header information for every cylinder group might be on the same disk platter and a single disk head crash could wipe out all of them. Therefore, each cylinder group has its header information at a different offset from the beginning of the group.

The directory-listing command *ls* commonly reads all the inodes of every file in a directory, making it desirable for all such inodes to be close together on the disk. For this reason, the inode for a file is usually allocated from the cylinder group containing the inode of the file's parent directory. Not everything can be localized, however, so an inode for a new directory is put in a *different* cylinder group from that of its parent directory. The cylinder group chosen for such a new directory inode is that with the greatest number of unused inodes.

To reduce disk head seeks involved in accessing the data blocks of a file, we allocate blocks from the same cylinder group as often as possible. Because a single file cannot be allowed to take up all the blocks in a cylinder group, a file exceeding a certain size (such as 2 MB) has further block allocation redirected to a different cylinder group; the new group is chosen from among those having more than average free space. If the file continues to grow, allocation is again redirected (at each megabyte) to yet another cylinder group. Thus, all the blocks of a small file are likely to be in the same cylinder group, and the number of long head seeks involved in accessing a large file is kept small.

There are two levels of disk-block-allocation routines. The global policy routines select a desired disk block according to the considerations already discussed. The local policy routines use the specific information recorded in the cylinder blocks to choose a block near the one requested. If the requested block is not in use, it is returned. Otherwise, the routine returns either the block rotationally closest to the one requested in the same cylinder or a block in a different cylinder but in the same cylinder group. If no more blocks are in the cylinder group, a quadratic rehash is done among all the other cylinder groups to find a block; if that fails, an exhaustive search is done. If enough free

space (typically 10 percent) is left in the file system, blocks are usually found where desired, the quadratic rehash and exhaustive search are not used, and performance of the file system does not degrade with use.

Because of the increased efficiency of the Fast File System, typical disks are now utilized at 30 percent of their raw transfer capacity. This percentage is a marked improvement over that realized with the Version 7 file system, which used about 3 percent of the bandwidth.

BSD Tahoe introduced the Fat Fast File System, which allows the number of inodes per cylinder group, the number of cylinders per cylinder group, and the number of distinguished rotational positions to be set when the file system is created. FreeBSD used to set these parameters according to the disk hardware type.

A.8 I/O System

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, the file system presents a simple, consistent storage facility (the file) independent of the underlying disk hardware. In UNIX, the peculiarities of I/O devices are also hidden from the bulk of the kernel itself by the *I/O system*. The I/O system consists of a buffer caching system, general device-driver code, and drivers for specific hardware devices. Only the device driver knows the peculiarities of a specific device. The major parts of the I/O system are diagrammed in Figure A.10.

There are three main kinds of I/O in FreeBSD: block devices, character devices, and the *socket* interface. The socket interface, together with its protocols and network interfaces, will be described in Section A.9.1.

Block devices include disks and tapes. Their distinguishing characteristic is that they are directly addressable in a fixed block size—usually, 512 bytes. A block-device driver is required to isolate details of tracks, cylinders, and so on from the rest of the kernel. Block devices are accessible directly through appropriate device special files (such as */dev/rp0*), but they are more commonly accessed indirectly through the file system. In either case, transfers are buffered through the *block buffer cache*, which has a profound effect on efficiency.

Character devices include terminals and line printers but also include almost everything else (except network interfaces) that does not use the block buffer

system-call interface to the kernel					
socket	plain file	cooked block interface	raw block interface	raw tty interface	cooked TTY
protocols	file system				line discipline
network interface	block-device driver			character-device driver	
the hardware					

Figure A.10 4.3 BSD kernel I/O structure.

cache. For instance, `/dev/mem` is an interface to physical main memory, and `/dev/null` is a bottomless sink for data and an endless source of end-of-file markers. Some devices, such as high-speed graphics interfaces, may have their own buffers or may always do I/O directly into the user's data space; because they do not use the block buffer cache, they are classed as character devices. Terminals and terminal-like devices use *C-lists*, which are buffers smaller than those of the block buffer cache.

Block devices and *character* devices are the two main device classes. Device drivers are accessed by one of two arrays of entry points. One array is for block devices; the other is for character devices. A device is distinguished by a class (block or character) and a *device number*. The device number consists of two parts. The *major device number* is used to index the array for character or block devices to find entries into the appropriate device driver. The *minor device number* is interpreted by the device driver as, for example, a logical disk partition or a terminal line.

A device driver is connected to the rest of the kernel only by the entry points recorded in the array for its class and by its use of common buffering systems. This segregation is important for portability and for system configuration.

A.8.1 Block Buffer Cache

The block devices, as mentioned, use a block buffer cache. The buffer cache consists of a number of buffer headers, each of which can point to a piece of physical memory, as well as to a device number and a block number on the device. The buffer headers for blocks not currently in use are kept in several linked lists, one each for

- Buffers recently used, linked in LRU order (the LRU list)
- Buffers not recently used or without valid contents (the AGE list)
- EMPTY buffers with no physical memory associated with them

The buffers in these lists are also hashed by device and block number for search efficiency.

When a block is wanted from a device (a read), the cache is searched. If the block is found, it is used, and no I/O transfer is necessary. If it is not found, a buffer is chosen from the AGE list or, if that list is empty, the LRU list. Then the device number and block number associated with it are updated, memory is found for it if necessary, and the new data are transferred into it from the device. If there are no empty buffers, the LRU buffer is written to its device (if it is modified), and the buffer is reused.

On a write, if the block in question is already in the buffer cache, the new data are put in the buffer (overwriting any previous data), the buffer header is marked to indicate that the buffer has been modified, and no I/O is immediately necessary. The data will be written when the buffer is needed for other data. If the block is not found in the buffer cache, an empty buffer is chosen (as with a read), and a transfer is done to this buffer. Writes are periodically forced for dirty buffer blocks to minimize potential file-system inconsistencies after a crash.

The number of data in a buffer in FreeBSD is variable, up to a maximum over all file systems, usually 8 KB. The minimum size is the paging-cluster size, usually 1,024 bytes. Buffers are page-cluster aligned, and any page cluster may be mapped into only one buffer at a time, just as any disk block may be mapped into only one buffer at a time. The EMPTY list holds buffer headers, which are used if a physical memory block of 8 KB is split to hold multiple, smaller blocks. Headers are needed for these blocks and are retrieved from EMPTY.

The number of data in a buffer may grow as a user process writes more data following those already in the buffer. When this increase in the data occurs, a new buffer large enough to hold all the data is allocated, and the original data are copied into it, followed by the new data. If a buffer shrinks, a buffer is taken off the empty queue, excess pages are put in it, and that buffer is released to be written to disk.

Some devices, such as magnetic tapes, require blocks to be written in a certain order. Facilities are therefore provided to force a synchronous write of buffers to these devices in the correct order. Directory blocks are also written synchronously, to forestall crash inconsistencies. Consider the chaos that could occur if many changes were made to a directory but the directory entries themselves were not updated.

The size of the buffer cache can have a profound effect on the performance of a system, because, if it is large enough, the percentage of cache hits can be high and the number of actual I/O transfers low. FreeBSD optimizes the buffer cache by continually adjusting the amount of memory used by programs and the disk cache.

Some interesting interactions occur among the buffer cache, the file system, and the disk drivers. When data are written to a disk file, they are buffered in the cache, and the disk driver sorts its output queue according to disk address. These two actions allow the disk driver to minimize disk head seeks and to write data at times optimized for disk rotation. Unless synchronous writes are required, a process writing to disk simply writes into the buffer cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much nearer to asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for large transfers, counter to intuition.

A.8.2 Raw Device Interfaces

Almost every block device also has a character interface, and these are called *raw device interfaces*. Such an interface differs from the *block interface* in that the block buffer cache is bypassed.

Each disk driver maintains a queue of pending transfers. Each record in the queue specifies whether it is a read or a write and gives a main memory address for the transfer (usually in 512-byte increments), a device address for the transfer (usually the address of a disk sector), and a transfer size (in sectors). It is simple to map the information from a block buffer to what is required for this queue.

It is almost as simple to map a piece of main memory corresponding to part of a user process's virtual address space. This mapping is what a raw disk interface, for instance, does. Unbuffered transfers directly to or from a user's

virtual address space are thus allowed. The size of the transfer is limited by the physical devices, some of which require an even number of bytes.

The kernel accomplishes transfers for swapping and paging simply by putting the appropriate request on the queue for the appropriate device. No special swapping or paging device driver is needed.

The 4.2 BSD file-system implementation was actually written and largely tested as a user process that used a raw disk interface, before the code was moved into the kernel. In an interesting about-face, the Mach operating system has no file system per se. File systems can be implemented as user-level tasks (see Appendix B).

A.8.3 C-Lists

As mentioned, terminals and terminal-like devices use a character-buffering system that keeps small blocks of characters (usually 28 bytes) in linked lists called C-lists. There are routines to enqueue and dequeue characters for such lists. Although all free character buffers are kept in a single free list, most device drivers that use them limit the number of characters that may be queued at one time for any given terminal line.

A `write` system call to a terminal enqueues characters on a list for the device. An initial transfer is started, and interrupts cause dequeuing of characters and further transfers.

Input is similarly interrupt driven. Terminal drivers typically support *two* input queues, however, and conversion from the first (raw queue) to the other (canonical queue) is triggered when the interrupt routine puts an end-of-line character on the raw queue. The process doing a read on the device is then awakened, and its system phase does the conversion; the characters thus put on the canonical queue are then available to be returned to the user process by the read.

The device driver can bypass the canonical queue and return characters directly from the raw queue. This mode of operation is known as *raw mode*. Full-screen editors, as well as other programs that need to react to every keystroke, use this mode.

A.9 Interprocess Communication

Although many tasks can be accomplished in isolated processes, many others require interprocess communication. Isolated computing systems have long served for many applications, but networking is increasingly important. Furthermore, with the increasing use of personal workstations, resource sharing is becoming more common. Interprocess communication has not traditionally been one of UNIX's strong points.

A.9.1 Sockets

The *pipe* (discussed in Section A.4.3) is the IPC mechanism most characteristic of UNIX. A pipe permits a reliable unidirectional byte stream between two processes. It is traditionally implemented as an ordinary file, with a few exceptions. It has no name in the file system, being created instead by the `pipe` system call. Its size is fixed, and when a process attempts to write to a

Appendix A BSD UNIX

full pipe, the process is suspended. Once all data previously written into the pipe have been read out, writing continues at the beginning of the file (pipes are not true circular buffers). One benefit of the small size (usually 4,096 bytes) of pipes is that pipe data are seldom actually written to disk; they usually are kept in memory by the normal block buffer cache.

In FreeBSD pipes are implemented as a special case of the *socket* mechanism. The socket mechanism provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities. Even on the same machine, a pipe can be used only by two processes related through use of the **fork** system call. The socket mechanism can be used by unrelated processes.

A socket is an endpoint of communication. A socket in use usually has an *address* bound to it. The nature of the address depends on the *communication domain* of the socket. A characteristic property of a domain is that processes communicating in the same domain use the same *address format*. A single socket can communicate in only one domain.

The three domains currently implemented in FreeBSD are the UNIX domain (AF_UNIX), the Internet domain (AF_INET), and the XEROX Network Services (NS) domain (AF_NS). The address format of the UNIX domain is that of an ordinary file-system path name, such as */alpha/beta/gamma*. Processes communicating in the Internet domain use DARPA Internet communications protocols (such as TCP/IP) and Internet addresses, which consist of a 32-bit host number and a 32-bit port number (representing a rendezvous point on the host).

There are several **socket types**, which represent classes of services. Each type may or may not be implemented in any communication domain. If a type is implemented in a given domain, it may be implemented by one or more protocols, which may be selected by the user:

- **Stream sockets:** These sockets provide reliable, duplex, sequenced data streams. No data are lost or duplicated in delivery, and there are no record boundaries. This type is supported in the Internet domain by TCP. In the UNIX domain, pipes are implemented as a pair of communicating stream sockets.
- **Sequenced packet sockets:** These sockets provide data streams like those of stream sockets, except that record boundaries are provided. This type is used in the XEROX AF_NS protocol.
- **Datagram sockets:** These sockets transfer messages of variable size in either direction. There is no guarantee that such messages will arrive in the same order they were sent, or that they will be unduplicated, or that they will arrive at all, but the original message (or record) size is preserved in any datagram that does arrive. This type is supported in the Internet domain by UDP.
- **Reliably delivered message sockets:** These sockets transfer messages that are guaranteed to arrive and that otherwise are like the messages transferred using datagram sockets. This type is currently unsupported.
- **Raw sockets:** These sockets allow direct access by processes to the protocols that support the other socket types. The protocols accessible include not only the uppermost ones but also lower-level protocols. For

example, in the Internet domain, it is possible to reach TCP, IP beneath that, or an Ethernet protocol beneath that. This capability is useful for developing new protocols.

A set of system calls is specific to the socket facility. The `socket` system call creates a socket. It takes as arguments specifications of the communication domain, the socket type, and the protocol to be used to support that type. The value returned by the call is a small integer called a **socket descriptor**, which occupies the same name space as file descriptors. The socket descriptor indexes the array of open *files* in the *u* structure in the kernel and has a file structure allocated for it. The FreeBSD file structure may point to a socket structure instead of to an inode. In this case, certain socket information (such as the socket's type, its message count, and the data in its input and output queues) is kept directly in the socket structure.

For another process to address a socket, the socket must have a name. A name is bound to a socket by the `bind` system call, which takes the socket descriptor, a pointer to the name, and the length of the name as a byte string. The contents and length of the byte string depend on the address format. The `connect` system call is used to initiate a connection. The arguments are syntactically the same as those for `bind`; the socket descriptor represents the local socket, and the address is that of the foreign socket to which the attempt to connect is made.

Many processes that communicate using the socket IPC follow the client–server model. In this model, the *server* process provides a *service* to the *client* process. When the service is available, the server process listens on a well-known address, and the client process uses `connect` to reach the server.

A server process uses `socket` to create a socket and `bind` to bind the well-known address of its service to that socket. Then, it uses the `listen` system call to tell the kernel that it is ready to accept connections from clients and to specify how many pending connections the kernel should queue until the server can service them. Finally, the server uses the `accept` system call to accept individual connections. Both `listen` and `accept` take as an argument the socket descriptor of the original socket. The system call `accept` returns a new socket descriptor corresponding to the new connection; the original socket descriptor is still open for further connections. The server usually uses `fork` to produce a new process after the `accept` to service the client while the original server process continues to listen for more connections. There are also system calls for setting parameters of a connection and for returning the address of the foreign socket after an `accept`.

When a connection for a socket type such as a stream socket is established, the addresses of both endpoints are known, and no further addressing information is needed to transfer data. The ordinary `read` and `write` system calls may then be used to transfer data.

The simplest way to terminate a connection, and to destroy the associated socket, is to use the `close` system call on its socket descriptor. We may also wish to terminate only one direction of communication of a duplex connection; the `shutdown` system call can be used for this purpose.

Some socket types, such as datagram sockets, do not support connections; instead, their sockets exchange datagrams that must be addressed individually. The system calls `sendto` and `recvfrom` are used for such connections. Both take

as arguments a socket descriptor, a buffer pointer and length, and an address-buffer pointer and length. The address buffer contains the appropriate address for `sendto` and is filled in with the address of the datagram just received by `recvfrom`. The number of data actually transferred is returned by both system calls.

The `select` system call can be used to multiplex data transfers on several file descriptors and/or socket descriptors. It can even be used to allow one server process to listen for client connections for many services and to `fork` a process for each connection as the connection is made. The server does a `socket`, `bind`, and `listen` for each service and then does a `select` on all the socket descriptors. When `select` indicates activity on a descriptor, the server does an `accept` on it and forks a process on the new descriptor returned by `accept`, leaving the parent process to do a `select` again.

A.9.2 Network Support

Almost all current UNIX systems support the UUCP network facilities, which are mostly used over dial-up telephone lines to support the UUCP mail network and the USENET news network. These are, however, rudimentary networking facilities; they do not support even remote login, much less remote procedure calls or distributed file systems. These facilities are almost completely implemented as user processes and are not part of the operating system itself.

FreeBSD supports the DARPA Internet protocols UDP, TCP, IP, and ICMP on a wide range of Ethernet, token-ring, and ARPANET interfaces. The framework in the kernel to support these protocols is intended to facilitate the implementation of further protocols, and all protocols are accessible via the socket interface. Rob Gurwitz of BBN wrote the first version of the code as an add-on package for 4.1BSD.

The International Standards Organization's (ISO) Open System Interconnection (OSI) Reference Model for networking prescribes seven layers of network protocols and strict methods of communication between them. An implementation of a protocol may communicate only with a peer entity speaking the same protocol at the same layer or with the protocol–protocol interface of a protocol in the layer immediately above or below in the same system. The ISO networking model is implemented in FreeBSD Reno and 4.4BSD.

The FreeBSD networking implementation, and to a certain extent the socket facility, is more oriented toward the ARPANET Reference Model (ARM). The ARPANET in its original form served as a proof of concept for many networking ideas, such as packet switching and protocol layering. The ARPANET was retired in 1988 because the hardware that supported it was no longer state of the art. Its successors, such as the NSFNET and the Internet, are even larger and serve as communications utilities for researchers and testbeds for Internet gateway research. The ARM predates the ISO model; the ISO model was in large part inspired by the ARPANET research.

Although the ISO model is often interpreted as setting a limit of one protocol communicating per layer, the ARM allows several protocols in the same layer. There are only four protocol layers in the ARM:

- **Process/applications:** This layer subsumes the application, presentation, and session layers of the ISO model. Such user-level programs as the file-transfer protocol (FTP) and Telnet (remote login) exist at this level.
- **Host–host:** This layer corresponds to ISO’s transport and the top part of its network layers. Both the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are in this layer, with TCP on top of IP. TCP corresponds to an ISO transport protocol, and IP performs the addressing functions of the ISO network layer.
- **Network interface:** This layer spans the lower part of the ISO network layer and the entire data-link layer. The protocols involved here depend on the physical network type. The ARPANET uses the IMP-Host protocols, whereas an Ethernet uses Ethernet protocols.
- **Network hardware:** The ARM is primarily concerned with software, so there is no explicit network hardware layer; however, any actual network will have hardware corresponding to the ISO physical layer.

The networking framework in FreeBSD is more generalized than is either the ISO model or the ARM, although it is most closely related to the ARM (Figure A.11).

User processes communicate with network protocols (and thus with other processes on other machines) via the socket facility. This facility corresponds to the ISO session layer, as it is responsible for setting up and controlling communications.

Sockets are supported by protocols—possibly by several, layered one on another. A protocol may provide services such as reliable delivery, suppression of duplicate transmissions, flow control, and addressing, depending on the socket type being supported and the services required by any higher protocols.

A protocol may communicate with another protocol or with the network interface that is appropriate for the network hardware. There is little restriction in the general framework on what protocols may communicate with what other

ISO reference model	ARPANET reference model	4.2BSD layers	example layering
application	process applications	user programs and libraries	telnet
presentation		sockets	sock_stream
session transport		protocol	TCP IP
network data link	network interface	network interfaces	Ethernet driver
hardware		network hardware	interlan controller

Figure A.11 Network reference models and layering.

protocols or on how many protocols may be layered on top of one another. The user process may, by means of the raw socket type, directly access any layer of protocol from the uppermost used to support one of the other socket types, such as streams, down to a raw network interface. This capability is used by routing processes and also for new protocol development.

There tends to be one **network-interface driver** per network controller type. The network interface is responsible for handling characteristics specific to the local network being addressed. This arrangement ensures that the protocols using the interface do not need to be concerned with these characteristics.

The functions of the network interface depend largely on the **network hardware**, which is whatever is necessary for the network to which it is connected. Some networks may support reliable transmission at this level, but most do not. Some networks provide broadcast addressing, but many do not.

The socket facility and the networking framework use a common set of memory buffers, or *mbufs*. These are intermediate in size between the large buffers used by the block I/O system and the C-lists used by character devices. An *mbuf* is 128 bytes long; 112 bytes may be used for data, and the rest is used for pointers to link the *mbuf* into queues and for indicators of how much of the data area is actually in use.

Data are ordinarily passed between layers—socket–protocol, protocol–protocol, or protocol–network interface—in *mbufs*. This ability to pass the buffers containing the data eliminates some data copying, but there is still frequently a need to remove or add protocol headers. It is also convenient and efficient for many purposes to be able to hold data that occupy an area the size of the memory-management page. Thus, the data of an *mbuf* may reside not in the *mbuf* itself but elsewhere in memory. There is an *mbuf* page table for this purpose, as well as a pool of pages dedicated to *mbuf* use.

A.10 Summary

The early advantages of UNIX were that this system was written in a high-level language, was distributed in source form, and provided powerful operating-system primitives on an inexpensive platform. These advantages led to UNIX's popularity at educational, research, and government institutions and eventually in the commercial world. This popularity first produced many strains of UNIX with variant and improved facilities.

UNIX provides a file system with tree-structured directories. The kernel supports files as unstructured sequences of bytes. Direct access and sequential access are supported through system calls and library routines.

Files are stored as an array of fixed-size data blocks with perhaps a trailing fragment. The data blocks are found by pointers in the inode. Directory entries point to inodes. Disk space is allocated from cylinder groups to minimize head movement and to improve performance.

UNIX is a multiprogrammed system. Processes can easily create new processes with the *fork* system call. Processes can communicate with pipes or, more generally, sockets. They may be grouped into jobs that may be controlled with signals.

Processes are represented by two structures: the process structure and the user structure. CPU scheduling is a priority algorithm with dynamically computed priorities that reduces to round-robin scheduling in the extreme case.

FreeBSD memory management is swapping supported by paging. A *pagedaemon* process uses a modified second-chance page-replacement algorithm to keep enough free frames to support the executing processes.

Page and file I/O uses a block buffer cache to minimize the amount of actual I/O. Terminal devices use a separate character-buffering system.

Networking support is one of the most important features in FreeBSD. The socket concept provides the programming mechanism to access other processes, even across a network. Sockets provide an interface to several sets of protocols.

Exercises

- A.1** Does FreeBSD give scheduling priority to I/O- or CPU-bound processes? For what reason does it differentiate between these categories, and why is one given priority over the other? How does it know which of these categories fits a given process?
- A.2** Early UNIX systems used swapping for memory management, but FreeBSD uses paging and swapping. Discuss the advantages and disadvantages of the two memory methods.
- A.3** Describe the modifications to a file system that FreeBSD makes when a process requests the creation of a new file */tmp/foo* and writes to that file sequentially until the file size reaches 20 KB.
- A.4** Directory blocks in FreeBSD are written synchronously when they are changed. Consider what would happen if they were written asynchronously. Describe the state of the file system if a crash occurred after all the files in a directory were deleted but before the directory entry was updated on disk.
- A.5** Describe the process to recreate the free list after a crash in 4.1BSD.
- A.6** What effects on system performance would the following changes to FreeBSD have? Explain your answers.
 - a. Clustering disk I/O into larger chunks
 - b. Implementing and using shared memory to pass data between processes, rather than using RPC or sockets
 - c. Using the ISO seven-layer networking model, rather than the ARM model
- A.7** What socket type should be used to implement an intercomputer file-transfer program? What type should be used for a program that periodically tests to see whether another computer is up on the network? Explain your answer.

Bibliographical Notes

Textbooks describing variants of the UNIX system are those by Holt [1983], discussing the Tunis operating system; Comer [1984, 1987], discussing the Xinu operating system; and Tanenbaum and Woodhull [1997], describing the Minix operating system.

FreeBSD is described in *The FreeBSD Handbook* can be downloaded from <http://www.freebsd.org/>.

The Mach System



In this appendix we examine the Mach operating system. Mach is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced system. Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. Its multiprocessing support is also exceedingly flexible, accommodating shared-memory systems as well as systems with no memory shared between processors. Mach is designed to run on computer systems ranging from one processor to thousands of processors. In addition, Mach is easily ported to many varied computer architectures. A key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware.

Although many experimental operating systems are being designed, built, and used, Mach satisfies the needs of the masses better than the others because it offers full compatibility with UNIX 4.3 BSD. As such, it provides a unique opportunity for us to compare two functionally similar, but internally dissimilar, operating systems. Mach and UNIX differ in their emphases, so our Mach discussion does not exactly parallel our UNIX discussion. In addition, we do not include a section on the user interface, because that component is similar to the user interface in 4.3 BSD. As you will see, Mach provides the ability to layer emulation of other operating systems as well; other operating systems can even run concurrently with Mach.

B.1 History of the Mach System

Mach traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Although Accent pioneered a number of novel operating system concepts, its utility was limited by its inability to execute UNIX applications and its strong ties to a single hardware architecture, which made it difficult to port. Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system and the management of tasks and threads) were developed from scratch. An important goal of the Mach effort was support for multiprocessors.

Appendix B The Mach System

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2 BSD kernel, with BSD kernel components being replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3 BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for Sun 3 workstations followed shortly; 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provides compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach make the kernels in these releases larger than the corresponding BSD kernels. Mach 3 (Figure B.1) moves the BSD code outside of the kernel, leaving a much smaller microkernel. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows replacement of BSD with another operating system or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the virtual-machine concept, but the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. As of Release 3.0, Mach became available on a wide variety of systems, including single-processor Sun, Intel, IBM, and DEC machines and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled into the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. The release of OSF/1 occurred a year later, and it now competes with UNIX System V, Release 4, the operating system of choice among *UNIX International (UI)* members. OSF members include key technological companies such as IBM, DEC, and HP. Mach 2.5 is also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs, of Apple Computer fame. OSF is evaluating Mach 3 as the basis for a

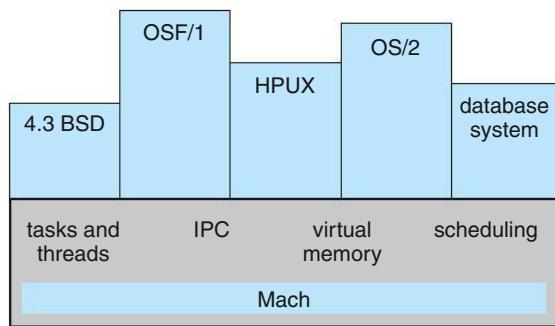


Figure B.1 Mach 3 structure.

future operating-system release, and research on Mach continues at CMU, OSF, and elsewhere.

