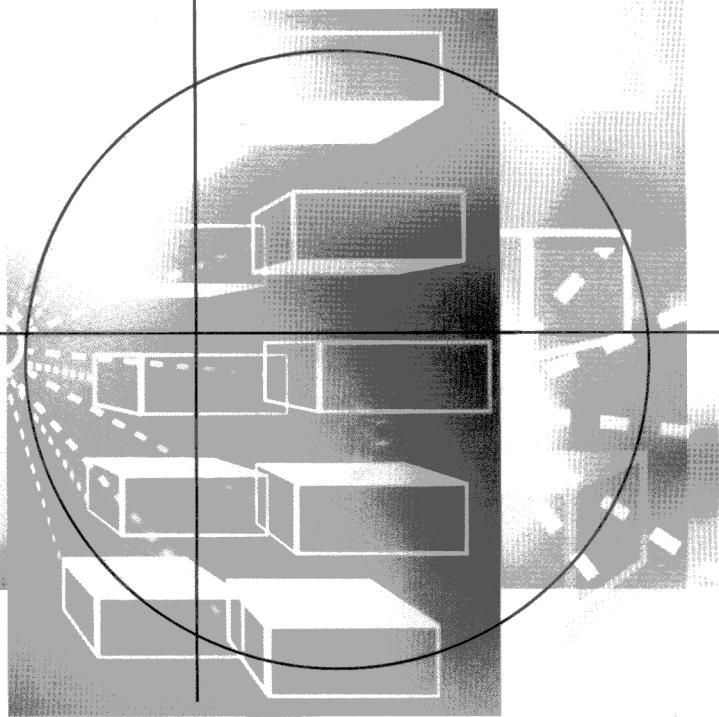


Paolo Ancilotti
Maurelio Boari
Anna Ciampolini
Giuseppe Lipari



Sistemi operativi

McGraw-Hill

Indice

Prefazione

XI

1 Concetti introduttivi

1

1.1	Principali funzioni di un sistema operativo	1
1.1.1	Facilità di programmazione e portabilità dei programmi	3
1.1.2	Gestione delle risorse	4
1.1.3	Protezione, sicurezza, tolleranza ai guasti	5
1.1.4	Astrazione di macchina virtuale	6
1.2	Cenni storici sull'evoluzione dei sistemi operativi	7
1.2.1	I primi sistemi di elaborazione	7
1.2.2	I primi sistemi batch	8
1.2.3	Sistemi batch multiprogrammati	9
1.2.4	Sistemi time-sharing	14
1.2.5	Sistemi in "tempo reale"	16
1.2.6	Sistemi operativi per personal computer	18
1.2.7	Sistemi operativi per sistemi paralleli e distribuiti	19
1.3	Richiami di architetture dei sistemi di elaborazione	20
1.3.1	Il processore	21
1.3.2	La memoria	22
1.3.3	Dispositivi d'ingresso/uscita	25
1.3.4	Meccanismo d'interruzione	26
1.3.5	Accessi diretti alla memoria (DMA)	30
1.3.6	Meccanismi di protezione	32
1.4	Struttura dei sistemi operativi	34
1.4.1	Principali componenti di un sistema operativo	34
1.4.2	Principali modelli strutturali	39
1.5	Sintassi utilizzata nel testo	44
1.6	Sommario	45
1.7	Note bibliografiche	46

2 Gestione dei processi

49

2.1	Definizione di processo	50
2.2	Stati di un processo	51

2.3	Descrittore di un processo	54
2.4	Code di processi	55
2.5	Cambio di contesto	57
2.6	Creazione e terminazione dei processi	57
2.7	Interazione tra i processi	58
2.8	Nucleo di un sistema a processi	60
2.9	Scheduling	63
2.10	Processi leggeri (threads)	67
2.11	Sommario	70
2.12	Note bibliografiche	70
3	Sincronizzazione dei processi	73
3.1	Tipi di interazione tra i processi	73
3.2	Problema della mutua esclusione	77
3.2.1	Soluzioni al problema della mutua esclusione	78
3.3	Problema della comunicazione	80
3.4	Semafori	82
3.4.1	Soluzione al problema della mutua esclusione	82
3.4.2	Soluzione al problema della comunicazione	84
3.5	Primitive send e receive	86
3.6	Soluzione al problema di comunicazione tra processi	88
3.6.1	Sincronizzazione tra processi comunicanti	90
3.7	Blocco critico	91
3.7.1	Condizioni per il blocco critico	96
3.7.2	Metodi per il trattamento del blocco critico	97
3.7.3	Individuazione di eventuali blocchi critici e successivo ripristino	99
3.8	Sommario	100
3.9	Note bibliografiche	101
4	Gestione della memoria	103
4.1	Introduzione alla gestione della memoria	103
4.1.1	Analogie con la gestione della CPU	104
4.1.2	Differenze rispetto alla gestione della CPU	105
4.2	Aspetti caratterizzanti la gestione della memoria	106
4.2.1	La memoria virtuale	107
4.2.2	Rilocazione statica e dinamica - Memory Management Unit	109
4.2.3	Organizzazione della memoria virtuale	115
4.2.4	Allocazione della memoria fisica	117
4.2.5	Dimensionamento della memoria virtuale	120
4.3	Tecniche di gestione della memoria	121
4.3.1	Memoria partizionata	122
4.3.2	Memoria segmentata	128
4.3.3	Memoria paginata	137
4.3.4	Memoria segmentata e paginata	149

4.3.5	Gestione degli spazi virtuali	150
4.4	Sommario	154
4.5	Note bibliografiche	154
5	Gestione delle periferiche (I/O)	155
5.1	Concetti generali	156
5.2	Organizzazione logica del sottosistema di I/O	159
5.2.1	Livello indipendente dai dispositivi	159
5.2.2	Livello dipendente dai dispositivi	165
5.3	Gestore di un dispositivo	167
5.3.1	Processi esterni	169
5.3.2	Gestione di un dispositivo a controllo di programma	170
5.3.3	Gestione di un dispositivo a interruzione	171
5.3.4	Gestione di un dispositivo in DMA	177
5.3.5	Flusso di controllo durante un trasferimento	178
5.3.6	Gestione del temporizzatore	179
5.4	Gestione e organizzazione dei dischi	181
5.4.1	Organizzazione fisica dei dischi	181
5.4.2	Criteri di ordinamento dei dati su disco e di scheduling delle richieste di trasferimento	184
5.4.3	Dischi RAID	187
5.5	Sommario	190
5.6	Note bibliografiche	190
6	Il File System	193
6.1	Organizzazione del file system	193
6.2	La struttura logica del file system	195
6.2.1	Il file	195
6.2.2	Il directory	196
6.2.3	Gestione della struttura logica del file system	198
6.3	Accesso al file system	199
6.3.1	Strutture dati e operazioni per l'accesso ai file	199
6.3.2	Metodi di accesso	201
6.3.3	Protezione di file e directory	203
6.4	Organizzazione fisica	206
6.4.1	Tecniche di allocazione dei file	206
6.5	Sommario	212
6.6	Note bibliografiche	213
7	I sistemi operativi Unix e Linux	215
7.1	Introduzione a Unix	215
7.1.1	La storia di Unix	215
7.1.2	Linux	216
7.2	Architettura del sistema operativo UNIX	217
7.2.1	Organizzazione	217

7.3	Interazione con l'utente	218
7.3.1	Lo shell	219
7.4	I processi nel sistema operativo Unix	220
7.4.1	Caratteristiche	220
7.4.2	Immagine di un processo Unix	222
7.4.3	System Call per la gestione di processi	224
7.4.4	Lo scheduling in Unix	229
7.5	La gestione della memoria nel sistema Unix	229
7.6	Il file system	230
7.6.1	Organizzazione logica del File System di Unix	230
7.6.2	Protezione	232
7.6.3	Organizzazione fisica del File System di Unix	234
7.6.4	Strutture dati del kernel per l'accesso a file	236
7.6.5	System Call per l'accesso a file	239
7.7	Interazione tra processi	241
7.7.1	Sincronizzazione: i segnali	242
7.7.2	System Call per l'uso dei segnali	243
7.7.3	Comunicazione: pipe	246
7.8	I Threads nel sistema Linux	248
7.8.1	Gestione di thread secondo lo standard POSIX: la libreria pthreads	248
7.8.2	Sincronizzazione tra thread Linux	250
7.8.3	Un esempio di sincronizzazione tra thread	256
7.9	Sommario	259
7.10	Note bibliografiche	260
8	Il Sistema Operativo Windows	261
8.1	Struttura generale	261
8.1.1	Evoluzione storica	261
8.1.2	Sottosistemi e moduli del nucleo	262
8.1.3	Struttura a microkernel	265
8.1.4	Kernel Objects	266
8.1.5	Sicurezza e controllo degli accessi	266
8.2	Gestione dei processi e dei thread	269
8.2.1	Lo Schedulatore	271
8.3	Sincronizzazione fra thread	274
8.3.1	La famiglia di funzioni Interlocked	274
8.3.2	Oggetti di sincronizzazione	276
8.4	Gestione della memoria virtuale	282
8.5	File system	283
8.6	Sommario	285
8.7	Note bibliografiche	285
A	Elementi di sincronizzazione in ambiente distribuito	287
A.1	Introduzione alle reti	287

A.1.1	Il modello OSI e la rete Internet	288
A.2	Sviluppo delle applicazioni di rete	291
A.2.1	Il modello cliente-servitore	292
A.2.2	Le socket di Unix BSD	292
A.2.3	Strutture dati associate alle socket	296
A.2.4	System Call per l'uso delle socket	297
A.3	Un esempio: esecuzione remota di comandi	304
A.4	Sommario	307
A.5	Note bibliografiche	307
B	Multithreading in Java	309
B.1	Ambiente di sviluppo e ambiente di esecuzione	310
B.2	I threads in Java	311
B.2.1	Creazione di threads mediante estensione della classe Thread	312
B.2.2	Creazione di threads mediante implementazione dell'interfaccia Runnable	313
B.2.3	Grafo di stato dei threads Java	314
B.2.4	Priorità e algoritmi di scheduling dei threads Java	315
B.3	Sincronizzazione in Java	316
B.3.1	Accessi esclusivi a un oggetto e sezioni critiche	317
B.3.2	Sincronizzazione diretta: metodi wait e notify	320
B.4	Sommario	325
B.5	Note bibliografiche	325
Bibliografia	327	
Indice analitico	333	

1

Concetti introduttivi

In questo primo capitolo vengono presentate le principali funzioni che il sistema operativo svolge nell'ambito di un sistema di elaborazione. Viene anche presentata una breve panoramica storica sull'evoluzione dei sistemi operativi per meglio comprendere i principi che stanno alla base del loro progetto e le motivazioni che hanno portato al loro sviluppo. Vengono, quindi, passate in rassegna le varie tipologie di sistemi operativi, distinguendoli in base alle finalità per cui sono stati progettati.

Per affrontare le problematiche relative allo sviluppo delle componenti di un sistema operativo è necessario conoscere la struttura di un sistema di elaborazione e, in particolare, le componenti hardware con le quali il sistema operativo deve interfacciarsi. Per questo motivo vengono brevemente richiamate quelle componenti della struttura fisica di un sistema di elaborazione a cui verrà fatto riferimento nei capitoli successivi del testo.

Vengono quindi presentate le principali strutture interne di un sistema operativo, le componenti che lo caratterizzano e il modo in cui tali componenti sono organizzate.

Infine, viene sinteticamente presentata la sintassi di uno pseudo-linguaggio di programmazione che, nei successivi capitoli del testo, verrà utilizzato per descrivere gli algoritmi e presentare gli esempi.

1.1 Principali funzioni di un sistema operativo

Un sistema operativo è un componente software di un sistema di elaborazione il cui compito principale è quello di controllare l'esecuzione dei programmi applicativi e di agire come *intermediario* tra questi e la macchina fisica (*hardware*) con lo scopo di facilitarne l'uso e, al tempo stesso, di garantire che tale uso sia effettuato in maniera efficace ed efficiente.

In modo molto schematico, potremmo rappresentare un sistema di elaborazione come un insieme di tre macro-componenti, organizzati gerarchicamente in tre livelli: il livello fisico, sul quale gira il sistema operativo che a sua volta offre le proprie

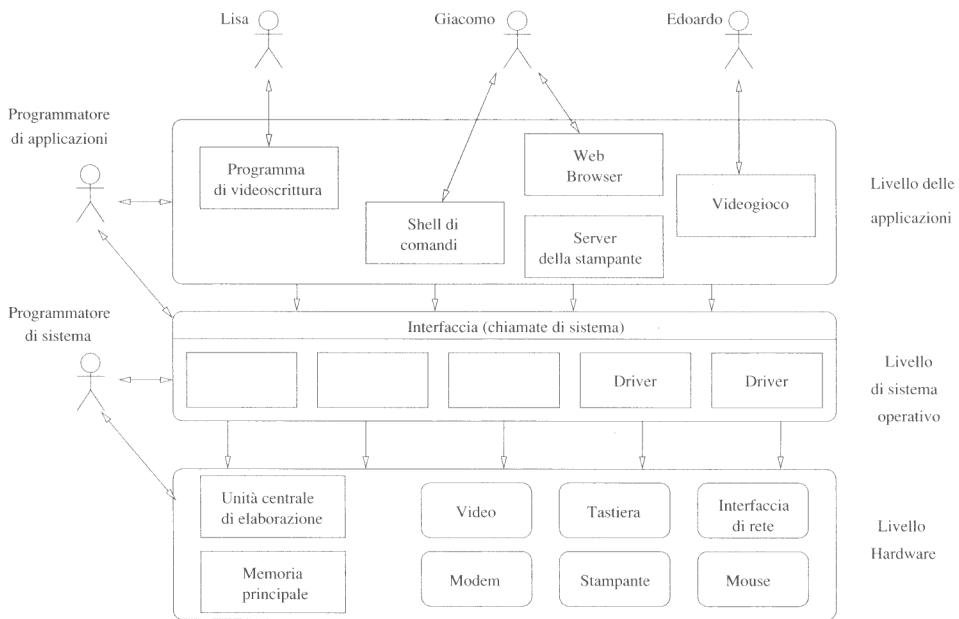


Figura 1.1 Struttura schematica di un sistema di elaborazione.

funzioni ai programmi applicativi richiesti dagli utenti. In figura 1.1 viene illustrata schematicamente la relazione fra i tre componenti e gli attori di un sistema di elaborazione:

- Il livello hardware, o della macchina fisica, è quello che corrisponde ai componenti fisici del sistema e comprende il processore, la memoria principale (volatile) e le unità periferiche, come la tastiera, il video, il mouse, le memorie di massa ecc. Lo scopo di questo livello è quello di mettere a disposizione dei programmi le risorse necessarie per la loro esecuzione. Per questo motivo, nel seguito, ci riferiremo ai vari componenti hardware con il termine di *risorse fisiche*. In questo senso il processore è una risorsa così come lo sono la memoria, la stampante ecc.
- Il livello del sistema operativo, che corrisponde a un insieme di componenti software che hanno il compito di gestire le risorse fisiche della macchina offrendo ai programmi applicativi un'interfaccia *standard* più semplice da usare rispetto a quella offerta direttamente dai componenti hardware. Tale interfaccia è composta da una serie di funzioni che possono essere invocate dai programmi applicativi per intervenire sulle componenti hardware del sistema in maniera controllata ed efficiente.
- Il livello dei programmi applicativi che, come dice il nome, corrisponde all'insieme delle applicazioni utilizzate direttamente dagli utenti del sistema. I programmi utente non interagiscono mai direttamente con l'hardware, ma sempre tramite l'interfaccia del sistema operativo. Quindi, ad esempio, se un programma utente deve visualizzare un messaggio sul video deve invocare delle opportune funzioni del sistema operativo che si occuperanno di eseguire tale compito. Questo tipo di

indirezione si rende necessaria per controllare l'accesso alle risorse della macchina fisica. A questo livello appartengono anche i componenti del *sistema di programmazione* (compilatori, caricatori, linker). Analogamente appartiene a questo livello *l'interprete dei comandi o shell*, cioè quel particolare programma tramite il quale l'utente specifica al sistema i compiti che questo deve svolgere.

In definitiva, il sistema operativo non è altro che un insieme di programmi che nascondono la macchina fisica alle applicazioni. Un programma applicativo opera sulle risorse fisiche per il tramite delle funzioni del sistema operativo il quale, quindi, fornisce ai programmi applicativi un'interfaccia costituita da un insieme di funzioni che mascherano la struttura della macchina fisica facendo vedere loro una *macchina virtuale* più semplice da usare. Tale astrazione permette al programmatore di strutturare un programma applicativo in maniera indipendente dalla maggior parte dei dettagli architetturali della macchina fisica. Programmare direttamente l'hardware è un compito complesso, tedioso e soggetto ad errori. Inoltre, tale astrazione facilita la *portabilità* dei programmi su differenti architetture hardware. Questo aspetto relativo alla facilità di programmazione e alla portabilità dei programmi rappresenta sicuramente uno dei principali obiettivi di un sistema operativo.

Il fatto che l'accesso alle risorse della macchina fisica avvenga sempre tramite le funzioni del sistema operativo consente a quest'ultimo di gestire le risorse in modo tale da ottimizzarne l'uso secondo quelle che sono le finalità del sistema di elaborazione. È questa una delle funzioni più importanti svolte dal sistema operativo tanto che, in taluni casi, si tende a rappresentare il sistema operativo come un insieme di gestori di risorse.

Un'altra funzione svolta dal sistema operativo è quella di garantire la protezione e la sicurezza dei dati dei vari utenti del sistema di elaborazione. Connessa con questa c'è anche quella relativa alla gestione di eventuali malfunzionamenti che si possono verificare durante l'esecuzione di un programma. L'obiettivo, in questo caso, è quello di verificare la possibilità di recuperare il corretto funzionamento del sistema.

Queste sono solo alcune delle principali funzioni svolte da un sistema operativo che adesso cercheremo di approfondire brevemente. In realtà, non è possibile fornire un elenco esaustivo poiché tali funzioni cambiano spesso da sistema a sistema.

1.1.1 Facilità di programmazione e portabilità dei programmi

Perché un programma possa essere eseguito è necessario che allo stesso siano dedicate alcune risorse della macchina fisica. Per prima cosa è necessario caricare il programma in memoria principale e quindi è necessario che una porzione di questa gli venga dedicata. È poi necessario dedicare al programma l'uso del processore affinché questo possa eseguire le istruzioni che lo compongono. Può darsi poi che il programma durante la sua esecuzione debba operare su alcuni dispositivi d'ingresso/uscita, per esempio per leggere dei dati dal disco e per fornire i risultati sul video o stamparli sulla stampante.

Per chiarire meglio tale concetto, consideriamo ad esempio un programma utente che legge dei dati da un dispositivo di memorizzazione di massa (ad esempio il disco fisso) e li mostra a video (ad esempio il comando cat nei sistemi operativi Unix, o il comando type nei sistemi Windows).

Supponiamo inizialmente che il sistema di elaborazione non preveda alcun sistema operativo. Il programma quindi deve interagire direttamente con diverse parti della macchina fisica. Innanzitutto è necessario caricare il programma in memoria e attivare la sua esecuzione caricando nel registro “*contatore delle istruzioni*” (*program counter* o *instruction pointer*) l’indirizzo della prima istruzione del programma. Successivamente, quando il programma deve interagire con l’unità di memorizzazione di massa per leggere i dati memorizzati sul supporto magnetico, è necessario che lo stesso esegua tutta una serie di istruzioni particolari, che sono quelle previste per accedere allo specifico dispositivo. Ciò implica, per il programmatore, una conoscenza approfondita di tutti i dettagli realizzativi di ogni dispositivo con cui il suo programma dovrà operare.

Purtroppo in commercio esistono molti modelli diversi di dispositivi di memorizzazione di massa, ognuno con le proprie caratteristiche e con le proprie funzionalità. Per cui, se il programma dovesse essere in grado di funzionare anche su un altro sistema su cui è installato un diverso dispositivo, sarebbe stato necessario programmarlo in modo da riconoscere il particolare dispositivo e adattare le proprie routine di lettura dati al dispositivo in questione. Inoltre, il programma deve essere in grado di individuare i propri dati all’interno del dispositivo che può contenere anche molti dati diversi (*file*).

Lo stesso problema si ha nel caso in cui si vogliano visualizzare i dati sul dispositivo di uscita (ad esempio il terminale video del sistema). Il programmatore deve conoscere i dettagli interni del dispositivo per poter scrivere un programma in grado di interagire con esso.

Nel caso in cui il sistema di elaborazione comprenda un sistema operativo il compito del programmatore risulta estremamente più semplice poiché non è più necessario conoscere i dettagli della macchina fisica ed è possibile concentrarsi esclusivamente sulla funzionalità del programma in quanto le interazioni con l’hardware vengono demandate al sistema operativo. In pratica, ogni interazione con un particolare dispositivo fisico può essere programmata chiamando altrettante funzioni offerte dal sistema operativo. Sono queste funzioni che hanno il compito di interagire direttamente con i dispositivi e quindi di conoscere tutti i dettagli della macchina fisica.

1.1.2 Gestione delle risorse

I moderni sistemi di elaborazione sono in grado di eseguire più programmi contemporaneamente. Questo è sicuramente vero per tutti quei sistemi che offrono il loro servizio a più utenti contemporaneamente (*sistemi multiutente*). Ad esempio, nello schema di figura 1.1 Edoardo accede al sistema localmente, tramite la tastiera e il video; Lisa e Giacomo accedono al sistema da remoto, tramite un’interfaccia di rete. Oppure si pensi a un sistema di elaborazione dedicato alla gestione di una banca. In questo caso, l’accesso al sistema avviene da parte degli operatori che, tramite i terminali presenti agli sportelli della banca, possono eseguire contemporaneamente varie transazioni.

Ma l’esecuzione contemporanea di più programmi è desiderabile anche all’interno di un normale personal computer. Si pensi ad un normale PC tramite il quale si può navigare su Internet e, allo stesso tempo, ascoltare della musica, editare dei file di testo, e così via. La tecnica utilizzata per consentire l’esecuzione contemporanea

di più programmi all'interno di un unico sistema di elaborazione è nota come *tecnica di multiprogrammazione* e sarà esaminata in dettaglio nel seguito.

Ciò che è necessario evidenziare fin da ora è il fatto che, se più programmi sono contemporaneamente in esecuzione su uno stesso sistema, nasce la necessità di coordinare l'accesso alle risorse della macchina da parte dei vari programmi al fine di evitare comportamenti indesiderati. Si pensi, ad esempio, al caso di due programmi che vogliono stampare il loro output sulla stampante. È possibile che i due programmi eseguano l'operazione di stampa "contemporaneamente" generando quindi delle stampe incomprensibili. È compito del sistema operativo garantire che le due operazioni di stampa siano eseguite in sequenza.

La stessa cosa vale per tutte le altre risorse fisiche per le quali i programmi in esecuzione necessariamente competono. Se, ad esempio, due programmi sono in esecuzione ma il sistema dispone di un solo processore, è compito del sistema operativo ripartire il suo uso tra i due programmi e consentire quindi che gli stessi possano procedere, un po' l'uno un po' l'altro. Analogamente è necessario gestire la memoria fisica affinché sia possibile caricarvi i diversi programmi in esecuzione.

Anche lo spazio fisico su disco costituisce una risorsa da gestire opportunamente affinché ciascun programma possa memorizzarvi i propri file senza compromettere quelli degli altri.

Oltre alle risorse fisiche il sistema operativo deve poi gestire anche altre risorse, costituite da dati e funzioni comuni (*risorse logiche*) qualora queste possano essere utilizzate da più programmi.

Le tecniche adottate dal sistema operativo per la gestione delle risorse e le varie strategie adottate per una loro corretta allocazione ai programmi in esecuzione, variano molto da sistema a sistema. È dalla scelta di queste strategie che dipende la tipologia di applicazioni a cui il sistema è in grado di fornire un migliore servizio.

1.1.3 Protezione, sicurezza, tolleranza ai guasti

Gli aspetti relativi alla protezione e alla sicurezza sono particolarmente importanti nei sistemi multiutente. In questi, ogni utente deve essere protetto dalle possibili interferenze degli altri. Quindi, un sistema deve garantire la **riservatezza** (*privacy*) e la **protezione** delle informazioni. Questi aspetti sono diventati particolarmente critici con l'avvento delle reti di calcolatori e la possibilità di accedere a sistemi remoti collegati in rete. È ormai noto che ogni sistema collegato in rete è sempre soggetto alla possibilità di "*attacchi*" da parte di utenti "*maliziosi*" (*hacker*). I moderni sistemi dedicano meccanismi sempre più raffinati per garantire che un utente "*malizioso*" non possa accedere alle informazioni riservate degli altri utenti. Conseguenza dello sviluppo delle reti di calcolatori è anche la diffusione di programmi particolarmente nocivi (i cosiddetti "*virus*") che vengono spesso ricevuti tramite messaggi di posta elettronica e che, una volta caricati nel sistema colpito, producono danni anche molto gravi all'intero sistema di elaborazione.

Nei moderni sistemi operativi multiutente è facoltà di ogni utente decidere quali dei suoi dati rendere pubblici e quali mantenere riservati. Ad esempio, il sistema operativo Windows 2000 (e la sua evoluzione Windows XP) sono sistemi multiutente e, come tali, forniscono la protezione delle informazioni riservate e l'accesso contemporaneo a più utenti.

La protezione e la sicurezza del sistema rispetto ad intrusioni sono tra i compiti più importanti di un sistema operativo. Aziende, istituti bancari ed altri enti possono subire notevoli danni economici nel caso in cui un utente non autorizzato acceda ai dati del sistema dedicato alla loro gestione.

Anche la protezione da eventuali guasti che si possono verificare nei componenti fisici è un aspetto importante di ogni sistema operativo. Vedremo, ad esempio nel capitolo 5 riservato alla gestione dei dispositivi d'ingresso/uscita, le tecniche adottate per far fronte ai molti eventi anomali che si possono verificare durante un'operazione d'ingresso dati o un trasferimento in uscita dei risultati.

Altri eventi anomali che si possono verificare durante la vita di un sistema non sono necessariamente riconducibili a guasti. In certi casi, ad esempio come conseguenza delle tecniche di allocazione delle risorse, il sistema può generare dei malfunzionamenti. Un esempio, il *blocco critico*, sarà trattato nel capitolo 3, dove verranno illustrate le varie strategie adottate per far fronte a questi particolari tipi di malfunzionamenti.

1.1.4 Astrazione di macchina virtuale

Come già indicato precedentemente, il sistema operativo – realizzando le funzioni indicate nei precedenti paragrafi – genera una macchina virtuale sulla quale girano i programmi applicativi.

L'interfaccia di programmazione che il sistema operativo fornisce ai programmi utente viene indicata genericamente come **API** (*Application Programming Interface*) di sistema. Le operazioni offerte dall'API di un sistema operativo vengono indicate anche col termine di *chiamate di sistema* (*system calls*), a indicare che tali operazioni sono realizzate in software tramite funzioni che vengono chiamate dai programmi applicativi. Spesso sono anche indicate con il termine di *primitive* in quanto, per un programma applicativo, una chiamata di sistema deve essere vista come se fosse un'istruzione macchina anche se, in realtà, viene realizzata in software all'interno del sistema operativo. Il termine *primitiva* indica proprio che l'esecuzione di una funzione di sistema deve avvenire in modo tale che il suo comportamento sia equivalente a quello di un'istruzione macchina, cioè in forma atomica (non divisibile). Questo aspetto delle primitive e la tecnica per la loro realizzazione verranno illustrate in dettaglio nel paragrafo 3.4.1.

Come indicato in figura 1.1, il programmatore applicativo utilizza le funzioni offerte dall'API del sistema operativo per scrivere le proprie applicazioni.

È, viceversa, compito del programmatore di sistema realizzare tutte le funzioni del sistema operativo operando direttamente sulla macchina fisica (interfaccia hardware).

Lo sviluppo dei sistemi operativi ha profondamente influenzato anche lo sviluppo delle architetture dei moderni processori. Molte delle caratteristiche presenti nei moderni microprocessori derivano dalla necessità di fornire al sistema operativo supporti sempre più efficienti per l'esecuzione delle sue funzioni.

Le funzionalità offerte dalla macchina virtuale dipendono dal tipo di sistema operativo che, a sua volta, dipende dall'ambito applicativo per cui è stato progettato. Infatti, settori applicativi diversi hanno bisogno di sistemi operativi con differenti caratteristiche. Ad esempio, applicazioni di controllo per sistemi dedicati hanno neces-

sità completamente diverse dalle applicazioni transazionali e queste, a loro volta, esigenze diverse da applicazioni di calcolo intensivo come nel caso di complessi programmi scientifici. Nei paragrafi successivi verrà presentata una panoramica delle diverse categorie di sistemi operativi.

1.2 Cenni storici sull'evoluzione dei sistemi operativi

I moderni sistemi di elaborazione sono molto potenti e complicati. Di conseguenza, anche i sistemi operativi sono programmi complessi che devono svolgere molti compiti. Per comprendere meglio le funzionalità di un sistema operativo moderno, è utile fare una breve panoramica sull'evoluzione storica dei sistemi operativi.

1.2.1 I primi sistemi di elaborazione

I primi calcolatori, fino alla metà degli anni '50, erano macchine enormi ed estremamente costose, che venivano usate soprattutto in campo militare o scientifico. Essi avevano una forma ben diversa da quella attuale. Le uniche periferiche d'ingresso erano costituite dai lettori di schede perforate e di nastri perforati, mentre le periferiche di uscita erano le stampanti e i perforatori di schede. Non era previsto nessun sistema operativo e l'utente programmatore interagiva direttamente col calcolatore per il tramite della consolle della macchina. Quando sulla macchina non giravano programmi (*stand-by*), era possibile prendere visione del contenuto (in binario) dei vari registri di macchina mediante opportune file di lampadine sulla consolle. Analogamente, mediante opportuni interruttori sempre sulla consolle, era possibile impostare un valore binario ed inserirlo manualmente in un qualunque registro della macchina o in una qualunque locazione di memoria. In questo modo, era possibile per l'utente caricare un programma in memoria e, inserendo l'indirizzo della prima istruzione nel *program counter*, attivarne manualmente l'esecuzione. Un tipico programma scritto in linguaggio sorgente, ad esempio in FORTRAN, veniva preliminarmente perforato su schede. Ogni scheda conteneva un'istruzione dove ogni carattere era rappresentato mediante un codice di perforazione. Quindi veniva caricato in memoria il compilatore che, una volta attivata manualmente la sua esecuzione, produceva il programma tradotto perforando il risultato della traduzione ancora su schede. A questo punto, posizionato il programma tradotto sul lettore di schede, veniva attivato ancora manualmente il caricatore per trasferire in memoria il programma tradotto. E infine, di nuovo manualmente, ne veniva attivata l'esecuzione.

Questo tipo di comportamento, che prevedeva di continuo l'intervento manuale dell'utente, implicava un'efficienza di uso delle costose risorse di macchina paurosamente bassa, meno dell'1%. È sufficiente pensare che per eseguire un tipico lavoro (*job*) costituito dall'esecuzione in sequenza di tre programmi, il compilatore, il caricatore e il programma utente, era necessario un tempo complessivo dell'ordine di decine di minuti, quando il tempo di reale esecuzione della CPU era limitato a pochi secondi. E tutto ciò, ovviamente, in assenza di eventuali errori di programmazione. Inoltre questo tipo di comportamento prevedeva che l'utente fosse particolarmente esperto del funzionamento del sistema. L'unico reale vantaggio era costituito dal fatto che, operando direttamente sulla macchina, l'utente poteva effettuare la fase di *test*.

e *debugging* di un programma in modo molto efficiente operando direttamente tramite la consolle.

Gli unici programmi di sistema presenti su questi primi sistemi di elaborazione erano costituiti dai componenti del *sistema di programmazione* (compilatori, caricatori, debugger) e, più tardi, anche da librerie, soprattutto librerie d'ingresso/uscita, utili per semplificare l'accesso ai dispositivi senza doverne conoscere tutti i complessi dettagli realizzativi.

1.2.2 I primi sistemi batch

È per far fronte ai gravi problemi di efficienza di uso delle costose risorse di macchina, tipici dei primi sistemi di elaborazione, che nacquero, tra la fine degli anni '50 e l'inizio degli anni '60, i primi sistemi operativi.

Il primo obiettivo fu quello di ridurre l'intervento manuale dell'utente. Ciò fu ottenuto nel seguente modo:

1. I programmi componenti il sistema di programmazione furono resi disponibili su memoria di massa, all'epoca nastri magnetici.
2. L'utente-programmatore forniva il proprio programma sorgente ad un operatore, unico abilitato ad operare direttamente sulla macchina. Per abilitare l'utente a specificare di quali programmi di sistema avesse bisogno per portare a termine il suo lavoro (job) fu definito un linguaggio di controllo, detto anche *job control language* (JCL) (l'antenato dei moderni shell). Utilizzando tale linguaggio l'utente presentava all'operatore il proprio pacco di schede contenente, oltre alle schede su cui erano perforati sia il programma sorgente che i dati da elaborare, anche le cosiddette schede di controllo (nelle quali erano perforati i comandi del JCL), caratterizzate dall'avere un primo carattere particolare come, ad esempio, il carattere \$, in modo tale da poterle facilmente riconoscere dalle schede contenenti le istruzioni e i dati del programma.
3. Fu definito un nuovo programma di sistema, detto *monitor* (il primo esempio di sistema operativo), il quale era residente in una porzione della memoria ad esso riservata. Questo era un semplice programma ciclico che si limitava a leggere le schede e, per ogni scheda di controllo, ne interpretava il contenuto nel senso di caricare in memoria, prelevandolo dal nastro magnetico, il programma di sistema richiesto mediante quella scheda. Terminato il caricamento in memoria del programma richiesto, il monitor trasferiva il controllo della CPU alla sua prima istruzione tramite un'istruzione di salto. Al termine della sua esecuzione il programma così attivato invece di eseguire l'istruzione di **halt**, che avrebbe fermato la macchina, restituiva il controllo al monitor ancora tramite un salto e questo continuava con l'esame della successiva scheda di controllo e così via fino alla particolare scheda di controllo che rappresentava la fine dell'esecuzione dell'intero job. Solo a questo punto era previsto un nuovo intervento manuale da parte dell'operatore per caricare un nuovo job sul lettore di schede.

Al fine di ridurre ulteriormente l'intervento manuale dell'operatore fu inoltre deciso di organizzare il lavoro del sistema caricando sul lettore di schede invece di un solo pacco, corrispondente al job di un singolo utente, un lotto di pacchi di schede, corrispondenti ad altrettanti job di utenti diversi che l'operatore aveva raccolto preceden-

temente (da ciò deriva il nome di sistemi *batch* con cui questi primi sistemi operativi sono stati conosciuti). In questo modo il monitor poteva passare dall'esecuzione di un job a quella del job successivo senza soluzione di continuità. L'intervento manuale dell'operatore avveniva solo alla fine dell'esecuzione di un batch di job. A quel punto veniva inserito sul lettore un nuovo batch, raccolto durante l'esecuzione del batch precedente, e la macchina veniva di nuovo riattivata manualmente.

Con questa tecnica l'efficienza aumentò pur rimanendo molto bassa (nell'ordine di poche unità percentuali).

Le caratteristiche essenziali di questo tipo di sistemi erano tre:

- La prima, positiva, legata all'incremento dell'efficienza di uso delle risorse di macchina. Tale incremento fu ottenuto semplicemente limitando l'intervento manuale anche se ciò implicò la necessità di dedicare al nuovo sistema operativo parte della memoria e anche parte del tempo di CPU durante il quale non potevano essere eseguiti programmi applicativi (*overhead di sistema*). Il sistema operativo era fondamentalmente costituito dal monitor e da un insieme di semplici routine di gestione dei dispositivi d'ingresso/uscita chiamato **BIOS** (*Basic Input Output System*)).
- La seconda, negativa, implicò un allontanamento dell'utente dalla macchina. Il programmatore che portava il proprio pacco di schede all'operatore riceveva i risultati (sotto forma di tabulato) dopo ore se non giorni. Ciò ebbe conseguenze molto negative sulla lunghezza e complessità della fase di debugging di un programma.
- Una terza caratteristica, ancora negativa, era legata al fatto che i programmi venivano eseguiti in modo completamente sequenziale e nello stesso ordine con cui comparivano i relativi pacchi di schede sul lettore. Poteva quindi accadere che un utente che aveva presentato un programma molto semplice, e quindi eseguibile molto rapidamente, dovesse aspettare anche per un lungo tempo la terminazione di un programma molto più complesso e richiedente molto tempo di esecuzione, solo per il fatto che il pacco di schede del secondo programma era stato posizionato sul lettore prima di quello relativo al programma semplice. In altri termini il sistema era caratterizzato dalla impossibilità di intervenire sull'ordine di esecuzione dei programmi (*scheduling*).

1.2.3 Sistemi batch multiprogrammati

Il principale motivo per cui l'efficienza rimaneva ancora molto bassa nei primi sistemi batch era legata alla notevole differenza fra la velocità della CPU e quella, molto più lenta, dei dispositivi periferici (soprattutto lettore di schede e stampante). Perciò, quando la CPU attivava un trasferimento di dati (per leggere le schede o stampare i risultati di un programma) doveva restare in attesa della fine del trasferimento, e questi tempi di attesa erano di alcuni ordini di grandezza superiori a quelli con cui la CPU era in grado di eseguire un intero programma.

Fu con l'introduzione a livello hardware, da un lato delle memorie di massa ad accesso casuale (dischi) e dall'altro dei meccanismi di interruzione e di DMA (*Direct Memory Access*) che verranno richiamati nei successivi paragrafi, che fu possibile apportare significativi miglioramenti.

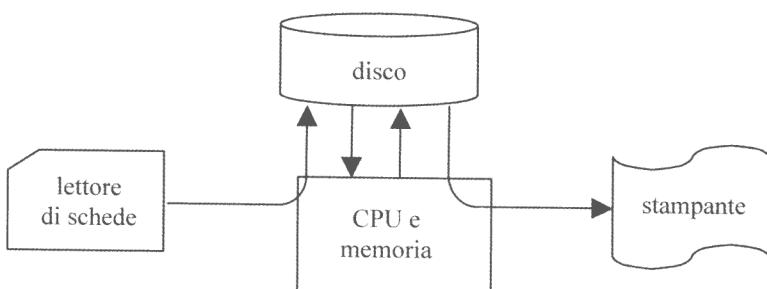


Figura 1.2 Sistema di spooling.

Un primo miglioramento si ebbe con l'introduzione della tecnica dello *spooling*¹. Tutti i job di un batch venivano preventivamente caricati su disco in modo tale che durante le loro esecuzioni la CPU leggesse i dati e producesse i risultati operando solo col disco (molto più veloce dei lettori di schede e della stampante). Mentre il calcolatore eseguiva i programmi di un batch, operando in DMA venivano contemporaneamente caricati su disco i job del batch successivo e, in parallelo, venivano stampati i risultati del batch precedente prelevandoli ancora dal disco (vedi figura 1.2). In questo modo si ottennero due vantaggi:

- Operando la CPU direttamente solo con periferiche veloci, si ridusse drasticamente il suo tempo di attesa aumentando quindi la sua efficienza di uso.
- Avendo su disco l'insieme di programmi da eseguire (detto anche *pool di job*) fu possibile implementare una politica di scheduling scegliendo, di volta in volta, quale eseguire per primo in base a una certa regola. Per esempio, una possibile regola dava la precedenza ai programmi con tempo di esecuzione più breve (*shortest job first*) eliminando così l'inconveniente visto precedentemente.

Però, fu possibile ottenere un drastico miglioramento nell'efficienza di uso delle risorse di macchina soltanto quando, utilizzando il meccanismo fisico delle interruzioni, fu realizzata la tecnica della *multiprogrammazione (multitasking)*. Questa tecnica costituisce la vera chiave di volta nello sviluppo dei sistemi operativi e si basa su un'osservazione molto semplice. Invece di caricare in memoria fisica un solo programma ed eseguirlo fino al suo completamento prima di caricarne un successivo, si caricano due o più programmi contemporaneamente in diverse aree di memoria (vedi figura 1.3).

In questo modo, è possibile iniziare l'esecuzione di uno dei programmi e, appena questo richiede l'intervento di un dispositivo d'ingresso/uscita e si pone in attesa che il dispositivo termini l'operazione prima di poter continuare, è possibile dedicare la CPU ad un altro dei programmi presenti in memoria evitando così che questa resti inattiva. Così facendo, tutti i programmi presenti in memoria vengono fatti evolvere "contemporaneamente", anche se in ogni istante uno solo di essi è in reale esecu-

¹ Il termine *spool* è l'acronimo dell'inglese *simultaneous peripheral operation on-line*.

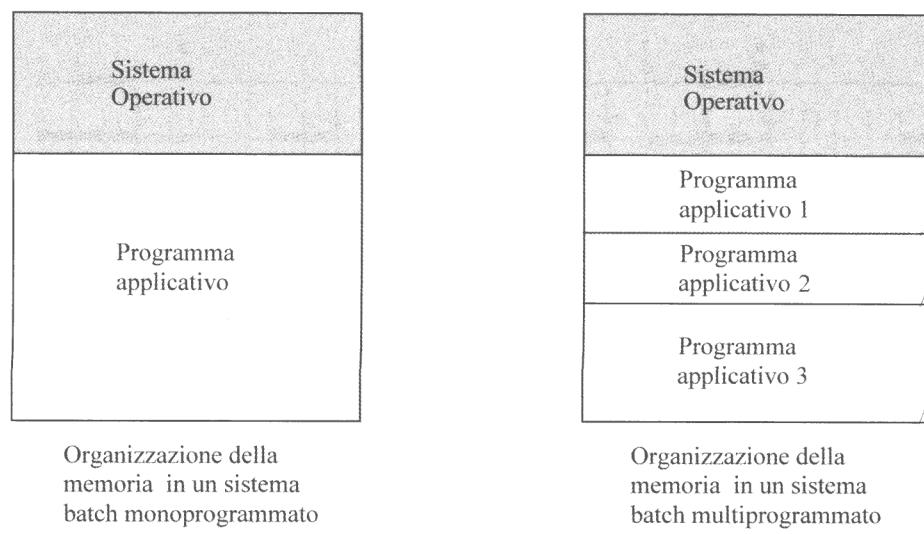


Figura 1.3 Configurazione della memoria nei sistemi mono e multiprogrammati.

zione. Questo schema rispecchia, in parte, l'organizzazione del lavoro di un essere umano. Raramente il nostro cervello si limita ad eseguire un solo compito alla volta dall'inizio fino alla fine. Molto più spesso, ciascuno di noi inizia varie attività e le porta avanti contemporaneamente, o meglio dedicando una frazione di tempo alternativamente a ciascuna attività. Un tipico esempio può essere quello di un maestro di scacchi che gioca contemporaneamente contro più avversari. Seguendo lo stesso criterio, un sistema di elaborazione può eseguire più programmi contemporaneamente anche se in ogni istante dedica le proprie risorse a uno solo di essi.

Con la tecnica della multiprogrammazione è possibile ridurre drasticamente il tempo di attesa della CPU e quindi migliorare notevolmente la sua efficienza fino ad ottenere valori che superano l'80%.

Per chiarire il concetto, supponiamo di dover eseguire tre programmi P1, P2 e P3 con le seguenti caratteristiche: P1 esegue per 1 unità di tempo (espressa, per esempio, in multipli di millisecondi) quindi attiva un trasferimento e resta in attesa per 4 unità. Successivamente esegue ancora per altre 2 unità prima di attivare un nuovo trasferimento e restare in attesa per ulteriori 3 unità di tempo. Infine, riprende l'esecuzione e termina dopo un'ultima unità di tempo. P2 esegue per 1 unità prima di bloccarsi per 6 unità. Quindi riprende l'esecuzione e termina dopo un'ultima unità di tempo. Infine, P3 esegue per 1 unità, successivamente si blocca per 4 unità e poi riprende l'esecuzione e termina dopo ulteriori 3 unità di tempo. In figura 1.4 viene riportato un diagramma temporale relativo all'esecuzione sequenziale dei tre programmi, uno alla volta. Nella figura, i tratti pieni rappresentano periodi di esecuzione mentre quelli tratteggiati periodi di attesa di un programma (e quindi anche della CPU). Inoltre, con il simbolo “↓” viene rappresentata la fine di un periodo di attesa (corrispondente all'interruzione di fine trasferimento da parte del dispositivo) e con l'asterisco (*) la terminazione di un programma.

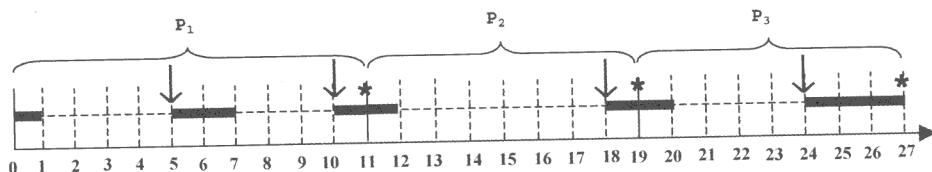


Figura 1.4 Esecuzione sequenziale.

Come si può notare, per completare l'esecuzione dei tre programmi sono necessarie 27 unità di tempo durante le quali, però, la CPU esegue programmi solo per 10 unità di tempo. L'efficienza complessiva con cui viene usata la CPU è quindi pari a $10/27$ e cioè pari al 37% del suo tempo.

In figura 1.5 viene riportato un analogo diagramma temporale nel caso in cui il sistema sia multiprogrammato, e cioè con i tre programmi presenti contemporaneamente in memoria e con la CPU che, quando non può proseguire l'esecuzione di un programma, viene commutata, se possibile, su un diverso programma in grado di essere eseguito.

Come si può notare, con la multiprogrammazione si ha un notevole vantaggio per quanto riguarda la velocità di esecuzione dell'insieme complessivo dei programmi. In esecuzione sequenziale erano necessarie 27 unità di tempo, adesso l'esecuzione di tutti e tre i programmi termina dopo 12 unità. Questo è ovviamente una conseguenza della maggiore efficienza con cui la CPU viene usata. Come si può notare nell'esempio, la CPU resta inattiva per due sole unità di tempo (fra le unità 3 e 5). L'efficienza diventa quindi pari a $10/12$ e cioè superiore a 83%. Aumentando il numero di programmi in esecuzione contemporanea (valore noto col termine di *grado di multiprogrammazione*) in teoria l'efficienza può essere ulteriormente aumentata. Infatti, nel precedente esempio, nelle due unità di tempo in cui la CPU è restata inattiva questa avrebbe potuto eseguire un eventuale quarto programma. Questo valore di efficienza è in realtà solo teorico poiché in un sistema multiprogrammato è necessario tener conto che la CPU deve dedicare una porzione del proprio tempo alla realizzazione del meccanismo necessario a trasferire il controllo della stessa CPU dall'esecuzione di un programma a quella di un altro. Questo meccanismo è noto col termine di *cambio di contesto* e, come verrà illustrato in dettaglio nel capitolo 2, comporta un aumento dell'overhead di sistema. Infatti, quando un programma non può proseguire poiché, avendo attivato un trasferimento di dati deve attendere il suo completamento, viene chiamata una funzione del sistema operativo che salva lo stato della CPU (e cioè i valori di tutti i suoi registri) in un'area di memoria da cui verrà successivamente recuperato quando il programma dovrà tornare in esecuzione. Inoltre, parte del tempo di CPU è necessario per eseguire le routine di risposta delle interruzioni provenienti dai dispositivi. È infatti all'arrivo di un'interruzione che il sistema operativo deve essere avvertito che è terminata un'operazione di trasferimento dati e che quindi il programma che l'aveva precedentemente attivata è, da adesso, in grado di tornare in esecuzione.

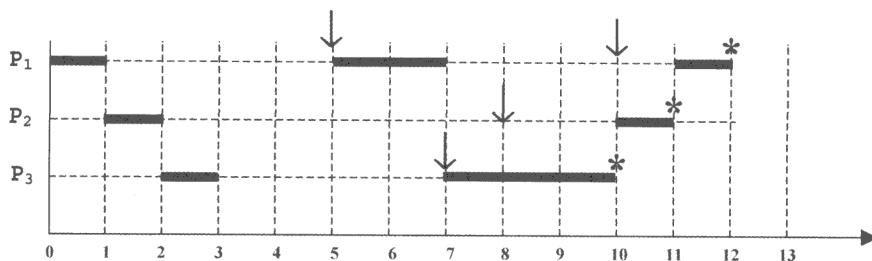


Figura 1.5 Esecuzione in multi-tasking.

Con l'introduzione della multiprogrammazione, i sistemi operativi non sono più semplici programmi come il monitor dei primi sistemi batch, che si limitavano a sequenzializzare i programmi applicativi richiesti dai vari utenti. Adesso, avendo più programmi in esecuzione, è necessario che il sistema operativo si occupi anche della gestione di tutte le risorse per evitare interferenze tra programmi diversi. Infatti, se l'accesso alle risorse non fosse controllato e regolato dal sistema operativo, un programma in esecuzione potrebbe tentare di operare su un dispositivo quando lo stesso sta già lavorando a favore di un altro programma. Inoltre, è necessario che il sistema operativo implementi dei criteri in base ai quali assegnare le risorse ai vari programmi in esecuzione che ne facciano richiesta (*algoritmi di scheduling*). Ad esempio, se non è possibile caricare in memoria contemporaneamente tutti i programmi appartenenti ad un batch, e presenti su memoria di massa in attesa di essere eseguiti, il sistema deve sceglierne un sottoinsieme da eseguire in multiprogrammazione (*job mix*) mentre gli altri verranno eseguiti successivamente. Un criterio spesso adottato è stato quello di scegliere un mix di programmi con caratteristiche diverse, ad esempio alcuni richiedenti molti calcoli e pochi trasferimenti di dati (*CPU-bound*) e altri richiedenti pochi calcoli ma molti trasferimenti e su periferiche diverse (*I/O-bound*) in modo tale da ottimizzare l'uso di tutte le risorse di macchina. Inoltre, quando un programma in esecuzione non può proseguire a causa dell'attivazione di un trasferimento dati ed è quindi necessario commutare la CPU su un altro programma, si pone il problema di scegliere a quale programma cedere il controllo. Per esempio in figura 1.5, all'istante 10 la CPU termina l'esecuzione del programma P3 e a quel punto sono già terminati anche i periodi di attesa dei programmi P2 e P1, ciascuno dei quali può quindi riprendere la sua esecuzione. Si pone quindi il problema di quale fra questi eseguire per primo. Un criterio spesso adottato è quello di scegliere (come mostrato in figura) quello che da più tempo è pronto per essere eseguito, adottando quindi un criterio First-In-First-Out. In altri sistemi, all'epoca spesso utilizzati nell'ambito di centri di calcolo, poteva essere implementato il criterio di scegliere per primi i programmi di utenti privilegiati, ad esempio in base a diversi livelli di tariffe.

Con l'introduzione della multiprogrammazione, e di conseguenza con l'aumento della complessità del sistema operativo, furono introdotti nuovi meccanismi hardware necessari per garantire una corretta ed efficiente esecuzione delle funzioni del si-

stema. Come vedremo nel paragrafo 1.3, sono stati introdotti meccanismi di protezione sia della memoria (per evitare che lo spazio dedicato ad un programma sia erroneamente manomesso durante l'esecuzione di un altro programma), sia della CPU, con l'introduzione del doppio stato di esecuzione (privilegiato/non privilegiato) per garantire che le istruzioni macchina con cui è possibile modificare lo stato di una qualunque risorsa fisica siano eseguibili solo da parte di funzioni del sistema operativo.

I sistemi batch multiprogrammati hanno costituito i primi complessi sistemi operativi funzionanti sui grossi mainframe. Un classico esempio è costituito dal sistema OS/360 funzionante sulle macchine IBM della serie 360 e, successivamente 370 [13].

1.2.4 Sistemi time-sharing

La multiprogrammazione da un lato e il controllo dell'esecuzione dei programmi secondo la filosofia batch dall'altro, ebbero un notevole successo e favorirono lo sviluppo dei sistemi operativi. I motivi del loro successo erano legati alle caratteristiche dei sistemi di elaborazione che, in quel periodo (fino alla metà degli anni '60), erano costituiti esclusivamente dai grossi sistemi (*mainframe*) presenti nei centri di calcolo, estremamente costosi e fondamentalmente dedicati all'esecuzione di programmi applicativi di tipo tecnico/scientifico che non avevano particolari necessità di interagire direttamente con il programmatore. Le caratteristiche peculiari di questi sistemi erano:

- l'alta efficienza raggiunta nell'uso delle costose risorse di calcolo;
- l'impossibilità da parte del programmatore di interagire direttamente col sistema;
- il lungo *tempo di risposta* di ogni programma (inteso come l'intervallo temporale compreso fra l'istante in cui il programmatore consegnava il proprio pacco di schede all'operatore e l'istante in cui otteneva il tabulato con i risultati).

Con lo sviluppo dei primi programmi applicativi di tipo interattivo i sistemi batch iniziarono a mostrare il loro limite. Questa categoria di applicazioni prevede che il programma in esecuzione debba interagire con l'utente. Si pensi, ad esempio ad un sistema *transazionale* tipico di molte applicazioni gestionali (sistemi bancari, sistemi per la prenotazione di posti su aerei, ecc). In questi casi è necessario che l'utente, tramite un terminale collegato al sistema, e disponendo di un opportuno linguaggio di controllo analogo al JCL dei sistemi batch, ma con i singoli comandi battuti direttamente sul terminale, indichi al sistema quali operazioni sono richieste. Normalmente, il sistema resta in attesa che l'utente batta una comando, visualizzando sul video del terminale un particolare carattere (*prompt*). Quando l'utente ha terminato di battere un comando il sistema carica in memoria, ed esegue, un programma che svolge il compito richiesto dall'utente alla fine del quale visualizza di nuovo il segnale di prompt e resta in attesa del prossimo comando. Questo tipo di comportamento implica che l'utente operi sul sistema all'interno di *sessions di lavoro* che iniziano quando l'utente si siede ad uno dei terminali collegati al sistema e, specificando il proprio codice identificativo (*user-ID*) e la propria parola d'ordine (*password*), si collega col sistema operativo (*login*). Una sessione di lavoro coincide, sostanzialmente, con l'invio di un comando al sistema, con l'esecuzione da parte del sistema del compito richiesto e quindi, di nuovo, con l'invio del comando successivo e così

via fino a quando, inviando il comando di chiusura (*logout*), l'utente non decide di terminare la sessione.

Anche in questo tipo di sistemi la multiprogrammazione costituisce una tecnica fondamentale. Infatti, normalmente, più utenti sono contemporaneamente collegati al sistema. Ciò significa che il sistema operativo deve essere in grado di controllare l'esecuzione concorrente dei programmi che svolgono i compiti richiesti dai vari utenti. Essendo il comportamento del sistema conversazionale, durante una sessione di lavoro la frazione di tempo che la CPU dedica all'esecuzione dei programmi richiesti dall'utente è sicuramente trascurabile nei confronti del tempo che l'utente dedica a battere i vari comandi sul terminale. Per cui il sistema operativo può sfruttare questi tempi "morti" di un utente per far avanzare i programmi richiesti da altri utenti. Il risultato è che ciascun utente vede il comportamento della macchina come se questa fosse dedicata tutta a lui (*macchina virtuale*). In effetti, come vedremo nel seguito del testo, il sistema operativo distribuendo la CPU e la memoria fra i vari programmi in esecuzione è come se generasse una CPU virtuale e una memoria virtuale per ciascuno dei programmi in esecuzione.

La differenza fondamentale fra i sistemi batch e questa nuova categoria di sistemi è legata alle loro diverse finalità. Nei sistemi batch, come è stato illustrato, l'obiettivo fondamentale era quello di ottenere un'elevata efficienza nell'uso delle costose risorse di macchina. Adesso l'efficienza acquista una minore importanza. Per prima cosa diminuisce il costo delle macchine, anche per l'evoluzione delle tecniche hardware e l'introduzione sul mercato dei *minicalcolatori*, molto meno costosi dei grossi *mainframe*. Ma soprattutto cambia l'obiettivo principale del sistema operativo che, in sistemi di tipo interattivo, diventa quello di minimizzare il tempo di risposta dei vari programmi, in modo tale da minimizzare il tempo che l'utente deve attendere di fronte al proprio terminale per aspettare il completamento di un servizio richiesto al sistema.

Non solo è importante minimizzare il tempo di risposta di un programma, ma ancora più importante è rendere questo tempo proporzionale alla complessità del programma. Un utente è tendenzialmente disponibile ad attendere al terminale il completamento di un programma molto complesso per un tempo maggiore rispetto a quanto sia disponibile ad attendere il completamento di un programma banale. Per esempio, se un utente batte un comando per conoscere l'ora attuale si aspetta di ricevere immediatamente la risposta, mentre nel caso di un comando per compilare un programma costituito da molte migliaia di linee di codice è disponibile ad attendere un tempo più lungo. I criteri utilizzati nei sistemi batch non sono adatti a fornire questo tipo di comportamento. In questi sistemi, infatti, se entra in esecuzione un programma *CPU-bound*, che non richiede trasferimenti per molto tempo, questo manterrà il controllo della CPU non consentendo l'esecuzione di nessun altro programma, neppure di quelli semplici che, viceversa, dovrebbero terminare il più rapidamente possibile.

Il criterio che fu adottato nei nuovi sistemi fu quello di eseguire i vari programmi assegnando loro una porzione di tempo di CPU (*time slice*) dell'ordine di alcune decine di millisecondi. Durante tale intervallo il programma poteva terminare, o comunque bloccarsi per aver lanciato operazioni d'ingresso/uscita, consentendo quindi la scelta di un diverso programma. Se però, allo scadere della porzione di tempo il

programma era ancora in esecuzione, tramite una interruzione lanciata dal timer, la CPU veniva comunque commutata su un diverso programma mediante un cambio di contesto. In questo modo era impossibile che un programma potesse monopolizzare la CPU, e quindi ritardare per molto tempo l'esecuzione degli altri. Inoltre, questo criterio di assegnazione della CPU privilegiava i programmi semplici, quelli cioè in grado di terminare la loro esecuzione all'interno del *time slice* assegnato. Gli altri, una volta revocati, tornavano fra i programmi in attesa di ricevere una nuova fetta di tempo, ma per poterla ricevere dovevano attendere che tutti gli altri programmi l'avessero a loro volta ricevuta (*round robin scheduling*). Chiaramente, aumentando il numero delle commutazioni di contesto della CPU aumentava l'overhead del sistema e, conseguentemente, diminuiva l'efficienza di uso della CPU.

I sistemi operativi che operano secondo questi criteri sono noti col nome di sistemi *time-sharing* o “*a divisione di tempo*”. Rispetto ai precedenti sistemi, aumenta molto la complessità del sistema operativo. A causa della multiutenza è ora necessario controllare e gestire gli accessi al sistema (gestione degli *account*) e complicare le funzioni di protezione sia del sistema che dei dati e dei programmi dei singoli utenti. Anche la gestione della memoria si complica poiché, richiedendo un più alto livello di multiprogrammazione, diventa necessario revocare un programma che non può proseguire non solo la CPU ma anche la memoria.

Fra i primi sistemi time-sharing è famoso il sistema Multics [21], realizzato intorno alla metà degli anni '60 al MIT, importante per la quantità di contributi che tale sistema ha fornito allo sviluppo dei sistemi operativi. Da esso sono poi derivati molti altri sistemi time-sharing come il VMS della Digital [30] e lo stesso sistema UNIX nelle sue diverse versioni [6].

1.2.5 Sistemi in “tempo reale”

L'applicazione dei sistemi di elaborazione ai problemi di controllo favorì lo sviluppo di una nuova tipologia di sistemi operativi: i *sistemi in tempo reale*. In questo tipo di applicazioni il calcolatore è dedicato a controllare il corretto funzionamento di un sistema fisico (*ambiente operativo*), che può essere costituito da un impianto industriale, da una centrale elettrica o di telecomunicazioni, da una centralina per il controllo motore di un'auto, da un robot o comunque da un'apparecchiatura di cui sia necessario mantenere sotto controllo le grandezze fisiche che la caratterizzano (vedi figura 1.6). Il sistema di calcolo, mediante opportuni sensori, legge periodicamente il valore delle grandezze fisiche da controllare (pressioni, portate, temperature ecc.). Una volta letto il valore di una grandezza, esegue un programma (spesso indicato col termine di *task*) che stabilisce se tale valore rientra nell'ambito dei valori corretti oppure se si verifica uno scostamento da questi. In quest'ultimo caso il sistema di calcolo retroagisce sull'ambiente operativo mediante opportuni attuatori cercando di riportare la grandezza in questione nell'intervallo dei valori corretti.

Anche per questo tipo di sistemi la multiprogrammazione rappresenta una tecnica fondamentale, poiché i task che devono essere eseguiti dal sistema sono molti (almeno uno per ciascuna delle grandezze da tenere sotto controllo). Ciascuno di questi task costituisce un programma ciclico che, periodicamente, va in esecuzione per svolgere il proprio compito. Una volta terminata la sua esecuzione ricomincia dall'inizio avviando una nuova esecuzione. Per evitare di sovraccaricare inutilmente il si-

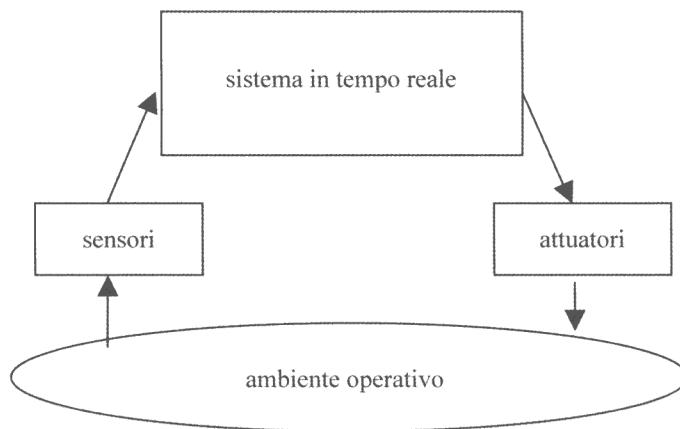


Figura 1.6 Sistema in tempo reale.

stema ogni task, terminata un'esecuzione e prima di iniziare la successiva, resta in attesa per un periodo di tempo che dipende dalla velocità con cui varia la grandezza da controllare. Durante i tempi di attesa di un task il sistema può eseguire gli altri task che siano pronti per essere eseguiti.

Ciò che caratterizza il comportamento di un sistema in tempo reale è il fatto che ogni task deve completare ciascuna delle proprie esecuzioni entro un intervallo temporale imposto dall'applicazione (*deadline* del task). L'obiettivo fondamentale di questi sistemi operativi non è, quindi, quello di massimizzare l'efficienza di uso delle risorse come nei sistemi batch, o quello di minimizzare i tempi di risposta come nei sistemi time-sharing, ma quello di garantire, a priori, che tutti i task del sistema completino le loro esecuzioni entro la rispettive deadline. In altri termini, la validità dei risultati prodotti da un programma applicativo (task) dipende non solo dalla correttezza del programma, ma anche dall'intervallo temporale entro il quale i risultati sono prodotti. In questo senso la parola *tempo* entra nel nome di questo tipo di sistemi operativi. L'aggettivo *reale* denota poi il fatto che la risposta del sistema agli eventi che si verificano nell'ambiente operativo deve avvenire durante l'evolversi degli eventi stessi.

Inoltre, normalmente, in un sistema in tempo reale, esistono task con diverso grado di criticità. È quindi naturale che la gestione della multiprogrammazione, in questi sistemi, preveda che quando un programma si blocca ed è necessario assegnare la CPU ad un altro programma pronto per l'esecuzione, la scelta venga fatta in base alla priorità che il sistema assegna a ciascun task. Tale priorità può essere definita una volta per tutte, prima che il sistema inizi la sua esecuzione (*priorità statiche*), o calcolata dinamicamente in base alle caratteristiche dei singoli task, ivi inclusa la necessità di completarli entro le rispettive deadline (*priorità dinamiche*).

Esistono molte categorie di sistemi in tempo reale. Alcuni, noti come sistemi “*hard real-time*” sono dedicati ad applicazioni di controllo critiche, nel senso che la violazione di anche una sola deadline può compromettere il corretto funzionamen-

to del sistema controllato, con conseguenze che possono essere anche catastrofiche per tale sistema. Si pensi ad esempio ai sistemi che controllano l'assetto di volo di un aereo. Altri sistemi, noti col nome di “*soft real-time*” hanno esigenze meno stringenti. In questi casi, la violazione di una deadline non compromette il corretto funzionamento dell'ambiente operativo ma ne riduce le prestazioni e quindi la qualità del servizio (QoS – *Quality of Service*).

Una categoria di sistemi in tempo reale molto importante per la loro diffusione è costituita dai cosiddetti *sistemi immersi* (*embedded systems*). Si tratta di sistemi progettati su particolari specifiche e dedicati a singole applicazioni di grande consumo. In questa categoria di applicazioni non è sempre facile distinguere fra il sistema operativo e l'applicazione. In realtà, come indicato anche dal nome, tutto il software è al tempo stesso software di sistema e software applicativo, immerso spesso nell'ambito di una singola scheda, o di un singolo chip, e dedicato a un'unica applicazione. Rientrano in questa categoria tutte le schede di controllo di varie apparecchiature, dalle schede controllo motore di un'auto a quelle inserite all'interno di normali elettrodomestici, ecc.

I primi sistemi in tempo reale erano costituiti da programmi ad hoc scritti direttamente in assembler e adatti a gestire una sola particolare applicazione. Successivamente, sfruttando lo sviluppo dei sistemi operativi time-sharing, sono stati sviluppati molti sistemi operativi in tempo reale di tipo generale sia per applicazioni hard che soft. Fra questi possiamo citare il sistema VxWorks [101] o, in alcuni casi, varianti di sistemi time sharing adattate per funzionare con applicazioni in tempo reale, come RT-Linux [61].

1.2.6 Sistemi operativi per personal computer

Lo sviluppo della microelettronica e dei microprocessori ha contribuito enormemente a far diminuire i costi delle componenti hardware dei sistemi di elaborazione, e di conseguenza, alla diffusione dei calcolatori personali (*Personal Computer – PC*). Come indica il nome, l'uso del PC è personale, riservato ad una sola persona alla volta e quindi non presenta tutte le problematiche tipiche della multiutenza, come ad esempio i sistemi time-sharing. Per questo motivo i primi sistemi operativi realizzati per PC sono stati sistemi molto semplici, progettati senza ricorrere alla tecnica della multiprogrammazione. I principali obiettivi di questi sistemi sono stati quelli di aiutare l'utente sia nella gestione delle proprie informazioni (dati e/o programmi) che nella realizzazione di nuove applicazioni fornendo un'interfaccia di programmazione (API) atta a facilitare l'accesso ai dispositivi d'ingresso/uscita e alle informazioni memorizzate nel sottosistema per la gestione degli archivi (*file system*). Un tipico esempio è stato il sistema MS-DOS sviluppato da Microsoft [67].

Con l'aumento delle prestazioni dei microprocessori e delle dimensioni della memoria principale, la tecnica della multiprogrammazione si è rilevata molto utile anche nei sistemi operativi per PC. In questi casi, infatti, è ormai usuale consentire all'utente di operare sul proprio PC lanciando, contemporaneamente, più applicazioni. Si pensi, ad esempio, a un utente che ha attivato un programma per la compilazione di un insieme di trasparenze (*slide*) da proiettare ad una presentazione. La compilazione di una trasparenza può richiedere di prelevare il testo da un documento precedentemente realizzato mediante un programma di videoscrittura. È quindi utile lan-

ciare, contemporaneamente, sia il programma per la compilazione delle trasparenze che quello di videoscrittura per recuperare dal documento il testo da utilizzare nella trasparenza in fase di compilazione. Analogamente può essere utile attivare anche un'applicazione di grafica per recuperare un grafico, anch'esso da utilizzare per la compilazione della trasparenza. E infine, attivare anche un ulteriore programma applicativo per l'ascolto di un CD di musica in sottofondo. Ciò che caratterizza il sistema, in questo caso, è il criterio con cui la CPU viene commutata da un programma all'altro. È ora direttamente l'utente che, mediante la propria interfaccia di input, costituita da un sistema di puntamento sul video (*mouse*), seleziona (“*clicca*”) esplicitamente il programma su cui desidera che la CPU venga commutata.

Rispetto a un sistema time-sharing, in cui ciascun programma applicativo può visualizzare i messaggi sul terminale video a cui è collegato l'utente che ha richiesto quel programma, adesso il terminale video è uno solo e tutte le applicazioni devono condividerlo. È quindi naturale, in questo caso, che oltre ai concetti di CPU virtuale e memoria virtuale, il sistema operativo gestisca anche quello di video virtuale sovrapponendo sull'unico terminale video tante *finestre*, ciascuna dedicata ad uno dei programmi applicativi in esecuzione.

Tutti gli attuali sistemi per PC hanno ormai adottato la multiprogrammazione; alcuni di questi sono stati realizzati esplicitamente per funzionare su personal computer, come Windows di Microsoft [24] o MacOS di Macintosh [4], altri sono dei veri e propri sistemi time-sharing adattati a funzionare con interfacce tipiche di sistemi personali (icone, finestre e mouse) come Linux [98].

1.2.7 Sistemi operativi per sistemi paralleli e distribuiti

Come già indicato precedentemente, l'evoluzione delle architetture e quella dei sistemi operativi si sono vicendevolmente influenzate. In particolare, la complessità dei sistemi operativi è cresciuta nel tempo anche come conseguenza della maggiore complessità delle attuali soluzioni architettoniche tipiche dei sistemi multielaboratore o dei sistemi distribuiti in rete.

In un normale sistema monoelaboratore la concorrenza introdotta con la multiprogrammazione è una concorrenza macroscopica, nel senso che l'esecuzione di un programma può iniziare prima che sia terminata l'esecuzione di un altro programma. In realtà, però, in ogni istante uno solo di essi sta eseguendo. Con l'introduzione di architetture multielaboratore la concorrenza tra programmi applicativi diventa reale nel senso che, disponendo di più CPU, più programmi sono in esecuzione nello stesso istante. Ciò crea una serie di problemi, in particolare quando sono in esecuzione contemporanea funzioni diverse del sistema operativo su diverse CPU. È necessario che il progettista del sistema operativo tenga conto di queste eventualità per prevenire possibili condizioni di inconsistenza sulle strutture dati del sistema, che si potrebbero verificare se queste fossero modificate contemporaneamente da funzioni di sistema eseguite su CPU diverse.

Infine, l'introduzione delle reti di calcolatori, sia locali che geografiche, ha richiesto lo sviluppo di ulteriori moduli del sistema operativo, quelli necessari alla gestione delle interfacce di rete e, al tempo stesso, la creazione di nuovi servizi che hanno permesso ad un sistema collegato in rete di accedere a informazioni distribuite sui vari nodi; si pensi ad esempio allo sviluppo del *world wide web* o alla possibi-

lità di realizzare applicazioni distribuite, composte cioè da vari programmi eseguiti su nodi diversi della rete.

I sistemi operativi concepiti per funzionare su calcolatori collegati in rete vengono, normalmente, classificati in due categorie: i *sistemi operativi di rete* e i *sistemi operativi distribuiti*. Nei primi, ogni nodo della rete dispone di un proprio sistema operativo (spesso anche diversi fra di loro). Ciascun sistema operativo è però abilitato a cooperare con gli altri, per esempio per consentire il trasferimento di file tra computer diversi collegati in rete. Un sistema operativo distribuito prevede, viceversa, che lo stesso sistema sia caricato su tutti i nodi della rete e funzioni in modo tale che l'utente collegato ad uno dei nodi “veda” un sistema unico nascondendo il fatto che in realtà il sistema è distribuito. In questo caso, ad esempio, il sistema offre all'utente l'astrazione di un singolo spazio di memoria centrale e secondaria indipendentemente dal fatto che tale spazio sia costituito dall'insieme dei singoli spazi distribuiti sui nodi della rete.

Per una trattazione più approfondita dell'argomento si rimanda alla bibliografia [97].

1.3 Richiami di architetture dei sistemi di elaborazione

Poiché il sistema operativo interagisce direttamente con le componenti hardware del sistema di elaborazione, per meglio comprendere sia i principi di funzionamento che le tecniche di realizzazione delle varie componenti di un sistema operativo è necessaria una buona conoscenza della struttura fisica del sistema. Una descrizione approfondita di questo argomento esula, ovviamente, dagli scopi del testo per cui si rimanda alla bibliografia per eventuali approfondimenti. Data quindi per scontata una generale conoscenza dei principi di funzionamento di un calcolatore, scopo di questo paragrafo è quello di richiamare, brevemente, quegli aspetti architetturali che più direttamente sono correlati al funzionamento delle componenti del sistema operativo e a cui verrà fatto riferimento nei capitoli successivi del testo.

In figura 1.7 viene mostrata una visione molto semplificata di un moderno calcolatore. In essa sono evidenziati i principali componenti dell'architettura di un sistema monoelaboratore: l'*unità centrale di elaborazione* (indicata nel testo anche con i nomi di *processore* o *Central Processing Unit* o, più brevemente, *CPU*), la *memoria centrale* (*Random Access Memory – RAM*), i vari *dispositivi d'ingresso/uscita*. Tali componenti sono tra loro interconnessi tramite un *bus* di sistema. Il bus può essere visto come un insieme di “fili” su cui viaggiano le informazioni. Esso si distingue in una parte di indirizzi, la quale serve per comandare il controllore della memoria, e una parte dati su cui viaggiano i dati veri e propri. La dimensione del bus (ovvero il numero di fili) corrisponde al numero di bit che possono essere letti e/o scritti in un ciclo di clock del bus. Tale grandezza è così importante che viene utilizzata per classificare i processori. In questo senso, abbiamo processori a 8, 16, 32 o 64 bit a seconda della dimensione del loro bus dati. Il bus di solito lavora ad una frequenza inferiore alla frequenza di funzionamento del processore. Quindi, l'accesso al bus può comportare un “rallentamento” dell'attività del processore.

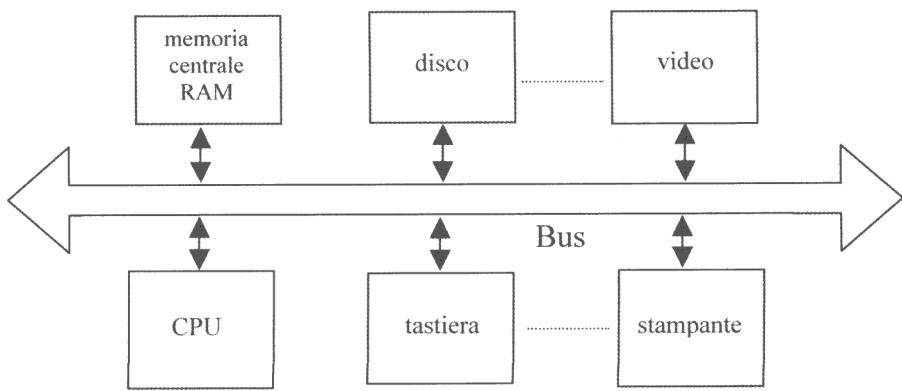


Figura 1.7 Visione semplificata dell'architettura di un sistema a processore singolo.

1.3.1 Il processore

È compito del processore eseguire, una dopo l'altra, le istruzioni di un programma contenuto nella memoria centrale. Durante l'esecuzione di un'istruzione, il processore attraversa diverse fasi. I processori esistenti in commercio sono molto diversi l'uno dall'altro. Nel seguito ci limiteremo a descrivere alcune delle loro principali caratteristiche, comuni a quasi tutti i processori.

L'aspetto che maggiormente caratterizza un processore, dal punto di vista del programmatore, è costituito dall'insieme dei registri interni (unità di memoria interne al processore e ad accesso molto più veloce rispetto alle locazioni della memoria centrale). Normalmente i registri interni vengono distinti in *registri generali* o *registri programmabili* (e cioè direttamente visibili al programmatore) e in *registri di stato e controllo*. I registri generali sono quelli utilizzati per contenere dati e indirizzi di locazioni di memoria durante l'esecuzione di un programma. Ad esempio appartengono a questa categoria i registri utilizzati come accumulatori e per contenere operandi di istruzioni macchina, i registri utilizzati per contenere indirizzi di locazioni di memoria o le informazioni utilizzate per calcolare un indirizzo, come i registri indice, i registri base, i registri di segmento ecc., a seconda delle modalità di indirizzamento previste dal processore. Un particolare registro programmabile, ormai presente in tutti i processori moderni, è il registro *Stack Pointer* (*SP*) che è destinato a contenere, in ogni stante durante l'esecuzione di un programma, l'indirizzo dell'ultimo dato (*top*) inserito nella pila utilizzata dal programma (*stack*). Tale registro rappresenta l'operando隐式 delle istruzioni macchina *push* e *pop* utilizzate per inserire o togliere rispettivamente dati dalla pila. Ricordiamo che la pila viene utilizzata da un programma per memorizzare gli indirizzi di ritorno delle chiamate di procedura, i parametri della procedura e le sue variabili locali.

I registri di stato e controllo non sono, normalmente, visibili al programmatore e contengono informazioni sullo stato del processore. Ad esempio, il registro *contatore delle istruzioni* (detto anche *Instruction Pointer* o *Program Counter* o brevemente

te *IP* o *PC*) è destinato a contenere l'indirizzo in memoria della prossima istruzione da eseguire. È tramite questo registro che il processore implementa il suo ciclo di esecuzione delle istruzioni di un programma composto dalle due fasi:

- *Prelievo (Fetch)*, durante la quale il processore trasferisce in un registro interno (*Instruction Register – IR*) l'istruzione da eseguire, prelevandola dalla locazione di memoria il cui indirizzo è in *PC*, e incrementa il contenuto del *PC* in modo tale da predisporlo al prelievo dell'istruzione successiva;
- *Decodifica ed Esecuzione (Decode and Execute)*, durante la quale l'istruzione viene decodificata ed eseguita, prelevando eventuali operandi dalla memoria o inserendovi eventualmente il risultato.

Un altro importante registro, anch'esso disponibile su tutti i processori, anche se individuato con nomi diversi (e in certi casi costituito anche da un insieme di più registri), è quello utilizzato per contenere alcune informazioni sullo stato interno del processore (ad esempio i bit che codificano il risultato di un'istruzione e sono utilizzati per condizionare una successiva istruzione di salto (*condition code*) o per contenere i bit che condizionano le modalità di funzionamento del processore (a cui accenniamo nel seguito di questo paragrafo) come, ad esempio, i bit che specificano il livello di privilegio con cui il processore sta eseguendo le istruzioni o il bit che abilita il processore a ricevere interruzioni. Normalmente tale registro è noto col nome di *Program Status Word (PS)*.

Come è stato indicato nei precedenti paragrafi, un programma applicativo, durante la sua esecuzione, può invocare una funzione del sistema operativo per chiedere l'esecuzione di un servizio, alla fine del quale il controllo può essere restituito al programma chiamante. Al fine di rendere più efficiente il meccanismo delle chiamate di sistema alcuni processori hanno disponibili due copie identiche dei registri programmabili, ivi incluso il registro SP. Quando è in esecuzione un programma applicativo, il processore utilizza i registri appartenenti a una delle due copie, mentre quando gira una funzione di sistema vengono utilizzati i registri dell'altra copia. In questo modo, all'atto della chiamata di una funzione di sistema non è necessario salvare i contenuti dei registri generali e poi ripristinarli al ritorno al programma chiamante, operazioni altrimenti necessarie se fosse presente una sola copia dei registri generali per evitare che la funzione di sistema, utilizzando un registro modifichi l'informazione in esso contenuta e relativa al programma chiamante. Ovviamente, come vedremo nel seguito, il processore deve essere in grado, istante per istante, di discriminare se il programma in esecuzione è applicativo o di sistema, al fine di utilizzare la corretta copia dei registri generali.

1.3.2 La memoria

La memoria di un sistema di elaborazione si distingue in *memoria volatile* (o memoria *RAM – Random Access Memory*), *memoria non volatile* (o memoria *ROM – Read Only Memory*) e *memoria di massa esterna*. La memoria ROM contiene delle informazioni non modificabili. Di solito, essa contiene il programma di inizializzazione del sistema (*Bootstrap*).

La memoria RAM è la memoria principale ed è utilizzata per contenere i programmi in esecuzione ed i loro dati, nonché il codice del sistema operativo ed i suoi

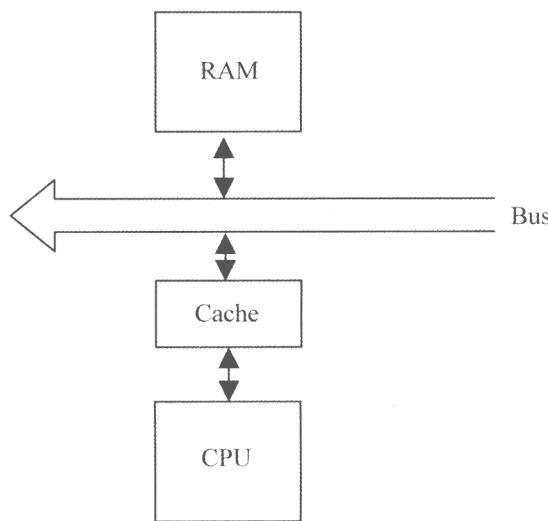


Figura 1.8 La memoria cache.

dati. La memoria RAM mantiene le sue informazioni finché viene alimentata da una fonte di energia (rete o batteria). Quindi, prima di spengere il sistema, le informazioni che non devono essere perdute devono essere salvate sui dispositivi di memoria di massa esterna, quali dischi e nastri magnetici.

Poiché l'accesso alla memoria RAM può rallentare la velocità con cui il processore esegue un programma, i processori di fascia alta implementano uno o più livelli di *memoria cache*. La memoria cache è una memoria molto veloce e molto costosa, che viene utilizzata per velocizzare gli accessi alle informazioni, e viene inserita tra il processore e il bus di sistema, come indicato in figura 1.8.

La cache contiene, di volta in volta, una “immagine” ridotta del contenuto della memoria principale. Ogni volta che il processore esegue un accesso in memoria, come prima cosa si controlla se la locazione riferita si trova nella cache. Se la risposta è positiva (*cache hit*), alla cache si accede direttamente, senza eseguire l'accesso in memoria, riducendo quindi la durata dell'operazione. Se invece la locazione riferita non è presente nella cache (*cache miss*), si effettua un accesso in memoria e contemporaneamente la locazione viene memorizzata nella cache. Quindi, la prossima volta che il processore accederà a quella locazione, molto probabilmente la troverà nella cache. Il meccanismo di funzionamento della memoria cache sfrutta il *principio di località dei riferimenti*. Secondo questo principio i riferimenti alla memoria da parte di un programma, in un certo intervallo di tempo, tendono ad essere “vicini” fra loro. Consideriamo ad esempio un programma che, durante la sua esecuzione, esegue un ciclo `while(...)` per un certo numero di volte. Durante l'intervallo di tempo in cui viene eseguito il ciclo, il processore accederà allo stesso gruppo di istruzioni che fanno parte del ciclo. Molto probabilmente, dopo la prima esecuzione del ciclo, tutte le istruzioni si troveranno nella cache, quindi i cicli successivi saranno molto più veloci ed efficienti.

V	tag	contenuto
---	-----	-----------

Figura 1.9 Elemento della cache.

Per questo motivo, normalmente le locazioni di memoria non vengono memorizzate nella cache singolarmente ma in blocchi di locazioni contigue (ad esempio blocchi di 32 locazioni). Ogni elemento della cache, atto quindi a contenere un blocco di locazioni, è caratterizzato da tre elementi (vedi figura 1.9): un bit di validità (V), un campo, detto *tag*, con cui viene identificato il blocco all'interno della cache e che è costituito da una parte dell'indirizzo fisico di tutte le locazioni del blocco, parte comune a tutti gli indirizzi di quelle locazioni, e il contenuto delle locazioni del blocco. Ad esempio, se il blocco è composto da 32 locazioni, i loro indirizzi si differenziano per i valori relativi ai 5 bit meno significativi dell'indirizzo mentre i restanti bit più significativi dell'indirizzo sono ovviamente uguali per tutte le locazioni del blocco e costituiscono quindi il campo *tag*.

Il bit di validità, come dice il nome, denota col suo valore la validità dei contenuti dell'elemento. Ad esempio, all'inizio dell'esecuzione di un programma, quando ancora non è stato fatto nessun riferimento alla memoria, tutti gli elementi della cache sono invalidi e vengono caricati, e resi validi, ad ogni nuovo riferimento. Ovviamente, quando la cache è piena (tutti gli elementi sono validi) ed è necessario memorizzare un ulteriore blocco di locazioni, ne viene scelto uno il cui contenuto viene rimpiazzato, normalmente quello meno recentemente utilizzato.

All'atto dell'accesso ad una locazione di memoria è quindi necessario verificare, per prima cosa, se tale locazione appartiene ad un blocco già presente in cache, e cioè se la porzione del suo indirizzo, corrispondente al tag, coincide col campo tag di un elemento della cache il cui bit di validità sia 1. Normalmente, per rendere più veloce l'accesso alla cache, tale verifica viene effettuata in parallelo su tutti gli elementi della cache (*accesso associativo*).

La presenza di memorie cache tende ad aumentare percentualmente l'overhead di un sistema operativo multiprogrammato in conseguenza di un cambio di contesto, e cioè della commutazione della CPU fra programmi diversi. Infatti, quando la CPU viene commutata da un programma all'altro, il sistema operativo deve invalidare tutti gli elementi della cache, poiché cambia completamente l'insieme delle locazioni a cui il nuovo programma dovrà fare riferimento aumentando quindi, almeno nella fase iniziale della sua esecuzione dopo il cambio di contesto, il tempo medio di accesso alle locazioni a cui fa riferimento.

Come sarà illustrato nel capitolo 4, nei moderni sistemi che forniscono un supporto hardware al sistema operativo per la gestione della memoria principale (sistemi con *paginazione e/o segmentazione*), esistono anche altre piccole unità di memoria con accesso associativo (*Memory Management Unit*) che si comportano come un'ulteriore memoria cache (*cache di indirizzi o Translation Lookaside Buffer*).

Le differenti memorie di un processore sono organizzate logicamente secondo una gerarchia, come mostrato in figura 1.10. Man mano che ci allontaniamo dal pro-

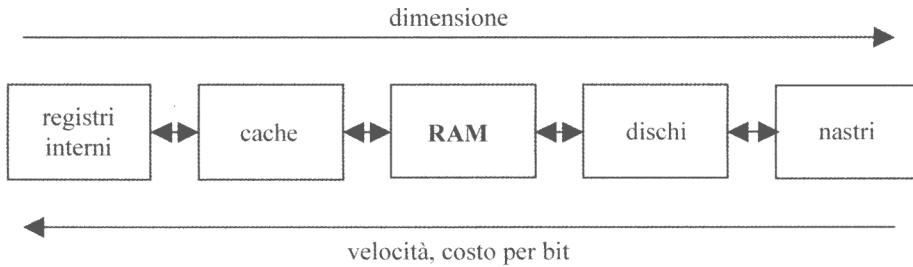


Figura 1.10 Gerarchia delle memorie.

cessore le dimensioni delle unità di memoria crescono, ma diminuiscono le velocità di accesso oltre che il costo per bit di tali unità. La memoria più veloce è costituita dai registri interni del processore, cui si può accedere in un solo ciclo di clock. Subito dopo, in termini di velocità, viene la cache. Dato il costo di tale dispositivo, le dimensioni tipiche di una cache si aggirano intorno a qualche centinaio di kilobyte anche se, nei sistemi moderni, sono presenti ormai memorie cache molto più capienti; inoltre è ormai normale avere più livelli di memorie cache. Segue la memoria RAM, che è più lenta di una cache, ma che nei moderni PC può arrivare fino a qualche gigabyte. Abbiamo quindi i dischi magneticici che vanno dalla decina di gigabyte fino al terabyte (10^{12} byte). Infine, i nastri magnetici e i dischi ottici che vengono utilizzati per memorizzare grandi archivi di dati che non sono di facile accessibilità (ad esempio gli archivi storici delle operazioni contabili di una banca).