B.2 Design Principles

The Mach operating system was designed to provide basic mechanisms that most current operating systems lack. The goal is to design an operating system that is BSD-compatible and, in addition, excels in the following areas:

- Support for diverse architectures, including multiprocessors with varying degrees of shared memory access: uniform memory access (UMA), non-uniform memory access (NUMA), and no remote memory access (NORMA)
- Ability to function with varying intercomputer network speeds, from wide-area networks to high-speed local-area networks and tightly coupled multiprocessors
- Simplified kernel structure, with a small number of abstractions (In turn, these abstractions are sufficiently general to allow other operating systems to be implemented on top of Mach.)
- Distributed operation, providing network transparency to clients and an object-oriented organization both internally and externally
- Integrated memory management and interprocess communication, to provide efficient communication of large numbers of data as well as communication-based memory management
- Heterogeneous system support, to make Mach widely available and interoperable among computer systems from multiple vendors

The designers of Mach have been heavily influenced by BSD (and by UNIX, in general), whose benefits include

- A simple programmer interface, with a good set of primitives and a consistent set of interfaces to system facilities
- Easy portability to a wide class of uniprocessors
- An extensive library of utilities and applications
- The ability to combine utilities easily via pipes

Of course, the designers also wanted to redress what they saw as the drawbacks of BSD:

- A kernel that has become the repository of many redundant features—and that consequently is difficult to manage and modify
- Original design goals that made it difficult to provide support for multiprocessors, distributed systems, and shared program libraries (For instance, because the kernel was designed for uniprocessors, it has no provisions for locking code or data that other processors might be using.)

- Too many fundamental abstractions, providing too many similar, competing means with which to accomplish the same task

The development of Mach continues to be a huge undertaking. The benefits of such a system are equally large, however. The operating system runs on many existing uni- and multiprocessor architectures, and it can be easily ported to future ones. It makes research easier, because computer scientists can add features via user-level code, instead of having to write their own tailor-made operating system. Areas of experimentation include operating systems, databases, reliable distributed systems, multiprocessor languages, security, and distributed artificial intelligence. In its current version, the Mach system is usually as efficient as other major versions of UNIX when performing similar tasks.

B.3 System Components

To achieve the design goals of Mach, the developers reduced the operating-system functionality to a small set of basic abstractions, out of which all other functionality can be derived. The Mach approach is to place as little as possible within the kernel but to make what is there powerful enough that all other features can be implemented at the user level.

Mach's design philosophy is to have a simple, extensible kernel, concentrating on communication facilities. For instance, all requests to the kernel, and all data movement among processes, are handled through one communication mechanism. Mach is therefore able to provide system-wide protection to its users by protecting the communications mechanism. Optimizing this one communications path can result in significant performance gains, and it is simpler than trying to optimize several paths. Mach is extensible, because many traditionally kernel-based functions can be implemented as user-level servers. For instance, all pagers (including the default pager) can be implemented externally and called by the kernel for the user.

Mach is an example of an object-oriented system where the data and the operations that manipulate that data are encapsulated into an abstract object. Only the operations of the object are able to act on the entities defined in it. The details of how these operations are implemented are hidden, as are the internal data structures. Thus, a programmer can use an object only by invoking its defined, exported operations. A programmer can change the internal operations without changing the interface definition, so changes and optimizations do not affect other aspects of system operation. The object-oriented approach supported by Mach allows objects to reside anywhere in a network of Mach systems, transparent to the user. The port mechanism, discussed later in this section, makes all of this possible.

Mach's primitive abstractions are the heart of the system and are as follows:

- A **task** is an execution environment that provides the basic unit of resource allocation. It consists of a virtual address space and protected access to system resources via ports, and it may contain one or more threads.
- A **thread** is the basic unit of execution and must run in the context of a task (which provides the address space). All threads within a task share

the tasks' resources (ports, memory, and so on). There is no notion of a *process* in Mach. Rather, a traditional process would be implemented as a *task* with a single thread of control.

- A **port** is the basic object-reference mechanism in Mach and is implemented as a kernel-protected communication channel. Communication is accomplished by sending messages to ports; messages are queued at the destination port if no thread is immediately ready to receive them. Ports are protected by kernel-managed capabilities, or **port rights**; a task must have a port right to send a message to a port. The programmer invokes an operation on an object by *sending* a message to a port associated with that object. The object being represented by a port *receives* the messages.
- A **port set** is a group of ports sharing a common message queue. A thread can receive messages for a port set and thus service multiple ports. Each received message identifies the individual port (within the set) from which it was received; the receiver can use this to identify the object referred to by the message.
- A **message** is the basic method of communication between threads in Mach. It is a typed collection of data objects; for each object, it may contain the actual data or a pointer to out-of-line data. Port rights are passed in messages; this is the only way to move them among tasks. (Passing a port right in shared memory does not work, because the Mach kernel will not permit the new task to use a right obtained in this manner.)
- A **memory object** is a source of memory; tasks can access it by mapping portions of an object (or the entire object) into their address spaces. The object can be managed by a user-mode external memory manager. One example is a file managed by a file server; however, a memory object can be any object for which memory-mapped access makes sense. A mapped buffer implementation of a UNIX pipe is one example.

Figure B.2 illustrates these abstractions, which we elaborate in the remainder of this chapter.

An unusual feature of Mach, and a key to the system's efficiency, is the blending of memory and interprocess-communication (IPC) features. Whereas some other distributed systems (such as Solaris, with its NFS features) have special-purpose extensions to the file system to extend it over a network, Mach provides a general-purpose, extensible merger of memory and messages at the heart of its kernel. This feature not only allows Mach to be used for distributed and parallel programming but also helps in the implementation of the kernel itself.

Mach connects memory management and IPC by allowing each to be used in the implementation of the other. Memory management is based on the use of memory objects. A memory object is represented by a port (or ports), and IPC messages are sent to this port to request operations (for example, `pagein`, `pageout`) on the object. Because IPC is used, memory objects can reside on remote systems and be accessed transparently. The kernel caches the contents of memory objects in local memory. Conversely, memory-management techniques are used in the implementation of message passing.

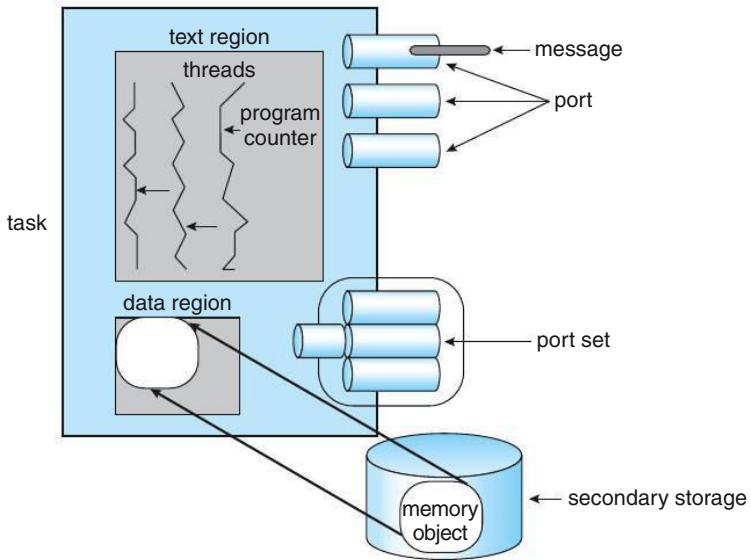


Figure B.2 Mach's basic abstractions.

Where possible, Mach passes messages by moving pointers to shared memory objects, rather than by copying the objects themselves.

IPC tends to involve considerable system overhead; for intrasystem messages, it is generally less efficient than is communication accomplished through shared memory. Because Mach is a message-based kernel, message handling must be carried out efficiently. Most of the inefficiency of message handling in traditional operating systems is due to either the copying of messages from one task to another (for intracomputer messages) or low network-transfer speed (for intercomputer messages). To solve these problems, Mach uses virtual-memory remapping to transfer the contents of large messages. In other words, the message transfer modifies the receiving task's address space to include a copy of the message contents. Virtual copy (or copy-on-write) techniques are used to avoid or delay the actual copying of the data. This approach has several advantages:

- Increased flexibility in memory management to user programs
- Greater generality, allowing the virtual copy approach to be used in tightly and loosely coupled computers
- Improved performance over UNIX message passing
- Easier task migration (Because ports are location independent, a task and all its ports can be moved from one machine to another; all tasks that previously communicated with the moved task can continue to do so because they reference the task only by its ports and communicate via messages to these ports.)

In the sections that follow, we detail the operation of process management, IPC, and memory management. Then, we discuss Mach’s chameleonlike ability to support multiple operating-system interfaces.

B.4 Process Management

A task can be thought of as a traditional process that does not have an instruction pointer or a register set. A task contains a virtual-address space, a set of port rights, and accounting information. A task is a passive entity that does nothing unless it has one or more threads executing in it.

B.4.1 Basic Structure

A task containing one thread is similar to a UNIX process. Just as a `fork` system call produces a new UNIX process, Mach creates a new task to emulate this behavior. The new task’s memory is a duplicate of the parent’s address space, as dictated by the **inheritance attributes** of the parent’s memory. The new task contains one thread, which is started at the same point as the creating `fork` call in the parent. Threads and tasks can also be suspended and resumed.

Threads are especially useful in server applications, which are common in UNIX, since one task can have multiple threads to service multiple requests to the task. Threads also allow efficient use of parallel computing resources. Rather than having one process on each processor, with the corresponding performance penalty and operating-system overhead, a task can have its threads spread among parallel processors. Threads add efficiency to user-level programs as well. For instance, in UNIX, an entire process must wait when a page fault occurs or when a system call is executed. In a task with multiple threads, only the thread that causes the page fault or executes a system call is delayed; all other threads continue executing. Because Mach has kernel-supported threads (Chapter 4), the threads have some cost associated with them. They must have supporting data structures in the kernel, and more complex kernel-scheduling algorithms must be provided. These algorithms and thread states are discussed in Chapter 4.

At the user level, threads may be in one of two states:

- **Running:** The thread is either executing or waiting to be allocated a processor. A thread is considered to be running even if it is blocked within the kernel (waiting for a page fault to be satisfied, for instance).
- **Suspended:** The thread is neither executing on a processor nor waiting to be allocated a processor. A thread can resume its execution only if it is returned to the running state.

These two states are also associated with a task. An operation on a task affects all threads in a task, so suspending a task involves suspending all the threads in it. Task and thread suspensions are separate, independent mechanisms, however, so resuming a thread in a suspended task does not resume the task.

Mach provides primitives from which thread-synchronization tools can be built. This practice is consistent with Mach’s philosophy of providing minimum

yet sufficient functionality in the kernel. The Mach IPC facility can be used for synchronization, with processes exchanging messages at rendezvous points. Thread-level synchronization is provided by calls to start and stop threads at appropriate times. A *suspend count* is kept for each thread. This count allows multiple suspend calls to be executed on a thread, and only when an equal number of resume calls occur is the thread resumed. Unfortunately, this feature has its own limitation. Because it is an error for a `start` call to be executed before a `stop` call (the *suspend count* would become negative), these routines cannot be used to synchronize shared data access. However, `wait` and `signal` operations associated with semaphores, and used for synchronization, can be implemented via the IPC calls. We discuss this method in Section B.5.

B.4.2 The C Threads Package

Mach provides low-level but flexible routines instead of polished, large, and more restrictive functions. Rather than making programmers work at this low level, Mach provides many higher-level interfaces for programming in C and other languages. For instance, the C threads package provides multiple threads of control, shared variables, mutual exclusion for critical sections, and condition variables for synchronization. In fact, C threads is one of the major influences on the POSIX Pthreads standard, which many operating systems are being modified to support. As a result there are strong similarities between the C threads and Pthreads programming interfaces. The thread-control routines include calls to perform these tasks:

- Create a new thread within a task, given a function to execute and parameters as input. The thread then executes concurrently with the creating thread, which receives a thread identifier when the call returns.
- Destroy the calling thread, and return a value to the creating thread.
- Wait for a specific thread to terminate before allowing the calling thread to continue. This call is a synchronization tool, much like the UNIX `wait` system calls.
- Yield use of a processor, signaling that the scheduler can run another thread at this point. This call is also useful in the presence of a preemptive scheduler, as it can be used to relinquish the CPU voluntarily before the time quantum (or scheduling interval) expires if a thread has no use for the CPU.

Mutual exclusion is achieved through the use of spinlocks, as discussed in Chapter 6. The routines associated with mutual exclusion are these:

- The routine `mutex_alloc` dynamically creates a mutex variable.
- The routine `mutex_free` deallocates a dynamically created mutex variable.
- The routine `mutex_lock` locks a mutex variable. The executing thread loops in a spinlock until the lock is attained. A deadlock results if a thread with a lock tries to lock the same mutex variable. Bounded waiting is not guaranteed by the C threads package. Rather, it is dependent on the hardware instructions used to implement the mutex routines.

- The routine `mutex_unlock` unlocks a mutex variable, much like the typical `signal` operation of a semaphore.

General synchronization without busy waiting can be achieved through the use of condition variables, which can be used to implement a condition critical region (or monitor), as described in Chapter 6. A condition variable is associated with a mutex variable and reflects a Boolean state of that variable. The routines associated with general synchronization are these:

- The routine `condition_alloc` dynamically allocates a condition variable.
- The routine `condition_free` deletes a dynamically created condition variable allocated as a result of `condition_alloc`.
- The routine `condition_wait` unlocks the associated mutex variable and blocks the thread until a `condition_signal` is executed on the condition variable, indicating that the event being waited for may have occurred. The mutex variable is then locked, and the thread continues. A `condition_signal` does not guarantee that the condition still holds when the unblocked thread finally returns from its `condition_wait` call, so the awakened thread must loop, executing the `condition_wait` routine until it is unblocked and the condition holds.

As an example of the C threads routines, consider the bounded-buffer synchronization problem of Section 6.6.1. The producer and consumer are represented as threads that access the common bounded-buffer pool. We use a mutex variable to protect the buffer while it is being updated. Once we have exclusive access to the buffer, we use condition variables to block the producer thread if the buffer is full and to block the consumer thread if the buffer is empty. Although this program normally would be written in the C language on a Mach system, we shall use the familiar Pascal-like syntax of previous chapters for clarity. As in Chapter 6, we assume that the buffer consists of n slots, each capable of holding one item. The `mutex` semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers, respectively. The semaphore `empty` is initialized to the value n ; the semaphore `full` is initialized to the value 0. The condition variable `nonempty` is true while the buffer has items in it, and `nonfull` is true if the buffer has an empty slot.

The first step includes the allocation of the mutex and condition variables:

```
mutex_alloc(mutex); condition_alloc(nonempty, nonfull);
```

The code for the producer thread is shown in Figure B.3; the code for the consumer thread is shown in Figure B.4. When the program terminates, the mutex and condition variables need to be deallocated:

```
mutex_free(mutex); condition_free(nonempty, nonfull);
```

B.4.3 The CPU Scheduler

The CPU scheduler for a thread-based multiprocessor operating system is more complex than are its process-based relatives. There are generally more threads

```

do {
    . . .
    // produce an item into nextp
    . . .
    mutex_lock(mutex);
    while(full)
        condition_wait(nonfull, mutex);
    . . .
    // add nextp to buffer
    . . .
    condition_signal(nonempty);
    mutex_unlock(mutex);
} while(TRUE);

```

Figure B.3 The structure of the producer process.

in a multithreaded system than there are processes in a multitasking system. Keeping track of multiple processors is also difficult and is a relatively new area of research. Mach uses a simple policy to keep the scheduler manageable. Only threads are scheduled, so no knowledge of tasks is needed in the scheduler. All threads compete equally for resources, including time quanta.

Each thread has an associated priority number ranging from 0 through 127, which is based on the exponential average of its usage of the CPU. That is, a thread that recently used the CPU for a large amount of time has the lowest priority. Mach uses the priority to place the thread in one of 32 global run queues. These queues are searched in priority order for waiting threads when a processor becomes idle. Mach also keeps per-processor, or local, run queues. A local run queue is used for threads that are bound to an individual processor. For instance, a device driver for a device connected to an individual CPU must run on only that CPU.

Instead of a central dispatcher that assigns threads to processors, each processor consults the local and global run queues to select the appropriate next thread to run. Threads in the local run queue have absolute priority over

```

do {
    . . .
    // produce an item into nextp
    . . .
    mutex_lock(mutex);
    while(full)
        condition_wait(nonfull, mutex);
    . . .
    // add nextp to buffer
    . . .
    condition_signal(nonempty);
    mutex_unlock(mutex);
} while(TRUE);

```

Figure B.4 The structure of the consumer process.

those in the global queues, because it is assumed that they are performing some chore for the kernel. The run queues—like most other objects in Mach—are locked when they are modified to avoid simultaneous changes by multiple processors. To speed dispatching of threads on the global run queue, Mach maintains a list of idle processors.

Additional scheduling difficulties arise from the multiprocessor nature of Mach. A fixed time quantum is not appropriate because, for instance, there may be fewer runnable threads than there are available processors. It would be wasteful to interrupt a thread with a context switch to the kernel when that thread's quantum runs out, only to have the thread be placed right back in the running state. Thus, instead of using a fixed-length quantum, Mach varies the size of the time quantum inversely with the total number of threads in the system. It keeps the time quantum over the entire system constant, however. For example, in a system with 10 processors, 11 threads, and a 100-millisecond quantum, a context switch needs to occur on each processor only once per second to maintain the desired quantum.

Of course, complications still exist. Even relinquishing the CPU while waiting for a resource is more difficult than it is on traditional operating systems. First, a thread must issue a call to alert the scheduler that the thread is about to block. This alert avoids race conditions and deadlocks, which could occur when the execution takes place in a multiprocessor environment. A second call actually causes the thread to be moved off the run queue until the appropriate event occurs. The scheduler uses many other internal thread states to control thread execution.

B.4.4 Exception Handling

Mach was designed to provide a single, simple, consistent exception-handling system, with support for standard as well as user-defined exceptions. To avoid redundancy in the kernel, Mach uses kernel primitives whenever possible. For instance, an exception handler is just another thread in the task in which the exception occurs. Remote procedure call (RPC) messages are used to synchronize the execution of the thread causing the exception (the *victim*) and that of the handler and to communicate information about the exception between the victim and handler. Mach exceptions are also used to emulate the BSD signal package.

Disruptions to normal program execution come in two varieties: internally generated exceptions and external interrupts. Interrupts are asynchronously generated disruptions of a thread or task, whereas exceptions are caused by the occurrence of unusual conditions during a thread's execution. Mach's general-purpose exception facility is used for error detection and debugger support. This facility is also useful for other functions, such as taking a core dump of a bad task, allowing tasks to handle their own errors (mostly arithmetic), and emulating instructions not implemented in hardware.

Mach supports two different granularities of exception handling. Error handling is supported by per-thread exception handling, whereas debuggers use per-task handling. It makes little sense to try to debug only one thread or to have exceptions from multiple threads invoke multiple debuggers. Aside from this distinction, the only difference between the two types of exceptions lies in their inheritance from a parent task. Task-wide exception-handling facilities are

passed from the parent to child tasks, so debuggers are able to manipulate an entire tree of tasks. Error handlers are not inherited and default to no handler at thread- and task-creation time. Finally, error handlers take precedence over debuggers if the exceptions occur simultaneously. The reason for this approach is that error handlers are normally part of the task and therefore should execute normally even in the presence of a debugger.

Exception handling proceeds as follows:

- The victim thread causes notification of an exception's occurrence via a `raise` RPC message sent to the handler.
- The victim then calls a routine to wait until the exception is handled.
- The handler receives notification of the exception, usually including information about the exception, the thread, and the task causing the exception.
- The handler performs its function according to the type of exception. The handler's action involves *clearing* the exception, causing the victim to *resume*, or *terminating* the victim thread.

To support the execution of BSD programs, Mach needs to support BSD-style signals. Signals provide software-generated interrupts and exceptions. Unfortunately, signals are of limited functionality in multithreaded operating systems. The first problem is that, in UNIX, a signal's handler must be a routine in the process receiving the signal. If the signal is caused by a problem in the process itself (for example, a division by zero), the problem cannot be remedied, because a process has limited access to its own context. A second, more troublesome aspect of signals is that they were designed for only single-threaded programs. For instance, it makes no sense for all threads in a task to get a signal, but how can a signal be seen by only one thread?

Because the signal system must work correctly with multithreaded applications for Mach to run 4.3 BSD programs, signals could not be abandoned. Producing a functionally correct signal package required several rewrites of the code, however! A final problem with UNIX signals is that they can be lost. This loss occurs when another signal of the same type occurs before the first is handled. Mach exceptions are queued as a result of their RPC implementation.

Externally generated signals, including those sent from one BSD process to another, are processed by the BSD server section of the Mach 2.5 kernel. Their behavior is therefore the same as it is under BSD. Hardware exceptions are a different matter, because BSD programs expect to receive hardware exceptions as signals. Therefore, a hardware exception caused by a thread must arrive at the thread as a signal. So that this result is produced, hardware exceptions are converted to exception RPCs. For tasks and threads that do not make explicit use of the Mach exception-handling facility, the destination of this RPC defaults to an in-kernel task. This task has only one purpose: Its thread runs in a continuous loop, receiving these exception RPCs. For each RPC, it converts the exception into the appropriate signal, which is sent to the thread that caused the hardware exception. It then completes the RPC, clearing the original exception condition. With the completion of the RPC, the initiating thread reenters the run state. It immediately sees the signal and executes its signal-handling code.

In this manner, all hardware exceptions begin in a uniform way—as exception RPCs. Threads not designed to handle such exceptions, however, receive the exceptions as they would on a standard BSD system—as signals. In Mach 3.0, the signal-handling code is moved entirely into a server, but the overall structure and flow of control is similar to those of Mach 2.5.

B.5 Interprocess Communication

Most commercial operating systems, such as UNIX, provide communication between processes and between hosts with fixed, global names (or Internet addresses). There is no location independence of facilities, because any remote system needing to use a facility must know the name of the system providing that facility. Usually, data in the messages are untyped streams of bytes. Mach simplifies this picture by sending messages between location-independent ports. The messages contain typed data for ease of interpretation. All BSD communication methods can be implemented with this simplified system.

The two components of Mach IPC are ports and messages. Almost everything in Mach is an object, and all objects are addressed via their communications ports. Messages are sent to these ports to initiate operations on the objects by the routines that implement the objects. By depending on only ports and messages for all communication, Mach delivers location independence of objects and security of communication. Data independence is provided by the NetMsgServer task (Section B.5.3).

Mach ensures security by requiring that message senders and receivers have *rights*. A right consists of a port name and a capability—send or receive—on that port, and is much like a capability in object-oriented systems. Only one task may have receive rights to any given port, but many tasks may have send rights. When an object is created, its creator also allocates a port to represent the object and obtains the access rights to that port. Rights can be given out by the creator of the object, including the kernel, and are passed in messages. If the holder of a receive right sends that right in a message, the receiver of the message gains the right, and the sender loses it. A task may allocate ports to allow access to any objects it owns or for communication. The destruction of either a port or the holder of the receive right causes the revocation of all rights to that port, and the tasks holding send rights can be notified if desired.

B.5.1 Ports

A port is implemented as a protected, bounded queue within the kernel of the system on which the object resides. If a queue is full, a sender may abort the send, wait for a slot to become available in the queue, or have the kernel deliver the message for it.

Several system calls provide the port with the following functionalities:

- Allocate a new port in a specified task and give the caller's task all access rights to the new port. The port name is returned.
- Deallocate a task's access rights to a port. If the task holds the receive right, the port is destroyed, and all other tasks with send rights are, potentially, notified.

- Get the current status of a task's port.
- Create a backup port, which is given the receive right for a port if the task containing the receive right requests its deallocation or terminates.

When a task is created, the kernel creates several ports for it. The function `task_self` returns the name of the port that represents the task in calls to the kernel. For instance, for a task to allocate a new port, it would call `port_allocate` with `task_self` as the name of the task that will own the port. Thread creation results in a similar `thread_self` thread kernel port. This scheme is similar to the standard process-ID concept found in UNIX. Another port created for a task is returned by `task_notify` and is the port to which the kernel will send event-notification messages (such as notifications of port terminations).

Ports can also be collected into port sets. This facility is useful if one thread is to service requests coming in on multiple ports—for example, for multiple objects. A port may be a member of no more than one port set at a time; and, if a port is in a set, it may not be used directly to receive messages. Instead, messages will be routed to the port set's queue. A port set may not be passed in messages, unlike a port. Port sets are objects that serve a purpose similar to the 4.3 BSD `select` system call, but they are more efficient.

B.5.2 Messages

A message consists of a fixed-length header and a variable number of typed data objects. The header contains the destination's port name, the name of the reply port to which return messages should be sent, and the length of the message (Figure B.5). The data in the message (or in-line data) were limited to less than 8 KB in Mach 2.5 systems, but Mach 3.0 has no limit. Any data

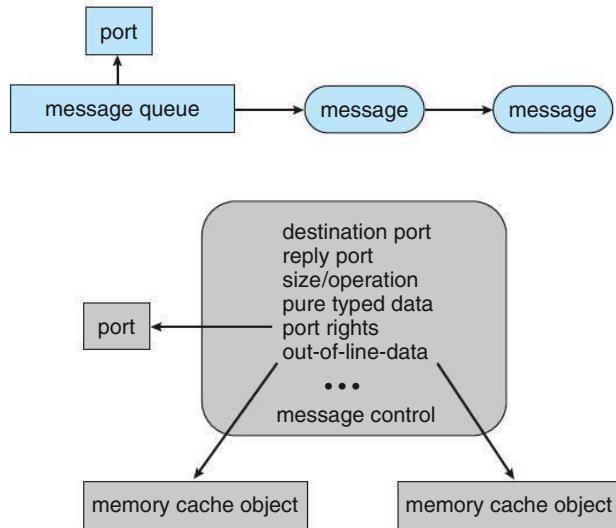


Figure B.5 Mach messages.

exceeding that limit must be sent either in multiple messages or, more likely, via reference by a pointer in a message (or out-of-line data). Each data section may be a simple type (numbers or characters), port rights, or pointers to out-of-line data. Each section has an associated type, so that the receiver can unpack the data correctly even if it uses a byte ordering different from that used by the sender. The kernel also inspects the message for certain types of data. For instance, the kernel must process port information within a message, either by translating the port name into an internal port data structure address or by forwarding it for processing to the NetMsgServer (Section B.5.3).

The use of pointers in a message provides the means to transfer the entire address space of a task in one single message. The kernel also must process pointers to out-of-line data, as a pointer to data in the sender's address space would be invalid in the receiver's—especially if the sender and receiver reside on different systems! Generally, systems send messages by copying the data from the sender to the receiver. Because this technique can be inefficient, especially in the case of large messages, Mach optimizes this procedure. The data referenced by a pointer in a message being sent to a port on the same system are not copied between the sender and the receiver. Instead, the address map of the receiving task is modified to include a copy-on-write copy of the pages of the message. This operation is *much* faster than a data copy and makes message passing efficient. In essence, message passing is implemented via virtual-memory management.

In Version 2.5, this operation was implemented in two phases. A pointer to a region of memory caused the kernel to map that region of memory into its own space temporarily, setting the sender's memory map to copy-on-write mode to ensure that any modifications did not affect the original version of the data. When a message was received at its destination, the kernel moved its mapping to the receiver's address space, using a newly allocated region of virtual memory within that task.

In Version 3, this process was simplified. The kernel creates a data structure that would be a copy of the region if it were part of an address map. On receipt, this data structure is added to the receiver's map and becomes a copy accessible to the receiver.

The newly allocated regions in a task do not need to be contiguous with previous allocations, so Mach virtual memory is said to be *sparse*, consisting of regions of data separated by unallocated addresses. A full message transfer is shown in Figure B.6.

B.5.3 The NetMsgServer

For a message to be sent between computers, the destination of a message must be located, and the message must be transmitted to the destination. UNIX traditionally leaves these mechanisms to the low-level network protocols, which require the use of statically assigned communication endpoints (for example, the port number for services based on TCP or UDP). One of Mach's tenets is that all objects within the system are location independent and that the location is transparent to the user. This tenet requires Mach to provide location-transparent naming and transport to extend IPC across multiple computers.

This naming and transport are performed by the [Network Message Server \(NetMsgServer\)](#), a user-level, capability-based networking daemon

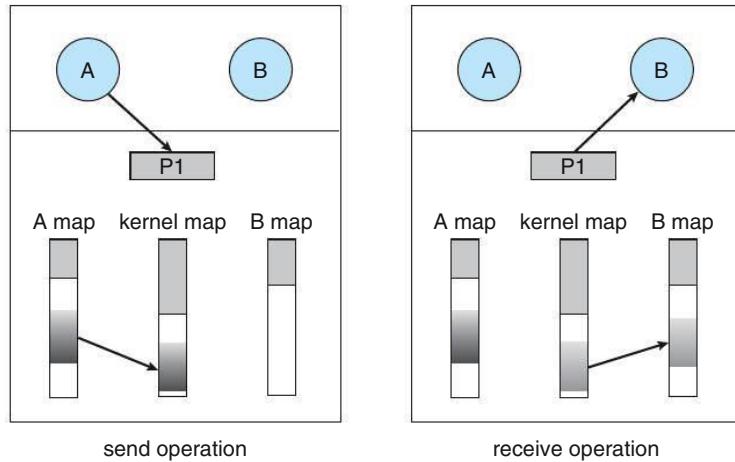


Figure B.6 Mach message transfer.

that forwards messages between hosts. It also provides a primitive network-wide name service that allows tasks to register ports for lookup by tasks on any other computer in the network. Mach ports can be transferred only in messages, and messages must be sent to ports; the primitive name service solves the problem of transferring the first port that allows tasks on different computers to exchange messages. Subsequent IPC interactions are fully transparent; the NetMsgServer tracks all rights and out-of-line memory passed in intercomputer messages and arranges for the appropriate transfers. The NetMsgServers maintain among themselves a distributed database of port rights that have been transferred between computers and of the ports to which these rights correspond.

The kernel uses the NetMsgServer when a message needs to be sent to a port that is not on the kernel's computer. Mach's kernel IPC is used to transfer the message to the local NetMsgServer. The NetMsgServer then uses whatever network protocols are appropriate to transfer the message to its peer on the other computer; the notion of a NetMsgServer is protocol-independent, and NetMsgServers have been built that use various protocols. Of course, the NetMsgServers involved in a transfer must agree on the protocol used. Finally, the NetMsgServer on the destination computer uses that kernel's IPC to send the message to the correct destination task.

The ability to extend local IPC transparently across nodes is supported by the use of proxy ports. When a send right is transferred from one computer to another, the NetMsgServer on the destination computer creates a new port, or proxy, to represent the original port at the destination. Messages sent to this proxy are received by the NetMsgServer and are forwarded transparently to the original port; this procedure is one example of how the NetMsgServers cooperate to make a proxy indistinguishable from the original port.

Because Mach is designed to function in a network of heterogeneous systems, it must provide a way to send between systems data formatted in a way that is understandable by both the sender and the receiver. Unfortunately,

computers differ in the formats they use to store various types of data. For instance, an integer on one system might take 2 bytes to store, and the most significant byte might be stored before the least significant one. Another system might reverse this ordering. The NetMsgServer therefore uses the type information stored in a message to translate the data from the sender's to the receiver's format. In this way, all data are represented correctly when they reach their destination.

The NetMsgServer on a given computer accepts RPCs that add, look up, and remove network ports from the NetMsgServer's name service. As a security precaution, a port value provided in an add request must match that in the remove request for a thread to ask for a port name to be removed from the database.

As an example of the NetMsgServer's operation, consider a thread on node A sending a message to a port that happens to be in a task on node B. The program simply sends a message to a port to which it has a send right. The message is first passed to the kernel, which delivers it to its first recipient, the NetMsgServer on node A. The NetMsgServer then contacts (through its database information) the NetMsgServer on node B and sends the message. The NetMsgServer on node B then presents the message to the kernel with the appropriate local port for node B. The kernel finally provides the message to the receiving task when a thread in that task executes a `msg_receive` call. This sequence of events is shown in Figure B.7.

Mach 3.0 provides an alternative to the NetMsgServer as part of its improved support for NORMA multiprocessors. The NORMA IPC subsystem of Mach 3.0 implements functionality similar to the NetMsgServer directly in the Mach kernel, providing much more efficient internode IPC for multicomputers with fast interconnection hardware. For example, the time-consuming copying of messages between the NetMsgServer and the kernel is eliminated. Use of NORMA IPC does not exclude use of the NetMsgServer; the NetMsgServer can

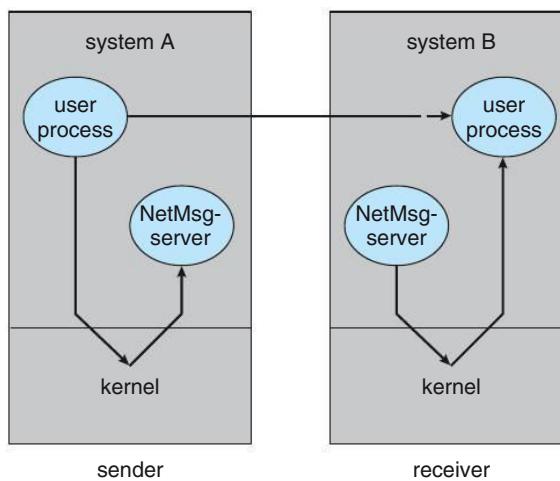


Figure B.7 Network IPC forwarding by NetMsgServer.

still be used to provide Mach IPC service over networks that link a NORMA multiprocessor to other computers. In addition to NORMA IPC, Mach 3.0 also provides support for memory management across a NORMA system and the ability for a task in such a system to create child tasks on nodes other than its own. These features support the implementation of a single-system-image operating system on a NORMA multiprocessor; the multiprocessor behaves like one large system rather than an assemblage of smaller systems (for both users and applications).

B.5.4 Synchronization Through IPC

The IPC mechanism is extremely flexible and is used throughout Mach. For example, it may be used for thread synchronization. A port may be used as a synchronization variable and may have n messages sent to it for n resources. Any thread wishing to use a resource executes a receive call on that port. The thread will receive a message if the resource is available; otherwise, it will wait on the port until a message is available there. To return a resource after use, the thread can send a message to the port. In this regard, receiving is equivalent to the semaphore operation `wait`, and sending is equivalent to `signal`. This method can be used for synchronizing semaphore operations among threads in the same task, but it cannot be used for synchronization among tasks, because only one task may have receive rights to a port. For more general-purpose semaphores, a simple daemon can be written that implements the same method.

B.6 Memory Management

Given the object-oriented nature of Mach, it is not surprising that a principal abstraction in Mach is the memory object. Memory objects are used to manage secondary storage and generally represent files, pipes, or other data that are mapped into virtual memory for reading and writing (Figure B.8). Memory objects may be backed by user-level memory managers, which take the place of the more traditional kernel-incorporated virtual-memory pager found in other operating systems. In contrast to the traditional approach of having the kernel manage secondary storage, Mach treats secondary-storage objects (usually files) as it does all other objects in the system. Each object has a port associated with it and may be manipulated by messages sent to its port. Memory objects—unlike the memory-management routines in monolithic, traditional kernels—allow easy experimentation with new memory-manipulation algorithms.

B.6.1 Basic Structure

The virtual address space of a task is generally sparse, consisting of many holes of unallocated space. For instance, a memory-mapped file is placed in some set of addresses. Large messages are also transferred as shared memory segments. For each of these segments, a section of virtual-memory address is used to provide the threads with access to the message. As new items are mapped or removed from the address space, holes of unallocated memory appear in the address space.

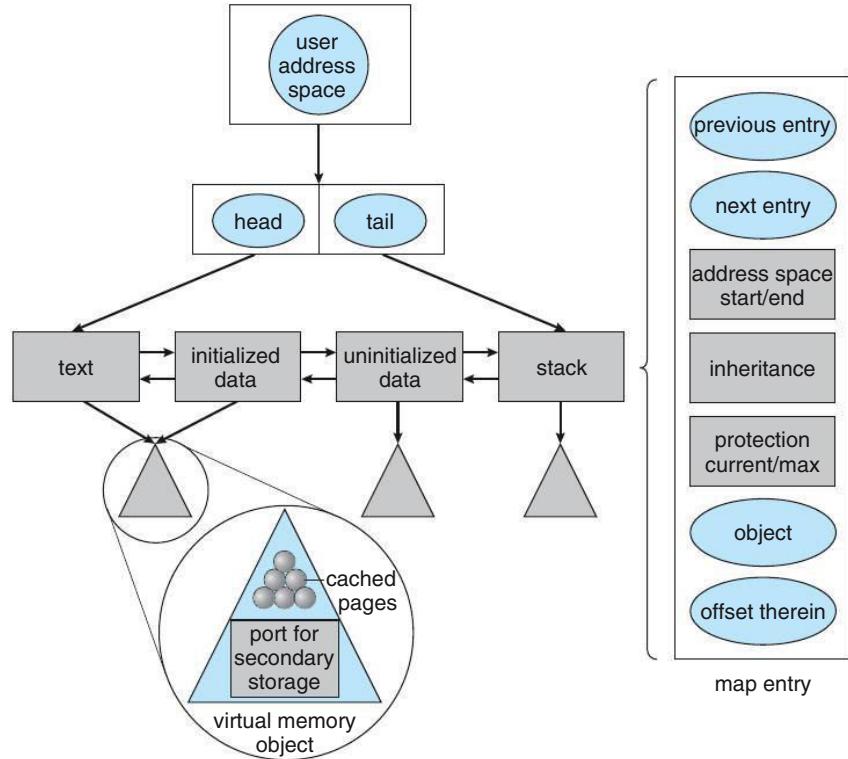


Figure B.8 Mach virtual memory task address map.

Mach makes no attempt to compress the address space, although a task may fail (or crash) if it has no room for a requested region in its address space. Given that address spaces are 4 GB or more, this limitation is not currently a problem. However, maintaining a regular page table for a 4-GB address space for each task, especially one with holes in it, would use excessive amounts of memory (1 MB or more). The key to sparse address spaces is that page-table space is used for only currently allocated regions. When a page fault occurs, the kernel must check to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry. Although the resulting lookup is more complex, the benefits of reduced memory-storage requirements and simpler address-space maintenance make the approach worthwhile.

Mach also has system calls to support standard virtual-memory functionality, including the allocation, deallocation, and copying of virtual memory. When allocating a new virtual-memory object, the thread may provide an address for the object or may let the kernel choose the address. Physical memory is not allocated until pages in this object are accessed. The object's backing store is managed by the default pager (Section B.6.2). Virtual-memory objects are also allocated automatically when a task receives a message containing out-of-line data.

Associated system calls return information about a memory object in a task's address space, change the access protection of the object, and specify how an object is to be passed to child tasks at the time of their creation (shared, copy-on-write, or not present).

B.6.2 User-Level Memory Managers

A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual-memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept that a memory object can be created and serviced by nonkernel tasks (unlike threads, for instance, which are created and maintained by only the kernel) is important. The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage. No assumptions are made by Mach about the content or importance of memory objects, so the memory objects are independent of the kernel.

In several circumstances, user-level memory managers are insufficient. For instance, a task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object (but must be paged), or a memory manager might fail to perform pageout. Mach itself also needs a memory manager to take care of its memory needs. For these cases, Mach provides a default memory manager. The Mach 2.5 default memory manager uses the standard file system to store data that must be written to disk, rather than requiring a separate swap space, as in 4.3 BSD. In Mach 3.0 (and OSF/1), the default memory manager is capable of using either files in a standard file system or dedicated disk partitions. The default memory manager has an interface similar to that of the user-level ones, but with some extensions to support its role as the memory manager that can be relied on to perform pageout when user-level managers fail to do so.

Pageout policy is implemented by an internal kernel thread, the *pageout daemon*. A paging algorithm based on FIFO with second chance (Section 9.4.5) is used to select pages for replacement. The selected pages are sent to the appropriate manager (either user level or default) for actual pageout. A user-level manager may be more intelligent than the default manager, and it may implement a different paging algorithm suitable to the object it is backing (that is, by selecting some other page and forcibly paging it out). If a user-level manager fails to reduce the resident set of pages when asked to do so by the kernel, the default memory manager is invoked, and it pages out the user-level manager to reduce the user-level manager's resident set size. Should the user-level manager recover from the problem that prevented it from performing its own pageouts, it will touch these pages (causing the kernel to page them in again) and can then page them out as it sees fit.

If a thread needs access to data in a memory object (for instance, a file), it invokes the `vm_map` system call. Included in this system call is a port that identifies the object and the memory manager that is responsible for the region. The kernel executes calls on this port when data are to be read or written in that region. An added complexity is that the kernel makes these

calls asynchronously, since it would not be reasonable for the kernel to be waiting on a user-level thread. Unlike the situation with pageout, the kernel has no recourse if its request is not satisfied by the external memory manager. The kernel has no knowledge of the contents of an object or of how that object must be manipulated.

Memory managers are responsible for the consistency of the contents of a memory object mapped by tasks on different machines. (Tasks on a single machine share a single copy of a mapped memory object.) Consider a situation in which tasks on two different machines attempt to modify the same page of an object concurrently. It is up to the manager to decide whether these modifications must be serialized. A conservative manager implementing strict memory consistency would force the modifications to be serialized by granting write access to only one kernel at a time. A more sophisticated manager could allow both accesses to proceed concurrently (for example, if the manager knew that the two tasks were modifying distinct areas within the page and that it could merge the modifications successfully at some future time). Most external memory managers written for Mach (for example, those implementing mapped files) do not implement logic for dealing with multiple kernels, due to the complexity of such logic.

When the first `vm_map` call is made on a memory object, the kernel sends a message to the memory manager port passed in the call, invoking the `memory_manager_init` routine, which the memory manager must provide as part of its support of a memory object. The two ports passed to the memory manager are a **control port** and a **name port**. The control port is used by the memory manager to provide data to the kernel—for example, pages to be made resident. Name ports are used throughout Mach. They do not receive messages but are used simply as a point of reference and comparison. Finally, the memory object must respond to a `memory_manager_init` call with a `memory_object_set_attributes` call to indicate that it is ready to accept requests. When all tasks with send rights to a memory object relinquish those rights, the kernel deallocates the object's ports, thus freeing the memory manager and memory object for destruction.

Several kernel calls are needed to support external memory managers. The `vm_map` call has already been discussed in the paragraph above. In addition, some commands get and set attributes and provide page-level locking when it is required (for instance, after a page fault has occurred but before the memory manager has returned the appropriate data). Another call is used by the memory manager to pass a page (or multiple pages, if read-ahead is being used) to the kernel in response to a page fault. This call is necessary since the kernel invokes the memory manager asynchronously. Finally, several calls allow the memory manager to report errors to the kernel.

The memory manager itself must provide support for several calls so that it can support an object. We have already discussed `memory_object_init` and others. When a thread causes a page fault on a memory object's page, the kernel sends a `memory_object_data_request` to the memory object's port on behalf of the faulting thread. The thread is placed in wait state until the memory manager either returns the page in a `memory_object_data_provided` call or returns an appropriate error to the kernel. Any of the pages that have been modified, or any *previous pages* that the kernel needs to remove from resident memory (due to page aging, for instance), are sent to the memory object via

`memory_object_data_write`. Precious pages are pages that may not have been modified but that cannot be discarded as they otherwise would because the memory manager no longer retains a copy. The memory manager declares these pages to be precious and expects the kernel to return them when they are removed from memory. Precious pages save unnecessary duplication and copying of memory.

Again, there are several other calls for locking, protection information and modification, and the other details with which all virtual memory systems must deal.

In the current version, Mach does not allow external memory managers to affect the page-replacement algorithm directly. Mach does not export the memory-access information that would be needed for an external task to select the least recently used page, for instance. Methods of providing such information are currently under investigation. An external memory manager is still useful for a variety of reasons, however:

- It may reject the kernel's replacement victim if it knows of a better candidate (for instance, MRU page replacement).
- It may monitor the memory object it is backing and request pages to be paged out before the memory usage invokes Mach's pageout daemon.
- It is especially important in maintaining consistency of secondary storage for threads on multiple processors, as we show in Section B.6.3.
- It can control the order of operations on secondary storage to enforce consistency constraints demanded by database management systems. For example, in transaction logging, transactions must be written to a log file on disk before they modify the database data.
- It can control mapped file access.

B.6.3 Shared Memory

Mach uses shared memory to reduce the complexity of various system facilities, as well as to provide these features in an efficient manner. Shared memory generally provides extremely fast interprocess communication, reduces overhead in file management, and helps to support multiprocessing and database management. Mach does not use shared memory for all these traditional shared-memory roles, however. For instance, all threads in a task share that task's memory, so no formal shared-memory facility is needed within a task. However, Mach must still provide traditional shared memory to support other operating-system constructs, such as the UNIX `fork` system call.

It is obviously difficult for tasks on multiple machines to share memory and to maintain data consistency. Mach does not try to solve this problem directly; rather, it provides facilities to allow the problem to be solved. Mach supports consistent shared memory only when the memory is shared by tasks running on processors that share memory. A parent task is able to declare which regions of memory are to be *inherited* by its children and which are to be readable-writable. This scheme is different from copy-on-write inheritance, in which each task maintains its own copy of any changed pages. A writable object is addressed from each task's address map, and all changes are made to the same copy. The threads within the tasks are responsible for coordinating

changes to memory so that they do not interfere with one another (by writing to the same location concurrently). This coordination can be done through normal synchronization methods: critical sections or mutual-exclusion locks.

For the case of memory shared among separate machines, Mach allows the use of external memory managers. If a set of unrelated tasks wishes to share a section of memory, the tasks can use the same external memory manager and access the same secondary-storage areas through it. The implementor of this system would need to write the tasks and the external pager. This pager could be as simple or as complicated as needed. A simple implementation would allow no readers while a page was being written to. Any write attempt would cause the pager to invalidate the page in all tasks currently accessing it. The pager would then allow the write and would revalidate the readers with the new version of the page. The readers would simply wait on a page fault until the page again became available. Mach provides such a memory manager: the Network Memory Server (NetMemServer). For multicomputers, the NORMA configuration of Mach 3.0 provides similar support as a standard part of the kernel. This XMM subsystem allows multicomputer systems to use external memory managers that do not incorporate logic for dealing with multiple kernels; the XMM subsystem is responsible for maintaining data consistency among multiple kernels that share memory and makes these kernels appear to be a single kernel to the memory manager. The XMM subsystem also implements virtual copy logic for the mapped objects that it manages. This virtual copy logic includes both copy-on-reference among multicomputer kernels and sophisticated copy-on-write optimizations.

B.7 Programmer Interface

A programmer can work at several levels within Mach. There is, of course, the system-call level, which, in Mach 2.5, is equivalent to the 4.3 BSD system-call interface. Version 2.5 includes most of 4.3 BSD as one thread in the kernel. A BSD system call traps to the kernel and is serviced by this thread on behalf of the caller, much as standard BSD would handle it. The emulation is not multithreaded, so it has limited efficiency.

Mach 3.0 has moved from the single-server model to support of multiple servers. It has therefore become a true microkernel without the full features normally found in a kernel. Rather, full functionality can be provided via emulation libraries, servers, or a combination of the two. In keeping with the definition of a microkernel, the emulation libraries and servers run outside the kernel at user level. In this way, multiple operating systems can run concurrently on one Mach 3.0 kernel.

An emulation library is a set of routines that lives in a read-only part of a program's address space. Any operating-system calls the program makes are translated into subroutine calls to the library. Single-user operating systems, such as MS-DOS and the Macintosh operating system, have been implemented solely as emulation libraries. For efficiency reasons, the emulation library lives in the address space of the program needing its functionality; but in theory, it could be a separate task.

More complex operating systems are emulated through the use of libraries and one or more servers. System calls that cannot be implemented in the

library are redirected to the appropriate server. Servers can be multithreaded for improved efficiency; BSD and OSF/1 are implemented as single multi-threaded servers. Systems can be decomposed into multiple servers for greater modularity.

Functionally, a system call starts in a task and passes through the kernel before being redirected, if appropriate, to the library in the task's address space or to a server. Although this extra transfer of control will decrease the efficiency of Mach, this decrease is somewhat ameliorated by the ability for multiple threads to execute BSD-like code concurrently.

At the next higher programming level is the C threads package. This package is a run-time library that provides a C language interface to the basic Mach threads primitives. It provides convenient access to these primitives, including routines for the forking and joining of threads, mutual exclusion through mutex variables (Section B.4.2), and synchronization through use of condition variables. Unfortunately, it is not appropriate for the C threads package to be used between systems that share no memory (NORMA systems), since it depends on shared memory to implement its constructs. There is currently no equivalent of C threads for NORMA systems. Other run-time libraries have been written for Mach, including threads support for other languages.

Although the use of primitives makes Mach flexible, it also makes many programming tasks repetitive. For instance, significant amounts of code are associated with sending and receiving messages in each task that uses messages (which, in Mach, is most tasks). The designers of Mach therefore provide an interface generator (or stub generator) called *MIG*. MIG is essentially a compiler that takes as input a definition of the interface to be used (declarations of variables, types and procedures) and generates the RPC interface code needed to send and receive the messages fitting this definition and to connect the messages to the sending and receiving threads.

B.8 Summary

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced operating system.

The Mach operating system was designed with three critical goals in mind:

- Emulate 4.3 BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.
- Have a modern operating system that supports many memory models and parallel and distributed computing.
- Design a kernel that is simpler and easier to modify than is 4.3 BSD.

As we have shown, Mach is well on its way to achieving these goals.

Mach 2.5 includes 4.3 BSD in its kernel, which provides the emulation needed but enlarges the kernel. This 4.3 BSD code has been rewritten to provide the same 4.3 functionality but to use the Mach primitives. This change allows the 4.3 BSD support code to run in user space on a Mach 3.0 system.

Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communications method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual-memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual-memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks.

By providing low-level, or primitive, system calls from which more complex functions can be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual-machine systems.

Exercises

- B.1** What three features of Mach make it appropriate for distributed processing?
- B.2** Name two ways in which port sets are useful in implementing parallel programs.
- B.3** Consider an application that maintains a database of information and provides facilities for other tasks to add, delete, and query the database. Give three configurations of ports, threads, and message types that could be used to implement this system. Which is the best? Explain your answer.
- B.4** Outline a task that would migrate subtasks (tasks it creates) to other systems. Include information about how it would decide when to migrate tasks, which tasks to migrate, and how the migration would take place.
- B.5** Name two types of applications for which you would use the MIG package.
- B.6** Why would someone use low-level system calls instead of the C threads package?
- B.7** Why are external memory managers not able to replace internal page-replacement algorithms? What information would the external managers need in order to make page-replacement decisions? Why would providing this information violate the principle behind the external managers?
- B.8** Why is it difficult to implement mutual exclusion and condition variables in an environment where similar CPUs do not share any memory? What approach and mechanism could be used to make such features available on a NORMA system?
- B.9** What are the advantages of rewriting the 4.3 BSD code as an external, user-level library, rather than leaving it as part of the Mach kernel? Are there any disadvantages? Explain your answer.