1.3.3 Dispositivi d'ingresso/uscita

I dispositivi d'ingresso/uscita costituiscono i componenti del sistema di elaborazione preposti al trasferimento di dati fra il sistema (unità centrale di elaborazione e memoria centrale) e l'ambiente esterno. Per questo motivo i dispositivi sono spesso indicati anche col termine di *dispositivi periferici* o, semplicemente, *periferiche*. L'ambiente esterno può essere rappresentato sia da un supporto esterno per la memorizzazione dei dati (supporti magnetici, carta di una stampante, ecc.) o da dispositivi mediante i quali l'operatore umano può direttamente leggere i dati inviati all'esterno del sistema o scrivere dati da inviare al sistema (ad esempio terminali video e tastiere).

Come verrà illustrato con maggiori dettagli nel capitolo 5, relativo alla gestione dei dispositivi periferici, ogni dispositivo è collegato al bus del sistema tramite un particolare circuito, detto *controllore*, contenente una serie di registri ciascuno dei quali individuato da un proprio indirizzo tramite il quale può essere acceduto dal processore. L'insieme degli indirizzi dei registri di tutti i controllori viene spesso indicato come spazio di I/O del processore. Eseguendo apposite istruzioni (*istruzioni di I/O*) il processore è in grado di operare sui registri del controllore di un dispositivo e attivare quindi il dispositivo per effettuare un trasferimento di dati.

1.3.4 Meccanismo d'interruzione

Come è già stato indicato nel paragrafo 1.2.3, l'aspetto che maggiormente caratterizza il funzionamento di un dispositivo periferico è legato alla differenza di velocità con cui opera rispetto alla velocità con cui la CPU esegue le proprie istruzioni, velocità spesso inferiore di diversi ordini di grandezza a quella della CPU. Questo aspetto crea notevoli problemi di efficienza se la CPU, una volta attivato un dispositivo per leggere ad esempio dei dati, deve attendere il completamento dell'operazione di lettura prima di poter proseguire. Tale attesa può essere facilmente realizzata facendo eseguire alla CPU un ciclo di attesa all'interno del quale rimane continuando a leggere i registri del controllore per verificare la fine del trasferimento (*polling*). È per evitare questi lunghi (per la CPU) cicli di attesa che è stato introdotto il meccanismo hardware delle interruzioni. Infatti, dopo aver attivato un dispositivo il programma, invece di entrare nel ciclo di attesa perdendo inutilmente tempo, potrebbe eseguire altre sequenze di istruzioni in parallelo con l'operazione del dispositivo. Così facendo però la CPU non esegue più il test per verificare la fine del trasferimento ed è quindi necessario che sia il dispositivo ad avvertire la CPU al completamento del trasferimento inviando alla stessa un segnale (*segnale di interruzione*). Questo tipo di funzionamento del dispositivo lo si ottiene se, al momento della sua attivazione, è stato selezionato il funzionamento “a interruzione di programma”.

Nel registro *PS (Program Status Word)* della CPU è presente un bit che, col proprio valore caratterizza il funzionamento del processore nei confronti dei segnali di interruzione. Quando questo bit è posto al valore uno il processore è abilitato a ricevere segnali d'interruzione, mentre i segnali non vengono ricevuti, pur rimanendo pendenti in attesa di essere ricevuti, se tale bit ha valore zero. Questo bit può essere modificato mediante apposite istruzioni, *STI (Set Interrupt)* e *CLI (Clear Interrupt)*.

Quando la CPU è abilitata a ricevere segnali di interruzione il suo normale ciclo di funzionamento (prelievo - decodifica - esecuzione), visto nel paragrafo 1.3.1, viene modificato in modo tale che, terminata l'esecuzione dell'istruzione corrente e prima del prelievo della successiva, la CPU verifica se ci sono segnali di interruzione pendenti. Se non ci sono il ciclo prosegue come di consueto, altrimenti viene servito il segnale d'interruzione a più alta priorità (la priorità fra i segnali d'interruzione è definita in hardware). Per servire il segnale d'interruzione, il normale flusso di esecuzione del programma che la CPU stava eseguendo viene interrotto e il controllo viene trasferito a una funzione di servizio (*interrupt handler*) il cui scopo è quello di gestire l'evento associato al segnale. In pratica è come se il programma interrotto fosse obbligato, via hardware, ad eseguire un sottoprogramma (la funzione di servizio), alla fine del quale il programma può riprendere la sua esecuzione.

Per mostrare più in dettaglio il meccanismo d'interruzione e verificarne l'analogia col meccanismo delle chiamate di sottoprogramma, per prima cosa richiamiamo brevemente il funzionamento delle istruzioni di chiamata a sottoprogramma (*call*) e di ritorno da sottoprogramma (*return*) che utilizzano il meccanismo hardware della pila (stack). In figura 1.11 viene illustrato un semplice esempio nel quale un programma in esecuzione (*main*) ad un certo momento esegue la chiamata di una funzione (*proc*). L'esecuzione dell'istruzione *call* trasferisce il controllo alla prima istruzione della funzione chiamata (*l*), la quale termina eseguendo l'istruzione

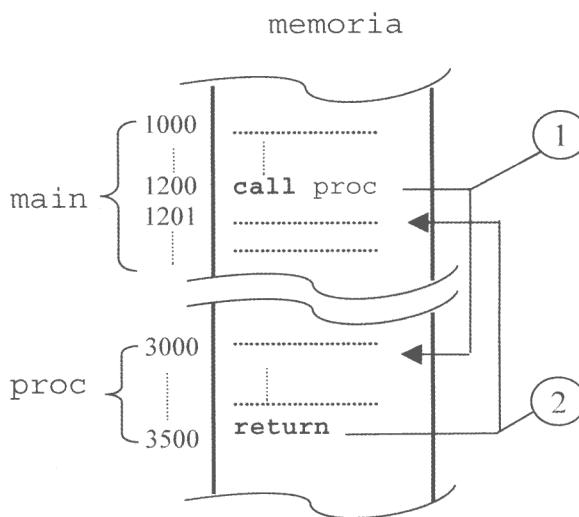


Figura 1.11 Meccanismo di chiamata e ritorno da sottoprogramma.

return, che restituisce il controllo al programma chiamante (2) che quindi continua la sua esecuzione con l'istruzione successiva alla call.

In figura 1.12 sono riportati, in particolare, i valori dei registri di macchina coinvolti: *PC* e *SP* nell'ipotesi che la pila del programma sia localizzata in memoria a partire dall'indirizzo 3900 e cresca verso le locazioni di indirizzi decrescenti.

In particolare, nella parte in alto a sinistra (1a) sono indicati i valori dei due registri dopo che l'istruzione call è stata prelevata ma prima che inizi la sua esecuzione. Nella parte in alto a destra (1b) sono riportati, invece, i valori dei registri alla fine dell'esecuzione della call. Come si può notare, l'indirizzo di ritorno è stato memorizzato in cima alla pila e nel *PC* è stato caricato l'indirizzo della prima istruzione della funzione chiamata. Analogamente, nella parte bassa, vengono riportati i valori contenuti nei registri rispettivamente all'inizio (a sinistra) e alla fine (a destra) dell'esecuzione dell'istruzione return, la quale riprende dalla pila l'indirizzo di ritorno (indirizzo dell'istruzione successiva alla call) e lo ricarica nel *PC* per riprendere l'esecuzione del programma chiamante. Ovviamente se la funzione chiamata deve usare dei registri generali, prima di usarli è necessario salvarne i valori nella pila al fine di ripristinare tali valori prima di ritornare al programma chiamante. Però queste operazioni di salvataggio e ripristino, a differenza del salvataggio e ripristino del valore del registro *PC*, vengono eseguite esplicitamente via software.

Come indicato precedentemente, l'accettazione di un segnale d'interruzione forza la CPU a trasferire il controllo a una funzione specifica, appositamente programmata per gestire l'evento segnalato tramite il segnale d'interruzione, mediante un meccanismo del tutto simile a quello visto precedentemente con la differenza che, in questo caso, viene salvato via hardware nella pila non solo il valore del registro *PC*, ma anche quello del registro *PS*. Il motivo di ciò verrà spiegato più in dettaglio nel paragrafo 1.3.5.

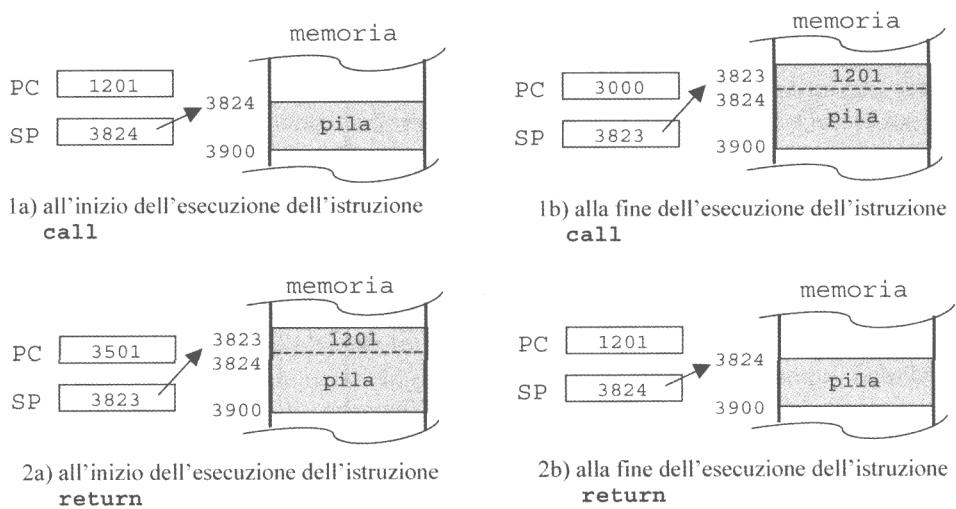


Figura 1.12 Funzionamento delle istruzioni di `call` e `return`.

Poiché tale funzione non è chiamata esplicitamente dal programma in esecuzione ma viene attivata via hardware, l'indirizzo di memoria a partire dal quale tale funzione è allocata deve essere esso stesso definito via hardware. Una semplice tecnica, spesso utilizzata, prevede che a ogni segnale d'interruzione siano riservate due locazioni a indirizzi fissi (dette anche *vettore d'interruzione*). La prima locazione è destinata a contenere l'indirizzo della funzione di gestione e la seconda il valore che dovrà essere contenuto nel registro *PS* durante l'esecuzione della funzione. In figura 1.13 viene illustrato il meccanismo hardware d'interruzione. In particolare, viene supposto che sia in esecuzione un programma (*prog*) e che durante l'esecuzione dell'istruzione di indirizzo 1200 la CPU riceva un segnale d'interruzione proveniente da un dispositivo. Il vettore d'interruzione è fissato nelle due locazioni di indirizzo *i* ed *i + 1*. Nella prima è contenuto l'indirizzo (100) della prima istruzione della funzione di gestione (*inth*), mentre nella seconda è contenuto il valore da caricare nel registro *PS* al verificarsi dell'interruzione.

Una volta terminata l'istruzione corrente e verificato l'arrivo dell'interruzione (1), la CPU trasferisce il controllo alla funzione di gestione prelevando l'indirizzo della sua prima istruzione dal vettore d'interruzione (2) e salvando i valori dei registri *PC* e *PS* in cima alla pila da cui l'istruzione di ritorno può prelevarli per ricaricarli nei registri e restituire il controllo al programma interrotto. L'istruzione di ritorno non è, in questo caso, una `return` poiché oltre che ripristinare il registro *PC* deve essere ripristinato anche il registro *PS*. Per questo motivo (3) esiste un'apposita istruzione di ritorno da interruzione (*iret*).

In figura 1.14 sono riportati i valori dei registri di macchina coinvolti: *PC*, *PS* e *SP* nell'ipotesi che la pila del programma sia localizzata in memoria a partire dall'indirizzo 3900. In particolare, nella parte in alto a sinistra (1a) sono indicati i valori

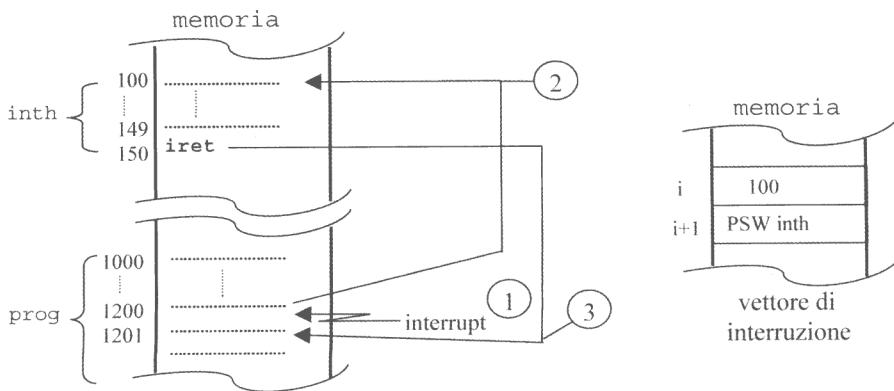


Figura 1.13 Meccanismo d'interruzione.

dei tre registri alla fine dell'istruzione durante la quale è arrivato il segnale d'interruzione. Il valore contenuto nel registro *PS* è indicato come "PSW prog". Nella parte in alto a destra (1b) sono riportati, invece, i valori dei registri dopo che è stato accettato il segnale d'interruzione. Come si può notare, in cima alla pila sono stati salvati i valori dei registri *PC* e *PS* relativi al programma interrotto e in essi sono stati caricati i valori prelevati dal vettore d'interruzione. Analogamente, nella parte bassa, vengono riportati i valori contenuti nei registri rispettivamente all'inizio (a sinistra) e alla fine (a destra) dell'esecuzione dell'istruzione *iret*, la quale recupera dalla pila i valori dei registri *PC* e *PS* per riprendere l'esecuzione del programma interrotto.

Il meccanismo delle interruzioni, come è stato indicato nel paragrafo 1.2.3, è alla base della tecnica della multiprogrammazione. Infatti, quando un programma in esecuzione attiva una periferica tramite una chiamata di sistema perde il controllo della CPU (cambio di contesto). Quando, successivamente, arriva l'interruzione di fine trasferimento il programma in esecuzione in quel momento viene interrotto e la funzione di gestione dell'interruzione viene utilizzata per indicare al sistema operativo che il programma che precedentemente aveva attivato il dispositivo è adesso in grado di tornare di nuovo in esecuzione e competere per l'uso della CPU.

Oltre alle interruzioni a cui abbiamo fatto riferimento fino ad ora, nei moderni sistemi sono previste molte altre categorie di interruzioni, ciascuna delle quali ha lo scopo di segnalare alla CPU il verificarsi di un evento che richiede la sospensione forzata del programma in esecuzione e il trasferimento del controllo a una specifica funzione di servizio del sistema operativo al fine di garantire il corretto funzionamento del sistema. La prima categoria (*interruzioni esterne*) corrisponde alle interruzioni segnalate dai dispositivi periferici a cui abbiamo fatto riferimento fino ad ora. Una seconda categoria (*eccezioni nel processore*) riguarda tutta una serie di eventi eccezionali che si possono verificare durante l'esecuzione di un'istruzione. Ad esempio, divisioni per zero, overflow aritmetici, tentativi di esecuzione di istruzioni illegali ecc. Un sottoinsieme particolarmente importante di interruzioni di questa categoria – che verrà dettagliatamente esaminato nel capitolo 5 – riguarda le eccezioni

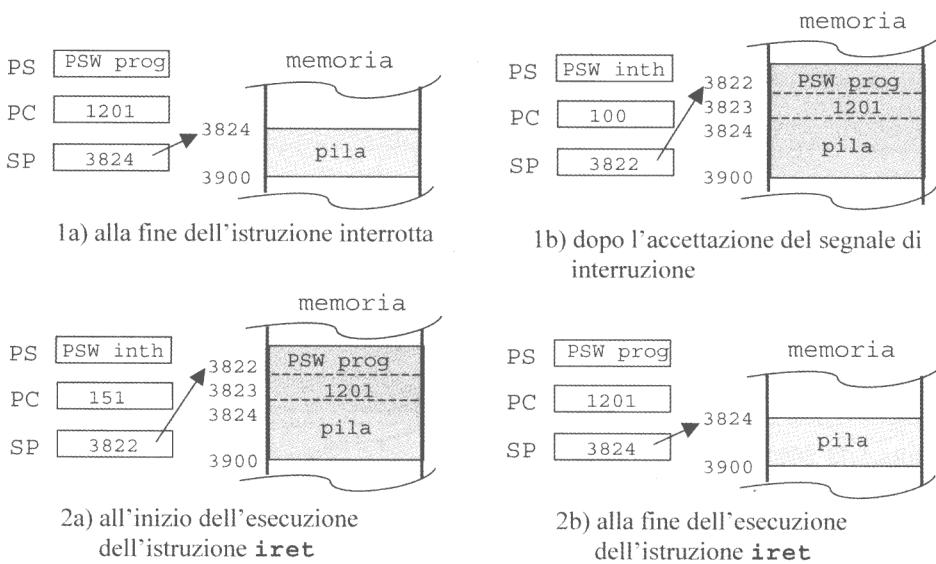


Figura 1.14 Funzionamento dell'istruzione iret.

che si possono verificare durante gli accessi alla memoria nei sistemi con paginazione e segmentazione. Una terza categoria riguarda le interruzioni che segnalano malfunzionamenti hardware (*hardware failure*) come, ad esempio, l'improvvisa mancanza di tensione (*power failure*) o un errore di parità nell'accesso ad una locazione di memoria. Infine, un'ultima categoria, estremamente importante nella realizzazione di un sistema operativo, come verrà messo in evidenza nel paragrafo successivo, è costituita dalle *interruzioni software* generate dall'esecuzione di un'apposita istruzione (istruzioni `int` o `svc – supervisor call`).

1.3.5 Accessi diretti alla memoria (DMA)

Abbiamo visto come sia possibile aumentare l'efficienza del sistema utilizzando il meccanismo d'interruzione e realizzando la tecnica della multiprogrammazione. In questo modo, dopo che un dispositivo è stato attivato, per esempio per leggere un dato, e dopo che l'operazione di lettura è terminata e il dato è quindi disponibile nei registri del controllore, è il dispositivo stesso che interrompe la CPU per richiedere il suo intervento, necessario per trasferire il dato dai registri del controllore alla locazione di memoria in cui deve essere caricato. Le istruzioni per il trasferimento del dato dai registri del controllore alla memoria fanno parte della funzione di servizio dell'interruzione. Questo modo di operare presenta ancora degli inconvenienti, in particolare nel caso in cui si voglia trasferire, invece di un solo dato, una sequenza di dati e ancora di più nell'ipotesi in cui il dispositivo da cui prelevare, o a cui trasferire, i dati sia un dispositivo veloce come è il caso dei dispositivi di memoria di massa (nastri e dischi).

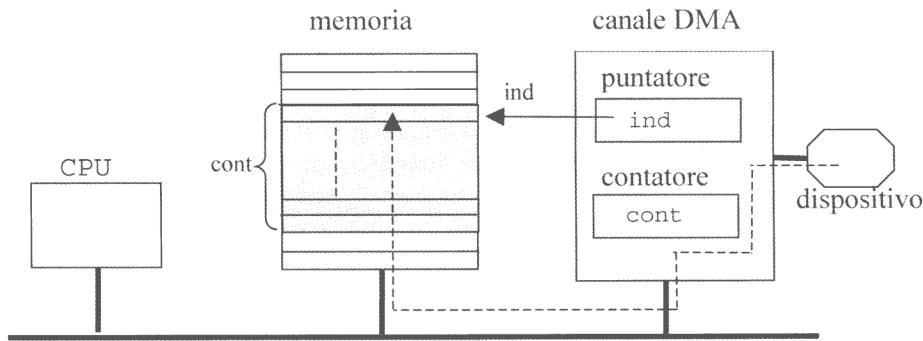


Figura 1.15 Canale di DMA.

Il primo inconveniente è legato al fatto che, per ogni dato della sequenza, deve essere generata un'interruzione per far intervenire la CPU al fine di trasferire il dato in (o dalla) memoria. Ciò implica un overhead di sistema non indifferente poiché la funzione di servizio dell'interruzione, oltre a trasferire il dato, deve anche salvare il valore degli eventuali registri di macchina utilizzati e, successivamente, ripristinare il loro valore prima di restituire il controllo al programma interrotto tramite l'istruzione `iret`. Il secondo inconveniente è legato alla velocità di trasferimento dati del dispositivo. Se questa è alta (paragonabile alla velocità di trasferimento dati fra CPU e memoria), può accadere che il tempo necessario alla CPU per la gestione di un'interruzione sia maggiore del tempo che intercorre tra due successive interruzioni dallo stesso dispositivo. In questo caso è ovvio che alcuni dati vengono perduti.

Per risolvere i due inconvenienti sarebbe necessario evitare che il dispositivo, una volta terminata la lettura di uno dei dati della sequenza (o una volta pronto per ricevere uno dei dati da inviare in uscita), invece di richiedere l'intervento della CPU tramite un'interruzione, fosse in grado di accedere da solo direttamente alla memoria per inserire il dato letto o prelevare il dato da inviare in uscita.

Questo tipo di funzionamento di un dispositivo viene reso possibile tramite i cosiddetti *canali di DMA* (*Direct Memory Access*). Un canale di DMA è un circuito che viene utilizzato per collegare un dispositivo periferico al bus del sistema e che lo abilita, come dice il nome, ad accedere direttamente alla memoria senza richiedere l'intervento del processore. In pratica tale circuito contiene due registri (vedi figura 1.15): uno (il *puntatore*) che viene inizializzato con l'indirizzo di memoria a partire dal quale inizia la sequenza dei dati da trasferire, il secondo (il *contatore*) che viene inizializzato con il numero dei dati della sequenza da trasferire.

Una volta inizializzato il canale, quando il dispositivo è pronto a trasferire uno dei dati della sequenza, il canale di DMA entra in competizione con la CPU per accedere al bus di sistema e, quando viene abilitato ad usarlo, effettua un accesso alla locazione di memoria il cui indirizzo è contenuto nel registro puntatore. Quindi il registro puntatore viene incrementato per renderlo pronto al successivo trasferimento e il registro contatore viene decrementato. Quando il contatore arriva a zero, indicando la fine dell'intero trasferimento, il canale lancia l'unica interruzione di tutta

l'operazione per indicare la fine del trasferimento. Questo tipo di funzionamento del canale viene anche indicato col nome di funzionamento in “*cycle stealing*” e cioè a *sottrazione di cicli* nel senso che, per ogni accesso alla memoria, il canale ruba un ciclo di memoria alla CPU, la quale viene quindi leggermente rallentata ma senza l'onere di dover eseguire un'intera funzione di servizio di un'interruzione.

1.3.6 Meccanismi di protezione

La contemporanea presenza di più programmi in memoria, e in contemporanea esecuzione fra di loro, crea nei sistemi multiprogrammati, e ancora di più in quelli multutente, notevoli problemi di protezione. È importante proteggere un programma da eventuali errori che si possono verificare durante l'esecuzione degli altri programmi. Particolarmente critica è poi la protezione dello stesso sistema operativo nei confronti di errori che si verificano durante le esecuzioni dei programmi applicativi. Analogamente deve essere garantita l'integrità dei dati appartenenti ad un utente nei confronti di accessi erronei o maliziosi da parte di altri utenti. Vedremo nel seguito del testo che nelle varie componenti del sistema operativo viene dedicata molta attenzione a questi problemi. In certi casi la soluzione viene offerta dal sistema operativo mediante la realizzazione di particolari meccanismi all'interno dei propri componenti (nel capitolo 6 verrà illustrata la strategia spesso utilizzata per garantire la protezione dei file residenti sul sistema). Alcuni meccanismi di base sono però generalmente disponibili su tutti i processori direttamente in hardware in modo tale da semplificare la realizzazione delle varie componenti del sistema operativo e, al tempo stesso, da renderle più efficienti.

Un meccanismo molto semplice di cui normalmente dispone il processore è rappresentato dal meccanismo di protezione della memoria. Esistono molti tipi di questi meccanismi e nel capitolo 4 alcuni di questi verranno esaminati più in dettaglio. Uno dei più semplici è costituito da una coppia di registri interni al processore (detti registri *base/limite* o *registri frontiera*) nei quali, quando un programma viene mandato in esecuzione vengono caricati, rispettivamente, l'indirizzo della prima e dell'ultima locazione dell'area di memoria nella quale il programma è caricato. Ad ogni accesso alla memoria effettuato dal programma in esecuzione, l'indirizzo della locazione a cui accedere viene confrontato con i valori dei registri frontiera. Se questo è contenuto all'interno all'intervallo compreso fra i valori dei registri frontiera l'accesso viene consentito e il programma prosegue correttamente. Se però la locazione è caratterizzata da un indirizzo esterno a tale intervallo, viene generata un'interruzione appartenente alla categoria “*eccezioni del processore*” vista precedentemente, che può essere servita in modo tale da segnalare un tentativo di accesso erroneo e comunque da prevenire che tale accesso modifichi il contenuto di locazioni appartenenti ad altri programmi.

Un altro meccanismo di protezione, importantissimo per la garanzia di integrità del sistema operativo, è rappresentato dal doppio stato di esecuzione del processore. Tutti i processori moderni presentano almeno due modi di funzionamento: il *modo utente* (*user mode*, detto anche *modo non privilegiato*) e il *modo supervisore* (*kernel mode* o *modo privilegiato*). Quando il processore gira in modo utente non è permessa l'esecuzione di alcune delle istruzioni del processore (*istruzioni privilegiate*). Ciò significa che, in questo caso, il tentativo di esecuzione di un'istruzione privilegiata

viene proibito mediante la generazione di un'interruzione appartenente alla categoria “*eccezioni del processore*”. Quando il processore si trova in modo supervisore, invece, tutte le istruzioni sono permesse. Il modo di funzionamento del processore è discriminato dal valore di un bit presente nel registro *PS*.

La necessità del doppio stato deriva proprio dalle esigenze del sistema operativo che, come abbiamo già messo in evidenza, ha il compito di gestire tutte le risorse della macchina fisica. È proprio per non vanificare le scelte implementate dal sistema operativo che è necessario “obbligare” un programma applicativo ad accedere alle risorse fisiche esclusivamente per mezzo delle chiamate di sistema. Per questo motivo, quando gira un programma applicativo, lo stato del processore deve essere quello utente, nel quale viene proibita l'esecuzione di tutte le istruzioni che modificano lo stato di una risorsa fisica (*istruzioni privilegiate*). Viceversa, una qualunque funzione del sistema operativo deve girare col processore in modo supervisore così da avere il controllo completo della macchina fisica.

Come si può facilmente intuire, le istruzioni che modificano il contenuto del registro *PS* devono necessariamente essere privilegiate per evitare che un programma applicativo, modificando il registro *PS*, possa commutare lo stato del processore da utente a privilegiato in qualunque momento. Ciò porta a un'ulteriore conseguenza: le chiamate di sistema non possono essere delle semplici chiamate di procedura, altrimenti la funzione chiamata del sistema operativo verrebbe eseguita con lo stesso valore del registro *PS* con cui stava eseguendo il programma chiamante e cioè con lo stato del processore in modo utente. Nel paragrafo 1.3.3, nel descrivere il funzionamento del meccanismo d'interruzione, è stato messo in evidenza che questo coincide con il meccanismo di chiamata di funzione con la sola variante di salvare nella pila, oltre al valore del registro *PC*, anche il valore del registro *PS* e forzare nello stesso il valore prelevato dalla corrispondente locazione del vettore d'interruzione. Per chiamare una funzione del sistema operativo può essere quindi utilizzato il meccanismo d'interruzione che consente di eseguire la funzione chiamata con un valore del registro *PS* corrispondente allo stato privilegiato. È questo il motivo per cui, nei moderni processori, sono disponibili l'istruzioni come la `int`. L'esecuzione di queste istruzioni corrisponde esattamente alla generazione di un'interruzione utilizzata, al posto della `call`, per le chiamate di sistema in modo tale che tramite tale istruzione sia possibile chiamare la funzione richiesta e, contestualmente, cambiare lo stato di esecuzione del processore.

Per lo stesso motivo per cui, in stato utente, non è possibile modificare il registro *PS*, risultano privilegiate anche le istruzioni `STI` e `CLI` che modificano il bit di *PS* relativo all'abilitazione delle interruzioni. In pratica, il modo utente in cui girano i programmi applicativi è caratterizzato dall'abilitazione della CPU a ricevere le interruzioni. Viceversa, il modo privilegiato in cui girano le funzioni di sistema è normalmente caratterizzato dalla non abilitazione della CPU a ricevere interruzioni. Questo aspetto, come sarà meglio illustrato nel capitolo 2, è legato alla necessità di realizzare le funzioni di sistema in modo tale che il loro comportamento sia analogo al comportamento delle operazioni primitive della macchina fisica a cui abbiamo già fatto riferimento.

Ogni interruzione, a qualunque categoria appartenga, corrisponde all'invocazione di una funzione del sistema operativo. Le funzioni appartenenti alle *API* di siste-

ma sono chiamate mediante le interruzioni software (*int*), le funzioni richieste su sollecitazione dei dispositivi o delle componenti hardware della macchina, tramite le interruzioni esterne o le eccezioni del processore.

Sono, infine, istruzioni privilegiate anche quelle che modificano lo stato delle altre risorse fisiche, ad esempio quelle relative alle operazioni per modificare il meccanismo di gestione della memoria (*Memory Management Unit*), che verrà illustrato con maggiori dettagli nel capitolo 4.

1.4 Struttura dei sistemi operativi

Nei primi paragrafi di questo capitolo sono state esaminate le cause che hanno determinato la nascita e lo sviluppo dei sistemi operativi, identificando le principali funzioni che vengono svolte da questo importante componente di ogni sistema di elaborazione e, al tempo stesso, caratterizzando le diverse tipologie di sistemi operativi che col tempo si sono affermate. Da quest'ultimo punto di vista possiamo, grossolanamente, classificare le tipologie dei sistemi operativi in tre grosse categorie in base al tipo di applicazioni a cui sono destinati ad offrire prioritariamente il loro supporto e quindi in base ai loro requisiti di progetto. La prima categoria corrisponde ai sistemi operativi batch, prioritariamente destinati ad offrire il loro supporto ad applicazioni di calcolo intensivo (applicazioni *CPU-bound*) e con l'obiettivo di ottimizzare l'efficienza di uso delle risorse. La seconda categoria corrisponde ai sistemi time-sharing, adatti a supportare applicazioni interattive (*I/O-bound*) con l'obiettivo di minimizzare i tempi medi di risposta. L'ultima categoria è quella dei sistemi real-time, destinati a fornire il loro supporto alle applicazioni di controllo con l'obiettivo di rispettare tutti i loro vincoli temporali (deadline). Pur nella loro diversità, alcune caratteristiche sono comuni a tutti i moderni sistemi operativi. Prima fra tutte la tecnica della multiprogrammazione, spesso indicata anche con i termini di *multitasking* o *multithreading*, come si vedrà nel capitolo 2, che rappresenta la tecnologia di base per tutte le tipologie di sistemi operativi. Da questa, come vedremo, discende poi la necessità da parte del sistema operativo di fornire tutta una serie di componenti dedicati alla gestione delle diverse risorse del sistema di elaborazione. Tali componenti, pur adottando criteri di gestione delle risorse diversi nei singoli casi al fine soddisfare gli specifici requisiti di progetto, sono comunque presenti in tutti i sistemi operativi. Lo scopo dei prossimi paragrafi è quello di passare brevemente in rassegna l'insieme dei principali componenti di un sistema operativo e, successivamente, descrivere i principali modelli strutturali adottati nel progetto di un sistema operativo per far fronte alla complessità della sua realizzazione.

1.4.1 Principali componenti di un sistema operativo

Come è stato ormai chiarito, il principale compito di un sistema operativo multiprogrammato è quello di coordinare l'evoluzione di più programmi applicativi concorrentemente. Lo strumento che il sistema ha a disposizione per svolgere questo compito è quello di allocare le risorse ai vari programmi in esecuzione in base alle loro esigenze ma anche secondo alcuni criteri che gli consentano di controllarne lo stato di avanzamento con l'obiettivo di rispettare i requisiti di progetto del sistema stesso.

Dato un programma applicativo, le esigenze di risorse di cui questo può aver bisogno durante la sua esecuzione (tempo di processore, spazio di memoria, dispositivi periferici, file ecc.) dipendono ovviamente da come il programma è stato scritto ma, per lo stesso programma, possono cambiare anche di molto fra due successive attivazioni. Si pensi ad esempio ad un programma che contiene un ciclo controllato da una condizione logica sui valori di alcune delle sue variabili. Durante una particolare esecuzione del programma può accadere che il ciclo debba essere ripetuto 10 volte prima che la condizione logica che lo controlla diventi falsa. In una seconda esecuzione, cambiando i dati d'ingresso del programma, magari il numero di iterazioni del ciclo può essere maggiore anche di diversi ordini di grandezza. Ciò significa che la seconda attivazione del programma richiederà la disponibilità del processore per molto più tempo rispetto alla prima attivazione. Analogamente ciò può accadere anche per altre risorse. Ad esempio, durante la prima attivazione, arrivato all'esecuzione di una istruzione `if...then...else`, può essere eseguito il ramo `then` all'interno del quale, supponiamo, sono contenute istruzioni per operare sia su un nastro magnetico che sulla stampante. Viceversa, nella seconda attivazione, con differenti dati d'ingresso, può essere eseguito il ramo `else` dove non viene eseguita nessuna operazione d'ingresso/uscita. Anche in questo caso le esigenze di risorse per lo stesso programma sono profondamente diverse da attivazione a attivazione. Chiaramente, come si può ben capire, ciò che il sistema operativo deve coordinare è l'evoluzione delle specifiche attivazioni dei programmi. Per questo motivo, fin dai primi sistemi multiprogrammati è stato introdotto un concetto che gioca un ruolo fondamentale nello sviluppo di un sistema operativo: il concetto di *processo*, inteso appunto come l'attività svolta dal processore durante una specifica esecuzione di un programma. È questo concetto, più che quello statico di programma inteso come sequenza di istruzioni, a giocare un ruolo essenziale nello sviluppo di un sistema operativo.

Un primo componente di ogni sistema operativo è quello dedicato alla *gestione dei processi* (*scheduler*), argomento trattato nel capitolo 2. È compito di questo componente ripartire l'uso del processore fra i vari programmi caricati in memoria in modo tale che lo stesso sviluppi un insieme di processi contemporaneamente, dando l'illusione che ciascuno di essi sia sviluppato da un diverso processore "virtuale". Sono i criteri con cui il processore viene distribuito fra i vari programmi che contribuiscono a caratterizzare il sistema operativo come un sistema batch piuttosto che time-sharing o real-time.

Per svolgere questo compito, il sistema operativo deve mantenere aggiornato lo stato di avanzamento di ciascun processo, caratterizzandolo anche in base al fatto se lo stesso sia impossibilitato a proseguire, per esempio perché in attesa del completamento di un'operazione d'ingresso/uscita, oppure se possa continuare la propria esecuzione e quindi ricevere il controllo del processore. Inoltre, deve gestire la terminazione di un processo, ad esempio in seguito al completamento dell'esecuzione di un programma applicativo e conseguentemente consentire l'attivazione di nuovi processi.

I vari processi sviluppati dal sistema sono raramente indipendenti fra di loro. Molto più spesso devono interagire, nel senso che l'avanzamento dell'uno può in certi casi influenzare l'avanzamento di un altro. Un ovvio motivo di questi mutui condizionamenti è legato al fatto che tutti i processi devono operare sullo stesso in-

sieme di risorse, per cui può accadere che una risorsa necessaria ad uno di essi sia in quel momento dedicata ad un altro. Come vedremo nel terzo capitolo, ciò comporterà la necessità di sviluppare appositi meccanismi per garantire una corretta evoluzione dei processi ed evitare interferenze sull'uso delle stesse risorse (*meccanismi di sincronizzazione*). Ma in certi casi sarà anche necessario consentire il passaggio di informazioni tra processi diversi. Riprendiamo, ad esempio, quanto visto nel paragrafo 1.2.6 relativo a un utente che, attivati contemporaneamente sia un programma per la produzione di trasparenze che un programma di videoscrittura, mediante semplici operazioni di "taglia" e "incolla" prelevi del testo da un documento realizzato col programma di videoscrittura per "importarlo" all'interno di una *slide* che sta preparando con l'altro programma. Questo passaggio di informazione dal documento di testo alla *slide* corrisponde ad un trasferimento di informazioni fra i due processi attivi, relativi alle esecuzioni dei due programmi applicativi. È come se il processore virtuale che esegue il primo dei due programmi inviasse dei dati al processore virtuale che esegue il secondo. Come vedremo nel terzo capitolo, il sistema operativo fornisce un apposito meccanismo per consentire questo scambio di informazioni tra processi, spesso indicato con l'acronimo *IPC* (*Interprocess Communication Mechanism*).

Un ulteriore componente del sistema operativo è dedicato alla **gestione della memoria principale**. Per consentire l'evoluzione concorrente di un insieme di processi è necessario che i corrispondenti programmi siano allocati in memoria principale affinché possano essere eseguiti. Ciò crea tutta una serie di problemi che sono legati alla necessità di mantenere contemporaneamente in memoria più programmi, ciascuno dei quali ha esigenze diverse.

Una parte della memoria deve necessariamente essere riservata al sistema operativo affinché possa svolgere le proprie funzioni di gestione. Normalmente però il sistema operativo è un programma molto complesso, costituito da un insieme di funzioni e strutture dati di dimensioni spesso superiori a quelle della stessa memoria. Per questo motivo soltanto una porzione della memoria principale è riservata al sistema operativo e questa stessa porzione è suddivisa in due parti. Nella prima vengono permanentemente mantenute alcune funzioni del sistema, caricate al momento dell'accensione e caricamento iniziale (*bootstrap*) e costituenti quella parte del sistema operativo nota come *parte residente*, contenente il sottoinsieme delle funzioni più spesso utilizzate. La restante parte della memoria dedicata al sistema operativo viene utilizzata per caricarci, prelevandole dalla memoria di massa, le altre funzioni soltanto quando queste sono richieste.

Tutto il resto della memoria viene gestito dal sistema per allocarvi i programmi che devono essere eseguiti in multiprogrammazione. È questa parte della memoria che costituisce una delle risorse più critiche dal punto di vista dell'efficienza dell'intero sistema. Come infatti è stato illustrato nel paragrafo 1.2.3, l'introduzione della multiprogrammazione è derivata dalla necessità di non mantenere inutilizzato il processore quando il programma in esecuzione deve attendere che si verifichi un evento necessario per la sua prosecuzione. Si può facilmente intuire, come è anche illustrato nell'esempio riportato in figura 1.5, che la possibilità di garantire che il processore non rimanga inutilizzato è tanto maggiore quanto maggiore è il *grado di multiprogrammazione* e cioè il numero massimo di processi che il sistema può supportare.

Tale numero è fortemente influenzato dalle dimensioni della memoria principale nella quale i programmi devono contemporaneamente risiedere. Un criterio a cui è necessario fare riferimento per aumentare il grado di multiprogrammazione è quello di consentire a più processi che devono svolgere le stesse azioni su dati diversi di condividere il programma da eseguire caricando in memoria un'unica copia dello stesso. Si consideri, ad esempio, un sistema time-sharing nel quale molti utenti sono contemporaneamente collegati al sistema tramite i loro terminali. È chiaro che per ogni utente è attivo un processo che gestisce il colloquio col sistema controllando il rispettivo terminale. È altrettanto ovvio che tali processi eseguono tutti lo stesso programma (*shell*) e che sarebbe del tutto inutile caricarne in memoria più copie identiche occupando così, inutilmente, aree che viceversa possono essere rese disponibili per altri programmi. Una caratteristica che un programma deve possedere affinché possa essere condiviso è quella nota col termine di *rientranza* che, come dice il nome, implica la possibilità che un'esecuzione del programma possa iniziare prima che una precedente esecuzione sia terminata. Questa caratteristica implica, ovviamente, che il programma non venga modificato durante la sua esecuzione o, come viene anche detto, che sia costituito da una *pura procedura*. Si pensi ad esempio a una procedura, o funzione, scritta in un qualunque linguaggio di alto livello. La procedura può operare sui propri parametri, sulle variabili locali e sui dati globali. Se più processi devono eseguire questa stessa procedura è possibile consentire agli stessi di condividere il solo codice della procedura se ciascuno di essi ha una propria pila nella quale vengono allocati i rispettivi record di attivazione contenenti gli specifici valori dei parametri, delle variabili locali e l'indirizzo dei dati globali sui quali ciascun processo deve operare.

I problemi che il gestore della memoria deve risolvere sono molteplici e verranno esaminati in dettaglio nel capitolo 4. È necessario garantire che ciascun processo non possa erroneamente accedere alle aree di memoria in cui sono caricati programmi eseguiti da altri processi. Inoltre, non potendo prevedere durante la compilazione di un programma in quale porzione della memoria questo verrà caricato, è necessario realizzare un meccanismo, noto col nome di *meccanismo di rilocazione*, che abiliti l'esecuzione del programma qualunque sia l'area di memoria in cui questo verrà caricato. Infine, quando il grado di multiprogrammazione è tale per cui la memoria non è sufficiente a contenere tutti i programmi, è necessario realizzare un meccanismo di allocazione dinamica della memoria in modo tale che, in ogni istante, solo un sottoinsieme dei programmi da eseguire è mantenuto in memoria principale mentre gli altri sono mantenuti su memoria di massa, abilitando il sistema operativo a gestire, in forma automatica e trasparente per il programmatore, il trasferimento dei programmi fra i due livelli di memoria. In questo modo, utilizzando anche specifici meccanismi hardware di supporto, il sistema operativo crea l'illusione che ciascun processo abbia a sua disposizione una propria *memoria virtuale* esattamente con lo stesso criterio con cui vengono creati i processori virtuali.

La **gestione dei dispositivi periferici** costituisce un ulteriore componente del sistema operativo. Molte sono le funzioni svolte da questo componente e verranno illustrate in dettaglio nel capitolo 5. Fra queste, due categorie di funzioni sono particolarmente importanti per raggiungere gli obiettivi di funzionalità e di facilità di uso del sistema. Le prime sono quelle necessarie per garantire un corretto accesso ai dispositivi

da parte dei processi, coordinando tali accessi in modo tale da evitare possibili interferenze legate alla competizione per il loro uso. Le altre sono quelle destinate a mascherare la complessità e la diversità dei vari dispositivi hardware. Per questo motivo, il sistema operativo fornisce agli utenti un'interfaccia che consente al programmatore di operare sui dispositivi sollevandolo dalla necessità di conoscere tutti i loro dettagli realizzativi e quindi molto più semplice da usare. Tutte le operazioni necessarie per controllare il corretto funzionamento di un dispositivo, ivi incluse le funzioni di risposta alle interruzioni che il dispositivo può inviare al processore, sono “nascoste” all'interno di un apposito modulo del sistema (*driver del dispositivo*).

Un sistema di elaborazione non viene utilizzato esclusivamente per eseguire programmi applicativi destinati ad elaborare dati, ma anche per memorizzare dati in modo tale da costituire veri e propri archivi elettronici e, tramite questi, basi di dati. Per questo motivo ogni sistema operativo fornisce un ulteriore componente destinato alla **gestione degli archivi** (*file system*) a cui sarà completamente dedicato il sesto capitolo. Lo scopo di questo componente è ancora quello di semplificare l'usabilità del sistema da questo punto di vista, sollevando l'utente da tutti i problemi legati alla conoscenza delle tecniche di memorizzazione dei file su memoria di massa, dei criteri di individuazione dei file all'interno del sistema (*naming*), di controllo degli accessi ai file al fine di garantire protezione e privacy dei dati privati di ciascun utente e, al tempo stesso, garantire un corretto accesso ai dati condivisi. Per risolvere questi problemi, il sistema deve gestire un'ulteriore risorsa, lo spazio fisico di memorizzazione su ciascun specifico supporto dell'informazione (o *volume*, come viene spesso chiamato) e cioè su ciascun disco o nastro che è possibile montare sul sistema.

I precedenti componenti sono presenti, in varie forme, in tutti i sistemi operativi multiprogrammati. A seconda del sistema è però possibile avere anche ulteriori componenti. Ad esempio, in un sistema di elaborazione distribuito, cioè costituito da vari calcolatori collegati in rete, è possibile realizzare i sistemi operativi relativi ai singoli nodi della rete in modo tale che ciascuno di essi offre un'ulteriore funzionalità atta a rendere accessibili, e quindi condivisibili, le risorse presenti su ciascun nodo a tutti gli utenti dell'intera rete. Per questo motivo è necessario aggiungere ad ogni sistema operativo un componente destinato alla **gestione delle comunicazioni sulla rete**, con l'obiettivo di fornire una visione omogenea di un unico sistema anche se, potenzialmente, i singoli nodi sono fra loro eterogenei. Un esempio molto importante è quello destinato a offrire un unico file system distribuito su tutti i nodi della rete.

Infine, un ultimo componente è rappresentato dal cosiddetto **interprete del linguaggio di controllo** e cioè del linguaggio tramite il quale l'utente si interfaccia col sistema. Questo componente è un vero e proprio *interprete* anche se il linguaggio di controllo è normalmente molto più semplice di un normale linguaggio di programmazione. In pratica questo componente, oggi noto anche col nome di *shell*, è il successore del semplice interprete presente nei primi sistemi batch, dove il linguaggio di controllo era costituito dai pochi comandi del *job control language*. Oggi il linguaggio di controllo è molto più ricco e può essere costituito da un insieme di comandi di sistema che l'utente batte sul proprio terminale o, molto più spesso ormai, da un linguaggio grafico col quale l'utente specifica le proprie esigenze tramite i dispositivi di puntamento sul video (*mouse*). In questi casi, l'interfaccia col sistema, costituita da mouse, finestre e liste di menu, è molto più intuitiva e semplice da usa-

re. Nei primi sistemi l'interprete era parte integrante del sistema operativo. Oggi, per garantire all'utente di poter modificare semplicemente il linguaggio di controllo secondo le proprie esigenze o, al limite, di proporne uno proprio, si tende a far girare questo componente all'esterno del sistema operativo come un normale programma applicativo (vedi figura 1.1).

1.4.2 Principali modelli strutturali

Come si può facilmente intuire anche dal semplice elenco precedente relativo ai principali componenti di un sistema operativo, quest'ultimo costituisce generalmente un programma di notevole complessità e dimensione. È quindi fondamentale applicare durante il suo progetto e la sua realizzazione le più sofisticate tecniche proprie dell'ingegneria del software, al fine di garantire un risultato che goda di tutte le proprietà che caratterizzano la qualità di un sistema software: correttezza, modularità, facilità di manutenzione, efficienza di esecuzione ecc. Per questo motivo sono stati proposti, nel tempo, vari modelli strutturali a cui fare riferimento nell'organizzare i componenti durante le fasi di progetto, realizzazione e test del sistema.

I primi sistemi, a cominciare dai monitor dei primi sistemi batch, erano costituiti da un unico programma (**sistemi monolitici**) senza una particolare suddivisione dello stesso in moduli distinti. Il sistema operativo era costituito da un insieme di procedure di servizio a ciascuna delle quali corrispondeva una chiamata al sistema. Normalmente tali procedure erano scritte in linguaggio assembler per consentire un più efficiente accesso alle risorse hardware della macchina. Questo tipo di approccio al progetto del sistema poteva essere adeguato soltanto nel caso di sistemi molto semplici, come nel caso dei primi sistemi operativi e, successivamente, come nel caso di semplici sistemi non multiprogrammati, ad esempio il sistema operativo MS-DOS. Col crescere della complessità tipica dei moderni sistemi multiprogrammati questo approccio ha rapidamente mostrato tutti i propri limiti. È rimasto famoso il caso del sistema operativo IBM OS-360, un sistema batch completamente realizzato in assembler per architetture IBM 360. Tale sistema, costituito da oltre un milione di istruzioni macchina ha continuato a produrre errori per molto tempo durante tutto l'arco della sua vita [13].

Un valido approccio utilizzato per affrontare la complessità di un sistema è quello di fare riferimento a tecniche di modularizzazione in modo tale da suddividere il sistema in componenti (moduli), ciascuno destinato a fornire una delle funzionalità del sistema e realizzato usualmente in un linguaggio di alto livello seguendo i criteri tipici della programmazione strutturata (**sistemi modulari**) [71, 72]. In base a questi criteri ogni modulo è caratterizzato da una ben precisa interfaccia che specifica la funzionalità offerta dal modulo e il modo di utilizzarla, e da un corpo contenente l'implementazione del modulo, non visibile all'esterno del modulo in base alle regole di visibilità del linguaggio (vedi figura 1.16). In questo modo ogni modifica all'implementazione del modulo che mantiene inalterata la sua interfaccia, non ha nessuna influenza sul resto del programma.

Seguendo questi criteri un primo esempio di strutturazione di un sistema operativo fu quello che si ottenne modificando la struttura dei primi sistemi monolitici in modo tale da identificare i vari moduli componenti e dettagliando con cura le rispettive interfacce. I vari moduli furono distinti in due categorie (vedi figura 1.17): le

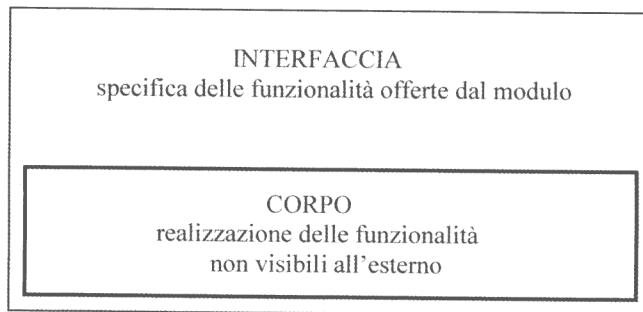


Figura 1.16 Modulo di programma.

procedure di servizio offerte dal sistema ai programmi applicativi tramite chiamate di sistema, e le procedure di utilità, utilizzate dalle prime ma non direttamente visibili ai processi. Ogni chiamata implicava, ovviamente, il cambio di stato del processore da stato utente e stato supervisore e viceversa al completamento.

Questo tipo di struttura è anche quello adottato nel sistema Unix (vedi figura 1.18). In questo caso fanno parte del sistema sia le tipiche componenti di un sistema operativo, invocate tramite le chiamate di sistema, eseguite in stato privilegiato e identificate globalmente con il termine di *kernel (nucleo)*, sia l'insieme dei programmi di utilità del sistema costituiti dalla shell, dai compilatori, caricatori, linker e dalle librerie di sistema, eseguiti in stato non privilegiato come i normali programmi applicativi.

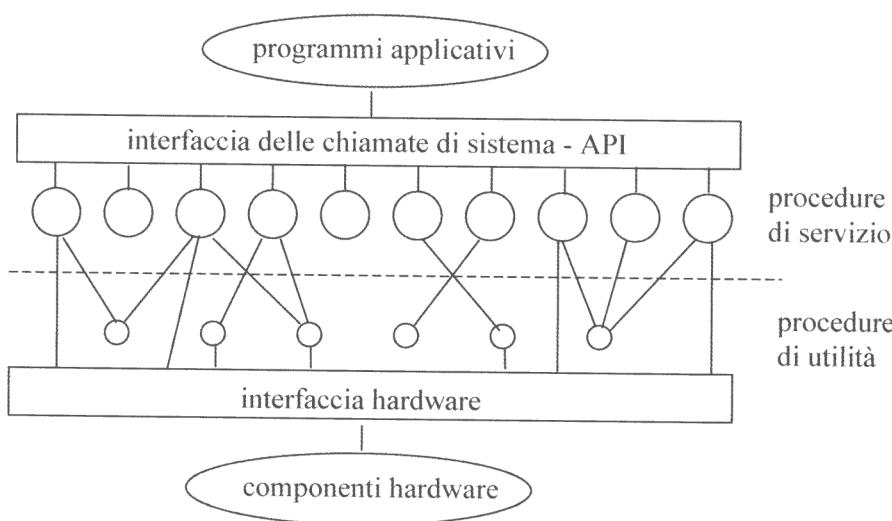


Figura 1.17 Semplice modello strutturale di un sistema modulare.

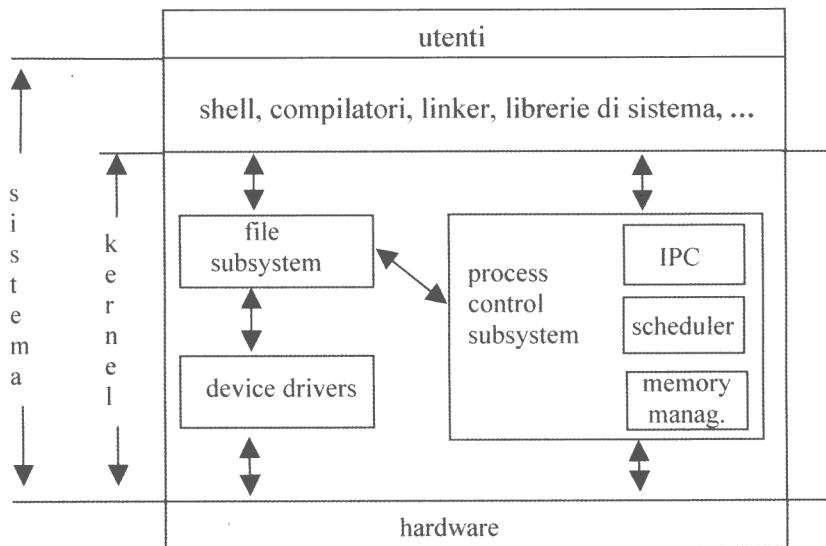


Figura 1.18 Struttura del sistema Unix.

Nonostante gli indubbi vantaggi indotti dall'uso di tecniche di modularità, la complessità dei sistemi operativi è andata progressivamente crescendo, richiedendo quindi ulteriori paradigmi di progetto in grado di affrontare in modo più idoneo la crescente complessità. Fra questi ha svolto un ruolo particolarmente importante la tecnica di organizzazione del software a *livelli gerarchici di astrazione*. Senza entrare nel merito di queste tecniche, che vanno ben oltre gli scopi del testo, e per le quali si rimanda alla bibliografia, ci limiteremo a illustrarne i concetti fondamentali. Uno degli aspetti principali della tecnica di organizzazione gerarchica consiste nel ridurre il numero di possibili interconnessioni fra i moduli di un sistema, semplificando quindi sia la fase di progetto che quella di verifica.

Tale tecnica può essere applicata seguendo due paradigmi diversi e complementari. Il primo, noto col nome di approccio *top-down*, o a *livelli di raffinamento successivi*, consiste nello scomporre il sistema software nei suoi componenti principali, identificandone le funzionalità ma astraendo dai dettagli realizzativi di tali funzionalità. Successivamente, ciascuno di questi componenti viene progettato dettagliando la sua struttura in termini di moduli di più basso livello che, a loro volta, vengono identificati mediante le loro funzionalità ma, di nuovo, astraendo da come verranno realizzati.

L'approccio duale, *approccio bottom-up*, consiste nel partire dal basso e cioè dalla macchina su cui il sistema deve essere realizzato. Vengono definiti e realizzati un certo numero di moduli software che implementano alcune funzionalità non disponibili direttamente in hardware, e costituiti da nuove operazioni (*procedure e funzioni*) e nuovi tipi di dati (*classi*). Successivamente, le nuove funzionalità possono essere utilizzate insieme a quelle direttamente offerte dalla macchina, ed astraen-

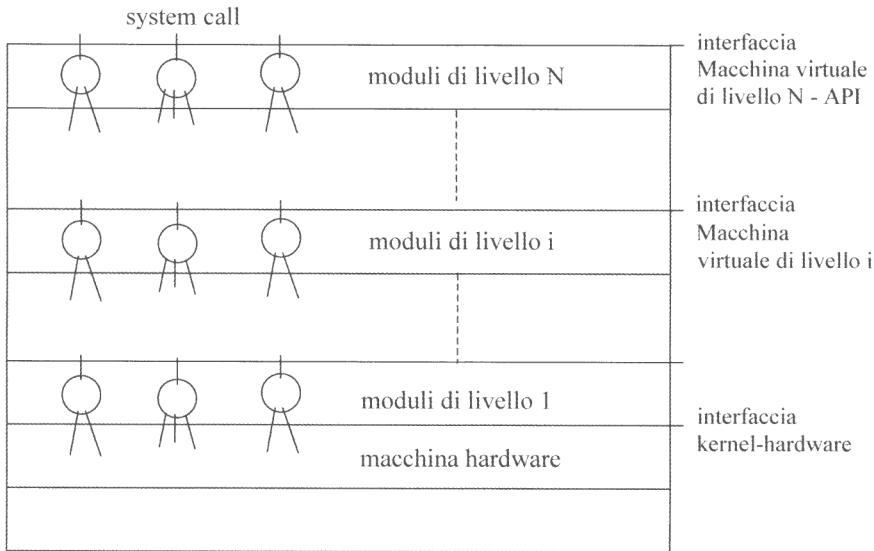


Figura 1.19 Struttura a livelli gerarchici.

do da come sono state realizzate (*funzionalità astratte*) per implementare un secondo livello di moduli che forniscono ulteriori nuove funzionalità di livello più alto, e così via fino ad arrivare ad un livello di astrazione le cui funzionalità corrispondono a quelle che devono essere offerte dall'intero sistema.

Un utilizzo combinato dei due precedenti approcci consente di strutturare il sistema in insiemi di moduli organizzati gerarchicamente in diversi livelli di astrazione (vedi figura 1.19) in modo tale che i moduli realizzati a un certo livello utilizzano esclusivamente le funzionalità offerte dai moduli di livello più basso e forniscono le loro funzionalità ai moduli di livello più alto. È come se ogni livello definisse una nuova macchina astratta, le cui funzionalità sono quelle offerte dalla macchina di livello più basso a cui si aggiungono quelle realizzate a quel livello.

Il primo sistema operativo realizzato seguendo questo modello strutturale è stato il sistema THE [32], un sistema batch sviluppato presso l'università di Eindhoven e strutturato su 4 livelli: il primo destinato a realizzare la tecnica della multiprogrammazione mediante la schedulazione della CPU (generazione dei processori virtuali), il secondo destinato alla gestione della memoria principale (generazione delle memorie virtuali), il terzo riservato alla gestione della console dell'operatore (generazione delle console virtuali), il quarto alla gestione dei dispositivi d'ingresso/uscita. Spesso, nei sistemi organizzati secondo questo modello, col termine *nucleo* o *kernel* del sistema, come si vedrà nel capitolo 2, viene indicato esclusivamente il primo livello, quello a diretto contatto con la parte hardware della macchina.

La necessità di proteggere le componenti del sistema operativo ha portato, come è stato illustrato nei precedenti paragrafi, alla necessità di introdurre il doppio stato di esecuzione e garantire che soltanto le componenti del sistema operativo possano

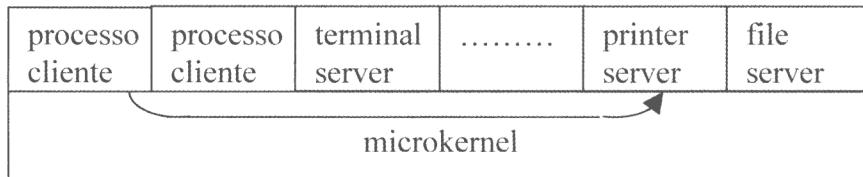


Figura 1.20 Struttura a microkernel.

girare in stato privilegiato. Ciò ha contribuito però a rendere un sistema più difficilmente modificabile ed estensibile dovendo, in questi casi, intervenire su parti protette del sistema. Modifiche o estensioni potrebbero risultare necessarie in vari casi. Ad esempio, nel caso in cui si debba aggiungere un nuovo driver in seguito alla connessione di un nuovo dispositivo; oppure nel caso in cui si ritenga opportuno modificare alcune delle scelte relative alla gestione di una o più risorse al fine di rendere il sistema più adatto ai requisiti imposti dai programmi applicativi.

Per dare soluzione a questo tipo di problemi è stata proposta una soluzione nota col nome di **struttura a microkernel**. In particolare, relativamente alla gestione di ogni risorsa, vengono definite due componenti del sistema operativo: i meccanismi che il sistema deve fornire al fine di consentire la gestione della risorsa e le specifiche strategie di gestione realizzate utilizzando i precedenti meccanismi. Ad esempio, per quanto riguarda il processore il *cambio di contesto* costituisce il meccanismo utilizzato per generare i processori virtuali, mentre lo scheduler implementa la strategia con cui il processore viene allocato all'atto di un cambio di contesto. Analogamente, per la memoria principale i meccanismi di *paginazione* o di *segmentazione*, a cui faremo riferimento nel capitolo 4, costituiscono i meccanismi utilizzati per allocare la memoria, mentre le scelte di come e quando trasferire le informazioni dalla memoria di massa alla memoria principale rappresentano le strategie di gestione della risorsa. Nei sistemi che adottano la struttura a microkernel, l'insieme dei meccanismi costituisce il microkernel del sistema, che è l'unico componente a girare in stato privilegiato. Tutte le strategie, viceversa, fanno parte di programmi di sistema che girano come normali processi applicativi. In questo senso sono facilmente modificabili ed estensibili. Uno dei primi sistemi operativi realizzati seguendo questo criterio è il sistema Mach sviluppato presso la Carnegie Mellon University [1].

Nei sistemi che seguono il modello a microkernel i gestori delle varie risorse sono quindi particolari processi di sistema, spesso indicati col nome di *server* (*file server*, *terminal server*, *printer server* ecc.). Ciò presuppone che, quando un processo applicativo (processo cliente) deve richiedere l'uso di una risorsa, debba interagire col corrispondente processo server mediante un meccanismo di comunicazione tra processi (*IPC*) fornito dal microkernel (vedi figura 1.20).

I principali vantaggi di queste strutture, come è stato precedentemente indicato, sono legati ad una maggiore modularità del sistema e ad una più semplice modificabilità e portabilità, anche se si registra una perdita di efficienza legata al fatto che ogni chiamata di sistema, in questi casi, si traduce in una comunicazione tra proces-

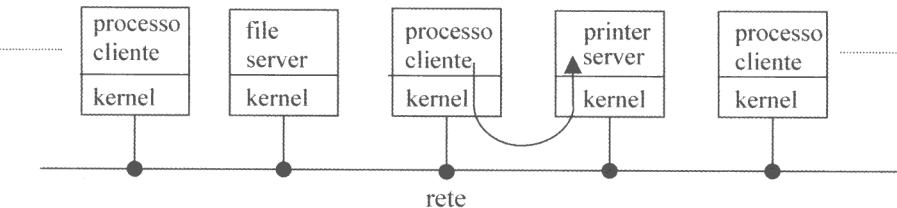


Figura 1.21 Modello client-server.

si. Per questo motivo i progettisti di Windows NT [24], che avevano originariamente realizzato il sistema secondo lo schema a microkernel, successivamente ne modificarono la struttura riportando all'interno del kernel alcune funzionalità originariamente previste all'esterno al fine di migliorare le prestazioni del sistema.

Lo schema precedente ha contribuito alla diffusione della struttura nota col nome di **struttura client-server** tipica dei molti sistemi distribuiti in rete (vedi figura 1.21).

In questi casi ciascun nodo della rete dispone di un proprio kernel. Su alcuni nodi della rete sono operativi specifici processi server, ad esempio un file server, mentre su altri girano normali processi applicativi (processi cliente). Anche in questi casi, come nell'esempio precedente, le interazioni tra processi cliente e servitore avvengono utilizzando il meccanismo di comunicazione tra processi (IPC) fornito dal kernel su ogni nodo che, a sua volta, viene implementato mediante i driver delle interfacce di rete.

1.5 Sintassi utilizzata nel testo

Nei successivi capitoli del testo quando sarà necessario, al fine di chiarire un concetto, o per meglio descrivere un algoritmo, verranno riportati programmi, o singole funzioni totalmente o parzialmente specificati. Nel presentare questi esempi non verrà fatto riferimento a nessun linguaggio di programmazione in particolare ma, dando per scontata una generale conoscenza dei principi della programmazione e dei linguaggi ad alto livello, argomenti questi che vanno oltre gli scopi di questo testo, verrà usato uno pseudolinguaggio la cui sintassi è simile a quella del C o di linguaggi con sintassi derivata dal C (come ad esempio il C++ o il Java). Verranno utilizzati, quando necessari, gli usuali costrutti per l'identificazione di blocchi, degli statement ripetitivi e alternativi. In particolare verranno utilizzate le seguenti convenzioni:

- I blocchi di codice saranno delimitati dalle parentesi graffe.
- Per le istruzioni condizionali verrà utilizzato il costrutto:

```
if (condizione) statement else statement;
```

dove la parte `else` è opzionale.

- Per le istruzioni ripetitive verranno utilizzati sia il costrutto:

```
for (statement iniziale, condizione, condizione) statement;
```

per i cicli incondizionati, che i costrutti:

```
while (condizione) statement;
e
do statement while (condizione);
```

per i cicli condizionati.

Per quanto riguarda le dichiarazioni di variabili verrà fatto riferimento alla usuale sintassi di una dichiarazione in C:

```
tipo identificatore;
```

e per accedere alle componenti di una struttura verrà utilizzata la normale “notazione a punto” (*dot notation*):

```
<identificatore di variabile>. <campo della struttura>
```

utilizzata anche per identificare le operazioni da eseguire su istanze di tipi astratti (classi). Quest’ultimo aspetto verrà approfondito più in dettaglio nel capitolo 8 relativamente al supporto offerto dal linguaggio Java per la programmazione di applicazioni con più thread.

Nelle chiamate di funzioni verranno utilizzate sia la modalità di passaggio di parametri per valore che quella per riferimento (indicato mediante l’operatore &).

Infine, tutte le componenti di un programma: tipi, variabili, funzioni, parole chiave, ecc, anche quando richiamate nel testo, verranno presentate utilizzando il tipo di caratteri *courier*.

1.6 Sommario

Il sistema operativo rappresenta un componente fondamentale di ogni sistema di elaborazione. I principali compiti che questo deve svolgere sono classificabili in due categorie. Da un lato, ha il compito di rendere più semplice e gradevole l’accesso al sistema di elaborazione da parte degli utenti. Per questo motivo offre un’interfaccia costituita da un insieme di comandi tramite i quali l’utente può agevolmente specificare i compiti che richiede al sistema. L’insieme dei comandi costituisce un semplice linguaggio tramite il quale l’utente colloquia col sistema, sia esso un generico utente che accede al sistema per richiedere l’esecuzione di programmi applicativi, sia nel caso più particolare di utenti che accedono al sistema per progettare e realizzare nuove applicazioni. Per questo motivo il sistema operativo dispone di un interprete del linguaggio di comandi (shell), che in alcuni casi si presenta come un tradizionale linguaggio simbolico mentre in altri, più recenti, come un linguaggio grafico nel quale i comandi sono specificati tramite liste di menù e icone selezionate mediante un dispositivo di puntamento sul video (mouse).

Il secondo compito del sistema operativo riguarda il controllo dell’esecuzione dei programmi applicativi allocando agli stessi le risorse di calcolo necessarie alle loro evoluzioni. Per questo motivo offre una seconda interfaccia, costituita dall’insieme delle system call che ogni programma applicativo può invocare. È l’insieme

delle strategie con cui vengono allocate le risorse che orienta il sistema verso un settore applicativo piuttosto che un altro.

Esistono oggi molte categorie di sistemi operativi, diverse le une dalle altre in base ad alcuni fattori. Il primo fattore di diversificazione riguarda il settore applicativo per il quale un sistema operativo viene progettato e realizzato. Da questo punto di vista si possono classificare i sistemi in tre principali categorie. I sistemi batch sono destinati a fornire il loro supporto prioritariamente ad applicazioni di calcolo intensivo con l'obiettivo, quindi, di ottimizzare l'uso delle risorse di macchina. La seconda categoria riguarda i sistemi time-sharing, destinati a supportare applicazioni interattive con l'obiettivo di minimizzare i tempi medi di risposta. Infine, la categoria dei sistemi in tempo reale dedicati all'esecuzione di applicazioni di controllo con l'obiettivo di rispettare tutti i vincoli temporali imposti dall'applicazione stessa. Un secondo fattore di diversificazione riguarda l'architettura hardware della macchina su cui il sistema operativo dovrà funzionare. Da questo punto di vista i sistemi operativi si possono classificare in sistemi per macchine monoelaboratore, per macchine multielaboratore e per architetture distribuite. Infine, un ultimo fattore di diversificazione riguarda i criteri di progetto che definiscono la struttura interna di un sistema operativo. Da questo punto di vista possiamo avere sistemi monolitici, modulari, strutturati a livelli di astrazione gerarchici e con architettura a microkernel.

In questo primo capitolo introduttivo è stata brevemente ripercorsa la fase di evoluzione dei sistemi operativi, cercando di classificarli in base alle loro principali caratteristiche. Inoltre, è stato messo in evidenza che, pur nella loro diversità, esiste una tecnologia di base che si è imposta come strumento fondamentale per la loro realizzazione, la tecnologia della multiprogrammazione (multitasking) che, opportunamente sfruttata, aiuta a raggiungere gli obiettivi di qualunque sistema operativo.

Con l'aumentare della complessità di un sistema operativo è cresciuta, di pari passo, la quantità di risorse di macchina che questo richiede per i propri scopi (*overhead*), sia in termini di tempo del processore che di spazio di memoria. Tale overhead può raggiungere livelli così alti da rischiare di compromettere il corretto funzionamento del sistema. Per questo motivo è andata aumentando col tempo la quantità di supporti hardware che sono finalizzati ad una efficiente realizzazione dei meccanismi del sistema operativo. In pratica, lo sviluppo dei sistemi operativi e quello delle architetture si sono vicendevolmente influenzati. Oggi sono disponibili su tutti gli attuali microprocessori molti meccanismi dedicati al supporto delle funzioni del sistema operativo (meccanismi di protezione, meccanismi per la gestione dei processi e del cambio di contesto, meccanismi per la gestione della memoria principale). Per questo motivo sono stati brevemente richiamati in questo capitolo alcuni elementi architetturali che, pur escludendo dagli argomenti del testo, vengono utilizzati nei successivi capitoli in quanto strumenti ormai indispensabili per il progetto di un sistema operativo.

1.7 Note bibliografiche

Una descrizione generale sull'evoluzione dei sistemi operativi si può trovare su alcuni testi relativi ai principi di funzionamento e di progetto di un sistema operativo quali [85], [79], [81], [96].

In [13] viene riportata l'esperienza relativa allo sviluppo di un complesso sistema non strutturato come il sistema OS/360, uno dei primi grossi sistemi batch multiprogrammati. Uno dei primi sistemi operativi time-sharing, il CTSS (*Compatible Time-Sharing System*) sviluppato al MIT, è descritto in [90]. In [21] viene descritto il sistema MULTICS (MULTIplexed Information and Computing Services) sviluppato ancora al MIT e che rappresenta una delle pietre miliari nello sviluppo dei sistemi operativi. In [6] viene descritto il sistema UNIX. La versione BSD di UNIX è descritta in [23], mentre la versione System V in [5]. In [30] è descritta la struttura del sistema VMS della Digital. Una descrizione approfondita di LINUX si può trovare in [17], mentre molte ulteriori informazioni realtive al sistema LINUX si possono trovare sul sito <http://tldp.org/guides.html>. In [86] viene fornita una breve descrizione delle caratteristiche dei sistemi real-time con particolare riferimento al sistema SPRING, mentre in [14] vengono affrontati i problemi relativi agli algoritmi di scheduling e di gestione delle risorse in questo tipo di sistemi. Informazioni e dettagli su alcuni sistemi operativi real-time si possono trovare su alcuni siti.

In particolare sul sito web <http://www.windriver.com/products/vxworks5/> della società Windriver si trova la descrizione del sistema VxWorks, mentre sul sito web <http://www.linuxworks.com/> si può trovare una versione real-time di Linux. Il primo sistema per personal computer, MS-DOS, è descritto in [67]. In [4] si può trovare la descrizione del sistema operativo Macintosh e in [24] quella di Windows, mentre in [82] viene descritta la struttura di Windows NT. Una panoramica sui sistemi distribuiti si trova in [93], mentre in [97] è disponibile una trattazione più approfondita dei paradigmi di programmazione distribuita.

Gli argomenti relativi agli aspetti architetturali e ai supporti hardware per il sistema operativo possono essere approfonditi su un testo di architettura dei sistemi di elaborazione, come ad esempio in [38], [22], [73], [84].

In [71] e [72] vengono descritte le tecniche di modularizzazione di un sistema software, mettendo in evidenza l'importanza di separare la specifica di un modulo dalla sua implementazione e quindi l'organizzazione gerarchica di un sistema. In [32] si può trovare la descrizione del sistema operativo THE, il primo organizzato a livelli gerarchici, mentre in [1] si trova la descrizione di Mach, un sistema "UNIX-like" organizzato mediante la struttura a microkernel.

2

Gestione dei processi

Come si è visto nel primo capitolo, in un sistema multiprogrammato più programmi possono essere caricati contemporaneamente nella memoria principale ed eseguiti in modo *concorrente*. La CPU passa cioè da un programma all'altro, eseguendo ciascuno di essi per decine o centinaia di millisecondi. Per indicare questo alternarsi della CPU fra i vari programmi si parla, talvolta, di *pseudo parallelismo* per distinguerlo dal vero parallelismo hardware ottenibile solo nel caso dei sistemi *multiprocessore*, dotati cioè di più CPU.

Per tenere traccia delle molte attività parallele derivanti dall'esecuzione “contemporanea” di un insieme di programmi i progettisti dei primi sistemi multiprogrammati hanno introdotto il concetto di processo.

In termini elementari, un processo può identificarsi con un programma in esecuzione; poiché esso utilizza in modo non continuativo la CPU, il sistema operativo deve provvedere a salvare lo stato della sua esecuzione quando questa viene interrotta e a ripristinarlo quando riacquista il controllo della CPU. Per effetto di questa attività del sistema operativo è come se ogni processo possedesse la sua *CPU virtuale* dotata del proprio program counter e dell'insieme di registri contenenti le informazioni necessarie per la sua esecuzione. Se il processo è in esecuzione il program counter e i registri coincidono con quelli della macchina fisica, diversamente sono temporaneamente contenuti in un'apposita area a esso dedicata e gestita dal sistema operativo.

Obiettivo di questo capitolo è, dopo aver meglio precisato la definizione di processo e le sue proprietà fondamentali, mostrare come viene realizzata l'astrazione *CPU virtuale* e come, per effetto di questa astrazione, aspetti quali sospensione e riattivazione dei processi, criteri di assegnazione della CPU e trattamento dei segnali di interruzione possono essere ignorati dai singoli processi. Verrà mostrato, inoltre, come la CPU virtuale risulti arricchita, in termini di istruzioni elementari, di primitive per la comunicazione e sincronizzazione con gli altri processi (compresa la comunicazione con i dispositivi di ingresso/uscita visti come processi esterni) e per la creazione ed eliminazione dei processi.

La parte del sistema operativo cui è assegnato il compito della realizzazione dell'astrazione CPU virtuale prende il nome di *nucleo* o *kernel* (*process manager*) del sistema operativo. Il nucleo costituisce la parte più interna di un sistema operativo e si interfaccia direttamente con l'hardware del sistema di elaborazione. Nel seguito verranno descritte le proprietà del nucleo, precisandone le funzioni e le strutture dati sulle quali opera, limitatamente al problema della gestione della assegnazione della CPU ai processi.

Ovviamente i processi necessitano per la loro esecuzione di altre risorse sia fisiche (memoria, dispositivi di I/O) che software (file, pool di buffer). Le tecniche con le quali il sistema operativo provvede ad assegnare tali risorse ai processi saranno presentate nei capitoli successivi.

2.1 Definizione di processo

Per comprendere il significato del concetto di processo è necessario prima di tutto precisare la sua relazione con il termine *programma*. Un programma rappresenta la descrizione del procedimento logico (*algoritmo*) che deve essere eseguito per risolvere un determinato problema, mediante un opportuno formalismo (il *linguaggio di programmazione*) che ne rende possibile l'esecuzione da parte di un particolare elaboratore. Con tale termine si intende un'entità astratta, parzialmente realizzata in hardware e parzialmente in software, in grado di eseguire algoritmi specificati in un linguaggio di alto livello. Parleremo quindi di elaboratori *C*, *Java* ecc. Le informazioni e gli algoritmi vanno codificati in termini di strutture dati e costrutti linguistici offerti dal linguaggio.

Nel caso del linguaggio C, ad esempio, l'elaboratore astratto viene realizzato tramite il compilatore del linguaggio in termini di un elaboratore reale e di un insieme di procedure di supporto (*supporto a tempo di esecuzione*) che simulano quelle operazioni del linguaggio sorgente che non sono direttamente traducibili in operazioni macchina. Ad esempio, le istruzioni di ingresso-uscita del linguaggio vengono tradotte dal compilatore sfruttando le procedure fornite dal sistema operativo per la gestione delle periferiche e dei file. Queste procedure devono quindi coesistere in memoria con il programma tradotto quando lo stesso verrà eseguito.

Un discorso analogo vale per il linguaggio Java, la cui macchina virtuale (*Java Virtual Machine*, *JVM*) è un calcolatore astratto che consiste di un interprete che esegue il codice prodotto dal compilatore del linguaggio Java (*bytecode*) e di un caricatore delle classi utilizzate come supporto a tempo di esecuzione.

Un elaboratore quando è in esecuzione produce una *sequenza di eventi*, dove il termine *evento* in questo contesto corrisponde all'esecuzione di un'operazione tra quelle appartenenti all'insieme che l'elaboratore sa riconoscere ed eseguire. Parliamo di sequenza in quanto nel seguito faremo l'ipotesi che gli elaboratori presi in considerazione siano dispositivi sequenziali, in grado cioè di eseguire una operazione alla volta sotto il controllo di un programma memorizzato.

La sequenza di eventi cui dà luogo un elaboratore quando opera sotto il controllo di un particolare programma, prende il nome di *processo sequenziale* o, più semplicemente, *processo*.

Esiste quindi una distinzione concettuale tra programma e processo: mentre un

programma descrive un algoritmo, il processo è quell'entità astratta che identifica l'attività dell'elaboratore relativa all'esecuzione del programma.

Esistono molti modi per schematizzare un processo. Ad esempio potremmo identificarlo con la sequenza di stati attraverso i quali passa l'elaboratore durante l'esecuzione di un programma. Questa sequenza prende il nome di *storia del processo* o *traccia dell'esecuzione del programma*.

Facciamo un esempio specifico per chiarire ulteriormente i concetti di programma e di processo. Si consideri il seguente programma scritto in linguaggio C che valuta, adottando un opportuno algoritmo, il massimo comun divisore (M.C.D) tra due numeri naturali x e y :

```
{
    a=x; b=y;
    while (a != b)
        if (a > b) then a=a - b;
        else b=b - a;
}
```

dove a e b sono due variabili locali al programma. Evidentemente il programma termina quando i valori di a e b coincidono con il M.C.D. di x e y .

Per schematizzare il processo svolto da un elaboratore C durante l'esecuzione del precedente programma, possiamo ricostruire la storia del processo evidenziando gli stati attraverso i quali passa l'elaboratore in termini di valori delle variabili coinvolte. Per esempio, il processo svolto quando il programma viene eseguito con i valori di ingresso $x=18$ e $y=24$ è mostrato in figura 2.1.

	Stato iniziale					Stato finale
x	18	18	18	18	18	18
y	24	24	24	24	24	24
a	-	18	18	18	12	6
b	-	-	24	6	6	6

Figura 2.1 Storia del processo.

Ovviamente il processo sarebbe stato diverso se diversi fossero stati i valori di ingresso. Ciò mette in luce un'ulteriore distinzione tra i due concetti di programma (*entità statica*) e processo (*entità dinamica*). Possiamo cioè dire che un programma descrive in realtà non un processo, ma un insieme di processi (*istanze del programma*), ognuno dei quali è relativo all'esecuzione del programma da parte dell'elaboratore per un ben determinato insieme di dati di ingresso.

2.2 Stati di un processo

Mentre un processo è in esecuzione, è soggetto a *transizioni di stato* definite in parte dall'attività corrente del processo stesso ed in parte da eventi esterni asincroni con la



Figura 2.2 Stati di un processo.

sua esecuzione. Da un punto di vista astratto, se esistessero tante unità di elaborazione fisiche quanti sono i processi, gli stati di un processo sarebbero *attivo* o *bloccato*, come descritto in figura 2.2.

Un processo attivo transita nello stato di bloccato (*sospensione*) quando attende il verificarsi di qualche evento. Per esempio un processo può sospendersi in attesa di un'interruzione da parte di un dispositivo, oppure in attesa di ricevere un messaggio, oppure in attesa della sincronizzazione con un altro processo. Il significato dello scambio messaggi e della sincronizzazione sarà chiarito più avanti. Il processo può transitare allo stato di bloccato anche per una decisione presa autonomamente dal sistema operativo quando, ad esempio, l'esecuzione di una istruzione genera una condizione d'eccezione (riferimento ad un indirizzo di memoria virtuale che non si trova in memoria principale ecc.).

Il processo è riattivato quando l'evento atteso si è verificato (*riattivazione*). Lo schema tipico è quello in cui un altro processo in esecuzione genera l'evento atteso (segnale di sincronizzazione, messaggio). L'interruzione da parte di un dispositivo esterno si può considerare, come si vedrà nel capitolo 5, come un segnale inviato da un processo esterno.

In generale il numero di unità di elaborazione è inferiore al numero dei processi. Nel seguito, per semplicità, supporremo di operare su un sistema dotato di una sola unità di elaborazione fisica (sistema *monoprocesso*). In questo caso, di tutti i processi attivi uno solo può utilizzare l'unità di elaborazione in ogni istante; tutti gli altri sono pronti per essere eseguiti ma, in realtà, in attesa essi stessi che venga loro assegnata l'unità di elaborazione. Gli stati di un processo possono quindi essere descritti come illustrato in figura 2.3.

Lo stato *attivo* viene suddiviso in due stati, *pronto* ed *in esecuzione*. La transizione dallo stato di pronto a quello di esecuzione (*assegnazione dell'unità di elaborazione*) avviene quando, in seguito al blocco del processo in esecuzione, il processo viene scelto tra tutti i processi pronti per essere eseguito. La transizione dallo stato di esecuzione a quello di pronto è chiamata *revoca* o *prerilascio* (*preemption*) e può avvenire per vari motivi. Ad esempio, nei sistemi in cui la scelta del processo da mandare in esecuzione è fatta in base ad una priorità fra i processi, può accadere che il processo in esecuzione (quello a priorità più alta rispetto a tutti i processi pronti) generi un segnale che sveglia un processo a priorità più alta della sua. In questo caso l'unità di elaborazione viene revocata al processo in esecuzione per essere assegnata al processo svegliato.

Figu

In al
quan
re (ti
inter
zione
pront

Il
tutti
proce
è assi
avanz

Pe
e ter
può a
in un
intend
zione
decide

Lo
proces
delle s
un uso
tempo
secuzio
stato d

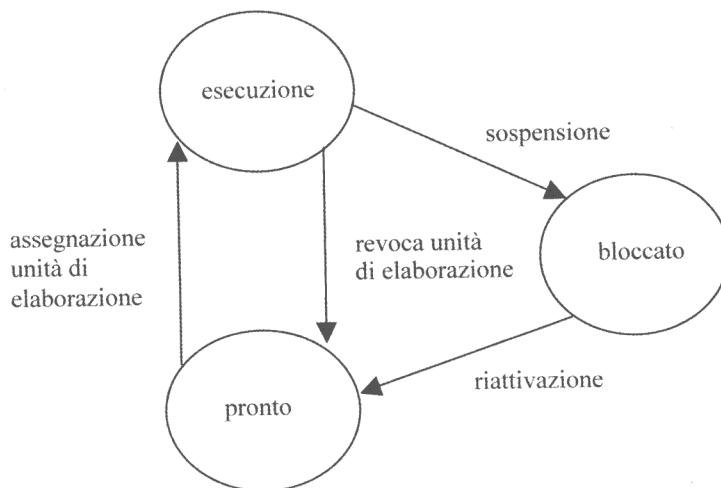


Figura 2.3 Stati di un processo in sistemi in cui il numero di processi supera il numero delle unità di elaborazione.

In altri sistemi in cui l'unità di elaborazione viene assegnata a divisione di tempo, quando un processo va in esecuzione viene inizializzato un dispositivo temporizzato (*timer*) che lancia un'interruzione dopo un certo intervallo di tempo. Se durante tale intervallo il processo non si è bloccato, all'arrivo dell'interruzione l'unità di elaborazione viene revocata dal sistema operativo per essere assegnata ad un altro processo pronto. Il processo precedentemente in esecuzione transita nello stato di pronto.

Il passaggio dallo stato di pronto a quello in esecuzione comporta una scelta tra tutti i processi pronti. Questa scelta viene compiuta da una funzione di gestione del processore (*scheduling*) che fa parte del nucleo del sistema operativo il cui compito è assicurare che tutti i processi pronti possano avanzare controllandone la velocità di avanzamento.

Per completare il grafo di figura 2.3 occorre aggiungere due ulteriori stati, *nuovo* e *terminato*. Lo stato *nuovo* corrisponde alla creazione di un nuovo processo che può avvenire, ad esempio, per effetto del collegamento al sistema di un nuovo utente in un sistema interattivo o per la richiesta da parte di un processo già esistente che intende delegare al nuovo processo parte del compito che deve svolgere. La transizione dallo stato di creato allo stato di pronto avviene quando il sistema operativo decide che il nuovo processo può rientrare tra quelli eseguibili.

Lo stato *terminato* può corrispondere o ad una terminazione normale in cui il processo esegue una chiamata al sistema operativo per indicare il completamento delle sue attività, oppure ad una terminazione anormale provocata, ad esempio, da un uso scorretto delle risorse (superamento dei limiti di memoria, superamento del tempo massimo di utilizzo della CPU consentito per il processo ecc.) oppure dall'esecuzione di istruzioni non consentite o non valide. Il processo transita, quindi, dallo stato di esecuzione allo stato terminato.

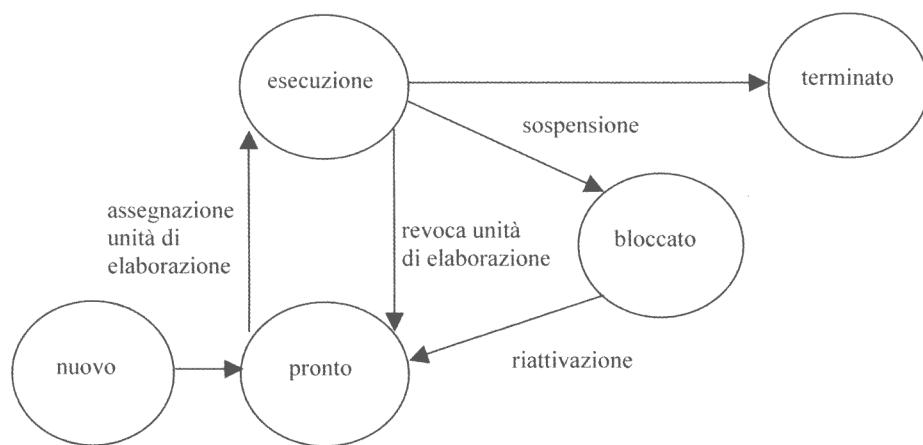


Figura 2.4 Modello a cinque stati.

Si ottiene in questo modo il modello a cinque stati riportato in figura 2.4. Si noti, per concludere, che in taluni sistemi è prevista la possibilità per un processo presente in memoria principale di essere temporaneamente spostato in memoria secondaria (*swapping*) in modo da liberare spazio per altri processi (vedi capitolo 4). Le motivazioni di questa scelta sono generalmente legate a motivi di efficienza: i processi che vengono spostati in memoria secondaria sono generalmente bloccati in attesa del verificarsi di qualche evento e quindi impossibilitati ad usare la unità di elaborazione. Agli stati precedenti va quindi aggiunto un ulteriore stato corrispondente ad un processo temporaneamente spostato in memoria secondaria.

2.3 Descrittore di un processo

Ad ogni processo è associata una struttura dati, detta descrittore del processo (*PCB—Process Control Block*). I descrittori dei processi sono a loro volta organizzati in una tabella, chiamata comunemente *tavella dei processi*.

I dati utilizzati per descrivere i processi dipendono dalle scelte di progetto del sistema operativo e dalla struttura della macchina fisica. In linea del tutto generale le informazioni da registrare nel descrittore possono essere così classificate:

- Nome del processo.** La scelta più comune è quella di identificare ogni processo con un indice, che coincide con quello della sua posizione nella tabella dei processi. Quando i processi sono organizzati in gruppi, oltre al nome è necessario registrare anche il gruppo di appartenenza, come si vedrà parlando del sistema operativo UNIX (vedi capitolo 7).
- Stato del processo.** Questa informazione può essere registrata esplicitamente nel descrittore o risultare implicitamente dall'appartenenza del descrittore ad una delle code gestite dal nucleo del sistema operativo (paragrafo 2.4).

- c) *Modalità di servizio dei processi.* La scelta a quale processo pronto assegnare l'unità di elaborazione può essere effettuata secondo diversi criteri. Il più semplice è quello FIFO (*First-In-First-Out*), che prevede l'assegnazione dell'unità di elaborazione al processo pronto in attesa da più tempo. Diversamente ad ogni processo può essere assegnata una priorità, fissa o variabile, contenuta nel suo descrittore che indica la sua importanza relativa nei confronti degli altri processi. Tale informazione può essere espressa anche in termini di intervallo di tempo in cui l'esecuzione del processo deve essere portata a termine (*deadline*). Nel descrittore del processo, in questo caso, è contenuto un valore che, sommato all'istante della richiesta di servizio da parte del processo, determina il tempo massimo entro il quale la richiesta deve essere soddisfatta (*sistemi in tempo reale*). Per modalità di servizio del tipo a suddivisione di tempo, nei descrittori è contenuto un valore che rappresenta il quanto di tempo consecutivo che l'unità di elaborazione può dedicare allo stesso processo.
- d) *Informazioni sulla gestione della memoria.* Contiene le informazioni necessarie ad individuare l'area di memoria principale nella quale sono caricati il codice ed i dati del processo. Queste informazioni si possono esprimere attraverso i valori dei *registri base e limite*, le *tabelle delle pagine* o le *tabelle dei segmenti*, secondo il sistema di gestione della memoria usato dal sistema operativo (vedi capitolo 4).
- e) *Contesto del processo.* Tutte le informazioni contenute nei registri di macchina all'atto della sospensione dell'esecuzione di un processo vengono salvate nel suo descrittore. Tali informazioni (*contesto*) riguardano il contatore di programma ed un insieme di registri che variano a seconda del tipo di architettura dell'elaboratore. Essi comprendono registri contenenti informazioni relative ai codici di condizione (*PS – Program Status word*), di cui un esempio è il registro EFLAGS del Pentium [48], puntatore allo stack (*stack pointer*), registri indice, accumulatori, registri di uso generale. Il contesto viene ricopiatto dal descrittore ai registri quando il processo torna in esecuzione.
- f) *Utilizzo delle risorse.* Queste informazioni comprendono la lista dei dispositivi di I/O assegnati al processo, i file aperti, il tempo di uso della CPU ecc.
- g) *Identificazione del processo successivo.* Come si è detto, a seconda del loro stato (pronto o bloccato), i processi vengono inseriti, in apposite code. Ogni descrittore contiene pertanto il nome del processo successivo nella stessa coda.

Le informazioni contenute nei descrittori sono vitali per il sistema operativo e devono essere opportunamente protette. I descrittori devono quindi essere memorizzati in un'area di memoria accessibile solo dal nucleo del sistema operativo.

2.4 Code di processi

Come si è visto, nell'ipotesi di una sola unità di elaborazione, ad ogni istante un solo processo è in esecuzione, mentre gli altri sono o bloccati, in attesa del verificarsi di un determinato evento, o pronti per essere eseguiti. In generale, l'indicazione di quale processo è in esecuzione viene mantenuta dal sistema operativo in un registro del processore il cui contenuto è il puntatore al descrittore del processo (*registro del processo in esecuzione*).

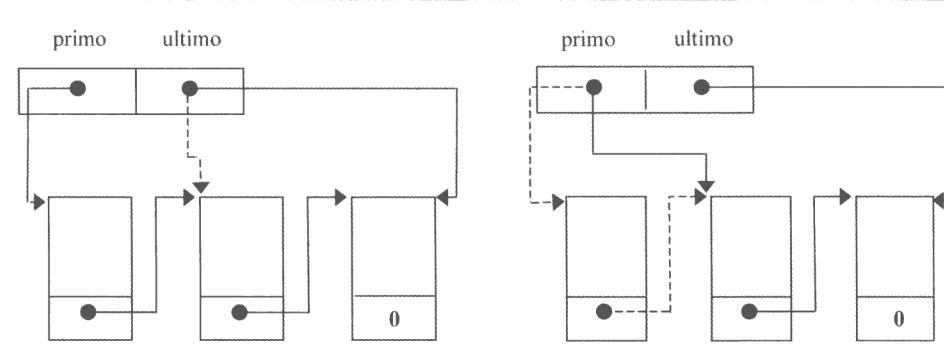


Figura 2.5 Inserimento e prelievo di un descrittore.

I processi presenti nella memoria principale che sono pronti e nell'attesa di essere eseguiti sono organizzati in una (o più) code (*code dei processi pronti*). Ad ogni coda è associato un descrittore di coda contenente una coppia di valori che rappresentano rispettivamente l'indice del primo e ultimo descrittore di processo in coda; ciascun descrittore contiene l'indicazione del processo successivo contenuto nella coda dei processi pronti.

L'operazione di inserimento di un processo in una coda registra la transizione in stato di pronto di un processo sospeso o in esecuzione; l'operazione di prelievo corrisponde al passaggio in stato di esecuzione di un processo pronto. Nell'ipotesi di un ordinamento FIFO delle code, le due operazioni sono illustrate nella figura 2.5, dove il tratteggio indica la situazione prima delle operazioni. Nella figura, a sinistra è rappresentata l'operazione di inserimento e a destra quella di prelievo.

Tra i processi pronti ne esiste uno, chiamato *processo fittizio (dummy process)*, che va in esecuzione quando tutti i processi sono temporaneamente bloccati. Il processo fittizio ha la priorità più bassa ed è sempre nello stato di pronto. Esso rimane in esecuzione fino a quando qualche altro processo diventa pronto, eseguendo un ciclo senza fine oppure una funzione di servizio di bassa priorità che non deve mai risultare bloccante per il processo.

Come si vedrà nel seguito (vedi paragrafo 2.9), è compito di un particolare componente del sistema operativo, lo *short term scheduler*, la scelta di quale processo mettere in esecuzione tra quelli pronti sulla base di prefissati criteri.

Esistono inoltre *code per i processi bloccati*, una per ogni evento o condizione per cui i processi possono attendere. A titolo di esempio, esistono code per ogni dispositivo di I/O in cui vengono inseriti i descrittori dei processi in attesa del completamento di una richiesta di I/O su quel particolare dispositivo.

In alcuni sistemi può essere presente una coda nella quale sono concatenati i *descrittori disponibili* per la creazione di nuovi processi. L'operazione di creazione estrae da questa coda il descrittore del processo da creare che verrà successivamente concatenato nella coda dei processi pronti. Viceversa l'operazione di eliminazione di un processo restituisce nella coda il descrittore del processo eliminato. In alternativa l'operazione di creazione può comportare la definizione di un nuovo descrittore e l'operazione di terminazione la sua distruzione.

2.5 Cambio di contesto

Il cambio del processo correntemente in esecuzione con un altro scelto dalla coda dei processi pronti comporta l'esecuzione di un insieme di operazioni che vanno sotto il nome di *cambio di contesto*. Tali operazioni sono:

1. Salvataggio del contesto del processo in esecuzione nel suo descrittore.
2. Inserimento del descrittore nella coda dei processi bloccati o dei processi pronti.
3. Selezione di un altro processo tra quelli contenuti nella coda dei processi pronti e caricamento del nome di tale processo nel registro processo in esecuzione.
4. Caricamento del contesto del nuovo processo nei registri del processore.

Le operazioni 1 e 4, che prendono il nome di *salvataggio stato* e *ripristino stato*, comportano il trasferimento di informazioni dai registri di macchina alla memoria centrale e viceversa. La complessità della loro realizzazione (numero dei registri su cui intervenire, tipi di istruzioni disponibili) è strettamente dipendente dal tipo di architettura del sistema.

L'operazione 2 prevede l'inserimento del descrittore nella coda associata all'evento atteso (se il cambio di contesto è dovuto alla sospensione del processo) oppure nella coda dei processi pronti (se il cambio di contesto è dovuto ad una revoca).

L'operazione 3 (*short term scheduling*) può operare sulla base di un algoritmo di tipo FIFO nel caso esista una sola coda per tutti i processi pronti o utilizzando l'informazione relativa alla priorità dei processi nel caso di più code di processi (una per livello di priorità).

Come si è detto all'inizio, in questo capitolo l'accento è posto unicamente sulle operazioni che il nucleo del sistema operativo deve compiere per consentire l'assegnazione della CPU ai processi (astrazione CPU virtuale).

Occorre ricordare, tuttavia, che il cambio di contesto comporta l'aggiornamento di tutte le strutture dati che rappresentano le risorse utilizzate dai processi: memoria, dispositivi di I/O, file aperti ecc. Questi aspetti verranno esaminati nei capitoli successivi. Va fin da ora precisato, comunque, che (in particolare per quanto riguarda la memoria) si tratta di operazioni che possono fortemente influenzare l'efficienza dell'intero meccanismo.

2.6 Creazione e terminazione dei processi

Si possono avere applicazioni nelle quali il numero dei processi è definito inizialmente e non viene più modificato durante il tempo di vita dell'applicazione. Situazioni di questo tipo si possono trovare in alcune applicazioni in tempo reale, ad esempio per il controllo di impianti fisici, in cui tutti i processi vengono creati all'avvio dell'applicazione (*creazione statica*).

In generale, tuttavia, durante la sua esecuzione un processo può creare altri processi utilizzando un'apposita primitiva fornita dal nucleo. Il processo creante prende il nome di processo *padre* ed il processo creato quello di processo *figlio*. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un albero di processi, come mostrato in figura 2.6.

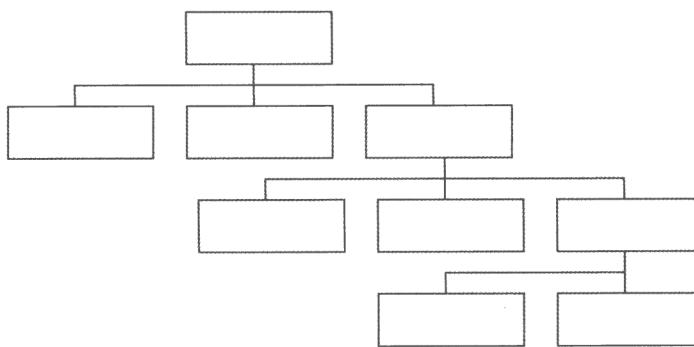


Figura 2.6 Gerarchia di processi.

Le politiche di scelta di quale programma il processo creato eseguirà, di condivisione di dati e risorse tra processi padri e figli e della loro sincronizzazione possono variare da sistema a sistema.

Anche nel caso di terminazione di un processo, questa può avvenire secondo diverse politiche di segnalazione al processo padre o più in generale ad un processo antenato.

Rinviamo alla soluzione adottata dal sistema operativo UNIX per la definizione di un esempio di politiche sia per la creazione che per la terminazione dei processi (vedi capitolo 7), si sottolinea qui come il nucleo debba offrire i meccanismi per la realizzazione delle politiche. Tali meccanismi consistono in due primitive di *creazione* e *terminazione* dei processi. La primitiva di creazione dovrà, in base ai parametri ricevuti, inizializzare il descrittore del processo da creare ed inserirlo nella coda dei processi pronti. Analogamente, la primitiva di terminazione provocherà la restituzione del descrittore alla coda dei descrittori liberi o la segnalazione che l'area di memoria può essere recuperata dal sistema operativo.

2.7 Interazione tra i processi

Con il termine *processi concorrenti* viene indicato un insieme di processi la cui esecuzione si sovrappongono nel tempo. La figura 2.7 rappresenta il caso in cui ogni pro-

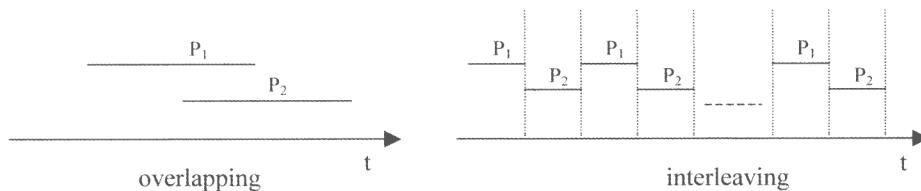


Figura 2.7 Processi concorrenti.

cesso possiede una propria unità di elaborazione (*overlapping*) ed il caso in cui i processi condividono la stessa unità di elaborazione (*interleaving*).

Una definizione generale di concorrenza che comprende entrambi i casi stabilisce che due processi si dicono concorrenti quando la prima operazione di uno inizia prima dell'ultima dell'altro.

I processi concorrenti possono comportarsi come *indipendenti* o come *interagenti*. Un insieme di processi si dice indipendente quando nessuno di essi può influenzare l'esecuzione degli altri. Processi che non condividono dati e che non si scambiano informazioni sono processi *indipendenti*. Come esempio di processi indipendenti si pensi ad un sistema di elaborazione al quale utenti diversi sottopongono in modalità time-sharing i loro programmi. I processi corrispondenti competono per l'utilizzo della unità di elaborazione, ma il nucleo del sistema operativo provvede a creare per ciascuno di essi un processore virtuale. Lo stesso succede per la competizione per altre risorse (memoria, dispositivi di I/O) che viene risolta da altre parti del sistema operativo.

Caratteristica fondamentale di un processo indipendente è che il suo comportamento è *riproducibile*; in altri termini lo stesso processo eseguito in momenti diversi, contemporaneamente ad insiemi diversi di altri processi indipendenti, produce sempre lo stesso risultato, se i dati di ingresso rimangano gli stessi.

Caratteristica dei processi interagenti è la possibilità di *influenzarsi vicendevolmente* durante l'esecuzione. Questo può avvenire in modo esplicito mediante scambio di *messaggi* o *segnali temporali* (*cooperazione*), oppure in modo implicito tramite *competizione* per la stessa risorsa (vedi capitolo 3).

Gli effetti delle interazioni dipendono dalle velocità relative di avanzamento dei processi. Poiché in generale tali velocità non sono predicibili, essendo funzione di eventi (ad esempio le interruzioni) non dipendenti dal programmatore, il comportamento dei processi *non è riproducibile*. Se si ripete l'esecuzione di un processo con gli stessi dati non si può garantire che le velocità relative siano le stesse e che quindi le interazioni avvengano nello stesso ordine.

Come esempio del primo tipo di interazione (cooperazione) si consideri il caso in cui un processo P_1 (*produttore*) produce ciclicamente un messaggio e lo inserisce in un buffer B con capacità unitaria ed un processo P_2 (*consumatore*), anch'esso circolare, preleva il messaggio da B e lo consuma (figura 2.8).

Appare evidente che esiste una sola sequenza delle operazioni dei due processi che porta ad un corretto funzionamento del sistema e cioè (nell'ipotesi che il buffer sia inizialmente vuoto) *inserimento-prelievo-inserimento-prelievo...* Ogni altra sequenza porterebbe o alla cancellazione del messaggio già depositato, ma non ancora

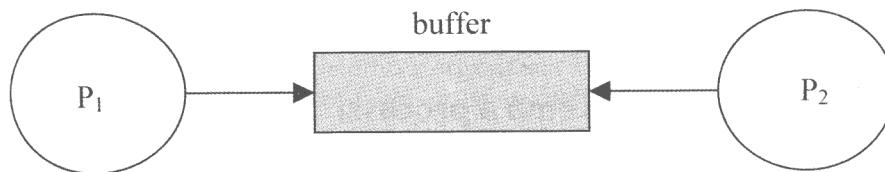


Figura 2.8 Produttore e consumatore.

prelevato (*inserimento-inserimento-prelievo*) o al prelievo ed utilizzo dello stesso messaggio più volte (*inserimento-prelievo-prelievo*).

Come esempio del secondo tipo di interazione (competizione), si consideri il caso di due processi P_1 e P_2 che condividono una variabile `contatore` che devono incrementare ognqualvolta effettuano una determinata azione. Al completamento dell'esecuzione dei processi, la variabile `contatore` deve contenere un valore pari al numero complessivo delle azioni effettuate dai due processi.

In termini di istruzioni assembler, l'istruzione:

```
contatore = contatore + 1
```

può essere espressa nel seguente modo:

```
LD contatore  
AD 1  
STO contatore
```

Ogni processo, cioè, carica il valore corrente della variabile su un proprio registro, lo modifica incrementandolo di 1 e memorizza il nuovo valore così ottenuto in `contatore`.

Se al termine di un'azione i processi eseguono concorrentemente la modifica di `contatore`, è possibile una sequenza del tipo:

t_0	:	LD contatore	(P_1)
t_1	:	LD contatore	(P_2)
t_2	:	AD 1	(P_2)
t_3	:	STO contatore	(P_2)
t_4	:	AD 1	(P_1)
t_5	:	STO contatore	(P_1)

in corrispondenza della quale il valore della variabile `contatore` viene erroneamente incrementato di una sola unità. Il risultato sarebbe corretto se il processo P_1 (P_2) avesse potuto eseguire tutte e tre le istruzioni senza essere interrotto dal processo P_2 (P_1).

Nel caso di processi interagenti è pertanto fondamentale, per un corretto funzionamento del sistema, imporre dei vincoli, diversi a seconda del tipo di interazione, nell'esecuzione di alcune operazioni (*sincronizzazione*).

Nel caso della cooperazione tra processi i vincoli riguardano l'*ordine* con cui le operazioni dei processi possono essere compiute (*sincronizzazione diretta* o *esplicita*). Nel caso della competizione occorre garantire che un solo processo alla volta abbia accesso alla risorsa comune (*sincronizzazione indiretta* o *implicita*). Come si vedrà nel capitolo 3, l'imposizione dei vincoli avviene mediante l'uso di apposite funzioni realizzate dal nucleo (*primitive di sincronizzazione e comunicazione*).

2.8 Nucleo di un sistema a processi

Le strutture dati e le funzioni illustrate nei paragrafi precedenti sono parte integrante del nucleo (*kernel* o *process manager*) di un sistema operativo il cui scopo è quello, come si è detto, di realizzare l'astrazione di CPU virtuale.

Il cambio di contesto tra due processi, la scelta del processo pronto da mettere in esecuzione, la risposta alle interruzioni provenienti dai dispositivi esterni, la realizzazione delle primitive di sincronizzazione dei processi e le primitive per la loro creazione e terminazione rappresentano le funzioni fondamentali svolte dal nucleo. Nella realizzazione di tali funzioni il nucleo utilizza come strutture dati i descrittori e le code dei processi e gli strumenti di sincronizzazione come i semafori (vedi capitolo 3).

Per quanto riguarda la gestione delle interruzioni esterne, occorre notare che il nucleo è il solo modulo del sistema operativo che è consci della loro esistenza. Come si vedrà nel capitolo 5, i processi che colloquiano con i dispositivi utilizzano opportune primitive realizzate dal nucleo, che provvedono (in generale) a sospenderli in attesa del completamento dell'azione richiesta. Quando l'azione è completata, il relativo segnale di interruzione inviato dal dispositivo all'unità di elaborazione viene catturato dal nucleo che provvede a risvegliare il processo sospeso.

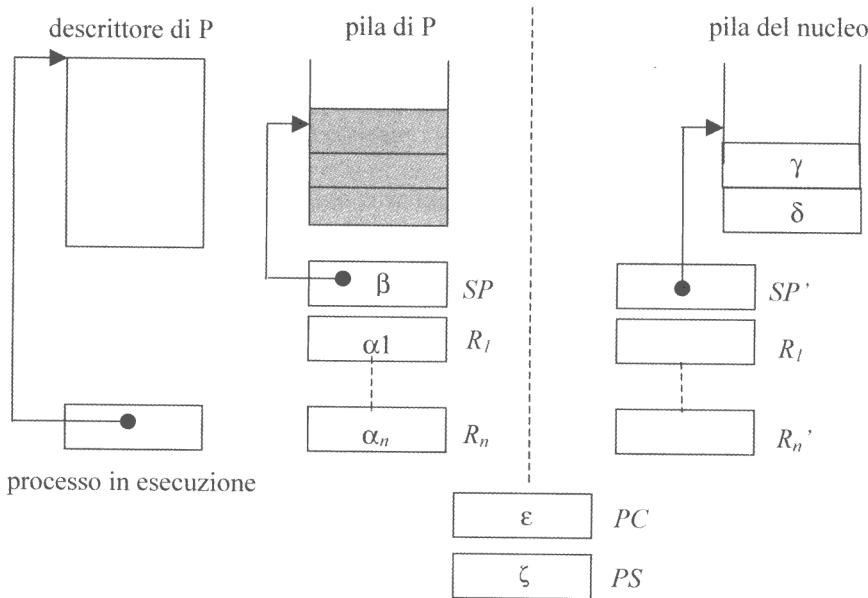
La gestione delle interruzioni è quindi invisibile ai processi ed ha come unico effetto il rallentamento della loro esecuzione sui rispettivi processori virtuali.

Una proprietà fondamentale nella strutturazione di un sistema a processi è la separazione tra meccanismi e politiche di gestione delle risorse. Il nucleo deve, per quanto è possibile, contenere solo meccanismi consentendo così, a livello di processi, di utilizzare tali meccanismi per la realizzazione di opportune politiche di gestione. Ciò consente una notevole flessibilità potendosi realizzare politiche diverse a seconda del tipo di applicazione.

Una proprietà molto importante del nucleo è l'efficienza con cui devono essere eseguite le sue operazioni. È evidente che essa condiziona l'intera struttura a processi. Per questo motivo esistono sistemi in cui alcune o tutte le operazioni del nucleo sono realizzate in hardware o mediante microprogrammi. Laddove realizzato in software, il nucleo deve essere programmato, almeno in parte, in linguaggio assembler in quanto è necessario operare direttamente sui registri dell'unità centrale.

Il meccanismo di passaggio tra i due ambienti di nucleo e dei processi è costituito dal meccanismo di interruzione. Più precisamente, nel caso di funzioni chiamate da processi esterni il passaggio all'ambiente di nucleo è ottenuto tramite il meccanismo di risposta al segnale di interruzione (*interruzioni esterne* o *asincrone*); nel caso di funzioni chiamate da processi interni, il passaggio è ottenuto tramite l'esecuzione di apposite istruzioni di tipo chiamata al supervisore, il cui effetto è del tutto analogo all'arrivo di un segnale esterno (*interruzioni interne* o *sincrone*). In entrambi i casi, al completamento della funzione richiesta, il trasferimento all'ambiente utente avviene utilizzando l'istruzione di *ritorno da interruzione* (*iret*).

Nel seguito verrà illustrato il meccanismo di trasferimento tra l'ambiente dei processi e l'ambiente del nucleo e viceversa, facendo riferimento ad una ipotetica architettura del sistema di elaborazione. In particolare si supporrà che ad ogni processo sia associata un'area di memoria privata organizzata a pila (stack) e gestita tramite il registro *SP*. La pila rappresenta l'area di lavoro del processo e contiene variabili temporanee ed i record di attivazione delle procedure chiamate. Oltre ai registri *PC* e *PS* si supporrà che l'unità di elaborazione possegga due insiemi di registri generali R_1, R_2, \dots, R_n e R'_1, R'_2, \dots, R'_n e due registri *SP* e *SP'* associati rispettivamente agli ambienti dei processi e di nucleo.



- γ valore del PC relativo al processo che ha eseguito la SVC
- δ valore del PS relativo al processo che ha eseguito la SVC
- ϵ indirizzo della procedura di risposta all'interruzione
- ζ parola di stato dell'ambiente del nucleo

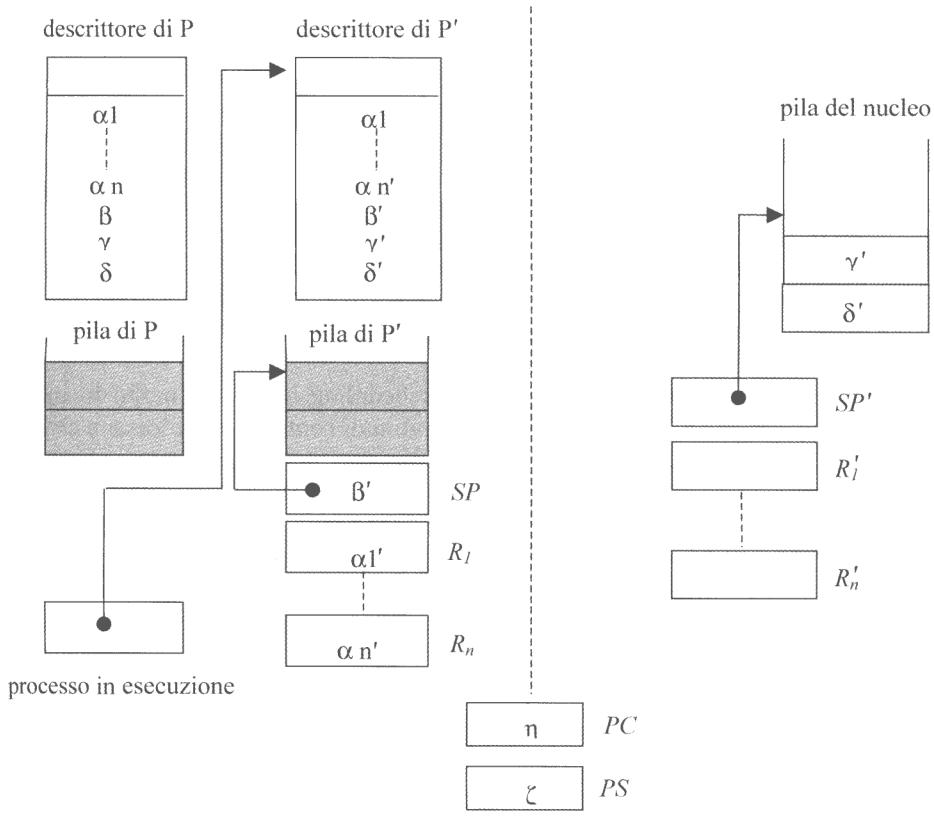
Figura 2.9 Situazione dei registri di macchina dopo l'esecuzione della SVC.

L'esecuzione di una primitiva da parte di un processo P corrisponde all'esecuzione di un'istruzione di tipo SVC (*Super Visor Call*) che genera un'interruzione di tipo sincrono, con salvataggio dei valori dei registri PC e PS relativi a P in cima alla pila del nucleo e con caricamento in PC e PS rispettivamente dell'indirizzo della procedura di risposta alle interruzioni e della parola di stato propria dell'ambiente nucleo (figura 2.9).

La procedura di risposta all'interruzione opera ora utilizzando l'insieme dei registri dell'ambiente del nucleo e, sulla base dei parametri passati, provvede a chiamare la relativa funzione richiesta.

Nell'ipotesi che la funzione richiesta non risulti bloccante per il processo, il nucleo provvede a rimettere in esecuzione il processo P mediante l'istruzione `iret` che ripristina nei registri PC e PS i corrispondenti valori γ e δ contenuti nella sua pila.

Se invece l'esecuzione della funzione comporta il blocco del processo, viene eseguita innanzitutto la procedura per il salvataggio dello stato del processo che provvede ad inserire nel suo descrittore i valori dei registri di macchina dell'ambiente processi, nonché i valori γ e δ .



η indirizzo dell'istruzione RTI (ritorno da interruzione)

ζ parola di stato dell'ambiente nucleo

Figura 2.10 Situazione dei registri di macchina prima dell'esecuzione della `iret`.

Al completamento della funzione richiesta, viene eseguita la procedura per il ripristino dello stato, che provvede a caricare nei registri di macchina dell'ambiente processi i corrispondenti valori contenuti nel descrittore del nuovo processo P' ed i valori γ' e δ' in cima alla pila dell'ambiente nucleo (figura 2.10).

Il nucleo provvede infine a rimettere in esecuzione il processo P' eseguendo l'istruzione `iret`.

2.9 Scheduling

Abbiamo visto precedentemente che con il termine *short term scheduling* si intende quella funzione del nucleo che ha il compito di scegliere a quale tra i processi pronti assegnare la CPU.

Essa interviene ogni qualvolta il processo in esecuzione perde il controllo della CPU. Ciò può avvenire quando il processo termina o viene sospeso in quanto richiede esplicitamente una funzione del sistema operativo. Si parla in questo caso di scheduling senza diritto di prelazione (*non preemptive scheduling*): è il processo stesso che abbandona l'uso della CPU.

La funzione di scheduling interviene anche quando un processo passa dallo stato di esecuzione allo stato di pronto, ad esempio per l'arrivo di un'interruzione che indica lo scadere del quanto di tempo assegnato al processo, o dallo stato di bloccato allo stato di pronto, ad esempio per il completamento di un'operazione di I/O. Nel primo caso il processo in esecuzione viene forzato ad abbandonare l'uso della CPU, nel secondo caso ciò avviene se il processo risvegliato è "più importante" del processo in esecuzione. In entrambi i casi si parla di scheduling con diritto di prelazione (*preemptive scheduling*).

Premesso che la funzione di short term scheduling interviene molto frequentemente in quanto legata, come si è visto, al cambio di contesto tra processi e che pertanto deve essere realizzata in modo molto efficiente, nel seguito verranno riportati in modo sintetico alcuni tra i principali algoritmi utilizzati nella scelta di uno tra i processi pronti.

Il più semplice tra gli algoritmi di scheduling della CPU è l'algoritmo *FCFS* (*First-Come-First-Served*) che assegna la CPU al processo pronto in attesa da più tempo. Quando un processo entra nella coda dei processi pronti il suo descrittore viene collegato all'ultimo elemento della coda. Quando la CPU è libera viene assegnata al processo il cui descrittore si trova nella testa della coda rimuovendolo da essa (*coda FIFO*).

Questo tipo di algoritmo risulta inadatto nel caso di applicazioni caratterizzate da molti processi che si sospendono frequentemente o nel caso di sistemi interattivi. Il tempo medio di attesa risulterebbe mediamente troppo elevato. L'algoritmo FCFS rientra in uno schema di scheduling senza diritto di prelazione e lo si può utilizzare nel caso di applicazioni di tipo *batch*.

L'algoritmo *Round Robin* (*RR*) è stato progettato appositamente per i sistemi a partizione di tempo. Rientra in uno schema di scheduling con diritto di prelazione. La coda dei processi pronti è realizzata come una coda circolare: ogni processo ottiene la CPU per un quanto di tempo (da 10 a 100 millisecondi) al termine del quale perde il controllo della CPU ed il suo descrittore viene inserito nella coda. La coda viene gestita con modalità *FIFO*: il processo che abbandona la CPU viene inserito in fondo alla coda e quello che la riceve è il primo della coda.

L'algoritmo *RR* è particolarmente adatto per i sistemi interattivi in quanto è in grado di assicurare tempi di risposta abbastanza brevi agli utenti, determinati esclusivamente dal valore del quanto di tempo e dal numero medio di processi pronti.

Le prestazioni del sistema possono essere fortemente influenzate dalla scelta del quanto di tempo. Il tempo di risposta è teoricamente migliore per piccoli valori del quanto di tempo. Va tuttavia considerato che il cambio di contesto tra processi, a meno che il processore disponga di efficienti supporti hardware, comporta un consumo di tempo in operazioni improduttive (*overhead*).

Fino ad ora si è supposto che tutti i processi pronti abbiano la stessa priorità. In molte applicazioni, invece, è necessario assegnare ai processi priorità diverse. Si

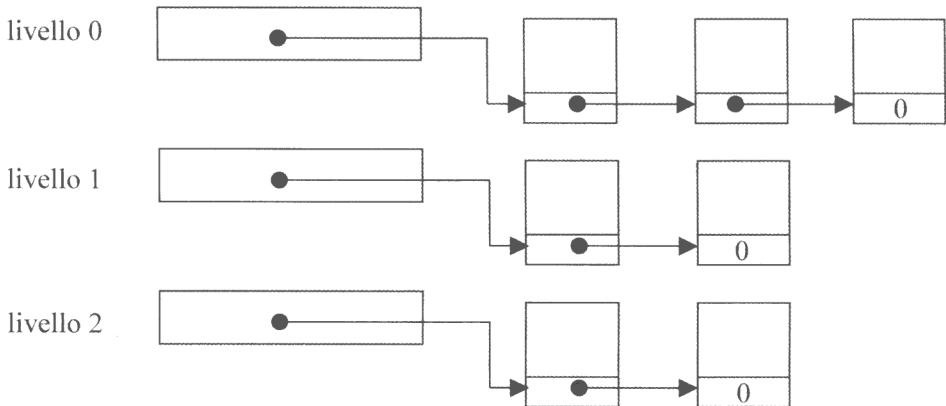


Figura 2.11 Livelli di priorità.

pensi ad esempio ad un sistema in tempo reale nel quale l'esecuzione di alcune funzioni (ad esempio il controllo di dispositivi esterni) può risultare critica e pertanto deve essere eseguita prima di altre funzioni. Le priorità sono indicate con un intervallo di numeri (da 0 a n), con la convenzione, ad esempio, che i numeri bassi indichino priorità alte. Un algoritmo di scheduling basato sulle priorità assegna la CPU al processo pronto con la priorità più alta. I processi con la stessa priorità si ordinano secondo uno schema FCFS.

Le priorità possono essere *statiche* o *dinamiche*. Sono statiche se non si modificano durante il periodo di vita dei processi. Diventano dinamiche se durante la loro esecuzione i processi possono modificare la loro priorità in funzione ad esempio del tempo di CPU già utilizzato (la priorità diminuisce al crescere del tempo di CPU utilizzato) o del tipo di operazione che il processo sta eseguendo (la priorità cresce se il processo deve eseguire una operazione di I/O).

I processi possono essere raggruppati in classi di priorità; in questo caso l'algoritmo di scheduling dovrà scegliere un processo pronto che appartiene alla classe a maggiore priorità, applicando all'interno di ogni classe un algoritmo del tipo, ad esempio, round robin.

Si consideri il caso di tre classi di priorità 0, 1, 2 in ordine decrescente (figura 2.11).

L'algoritmo di schedulazione prevede che fino a che esistono processi nella classe 0, ciascuno di questi viene messo in esecuzione per un quanto di tempo (round robin); se la classe zero è vuota vengono scelti con lo stesso criterio processi della classe 1; lo stesso per la classe 2 se la classe 1 è vuota.

Si noti che un algoritmo basato sulla priorità dei processi può portare a situazioni di attesa infinita per i processi a più bassa priorità (*starvation*). Anche per questo motivo le priorità vengono, di norma, aggiornate dinamicamente.

Un esempio di algoritmo di scheduling basato su classi di priorità con scelta del processo all'interno della classe basato su un algoritmo round robin è quello

realizzato nel sistema operativo UNIX. Le priorità sono inoltre dinamiche e vengono ricalcolate periodicamente (ad esempio ad ogni quanto di tempo) con l'obiettivo di ridurle rispetto al valore iniziale al crescere del tempo di CPU utilizzato dai processi. L'utente ha a disposizione una system call `nice` per ridursi spontaneamente la priorità.

Nell'assegnare le priorità ai processi applicativi viene spesso fatta una distinzione tra due gruppi di processi: interattivi (*foreground*) e batch (*background*). Questi due tipi di processi hanno tempi di risposta diversi e quindi diverse esigenze in termini di priorità. Inoltre, come si è accennato, hanno la massima priorità i processi il cui compito è l'esecuzione di funzioni di I/O.

Per concludere è opportuno ricordare che in un sistema operativo esistono altri due livelli di scheduling, scheduling a lungo termine (*long term scheduling*) e scheduling a medio termine (*medium term scheduling*).

Con l'espressione *scheduling a lungo termine* si intende quella funzione del sistema operativo che, in un sistema di tipo batch, sceglie, fra tutti i programmi caricati in memoria di massa per essere eseguiti, quelli da trasferire in memoria centrale e da inserire nella coda dei processi pronti. La scelta viene fatta, in generale, in modo da equilibrare la presenza in memoria principale di processi del tipo *CPU-bound* e del tipo *I/O-bound*. I primi richiedono poche operazioni di I/O ed impiegano la maggior parte del loro tempo in operazioni di elaborazione; i secondi impiegano la maggior parte del loro tempo in operazioni di I/O. È intuitibile come uno sbilanciamento del carico verso uno qualunque dei due tipi di processi porterebbe ad un uso non efficiente dell'intero sistema. Se fossero prevalenti i processi del secondo tipo (I/O bound), la coda dei processi pronti sarebbe quasi sempre vuota e la CPU non sarebbe utilizzata in modo efficiente. Se fossero prevalenti i processi del primo tipo (CPU-bound), i dispositivi di I/O sarebbero poco utilizzati ed il sistema sarebbe nuovamente sbilanciato.

Compito dello scheduling a lungo termine è anche quello di controllare il numero dei processi presenti contemporaneamente in memoria centrale (*grado di multipro-*

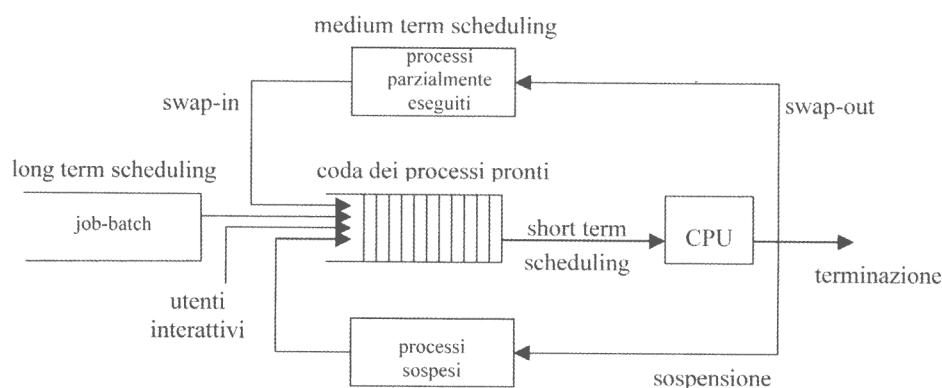


Figura 2.12 Livelli di scheduling.

grammazione). Più processi sono presenti in memoria centrale, minore è il tempo a disposizione di ogni processo per l'esecuzione. Perciò lo scheduler a lungo termine può limitare il grado di multiprogrammazione per fornire un servizio soddisfacente all'insieme di processi che deve gestire.

Lo scheduling a medio termine (*medium term scheduling*) rappresenta quella funzione del sistema operativo che ha il compito di trasferire temporaneamente processi dalla memoria centrale alla memoria di massa (*swap-out*) e viceversa (*swap-in*). La funzione viene utilizzata per liberare parte della memoria centrale necessaria ad altri processi già presenti o per rendere possibile il caricamento di altri processi così da migliorare, ad esempio, la combinazione dei processi presenti.

Entrambe le funzioni di long term scheduling e medium term scheduling vengono eseguite dal sistema operativo con una frequenza nettamente inferiore a quella dello short term scheduling e quindi è giustificabile una loro certa complessità. La figura 2.12 riassume le funzioni svolte dai tre livelli di scheduling.

2.10 Processi leggeri (threads)

Come si è visto precedentemente, oltre che dal codice di uno o più programmi, un processo è costituito da un insieme di locazioni per i dati, le variabili globali e locali, dallo stack e dal suo descrittore. L'insieme di queste informazioni prende il nome di immagine del processo. Al processo sono inoltre associate alcune risorse, come file aperti, processi figli, gestori di segnali, dispositivi di I/O ecc. L'immagine del processo e le risorse da esso possedute costituiscono il suo spazio di indirizzamento. Ogni processo ha uno spazio di indirizzamento distinto da quello di altri processi.

La locazione dello spazio di indirizzamento dipende dalla tecnica di gestione della memoria adottata. Potrà essere mantenuto completamente o solo in parte nella memoria principale, in modo contiguo o in blocchi non necessariamente contigui, di dimensioni fisse (*pagine*) o variabili (*segmenti*) (vedi capitolo 4).

In un sistema di processi concorrenti le operazioni di scambio tra due processi possano risultare onerose in termini di tempo di esecuzione (*overhead*) comportando il salvataggio e ripristino del loro spazio di indirizzamento. Lo stesso dicasi per la creazione e la terminazione di un processo. Inoltre, la *separazione* degli spazi di indirizzamento, con le conseguenti proprietà di protezione dei dati locali dei singoli processi, ne favorisce l'utilizzo nei casi di interazioni basate sullo scambio di messaggi, mentre rende complesso il loro uso nel caso di frequenti interazioni basate sull'accesso a strutture dati comuni.

D'altra parte sono numerosi i casi di applicazioni che, presentando un intrinseco grado di parallelismo, possono essere decomposte in attività che procedono parallelamente condividendo un insieme di dati e risorse comuni.

Si pensi, ad esempio, ad applicazioni in tempo reale per il controllo di impianti fisici in cui si possono individuare attività quali il controllo dei singoli dispositivi di I/O dedicati a raccogliere dati dal processo fisico o a inviare comandi verso di esso. Queste attività devono frequentemente accedere a strutture dati comuni rappresentanti lo stato complessivo del sistema da controllare.

Un altro esempio è rappresentato da un programma di elaborazione dei testi nel quale sono individuabili attività quali la scrittura sul video, la lettura dei dati immes-

si da tastiera, la correzione ortografica e grammaticale ed il salvataggio periodico del file su disco. Anche in questo caso siamo in presenza di attività concorrenti che operano sugli stessi dati (il testo che viene prodotto).

Per ottenere una soluzione efficiente a questi problemi è stato introdotto nei moderni sistemi operativi il concetto di *thread* o *processo leggero*. Un thread rappresenta un *flusso di esecuzione* all'interno di un processo (*processo pesante*). Come vedremo, all'interno di un processo è possibile definire più threads (*multithreading*), ciascuno dei quali condivide le risorse del processo, risiede nello stesso spazio di indirizzamento e accede agli stessi dati.

Le singole attività in cui si decompone l'applicazione che condividono gli stessi dati possono essere rappresentate mediante threads.

Non possedendo risorse, i threads possono essere creati e distrutti più facilmente rispetto ai processi ed inoltre il cambio di contesto tra due threads risulta molto più efficiente.

Alla base del concetto di thread sta la constatazione che il concetto di processo che abbiamo esaminato è basato su due aspetti indipendenti: *possesso delle risorse* ed *esecuzione*. Come si è visto, ogni processo ha il proprio spazio di indirizzamento che contiene l'immagine del processo e le risorse ad esso assegnate; al tempo stesso rappresenta un *flusso di esecuzione*, all'interno di uno o più programmi, che condivide l'unità di elaborazione con altri flussi, possiede uno stato e viene messo in esecuzione sulla base della politica di scheduling adottata dal sistema operativo.

I due aspetti sono indipendenti e possono essere gestiti dal sistema operativo in modo indipendente. L'elemento che viene eseguito prende il nome di *thread*, mentre l'elemento che possiede le risorse viene chiamato *processo* o *task*.

Nei sistemi operativi tradizionali (ad esempio UNIX) ogni processo è caratterizzato dal possedere un solo thread. Il termine *multithreading* è utilizzato per descrivere la situazione in cui ad un processo sono associati più thread. Tutti i thread condividono le risorse di quel processo, risiedono nello stesso spazio di indirizzamento ed hanno accesso agli stessi dati. Se, ad esempio, un thread apre un file con determinati diritti di accesso, tutti i thread dello stesso processo possono esercitare sul file gli stessi diritti di accesso.

MS-DOS è un esempio di un sistema operativo che supporta un solo processo utente ed un solo thread; UNIX è caratterizzato da più processi utenti ciascuno con un solo thread; un esempio del caso di un solo processo utente con più thread è rappresentato dal supporto a tempo di esecuzione del linguaggio Java, mentre Windows NT, Solaris e Linux rappresentano casi di sistemi operativi che supportano più processi utenti ciascuno dei quali con più thread (figura 2.13).

Ad ogni thread sono associati uno stato di esecuzione (pronto, bloccato, in esecuzione), uno spazio di memoria per le variabili locali, uno stack, un contesto rappresentato dal valore dei registri del processore utilizzati dal thread ed un descrittore. Non appartengono ad un thread risorse in quanto esso accede (come gli altri thread) a tutte le risorse di proprietà del processo che lo contiene.

Essendo ridotte le informazioni associate al thread, le operazioni di cambio di contesto, di creazione e terminazione risultano molto semplificate rispetto a quelle equivalenti dei processi.

La gestione dei thread può avvenire sia a *livello utente* che a *livello di nucleo*.

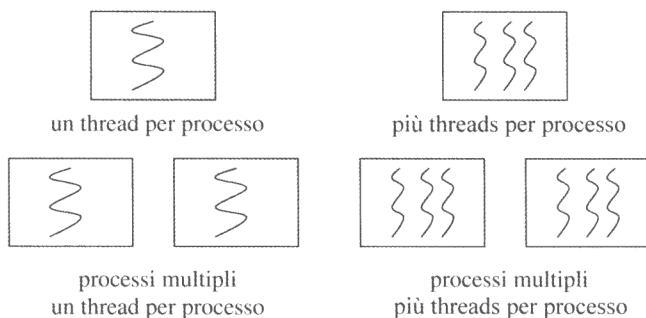


Figura 2.13 Multithreading.

Nel primo caso si fa uso di una libreria di funzioni (*thread package*) realizzata a livello utente che fornisce il supporto per la creazione e terminazione dei thread, per la loro sincronizzazione nell'accesso alle variabili locali del processo, per la scelta di quale thread mettere in esecuzione ecc.

Quello che è importante sottolineare è che tutte queste funzioni sono realizzate nello spazio utente; il sistema operativo ignora la presenza dei thread continuando a gestire solo i processi. Quando un processo è in esecuzione normalmente parte con un solo thread che ha la capacità di creare nuovi thread chiamando una apposita funzione di libreria. Si può realizzare una gerarchia di thread (padre e figlio) oppure tutti i thread sono allo stesso livello. Durante l'esecuzione un thread può passare allo stato di bloccato per effetto della chiamata ad una funzione che deve assicurare l'utilizzo esclusivo delle risorse del processo. Infine, quando un thread ha terminato il suo lavoro può richiamare una funzione di libreria che recupera la memoria ad esso associata.

Non essendo previsto l'uso di system call per la realizzazione di queste funzioni, ma semplici chiamate a procedure di libreria nello spazio utente il loro utilizzo risulta particolarmente efficiente. I thread possono, tuttavia, utilizzare le system call, ad esempio per eventuali operazioni di I/O; in questo caso interviene il sistema operativo che blocca il processo e conseguentemente l'esecuzione di tutti i thread definiti al suo interno.

Un altro caso di intervento del sistema operativo si ha con i sistemi a divisione di tempo, allo scadere del quanto di tempo assegnato al processo. Anche in questo caso il processo perde il controllo della CPU e di conseguenza tutti i suoi thread vengono sospesi.

La soluzione descritta ha il vantaggio di poter essere realizzata su sistemi operativi che non supportano direttamente i thread, come ad esempio UNIX. Inoltre è possibile adottare politiche di scheduling diverse a seconda del tipo di applicazione (a priorità, round robin) senza dover modificare lo scheduler del sistema operativo.

Non è invece possibile, con questo tipo di soluzione, sfruttare il parallelismo proprio di architetture multiprocessore. Poiché un processo è assegnato ad uno dei processori, tutti i suoi thread possono operare uno alla volta su quel solo processore.

La gestione dei thread a livello di nucleo prevede invece che il sistema operati-

vo si faccia carico di tutte le funzioni per la realizzazione dei thread: creazione, terminazione, scheduling, sincronizzazione. A ciascuna di queste funzioni corrisponde quindi una chiamata di sistema ed il nucleo deve contenere non solo i descrittori dei singoli processi, ma anche di tutti i thread. In caso di sospensione di un thread, a differenza del caso precedente, il nucleo potrà scegliere di mettere in esecuzione un thread dello stesso processo.

La soluzione, adottata in sistemi operativi come Windows NT e Linux, risulta meno efficiente della precedente a causa della gestione delle system call, ma presenta il vantaggio che in un ambiente multiprocessore thread diversi appartenenti allo stesso processo possono essere eseguiti su diverse unità di elaborazione.

In alcuni sistemi (ad esempio Solaris) viene adottata una soluzione mista con l'obiettivo di mettere assieme i vantaggi di entrambe le soluzioni.

Per un'analisi più approfondita sul tema della realizzazione dei thread si rinvia alla bibliografia.

2.11 Sommario

Obiettivo fondamentale di questo capitolo è stato quello di introdurre l'astrazione CPU virtuale, mostrando come l'uso del processore fisico possa essere assegnato alternativamente a più processi, dando loro l'illusione di possedere una propria CPU, di precisare il concetto di processo e di definirne le principali proprietà.

Il concetto di processo è fondamentale per la comprensione del funzionamento di un moderno sistema operativo il cui compito, come si vedrà nei prossimi capitoli, è quello di allocare le risorse di un sistema di elaborazione ai processi, di consentire loro la condivisione e lo scambio di informazioni, di proteggere le risorse di ciascuno di essi e permetterne la sincronizzazione.

Dopo aver evidenziato la differenza tra il concetto di processo e quello di programma, sono stati introdotti gli stati in cui un processo può trovarsi durante il suo tempo di vita e sono state illustrate le principali strutture dati necessarie per la realizzazione del concetto di processo. Si è quindi affrontato il problema delle diverse possibili forme di interazioni tra processi concorrenti, mostrando come durante la loro esecuzione i processi possono cooperare o scambiandosi informazioni o competere per le stesse risorse.

È stata poi discussa la struttura di un componente fondamentale di un sistema operativo, il nucleo o kernel, il cui compito è quello di definire l'astrazione CPU virtuale rendendo così possibile la realizzazione del concetto di processo.

Infine è stato affrontato il problema del multithreading, mostrando dapprima le motivazioni per la sua introduzione ed in seguito le modalità di realizzazione del concetto di thread.

Come si è detto, il capitolo ha nel suo complesso un carattere introduttivo. Molti dei problemi presentati troveranno un approfondimento nei capitoli successivi.

2.12 Note bibliografiche

Un'analisi approfondita del concetto di processo e delle sue proprietà si può trovare in [2], [42], [39] mentre una descrizione approfondita di come il concetto di proces-

so sia centrale per la costruzione e gestione di un sistema operativo si può trovare su alcuni testi classici di sistemi operativi come [84], [80], [97].

Gli stessi testi si possono consultare per un'analisi più approfondita degli algoritmi di scheduling, mentre per un approfondimento delle proprietà del nucleo di un sistema operativo, anche nel caso di un sistema multiprocessore, si rimanda ancora a [2].

Per la parte relativa ai thread; [59] e [55] forniscono una buona panoramica del concetto di thread e del suo utilizzo nella programmazione.

3

Sincronizzazione dei processi

Nel capitolo 2 è stato introdotto il concetto di processi concorrenti mettendo in evidenza come, in presenza di interazione, il loro comportamento risulti in genere non riproducibile e come sia pertanto necessario imporre dei vincoli, diversi a seconda del tipo di interazione, nell'esecuzione delle loro operazioni. In questo capitolo, dopo aver messo in evidenza le diverse modalità di interazione tra i processi, verranno presentati gli strumenti adottati per la loro sincronizzazione. Per semplicità si farà riferimento all'esecuzione concorrente di processi anche se, come si è visto, l'unità fondamentale di concorrenza è, in alcuni sistemi, il thread e alcuni tipi di interazione (quelli basati sull'accesso a variabili comuni) sono tipici dei thread anziché dei processi. Esempi di sincronizzazione nei thread saranno introdotti mediante l'utilizzo della libreria P-thread (vedi capitolo 7) e l'utilizzo del linguaggio Java (vedi appendice B).

3.1 Tipi di interazione tra i processi

Esistono due tipi di interazione tra i processi: *cooperazione* e *competizione*.

- a) *Cooperazione*. La cooperazione tra processi prevede tra gli stessi uno scambio di informazioni. Il caso più semplice è quello in cui l'unica informazione scambiata è costituita da un *segnaletemporeale* senza trasferimento di dati. Si pensi ad esempio ad un sistema *real-time* dedicato al controllo di un impianto industriale. In questo caso è abbastanza usuale che il sistema di elaborazione debba eseguire periodicamente alcune attività relative alla lettura di dati provenienti da appositi sensori ed all'elaborazione di tali dati per agire sull'impianto mediante opportuni attuatori. I processi che eseguono queste attività devono essere attivati da un processo gestore che ha il compito di registrare il passare del tempo e di inviare loro segnali temporali. Conclusa un'esecuzione, ogni processo deve attendere un nuovo segnale di attivazione prima di ricominciare. Esiste pertanto un vincolo di pre-

cedenza tra l'operazione con la quale viene inviato il segnale temporale da parte del processo gestore e la prima operazione di un processo ricevente. Il rispetto di questo vincolo impone una *sincronizzazione* dei due processi, nel senso che il processo che esegue una specifica attività non può iniziare la sua esecuzione prima dell'arrivo del segnale da parte del processo gestore. Nel caso generale di cooperazione, in cui sia previsto anche uno scambio di dati, l'interazione tra i due processi consiste, oltre che in una sincronizzazione tra gli stessi, anche in una *comunicazione*. Come esempio di comunicazione si può considerare il caso di un processo (*produttore*) che produce linee di stampa e le deposita in un buffer da dove sono prelevate da un altro processo (*consumatore*) che provvede alla loro stampa su dispositivo. Nell'ipotesi che il buffer possa contenere una sola linea di stampa alla volta, i vincoli imposti nell'esecuzione dei due processi prevedono che il produttore non possa inserire una nuova linea nel buffer prima che il consumatore abbia prelevato la precedente e che il consumatore non possa prelevare una linea dal buffer prima che la stessa vi sia stata inserita da parte del produttore. Anche in questo caso esiste un vincolo di precedenza tra le operazioni dei processi, che devono pertanto essere sincronizzati.

- b) *Competizione*. La competizione tra processi si ha quando questi richiedono l'uso di risorse comuni che non possono essere usate contemporaneamente. Facciamo, ad esempio, il caso che due o più processi debbano stampare dei messaggi durante la loro esecuzione. Se le velocità dei processi sono tali che le operazioni di stampa vengono eseguite concorrentemente e se il sistema di elaborazione dispone di una sola stampante, il risultato sarà evidentemente costituito da una stampa senza significato. Questo tipo di interazione non è insito nella logica dei processi (potendosi teoricamente eliminare ogni forma di competizione aumentando il numero di risorse), ma imposto da vincoli di reale disponibilità delle risorse stesse. Anche in questo caso esiste un vincolo di precedenza (sincronizzazione) tra le operazioni con le quali i processi possono accedere alla risorsa comune, ma mentre nel caso precedente il vincolo richiedeva un ordinamento tra le operazioni (prima l'invio del segnale poi l'esecuzione dell'operazione da parte del processo ricevente), in questo caso l'ordine di accesso alla risorsa è indifferente purché le operazioni siano *mutuamente esclusive nel tempo*.

La natura dei due problemi di sincronizzazione è quindi concettualmente diversa. Si parla di *sincronizzazione diretta* o *esplicita* per indicare i vincoli propri di una cooperazione e *sincronizzazione indiretta* o *implicita* per indicare i vincoli imposti dalla competizione.

Esiste un'altra forma di interazione tra due processi che va sotto il nome di *interferenza*, di cui alcuni esempi sono stati mostrati nel paragrafo 2.8, provocata da una erronea soluzione a problemi di cooperazione e competizione. Caratteristica di ogni forma di interferenza è che il manifestarsi dei propri effetti erronei dipende dai rapporti di velocità tra i processi. Con ciò si intende dire che tali effetti possono o no manifestarsi nel corso dell'esecuzione del programma a seconda delle differenti condizioni di velocità di esecuzione dei processi (*errori dipendenti dal tempo*).

Come si vedrà, obiettivo fondamentale degli strumenti di sincronizzazione è proprio quello di evitare condizioni di interferenza tra i processi.

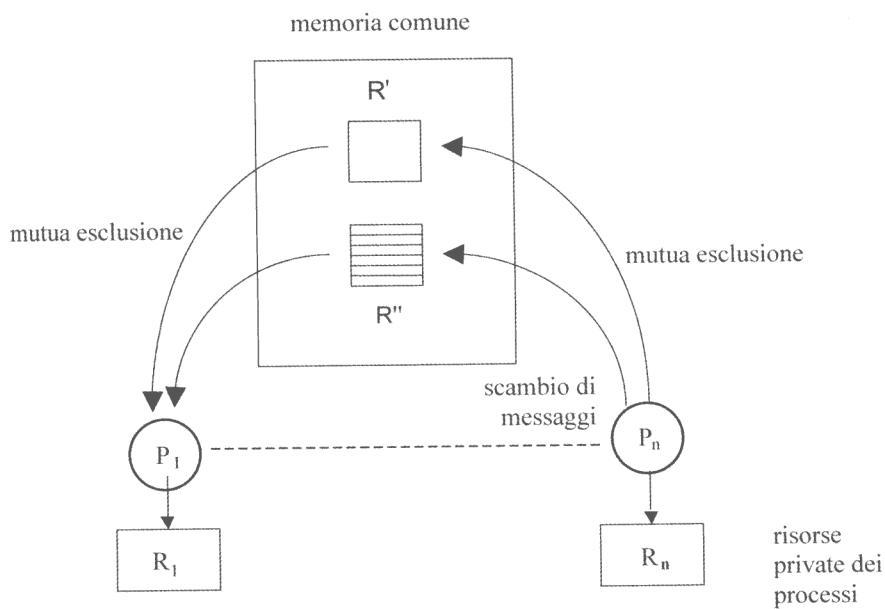


Figura 3.1 Interazioni tra processi in sistemi a memoria comune.

La soluzione ai problemi di interazione illustrati può essere ottenuta con differenti strumenti di sincronizzazione a seconda del tipo di *modello di interazione* tra processi: *modello ad ambiente globale* e *modello ad ambiente locale*.

In un modello ad ambiente globale qualunque tipo d'interazione tra i processi avviene tramite la memoria comune. Ogni applicazione viene strutturata come un insieme di processi e di risorse, intendendo con questo termine qualunque oggetto, fisico o logico (ad esempio, stampante o buffer), di cui un processo necessita per portare a termine il compito a esso affidato. Ogni risorsa viene rappresentata da una struttura dati allocata nella memoria comune; per le risorse fisiche, ad esempio dispositivi di I/O, la struttura dati è rappresentata dai descrittori dei dispositivi (vedi capitolo 5), cioè da un insieme di variabili che identificano le loro proprietà ed il loro stato. Una risorsa può essere *privata* di un processo (o *locale* al processo) quando quel processo è il solo che può operare sulla risorsa; *comune* (o *globale*) quando più processi possono operare su di essa.

In un modello ad ambiente globale entrambe le forme d'interazione avvengono tramite l'utilizzo di risorse globali; i processi competono per l'utilizzo di risorse comuni e le utilizzano per lo scambio di informazioni (figura 3.1).

Questo tipo di modello è utilizzato nel caso generale di architetture multiprocessore, caratterizzate cioè da più unità di elaborazione sulle quali operano i singoli processi, tutte collegate ad un'unica memoria principale dove risiedono gli oggetti comuni. Il caso di una sola unità di elaborazione sulla quale operano tutti i processi rientra nel caso generale.

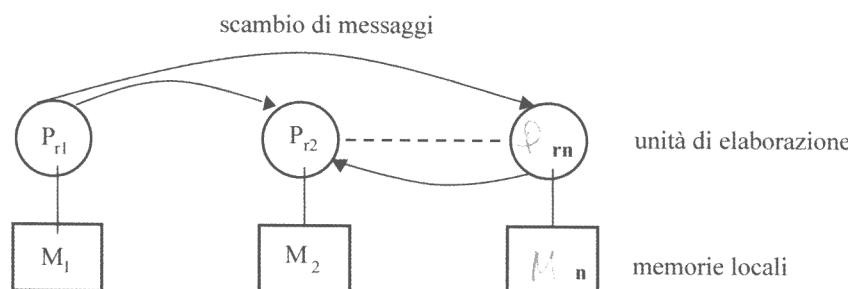


Figura 3.2 Interazioni tra processi in sistemi a memoria locale.

Un esempio di strumento di sincronizzazione particolarmente diffuso in questo modello è rappresentato dal *semaforo* e dalle *primitive di sincronizzazione* *wait* e *signal* (paragrafo 3.4).

Nel modello ad ambiente locale ogni processo opera esclusivamente su proprie variabili che non possono essere direttamente accedute da altri processi.

Non essendo presenti risorse direttamente accessibili da più processi, qualunque forma d'interazione tra processi può avvenire solo tramite scambio di messaggi (figura 3.2).

Dal punto di vista logico, esiste anche in questo modello il concetto di risorsa comune, cioè utilizzata da più processi. Tale risorsa, tuttavia, è locale ad un processo che risulta a tutti gli effetti il suo gestore. Quando un processo intende operare sulla risorsa deve comunicare al processo gestore questa esigenza inviando un opportuno messaggio; il processo gestore svolgerà l'operazione richiesta agendo sulla risorsa e comunicherà eventualmente l'esito al processo richiedente.

Questo tipo di modello, tipicamente utilizzato in reti di elaboratori senza memoria comune, può essere realizzato anche nel caso di sistemi monoprocesso e multiprocessore con memoria condivisa. Lo scambio di messaggi può avvenire infatti utilizzando sia la rete di comunicazione sia opportune aree di memoria dove i messaggi vengono depositati e prelevati.

Un esempio di strumento di sincronizzazione particolarmente diffuso in questo modello è quello basato sull'utilizzo delle primitive *send* e *receive* (paragrafo 3.5). Sia i *semafori* con le primitive *wait* e *signal* sia le primitive *send* e *receive*, rappresentano strumenti di sincronizzazione molto elementari, a livello del linguaggio assembler come potenza espressiva e messi a disposizione, in generale, dal nucleo di ogni sistema operativo. Esistono strumenti a più alto livello che facilitano la soluzione di problemi di sincronizzazione e rendono più semplice e più comprensibile verificarne la correttezza. Per il modello a memoria comune si può citare il costrutto *monitor* e per il modello ad ambiente locale il costrutto *chiamata di procedura remota* (*RPC - Remote Procedure Call*). L'analisi di questi ed altri costrutti linguistici per la sincronizzazione esula dagli obiettivi di questo testo e per essi si rimanda alla bibliografia [2], [97].

3.2 Problema della mutua esclusione

Come si è detto, la competizione tra processi su risorse comuni che per loro natura o per il modo con cui vengono impiegate possono essere utilizzate da un solo processo alla volta, impone il vincolo di *mutua esclusione* delle operazioni con cui i processi accedono alle risorse stesse.

In altre parole il vincolo di mutua esclusione richiede che le operazioni con le quali i processi accedono alle variabili comuni che rappresentano le risorse, non si sovrappongano nel tempo. Nessun vincolo è invece imposto sull'ordine con il quale le operazioni sulle variabili comuni vengono eseguite. Tali operazioni prendono il nome di *sezioni critiche*. Con tale termine si intende quindi l'insieme di operazioni con le quali ciascun processo agisce sulle variabili comuni.

Si consideri il seguente caso. Due processi P_1 e P_2 hanno accesso ad una struttura organizzata a pila per prelevare ed inserire dati. La struttura dati può essere rappresentata da un vettore `stack` i cui elementi costituiscono i singoli dati e dalla variabile `top` che indica la posizione dell'ultimo elemento contenuto nella pila. I processi utilizzano le operazioni `Inserimento` e `Prelievo` rispettivamente per depositare e prelevare i messaggi dalla pila. Tali operazioni, che accedendo a variabili comuni rappresentano delle sezioni critiche, possono essere così descritte (tralasciando il trattamento delle condizioni eccezionali di pila piena e pila vuota):

```
T stack[n];
int top = 1;

void Inserimento(T y) {
    >top++;
    >stack[top]=y;
}

T Prelievo() {
    >T temp;
    >temp=stack[top];
    >top--;
    >return temp;
}
```

Entrambe le sezioni critiche accedono e modificano le variabili comuni `stack` e `top`. L'esecuzione contemporanea di queste operazioni da parte dei processi può portare ad un uso scorretto della risorsa. Una possibile sequenza nell'esecuzione delle due sezioni critiche è infatti la seguente:

$t_0 : top++;$	(P_1)
$t_1 : temp=stack[top];$	(P_2)
$t_2 : top--;$	(P_2)
$t_3 : stack[top]=y;$	(P_1)

Come risultato viene prelevato dalla pila un valore non definito e l'ultimo valore valido contenuto nella pila viene cancellato dal nuovo valore.

Un risultato analogo si avrebbe nel caso di esecuzione contemporanea di una qualunque delle due sezioni critiche da parte dei due processi. Per una corretta soluzione del problema occorre quindi che si escludano mutuamente nel tempo sia esecuzioni diverse della stessa sezione critica sia le esecuzioni delle due sezioni critiche.

Un secondo esempio della necessità della mutua esclusione nelle operazioni tra i due processi è quello riportato nel nucleo-di-un del capitolo-2 relativo all'utilizzo della stessa variabile `contatore` da parte di due processi. Anche in quel caso per una corretta soluzione al problema è necessario che le esecuzioni della stessa sezione critica:

```
LD contatore
AD 1
STO contatore
```

con la quale entrambi i processi accedono alla variabile `contatore` si escludano mutuamente nel tempo.

3.2.1 Soluzioni al problema della mutua esclusione

Una soluzione al problema della mutua esclusione si potrebbe ottenere tempificando l'esecuzione dei singoli processi in modo tale che sezioni critiche riferite allo stesso insieme di variabili comuni vengano sempre eseguite in differenti intervalli di tempo. Questa soluzione, tuttavia, non può in generale essere accettata, in quanto presuppone da parte del programmatore una conoscenza della velocità relativa dei singoli processi.

Una seconda soluzione si può ottenere ponendo come condizione l'inibizione del sistema di interruzione dell'elaboratore sul quale sono eseguite le sezioni critiche, durante l'esecuzione di ciascuna di esse. La soluzione è parziale, perché limitata ovviamente al caso in cui le sezioni critiche siano eseguite sullo stesso elaboratore; anche in questo caso, tuttavia, la soluzione presenta l'inconveniente di inibire, durante l'esecuzione di una sezione critica, l'esecuzione di eventuali operazioni prioritarie e indipendenti dalle sezioni critiche.

La ricerca di una soluzione generale al problema richiede la definizione di un opportuno protocollo che i processi devono adottare per interagire correttamente sulla risorsa comune. In altre parole, ogni processo prima di entrare in una sezione critica dovrà chiedere l'autorizzazione eseguendo una serie di operazioni che gli garantiscono l'accesso esclusivo alla risorsa, se questa è libera, oppure ne impediscono l'accesso, se questa è già occupata. Questo insieme di istruzioni prende il nome di *prologo*, mentre con il nome di *epilogo* verrà indicato l'insieme di istruzioni eseguite dal processo per dichiarare libera la sezione critica al completamento della sua azione.

Il modo più semplice per definire il prologo e l'epilogo è mediante l'utilizzo di una variabile condivisa `occupato` che può assumere il valore 1 (risorsa occupata) o il valore 0 (risorsa libera). Sia 0 il valore iniziale di `occupato`:

```
P1
prologo : while(occupato == 1);
           occupato =1;
           <sezione critica A>;
epilogo : occupato = 0;
```

```
P2
prologo: while(occupato == 1);
          occupato = 1;
          <sezione critica B>;
epilogo:  occupato = 0;
```

Si può facilmente notare che la soluzione non soddisfa la proprietà di mutua esclusione nell'esecuzione delle sezioni critiche. A seconda dei rapporti di velocità dei processi, essi possono essere abilitati ad eseguire contemporaneamente le rispettive sezioni critiche.

Si consideri, ad esempio, la sequenza di esecuzione delle operazioni del prologo da parte dei due processi:

$t_0 : P_1$ esegue l'istruzione `while` e trova `occupato=0`
 $t_1 : P_2$ esegue l'istruzione `while` e trova `occupato=0`
 $t_2 : P_1$ pone `occupato=1` ed entra nella sezione critica
 $t_3 : P_2$ pone `occupato=1` ed entra nella sezione critica

Tale sequenza ha come risultato che entrambi i processi sono contemporaneamente nella loro sezione critica.

È importante notare che nella soluzione precedente si è supposto che l'hardware garantisca la mutua esclusione solo a livello di lettura e scrittura di una singola parola in memoria (nel senso che l'indivisibilità è assicurata solo riguardo all'ispezione o assegnamento di un valore ad una singola variabile comune).

Molte macchine posseggono particolari istruzioni che consentono di esaminare e modificare il contenuto di una parola o di scambiare il contenuto di due parole in un ciclo di memoria. Ciò consente, nel caso dell'esempio precedente, al processo P_1 di leggere e modificare la variabile `occupato` senza possibilità di interferenza da parte di P_2 . Un esempio tipico di queste istruzioni è rappresentato dall'istruzione TSL (*Test and Set Lock*).

L'esecuzione di `TSL R, x` funziona nel seguente modo: il valore contenuto nella locazione `x` viene copiato nel registro `R` del processore e viene scritto in `x` un valore diverso da zero. Le operazioni di lettura e scrittura sono eseguite in modo indivisibile: nessun altro processore può accedere a `x` finché l'istruzione non è finita. La CPU che esegue l'istruzione TSL blocca il bus di memoria per impedire ad altre CPU di accedere alla memoria finché non ha completato l'esecuzione di TSL.

La mutua esclusione nella esecuzione delle due sezioni critiche A e B dell'esempio precedente si ottiene introducendo due funzioni `lock(x)` e `unlock(x)` così definite:

```
lock(x):
: TSL registro,x      (copia x nel registro e pone x=1)
CMP registro,0        (il valore di x era 0 ?)
JNE lock              (se no, ricomincia il ciclo)
RET                  (ritorna al chiamante;)
                     (accesso alla sezione critica)
```

```
unlock(x) :
    MOVE x, 0          (inserisce 0 in x)
    RET                (ritorna al chiamante)
```

dove il valore 0 per la variabile x indica che nessuna delle due sezioni critiche è in esecuzione, mentre il valore 1 indica che una tra A e B è in esecuzione. Lo schema seguito dai due processi P_1 e P_2 è ora il seguente:

```
P1
prologo:      lock(x);
               <sezione critica A>;
epilogo:       unlock(x);

P2
prologo:      lock(x);
               <sezione critica B>;
epilogo:       unlock(x);
```

La soluzione precedente è caratterizzata da condizioni di *attesa attiva* dei processi che non possono entrare nella sezione critica richiesta (*busy form of waiting*). Tali processi ripetono infatti la sequenza di istruzioni con la quale richiedono l'accesso alla sezione critica tenendo occupato il processore sul quale sono in esecuzione ed impedendo quindi che esso venga assegnato ad altri processi. La soluzione è quindi applicabile solamente in sistemi multiprocessore, con processi cioè operanti su processori diversi e si presta al caso di sezioni critiche "molto brevi".

Sono stati realizzati algoritmi che assicurano una corretta soluzione al problema della mutua esclusione tra n processi senza utilizzare l'intervento dell'hardware come nel caso visto precedentemente. Essi risultano tuttavia complessi, sono caratterizzati da forme di attesa attiva e possono, in alcuni casi, non garantire l'assenza di condizioni in cui processi pronti per entrare nella loro sezione critica vengano indefinitely ritardati a causa dell'esecuzione della stessa sezione da parte di altri processi (*starvation*). Per un loro esame si rimanda alla bibliografia [74].

3.3 Problema della comunicazione

Un ben noto problema nel campo della comunicazione tra processi è quello che va sotto il nome di produttore-consamatore.

Un processo (*produttore*) genera ciclicamente un messaggio e lo deposita in un'area di memoria, *buffer*, capace di contenere un solo messaggio alla volta; un secondo processo (*consumatore*) preleva dall'area di memoria il messaggio e lo utilizza (figura 3.3).

I vincoli imposti nell'esecuzione delle operazioni dei due processi sono i seguenti:

1. il produttore non può inserire nel buffer un nuovo messaggio prima che il consumatore abbia prelevato il precedente;
2. il consumatore non può prelevare dal buffer un nuovo messaggio prima che il produttore l'abbia depositato.

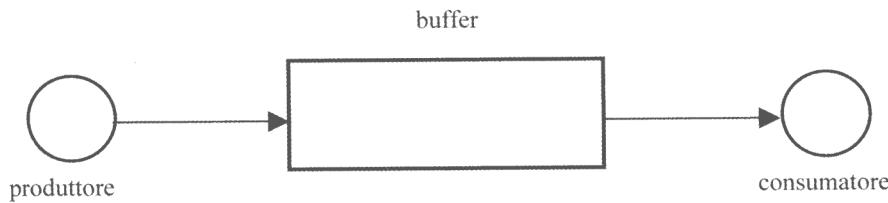


Figura 3.3 Produttore-consumatore.

Si noti che, pur esistendo un problema di mutua esclusione nell'utilizzo del buffer comune da parte dei processi, la soluzione a questo problema impone, a differenza dei problemi esaminati nel paragrafo precedente, un ordinamento nelle operazioni dei due processi. Per ottenere questo ordinamento è necessario che produttori e consumatori si scambino segnali per indicare rispettivamente l'avvenuto deposito e prelievo del messaggio dal buffer. Ciascuno dei due processi deve attendere, per compiere la sua azione, l'arrivo del segnale dell'altro processo.

Nel caso più generale, produttori e consumatori possono utilizzare un buffer contenente più messaggi (figura 3.4). Le condizioni 1) e 2) diventano in questo caso:

1. il produttore non può inserire un messaggio nel buffer se questo è pieno;
2. il consumatore non può prelevare un messaggio dal buffer se questo è vuoto.

Anche in questo caso, come nel paragrafo precedente, sarebbe possibile costruire soluzioni per il problema che presenterebbero tuttavia i problemi già discussi nel caso della mutua esclusione. Si preferisce quindi, per semplicità, riportare direttamente la soluzione basata sui semafori.

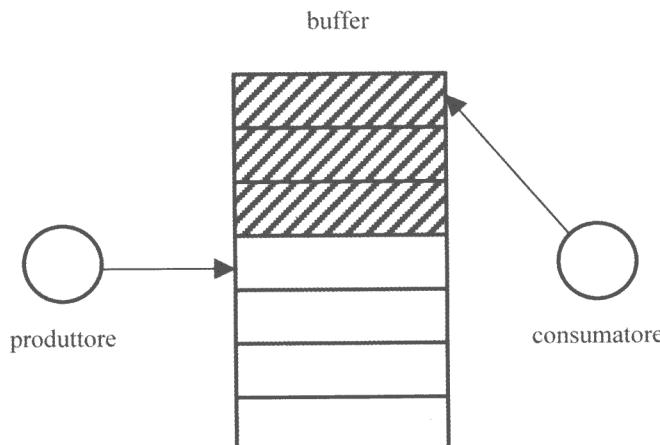


Figura 3.4 Produttore-consumatore con buffer di dimensione N .

3.4 Semafori

Nel seguito verranno presentati il semaforo e le primitive `wait` e `signal` e verrà mostrato come con tale strumento sia possibile, in ambienti a memoria comune, risolvere sia il problema della mutua esclusione che quello della comunicazione.

Un semaforo `s` è una struttura dati alla quale sono applicabili solo due operazioni primitive (indivisibili) `wait(s)` e `signal(s)`. La struttura dati è costituita da una variabile intera non negativa, che indicheremo come `s.value`, con valore iniziale $s_0 \geq 0$ e da una coda di processi sospesi, che indicheremo con `s.queue`.

L'operazione `wait` viene utilizzata da un processo per verificare lo stato di un semaforo secondo lo schema di seguito riportato:

```
void wait(s)
{
    if (s.value == 0)
        <il processo viene sospeso>
        <il suo descrittore inserito in s.queue>;
    else s.value = s.value - 1;
}
```

Se il valore del semaforo è positivo, questo viene decrementato di una unità ed il processo prosegue l'esecuzione; se il valore è nullo, l'esecuzione della `wait` provvede a modificare lo stato del processo che da attivo diventa bloccato e ad inserire il suo descrittore nella coda associata al semaforo. L'esecuzione del processo è sospesa e l'unità di elaborazione così liberata viene assegnata ad un altro processo.

L'operazione `signal` viene utilizzata per risvegliare eventuali processi sospesi sul semaforo secondo lo schema di seguito riportato:

```
void signal(s)
{
    if (<esiste almeno un processo nella coda s.queue>)
        <il descrittore del primo di questi viene estratto
         da s.queue ed il suo stato modificato in pronto>;
    else s.value = s.value + 1;
}
```

Se non esistono processi in coda, il valore del semaforo viene incrementato di una unità; diversamente viene riattivato il primo processo in coda.

A differenza della `wait`, la primitiva `signal` non prevede la sospensione del processo che la esegue, nel senso che il processo può logicamente proseguire la sua esecuzione. Tuttavia, nel caso in cui lo scheduling sia organizzato per livelli di priorità, il processo risvegliato, P_k , potrebbe essere prioritario rispetto al processo in esecuzione, P_i . In questo caso lo stato di P_i viene salvato nel suo descrittore che viene inserito nella coda dei processi pronti e P_k viene posto in esecuzione.

3.4.1 Soluzione al problema della mutua esclusione

La soluzione al problema della mutua esclusione si ottiene associando alla risorsa condivisa, il cui utilizzo deve essere mutuamente esclusivo da parte di un insieme di

processi, un semaforo mutex inizializzato al valore 1 e adottando per ogni processo dell'insieme il seguente schema:

```
P
prologo    wait (mutex);
            <sezione critica>;
epilogo:   signal (mutex);
```

Per illustrare la soluzione al problema della mutua esclusione facendo uso dei semafori e delle `wait` e `signal` si consideri il caso di tre processi P_1, P_2, P_3 che accedono alla stessa risorsa utilizzando rispettivamente le sezioni critiche A, B, C.

```
P1
prologo:   wait(mutex);
            <sezione critica A>;
epilogo:   signal(mutex);

P2
prologo:   wait(mutex);
            <sezione critica B>;
epilogo:   signal(mutex);

P3
prologo:   wait(mutex);
            <sezione critica C>;
epilogo:   signal(mutex);
```

Si supponga, ad esempio, che il primo processo ad accedere alla sua sezione critica sia P_2 . Essendo il valore di `mutex` uguale a 1, l'esecuzione della `wait(mutex)` decremente il valore a 0 ed il processo P_2 può entrare in B. Si supponga che P_2 passi nello stato di pronto (ad esempio, revoca della CPU) e che il processo P_1 entri in esecuzione. Se P_1 tenta di entrare nella sua sezione critica A, l'esecuzione di `wait(mutex)` lo sospende (essendo `mutex = 0`) ed il suo descrittore viene inserito nella coda del semaforo `mutex`. Se è P_3 il nuovo processo in esecuzione, un suo tentativo di entrare nella sezione critica C provoca ugualmente la sua sospensione e l'inserimento del suo descrittore nella coda del semaforo `mutex`.

Dei tre processi uno, P_2 , si trova nello stato di pronto, gli altri due, P_1 e P_3 nello stato di bloccato (entrambi nella coda associata a `mutex`). P_2 può riprendere l'esecuzione e completare la sua sezione critica. L'esecuzione di `signal(mutex)` provoca il risveglio di uno dei processi in coda a `mutex`, ad esempio P_1 , che transita nello stato di pronto. La scelta di quale tra i due processi P_2 e P_1 mettere in esecuzione dipende dalla priorità relativa dei due processi. Nell'ipotesi che tale processo sia P_2 , esso potrà proseguire e terminare la sua esecuzione (o essere nuovamente sospeso). Verrà quindi messo in esecuzione P_1 , che al completamento della sua sezione critica eseguirà `signal(mutex)` risvegliando il processo P_3 . Nell'ipotesi che P_1 rimanga attivo, al suo completamento (o sospensione) verrà messo in esecuzione P_3 che, completata la sua sezione critica, eseguirà la `signal(mutex)`. Non essendo processi sospesi in coda a `mutex`, il suo valore verrà riportato a 1.

Le sezioni critiche A, B, C sono state eseguite dai processi in modo mutuamente esclusivo. Si può facilmente verificare che lo stesso risultato si sarebbe potuto ottenere con qualunque sequenza di esecuzione dei processi.

La soluzione presentata evita condizioni di *attesa attiva* in quanto l'impossibilità di accedere alla propria sezione critica comporta la sospensione del processo. Inoltre, se si adotta una politica di risveglio dei processi sospesi sul semaforo del tipo FCFS si evitano anche situazioni di *attesa indefinita (starvation)* dei processi sospesi.

Il semaforo `mutex` prende il nome di *semaforo binario* in quanto può assumere solo i valori 1 e 0. Si noti che la correttezza della soluzione è legata all'inizializzazione del semaforo `mutex` al valore 1 e al corretto posizionamento delle primitive `wait` e `signal` (subito prima e subito dopo la sezione critica). Per questi motivi, come si è accennato precedentemente, il semaforo e le primitive `wait` e `signal` rappresentano uno strumento molto elementare per la sincronizzazione dei processi, soprattutto per problemi più complessi di quello della mutua esclusione.

Rimane, per completare, il problema fondamentale di come ottenere la *indivisibilità* delle operazioni `wait` e `signal`. Nell'ipotesi che tutti i processi coinvolti nel problema di mutua esclusione operino sullo stesso processore l'indivisibilità della `wait` e `signal` è garantita dalla disabilitazione delle interruzioni del processore durante la loro esecuzione. D'altra parte `wait` e `signal` sono operazioni fornite dal nucleo del sistema operativo e chiamabili dai processi tramite chiamate di sistema.

Qualora i processi siano eseguiti su processori diversi, per garantire che `wait` e `signal` operino in modo mutuamente esclusivo sul semaforo `mutex` bisogna ricorrere alle funzioni `lock(x)` e `unlock(x)`, descritte precedentemente, secondo il seguente schema:

```
lock(x);
  wait(mutex);
unlock(x);
  <sezione critica>;
lock(x);
  signal(mutex);
unlock(x);
```

L'esecuzione della `lock(x)` garantisce che la `wait(mutex)` possa essere eseguita da un solo processo alla volta. Durante l'esecuzione della `wait(mutex)` da parte di un processo, gli altri processi che chiamano la `wait(mutex)` si chiudono in un ciclo fino all'esecuzione della `unlock`. Lo stesso ragionamento vale per la `signal(mutex)`.

In conclusione il semaforo `mutex` (con le primitive `wait` e `signal`) assicura la mutua esclusione delle sezioni critiche su una risorsa R, mentre la variabile `x` (con le primitive `lock` e `unlock`) assicura la mutua esclusione delle primitive `wait` e `signal` sul semaforo `mutex`.

3.4.2 Soluzione al problema della comunicazione

La soluzione al problema del produttore-consomatore con buffer di capacità unitaria (riportato al sinc-problema-comunicazione) utilizzando i semafori e le primitive `wait` e `signal` è la seguente:

```
Processo produttore
{
  do {
```

```

<produzione del nuovo messaggio>;
wait (spazio_disponibile);
<deposito del messaggio nel buffer>;
signal (messaggio_disponibile);
} while (!fine);
}

Processo consumatore
{
    do {
        wait (messaggio_disponibile);
        <prelievo del messaggio dal buffer>;
        signal (spazio_disponibile);
        <consumo del messaggio>
    }
    while (!fine);
}

```

Si fa l'ipotesi che il buffer sia inizialmente vuoto. I due semafori sono indicati con `messaggio_disponibile` e `spazio_disponibile`, con valore iniziale rispettivamente 0 e 1. Si può verificare facilmente che la soluzione è corretta qualunque sia la velocità relativa dei processi.

La soluzione al problema con buffer contenente più messaggi è analoga alla precedente, con la differenza (sempre nell'ipotesi che il buffer sia inizialmente vuoto) che il valore iniziale del semaforo `spazio_disponibile` vale N, se N è la dimensione del buffer, mentre il valore iniziale di `messaggio_disponibile` rimane 0.

Il produttore verifica la disponibilità di spazio nel buffer tramite la primitiva `wait(spazio_disponibile)`. Se il buffer è pieno il valore del semaforo è 0 e il produttore viene bloccato. Ogni qualvolta, tramite l'esecuzione della primitiva `signal(spazio_disponibile)`, il consumatore rende disponibile una porzione di buffer, risveglia il processo produttore, se sospeso, o incrementa il valore del contatore `spazio_disponibile`.

Analogamente, il consumatore verifica la disponibilità di un messaggio tramite la primitiva `wait(messaggio_disponibile)`. Se il buffer è vuoto il consumatore viene bloccato. Ognqualvolta il produttore rende disponibile un messaggio, tramite l'esecuzione della primitiva `signal(messaggio_disponibile)` risveglia il processo consumatore, se sospeso, oppure incrementa il valore di `messaggio_disponibile`.

Affinché la soluzione sia corretta è anche necessario che produttore e consumatore non accedano mai contemporaneamente alla stessa posizione del buffer. Per dimostrare che tale condizione è verificata, supponiamo che il buffer sia organizzato come un vettore circolare e gestito tramite i due puntatori `coda` e `testa` che individuano rispettivamente la prima porzione vuota e piena del buffer. Inizialmente sia `coda=testa`. Il deposito di un messaggio nel buffer comporta le seguenti operazioni:

```
vettore [coda]=<messaggio prodotto>;
coda=(coda+1)%N;
```

Il prelievo di un messaggio da parte del consumatore avviene nel modo seguente:

```
<messaggio prelevato>=vettore[testa];
testa=(testa+1)%N;
```

Il programma per il processo produttore e il processo consumatore diventa quindi:

```
Processo produttore
{
    do {
        <produzione del messaggio x>;
        wait(spazio_disponibile);
        vettore[coda]=x;
        coda=(coda+1)% N;
        signal(messaggio_disponibile);
    }
    while(!fine);
}

Processo consumatore
{
    >do {
        wait (messaggio disponibile);
        x=vettore[testa];
        testa=(testa+1) % N;
        signal (spazio disponibile);
        <consumo del messaggio x>;
    }
    while (!fine);
}
```

Indicando con p_1 e p_2 rispettivamente il numero di volte in cui `coda` e `testa` sono stati incrementati e tenendo conto che i loro valori iniziali coincidono, le operazioni di deposito e prelievo agiscono sulla stessa porzione di buffer se:

$$p_1 = p_2 \% N$$

Si può facilmente dimostrare che tale condizione, che corrisponde a situazioni di buffer pieno o buffer vuoto, non può mai verificarsi. Infatti, se il buffer è pieno il processo produttore viene bloccato sul semaforo `spazio_disponibile`, mentre se il buffer è vuoto il processo consumatore viene bloccato sul semaforo `messaggio_disponibile`.