Bibliographical Notes

The Accent operating system was described by Rashid and Robertson [1981]. A historical overview of the progression from an even earlier system, RIG, through Accent to Mach was given by Rashid [1986]. General discussions concerning the Mach model were offered by Tevanian and Smith [1989].

Accetta et al. [1986] presented an overview of the original design of Mach. The Mach scheduler was described in detail by Tevanian et al. [1987a] and Black [1990]. An early version of the Mach shared memory and memory-mapping system was presented by Tevanian et al. [1987b].

The most current description of the C threads package appeared in Cooper and Draves [1987]; MIG was described by Draves et al. [1989]. An overview of these packages' functionality and a general introduction to programming in Mach was presented by Walmer and Thompson [1989] and Boykin et al. [1993].

Black et al. [1988] discussed the Mach exception-handling facility. A multithreaded debugger based on this mechanism was described in Caswell and Black [1989].

A series of talks about Mach sponsored by the OSF UNIX consortium is available on videotape from OSF. Topics include an overview, threads, networking, memory management, many internal details, and some example implementations of Mach. The slides from these talks were given in OSF [1989].

The news group `comp.os.mach`, which is accessible on systems where USENET News is available (most educational institutions in the United States, and some overseas), is used to exchange information on the Mach project and its components.

An overview of the microkernel structure of Mach 3.0, complete with performance analysis of Mach 2.5 and 3.0 compared with other systems, was given in Black et al. [1992]. Details of the kernel internals and interfaces of Mach 3.0 were provided in Loepere [1992]. Tanenbaum [1992] presented a comparison of Mach and Amoeba. Discussions concerning parallelization in Mach and 4.3 BSD were offered by Boykin and Langerman [1990].

Ongoing research was presented at USENIX Mach and Micro-kernel Symposia [USENIX 1990, USENIX 1991, and USENIX 1992b]. Active research areas include virtual memory, real time, and security (McNamee and Armstrong [1990]).

Credits

Figures B.1, B.6, and B.8 reproduced with permission from Open Software Foundation, Inc. Excerpted from Mach Lecture Series, OSF, October 1989, Cambridge, Massachusetts.

Figures B.1 and B.8 presented by R. Rashid of Carnegie Mellon University and Figure B.7 presented by D. Julin of Carnegie Mellon University.

Figure B.6 from Accetta, Baron, Bolosky, Golub, Rashid, Tevanian, and Young, "Mach: A New Kernel Foundation for UNIX Development," Proceedings of Summer USENIX, June 1986, Atlanta, Georgia. Reprinted with permission of the authors.

Capitolo 1 – Introduzione

Esercizi di ripasso

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

Risposta:

I tre scopi principali di un sistema operativo sono:

- Fornire agli utenti un ambiente per l'esecuzione dei programmi sull'hardware del computer in modo comodo ed efficiente.
- Allocare le risorse del computer in base alle necessità del dato problema. Il processo di assegnazione delle risorse deve essere il più equo ed efficiente possibile.
- Assolvere, come programma di controllo, due compiti principali: (1) Supervisione dell'esecuzione dei programmi degli utenti allo scopo di evitare errori e un utilizzo improprio del computer; (2) gestione del funzionamento e controllo dei dispositivi di I/O.

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

Risposta:

I sistemi monoutente dovrebbero massimizzare l'uso del sistema per l'utente. Una GUI può "sprecare" cicli di CPU, ma ottimizza l'interazione dell'utente con il sistema.

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

Risposta:

La difficoltà principale è mantenere il sistema operativo entro i limiti di tempo fissati di un sistema real-time. Se il sistema non completa un task in un determinato periodo di tempo, può causare il collasso di tutto il sistema in esecuzione. Pertanto, nello scrivere un sistema operativo per un sistema real-time il programmatore deve assicurarsi che gli algoritmi di scheduling non consentano tempi di risposta superiori ai vincoli fissati.

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

Risposta:

Un argomento a favore dell'inclusione di applicazioni diffuse all'interno del sistema operativo è che quando l'applicazione è incorporata nel sistema è probabile che sia maggiormente in grado di sfruttare le funzionalità del kernel, ottenendo quindi vantaggi in termini di prestazioni rispetto a un'applicazione eseguita al di fuori del kernel. Gli argomenti contro l'inclusione di applicazioni all'interno del sistema operativo sono comunque, in genere, predominanti: (1) le applicazioni sono applicazioni e non parte di un sistema operativo, (2) tutti i vantaggi prestazionali dell'esecuzione all'interno del kernel sono controbilanciati da vulnerabilità di sicurezza, (3) il sistema operativo cresce in dimensione.

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

Risposta:

La distinzione tra modalità kernel e modalità utente fornisce una forma rudimentale di protezione nel seguente modo. E' possibile eseguire determinate istruzioni solo quando la CPU è in modalità kernel. Allo stesso modo, i dispositivi hardware possono essere accessibili solo quando il programma viene eseguito in modalità kernel. Anche il controllo su quando gli interrupt possono essere abilitati o disabilitati è possibile solo quando la CPU è in modalità kernel. Di conseguenza, la CPU ha una capacità molto limitata durante l'esecuzione in modalità utente e ciò permette la protezione delle risorse critiche.

1.6 Quali delle seguenti istruzioni dovrebbero essere privilegiate?

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

Risposta:

Devono essere privilegiate le seguenti istruzioni: impostare il timer, cancellare la memoria, disattivare le interruzioni, modificare le informazioni nella tabella che indica lo status dei dispositivi, accedere a un dispositivo I/O. Il resto può essere eseguito in modalità utente.

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

Risposta:

I dati richiesti dal sistema operativo (password, controlli di accesso, informazioni sugli account, e così via) dovevano essere conservati nella memoria non protetta (o passare per questa zona di memoria) e quindi essere accessibili agli utenti non autorizzati.

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

Risposta:

Sebbene la maggior parte dei sistemi distinguono solo tra le modalità kernel e utente, alcune CPU supportano altre modalità, che possono essere utilizzate per fornire una politica di sicurezza più raffinata. Ad esempio, invece di distinguere soltanto tra modalità utente e modalità kernel, è possibile identificare diversi tipi di modalità utente. Agli utenti appartenenti allo stesso gruppo potrebbe essere permesso di eseguire codice l'uno dell'altro. Per permettere questa operazione la macchina può entrare in una modalità specifica nel momento in cui uno di questi utenti manda in esecuzione del codice. Quando la macchina è in questa modalità, un membro del gruppo è in grado di eseguire codice appartenente a qualsiasi altro membro.

Un'altra possibilità è quella di fornire una diversificazione all'interno del codice del kernel. Ad esempio, una modalità specifica potrebbe essere utilizzata per consentire ai driver di periferica USB di funzionare. In tal modo, i dispositivi USB potrebbero essere serviti senza dover passare alla modalità kernel, consentendo sostanzialmente ai driver di periferica USB l'esecuzione in una modalità ibrida.

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

Risposta:

Un programma può utilizzare il seguente metodo per calcolare l'ora corrente utilizzando i timer. Innanzitutto, il programma imposta un timer per un certo istante di tempo nel futuro e va in sleep. Quando viene risvegliato dall'interrupt, il programma aggiorna il suo stato locale, che tiene traccia del numero di interrupt ricevuti complessivamente. Si ripete quindi questo processo, continuando a impostare gli interrupt del timer e aggiornando lo stato locale quando gli interrupt vengono effettivamente sollevati.

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

Risposta:

Le cache sono utili quando due o più componenti devono scambiarsi dei dati e i componenti eseguono trasferimenti a velocità differenti. Le cache risolvono il problema di trasferimento fornendo un buffer di velocità intermedia tra i componenti. Se il dispositivo veloce trova i dati di cui ha bisogno nella cache, non ha bisogno di aspettare il dispositivo più lento. I dati nella cache devono essere mantenuti coerenti con i dati dei componenti. Se viene modificato il valore di un dato su un componente e il dato è presente nella cache, la cache deve essere aggiornata. Ciò costituisce un problema in particolare sui sistemi multiprocessore, dove più processi possono eseguire l'accesso a un dato. Un componente può essere sostituito da una cache di pari dimensioni, ma solo se (a) la cache e il componente hanno equivalenti capacità di memorizzazione dello stato (vale a dire, se il componente è in grado di conservare i dati in assenza di elettricità, anche la cache deve essere in grado di conservarli), e se (b) la cache è conveniente, perché di solito i dispositivi di memorizzazione più veloci tendono ad essere più costosi.

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

Risposta:

Il modello client-server distingue in maniera netta i ruoli di client e server. In questo modello, il client richiede servizi che vengono forniti dal server. Il modello peer-to-peer non ha una rigida distinzione tra i ruoli. Tutti i nodi del sistema sono considerati pari e possono quindi agire sia come client che come server, o anche in entrambi i modi. Un nodo può richiedere un servizio a un altro peer, oppure può fornire tale servizio ad altri peer nel sistema.

Ad esempio, consideriamo un sistema di nodi che condividono ricette di cucina. Nel modello client-server, tutte le ricette vengono memorizzati sul server e se un client vuole accedere a una ricetta deve richiederla al server. Utilizzando il modello peer-to-peer, un nodo peer può richiedere la ricetta agli altri nodi e il nodo (anche più d'uno) che possiede la ricetta richiesta può fornirla al nodo richiedente. Notate come ogni peer può agire sia come client (può chiedere ricette) che come server (può fornire ricette).

Capitolo 2 – Strutture dei sistemi operativi

Esercizi di ripasso

2.1 Qual è lo scopo delle chiamate di sistema?

Risposta:

Le chiamate di sistema permettono ai processi di livello utente di richiedere servizi al sistema operativo.

2.2 Quali sono le cinque attività principali di un sistema operativo riguardanti la gestione dei processi?

Risposta:

Le cinque principali attività di un sistema operativo riguardanti la gestione dei processi sono:

- a. Creare ed eliminare processi utente e di sistema;
- b. Sospendere e riprendere i processi;
- c. Fornire meccanismi per la sincronizzazione dei processi;
- d. Fornire meccanismi per la comunicazione tra processi;
- e. Fornire meccanismi per la gestione delle situazioni di stallo.

2.3 Quali sono le tre attività principali di un sistema operativo riguardanti la gestione della memoria?

Risposta:

Le tre principali attività di un sistema operativo riguardanti la gestione della memoria sono:

- a. Tenere traccia delle porzioni di memoria utilizzate e degli utilizzatori di queste aree;
- b. Decidere quali processi devono essere caricati quando diventa disponibile uno spazio di memoria;
- c. Allocare e deallocare memoria al bisogno.

2.4 Quali sono le tre attività principali di un sistema operativo riguardanti la gestione della memoria secondaria?

Risposta:

Le tre principali attività di un sistema operativo riguardanti la gestione della memoria secondaria sono:

- a. Gestione dello spazio di memorizzazione disponibile;
- b. Allocazione dello storage;
- c. Scheduling del disco.

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

Risposta:

L'interprete dei comandi legge i comandi dell'utente (o i comandi presenti in un file) e li esegue, generalmente convertendoli in una o più chiamate di sistema. Non è di solito parte del kernel, perché è soggetto a modifiche.

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

Risposta:

Per avviare un nuovo processo nei sistemi UNIX deve essere eseguita una chiamata di sistema fork, seguita da una chiamata exec. La fork clona il processo attualmente in esecuzione, mentre la exec sovrappone al processo chiamante un nuovo processo basato su un eseguibile distinto.

2.7 Qual è lo scopo dei programmi di sistema?

Risposta:

I programmi di sistema possono essere pensati come una collezione di utili chiamate di sistema. Essi forniscono all'utente funzionalità di base, di modo che l'utente non abbia bisogno di scrivere programmi per risolvere problemi generali.

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

Risposta:

Come in tutti i casi di modularità, il progetto modulare di un sistema operativo porta diversi vantaggi. Innanzitutto, è più semplice eseguire il debug e modificare il sistema, perché i cambiamenti riguardano solo un limitato numero di sezioni del sistema, e non tutte. Inoltre, le informazioni vengono conservate solo dove serve e sono accessibili solo all'interno di una zona definita e limitata, quindi eventuali bug che colpiscono i dati sono limitati ad un modulo o ad un livello specifico.

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

Risposta:

I cinque servizi sono i seguenti.

- a. Esecuzione dei programmi . Il sistema operativo carica il contenuto di un file (o una sua parte) in memoria e comincia la sua esecuzione. Non ci si può affidare a un programma di livello utente per una corretta allocazione del tempo di CPU.
- b. Operazioni di I/O. La comunicazione con dischi, nastri, linee seriali e altri dispositivi avviene a un livello molto basso. L'utente deve specificare solo il dispositivo e l'operazione da eseguire su di esso, mentre il sistema converte la richiesta in comandi specifici del dispositivo o del controller. Non ci si può affidare ai programmi di livello utente per gestire l'accesso, che deve avvenire esclusivamente verso i dispositivi autorizzati e solo quando questi sono inutilizzati.
- c. Manipolazione dei file-system . Molti dettagli nella creazione, cancellazione, assegnazione e denominazione dei file non devono essere realizzati dagli utenti. I blocchi di spazio su disco vengono utilizzati dai file e devono essere tracciati. L'eliminazione di un file richiede di rimuovere le informazioni sul file e di liberare i blocchi allocati. Deve inoltre essere realizzato un controllo sulle protezioni per assicurare l'accesso corretto ai file. I programmi utente non possono garantire il

- rispetto dei metodi di protezione, né essere considerati attendibili per l'assegnazione dei blocchi liberi e la deallocazione dei blocchi in caso di cancellazione di un file.
- d. Comunicazioni. Lo scambio di messaggi tra sistemi richiede la trasformazione dei messaggi in pacchetti di informazioni, l'invio al controller di rete, la trasmissione attraverso un mezzo di comunicazione e il riassemblaggio sul sistema di destinazione. Devono essere gestiti l'ordinamento dei pacchetti e la correzione dei dati. Anche in questo caso, i programmi utente potrebbero non essere in grado di coordinare l'accesso al dispositivo di rete, oppure potrebbero ricevere i pacchetti destinati ad altri processi.
 - e. Rilevamento di errori. Il rilevamento degli errori avviene sia a livello hardware che a livello software. A livello hardware, tutti i trasferimenti di dati devono essere controllati per assicurare che i dati non siano stati danneggiati durante il trasporto. Tutti i dati presenti su supporti devono essere controllati per assicurare che i dati non siano cambiati da quando sono stati scritti. A livello software, i supporti devono essere controllati per garantire la coerenza dei dati; per esempio, deve essere verificato che il numero di blocchi allocati e deallocati corrisponda al numero totale di blocchi sul dispositivo. In questi casi, gli errori sono spesso indipendenti dai processi (ad esempio, in caso di dati corrotti sul dispositivo), quindi ci deve essere un programma globale (il sistema operativo) che gestisce tutti i tipi di errori. Inoltre, affidando la gestione degli errori al sistema operativo, i processi non devono contenere codice per intercettare e correggere tutti i possibili errori in un sistema.

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

Risposta:

Su alcuni dispositivi, come i palmari e i telefoni cellulari, potrebbe non essere disponibile un disco dotato di un file-system. In queste situazioni il sistema operativo deve essere memorizzato nel firmware.

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

Risposta:

Si consideri un sistema su cui si vogliono eseguire sia Windows XP che tre diverse distribuzioni di Linux (ad esempio, RedHat, Debian e Mandrake). Ogni sistema operativo sarà memorizzato sul disco. Durante l'avvio del sistema, un programma speciale (che chiamiamo boot manager) determinerà quale sistema operativo avviare. Ciò significa che durante l'avvio del sistema, invece di eseguire un sistema operativo, viene eseguito il boot manager, responsabile di determinare quale sistema avviare. Tipicamente i boot manager devono essere conservati in determinate posizioni del disco rigido per essere riconosciuti durante l'avvio del sistema. I boot manager forniscono all'utente una selezione di sistemi che è possibile avviare e sono in genere progettati per permettere l'avvio di un sistema operativo di default qualora non venga effettuata nessuna scelta da parte dell'utente.

Capitolo 3 – Processi

Esercizi di ripasso

3.1 Con riferimento al programma mostrato nella Figura 3.30, descrivete l'output prodotto alla LINEA A.

Risposta:

Il risultato è ancora 5, visto che il figlio aggiorna la sua copia del valore. Quando il controllo torna al genitore, il suo valore resta 5.

3.2 Considerando anche il processo padre iniziale, quanti processi vengono creati dal programma della Figura 3.31?

Risposta:

Vengono creati 8 processi.

3.3 Le versioni originali del sistema operativo Apple iOS non fornivano alcuno strumento di elaborazione concorrente. Discutete tre principali complicazioni che l'elaborazione concorrente aggiunge a un sistema operativo.

Risposta:

(Manca sul testo inglese, N.d.T.)

3.4 Il processore Sun UltraSPARC ha diversi set di registri. Descrivete ciò che avviene quando si ha un cambio di contesto nel caso in cui il contesto successivo sia già caricato in un determinato set di registri. Che cosa succede se il contesto successivo è nella memoria (invece che in un set di registri) e tutti i registri sono in uso?

Risposta:

Il puntatore al set di registri corrente viene modificato per puntare al set contenente il nuovo contesto; questa operazione richiede poco tempo. Se il contesto è in memoria, deve essere spostato in memoria uno dei contesti in un set di registri, e il nuovo contesto deve essere caricato dalla memoria nel set di registri. Questo processo richiede un po' più di tempo rispetto ai sistemi con un solo set di registri, variabile a seconda di come avviene la selezione del set di registri che sarà soggetto alla sostituzione.

3.5 Quando un processo crea un nuovo processo utilizzando l'istruzione fork(), quale dei seguenti stati è condiviso tra il processo padre e il processo figlio?

- a. Stack
- b. heap
- c. Segmenti di memoria condivisa

Risposta:

Solo i segmenti di memoria condivisa sono condivisi tra il processo padre e il processo figlio appena creato. Per quanto riguarda stack e heap, vengono effettuate copie per il nuovo processo.

3.6 A proposito del meccanismo RPC, considerate la semantica “esattamente una volta” (*exactly once*). L’algoritmo che implementa questa semantica funziona correttamente anche se il messaggio ACK che restituisce al client va perso a causa di un problema di rete? Descrivete la sequenza di messaggi scambiati e indicate se la semantica “esattamente una volta” viene ancora preservata.

Risposta:

La semantica "esattamente una volta" assicura che una procedura verrà eseguita una e una sola volta. L'algoritmo generale per garantire tale meccanismo utilizza uno schema di ACK combinato con alcuni timestamp (o qualche altro contatore incrementale che consenta al server di distinguere tra i messaggi duplicati).

La strategia generale consiste nell'invio da parte del client di una RPC insieme a un timestamp. Il client farà inoltre partire un orologio di timeout. Il client resterà poi in attesa di uno tra i seguenti due eventi: (1) ricezione di un ACK dal server che indichi che la procedura remota è stata eseguita, o (2) timeout. In caso di timeout del client, si assume che il server non sia stato in grado di eseguire la procedura remota, dunque il client richiama la RPC una seconda volta, con un ulteriore invio di timestamp. Il client non riceve l'ACK nei seguenti casi: (1) la RPC originale non è mai stata ricevuta dal server, o (2) la RPC è stata ricevuta ed eseguita dal server, ma l'ACK è stato perso. Nella situazione (1), l'utilizzo di ACK consente al server di ricevere ed eseguire, alla fine, la RPC. Nella situazione (2), il server riceverà una RPC duplicata e utilizzerà il timestamp per identificare questa richiesta come un duplicato in modo da non eseguire la RPC una seconda volta. È importante notare che il server deve inviare un secondo ACK al client per informarlo che la RPC è stata eseguita .

3.7 Assumendo che un sistema distribuito sia soggetto a malfunzionamento del server, quali meccanismi sarebbero richiesti per garantire la semantica “esattamente una volta” per l'esecuzione di rpc?

Risposta:

Il server deve tenere stabilmente traccia in memoria (ad esempio, per mezzo di un log su disco) delle informazioni riguardanti le operazioni RPC ricevute, l'esito della loro esecuzione e i risultati associati alle operazioni. Quando si verifica un crash del server e viene ricevuto un messaggio RPC, il server può verificare se la RPC era stata precedentemente eseguita e quindi garantire la semantica "esattamente una volta".

Capitolo 4 – Thread

Esercizi di ripasso

4.1 Fornite due esempi di programmi nei quali il multithread offra prestazioni migliori rispetto a soluzioni con un singolo thread.

Risposta:

- a. Un Web server che serve ogni richiesta in un thread separato.
- b. Un'applicazione parallelizzata come una moltiplicazione di matrici in cui diverse parti della matrice possono essere processate in parallelo.
- c. Un programma interattivo come un debugger in cui un thread è utilizzato per monitorare l'input dell'utente, un altro thread rappresenta l'applicazione in esecuzione ed un terzo thread controlla le prestazioni.

4.2 Quali sono due differenze tra i thread a livello utente e i thread a livello kernel? In quali circostanze un tipo è meglio dell'altro?

Risposta:

- a. I thread a livello utente sono sconosciuti al kernel, il quale è a conoscenza soltanto dei thread del kernel.
- b. Su sistemi che utilizzano mappature M: 1 o M:N i thread utente sono pianificati dalla libreria dei thread, mentre il kernel si occupa di pianificare i thread del kernel.
- c. I thread del kernel non hanno bisogno di essere associati ad un processo, mentre ogni thread utente appartiene a un processo. I thread del kernel sono generalmente più costosi da mantenere rispetto ai thread utente, perché devono essere rappresentati con strutture dati del kernel.

4.3 Descrivete le azioni intraprese da un kernel per cambiare contesto tra i thread a livello kernel.

Risposta:

Il cambio di contesto tra i thread richiede tipicamente il salvataggio dei valori dei registri della CPU del thread in uscita e il ripristino dei registri relativi al nuovo thread.

4.4 Quali risorse vengono utilizzate quando si crea un thread? Come differiscono da quelle utilizzate quando si crea un processo?

Risposta:

Poiché un thread è più piccolo di un processo, la creazione di un thread utilizza in genere meno risorse rispetto alla creazione di un processo. La creazione di un processo richiede l'allocazione di un blocco di controllo di processo (PCB), una struttura dati piuttosto grande. La PCB include una mappa della memoria, l'elenco dei file aperti e le variabili di ambiente. L'allocazione e la gestione della mappa di memoria è di solito l'attività più dispendiosa in termini di tempo. La creazione di un thread utente o kernel comporta l'allocazione di una piccola struttura dati per contenere un set di registri, lo stack e la priorità.

4.5 Assumete che un sistema operativo mappi i thread a livello utente sul kernel utilizzando il modello molti a molti e che la mappatura avvenga tramite LWP. Assumete inoltre che il sistema permetta agli sviluppatori di creare dei thread real-time da utilizzare in sistemi real-time. È necessario vincolare un thread real-time a un LWP? Fornite una spiegazione.

Risposta:

Sì, la temporizzazione è fondamentale per le applicazioni real-time. Se un thread viene contrassegnato come real-time, ma non è legato a un LWP, potrebbe essere necessario attendere il collegamento a un LWP prima dell'esecuzione. Considerate un thread in tempo reale in esecuzione (collegato a un LWP) che successivamente si blocca (per eseguire operazioni di I/O, perché è stato prelazionato da un thread real-time con priorità più alta, perché è in attesa di un lock di mutua esclusione, ecc.). Mentre il thread real-time è bloccato, l'LWP a cui è stato collegato viene assegnato a un altro thread. Quando il thread real-time deve essere mandato nuovamente in esecuzione, deve prima aspettare di essere collegato a un LWP. Vincolando un LWP a un thread real-time si assicura che il thread sarà in grado di essere eseguito con un ritardo minimo una volta che viene pianificata la sua esecuzione.

Capitolo 5 – Sincronizzazione dei processi

Esercizi di ripasso

5.1 Nel Paragrafo 5.4 si afferma che disabilitando di frequente le interruzioni si può influenzare l'orologio di sistema. Spiegate perché ciò può succedere e come tali effetti possono essere mitigati.

Risposta:

L'orologio di sistema viene aggiornato ad ogni interrupt del clock. Se gli interrupt sono stati disattivati -in particolare se ciò avviene per un lungo periodo di tempo- è possibile che l'orologio di sistema perda l'orario corretto. L'orologio di sistema è utilizzato anche nello scheduling. Ad esempio, il quanto di tempo assegnato ad un processo viene espresso come numero di segnali del clock. Ad ogni interrupt del clock, lo scheduler determina se il quanto di tempo per il processo attualmente in esecuzione è scaduto. Se sono stati disabilitati gli interrupt del clock, lo scheduler non può assegnare con precisione i quanti di tempo. Questo effetto può essere minimizzato disabilitando gli interrupt solo per periodi molto brevi.

5.2 Spiegate perché Windows, Linux e Solaris implementano multipli meccanismi di locking. Descrivete le circostanze in cui questi sistemi operativi utilizzano spinlock, mutex, semafori, lock mutex adattivi e variabili condizionali. Spiegate, in ciascun caso, perché occorre un tale meccanismo.

Risposta:

Questi sistemi operativi forniscono diversi meccanismi di lock a seconda delle esigenze degli sviluppatori di applicazioni. Gli spinlock sono utili per sistemi multiprocessore in cui un thread può essere eseguito in un busy-loop (per un breve periodo di tempo) per non essere soggetti all'overhead che comporta l'inserimento del thread in una coda di sleep. I mutex sono utili per il lock di risorse. Solaris 2 utilizza mutex adattivi, ovvero mutex realizzati con uno spinlock su macchine multiprocessore. Semafori e variabili condizionali sono strumenti appropriati per la sincronizzazione nei casi in cui una risorsa deve essere trattenuta per un lungo periodo di tempo, poiché in questi casi lo spinning risulta inefficiente.

5.3 Qual è il significato della locuzione *attesa attiva*? Esistono attese di altro genere in un sistema operativo? È possibile evitare completamente l'attesa attiva? Spiegate le vostre risposte.

Risposta:

"Attesa attiva" significa che un processo resta ciclicamente in attesa che una condizione sia soddisfatta senza rilasciare il processore. In alternativa, un processo potrebbe restare in attesa rilasciando il processore, bloccandosi su una condizione e aspettando di essere risvegliato in un momento opportuno. L'attesa attiva può essere evitata, ma ciò comporta il carico aggiuntivo associato alla messa in stato di sleep del processo e al suo risveglio quando viene raggiunto un opportuno stato.

5.4 Spiegate perché gli spinlock non sono adatti ai sistemi monoprocesso, ma sono spesso usati nei sistemi multiprocesso.

Risposta:

Gli spinlock non sono adatti ai sistemi monoprocesso, perché la condizione che permetterebbe l'uscita di un processo dallo spinlock può essere ottenuta soltanto mediante l'esecuzione di un altro processo. Se il processo non rilascia il processore, gli altri processi non hanno la possibilità di impostare la condizione richiesta affinché il primo processo possa fare progressi. In un sistema multiprocesso gli altri processi sono eseguiti su altri processori e possono quindi modificare lo stato e permettere l'uscita del primo processo dallo spinlock.

5.5 Si dimostri che, se le operazioni `wait()` e `signal()` dei semafori non sono eseguite in modo atomico, la mutua esclusione rischia di essere violata.

Risposta:

Un'operazione di `wait` diminuisce il valore associato a un semaforo in maniera atomica. Se vengono eseguite due operazioni di `wait` su un semaforo quando il suo valore è 1, e se queste non sono eseguite atomicamente, diventa possibile che entrambe le operazioni decrementino il valore del semaforo, violando la mutua esclusione.

5.6 Mostrate come si può utilizzare un semaforo binario per implementare la mutua esclusione tra n processi.

Risposta:

Gli n processi condividono un semaforo, `mutex`, inizializzato a 1. Ogni processo P_i è organizzato nel seguente modo.

```
do {  
    wait(mutex);  
    /* sezione critica */  
    signal(mutex);  
    /* sezione non critica */  
} while (true);
```

Capitolo 6 – Scheduling della CPU

Esercizi di ripasso

6.1 Un algoritmo di scheduling della CPU stabilisce un ordine per l'esecuzione dei processi pianificati. Dati n processi da mandare in esecuzione su di un singolo processore, quanti differenti scheduling sono possibili? Date una formula in funzione di n .

Risposta:

$n!$ (n factorial = $n \times n - 1 \times n - 2 \times \dots \times 2 \times 1$).

6.2 Spiegate la differenza tra lo scheduling con e senza prelazione.

Risposta:

Lo scheduling con prelazione permette che un processo sia interrotto nel mezzo della sua esecuzione, prendendo possesso della CPU e assegnandola ad un altro processo. Lo scheduling senza prelazione garantisce che un processo ceda il controllo della CPU solo quando si conclude il tempo di CPU che gli è stato assegnato.

6.3 Supponete che i seguenti processi siano pronti per l'esecuzione agli istanti di tempo indicati nella tabella che segue. Nella tabella è anche indicata la durata della sequenza di operazioni per ogni processo. Nel rispondere alle domande seguenti utilizzate uno scheduling senza prelazione e basate le vostre decisioni sulle informazioni che avete a disposizione nel momento in cui la decisione deve essere presa.

processo Istante di arrivo Durata della sequenza

p_1 0,0 8

p_2 0,4 4

p_3 1,0 1

- Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling FCFS?
- Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling SJF?
- L'algoritmo SJF dovrebbe migliorare le prestazioni, ma si noti che al tempo 0 viene scelto il processo p_1 , perché non si sa ancora che presto arriveranno due processi più brevi. Calcolate il tempo di completamento medio ipotizzando che la CPU venga lasciata inattiva per il primo istante di tempo e che successivamente venga utilizzato l'algoritmo SJF. Ricordate che i processi p_1 e p_2 restano in attesa durante il periodo di inattività, e quindi il loro tempo di attesa può aumentare. Questo algoritmo si potrebbe chiamare scheduling con conoscenza del futuro.

Risposta:

- 10,53
- 9,53
- 6,86

Si ricordi che il tempo di completamento medio è il tempo di completamento meno il tempo di arrivo; si devono dunque sottrarre i tempi di arrivo per effettuare i calcoli. Per FCFS si ottiene il valore 11 se ci si dimentica di sottrarre il tempo di arrivo.

6.4 Quali vantaggi si hanno nell'avere quanti di tempo di dimensioni differenti a differenti livelli in un sistema di code multilivello?

Risposta:

I processi che richiedono di essere serviti con più frequenza, ad esempio i processi interattivi come gli editor, possono essere inseriti in una coda con un piccolo quanto di tempo. I processi senza necessità di alte frequenze di servizio possono essere inseriti in una coda con un quanto più grande, in modo da richiedere meno cambi di contesto per completare l'esecuzione e permettendo dunque un uso più efficiente del computer.

6.5 Molti algoritmi di scheduling della CPU sono parametrici. L'algoritmo RR, per esempio, richiede un parametro che specifichi l'intervallo di tempo. Le code multilivello con retroazione richiedono parametri per specificare il numero di code, l'algoritmo di scheduling da utilizzare per ogni coda, il criterio usato per muovere i processi tra le code, e così via. Questi algoritmi sono dunque classi di algoritmi (per esempio, la classe di algoritmi RR per tutti gli intervalli di tempo, e così via). Una classe di algoritmi ne può includere un'altra (per esempio, l'algoritmo FCFS è un algoritmo RR con intervallo di tempo infinito). Quale relazione intercorre (se esiste una relazione) tra le seguenti coppie di classi di algoritmi?

- a. Priorità e SJF.
- b. Code multilivello con retroazione e FCFS.
- c. Priorità ed FCFS.
- d. RR e SJF.

Risposta:

- a. Il job più breve ha la massima priorità.
- b. Il livello più basso di MLFQ è FCFS.
- c. FCFS assegna la massima priorità al job che è rimasto in vita più a lungo.
- d. Nessuna.

6.6 Supponete che un algoritmo di scheduling (a livello dello scheduling della CPU a breve termine) favorisca quei processi che hanno usato meno CPU in un passato recente. Spiegate perché questo algoritmo favorirà programmi con prevalenza di I/O senza bloccare permanentemente i programmi con prevalenza di elaborazione.

Risposta:

L'algoritmo favorirà i programmi con prevalenza di I/O a causa della relativamente scarsa richiesta di CPU da parte loro; tuttavia, i programmi con prevalenza di elaborazione non rimarranno permanentemente bloccati, perché i programmi con prevalenza di I/O rilasceranno la CPU abbastanza spesso per effettuare il loro I/O.

6.7 Individuate le differenze tra gli scheduling PCS e SCS.

Risposta:

Lo scheduling PCS è locale al processo e consiste nel modo in cui la libreria dei thread pianifica i thread sugli LWP disponibili. Nello scheduling SCS il sistema operativo pianifica i thread del kernel. Su sistemi che utilizzano modelli molti-a-uno o molti-a-molti, i due scheduling sono fondamentalmente diversi. Su sistemi che utilizzano un modello uno-a-uno, PCS e SCS coincidono.

6.8 Supponete che un sistema operativo mappi thread a livello utente in thread a livello kernel utilizzando il modello molti-a-molti e che il mapping venga effettuato mediante l'utilizzo di LWP. Inoltre, il sistema consente agli utenti di creare thread real-time. È necessario legare rigidamente un thread real-time a un LWP?

Risposta:

Sì, altrimenti un thread utente potrebbe dover competere per un LWP libero prima di essere effettivamente pianificato. Legando il thread a un LWP, non c'è latenza in attesa di un LWP disponibile e il thread utente real-time può essere pianificato immediatamente.

6.9 Lo scheduler tradizionale di uNIX stabilisce una relazione inversa tra numeri e priorità: più alto è il numero, minore è la priorità. Lo scheduler ricalcola le priorità dei processi una volta al secondo utilizzando la seguente funzione:

$$\text{priorità} = (\text{utilizzo recente della cpu} / 2) + \text{base}$$

dove base = 60 utilizzo e *utilizzo recente della cpu* è un valore che indica in che misura un processo ha utilizzato la CPU dall'ultima volta in cui le priorità sono state ricalcolate.

Si supponga che l' utilizzo recente della CPU sia 40 per il processo p_1 , 18 per il processo p_2 e 10 per il processo p_3 . Quali saranno le nuove priorità di questi tre processi dopo il ricalcolo? Sulla base di questa informazione, dite se lo scheduler tradizionale di UNIX alza o abbassa la priorità relativa di un processo CPU-bound.

Risposta:

Le priorità assegnate ai processi sono rispettivamente 80, 69 e 65. Lo scheduler abbassa la priorità relativa dei processi CPU-bound.

Capitolo 7 – Stallo dei processi

Esercizi di ripasso

7.1 Elencate tre esempi di stallo che non riguardino l'informatica.

Risposta:

- Due auto che attraversano un ponte ad una sola corsia in direzioni opposte.
- Una persona che scende una scala mentre un'altra persona sta salendo.
- Due treni che viaggiano l'uno verso l'altro sullo stesso binario.

7.2 Supponete che un sistema sia in uno stato non sicuro. Mostrate che è possibile che i processi completino l'esecuzione senza entrare in una situazione di stallo.

Risposta:

Uno stato non sicuro non porta necessariamente a una situazione di stallo, ma si tratta di una situazione in cui non è possibile garantire che la situazione di stallo non si verifichi. E' dunque possibile che un sistema in uno stato non sicuro permetta il completamento di tutti i processi senza il verificarsi di situazioni di stallo. Si consideri una situazione in cui il sistema dispone di 12 risorse assegnate tra i processi P0, P1 e P2. Le risorse sono assegnate secondo il seguente criterio:

Max - Situazione corrente - Fabbisogno

P0 10 5 5

P1 4 2 2

P2 9 3 6

Attualmente ci sono due risorse disponibili. Questo sistema si trova in uno stato non sicuro in quanto il processo P1 può essere completato, liberando così un totale di quattro risorse, ma non possiamo garantire che l'elaborazione di P0 e P2 termini. E' tuttavia possibile che un processo rilasci risorse prima di richiederne ulteriori. Ad esempio, il processo P2 potrebbe rilasciare una risorsa, aumentando così il numero totale di risorse a cinque. Ciò permetterebbe il completamento di P0, con il conseguente rilascio di un totale di nove risorse, consentendo così il completamento del processo P2.

7.3 Considerate la seguente situazione istantanea di un sistema:

Assegnate – Massimo - Disponibili

Rispondete alle seguenti domande servendovi dell'algoritmo del banchiere.

- a. Qual è il contenuto della matrice *Necessità*?
- b. Il sistema è in uno stato sicuro?
- c. Se arriva una richiesta dal processo p_1 di (0,4,2,0) tale richiesta può venire accolta immediatamente?

Risposta:

- a. I valori di Necessità per i processi da P0 a P4 sono, rispettivamente, (0, 0, 0, 0), (0, 7, 5, 0), (1, 0, 0, 2), (0, 0, 2, 0) e (0, 6, 4, 2).
- b. Il sistema è in uno stato sicuro? Sì. Essendo Disponibili pari a (1, 5, 2, 0), sia P0 che P3 possono essere eseguiti. Una volta che il processo P3 è terminato rilascia le sue risorse, che permettono a tutti gli altri processi esistenti di essere eseguiti.
- c. La richiesta può essere accolta immediatamente? Ciò si traduce in un valore di Disponibili pari a (1, 1, 0, 0). Un ordinamento possibile dei processi in grado di terminare è P0, P2, P3, P1 e P4.

7.4 Un metodo possibile per prevenire gli stalli consiste nel disporre di una singola risorsa di ordine più elevato che deve essere richiesta prima di ogni altra risorsa. Lo stallo è possibile, per esempio, se più thread tentano di accedere agli oggetti di sincronizzazione A…E (tali oggetti di sincronizzazione possono includere

mutex, semafori, variabili condition, e simili). Possiamo prevenire gli stalli aggiungendo un sesto oggetto F . Ogni volta che un thread vuole acquisire un lock di sincronizzazione per ciascun oggetto $A \cdots E$, deve prima acquisire il lock per l'oggetto F . Questa soluzione è conosciuta come **inclusione (containment)**: i lock per gli oggetti $A \cdots E$ sono inclusi all'interno del lock per l'oggetto F . Paragonate questo schema con quello ad attesa circolare del paragrafo 7.4.4.

Risposta:

Non si tratta probabilmente di una buona soluzione, perché allarga troppo il campo di applicazione. E' meglio definire una politica di lock con un ambito il più ristretto possibile.

7.5 Dimostrate che l'algoritmo di sicurezza presentato nel paragrafo 7.5.3 richiede un numero di operazioni dell'ordine di $m \times n^2$.

Risposta:

La Figura 7.1 mostra il codice Java che implementa l'algoritmo di sicurezza dell'algoritmo del banchiere (l'implementazione completa dell'algoritmo del banchiere è disponibile per il download).

```
for (int i = 0; i < n; i++) {  
    // trova un thread in grado di terminare  
    for (int j = 0; j < n; j++) {  
        if (!finish[j]) {  
            boolean temp = true;  
            for (int k = 0; k < m; k++) {  
                if (need[j][k] > work[k])  
                    temp = false;  
            }  
            if (temp) { // se questo thread può terminare  
                finish[j] = true;  
                for (int x = 0; x < m; x++)  
                    work[x] += work[j][x];  
            }  
        }  
    }  
}
```

Figure 7.1 L'algoritmo di sicurezza dell'algoritmo del banchiere.

Come è possibile notare, i cicli nidificati più esterni, che vengono eseguiti n volte, danno origine al fattore n^2 . All'interno di questi cicli vi sono due cicli interni sequenziali che vengono ripetuti m volte. L'O grande di questo algoritmo è quindi $O(m \times n^2)$.

7.6 Considerate un sistema informatico che esegua 5000 task al mese e non disponga ne di uno schema per prevenire gli stalli né di uno schema per evitarli. Gli stalli si verificano circa due volte al mese e l'operatore deve terminare e rieseguire circa 10 task per ogni stallo. Ogni task costa circa 2 dollari (in tempo del processore), e i task terminati tendono a essere circa a metà del loro lavoro quando vengono interrotti. Un programmatore di sistema ha stimato che un algoritmo per evitare gli stalli (come l'algoritmo del banchiere) potrebbe essere installato nel sistema con un incremento del 10% circa del tempo medio di esecuzione per task. Siccome la macchina ha attualmente il 30% di periodo d'inattività, tutti i 5000 task al mese potrebbero ancora essere eseguiti, nonostante il tempo di completamento aumenterebbe in media del 20 per cento circa.

- Quali sono i vantaggi dell'installazione dell'algoritmo per evitare gli stalli?
- Quali sono gli svantaggi dell'installazione dell'algoritmo per evitare gli stalli?

Risposta:

Un argomento a favore dell'installazione sul sistema dell'algoritmo per evitare gli stalli è che è possibile garantire l'assenza di situazioni di stallo. Inoltre, nonostante l'aumento del tempo di risposta, tutti i 5.000 task possono essere eseguiti.

Un argomento contro l'installazione dell'algoritmo è che i deadlock si verificano raramente e costano poco quando si verificano.

7.7 Un sistema può individuare che alcuni dei suoi processi sono in una situazione di attesa indefinita? Se la vostra risposta è affermativa, spiegate com'è possibile. Se è negativa, spiegate come il sistema può trattare il problema dell'attesa indefinita.

Risposta:

L'attesa indefinita è un argomento di difficile trattazione in quanto può assumere significati differenti su sistemi diversi. Ai fini della presente domanda, definiremo l'attesa indefinita come una situazione in cui un processo deve attendere oltre un ragionevole periodo di tempo, forse a tempo indeterminato, prima di ricevere una risorsa richiesta. Un modo per rilevare l'attesa indefinita è quello di individuare prima di tutto un periodo di tempo, T, che viene considerato irragionevole. Quando un processo richiede una risorsa, viene avviato un timer e se il tempo trascorso supera T, allora il processo è considerato in attesa indefinita. Una strategia per affrontare l'attesa indefinita consiste nell'adottare una politica in cui le risorse vengono assegnate solo al processo che ha atteso più a lungo. Ad esempio, se il processo Pa ha atteso più a lungo la risorsa X rispetto al processo Pb, la richiesta del processo Pb viene differita fino a quando la richiesta del processo Pa è stata soddisfatta.

Un'altra strategia, meno rigorosa di quella appena descritta, prevede che una risorsa possa essere concessa ad un processo che ha atteso meno di un altro processo a condizione che il secondo non sia in attesa indefinita. Se però un altro processo è considerato in attesa indefinita, allora la sua richiesta viene soddisfatta prima.

7.8 Considerate la seguente politica di allocazione delle risorse. Le richieste e i rilasci di risorse sono sempre possibili. Se una richiesta di risorse non può essere soddisfatta perché queste non sono disponibili, si controllano tutti i processi bloccati in attesa di risorse. Se un processo bloccato possiede le risorse desiderate, queste vengono prelevate e assegnate al processo che le richiede. Il vettore delle risorse attese dal processo bloccato è aggiornato in modo da includere le risorse sottratte al processo.

Si consideri per esempio un sistema con tre tipi di risorse e il vettore *Disponibili* inizializzato a (4,2,2). Se il processo p_0 richiede (2,2,1), le ottiene. Se p_1 richiede (1,0,1), le ottiene. Poi, se p_0 chiede (0,0,1) viene bloccato (risorsa non disponibile). Se p_2 ora chiede (2,0,0) ottiene la risorsa disponibile (1,0,0) e quella che è stata assegnata a p_0 (poiché p_0 è bloccato). Il vettore *Assegnate* di p_0 scende a (1,2,1) e il suo vettore *Necessità* passa a (1,0,1).

a. Possono verificarsi stalli? Se la risposta è affermativa, fornite un esempio. In caso di risposta negativa, specificate quale condizione necessaria non si può verificare.

b. Può verificarsi un blocco indefinito? Spiegate la risposta.

Risposta:

- a. Non può verificarsi una situazione di stallo, perché esiste la prelazione.
- b. Sì. Un processo potrebbe non acquisire mai tutte le risorse necessarie se queste sono continuamente prelazionate da una serie di richieste come quelle del processo C.

7.9 Supponete di aver codificato l'algoritmo di sicurezza per evitare gli stalli e che ora vi sia richiesto di implementare l'algoritmo di rilevamento delle situazioni di stallo. È possibile farlo utilizzando semplicemente il codice dell'algoritmo di sicurezza e ridefinendo $\text{Massimo}_i = \text{Attesa}_i + \text{Assegnate}_i$, dove Attesa_i è un vettore che specifica le risorse attese dal processo i e Assegnate_i è definito come nel paragrafo 7.5? Motivate la risposta.

Risposta:

Sì. Il vettore Massimo rappresenta la massima richiesta che un processo può effettuare. Quando l'algoritmo di sicurezza calcola, usa la matrice Necessità, che rappresenta la differenza Massimo-Assegnate. Un altro modo di pensare la stessa cosa è $\text{Massimo} = \text{Necessità} + \text{Assegnate}$. Secondo la domanda, la matrice Attesa svolge una funzione simile alla matrice Necessità, quindi $\text{Massimo} = \text{Attesa} + \text{Assegnate}$.

7.10 E' possibile che uno stallo coinvolga un solo processo con un singolo thread? Argomentate la risposta.

Risposta:

No. Ciò deriva direttamente dalla condizione di hold-and-wait.

Capitolo 8 – Memoria centrale

Esercizi di ripasso

8.1 Citate due differenze tra indirizzi logici e fisici.

Risposta:

Un indirizzo logico non fa riferimento a un indirizzo reale esistente; piuttosto, si riferisce a un indirizzo astratto in uno spazio di indirizzi astratto. Per contro, un indirizzo fisico si riferisce a un indirizzo fisico di memoria. Un indirizzo logico è generato dalla CPU e viene tradotto in un indirizzo fisico dall'unità di gestione della memoria (MMU). Pertanto, gli indirizzi fisici sono generati dalla MMU.

8.2 Considerate un sistema nel quale un programma possa essere separato in due parti: codice e dati. Il processore sa se necessita di un'istruzione (prelievo di istruzione) o di un dato (prelievo o memorizzazione di dati). perciò, vengono fornite due coppie di registri base e limite: una per le istruzioni e una per i dati. La coppia di registri base e limite per le istruzioni è automaticamente a sola lettura, di modo che i programmi possano essere condivisi tra i diversi utenti. Discutete i vantaggi e gli svantaggi di questo schema.

Risposta:

Il principale vantaggio di questo schema è che si tratta un meccanismo efficace per la condivisione di codice e dati. Ad esempio, una sola copia di un editor o di un compilatore deve essere mantenuta in memoria e il codice può essere condiviso da tutti i processi che necessitano l'accesso al codice dell'editor o del compilatore. Un altro vantaggio è la protezione del codice da modifiche errate. L'unico svantaggio è che il codice e i dati devono essere separati. Questa condizione è di solito rispettata in un codice generato da un compilatore.

8.3 Perché la dimensione delle pagine è sempre una potenza di due?

Risposta:

Ricordiamo che la paginazione è implementata mediante la scomposizione di un indirizzo in un numero di pagina e uno scostamento (offset). Risulta più efficace suddividere l'indirizzo in X bit di pagina e Y bit di offset, anziché effettuare calcoli aritmetici sull'indirizzo per ottenere il numero di pagina e l'offset. Poiché ogni posizione dei bit rappresenta una potenza di 2, la suddivisione di un indirizzo in bit porta a una dimensione di pagina che è una potenza di 2.

8.4 Considerate uno spazio degli indirizzi logici di 64 pagine, ciascuna delle quali di 1024 parole, mappato su una memoria fisica di 32 frame.

- a. Quanti bit ci sono nell'indirizzo logico?
- b. Quanti bit ci sono nell'indirizzo fisico?

Risposta:

- a. Indirizzo logico: 16 bit.
- b. Indirizzo fisico: 15 bit.

8.5 Quale effetto si verifica se si permette a due voci di una tabella delle pagine di puntare allo stesso frame di pagina della memoria? Spiegate come questo effetto potrebbe essere utilizzato per diminuire il tempo necessario per copiare una grande quantità di memoria da uno spazio a un altro. Nel caso in cui vengano aggiornati alcuni byte della prima pagina, quale effetto si avrebbe sulla seconda pagina?

Risposta:

Consentendo a due voci di una tabella delle pagine di puntare allo stesso frame di pagina della memoria, gli utenti possono condividere codice e dati. Se il codice è rientrante, una gran quantità di spazio di memoria può essere risparmiata attraverso l'uso condiviso di programmi grandi come editor di testo, compilatori e sistemi di database. La copia di grandi quantità di memoria potrebbe essere effettuata per mezzo di diverse tabelle delle pagine che puntano alla stessa locazione di memoria.

Tuttavia, la condivisione di codice non-rientrante o di dati permette a qualsiasi utente con accesso al codice di modificarlo e queste modifiche si rifletterebbero sulla "copia" degli altri utenti.

8.6 Descrivete un meccanismo per il quale un segmento potrebbe appartenere allo spazio degli indirizzi di due processi differenti.

Risposta:

Poiché le tabelle dei segmenti sono una raccolta di registri di base-limite, i segmenti possono essere condivisi quando le voci nella tabella dei segmenti di due task differenti puntano alla stessa posizione fisica. Le due tabelle dei segmenti devono avere puntatori di base identici e il numero di segmento condiviso deve essere lo stesso nei due processi.

8.7 In un sistema di segmentazione linkato dinamicamente è possibile condividere segmenti tra processi senza richiedere che abbiano lo stesso numero di segmento.

a. Definite un sistema che permetta il linking statico e la condivisione di segmenti senza richiedere che il numero del segmento sia lo stesso.

b. Descrivete uno schema di paginazione che permetta alle pagine di essere condivise senza richiedere ai numeri delle pagine di essere gli stessi.

Risposta:

Entrambi questi problemi si riconducono a un programma in grado di fare riferimento sia al proprio codice che ai propri dati senza conoscere il numero di segmento o di pagina associato all'indirizzo. MULTICS risolse questo problema associando quattro registri ad ogni processo. Un registro aveva l'indirizzo del segmento di programma attuale, un altro aveva un indirizzo di base per lo stack, un altro aveva un indirizzo di base per i dati globali, e così via. L'idea è che tutti i riferimenti devono essere indiretti, attraverso un registro che fa riferimento al numero di segmento o di pagina corrente. Modificando questi registri, lo stesso codice può essere eseguito da processi diversi, senza lo stesso numero di pagina o di segmento.

8.8 Nell'IBM/370 la memoria viene protetta attraverso l'uso di *chiavi*. Una chiave è di 4 bit. Ogni blocco di memoria di 2 k è associato a una chiave (la chiave di memoria). Anche il processore è associato a una chiave (la chiave di protezione). Un'operazione di memorizzazione è permessa solo se entrambe le chiavi sono uguali oppure se una è uguale zero. Quale dei seguenti schemi di gestione della memoria potrebbe essere usato con successo con questo hardware?

- a. Macchina nuda.
- b. Sistema a singolo utente.
- c. Programmazione multipla con un numero fisso di processi.
- d. Programmazione multipla con un numero variabile di processi.
- e. Paginazione.
- f. Segmentazione.

Risposta:

- a. Protezione non necessaria; chiave di sistema impostata a 0.
- b. Chiave di sistema impostata a 0 quando si è in modalità supervisor.