3.5 Primitive send e receive

Nel paragrafo precedente abbiamo visto come attraverso l'uso del semaforo e delle primitive `wait` e `signal` sia possibile risolvere problemi di competizione e cooperazione tra processi operanti in un ambiente a memoria comune. In un ambiente a memoria locale qualunque forma d'interazione tra processi avviene attraverso la *comunicazione*, cioè lo scambio di informazioni tra i processi sotto forma di messaggi.

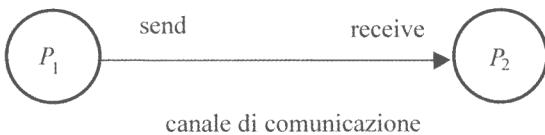


Figura 3.5 Canale di comunicazione.

Con il termine IPC (*Inter-Process-Communication*) si intende un meccanismo offerto dal nucleo del sistema operativo mediante il quale avviene la comunicazione tra i processi.

Lo strumento IPC più elementare è quello basato sull'utilizzo delle due primitive `send` e `receive`, il cui schema generale è il seguente:

```
send(destinazione, messaggio);
receive(origine, messaggio);
```

La primitiva `send` spedisce un messaggio ad una determinata destinazione e la `receive` riceve un messaggio da una determinata origine (o da qualunque origine, se al ricevente non interessa una particolare origine).

I sistemi di comunicazione basati sulla `send` e `receive` vengono in generale classificati a seconda delle seguenti caratteristiche:

- designazione di `origine` e `destinazione` di ogni comunicazione;
- tipo di sincronizzazione tra i processi comunicanti.

Prima di descrivere le proprietà fondamentali di queste primitive va ricordato che, seppure il loro uso risulti particolarmente utile in ambienti distribuiti dove i processi comunicanti possono risiedere in macchine diverse collegate tra loro in rete, esse possono essere impiegate anche in sistemi multiprocessori a memoria condivisa o in sistemi a singolo processore.

Il concetto fondamentale che sta alla base del loro utilizzo è infatti la separazione degli spazi di indirizzi dei singoli processi, tipica di un modello a memoria locale, che è possibile ottenere anche in presenza di memoria comune.

Per comunicare, i processi necessitano di un *canale di comunicazione* (figura 3.5).

Nel caso di sistemi privi di memoria comune il canale è realizzato tramite una qualche forma di collegamento fisico tra i processori sui quali operano i processi; nel caso di sistemi con memoria comune il canale è realizzato da una zona di memoria, gestita dal sistema operativo, nella quale vengono inseriti e prelevati i messaggi.

La figura 3.6 riporta il formato tipico di un messaggio il cui contenuto è rappresentato da due parti, l'*intestazione* ed il *corpo*. Fanno parte dell'intestazione l'identificazione del mittente e del destinatario ed eventuali informazioni relative al tipo di messaggio, alla sua lunghezza, ecc. Il corpo contiene il messaggio vero e proprio.



Figura 3.6 Formato di un messaggio.

3.6 Soluzione al problema di comunicazione tra processi

In un sistema a scambio di messaggi i processi possono comunicare tra loro *direttamente* o *indirettamente*. Nel caso di comunicazione diretta è necessario indicare esplicitamente nelle primitive `send` e `receive` i nomi dei processi cui si vuole inviare un messaggio o da cui si vuole ricevere un messaggio. La situazione descritta dalle due primitive seguenti, eseguite rispettivamente dai due processi P_1 e P_2 ,

```
send (P2,messaggio);
receive (P1,messaggio);
```

sta ad indicare che il processo P_1 invia `messaggio` al processo P_2 e che il processo P_2 riceve, in `messaggio`, le informazioni inviate da P_1 . La comunicazione, di *tipo simmetrico* avviene attraverso un canale che è associato in modo univoco ai due processi.

Una variante al caso mostrato si ha quando il processo mittente nomina esplicitamente il destinatario, mentre non è possibile per il processo ricevente definire un nome di mittente in quanto più processi possono inviargli messaggi. È questo il caso di un processo adibito, ad esempio, alla gestione di un disco che riceve richieste da più processi. La comunicazione (*asimmetrica*) è descritta dalle due primitive seguenti,

```
send (P2,messaggio);
receive (id,messaggio);
```

la prima delle quali eseguita da un generico processo P_i che invia un messaggio a P_2 (ad esempio, per richiedergli l'esecuzione di un'operazione) e la seconda eseguita da P_2 con `id` che, di volta in volta, assume il nome del processo con cui è avve-

Processo produttore P	Processo consumatore C
<pre>pid C=.....; main () { msg M; do { produci(&M); ... send (C,M); } while (!fine); }</pre>	<pre>pid P=.....; main () { msg M; do { receive (P,&M); ... consuma (M); } while(!fine); }</pre>

Figura 3.7 Comunicazione diretta simmetrica.

nuta la comunicazione; *id* può essere utilizzato per rispondere al mittente del messaggio quando l'operazione è compiuta.

Entrambi i casi di comunicazione diretta presentano lo svantaggio che una modifica del nome di un processo implica la revisione di tutte le operazioni di comunicazione che vedono coinvolto il processo.

In figura 3.7 e in figura 3.8 sono riportati gli schemi di due processi produttore e consumatore nel caso di comunicazione diretta simmetrica e asimmetrica. L'identificatore *pid* (*process identifier*) rappresenta un tipo di variabili utilizzate per identificare processi.

La *comunicazione indiretta* prevede che i messaggi non siano inviati direttamente ai processi, ma depositati in, e prelevati da, una struttura dati detta *porta*. Nell'ipotesi che il modello a memoria locale sia realizzato su sistemi mono o multiprocessore, la porta, che prende il nome di *mailbox*, è creata e gestita dal sistema operativo e resa accessibile ai processi attraverso apposite chiamate di sistema.

La comunicazione tra due processi P_1 e P_2 avviene ora nel seguente modo:

```
send(mailbox, messaggio);
receive(mailbox, messaggio);
```

Processo produttore P	Processo consumatore C
<pre>pid C=... main () { msg M; do { produci(&M); ... send (C,M); } while (!fine); }</pre>	<pre>... main () { msg M; pid id; do { receive(&id,&M); ... consuma(M); } while(!fine); }</pre>

Figura 3.8 Comunicazione diretta asimmetrica.

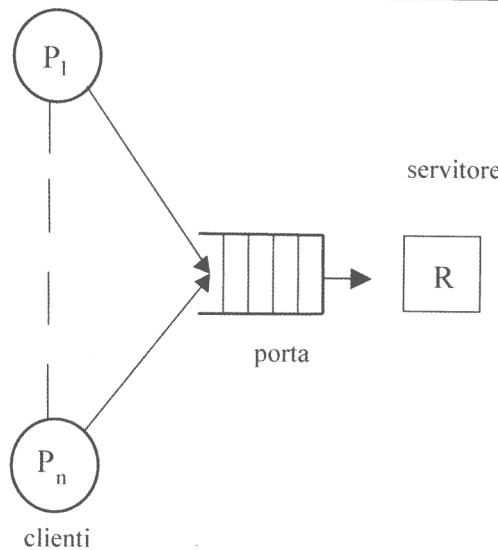


Figura 3.9 Modello cliente-servitore.

La primitiva `send` viene utilizzata da P_1 per inviare un messaggio alla `mailbox` e la primitiva `receive` è utilizzata da P_2 per prelevare un messaggio dalla `mailbox`.

Nell'ipotesi che il modello a memoria locale sia realizzato su sistemi distribuiti, le porte sono create e gestite dai sistemi operativi sui processori su cui sono attivi i processi.

Per un'analisi più approfondita del problema della comunicazione tra processi tramite `mailbox` si rinvia al capitolo relativo ai sistemi operativi Unix e Linux. Per la comunicazione in ambito di reti di calcolatori e sistemi distribuiti si rinvia alla bibliografia.

Si vuole qui ricordare che tra tutte le relazioni tra mittenti e riceventi (*uno-a-uno*, *uno-a-molti*, *molti-a-uno* e *molti-a-molti*), di particolare importanza è quello *molti-a-uno* proprio di collegamenti tipo *cliente-servitore* (*client-server*). Lo schema cliente-servitore corrisponde all'uso di un processo come gestore di una risorsa R (processo *servitore*). Ogni processo cliente che necessiti di operare sulla risorsa invia un messaggio alla porta cui è associato il processo servitore, specificando una richiesta di servizio (cioè un tipo di operazione sulla risorsa R); il processo servitore analizza le varie richieste di servizio contenute nella porta, ne sceglie una in base a determinati criteri ed opera sulla risorsa R rispondendo, se previsto, al corrispondente processo cliente (figura 3.9).

3.6.1 Sincronizzazione tra processi comunicanti

Per quanto riguarda la sincronizzazione tra processi indotta dall'uso delle primitive `send` e `receive`, esistono varie possibilità legate alle proprietà delle primitive stesse.

Per la `send`, possiamo distinguere tre casi:

- *send asincrona*
- *send sincrona*
- *send tipo chiamata di procedura remota*

Per la `receive` si hanno due alternative:

- *receive bloccante*
- *receive non bloccante*

La `send` di tipo asincrono consente al processo di proseguire la sua esecuzione immediatamente dopo l'esecuzione della `send` stessa e quindi l'invio del messaggio.

La `send` di tipo sincrono blocca il processo mittente fino al ricevimento del messaggio da parte del processo destinatario.

La `send` tipo chiamata di procedura remota viene utilizzata quando il processo mittente chiede l'esecuzione di un servizio (procedura remota) al processo destinatario e i due processi operano su due processori collegati da una rete di comunicazione. Il processo mittente rimane in attesa fino a quando il servizio richiesto non è terminato e il risultato ricevuto.

La `receive` bloccante provoca la sospensione del processo che la esegue nell'ipotesi che non ci siano messaggi in attesa di essere serviti; all'arrivo del primo messaggio il processo viene risvegliato.

La `receive` non bloccante consente la prosecuzione dell'esecuzione del processo anche in assenza di messaggi.

Sono possibili diverse combinazioni di `send` e `receive` tra quelle illustrate. Ad esempio, per quanto riguarda il modello client-server, oltre alla `send` anche la `receive` deve risultare bloccante. Il processo server, infatti, non può eseguire la sua azione in assenza di una specifica richiesta.

La primitiva `send` non bloccante risulta, in generale, molto diffusa proprio per la proprietà di consentire l'immediata prosecuzione del processo che l'ha eseguita. In generale, tuttavia, lascia al programmatore il compito di controllare che il messaggio sia stato ricevuto senza errori: i processi riceventi devono infatti farsi carico di inviare un messaggio che segnali al mittente l'avvenuta ricezione del messaggio. Per la `receive` può risultare più conveniente una forma non bloccante che consenta di analizzare, secondo un ordine stabilito, lo stato di più canali di ingresso, anche se, in assenza di messaggi su tutti i canali, questo può portare a forme di attesa attiva.

Come si è già detto a proposito delle forme di comunicazione dei processi, anche per un'analisi approfondita delle proprietà delle primitive `send` e `receive` si rinvia allo studio delle reti di calcolatori e dei sistemi distribuiti.

3.7 Blocco critico

Un sistema multiprogrammato è caratterizzato dal fatto che più processi possono competere per ottenere l'uso di un numero definito di risorse. La situazione di *blocco critico* o di *stallo (deadlock)* si può verificare tra un gruppo di due o più processi quando ciascun processo possiede almeno una risorsa e fa richiesta per un'altra. La

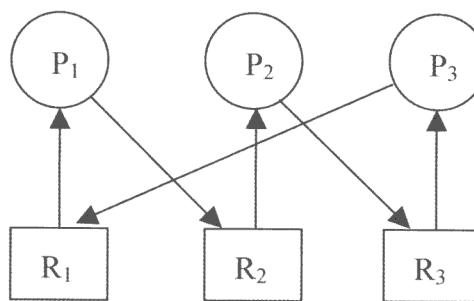


Figura 3.10 Blocco critico.

richiesta non può essere soddisfatta perché la risorsa richiesta è trattenuta da un altro processo che è a sua volta bloccato in attesa della risorsa posseduta dal primo.

Si consideri il seguente caso di blocco critico tra tre processi P_1, P_2, P_3 che competono per l'utilizzo di tre risorse R_1, R_2, R_3 . Adottando una rappresentazione grafica nella quale i processi sono rappresentati con dei cerchi, le risorse con dei rettangoli, e dove un arco diretto da una risorsa ad un processo significa che la risorsa è stata assegnata al processo ed un arco diretto dal processo alla risorsa indica che il processo richiede la risorsa, la situazione precedente può essere descritta dalla figura 3.10.

Il processo P_1 trattiene la risorsa R_1 e richiede la risorsa R_2 ; il processo P_2 trattiene la risorsa R_2 e richiede la risorsa R_3 ; il processo P_3 trattiene la risorsa R_3 e richiede la risorsa R_1 . Nessuno dei processi può proseguire l'esecuzione poiché tutti sono in attesa di una risorsa che è trattenuta da un altro processo bloccato. Nell'ipotesi che le risorse possedute dai processi possano essere rilasciate da questi solo al termine del loro uso, il risultato è che P_1, P_2, P_3 non sono in grado di proseguire nella loro esecuzione e rimarranno *bloccati indefinitamente (blocco critico)*.

Le risorse cui si è fatto riferimento nell'esempio prendono il nome di *risorse riusabili*. Una risorsa riusabile può essere utilizzata da un processo alla volta e non viene distrutta dopo l'uso. I processi acquisiscono in modo esclusivo queste risorse, le utilizzano e poi le rilasciano per permettere ad altri processi di utilizzarle. Esempi di risorse riusabili sono risorse hardware come stampanti, scanner, CD, registratori di nastro e risorse software come file, record, tabelle ecc.

Nel caso considerato, ad esempio, il processo P_1 dopo aver acquisito un file (R_1) per modificarne il contenuto richiede l'utilizzo di una stampante (R_2), che è attualmente impegnata dal processo P_2 , senza rilasciare il file. Il processo P_2 richiede a sua volta l'utilizzo di una unità a nastri, che è attualmente impegnata dal processo P_3 , senza rilasciare la stampante. Il processo P_3 chiede di poter utilizzare lo stesso file posseduto da P_1 senza rilasciare l'unità a nastri.

E opportuno mettere subito in evidenza come la situazione di blocco critico sia dipendente dalla *velocità relativa* di esecuzione dei processi. Per illustrare quanto detto semplifichiamo l'esempio precedente considerando il caso di due processi P_1 e P_2 e di due risorse riusabili R_1 e R_2 e utilizziamo le primitive di sincronizzazione *wait* e *signal*.

Siano mutex_1 e mutex_2 i due semafori di mutua esclusione, inizializzati ad 1, associati rispettivamente alle risorse R_1 e R_2 ed utilizzati per assicurare che R_1 e R_2 vengano usate dai processi P_1 e P_2 in modo mutuamente esclusivo. La struttura dei due processi sia la seguente:

```

P1
...
wait(mutex1);
<inizio utilizzazione di R1>;
...
wait (mutex2);
<inizio utilizzazione di R2>;
...
signal(mutex2);
<rilascio di R2>;
...
signal (mutex1);
<rilascio di R1>;
...

P2
...
wait(mutex2);
<inizio utilizzazione di R2>;
...
wait(mutex1);
<inizio utilizzazione di R1>;
...
signal (mutex1);
<rilascio di R1>;
...
signal (mutex2);
<rilascio di R2>;
...

```

Se si verifica la seguente sequenza temporale di azioni da parte dei due processi:

P_1 esegue `wait(mutex1)` ed acquisisce la risorsa R_1

P_2 esegue `wait(mutex2)` ed acquisisce la risorsa R_2

P_1 esegue `wait(mutex2)` e si blocca

P_2 esegue `wait(mutex1)` e si blocca

i due processi rimangono bloccati rispettivamente sui semafori mutex_2 e mutex_1 e non esiste alcuna possibilità da parte dei processi di uscire dalla situazione di blocco.

Come si vede dalla figura 3.11, esiste una condizione di *catena circolare* tra processi e risorse.

Si può facilmente verificare che altre condizioni di velocità relativa nell'esecuzione dei processi non avrebbero portato a situazioni di blocco critico. Ad esempio, se il processo P_1 avesse acquisito anche la risorsa R_2 prima del processo P_2 , avrebbe potuto utilizzare entrambe le risorse, mentre il processo P_2 si sarebbe bloccato sul semaforo mutex_2 .

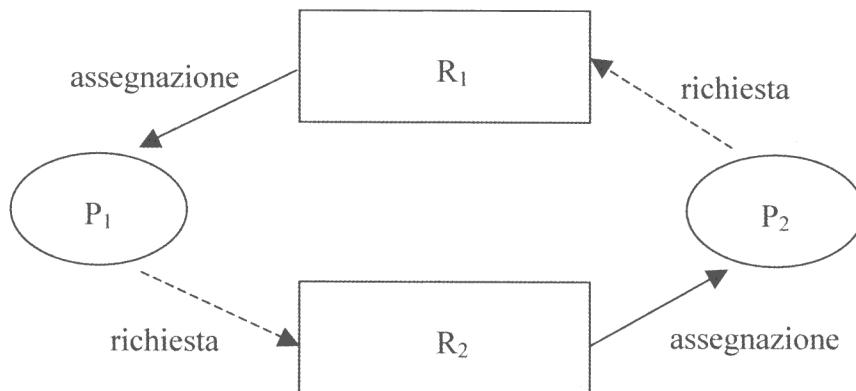


Figura 3.11 Catena circolare di bloccaggi.

La dipendenza del verificarsi delle situazioni di blocco critico dalla velocità relativa dei processi rende particolarmente complessa la loro individuazione a priori. In generale il blocco critico può avvenire solo dopo una rara combinazione di eventi, quindi un programma potrebbe funzionare per parecchio tempo, anche anni, prima di evidenziare un problema. Una strategia per gestire blocchi critici di questo tipo è di imporre a livello di progettazione, come vedremo nel seguito, dei vincoli sull'ordine delle richieste delle risorse.

In generale in un sistema di elaborazione esistono *diversi tipi* di risorse riusabili per ciascuna delle quali possano essere presenti più unità (*istanze*). Se un processo richiede un'istanza relativa ad un tipo di risorsa, l'assegnazione di qualsiasi istanza di quel tipo può soddisfare la richiesta.

Nell'utilizzo delle risorse riusabili i processi devono rispettare il seguente protocollo:

Richiesta: L'operazione può risultare bloccante se la risorsa non può essere immediatamente assegnata al processo, ad esempio in quanto già impegnata da un altro processo.

Utilizzo: Il processo opera sulla risorsa.

Rilascio: Il processo libera la risorsa.

La richiesta ed il rilascio avvengono tramite chiamate al sistema operativo mediante l'utilizzo di funzioni di sistema tipiche di ogni risorsa (*open-file*, *close-file*, *allocate-memory*, *free-memory* ecc.) che fanno uso di meccanismi di sincronizzazione.

Accanto alle risorse riusabili esistono in un sistema anche risorse non riusabili o consumabili.

Esempi di risorse consumabili sono i messaggi, i segnali e le interruzioni. Caratteristica fondamentale di questo tipo di risorsa è di essere distrutte (consumate) dopo il loro uso e di essere potenzialmente in numero infinito. Anche l'utilizzo di tali risorse può portare a delle situazioni di blocco critico.

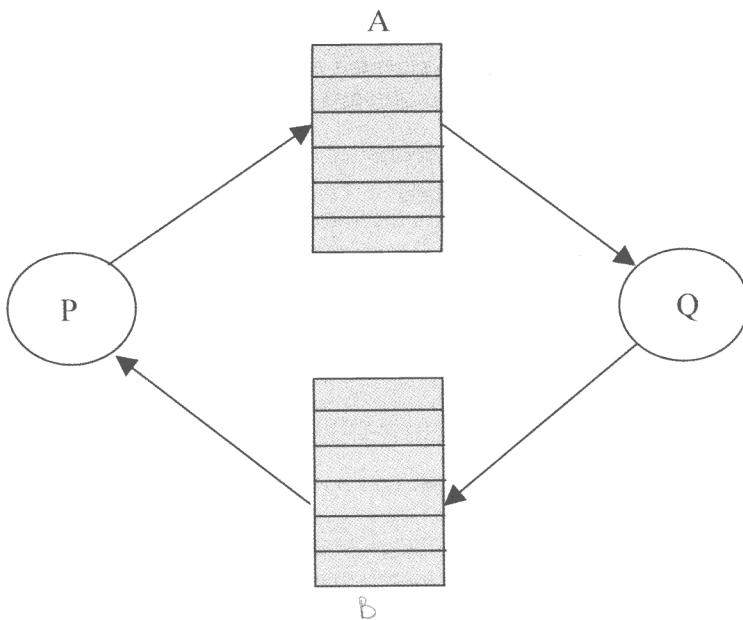


Figura 3.12 Blocco critico con risorse consumabili.

Si consideri l'esempio riportato nella figura 3.12. I processi P e Q sono rispettivamente produttore e consumatore con riferimento al buffer A e consumatore e produttore con riferimento al buffer B. Se i due buffer sono pieni, P non può inserire il suo messaggio nel buffer A e quindi si blocca in attesa che intervenga Q, il quale a sua volta può essere bloccato in quanto impossibilitato ad inserire il suo messaggio nel buffer B.

Per concludere, si può notare come il blocco critico rappresenti un problema di particolare importanza soprattutto in quei sistemi nei quali il blocco di un certo numero di processi e quindi l'impossibilità di eseguire particolari funzioni può provare gravi malfunzionamenti (si pensi ad esempio a sistemi per il controllo di impianti industriali). In generale poi, la situazione di blocco critico tende a coinvolgere gradualmente tutti i processi del sistema fino al suo completo arresto. Il problema è reso complesso dal fatto che, come si è visto, il blocco critico può avvenire solo dopo una ben precisa combinazione di eventi e quindi un programma potrebbe funzionare per molto tempo prima di evidenziare un problema.

La maggior parte dei sistemi operativi in uso non offre strumenti di prevenzione delle situazioni di stallo. D'altra parte l'aumento del numero dei processi e delle risorse all'interno dei sistemi e la diffusione di tecniche multi-threading fanno sì che il problema del blocco critico acquisti sempre maggiore importanza. In particolare, i programmi multithread possono essere soggetti a condizioni di blocco critico poiché proprietà fondamentale dei thread è quella di condividere e competere per le risorse definite all'interno dei processi.

Non esiste un'unica strategia efficace per trattare tutti i casi di blocco critico. Sono stati sviluppati diversi approcci che rientrano nelle due categorie di prevenzione, rilevamento e ripristino.

Nel seguito verranno indicati brevemente i principi su cui si basano tali tecniche, rinviando per una loro analisi più approfondita alla numerosa bibliografia in proposito.

3.7.1 Condizioni per il blocco critico

Dato un insieme di n processi P_1, P_2, \dots, P_n e di m tipi di risorse R_1, R_2, \dots, R_m si può verificare una condizione di blocco critico se sono vere *contemporaneamente* tutte le seguenti condizioni:

1. Le risorse possono essere utilizzate da un solo processo alla volta (*mutua esclusione*).
2. I processi trattengono le risorse che già possiedono mentre chiedono risorse addizionali (*possesso e attesa*).
3. Le risorse già assegnate ai processi non possono essere a questi sottratte (*mancanza di pre-rilascio*).
4. Esiste un insieme di processi P_i, P_{i+1}, \dots, P_k , tali che P_i è in attesa di una risorsa posseduta da P_{i+1} , P_{i+1} è in attesa di una risorsa posseduta da P_{i+2}, \dots, P_k è in attesa di una risorsa posseduta da P_i (*attesa circolare*).

Le prime tre condizioni sono *necessarie* ma non sufficienti per il verificarsi di una condizione di blocco critico. La quarta condizione deriva potenzialmente dalle prime tre, nel senso che in loro presenza si può verificare una condizione di attesa circolare.

Anche la quarta condizione è in generale una condizione solo necessaria per il blocco critico. Diventa *sufficiente* solo nel caso che per ogni tipo di risorsa riusabile sia definita una sola istanza.

Si consideri ad esempio il caso riportato nella figura 3.13, nella quale si è messo in evidenza che per il tipo di risorsa R_1 sono definite due istanze.

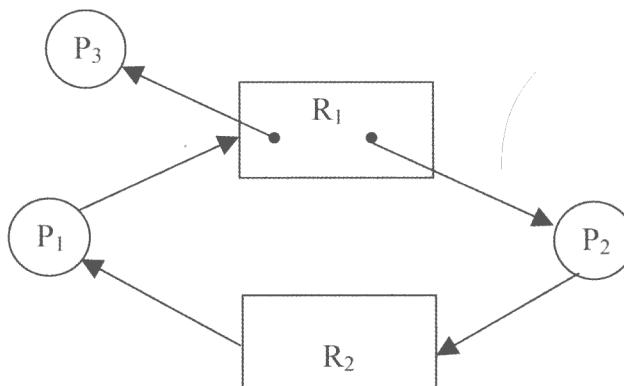


Figura 3.13 Più istanze della stessa risorsa.

La catena circolare costituita da $P_1 - R_1 - P_2 - R_2$ non porta ad un blocco critico in quanto il processo P_3 può rilasciare la propria istanza di R_1 che può quindi essere assegnata al processo P_1 rompendo la catena.

3.7.2 Metodi per il trattamento del blocco critico

Il problema del blocco critico si può affrontare utilizzando o tecniche di prevenzione che assicurino che la situazione di blocco critico non possa mai capitare oppure tecniche di individuazione di eventuali blocchi critici e successivo ripristino.

Per quanto riguarda la prevenzione si può distinguere tra *prevenzione statica* e *prevenzione dinamica*.

Prevenzione statica Consiste essenzialmente nell'assicurare, all'atto della scrittura dei programmi, che almeno una delle quattro condizioni necessarie descritte precedentemente non possa avverarsi. Tralasciando la condizione di mutua esclusione, che è una delle proprietà fondamentali delle risorse riusabili, si può intervenire sulle restanti tre condizioni.

a) Possesso ed attesa.

Si può imporre che un processo richieda all'inizio tutte le risorse che gli serviranno durante la sua esecuzione. Se tutte le risorse sono disponibili il processo può proseguire, diversamente viene bloccato. Occorre definire un'apposita chiamata di sistema che garantisca l'acquisizione *contemporanea* di tutte le risorse richieste.

La soluzione presenta lo svantaggio di obbligare il programmatore a conoscere in anticipo quali risorse richiederà il processo, il che è in contrasto con la struttura modulare e multithreading delle applicazioni. Inoltre, il processo è costretto ad attendere che tutte le risorse siano disponibili invece di cominciare l'esecuzione con quelle libere. Infine si ottiene una bassa utilizzazione delle risorse in quanto queste rimangano assegnate al processo per tutto il tempo della sua esecuzione anche se non utilizzate.

b) Mancanza di prerilascio.

Si può imporre che un processo che possiede una o più risorse, nel caso gli vengano rifiutate altre risorse, rilasci le risorse possedute. Il processo verrà nuovamente avviato solo quando potrà ottenere le risorse già in suo possesso e quelle che stava richiedendo.

In alternativa, le risorse richieste da un processo possono essere ottenute dal sistema operativo sottraendole ad un altro processo che è in attesa di altre risorse.

In generale la sottrazione di risorse ad un processo è un'operazione complicata, a meno che la natura delle risorse sia tale che il loro stato possa essere facilmente salvato e ripristinato in seguito, come nel caso dello spazio di memoria e del programma.

c) Attesa circolare

Si può imporre che le risorse vengano acquisite dai processi seguendo un ordine prefissato.

In particolare, le risorse vengono ordinate in una *gerarchia di livelli*: se un processo possiede una risorsa appartenente al livello d , nel seguito può richiedere solo risorse di livello k con $k > d$ (allocazione gerarchica delle risorse). Si può

facilmente dimostrare che tale ordinamento impedisce il sorgere di condizioni di blocco critico. Per assurdo, si potrebbe avere una situazione di stallo qualora il processo P_1 che possiede R_k richiedesse R_d posseduta da P_2 e questo a sua volta richiedesse R_k . Ne deriverebbe, stante la tecnica di allocazione gerarchica tra le risorse, che $k > d$ e $d > k$.

L'acquisizione delle risorse solo nell'ordine prestabilito può comportare un utilizzo inefficiente delle risorse stesse che possono risultare acquisite da un processo, ma non immediatamente utilizzate.

Prevenzione dinamica Le tecniche di prevenzione statica esaminate precedentemente sono basate sull'imposizione di vincoli nell'acquisizione delle risorse, il che, come si è visto, può provocare un uso non efficiente delle risorse ed un rallentamento dei processi.

Le tecniche di prevenzione dinamica non impongono vincoli preventivi sull'utilizzo delle risorse; si basano su algoritmi in grado di verificare sulla base dello stato corrente di allocazione delle risorse e delle richieste complessive dei processi se il soddisfacimento di una particolare richiesta da parte di un processo può portare ad una situazione di blocco critico (*stato non salvo*). Il più noto di questi algoritmi, *l'algoritmo del banchiere* [31] verifica se a fronte dell'eventuale soddisfacimento di una richiesta di un processo per una risorsa, esiste una sequenza di esecuzione per la quale tutti i processi possono completare la loro esecuzione utilizzando le risorse libere e quelle che vengono man mano liberate dai processi che terminano (*sequenza salva*).

Per illustrare brevemente il principio di funzionamento di una tecnica di prevenzione dinamica, consideriamo l'esempio di due processi P_1 e P_2 che utilizzano le due risorse riusabili R_1 e R_2 .

La figura 3.14 riporta sull'asse delle ascisse il progresso dell'esecuzione di P_1 e sull'asse delle ordinate il progresso dell'esecuzione di P_2 . Gli istanti t_1 e t_2 rappresentano i momenti in cui il processo P_1 richiede l'utilizzo rispettivamente di R_1 e R_2 , mentre t_3 e t_4 rappresentano i momenti in cui tali risorse vengono rilasciate. Lo stesso dicasi per il processo P_2 che richiede nell'ordine le risorse R_2 (istante t_5) e R_1 (istante t_6) che vengono rilasciate nello stesso ordine (istanti t_7 e t_8).

La traiettoria nella figura rappresenta il cammino di esecuzione dei due processi (orizzontale per P_1 , verticale per P_2) che parte dall'origine e prosegue alternando l'esecuzione di P_1 e P_2 .

Le zone grigie identificano *zone di impossibilità*, in cui, cioè, la traiettoria non può passare, essendo corrispondenti a stati di assegnazione delle risorse per i quali R_1 o R_2 o entrambe le risorse risulterebbero assegnate ad entrambi i processi.

Se la traiettoria di esecuzione entra nella *regione non salva* diventa inevitabile, come si può facilmente verificare, una situazione di blocco critico rappresentato dal punto A che corrisponde alla situazione in cui i due processi, possedendo ciascuno una risorsa, chiedono di poter utilizzare l'altra.

Obiettivo della prevenzione dinamica è impedire che un processo entri in una zona non salva. A tale scopo, nell'esempio di figura 3.14, la richiesta di P_2 per la risorsa R_2 all'istante t_5 deve essere rifiutata dal sistema operativo. Se la richiesta fosse accettata non esisterebbe infatti alcuna sequenza di esecuzione per la quale entrambi i processi possono completare la loro esecuzione.

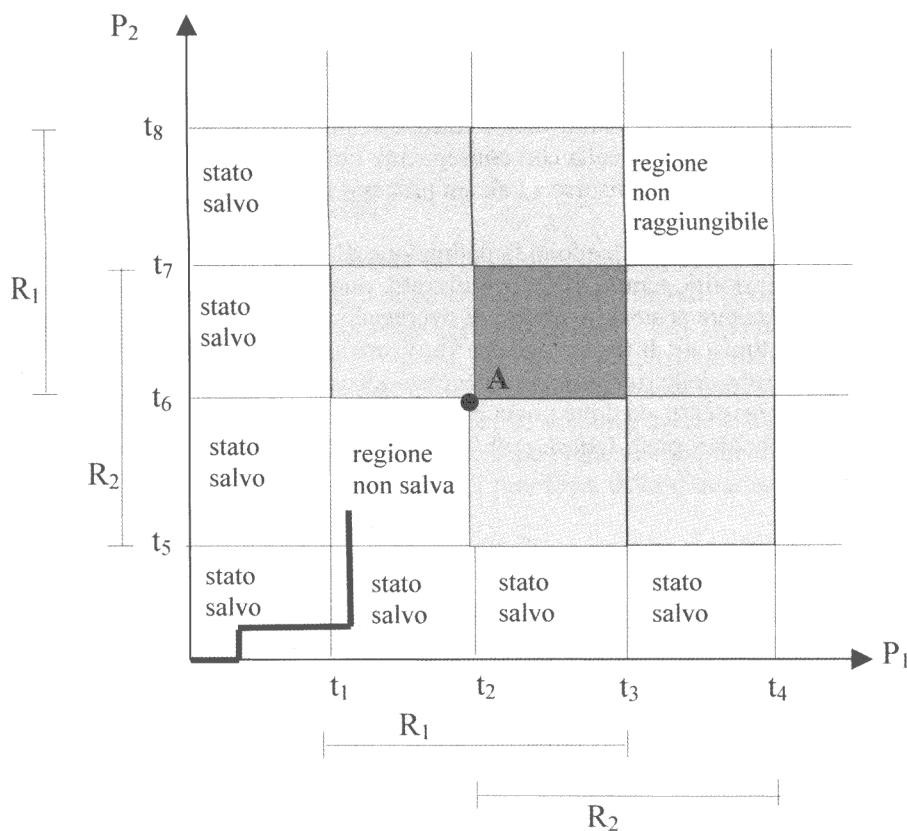


Figura 3.14 Prevenzione dinamica del blocco critico.

3.7.3 Individuazione di eventuali blocchi critici e successivo ripristino

Se non vengono adottate tecniche di prevenzione statica o dinamica del blocco critico, è possibile che una tale situazione si verifichi coinvolgendo un certo numero di processi e di risorse. La tecnica di individuazione e successivo ripristino prevede che il sistema operativo periodicamente metta in esecuzione un algoritmo per verificare se esistono processi coinvolti in un blocco critico ed in caso affermativo intervenga per ripristinare il sistema eliminando tale situazione.

L'algoritmo per l'individuazione del blocco critico opera verificando se tra tutti i processi ne esiste uno le cui richieste possono essere soddisfatte con le risorse non già assegnate ad altri processi. La ricerca viene ripetuta supponendo che il processo completi l'esecuzione e liberando quindi tutte le risorse possedute. Se tutti i processi possono completare l'esecuzione non c'è blocco critico, diversamente l'algoritmo determina quali processi sono coinvolti in un blocco critico.

Come si è detto, di norma l'algoritmo viene messo in esecuzione periodicamente con una frequenza che dipende dal tipo di applicazioni; un'alternativa potrebbe esse-

re quando il grado di utilizzazione della CPU scende sotto certi livelli in quanto una condizione di blocco critico può rendere inefficienti le prestazioni del sistema.

Il *ripristino* dal blocco critico può essere ottenuto con diverse strategie, la più comune delle quali è quella di abortire tutti i processi coinvolti. Per limitare la perdita di informazioni si può adottare una tecnica che prevede la terminazione forzata (*abort*) di un processo alla volta con conseguente liberazione delle risorse impegnate, oppure la sottrazione di risorse ad alcuni processi fino ad arrivare alla eliminazione del blocco critico.

Entrambe le strategie richiedono la definizione di criteri di scelta dei processi sui quali operare (priorità, tempo di CPU utilizzato, numero di risorse impegnate ecc.) che possono risultare costosi in termini di overhead in quanto dopo ogni terminazione forzata o sottrazione di risorse occorre verificare se c'è ancora blocco critico. Nel caso di sottrazione delle risorse ad un processo occorre inoltre che esso sia riportato in uno stato consistente, da dove possa proseguire l'esecuzione. Ciò è possibile se è previsto in particolari punti (*check point*) il salvataggio dello stato corrente del processo.

3.8 Sommario

Nel capitolo precedente si è visto come in presenza di interazioni il comportamento di un insieme di processi concorrenti risulti in genere non riproducibile e come sia necessario imporre vincoli, diversi a seconda del tipo di interazione, nell'esecuzione delle loro operazioni.

In questo capitolo si è provveduto innanzitutto ad un'analisi delle modalità d'interazione tra i processi, facendo riferimento sia al modello ad ambiente globale dove le interazioni tra processi avvengono utilizzando risorse comuni, sia al modello ad ambiente locale dove le interazioni avvengono utilizzando il meccanismo di scambio di messaggi.

Sono stati poi illustrati gli strumenti di sincronizzazione principali utilizzati per risolvere i problemi di mutua esclusione e comunicazione: i semafori e le primitive *wait* e *signal* e le primitive *send* e *receive*. È stato sottolineato come questi strumenti, seppur molto potenti in termini di flessibilità, possano risultare troppo elementari e quindi di non semplice uso quando il problema di sincronizzazione diventa complesso e come siano stati messi a punto strumenti di sincronizzazione di più alto livello quali il *monitor* e la *RPC*, per i quali si rimanda alla bibliografia.

L'utilizzo delle *variabili condizione*, tipiche del costrutto monitor, verrà affrontato nel capitolo 7 dedicato all'utilizzo dei thread come parte della libreria P-Thread.

È stato, infine, trattato il problema del blocco critico, cioè della situazione che si verifica quando in un insieme di processi ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme.

La maggior parte dei sistemi operativi in uso non offre strumenti di prevenzione di situazioni di blocco critico. D'altra parte l'aumento del numero di risorse e processi all'interno dei sistemi di elaborazione e la diffusione di tecniche di multithreading fanno sì che il problema del blocco critico acquisti sempre maggiore importanza. Dopo aver precisato le condizioni che devono verificarsi perché si instauri un blocco critico, sono state affrontate le tecniche di prevenzione, statica e dinamica, in

grado di assicurare che tale situazione non possa mai capitare. Infine, si è affrontato il problema del recupero da situazioni di blocco critico che possono essere utilizzate qualora non vengano adottate tecniche di prevenzione. Per un'analisi approfondita del problema del blocco critico e delle tecniche che si possono adottare si rinvia alla bibliografia.

3.9 Note bibliografiche

Un'analisi approfondita del problema dell'interazione tra processi si può trovare in [2], così come la descrizione degli strumenti di sincronizzazione per i due modelli a memoria comune e a scambio di messaggi. Testi classici sull'argomento sono [42] sulle modalità di interazione tra i processi, [31] per la descrizione del concetto di semaforo, [36] per l'utilizzo dei semafori nei problemi di mutua esclusione e comunicazione. Le principali notazioni adottate per la comunicazione e sincronizzazione dei processi nel modello ad ambiente locale sono contenute nel lavoro [3]. In [74] è contenuto un algoritmo per la soluzione del problema della mutua esclusione senza l'utilizzo dei semafori.

Strumenti di sincronizzazione a più alto livello rispetto ai semafori ed alle primitive `wait` e `signal`, `send` e `receive` si trovano in [2], [40], [9].

Per l'estensione al caso distribuito si può fare riferimento a [97].

Riferimenti classici per il problema del blocco critico sono costituiti da [41] e [20]. In [31] viene presentato l'algoritmo del banchiere per la rilevazione dinamica di condizioni di blocco.

4

Gestione della memoria

La memoria principale costituisce, insieme alla CPU, una delle risorse necessarie per fornire il supporto al concetto di processo. Infatti, per parlare di processo, e cioè di programma in esecuzione, è necessario avere a disposizione sia un'unità di elaborazione che esegue il programma sia un'area di memoria nella quale il programma deve essere allocato affinché possa essere eseguito. Compito di questo capitolo è quello di illustrare le varie tecniche utilizzate per gestire la memoria principale ed allocarla ai vari processi presenti nel sistema.

Partendo dagli aspetti che accomunano i criteri di gestione delle due risorse CPU-e memoria, quali ad esempio i meccanismi per la creazione e gestione di risorse virtuali, vengono successivamente illustrati gli aspetti caratterizzanti la gestione della memoria, quali la sua organizzazione sia logica che fisica, la rilocazione degli indirizzi logici nei corrispondenti indirizzi fisici, la protezione delle informazioni e la condivisione di codice e dati tra processi diversi.

Alla luce dei precedenti aspetti, vengono quindi presentati i principali meccanismi e i relativi algoritmi di gestione della memoria con riferimento, soprattutto, ai supporti forniti a questo proposito direttamente in hardware, noti con l'acronimo di dispositivi MMU (*Memory Management Unit*).

4.1 Introduzione alla gestione della memoria

Come è stato indicato nel primo capitolo, uno dei principali compiti di un sistema operativo è quello di controllare l'esecuzione dei programmi applicativi coordinando il regolare svolgimento dei relativi processi. Per assolvere a questo compito, il sistema operativo deve garantire a ciascun processo il corretto accesso alle risorse di cui ha bisogno. È quindi necessario che il sistema operativo fornisca gli strumenti per coordinare gli accessi alle risorse da parte dei processi secondo precise politiche di allocazione. Fra le risorse che il sistema operativo gestisce, e di cui un processo può aver bisogno durante la sua esecuzione, ve ne sono due senza le quali il concetto

stesso di processo perde di significato. Infatti, come è stato indicato nel paragrafo 2.1, il concetto di processo coincide con quello di programma in esecuzione, ma affinché un programma possa essere in esecuzione, è necessario che siano disponibili le seguenti risorse:

- un'unità di elaborazione (l'unità che esegue il programma);
- un'area di memoria principale (l'area nella quale il programma è residente).

Partendo da questa semplice osservazione è possibile identificare tutta una serie di analogie nella gestione delle due risorse, CPU e memoria, e introdurre quindi le problematiche relative alla gestione della memoria partendo da quelle già note, esaminate nel capitolo 2, relative alla gestione della CPU. Chiaramente, la diversa natura delle due risorse implica che, al di là delle analogie, siano presenti anche molte differenze sia nei meccanismi di supporto che nelle strategie adottate per una loro efficiente gestione.

4.1.1 Analogie con la gestione della CPU

Il criterio con cui sono allocate ai processi molte delle risorse di un sistema (ad esempio le periferiche di ingresso/uscita o i files) è quello di lasciare ad ogni processo il compito di richiedere, tramite opportune chiamate di sistema, il diritto di operare su una risorsa quando la stessa è necessaria per la prosecuzione del processo. Sarà poi compito del gestore di tale risorsa accordare il diritto, o temporaneamente rifiutarlo nel caso in cui la risorsa sia stata già allocata ad un altro. Una volta che il processo ha ottenuto il diritto di operare sulla risorsa richiesta, e dopo avervi operato, è ancora compito del processo indicare al gestore della risorsa che la stessa non è più necessaria e che può essere quindi disponibile per altri. Questo criterio è spesso noto con il termine di *allocazione non revocabile (non preemptive)* in quanto denota che una risorsa, una volta allocata ad un processo, non può essere revocata d'autorità dal sistema, ma è il processo stesso che decide quando rilasciarla.

La principale analogia che possiamo riscontrare nei criteri di gestione delle due risorse CPU e memoria deriva proprio, come è stato già detto, dalla stessa definizione di processo e consiste nell'adozione di un criterio di allocazione diametralmente opposto a quello indicato precedentemente. Infatti, si capisce subito che non ha senso pretendere che sia il processo a richiedere l'uso della CPU perché ciò implica che il processo sia in esecuzione, e quindi che abbia già la disponibilità della CPU. Analogamente, e per lo stesso motivo, non ha senso pretendere che sia il processo a richiedere la memoria perché per richiederla il processo deve essere in esecuzione e quindi il programma che lo stesso sta eseguendo deve già essere allocato in memoria.

Questa considerazione ci spinge a ritenere che a ogni processo debbano necessariamente essere biunivocamente associate le due risorse che sono necessarie per la sua stessa esistenza: un processore e un'area di memoria in cui allocare il programma che il processo deve eseguire.

Come è stato già indicato nel secondo capitolo, ciò crea un'apparente contraddizione nel momento in cui nel sistema il numero dei processi è superiore a quello dei processori. Come è noto, tale contraddizione viene eliminata mediante la tecnica della virtualizzazione della CPU. Ad ogni processo, all'atto della sua creazione, viene biunivocamente associato un processore virtuale, che come è stato indicato nel

capitolo 2, è rappresentato da un campo del descrittore del processo, il campo dove trovano spazio i valori di tutti i registri del processore virtuale. È il processore reale che viene allocato dinamicamente dal sistema per supportare, di volta in volta, i diversi processori virtuali. E ciò avviene quando lo decide il sistema, e non su richiesta dei singoli processi, secondo le scelte implementate dall'algoritmo di *scheduling* del processore. Ebbene, questo criterio di allocazione della risorsa CPU, mediante virtualizzazione (assegnazione di una CPU virtuale ad ogni processo e allocazione dinamica della CPU reale) viene adottato anche per l'allocazione della memoria. Anche in questo caso, e per gli stessi motivi, ad ogni processo viene biunivocamente associata una memoria virtuale nella quale viene allocato il programma che il processo deve eseguire. La somma delle dimensioni di tutte le memorie virtuali dei processi è normalmente superiore alla dimensione della memoria reale. Sarà quindi compito del sistema operativo allocare la memoria reale per fornire, di volta in volta, il supporto alle diverse memorie virtuali.

4.1.2 Differenze rispetto alla gestione della CPU

A fronte delle precedenti analogie di criteri di allocazione delle due risorse CPU e memoria, molte sono ovviamente le differenze fra le tecniche utilizzate nei due casi, differenze legate alla diversa natura delle due risorse.

La prima differenza sostanziale fra la gestione della memoria e quella della CPU è legata alla diversa struttura fisica delle due risorse e quindi alla diversa organizzazione delle corrispondenti risorse virtuali. Una risorsa virtuale non è altro che una struttura dati che deve rappresentare lo stato della risorsa fisica quando questa non è allocata al corrispondente processo. In questo senso la CPU virtuale è costituita da una struttura dati allocata nella memoria del sistema e, in particolare, in un campo del descrittore del processo (campo *contesto del processo*, vedi paragrafo 2.3). Ovviamente ciò non potrà essere vero per la memoria virtuale, che quindi dovrà essere allocata in una memoria diversa dalla memoria principale. Possiamo ipotizzare che sia riservata, per questo scopo, un'area nel disco di sistema, spesso indicata come *swap-area*, dedicata a contenere le memorie virtuali di tutti i processi non allocati in memoria principale.

Una seconda differenza riguarda il fatto che la risorsa fisica, e cioè la memoria principale, a differenza della CPU può essere utilizzata per fornire il supporto a diversi processi contemporaneamente. Infatti, più processi contemporaneamente possono trovare spazio in memoria principale. Questo fatto, come vedremo, crea tutta una serie di conseguenze di cui sarà necessario tener conto.

Per prima cosa nasce un problema di *protezione*, legato al fatto che un processo in esecuzione potrebbe, erroneamente, eseguire istruzioni che modificano locazioni di memoria allocate ad un diverso processo, compromettendone così la correttezza. Una seconda conseguenza della contemporanea presenza di più processi in memoria, che in questo caso costituisce un'opportunità, è legata al fatto che più processi possono eseguire lo stesso programma, chiaramente operando su dati diversi. Si pensi ad esempio a due processi ciascuno dei quali sta editando un diverso testo con lo stesso word processor. L'opportunità che nasce in questo caso è quella della *condivisione (sharing)* del codice se viene consentito ai due processi di eseguire la stessa copia del programma in memoria senza doverlo caricare due volte in due diverse

arie contemporaneamente. Analogamente possiamo pensare a due o più processi che condividono dei dati.

4.2 Aspetti caratterizzanti la gestione della memoria

Come vedremo, molte sono le tecniche adottate nei sistemi operativi per gestire la memoria, spesso molto diverse tra loro, tecniche che dipendono da alcuni fattori tra loro interconnessi e che a loro volta dipendono da un lato dall'architettura fisica del processore, dall'altro dalle scelte operate in fase di progetto del sistema operativo.

Un primo aspetto è relativo alla disponibilità di eventuali supporti hardware per la gestione della memoria (*MMU – Memory Management Unit*). Per comprendere l'importanza di questo aspetto, supponiamo che il numero di processi presenti nel sistema sia tale per cui non sia possibile caricarli tutti in memoria contemporaneamente. In questo caso, possiamo adottare la stessa politica di allocazione con revoca (*preemption*), già vista per la CPU, in base alla quale ad un processo può essere revocata la memoria ad esso assegnata prima che lo stesso sia terminato. Quando ciò accade, il processo viene trasferito su memoria di massa nella *swap-area* (*swap_out*) per far posto ad altri processi. Successivamente verrà ricaricato in memoria (*swap_in*) per continuare la sua esecuzione. In generale non sarà sempre facile, a meno di non porre forti vincoli al gestore della memoria, allocare il processo nelle stesse locazioni in cui era presente prima della revoca. D'altra parte, come vedremo, a meno di non disporre di opportuni dispositivi hardware (MMU), il processo non potrà essere rilocato in una diversa porzione di memoria quando, dopo essere stato trasferito sulla *swap-area*, viene deciso di ricaricarlo in memoria. Useremo il termine di *rilocazione dinamica* per indicare la capacità di un sistema di caricare un processo in aree di memoria diverse da quella in cui era stato precedentemente allocato.

Un secondo aspetto riguarda l'*organizzazione logica della memoria virtuale*. Come vedremo, la memoria virtuale non è altro che una struttura dati destinata a rappresentare le esigenze di memoria di un processo. Tali esigenze potranno essere rappresentate in termini di un unico insieme di locazioni, con indirizzi contigui, oppure mediante più insiemi di locazioni, spesso indicati col termine di *segmenti*. Ciascun segmento è ancora un insieme di locazioni contigue, ma ogni segmento è completamente indipendente dagli altri.

Il terzo aspetto riguarda l'*organizzazione fisica* della porzione di memoria utilizzata per allocare un processo. Il caso più semplice è quello di riservare per la memoria virtuale del processo, o per ciascuno dei suoi segmenti, un'area di memoria fisica costituita da un insieme di locazioni contigue. L'alternativa, sicuramente più complicata ma come vedremo molto più efficiente, è quella di allocare il processo in locazioni di memoria non necessariamente contigue.

Un ultimo aspetto riguarda la *dimensione della memoria virtuale*. Il caso più ovvio è quello di limitare la memoria virtuale di un processo in modo tale che non superi la dimensione della memoria fisica. Ciò pone ovviamente un limite superiore alla dimensione dei programmi che possono essere eseguiti nel sistema, a meno di non obbligare il programmatore all'uso di complicate tecniche di caricamento dina-

mico, (tecniche di *overlay*, vedi paragrafo 4.2.5) ormai cadute in disuso. L'alternativa, di gran lunga più semplice da usare, consiste nel consentire dimensioni della memoria virtuale superiori a quelle della memoria fisica, delegando al sistema operativo il compito di caricare un po' per volta, *a domanda*, singole porzioni della memoria virtuale, e cioè soltanto quando queste sono richieste, scaricando le porzioni non più necessarie.

4.2.1 La memoria virtuale

Cerchiamo adesso di caratterizzare meglio cosa intendiamo con il termine di memoria virtuale¹ e chi ha il compito di crearla.

Per definire questo concetto possiamo adottare lo stesso ragionamento già visto per la CPU. In quel caso, il processore virtuale coincide con l'insieme dei registri di macchina, ivi inclusi i valori in essi contenuti, e a cui il processo può fare riferimento durante la sua esecuzione. Infatti, le informazioni che devono essere salvate durante una commutazione di contesto corrispondono proprio ai valori dei registri di macchina da memorizzare all'interno dell'area *contesto* del descrittore del processo. Per analogia, possiamo definire la memoria virtuale di un processo come l'insieme di tutte le locazioni di memoria e delle informazioni in esse contenute i cui indirizzi possono essere generati dalla CPU durante l'esecuzione del processo. Quindi tutte le locazioni contenenti le istruzioni del programma eseguito dal processo, quelle contenenti i dati su cui il programma opera, oltre alle locazioni utilizzate per contenere lo stack del programma.

Da un punto di vista del tutto concettuale, potremmo quindi identificare la memoria virtuale di un processo come la memoria necessaria a contenere il suo programma (il codice), i dati su cui deve operare e l'area per lo stack, nell'ipotesi che la memoria fisica fosse tutta dedicata a questo unico processo, come indicato nella figura 4.1 dove è riportata quella che viene indicata normalmente come immagine del processo in memoria.

In questo semplice esempio si è supposto che le istruzioni del programma occupino le prime 4K locazioni. A partire dalla locazione di indirizzo 4096 sono contenuti i dati, per un totale di 1.5K locazioni e a seguire l'area di stack per ulteriori 0.5K locazioni. Nell'esempio si è supposto che l'*entry point* del programma, cioè la prima istruzione che dovrà essere eseguita, sia contenuta nella locazione di indirizzo 160 e che la base dello stack sia costituita dalla locazione 6140, nell'ipotesi che lo stack cresca verso le locazioni di indirizzo decrescente. Con queste ipotesi, all'inizio dell'esecuzione, i due valori 160 e 6140 devono essere caricati rispettivamente nei due registri PC e SP.

Avendo quindi definito la memoria virtuale di un processo come la porzione di memoria fisica da esso utilizzata a partire dalla locazione fisica di indirizzo zero e quindi nell'ipotesi che nessun altro processo sia contemporaneamente allocato in memoria, ne consegue che la struttura della memoria virtuale viene a coincidere con

¹ È opportuno notare che in molti testi di sistemi operativi il concetto di memoria virtuale viene spesso associato al caso particolare in cui la memoria richiesta dal processo abbia dimensioni superiori a quelle della memoria fisica.

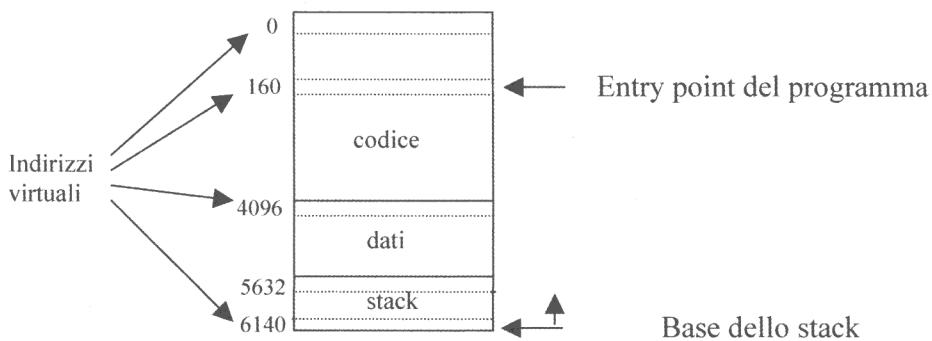
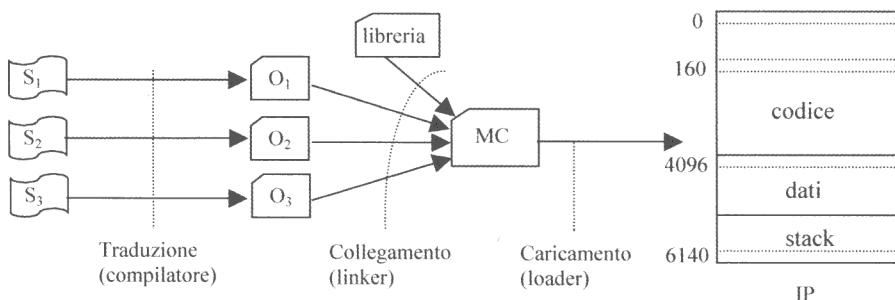


Figura 4.1 Immagine di un processo.

il contenuto di quel particolare file (*file eseguibile*) nel quale sono presenti, in formato binario e pronte per essere caricate in memoria, tutte le informazioni relative al programma da eseguire (immagine del processo).

E utile, al fine di chiarire questi concetti, richiamare brevemente come viene creato il file eseguibile per mezzo dei normali strumenti disponibili in un ambiente di programmazione (compilatori e programma di collegamento, spesso indicato col termine di *linker* o *linkage editor*).

Nel caso più generale, un programma consiste in un insieme di moduli, ciascuno scritto in un particolare linguaggio di programmazione (*moduli sorgente*). Ogni modulo viene separatamente compilato (o assemblato), generando un file che contiene il relativo codice oggetto, cioè le istruzioni e i dati relativi al modulo sorgente tradotti in linguaggio macchina (*modulo oggetto*). In realtà non tutte le informazioni presenti in un modulo sorgente possono esser tradotte in linguaggio macchina dal compilatore. Infatti, se in un modulo sorgente sono presenti riferimenti a informazioni



S_1, S_2, S_3 : moduli sorgente; O_1, O_2, O_3 : moduli oggetto;
 MC: modulo di caricamento (file eseguibile); IP: immagine del processo

Figura 4.2 Preparazione di un programma per l'esecuzione.

relative ad altri moduli compilati separatamente, questi riferimenti restano ancora in forma simbolica nel modulo oggetto, in quanto il compilatore non è in grado di risolverli. Successivamente tutti i file contenenti i moduli oggetto devono essere collegati insieme in modo tale da risolvere tutti i riferimenti fra moduli diversi. Allo stesso tempo, qualora in uno qualunque dei moduli sorgente siano presenti riferimenti a funzioni di libreria (moduli già disponibili come codice oggetto) anche questi ultimi vengono collegati al fine di ottenere un unico file eseguibile (*modulo di caricamento*) dove tutti i riferimenti sono risolti e dove a tutte le informazioni costituenti il programma, istruzioni e dati, sono assegnati indirizzi consecutivi a partire dall'indirizzo zero (vedi figura 4.2).

Ovviamente, quanto riportato nelle due figure 4.1 e 4.2 vuole essere solo un semplice esempio. In generale, la struttura del modulo di caricamento, e quindi della memoria virtuale di un processo, dipende molto dall'architettura hardware del processore.

4.2.2 Rilocazione statica e dinamica - Memory Management Unit

Rilasciando adesso l'ipotesi che tutta la memoria fisica sia a disposizione di un solo processo e considerando, quindi, il caso reale in cui ogni processo debba occupare una porzione della memoria, mentre altre porzioni saranno occupate da processi diversi, è necessario tener conto che l'area di memoria assegnata ad un processo non potrà coincidere con quella indicata in figura 4.1, che inizia con la locazione di indirizzo zero.

In generale, l'area assegnata ad un processo dovrà avere una dimensione in byte sufficiente a contenere l'intera memoria virtuale del processo ma localizzata nella memoria fisica a partire da un qualunque *indirizzo I* diverso da zero. Sarà quindi opportuno, da ora in poi, distinguere gli indirizzi della memoria virtuale di un processo (*indirizzi virtuali*) dagli indirizzi di memoria fisica corrispondenti alle locazioni fisiche nelle quali il codice e i dati del processo sono stati caricati (*indirizzi fisici*). Ad esempio, nel caso di un processo che debba eseguire il programma il cui modulo di caricamento è quello riportato in figura 4.1, l'insieme di tutti gli indirizzi virtuali del processo (*spazio degli indirizzi virtuali*) corrisponde a tutti i valori compresi tra 0 e 6140 (6 Kbyte). Se tale processo fosse allocato a partire dalla locazione di indirizzo fisico 10240, gli indirizzi fisici ad esso associati (*spazio degli indirizzi fisici*) sarebbero tutti quelli relativi ai 6 Kbyte a partire da quello di indirizzo 10240.

I due spazi di indirizzi, quello virtuale e quello fisico, non sono ovviamente indipendenti, bensì legati da una funzione, nota con il termine di *funzione di rilocazione*, che appunto dipende dalla porzione di memoria fisica utilizzata per caricare l'immagine del processo:

$$y = f(x)$$

e che fa corrispondere ad ogni indirizzo virtuale x l'indirizzo fisico y della locazione di memoria nella quale è stata caricata l'informazione (istruzione o dato) il cui indirizzo virtuale è x . Nella figura 4.3 viene indicato, con riferimento allo stesso esempio delle precedenti figure, la generazione del modulo di caricamento da parte del linker, e l'immagine del processo in memoria successivamente al suo caricamento.

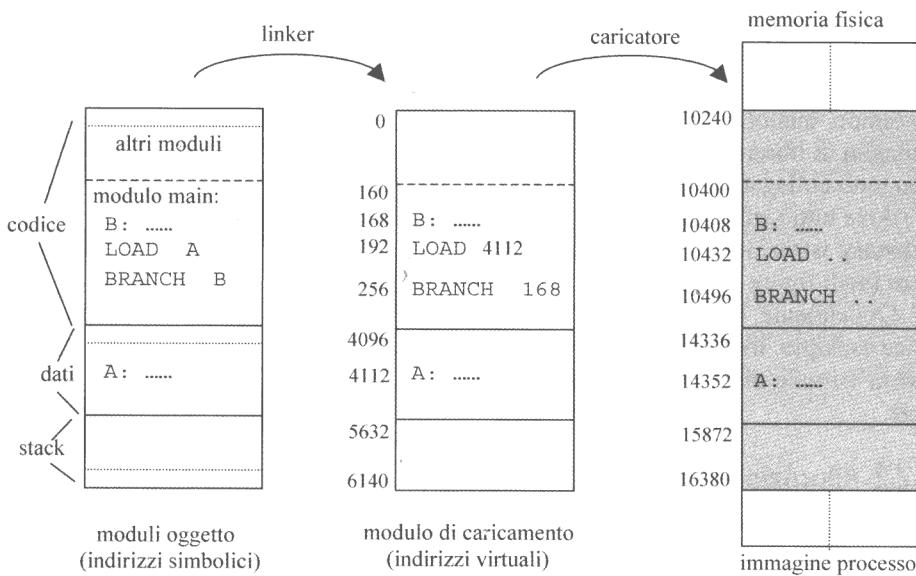


Figura 4.3 Generazione della memoria virtuale e dell'immagine di un processo.

Come si può notare, il modulo di caricamento non è altro che l'immagine del processo allocato nella sua memoria virtuale a cui corrisponde, dopo il caricamento in memoria a partire dall'indirizzo fisico 10240, la sua immagine in memoria fisica pronta per l'esecuzione.

In questo semplice esempio la funzione di rilocazione si riduce a una somma. Infatti l'indirizzo fisico y che corrisponde ad un indirizzo virtuale x si ottiene sommando a quest'ultimo l'indirizzo fisico a partire dal quale il processo è stato caricato:

$$y = x + 10240$$

Per un motivo di leggibilità, in figura 4.3, nei moduli oggetti, nel modulo di caricamento e nell'immagine del processo in memoria, sono riportate alcune istruzioni ancora in forma simbolica, e non in linguaggio macchina come in realtà dovrebbero comparire.

Quando il processo è in esecuzione, la CPU genera sia gli indirizzi delle istruzioni da eseguire, durante le fasi di prelievo (*fetch*), sia gli indirizzi degli operandi, durante le fasi di esecuzione delle istruzioni. Tali indirizzi sono quindi inviati all'unità di memoria per accedere alla corrispondenti locazioni fisiche. Ad esempio, con riferimento al caso presentato in figura 4.3, per eseguire l'istruzione LOAD A, che nella memoria virtuale del processo occupa la locazione 192, è necessario prelevare l'istruzione dalla locazione fisica $192 + 10240 = 10432$. Analogamente, durante la sua esecuzione l'istruzione deve accedere all'operando A il cui indirizzo virtuale è 4112 e quindi leggere la locazione fisica $4112 + 10240 = 14352$. In altri termini, per accedere alle locazioni fisiche in cui si trovano sia le istruzioni da eseguire che i dati da elaborare, è necessario rilocare gli indirizzi virtuali nei corrispondenti indirizzi fisici.

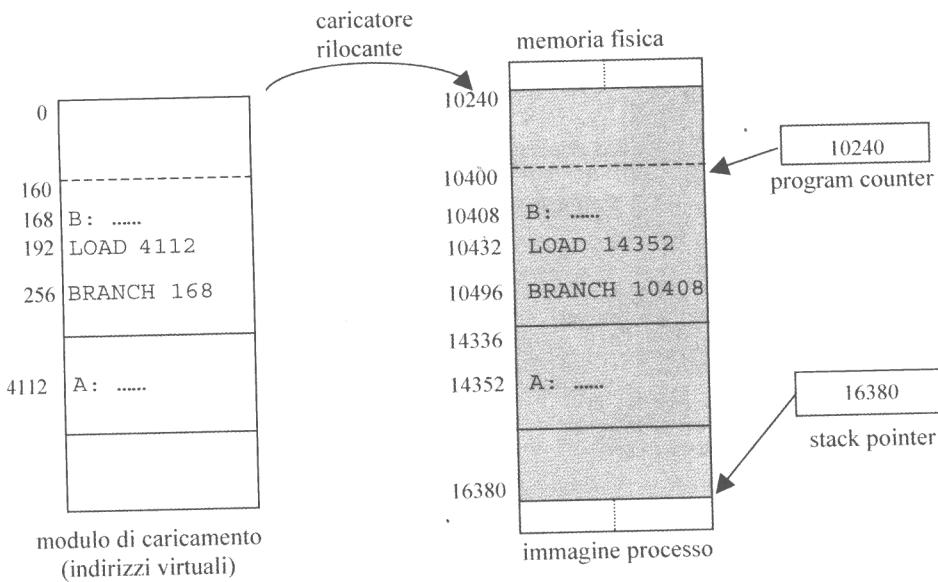


Figura 4.4 Rilocazione statica.

Due sono le tecniche che possono essere utilizzate per effettuare tale rilocazione. La più semplice consiste nel rilocare tutti gli indirizzi da virtuali a fisici prima che il processo inizi la sua esecuzione (*rilocazione statica*), e cioè durante la fase di caricamento. In questo caso il caricatore, che per questo viene anche chiamato *caricatore rilocante*, nel trasferire in memoria il modulo di caricamento, modifica tutti gli indirizzi da virtuali a fisici sommandovi l'indirizzo iniziale di caricamento. L'immagine del processo in memoria contiene quindi tutti gli indirizzi già rilocati rispetto alla porzione di memoria fisica utilizzata per caricare il processo (vedi figura 4.4).

All'atto della creazione del processo i due registri PC e SP (*stack pointer*) vengono inizializzati rispettivamente con il valore dell'indirizzo fisico corrispondente all'*entry point* del programma e con quello corrispondente alla base dello stack.

Questa soluzione, per quanto semplice, è molto limitativa in quanto, una volta caricato in memoria un programma, questo non può essere più rilocato in una diversa porzione di memoria. Supponiamo, ad esempio, che la memoria assegnata a un processo debba essere revocata per fare spazio ad altri processi. In questo caso l'immagine del processo viene trasferita nella *swap-area* su memoria di massa (*swap_out*) da cui, successivamente, verrà recuperata e ricaricata in memoria (*swap_in*). Ebbene, la scelta di aver rilocato staticamente in memoria il processo, e cioè di aver definito il suo spazio degli indirizzi fisici prima dell'inizio della sua esecuzione, non consente all'atto dell'operazione di *swap_in* di scegliere una diversa porzione di memoria in cui riallocare il processo. Come si può facilmente intuire, questo vincolo limita fortemente la flessibilità con cui la memoria può essere allocata ai processi, riducendo notevolmente l'efficienza del sistema.

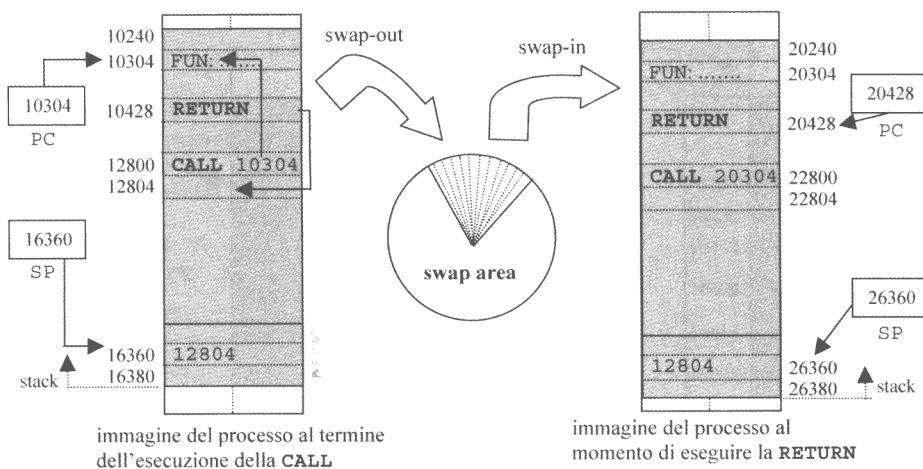


Figura 4.5 Revoca di memoria e successiva riallocazione in una diversa partizione.

Per renderci conto del perché di questa limitazione, supponiamo di fare ancora riferimento all'esempio riportato in figura 4.4. Supponiamo anche che il processo, una volta caricato nella porzione di memoria di indirizzo iniziale 10240, inizi la sua esecuzione e, a un certo punto, esegua una **CALL FUN** per eseguire la funzione **FUN**. Supponendo che la funzione **FUN** sia allocata all'indirizzo virtuale 64, e quindi all'indirizzo fisico $10240 + 64 = 10304$, nella parte sinistra della figura 4.5 viene riportata l'immagine del processo subito dopo l'esecuzione della **CALL**.

Come si può notare, il *program counter* contiene l'indirizzo fisico 10304 della prima istruzione della funzione **FUN**, mentre in cima allo stack è stato salvato l'indirizzo di ritorno, cioè l'indirizzo 12804 dell'istruzione successiva alla **CALL**. Nell'esempio si è supposto che l'indirizzo della cima dello stack sia 16360, come indicato in figura, indirizzo che è contenuto nel registro *stack pointer*. Supponiamo quindi che il processo, durante l'esecuzione della funzione **FUN**, subisca una revoca di memoria e venga trasferito sulla *swap-area* da cui, successivamente, venga ricaricato in memoria. Se a questo punto il processo fosse rilocato in una diversa partizione della memoria, ad esempio quella che parte dall'indirizzo fisico 20240, tutti gli indirizzi virtuali sarebbero di nuovo rilocati nella nuova partizione. Nella parte destra della figura 4.5 viene illustrato la nuova immagine del processo, relativa in particolare all'istante di esecuzione della **RETURN** dalla funzione **FUN**. Come si può notare, pur avendo diversamente rilocato il processo, purtroppo alcune delle informazioni che il processo ha prodotto prima della revoca, ad esempio le informazioni contenute nello stack, restano legate agli indirizzi fisici originali. Per cui dopo l'esecuzione dell'istruzione **RETURN** la CPU andrebbe a cercare la successiva istruzione da eseguire nella vecchia locazione 12804 e non in quella adesso utilizzata di indirizzo 22804, facendo fallire l'esecuzione del programma.

Per evitare questi inconvenienti legati alla rilocazione statica, è necessario ritardare la rilocazione stessa attuandola in fase di esecuzione (*rilocazione dinamica*). In

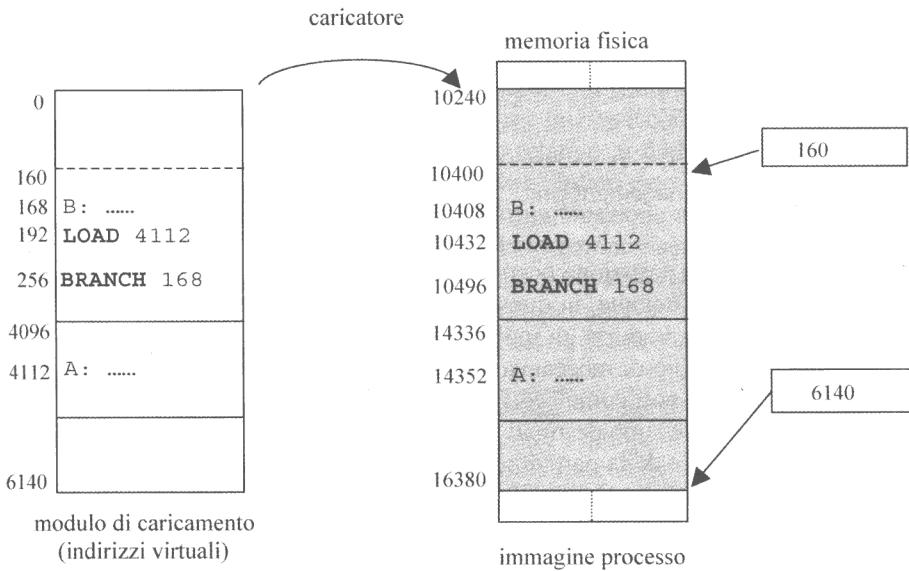


Figura 4.6 Rilocazione dinamica.

questo caso, quando un processo viene allocato in memoria, il caricatore trasferisce nelle locazioni della partizione scelta, direttamente i contenuti del modulo di caricamento contenente gli indirizzi virtuali, quindi senza rilocare il programma (vedi figura 4.6).

Come si può notare, nell'immagine del processo in memoria le singole istruzioni contengono gli indirizzi ancora nella loro versione virtuale, che è quindi indipendente dalle particolari locazioni fisiche contenenti le istruzioni e i dati. Per lo stesso motivo, all'inizio dell'esecuzione del processo, nel registro *program counter* viene caricato l'indirizzo virtuale della prima istruzione da eseguire e non l'indirizzo fisico della locazione in cui tale istruzione è contenuta. Analogamente nello *stack pointer* viene posto l'indirizzo virtuale della base dello stack.

In questo modo la CPU, sia in fase di prelievo (*fetch*) che in fase di esecuzione di ogni istruzione, genera indirizzi virtuali e non indirizzi fisici. Per accedere alla memoria è però necessario fornire degli indirizzi fisici. Per questo motivo, l'architettura hardware di un processore che permette di effettuare la rilocazione dinamica presenta un particolare dispositivo (noto con il termine di *Memory Management Unit*) che interfaccia la CPU all'unità di memoria (vedi figura 4.7) e che implementa la funzione di rilocazione $y = f(x)$ traducendo dinamicamente ogni indirizzo virtuale nel corrispondente indirizzo fisico.

Ciò consente di rilicare dinamicamente un processo in memoria. Ad esempio, con riferimento al caso precedentemente esaminato, qualora un processo subisca una revoca di memoria e successivamente venga riallocato in una diversa partizione,

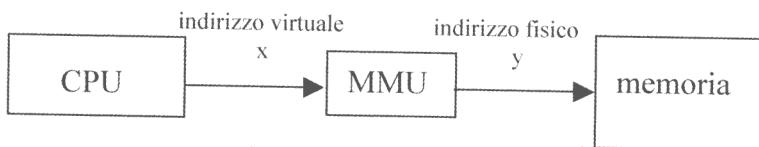
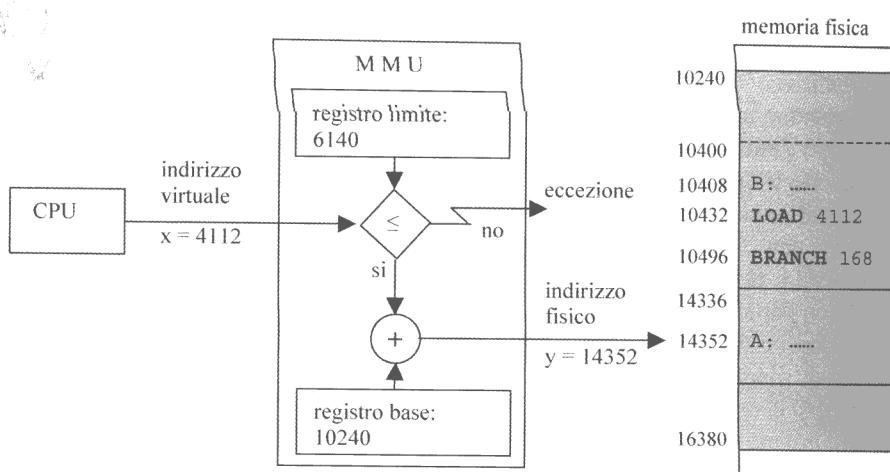


Figura 4.7 Memory Management Unit.

dipendentemente dalla partizione occupata. Affinché il processo possa ripartire nella nuova partizione è sufficiente, in questo caso, modificare i parametri con cui opera la MMU abilitandola a tradurre gli indirizzi virtuali nei nuovi indirizzi fisici.

Un semplice esempio di meccanismo per la rilocazione dinamica è costituito da una coppia di registri, noti come registro base e registro limite, che fanno parte del dispositivo MMU e che sono destinati a contenere, rispettivamente, l'indirizzo fisico iniziale e la dimensione della partizione di memoria nella quale è caricata l'immagine del processo. Ogni volta che la CPU genera un indirizzo virtuale x , questo viene sommato al contenuto del registro base al fine di ottenere il corrispondente indirizzo fisico y . L'indirizzo virtuale x viene anche confrontato con il valore del registro limite. Solo se x è inferiore o uguale al valore del registro limite l'indirizzo fisico y viene inviato alla memoria fisica per accedere all'informazione desiderata. Viceversa, se x supera il valore del registro limite, viene generata un'interruzione per segnalare che si è usciti dallo spazio di memoria assegnato. In definitiva il confronto col registro limite rappresenta un semplice meccanismo di protezione che garantisce l'impossibilità per un processo di accedere a informazioni che non gli appartengono.

In figura 4.8 viene illustrato questo semplice esempio di Memory Management Unit che garantisce la rilocabilità dinamica. Infatti, se a un processo viene revocata



memoria (`swap_out`) e successivamente viene riallocata una diversa partizione, quando il processo viene di nuovo schedulato, è sufficiente caricare nel registro base l'indirizzo iniziale della nuova partizione assegnata al processo.

4.2.3 Organizzazione della memoria virtuale

Negli esempi visti nel precedente paragrafo si è fatto implicitamente riferimento al caso in cui la memoria virtuale di un processo sia costituita da un unico spazio di indirizzi virtuali (gli indirizzi compresi tra zero e 6140 nell'esempio riportato in figura 4.1). Ciò perché è stata fatta implicitamente l'ipotesi che il linker allochi tutti i moduli componenti il programma a indirizzi virtuali contigui. Questa ipotesi però non è necessariamente l'unica possibile. In particolare, con riferimento all'immagine del processo indicata in figura 4.1, si può notare che questa è composta da tre regioni, o segmenti: quello contenente il codice, quello relativo ai dati e quello per lo stack, che sono del tutto indipendenti l'uno dall'altro. Infatti non c'è nessun particolare motivo che obblighi il linker a riservare la prima locazione del segmento dati nella locazione di indirizzo successivo a quello della locazione riservata per l'ultima istruzione del segmento codice. La stessa cosa vale per quanto riguarda l'indipendenza delle locazioni dello stack rispetto a quelle usate per allocare i dati. In questo senso i tre segmenti potrebbero essere rilocati in memoria fisica l'uno indipendentemente dall'altro.

Per ottenere questo risultato dovremmo complicare leggermente il linker in modo tale che, nel predisporre il modulo di caricamento, e quindi nel predisporre la memoria virtuale del processo, riservasse non un unico spazio virtuale contiguo nel quale allocare tutte le informazioni, ma tre diversi sotto-spazi, uno per il codice (segmento 0), uno per i dati (segmento 1) e uno per lo stack (segmento 2), ciascuno contenente indirizzi che partono da zero. Successivamente, nel modulo di caricamento, ogni indirizzo virtuale del processo dovrebbe essere rappresentato mediante un coppia di interi: un intero che identifica il segmento a cui l'indirizzo appartiene e l'altro intero che rappresenta la locazione nell'ambito del segmento. Per esemplificare, possiamo ancora fare riferimento allo stesso esempio della figura 4.1. In particolare, nella figura 4.9, nella parte a) alla sinistra, viene di nuovo indicato come sarebbe strutturato il modulo di caricamento nell'ipotesi di uno spazio virtuale unico (come già visto in figura 4.1), mentre nella parte b) al centro, viene indicato come sarebbe strutturato lo spazio virtuale nell'ipotesi che questo fosse segmentato. Come si può notare, l'istruzione simbolica `LOAD A`, presente alla locazione 192 del segmento di codice, viene tradotta come `LOAD (1..16)`. Infatti, il dato di indirizzo simbolico `A` a cui fa riferimento, è allocato nella locazione 16 del segmento dati, ed è quindi caratterizzato da un indirizzo virtuale costituito dalla coppia (segmento 1, locazione 16).

Se lo spazio virtuale è segmentato, ognuno dei segmenti può essere rilocato in memoria fisica indipendentemente dagli altri. In particolare, se facciamo riferimento ad un sistema con rilocazione statica, il caricamento di un processo implicherà l'allocatione di tre porzioni di memoria nelle quali trasferire i tre segmenti, delegando al caricatore rilocatione il compito di rilocarli nelle corrispondenti porzioni. Ad esempio, in figura 4.9 nella parte c) a destra viene illustrata come risulterebbe l'immagine del processo in memoria se il segmento codice fosse rilocato a partire dall'indirizzo fisico 10240, il segmento di dati a partire dall'indirizzo fisico 34000 e il segmento di

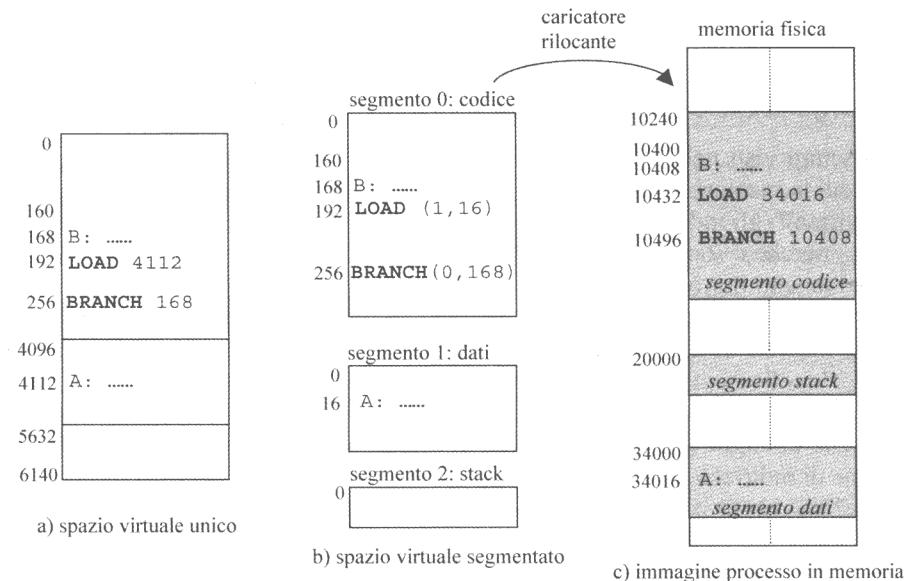


Figura 4.9 Spazio virtuale segmentato.

stack a partire dall'indirizzo fisico 20000. Il caricatore rilocante, mantenendo aggiornata una tabella con gli indirizzi iniziali di ciascun segmento, è in grado di rilocare tutti gli indirizzi virtuali. Ad esempio l'indirizzo virtuale dell'operando A dell'istruzione di LOAD, rappresentato dalla coppia (1, 16), verrebbe tradotto nell'indirizzo fisico 34016 ottenuto sommando all'indirizzo iniziale del segmento 1 (34000 nell'esempio) l'indirizzo della locazione A all'interno del segmento dati (16 nell'esempio).

Nel caso di rilocazione dinamica, la traduzione da indirizzo virtuale, espresso come coppia (segmento, locazione), nel corrispondente indirizzo fisico, avverrebbe durante l'esecuzione del processo all'atto di ogni accesso alla memoria tramite il meccanismo di MMU. Rispetto al caso di uno spazio virtuale unico, visto in figura 4.8, in questo caso la MMU dovrebbe contenere tre coppie di registri base/limite per rilocare dinamicamente i tre segmenti l'uno indipendentemente dagli altri. Per selezionare la specifica coppia di registri base/limite da utilizzare per tradurre un indirizzo virtuale x , è però necessario che la CPU generi gli indirizzi virtuali come coppia di interi (segmento, locazione) in modo tale da utilizzare il numero del segmento per selezionare l'opportuna coppia di registri. L'architettura di un elaboratore che adotta questo criterio viene appunto indicata con il termine di architettura con indirizzamento segmentato.

In questo caso il numero di segmenti può essere del tutto arbitrario, non più limitato a tre come nell'esempio precedente. In pratica ad ogni modulo di programma (funzione, struttura dati ecc) può essere associato un diverso segmento. La struttura della memoria virtuale viene quindi a corrispondere alla struttura di un programma così come viene concepita direttamente dal programmatore, in termini di segmenti

intesi come componenti logiche del programma, indipendentemente da quella che è la struttura fisica della memoria. Ciò permette di definire, in maniera più precisa e selettiva, un certo numero di parametri che caratterizzano i singoli segmenti visti come componenti logiche di un programma. In particolare i parametri relativi alla protezione e alla condivisione. Come vedremo nel seguito, è possibile associare selettivamente ad ogni segmento dei diritti di accesso che caratterizzano le modalità con cui è possibile accedere correttamente alle informazioni del segmento. Ad esempio, per un segmento di codice possiamo associare il solo diritto di lettura (non deve essere possibile modificare, erroneamente, un'istruzione). Viceversa, ad un segmento di dati possiamo associare i diritti in lettura e scrittura o, nel caso di un segmento di costanti, il solo diritto in lettura. Analogamente, possiamo caratterizzare ogni segmento, sia esso di codice o di dati, riguardo alla possibilità che lo stesso sia o meno condiviso fra più processi.

L'organizzazione di una memoria virtuale segmentata, rispetto al caso più semplice di memoria virtuale unica, implica sicuramente una maggiore complessità del linker e del meccanismo di rilocazione. Offre però tutti i vantaggi che sono stati indicati precedentemente. Inoltre, in generale, consente di migliorare l'efficienza di uso della memoria fisica come verrà indicato nel successivo paragrafo.

4.2.4 Allocazione della memoria fisica

Nei sistemi che adottano la rilocazione statica degli indirizzi lo spazio di memoria utilizzato per caricare un programma è costituito, come è stato mostrato negli esempi precedenti, da una porzione di memoria composta da un insieme di locazioni contigue e di dimensioni sufficienti a contenere lo spazio virtuale del processo. Nel caso in cui lo spazio virtuale sia segmentato vengono utilizzate più porzioni di memoria, ma ciascuna di esse è ancora costituita da un insieme di locazioni contigue (vedi figura 4.9). In altri termini alla contiguità di indirizzi virtuali corrisponde contiguità di indirizzi fisici.

Questa caratteristica costituisce un vincolo nella scelta dello spazio di memoria da assegnare a un processo, vincolo che spesso produce inefficienza nella gestione della memoria. Ad esempio, se dobbiamo allocare un intero spazio virtuale costituito da un certo numero N di Kbyte è necessario trovare un'area libera in memoria di dimensioni maggiori o uguali a N Kbyte contigui. Se per ipotesi fossero disponibili molte aree libere tutte di dimensioni minori di N Kbyte e non contigue in memoria, queste non sarebbero utilizzabili neppure se la somma delle loro dimensioni superasse il valore N . Questo fenomeno, che tende a ridurre l'utilizzazione della memoria, è noto col termine di *frammentazione* e si verifica a causa della necessità di allocare lo spazio fisico a indirizzi contigui.