- c. Dimensione delle regioni fissata, con incrementi di 2k byte; chiave allocata con i blocchi di memoria.
- d. Come sopra.
- e. Dimensione dei frame con incrementi di 2k byte; chiave allocata con le pagine.
- f. Dimensione dei segmenti con incrementi di 2k byte; chiave allocata con i segmenti.

Capitolo 9 – Memoria virtuale

Esercizi di ripasso

9.1 In quali circostanze si verifica un page fault? Descrivete le azioni che vengono intraprese dal sistema operativo in questo caso.

Risposta:

Un page fault si verifica quando si effettua l'accesso a una pagina che non è stata portata nella memoria principale. Il sistema operativo verifica l'accesso alla memoria, interrompendo il programma se l'accesso non è valido. Se invece si tratta di un accesso valido, viene individuato un frame libero e richiesto l'I/O per portare la pagina necessaria nel frame. Al completamento dell'I/O vengono aggiornate la tabella dei processi e la tabella delle pagine e l'istruzione viene riavviata.

9.2 Considerate una successione di riferimenti alle pagine di memoria per un processo con m frame (inizialmente tutti vuoti). La successione ha lunghezza p ; in essa vi sono n distinti numeri di pagina.

Rispondete alle seguenti domande relative agli algoritmi di sostituzione delle pagine in generale:

- Qual è un limite inferiore del numero di page fault?
- Qual è un limite superiore del numero di page fault?

Risposta:

- n
- p

9.3 Considerate la tabella delle pagine per un sistema con indirizzi virtuali e fisici a 12 bit con pagine di 256 byte mostrata nella Figura 9.30. La lista dei frame di pagina liberi è D, E, F (dove D è in testa alla lista, E al secondo posto ed F è in coda).

Convertite i seguenti indirizzi virtuali nei corrispondenti indirizzi fisici, in esadecimale.

Tutti i numeri dati sono esadecimali. (Il trattino nella colonna dei frame di pagina indica che la pagina non è in memoria.)

- 9EF
- 111
- 700
- OFF

Risposta:

- 9EF - OEF
- 111 - 211
- 700 - D00
- OFF - EFF

9.4 Considerate i seguenti algoritmi di sostituzione delle pagine e valutateli basandovi su di una scala a cinque valori da “pessimo” a “ottimo” a seconda del tasso di page fault. Separate gli algoritmi che soffrono dell'anomalia di Belady da quelli che non ne sono affetti:

- sostituzione delle pagine usate meno recentemente (LRU);
- sostituzione delle pagine secondo l'ordine d'arrivo (FIFO);
- sostituzione ottimale;
- sostituzione alla seconda chance.

Risposta:

Valutazione - Algoritmo - Soffre dell'anomalia di Belady?

- 1 - Ottimale - No
- 2 - LRU - No
- 3 - Seconda chance - Sì
- 4 - FIFO - Sì

9.5 Analizzate l'hardware necessario alla paginazione su richiesta.

Risposta:

Per ogni operazione di accesso alla memoria deve essere consultata la tabella delle pagine per verificare se la pagina corrispondente è residente o meno e se il programma ha privilegi di lettura o scrittura sulla pagina. Questi controlli devono essere eseguiti in hardware. Può essere utilizzata come cache una TLB, per migliorare le prestazioni dell'operazione di ricerca.

9.6 Un sistema operativo supporta la memoria virtuale paginata, utilizzando un processore centrale con una durata di ciclo di 1 microsecondo. L'accesso a una pagina diversa da quella corrente richiede 1 ulteriore microsecondo. Le pagine hanno 1.000 parole e lo strumento di paginazione è un tamburo che ruota a 3.000 giri al minuto e trasferisce un milione di parole al secondo. Dal sistema si ottengono le seguenti misurazioni statistiche.

- L'1 per cento di tutte le istruzioni eseguite hanno avuto accesso a una pagina diversa dalla pagina corrente.
- L'80 per cento di queste istruzioni – che hanno cioè avuto accesso a un'altra pagina – hanno avuto accesso a una pagina già in memoria.
- Nel caso in cui sia stata richiesta una nuova pagina, la pagina sostituita è stata modificata nel 50 per cento dei casi.

Calcolate il tempo effettivo di esecuzione delle istruzioni di questo sistema, assumendo che il sistema stia eseguendo un unico processo e che il processore sia fermo durante i trasferimenti dal tamburo.

Risposta:

$$\begin{aligned}\text{tempo effettivo di accesso} &= 0,99 \times (1 \mu\text{sec} + 0,008 \times (2 \mu\text{sec}) \\ &+ 0,002 \times (10000 \mu\text{sec} + 1000 \mu\text{sec}) \\ &+ 0,001 \times (10000 \mu\text{sec} + 1000 \mu\text{sec}) \\ &= (0,99 + 0,016 + 22,0 + 11,0) \mu\text{sec} \\ &= 34,0 \mu\text{sec}\end{aligned}$$

9.7 Considerate un array bidimensionale A:

int A [] [] = new int[100][100];

dove A[0][0] si trova nella posizione 200 in un sistema di memoria paginato con pagine di dimensione 200. Un piccolo processo che manipola la matrice risiede alla pagina 0 (posizioni da 0 a 199); ogni fetch di istruzioni avverrà così dalla pagina 0.

Per tre frame di pagina, quanti page fault vengono generati dai seguenti cicli di inizializzazione dell'array? Utilizzate la sostituzione LRU e ipotizzate che un frame contenga il processo e gli altri due frame siano inizialmente vuoti.

- a. for (int j = 0; j < 100; j++)
for (int i = 0; i < 100; i++)
A[i][j] = 0;
- b. for (int i = 0; i < 100; i++)
for (int j = 0; j < 100; j++)
A[i][j] = 0;

Risposta:

- a. 5.000
- b. 50

9.8 Considerate la seguente successione di riferimenti a pagine di memoria:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Quante eccezioni di page fault si verificherebbero per i seguenti algoritmi di sostituzione, assumendo uno, due, tre, quattro, cinque, sei e sette frame? Ricordate che tutti i frame sono inizialmente vuoti, per cui le vostre prime pagine uniche costeranno un'eccezione ciascuna.

- Sostituzione LRU.
- Sostituzione FIFO.
- Sostituzione ottimale.

Risposta:

Numero di frame - LRU - FIFO - Ottimale

1 20 20 20

2 18 18 15

3 15 16 11

4 10 14 8

5 8 10 7

6 7 10 7

7 7 7 7

9.9 Supponete di voler utilizzare un algoritmo di paginazione che richiede un bit di riferimento (come nella sostituzione alla seconda chance o nel modello del working set) che non viene però fornito dall'hardware. Delineate come potreste simulare un bit di riferimento anche se non fornito dall'hardware, oppure spiegate perché non è possibile mettere in pratica una tale ipotesi. Se possibile, calcolate il costo di questa soluzione.

Risposta:

È possibile utilizzare il bit valido/non valido, fornito dall'hardware, per simulare il bit di riferimento. Si setta inizialmente il bit a non valido. Al primo riferimento viene sollevata una trap per il sistema operativo. Il sistema operativo imposterà un bit software a 1 e il bit valido/non valido al valore valido.

9.10 Avete progettato un nuovo algoritmo per la sostituzione delle pagine che pensate possa essere ottimale. In alcuni complicati test di controllo si verifica l'anomalia di Belady. L'algoritmo può essere considerato ottimale? Argomentate la vostra risposta.

Risposta:

No, un algoritmo ottimale non può soffrire dell'anomalia di Belady perché, per definizione, sostituisce la pagina che non sarà utilizzata per più tempo. L'anomalia di Belady si verifica quando un algoritmo per la sostituzione delle pagine espelle una pagina che sarà necessaria in un futuro immediato, e un algoritmo ottimale non selezionerebbe una tale pagina.

9.11 La segmentazione è simile alla paginazione, ma utilizza "pagine" di dimensione variabile. Definite due algoritmi di sostituzione dei segmenti basati sugli schemi di sostituzione delle pagine FIFO e LRU. Ricordate che, siccome i segmenti non hanno la stessa dimensione, il segmento scelto per essere sostituito può essere troppo piccolo per poter contenere abbastanza locazioni di memoria consecutive per il segmento

richiesto. Considerate strategie per sistemi nei quali i segmenti non possono essere rilocati e per sistemi nei quali ciò è invece possibile.

Risposta:

- a. FIFO. Trova il primo segmento sufficientemente ampio da accogliere il segmento in arrivo. Se la rilocazione non è possibile e nessun segmento è abbastanza grande, seleziona una combinazione di segmenti che siano contigui in memoria, che siano più "vicini al primo della lista" e che possano ospitare il nuovo segmento. Se la rilocazione è possibile, riorganizza la memoria in modo che i primi N segmenti abbassanza grandi per il segmento in arrivo siano contigui in memoria. In entrambi i casi, aggiungi ogni spazio rimanente all'elenco degli spazi liberi.
- b. LRU. Seleziona il segmento che non è stato utilizzato per il più lungo periodo di tempo e che sia abbastanza grande, aggiungendo ogni spazio rimanente all'elenco degli spazi liberi. Se nessun segmento è abbastanza grande, seleziona una combinazione di segmenti "vecchi", contigui in memoria (se la rilocazione non è disponibile) e che siano abbastanza grandi. Se la rilocazione è disponibile, riorganizza gli N segmenti più vecchi affinché siano contigui in memoria e sostituisci con il nuovo segmento.

9.12 Considerate un sistema informatico a paginazione su richiesta nel quale il grado di multiprogrammazione sia attualmente fissato a quattro. Il sistema è stato recentemente sottoposto a misurazioni volte a determinare l'utilizzo del processore e del disco di paginazione. Le alternative che seguono mostrano tre possibili risultati. Per ognuno di questi casi, che cosa sta avvenendo? Il livello di multiprogrammazione può essere incrementato per migliorare l'utilizzo del processore? La paginazione sta dimostrandosi utile?

- a. Utilizzo del processore 13 per cento; utilizzo del disco 97 per cento.
- b. Utilizzo del processore 87 per cento; utilizzo del disco 3 per cento.
- c. Utilizzo del processore 13 per cento; utilizzo del disco 3 per cento.

Risposta:

- a. Si sta verificando un fenomeno di thrashing.
- b. L'utilizzo della CPU è sufficientemente alto da lasciare che le cose proseguano così. Si può incrementare il livello di multiprogrammazione.
- c. Si può incrementare il livello di multiprogrammazione.

9.13 Considerate un sistema operativo per una macchina che utilizza registri base e limite, ma supponete di aver modificato la macchina di modo che metta a disposizione una tabella delle pagine. Si possono configurare le tabelle in modo da simulare registri base e limite? Come? Oppure, perché la cosa non è possibile?

Risposta:

La tabella delle pagine può essere impostata per simulare i registri di base e limite a condizione che la memoria sia allocata in segmenti di dimensione fissa. In questo modo, la base di un segmento può essere inserita nella tabella delle pagine e il bit valido/non valido può essere utilizzato per indicare che quella porzione del segmento è residente nella memoria. Si verificherà qualche problema di frammentazione interna.

Capitolo 10 – Memoria secondaria

Esercizi di ripasso

10.1 Lo scheduling del disco, con algoritmi diversi dall'FCFS, è utile in un ambiente con un solo utente? Argomentate.

Risposta:

In un ambiente con un solo utente la coda di I/O è di solito vuota. Le richieste arrivano generalmente da un unico processo e riguardano un blocco o una sequenza di blocchi consecutivi. In questi casi FCFS è un metodo economico di scheduling del disco. LOOK è però altrettanto semplice da programmare e darà migliori prestazioni quando più processi eseguono operazioni concorrenti di I/O, come quando un browser Web recupera i dati in background mentre il sistema operativo sta paginando e un'altra applicazione è attiva in primo piano.

10.2 Spiegate perché l'SSTF tende a favorire i cilindri centrali rispetto a quelli più interni e più esterni.

Risposta:

Il centro del disco è la posizione avente la più piccola distanza media da tutte le altre tracce. La testina del disco tende quindi ad allontanarsi dai bordi del disco. Un altro modo di vedere la stessa cosa è di pensare che la posizione corrente della testina divide i cilindri in due gruppi. Se la testina non si trova al centro del disco e arriva una nuova richiesta, è più probabile che questa nuova richiesta sia nel gruppo che comprende il centro del disco; la testina, quindi, si muoverà con maggior probabilità in quella direzione.

10.3 Perché la latenza di rotazione non viene solitamente presa in considerazione nello scheduling del disco? Come potreste modificare lo scheduling SSTF, lo SCAN e quello C-SCAN per includervi l'ottimizzazione della latenza?

Risposta:

La maggior parte dei dischi non forniscono all'host le informazioni sulla posizione di rotazione. Anche se lo facessero, il tempo necessario affinché queste informazioni raggiungano lo scheduler sarebbe soggetto a imprecisione ed essendo il tempo impiegato dallo scheduler variabile, le informazioni di posizione rotazionale diverrebbero errate. Inoltre, le richieste del disco sono di solito espresse in termini di numeri di blocchi logici, e il mapping tra blocchi logici e posizioni fisiche è molto complesso.

10.4 Perché è importante bilanciare gli I/O del file system tra i dischi e i controllori su un sistema in un ambiente multitasking?

Risposta:

Un sistema può viaggiare ad una velocità massima pari al suo più lento "collo di bottiglia". Dischi o controller del disco costituiscono spesso il collo di bottiglia nei sistemi moderni, perché le loro prestazioni non possono tenere il passo di quelle della CPU e del bus di sistema. Bilanciando l'I/O tra dischi e controller, non si sovraccarica né un singolo disco, né un controller, e il collo di bottiglia viene evitato.

10.5 Quali sono i tradeoff fra la rilettura delle pagine di codice da un file system e l'utilizzo dell'area di avvicendamento per memorizzarli?

Risposta:

Se le pagine di codice vengono memorizzate nello spazio di swap, possono essere trasferite più rapidamente alla memoria principale (perché l'allocazione dello spazio di swap è ottimizzata per fornire prestazioni migliori rispetto all'allocazione del file-system generale). L'utilizzo dello spazio di swap può richiedere un tempo aggiuntivo di avvio se le pagine vengono copiate al momento dell'invocazione del processo piuttosto che essere copiate su richiesta. Inoltre, deve essere allocato un maggiore spazio di swap se questo viene utilizzato sia per le pagine di codice che per le pagine di dati.

10.6 Esiste un modo per realizzare una memorizzazione delle informazioni veramente stabile? Argomentate la risposta.

Risposta:

Una memorizzazione realmente stabile non perderebbe mai i dati. La tecnica fondamentale per la memorizzazione stabile è il mantenimento di molteplici copie dei dati, in modo che se una copia viene distrutta un'altra copia resta disponibile per l'uso. Per qualsiasi schema è tuttavia possibile immaginare un disastro sufficiente grande da distruggere tutte le copie dei dati.

10.7 A volte un nastro è detto mezzo ad accesso sequenziale, mentre un disco magnetico è considerato un mezzo ad accesso diretto. In realtà, l'idoneità di un dispositivo di memorizzazione all'accesso diretto dipende dalla grandezza del trasferimento. Il termine *velocità di trasferimento in streaming* denota la velocità di un trasferimento di dati che è in corso, escluso l'effetto della latenza di accesso. La *velocità effettiva di trasferimento*, invece, è il rapporto dei byte totali per il totale dei secondi, inclusi i tempo di overhead come la latenza di accesso.

Ipotizzate che, in un computer, la cache di livello 2 abbia una latenza di accesso di 8 nanosecondi e una velocità di trasferimento in streaming di 800 megabyte al secondo, che la memoria principale abbia una latenza di accesso di 60 nanosecondi e una velocità di trasferimento in streaming di 80 megabyte al secondo, che il disco magnetico abbia una latenza di accesso di 15 millisecondi e una velocità di trasferimento in streaming di 5 megabyte al secondo, e che una unità a nastro abbia una latenza di accesso di 60 secondi e una velocità di trasferimento in streaming di 2 megabyte al secondo.

- a. L'accesso diretto causa la diminuzione della velocità effettiva di trasferimento di un dispositivo, perché non vengono trasferiti dati durante il tempo di accesso. Per il disco descritto, qual è la velocità di trasferimento effettiva se in media un accesso è seguito da un trasferimento in streaming di (1) 512 byte, (2) 8 kilobyte, (3) 1 megabyte, e (4) 16 megabyte?
- b. L'utilizzazione di un dispositivo è definita come il rapporto tra la velocità effettiva di trasferimento e la velocità di trasferimento in streaming. Calcolate l'utilizzazione dell'unità a disco per ognuna delle quattro grandezze di trasferimento indicate al punto a.
- c. Supponete che un'utilizzazione del 25 per cento o più sia considerata accettabile. Utilizzando i valori di prestazione indicati, calcolate la dimensione minima di trasferimento per un disco che dia un'utilizzazione accettabile.
- d. Completate la frase seguente: Un disco è un dispositivo ad accesso diretto per trasferimenti superiori a _____ byte ed è un dispositivo ad accesso sequenziale per trasferimenti inferiori.
- e. Calcolate le dimensioni minime di trasferimento che diano utilizzazioni accettabili per cache, memoria e nastro.
- f. Quando un nastro può essere considerato un dispositivo ad accesso diretto e quando invece è un dispositivo ad accesso sequenziale?

Risposta:

- a. La velocità di trasferimento effettiva (ETR) nel caso di 512 byte è calcolata come segue.
ETR = grandezza del trasferimento/tempo di trasferimento.
Se X è la grandezza del trasferimento, allora il tempo del trasferimento è $((X/STR) + \text{latenza})$, dove STR indica la velocità di trasferimento in streaming.
Il tempo di trasferimento è di $15\text{ms} + (512\text{B}/5\text{MB al secondo}) = 15,0097\text{ms}$.

La velocità di trasferimento effettiva è quindi $512B/15,0097ms = 33,12\text{ KB/sec}$.

ETR per 8KB = 0,47MB/sec.

ETR per 1MB = 4,65MB/sec.

ETR per 16MB = 4,98MB/sec.

- b. L'utilizzazione del dispositivo per 512B è $33,12\text{ KB/sec} / 5\text{MB/sec} = 0,0064 = 0,64\%$
Per 8KB = 9,4%.
Per 1MB = 93%.
Per 16MB = 99,6%.
- c. Si risolve $0,25 = \text{ETR}/\text{STR}$ per una grandezza di trasferimento X.
 $\text{STR} = 5\text{MB}$, quindi $1,25\text{MB/S} = \text{ETR}$.
 $1,25\text{MB/S} * ((X/5) + 0,015) = X$.
 $0,25X + 0,01875 = X$.
 $X = 0,025\text{MB}$.
- d. Un disco è un dispositivo ad accesso casuale per trasferimenti maggiori di K bytes (dove K > dimensione del blocco del disco), mentre è un dispositivo ad accesso sequenziale per trasferimenti più piccoli.
- e. Per ottenere le dimensioni minime di trasferimento che danno un'utilizzazione accettabile per la cache si procede come segue.
 $\text{STR} = 800\text{MB}$, $\text{ETR} = 200$, latenza = $8 * 10^{-9}$.
 $200(X\text{MB}/800 + 8 * 10^{-9}) = X\text{MB}$.
 $0,25X\text{MB} + 1600 * 10^{-9} = X\text{MB}$.
 $X = 2,24\text{ byte}$.
Per la memoria:
 $\text{STR} = 80\text{MB}$, $\text{ETR} = 20$, $L = 60 * 10^{-9}$.
 $20(X\text{MB}/80 + 60 * 10^{-9}) = X\text{MB}$.
 $0,25X\text{MB} + 1200 * 10^{-9} = X\text{MB}$.
 $X = 1,68\text{ byte}$.
Per il nastro:
 $\text{STR} = 2\text{MB}$, $\text{ETR} = 0,5$, $L = 60\text{s}$.
 $0,5(X\text{MB}/2 + 60) = X\text{MB}$.
 $0,25X\text{MB} + 30 = X\text{MB}$.
 $X = 40\text{MB}$.
- f. Dipende da come viene utilizzato. Assumiamo di usare il nastro per ripristinare un backup. In questo caso il nastro agisce come un dispositivo ad accesso sequenziale di cui stiamo sequenzialmente leggendo il contenuto. D'altro canto, supponiamo di usare il nastro per accedere a diversi record memorizzati sul nastro. In questo esempio, l'accesso al nastro è arbitrario e viene quindi considerato casuale.

10.8 Può un'organizzazione RAID a livello 1 ottenere prestazioni migliori sulle richieste di lettura rispetto a un'organizzazione RAID a livello 0 (con striping senza ridondanza)? Se sì, come?

Risposta:

Sì, un'organizzazione RAID a livello 1 potrebbe raggiungere prestazioni migliori sulle richieste di lettura. Quando viene eseguita un'operazione di lettura, un sistema RAID a livello 1 può decidere a quale delle due copie del blocco accedere per soddisfare la richiesta. Questa scelta potrebbe essere basata sulla posizione corrente della testina del disco e potrebbe pertanto portare a un'ottimizzazione delle prestazioni quando si sceglie una testina più vicina ai dati da leggere.

Capitolo 11 – Interfaccia del file system

Esercizi di ripasso

11.1 Alcuni sistemi cancellano automaticamente tutti i file utente quando un utente si disconnette oppure quando un job termina, a meno che l'utente non richieda esplicitamente che questi vengano conservati; altri sistemi conservano tutti i file a meno che l'utente non li cancelli esplicitamente. Discutete i meriti di ciascun approccio.

Risposta:

L'eliminazione di tutti i file non espressamente salvati dall'utente ha il vantaggio di ridurre al minimo lo spazio necessario per ciascun utente, non salvando i file indesiderati o non necessari. Il salvataggio di tutti i file che non vengono espressamente eliminati è più sicuro per l'utente, in quanto non è possibile perdere inavvertitamente file quando ci si è dimenticati di salvarli.

11.2 Perché alcuni sistemi tengono traccia del tipo di un file, mentre altri lasciano questo compito all'utente e altri semplicemente non implementano tipi di file multipli? Quale sistema è “migliore”?

Risposta:

Alcuni sistemi permettono diverse operazioni sui file in base alla loro tipologia (ad esempio, un file ASCII può essere letto come uno stream, mentre un file di database può essere letto mediante l'indice dei blocchi). Altri sistemi lasciano una tale interpretazione dei dati di un file al processo e non forniscono aiuto nell'accesso ai dati. Il metodo "migliore" dipende dalle esigenze dei processi del sistema e da quello che un utente richiede al sistema operativo. Se un sistema esegue prevalentemente applicazioni di database, può essere più efficiente che il sistema operativo implementi un file di tipo di database e fornisca opportune operazioni, piuttosto che obbligare ogni programma ad implementare la stessa cosa (magari in modo diverso). In sistemi general-purpose può essere preferibile implementare solo i tipi di file di base per mantenere ridotta la dimensione del sistema operativo e consentire la massima libertà ai processi nel sistema.

11.3 In modo simile, alcuni sistemi mettono a disposizione molte tipologie diverse di strutture per i dati di un file, mentre altri trattano solo un semplice flusso di byte. Quali sono i vantaggi e gli svantaggi di tali approcci?

Risposta:

Un vantaggio di avere il supporto a diverse strutture di file da parte del sistema è che il supporto viene fornito direttamente dal sistema e non dalle singole applicazioni. Inoltre, se il sistema fornisce il supporto per strutture di file differenti, questo sarà implementato presumibilmente in maniera più efficiente rispetto al supporto implementato da un'applicazione.

Lo svantaggio di avere il supporto a diverse strutture di file da parte del sistema è l'aumento delle dimensioni del sistema. Inoltre, applicazioni che hanno bisogno di tipi di file diversi da quelli implementati potrebbero non essere in grado di girare su questi sistemi.

Una strategia alternativa è che il sistema operativo non definisca alcun supporto per le strutture di file, trattando tutti i file come sequenze di byte. Questo è l'approccio adottato dai sistemi UNIX. Il vantaggio di questo approccio è che semplifica il supporto del sistema operativo per il file system, visto che il sistema non deve fornire la struttura per diversi tipi di file. Inoltre, questo approccio consente alle applicazioni di definire strutture di file, evitando così la situazione in cui un sistema non è in grado di fornire la definizione del tipo di file richiesto da una specifica applicazione.

11.4 È possibile simulare una struttura di directory multilivello con una struttura di directory a un livello nella quale possono essere usati nomi arbitrariamente lunghi? Se la risposta è affermativa, spiegate come ciò sia fattibile e confrontate questo schema con lo schema a directory multilivello. In caso di risposta negativa, spiegate che cosa impedisce il successo della simulazione. Come cambierebbe la vostra risposta se la lunghezza del nome dei file fosse limitata a sette caratteri?

Risposta:

Quando possono essere usati nomi arbitrariamente lunghi è possibile simulare una struttura di directory multilivello. Questo può essere fatto, per esempio, utilizzando il carattere "." per indicare la fine di una sottodirectory. Così, per esempio, il nome jim.java.F1 specifica che F1 è un file della sottodirectory java che a sua volta è nella directory principale jim.

Se i nomi di file sono limitati a sette caratteri lo schema appena descritto potrebbe non essere utilizzato e quindi, in generale, la risposta è no. Il miglior approccio in questa situazione è di utilizzare un file specifico come tabella dei simboli (directory) per mappare nomi arbitrariamente lunghi (come jim.java.F1) in nomi arbitrari più brevi (come XX00743), che vengono poi utilizzati per l'accesso vero e proprio ai file.

11.5 Spiegate lo scopo delle operazioni open() e close().

Risposta:

Lo scopo delle operazioni open() e close() è il seguente:

- L'operazione open() informa il sistema che il dato file sta per diventare attivo.
- L'operazione close() informa il sistema che il dato file non viene più utilizzato attivamente dall'utente che ha eseguito l'operazione stessa.

11.6 In alcuni sistemi, una sottodirectory può essere letta e scritta da un utente autorizzato esattamente come un file ordinario. Descrivete i possibili problemi di protezione che ne derivano. Suggerite un metodo per risolvere ciascuno di questi problemi.

Risposta:

- Una delle informazioni conservative in una voce di directory è la posizione del file. Se un utente potesse modificare questa posizione, allora riuscirebbe ad accedere ad altri file indipendentemente dallo schema di protezione degli accessi.
- Non permettere all'utente di scrivere direttamente sulla sottodirectory, ma, piuttosto, fornire operazioni di sistema per farlo.

11.7 Considerate un sistema che supporti 5.000 utenti. Supponete di voler permettere a 4.990 utenti di accedere a un dato file.

- a. Come specifichereste questo schema di protezione in UNIX?
- b. Potete suggerire un altro schema di protezione utilizzabile a questo scopo più efficacemente dello schema fornito da UNIX?

Risposta:

- a. Ci sono due metodi per raggiungere questo obiettivo:
 - i. Creare una lista di controllo degli accessi con i nomi di tutti i 4.990 utenti.
 - ii. Inserire questi 4.990 utenti in un gruppo e impostare di conseguenza l'accesso al gruppo. Questo schema non può sempre essere utilizzato, perché i gruppi di utenti sono limitati dal sistema.
- b. L'accesso universale ai file si applica a tutti gli utenti a meno che il loro nome compaia nella lista di controllo degli accessi con autorizzazioni di accesso differenti. Con questo schema è sufficiente

mettere i nomi dei restanti dieci utenti nella lista di controllo degli accessi privandoli dei privilegi che consentono l'accesso.

11.8 Alcuni ricercatori hanno suggerito che, invece di associare una lista di accesso a ogni file (dove la lista specifica quali utenti possono accedere al file e come), dovremmo avere *una lista di controllo degli utenti* associata a ogni utente (dove la lista specifica a quali file un utente può accedere e come). Discutete i meriti relativi a questi due schemi.

Risposta:

Lista di controllo dei file. Poiché le informazioni di controllo di accesso sono concentrate in un'unica locazione è più facile modificare le informazioni di controllo di accesso e lo spazio richiesto è minore.

Lista di controllo degli utenti. Si ha meno overhead quando si apre un file.

Capitolo 12 – Realizzazione del file system

Esercizi di ripasso

12.1 Prendete in considerazione un file costituito da 100 blocchi. Assumete che il blocco di controllo del file (e il blocco dell'indice, in caso di allocazione indicizzata) sia già in memoria. Calcolate quante operazioni di I/O del disco sono necessarie con le strategie di allocazione contigue, concatenate e indicizzate (singolo livello), se per un blocco, valgono le condizioni che seguono. Nel caso di allocazione contigua, assumete che non ci sia spazio di crescita all'inizio, ma solo alla fine. Assumete inoltre che il blocco di informazione da aggiungere sia salvato in memoria.

- a. Il blocco viene aggiunto all'inizio.
- b. Il blocco viene aggiunto al centro.
- c. Il blocco viene aggiunto alla fine.
- d. Il blocco viene rimosso dall'inizio.
- e. Il blocco viene rimosso dal centro.
- f. Il blocco viene rimosso dalla fine.

Risposta:

I risultati sono i seguenti.

Contigue – Concatenate – Indicizzate

- a. 201 1 1
- b. 101 52 1
- c. 1 3 1
- d. 198 1 0
- e. 98 52 0
- f. 0 100 0

12.2 Quali problemi potrebbero verificarsi se un sistema permettesse di montare il file system simultaneamente in più di una posizione?

Risposta:

Ci sarebbero percorsi multipli allo stesso file e ciò potrebbe confondere gli utenti o incoraggiare errori (l'eliminazione di un file con un percorso elimina il file in tutti gli altri percorsi).

12.3 Perché la mappa di bit per l'allocazione dei file deve essere conservata nella memoria di massa, e non nella memoria principale?

Risposta:

In caso di blocco del sistema (errore di memoria) non si perderebbe la lista dello spazio libero, cosa che invece accadrebbe salvando la mappa di bit nella memoria principale.

12.4 Considerate un sistema che supporta le strategie di allocazione contigua, concatenata e indicizzata. Quali criteri dovrebbero essere impiegati per decidere quale strategia è migliore per un dato file?

Risposta:

- Contigua - se di solito si accede al file sequenzialmente e se il file è relativamente piccolo.
- Concatenata - se il file è di grandi dimensioni e di solito vi si accede sequenzialmente.
- Indicizzata - se il file è di grandi dimensioni e di solito vi si accede in modo casuale.

12.5 Un problema dell'allocazione contigua consiste nel fatto che l'utente deve preallocare abbastanza spazio per ogni file. Se il file diventa più grande dello spazio che gli è stato allocato, devono essere intraprese delle azioni specifiche. Questo problema può essere risolto definendo una struttura di file che consiste in un'area inizialmente contigua (di una dimensione specificata.) Se l'area viene riempita, il sistema operativo definisce automaticamente un'area di overflow linkata all'area contigua iniziale. Se l'area di overflow viene riempita, si alloca una seconda area di overflow. Paragonate questa implementazione con le versioni standard della allocazione contigua e concatenata.

Risposta:

Questo metodo richiede più overhead rispetto all'allocazione contigua standard, ma ne richiede meno rispetto all'allocazione concatenata standard.

12.6 Come possono le cache contribuire a migliorare le prestazioni? Perché i sistemi non utilizzano un maggior numero di cache, oppure cache più grandi, se esse sono così utili?

Risposta:

Le cache permettono a componenti aventi velocità diverse di comunicare in modo più efficiente, memorizzando temporaneamente i dati del dispositivo più lento in un dispositivo più veloce (la cache). Le cache sono, quasi per definizione, più costose rispetto ai dispositivi che servono, dunque l'aumento del numero o delle dimensioni delle cache aumenterebbe il costo del sistema.

12.7 Perché è vantaggioso per l'utente che il sistema operativo allochi dinamicamente le proprie tabelle interne? Quali sono le conseguenze negative in cui incorrono i sistemi operativi che si comportano in questo modo?

Risposta:

Le tabelle dinamiche consentono una maggiore flessibilità rispetto alla crescita dell'utilizzo del sistema – le dimensioni delle tabelle non vengono mai superate, evitando limiti di utilizzo artificiali. Purtroppo, le strutture del kernel e il codice diventano più complicati, con un conseguente maggior rischio di bug. Inoltre, l'utilizzo di una risorsa può occupare più risorse di sistema (che crescono per soddisfare le richieste) rispetto alle tabelle statiche.

12.8 Spiegate come lo strato VSF permetta al sistema operativo di supportare facilmente diversi tipi di file system.

Risposta:

VFS introduce un livello di riferimento indiretto nell'implementazione del file system. Per molti versi VFS è simile alle tecniche di programmazione orientate agli oggetti. Le chiamate di sistema possono essere effettuate in maniera generica (indipendentemente dal tipo di file system). Ogni tipo di file system fornisce proprie chiamate di funzioni e strutture dati allo strato VFS. Una chiamata di sistema viene tradotta a livello VFS nell'opportuna funzione specifica per il file system di destinazione. Il programma chiamante non contiene codice riferito ad uno specifico file system e i livelli superiori delle strutture delle chiamate di sistema sono, analogamente, indipendenti dal file system. La traduzione a livello VFS trasforma queste chiamate generiche in operazioni specifiche del file system.

Capitolo 13 – Sistemi di I/O

Esercizi di ripasso

13.1 Individuate tre vantaggi che si ottengono dal collocare funzionalità all'interno del controllore di un dispositivo piuttosto che nel kernel. Individuate poi tre svantaggi.

Risposta:

Tre vantaggi:

- a. I bug hanno meno probabilità di causare un crash del sistema operativo.
- b. Le prestazioni possono essere migliorate utilizzando hardware dedicato e algoritmi cablati.
- c. Il kernel è semplificato grazie allo spostamento di algoritmi al di fuori di esso.

Tre svantaggi:

- a. I bug sono più difficili da risolvere: è necessaria una nuova versione del firmware o un nuovo hardware.
- b. Analogamente, il miglioramento degli algoritmi richiede un aggiornamento hardware piuttosto che un aggiornamento del kernel o del driver di periferica.
- c. Gli algoritmi embedded potrebbero entrare in conflitto con l'uso del dispositivo da parte delle applicazioni, causando un degrado delle prestazioni.

13.2 L'esempio di handshaking del Paragrafo 13.2 utilizza 2 bit: un bit busy e un bit command-ready. È possibile implementare la stessa negoziazione con un solo bit? Se è possibile, descrivete il protocollo. Se non lo è, spiegate perché un bit non è sufficiente.

Risposta:

E' possibile, utilizzando il seguente algoritmo. Supponiamo di usare solamente il bit busy (ma lo stesso si può fare scegliendo di utilizzare il bit command-ready). Quando il bit è spento, il controller è inattivo. L'host predispone i dati in uscita e impone il bit per segnalare che l'operazione di scrittura è pronta (ciò equivale a impostare il bit command-ready). Quando il controller ha finito cancella il bit busy. L'host può quindi avviare l'operazione successiva. Questa soluzione richiede che sia l'host che il controller abbiano accesso in lettura e scrittura allo stesso bit e ciò può complicare la circuiteria e aumentare il costo del controller.

13.3 Perché un sistema potrebbe utilizzare I/O guidato dalle interruzioni per gestire una porta seriale singola e il polling per gestire un processore di front-end come un terminal concentrator?

Risposta:

Il polling può essere più efficiente dell'I/O guidato dalle interruzioni, ad esempio quando l'I/O è frequente e di breve durata. Anche se una singola porta seriale esegue I/O con poca frequenza e dovrebbe quindi utilizzare le interruzioni, un insieme di porte seriali come quelle di un terminal concentrator può produrre molte operazioni di I/O di breve durata e l'interruzione ad ogni I/O potrebbe appesantire notevolmente il carico del sistema. Un ciclo di polling ben temporizzato allevierebbe il carico senza sprecare troppe risorse per loop senza necessità di I/O.

13.4 Il polling per il completamento di I/O può sprecare un gran numero di cicli di CPU se il processore itera un ciclo busy-waiting molte volte prima che l'I/O sia terminato. D'altro canto, se il dispositivo di I/O è pronto per il servizio, l'interrogazione ciclica può essere molto più efficiente rispetto a rilevare e gestire un'interruzione. Descrivete una strategia ibrida per un dispositivo di I/O che combini polling, sleeping e interruzioni. Per ognuna di queste tre strategie (polling puro, interruzioni pure e ibrida) descrivete un contesto in cui quella strategia sia più efficiente delle altre.

Risposta:

Un approccio ibrido potrebbe passare dal polling alle interrupt a seconda della durata dell'attesa dell'operazione di I/O. Ad esempio, si potrebbe effettuare l'interrogazione N volte e, se il dispositivo è ancora occupato al tempo $N+1$, si potrebbe impostare un interrupt e passare in sleep. Questo approccio permetterebbe di evitare cicli busy-waiting lunghi. Questo metodo funzionerebbe al meglio per tempi d'attesa molto lunghi o molto corti, ma sarebbe inefficiente quando l'I/O viene completato in un tempo $N+T$ (dove T è un numero piccolo di cicli), a causa del sovraccarico portato dal polling con in aggiunta l'impostazione e la cattura delle interruzioni.

Il polling puro è migliore in presenza di tempi di attesa molto brevi. Le interruzioni si comportano meglio con tempi di attesa lunghi.

13.5 In che modo il DMA aumenta la concorrenza? In che senso complica il progetto dell'hardware?

Risposta:

Il DMA aumenta la concorrenza nel sistema permettendo alla CPU di eseguire attività mentre il sistema DMA trasferisce i dati attraverso i bus di sistema e di memoria. Il progetto dell'hardware si complica perché il controller DMA deve essere integrato nel sistema e il sistema deve consentire al controller DMA di comandare il bus. Può inoltre essere necessario il "furto di cicli" per permettere alla CPU e al controller DMA di condividere l'uso del bus di memoria.

13.6 Perché è importante aumentare la velocità del bus di sistema e delle periferiche all'aumentare della velocità della CPU?

Risposta:

Si consideri un sistema che esegue il 50% di I/O e il 50% di calcoli. Raddoppiando le prestazioni della CPU le prestazioni complessive del sistema aumenterebbe soltanto del 50%, mentre raddoppiando entrambi gli aspetti del sistema si otterebbe un incremento del 100%. In generale, è più importante rimuovere il collo di bottiglia e aumentare le prestazioni complessive del sistema, piuttosto che aumentare ciecamente le prestazioni di singole componenti del sistema.

13.7 Fate una distinzione tra un driver STREAMS e un modulo STREAMS.

Risposta:

Il driver STREAMS controlla un dispositivo fisico che può essere coinvolto in un'operazione STREAMS. Il modulo STREAMS modifica il flusso di dati tra l'interfaccia utente e il driver.

Capitolo 14 – Protezione

Esercizi di ripasso

14.1 Quali sono le principali differenze tra liste delle abilitazioni e liste d'accesso?

Risposta:

Una lista di accesso è, per ogni oggetto, una lista composta dai domini con un insieme non vuoto di diritti di accesso per l'oggetto in questione. Una lista delle abilitazioni è una lista di oggetti e delle operazioni consentite su tali oggetti per ciascun dominio.

14.2 Un file nel Burroughs B7000/B6000 MCP può essere contrassegnato come dato sensibile. Quando un tale file viene cancellato, l'area in cui era memorizzato viene sovrascritta da bit casuali. Per quali scopi un tale schema può essere utile?

Risposta:

Può essere utile come misura di sicurezza aggiuntiva per fare in modo che il vecchio contenuto della memoria non sia accessibile, intenzionalmente o accidentalmente, ad un altro programma. Ciò è particolarmente utile in presenza di informazioni altamente riservate.

14.3 In un sistema di protezione ad anello il livello 0 ha l'accesso più ampio agli oggetti, e il livello n (con $n > 0$) ha diritti di accesso più bassi. I diritti di accesso di un programma a un dato livello nell'anello sono considerati un insieme di abilitazioni. Che relazione c'è tra le abilitazioni a un oggetto per un dominio a livello j e un dominio a livello i (per $j > i$)?

Risposta:

D_j è un sottoinsieme di D_i .

14.4 Il sistema RC4000, come altri sistemi, ha definito un albero dei processi tale che tutti i discendenti di un processo possono ottenere risorse (oggetti) e diritti di accesso solo dai loro antenati. Un discendente non potrà quindi mai fare niente di quello che gli antenati non possono fare. La radice dell'albero è il sistema operativo, che ha la capacità di fare ogni cosa. Assumete che l'insieme dei diritti di accesso sia rappresentato tramite una matrice di accesso A . $A(x,y)$ definisce i diritti di accesso del processo x a un oggetto y . Se x è discendente di z , che relazione c'è tra $A(x,y)$ e $A(z,y)$, per un oggetto arbitrario y ?

Risposta:

$A(x,y)$ è un sottoinsieme di $A(z,y)$.

14.5 Quali problemi di protezione possono nascere se viene utilizzata uno stack condiviso per il passaggio di parametri?

Risposta:

Il contenuto dello stack può essere compromesso da altri processi che lo condividono.

14.6 Considerate un ambiente elaborativo in cui a ogni processo e a ogni oggetto del sistema sia associato un numero univoco. Supponete che un processo con numero n possa accedere a oggetti con numero m solo se $n > m$. Quale tipo di struttura di protezione avremmo?

Risposta:

Una struttura gerarchica.

14.7 Considerare un ambiente elaborativo in cui un processo ottiene il privilegio di accedere a un oggetto soltanto n volte. Suggerite uno schema per implementare una tale politica.

Risposta:

L'aggiunta di un contatore intero alle abilitazioni.

14.8 Se tutti i diritti di accesso di un oggetto vengono cancellati, l'oggetto non è più accessibile. A questo punto, lo stesso oggetto dovrebbe essere cancellato, e lo spazio da lui occupato dovrebbe essere restituito al sistema. Suggerite un'efficiente implementazione di questo schema.

Risposta:

Il conteggio dei riferimenti.

14.9 Quali difficoltà si incontrano nella protezione di un sistema in cui gli utenti hanno il permesso di eseguire le loro operazioni di I/O?

Risposta:

Nei capitoli precedenti abbiamo evidenziato una distinzione tra modalità kernel e modalità utente in cui la modalità kernel viene utilizzata per effettuare operazioni privilegiate come l'I/O. Una ragione per cui l'I/O deve essere eseguito in modalità kernel è che richiede l'accesso all'hardware, e un accesso corretto all'hardware è indispensabile per l'integrità del sistema. Se permetessimo agli utenti di eseguire il proprio I/O, non potremmo garantire l'integrità del sistema.

14.10 La lista delle abilitazioni è solitamente mantenuta nello spazio di indirizzi dell'utente. In che modo il sistema assicura che l'utente non possa modificare il contenuto della lista?

Una lista delle abilitazioni è considerata un "oggetto protetto" ed è accessibile all'utente solo indirettamente. Il sistema operativo garantisce che l'utente non possa accedere direttamente a tale lista.

Capitolo 17 – Sistemi distribuiti

Esercizi di ripasso

17.1 Perché il passaggio di pacchetti broadcast tra le reti sarebbe una pessima scelta per i gateway? Quali sarebbero i vantaggi di tale comportamento?

Risposta:

Tutte le trasmissioni broadcast sarebbero propagate a tutte le reti, causando un notevole traffico di rete. Se il traffico broadcast fosse limitato a (pochi) dati importanti, la trasmissione broadcast risparmierebbe ai gateway la fatica di dover eseguire uno speciale software per rilevare questi dati (come le informazioni di sul routing di rete) e ritrasmetterli.

17.2 Discutete vantaggi e svantaggi del salvataggio in una cache di traduzioni di nomi per computer localizzati in domini remoti.

Risposta:

Salvando in una cache le traduzioni dei nomi per i computer che si trovano in domini remoti si ottiene un vantaggio di prestazioni: la risoluzione ripetuta dello stesso nome da diversi computer situati nel dominio locale può essere eseguita localmente senza richiedere una operazione di ricerca del nome in remoto. Lo svantaggio è che ci possono essere incongruenze nelle traduzioni dei nomi quando vengono effettuati aggiornamenti nella mappatura dei nomi in indirizzi IP. Questi problemi di coerenza possono essere risolti invalidando le traduzioni, il che richiede la gestione di stati riguardanti la memorizzazione nella cache di una certa traduzione e alcuni messaggi di invalidazione, oppure contratti in base ai quali una traduzione viene invalidata dopo un certo periodo di tempo. Quest'ultimo approccio richiede meno condizioni e non richiede messaggi di invalidazione, ma potrebbe soffrire di incongruenze temporanee.

17.3 Quali sono i vantaggi e gli svantaggi nell'utilizzo della commutazione di circuito? Per quali tipologie di applicazioni la commutazione di circuito è una strategia applicabile?

Risposta:

La commutazione di circuito garantisce che le risorse di rete necessarie per un trasferimento siano riservate prima della trasmissione. Ciò assicura che i pacchetti non vengano eliminati e che la loro consegna soddisfi i requisiti di qualità del servizio. Lo svantaggio della commutazione di circuito è che richiede un messaggio di andata e ritorno per impostare le riserve. Inoltre, potrebbe essere riservata una quantità eccessiva di risorse, che non sarebbero dunque utilizzate in maniera ottimale. La commutazione di circuito è una strategia applicabile per le applicazioni che hanno continue richieste di risorse di rete e che richiedono le risorse per lunghi periodi di tempo, perché in questo caso i costi iniziali vengono ammortizzati.

17.4 Quali sono due difficili problemi che i progettisti devono risolvere nell'implementazione di un sistema di rete avente la qualità di essere trasparente?

Risposta:

Uno di questi problemi è di far apparire tutti i processori e i dispositivi di storage trasparenti attraverso la rete. In altre parole, il sistema distribuito dovrebbe apparire agli utenti come un unico sistema centralizzato. I file system Andrew e NFS forniscono questa funzione: il file system distribuito appare all'utente come un unico file system, ma in realtà può essere distribuito attraverso una rete.

Un'altra questione riguarda la mobilità degli utenti. Si vuole consentire agli utenti di connettersi al "sistema", piuttosto che ad una specifica macchina (anche se in realtà l'accesso può essere eseguito su una specifica macchina da qualche parte nel sistema distribuito).

17.5 La migrazione di processi su una rete eterogenea è di solito impossibile a causa delle differenze tra le architetture e tra i sistemi operativi. Descrivete un metodo per la migrazione di processi tra architetture diverse che eseguano:

- a. lo stesso sistema operativo;
- b. diversi sistemi operativi.

Risposta:

Nel caso di uno stesso sistema operativo la migrazione di processi è relativamente semplice, perché lo stato del processo deve migrare da un processore all'altro. Ciò comporta lo spostamento dello spazio degli indirizzi, dello stato dei registri della CPU e dei file aperti dal sistema di origine a quello di destinazione. Tuttavia, per garantire la compatibilità è importante che le copie del sistema operativo in esecuzione sui diversi sistemi siano identiche. Se il sistema operativo è lo stesso, ma sui diversi sistemi sono in esecuzione versioni differenti, allora i processi da migrare devono assicurarsi di seguire linee guida di programmazione coerenti tra le diverse versioni del sistema operativo.

Le applet Java forniscono un buon esempio di migrazione di processi tra sistemi operativi diversi. Per nascondere le differenze nel sistema sottostante, il processo da migrare (cioè l'applet Java) viene eseguito su una macchina virtuale piuttosto che su un sistema operativo specifico. L'unica richiesta è che la macchina virtuale sia in esecuzione sul sistema di destinazione.

17.6 Per costruire un sistema distribuito robusto occorre conoscere quali tipi di guasti possono verificarsi.

- a. Elencate tre tipi possibili di guasto in un sistema distribuito.
- b. Specificate quali di questi guasti si possono verificare anche su sistemi centralizzati.

Risposta:

Tre guasti comuni in un sistema distribuito sono: (1) errori nel collegamento di rete, (2) errori dell'host, (3) errori del supporto di memorizzazione. Sia (2) che (3) sono errori che potrebbero verificarsi anche in un sistema centralizzato, mentre l'errore nel collegamento di rete può avvenire solo in un sistema distribuito di rete.

17.7 È sempre di cruciale importanza sapere che un messaggio inviato è correttamente giunto a destinazione? Se la vostra risposta è sì, motivatela. Se la vostra risposta è no, fornite un esempio.

Risposta:

No. Molti programmi di raccolta di stati partono dal presupposto che un pacchetto possa non essere ricevuto dal sistema di destinazione. Questi programmi in genere trasmettono un pacchetto e assumono che almeno alcuni altri sistemi sulla loro rete riceveranno l'informazione. Per esempio, un demone su ciascun sistema potrebbe trasmettere il carico medio del sistema e il numero di utenti. Queste informazioni possono essere utilizzate per la selezione del destinatario di una migrazione di processo. Un altro esempio è un programma che determina se un sito remoto è in esecuzione ed è accessibile sulla rete. Se invia una query e non ottiene alcuna risposta sa che il sistema non può essere attualmente raggiunto.

17.8 Considerate un sistema distribuito con due siti A e B. Valutate se il sito A può distinguere fra i seguenti problemi

- a. Cade B.
- b. Cade il collegamento tra A e B.
- c. B è estremamente sovraccarico e il suo tempo di risposta è di 100 volte superiore al normale.

Quali implicazioni ha la vostra risposta sul ripristino in sistemi distribuiti?

Risposta:

Una tecnica consiste nell'invio periodico da parte di B di un messaggio "Sono attivo" rivolto ad A, ad indicare che B è ancora vivo. Se A non riceve un messaggio di "Sono attivo", può assumere che B è caduto, oppure è caduto il collegamento di rete. Si noti che un messaggio di "Sono attivo" non permette ad A di distinguere tra le tipologie di guasto. Una tecnica che permette ad A di rilevare cadute della rete consiste nell'invio da A a B del messaggio "Sei attivo" utilizzando un percorso alternativo. Se A riceve una risposta, può stabilire che in effetti il collegamento di rete è giù e che B è attivo.

Se assumiamo che A sappia che B è attivo e raggiungibile (tramite il meccanismo di "Sono attivo"), e che A sia in possesso di un certo valore N indicante un tempo di risposta normale, A può verificare il tempo di risposta di B e confrontarlo con N, per determinare se B è sovraccarico oppure no.

Grazie a queste tecniche A può scegliere un altro host -diciamo C- nel sistema se B è caduto, oppure se è irraggiungibile, o se è sovraccarico.

Capitolo 18 - Linux

Esercizi di ripasso

18.1 I moduli del kernel caricabili dinamicamente offrono flessibilità quando si aggiungono driver al sistema. Hanno anche qualche svantaggio? In quali circostanze conviene compilare il kernel in un singolo file binario e in quali conviene invece mantenerlo suddiviso in moduli? Spiegate la vostra risposta.

Risposta:

Esistono due principali inconvenienti nell'utilizzo dei moduli. Il primo è la dimensione: la gestione dei moduli consuma memoria kernel non paginabile e un kernel di base con una serie di moduli caricati consuma più memoria di un kernel equivalente con i driver compilati all'interno dell'immagine stessa del kernel. Questo problema può essere significativo su macchine con memoria fisica limitata.

Il secondo inconveniente è che i moduli possono aumentare la complessità del processo di avvio del kernel. E' difficile caricare un insieme di moduli dal disco se il driver necessario per accedere a tale disco è esso stesso un modulo che deve essere caricato. La gestione dell'avvio di un kernel con moduli può dunque richiedere lavoro supplementare da parte dell'amministratore: i moduli necessari per l'avvio devono essere posti in un'immagine ramdisk che viene caricata a fianco dell'immagine iniziale del kernel, quando il sistema viene inizializzato.