Se lo spazio virtuale fosse segmentato, il fenomeno sarebbe ancora vero a livello del singolo segmento anche se, essendo le dimensioni di ciascun segmento inferiori alle dimensioni dell'intero spazio, ogni segmento avrebbe maggiori probabilità di trovare una partizione che lo contenesse, riducendo quindi gli effetti della frammentazione.

Nei sistemi con rilocazione dinamica il precedente fenomeno potrebbe essere eliminato potendo rilocare i processi in memoria. Infatti, se fosse richiesta un'area libera di N Kbyte contigui, ma la memoria risultasse frammentata come indicato in fi-

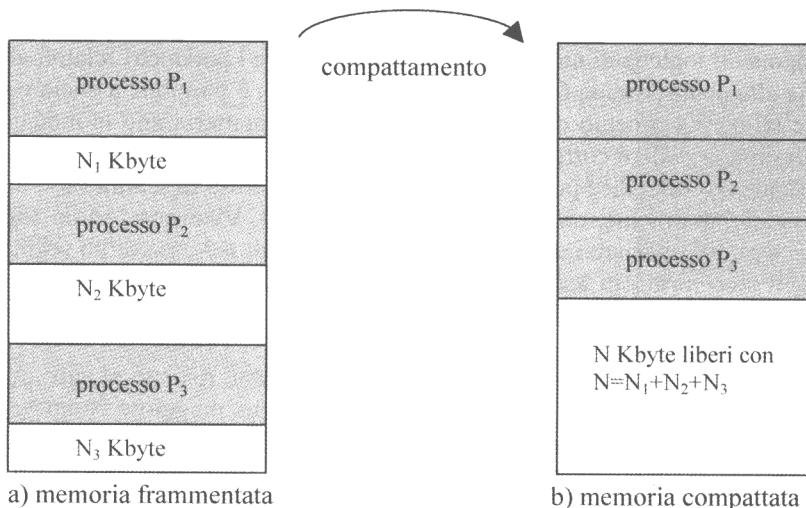


Figura 4.10 Compattamento.

gura 4.10-a), dove sono indicate aree occupate da processi ed aree libere ma ciascuna di queste di dimensioni inferiori a N , potremmo spostare tutti i processi in memoria compattando le aree libere in un'unica area di dimensioni pari alla somma delle aree precedenti come indicato in figura 4.10-b).

Questa tecnica, nota col termine di *compattamento*, elimina il fenomeno della frammentazione, ma il costo del compattamento è sicuramente molto elevato in termini di tempo di esecuzione richiesto al sistema operativo per ottenere un'area libera di dimensioni sufficienti a contenere il nuovo processo da caricare.

L'unica soluzione che consente di evitare l'inconveniente del compattamento è quella di permettere l'allocazione di uno spazio virtuale contiguo in locazioni di memoria fisica non necessariamente contigue. Infatti, se ciò fosse consentito, sarebbe possibile utilizzare tutte le aree di memoria disponibili, indipendentemente dalla loro distribuzione nella memoria.

Purtroppo, quando un programma viene rilocato staticamente mediante il caricatore, a indirizzi virtuali contigui devono necessariamente corrispondere indirizzi fisici contigui. Si pensi ad esempio a un segmento di codice nel quale due istruzioni del programma, una successiva all'altra, sono allocate nello spazio virtuale a due indirizzi contigui $iv1$ e $iv2 = iv1 + 1$. Se il caricatore trasferisse in memoria il segmento allocando la prima delle due precedenti istruzioni nella locazione di indirizzo fisico $if1$, allora sarà costretto ad allocare la seconda nella locazione di indirizzo fisico consecutivo ($if2 = if1 + 1$). Infatti, come è noto, durante l'esecuzione del programma, terminata l'esecuzione della prima delle due istruzioni, la CPU preleva la successiva istruzione da eseguire dalla locazione fisica di indirizzo successivo.

Questo vincolo non sussiste nel caso di rilocazione dinamica degli indirizzi. In-

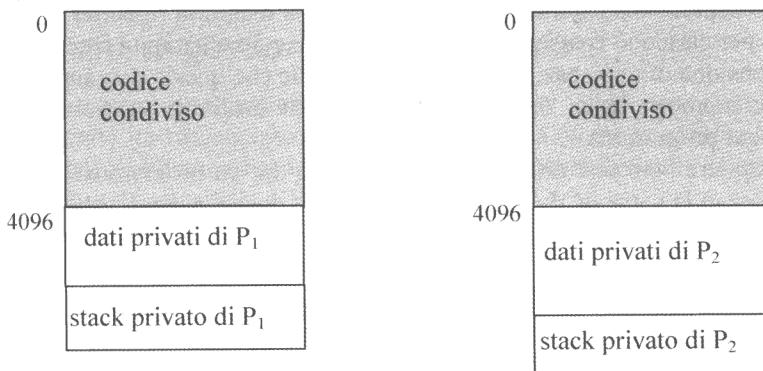


Figura 4.11 Codice condiviso.

fatti, con riferimento al precedente esempio, le due istruzioni, pur essendo consecutive nello spazio virtuale, potrebbero essere allocate, almeno in teoria, in locazioni fisiche non contigue. Ricordiamo che in questo caso, quando il programma è in esecuzione, nel registro PC è contenuto l'indirizzo virtuale dell'istruzione che viene correntemente eseguita. Quindi, terminata l'esecuzione della prima delle due precedenti istruzioni, la CPU preleva la successiva istruzione da eseguire dalla locazione di indirizzo virtuale successivo, ma ciò non implica necessariamente dalla locazione di memoria di indirizzo fisico successivo. Infatti, la disponibilità di un meccanismo di rilocazione dinamica (MMU) può essere sfruttata, complicando il meccanismo, in modo tale che la funzione di rilocazione $y = f(x)$, che per ogni indirizzo virtuale x fornisce l'indirizzo fisico y che gli corrisponde, permetta di associare indirizzi fisici non contigui a indirizzi virtuali contigui.

Questa caratteristica, nel caso di spazi virtuali non segmentati, è anche l'unica che consente a più processi di condividere parte del loro spazio. Infatti, supponiamo ad esempio che due processi P1 e P2 esegano lo stesso programma, ovviamente su dati diversi. In questo caso è opportuno caricare in memoria una sola copia del programma facendo condividere il codice ai due processi. Nella successiva figura 4.11 sono riportati, come esempio, gli spazi virtuali dei due processi P1 e P2. Come è indicato in figura, nei primi 4 Kbyte del loro spazio virtuale è contenuto il codice condiviso, mentre i dati e lo stack privati di ciascun processo sono allocati nella restante parte dei loro spazi.

Se gli spazi virtuali dovessero essere allocati in memoria fisica in aree contigue sarebbe chiaramente impossibile caricare, e quindi condividere, una sola copia del codice. Infatti, la porzione di memoria allocata a P1 conterrebbe nei primi 4 Kbyte il codice e nei successivi byte la parte privata di P1. Ma allora, per consentire a P2 di condividere il codice, anche il suo spazio virtuale dovrebbe essere allocato nella stessa porzione di memoria assegnata a P1, andando a sovrascrivere con la propria

4.2.5 Dimensionamento della memoria virtuale

Un ultimo aspetto che caratterizza la gestione della memoria riguarda le dimensioni ammesse per ciascuno spazio virtuale. Il caso più ovvio e naturale è quello di limitare la dimensione di uno spazio virtuale in modo tale che questo non superi le dimensioni della memoria fisica. Ovviamente ciò impone un limite superiore alle dimensioni di ogni programma.

Può capitare però che un processo debba eseguire un programma le cui dimensioni superano la capacità della memoria fisica. In questo caso il linker non riuscirà a creare un modulo di caricamento di dimensioni tali da poter essere allocato tutto insieme in memoria fisica. Per fornire una soluzione a questi casi eccezionali sono state concepite tecniche specifiche, come la tecnica dell'*overlay*, che prevedono un particolare tipo di linker, oltre che l'intervento da parte del programmatore il quale ha il compito di strutturare il suo programma in modo tale da non prevedere la presenza in memoria di tutte le sue componenti contemporaneamente. In pratica il programmatore deve specificare al linker quali componenti devono costituire il modulo iniziale di caricamento che verrà allocato in memoria alla creazione del processo. Successivamente, se previsto in base alla struttura del programma, quando è richiesta una componente non ancora caricata, viene richiamato il caricatore che, utilizzando una porzione dell'area di memoria già allocata al processo, sovrascrive alla parte di programma in essa caricata, e non più necessaria, la nuova componente che ovviamente il linker deve aver allocato nella stessa porzione di memoria virtuale della parte sovrascritta. Come si intuisce questa tecnica, utilizzata nei sistemi con rilocazione statica degli indirizzi, è relativamente complicata da usare ed è ormai caduta in disuso.

Nei sistemi che prevedono la rilocazione dinamica è possibile complicare leggermente il meccanismo di rilocazione (MMU) in modo tale da abilitare il linker a creare spazi virtuali di dimensioni superiori a quelle della memoria fisica. Ciò implica che uno spazio virtuale non potrà essere caricato in memoria tutto in una volta. La tecnica che viene utilizzata in questi casi è quella di caricare in memoria fisica le informazioni del programma (istruzioni o dati) soltanto quando la loro presenza in memoria è necessaria per l'esecuzione del processo. E cioè solo quando la CPU fa riferimento agli indirizzi virtuali di tali informazioni.

Per ottenere questo tipo di funzionamento il meccanismo MMU realizza la funzione di rilocazione $y = f(x)$ in modo tale che, dato un indirizzo virtuale x da tradurre, se l'istruzione o il dato di indirizzo virtuale x è già presente in memoria fisica la funzione restituisce l'indirizzo fisico y della corrispondente locazione di memoria. Se però tale informazione non è presente in memoria, viene generata un'interruzione che verrà gestita dal sistema operativo per caricare in memoria l'informazione richiesta (*caricamento a domanda*).

Con questa tecnica, complicando il meccanismo di rilocazione e, come vedremo, al costo di un maggiore *overhead* del sistema operativo, si ottengono numerosi vantaggi. Primo fra tutti, si svincolano le dimensioni di un programma da quelle della memoria fisica, liberando il programmatore da tutti i compiti previsti nel caso di uso delle tecniche di *overlay*. In pratica è il sistema operativo che si accolla l'onere di caricare le informazioni relative a un programma, un po' per volta e solo quando

della memoria fisica, considerate tutte quelle parti di un programma che durante l'esecuzione di un processo non sono necessarie perché costituiscono un ramo del programma che non viene eseguito da quel particolare processo (ad esempio il ramo `else` di uno statement `if` di cui viene eseguito il ramo `then`, oppure il modulo di programma per la gestione di una eccezione che non viene sollevata ecc.). Analogamente, parti di un programma che sono mutuamente esclusive nel tempo possono trovare spazio nelle stesse locazioni di memoria, riducendo la quantità di memoria complessivamente richiesta dal processo.

4.3 Tecniche di gestione della memoria

Come è stato mostrato nei precedenti paragrafi, quattro sono i principali parametri che caratterizzano un meccanismo per la gestione della memoria, parametri che dipendono sia dall'architettura hardware del processore che dalle scelte di progetto del sistema operativo (vedi tabella 4.1):

Rilocazione		Spazio virtuale		Allocazione della memoria fisica		Caricamento dello spazio virtuale	
Statica	Dinamica	Contigua	Non Contigua	Unico	Segmentato	Tutto insieme	A domanda

Tabella 4.1 Parametri caratterizzanti l'allocazione della memoria.

A ciascuna combinazione dei precedenti parametri corrisponde un diverso meccanismo e una diversa tecnica di gestione della memoria. In realtà alcune combinazioni di parametri non sono significative. Ad esempio, come è già stato detto precedentemente, nel caso di rilocazione statica l'allocazione della memoria fisica è sempre effettuata in maniera contigua e le dimensioni di uno spazio virtuale non possono superare quelle della memoria fisica (il caricamento di uno spazio virtuale avviene "tutto insieme").

Nel caso di sistemi con rilocazione dinamica e allocazione contigua di memoria lo spazio virtuale è sempre segmentato. Infatti, la scelta di realizzare uno spazio virtuale unico non permetterebbe nessuna forma di condivisione di informazioni tra processi diversi, come è già stato messo in evidenza nel paragrafo 4.2.4. Infine, nei sistemi più complessi che prevedono sia spazi virtuali segmentati sia tecniche di allocazione di memoria non contigua (*paginazione*), il caricamento è sempre "a domanda".

Nella successiva figura 4.12 viene riportato sia, in forma di albero, la gamma di tutte le varie tecniche oggi adottate in funzione dei diversi valori dei precedenti parametri.

Nei successivi paragrafi passeremo in rassegna le varie tecniche di gestione della memoria iniziando con quelle più semplici e passando quindi a quelle più complesse.

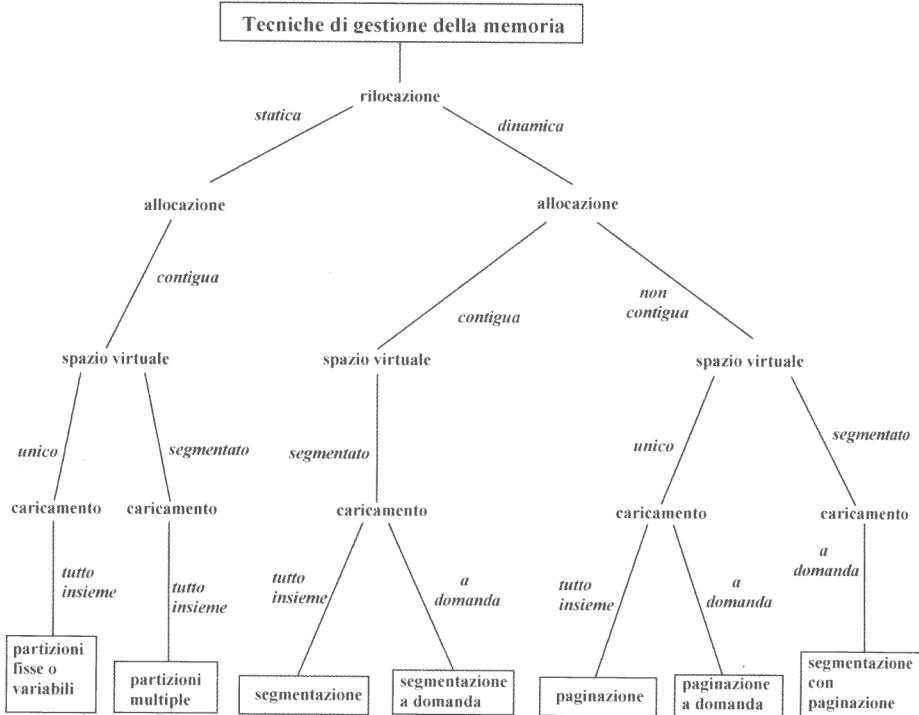


Figura 4.12 Tecniche di gestione della memoria.

4.3.1 Memoria partizionata

Seguendo lo schema riportato in figura 4.12, iniziamo ad esaminare le tecniche corrispondenti al sottoalbero di sinistra relative ai sistemi che, non disponendo di particolari meccanismi hardware per la gestione della memoria (MMU), adottano lo schema di rilocazione statica degli indirizzi effettuata mediante il cariatore rilocante.

Partizioni fisse e variabili In questo ambito iniziamo a vedere le tecniche di allocazione utilizzate in sistemi nei quali la memoria virtuale è costituita da un unico spazio virtuale contiguo, corrispondenti a quelle indicate all'estrema sinistra in figura 4.12, e caratterizzate dai seguenti parametri:

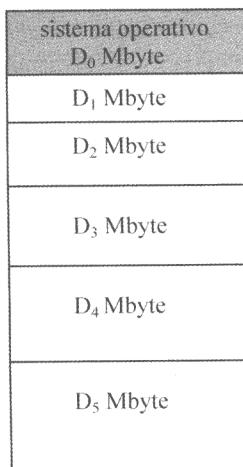
Rilocazione	Allocazione della memoria fisica	Spazio virtuale	Caricamento dello spazio virtuale
Statica	Contigua	Unico	Tutto insieme

In questi casi quando è necessario allocare memoria ad un processo (ad esempio quando questo viene creato), dovendo caricare tutto lo spazio virtuale del processo e

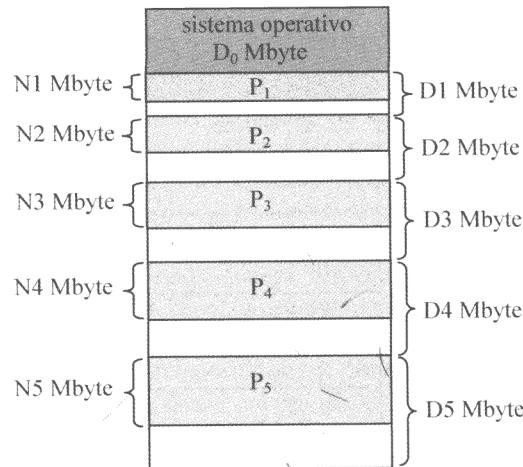
caricarlo in un'unica area di locazioni contigue, è necessario cercare una porzione di memoria (o *partizione*, come viene più spesso chiamata) che sia libera, cioè che non contenga informazioni di altri processi, e le cui dimensioni siano maggiori o uguali a quelle dello spazio virtuale da caricare. Una volta individuata la partizione, il caricatore vi rilocherà lo spazio virtuale del processo, con il vincolo che tale partizione rimarrà assegnata al processo fino ma quando lo stesso non sarà terminato, oppure se il processo subirà uno *swap_out* sarà necessario ricaricarlo nella stessa partizione all'atto del successivo *swap_in*.

Per risolvere questo problema sono stati adottati, nel tempo, due diversi schemi.

Il primo, noto col nome di *schema a partizioni fisse*, consiste nel suddividere la memoria in $(n + 1)$ partizioni di indirizzo iniziale e dimensioni fisse, definite in fase di installazione del sistema operativo. La prima di queste è destinata a contenere la parte residente del sistema stesso. Le successive n ulteriori partizioni sono riservate per ospitarvi le immagini di altrettanti processi una volta rilocati i rispettivi spazi virtuali al loro interno. In figura 4.13-a) viene illustrato un esempio in cui la memoria fisica è suddivisa in 6 partizioni, la prima di indirizzo iniziale 0 e dimensioni D_0 è riservata al sistema operativo. Le successive 5 partizioni, usate per allocare processi, hanno rispettivamente dimensioni D_1, D_2, D_3, D_4 e D_5 . Normalmente un processo viene allocato in una partizione che, fra tutte quelle disponibili, è quella di dimensioni più piccole che lo possa contenere. È però estremamente raro il caso in cui le dimensioni dello spazio virtuale coincidano esattamente con quelle della partizione utilizzata per contenerlo. Ne consegue che la differenza tra le dimensioni della partizione e quelle dello spazio virtuale in essa caricato corrisponde a una zona di memoria non utilizzata. La somma di tutte queste zone può costituire una porzione considerevole di memoria non utilizzata. Questo inconveniente porta il nome di *frammentazione interna* (l'aggettivo interna si riferisce alla partizione in quanto porzione di questa



a) partizioni libere



b) frammentazione interna

non utilizzata). Nella stessa figura 4.13, parte b), viene indicato il fenomeno della frammentazione che corrisponde alla somma delle differenze tra le dimensioni delle singole partizioni e quelle degli spazi virtuali dei processi in esse allocati. Nella figura questa quantità corrisponde a:

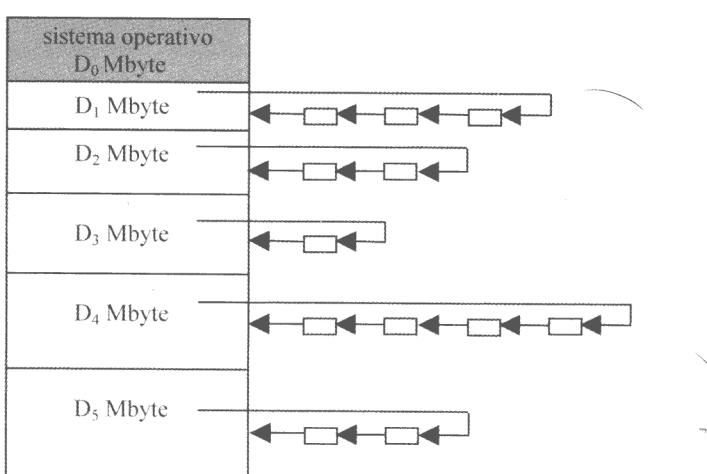
$$(D_1 - N_1) + (D_2 - N_2) + (D_3 - N_3) + (D_4 - N_4) + (D_5 - N_5)$$

dove N_i rappresenta la dimensione dello spazio virtuale di ciascun processo P_i .

Pur adottando il criterio di rilocazione statica degli indirizzi, la tecnica delle partizioni fisse si presta molto bene ad allocare dinamicamente la memoria ai processi, revocando loro memoria mediante la primitiva `swap_out` e successivamente riallocando il processo nella stessa partizione mediante la primitiva `swap_in`. Per questo motivo il sistema operativo, per ogni partizione, mantiene ordinata una coda di descrittori, corrispondenti ai processi destinati a quella partizione. Di questi, il primo processo nella coda è fisicamente allocato in memoria mentre tutti gli altri sono residenti nella *swap-area*. Periodicamente, secondo una politica *round robin*, la memoria viene revocata al primo processo di ogni partizione, il cui descrittore viene concatenato in fondo alla rispettiva coda e nella partizione viene allocato il processo successivo in coda (vedi figura 4.14).

Lo schema di allocazione mediante partizioni fisse, pur essendo estremamente semplice e richiedendo quindi un basso *overhead* di sistema, è comunque molto inefficiente per quanto riguarda l'uso della memoria. Infatti, la scelta di definire il numero e le dimensioni delle partizioni una volta per tutte in fase di installazione del sistema implica una totale mancanza di flessibilità. Inoltre il problema della frammentazione interna riduce drasticamente la possibilità di ottimizzare l'uso della memoria.

Per questo motivo è stato proposto un diverso schema per suddividere la memoria in partizioni, schema noto col nome di *partizioni variabili*, che consente di defi-



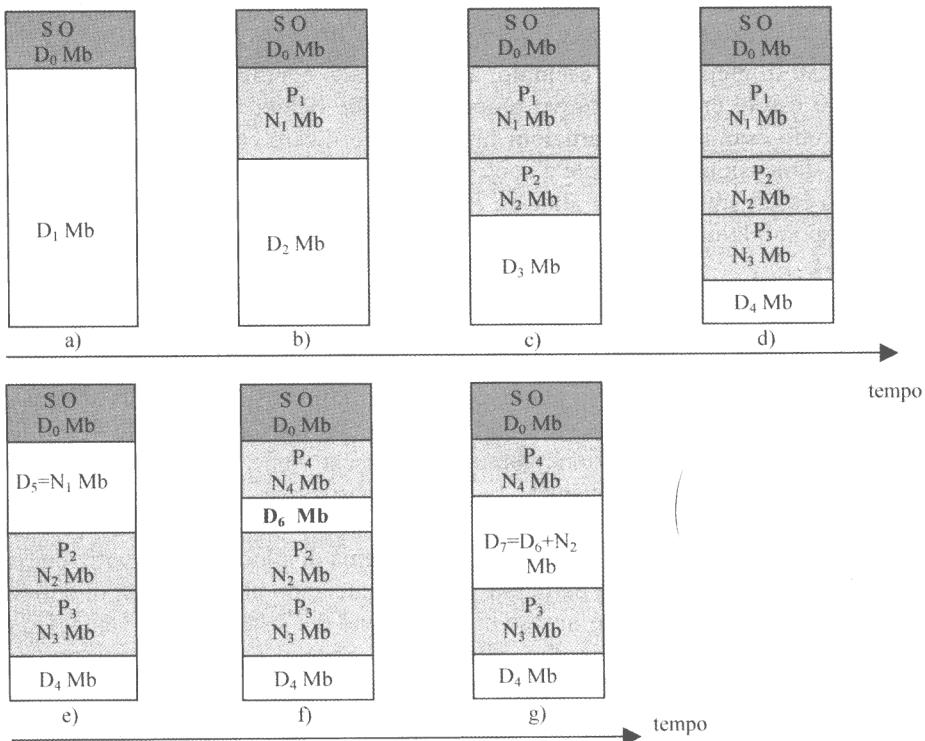


Figura 4.15 Partizioni variabili.

nire dinamicamente le caratteristiche delle singole partizioni in modo tale che queste corrispondano esattamente alle esigenze di memoria dei processi in esse caricati. Per chiarire questo schema facciamo riferimento all'esempio illustrato in figura 4.15.

Inizialmente tutta la memoria costituisce un'unica partizione libera di dimensioni D_1 Mbyte, dove D_1 coincide con la dimensione dell'intera memoria meno l'area D_0 occupata dal sistema operativo (vedi figura 4.15-a). Quando viene creato il primo processo P_1 , il cui spazio virtuale è costituito da N_1 locazioni, una partizione di dimensioni esattamente uguali a N_1 viene allocata al processo prelevandola dall'unica partizione disponibile. Dopo l'allocatione resta ancora disponibile una sola partizione di dimensioni $D_2 = D_1 - N_1$ (figura 4.15-b). Successivamente altri processi vengono creati, ad esempio P_2 che richiede N_2 Mbyte e P_3 che richiede N_3 Mbyte. Ad essi vengono quindi allocate le rispettive partizioni di dimensioni esattamente uguali a N_2 ed N_3 (figure 4.15-c e 4.15-d). Resta ancora disponibile una sola partizione che corrisponde a quella iniziale da cui sono state prelevate le aree necessarie a creare i tre precedenti processi e quindi una partizione di dimensione $D_4 = D_1 - N_1 - N_2 - N_3$ Mbyte. Se ora termina il processo P_1 rilasciando l'area da esso occupata, il sistema dovrà registrare che da ora in poi sono disponibili due

$D_5 = N_1$ Mbyte (figura 4.15-e). A questo punto altri processi possono terminare e altri possono essere creati, generando quindi uno scenario in cui la memoria viene dinamicamente suddivisa in partizioni libere intervallate da aree occupate dai processi allocati in memoria. Ad esempio, a partire dalla condizione indicata in figura 4.15-e può essere creato un nuovo processo P_4 che richiede N_4 Mbyte. In questo caso vengono esaminate le partizioni libere per verificare che almeno una di esse abbia una dimensione sufficiente a contenerlo. Per esempio se $D_5 > N_4$, la partizione D_5 può essere utilizzata per allocare P_4 . Di nuovo, dopo questa allocazione resta ancora disponibile una partizione di dimensione $D_6 = D_5 - N_4$ come indicato in figura 4.15-f.

Non sempre quando un processo termina rilasciando la partizione da esso occupata si viene necessariamente a creare una nuova partizione libera. Infatti se l'area rilasciata è adiacente (superiormente e/o inferiormente) ad altre partizioni libere è necessario compattarle in una sola nuova partizione di dimensioni pari alla somma di quelle compattate. Ad esempio se, a partire dallo stato indicato in figura 4.15-f termina il processo P_2 , la partizione che questo rilascia viene compattata con quella D_6 ad essa adiacente generando la nuova partizione libera di dimensioni $D_7 = D_6 + N_2$ (vedi figura 4.15-g).

Come illustrato nel precedente esempio, lo schema di allocazione a partizioni variabili, rispetto al precedente a partizioni fisse, rinunciando alla necessità di definire le partizioni una volta per tutte in fase di installazione del sistema, ha il vantaggio di essere molto più flessibile e inoltre, allocando a ciascun processo esattamente la quantità di memoria richiesta, elimina il problema della frammentazione interna. Peraltro, come già indicato nel paragrafo 4.2.4, dovendo ancora allocare memoria fisica in locazioni contigue, soffre di un'altra forma di frammentazione, nota come frammentazione esterna, quando le singole partizioni libere sono ciascuna di dimensione inferiore alla quantità di memoria richiesta per un nuovo processo anche se la somma delle loro dimensioni è maggiore della richiesta. Infatti, la rilocazione statica non consente di compattare le partizioni libere e ciò contribuisce ancora ad uno scarso livello di utilizzazione della memoria.

Inoltre questo schema prevede che il gestore della memoria mantenga aggiornata una struttura dati con la quale registrare, istante per istante, quante e quali sono le partizioni disponibili. Una possibile soluzione a questo problema, che viene spesso utilizzata, è quella di mantenere aggiornata una lista i cui elementi rappresentano le caratteristiche (indirizzo iniziale e dimensione) di ogni partizione libera. Per questo motivo si riserva nella memoria del sistema una locazione (indichiamola col nome di memoria libera) contenente l'indirizzo di una partizione libera. Le prime due locazioni della partizione contengono, rispettivamente, la dimensione in byte della partizione e l'indirizzo di una successiva partizione, e così via per tutte le partizioni. In questo modo la variabile memoria libera e gli elementi costituiti dalle prime due locazioni di ogni partizione libera costituiscono una lista (vedi figura 4.16) che concatena tutte le partizioni disponibili. L'esempio riportato in figura è quello relativo al caso in cui siano disponibili tre partizioni, la prima di indirizzo iniziale I_1 e dimensioni D_1 e le altre due di indirizzo e dimensioni I_2, D_2 e I_3, D_3 rispettivamente.

Normalmente, per semplificare sia la fase di richiesta di una partizione libera che



Figura 4.16 Lista delle partizioni libere.

pio potrebbe essere mantenuta ordinata per dimensioni crescenti delle partizioni. In questo caso, noto come schema *best-fit*, in fase di richiesta di una partizione di almeno N byte la lista viene scandita e la prima partizione che viene trovata in grado di soddisfare la richiesta è sicuramente quella più piccola tra tutte quelle di dimensioni superiori a N . Questo schema soffre di due inconvenienti. Il primo è legato al fatto che, una volta effettuata l'allocazione, la parte della partizione non utilizzata, e che quindi resta libera, è sicuramente quella di dimensioni più piccole e ciò, come ben si intuisce, tende a far crescere la frammentazione della memoria. Il secondo inconveniente è che in fase di rilascio è necessario verificare se la partizione resa disponibile è adiacente ad una o a due partizioni libere per, eventualmente, compattarle insieme. Ebbene, questa verifica implica una scansione dell'intera lista per controllare tali possibili adiacenze.

Un secondo schema, più spesso usato e noto col nome di *first-fit*, consiste nel mantenere la lista ordinata per indirizzi crescenti delle partizioni. Questo schema risulta particolarmente efficiente in fase di rilascio. Infatti, una volta trovata nella lista la posizione corrispondente alla partizione liberata, le eventuali partizioni adiacenti inferiormente e/o superiormente, se esistono, non possono che essere quelle caratterizzate dagli elementi della lista precedente e successivo a quello relativo alla partizione rilasciata.

Per completare l'argomento è opportuno accennare ai due aspetti relativi alla protezione e alla condivisione di informazioni. Il primo aspetto riguarda il problema di garantire la protezione fra processi contemporaneamente allocati in memoria. Questo problema viene spesso risolto a livello di architettura del processore. Ad esempio riservando due particolari registri, noti col nome di *registri limite* o *registri frontiera*, contenenti rispettivamente l'indirizzo iniziale e quello finale della partizione assegnata al processo in esecuzione. Ogni indirizzo generato dal processore viene confrontato con i valori di tali registri e, nel caso in cui l'indirizzo sia esterno alla partizione assegnata, viene generata un'interruzione di errore.

Viceversa, per quanto riguarda la condivisione di informazioni, come è già stato indicato nel paragrafo 4.2.4, il fatto di allocare in locazioni contigue uno spazio virtuale unico non consente a processi diversi nessuna forma di condivisione.

Partizioni multiple È proprio per consentire a due o più processi di condividere il codice ed eseguire quindi lo stesso programma, anche se su dati diversi, che è stata introdotta la possibilità di segmentare lo spazio virtuale, abilitando il linker a suddividerlo, ad esempio, nei tre segmenti di codice, dati e stack come visto in figura 4.9. In questo modo, ogni segmento pur essendo ancora singolarmente allocato in locazioni contigue, e cioè in una partizione di memoria, può essere rilocato in maniera completamente indipendente dalle partizioni usate per allocare gli altri segmenti.

Questo schema, noto col nome di *partizioni multiple*, coincide col precedente per quanto riguarda l'allocazione di una generica partizione, ma la creazione di un processo implica adesso la ricerca non di una sola grande partizione ma di tre più piccole, e non necessariamente contigue fra loro. I parametri di questo schema sono dunque:

Rilocazione	Allocazione della memoria fisica	Spazio virtuale	Caricamento dello spazio virtuale
Statica	Contigua	Segmentato	Tutto insieme

La maggiore complessità richiesta al linker è ampiamente controbilanciata dal fatto che il segmento di codice non essendo più allocato in locazioni contigue al segmento dati, può essere facilmente condiviso tra più processi. Inoltre, avendo spezzettato lo spazio virtuale, le singole partizioni necessarie per allocare un processo sono di dimensioni più piccole di quella che dovremmo trovare se lo spazio fosse unico. Ciò, pur non eliminando il fenomeno della frammentazione esterna, tende a ridurne gli effetti negativi.

4.3.2 Memoria segmentata

Come indicato nel paragrafo 4.2.4, per ridurre il fenomeno della frammentazione esterna sarebbe necessario compattare le varie partizioni libere in un'unica partizione di dimensione uguale alla somma delle loro dimensioni. Ma per spostare in memoria un processo è necessario ricorrere ad una tecnica di rilocazione dinamica, ad esempio facendo ricorso ad un meccanismo di MMU che contiene i registri base e limite come è stato mostrato in figura 4.8. Se però, come indicato nel precedente paragrafo, lo spazio virtuale è segmentato, per consentire a più processi di condividere almeno il segmento di codice, ogni segmento deve essere dinamicamente rilocato in modo indipendente dagli altri. La MMU deve quindi contenere non una coppia, bensì tre coppie di registri base/limite, una coppia per ogni segmento componente lo spazio virtuale e utilizzata per tradurre ogni indirizzo virtuale appartenente a quel segmento nell'indirizzo fisico corrispondente. Ad esempio, in figura 4.17 viene illustrato il caso di un processo i cui segmenti di codice, dati e stack sono stati allocati nelle tre partizioni caratterizzate, rispettivamente, dai seguenti indirizzi fisici iniziali e dimensioni: I_1 ed D_1 per il codice, I_2 e D_2 per i dati, I_3 e D_3 per lo stack. Nella MMU sono contenute le tre coppie di registri base/limite che, quando il processo è in esecuzione, contengono i valori corrispondenti alle tre partizioni. Nella figura, in particolare, viene illustrato il caso in cui la CPU ha generato l'indirizzo virtuale x appartenente al segmento di codice, ad esempio l'indirizzo virtuale di un'istruzione generato in fase di prelievo (*fetch*). La funzione di rilocazione $y = f(x)$ viene quindi realizzata, in questo caso, utilizzando la prima coppia di registri base/limite, quella corrispondente al segmento di codice.

Per garantire la corretta rilocazione degli indirizzi, per ogni indirizzo virtuale x generato dalla CPU, deve essere possibile capire a quale segmento esso appartiene in modo tale che possa essere selezionata nella MMU la corretta coppia di registri base/limite per tradurre x nell'indirizzo fisico y che gli corrisponde. Nei casi in cui

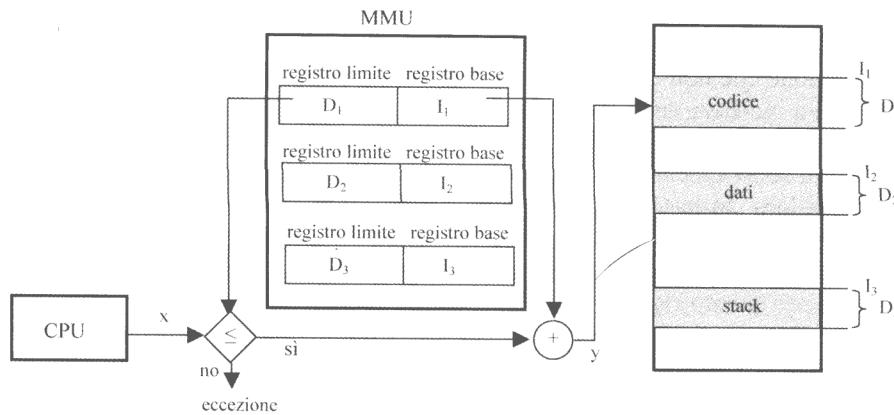


Figura 4.17 Tecnica della segmentazione.

lo spazio virtuale è costituito dai soli segmenti di codice, dati e stack, l'individuazione del segmento a cui un generico indirizzo virtuale appartiene può essere fatta in modo molto semplice e automatico. Ad esempio, tutti gli indirizzi generati in fase di prelievo di una istruzione appartengono al segmento di codice in quanto indirizzi di istruzioni. Analogamente appartengono a questo segmento tutti gli indirizzi generati durante le fasi di esecuzione di istruzioni di salto e/o di salto a sottoprogramma. Viceversa, appartengono al segmento di stack tutti gli indirizzi generati durante le fasi di esecuzione delle istruzioni push e pop di accesso allo stack. Infine, appartengono al segmento dati tutti gli indirizzi generati durante le fasi di esecuzione delle altre istruzioni.

Questa tecnica di allocazione della memoria è nota con il nome di *segmentazione* e corrisponde esattamente a quella precedentemente illustrata delle partizioni multiple con la variante di aver introdotto un meccanismo di rilocazione dinamica. Per questo motivo è stata anche chiamata *tecnica delle partizioni rilocabili* ed è caratterizzata dai seguenti parametri:

Rilocazione	Allocazione della memoria fisica	Spazio virtuale	Caricamento dello spazio virtuale
Dinamica	Contigua	Segmentato	Tutto insieme

L'introduzione della MMU per rilocare dinamicamente gli indirizzi consente di ottenere una maggiore flessibilità ed efficienza nell'allocazione della memoria rispetto alla precedente tecnica delle partizioni multiple. Infatti, pur continuando ad allocare la memoria fisica per partizioni, e quindi generando il fenomeno della frammentazione, la possibilità di rilocare i processi in memoria consente di compattare le partizioni libere quando ciò risulta necessario, riducendo gli effetti negativi della frammentazione. Inoltre, è adesso molto più semplice revocare memoria ad un processo mediante la primitiva di swap_out tenendo conto che all'atto della riallocazione

mediante la primitiva `swap_in` non è più necessario riallocare il processo nelle stesse partizioni occupate prima della revoca.

Nel paragrafo 4.2.3, lo spazio virtuale segmentato è stato introdotto in base alla considerazione che non c'è nessuna ragione che obblighi il linker ad allocare il segmento dati a indirizzi virtuali contigui a quelli in cui vengono allocate le istruzioni, e analogamente non è necessario allocare il segmento di stack a indirizzi contigui a quelli in cui sono allocati i dati o il codice, essendo i tre segmenti del tutto indipendenti tra di loro. In base alle stesse considerazioni, però, non c'è nessuna ragione neppure per allocare le istruzioni componenti una funzione in locazioni contigue a quelle in cui vengono allocate le istruzioni di un'altra funzione e la stessa considerazione vale per quanto riguarda due diverse strutture dati. In altri termini uno spazio virtuale segmentato non è vincolato ad essere strutturato nei tre soli segmenti a cui abbiamo fatto riferimento fino ad ora. Anzi, negli attuali sistemi segmentati il linker crea lo spazio virtuale riservando un diverso segmento per ciascun modulo di programma (procedure e funzioni, strutture dati ecc.). In questo modo lo spazio virtuale viene ad assumere una struttura che ricalca fedelmente la struttura del programma che in esso è stato allocato, così come viene percepito dal programmatore. A differenza della memoria fisica, che è costituita da una piatta sequenza di locazioni, la memoria virtuale è costituita da un insieme di moduli semanticamente significativi, i segmenti, ciascuno costituito da una sequenza di locazioni di dimensioni diverse rispetto a quelle di altri segmenti. In ogni segmento il linker alloca uno dei moduli componenti il programma.

Di seguito vengono analizzate, per prima cosa, le conseguenze che questa organizzazione crea a livello architettonale e, successivamente, vengono messi in evidenza i vantaggi di questo tipo di struttura della memoria virtuale, soprattutto per quanto riguarda la protezione e la condivisione.

Il passaggio da tre soli segmenti ad un numero superiore crea un primo problema: l'impossibilità di determinare in forma automatica il segmento a cui appartiene un indirizzo virtuale x generato dalla CPU. Ad esempio, se x è generato in fase di prelievo di un'istruzione evidentemente x apparterrà ad un segmento di codice, ma nel momento in cui sono presenti molti segmenti di codice non è possibile individuare automaticamente a quale di essi appartiene. Per questo motivo le architetture dei sistemi segmentati prevedono che la CPU generi indirizzi virtuali che indicano esplicitamente a quale segmento l'indirizzo appartiene. Ogni indirizzo virtuale x è quindi costituito da una coppia di interi:

$$x = \langle sg, of \rangle$$

dove sg denota il numero del segmento a cui x appartiene ed of indica lo scostamento di x all'interno del segmento (*offset*).

Inoltre, quando il numero di segmenti aumenta non è più possibile mantenere nei registri della MMU i due valori corrispondenti all'indirizzo di base e al limite per tutti i segmenti. Ad esempio nel microprocessore Intel 386 uno spazio virtuale può contenere fino a 2^{14} diversi segmenti. Non è quindi possibile mantenere un numero così elevato di coppie di registri all'interno della MMU. Per questo motivo i due valori corrispondenti all'indirizzo base e al valore limite di ogni segmento (spesso indicati con il termine di *descrittore del segmento*) sono mantenuti dal sistema operati-

vo in una tabella, chiamata *tabella dei segmenti*, allocata nella memoria fisica del sistema. La memoria virtuale di un processo è quindi caratterizzata dai vari segmenti che la compongono e, quando lo stesso è allocato in memoria fisica, dall'insieme delle partizioni nelle quali i segmenti sono stati caricati e i cui valori sono registrati nella sua tabella dei segmenti.

Il descrittore di un processo, così come contiene lo stato del proprio processore virtuale (il campo contesto visto nel capitolo 2), contiene anche un campo relativo alla sua memoria virtuale. In tale campo del descrittore vengono mantenute aggiornate due informazioni: il numero di segmenti che compongono la memoria virtuale e l'indirizzo in memoria della tabella dei segmenti.

I due precedenti valori contenuti nel descrittore di un processo, quando lo stesso viene schedulato e passa in esecuzione, vengono caricati in due particolari registri di macchina. Il primo, spesso indicato con l'acronimo *STLR* (*Segment Table Limit Register*), è destinato a contenere il numero di segmenti del processo, e cioè il numero di elementi della sua tabella dei segmenti. Il secondo, spesso indicato con l'acronimo *STBR* (*Segment Table Base Register*), è destinato a contenere l'indirizzo fisico della tabella dei segmenti. Tale tabella viene utilizzata per tradurre tutti gli indirizzi virtuali generati dalla CPU durante l'esecuzione del processo. Ad esempio, in figura 4.18 viene illustrato come viene tradotto l'indirizzo virtuale x generato dalla CPU nel caso in cui x sia relativo alla locazione di scostamento of nel segmento di indice sg .

Per prima cosa il valore sg del segmento viene confrontato col contenuto del registro *STLR* che contiene il numero n di segmenti appartenenti al processo. Se sg fosse superiore a n verrebbe generata una *trap* per indicare il tentativo di accedere ad un segmento inesistente. In caso contrario sg viene utilizzato per accedere alla tabella dei segmenti e trovare l'elemento contenente i valori relativi all'indirizzo iniziale I e alla dimensione D della partizione di memoria nella quale il segmento sg è stato caricato. Se lo scostamento of è inferiore o uguale alla dimensione D del segmento, il suo indirizzo di base I viene sommato allo scostamento of al fine di ricavare l'indirizzo fisico y che viene quindi inviato alla memoria per accedere all'informazione desiderata. Di nuovo, se of fosse superiore alla dimensione D del segmento verrebbe generata una *trap* per indicare il tentativo di accedere a un'informazione non appartenente al segmento.

Il ricorso alla tabella dei segmenti allocata in memoria principale genera una notevole perdita di efficienza della CPU rispetto al caso in cui i valori base e limite di ogni segmento sono contenuti in registri veloci all'interno della MMU. Infatti, per ogni indirizzo generato dalla CPU è ora necessario raddoppiare gli accessi alla memoria: uno per accedere alla tabella dei segmenti per tradurre l'indirizzo da virtuale a fisico, un secondo per accedere all'informazione desiderata una volta ottenuto l'indirizzo fisico. Al fine di ridurre questa perdita di efficienza, vengono mantenuti nella MMU alcuni registri associativi (normalmente da 4 a 8). In ognuno di tali registri viene memorizzato un numero di segmento e i corrispondenti valori dei registri base/limite. In pratica, se nella MMU sono presenti 8 registri associativi, in essi sono contenuti i valori base/limite degli ultimi 8 segmenti a cui è stato fatto riferimento. In questo modo, quando la CPU genera un indirizzo $x = <sg, of>$, la sua traduzione avviene per prima cosa accedendo alla memoria associativa (*Translation Look-*

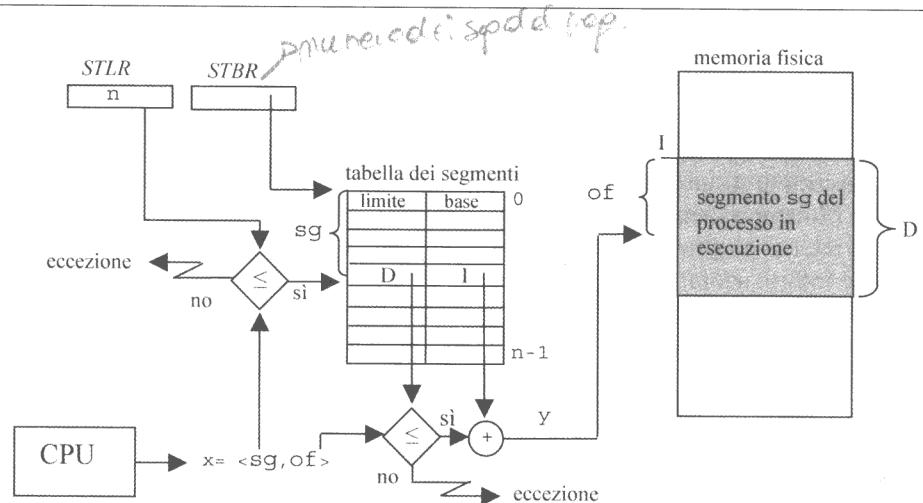


Figura 4.18 Traduzione degli indirizzi segmentati.

mento sg sono già presenti in uno dei registri. Se è così, la traduzione procede prelevando i valori base e limite dal registro associativo. Se viceversa, le informazioni relative a sg non sono presenti nella MMU la traduzione procede come indicato nella precedente figura 4.18 e, alla fine della traduzione, i valori base/limite di sg prelevati dalla tabella dei segmenti vengono anche caricati in uno dei registri associativi della MMU in modo tale che i prossimi riferimenti allo stesso segmento possano essere tradotti senza dover accedere alla tabella in memoria. Tenendo conto che gli accessi alla memoria sono spesso localizzati (per esempio all'interno di una funzione), con i pochi registri associativi presenti nella MMU, oltre l'80% degli indirizzi può essere tradotto senza accedere alla tabella dei segmenti.

La maggiore complicazione del meccanismo di traduzione degli indirizzi rispetto alle tecniche precedenti, trova la sua giustificazione nei vantaggi offerti dalla segmentazione, vantaggi correlati alla struttura della memoria virtuale che, come già indicato precedentemente, coincide con la struttura logica del programma. In base a tale struttura il programmatore può definire, per ogni segmento, alcuni parametri che sono correlati alla semantica del modulo di programma rappresentato dal segmento stesso. Due parametri particolarmente significativi riguardano la protezione e la condivisione di ogni segmento.

Per quanto riguarda la protezione, la segmentazione consente di effettuare tre diversi controlli nel momento in cui viene fatto l'accesso ad una locazione di memoria. I primi due controlli sono già stati messi in evidenza in figura 4.18, e corrispondono alla verifica che l'indirizzo virtuale generato dalla CPU appartenga allo spazio del processo. In particolare viene effettuato il controllo che il segmento sg a cui x appartiene sia uno dei segmenti del processo verificando che il suo valore sia minore o uguale al contenuto del registro $STLR$. L'altro controllo verifica che l'offset of sia inferiore o uguale alla dimensione del segmento. In entrambe i casi, se il controllo

base	limite	controllo
indirizzo della partizione	dimensione della partizione	R W

Figura 4.19 Descrittore di segmento.

do controllo tale interruzione non corrisponde necessariamente a un errore. Per esempio, può essere sfruttata dal sistema operativo per realizzare strutture dati variabili e essere interpretata come la necessità di allocare al segmento una partizione di maggiori dimensioni poiché il segmento in essa contenuto ha necessità di espandersi.

Oltre ai precedenti controlli, è inoltre possibile associare ad ogni segmento degli specifici diritti di accesso in modo tale che ogni riferimento a quel segmento sia consistente con tali diritti. Per esempio, se un segmento contiene del codice è ovvio che le sue locazioni non devono essere modificate e possiamo quindi definire per esso il solo diritto di accesso in lettura. La stessa cosa vale per un segmento che contiene soltanto delle costanti. Viceversa, un segmento di dati può essere abilitato agli accessi sia in lettura che in scrittura. In questo modo, sfruttando il significato di ogni segmento, è possibile realizzare un controllo molto più fine sulla correttezza degli accessi che un processo effettua alle informazioni contenute nel proprio spazio virtuale.

Per consentire tali ulteriori controlli, il generico elemento della tabella dei segmenti di un processo contiene, oltre all'indirizzo base e alla dimensione del segmento, anche un terzo campo di controllo (vedi figura 4.19) nel quale sono presenti alcuni bit che rappresentano i diritti di accesso al segmento. Ad esempio, i bit di accesso in lettura e in scrittura (bit R e W della figura 4.19). Soltanto se un bit ha il valore uno la corrispondente operazione di lettura o di scrittura viene abilitata mentre in caso contrario viene generata un'interruzione di errore.

Un secondo parametro che può essere specificato per ciascun segmento riguarda la condivisione. Di nuovo, essendo i singoli segmenti corrispondenti a specifici moduli di programma, è possibile abilitare processi diversi a condividere uno o più segmenti allocando in memoria fisica una sola copia degli stessi. Ad esempio nella figura 4.20 viene indicato il caso in cui due processi P1 e P2 condividono un segmento contenente il codice della funzione fun.

Nella figura sono rappresentati i due spazi virtuali dei processi P1 e P2 mediante le rispettive tabelle dei segmenti. Ciascun processo è caratterizzato da 4 segmenti: il proprio main program la funzione fun condivisa e i propri segmenti di dati e di stack. La condivisione della funzione viene evidenziata dal fatto che il descrittore del segmento di indice 1 nelle due tabelle contiene i dati relativi alla stessa partizione di memoria nella quale la funzione è stata caricata.

Relativamente alla condivisione è necessario mettere in evidenza un vincolo a cui è necessario prestare attenzione. Qualora le informazioni condivise contengano degli indirizzi di memoria, essendo questi degli indirizzi virtuali, ciò implica che le informazioni a cui tali indirizzi fanno riferimento devono essere allocate in posizioni

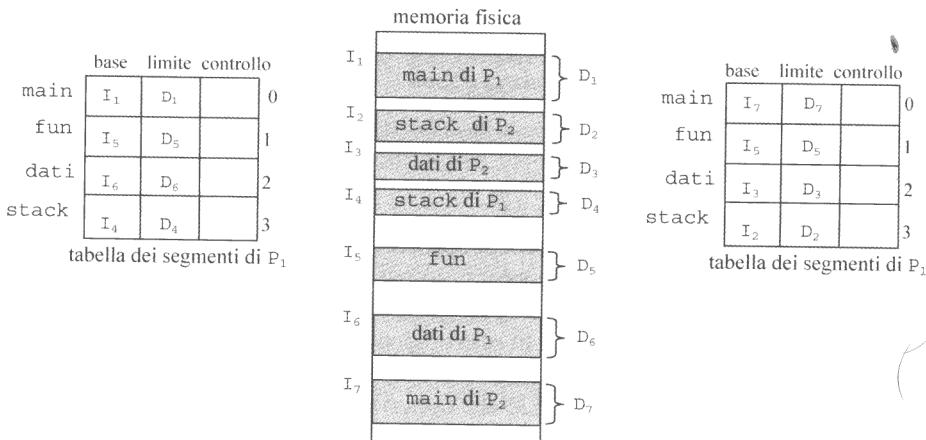


Figura 4.20 Condivisione di segmenti.

zione `fun` contenesse al proprio interno un riferimento ad una propria istruzione (ad esempio in una istruzione di salto) il numero di segmento assegnato a `fun` in uno spazio virtuale deve coincidere con lo stesso numero assegnato alla funzione anche nell'altro spazio (il comune segmento 1 nel precedente esempio).

Segmentazione a domanda Con la tecnica della segmentazione, per quanto riguarda l'allocazione della memoria fisica, un processo può essere in due diverse condizioni: o è allocato in memoria, e allora tutti i segmenti che compongono il suo spazio virtuale sono residenti in altrettante partizioni, oppure non è allocato in memoria, e allora tutti i suoi segmenti sono contenuti su memoria di massa in opportune porzioni della *swap-area*. Per questo motivo è necessario aumentare il numero degli stati in cui un processo può trovarsi. Oltre ai tre stati illustrati nella figura 2.3 (stati di *esecuzione*, *pronto* e *bloccato*), o ai cinque stati indicati in figura 2.4, con le relative transizioni già esaminate nel secondo capitolo, è necessario aggiungere due ulteriori stati che corrispondono agli stati *pronto* e *bloccato* ma con spazio virtuale non allocato in memoria fisica. La necessità di mantenere aggiornati anche questi due stati deriva dal fatto che la CPU non è assegnabile ad un processo *pronto* se lo stesso non è anche allocato in memoria. Le transizioni fra gli stati *pronto* o *bloccato* e i corrispondenti stati con spazio virtuale non allocato in memoria fisica (*swapped*) corrispondono all'esecuzione delle primitive di *swap_in* e *swap_out* (vedi figura 4.21). La scelta del processo pronto a cui assegnare la CPU (assegnazione_CPU) e quella del processo pronto, ma *swapped*, da caricare in memoria (*swap_in*) corrispondono ai due livelli di schedulazione che nel capitolo 2 sono stati indicati come *short-term scheduling* e *medium-term scheduling* rispettivamente.

L'allocazione in memoria dell'intero spazio virtuale di un processo, e cioè di tutti i suoi segmenti, limita la dimensione complessiva dello spazio virtuale stesso. In particolare, la somma delle dimensioni di tutti i segmenti deve essere inferiore alla di-

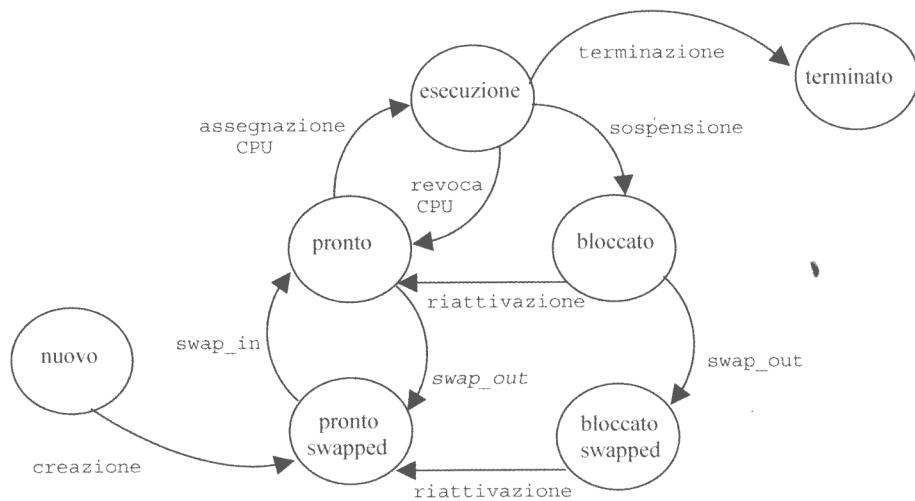


Figura 4.21 Stati swapped di un processo.

sibile complicare la funzione di rilocazione $y = f(x)$ in modo tale che, durante l'esecuzione di un processo, in ogni istante, sia possibile mantenere in memoria soltanto una parte del suo spazio virtuale, e cioè solo un sottoinsieme dei suoi segmenti, quelli strettamente necessari a quel punto della sua esecuzione. Per questo motivo è necessario abilitare la funzione di rilocazione, dato un indirizzo virtuale $x = \langle sg, of \rangle$ a restituire il corrispondente indirizzo fisico y se il segmento sg è presente in memoria e a generare un'interruzione in caso contrario (interruzione di segment-fault). La funzione di gestione di questa interruzione ha il compito di cercare in memoria fisica una partizione libera nella quale caricare il segmento richiesto dal processo e riattivare quindi l'esecuzione del processo stesso che dovrà rieseguire l'istruzione che ha generato l'interruzione di segment-fault per poter proseguire.

Questa tecnica viene spesso indicata col nome di segmentazione a domanda perché corrisponde esattamente alla tecnica della segmentazione già vista, ma con il criterio che i segmenti di un processo sono caricati soltanto quando, e se, sono necessari. Se all'atto del caricamento di un segmento non c'è spazio in memoria fisica, è possibile scaricare sulla swap-area uno o più segmenti dello stesso processo, o di altri processi (rimpiazzamento), al fine di fare spazio per il nuovo segmento. In questo modo le primitive di swap_out e di swap_in operano soltanto su singoli segmenti e non su un intero spazio virtuale. Cade, quindi, la necessità di mantenere aggiornati i due stati swapped visti nella figura 4.21 poiché adesso è possibile schedulare anche un processo che non ha neppure un segmento in memoria. Infatti, in questo caso, al primo tentativo di accesso alla memoria (il prelievo della prima istruzione) viene subito generato un segment-fault in seguito al quale il gestore della memoria carica il primo segmento di codice del processo e, successivamente, ancora a domanda, tutti gli altri segmenti richiesti.

I parametri di questa tecnica sono i seguenti:

Rilocazione	Allocazione della memoria fisica	Spazio virtuale	Caricamento dello spazio virtuale
Dinamica	Contigua	Segmentato	A domanda

Per abilitare la funzione di traduzione degli indirizzi a generare un *segment-fault* in mancanza del segmento a cui fare riferimento viene leggermente complicata la struttura di ogni descrittore di segmento rispetto a quella già vista nella precedente figura 4.19. Come indicato in figura 4.22, nella parte controllo di ogni descrittore, oltre ai bit relativi ai diritti di accesso già esaminati precedentemente, trovano spazio altri bit. Uno in particolare, indicato come bit di presenza (P), serve per indicare la presenza o meno del corrispondente segmento in memoria fisica. Quando $P = 1$ il segmento è allocato in memoria e, in questo caso, i campi base e limite contengono i dati della partizione nella quale il segmento è caricato. Se $P = 0$ ciò significa che il segmento non è in memoria e quindi che gli altri campi del descrittore non sono significativi. Ogni tentativo di tradurre un indirizzo virtuale $x = \langle sg, of \rangle$ il cui segmento sg abbia un descrittore nella tabella dei segmenti con bit P = 0 genera via hardware un'interruzione di *segment-fault*.

Come è già stato indicato, un'interruzione di *segment-fault* viene gestita in modo tale da caricare in memoria il segmento richiesto. Per questo motivo, viene ricercata una partizione disponibile, se esiste, altrimenti viene compattata la memoria se questa è frammentata. Ma se, anche compattando non c'è sufficiente memoria disponibile, è necessario scaricare uno o più segmenti, appartenenti sia al processo in esecuzione sia ad altri processi, eventualmente sospesi, in modo tale da liberare una partizione sufficiente a contenere il nuovo segmento. La scelta dei segmenti da rimpiazzare costituisce uno dei parametri fondamentali agli effetti dell'efficienza complessiva del sistema. Infatti, una scelta sbagliata che rimpiazzasse un segmento ancora necessario al processo genererebbe una nuova interruzione di *segment-fault* creando un eccessivo *overhead*.

Al fine di facilitare l'implementazione di algoritmi di rimpiazzamento efficienti, sono spesso disponibili ulteriori bit di controllo nel descrittore di un segmento. Ad esempio i bit U e M indicati in figura 4.22, e noti come bit di *uso* e di *modifica*. Questi bit possono essere letti e azzerati via software e vengono automaticamente posti al valore uno via hardware ogni volta che viene fatto un riferimento al segmento (bit U) e ogni volta che viene modificato il contenuto del segmento (bit M). L'importanza del bit di modifica deriva dal fatto che se viene scelto come segmento da rimpiazzare un segmento il cui bit M sia a zero (ammesso che tale bit fosse stato azzerato all'atto del caricamento del segmento stesso) ciò significa che tale segmento non è stato

base	limite	controllo
indirizzo della partizione	dimensione della partizione	R W U M P

Figura 4.22 Bit di presenza, di modifica e di uso.

modificato dopo essere stato allocato in memoria e quindi la partizione che lo contiene può essere liberata senza la necessità di riscrivere sulla *swap-area* il contenuto del segmento stesso con un conseguente risparmio di tempo. Il bit U viene invece utilizzato per verificare se un segmento è stato usato dal momento in cui il relativo bit di uso è stato azzerato. Ciò consente di valutare la frequenza di uso di ogni segmento e quindi di implementare algoritmi di scelta del segmento da rimpiazzare sulla base della frequenza dei riferimenti ai vari segmenti presenti in memoria. Una descrizione approfondita di tali algoritmi esula dagli scopi di questo testo. Per eventuali approfondimenti si rimanda alla bibliografia.

4.3.3 Memoria paginata

L'allocazione della memoria fisica mediante partizioni, che è alla base di tutte le tecniche viste precedentemente, soffre dell'inconveniente della frammentazione. Tale inconveniente può essere limitato nei sistemi segmentati compattando le partizioni libere. Ma il compattamento richiede comunque un tempo di CPU non indifferente. Per eliminare alla radice il problema della frammentazione sarebbe necessario poter allocare in memoria, in locazioni fisiche non necessariamente contigue, informazioni i cui indirizzi virtuali sono contigui. Utilizzando una funzione di rilocazione dinamica $y = f(x)$, ciò è concettualmente possibile. È sufficiente implementare nella MMU una funzione che a ogni indirizzo virtuale faccia corrispondere una locazione fisica qualsiasi, senza vincoli di contiguità. Come si può facilmente intuire, però, tale funzione dovrebbe costituire una tabella di corrispondenza di dimensioni esattamente uguali a quelle dello spazio virtuale e ciò vanificherebbe il tentativo di raggiungere il risultato desiderato. È possibile però raggiungere un compromesso se le locazioni dello spazio virtuale, invece che essere allocate singolarmente in locazioni fisiche indipendenti, vengono allocate a gruppi di locazioni virtuali contigue di dimensioni fisse, ad esempio di dimensioni d, allocando ogni gruppo in un gruppo di locazioni fisiche contigue anch'esso di dimensioni d, ma senza il vincolo che i gruppi fra loro contigui in memoria virtuale debbano essere fra loro contigui anche in memoria fisica. Ad esempio, in figura 4.23 è illustrato il caso in cui la dimensione d del gruppo di locazioni contigue (detto anche *pagina*) è 1024. L'esempio riportato in figura rappresenta uno spazio virtuale di 4096 locazioni (costituito quindi da 4 pagine) mentre la memoria fisica contiene 7168 locazioni (corrispondenti a 7 pagine). Nell'esempio, la prima pagina dello spazio virtuale (ognuna delle quali è detta anche *pagina virtuale*) è stata caricata nella terza pagina della memoria fisica (detta anche *pagina fisica* o *frame*). Ma, come si può osservare, la seconda pagina virtuale è caricata in una pagina fisica, la sesta, che non è contigua a quella in cui è stata caricata la precedente pagina virtuale.

Per realizzare la funzione di traduzione degli indirizzi è quindi sufficiente una tabella che registri le corrispondenze tra le pagine virtuali e le pagine fisiche che le contengono. Ad esempio, all'indirizzo virtuale 1024 (il primo della seconda pagina virtuale) corrisponde l'indirizzo fisico 5120 (il primo della sesta pagina fisica). Analogamente, al successivo indirizzo virtuale appartenente alla stessa pagina virtuale (1025) corrisponderà il successivo indirizzo fisico appartenente alla stessa pagina fisica (5121). In pratica, per tradurre un indirizzo virtuale x è necessario sapere a quale

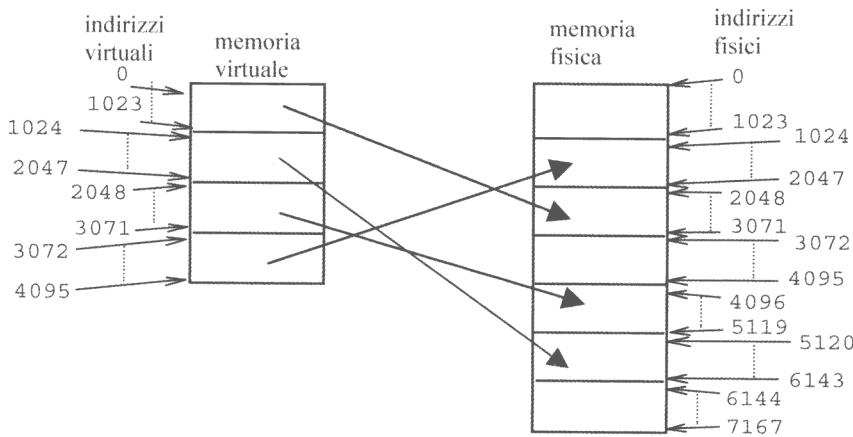
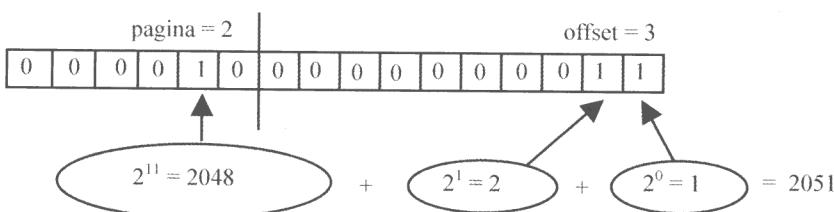


Figura 4.23 Paginazione.

ne d della pagina). Il quoziente della divisione fornisce l'indice dalla pagina virtuale a cui x appartiene mentre il resto fornisce lo scostamento di x (*offset*) nell'ambito della pagina. Se, come nell'esempio precedente, la dimensione d della pagina è una potenza di due ($d = 2^{10} = 1024$ nell'esempio), il resto della divisione è costituito dai 10 bit meno significativi di x mentre il quoziente intero dai restanti bit più significativi. Ad esempio, con un indirizzo virtuale a 16 bit, in figura 4.24 è riportato l'indirizzo virtuale 2051, che corrisponde all'indirizzo con scostamento 3 nell'ambito della pagina virtuale di indice 2.

Se per ogni processo viene mantenuta aggiornata una tabella di corrispondenza tra le pagine virtuali del suo spazio e le pagine fisiche in cui queste sono caricate (*tabella delle pagine*) la traduzione di un indirizzo virtuale x nel corrispondente indirizzo fisico y si effettua molto semplicemente ricavando, come è stato indicato precedentemente, la pagina virtuale pv e l'*offset* of di x , quindi, mediante l'indice pv si accede alla tabella delle pagine per verificare in quale pagina fisica pf la pagina virtuale pv è stata caricata. Noto pf , l'indirizzo y si ottiene semplicemente componendolo in modo tale che i suoi 10 bit meno significativi coincidano con of (lo scostamento nella pagina virtuale e nella corrispondente pagina fisica coincidono) mentre i bit più



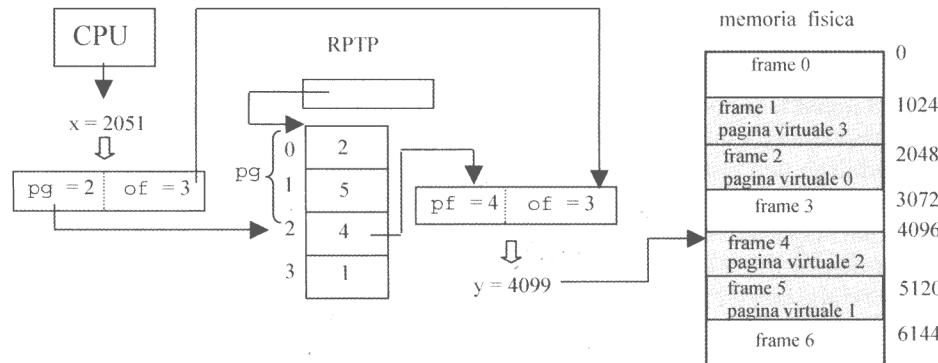


Figura 4.25 Meccanismo di traduzione degli indirizzi.

significativi coincidono con l'indice della pagina fisica pf individuata (vedi figura 4.25, dove la tabella delle pagine corrisponde all'esempio illustrato in figura 4.23).

Ogni processo allocato in memoria possiede una propria tabella delle pagine, necessaria per realizzare la sua funzione di traduzione degli indirizzi. La dimensione della tabella dipende dal numero di pagine che compongono lo spazio virtuale del processo e questo numero, a sua volta, dipende dalla dimensione dell'intero spazio e dalla dimensione delle pagine. Supponendo, come nel precedente esempio, che una pagina contenga 1024 locazioni, lo spazio virtuale di un programma di medie dimensioni, ad esempio di 256 kbyte, è composto da 256 pagine e quindi richiede una tabella delle pagine di 256 elementi. Come le tabelle dei segmenti nei sistemi segmentati, così anche le tabelle delle pagine non possono essere completamente allocate nei registri della MMU. È viceversa necessario allocarle in memoria fisica. Esiste, quindi, un particolare registro di macchina che in ogni istante contiene l'indirizzo fisico della tabella delle pagine del processo in esecuzione. In figura 4.25 è stato indicato con l'acronimo *RPTP* (*Registro Puntatore alla Tabella delle Pagine*).

La tecnica della paginazione è caratterizzata dai seguenti parametri:

Rilocazione	Allocazione della memoria fisica	Spazio virtuale	Caricamento dello spazio virtuale
Dinamica	Non contigua	Unico	Tutto insieme

Per ridurre il numero di accessi alla tabella delle pagine, anche nei sistemi che realizzano il meccanismo della paginazione, così come è stato visto nel caso della segmentazione, nella MMU è presente un numero limitato di registri associativi veloci (normalmente da 4 a 8) nei quali vengono mantenute le corrispondenze tra pagine virtuali e pagine fisiche relativamente alle ultime pagine virtuali a cui il processo ha fatto riferimento (*Translation Lookaside Buffer*). Quando la CPU genera l'indirizzo virtuale x , la MMU viene interrogata con l'indice della pagina virtuale pg a cui x appartiene. Se la corrispondenza tra pg e l'indice pf della pagina fisica in cui pg è ca-

ricata è presente in uno dei registri della MMU, l'indirizzo fisico viene calcolato prelevando p_f direttamente dalla MMU. Altrimenti la traduzione avviene normalmente come indicato nella precedente figura 4.25, e alla fine della traduzione la coppia $pg - p_f$ viene registrata in uno dei registri della MMU, eventualmente rimpiazzando uno dei registri, di norma con politica *First-In-First-Out*.

Il gestore della memoria fisica mantiene aggiornato l'elenco delle pagine fisiche disponibili in una propria struttura dati, normalmente una tabella con tanti elementi quante sono le pagine fisiche della memoria (detta *tabella delle pagine fisiche* o *frame table*). Ogni elemento della tabella contiene l'indicazione se la corrispondente pagina fisica è libera o occupata e, in quest'ultimo caso, contiene l'indice del processo a cui è allocata e l'indice della pagina virtuale in essa caricata.

Quando un processo deve essere caricato in memoria, viene richiesto al gestore un numero di pagine fisiche disponibili uguale al numero di pagine virtuali del suo spazio. Le pagine ottenute, anche se non contigue, vengono utilizzate per caricarci le pagine virtuali e generare quindi la tabella delle pagine del processo. Se le pagine fisiche disponibili non sono sufficienti, non è necessaria nessuna forma di compattamento. È sufficiente scaricare, mediante la primitiva di `swap_out` le pagine occupate da uno o più processi, fino ad ottenere il numero sufficiente di pagine fisiche libere. Anche nei sistemi che realizzano la paginazione gli stati di un processo sono quelli rappresentati in figura 4.21. Potendo infatti revocare dinamicamente memoria sono necessari i due stati *swapped* quando lo spazio virtuale di un processo non si trova allocato in memoria fisica.

Il campo del descrittore di ogni processo relativo alla memoria virtuale contiene, in questo caso, due informazioni: l'indirizzo in memoria della tabella delle pagine del processo e un intero che corrisponde al numero delle pagine virtuali del suo spazio, e cioè al numero di elementi della tabella delle pagine. Queste due informazioni descrivono completamente lo spazio di memoria fisica allocata al processo.

Un parametro importante nei sistemi paginati è costituito dalla dimensione di ogni pagina. Al diminuire di queste dimensioni aumenta di conseguenza la dimensione della tabella delle pagine. Al limite, come è già stato messo in evidenza, se la pagina contenesse un solo byte, la non contiguità delle locazioni fisiche sarebbe completa ma la dimensione della tabella coinciderebbe con la dimensione dello stesso spazio virtuale. D'altra parte, se la dimensione della pagina viene aumentata eccessivamente s'introduce di nuovo una forma di frammentazione interna. Infatti, difficilmente la dimensione di uno spazio virtuale è multipla della dimensione delle pagine. Per cui la pagina fisica che contiene l'ultima pagina virtuale è utilizzata solo in parte. Mediamente, metà della pagina resta inutilizzata generando quindi questa forma di frammentazione interna alla pagina. All'aumentare della dimensione della pagina aumenta l'effetto negativo di questo tipo di frammentazione. È perciò necessario ricorrere ad un compromesso fra i due precedenti inconvenienti. Valori tipici della dimensione delle pagine oscillano fra 512 byte e 4 Kbyte.

Per quanto riguarda la protezione, con la paginazione possono essere effettuati gli stessi controlli visti nel caso della segmentazione anche se alcuni di questi, come vedremo, sono in questo caso molto meno significativi. Il primo controllo riguarda la verifica che ogni indirizzo virtuale x generato dalla CPU appartenga allo spazio del processo. In particolare viene effettuato il controllo che la pagina pg a cui x ap-

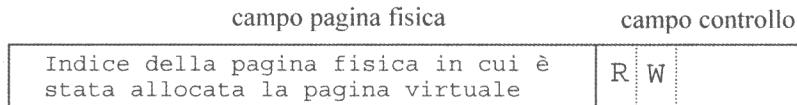


Figura 4.26 Elemento della tabella delle pagine.

partiene sia una delle pagine del processo verificando che il suo valore sia minore o uguale al contenuto di un particolare registro di macchina (il registro *PTLR – Page Table Length Register*) nel quale, al momento della schedulazione di un processo, viene caricato il numero di pagine virtuali del suo spazio. Se pg supera il valore di tale registro viene generata un'eccezione. È poi possibile associare ad ogni pagina virtuale degli specifici diritti di accesso, come è stato visto nel caso della segmentazione, in modo tale che ogni riferimento a quella pagina sia consistente con tali diritti. Ciò può essere ottenuto associando ad ogni elemento della tabella delle pagine un nuovo campo *controllo* (vedi figura 4.26) all'interno del quale sono specificati i diritti di accesso, ad esempio in lettura (R) e in scrittura (W).

La minore significatività di questo controllo, nei confronti dell'analogo controllo effettuato nel caso della segmentazione, deriva dal fatto che la pagina virtuale, a differenza di un segmento, non individua un elemento logico del programma ma una parte del suo spazio di memoria ed è quindi più difficile associarle dei diritti di accesso. Ad esempio, tenendo conto che lo spazio virtuale paginato è contiguo, se l'ultima pagina in cui è allocato il codice contenesse anche parte dei dati, sarebbe impossibile associarle dei diritti di accesso in quanto relativi ad una pagina contenente informazioni di natura diversa.

Le stesse considerazioni possono essere fatte per quanto riguarda la condivisione. In questo caso è possibile abilitare due o più processi a condividere delle pagine (vedi figura 4.27) ma, di nuovo, ciò che viene condiviso è un insieme di pagine che non necessariamente coincide con un insieme di moduli di programma.

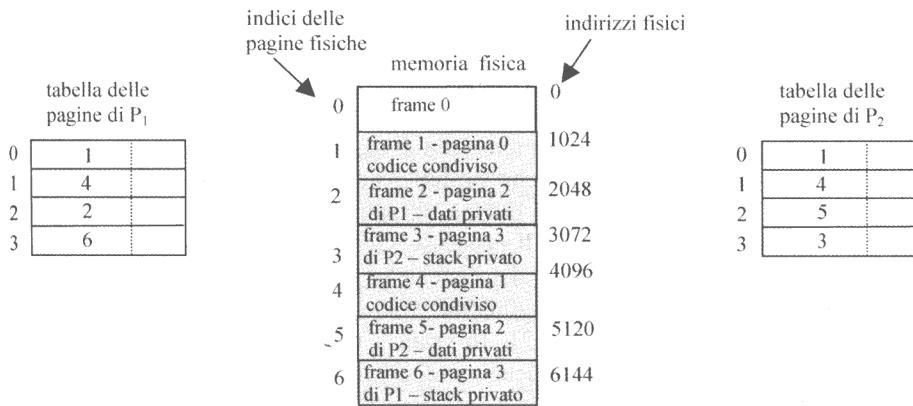


Figura 4.27 Condivisione di pagine.

In figura 4.27, in particolare, viene illustrato il caso di due processi che condividono lo stesso codice (allocato nelle prime due pagine virtuali nei rispettivi spazi e nelle pagine fisiche 1 e 4 rispettivamente). Il processo P1 contiene poi una pagina di dati privati (la numero 2 allocata nella pagina fisica 2) e una pagina di stack (la 3 allocata nella pagina fisica 6). Analogamente P2 contiene le due pagine dati e stack allocate rispettivamente nelle pagine fisiche 5 e 3.

Infine, per quanto riguarda la condivisione, è opportuno ricordare che se le pagine condivise contengono degli indirizzi, essendo questi indirizzi virtuali, ciò comporta il vincolo che le informazioni condivise devono essere allocate nelle stesse locazioni dei rispettivi spazi virtuali.

Paginazione a domanda Con la tecnica della paginazione un processo deve avere l'intero spazio virtuale in memoria per poter eseguire. Come indicato nel paragrafo 4.2.5, però, sono molti i vantaggi che possiamo ottenere abilitando un processo a caricare in memoria le parti del proprio spazio virtuale soltanto se, e quando, sono necessarie. Questa tecnica, analoga a quella già vista nel caso della segmentazione a domanda, è quella più usata anche nei sistemi paginati. È infatti sufficiente sofisticare leggermente il meccanismo hardware di traduzione di un indirizzo virtuale paginato per consentire la realizzazione di questa tecnica, molto più complessa da un punto di vista software, come vedremo, ma molto più flessibile ed efficiente, nota col nome di *paginazione a domanda (demand paging)*. I parametri caratteristici di questa tecnica sono i seguenti:

Rilocazione	Allocazione della memoria fisica	Spazio virtuale	Caricamento dello spazio virtuale
Dinamica	Non contigua	Unico	A domanda

Il meccanismo hardware è molto simile a quello già visto nel caso della segmentazione a domanda. Nel campo *controllo* di ogni elemento della tabella delle pagine di un processo viene inserito il bit di presenza (P) che caratterizza la presenza in memoria, o meno, della corrispondente pagina virtuale. Analogamente sono utilizzati i bit di uso (U) e di modifica (M) da utilizzare con gli stessi significati visti nel caso della segmentazione, quando sia necessario rimpiazzare delle pagine (vedi figura 4.28).

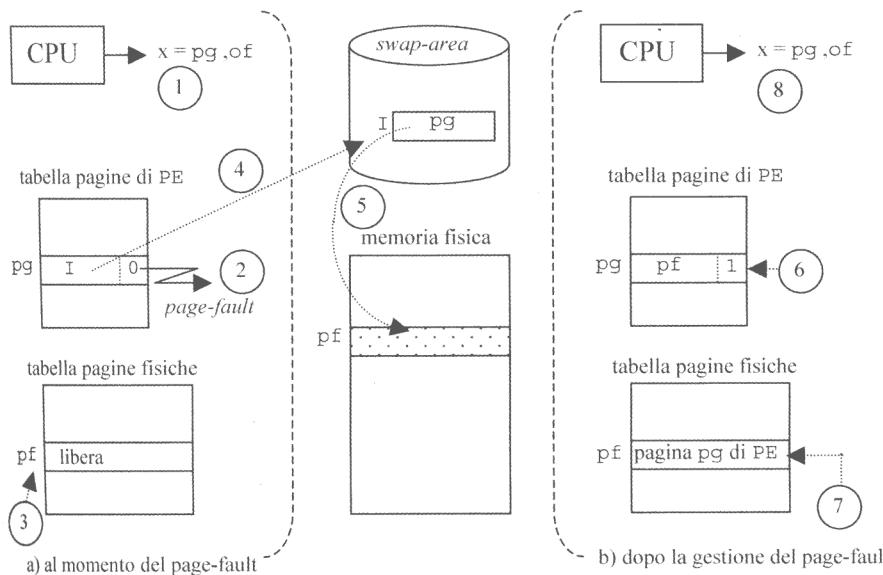
Questo tipo di gestione prevede che lo spazio virtuale di un processo, alla sua creazione, risieda completamente su memoria di massa. Normalmente le caratteristiche del disco di sistema sono tali che le dimensioni di un settore coincidono con quelle della pagina. Per cui, tutte le pagine virtuali inizialmente risiedono su altrettanti settori fisici sulla *swap-area*. Il processo viene creato senza caricare nessuna pagina, ma può subito essere schedulato anche se non ha niente in memoria utilizzando la tecnica del caricamento a domanda. Cade quindi l'esigenza di mantenere i due stati *swapped* visti in figura 4.21. In pratica, a regime un processo non è mai completamente in memoria o completamente su *swap-area*. La tabella delle pagine viene creata con tutti i bit di presenza a zero mentre nel campo destinato a contenere l'indice della pagina fisica quando la pagina è in memoria, viene inserito l'indirizzo della pagina su *swap-area*.

campo pagina fisica	campo controllo
indice della pagina fisica se P=1; indirizzo su disco se P=0	R W U M P

Figura 4.28 Bit di presenza (P), di uso (U) e di modifica (M).

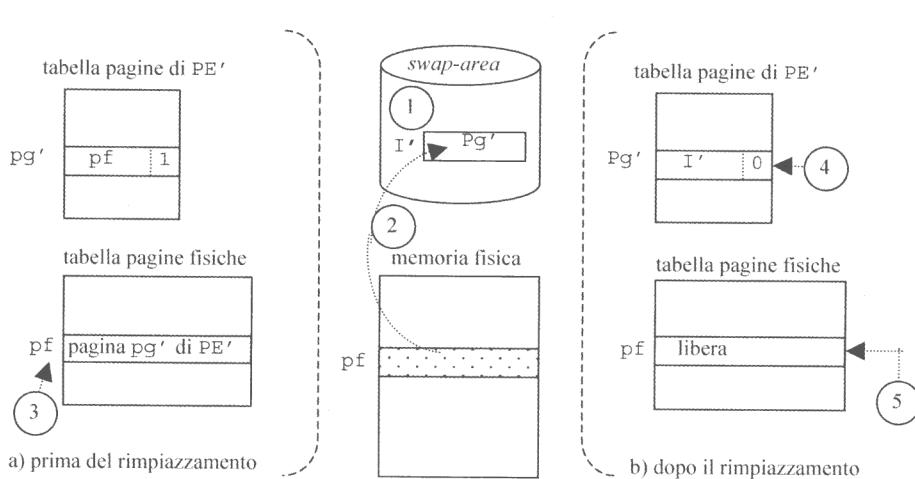
Ad ogni riferimento relativo a una pagina non in memoria (bit $P = 0$) viene generata un'interruzione di *mancanza di pagina* (*page-fault*) che viene gestita dal sistema operativo per ricercare nella tabella delle pagine fisiche, una pagina fisica in cui caricare la pagina virtuale richiesta. La pagina viene caricata prelevandola dalla *swap-area* all'indirizzo contenuto nel corrispondente elemento della tabella delle pagine. Una volta caricata in memoria la pagina virtuale richiesta e aggiornata di conseguenza la tabella delle pagine del processo, l'istruzione che ha generato *page-fault* può essere rieseguita e il processo può continuare la sua esecuzione. Se, al momento della ricerca di una pagina fisica disponibile, tutte le pagine fossero occupate, sarebbe necessario rimpiazzarne una. Ma il rimpiazzamento di una pagina è molto più semplice rispetto al rimpiazzamento di un segmento essendo tutte le pagine di uguali dimensioni.

Nella figura 4.29 viene illustrato il caso in cui il processo in esecuzione PE genera un *page-fault* nell'ipotesi che sia disponibile almeno una pagina fisica. Nella parte a), a sinistra, sono riportate la tabella delle pagine del processo e la tabella delle pagine fisiche del gestore della memoria all'atto della generazione del *page-fault*, mentre nella parte b), a destra, sono riportate le stessa tabelle alla fine della gestione del *page-fault*.



La CPU genera l'indirizzo virtuale x contenuto nella pagina virtuale pg (1). Poiché nell'elemento della tabella delle pagine di indice pg il bit di presenza P è uguale a zero, viene generata l'eccezione di *page-fault* (2). Viene fatta richiesta al gestore della memoria di una pagina fisica disponibile. Dalla tabella delle pagine fisiche viene ricavato l'indice pf di una pagina disponibile (3). Quindi dalla tabella delle pagine si ricava l'indirizzo I' sulla *swap-area* della pagina pg richiesta (4) e viene attivato un trasferimento da disco a memoria principale per caricare la pagina virtuale pg nella pagina fisica pf (5). A questo punto della gestione dell'interruzione di *page-fault* il processo PE perde il controllo e, mediante una commutazione di contesto, la CPU viene assegnata ad un altro processo pronto. Al termine del caricamento il processo PE può riprendere il controllo della CPU e a questo punto vengono aggiornate sia la tabella delle pagine di PE (6) che la tabella delle pagine fisiche, indicando che pf non è più disponibile (7) e, infine, viene riattivato il processo PE rieseguendo l'istruzione che ha precedentemente generato *page-fault* (8). Nel caso in cui al precedente punto (3) non fosse trovata nessuna pagina fisica libera, con un particolare algoritmo, detto *algoritmo di rimpiazzamento*, verrebbe scelta la pagina pf da rimpiazzare.

Per effettuare il rimpiazzamento, vedi figura 4.30, viene ricercata sulla *swap-area* un'area disponibile di indirizzo I' nella quale trasferire il contenuto della pagina fisica pf da rimpiazzare (1) e viene quindi trasferito il contenuto di pf su disco a tale indirizzo (2). Anche in questo caso il processo PE perderebbe il controllo in attesa della fine del trasferimento. Quindi, dalla tabella delle pagine fisiche, dall'elemento di indice pf , viene ricavato l'indice PE' del processo a cui la pagina è stata revocata e l'indice pg' della sua pagina virtuale caricata in pf (3). Per completare il rimpiazzamento è ora necessario porre a zero il bit di presenza P nell'elemento di indice pg' della tabella delle pagine di PE' (4). In tale elemento viene quindi scritto l'indirizzo I' della *swap-area* dove la pagina è stata trasferita. Infine, nella tabella



delle pagine fisiche, si indica che ora p_f è libera (5). Nella parte a), a sinistra nella figura, è indicato lo stato delle tabelle coinvolte prima del rimpiazzamento, mentre nella parte b), a destra, il loro stato alla fine del rimpiazzamento.

In caso di necessità di rimpiazzamento, il trasferimento su disco della pagina scelta sarebbe evitato se tale pagina non fosse stata modificata dopo il suo caricamento in memoria (bit di modifica $M = 0$).

Nell'eseguire l'algoritmo di rimpiazzamento, per trovare la pagina fisica più adatta ad essere rimpiazzata è necessario che certe pagine fisiche non siano prese in considerazione dall'algoritmo. Per esempio, se una pagina fisica p_f contiene la pagina virtuale pg di un processo P il quale ha attivato, tramite un canale di DMA, un trasferimento di dati da un dispositivo periferico in un suo buffer contenuto nella propria pagina virtuale pg (o viceversa), p_f non deve essere scelta. Infatti il processo P è sicuramente bloccato in attesa che il trasferimento di I/O termini, ma il canale di DMA, che è stato attivato per trasferire i dati nel buffer, continua ad accedere alla pagina fisica p_f e a inserire dati nel buffer in essa allocato. Se p_f fosse rimpiazzata il canale di DMA continuerebbe a trasferire dati dentro questa pagina fisica andando in collisione col contenuto della nuova pagina virtuale caricata in p_f . Per questo motivo nella tabella delle pagine fisiche ogni elemento, oltre al campo che indica se la corrispondente pagina fisica è libera oppure no, contiene anche un campo (è sufficiente un solo bit), detto campo di *lock*, per indicare, quando posto al valore 1, che la corrispondente pagina fisica non deve essere presa in considerazione come candidata ad un rimpiazzamento (vedi figura 4.31).

Dalle precedenti considerazioni emergono chiaramente due aspetti contrastanti della paginazione a domanda. Da un lato, la possibilità di rendere lo spazio virtuale indipendente dalle dimensioni della memoria fisica e di caricare le parti di un programma solo quando, e se, sono necessarie mediante le interruzioni di *page-fault*, tende ad aumentare la flessibilità del sistema e l'efficienza della gestione della memoria fisica. Dall'altro, la necessità di gestire tali interruzioni, con la conseguente necessità di commutare la CPU, riduce l'efficienza con cui un processo esegue il proprio programma. È quindi evidente che, al fine di non ridurre eccessivamente l'efficienza, è necessario minimizzare il numero di *page-fault* che un processo può generare durante la sua esecuzione. Tale numero dipende, ovviamente, dal programma che il processo esegue. Tanto più il programma è ben strutturato tanto minore sarà il numero di *page-fault*. All'aumentare però del numero di processi gestiti dal sistema, aumenta la richiesta di pagine fisiche e ciò porta inevitabilmente a saturare la memoria. In questo caso, ad ogni *page-fault* è necessario rimpiazzare una delle pagine già occupate. Se l'algoritmo di rimpiazzamento non effettua una scelta oculata della pagina da rimpiazzare, il numero di *page-fault* può aumentare considerevolmente. Infatti, può accadere che la pagina scelta per il rimpiazzamento sia

Indicazione se la pagina è libera o, se occupata, quale pagina virtuale contiene e di quale processo	bit di lock
--	-------------

Figura 4.31 Elemento della tabella delle pagine fisiche.

di nuovo richiesta nell'immediato futuro generando a sua volta un nuovo *page-fault*. Se poi questo fenomeno si ripete, il sistema rischia di entrare in uno stato, noto col nome di *trashing*, nel quale l'attività della CPU è fondamentalmente dedicata a trasferire pagine avanti e indietro dalla *swap-area*, riducendo l'efficienza del sistema a livelli molto bassi. Si capisce quindi l'estrema importanza che in questo tipo di sistemi acquisisce la scelta di un buon algoritmo di rimpiazzamento delle pagine.

Rimpiazzamento delle pagine Per valutare la bontà di un algoritmo di rimpiazzamento si può ipotizzare di avere un solo processo in memoria e, nell'ipotesi di avere un numero di pagine fisiche inferiore al numero di pagine virtuali referenziate dal processo, verificare il numero di *page-fault* generati dal processo durante la sua esecuzione. È chiaro che tale numero dipende dal programma eseguito e, per ogni programma, dall'algoritmo di rimpiazzamento. Infatti, dopo che sono state caricate tante pagine virtuali quante sono le pagine fisiche disponibili, al primo riferimento ad una nuova pagina è necessario rimpiazzarne una di quelle già caricate. Da questo momento in poi il numero di *page-fault* dipende dalle scelte effettuate dall'algoritmo di rimpiazzamento.

Esiste un algoritmo, detto *algoritmo ottimo*, che sicuramente è quello che dà luogo al minor numero possibile di *page-fault*. Tale algoritmo è quello che sceglie come pagina da rimpiazzare una di quelle che sicuramente non verranno più riferite nel futuro o, se tutte le pagine in memoria sono ancora necessarie, quella che verrà riferita più tardi nel tempo. Sfortunatamente, come si può capire, tale algoritmo è del tutto teorico e irrealizzabile, poiché prevede che si abbia una conoscenza di come il programma si comporterà nel futuro. È stato proposto esclusivamente come algoritmo di riferimento, da utilizzare come termine di paragone nei confronti dei vari algoritmi realizzati nei sistemi reali. Questi ultimi, non potendo prevedere il futuro, si basano tutti su informazioni relative agli accessi effettuati nell'immediato passato. Infatti, a partire da un generico istante e per un certo periodo di tempo, i riferimenti alla memoria generati da un processo sono localizzati all'interno di un numero limitato di pagine virtuali (spesso indicate come *insieme di lavoro* del processo o *working set*). Ad esempio le pagine nelle quali è contenuto il codice della funzione eseguita, la pagina contenente la cima dello stack e quelle dei dati riferiti dalla funzione. L'insieme di lavoro ovviamente cambia nel tempo ma in modo graduale, mentre eventuali discontinuità si verificano, ad esempio, al momento della chiamata di una nuova funzione. Questo aspetto del comportamento di un programma è noto anche col nome di *località dei riferimenti*.

Uno degli algoritmi più semplici da realizzare è quello *FIFO* (*First-In-First-Out*). In questo caso si tratta di scegliere come pagina da rimpiazzare quella che da più tempo è in memoria, sperando che sia anche una di quelle non più necessarie. Purtroppo questa ipotesi non è per niente certa per cui viene spesso rimpiazzata una pagina ancora necessaria e che quindi verrà di nuovo richiesta generando un successivo *page-fault*. L'unico aspetto positivo di questo algoritmo è la semplicità di realizzazione. È infatti sufficiente concatenare tra loro gli elementi della tabella delle pagine fisiche in modo tale che compongano una coda.

Un algoritmo molto più complesso da realizzare, ma molto più efficiente in termini di *page-fault*, è quello noto con l'acronimo *LRU* (*Least Recently Used*) che sce-

glie come pagina da rimpiazzare quella meno recentemente utilizzata, indipendentemente da quando è stata caricata. È stato verificato sperimentalmente che questo algoritmo, fra tutti quelli che sono stati proposti, ha delle prestazioni che più si avvicinano a quelle dell'algoritmo ottimo. Purtroppo la sua realizzazione è estremamente costosa sia in termini di ausili hardware che di supporto software necessari per una sua corretta implementazione. Per questo motivo non è mai stato realizzato completamente.

Molti algoritmi utilizzati nella pratica sono delle approssimazioni, più o meno grossolane, dell'algoritmo LRU. Una fra queste è quella che corrisponde all'algoritmo noto col nome di *second-chance* (o anche *clock algorithm*) descritto in figura 4.32. Per valutare se le pagine presenti in memoria sono state recentemente utilizzate oppure no viene utilizzato il bit di uso (U) ad esse associato. In pratica le pagine vengono suddivise in due categorie: quelle col bit di uso a 1 (le più recentemente usate) e quelle col bit di uso a zero (le meno recentemente usate). Si sceglie con tecnica FIFO una di quelle meno recentemente usate, se ci sono, altrimenti, sempre con tecnica FIFO, si sceglie una appartenente all'altra categoria.

Per questo motivo, la tabella delle pagine fisiche viene gestita come un array circolare (*round robin*). Viene mantenuta aggiornata la variabile *vittima*, un puntatore contenente l'indice della pagina fisica successiva a quella che è stata rimpiazzata per ultima. Al prossimo *page-fault* si inizia a verificare la pagina il cui indice è contenuto nella variabile *vittima*. In particolare si esamina il bit di uso ad essa associato. Se questo è zero la pagina viene scelta per il rimpiazzamento. In caso contrario, si azzera il bit di uso e si incrementa la variabile *vittima* per esaminare la successiva pagina fisica fino a quando non se ne trova una col bit di uso uguale a zero. Poiché l'incremento della variabile *vittima* è fatto operando il modulo M (se M è il numero delle pagine fisiche), abbiamo la garanzia che una pagina col bit di uso uguale a zero si trova certamente. Nel caso più sfavorevole è quella da cui è iniziata la scansione dopo aver scandito l'intera tabella (da cui il nome *second chance* del-

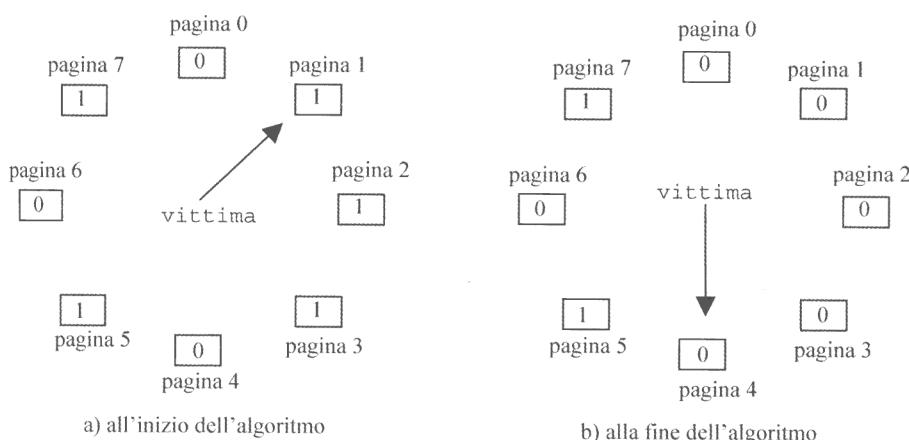


Figura 4.32 Algoritmo second chance.

l'algoritmo). Nella figura 4.32 viene illustrato il caso in cui le pagine fisiche sono 8 e il bit di uso è indicato all'interno del quadratino che rappresenta ogni elemento della tabella delle pagine fisiche. Dalla figura si capisce anche il perché dell'altro nome con cui questo algoritmo è noto (*clock algorithm*). Spesso, per ridurre il numero di trasferimenti tra memoria e disco, viene preso in considerazione anche il bit di modifica (M). In questo caso, le pagine sono classificate in quattro categorie in base ai valori dei due bit di uso e di modifica: (U - M: 0-0, 0-1, 1-0, 1-1). Rispetto alla scelta basata sul solo bit di uso si privilegiano le pagine col bit di modifica a zero.

Molti altri algoritmi di rimpiazzamento sono stati proposti e realizzati in vari sistemi operativi. Per un loro esame più approfondito si rimanda alla bibliografia.

Molte sono anche le varianti di ogni algoritmo a seconda di alcune scelte relative ai criteri di allocazione delle pagine fisiche. In particolare, tenendo conto che a differenza di quanto esposto precedentemente, le pagine fisiche non sono allocate a un solo processo, all'atto di un *page-fault* se è necessario rimpiazzare una pagina, si hanno due possibilità: scegliere la pagina da rimpiazzare fra tutte le pagine fisiche indipendentemente dai processi a cui sono allocate (tecnica di *rimpiazzamento globale*), oppure scegliere come vittima da rimpiazzare una delle pagine già allocate al processo che ha generato *page-fault* (*rimpiazzamento locale*). Il primo criterio è sicuramente quello che offre la maggiore flessibilità nell'effettuare la scelta più opportuna. Però l'efficienza con cui un processo esegue il suo programma può essere influenzata negativamente dal fatto che un altro processo, eseguendo un programma non strutturato e generando quindi molti *page-fault*, gli può revocare alcune pagine obbligandolo, a sua volta, a richiamarle successivamente.

La tecnica del rimpiazzamento locale implica poi che ogni processo abbia un pool di pagine fisiche a sua disposizione. Ciò comporta la necessità di definire un criterio con cui allocare le pagine fisiche ai processi (per esempio suddividendole in ugual numero fra tutti oppure in proporzione alle dimensioni dei rispettivi spazi virtuali).

Un'altra variante riguarda il numero di pagine virtuali caricate in memoria alla creazione di un processo. Fino ad ora abbiamo esaminato il caso che prevede di schedulare un processo, dopo che è stato creato, senza aver caricato in memoria nessuna delle sue pagine virtuali (criterio indicato anche come paginazione a domanda senza *pre-paging*). In alternativa, alcuni sistemi pre-allocano un certo numero di pagine virtuali al fine di ridurre il numero di *page-fault* iniziali (tecnica del *pre-paging*).

Molti sistemi, fra cui Unix, prevedono di mantenere sempre disponibili un certo numero di pagine fisiche in modo tale che, all'atto di un *page-fault*, la sua gestione possa essere effettuata più velocemente. Ciò implica che, periodicamente, il sistema rimpiazza un certo numero di pagine fisiche in modo tale da inserirle nella lista delle pagine libere. Se una delle pagine rimpiazzate e inserite nella lista delle pagine libere, fosse richiesta dal processo a cui è stata tolta prima di essere usata per caricarvi una nuova pagina virtuale, il processo potrebbe riprenderla senza la necessità di effettuare un trasferimento da disco.

Infine, in alcuni sistemi, al fine di evitare il fenomeno del *trashing*, viene effettuata una valutazione approssimativa dell'insieme di pagine virtuali che caratterizzano il suo *working set*. Tale valutazione viene effettuata monitorando i bit di uso delle

pagine a intervalli Δt costanti. Quando un processo genera *page-fault*, se tutte le pagine virtuali che il processo ha in memoria appartengono ancora al suo *working set*, ciò significa che questo insieme si sta espandendo e che, quindi, è necessario disporre di una ulteriore pagina fisica. Ma se la memoria è satura, invece di rimpiazzare una delle sue pagine che sarebbe richiesta di nuovo nell'immediato futuro, si preferisce effettuare lo *swap-out* di tutte le pagine di quel processo, introducendo di nuovo gli stati *swapped*.

4.3.4 Memoria segmentata e paginata

Le due tecniche della segmentazione e della paginazione, esaminate nei precedenti paragrafi, pur avendo molte somiglianze dal punto di vista del meccanismo di traduzione degli indirizzi, sono, in realtà, profondamente diverse. La prima corrisponde a un criterio di organizzazione della memoria virtuale, non più vista come una piatta sequenza di byte, ma come un insieme di moduli separati e semanticamente indipendenti. La seconda corrisponde ad una tecnica per gestire la memoria fisica, permettendo la sua allocazione in maniera non contigua ed eliminando così il fenomeno della frammentazione esterna. È quindi possibile concepire una tecnica che preveda sia la strutturazione dello spazio virtuale in segmenti sia l'allocazione di ogni segmento mediante la tecnica della paginazione.

Questa tecnica, che offre i vantaggi sia della segmentazione che della paginazione, viene anche chiamata segmentazione paginata, ed è caratterizzata dai seguenti parametri:

Rilocazione	Allocazione della memoria fisica	Spazio virtuale	Caricamento dello spazio virtuale
Dinamica	Non contigua	Segmentato	A domanda

Il primo sistema operativo che ha adottato questa tecnica è il Multics, un sistema ormai obsoleto ma famoso per aver introdotto molti dei principali concetti che stanno alla base del progetto dei moderni sistemi operativi. In figura 4.33 è riportato uno schema semplificato del meccanismo utilizzato nel Multics. Come si può notare, alla base del meccanismo c'è la segmentazione. La CPU genera un indirizzo virtuale x composto dalle due componenti segmento sg e scostamento sc nel segmento. La componente segmento, a parte i vari controlli di protezione già visti nella figura 4.18, viene utilizzato per accedere al descrittore del segmento, presente nella tabella dei segmenti. Il descrittore contiene ancora i due campi: valore *limite* e *base*. Solo che ora la *base* non rappresenta l'indirizzo in memoria di una partizione usata per contenere il segmento poiché quest'ultimo viene allocato in forma paginata. Per cui la *base* è l'indirizzo in memoria della tabella delle pagine del segmento stesso. Ogni segmento ha una sua propria tabella delle pagine. Lo scostamento sc , che rappresenta un indirizzo lineare nel segmento, viene paginato scomponendolo nei bit meno significativi (*offset of* nella pagina) e in quelli più significativi (pagina virtuale pg). L'indice pg della pagina virtuale viene usato per accedere alla tabella delle pagine e ricavare l'indice della pagina fisica pf nella quale pg è stata caricata. Quindi l'indirizzo fisico con cui accedere alla memoria viene composto tramite pf e of .

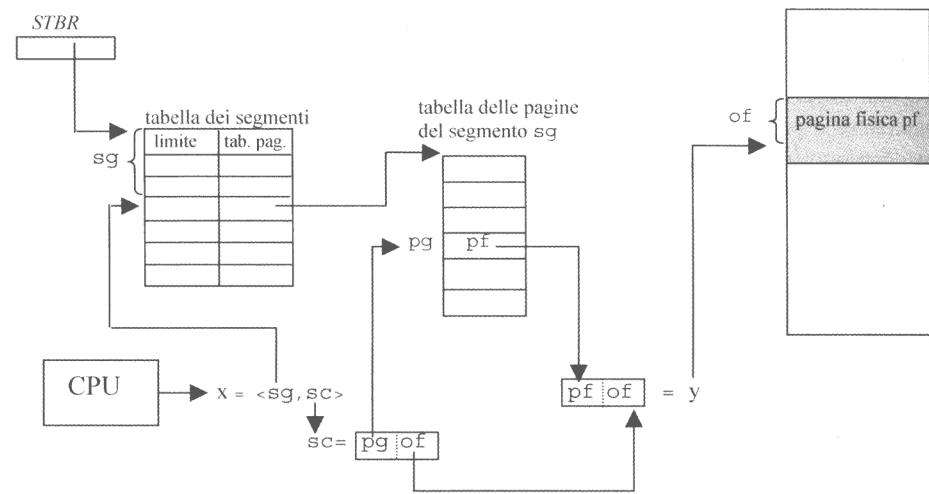


Figura 4.33 Segmentazione paginata.

Ogni indirizzo virtuale può generare più interruzioni. Può essere a zero il bit di presenza nel descrittore del segmento. Ciò implica che la tabella delle pagine del segmento non è in memoria e quindi l'interruzione dei *segment-fault* viene gestita per caricare tale tabella. Inoltre può essere a zero il bit di presenza nell'elemento della tabella delle pagine, generando quindi un normale *page-fault*.

Uno schema simile, anche se diverso nei dettagli e più flessibile, è utilizzato nei microprocessori della famiglia Intel. In questi è possibile utilizzare indirizzi virtuali segmentati e, volendo, anche paginati. Ogni indirizzo virtuale è costituito dalla coppia segmento sg e scostamento sc e tradotto, secondo il meccanismo tipico della segmentazione, come indicato in figura 4.18 per ottenere un indirizzo lineare a 32 bit. Questo, se la paginazione non è abilitata, costituisce l'indirizzo fisico. Se però, in base al valore di un bit particolare del registro di stato della CPU, la paginazione è abilitata, tale indirizzo lineare viene poi paginato e ulteriormente tradotto tramite tabelle delle pagine per ottenere l'indirizzo fisico corrispondente.

4.3.5 Gestione degli spazi virtuali

L'uso di meccanismi di rilocazione dinamica crea una serie di problemi relativi alla gestione degli spazi virtuali, problemi che sono comuni sia ai meccanismi della segmentazione che a quelli della paginazione.

Un primo problema è già stato messo in evidenza nei precedenti paragrafi, ed è relativo alla condivisione delle informazioni. Tale problema deriva dal fatto che quando parte di uno spazio virtuale è caricato in memoria, se contiene degli indirizzi questi vengono trasferiti in memoria senza essere rilocati. Se questa porzione di spazio virtuale fosse condivisa tra più processi, sarebbero condivisi anche gli indirizzi virtuali in essa contenuti. È quindi evidente che le informazioni condivise dovrebbero essere allocate nelle stesse posizioni nei singoli spazi virtuali.

Un secondo problema riguarda gli indirizzi generati dalla CPU quando vengono eseguite funzioni del sistema operativo. Nei precedenti paragrafi è stato messo in evidenza che ogni processo possiede una propria funzione di rilocazione $y = f(x)$. Quando è in esecuzione un certo processo P , è tramite la funzione ad esso associata che la MMU traduce un indirizzo virtuale x nel corrispondente indirizzo fisico y . Se però, durante la sua esecuzione, P tramite una *SVC* (*Super Visor Call*) invoca una primitiva del sistema, la CPU inizia a generare indirizzi relativi a istruzioni e dati del sistema operativo. Il problema che nasce è il seguente: a quale spazio virtuale appartengono tali indirizzi?

Nei primi sistemi che hanno adottato i meccanismi di rilocazione dinamica la soluzione che fu adottata fu quella di riservare anche al sistema operativo un proprio spazio virtuale, indipendente da quello di qualunque processo. Tale soluzione presenta però due inconvenienti.

Il primo è relativo ad un appesantimento del meccanismo delle chiamate al sistema. Infatti, in questo caso, oltre a invocare una primitiva tramite una *SVC*, cambiando lo stato di esecuzione come è stato indicato nel primo capitolo, è necessario cambiare anche la funzione di rilocazione, commutando le informazioni presenti nella MMU in modo tale che gli indirizzi vengano tradotti mediante la funzione del sistema. Ciò crea inevitabilmente un rallentamento nell'esecuzione dei programmi dovendo cambiare le tabelle (dei segmenti o delle pagine) e dando quindi luogo a un incremento di interruzioni di *segment* o *page-fault*.

Il secondo problema riguarda un appesantimento nel passaggio di parametri dal processo chiamante alla funzione di sistema chiamata tutte le volte che la funzione richiede un parametro corrispondente a un indirizzo. Per esempio, quando un processo chiama una funzione di sistema per leggere un certo numero di byte da un dispositivo collegato in DMA:

```
n = read(dispositivo, buffer, nbytes)
```

il processo è tenuto a indicare l'indirizzo del buffer in cui i byte letti devono finire. Ovviamente l'indirizzo *buffer*, passato come parametro attuale dal processo chiamante, è un indirizzo virtuale appartenente allo spazio del processo. Il sistema deve però tradurre tale indirizzo virtuale in un indirizzo fisico da utilizzare per inizializzare il canale di DMA e, ovviamente, non potrà tradurlo mediante la propria funzione di traduzione poiché il sistema gira in un diverso spazio virtuale rispetto a quello relativo al processo chiamante. Questo fatto implica un appesantimento della fase di traduzione e ciò rallenta l'esecuzione della funzione di sistema.

Una semplice soluzione a quest'ultimo problema è stata adottata in tutti i sistemi che realizzano spazi virtuali di dimensioni superiori a quelle della memoria fisica (segmentazione a domanda e paginazione a domanda). In questi sistemi lo spazio virtuale di un processo può raggiungere dimensioni molto elevate. Ad esempio, nei sistemi con paginazione a domanda è usuale avere indirizzi virtuali almeno a 32 bit che implicano spazi virtuali di 4 Gbyte. Uno spazio di queste dimensioni è sufficiente a contenere, non solo il codice, i dati e lo stack del processo, ma anche il codice e i dati dell'intero sistema operativo. La soluzione consiste quindi nell'allocare il sistema operativo in tutti gli spazi virtuali in modo tale che sia condiviso tra tutti i processi. Normalmente, ogni spazio virtuale è suddiviso in due metà: quella inferiore

occupata dai programmi e i dati del processo, quella superiore dai programmi e i dati del sistema operativo. Con questa soluzione, all'atto della chiamata di una primitiva di sistema da parte di un processo non viene cambiata la funzione di traduzione degli indirizzi poiché gli indirizzi virtuali del sistema operativo fanno parte dello spazio virtuale del processo, eliminando quindi il precedente problema.

La gestione di spazi virtuali di notevoli dimensioni crea però un altro problema, quello relativo all'enorme dimensione che le singole tabelle (delle pagine o dei segmenti) vengono ad assumere. Nei precedenti paragrafi è stato implicitamente assunto che la tabella delle pagine (o dei segmenti) del processo in esecuzione sia presente in memoria fisica per essere utilizzata durante la traduzione degli indirizzi. Facendo riferimento, ad esempio, ad un sistema paginato con indirizzo virtuale a 32 bit e pagina di 4 kbyte, ciò significa che dei 32 bit dell'indirizzo i 12 meno significativi individuano l'offset nell'ambito della pagina mentre i restanti 20 bit individuano la pagina virtuale. Con 20 bit per la pagina abbiamo quindi la possibilità di indirizzare fino a 2^{20} pagine. Quindi, abbiamo bisogno di una tabella delle pagine di 2^{20} elementi. Se ogni elemento della tabella fosse costituito da 4 byte, l'intera tabella occuperebbe da sola 4 Mbyte di memoria fisica.

Si capisce subito che il vincolo di avere l'intera tabella delle pagine completamente presente in memoria fisica, anche se limitatamente al solo processo in esecuzione, diventa un vincolo inaccettabile.

Negli attuali elaboratori predisposti per operare con ampi spazi virtuali si ricorre, quindi, alla tecnica di paginare anche le tabelle di traduzione degli indirizzi, al fine di allocarle in memoria, esse stesse a domanda, e cioè solo in parte e quando queste parti sono strettamente necessarie.

Con riferimento alla paginazione, ad esempio, si ricorre alla paginazione a più livelli. In figura 4.34 viene illustrato il meccanismo adottato nei microprocessori della famiglia Intel. L'indirizzo virtuale a 32 bit viene suddiviso nei due campi: offset

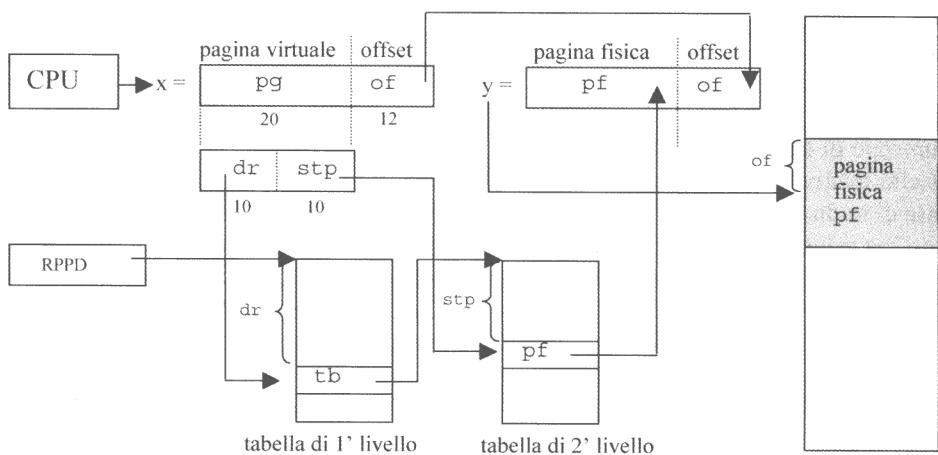


Figura 4.34 Paginazione a più livelli.

nella pagina of (i 12 bit meno significativi), e pagina virtuale pg (i 20 bit più significativi). La tabella delle pagine è però paginata nel seguente modo. Essendo composta da 2^{20} elementi di 4 byte ciascuno (per un totale di 4 Mbyte), tale tabella viene suddivisa in 2^{10} porzioni consecutive. Ciascuna porzione contiene quindi 2^{10} elementi della tabella ed occupa quindi esattamente una pagina fisica di 4 Kbyte (essendo ogni elemento di 4 byte). Le 2^{10} porzioni che compongono la tabella delle pagine possono essere allocate in memoria fisica in modo non contiguo e, solo se necessarie, mediante un'ulteriore tabella di primo livello (detta *page directory*) di 2^{10} elementi, dove ogni elemento contiene (quando il bit di presenza è uguale a uno) l'indirizzo in memoria della corrispondente porzione della tabella delle pagine. Per questo motivo, l'indice pg della pagina virtuale viene a sua volta suddiviso in due campi di 10 bit ciascuno. Il primo campo (dr), corrispondente ai 10 bit più significativi, viene usato per accedere alla tabella di 1° livello (page directory). Questa tabella è l'unica che deve essere garantita in memoria fisica per il processo in esecuzione. Il suo indirizzo è contenuto in un particolare registro di macchina (*PDAR – Page Directory Address Register*). Nell'elemento di indice dr di tale tabella (se il bit di presenza è uguale a uno) si trova l'indirizzo di memoria dove risiede la porzione pg -esima della tabella delle pagine (se il bit di presenza fosse zero si genererebbe un *fault* di tabella delle pagine utilizzato per caricare in una pagina fisica la porzione di tabella richiesta). Una volta caricata in memoria la porzione di tabella delle pagine, ad essa si accede per leggere l'elemento di indice $s_t p$ (scostamento nella tabella delle pagine, corrispondente ai 10 bit meno significativi dell'indice di pagina virtuale). Se in tale elemento il bit di presenza è uguale a uno, si trova l'indice pf della pagina fisica con cui, alla fine, si costruisce l'indirizzo fisico y e quindi si accede alla memoria. Se il bit di presenza è zero, si genera un normale *page-fault*.

Nei sistemi segmentati la tabella dei segmenti viene suddivisa in due sotto-tabelle distinte. Una relativa alla metà superiore dello spazio virtuale, che serve a tradurre gli indirizzi virtuali di sistema e che quindi non viene commutata quando si passa da un processo all'altro. La seconda, corrispondente alla prima metà della spazio virtuale, serve per tradurre gli indirizzi relativi ai segmenti propri del processo in esecuzione. Quest'ultima, in quanto struttura dati del sistema operativo, costituisce essa stessa un segmento nello spazio di sistema. Nei sistemi segmentati e paginati, è poi allocata in memoria fisica in maniera paginata.

Infine, si può mettere in evidenza un'ultimo problema relativo ai sistemi con rilocazione dinamica. In questi sistemi la commutazione della CPU da un processo all'altro è sicuramente più costosa rispetto agli altri sistemi. Infatti, oltre a comportare tutte le operazioni viste nel secondo capitolo, è necessario commutare anche la MMU e ciò comporta un maggior tempo di commutazione. Inoltre, quando un nuovo processo viene schedulato, tutti i registri associativi del *look aside buffer* devono essere invalidati poiché contendono informazioni relative alla funzione di traduzione degli indirizzi del precedente processo, e ciò comporta un aumento degli accessi alle tabelle, riducendo di conseguenza la velocità di accesso alle informazioni. Questo aspetto costituisce un ulteriore motivo per cui sono stati introdotti i thread. Infatti, poiché i thread di uno stesso processo girano tutti nello stesso spazio virtuale, una commutazione di contesto tra thread è molto più efficiente che non una commutazione di contesto fra processi.

4.4 Sommario

La gestione della memoria possiede alcune analogie con la gestione della CPU. In particolare, l'aspetto che accomuna la gestione di queste due risorse è la tecnica della virtualizzazione. In entrambe i casi, per ogni processo, il sistema operativo crea le rispettive risorse virtuali, allocando le corrispondenti risorse reali al fine di fornire il necessario supporto alle risorse virtuali. Partendo da questa analogia, in questo capitolo sono state evidenziate le peculiarità della gestione della memoria nei confronti di quella, precedentemente vista, della CPU. In particolare è stato mostrato come la gestione della memoria sia caratterizzata da quattro parametri fondamentali:

- la tecnica di rilocazione (statica o dinamica);
- il criterio di allocazione della memoria fisica (allocazione contigua o non contigua);
- l'organizzazione della memoria virtuale (unica o segmentata);
- il criterio di caricamento di un processo in memoria (caricamento “tutto insieme” o “a domanda”).

In base ai diversi valori dei precedenti parametri sono state illustrate le varie tecniche di allocazione della memoria principale utilizzate dal sistema operativo esaminando, per ciascuna di esse, i meccanismi hardware di supporto (MMU), i meccanismi per la protezione delle memorie e i criteri per la condivisione delle informazioni tra processi diversi.

4.5 Note bibliografiche

Per approfondire gli argomenti relativi al linker e al caricatore si può fare riferimento a vari testi sui sistemi di programmazione, ad esempio [19].

Un'approfondita trattazione delle tecniche delle partizioni si può trovare in [64], mentre in [56] si trova una completa descrizione delle varie tecniche di allocazione delle partizioni.

La segmentazione è stata introdotta nel MULTICS [69] e discussa, per primo, da [27]. La paginazione è stata introdotta nel sistema operativo Atlas [45] e [54]. I lavori di Belady hanno costituito delle pietre miliari sugli algoritmi di rimpiazzamento [7] e [8]. Una completa descrizione del clock algorithm si trova in [18]. Per un approfondimento sugli algoritmi di rimpiazzamento in generale si rimanda ad altri testi sui sistemi operativi [80], [85], [96]. In [28] viene introdotto il concetto di *working set*.

Ottimi lavori di rassegna sulle memorie virtuali sono costituiti dal classico lavoro di Denning [29] e quello di Jacob e Mudge [50].

In [50] vengono discussi i meccanismi hardware presenti nei moderni microprocessori a supporto della gestione della memoria. Per approfondire gli aspetti hardware relativi ai Translation Lookaside Buffers e, in generale, ai meccanismi presenti nelle MMU si può fare riferimento a [38].

5

Gestione delle periferiche (I/O)

Da un punto di vista macroscopico, le attività svolte da un calcolatore possono essere classificate in due distinte categorie: da un lato le attività d'ingresso/uscita (I/O) relative al trasferimento nel sistema dei dati da elaborare e alla restituzione dei risultati della elaborazione, dall'altro le attività di elaborazione svolte dalla CPU. I processi applicativi vengono spesso classificati anche in base alla prevalenza di un tipo di attività rispetto all'altro. Si parla di processi *CPU-bound* se le attività che questi svolgono sono prevalentemente attività di elaborazione dati. Un classico esempio di processi di questo tipo è quello relativo a una complessa elaborazione di dati scientifici.

Se le operazioni svolte da un processo sono, viceversa, prevalentemente operazioni d'ingresso/uscita si parla di processi *I/O-bound*. Appartengono a questa categoria, ad esempio, processi che svolgono attività di editing di un file di testo o dedicati ad eseguire attività di *navigazione su web*.

In questo capitolo vengono descritti i principali servizi offerti dal sistema operativo per assicurare ai processi un adeguato supporto all'esecuzione delle operazioni d'ingresso/uscita. Tale supporto è teso a garantire un uso semplice, efficiente e sicuro dei dispositivi periferici che sono deputati all'implementazione delle operazioni d'ingresso/uscita. Per questo motivo, il sottosistema di I/O del sistema operativo è destinato a fornire un'interfaccia uniforme che, da un lato ha il compito di mascherare le peculiarità dei singoli dispositivi d'ingresso/uscita, dall'altro deve garantire l'accesso ai dispositivi nel modo più efficiente possibile gestendo anche eventuali malfunzionamenti che si possono verificare durante un trasferimento di dati.

Per prima cosa verranno brevemente richiamate alcune caratteristiche relative all'architettura hardware di interconnessione tra dispositivi periferici, CPU e memoria (interfaccia hardware) per poi passare a descrivere le singole componenti del sottosistema di I/O che è preposto all'implementazione dell'interfaccia uniforme fornita ai processi applicativi.

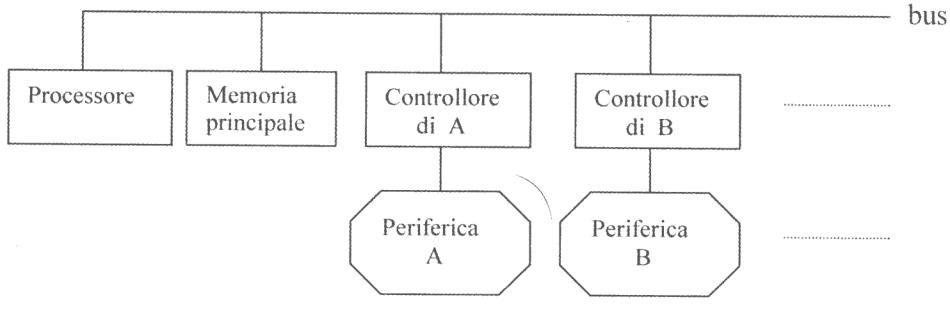


Figura 5.1 Schema architetturale semplificato di un calcolatore.

5.1 Concetti generali

La necessità di trasferire da un supporto esterno alla memoria principale i dati da elaborare e, viceversa, di trasferire dalla memoria all'esterno del sistema i risultati di una elaborazione, implica per un processo la necessità di eseguire, oltre alle normali operazioni di elaborazione, anche operazioni d'ingresso/uscita. Per questo motivo ogni processore dispone di un particolare insieme di istruzioni macchina deputate al trasferimento di dati tra i registri interni al processore e i dispositivi periferici (istruzioni d'ingresso/uscita).

Richiamando brevemente alcune semplici nozioni di architettura dei calcolatori, possiamo rappresentare in figura 5.1 lo schema architetturale semplificato di un calcolatore, dove, come è noto, ogni dispositivo periferico è collegato al bus del sistema (e quindi al processore) tramite un circuito di interfaccia, spesso indicato come *controllore* del dispositivo. Compito del controllore è appunto quello di controllare il funzionamento del dispositivo esterno (trasduttore) inviando e ricevendo dallo stesso segnali mediante un ben definito protocollo. Al tempo stesso, è collegato tramite il bus con il processore al quale si presenta come una piccola memoria costituita da un insieme di registri ciascuno dei quali è caratterizzato da un proprio indirizzo appartenente allo spazio di I/O. È questo lo spazio che viene riferito durante l'esecuzione delle istruzioni appartenenti a quel particolare sottoinsieme noto appunto col nome di istruzioni d'ingresso/uscita.

Volendo programmare direttamente in linguaggio assembler un trasferimento di dati da, o verso, un particolare dispositivo periferico sarebbe quindi necessario conoscere, in tutti i dettagli, l'insieme dei registri contenuti nel controllore del dispositivo al fine di controllarne il funzionamento secondo le specifiche necessità.

Un primo compito del sottosistema di I/O è proprio quello di evitare che il programmatore debba conoscere tutti i dettagli delle interfacce hardware, spesso molto complesse e diverse tra loro, fornendo delle astrazioni dei dispositivi periferici. Ciascuna periferica viene quindi vista come una risorsa astratta alla quale sia possibile accedere tramite un insieme di funzioni. L'insieme di tutte le funzioni di accesso alle periferiche costituisce l'interfaccia di programmazione applicativa (*I/O API - Input/Output API*).

Output Application Programming Interface) che i singoli processi possono utilizzare per le loro esigenze di trasferimento dati da e verso unità periferiche.

Pur essendo molto ampia la gamma di dispositivi periferici, un secondo obiettivo, strettamente correlato col precedente, riguarda la necessità di omogeneizzare, a livello di *API*, le funzioni di accesso ai singoli dispositivi, cercando di nascondere le loro singole specificità all'interno delle astrazioni che li rappresentano. I dispositivi possono essere classificati in diverse categorie in base a vari criteri. Per esempio, con riferimento alla sorgente o alla destinazione dei dati a cui sono collegati, possiamo distinguere tre categorie di dispositivi: quelli destinati a collegare il sistema con l'operatore umano (terminali video, tastiera, mouse, stampanti ecc.), quelli destinati a collegare il sistema con altre apparecchiature (sensori, attuatori, dischi, nastri ecc.), quelli destinati a collegare il sistema con apparecchiature o utenti remoti (interfacce di rete, modem ecc.).

Le velocità con cui i dispositivi trasmettono o ricevono i dati sono molto diverse tra di loro, anche di molti ordini di grandezza. Nella seguente tabella viene fornito, come esempio, un elenco di dispositivi con le relative velocità di trasferimento dati:

dispositivo	velocità di trasferimento
tastiera	10 bytes/sec.
mouse	100 bytes/sec.
modem	10 Kbytes/sec.
linea ISDN	16 Kbytes/sec.
stampante laser	100 Kbytes/sec.
scanner	400 Kbytes/sec.
porta USB (Universal Serial Bus)	1.5 Mbytes/sec.
disco IDE	5 Mbytes/sec.
CD-ROM	6 Mbytes/sec.
Fast Ethernet	12.5 Mbytes/sec.
FireWire (IEEE 1394)	50 Mbytes/sec.
monitor XGA	60 Mbytes/sec.
Ethernet gigabit	125 Mbytes/sec.
Sun Gigaplane XP backplane	20 Gbytes/sec.

Tabella 5.1 Esempi di velocità di trasferimento dati.

Diversi sono anche il numero di funzioni che un dispositivo può svolgere e la quantità di errori o malfunzionamenti che si possono originare durante il loro funzionamento.

Fra le varie classificazioni dei dispositivi periferici, una particolarmente utile ai fini della loro gestione riguarda il modo in cui sono organizzati i dati da trasferire. Da questo punto di vista possiamo distinguere i dispositivi in *dispositivi a blocchi*, *dispositivi a carattere* e *dispositivi speciali*. Nella prima categoria rientrano tutti i dispositivi di memoria di massa (dischi, nastri ecc.) che registrano i dati in blocchi di dimensione fissa. Ogni blocco è caratterizzato da un proprio indirizzo e può essere letto o scritto l'uno indipendentemente dagli altri. Viceversa, i dispositivi a carattere non prevedono nessuna strutturazione interna ed accettano, o inviano, stringhe di caratteri senza poterli indirizzare al loro interno. Appartengono a questa categoria le

tastiere e i *mouse*, le stampanti, le interfacce di rete ecc. Vi sono poi alcuni dispositivi molto particolari che non rientrano in nessuna di queste due categorie e, come tali, vengono indicati come dispositivi speciali. Un tipico esempio è costituito dal dispositivo di temporizzazione (*timer*) che non accetta né blocchi di dati né singoli caratteri, ma si limita a generare interruzioni a istanti programmati.

Indipendentemente da tutta questa variabilità di dispositivi, l'obiettivo è quello di fornire ai processi applicativi un insieme di funzioni generiche come `read` e `write`, `open` e `close` ecc. Quando, ad esempio, un processo deve leggere il record di un file è opportuno che il codice del programma che il processo esegue non cambi al variare del supporto su cui il file risiede, e quindi dello specifico dispositivo coinvolto.

Un'ulteriore necessità riguarda la gestione dei malfunzionamenti che si possono verificare durante un trasferimento di dati. È compito del sistema fornire i gestori di tutte le eventuali eccezioni che possono essere sollevate durante un trasferimento. Alcuni di questi eventi eccezionali possono essere direttamente risolti dal sistema che è in grado di recuperare il corretto funzionamento (per esempio ripetendo l'operazione in caso di un malfunzionamento non permanente). In altri casi è necessario sollevare l'eccezione al processo che ha invocato l'operazione di I/O, o rendere evidente l'evento con un messaggio sulla consolle, come nel caso di mancanza di carta sulla stampante.

Un altro compito che viene delegato al sottosistema di I/O è quello di *naming*, cioè quello relativo alla definizione di uno *spazio dei nomi* tramite i quali i singoli dispositivi (e i file su essi residenti) vengono identificati dai processi che su essi devono operare. Per semplificare l'interfaccia di programmazione, viene consentito ai processi di identificare file e dispositivi di I/O mediante nomi simbolici mentre a livello di sistema ogni entità, ivi inclusi file e dispositivi, vengono identificati in modo univoco mediante numeri interi. Tali interi costituiscono, spesso, indici in un array di puntatori a strutture dati (i descrittori) che rappresentano in memoria i dispositivi stessi.

Con l'obiettivo di nascondere ai processi applicativi le peculiarità hardware dei singoli dispositivi il sottosistema di I/O fornisce anche un servizio di *bufferizzazione* dei dati da leggere, o da inviare in uscita. Per quanto riguarda i dispositivi a carattere ciò è necessario tenendo conto dell'enorme differenza fra la velocità con cui l'elaboratore produce i dati da inviare in uscita (o può elaborare i dati da leggere) rispetto alla velocità con cui operano i dispositivi di I/O. Per quanto riguarda, invece, i dispositivi a blocchi, l'esigenza nasce dal fatto che la struttura hardware di un dispositivo di memoria di massa prevede, normalmente, la lettura o la scrittura di un numero intero di blocchi fisici, mentre un processo applicativo può essere interessato a trasferire in ingresso o in uscita soltanto un numero limitato di bytes. In questo senso, le funzioni di I/O chiamate dai processi applicativi per trasferire dati in uscita inseriscono tali dati nel buffer offerto dal sistema; soltanto quando il buffer è completo questo viene trasferito su memoria di massa. Analogamente, in lettura, i dati vengono trasferiti nel buffer da cui, successivamente, il processo estrae i dati richiesti.

Come già indicato, tutte le precedenti funzionalità vengono offerte dal sottosistema di I/O omogeneizzando, per quanto possibile, le funzioni di accesso ai singoli dispositivi indipendentemente dalle loro peculiarità. È compito di un'altra componente del sottosistema di I/O, la componente *device-dependent*, nascondere le peculiarità

dei dispositivi, interfacciandosi direttamente con i corrispondenti controllori hardware e tenendo quindi conto di tutti i dettagli necessari per controllare il loro funzionamento. In pratica, per ogni dispositivo, in certi casi per un insieme di dispositivi omogenei, esiste un componente dedicato alla sua gestione: il *device driver*.

Un processo applicativo invoca le funzioni fornite dall'interfaccia di programmazione. Queste, a loro volta, invocano le funzioni del driver del dispositivo coinvolto affinché, operando direttamente sui registri del corrispondente controllore, ne controlli il funzionamento fino al completamento della funzione richiesta.

Infine, quando un processo applicativo attiva un trasferimento di dati, normalmente deve attendere il suo completamento per poter proseguire. Tenendo conto della differenza di velocità tra il processore e un qualunque dispositivo di I/O, normalmente molto più lento, il tempo di attesa può essere anche molto lungo. In un sistema multiprogrammato, come è ben noto, quando un processo deve attendere che si verifichi un evento particolare è necessario sospenderlo e, con un cambio di contesto, attivare un altro processo pronto per essere poi successivamente riattivato quando l'evento si sarà verificato. In questo caso l'evento in questione è il completamento del trasferimento di dati che il dispositivo può segnalare via hardware mediante un segnale d'interruzione. Faranno quindi parte del sottosistema di I/O anche tutte le funzioni di risposta alle interruzioni generate dai dispositivi periferici.

5.2 Organizzazione logica del sottosistema di I/O

La quantità e la complessità delle diverse funzioni che il sottosistema di I/O deve svolgere suggeriscono di organizzare le sue componenti adottando lo stesso approccio tipico di tutti i sistemi complessi e cioè, facendo ricorso alle tecniche di astrazione e di strutturazione a livelli gerarchici.

Un esempio di come viene normalmente strutturato il sottosistema di I/O è riportato nella successiva figura 5.2, dove sono stati messi in evidenza i due principali livelli di software che lo compongono e le diverse interfacce che gli stessi generano. In figura, oltre ai due livelli che caratterizzano il sottosistema di I/O (livello *device independent* e livello dipendente dalle caratteristiche dei singoli dispositivi), sono disegnati anche i due ulteriori livelli coinvolti in ogni trasferimento di I/O: il livello utente, nel quale girano i processi applicativi che, direttamente o tramite le librerie di I/O tipiche di ogni linguaggio di programmazione, invocano le funzioni offerte a livello d'interfaccia applicativa dal sottosistema di I/O per attivare operazioni trasferimento di dati da, o verso, un dispositivo. Analogamente, viene messo in evidenza il livello hardware, proprio dei dispositivi con i loro controllori, che forniscono l'interfaccia di basso livello tramite la quale il processore può interagire con i dispositivi stessi. Tale interfaccia permette di operare direttamente sui registri di ogni controllore mediante le istruzioni d'ingresso/uscita.

5.2.1 Livello indipendente dai dispositivi

Come già indicato nel paragrafo introduttivo, alcune funzioni del sottosistema di I/O sono indipendenti dai particolari dispositivi e, come tali, fanno parte di quel componente del sottosistema che ha il compito, da un lato, di definire l'interfaccia applica-

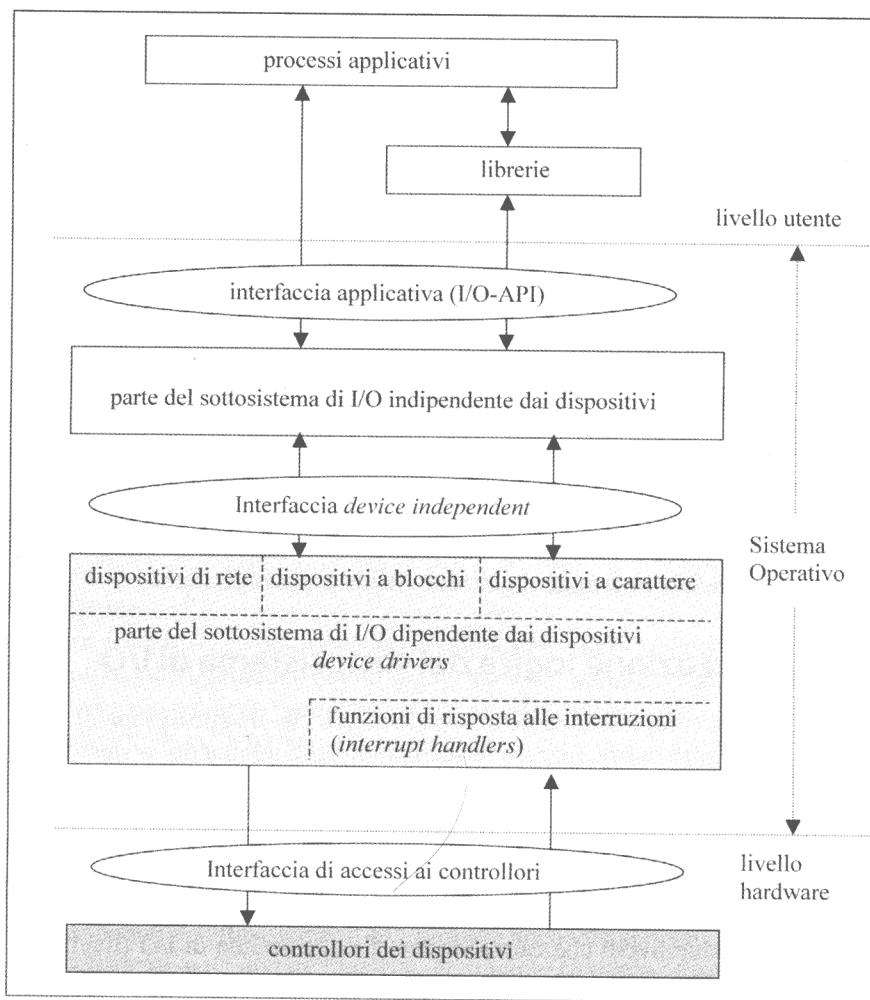


Figura 5.2 Organizzazione logica del sottosistema di I/O.

tiva d'ingresso/uscita, e cioè di fornire l'insieme di tutte le chiamate di sistema dedicate ai trasferimenti di I/O, dall'altro di interfacciarsi con la componente *device dependent*, e per il suo tramite, di operare sui dispositivi.

I servizi offerti da questo livello hanno il compito di rendere più semplice, sicuro ed efficiente l'uso dei dispositivi e variano molto da sistema a sistema, ma alcune di queste sono normalmente disponibili nella maggior parte dei sistemi operativi e spesso coincidono con alcune funzioni del file system essendo uniformata, a livello di interfaccia applicativa, la gestione dei file e quella dei dispositivi periferici. Per questo motivo alcune delle funzioni di questo livello saranno esaminate con maggiori dettagli nel capitolo 6 dedicato al file system.

Alcune delle funzioni che non dipendono dal particolare dispositivo, e che quindi vengono realizzate in questo livello, riguardano le funzioni di: *naming* dei file e dei dispositivi, di protezione, di disaccoppiamento tra i registri del controllore e la memoria virtuale del processo applicativo che richiede un trasferimento dati (*buffering*), di gestione degli eventi eccezionali non gestiti a livelli di driver, di allocazione dinamica dei dispositivi e di *spooling*.

La funzione di *naming* dei dispositivi ha il compito di consentire l'identificazione di ogni dispositivo mediante un nome simbolico e quindi di mettere in corrispondenza tale nome con il corrispondente driver che gestisce il dispositivo. Per quanto riguarda questa funzione, viene spesso utilizzato il criterio di inserire i nomi simbolici dei dispositivi all'interno dello spazio dei nomi del *file system*. In pratica un dispositivo periferico è visto come un file di tipo speciale. Per questo motivo, anche tutti i meccanismi di protezione e controllo degli accessi sono gli stessi utilizzati per controllare gli accessi ai file. Rimandiamo, quindi, al successivo capitolo il dettaglio degli aspetti relativi a queste funzionalità.

Buffering Una delle funzioni più importanti offerte dal livello di software indipendente dai dispositivi è quella di fornire, per ogni operazione di trasferimento dati, e in modo del tutto trasparente per il processo applicativo, una o più aree di memoria tampone (*buffer di sistema*) destinate a contenere i dati durante il trasferimento fra il dispositivo e l'area di memoria virtuale del processo applicativo da cui i dati devono essere prelevati, nel caso di un trasferimento in uscita, o in cui devono essere inseriti se il trasferimento è d'ingresso.

Le ragioni che inducono a riservare questi *buffer* nella memoria del sistema sono molteplici. Prima fra tutte, come in ogni problema di tipo produttore/consumatore, è quella di far fronte alla notevole differenza fra la velocità con cui il processore può produrre (o consumare) informazioni e quella con cui un dispositivo può consumarle (o produrle). Per meglio comprendere queste ragioni distinguiamo il caso di operazioni con un dispositivo a blocchi (disco) da quello di operazioni con un dispositivo a carattere (stampante o tastiera).

Nel primo caso, i dati vengono trasferiti in blocchi di dimensione fissa. Supponiamo, ad esempio, che un processo invochi una chiamata di sistema per leggere un blocco di dati. Una tipica operazione di lettura potrebbe essere la seguente:

```
n = read(fd, ubuf, nbytes);
```

dove *fd* è il nome simbolico del file da cui leggere, *ubuf* rappresenta l'indirizzo del buffer, nella memoria virtuale del processo chiamante, in cui trasferire i dati richiesti, *nbytes* rappresenta il numero di bytes da trasferire e *n* il valore restituito dalla funzione relativo ai bytes realmente trasferiti. Normalmente, il processo applicativo, una volta invocata la funzione di lettura deve sospendere la propria esecuzione fino a quando i dati richiesti sono arrivati nell'area *ubuf* destinata a contenerli. Se la funzione *read* viene realizzata facendo ricorso a un buffer di sistema come indicato nella seguente figura 5.3, l'operazione di lettura trasferisce i dati dal disco al buffer e quindi da questo all'area *ubuf* dello spazio utente. A trasferimento avvenuto il processo applicativo può riprendere la sua esecuzione. Il vantaggio di questa organizzazione si rileva quando, come normalmente avviene, il processo deve leggere ed elaborare non un solo blocco di dati ma una sequenza di blocchi. In questo caso,

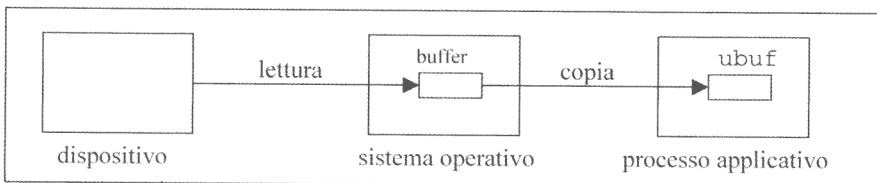


Figura 5.3 Operazione di lettura con singolo buffer.

il tempo di attesa del processo, e il numero di commutazioni di contesto, può essere ridotto se l'elaborazione di un blocco di dati da parte del processo viene eseguita in parallelo alla lettura nel buffer del successivo blocco di dati (*read ahead*). Un ulteriore livello di parallelismo può essere ottenuto utilizzando la tecnica del doppio buffer (vedi figura 5.4) che consente di eseguire in concorrenza con la lettura di un blocco il trasferimento del blocco precedente dal buffer di sistema all'area *ubuf* di utente e la sua successiva elaborazione da parte del processo.

Un livello di disaccoppiamento ancora maggiore può essere ottenuto utilizzando un array circolare di buffers e adottando la stessa soluzione già vista per il classico problema produttore/consumatore.

Le stesse considerazioni possono essere ovviamente fatte nell'ipotesi di un'operazione di scrittura.

Un secondo vantaggio offerto dalla tecnica di *bufferizzazione* riguarda il fatto che un processo applicativo, mediante la funzione `read`, può richiedere il trasferimento di un numero di bytes anche inferiore a quello di un blocco di dati. In questo caso viene comunque letto nel buffer di sistema l'intero blocco, salvo trasferire successivamente nell'area *ubuf* di utente soltanto i bytes richiesti.

Nel caso di operazioni con dispositivi a carattere, normalmente i dati vengono trasferiti come stringhe di bytes, per esempio una linea di testo da inviare alla stampante. Anche in questo caso l'uso di buffers di sistema consente di far fronte alle diverse velocità fra il produttore e il consumatore dei dati da trasferire. Consente inoltre di gestire automaticamente, nel buffer di sistema, la cancellazione di caratteri letti da tastiera, ma successivamente cancellati tramite il tasto *backspace* prima di battere il tasto *invio*, e ciò in maniera del tutto trasparente per il processo applicativo.

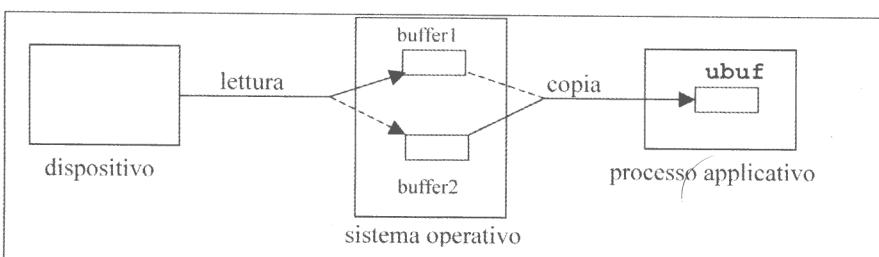


Figura 5.4 Operazione di lettura con doppio buffer.

Il fatto che un dispositivo inserisca, o prelevi, i dati soltanto da un buffer di sistema, consente anche di semplificare la gestione delle pagine fisiche in un sistema con gestione paginata della memoria principale. Infatti, in assenza di buffer di sistema il dispositivo dovrebbe operare direttamente con le pagine fisiche contenenti l'area di memoria virtuale *ubuf* del processo applicativo. Poiché, normalmente il processo, una volta invocata l'operazione *read* o *write*, si sospende in attesa del completamento dell'operazione, le sue pagine virtuali potrebbero essere rimpiazzate in seguito a *page faults* generati da altri processi. Per evitare che ciò accada mentre le pagine sono soggette ad un trasferimento di dati, sarebbe necessario bloccarle in memoria tramite il meccanismo di *lock* visto nel precedente capitolo.

Infine, i buffers di sistema possono essere utilizzati anche come una *cache* di blocchi nei confronti di una memoria di massa. Per esempio, quando un file viene condiviso tra più processi applicativi la lettura di un blocco prevede, per prima cosa, la verifica che il blocco non sia già presente nella cache in quanto già letto precedentemente da un altro processo. In questo caso può essere evitato un trasferimento da disco.

Gestione degli errori, dei malfunzionamenti e degli eventi eccezionali Durante un'operazione d'ingresso/uscita si possono verificare molti eventi anomali. Alcuni di questi possono essere dei malfunzionamenti originati da guasti hardware che si possono verificare nei dispositivi periferici o nei relativi controllori come, ad esempio, la rottura di una testina di lettura/scrittura di un disco o l'inceppamento della carta in una stampante laser. In altri casi l'evento anomalo, pur se non classificabile come malfunzionamento, costituisce comunque un evento eccezionale che non consente, se non opportunamente gestito, il corretto completamento dell'operazione. Ad esempio, in fase di stampa la mancanza della carta nel vassoi della stampante o, in fase di lettura di un file, la lettura della marca di fine file (*End-Of-File*). Altri casi anomali possono essere originati da semplici errori commessi in fase di programmazione come, ad esempio, il tentativo di operare su un dispositivo non connesso al calcolatore, o semplicemente spento, oppure il tentativo di aprire un file inesistente.

In tutti i precedenti casi è necessario che i singoli eventi anomali, di qualunque natura essi siano, vengano opportunamente gestiti affinché i processi applicativi possano essere messi nella condizione di completare correttamente la loro esecuzione, se possibile, oppure di eseguire rami di programma alternativi, nell'ipotesi di malfunzionamenti non recuperabili.

Il compito di gestire gli eventi anomali non è limitato ad una sola componente del sottosistema di I/O ma è svolto a diversi livelli da ciascuna delle sue componenti e in parte anche dal software applicativo. In genere è buona norma gestire gli eventi anomali il più vicino possibile a dove si verificano, propagando un'eccezione ad un livello superiore soltanto nel caso in cui non sia stato possibile gestire completamente l'evento. Per esempio, numerosi malfunzionamenti rilevati a livello di controllore di un dispositivo sono dovuti a errori transitori come nel caso di un errore in lettura da disco causato dalla presenza di polvere sulla testina oppure causato da un disturbo elettromagnetico e rilevato dal controllo di parità o di *checksum*. In questi casi è il driver del dispositivo che, rilevato l'evento, può prendersi cura di recuperare il

corretto funzionamento, ad esempio ripetendo semplicemente l'operazione in modo tale che ai livelli superiori di software l'errore sia completamente mascherato e tutto avvenga come se niente di anomalo fosse accaduto. Nei casi in cui il driver, pur rilevando l'evento, non è in grado di risolvere completamente il problema, viene sollevata un'eccezione a livello superiore. Per esempio, nel caso in cui il driver rilevi un errore in fase di lettura da disco l'operazione viene ripetuta. Ma se l'evento si ripresenta costantemente per un certo numero prestabilito di volte, evidentemente ciò è indice di un malfunzionamento permanente e come tale non recuperabile a livello del driver che propaga quindi l'evento al livello superiore.

Anche a livello di software indipendente dai dispositivi esistono quindi i gestori di eventi anomali, in particolare di tutti quelli che si verificano a questo livello oltre a quelli propagati dal livello inferiore. E anche in questo caso si cerca, se possibile, di gestire completamente tali eventi mascherandone gli effetti al livello applicativo. Solo se ciò non è possibile l'evento viene propagato mediante un'eccezione sollevata a livello di software applicativo. Per esempio, nel caso in cui siano presenti più stampanti, di cui una sia quella usata prioritariamente (stampante di *default*), se durante l'esecuzione della funzione di stampa viene rilevato dal driver l'evento '*stampante non collegata*', tale evento viene propagato al livello superiore che, in questo caso, può risolvere completamente il problema tentando di ripetere l'operazione di stampa su una stampante alternativa. In altri casi l'evento anomalo può essere risolto, mascherandone quindi gli effetti ai livelli superiori, solo dopo l'intervento umano, come nel caso di mancanza della carta nel vassoio della stampante. In questo caso è sufficiente l'invio di un messaggio sulla consolle del sistema rimanendo in attesa che la condizione anomala sia eliminata. Infine, quando l'errore non è recuperabile è necessario propagare l'evento a livello applicativo. Sarà il processo che dovrà prendere in considerazione quale alternativa eseguire. Da questo punto di vista, in molti casi è previsto che le chiamate al sistema restituiscano al processo chiamante un eventuale codice di errore.

Allocazione dei dispositivi e tecniche di spooling Come avviene per altri tipi di risorse, anche i dispositivi periferici possono essere dedicati ad un processo alla volta o essere condivisi tra più processi contemporaneamente. Per esempio, un'unità a disco può essere condivisa tra più processi, tipicamente quando gli stessi aprono dei file residenti sullo stesso disco. In altri casi, ad esempio nel caso di una stampante, il dispositivo deve essere allocato ad un processo alla volta se vogliamo evitare che le stampe di uno vengano intercalate con le stampe degli altri. In questo caso, compito del livello *device independent* è anche quello di fornire i gestori dei dispositivi che dinamicamente devono essere allocati ad un processo alla volta. Ciò presuppone che un processo applicativo operi seguendo il tradizionale protocollo che prevede la richiesta al gestore e la relativa acquisizione in modo esclusivo del dispositivo prima di poterci operare, e la successiva operazione di rilascio quando l'intera sequenza di operazioni è terminata.

Una tecnica spesso utilizzata in alternativa all'allocazione dinamica di un dispositivo, dedicandolo ad un processo alla volta, è quella di *spooling* il cui scopo è quello di ridurre i tempi di attesa per un processo applicativo che, nel richiedere l'uso esclusivo del dispositivo, e trovandolo già occupato, deve attendere il suo rilascio

per un tempo non facilmente prevedibile. Tale tecnica è spesso adottata con risorse come la stampante. In questo caso il processo applicativo non opera direttamente sulla stampante ma su una sua copia virtuale, e privata del processo, realizzata mediante un file su disco. Il processo non è quindi tenuto a richiedere l'uso della sua stampante che è sempre disponibile per lui. Le operazioni di stampa creano un file su disco che, successivamente verrà letto da un particolare processo di sistema (*daemon*), l'unico ad avere accesso alla stampante reale e quindi inviato realmente in stampa. Dinamicamente si viene quindi a creare una lista di *file di spool* in attesa di essere serviti e stampati dal processo di sistema uno alla volta. La coda dei file già creati e in attesa di essere stampati è sempre disponibile all'utente (sistemista) che, in ogni istante può riordinare tale coda o, in casi eccezionali, rimuovere manualmente alcuni dei file in attesa.

Strettamente connessa alle funzioni di allocazione dei dispositivi è anche la funzione di ordinamento (*scheduling*) dei trasferimenti fra settori di un disco e i buffer nella memoria del sistema. Come indicato precedentemente, un disco è normalmente condiviso tra più processi, il che ovviamente non significa che più operazioni di lettura e/o scrittura possano avvenire contemporaneamente, ma solo che fra due successive operazioni invocate da un processo può essere eseguita l'operazione invocata da un altro. Dinamicamente si viene quindi a generare una coda di richieste di trasferimento di settori, da, o verso, il disco relative ad altrettante operazioni invocate da processi diversi ma che devono esser eseguite una alla volta anche se in un qualunque ordine. Nasce quindi il problema di come ordinare le esecuzioni di tutti i trasferimenti pendenti.

Le politiche di schedulazione più ovvie possono essere quella FIFO, che ha il pregio della semplicità e anche quello di evitare ogni forma di *starvation*, oppure quella prioritaria, legata alla priorità dei processi che hanno richiesto i trasferimenti. Spesso, però, tenendo conto delle caratteristiche hardware dei dispositivi si preferisce realizzare una diversa politica di schedulazione. Infatti il tempo medio di accesso al disco dipende fondamentalmente da due parametri: il primo, e più significativo, è il cosiddetto tempo di *seek* che corrisponde all'intervallo temporale necessario per spostare la testina di lettura/scrittura dalla traccia su cui si trova a quella su cui è richiesto di operare. Questo tempo di spostamento meccanico della testina è di alcuni ordini di grandezza superiore a quello di effettivo trasferimento dei dati. Vi è poi da attendere anche il tempo di rotazione necessario affinché il settore su cui operare passi sotto la testina di lettura/scrittura. Tenendo conto di queste considerazioni, al fine di ridurre il tempo medio di accesso, è necessario ridurre il tempo di seek. Per questo motivo viene in genere preferito un algoritmo di schedulazione che, istante per istante, privilegia il trasferimento che coinvolge una delle tracce più vicine a quella su cui è posizionata la testina di lettura/scrittura.

5.2.2 Livello dipendente dai dispositivi

È compito del livello inferiore del sottosistema di I/O nascondere al proprio interno tutti i dettagli relativi ai singoli dispositivi e ai relativi controllori definendo un'interfaccia (vedi figura 5.2) che consente alle funzioni del livello superiore di operare sui dispositivi astraendo da tali dettagli (interfaccia *device-independent*). È all'interno di questo livello di software che vengono definite tutte le astrazioni (classi) che rap-

presentano le varie tipologie di dispositivi. In pratica ogni dispositivo fisico viene rappresentato mediante una struttura dati (*descrittore di periferica*) a cui è possibile accedere mediante un insieme di funzioni (funzioni membro della classe) che fanno parte della precedente interfaccia. È compito di ognuna di queste funzioni inviare gli opportuni comandi ai registri del controllore del dispositivo mediante le istruzioni di I/O, e coordinare così le operazioni del dispositivo fisico al fine di completare la corretta esecuzione della funzione stessa.

In realtà, tenendo conto della grande variabilità dei dispositivi periferici e, soprattutto del loro diverso modo di operare, l'interfaccia uniforme fornita al livello superiore prevede spesso la distinzione tra le funzioni di accesso ai dispositivi distinguendoli in base alla categoria a cui appartengono. Per esempio è abbastanza usuale distinguere tra le funzioni di accesso ai dispositivi a blocchi rispetto a quelle per accedere ai dispositivi a carattere, e spesso anche nei confronti dei dispositivi di rete (vedi figura 5.2). Sono infatti diverse, anche a livello applicativo, le funzioni che un processo può eseguire su un dispositivo a blocchi (*read*, *write*, *seek*, ecc.) rispetto a quelle relative ad un dispositivo a carattere (*get* o *put*). Anche i dispositivi di rete si distinguono da questo punto di vista poiché la maggior parte dei sistemi operativi offre un'interfaccia corrispondente ai *socket* che verranno illustrati nell'appendice A.

Le funzioni offerte al livello di interfaccia *device-independent* sono praticamente le stesse offerte a livelli di interfaccia applicativa. Ad esempio per quanto riguarda le due tipiche funzioni di accesso in lettura e scrittura su disco, queste possono presentarsi come:

```
n = read (dispositivo, buffer, nbytes);
n = write (dispositivo, buffer, nbytes);
```

dove, rispetto alle corrispondenti funzioni dell'interfaccia applicativa è diverso il modo di individuare il dispositivo (non più mediante nomi simbolici) e l'area di memoria interessata che in questo caso è un *buffer* di sistema e non l'area *ubuf* indicata dal processo applicativo.

Le precedenti funzioni vengono realizzate andando ad operare direttamente sui registri del corrispondente controllore tramite l'interfaccia di basso livello con i seguenti obiettivi:

- a)** attivare il dispositivo affinché esegua l'operazione richiesta;
- b)** controllore che durante l'operazione non si verifichino eventi anomali. Se ciò accade verificare la possibilità di gestire localmente l'evento mascherandone le conseguenze o, in caso contrario, interrompere l'operazione e sollevare un'eccezione a livello superiore;
- c)** sincronizzare il processo applicativo che ha richiesto l'operazione con la fine dell'operazione stessa;
- d)** disattivare il dispositivo alla fine dell'operazione.

L'insieme delle funzioni di accesso a un dispositivo fisico, realizzate secondo questi criteri, costituisce quel componente del sottosistema di I/O che è dedicato alla gestione del dispositivo e che è noto col termine di *device driver*. Nel successivo paragrafo verranno illustrati in dettaglio alcuni esempi di realizzazione di un driver.

Con riferimento al precedente punto c) è opportuno notare che le azioni svolte da un dispositivo sono del tutto concorrenti e asincrone con le operazioni svolte dal processore. In effetti, l'insieme del dispositivo col proprio controllore, può essere considerato come un vero e proprio processore *special purpose* dedicato a svolgere solo operazioni particolari e in modo autonomo dalla CPU una volta attivato. È quindi possibile scrivere un programma che, una volta eseguita l'operazione di attivazione di un dispositivo, continua la sua esecuzione in maniera del tutto asincrona con l'esecuzione delle azioni fisiche svolte dal dispositivo. Tenendo conto di queste considerazioni, è possibile definire le funzioni di I/O in modo tale che si comportino in maniera asincrona nei confronti del processo applicativo che le invoca. Ad esempio la funzione `read` quando, invocata da un processo, in base a quanto detto precedentemente, attiva il dispositivo per leggere i dati e termina restituendo il controllo al processo chiamante il quale continua la sua esecuzione concorrentemente con le operazioni svolte dal dispositivo. Questa modalità di funzionamento delle funzioni d'ingresso/uscita viene offerta da alcuni sistemi operativi ed è nota col termine di *modalità di funzionamento asincrona*.

Raramente, però, la precedente modalità costituisce la sola offerta del sistema in quanto, anche se potenzialmente più efficiente di altre modalità, è sicuramente più complessa da utilizzare poiché, durante la sua esecuzione il processo non è in grado di stabilire se l'operazione è già terminata oppure è ancora in corso. Per questo motivo viene sempre offerta anche l'altra modalità, nota col termine di *modalità sincrona*, la quale prevede che un processo, all'atto dell'invocazione di una funzione di I/O, si sospenda subendo un cambio di contesto per essere riattivato soltanto alla fine dell'operazione. È a tale modalità di comportamento delle funzioni d'ingresso/uscita che faremo riferimento nel prossimo paragrafo essendo questa la modalità più spesso realizzata e sicuramente quella più usata.

Essendo naturalmente asincrone le attività svolte da un dispositivo rispetto a quelle svolte da un processo applicativo, e volendo sincronizzare quest'ultimo con il completamento delle attività del primo, è necessario fare ricorso al meccanismo hardware delle interruzioni per garantire questo tipo di comportamento. Se infatti un dispositivo viene attivato a operare a interruzione di programma, alla fine di ogni operazione il processore viene interrotto e questo segnale può essere utilizzato per realizzare la corretta sincronizzazione. È per questo motivo che in figura 5.2 l'insieme delle funzioni di risposta alle interruzioni provenienti dai dispositivi (*interrupt handler*) sono indicate come parte integrante del livello *device-dependent*.

5.3 Gestore di un dispositivo

Per comprendere la struttura di un driver è conveniente partire da un possibile schema semplificato di come si presenta al processore il controllore di un dispositivo. Come già indicato nei paragrafi iniziali, la CPU agisce sul controllore tramite i registri di cui questo dispone e che vengono indirizzati mediante le istruzioni macchina di I/O. La quantità e la tipologia dei registri presenti nel controllore varia molto da caso a caso e dipende dalla complessità delle funzioni che il dispositivo è in grado di svolgere. In modo estremamente semplificato possiamo però distinguere la presenza di tre registri (o gruppi di registri) che identificheremo con i nomi di *registro*

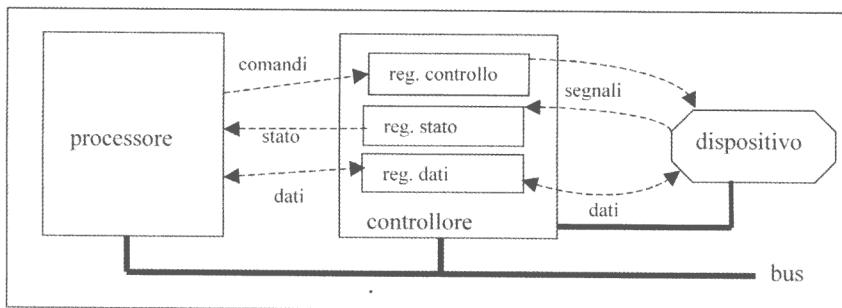


Figura 5.5 Controllo di un dispositivo.

di controllo, registro di stato e registro dati (o buffer del controllore) come indicato nella successiva figura 5.5.

Le funzioni di questi registri sono ben indicate dai loro nomi. Il registro di controllo è quello che consente alla CPU di controllare il funzionamento del dispositivo. In pratica, rappresenta un registro in sola scrittura nel quale la CPU può trasferire valori al fine di attivare il dispositivo affinché lo stesso svolga un certo compito. Come è indicato genericamente nella successiva figura 5.6, è possibile ipotizzare la presenza in questo registro di un bit (*bit di start*) che, quando posto al valore uno, consente al controllore di attivare il dispositivo inviando gli opportuni segnali.

Qualora il dispositivo sia in grado di svolgere più operazioni, come ad esempio nel caso di un disco (read, write, seek ecc.) saranno presenti altri bit utilizzati per selezionare, in fase di attivazione, l'operazione richiesta. È poi presente un bit (*bit di abilitazione alle interruzioni*) che, se posto al valore uno in fase di attivazione, abilita il controllore a inviare un segnale di interruzione alla CPU alla fine dell'operazione del dispositivo.

Il registro di stato è viceversa un registro in sola lettura per la CPU ed è utilizzato dal dispositivo per mantenere aggiornato lo stato in cui si trova, stato che può quindi essere letto dalla CPU. Anche in questo caso possiamo ipotizzare la presenza di un bit (*il flag*) che viene posto a uno alla fine di un'operazione da parte del dispositivo. Quando il flag commuta da zero a uno, se il dispositivo è stato abilitato ad interrompere viene inviato un segnale d'interruzione alla CPU. È inoltre presente un bit che

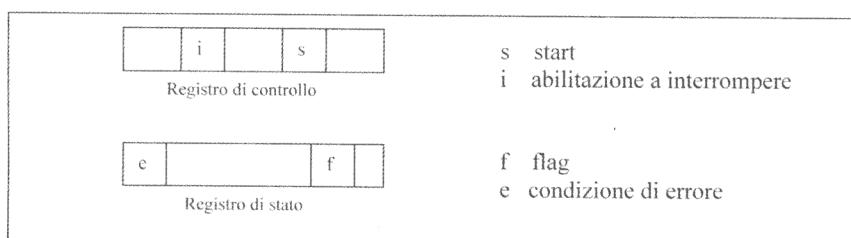


Figura 5.6 Registri di stato e di controllo.

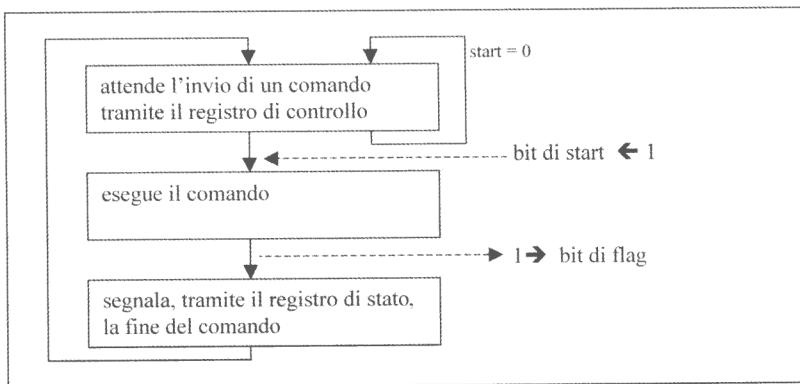


Figura 5.7 Attività del processo esterno.

viene posto a uno tutte le volte che si verifica un evento anomalo durante l'operazione del dispositivo. Anche in questo caso, se gli eventi anomali sono diversi, saranno presenti più bit di errore, uno per ciascuna causa di malfunzionamento.

Infine, il registro dati rappresenta il buffer del controllore nel quale la CPU inserisce il dato da trasferire in uscita, o dal quale preleva il dato letto in fase di input.

5.3.1 Processi esterni

Possiamo adesso illustrare il comportamento di un dispositivo con il relativo controllore come indicato nella successiva figura 5.7. In pratica, ogni dispositivo esegue permanentemente una sequenza di azioni che possiamo considerare come l'esecuzione di un processo anche se, in questo caso, il processo non corrisponde all'esecuzione di un programma ma ad una sequenza di azioni fisse (*cablate*) svolte da un processore particolare (il dispositivo) che opera in parallelo alla CPU. Nel seguito identificheremo questo particolare processo col termine di *processo esterno*.

Normalmente il dispositivo è in *stand-by* in attesa che il bit di start del registro di controllo venga posto a uno. È come se il processo eseguisse un ciclo di attesa attiva aspettando tale evento (vedi figura 5.7). Una volta attivato, il dispositivo esegue il comando e, alla fine di tale operazione, registra questo evento nel registro di stato ponendo a uno il relativo bit di flag. Quindi torna in testa al ciclo e si pone di nuovo in *stand-by* in attesa di una nuova attivazione.

In pratica è come se il processo esterno eseguisse il seguente codice:

```

processo esterno:
{
    while (true) {
        do{} 
        while(Start == 0); // attesa invio di un comando;
        <esegue comando>;
        <registra esito comando, ivi incluso flag=1>;
    }
}
  
```

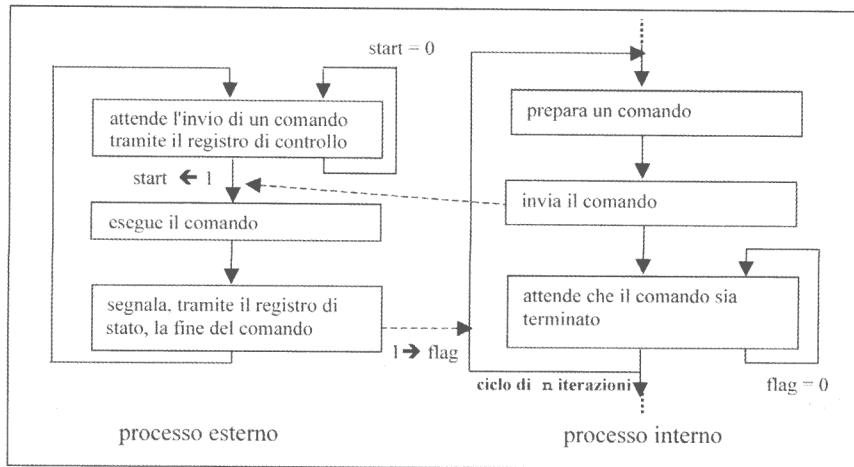


Figura 5.8 Interazione processo esterno - processo applicativo.

In realtà, come già indicato, il processo esterno non è l'attività di un programma in esecuzione e quindi il precedente pseudo-codice serve soltanto ad indicare il suo comportamento. In particolare il ciclo do {} while interno denota l'attesa di attivazione del dispositivo quando lo stesso è in *stand-by*.

5.3.2 Gestione di un dispositivo a controllo di programma

In base alle precedenti considerazioni, nella successiva figura 5.8 viene illustrato come può essere strutturata l'interazione tra il processo esterno e un processo applicativo che, ad un certo punto della sua evoluzione, entra in un ciclo di n iterazioni per leggere o scrivere n dati consecutivamente, operando con lo specifico dispositivo.

Come esempio, possiamo schematizzare con il seguente frammento di programma il ciclo che il processo applicativo esegue per trasferire n dati in lettura:

```

...
for (int i=0; i++; i<n)
{
    <prepara il comando>;
    // assembla in un registro interno alla
    // CPU il valore da trasferire nel
    // registro di controllo;

    <invia il comando>; // ponendo a uno il bit di start;

    do{} while flag == 0;// ciclo di attesa sul flag;

    <verifica l'esito>;
    // controlla tramite il registro di
    // stato che non ci siano stati errori e
}

```

```
// preleva il dato letto dal registro
// dati del controllore;
}
...
```

Come evidenziato dall'istruzione:

```
do ; while (flag == 0)
```

il processo, per ciascuno degli n trasferimenti, prima di poter prelevare il dato, deve attendere che lo stesso sia disponibile nel registro dati del controllore. Tale attesa viene realizzata mediante il ciclo sul flag (*ciclo di attesa attiva*).

Questo schema prevede quindi che sia il processo applicativo, per ciascun dato da trasferire, a controllare direttamente la disponibilità del dato nel registro del controllore. Per questo motivo è noto in letteratura come schema di gestione di un dispositivo a controllo di programma. Ma si capisce subito che tale schema non è adatto per essere utilizzato nell'ambito di un sistema operativo multiprogrammato dove, come già indicato nel capitolo 3, per evitare di mantenere inutilmente inutilizzata la CPU, quando un processo applicativo non può proseguire dovendo attendere un particolare evento, è necessario sospornerlo tramite un commutazione di contesto per riattivarlo all'istante in cui l'evento atteso si sarà verificato.

5.3.3 Gestione di un dispositivo a interruzione

Per forzare la sospensione del processo applicativo quando deve restare in attesa sul flag è possibile associare al dispositivo un semaforo inizializzato a zero e sostituire al ciclo di attesa attiva una *wait* su tale semaforo. Ad esempio indicando con dato_disponibile l'identificatore di questo semaforo:

```
semaforo dato_disponibile; dato_disponibile = 0;
```

il precedente frammento di programma eseguito dal processo applicativo potrebbe essere così modificato:

```
for (int i=0; i++ < n) {
    <prepara il comando>;
    // assembla in un registro interno alla
    // CPU il valore da trasferire nel
    // registro di controllo;

    <invia il comando>; // ponendo a uno il bit di start;

    dato_disponibile.wait(); // attesa dato;
    <verifica l'esito>; // controlla...
}
```

Adesso, però, il processo si sospende e non è quindi più in grado di controllare direttamente la disponibilità del dato nel registro del controllore. Dovrà quindi essere il controllore a segnalare questo evento e quindi a risvegliare il processo applicativo.

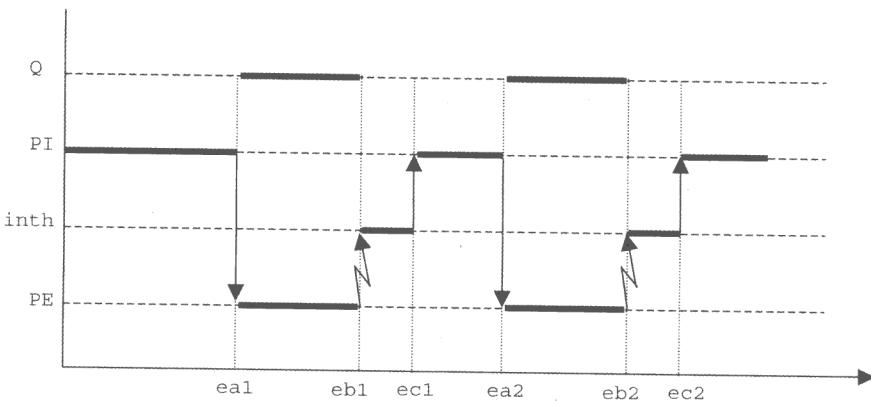


Figura 5.9 Diagramma temporale della gestione a interruzione.

Poiché il processo esterno non è un programma in esecuzione e non ha quindi la possibilità di eseguire la primitiva `signal` sul semaforo, è necessario che sia presente nel sistema una funzione a cui delegare tale compito, funzione che deve essere chiamata non appena si verifica l'evento corrispondente alla transizione da zero a uno del bit di flag.