In certi casi è meglio usare un kernel modulare e in altri casi è meglio usare un kernel con i driver di periferica precollegati. Dove è importante minimizzare la dimensione del kernel, la scelta dipenderà da quanto spesso vengono utilizzati i vari driver di periferica. Se sono utilizzati costantemente, allora i moduli sono inadatti. Questo è particolarmente vero quando i driver sono necessari per il processo di avvio. D'altra parte, se alcuni driver non sono sempre necessari, allora il meccanismo dei moduli permette di caricare e scaricare tali driver su richiesta, offrendo potenzialmente un risparmio netto di memoria fisica.

Quando un kernel deve essere costruito per essere utilizzabile su una grande varietà di macchine molto diverse tra loro, la struttura modulare è chiaramente preferibile rispetto all'utilizzo di un singolo kernel con decine di driver inutili che consumano memoria. Ciò è particolarmente vero per i kernel distribuiti commercialmente, per i quali il supporto di un'ampia varietà di hardware nel modo più semplice possibile è una priorità.

Tuttavia, se un kernel è costruito per un'unica macchina la cui configurazione è nota in anticipo, allora la compilazione e l'utilizzo dei moduli può costituire un'inutile complessità. In casi come questo l'uso o meno dei moduli può anche essere solamente una questione di gusti .

18.2 Il multithreading è una tecnica di programmazione comune. Descrivete tre modi diversi per implementare i thread e confrontate questi tre metodi con il meccanismo clone() di Linux. Quando un meccanismo alternativo può essere preferibile all'impiego di cloni? Quando può valere il viceversa?

Risposta:

Le implementazioni dei thread possono essere classificate in due gruppi: thread del kernel e thread in modalità utente. I pacchetti thread in modalità utente sfruttano il supporto del kernel – ad esempio, possono richiedere l'utilizzo di interrupt dei timer -, ma lo scheduling dei thread non viene effettuato dal kernel, bensì da una libreria in modalità utente . In questa implementazione un insieme di thread appare al sistema operativo come un unico contesto di esecuzione. Quando il processo multithread è in esecuzione, decide autonomamente quale thread eseguire, utilizzando salti non locali per spostarsi tra i thread secondo le proprie regole di pianificazione, con o senza prelazione.

In alternativa, il supporto per i thread può essere fornito direttamente dal kernel del sistema operativo. In questo caso i thread possono essere implementati come processi separati che condividono totalmente o parzialmente uno spazio di indirizzamento comune, oppure possono essere implementati come contesti di esecuzione separati all'interno di un singolo processo. Comunque vengano organizzati i thread, questi appaiono all'applicazione come contesti di esecuzione totalmente indipendenti.

Sono anche possibili implementazioni ibride in cui viene messo a disposizione dell'applicazione un gran numero di thread utilizzando un minor numero di thread del kernel. I thread utente eseguibili vengono eseguiti dal primo thread del kernel disponibile.

In Linux i thread sono implementati all'interno del kernel da un meccanismo di clonazione (clone) che crea un nuovo processo nello stesso spazio di indirizzamento virtuale del processo padre. A differenza di alcuni pacchetti di thread del kernel, il kernel Linux non fa alcuna distinzione tra thread e processi: un thread è semplicemente un processo che non ha creato un nuovo spazio di indirizzamento virtuale quando è stato inizializzato.

I principali vantaggi dell'implementazione dei thread nel kernel piuttosto che in una libreria in modalità utente sono:

- i sistemi che utilizzano thread del kernel possono trarre vantaggio dalla presenza di più processori, e
- se si blocca un thread in una routine di servizio del kernel (per esempio, una chiamata di sistema o un errore di pagina), gli altri thread possono proseguire la loro esecuzione.

Un vantaggio minore è la possibilità di assegnare attributi di sicurezza diversi per ogni thread.

Le implementazioni in modalità utente non hanno questi vantaggi. Poiché tali implementazioni vengono eseguite all'interno di un unico contesto di esecuzione del kernel, solo un thread alla volta può essere in esecuzione, anche se sono disponibili più CPU. Per lo stesso motivo, se un thread entra in una chiamata di sistema, nessun altro thread può essere eseguito fino al completamento di quella. Un thread che sta eseguendo una lettura bloccante su disco terrà dunque in sospeso ogni altro thread dell'applicazione.

Tuttavia, anche le implementazioni in modalità utente hanno i loro vantaggi. Il vantaggio più ovvio sono le prestazioni: l'invocazione dello scheduler del kernel per spostarsi tra i thread comporta l'ingresso in un nuovo dominio di protezione, perché la CPU passa alla modalità kernel, mentre la commutazione tra i thread in modalità utente può essere realizzata semplicemente salvando e ripristinando i principali registri della CPU. I thread in modalità utente possono anche consumare meno memoria di sistema: la maggior parte dei sistemi UNIX riservano almeno una pagina intera nello stack del kernel per ogni thread del kernel, e questo stack non può essere paginato.

L'approccio ibrido, che implementa più thread utente su un numero ridotto di thread del kernel, permette di raggiungere un compromesso. I thread del kernel permettono a più thread di essere eseguiti contemporaneamente in caso di chiamate del kernel bloccanti e consentono l'esecuzione su più CPU, mentre all'interno di ogni thread kernel potrà avvenire una commutazione tra i thread utente, più leggera, senza un eccessivo sovraccarico dovuto alla presenza di troppi thread del kernel. Il lato negativo di questo approccio è la complessità: il controllo di questo compromesso complica l'interfaccia utente della libreria dei thread.

18.3 Il kernel di Linux non permette la paginazione della memoria del kernel. Come influisce questa restrizione sul progetto del kernel? Elencate due vantaggi e due svantaggi di questa scelta di progetto.

Risposta:

L'effetto principale del non permettere la paginazione della memoria kernel in Linux è la conservazione della non prelazionabilità del kernel. Ogni processo che incorre in un errore di pagina, sia in modalità kernel che in modalità utente, rischia di essere ripianificato mentre i dati richiesti vengono paginati dal disco.

Poiché il kernel può contare sul fatto di non essere ripianificato durante l'accesso alle sue strutture dati primarie, i requisiti per proteggere l'integrità di tali strutture dati ne risultano molto semplificati. Oltre ad essere un vantaggio in sé, la semplicità fornisce anche un importante vantaggio in termini di prestazioni su macchine monoprocessoressi, grazie al fatto che non è necessaria un'attività di locking supplementare sulla maggior parte delle strutture dati interne.

Vi sono tuttavia un certo numero di svantaggi nella mancanza di memoria paginabile del kernel.

Innanzitutto, si impongono vincoli sulla quantità di memoria che il kernel può utilizzare. È irragionevole mantenere strutture dati molto grandi nella memoria non paginabile, dal momento che questa memoria fisica non può assolutamente essere utilizzata per altri scopi. Ciò ha due conseguenze: prima di tutto, il kernel deve sfoltire molte delle sue strutture dati interne manualmente, invece di poter contare su un unico meccanismo di memoria virtuale per mantenere sotto controllo l'utilizzo della memoria fisica. In

secondo luogo, si rende inattuabile l'implementazione di alcune caratteristiche che richiedono grandi quantità di memoria virtuale del kernel, come il /tmp-filesystem (un veloce file system virtuale basato sulla memoria che si trova su alcuni sistemi UNIX).

Si noti che la complessità della gestione degli errori di pagina durante l'esecuzione del codice del kernel non è rilevante in questo contesto. Il codice del kernel di Linux è già in grado di affrontare gli errori di pagina: deve essere in grado di gestire le chiamate di sistema i cui argomenti fanno riferimento alla memoria utente che può essere paginata su disco.

18.4 Discutete tre vantaggi del collegamento dinamico (condiviso) di librerie rispetto al collegamento statico. Descrivete due casi in cui sia preferibile il collegamento statico.

Risposta:

I vantaggi principali delle librerie condivise sono la riduzione della memoria e dello spazio su disco utilizzati da un sistema, e il miglioramento della manutenibilità.

Quando le librerie condivise vengono utilizzate da tutti i programmi in esecuzione, vi è una sola istanza di ogni routine di libreria di sistema sul disco e al massimo una sola istanza nella memoria fisica. Quando la libreria in questione viene utilizzata da molte applicazioni e programmi il risparmio di disco e memoria può essere abbastanza consistente. Inoltre, il tempo di avvio per l'esecuzione di nuovi programmi può essere ridotto, poiché molte delle funzioni comuni necessarie a tale programma possono essere già caricate nella memoria fisica.

Anche la manutenibilità costituisce un importante vantaggio del collegamento dinamico rispetto al collegamento statico. Se tutti i programmi in esecuzione utilizzano una libreria condivisa per accedere alle proprie routine di libreria di sistema, l'aggiornamento di queste procedure, per aggiungere nuove funzionalità o per correggere i bug, può essere fatto semplicemente sostituendo la libreria condivisa in questione. Non c'è bisogno di ricompilare o ricollegare tutte le applicazioni; tutti i programmi caricati dopo che l'aggiornamento è completato rileveranno automaticamente le nuove versioni delle librerie.

Vi sono anche ulteriori vantaggi. Un programma che utilizza librerie condivise può spesso essere adattato a scopi specifici semplicemente sostituendo una o più librerie, o anche (se il sistema lo permette, come nel caso della maggior parte dei sistemi UNIX, Linux compreso) aggiungendone una nuova a run-time. Ad esempio, una libreria di debug può sostituire la normale libreria per rintracciare un problema in un'applicazione. Le librerie condivise permettono anche ai programmi binari di collegare librerie commerciali proprietarie senza comprendere di fatto nessuna porzione di codice nel file eseguibile finale del programma. Ciò è importante perché sulla maggior parte dei sistemi UNIX molte delle librerie condivise standard sono proprietarie e le questioni di licenza possono vietare di includere il codice nei file eseguibili da distribuire a terzi.

In alcuni casi, tuttavia, è opportuno utilizzare il collegamento statico. Un esempio è costituito dagli ambienti di ripristino per gli amministratori di sistema. Se un amministratore di sistema fa un errore durante l'installazione di eventuali nuove librerie, o se l'hardware dà problemi, è possibile che le librerie condivise esistenti diventino inutilizzabili. Per questo motivo, un insieme di base di utilità di ripristino sono spesso collegate staticamente, in modo che ci sia la possibilità di correggere l'errore senza dover fare affidamento su librerie condivise funzionanti correttamente.

Vi sono anche vantaggi prestazionali che a volte rendono il collegamento statico preferibile in casi particolari. Innanzitutto, il collegamento dinamico fa aumentare il tempo di avvio di un programma, perché il collegamento deve avvenire in fase di esecuzione, piuttosto che in fase di compilazione. Il collegamento dinamico può, a volte, anche aumentare la dimensione massima del working set di un programma (il numero totale di pagine fisiche di memoria richieste per eseguire il programma). In una libreria condivisa, l'utente non ha alcun controllo sulla posizione in cui risiedono le varie funzioni nel file binario. Poiché la maggior parte delle funzioni non riempiono esattamente una o più pagine intere della libreria, il caricamento di una funzione provocherà solitamente il caricamento di alcune porzioni delle funzioni limitrofe. Con il collegamento statico, nessuna funzione alla quale non si fa riferimento (direttamente o indirettamente) sarà caricata in memoria.

Altre questioni riguardanti il collegamento statico riguardano la facilità di distribuzione: è più facile distribuire un file eseguibile con collegamento statico che con collegamento dinamico quando il distributore non è sicuro che il destinatario abbia installato in anticipo le librerie corrette. Ci possono inoltre essere alcune restrizioni commerciali nei confronti della ridistribuzione di alcuni binari come le librerie condivise. Ad esempio, la licenza per l'ambiente grafico "Motif" di UNIX consente ai binari che utilizzano Motif di essere distribuiti liberamente, purché con un collegamento statico, ma le librerie condivise non possono essere utilizzate senza licenza.

18.5 Confrontate l'utilizzo di socket di rete con l'utilizzo di memoria condivisa come meccanismi di comunicazione di dati tra processi in uno stesso computer. Quali sono i vantaggi di ciascun metodo? Quando uno è preferibile rispetto all'altro?

Risposta:

L'utilizzo di socket di rete piuttosto che di memoria condivisa per la comunicazione locale ha una serie di vantaggi. Il vantaggio principale è che l'interfaccia di programmazione per le socket è dotata di un ricco insieme di funzionalità di sincronizzazione. Un processo può facilmente determinare quando nuovi dati arrivano su una connessione socket, la quantità di dati presente e il mittente. I processi possono rimanere bloccati fino a quando arrivano nuovi dati su un socket, oppure possono richiedere l'invio di un segnale all'arrivo dei dati. Un socket gestisce anche connessioni separate. Un processo con un socket aperto in ricezione può accettare più connessioni a quel socket e verrà avvisato quando nuovi processi tentano di connettersi o quando vecchi processi lasciano cadere le loro connessioni.

La memoria condivisa non offre nessuna di queste caratteristiche. Per determinare se un altro processo ha inserito o modificato dati nella memoria condivisa un processo può soltanto andare a verificare il contenuto di tale memoria. Non è possibile fare in modo che un processo si blocchi in attesa e richieda un segnale di risveglio quando viene consegnata la memoria condivisa, e non vi è alcun meccanismo standard per permettere ad altri processi di stabilire un collegamento di memoria condivisa ad un processo esistente. Tuttavia, la memoria condivisa ha il vantaggio di essere spesso molto più veloce rispetto alla comunicazione mediante socket. Quando i dati vengono inviati tramite un socket, vengono solitamente copiati diverse volte da memoria a memoria. Gli aggiornamenti della memoria condivisa non richiedono copie dei dati: se un processo aggiorna una struttura dati nella memoria condivisa, l'aggiornamento è immediatamente visibile a tutti gli altri processi che condividono quella memoria. L'invio o la ricezione di dati su un socket richiede una chiamata di servizio di sistema del kernel per avviare il trasferimento, mentre la comunicazione mediante memoria condivisa può essere eseguita interamente in modalità utente, senza un trasferimento del controllo al kernel.

La comunicazione mediante socket è tipicamente preferibile quando è importante poter gestire la comunicazione o quando è richiesto di sincronizzare il mittente e il destinatario. Ad esempio, i processi server istanziano di solito un socket in ascolto a cui i client possono collegarsi quando vogliono utilizzare il servizio. Una volta che il socket è attivo, le singole richieste sono inviate tramite il socket, in modo che il server possa facilmente determinare quando una nuova richiesta arriva e il mittente della richiesta. In alcuni casi, tuttavia, è preferibile la memoria condivisa. La memoria condivisa è spesso la migliore soluzione quando deve essere trasferita una grande quantità di dati o quando due processi necessitano un accesso casuale a un grande insieme di dati comuni. In questo caso, però, i processi comunicanti possono aver bisogno di un meccanismo supplementare, oltre alla memoria condivisa, per ottenere la sincronizzazione. Il sistema X Window, un ambiente grafico di visualizzazione per UNIX, è un buon esempio: la maggior parte delle richieste grafiche vengono inviate su socket, ma viene offerta la memoria condivisa come meccanismo aggiuntivo da utilizzare in casi particolari in cui devono essere visualizzate sullo schermo bitmap di grandi dimensioni. In questo caso, una richiesta per visualizzare la bitmap verrà inviata su socket, ma i dati della bitmap saranno inviati tramite memoria condivisa.

18.6 Una volta i sistemi UNIX utilizzavano ottimizzazioni dell'organizzazione del disco basate sulla posizione dei dati rispetto alla rotazione del disco. Le moderne implementazioni, incluso Linux, effettuano invece un'ottimizzazione mirata semplicemente all'accesso sequenziale. Perché è stata fatta questa scelta? Da

quali caratteristiche hardware trae vantaggio l'accesso sequenziale? Perché l'ottimizzazione basata sulla rotazione non è più così utile?

Risposta:

Le caratteristiche hardware del disco sono molto cambiate in questi ultimi anni. In particolare, sono stati introdotti molti miglioramenti per aumentare la larghezza di banda massima che può essere realizzata su un disco. In un sistema moderno può essere presente una lunga pipeline tra il sistema operativo e la testina di lettura-scrittura del disco. Una richiesta di I/O deve passare attraverso il controller del disco locale del computer, sul bus fino al disco stesso, e poi internamente al disco, dove è probabile che vi sia un controller complesso in grado di memorizzare nella cache i dati cui si accede e, potenzialmente, di ottimizzare l'ordine delle richieste di I/O.

A causa di questa complessità il tempo necessario per esaudire una richiesta di I/O e perché la richiesta successiva sia generata e ricevuta dal disco può superare di molto la quantità di tempo che trascorre tra il passaggio della testina su un settore del disco e l'arrivo dell'header del settore successivo. Al fine di poter efficacemente leggere più settori contemporaneamente, i dischi utilizzano una cache readahead. Mentre un settore viene passato al computer host, il disco è impegnato a leggere i prossimi settori in previsione di una richiesta di lettura. Se le richieste di lettura cominciano ad arrivare in un ordine che rompe questa pipeline readahead, le prestazioni peggiorano. Si avranno dunque benefici prestazionali se il sistema operativo cerca di tenere le richieste di I/O in stretto ordine sequenziale.

Una seconda caratteristica dei dischi moderni è la complessità della loro geometria. Il numero di settori per cilindro può variare a seconda della posizione del cilindro: possono essere memorizzati più dati nelle tracce più vicine al bordo del disco rispetto a quelle più vicine al centro del disco. Un sistema operativo, per ottimizzare la posizione rotazionale dei dati su questi dischi, dovrebbe avere una completa comprensione di questa geometria, così come delle caratteristiche di temporizzazione del disco e del suo controller. In generale, solo la logica interna del disco può determinare la pianificazione ottimale dell'I/O e la geometria del disco smorza probabilmente qualsiasi tentativo da parte del sistema operativo di eseguire ottimizzazioni di rotazione.

Capitolo 19 – Windows 7

Esercizi di ripasso

19.1 Che tipo di sistema operativo è Windows? Descrivete due delle sue caratteristiche principali.

Risposta:

Windows è un sistema operativo a 32/64 bit, multitasking con prelazione, con il supporto di più utenti.

(1) La capacità di riparare automaticamente i problemi delle applicazioni e del sistema operativo.

(2) Un miglior supporto alla rete e ai dispositivi (tra cui il supporto alla fotografia e al video digitale).

19.2 Elencate gli obiettivi di progetto di Windows. Descrivetene due in dettaglio.

Risposta:

Tra gli obiettivi di progettazione di Windows vi sono la sicurezza, l'affidabilità, la compatibilità delle applicazioni Windows e POSIX, le alte prestazioni, l'estensibilità, la portabilità e il supporto internazionale.

(1) L'affidabilità è stata percepita come un requisito indispensabile e comprende una vasta verifica dei driver, strutture per la cattura di errori di programmazione nel codice a livello utente ed un rigoroso processo di certificazione per i driver, le applicazioni e i dispositivi di terze parti.

(2) Il raggiungimento di elevate prestazioni ha richiesto lo studio delle problematiche del passato, come le prestazioni di I/O, i colli di bottiglia della CPU del server e la scalabilità degli ambienti multithreading e multiprocessore.

19.3 Descrivete il processo di avvio di un sistema Windows.

Risposta:

(1) All'avvio dell'hardware il BIOS inizia la sua esecuzione da ROM e carica ed esegue il bootstrap loader dal disco. (2) Il programma NTLDR viene caricato dalla directory root del dispositivo di sistema identificato e determina quale dispositivo di avvio contiene il sistema operativo. (3) NTLDR carica la libreria HAL, il kernel e l'hive di sistema. L'hive di sistema indica i driver di boot richiesti e li carica. (4) L'esecuzione del kernel inizia con l'inizializzazione del sistema e la creazione di due processi: il processo di sistema che contiene tutti i thread di lavoro interni e il primo processo di inizializzazione in modalità utente: SMSS. (5) SMSS inizializza ulteriormente il sistema istanziando i file di paging e caricando i driver di periferica. (6) SMSS crea due processi: WINLOGON, che carica il resto del sistema, e CSRSS (il processo del sottosistema Win32).

19.4 Descrivete i tre livelli principali dell'architettura del kernel di Windows.

Risposta:

a. L'HAL (Hardware Abstraction Layer) crea la portabilità del sistema operativo nascondendo le differenze di hardware agli strati superiori del sistema. I dettagli di amministrazione delle funzionalità di basso livello sono forniti dalle interfacce HAL. HAL presenta un'interfaccia di macchina virtuale che viene utilizzata dal dispatcher del kernel, dal livello esecutivo e dai driver di periferica.

b. Il livello kernel fornisce una base per le funzioni esecutive e per i sottosistemi in modalità utente. Il kernel rimane in memoria e non viene mai interrotto. Le sue responsabilità sono lo scheduling dei thread, la gestione di interrupt e eccezioni, la sincronizzazione del processore a basso livello e il ripristino da interruzioni di corrente.

c. Il livello esecutivo fornisce un insieme di servizi utilizzati da tutti i sottosistemi: il gestore degli oggetti, il gestore della memoria virtuale, il gestore dei processi, le chiamate di procedura locale, il gestore dell'I/O, il controllo della sicurezza, il gestore del plug-and-play, i registri e l'avvio.

19.5 Che lavoro svolge il gestore degli oggetti?

Risposta:

Gli oggetti presentano un insieme generico di interfacce in modalità kernel ai programmi in modalità utente. Gli oggetti sono manipolati dal gestore degli oggetti del livello esecutivo. Il compito del gestore degli oggetti è quello di sorvegliare l'assegnazione e l'uso di tutti gli oggetti gestiti.

19.6 Che tipi di servizio offre il gestore dei processi?

Risposta:

Il gestore dei processi fornisce servizi per la creazione, l'eliminazione e l'utilizzo di processi, thread e job. Il gestore dei processi implementa anche l'accodamento e la consegna delle chiamate asincrone di procedura ai thread.

19.7 Che cos'è una chiamata di procedura locale?

Risposta:

La chiamata di procedura locale (LPC) è un sistema di scambio di messaggi. Il sistema operativo utilizza la LPC per trasferire le richieste e i risultati tra i processi client e server all'interno di una singola macchina, in particolare tra i sottosistemi Windows 7.

19.8 Quali sono le responsabilità del gestore dell'I/O?

Risposta:

Il gestore dell' I/O è responsabile del file system, dei driver di periferica e dei driver di rete. Il gestore di I/O tiene traccia di quali driver di periferica, driver di filtro e file system vengono caricati e gestisce i buffer per le richieste di I/O. Il gestore assiste, inoltre, la fornitura dei file di I/O mappati in memoria e controlla il gestore della cache per l'intero sistema di I/O.

19.9 Quali tipologie di networking sono supportate da Windows? Come sono implementati i protocolli di trasporto da Windows? Descrivete due protocolli di rete.

Risposta:

E' fornito il supporto sia per reti peer-to-peer che per reti client-server. I protocolli di trasporto sono implementati come driver. (1) Il pacchetto TCP/IP comprende SNMP, DHCP, WINS e NetBIOS. (2) Viene offerto il protocollo di tunneling punto-a-punto per la comunicazione tra i moduli ad accesso remoto in esecuzione sui server Windows 7 e su altri sistemi client connessi attraverso Internet. Con l'utilizzo di questo schema vengono supportate le reti private virtuali (VPN) multi-protocollo su Internet.

19.10 Descrivete l'organizzazione dello spazio dei nomi di NTFS.

Risposta:

Lo spazio dei nomi di NTFS è organizzato come una gerarchia di directory in cui ogni directory utilizza una struttura dati ad albero B+ per memorizzare un indice dei nomi dei file della directory. La radice indice di una directory contiene il livello superiore del B+ albero. Ogni voce della directory contiene il nome e il riferimento del file, così come il timestamp di aggiornamento e la dimensione del file.

19.11 Come tratta le strutture dati NTFS? Come viene ripristinato NTFS dopo un crash di sistema? Che cosa viene garantito dopo che è avvenuto un ripristino?

Risposta:

In NTFS tutti gli aggiornamenti delle strutture dati del file system vengono eseguiti all'interno di transazioni. Prima che una struttura di dati venga modificata, la transazione scrive un record nel log contenente le informazioni per l'annullamento e il ripristino. Un record di commit viene scritto nel log dopo che una transazione è riuscita. In caso di crash il file system può essere ripristinato ad uno stato consistente elaborando i record di log, in primo luogo ripetendo le operazioni per le transazioni eseguite, poi annullando le operazioni per le transazioni che non sono state eseguite correttamente. Questo schema non garantisce che il contenuto dei file dell'utente sia corretto dopo il ripristino, ma garantisce piuttosto che le strutture dati del file system (metadati dei file) non siano danneggiate e riflettano uno stato consistente che esisteva prima del crash.

19.12 Come alloca la memoria utente Windows?

Risposta:

La memoria utente può essere allocata seguendo diversi schemi: memoria virtuale, file mappati in memoria, heap e archiviazione locale dei thread.

19.13 Descrivete alcuni dei modi in cui un'applicazione utilizza la memoria per mezzo della API Win32.

Risposta:

- a. La memoria virtuale fornisce diverse funzioni che consentono ad un'applicazione di riservare e rilasciare memoria, specificando l'indirizzo virtuale in cui la memoria viene allocata.
- b. Un file può essere mappato in memoria nello spazio degli indirizzi, fornendo così uno strumento per la condivisione della memoria tra due processi.
- c. Quando viene inizializzato un processo Win32, viene dotato di uno heap di default. Possono essere creati degli heap privati che forniscono regioni di spazio di indirizzamento riservato per le applicazioni. Sono fornite funzionalità di gestione dei thread per allocare e controllare l'accesso dei thread a heap privati.
- d. Un meccanismo di archiviazione locale dei thread permette ai dati globali e statici di funzionare correttamente in un ambiente con multithreading. L'archiviazione bloccata dai thread alloca memoria globale per ogni thread.