Per risolvere il problema è sufficiente ricordarsi che, se il dispositivo viene inizializzato abilitandolo a interrompere (ponendo al valore uno il bit di abilitazione alle interruzioni del registro di controllo), allora quando si verifica la transizione del flag da zero a uno il controllore interrompe la CPU e, via hardware, va in esecuzione quella funzione che è stata precedentemente indicata come *funzione di risposta alle interruzioni del dispositivo*. Possiamo quindi demandare a tale funzione il compito di eseguire la primitiva `signal` sul semaforo. Nella successiva figura 5.9 viene riportato un diagramma temporale che illustra come si alternano le fasi di esecuzione del processo applicativo che esegue l'operazione di input degli n dati (indicato con l'identificatore PI), l'esecuzione delle attività del dispositivo (processo esterno PE), della funzione di risposta alle interruzioni del dispositivo (indicata con l'identificatore inth) e le fasi di esecuzione di un generico processo applicativo (processo Q) che si ipotizza venga schedulato quando PI, si sospende sul semaforo `dato_disponibile` e quindi subisce una commutazione di contesto.

In questo esempio si suppone che all'istante `ea1` il processo applicativo PI, eseguendo il primo ciclo del `for`, una volta attivato il dispositivo si blocchi sul semaforo `dato_disponibile`. Il processo subisce un cambio di contesto per cui viene schedulato un altro processo (Q) e, in parallelo all'attività della CPU inizia l'attività del dispositivo (PE). Quando termina questa attività e viene posto a uno il flag, la CPU viene interrotta (evento `eb1`) e va in esecuzione la routine di interruzione (inth) che, eseguendo una `signal` sul semaforo `dato_disponibile`, riattiva PI il quale, nell'ipotesi che venga subito schedulato, torna in esecuzione (evento `ec1`). Successivamente questa sequenza di eventi si ripete per ciascuno degli n dati da leggere e cioè per ciascuna delle n iterazioni del ciclo `for`. In pratica possiamo

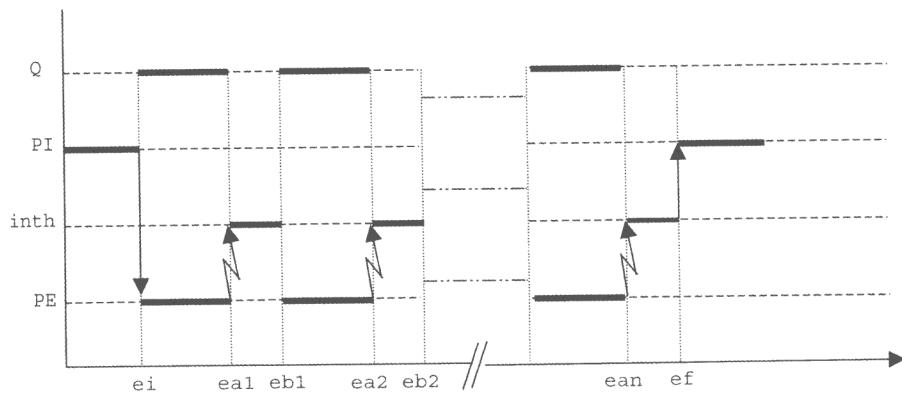


Figura 5.10 Trasferimento di n dati consecutivi.

vedere la routine di interruzione come un'appendice interna al sistema del processo esterno, necessaria per abilitare quest'ultimo a eseguire le operazioni che realizzano la sincronizzazione tra il processo esterno e il processo applicativo.

Il precedente schema, noto anche come schema di gestione di un dispositivo a interruzione di programma, soffre di un inconveniente: se il processo applicativo deve trasferire n dati, ed eseguire quindi n iterazioni del ciclo `for`, per n volte viene sospeso e di seguito riattivato inutilmente poiché deve subito sospendersi di nuovo. Soltanto quando viene riattivato per l'ultima volta il processo non si sospende più essendo terminato l'intero trasferimento. Tutta questa sequenza di inutili commutazioni di contesto genera esclusivamente una perdita di tempo. Sarebbe estremamente più utile fornire al processo applicativo una funzione con la quale specificare il numero n dei dati da trasferire e l'indirizzo in memoria dove i dati devono essere trasferiti (o da cui prelevarli nell'ipotesi di un trasferimento in uscita), attivare il dispositivo e quindi bloccare il processo fino a quando l'intero trasferimento non è completato, riducendo a una soltanto il numero delle commutazioni di contesto. Nella successiva figura 5.10 viene riportato lo schema temporale di questo tipo di comportamento.

Nella figura l'evento ei indica l'inizio dell'operazione di input con l'attivazione del dispositivo e la corrispondente sospensione del processo PI . I successivi eventi eai (con $i=1, 2, \dots, n$) rappresentano gli istanti in cui la CPU viene interrotta in corrispondenza dell' i -esima transizione da zero a uno del bit di flag. In corrispondenza di questi eventi si sospende temporaneamente il processo in esecuzione (Q nella figura) per eseguire la routine d'interruzione. Gli eventi ebl ($i=1, 2, \dots, n-1$) rappresentano gli istanti in corrispondenza dei quali la routine $inth$ termina avendo gestito la i -esima interruzione. In corrispondenza di questi istanti riprende l'esecuzione del processo Q interrotto e viene riattivato il dispositivo per leggere il dato successivo. Infine, dopo che è arrivata l'ultima interruzione relativa all'ultimo dato, terminata l'ultima esecuzione della routine $inth$ (evento ef), il trasferimento è completato e il processo PI può essere riattivato.

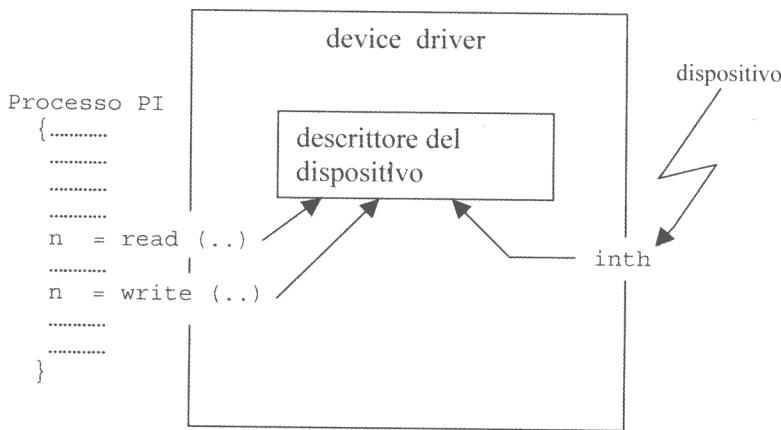


Figura 5.11 Astrazione di un dispositivo periferico.

Descrittore di un dispositivo Per ottenere questo tipo di comportamento è necessario che la routine di interruzione riesca a discriminare, fra le varie interruzioni che deve gestire, quali sono quelle intermedie di ogni blocco di trasferimenti e qual è quella finale in corrispondenza della quale deve essere riattivato il processo applicativo che ha richiesto il trasferimento.

Un modo semplice di risolvere il problema è quello di riservare in memoria una struttura dati destinata a rappresentare il dispositivo e sulla quale possa operare sia il processo applicativo, mediante le funzioni offerte dall'interfaccia *device independent*, sia la routine di gestione delle interruzioni lanciate dal dispositivo. È questa struttura dati, spesso indicata col termine di *descrittore del dispositivo*, che insieme con le relative funzioni di accesso, costituisce il *device driver* e cioè il modulo del sistema dedicato alla gestione del dispositivo fisico. Possiamo considerare questo modulo come un'astrazione del dispositivo fisico (vedi figura 5.11) così come viene visto da un processo applicativo tramite le funzioni offerte dall'interfaccia *device independent* e che racchiude al proprio interno tutti i dettagli dipendenti dal dispositivo stesso.

Il descrittore del dispositivo è una struttura dati che viene utilizzata per due scopi diversi:

- nascondere al proprio interno tutte le informazioni che sono associate al dispositivo;
- consentire la comunicazione di informazioni tra il processo applicativo e il dispositivo (numero di dati da trasferire, indirizzo del buffer in cui trasferirli) e viceversa (informazioni relative all'esito del trasferimento).

Nella successiva figura 5.12 viene riportato un esempio di possibili campi contenuti nel descrittore. Tale esempio è del tutto indicativo, tenendo conto che la struttura cambia molto da sistema a sistema e, nell'ambito dello stesso sistema, cambia da dispositivo a dispositivo.

Indirizzo registro di controllo
Indirizzo registro di stato
Indirizzo registro dati
Semaforo di sincronizzazione: <u>dato_disponibile</u>
Contatore dati da trasferire: <u>contatore</u>
Indirizzo del buffer: <u>puntatore</u>
Risultato del trasferimento: <u>esito</u>

Figura 5.12 Descrittore del dispositivo.

Possiamo immaginare un'area del descrittore riservata a contenere gli indirizzi dei registri del controllore del dispositivo. Sarà poi presente un campo di tipo semaforo per garantire la sincronizzazione tra il processo applicativo che attiva un trasferimento e la routine di risposta alle interruzioni (il semaforo dato_disponibile dell'esempio precedente).

I due campi relativi al numero dei dati da trasferire e all'indirizzo del buffer in cui trasferirli vengono inizializzati dalla funzione di lettura o scrittura eseguita dal processo applicativo e sono utilizzati dalla funzione di risposta alle interruzioni per ricavare l'indirizzo di memoria in cui trasferire il dato letto (o da cui prelevare il dato da scrivere), e per discriminare, fra le varie interruzioni da gestire, quale sia l'ultima di ogni trasferimento. Infine, il campo relativo al risultato viene compilato dalla funzione di risposta alle interruzioni alla fine del trasferimento, codificandovi il suo esito (trasferimento concluso correttamente oppure indicazioni degli eventi anomali che si possono essere verificati). Sarà compito della funzione di lettura o scrittura, una volta terminato il trasferimento, leggere tale campo e decidere come proseguire.

Driver di un dispositivo Per esemplificare quanto è stato detto, possiamo adesso illustrare come è strutturato il driver di un dispositivo, mostrando le funzioni di lettura o scrittura (funzioni appartenenti all'interfaccia device independent, vedi figura 5.2) e la funzione di risposta alle interruzioni, affinché il comportamento del trasferimento dati a interruzione di programma sia quello illustrato nel diagramma di figura 5.10.

Facciamo ad esempio riferimento alla funzione di lettura:

```
int read (int disp, char *pbuff, int cont)
```

dove disp identifica il dispositivo su cui operare, pbuff il puntatore al buffer di sistema in cui trasferire i dati letti, e cont il numero di dati da leggere. Supponiamo,

inoltre, che la funzione restituiscia -1 in caso di terminazione erronea, oppure il numero dei dati letti in caso di terminazione regolare e di indicare con `descrittore[disp]` il descrittore del dispositivo `disp`. La funzione per prima cosa trasferisce nei corrispondenti campi del descrittore i valori relativi al numero di dati da leggere e all'indirizzo del buffer di sistema in cui trasferirli. Quindi, attiva il dispositivo scrivendo nel registro di controllo del controllore e blocca il processo applicativo sul semaforo `dato_disponibile`. Alla fine del trasferimento il processo viene riattivato e la funzione termina leggendo il campo `esito` nel quale è stato codificato se il trasferimento è terminato erroneamente o in modo corretto. Nel primo caso viene restituito il valore -1, nell'altro il numero di dati effettivamente letti.

```
int read (int disp, char * pbuf, int cont)
{
    descrittore[disp].contatore = cont;
    descrittore[disp].puntatore = pbuf;
    <attivazione del dispositivo>; // bit di start=1
    // sospensione del processo
    descrittore[disp].dato_disponibile.wait();
    // in caso di errore restituisce -1
    if (descrittore[disp].esito == <codice di errore>)
        >return(-1);
    // altrimenti restituisce il numero di dati letti
    return(cont - descrittore[disp].contatore);
}
```

All'arrivo di un'interruzione il processo in esecuzione viene interrotto e, via hardware, va in esecuzione la funzione `inth`. Questa, per prima cosa, legge il registro di stato del controllore per verificare la causa dell'interruzione. Se questa non è dovuta a errori ma alla fine della lettura di un dato da parte del dispositivo, il dato stesso viene letto nella variabile locale `b`, prelevandolo dal registro dati, e successivamente trasferito nel buffer di sistema all'indirizzo contenuto nel campo `puntatore`. Quindi il campo `puntatore` viene incrementato per renderlo consistente col trasferimento del prossimo dato e il campo `contatore` viene decrementato per registrare che rimane da leggere un dato in meno. Se dopo il decremento il `contatore` è ancora diverso da zero ciò significa che il trasferimento non è ancora completato e quindi il dispositivo viene riattivato per il prossimo trasferimento. Altrimenti l'interruzione ricevuta è l'ultima dell'intero trasferimento quindi il processo applicativo può essere riattivato. Viene scritto nel campo `esito` che il trasferimento si è concluso correttamente e il dispositivo viene disattivato. A questo punto la routine termina con un ritorno da interruzione. Se l'interruzione è dovuta ad un errore, e qualora questo sia mascherabile, viene eseguita una routine di gestione dell'evento anomalo. Se non è possibile mascherare l'errore la routine termina riattivando il processo applicativo dopo aver codificati nel campo `esito` la causa della terminazione anomala. In questo caso, come già indicato, sarà compito del livello di software superiore gestire l'evento anomalo.

```
void inth() { // funzione di risposta alle interruzioni
    char b;
    <legge il registro di stato del controllore>;
    if (<bit di errore == 0>) { // assenza di errori
```

```

<lettura del registro dati>
<assegnamento alla variabile locale b>;
* descrittore[disp].puntatore = b;
descrittore[disp].puntatore++;
descrittore[disp].contatore--;
if(descrittore[disp].contatore != 0)
    <riattivazione dispositivo>;
else {
    >descrittore[disp].esito=<terminazione corretta>;
    <disattivazione del dispositivo>;
    // riattivazione processo
    descrittore[disp].dato_disponibile.signal();
}
else { // presenza di errori
    <routine di gestione errore>;
    if (<errore non recuperabile>)
        descrittore[disp].esito=<codice errore>;
    // riattivazione processo
    descrittore[disp].dato_disponibile.signal();
}
}
return; // ritorno da interruzione
}

```

5.3.4 Gestione di un dispositivo in DMA

Come è stato richiamato nel capitolo 2, i dispositivi a blocchi sono spesso collegati alla CPU e alla memoria per mezzo di un canale di DMA che consente di trasferire i dati dal registro del controllore direttamente in memoria tramite il bus del sistema, operando in *cycle stealing*, cioè competendo con la CPU nell'accedere alla memoria tramite il bus di sistema. Questa tecnica riduce notevolmente il tempo di trasferimento di un blocco di dati e, soprattutto, elimina la necessità da parte della CPU di eseguire la funzione di gestione delle interruzioni per ciascun dato letto. Infatti il canale di DMA lancia una sola interruzione per ogni blocco di dati in corrispondenza della fine dell'intero trasferimento.

Come è stato indicato, per garantire questo comportamento, nel canale di DMA sono presenti due registri che svolgono le stesse funzioni delle variabili, puntatore e contatore viste precedentemente e gestite, via hardware, dal canale di DMA esattamente come nel caso precedente erano gestiti dalla routine di gestione alle interruzioni (vedi figura 5.13).

Il *device driver* di un dispositivo connesso tramite un canale di DMA non è quindi molto diverso dal caso precedente. Le uniche varianti sono legate al fatto che le due informazioni relative all'indirizzo del buffer in memoria e del contatore dei dati da trasferire non fanno parte del descrittore del dispositivo in quanto sono presenti come registri fisici nel canale di DMA. Inoltre la funzione di gestione delle interruzioni *inth* viene eseguita una sola volta per ogni blocco di trasferimenti in corrispondenza della fine trasferimento dell'intero blocco, svolgendo le stesse funzioni che, nel caso precedente, venivano svolte durante la gestione dell'ultima interruzione.

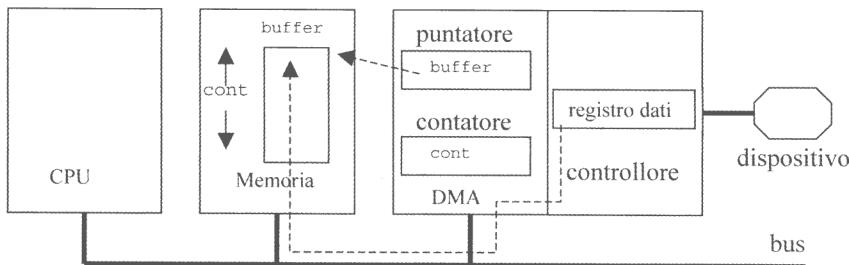


Figura 5.13 Canale di DMA.

5.3.5 Flusso di controllo durante un trasferimento

Al fine di chiarire quanto visto nei paragrafi precedenti, verrà adesso illustrato un esempio di come è strutturato il flusso di controllo durante l'esecuzione di una chiamata di sistema relativa ad un trasferimento di dati. Per questo motivo faremo riferimento all'esecuzione da parte di un processo applicativo PI di una chiamata di sistema per leggere una sequenza di byte, in modalità sincrona, da un dispositivo periferico (vedi figura 5.14).

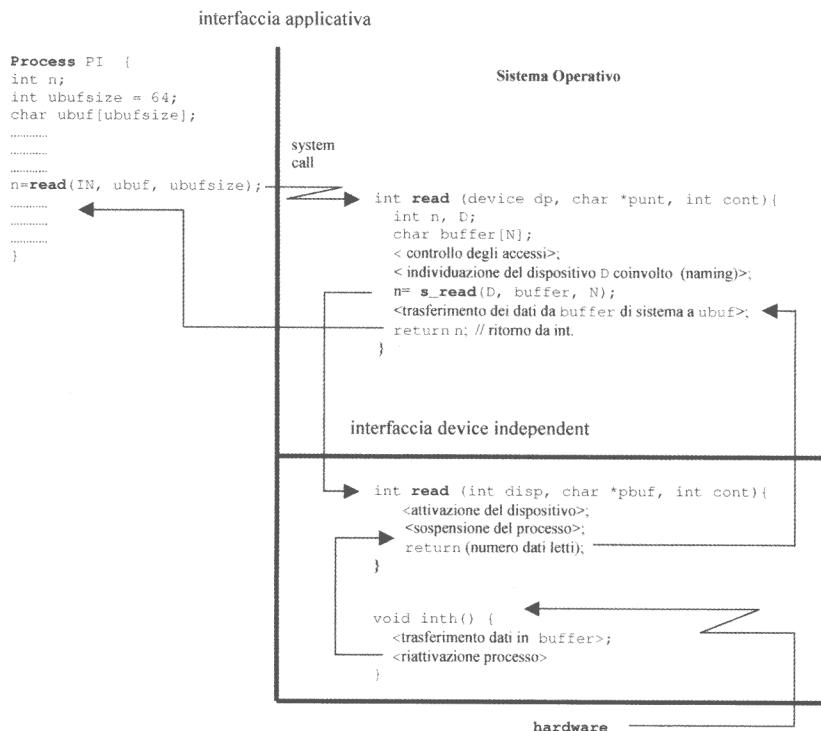


Figura 5.14 Flusso di controllo durante l'esecuzione di una primitiva di I/O.

Nell'esempio, il processo invoca, tramite una system call, una delle funzioni offerte dall'interfaccia applicativa, nella fattispecie la funzione di lettura, passando come parametri il nome simbolico del dispositivo (`IN`), l'indirizzo del buffer nella memoria virtuale del processo (`ubuf`) e il numero di byte da leggere (`ubufsize`). Il controllo entra quindi nel sistema operativo e inizia l'esecuzione della funzione `read` che svolge tutti i compiti già indicati come facenti parte del livello indipendente dai dispositivi. In particolare vengono eseguiti tutti i controlli relativi ai diritti di accesso e, se per semplicità supponiamo che non si verifichino malfunzionamenti, viene individuato il nome unico del dispositivo (`D`), e il buffer di sistema da usare per la lettura (`buffer`). Quindi viene invocata la funzione `read` di più basso livello (funzione del driver, indicata in figura con l'identificatore `_read` per distinguerla da quella di livello più alto) che attiva il dispositivo e blocca il processo applicativo. Ogni interruzione sarà quindi gestita dalla funzione `inth` di risposta alle interruzioni per trasferire nel buffer di sistema i singoli byte letti. Quando infine arriva l'interruzione di fine trasferimento il processo viene riattivato e può quindi completare la funzione di lettura che restituisce il numero di byte letti e il controllo torna quindi alla `read` di livello più alto che trasferisce i dati dal buffer di sistema al buffer del processo (alla locazione `ubuf` nella memoria virtuale del processo). Quindi, con un ritorno da interruzione, il controllo torna al processo applicativo.

5.3.6 Gestione del temporizzatore

Un dispositivo molto particolare è costituito dal timer. La sua particolarità risiede nel fatto che non viene utilizzato per trasferire dati ma semplicemente come una fonte di interruzioni cadenzate nel tempo. La necessità di avere uno o più dispositivi di questo tipo nasce da varie esigenze molto diverse tra loro.

Un primo esempio di utilizzazione di un timer è stato visto nel capitolo 2 in relazione agli algoritmi di schedulazione della CPU in sistemi time-sharing. Ad esempio in uno scheduler che funziona con strategia *round robin* ad ogni processo, quando viene schedulato, viene assegnato un intervallo massimo di esecuzione (*time-slice*) allo scadere del quale il processo, se non è terminato o se non si è sospeso, viene comunque revocato per consentire l'esecuzione di altri processi.

Un secondo esempio, sempre presente in tutti i sistemi, è quello connesso alla gestione della data e del tempo, in modo tale da mantenere costantemente aggiornati sia la data corrente sia l'orologio in tempo reale.

Un ulteriore esempio, estremamente importante soprattutto nei sistemi real-time, è relativo alla possibilità di fornire ai processi applicativi un insieme di servizi connessi alla gestione del tempo. Spesso è disponibile un servizio relativo alla possibilità di specificare attese programmate o di ricevere segnali di time-out.

In questo paragrafo illustriamo, come esempio, un semplice driver per un timer destinato a fornire il servizio di attese programmate ad un insieme di processi applicativi.

Da un punto di vista hardware, il controllore di un timer contiene, oltre ai registri di controllo e di stato, un registro contatore nel quale la CPU può trasferire un valore intero che, successivamente, il timer decrementa con una frequenza fissa, o in certi casi programmabile. Quando il registro contatore raggiunge il valore zero il controllore lancia un segnale d'interruzione.

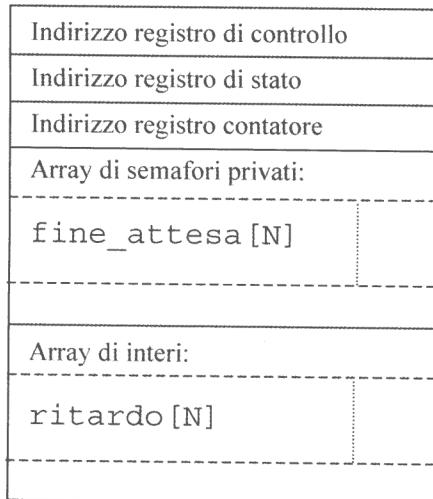


Figura 5.15 Descrittore del timer.

Come per ogni altro dispositivo, anche per il timer il relativo driver sarà costituito dalla struttura dati **descrittore**, dalla funzione di risposta alle interruzioni **inth** e inoltre offrirà ai processi applicativi la primitiva **delay** che un processo può invocare passandogli come parametro un valore intero che rappresenta (in termini di quanti di tempo) la durata dell'attesa richiesta dal processo.

In figura 5.15 è illustrato un esempio di come potrebbe essere strutturato il descrittore del timer.

Per semplicità, nell'esempio viene fatto riferimento al caso in cui i processi applicativi siano gli N processi P_0, P_1, \dots, P_{N-1} .

Nel descrittore, oltre agli indirizzi dei registri del controllore, è presente un array di N semafori (**fine_attesa [N]**), uno per ciascun processo e tutti inizializzati al valore zero. Ciascun semaforo viene utilizzato per bloccare il corrispondente processo che invoca la primitiva **delay**. È infine presente un array di interi che viene utilizzato per mantenere aggiornato il numero di quanti di tempo che, per ciascun processo, devono ancora passare prima che il processo possa essere riattivato. Utilizzando questa struttura possiamo quindi schematizzare il codice sia della primitiva **delay** che della routine di risposta alle interruzioni:

```

void delay (int n){
    int proc;
    proc = <indice del processo in esecuzione>;
    descrittore.ritardo[proc] = n;
    // sospensione del processo
    descrittore.fine_attesa[proc].wait();
}

void inth() {

```

```

for (int i=0; i<N, i++)
    if(desrittore.ritardo[i]!=0) {
        desrittore.ritardo[i]--;
        if (desrittore.ritardo[i]==0)
            desrittore.fine_attesa[proc].signal();
    }
}

```

La primitiva `delay` si limita a registrare nell'elemento corrispondente al processo in esecuzione del campo `ritardo`, il valore del numero di quanti di tempo che corrisponde al periodo di attesa richiesto dal processo. Quindi blocca il processo richiedente sul proprio semaforo.

All'arrivo di un'interruzione da parte del timer la procedura `inth` scandisce l'intero array `ritardo`. Se l'elemento di indice i di questo array è diverso da zero, ciò significa che il processo P_i è in attesa e tale elemento denota il numero di quanti che deve ancora attendere. Questo valore viene quindi decrementato e se raggiunge il valore zero ciò significa che l'attesa è terminata e perciò il processo viene riattivato.

5.4 Gestione e organizzazione dei dischi

Le unità di memoria di massa, e soprattutto i dischi, rappresentano dei dispositivi di particolare importanza agli effetti dell'efficienza e dell'operatività dell'intero sistema. Molte sono infatti le componenti di un sistema operativo che richiedono la disponibilità di aree di memoria di massa. Ad esempio, come è stato indicato nel precedente capitolo, per fornire il supporto al concetto di memoria virtuale di un processo è necessario disporre di un'area di memorizzazione di massa (la *swap area*) destinata a contenere tutte le informazioni (codice e dati) relative ai processi temporaneamente non allocati in memoria principale. Analogamente, come vedremo nel successivo capitolo, è la memoria di massa che fornisce il supporto alla memorizzazione dei file e quindi alla realizzazione di quella componente del sistema operativo nota come *file system*. È quindi evidente che dall'efficienza di uso dei dischi e dalla loro affidabilità dipende sia l'efficienza che l'affidabilità dell'intero sistema. Per questo motivo, nei successivi paragrafi verranno indicati i criteri oggi più usati per migliorare, da un lato l'efficienza di uso, dall'altro l'affidabilità, di un sistema di memoria di massa costituito da uno o più dischi.

5.4.1 Organizzazione fisica dei dischi

Un disco è fisicamente costituito da un piatto di plastica su cui viene depositato uno strato di materiale magnetico sul quale i dati possono essere scritti, e successivamente letti, tramite una testina di lettura/scrittura che modifica lo stato di magnetizzazione del materiale in fase di scrittura e ne rilava lo stato in fase di lettura. Durante le operazioni di lettura o scrittura, la testina resta fissa mentre il disco ruota sotto di essa. Per questo motivo i dati sono memorizzati su tracce concentriche del disco (vedi figura 5.16-a). Sono disponibili due categorie di dispositivi: quelli a testine fisse e quelli a testine mobili. Nei primi sono disponibili tante testine di lettura/scrittura quante sono le tracce del disco. Per cui le testine sono fisse e per leggere o scrivere

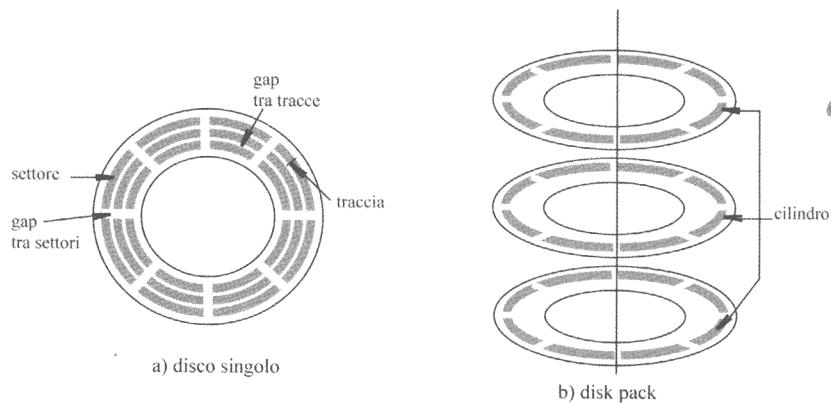


Figura 5.16 Organizzazione fisica dei dischi.

dati relativi ad una traccia è necessario selezionare la testina corrispondente alla traccia desiderata. Nei dischi a testine mobili esiste una sola testina montata su un braccio mobile che può traslare in senso radiale al disco. In questi casi per leggere o scrivere dati relativi ad una traccia è necessario preventivamente muovere il braccio al fine di posizionare la testina sopra la traccia desiderata. Il numero di tracce disponibili su un disco è dell'ordine di alcune migliaia. Come indicato in figura 5.16-a, due tracce contigue sono separate da uno spazio vuoto privo di materiale magnetico (*intertrack gap*).

Il numero di bit memorizzabili all'interno di ogni traccia è lo stesso per tutte le tracce. Ciò consente di semplificare la logica di controllo del dispositivo anche se implica che i bit siano memorizzati con diversa densità, maggiore per le tracce più interne e minore per quelle più esterne.

Il trasferimento di dati tra memoria principale e disco avviene in termini di unità di trasferimento di dimensione fissa corrispondenti ai settori nei quali è divisa ogni traccia (vedi figura 5.16-a). Tipiche dimensioni di un settore oscillano tra i 512 e i 1024 byte. Anche due settori adiacenti sono separati da uno spazio vuoto (*intersector gap*).

Il numero di settori all'interno di una stessa traccia è dell'ordine di alcune centinaia. Nella successiva Tabella 5.2 sono riportati, come esempio, i dati di alcuni dischi Western Digital: il disco AC2540 da 540 MB e il disco WDE18300 da 18.3 GB.

Per identificare in maniera univoca i singoli settori all'interno di una traccia, in fase di formattazione del disco vengono memorizzati alcuni dati di controllo che consentono al controllore del disco di identificare l'inizio di ogni traccia e l'inizio e la fine di ogni settore.

In molti dischi è possibile usare le due facce del piatto, raddoppiando così la sua capacità. Nei sistemi in cui è necessaria una memoria di massa di notevole capacità si ricorre spesso ai cosiddetti *disk pack* costituiti da molti dischi concentrici (vedi figura 5.16-b). In questi casi si parla spesso di *cilindro*, intendendo con ciò l'insieme di tutte le tracce concentriche.

Parametri	AC2540	WDE18300
Numero cilindri (N. di tracce per ogni faccia)	1048	13614
Tracce per cilindro	4	8
Settori per traccia	252	320
Byte per settore	512	512
Capacità	540 MB	18.3 GB
Tempo minimo di seek (tra cilindri adiacenti)	4 msec.	0.6 msec.
Tempo medio di seek	11 msec.	5.2 msec.
Tempo di rotazione	13 msec.	6 msec.
Tempo di trasferimento di un settore	53 msec.	19 msec.

Tabella 5.2 parametri caratterizzanti i due dischi WD AC2540 e WDE18300.

Un settore costituisce l'unità minima di trasferimento delle informazioni e ha un indirizzo fisico che è costituito dalle proprie coordinate:

(f, t, s)

dove f rappresenta il numero della faccia, t il numero della traccia (o cilindro) nell'ambito della faccia e s il numero del settore entro la traccia. Per semplificare il criterio di indirizzamento dei veri settori, e quindi le informazioni da passare come parametri alle funzioni di accesso alla memoria di massa, normalmente tutti i settori che compongono un disco (o un pacco di dischi), vengono considerati come costituenti un array dove, quindi, ogni settore viene identificato tramite il proprio indice nell'array. Per esempio il settore di indice zero coincide con quello di coordinate (faccia: zero, traccia: zero, indice del settore nella traccia: zero); il settore di indice uno a quello di coordinate (faccia: zero, traccia: zero, indice del settore nella traccia: uno), e così via fino al settore di indice N-1, se N corrisponde al numero di settori per traccia. Quindi si prosegue con la traccia successiva nell'ambito della stessa faccia, e cioè il settore di indice N corrisponde a quello di coordinate (faccia: zero, traccia: uno, indice del settore nella traccia: zero) e così via con tutti i settori delle tracce relative alla faccia zero. Quindi si prosegue con lo stesso criterio con le facce successive (vedi figura 5.17).

Quindi, indicando con M il numero di tracce per faccia e con N il numero di settori per traccia, un settore di coordinate (f, t, s), viene rappresentato nell'ambito dell'array di tutti i settori come quello di indice i calcolato con la seguente semplice formula:

$$i = f * M + t * N + s$$

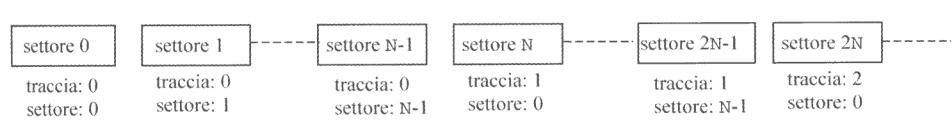


Figura 5.17 Indicizzazione dei settori.

5.4.2 Criteri di ordinamento dei dati su disco e di scheduling delle richieste di trasferimento

Per valutare l'efficienza di uso di un disco viene spesso preso in considerazione un indicatore, il *tempo medio di trasferimento* (*TF*), che corrisponde al tempo mediamente necessario per leggere o scrivere una certa quantità di byte, ad esempio i byte relativi ad un settore. Il valore di *TF* può essere scomposto in due componenti: il *tempo medio di accesso* (*TA*), relativo al tempo necessario per posizionare la testina di lettura/scrittura in corrispondenza dell'inizio del settore desiderato, e il tempo necessario a effettuare il vero e proprio trasferimento dei dati relativi al settore (*TT*):

$$TF = TA + TT \quad (5.1)$$

Con riferimento ad un disco a testine mobili possiamo scomporre ulteriormente il tempo *TA* in due componenti: il tempo necessario per posizionare la testina mobile sopra la traccia contenente il settore desiderato, indicato come tempo di *seek* (*ST*), e il tempo necessario per aspettare che il settore desiderato ruoti sotto la testina, indicato spesso con nome di *rotational latency* (*RL*). Il primo fra i due dipende dalla velocità con cui si muove il braccio che porta la testina e, ovviamente, dalla distanza tra la traccia su cui è posizionata la testina all'inizio del movimento, e la traccia su cui dovrà posizionarsi. Come indicato nella precedente tabella 5.2, tale tempo è dell'ordine di alcuni millisecondi (4 msec. come valore minimo, relativo allo spostamento tra due tracce consecutive, e 11 msec. come valore medio per il primo fra i due dischi, 0.6 msec. come minimo e 5.2 msec. come valore medio per il secondo). Per quanto riguarda *RL*, mediamente corrisponde al tempo richiesto al disco per compiere mezzo giro ed è anch'esso dell'ordine di alcuni millisecondi fino ad arrivare ad alcune centinaia di millisecondi per un floppy disk. Per i due dischi riportati in Tabella 5.2 abbiamo 6.5 msec. per il primo e 3 msec. per il secondo.

La precedente relazione (5.1) può quindi essere meglio specificata come:

$$TF = ST + RL + TT \quad (5.2)$$

Infine, il tempo di trasferimento *TT* corrisponde al tempo necessario per far transitare sotto la testina l'intero settore e cioè, se indichiamo con *t* il tempo impiegato per compiere un giro e con *s* il numero di settori per traccia, possiamo approssimare *TT* al valore *t/s*. Tale valutazione non è esatta poiché non tiene conto degli intersecor gap. Con riferimento ai due dischi riportati in tabella abbiamo i due valori di *TT* pari a 53 e 19 microsecondi rispettivamente.

In base a quanto indicato precedentemente risulta evidente che il *tempo medio di trasferimento* dipende sostanzialmente dal *tempo medio di accesso* e quindi dai due fattori *ST* e *RL* sui quali risulterà opportuno intervenire per ridurre *TF*.

Due sono in particolare i modi di intervento: uno riguarda i criteri con cui i dati dovranno essere memorizzati sul disco al fine di ridurre il *tempo medio di accesso*. Il secondo è relativo al criterio con cui schedulare le richieste di accesso al disco da parte di processi diversi, sempre con l'ottica di ridurre tale parametro.

Per quanto riguarda il primo aspetto supponiamo, ad esempio, che un processo desideri leggere in memoria principale un file che occupi su disco un certo numero di settori. Ad esempio, con riferimento al disco WDE18300 riportato in tabella 5.2,

supponiamo che il file richieda 640 settori (pari a un totale di 160 KB). Facciamo prima l'ipotesi che i 640 settori occupino completamente due tracce consecutive (*allocazione contigua*). Il tempo richiesto per leggere tutti i settori della prima traccia è pari al tempo di seek, più il *rotational latency*, più il tempo richiesto per il trasferimento dei 320 settori della prima traccia (che corrisponde al tempo di rotazione del disco) e quindi è pari a $5.2 + 3 + 6 = 14.2$ millisecondi. La stessa cosa vale per leggere i 320 settori della seconda traccia con la differenza, in questo caso, che il tempo di seek si riduce al minimo per il passaggio da una traccia alla successiva: $0.6 + 3 + 6 = 9.6$ millisecondi. Per leggere l'intero file sono quindi necessari 23.8 millisecondi.

Supponiamo adesso di valutare di nuovo il tempo necessario per leggere il file nell'ipotesi che i 640 settori siano allocati su disco in maniera casuale. In questo caso, il tempo necessario per trasferire ciascuno dei 640 settori corrisponde a $5.2 + 3 + 0.019 = 8.419$ millisecondi (il valore 0.019 è il tempo di trasferimento di un singolo settore espresso in millisecondi, come indicato nella tabella 5.2). Per leggere l'intero file sono quindi necessari $8.419 \times 640 = 5388.16$ millisecondi, e cioè oltre 5 secondi. Da ciò risulta evidente quanto sia importante la strategia di allocazione dei file su memoria di massa, argomento che sarà trattato nel prossimo capitolo.

Per il momento, nel valutare il tempo richiesto per completare un trasferimento di dati, abbiamo preso in esame esclusivamente il *tempo medio di trasferimento* corrispondente al tempo mediamente necessario per leggere o scrivere un dato settore. In realtà, per valutare il tempo di attesa di un processo che ha richiesto di effettuare un trasferimento dovremmo considerare anche il tempo che il processo resta sospeso nella coda di accesso al dispositivo nell'ipotesi che, al momento della richiesta di trasferimento, il dispositivo sia occupato da un altro processo. È ovvio che quest'ultimo tempo dipende strettamente dal criterio con cui verrà gestita la coda di attesa, cioè dalla strategia di scheduling delle richieste al disco.

Per esempio, supponiamo che un processo stia operando sul disco con la testina posizionata sul cilindro n. 20 e supponiamo che durante la sua operazione molti altri processi richiedano di operare sul disco richiedendo, in particolare, di operare sui cilindri 14, 40, 23, 47, 7. Supponiamo anche che questo sia l'ordine con cui i vari cilindri sono stati richiesti dai singoli processi. Al termine dell'operazione del processo che opera sul cilindro n. 20 si pone il problema di come gestire la coda delle richieste pendenti. Una semplice strategia, spesso utilizzata come criterio di scheduling, e che è già stata presentata anche in altri contesti, è la strategia *First-Come-First-Served (FCFS)* che, fra l'altro, ha il privilegio di eliminare ogni condizione di starvation. Purtroppo, si nota subito che questa strategia non risponde a nessun criterio di ottimalità. In particolare, in questo caso, il parametro da ottimizzare per migliorare l'efficienza di uso del dispositivo dovrebbe corrispondere ad una minimizzazione degli spostamenti della testina al fine di ridurre al minimo i tempi di seek. Con riferimento al caso precedente, nella successiva figura 5.18 viene riportato uno schema che illustra quali sarebbero i movimenti della testina se le richieste pendenti fossero servite con l'algoritmo FCFS. Come si può notare, in questo caso, per servire tutte le richieste pendenti, sarebbe necessario spostare la testina su un numero totale di 113 cilindri.

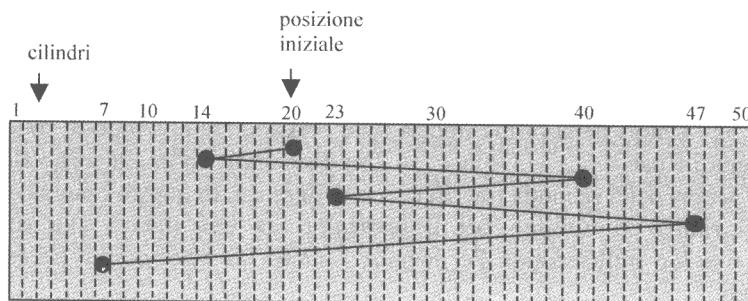


Figura 5.18 Algoritmo di scheduling FCFS.

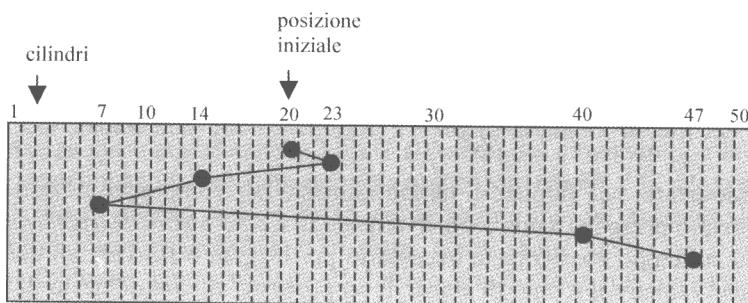


Figura 5.19 Algoritmo di scheduling SSTF.

Un criterio molto più efficiente del FCFS potrebbe essere quello che, alla fine di un’operazione, tra le varie richieste pendenti, sceglie quella che chiede di operare sul cilindro più vicino alla posizione attuale della testina, in modo tale da minimizzare il tempo di seek per ognuna delle operazioni pendenti. Con riferimento al precedente esempio, in questo caso le richieste sarebbero servite secondo il seguente ordine di richiesta dei cilindri: 23, 14, 7, 40, 47. Come si può notare dalla successiva figura 5.19, con questo criterio sarebbe necessario spostare la testina su un numero totale di 59 cilindri, nettamente inferiore al caso precedente. L’algoritmo che sceglie le richieste pendenti secondo questo criterio è noto col nome di *Shortest-Seek-Time-First* (SSTF).

L’algoritmo SSTF fornisce prestazioni sicuramente migliori di quelle FCFS anche se non è detto che il criterio di minimizzare il tempo di seek per ognuna delle richieste pendenti risulti poi in una minimizzazione del tempo medio di seek. Inoltre, tale algoritmo può generare l’inconveniente della *starvation* qualora siano presenti nella coda delle richieste pendenti molte richieste per operare su cilindri tra loro vicini insieme ad una richiesta per operare su un cilindro molto distante da questi. In tale caso verranno servite prima le richieste relative ai cilindri vicini e, se durante il loro servizio arrivano altre richieste anch’esse relative a cilindri vicini ai precedenti, è ovvio che la richiesta per il cilindro più lontano non viene servita e soprattutto non è

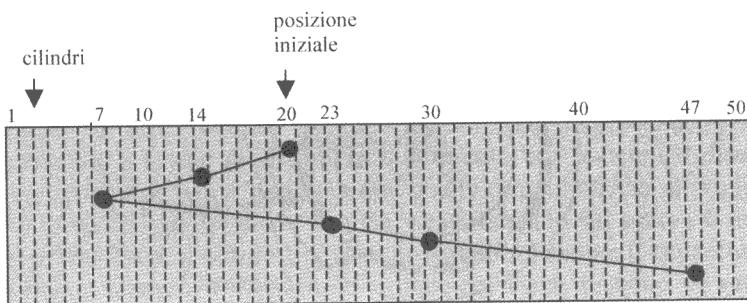


Figura 5.20 Algoritmo di scheduling SCAN.

possibile avere la sicurezza che, prima o poi, venga servita. È per questo motivo che è stato proposto un diverso algoritmo, noto col termine di *SCAN algorithm*. Con questo algoritmo la testina parte da un bordo del disco, per esempio dal cilindro di indice più basso, e serve tutte le richieste pendenti in ordine crescente di cilindri richiesti. Una volta raggiunto l'altro bordo, oppure se non ci sono più richieste pendenti per cilindri relativi a quella direzione di movimento, il movimento della testina viene invertito e vengono servite le richieste pendenti in ordine decrescente di cilindri richiesti. Questo algoritmo viene indicato anche col nome di *algoritmo dell'ascensore*, in quanto simula il comportamento del meccanismo di controllo delle chiamate di un ascensore in un palazzo.

Con riferimento all'esempio precedente, e nell'ipotesi che al termine dell'operazione corrente la direzione di movimento sia quella relativa all'ordine decrescente dei cilindri, le richieste vengono servite nell'ordine: 14, 7. A questo punto non avendo ulteriori richieste pendenti in ordine decrescente di cilindri, la direzione di movimento viene invertita andando a servire le ulteriori richieste pendenti: 23, 40, 47. Come indicato nella figura 5.20, con questo criterio sarebbe necessario spostare la testina su un numero totale di 53 cilindri.

Ulteriori algoritmi sono stati successivamente proposti. Per un dettagliato esame di questi ultimi si rimanda alla bibliografia.

5.4.3 Dischi RAID

Le attuali tecnologie non consentono di ottenere ulteriori miglioramenti delle prestazioni di una memoria di massa oltre quelle a cui si è accennato nel precedente paragrafo. Per migliorare ulteriormente le prestazioni, come è opportuno in tutti quei sistemi che fanno un uso intensivo della memoria di massa, ad esempio i sistemi per la gestione di basi di dati, è quindi necessario ricorrere a tecniche che, utilizzando più dischi contemporaneamente, consentano di operare in parallelo sui singoli dischi. Ciò significa organizzare i dati sui vari dischi in modo tale che sia possibile sfruttare l'implicito parallelismo dovuto alla presenza di più dischi fisicamente indipendenti tra di loro. Inoltre, questo livello di parallelismo consente anche di organizzare i dati sui diversi dischi in modo tale da migliorare l'affidabilità nei confronti di possibili guasti, ridondando i dati su dischi diversi.

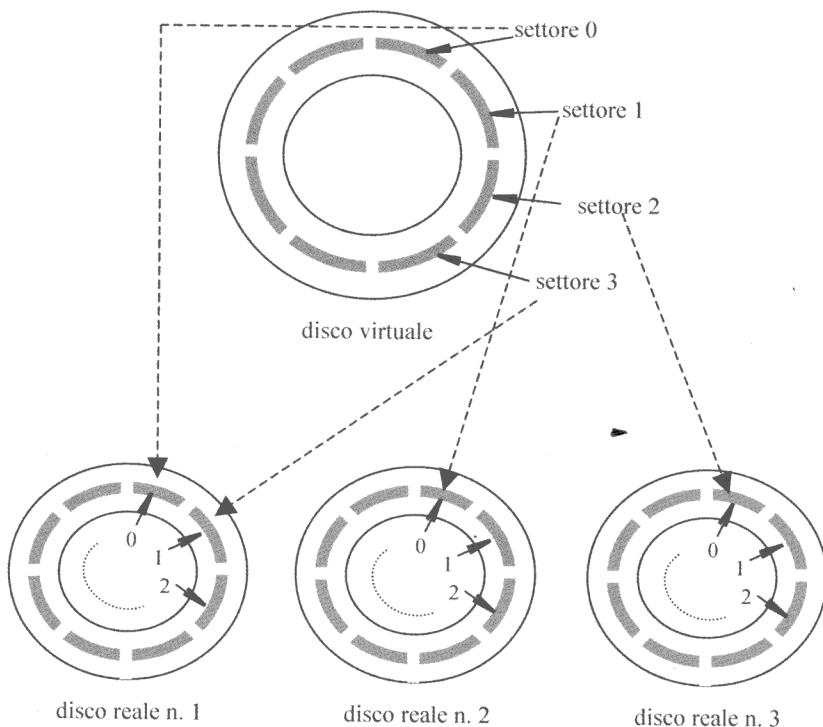


Figura 5.21 Dischi RAID.

Avendo più dischi a disposizione, è possibile concepire diversi modi per replicare i dati al fine di migliorare la robustezza del sistema nei confronti di possibili guasti. Chiaramente, per evitare che un sistema per la gestione di basi di dati venga a dipendere da questi dettagli, che possono variare da un sistema operativo all'altro, sono stati definiti alcuni schemi standard di organizzazione dei dati sui singoli dischi. La definizione di tali standard consente agli utilizzatori di vedere l'insieme dei vari dischi disponibili come se, complessivamente, costituissero un unico disco virtuale, caratterizzato da una grande capacità, da alta velocità di accesso (garantita dagli accessi contemporanei ai dischi componenti), e da alta affidabilità (garantita dal livello di ridondanza dei dati memorizzati su dischi diversi). È ovviamente compito del sistema operativo fornire questo livello di astrazione in modo tale che chi realizza una base di dati non sia costretto a tener conto del numero di dischi fisicamente disponibili e del modo in cui i dati vengono distribuiti e replicati sui singoli dischi.

Lo standard più usato è sicuramente lo standard *RAID (Redundant Array of Independent Disks)*, che definisce il criterio con cui i dati sono distribuiti sui vari dischi di un array di unità indipendenti, al fine di aumentare sia il parallelismo degli accessi che l'affidabilità.

Nella figura 5.21 viene riportato lo schema utilizzato nello standard *RAID* per parallelizzare gli accessi al disco virtuale.

Disponendo di un certo numero n (n è pari a tre nell'esempio di figura 5.21) di dischi reali di capacità limitata e relativamente poco costosi, è possibile realizzare l'astrazione di disco virtuale la cui capacità è pari a n volte quella dei dischi reali e con tempo medio di accesso notevolmente inferiore. Per questo motivo le informazioni residenti sul disco virtuale vengono memorizzate suddividendole sui dischi reali, per esempio a livello di settore. In questo caso una traccia del disco virtuale contiene un numero di settori pari a n volte quello di ogni traccia di un disco reale. Il settore 0 di un traccia del disco virtuale viene mappato sul settore 0 della stessa traccia del primo disco, il settore 1 del disco virtuale sul settore 0 del secondo disco, e così via fino al settore $n - 1$ del disco virtuale che viene mappato sul settore 0 del disco n . Quindi, operando con tecnica *round robin*, si ricomincia dal primo disco reale. Per cui il settore n del disco virtuale viene mappato sul settore 1 del primo disco reale e così via.

Questa tecnica consente di velocizzare tutte le operazioni di I/O che richiedono di operare su un insieme contiguo di settori del disco virtuale, consentendo un livello massimo di parallelismo pari a n .

Lo standard RAID contempla, in realtà, molte varianti che dipendono dal grado di ridondanza dei dati e quindi dal livello di affidabilità, oltre che di rapidità di accesso, richiesta dalle applicazioni. In particolare sono state definite 7 diverse varianti dello standard, identificate come varianti di livello 0, 1, 2, 3, 4, 5 e 6.

Lo standard RAID di livello 0 corrisponde esattamente a quello schematizzato in figura 5.21 il quale, quindi, non prevede nessun livello di ridondanza dei dati.

Il livello 1 corrisponde esattamente al livello 0 per quanto riguarda lo schema di distribuzione dei dati sui dischi fisici, ma prevede che tutti i dischi fisici siano duplicati in modo tale che di ogni disco fisico esistano due copie contenenti esattamente le stesse informazioni (con riferimento alla precedente figura, dovrebbero essere disponibili 6 dischi, i tre precedenti più le loro tre copie). Questa organizzazione (nota col nome di *mirroring*) è in grado di fornire un'alta affidabilità al sistema in quanto, in presenza di un guasto su uno dei dischi è comunque possibile continuare ad utilizzare la sua copia. La presenza delle copie non comporta inoltre nessuna perdita di efficienza in quanto le operazioni di lettura possono procedere operando su uno qualunque dei due dischi contenenti il settore richiesto (normalmente quello fra i due che richiede un tempo di seek inferiore). Le operazioni di scrittura possono procedere in parallelo sui due dischi contemporaneamente. Inoltre, avendo due copie di un disco, è possibile anche operare in parallelo letture di due settori residenti sullo stesso disco fisico, chiaramente leggendo in parallelo i due diversi settori l'uno da un disco l'altro dal disco contenente la copia.

Questo schema è sicuramente ottimo sia per quanto riguarda l'affidabilità che la velocità di accesso. Per contro, ha l'inconveniente di essere molto costoso a causa dell'elevato numero di dischi richiesti. Per questo motivo sono state proposte molte varianti (corrispondenti ai livelli 2, 3, 4, 5, e 6) che si differenziano dal precedente schema per il livello di ridondanza richiesto e per i meccanismi di rilevazione e recupero da guasti (meccanismi di controllo di parità o schemi di codifica Hamming).

Ad esempio, nello schema corrispondente al livello 5 (noto col nome di *block interleaved parity*) il livello di ridondanza è nettamente inferiore rispetto allo schema di *mirroring*. In questo caso viene utilizzata una piccola frazione dello spazio su

ogni disco per contenere dei controlli di parità. Ad esempio, se vengono utilizzati n dischi, per ogni gruppo di n settori consecutivi, memorizzati sugli n dischi dell'array, viene calcolato un ulteriore $n + 1$ esimo settore contenente bit di parità. In pratica, il primo bit del settore di parità corrisponde al bit di parità calcolato sui primi bit degli n settori regolari. Analogamente vale per gli altri bit di ogni settore. I settori di parità vengono allocati su dischi consecutivi in modo circolare (round robin). In questo caso, se uno qualunque dei settori di un disco si corrompe per un guasto, il suo contenuto può essere recuperato utilizzando il corrispondente settore di parità.

Per chi volesse esaminare in dettaglio gli schemi relativi ai livelli superiori dello standard RAID si rimanda alla bibliografia.

5.5 Sommario

La gestione dei dispositivi periferici costituisce una delle componenti di un sistema operativo che nei libri di testo viene spesso relegata fra le parti meno sviluppate, pur costituendo una frazione dell'intero sistema operativo sicuramente molto cospicua in termini di linee di codice. Ciò è principalmente dovuto alla grande variabilità dei dispositivi periferici che possono essere collegati a un sistema di elaborazione e all'elevato numero di dettagli che caratterizzano le interfacce di collegamento fra i dispositivi e il sistema, dettagli che rendono complessa una trattazione esaustiva dell'argomento.

In questo paragrafo abbiamo cercato di esemplificare alcune caratteristiche comuni presenti in molte interfacce relative a alcune categorie di dispositivi, sia per quelli orientati al trasferimento di singoli caratteri, sia per quelli orientati al trasferimento di blocchi di dati.

Sono stati illustrati alcuni semplici esempi di come nel kernel del sistema può essere realizzato il driver per un dispositivo caratterizzandone il funzionamento sia secondo il paradigma a controllo di programma che a interruzione e, per i dispositivi connessi mediante appositi canali, secondo lo schema di collegamento in DMA.

È stato inoltre presentato lo schema organizzativo del software di sistema costituenti l'intero sottosistema di I/O secondo un'organizzazione a livelli gerarchici così come viene realizzato nella maggior parte dei sistemi operativi.

Alcuni aspetti dei livelli superiori di questo schema saranno ripresi anche nel capitolo relativo al file system e, per quanto riguarda i dispositivi di rete, nel capitolo relativo ai sistemi distribuiti.

5.6 Note bibliografiche

Gli argomenti relativi agli aspetti architetturali del sottosistema d'ingresso/uscita possono essere approfonditi su un testo di architettura dei sistemi di elaborazione, come ad esempio in [38], [22], [73], [84]. Una descrizione generale sulla struttura e sui diversi compiti svolti dal sottosistema d'ingresso/uscita nell'ambito di un sistema operativo si può trovare su alcuni testi di sistemi operativi quali [85], [79], oppure su testi dedicati a questo argomento, come [78].

In [99] si trova un'approfondita descrizione dei dettagli interni del sottosistema di I/O di Unix, mentre in [26] e in [83] si può trovare un'analogia descrizione relativa

a Windows NT. In [96] è riportata un'approfondita descrizione del sottosistema di I/O di Minix, un sistema operativo utilizzato per scopi didattici.

In [33] e in [79] viene descritto come realizzare un device driver in sistemi Unix. Analoghe descrizioni con riferimento al sistema Windows si trovano in [16] e in [68].

Un'approfondita descrizione delle tecnologie per le memorie di massa si trova in [63]. Una completa descrizione degli standard RAID si può trovare in una pubblicazione della RAID Advisory Board, un'associazione che comprende fornitori e utilizzatori di dischi realizzati secondo gli schemi RAID [62].

6

Il File System

Per garantire ad ogni processo l'accesso a grandi insiemi di dati e a informazioni che sopravvivano alla sua terminazione, o anche all'interruzione dell'alimentazione elettrica, è necessario sfruttare dispositivi di memorizzazione con caratteristiche di persistenza e di grande capacità: nel primo capitolo è stato mostrato che i dispositivi di memoria secondaria assolvono proprio a questi requisiti.

Se, quindi, dal lato hardware è necessario disporre di queste particolari periferiche (come ad esempio i dischi, ai quali faremo implicitamente riferimento nel seguito del capitolo), dal lato software occorrono astrazioni e meccanismi che consentano la rappresentazione, l'archiviazione e l'accesso ai dati immagazzinati all'interno della memoria secondaria. Il sistema software che soddisfa questi requisiti è il *file system*, al quale è dedicato questo capitolo.

6.1 Organizzazione del file system

Il file system è quella parte del sistema operativo che fornisce i meccanismi necessari per l'accesso e l'archiviazione delle informazioni in memoria secondaria.

La struttura di un file system può, in generale, essere rappresentata (vedi figura 6.1) da un insieme di componenti organizzate in vari livelli.

Il livello più alto (*Struttura logica*) presenta all'utente una visione astratta delle informazioni presenti sul disco, che prescinde dalle caratteristiche del dispositivo, e dalle tecniche di allocazione e di accesso alle informazioni adottate dal sistema.

In particolare, le informazioni vengono generalmente memorizzate come *file* o insiemi di *file*, che possono essere collocati all'interno di "contenitori" detti *direttori* (o *cartelle*). La struttura astratta che viene presentata alle applicazioni appare quindi come un'organizzazione di file e direttori, che, come verrà illustrato nel paragrafo 6.2.2, può avere struttura e caratteristiche diverse da sistema a sistema.

Processi (e utenti) vedono quindi la memoria secondaria attraverso la *struttura astratta* presentata dal *file system*. I processi possono accedere a questa struttura logica, utilizzando le *system call* specifiche offerte dal sistema.

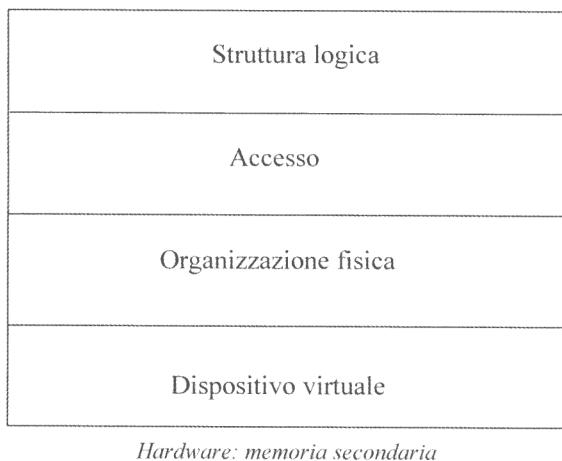
Applicazioni

Figura 6.1 Organizzazione del file system.

Il livello sottostante (*Accesso*), definisce e realizza i meccanismi mediante i quali è possibile eseguire operazioni sul contenuto dei file. L'accesso a file, in generale, può avere due diverse finalità: *leggere* (cioè, estrarre informazioni) oppure *scrivere* il file (cioè, aggiungere nuove informazioni). A questo livello ogni file è visto come un insieme di *record logici*.

Il *record logico* è l'unità di trasferimento tra processo e file: esso rappresenta, cioè, l'unità di informazione che ogni processo può estrarre (nel caso di lettura) o depositare (nel caso di scrittura) da o in ogni file a cui si accede.

Il record logico è caratterizzato da alcune proprietà come il *tipo* e la *dimensione*, che dipendono dal sistema operativo: ad esempio, nel sistema Unix, il record logico è il byte (vedi paragrafo 7.6.4).

In generale, esistono diversi meccanismi per accedere a un file (realizzati dai *metodi di accesso*, vedi paragrafo 6.3) ognuno dei quali sottintende una particolare organizzazione dei record logici all'interno del file stesso. Ad esempio, l'insieme dei record logici appartenenti a un file può essere organizzato *sequenzialmente* oppure in modo *casuale*; il metodo di accesso dipende strettamente da questa organizzazione: per esempio, nel primo caso si parla di *accesso sequenziale*, mentre nel secondo caso, non essendoci una relazione d'ordine prefissata tra i record logici, l'accesso è di tipo *casuale* o *diretto*. In generale, ogni sistema operativo compie delle scelte riguardo ai metodi di accesso, che (come si vedrà nel paragrafo 6.3.2) derivano da considerazioni relative a vari parametri: tipicamente l'efficienza nell'accesso, la gestione ottimizzata dello spazio disponibile, ed inoltre la robustezza del file system nei confronti di guasti.

Il livello di *Accesso* realizza le operazioni elementari di accesso ai file (lettura e scrittura) conformemente ai metodi di accesso scelti; tali operazioni vengono rese disponibili ai processi mediante opportune *system calls*.

Inoltre, va evidenziato che in un sistema multiutente l'accesso di un processo ad un file è solitamente subordinato al soddisfacimento delle politiche di protezione, che stabiliscono *chi* (cioè, quali utenti) ed *in che modo* è abilitato ad accedere al file. È ancora compito del livello di *Accesso* realizzare i meccanismi volti a controllare che ogni tentativo di accesso ad un file/direttorio avvenga in conformità con i vincoli di protezione applicati ad esso. I meccanismi di protezione più comunemente adottati nei sistemi operativi verranno illustrati nel paragrafo 6.3.3.

Lo strato sottostante al livello di Accesso è rappresentato dal livello di *Organizzazione fisica*. Il suo compito primario è allocare i record logici di ogni file nell'unità di memorizzazione secondaria. A questo livello, lo spazio disponibile per l'allocazione sul disco viene visto come un insieme di *blocchi fisici*.

Il *blocco fisico* (o, più semplicemente, *blocco*) è l'unità di allocazione e di trasferimento delle informazioni sul dispositivo; ad ogni blocco è quindi assegnata una posizione particolare sulla superficie del disco. Il livello di Organizzazione fisica realizza quindi opportuni meccanismi (*i metodi di allocazione*) che stabiliscono il collegamento tra ogni file (considerato come insieme di record logici) e l'insieme dei blocchi fisici sui quali esso è allocato. Nel paragrafo 6.4 verranno descritti i vari approcci utilizzabili per l'allocazione di file, sottolineando per ognuno di essi vantaggi e svantaggi.

Non tutto lo spazio disponibile sul disco viene utilizzato per l'allocazione dei file: come verrà approfondito nel seguito, infatti, il sistema mantiene alcune strutture dati per la descrizione della struttura logica del file system e per supportare la gestione degli accessi ai file da parte di processi (vedi paragrafo 6.3.1). Il livello di Organizzazione fisica, quindi, deve occuparsi anche dell'allocazione delle strutture dati necessarie al sistema per la rappresentazione e la gestione di file e direttori. A questo scopo, lo spazio di memorizzazione viene partizionato in aree separate per la memorizzazione dei file e per l'allocazione di tali strutture dati.

È possibile, infine, individuare un ulteriore livello (il livello di *Dispositivo virtuale*) che si "appoggia" direttamente sull'hardware e che si occupa di partizionare lo spazio disponibile nel dispositivo in un insieme di blocchi fisici di dimensione costante, offrendo al livello superiore (*Organizzazione fisica*) una visione dello spazio disponibile come un vettore lineare di blocchi fisici, come è stato indicato nel capitolo 5.

A parte, quindi, le semplici funzioni svolte dal livello più basso (*Dispositivo virtuale*) già illustrate precedentemente (vedi paragrafo 5.4.1), nel seguito di questo capitolo verranno approfonditi gli aspetti più importanti dei tre livelli superiori (più astratti), mostrando le tecniche più diffuse nei moderni sistemi per la realizzazione delle funzionalità di ogni componente.

6.2 La struttura logica del file system

Iniziando, quindi, dal livello più alto, viene descritta adesso la struttura logica del file system.

6.2.1 Il file

Il file costituisce l'unità logica di memorizzazione all'interno del *file system*. In pratica, esso rappresenta un contenitore di informazioni.

Le informazioni contenute nei file possono essere di varia natura: ad esempio, un file può essere impiegato per contenere del testo, un insieme di dati, un programma eseguibile, immagini, filmati, suoni, e altro ancora.

Ogni file è individuato all'interno della struttura logica realizzata dal file system, da (almeno) un nome simbolico mediante il quale può essere riferito (ad esempio, nell'invocazione di comandi o di *system call*).

Oltre al nome, il file è caratterizzato anche da un insieme di attributi. Gli attributi del file possono variare a seconda del sistema operativo; tuttavia, alcuni di essi ricorrono in molti sistemi operativi. Ad esempio:

- il *tipo*: è un attributo che stabilisce l'appartenenza del file a una classe; per esempio, può essere utilizzato per indicare il tipo di informazioni contenute nel file (eseguibile, batch, testo, ecc.). Talvolta al tipo vengono associate particolari convenzioni riguardanti il nome del file: per esempio, nel sistema Windows il nome dei file contenenti codice binario eseguibile, ha estensione “.exe”, mentre i file contenenti comandi di shell hanno nomi con estensione “.bat” ecc.;
- gli *indirizzi*: sono i puntatori che riferiscono direttamente o indirettamente i blocchi utilizzati per l'allocazione del file in memoria secondaria (vedi paragrafo 6.4.1);
- la *dimensione*: esprime il numero dei byte contenuti nel file;
- la *data* e l'*ora* di creazione: registrano l'istante in cui il file è stato creato;
- la *data* e l'*ora* di modifica: registrano l'istante più recente in cui il file è stato modificato.

Nei sistemi operativi multiutente è importante attribuire al file anche le informazioni relative all'utente che possiede il file, e le eventuali politiche di protezione da applicare ad esso; a questo scopo vengono spesso introdotti i seguenti attributi:

- *proprietario*: è un attributo che individua univocamente l'utente proprietario del file;
- *protezione*: esprime le politiche di accesso da applicare al file nei confronti di tutti gli utenti del sistema¹.

Gli attributi del file vengono registrati all'interno di una particolare struttura dati associata al file, detta *descrittore del file* (vedi paragrafo 6.3.1).

6.2.2 Il directory

Il directory è un'astrazione che consente di raggruppare più file. In pratica, quindi, un directory rappresenta contenitore di files.

Talvolta il directory può contenere, a sua volta, altri directory: in questo caso esso può essere visto come un operatore di composizione che consente di organizzare la struttura logica del file system come una composizione di file e directory.

¹ Come verrà illustrato nel seguito, questo non è l'unico modo per specificare i vincoli di accesso a una risorsa (in questo caso, il file). Nel paragrafo 6.3.3, questo tema verrà approfondito mostrando gli approcci più comuni per l'esplicitazione delle politiche di protezione.

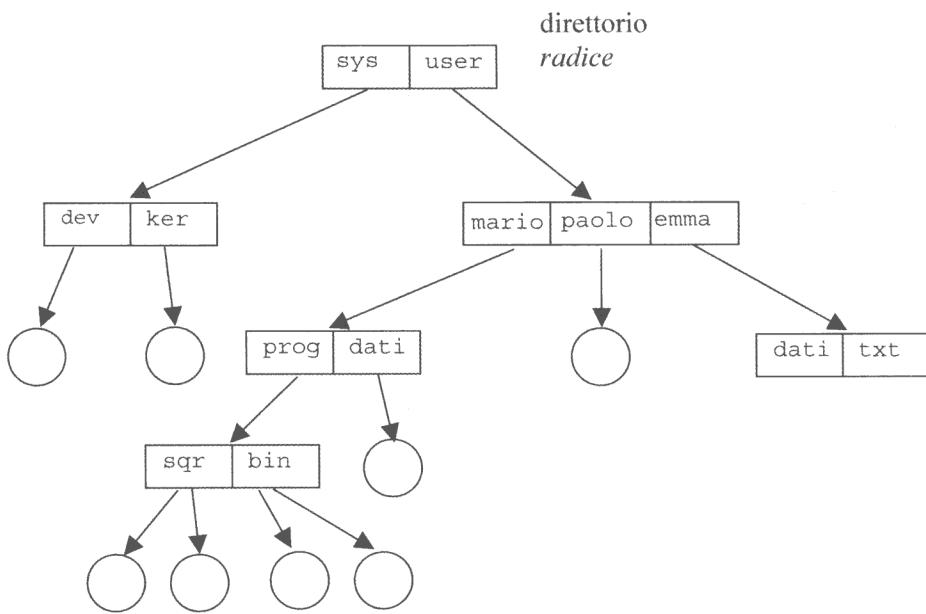


Figura 6.2 Esempio di file system ad albero.

La struttura logica del file system è pertanto un'aggregazione di file e direttori le cui caratteristiche dipendono quindi in modo stretto dalle caratteristiche del direttorio stesso.

Nei sistemi operativi moderni la soluzione più comune è rappresentata da file system *strutturati ad albero*: in questo caso un direttorio può contenere sia file che altri direttori. In particolare, ogni file è contenuto in un direttorio; ogni direttorio, tranne uno (il direttorio *radice*), appartiene ad un altro direttorio. È possibile rappresentare una struttura logica di questo tipo mediante un grafo ad albero *n*-ario (con *n* qualunque). Un esempio di file system ad albero è mostrato in figura 6.2: nello schema le caselle rettangolari rappresentano i direttori, mentre i cerchi rappresentano i file; gli archi descrivono la relazione di appartenenza ad un direttorio. Si può facilmente osservare che nel caso di file system ad albero si viene a creare una struttura logica organizzata in modo gerarchico, nella quale, al vertice della gerarchia, compare il direttorio *radice*.

In alcuni sistemi (per esempio Unix, vedi paragrafo 7.6.1), viene permesso a più direttori di condividere lo stesso file: questo meccanismo, che viene indicato con il termine “*linking*”, modifica la struttura logica del file system, che, pur mantenendo una rigida organizzazione gerarchica, diventa un grafo diretto aciclico (*Direct Acyclic Graph*, o *DAG*). Un esempio è rappresentato dalla struttura mostrata in figura 6.3, nella quale il file *f* appartiene a due direttori distinti.

Analogamente a quanto detto nel paragrafo 6.2 a proposito del file, anche ad ogni direttorio (o cartella) viene associato un descrittore che ne contiene gli attributi.

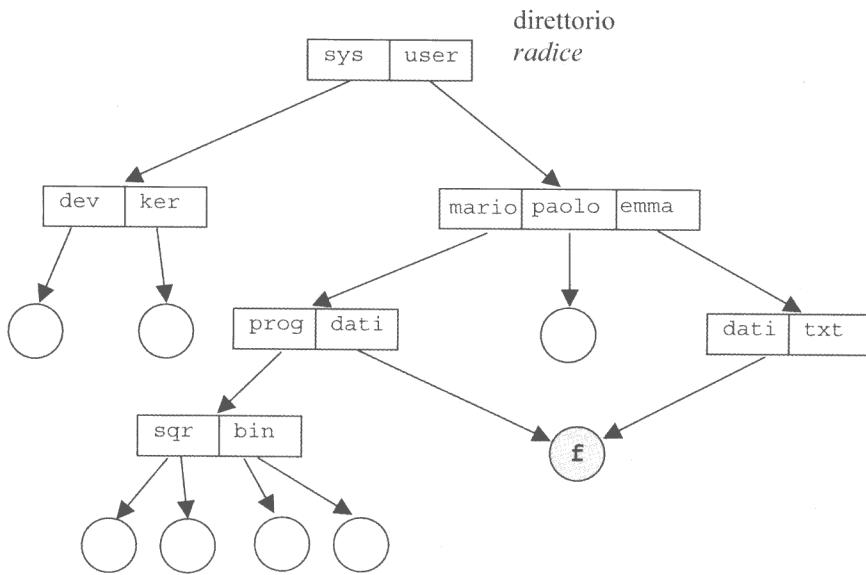


Figura 6.3 File system a grafo orientato aciclico.

In molti sistemi operativi (ad esempio Unix, vedi paragrafo 7.6) file e directory vengono rappresentati in modo omogeneo, cioè il directory viene rappresentato come un particolare tipo di file. In questo caso, quindi, file e directory sono descritti da un unico insieme omogeneo di descrittori di file.

6.2.3 Gestione della struttura logica del file system

Il sistema operativo realizza anche i meccanismi per la gestione del file system, che rende disponibili alle applicazioni mediante specifiche *system call*. Le operazioni fondamentali per la gestione del *file system* riguardano sia file che directory; tipicamente ogni sistema realizza almeno le seguenti funzioni di base:

- *Creazione e cancellazione* di directory: mediante queste operazioni è possibile apportare modifiche alla struttura logica del file system, aggiungendo/eliminando rami al grafo che rappresenta il file system.
- *Aggiunta/cancellazione* di file: mediante queste operazioni è possibile memorizzare/eliminare file.
- *Listing*: questa operazione consente di ispezionare il contenuto di uno (o più) directory, tramite la visualizzazione dell'elenco di tutti i file contenuti nel directory considerato.
- *Attraversamento* del directory: questa operazione consente di “navigare” attraverso la struttura logica del file system, spostando la posizione corrente da un directory di origine ad uno di destinazione.

Funzionalità analoghe sono disponibili per gli utenti sotto forma di comandi di *shell*.

6.3 Accesso al file system

Il secondo livello del file system riguarda la realizzazione delle varie modalità di accesso ai file.

6.3.1 Strutture dati e operazioni per l'accesso ai file

Nel paragrafo 6.2.1 è stato mostrato che ogni file è caratterizzato da un nome e da un insieme di attributi. L'insieme degli attributi di ogni file viene mantenuto all'interno di una particolare struttura dati, detta *descrittore del file*. Come il file, anche il descrittore deve avere caratteristiche di persistenza, e pertanto anch'esso viene mantenuto in memoria secondaria.

Poiché ogni file appartiene ad un direttorio, ogni direttorio necessita del collegamento con i descrittori dei file che gli appartengono.

Per soddisfare questo requisito, alcuni sistemi (per esempio, Windows) realizzano il direttorio come una struttura dati di tipo tabellare (vedi figura 6.4) che contiene un elemento per ogni file; ogni elemento include due campi: il nome del file ed il suo descrittore. In questo modo, i descrittori dei file presenti nel file system sono distribuiti all'interno delle strutture dati che implementano i direttori.

Una soluzione alternativa (adottata ad esempio in Unix) è mostrata in figura 6.5: essa prevede invece che l'insieme dei descrittori di tutti i file sia concentrato in un'apposita struttura dati globale (la *tabella dei descrittori dei file*). In questo caso, quindi, il direttorio viene rappresentato da una struttura dati che ad ogni nome di file associa il riferimento ai suoi attributi nella tabella dei descrittori.

Il secondo approccio presenta indubbiamente una minore efficienza rispetto al primo nel reperimento degli attributi associati ad un file, ma fornisce la possibilità di realizzare in modo semplice il meccanismo di *linking* tra file (vedi paragrafo 6.2.2): se, infatti, due direttori contengono lo stesso file (eventualmente riferito con nomi diversi) la condivisione può essere realizzata inserendo nei due direttori il riferimento allo stesso descrittore.

Uno dei compiti del file system è consentire l'accesso *on-line* ai file. Come è già stato mostrato, i processi possono accedere ai file in vari modi. Le modalità più comuni sono:

- *lettura*: al file si accede con l'obiettivo di estrarre informazioni contenute in esso.

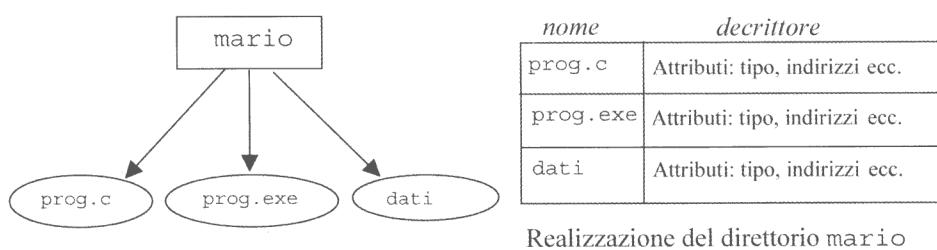


Figura 6.4 Realizzazione del direttorio: il descrittore fa parte del direttorio.

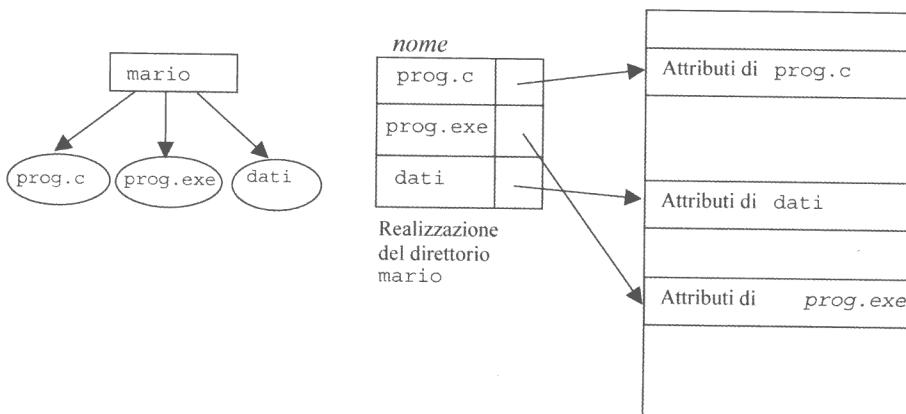


Figura 6.5 Realizzazione del direttorio: i descrittori sono concentrati nella Tabella dei descrittori.

- *scrittura*: l’accesso determina l’immissione nel file di nuove informazioni; in questo caso il contenuto del file prima dell’accesso in scrittura viene perso, e sostituito con le nuove informazioni scritte; questa è l’operazione mediante la quale, ad esempio, si inizializza il contenuto di un nuovo file.
- *scrittura in aggiunta (append)*: anche in questo caso vengono immesse nuove informazioni all’interno del file, ma in modo più “conservativo”: i record logici scritti vanno ad aggiungersi a quelli già presenti nel file.

Indipendentemente dalle scelte del sistema riguardo ai metodi di accesso, ognuna di queste operazioni richiederebbe preventivamente l’accesso al descrittore del file, che, come già detto, è memorizzato sul disco. Ad esempio, l’accesso in lettura di un processo P ad un file f richiede la localizzazione dei blocchi sui quali è allocato il record desiderato, mediante gli indirizzi contenuti proprio nel descrittore. Come è facile intuire, a queste condizioni il costo di ogni operazione di accesso sarebbe troppo oneroso.

Per questo motivo il sistema mantiene in memoria centrale una struttura dati (la *tabella dei file aperti*) che contiene alcune informazioni associate ai file attualmente in uso; tipicamente:

- una copia del descrittore del file;
- il puntatore al prossimo record logico da leggere/scrivere (nel caso di accesso sequenziale);
- le informazioni relative al processo che accede al file.

Il trasferimento di queste informazioni in memoria centrale riduce notevolmente il tempo di esecuzione di ogni singola operazione di accesso.

L’inserimento di un nuovo elemento all’interno della *tabella dei file aperti* viene provocata dall’esecuzione dell’operazione di *apertura* del file, che deve sempre essere effettuata prima di ogni sessione di accesso a file. Analogamente, ogni sessione

di accesso a file deve essere conclusa dall'operazione di *chiusura*, mediante la quale si elimina dalla tabella dei file aperti l'elemento corrispondente.

Per aumentare ulteriormente l'efficienza, molti sistemi adottano inoltre un meccanismo di *memory mapping* dei file: ogni file aperto (o la sua parte correntemente aperta) viene temporaneamente copiato in memoria centrale. Grazie a questo accorgimento, ogni operazione di lettura/scrittura può agire sulla copia *memory mapped* del file, conseguendo così tempi di accesso più veloci.

In definitiva, gli effetti dell'operazione di *apertura* di un file sono i seguenti:

- inserimento di un nuovo elemento nella *tabella dei file aperti*;
- eventuale *memory mapping* del file.

In modo simmetrico, gli effetti dell'operazione di *chiusura* di un file sono:

- salvataggio del file *memory mapped* sul disco;
- eliminazione dell'elemento corrispondente dalla *tabella dei file aperti*.

6.3.2 Metodi di accesso

Il metodo di accesso stabilisce le modalità con le quali i processi possono leggere o scrivere i file. Ogni metodo di accesso presuppone implicitamente una particolare organizzazione interna del file, che a questo livello viene visto come un insieme di record logici numerati $\{R_1, R_2, \dots, R_N\}$.

Il record logico è l'unità di lettura/scrittura sul file.

Come verrà mostrato, ogni metodo di accesso è, in generale, indipendente sia dal tipo di dispositivo utilizzato che dalla tecnica di allocazione dei blocchi in memoria secondaria.

I metodi di accesso più diffusi sono:

- l'accesso *sequenziale*;
- l'accesso *diretto*;
- l'accesso a *indice*.

Metodo di accesso sequenziale Nel caso di modalità di accesso sequenziale, si assume che ogni file sia organizzato come una sequenza di record logici.

Pertanto, all'insieme dei record logici contenuto in ogni file viene applicata una relazione d'ordine mediante la quale è possibile individuare univocamente il primo record della sequenza, il secondo, e così via.

Il metodo di accesso sequenziale prevede che i record logici appartenenti al file possano essere letti/scritti esattamente nell'ordine prefissato. Ad esempio, in una sessione di lettura, il primo record letto sarà il primo della sequenza; successivamente verrà letto il secondo, poi il terzo ecc.

Quindi, per accedere ad un particolare record logico R_i del file, è necessario accedere prima agli $(i - 1)$ record che lo precedono nella sequenza.

A questo scopo, durante una sessione di accesso (in lettura o scrittura) ad un file F , il sistema registra la posizione del prossimo elemento della sequenza in un *puntatore* (detto *puntatore a file* o *I/O pointer*).

Le operazioni di accesso sequenziale, realizzate dal sistema operativo come chiamate di sistema, sono:

- `readnext`, per leggere il prossimo record logico;
- `writenext`, per scrivere il prossimo record logico.

Entrambe, quindi, fanno riferimento implicitamente all'I/O pointer, per individuare il record logico al quale accedere.

In particolare, l'operazione `readnext(f, &V)` provoca un duplice effetto:

- assegna il valore del prossimo record logico del file `f` alla variabile `V`;
- posiziona il puntatore al file `f` sull'elemento successivo a quello letto.

Analogamente, l'operazione `>writenext(f, V)` provoca i seguenti effetti:

- scrive nel prossimo record logico del file `f` il valore della variabile `V`;
- posiziona il puntatore al file `f` sull'elemento successivo a quello scritto.

Accesso diretto La modalità di accesso diretto permette la lettura/scrittura di un qualunque record logico del file con operazioni del tipo:

- `readd(f, i, &V)`: lettura del record logico R_i dal file `f`; il valore letto viene assegnato alla variabile `V`;
- `>writed(f, i, V)`: scrittura del valore della variabile `V` nel record logico R_i del file `f`.

A differenza del metodo di accesso sequenziale, quindi, per leggere/scrivere un particolare record logico non è necessario accedere prima a tutti i record che lo precedono. Questo, soprattutto se il record da accedere si trova alla fine di una lunga sequenza, implica evidenti vantaggi dal punto di vista della semplicità di programmazione e talvolta anche dal punto di vista dell'efficienza².

Riguardo all'aspetto di programmazione, si consideri, per esempio, il caso di un processo che desidera leggere un particolare record logico R_k da un certo file aperto `f`. Se il metodo di accesso è sequenziale, la sequenza di istruzioni da eseguire sarà del tipo:

```
for (i=1; i<k; i++)
    readnext(f, &V); /* lettura dei record precedenti Rk */
readnext(f, &V); /* lettura del record Rk */
```

Se, invece, il metodo di accesso adottato è diretto, la lettura di R_k si può specificare in modo più sintetico con l'unica istruzione:

```
readd(f, k, &V);
```

Questo metodo è quindi particolarmente conveniente quando si vuole accedere a file di grandi dimensioni per estrarre o aggiornare soltanto poche informazioni, come ad esempio nelle applicazioni di gestione di grossi *databases*.

Va infine osservato che, a differenza dell'accesso sequenziale, in questo caso non è necessario mantenere un puntatore al file aperto perché l'accesso a un particolare record logico avviene in modo del tutto indipendente dalla posizione dell'ultimo record logico letto o scritto.

² Infatti, come si vedrà nel paragrafo 6.4, l'efficienza nell'accesso diretto dipende in larga misura dal metodo di allocazione adottato nel sistema.

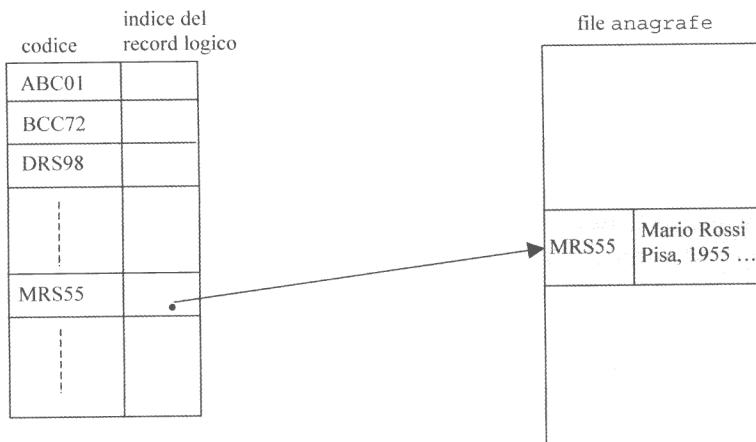


Figura 6.6 Accesso a indice.

Accesso a indice Questo metodo prevede che ogni record logico sia strutturato in almeno due campi, uno dei quali contiene un'informazione (la *chiave*) che serve ad identificare univocamente il record all'interno del file. Ad ogni file inoltre viene associata una struttura dati tabellare detta *indice*, nella quale vi è un elemento per ogni chiave che contiene il riferimento al record corrispondente nel file. In questo modo è possibile accedere a un particolare record del file specificandone la chiave, mediante le operazioni:

- `readk(f, key, &V)`: lettura del record logico con chiave uguale a `key` dal file `f` nella variabile `V`;
- `writek(f, key, V)`: scrittura del valore della variabile `V` nel record logico con chiave `key`.

La *chiave*, quindi consente di accedere direttamente al record desiderato, previa ricerca associativa nell'*indice* del file. Per esempio, si veda la figura 6.6: l'accesso al file anagrafe avviene in modo indicizzato sulla chiave `codice`. A questo scopo è associato al file l'indice mostrato in figura, nel quale ad ogni particolare `codice` è associato il riferimento al record corrispondente.

Questo metodo è presente in alcuni importanti sistemi operativi, come ad esempio il VAX VMS [53].

6.3.3 Protezione di file e direttori

La protezione è un importante requisito dei sistemi operativi multiutente: essa permette di imporre vincoli di accesso ai file e ai direttori presenti nel file system, per impedire l'esecuzione di operazioni non autorizzate su tali risorse. In generale, la protezione si realizza sia attraverso le *politiche*, mediante le quali si esplicitano i vincoli di accesso a file e direttori del sistema, sia mediante i *meccanismi* che consentono la rappresentazione e la verifica delle politiche.

In questo paragrafo vengono approfonditi alcuni aspetti legati alla protezione del file system e più in particolare ai meccanismi generalmente utilizzati nei sistemi operativi per la specifica di politiche di protezione; non verranno trattate, invece, le politiche, il cui progetto è solitamente affidato agli utenti proprietari dei file e direttori da proteggere.

Il ruolo svolto dai meccanismi di protezione è duplice:

- *Rappresentazione delle politiche*: ovvero, realizzazione delle strutture dati che contengono la specifica dei vincoli di accesso sulle risorse del sistema;
- *Controllo degli accessi*: cioè, verifica che ogni accesso di un utente a una data risorsa avvenga in conformità con la politica di protezione specificata per essa.

Con lo scopo di illustrare quali sono gli approcci più comuni per la rappresentazione delle politiche di protezione, è necessario introdurre preliminarmente i concetti che concorrono alla definizione delle stesse politiche: la *risorsa* ed il *dominio di protezione*.

Le *risorse* sono gli oggetti da proteggere: nell'ambito di questo capitolo identifieremo il concetto di *risorsa* con quello di file e di direttorio; ciò non toglie, tuttavia, che la risorsa possa rappresentare anche altri oggetti come dispositivi, segmenti di memoria, processi, porte ed altro ancora. Ad ogni risorsa è associato un utente proprietario che ha il potere di concedere o negare ad altri utenti il permesso di accedere ad essa.

Il *dominio di protezione* viene definito come un insieme di coppie *<risorsa, diritti>* [86]: nella maggior parte dei casi esso è associato univocamente ad un utente e rappresenta l'insieme delle risorse alle quali esso è abilitato ad accedere. In particolare, per ogni risorsa viene specificato un insieme di *diritti*, ognuno dei quali rappresenta il permesso di eseguire una certa operazione sulla risorsa data. Come è noto (vedi paragrafo 6.3.1), le operazioni di base su file e direttori sono la *lettura*, la *scrittura*. In aggiunta a queste, molti sistemi operativi introducono anche l'operazione di *esecuzione*, utilizzabile su file contenenti codice eseguibile. Negli esempi che seguiranno verranno quindi considerati i tre diritti di *lettura*, *scrittura* ed *esecuzione*; tuttavia la trattazione può essere facilmente estesa anche ad altri diritti (come ad esempio, la *cancellazione*).

Ogni processo, in ogni istante della sua esecuzione, è associato ad un particolare dominio (ad esempio, quello dell'utente che ha richiesto la creazione del processo): il processo può quindi accedere alle risorse appartenenti al dominio e con le modalità stabilite dai diritti specificati per esse. In alcuni sistemi operativi è previsto inoltre che un processo possa eventualmente cambiare dominio durante la sua esecuzione: questo significa che l'insieme di risorse sulle quali è abilitato ad operare viene dinamicamente sostituito da quelle contenute nel nuovo dominio, con nuovi diritti di accesso.

Un esempio di questo meccanismo è fornito da Unix (vedi paragrafo 7.6.2), dove, per default, è previsto che ad ogni processo vengano associati lo *userid* e il *groupid* dell'utente che ha richiesto la creazione del processo. Tuttavia, come si vedrà nel paragrafo 7.6.2, un processo P può cambiare dominio: infatti, se P passa ad eseguire un nuovo programma C (mediante la *system call exec*, vedi paragrafo 7.4.3) e

il file eseguibile contenente il codice C ha il bit *SUID* con valore 1, P passa nel dominio associato all'utente proprietario del file eseguibile.

Fatte queste premesse, la rappresentazione delle politiche di protezione da applicare alle risorse richiede che il sistema mantenga in opportune strutture dati le relazioni tra le risorse e i domini. Questo è possibile, a livello concettuale, mediante la definizione di una *matrice di protezione*, come quella mostrata in tabella 6.1.

Si può notare che ogni riga della matrice è associata a un *dominio* di protezione (un certo utente), e ogni colonna è associata a una *risorsa* (un file, un directory o un dispositivo): pertanto la coppia (D, R) individua nella matrice l'insieme dei permessi concessi all'utente D sulla risorsa R. In questo modo si ha la massima flessibilità nella specifica dei diritti per ogni file e per ogni utente. Ad esempio, File2 è leggibile e scrivibile per l'utente Elena, ed è soltanto leggibile per Mario; all'utente Paola, invece, non è concesso alcun tipo di accesso a File2. Nell'esempio si può inoltre osservare che le politiche di protezione possono essere applicate anche a oggetti diversi da file e directory, come le periferiche: in questo caso l'accesso alla stampante è concesso soltanto agli utenti Mario e Elena; data la natura della periferica, l'unico tipo di permesso previsto per l'accesso alla stampante è la scrittura.

	File1	File2	File3	File4	Dir01	Dir02	Stampante
Utente Mario	r-w	r				r-w	w
Utente Paola				r-w-x	r		
Utente Elena		r-w	r-w-x	r		r-w	w

Tabella 6.1 La matrice di protezione.

Chiaramente, la matrice proposta dalla tabella 6.1 ha scopo puramente esemplificativo e per questo motivo le sue dimensioni sono molto ridotte rispetto alle matrici di protezione di file system reali. Infatti, il numero di colonne (che rappresenta il numero di file e directory presenti nel file system) nella realtà è molto più elevato, visto che i file archiviati in un sistema possono ammontare a migliaia o addirittura a milioni. Analogamente, gli utenti possono essere molto numerosi, determinando così un alto numero di righe nella matrice di protezione. Inoltre ogni utente può generalmente accedere solo a un ridotto sottoinsieme dei file: la matrice di protezione, in questo caso, è pertanto una matrice sparsa.

Per questi motivi, raramente le politiche di protezione vengono rappresentate concretamente da una matrice, perché questo approccio richiederebbe l'allocazione di troppo spazio e costi di accesso elevati, a causa della complessità di ricerca delle informazioni all'interno della matrice.

I due approcci più comuni, invece, consistono nell'introduzione di *liste di controllo degli accessi* (*Access Control List*, o *ACL*) o delle *liste di capability* (*capability list*, o *C-list*).

In particolare, una *ACL* esprime la politica di protezione associata a una particolare risorsa. Essa quindi rappresenta una colonna della matrice di protezione: per ogni risorsa R vengono elencati i permessi concessi ad ogni utente del sistema per l'accesso a R.

Ad esempio, se consideriamo ancora la matrice di protezione di tabella 6.1, la lista di controllo degli accessi di File4 è [Paola: r-w-x, Elena: r].

Seguendo l'approccio basato su *ACL*, il sistema può quindi autorizzare (o meno) ogni tentativo di accesso a un file *f* da parte di un utente *U*, ricercando l'*ACL* di *f* e successivamente verificando i permessi associati a *U* all'interno dell'*ACL*.

Come verrà illustrato nel seguito (capitoli 7 e 8) sia Unix che Windows adottano tecniche di protezione basate sul concetto di *ACL*. In particolare, le *ACL* di risorse Unix sono più semplici rispetto al caso generale appena trattato, in quanto per ogni file vengono specificati i diritti di accesso relativi a soltanto tre classi di utenti: il *proprietario*, il suo *gruppo* di appartenenza e tutti gli *altri* utenti, ottenendo l'effetto di ridurre drasticamente l'occupazione di memoria di ogni *ACL* (9 bit).

Un approccio alternativo e complementare all'*ACL* è rappresentato dalla *capability list*: per ogni processo *P* il sistema costruisce una lista di permessi riferiti alle diverse risorse del sistema alle quali l'utente a cui *P* appartiene è abilitato ad accedere. In pratica, una *C-list* rappresenta una riga della matrice di protezione.

Per esempio, considerando ancora la matrice di protezione di tabella 6.1, ad ogni processo dell'utente Mario viene attribuita la seguente *C-list*:

```
[File1:r-w, File2:r, Dir01:r-w, Stampante1:w].
```

Da un lato questa tecnica offre una maggiore efficienza rispetto all'*ACL* nel controllo degli accessi, in quanto la *C-list* fa parte delle informazioni locali ad ogni processo e pertanto la ricerca dei diritti viene effettuata localmente al processo. Dall'altro lato, però, un'operazione come la revoca di tutti i diritti di accesso associati ad un file è un'operazione onerosa, in quanto essa richiede la ricerca e l'aggiornamento di tutte le *C-list*.

Per questo motivo alcuni sistemi operativi, come ad esempio MULTICS [69], adottano soluzioni ibride che sfruttano sia *C-list*, per garantire efficienza nell'accesso alle risorse, che *ACL* per ridurre l'overhead dovuto alla specifica di politiche riferite a particolari risorse.

6.4 Organizzazione fisica

Il livello di organizzazione fisica gestisce lo spazio disponibile sul dispositivo virtuale per l'allocazione dei file e delle strutture dati ad essi associate (come, ad esempio, la tabella dei descrittori). A questo scopo, lo spazio disponibile per l'allocazione sul disco è rappresentato da un insieme di *blocchi fisici*; esso può essere partizionato in due (o più) aree destinate rispettivamente all'allocazione dei file ed all'allocazione delle strutture dati associate al file system.

Nel prossimo paragrafo verranno esaminate le tecniche di allocazione dei file più comunemente utilizzate.

6.4.1 Tecniche di allocazione dei file

Le tecniche di allocazione dei file stabiliscono una corrispondenza tra i record logici contenuti in ogni file e l'insieme dei blocchi nei quali sono effettivamente memorizzati.

Infatti, come è stato già illustrato in precedenza, i processi accedono ai file facendo riferimento al *record logico*, come unità di accesso al file. L'unità di allocazione delle informazioni sul disco è invece il *blocco* (o *record fisico*), la cui dimensione è costante e dipende dal sistema operativo (ad esempio, Unix impiega blocchi di dimensioni comprese tra 512K e 4096K, a seconda della versione). In generale, la dimensione del blocco è comunque tale da poter contenere più record logici.

Lo spazio disponibile per l'allocazione dei file è visto come un vettore lineare di blocchi, ognuno dei quali, quindi, può essere utilizzato per l'allocazione di un sottoinsieme di record logici appartenenti ad uno stesso file.

Nel seguito, indicheremo con D_b la dimensione di un blocco e D_r la dimensione del record logico: il numero di record logici contenuti in ogni blocco (N_b) è quindi dato da $N_b = D_b/D_r$.

Esistono vari approcci riguardo a come scegliere i blocchi su cui allocare un file; nella maggior parte dei casi, ogni sistema operativo adotta un particolare criterio, che si concretizza nel metodo di allocazione: questa scelta è frutto di considerazioni relative sia alla rapidità nell'accesso ai file che all'efficienza nell'utilizzo dello spazio disponibile in memoria secondaria.

Nel seguito verranno analizzate e confrontate le tecniche di allocazione più diffuse nei sistemi moderni: allocazione *contigua*, a *lista* e ad *indice*.

Allocazione contigua Il metodo di allocazione contigua prevede che ogni file occupi un insieme di blocchi fisicamente contigui (vedi figura 6.7).

Questa collocazione sequenziale dei record logici sul dispositivo rende particolarmente semplice ed efficiente la realizzazione del metodo di accesso sequenziale.

Seguendo questo approccio, inoltre, è possibile realizzare in modo efficiente anche il metodo di accesso diretto. In questo caso, infatti, il descrittore di ogni file mantiene l'indirizzo B del primo blocco utilizzato per l'allocazione; l'indirizzo del blocco su cui è allocato un particolare record logico R_i viene quindi ottenuto direttamente mediante la formula $B + i/N_b$.

Il metodo di allocazione contigua presenta tuttavia alcuni svantaggi, legati soprattutto alla gestione dello spazio disponibile.

Un primo problema consiste nel costo indotto dalla ricerca dell'insieme di blocchi nei quali allocare un nuovo file: per ogni nuova allocazione, infatti, è necessario individuare sul disco un insieme di blocchi liberi contigui di estensione sufficiente a contenere il file.

Poiché raramente si troverà una zona della esatta dimensione del file f da allocare, l'area prescelta avrà in genere dimensioni superiori a quelle di f . A questo proposito, esistono vari criteri per la scelta: tra questi ricordiamo le tecniche *best fit* e *first fit* già esaminate nel quarto capitolo a proposito dell'allocazione in memoria centrale di partizioni di locazioni contigue (vedi paragrafo 4.3.1).

Nel primo caso il sistema tende a minimizzare il numero di blocchi non utilizzati nell'area prescelta per l'allocazione del nuovo file, individuando tra tutte le zone contigue di dimensione maggiore o uguale a quella di f , quella minima. Nel secondo caso viene scelta quella di indirizzo (su disco) inferiore. Esiste anche una terza possibilità, nota col nome di *worst fit*, in base alla quale viene cercata la zona di estensione massima, favorendo la possibilità di ulteriori allocazioni nella stessa zona.

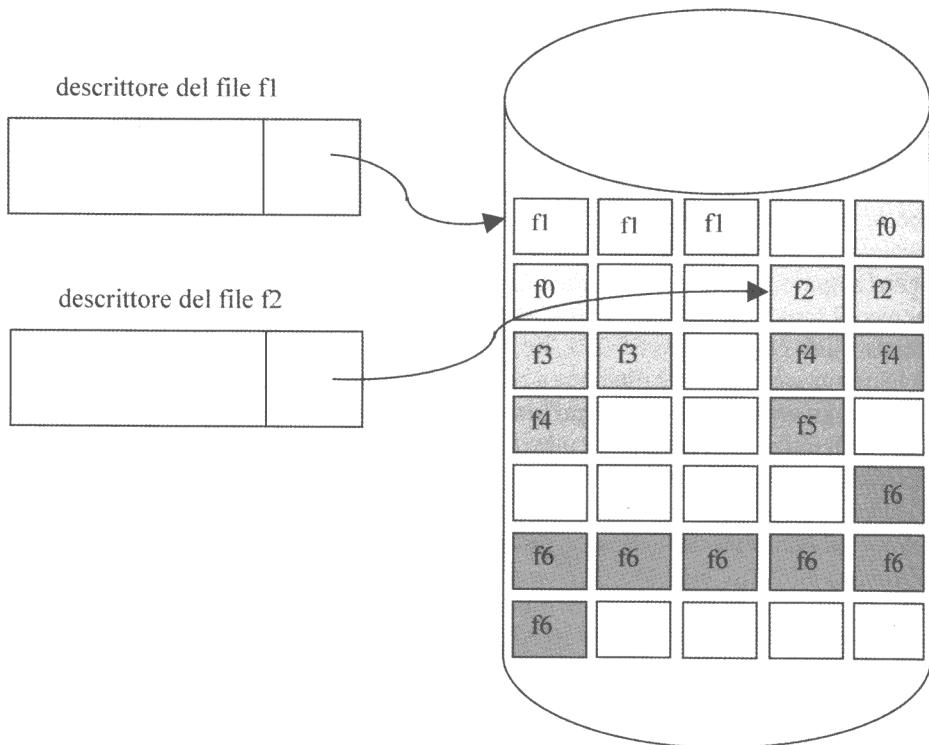


Figura 6.7 Allocazione contigua di file.

Un altro importante svantaggio dell’allocazione contigua consiste nella *frammentazione* dello spazio disponibile (vedi figura 6.8-a): man mano che si allocano file sul disco, rimangono zone contigue di blocchi liberi di dimensioni sempre più piccole, spesso inutilizzabili per l’allocazione di nuovi file. Per fare fronte a questo problema, i sistemi che adottano l’allocazione contigua prevedono anche l’intervento periodico di un meccanismo di *compattamento* (o *deframmentazione*): questa tecnica sposta i file già allocati in aree contigue tra loro, ricavando quindi dai numerosi frammenti di blocchi liberi, un’unica zona di blocchi liberi contigui di grandi dimensioni. La figura 6.8-b mostra l’effetto del compattamento sul file system (a). Lo stesso fenomeno della frammentazione e del relativo compattamento è stato già esaminato nel paragrafo 4.3.1 a proposito della tecnica di allocazione della memoria centrale mediante partizioni variabili. Il fenomeno è infatti imputabile allo stesso problema, e cioè a quello di allocare le informazioni in maniera contigua (in locazioni contigue nella memoria centrale o in blocchi contigui nella memoria di massa).

Allocazione a lista concatenata La tecnica di allocazione a lista memorizza ogni file in un insieme di blocchi non contigui organizzati in una lista concatenata (vedi figura 6.9).

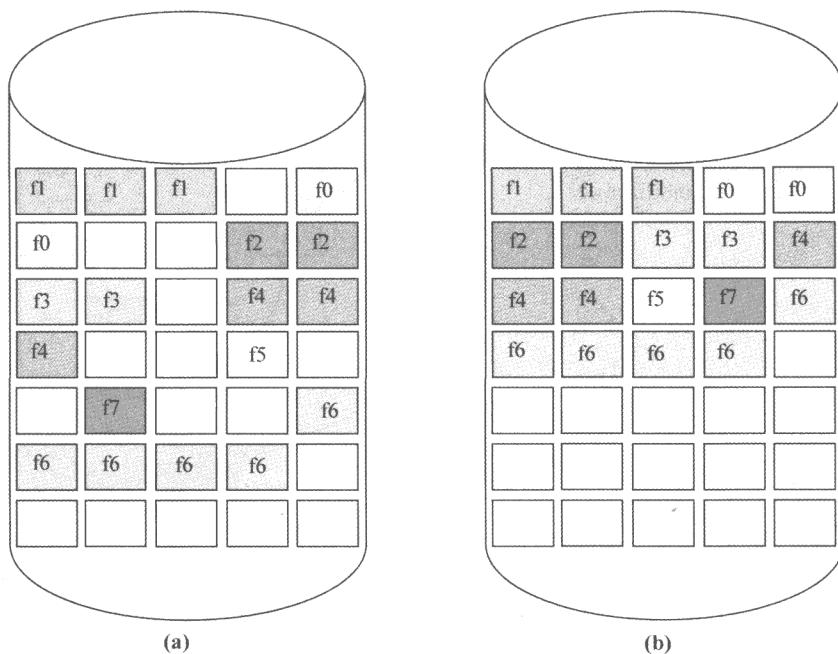


Figura 6.8 Frammentazione dello spazio disponibile su disco (a) e compattamento (b).

L'organizzazione dei blocchi utilizzati per uno stesso file è quindi ancora sequenziale, ma blocchi successivi non devono essere necessariamente vicini.

Per permettere l'accesso a file allocati con questo metodo è necessario mantenere nel descrittore del file il puntatore al primo blocco utilizzato; il collegamento tra blocchi successivi della stessa lista viene realizzato riservando all'interno di ogni blocco utilizzato lo spazio per il puntatore al blocco successivo nella lista (vedi figura 6.9).

Tutto ciò implica che in questo caso, il metodo di accesso sequenziale risulta poco costoso da realizzare, mentre l'accesso diretto risulta invece oneroso, in quanto il reperimento del blocco nel quale è allocato un particolare record R_i richiede preliminariamente l'effettuazione di i/N_b accessi in memoria prima di localizzare il blocco desiderato.

L'aspetto più vantaggioso dell'allocazione a lista riguarda l'eliminazione della frammentazione. Infatti, non dovendo rispettare il vincolo di contiguità tra blocchi, ogni blocco libero è effettivamente utilizzabile per allocare nuovi file. Inoltre, per lo stesso motivo il costo di ricerca dei blocchi liberi da utilizzare è molto ridotto rispetto al caso dell'allocazione contigua.

Un problema caratteristico di questa tecnica riguarda la scarsa robustezza del file system in caso di perdita di informazioni: se, infatti, per un guasto o un errore software si altera il valore di un puntatore ad un blocco b allocato a un file f , questo comporta inevitabilmente l'impossibilità di accedere a tutti i blocchi di f successivi a b .

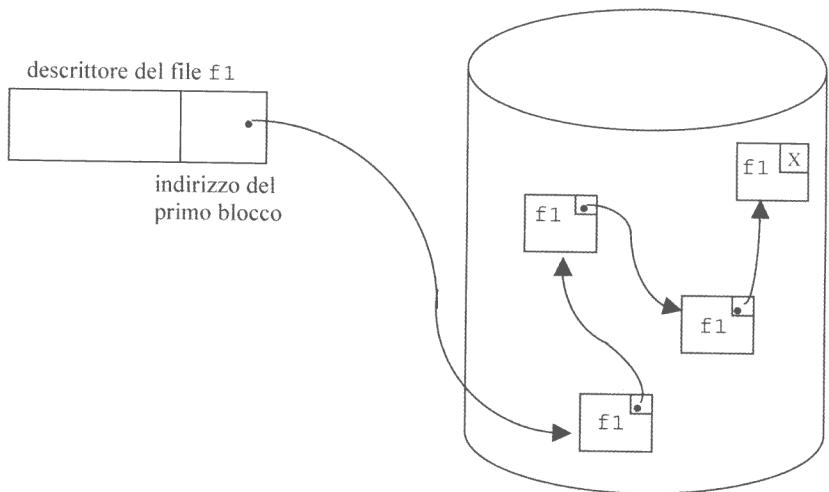


Figura 6.9 Allocazione a lista concatenata.

Per arginare questo problema, ad esempio, si può realizzare la lista con un doppio collegamento (vedi figura 6.10): ogni blocco contiene i puntatori al blocco successivo (puntatore *s*) e a quello precedente (puntatore *p*); in questo caso, quindi, è necessario salvare nel descrittore gli indirizzi del primo e dell'ultimo blocco della lista. Questo tipo di soluzione induce comunque una maggiore occupazione di memoria e un maggior costo di allocazione.

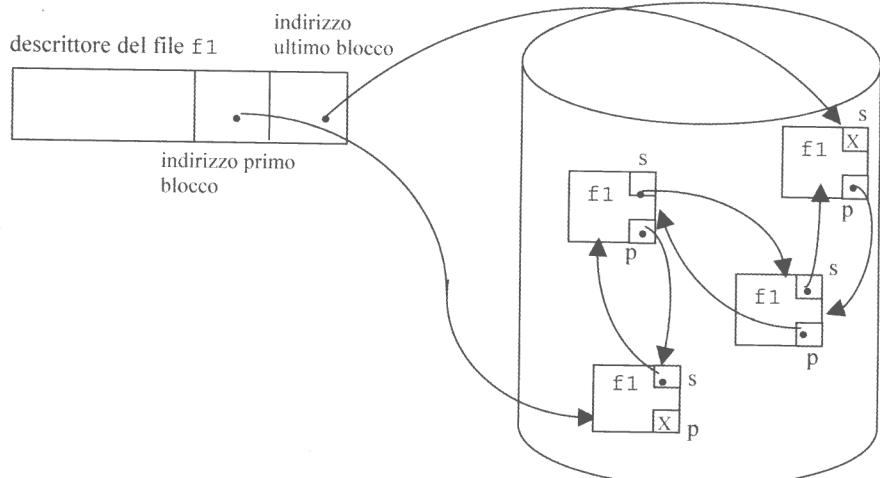


Figura 6.10 Allocazione a lista con doppio collegamento.

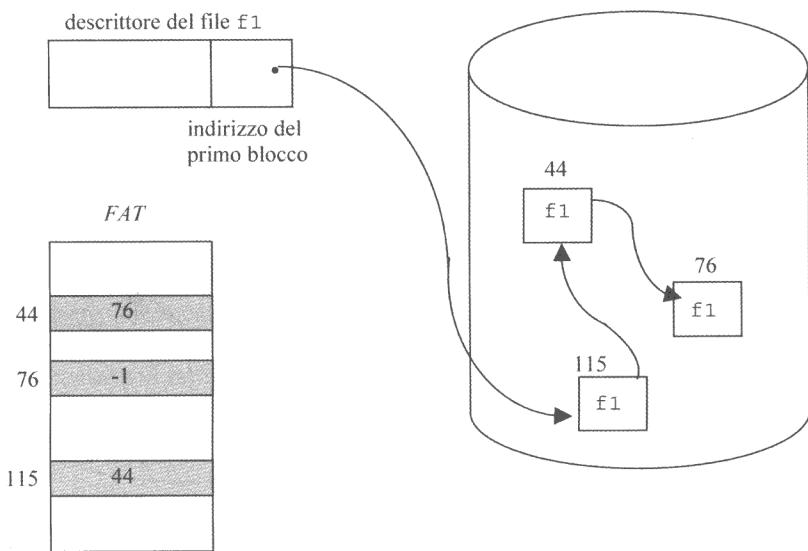


Figura 6.11 Allocazione a lista con FAT.

Alcuni sistemi operativi (per esempio MS/DOS), invece, affrontano il problema affiancando ad una allocazione basata ancora su una lista concatenata a collegamento semplice, l'introduzione di una struttura dati nella quale viene descritta la mappa di allocazione di tutti i blocchi. Tale struttura, chiamata *tabella di allocazione dei file* (*File Allocation Table, FAT*), viene memorizzata in una posizione predefinita sul dispositivo virtuale. Essa (vedi figura 6.11) contiene un elemento per ogni blocco del dispositivo il cui valore indica se il blocco è libero, o, in caso contrario, è l'indice dell'elemento della tabella che rappresenta il blocco successivo nella lista.

Con questa soluzione, in caso di perdita di un concatenamento, è possibile effettuare il recupero del puntatore perso accedendo alla *FAT*. Inoltre la *FAT* può essere copiata dal sistema in memoria centrale o in una cache: in questo modo è possibile velocizzare notevolmente l'accesso diretto, poiché l'indirizzo del blocco da accedere può essere ottenuto senza compiere accessi preliminari al disco.

Allocazione a indice L'allocazione ad indice si basa ancora sull'utilizzo di blocchi non contigui per l'allocazione di un file. In particolare, questo metodo prevede che ad ogni file venga associato un blocco *indice* che contiene gli indirizzi dei blocchi effettivamente utilizzati per l'allocazione del contenuto del file (vedi figura 6.12).

Con questa tecnica, analogamente all'allocazione a lista, si esclude la possibilità di frammentazione. In questo caso il descrittore del file contiene l'indirizzo del blocco indice, accedendo al quale è possibile ottenere l'indirizzo di ogni altro blocco utilizzato per l'allocazione del file. Perciò questo metodo consente di accedere efficientemente al file sia in modo sequenziale che diretto.

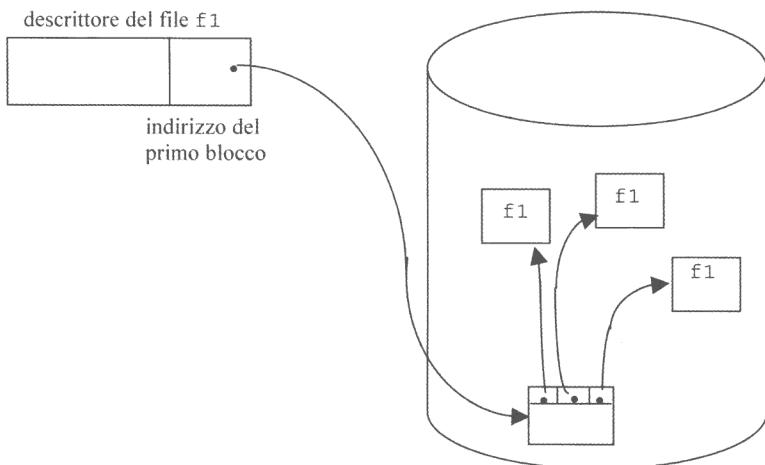


Figura 6.12 Allocazione a indice.

Il principale svantaggio di questo approccio riguarda la dimensione del blocco indice: essendo infatti limitata, questo implica che anche la dimensione del file da esso indirizzato sia limitata. Per ovviare a questa restrizione, alcuni sistemi operativi introducono più livelli di indice: per esempio, nel caso di indice a due livelli, ad ogni file è associato un indice di primo livello che contiene indirizzi di altri blocchi indice (di secondo livello), i quali contengono indirizzi di blocchi utilizzati per l'allocazione del file. Unix, ad esempio, utilizza un metodo basato su tre livelli di indice; il capitolo 7 mostrerà con maggior dettaglio le scelte di Unix a questo riguardo.

6.5 Sommario

In questo capitolo sono stati trattati i principali aspetti relativi alla gestione dei file, realizzata da una delle componenti principali del sistema operativo: il *file system*. Il compito di questa componente all'interno del sistema è di fornire adeguate astrazioni e meccanismi per l'archiviazione delle informazioni nei dispositivi di memoria di massa. Concetto centrale di questo capitolo è il file, che rappresenta l'unità di archiviazione delle informazioni nel *file system*.

È stata presentata l'organizzazione del *file system* mediante la definizione di quattro livelli, ognuno dedicato alla soluzione di particolari problemi; come è stato mostrato, l'adozione di un'organizzazione gerarchica di questo tipo ha permesso di trattare importanti temi, come ad esempio i metodi di accesso ai file, in modo astratto rispetto ad altri aspetti come, ad esempio, le tecniche di allocazione dei file sul dispositivo.

Quindi, sono state descritte le funzionalità di ciascun livello e le soluzioni comunemente adottate per la loro realizzazione. In particolare, partendo dal livello più alto, sono stati trattati gli aspetti relativi alla presentazione del *file system* alle

applicazioni (la struttura logica), introducendo il concetto di directory come contenitore di file.

Le caratteristiche del directory influiscono sulla struttura logica del *file system*: le soluzioni più comuni prevedono strutture logiche gerarchiche come, ad esempio, alberi *n*-ari.

Successivamente sono state prese in considerazione le problematiche relative all'accesso a file, includendo una breve trattazione sulla protezione del *file system*, requisito fondamentale nei sistemi multi-utente.

Infine è stato affrontato il tema dell'allocazione dei file, analizzando i pro ed i contro degli approcci più diffusi: in particolare si sono evidenziate le differenze tra tecniche basate su allocazione contigua e tecniche di allocazione non contigua, come allocazione a lista e a indice, sottolineando che, nonostante l'allocazione contigua permetta accessi più rapidi ai file, essa presenta come principale svantaggio la frammentazione del dispositivo. Questo problema viene superato invece dalle tecniche basate su allocazione non contigua.

6.6 Note bibliografiche

Le problematiche relative alla gestione dei files, trattate in modo più esteso nei testi [80], [96] e [85], possono essere ulteriormente approfondite consultando [35], dove tutti gli aspetti inerenti alla realizzazione del file system vengono esaminati in modo più dettagliato.

7

I sistemi operativi Unix e Linux

In questo capitolo viene fornita una breve panoramica di due fra i più diffusi sistemi operativi: Unix e Linux. Dopo una breve introduzione relativa alla loro evoluzione storica, l'attenzione verrà focalizzata principalmente su Unix e, successivamente, su Linux con particolare riferimento alla gestione dei threads.

7.1 Introduzione a Unix

Scopo di questa prima parte del capitolo è quello di esaminare le caratteristiche del sistema operativo Unix e di verificare come i vari concetti visti nei precedenti capitoli del testo stiano stati calati in pratica nell'ambito di questo sistema.

7.1.1 La storia di Unix

Le origini del sistema operativo UNIX risalgono al 1969 quando venne sviluppato nei *Bell Laboratories* dell'AT&T. L'obiettivo era quello di realizzare un ambiente di calcolo multiprogrammato portabile per macchine di medie dimensioni.

Nel 1970 viene realizzata la prima versione di UNIX, interamente sviluppata nel linguaggio assembler del calcolatore PDP-7; questa versione era multiprogrammata e monoutente. Negli anni successivi al 1970 furono prodotte nuove versioni, arricchite con altre caratteristiche e funzionalità. In particolare venne introdotto il supporto alla multiutenza. Nel 1973 Unix venne scritto utilizzando, per la maggior parte del progetto, il linguaggio di programmazione C. Il risultato fu un sistema operativo nel quale più del 60% del codice era scritto in un linguaggio di alto livello e quindi in buona parte indipendente dalla piattaforma hardware. Questa caratteristica conferì al nuovo sistema operativo un'ottima portabilità, intesa come capacità di fornire lo stesso ambiente di esecuzione e di sviluppo su architetture diverse. Questa prerogativa, decisamente inusuale all'epoca in cui Unix è nato, è stata la chiave di volta per la capillare diffusione che ha raggiunto nei giorni nostri. Di pari passo anche il lin-

guaggio C, nato come formalismo per lo sviluppo di Unix si è diffuso in maniera tale da diventare uno standard di fatto tra i linguaggi di programmazione imperativi. L'utilizzo di un linguaggio così conosciuto ha anche contribuito a rendere popolare il sistema operativo presso la comunità scientifica e accademica che ha potuto comprendere e apprezzare facilmente alcune componenti del sistema [96]. Per questo motivo il sistema Unix, è stato utilizzato dagli anni 80 fino ad oggi per la formazione di intere generazioni di persone operanti nel settore dell'informazione, diffondendo una cultura che ha contribuito in modo fondamentale all'attuale successo della famiglia Unix.

Negli anni '80 la grande popolarità di Unix ha anche inevitabilmente determinato il proliferare di svariate realizzazioni, spesso con notevoli differenze tra loro, che hanno ostacolato una piena portabilità delle applicazioni. In particolare, le due *famiglie* di sistemi Unix maggiormente affermatisi sono Unix System V [11], prodotto dai laboratori dell'AT&T, e Unix Berkeley Software Distribution [58] BSD, sviluppato presso l'University of California di Berkeley.

Nel 1988 L'*Institute of Electrical and Electronic Engineers* (IEEE) definisce lo standard POSIX (*Portable Operating Systems Interface*) [46] che stabilisce le caratteristiche relative alle modalità di utilizzo e di programmazione del sistema operativo. In particolare, lo standard POSIX definisce un'interfaccia tra il sistema operativo e i programmi (API). Applicazioni che utilizzano i servizi del sistema operativo utilizzando l'interfaccia POSIX, hanno ottime caratteristiche di portabilità verso altri sistemi operativi conformi allo standard.

Nel 1990 lo standard POSIX viene riconosciuto dall'*International Standards Organization* (ISO). Negli anni seguenti, le versioni successive di Unix SystemV e BSD (versione 4.3), pur mantenendo comunque alcune differenze tra loro, sono state sviluppate entrambe in maniera conforme allo standard POSIX.

7.1.2 Linux

Una delle realizzazioni di Unix attualmente più diffuse è Linux, un sistema operativo di pubblico dominio realizzato inizialmente per essere eseguito su personal computer Intel x86 ed oggi disponibile su varie architetture (ad esempio, SUN Sparc, Motorola 68k, PowerPC, Alpha, ecc.).

Linux è nato nel 1991, per iniziativa di Linus Torvalds, uno studente dell'Università di Helsinki che, per coltivare un interesse personale sviluppò il kernel di Minix (una versione semplificata di Unix, definita in ambito accademico da A. Tanenbaum) [95] per Personal Computer. I sorgenti della prima versione di Linux furono resi accessibili gratuitamente attraverso la rete Internet [60]. Questa possibilità invogliò molti programmati ad analizzare, modificare ed estendere le varie componenti del sistema operativo. Questo fenomeno provocò un velocissimo progresso nel completamento e nel miglioramento del sistema, che in breve tempo, da un prototipo "giocattolo" di sistema operativo divenne un sistema operativo completo, moderno, robusto ed efficiente. Tutto questo grazie al contributo e alla collaborazione di un vasto ed aperto gruppo di persone distribuito su una rete geografica. In questo modo nel 1994 venne prodotta la prima versione completa ed affidabile del sistema operativo. Negli anni seguenti Linux ha avuto una grande diffusione grazie alle sue caratteristiche di affidabilità ed efficienza, che lo mettono in diretta competizione con i

corrispondenti rivali commerciali. La distribuzione di Linux è gratuita, presso i numerosi siti Internet dai quali è possibile scaricare tutto il software necessario per l'installazione del sistema operativo. Nonostante questo, esistono varie distribuzioni "commerciali" di Linux [77] [25] che offrono tutto il software di Linux su CD-ROM spesso corredata da moduli aggiuntivi (come, ad esempio, particolari interfacce grafiche) e da ampia documentazione.

Il sistema operativo Linux, pur essendo nato come una realizzazione di Unix, nel tempo è stato modificato ed esteso ed ha acquisito alcune caratteristiche che oggi lo differenziano rispetto al sistema da cui ha tratto ispirazione. A questo proposito, una delle differenze più macroscopiche riguarda la gestione dei processi, che nel sistema Linux è attualmente basata sul concetto di *thread*, mentre in Unix è improntata a un modello di processo *pesante*. Questo aspetto verrà approfondito nel paragrafo 7.8, che tratterà proprio delle caratteristiche e della gestione dei *thread* nel sistema *Linux*.

7.2 Architettura del sistema operativo UNIX

Unix è un sistema operativo *multiutente* e *multiprogrammato*: il suo nucleo supporta la contemporanea esecuzione di più processi che vengono gestiti dallo scheduler a divisione di tempo (o *time sharing*, vedi capitolo 2).

Il sistema di gestione della memoria in Unix si basa su paginazione e segmentazione. Utilizzando questi meccanismi il sistema fornisce la *memoria virtuale* (vedi capitolo 4), grazie alla quale ogni processo può indirizzare un'area di memoria di dimensioni eventualmente superiori a quelle della memoria fisica effettivamente disponibile.

Grazie all'impiego del linguaggio C nella realizzazione del sistema e alla conformità con lo standard POSIX, il sistema fornisce un ambiente di esecuzione standard e portabile, ed è oggi disponibile su una vastissima gamma di architetture diverse.

Unix mantiene tuttora uno stretto legame con il linguaggio C: esso offre infatti un insieme piuttosto ricco di applicazioni di sistema che realizzano strumenti per lo sviluppo di applicazioni C (ad esempio, il compilatore cc).

Inoltre, Unix utilizza alcuni dei più diffusi servizi e protocolli di comunicazione della rete *Internet*, rendendo possibile una facile integrazione di sistemi Unix all'interno di reti di calcolatori.

7.2.1 Organizzazione

L'architettura di un sistema Unix è a livelli (figura 7.1), ad ognuno dei quali corrisponde una diversa funzionalità. In particolare, ogni livello presenta una specifica interfaccia (e quindi una diversa vista del sistema) al livello sovrastante.

Il livello più basso è costituito dall'hardware, al di sopra del quale è collocato il kernel del *Sistema operativo*¹. Questo livello realizza tutte le funzionalità del sistema

¹ Nella terminologia Unix il termine *kernel* assume un significato diverso da quello introdotto nei capitoli 1 e 2. Con tale termine, infatti, si intende comprendere tutte le funzioni svolte tradizionalmente da un sistema operativo.

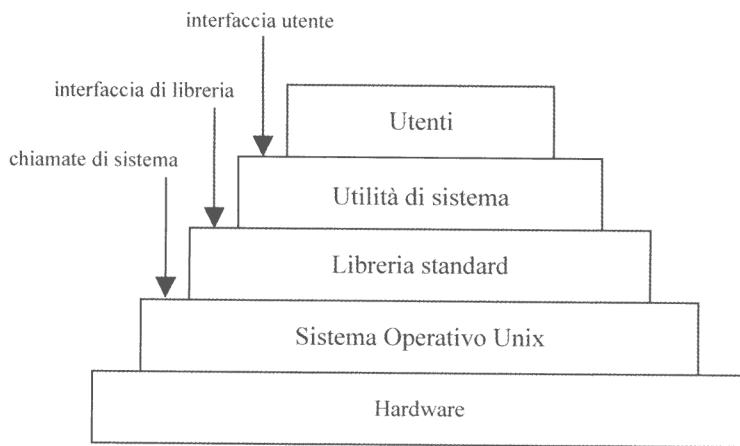


Figura 7.1 Architettura del sistema Unix.

operativo, tra cui la gestione dei processi, della memoria ed il file system. Nella maggior parte delle realizzazioni, Unix è *monolitico* (fanno eccezione alcuni sistemi derivati da Unix, come ad esempio Mach 3.0, realizzato presso l'Università americana Carnegie-Mellon, che adotta una organizzazione a *microkernel*, vedi capitolo 1) [76].

L'interfaccia del sistema operativo verso il livello superiore è costituito dalle *chiamate di sistema* (*system calls*). Queste funzioni mettono a disposizione del programmatore di sistema dei servizi che, una volta invocati, vengono eseguiti dal *kernel*. In particolare, il codice delle chiamate di sistema è parte del *kernel* stesso: ogni processo che ne richiede l'esecuzione, trasferisce il controllo al *kernel*, che effettua il servizio. I servizi tipicamente effettuati dalle chiamate di sistema riguardano azioni su processi (ad esempio, creazione, sincronizzazione e comunicazione) e su file (ad esempio, lettura e scrittura di informazioni da/verso file).

Sopra il livello delle chiamate di sistema è collocata una collezione di librerie C, il cui ruolo fondamentale è quello di fornire un'interfaccia C per le chiamate di sistema: ogni applicazione C può quindi richiedere l'esecuzione di una chiamata di sistema, attraverso una chiamata alla specifica funzione C di libreria che la rappresenta.

Il livello più alto del sistema è rappresentato dalle *Utilità di sistema*. Si tratta, in generale, di programmi applicativi il cui ruolo è quello di offrire all'utente alcuni servizi (tra cui, ad esempio, le interfacce utente e gli strumenti di sviluppo per programmi C).

7.3 Interazione con l'utente

Il sistema Unix prevede che l'interazione con l'utente avvenga attraverso un interprete dei comandi (lo *shell*) che interpreta ed esegue le richieste formulate dall'utente attraverso opportune direttive (i *comandi*). In particolare, allo shell è associato un vero e proprio linguaggio che definisce il vocabolario e la sintassi dei comandi che l'utente deve necessariamente conoscere per potere interagire con il sistema.

Tuttavia, nelle moderne versioni di Unix l'interazione dell'utente con il sistema è semplificata, grazie all'introduzione di interfacce grafiche che permettono di specificare le proprie richieste in modo più semplice ed intuitivo mediante finestre, icone, menu da utilizzare con un dispositivo di puntamento (ad esempio, il mouse). Una delle più diffuse interfacce grafiche per Unix è CDE (*Common Desktop Environment*), frutto di uno sforzo di standardizzazione portato avanti dalle più importanti aziende produttrici di Unix, che presenta caratteristiche analoghe a quelle dell'interfaccia grafica del sistema operativo Windows NT.

7.3.1 Lo shell

Nelle diverse versioni di Unix è possibile ritrovare vari tipi di shell che generalmente differiscono tra di loro più o meno marcatamente nel linguaggio di comandi. Nel seguito faremo riferimento allo shell di *Bourne*, introdotto nelle prime versioni di Unix, e tuttora presente nella maggior parte delle realizzazioni.

Lo shell può acquisire i comandi dalla linea di comando, oppure da un file (detto *file comandi*). Nel primo caso si parla di modalità *interattiva* di esecuzione: l'utente può impartire un comando alla volta allo shell, digitandolo direttamente dalla tastiera al *prompt* della linea di comando. Ogni shell prevede un prompt caratteristico: per lo shell di Bourne il classico prompt è il carattere “\$”. Nel secondo caso l'interazione avviene tramite file comandi: l'utente indica dalla linea di comando il nome del file comandi. Lo shell eseguirà, nello stesso ordine con cui appaiono nel file, i comandi specificati.

Ogni comando può essere visto come un *filtro*, con un canale d'ingresso, mediante il quale il comando acquisisce eventuali dati, ed un canale d'uscita, attraverso il quale vengono comunicati i risultati prodotti dalla sua esecuzione (figura 7.2):



Figura 7.2 Comandi di shell.

Il canale di ingresso (*input stream*), è normalmente associato al dispositivo di standard input (tipicamente, la tastiera del terminale). Il canale di uscita (*output stream*) è invece associato al dispositivo di standard output (di solito, questo corrisponde al video del terminale).

I canali di input e di output non vengono impiegati da tutti i comandi: in particolare, i comandi che sfruttano entrambi i canali di comunicazione vengono detti *comandi filtro*. L'associazione tra canali e dispositivi può essere modificata dall'utente, utilizzando opportuni operatori di *ridirezione* e *piping*.

Sintassi di comandi La sintassi tipo di un comando è la seguente:

\$ nomecomando [opzioni] [lista degli argomenti]

dove le opzioni indicano modalità specifiche di esecuzione del comando e la lista degli argomenti contiene gli eventuali parametri del comando.

Ad esempio, la linea di comando:

```
$ ls -l /home/andrea
```

richiede l'esecuzione del comando `ls` con l'opzione `-l`, riferito all'argomento `/home/andrea`; l'effetto di questo comando è la visualizzazione dei nomi e degli attributi (opzione `-l`) di tutti i file e i direttori contenuti nel directory specificato dell'argomento `/home/andrea`.

Una descrizione dettagliata del significato e delle modalità di utilizzo dei vari comandi andrebbe oltre gli obiettivi di questo testo; per una trattazione dettagliata dei comandi Unix il lettore interessato potrà consultare, ad esempio, [11].

7.4 I processi nel sistema operativo Unix

Unix è una famiglia di sistemi operativi multiprogrammati basati su *processi*: supporta quindi il concetto di processo (nell'accezione presentata nel paragrafo 2.1) e non prevede, invece, la possibilità di *multithreading*. Il processo Unix, quindi, contiene un solo thread.

Tuttavia, va sottolineato che in molte delle versioni attualmente commercializzate di Unix vengono fornite librerie addizionali (si vedano ad esempio le librerie specifiche di Solaris [92]) che forniscono funzioni per la creazione e la gestione di *threads* all'interno di processi. Poiché la realizzazione dei thread viene fatta a livello utente, in questo modo viene consentito il loro impiego anche in ambiente Unix.

Inoltre, esistono dei sistemi derivati da Unix (come, ad esempio, il sistema *Linux*) che realizzano i *thread* a livello kernel, adottando quindi un differente modello di processo (rispetto a Unix), basato sul *multithreading*.

7.4.1 Caratteristiche

Il processo Unix mantiene spazi di indirizzamento separati per dati e codice.

Ogni processo ha uno spazio di indirizzamento dei dati privato: non esiste, quindi, la possibilità di condivisione di variabili tra processi diversi. Questa caratteristica, come vedremo in seguito, impedisce ai processi di interagire mediante variabili condivise, ed impone che l'interazione avvenga mediante scambio di messaggi.

A differenza dello spazio di indirizzamento dei dati, lo spazio di indirizzamento del codice (*text*) è invece condivisibile: questa caratteristica del processo Unix, detta *rientranza* del codice (vedi primo capitolo, paragrafo 1.4.1), permette a più processi di eseguire lo stesso programma, facendo riferimento alla stessa area di codice nella memoria centrale.

Diagramma degli stati Il sistema operativo Unix adotta una politica di assegnazione della CPU ai processi basata sulla divisione di tempo. Il diagramma degli stati di un processo (come mostrato in figura 7.3) ricalca quasi completamente lo schema generale inizialmente presentato in figura 2.4 e successivamente esteso con gli stati *swapped* come indicato in figura 4.21.

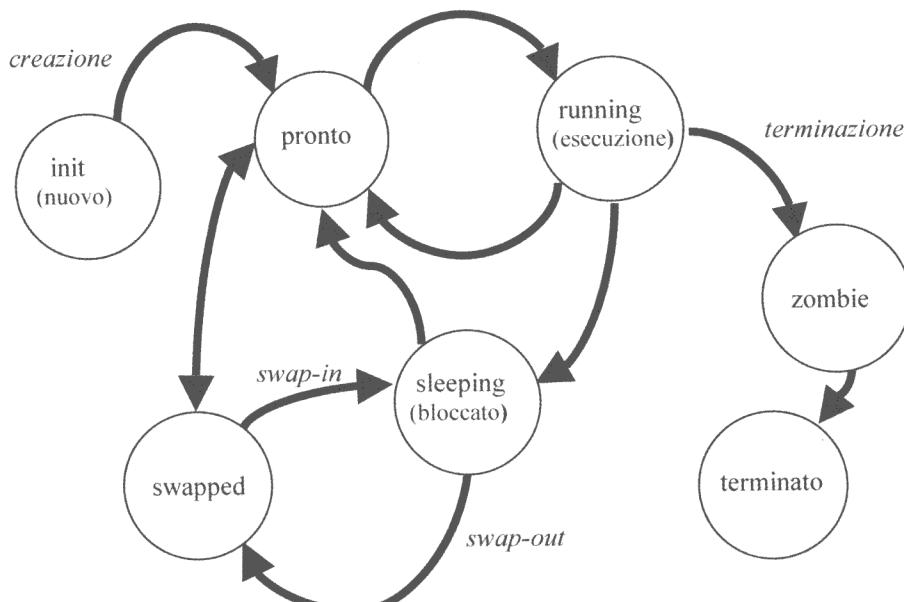


Figura 7.3 Diagramma degli stati di un processo Unix.

Come si può osservare nella figura 7.3, in aggiunta a quelli mostrati in figura 4.21, è presente un nuovo stato: *zombie*. È questo lo stato in cui si trova un processo che ha terminato di eseguire il codice ma che non può essere eliminato dal sistema perché la sua immagine è ancora necessaria. Il caso tipico in cui un processo entra nello stato *zombie* è il seguente: se un processo figlio termina prima del padre e, in particolare, prima che esso ne rilevi la terminazione, il sistema lo mantiene nello stato *zombie*; il processo figlio uscirà da tale stato non appena il padre avrà raccolto le informazioni relative alla sua terminazione, oppure (nel caso in cui il padre non effettui tali operazioni) quando il padre avrà terminato la propria esecuzione.

Riguardo allo stato *swapped*, va specificato che in Unix è presente uno scheduler di medio termine (*swapper*); è prevista quindi la possibilità, per un processo presente in memoria principale, di essere temporaneamente spostato in memoria secondaria in modo da liberare spazio per altri processi (vedi capitolo 4). Lo stato *swapped* è proprio lo stato in cui si trova un processo temporaneamente spostato dallo *swapper* in memoria secondaria. Questo trasferimento forzato in memoria secondaria (*swap-out*) si applica preferibilmente ai processi bloccati (*sleeping*), prendendo in considerazione parametri quali ad esempio il tempo di attesa, il tempo di permanenza in memoria e la dimensione del processo. La politica normalmente adottata nei sistemi Unix, ad esempio, applica lo *swap-out* preferibilmente ai processi la cui dimensione (intesa come estensione dell'area di memoria occupata) è maggiore. Un processo *swapped* verrà successivamente ripristinato in memoria mediante l'operazione di *swap-in*: in questo caso lo *swapper* selezionerà il processo da trasferire ancora in ba-

se a politiche proprie (ad esempio, la scelta può essere indirizzata verso il processo di minore dimensione).

7.4.2 Immagine di un processo Unix

Il sistema Unix suddivide le informazioni tipicamente contenute nel PCB di un processo (vedi capitolo 2) in due strutture dati distinte: la *Process Structure* e la *User Structure*.

La *Process Structure* contiene le informazioni indispensabili per la gestione del processo, anche se esso è nello stato *swapped*. La *User Structure*, invece, contiene le informazioni necessarie al sistema per la gestione del processo soltanto quando esso è residente in memoria (ovvero, non è nello stato *swapped*). Per questi motivi, come si vedrà nel seguito, la *User Structure* stessa potrà essere soggetta a *swap-out*, mentre la *Process Structure* deve rimanere residente in memoria per tutta la durata della vita del processo.

In particolare, le informazioni più importanti contenute nella *Process Structure* sono le seguenti (vedi figura 7.4): l'intero positivo che individua univocamente il processo nel sistema (*process identifier* o *PID*), lo stato del processo, i puntatori alle varie aree dati e stack associati al processo (nel caso di stato *swapped*, tali indirizzi saranno riferiti alla memoria secondaria), un riferimento (indiretto) al codice, le informazioni di scheduling (ad esempio, la priorità) e un riferimento al processo padre. Inoltre, vi sono il puntatore al processo successivo nella coda a cui appartiene il processo ed il riferimento alla User Structure del processo.

Tra le informazioni tipicamente contenute nella *User Structure* (vedi figura 7.4) vi sono: una copia dei registri di CPU (da utilizzare al cambio di contesto), le informazioni sulle risorse allocate al processo (ad esempio i file aperti), informazioni sulla gestione di eventi asincroni (o *segna*li, vedi paragrafo 7.7) e l'ambiente di esecuzione del processo (ad esempio, il direttorio corrente, etc.).

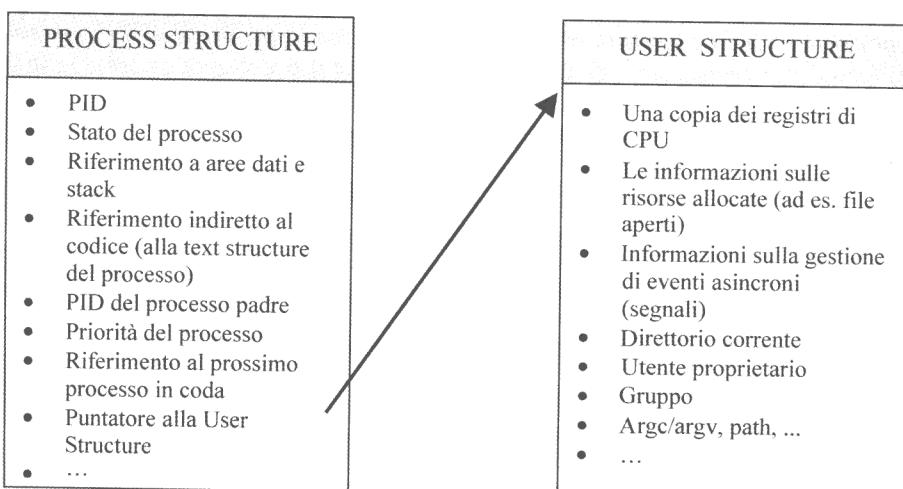


Figura 7.4 Descrittore di un processo Unix.

Le *Process Structure* di tutti i processi sono contenute all'interno di una struttura dati del kernel: la tavola dei processi (*process table*). Ad ogni istante, quindi, il numero degli elementi contenuti nella tavola dei processi coincide con il numero dei processi presenti nel sistema.

Il codice dei processi Unix è rientrante, cioè può essere condiviso da più processi. Per questo motivo il kernel gestisce una struttura dati globale, detta tavola dei codici (o *text table*) nella quale ogni elemento rappresenta il codice di un programma correntemente eseguito da uno o più processi. In particolare, ogni elemento della *text table* (detto *text structure*) contiene un puntatore all'area di memoria in cui è allocato il codice (se il processo è *swapped*, esso è un riferimento alla memoria secondaria). Nel descrittore del processo, quindi, il riferimento al codice è espresso mediante un riferimento all'elemento della *text table* che rappresenta il programma eseguito. In particolare, tale collegamento viene mantenuto nella *Process Structure* associata al processo.

Come si è visto nel paragrafo 2.10, l'immagine di un processo è costituita dal suo descrittore (PCB) e dall'insieme delle aree di memoria dove vengono allocati dati, codice e stack del processo. La struttura dell'immagine del processo, nel caso del sistema Unix, è quindi articolata in varie parti, come descritto dalla figura 7.5: in particolare, essa è costituita, oltre che dal descrittore (suddiviso in *Process* e *User Structure*), dalla Text Structure e dalle aree dati, codice e stack allocate per il processo. Il sistema prevede, oltre allo stack necessario al supporto delle normali chiamate a procedura/funzione in modalità utente, uno stack del kernel, a supporto delle chiamate al sistema operativo.

La figura 7.5 classifica le componenti dell'immagine in base alla visibilità (*user/kernel*) e alla possibilità di swapping (*residente/swappabile*).

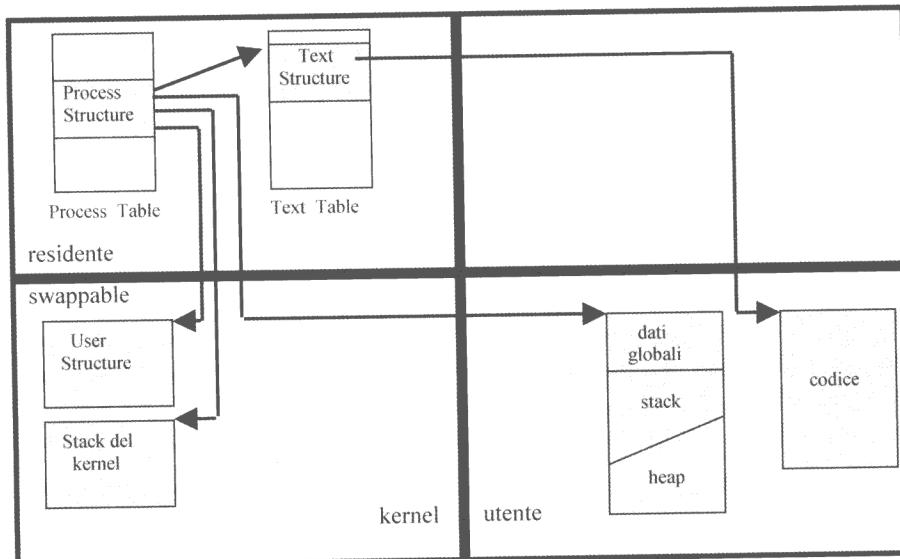


Figura 7.5 Immagine di un processo Unix.

Alcune parti dell'immagine, infatti, riguardano informazioni relative alla gestione del processo da parte del sistema operativo: queste parti, vengono pertanto protette permettendone l'accesso soltanto al kernel. Il processo può accedere (in modalità user) soltanto alle proprie aree di dati (dati globali, stack e heap) e di codice. Tutte le altre parti sono accessibili in modalità kernel: tra queste troviamo gli elementi delle strutture globali *Process Table* e *Text Table*, la *User Structure* del processo e il suo stack del kernel. Inoltre, non tutte le parti dell'immagine di un processo possono essere sottoposte a swapping. Infatti, alcune informazioni, come quelle contenute nelle *Process Structure* e *Text Structure* associate al processo, sono necessarie in ogni istante al sistema (indipendentemente dallo stato del processo) per una corretta gestione del processo e pertanto devono essere residenti in memoria.

7.4.3 System Call per la gestione di processi

Creazione di Processi Il meccanismo fondamentale a supporto del concetto di processo è il meccanismo di *creazione*: attraverso opportuni strumenti, ogni processo è in grado di creare dinamicamente altri processi. Lo strumento per la creazione dei processi è costituito dalla chiamata di sistema `fork`, mediante la quale un processo può creare un nuovo processo figlio. Il processo creato, pur mantenendo uno spazio di indirizzamento dei dati separato, per la caratteristica di rientranza condividerà con il processo padre il codice. Ogni processo creato può a sua volta generare altri processi, che possono generare altri, etc. Si vengono così a formare delle gerarchie di processi rappresentabili da strutture ad albero, come illustrato in figura 7.6.

A livello sintattico, la system call `fork` è caratterizzata dalla seguente dichiarazione:

```
int fork(void);
```

Come si può notare, quindi, la `fork` non richiede parametri e restituisce al padre un intero che, se la creazione è avvenuta con successo, ha valore positivo e rappresenta il PID del processo figlio; altrimenti, il valore restituito sarà negativo.

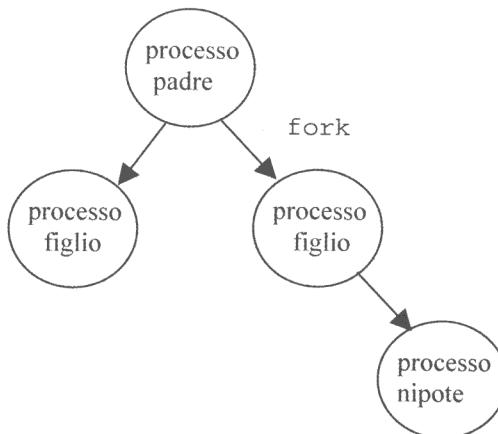


Figura 7.6 Gerarchia di processi.

Il nuovo processo (*il figlio*) condivide il codice con il *padre* ed eredita una copia delle aree dati globali, stack, heap, User Structure dal padre: nell'istante immediatamente successivo alla creazione, ogni variabile del figlio è inizializzata con il valore assegnato dal padre prima della `fork`.

Pertanto la User Structure del figlio è, inizialmente, una copia di quella del padre; ricordando che in questa struttura è conservato il valore del registro Program Counter, ciò significa che, non solo padre e figlio condividono il codice del programma eseguito, ma anche che la prima istruzione eseguita dal figlio sarà quella immediatamente successiva alla `fork` che l'ha creato.

Il figlio, quindi, può leggere l'intero restituito dalla `fork` che l'ha creato: per convenzione, il suo valore è sempre uguale a zero. In sostanza, padre e figlio ottengono valori diversi dalla chiamata a `fork`: questo, come si può vedere nell'esempio di figura 7.7, consente di differenziare il comportamento del padre da quello del figlio.

```
#include <stdio.h>
main()
{
    int pid;
    pid=fork();
    if (pid==0)
        /* codice figlio */
        printf("Sono il figlio! (pid: %d)\n", getpid());
        /* la system call getpid restituisce il pid */
        /* del processo che la chiama */
    else if (pid>0)
        /* codice padre */
        printf("Sono il padre: pid figlio: %d\n", pid);
    else printf("Creazione fallita!");
}
```

Figura 7.7 Creazione di un processo in Unix.

Nel programma mostrato in figura, infatti, viene definita una variabile `pid` mediante la quale viene ricevuto il valore restituito dalla `fork`. Se la creazione è avvenuta con successo (il valore di `pid` non è negativo) il padre otterrà un valore di `pid` strettamente positivo che rappresenta l'identificatore del figlio appena creato; il nuovo processo figlio, invece, avrà una diversa istanza della variabile `pid` con valore iniziale uguale a zero. È proprio il valore di `pid` che, nell'esempio, consente di esprimere diverse azioni per padre e figlio mediante l'istruzione `if`: in questo caso padre e figlio stampano messaggi diversi sullo standard output.

Terminazione dei processi Un processo può terminare *involontariamente* o *volontariamente*: nel primo caso la terminazione avviene in modo imprevisto per cause di forza maggiore, come nel caso di tentativi di azioni illegali (ad esempio, l'accesso a locazioni esterne al proprio spazio di indirizzamento) o di interruzioni causate dalla ricezione di segnali; invece, la terminazione di un processo è volontaria quando

esso termina l'esecuzione dell'ultima istruzione del suo codice, oppure se chiama esplicitamente un'apposita system call (`exit`).

La politica di Unix, per quanto concerne la sincronizzazione tra *padre* e *figlio*, prevede che ogni processo padre possa sospendersi (mediante la system call `wait`) in attesa della terminazione dell'esecuzione del processo figlio. Questa sincronizzazione permette al padre di raccogliere l'eventuale stato di terminazione del figlio. Infatti, ogni processo che termina volontariamente invocando la system call `exit`, può trasferire al padre un valore che rappresenta lo stato di terminazione: questo valore può essere quindi recepito dal padre mediante la `wait`.

La sintassi di `exit` è descritta dalla dichiarazione seguente:

```
void exit(int status);
```

La `exit` provoca la terminazione del processo che l'invoca e quindi è sempre una chiamata senza ritorno. Il parametro `status` permette al processo che termina di comunicare al padre informazioni (sintetizzate in un valore intero) sul suo stato di terminazione (ad esempio, l'esito della sua esecuzione).

La seguente dichiarazione descrive la sintassi di `wait`:

```
int wait(int *status);
```

Il parametro `status` è l'indirizzo della variabile nella quale verrà eventualmente memorizzato lo stato di terminazione del figlio. Il risultato prodotto dall'invocazione a `wait` è il pid del processo terminato, oppure (se la chiamata fallisce) un codice di errore (un valore intero < 0).

La `wait` può avere un effetto sospensivo sul processo che la chiama. Questo è il caso in cui il processo chiamante ha tutti i figli in attività (cioè, non ancora terminati); la sospensione del padre verrà interrotta non appena il primo figlio sarà terminato. Se, invece, almeno un figlio è già terminato ed il suo stato non è stato ancora rilevato (cioè, il figlio è nello stato *zombie*), la `wait` ritorna immediatamente il controllo al processo padre.

Il valore della variabile `*status`, in seguito ad una chiamata a `wait`, integra informazioni relative a come il figlio è terminato (volontariamente o involontariamente) ed eventualmente al suo stato di terminazione, in accordo con le seguenti convenzioni.

Se il byte meno significativo di `*status` è zero², la terminazione del figlio è avvenuta volontariamente (ad esempio mediante una `exit`). In questo caso, nel byte più significativo è memorizzato lo stato di terminazione (il valore del parametro attuale eventualmente passato alla `exit` dal figlio). Invece, se il byte meno significativo di `*status` è diverso da zero, la terminazione è avvenuta involontariamente. In particolare, il suo valore descrive l'evento (il *segnalet*) che ha provocato la terminazione del figlio. L'uso di `wait` e `exit` è esemplificato nel programma in figura 7.8.

² Assumiamo implicitamente che l'intero sia rappresentato da 16 bit. Se questa condizione non è verificata, nell'header file `<sys/wait.h>` sono definite alcune macro che consentono la gestione dello stato in modo astratto dalla rappresentazione.

```
#include <stdio.h>

int main()
{
    int pid, st;
    pid=fork();
    if (pid==0) {>>>
        printf("figlio");
        exit(0);
    }
    else {
        pid=wait(&st);
        printf("terminato processo figlio n.%d\n", pid);
        if ((char)st==0)
            printf("term. volontaria: stato %d\n",
                   st>>8);
        else
            printf("term. involontaria: segnale %d\n",
                   (char)st);
    }
}
```

Figura 7.8 Esempio di uso di exit e wait.

Come si è già detto, è quindi necessario che il sistema mantenga l'immagine di ogni figlio (anche se la sua esecuzione è già terminata) fino a che il padre non ne rilevi la terminazione (o non termini anch'esso la propria esecuzione). Per questo motivo, ogni processo che termina la propria esecuzione può transitare nello stato zombie per permettere la sincronizzazione con il processo padre.

In ogni caso, se il processo che termina ha uno o più figli, essi, dopo la terminazione del padre, vengono adottati da un processo di sistema (`init`, che ha sempre `pid` uguale a 1).

Sostituzione di codice Ogni nuovo processo, sebbene nasca eseguendo lo stesso programma del processo che l'ha creato, può tuttavia sostituire il programma (codice e dati) eseguito con un altro scelto arbitrariamente. Questa operazione può essere effettuata mediante la system call `exec1`³, la cui sintassi è descritta dalla seguente dichiarazione:

```
int exec1(char * path, char * arg0,...,char * argN, (char *)0);
```

Come si può notare, la `exec1` prevede una lista di parametri di lunghezza variabile, terminata dal puntatore nullo. Riguardo al significato dei parametri: `path` è il nome (assoluto o relativo) del file contenente il codice del programma da caricare. Così

³ Unix fornisce un'intera famiglia di system call per la sostituzione del codice; in questa sede, per esigenze di sintesi presenteremo solo `exec1`, rimandando ai testi specifici su Unix la trattazione delle altre system call della famiglia.

come dalla linea di comando dello *shell* di Unix è possibile richiedere l'esecuzione di un programma (o *comando*) *arg0* specificandone eventuali argomenti (*arg1*, *arg2*, ..., *argN*), la stessa convenzione è adottata nella *exec1*: il parametro formale *arg0* rappresenta il nome del programma da eseguire, mentre *arg1*, *arg2*, ..., *argN* rappresentano gli eventuali argomenti da trasferire ad esso. Se una chiamata a *exec1* viene eseguita con successo, il processo chiamante passa ad eseguire il comando *arg0* con gli argomenti *arg1*, *arg2*, ..., *argN*. In questo caso, quindi, al processo viene sostituito il codice e vengono aggiornate tutte le parti della sua immagine legate al codice.

In particolare, dopo l'esecuzione di *exec1*, il processo chiamante commuta a un nuovo programma e pertanto avrà *codice*, *dati globali*, *stack* e *heap* nuovi; di conseguenza nella sua immagine viene sostituita la *text structure* con una nuova struttura associata al nuovo codice. Tuttavia, nonostante il processo esegua un programma diverso, esso mantiene la stessa *process structure*, nella quale vengono modificate soltanto le informazioni dipendenti dal codice (ad esempio, il riferimento indiretto al codice); tutti gli altri attributi (come ad esempio, il pid del processo e del padre) vengono mantenuti invariati. Inoltre esso mantiene la stessa *user structure*, in cui vengono aggiornati soltanto quegli attributi dipendenti dal codice (ad esempio, il valore del Program Counter); di conseguenza, eventuali risorse allocate al processo prima della chiamata ad *exec1* (come, ad esempio, i file aperti), rimangono ancora assegnate al processo anche dopo la chiamata.

Va infine osservato che ogni chiamata a *exec1* eseguita con successo è una chiamata “senza ritorno”, in quanto il processo chiamante perde ogni riferimento al vecchio codice; ne consegue che nel testo di un programma Unix eventuali istruzioni successive a una chiamata a *exec1* verranno eseguite soltanto in caso di fallimento della chiamata stessa.

Nel programma mostrato in figura 7.9, viene esemplificato l'uso di *exec1* per impostare a un nuovo processo l'esecuzione del codice del comando di *shell ls*.

```
int main(int argc, char *argv[])
{
    int pid, status;
    pid=fork();
    if (pid==0) {
        exec1("/bin/ls", "ls", "-l", "pippo", (char *)0);
        printf("exec fallita!\n");
        exit(1);
    }
    else if (pid>0) {
        pid=wait(&status);
        /* gestione dello stato...*/
    }
    else printf("fork fallita!");
}
```

Figura 7.9 Esempio di uso di *exec1*.

7.4.4 Lo scheduling in Unix

Unix è un sistema a divisione di tempo, pertanto lo scheduler di CPU utilizza un algoritmo basato su prelazione per offrire tempi di risposta ridotti ai processi interattivi. In particolare, ad ogni processo è associato un livello di priorità. L'intervallo di variazione delle priorità comprende sia valori positivi che negativi: più grande è il valore, più bassa è la priorità. In particolare, valori negativi sono riservati ai processi eseguiti in modalità kernel, i valori positivi sono associati a processi di utente.

Lo scheduler utilizza un insieme di code, ad ognuna delle quali è associato un insieme (un sotto-intervallo) di priorità: processi con lo stesso livello di priorità sono collocati nella stessa coda di scheduling. Ad ogni quanto di tempo lo scheduler seleziona la coda con priorità più elevata (cioè con valore più basso) che contiene almeno un processo: su di essa esegue un algoritmo del tipo *round robin*, fino ad esaurimento dei processi in coda, o fino a che non compare un processo in una coda a più alta priorità.

I valori di priorità sono dinamici e vengono ricalcolati periodicamente ad ogni secondo con l'obiettivo di aumentarli rispetto ai valori iniziali al crescere del tempo di CPU utilizzato dai processi. Inoltre, ogni processo può volontariamente diminuire la propria priorità mediante la system call *nice*. È il caso, ad esempio, di un processo che ha bisogno di utilizzare la CPU per un lungo tempo e per il quale non è richiesta una particolare velocità di esecuzione: il processo, quindi, potrebbe diminuire la propria priorità (cioè, aumentare il proprio *valore* di priorità) per favorire gli altri processi.

7.5 La gestione della memoria nel sistema Unix

Le più recenti versioni di Unix (dalla versione 3 del BSD in avanti) forniscono l'astrazione di memoria virtuale (vedi paragrafo 4.2.1), sia per consentire ad ogni processo di indirizzare un'area di memoria di dimensioni superiori a quelle della memoria fisica, sia per svincolare il grado di multiprogrammazione dai limiti della memoria effettivamente disponibile.

Il sistema di gestione della memoria in Unix si basa su paginazione e segmentazione: in particolare viene adottato il modello a segmentazione paginata (vedi paragrafo 4.3.4). Infatti, lo spazio di indirizzamento di ogni processo è segmentato. Esso, cioè, viene partizionato in tre segmenti distinti: codice (*code segment*), dati (*data segment*) e stack (*stack segment*). L'allocazione dei segmenti viene gestita con la tecnica della paginazione a domanda (vedi capitolo 4) effettuata dal kernel, con il supporto del processo di sistema *pagedaemon* (individuato dal valore di pid 2), che periodicamente (ogni 250 ms) si occupa della sostituzione delle pagine.

Il collegamento tra pagine logiche e pagine fisiche viene mantenuto all'interno di una tabella delle pagine. Inoltre, il kernel mantiene una descrizione dello stato di allocazione della memoria all'interno della tabella delle pagine fisiche (la *core map*), nella quale ogni elemento rappresenta una pagina fisica, e contiene le informazioni relative ad essa. Ad esempio, ogni elemento della *core map*, oltre a contenere l'indirizzo della pagina fisica che rappresenta, indica se essa è libera oppure allocata; se allocata, registra inoltre le informazioni relative alla pagina logica in essa allocata e al processo a cui appartiene.

La sostituzione delle pagine si basa sull'algoritmo di seconda chance (vedi paragrafo 4.3.3). Il processo *pagedaemon* esegue l'algoritmo di sostituzione delle pagine soltanto se il numero delle pagine fisiche libere è inferiore a un valore di soglia prefissato, rappresentato dalla costante di sistema *lotsfree*. Può darsi, tuttavia, che nonostante l'intervento del *pagedaemon*, la frequenza di paginazione sia troppo elevata, ed il numero di pagine libere rimanga comunque inferiore a *lotsfree*: in questo caso interviene lo *swapper* (vedi capitolo 4) che, in modo più drastico, provvede al trasferimento di uno o più processi dalla memoria centrale a quella secondaria. Per rilevare questa situazione il sistema definisce altre due costanti di sistema: *minfree* (che esprime il numero minimo di pagine fisiche libere necessarie per evitare lo swap-out dei processi) e *desfree* (che esprime il numero medio *desiderabile* di pagine fisiche libere). La relazione tra le tre costanti di sistema è la seguente:

$$\text{lotsfree} > \text{desfree} > \text{minfree}$$

Lo *swapper*, quindi, interviene se sono soddisfatte le seguenti condizioni:

- il numero di pagine fisiche libere è minore di *minfree*;
- numero medio di pagine fisiche libere nell'unità di tempo è minore di *desfree*.

In questo modo si riesce a contenere l'uso della CPU da parte del *pagedaemon*.

7.6 Il file system

Il file system è la struttura all'interno della quale vengono memorizzati e gestiti i file. Come è stato mostrato nel capitolo 6, le astrazioni sulle quali si basa un file system sono il *file* ed il *direttorio*: il file è l'unità logica di memorizzazione e il diretto-rio è la struttura che permette di raggruppare file (e direttori).

Una delle caratteristiche fondamentali del file system di Unix è l'omogeneità: ogni risorsa del sistema è rappresentata all'interno del file system sotto forma di file. In particolare, esistono tre tipi di file: il file *ordinario*, il file *speciale* e il file *direttorio*.

Il file ordinario rappresenta un insieme di informazioni effettivamente allocate in memoria di massa. Il file speciale, invece, rappresenta un dispositivo fisico (ad esempio, una stampante o una porta di comunicazione): a questo scopo in tutte le installazioni di Unix è presente un diretto-rio (`/dev`) che contiene tutti i file speciali relativi alle periferiche installate nel sistema.

Il terzo tipo di file rappresenta il concetto astratto di diretto-rio: come si vedrà, esso è realizzato mediante un particolare file che contiene la descrizione dell'insieme dei file e direttori in esso contenuti. Questa omogeneità di rappresentazione per file e dispositivi consente di utilizzare le stesse modalità di input e output dei processi sia nei confronti di file che nei confronti di periferiche. Ad esempio, la stampa su un certo dispositivo di output può essere effettuata con gli stessi strumenti utilizzabili per la scrittura su file.

7.6.1 Organizzazione logica del File System di Unix

Il file system di Unix ha una struttura gerarchica che può essere rappresentata da un grafo aciclico diretto (nel caso più semplice un albero, vedi figura 7.10). Nella figu-

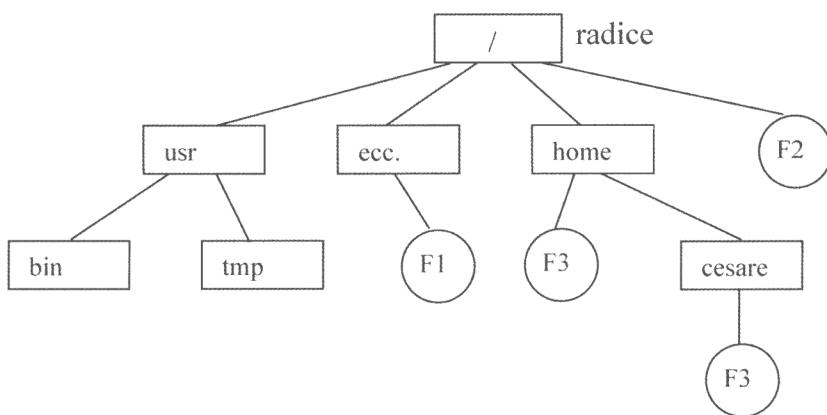


Figura 7.10 Organizzazione logica del file system di Unix.

ra i direttori sono rappresentati da nodi dell'albero (rettangoli), mentre i file ordinari e speciali vengono rappresentati da foglie (cerchi); è possibile notare che esiste un direttorio che rappresenta la *radice* dell'albero, che, per convenzione, viene denotato con il simbolo “/”.

Ogni utente può spostarsi attraverso il file system utilizzando il comando di shell `cd`: in ogni istante, ad ogni utente è associato un direttorio (il direttorio *corrente*) che rappresenta la sua posizione all'interno della struttura ad albero.

Il nome di un file in Unix indica il cammino da compiere per raggiungere il file a partire da un direttorio di riferimento. In particolare, per riferire un file (o un direttorio), è possibile utilizzare due modalità: il nome relativo o il nome assoluto. Il nome assoluto indica la posizione del file rispetto al direttorio radice (“/”), mentre il nome relativo indica la posizione del file rispetto al direttorio corrente. Nella costruzione dei nomi di file è importante conoscere le convenzioni previste per indicare il direttorio corrente, denotato con il simbolo “.”, ed il direttorio *padre* di un dato direttorio D (quello, cioè, a “monte” di D nel grafo), denotato con il simbolo “..”. Ogni passo del cammino che definisce il nome (relativo o assoluto) di un file è separato dal successivo mediante il simbolo “/”.

Ad esempio, si consideri il file system di figura 7.10. Nell'ipotesi che il direttorio corrente sia `home`, è possibile riferire il file `F4` con il suo nome assoluto (`/home/cesare/F4`) oppure con il suo nome relativo (`cesare/F4`). Analogamente, per riferire il direttorio `tmp` si possono usare indifferentemente il nome assoluto `/usr/tmp`, oppure il nome relativo `../usr/tmp`.

Nonostante finora siano stati mostrati esempi in cui il file system è rappresentabile da strutture ad albero, più in generale esso è rappresentato da grafici aciclici. Infatti ad ogni file possono essere associati uno o più nomi simbolici: questo comporta che la stessa foglia possa derivare da due nodi (direttori) diversi. In questo caso si parla di *linking* di file, come nell'esempio di figura 7.11: il file evidenziato ha 2 nomi (*link*) diversi; si può verificare che, in questo caso, la struttura del file system non è più un albero.

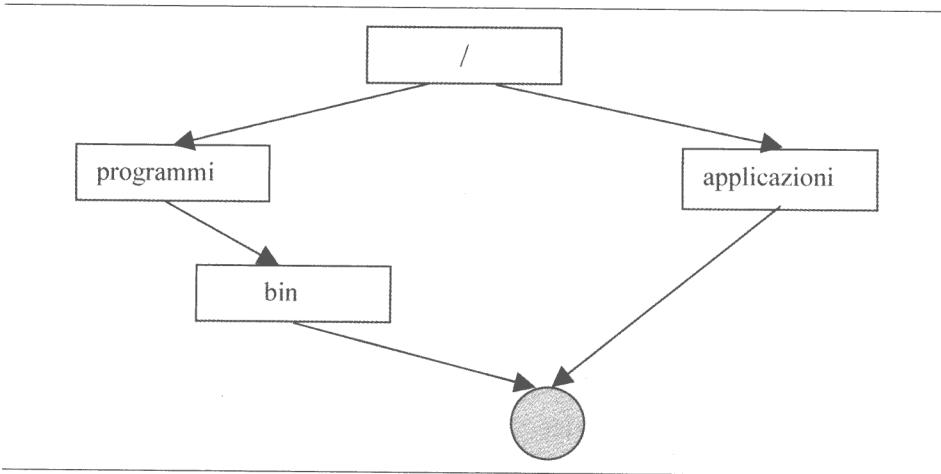


Figura 7.11 Link di file.

Va sottolineato che, nonostante il nome relativo di un file possa non essere unico, ad ogni file è associato uno ed un solo descrittore (detto *i-node*), univocamente identificato da un intero (chiamato *i-number*).

7.6.2 Protezione

Come ogni sistema operativo multiutente, anche Unix affronta e risolve alcune problematiche relative alla sicurezza ed alla protezione delle risorse, offrendo alcune funzionalità di sicurezza come l'autenticazione degli utenti ed il controllo nell'accesso delle risorse. Alcune delle più diffuse versioni di Unix hanno ottenuto la certificazione di classe C2, con riferimento al rapporto *Orange Book* del Ministero della difesa USA [66].

Il sistema Unix controlla l'accesso degli utenti utilizzando un meccanismo di autenticazione basato su password. In particolare, ogni utente è individuato univocamente da un nome logico (lo *username*) al quale è associato un valore intero (*user-id*). In ogni sistema è definito l'utente *root* (che ha sempre *user-id* uguale a 0), che rappresenta l'amministratore di sistema, e che, in generale, non ha alcuna limitazione nell'accesso alle risorse del sistema.

Gli utenti sono aggregati in gruppi, ciascuno individuabile univocamente mediante il nome logico *groupname* (o l'intero *group-id*).

Per ogni *username* è definita una *password* che consente l'autenticazione dell'utente. L'elenco degli utenti (e relativi attributi, come password, gruppo e directory *home*) è contenuto all'interno del file di sistema */etc/passwd*. Questo file, leggibile da ogni utente, può essere modificato soltanto dall'utente *root*. Per motivi di sicurezza, nel file */etc/passwd* le password degli utenti vengono memorizzate in forma crittografata⁴.

⁴ Ad ogni password, cioè, viene applicato un algoritmo di crittografia che la trasforma in modo tale che risalire alla password vera e propria da quella crittografata risulti un compito arduo.

L'accesso di ogni utente al sistema, avviene attraverso il *login*. In questa fase, l'utente deve fornire al sistema i propri *username* e *password*, affinché possa avvenire l'autenticazione. Solo se l'autenticazione ha successo (cioè, se la coppia di informazioni *<username, password>* è presente nel file */etc/passwd*), l'utente ottiene l'accesso al sistema.

Nel sistema Unix il meccanismo che controlla l'accesso alle risorse da parte degli utenti è basato sulle liste di controllo degli accessi (*Access Control List*, o *ACL*, vedi capitolo 6): per ogni risorsa sono stabiliti i diritti di accesso associati ad ogni utente. Ricordando che in Unix ogni risorsa è presente nel file system come file, la protezione interessa quindi file, direttori e periferiche.

Ad ogni file è associato un utente proprietario: il proprietario stabilisce la politica di protezione da applicare al file, concedendo o negando agli altri utenti il permesso di accedervi.

In particolare, per ogni file sono previste tre modalità di accesso:

Lettura: il contenuto del file può essere letto;

Scrittura: il contenuto del file può essere modificato;

Esecuzione: il file può essere eseguito; in questo caso è necessario che il file contenga del codice eseguibile, o una sequenza di comandi (nel caso di file comandi).

Per ogni file l'*ACL* è sintetizzata da 9 bit (i *bit di protezione*), che indicano quali sono i diritti di accesso al file per ogni utente del sistema. In particolare, dato un file, ogni utente del sistema può rientrare in una delle tre categorie seguenti:

User: l'utente è il proprietario;

Group: l'utente non è il proprietario, ma appartiene al gruppo del proprietario;

Others: l'utente non appartiene al gruppo del proprietario.

I 9 bit di protezione sono suddivisi in tre terne di bit: ogni terna è associata ad una delle tre categorie di utenti (*User, Group e Others*), come mostrato in figura 7.12.

I tre bit di ogni terna rappresentano rispettivamente il diritto di lettura (indicato con "r"), il diritto di scrittura (indicato con "w") ed il diritto di esecuzione (indicato con "x"): se un certo bit ha il valore 1, significa che il diritto di accesso che esso rappresenta è concesso agli utenti della categoria alla quale la tripletta è associata; se invece ha valore 0, il diritto è negato.

L'esempio di figura 7.12 mostra i bit di protezione associati a un file: in questo caso, il file è leggibile, scrivibile ed eseguibile per il proprietario (prima terna di bit), è soltanto leggibile per gli utenti del gruppo a cui appartiene il proprietario (seconda terna di bit), e non è accessibile in alcun modo da tutti gli altri utenti.

r	w	x	r	w	x	r	w	x
1	1	1	1	0	0	0	0	0
User			Group			Others		

Figura 7.12 Esempio di politica di protezione associata a un file.

Oltre ai 9 bit che esprimono i permessi di accesso, ad ogni file sono associati altri tre bit: il bit SUID (set user id), il bit SGID (set group id) e lo STIcky (Save Text Image) bit. Questa terna di bit è significativa soltanto quando il file è eseguibile. Come si è visto, infatti, l'esecuzione di un programma determina la creazione di un nuovo processo. Ad ogni processo il sistema associa dinamicamente uno *user-id* (e un *group-id*): il default prevede che siano quelli dell'utente che richiede l'esecuzione del programma. Tuttavia è possibile modificare questa impostazione settando il valore di SUID (e SGID): infatti, se il bit SUID di un file eseguibile vale 1, ogni processo che esegue il file assumerà l'*user-id* del *proprietario* del file; analogamente, se il bit SGID di un file eseguibile vale 1, ad ogni processo che esegue il file verrà attribuito il *group-id* del *proprietario* del file. In questo modo chi lancia un programma può assumere temporaneamente l'identità dell'utente proprietario del file eseguibile nel quale è contenuto il codice del programma.

Un esempio tipico dell'impiego di SUID e SGID è fornito dal meccanismo di aggiornamento delle password, realizzato dal comando `passwd`, il cui codice è contenuto nel file `/bin/passwd`. Come già visto, le password sono memorizzate all'interno del file di sistema `/etc/passwd`; per ovvi motivi di sicurezza soltanto l'utente *root* può scrivere su `/etc/passwd`. Tuttavia, per concedere la possibilità ad ogni utente di modificare la propria password (e quindi di modificare `/etc/passwd`), il file `/bin/passwd` ha il valore di entrambi i bit SUID e SGID uguale a 1: in questo modo, chiunque esegua il comando, crea un processo che assume lo *user-id* del proprietario di `/bin/passwd` (cioè, di *root*) e quindi acquisisce temporaneamente i diritti dell'utente *root*. Pertanto, nonostante il comando possa essere stato impartito da un utente non privilegiato, tale processo acquisirà il permesso di scrittura sul file delle password (`/etc/passwd`) e potrà quindi apportarvi le modifiche previste dal comando.

Infine, se lo STIcky bit ha valore 1, esso mantiene il codice del processo memorizzato nell'area di swap, anche dopo la sua terminazione. Ad esempio, comandi di uso frequente (come il compilatore `cc`) hanno lo STIcky bit a 1: in questo modo, essendo già parzialmente caricata l'immagine del processo, l'inizializzazione del processo che esegue il comando può avvenire in modo particolarmente rapido.

7.6.3 Organizzazione fisica del File System di Unix

Il file system di Unix trova allocazione in un unico dispositivo fisico, generalmente un disco. Il disco viene formattato in blocchi fisici di dimensione costante e prefissa (ad esempio 4096 Bytes). La superficie del disco viene suddivisa in 4 regioni (vedi figura 7.13): *BootBlock*, *SuperBlock*, *DataBlocks* e *i-List*.

La prima regione occupa un blocco fisico, allocato a un indirizzo prefissato, e contiene il programma di inizializzazione del sistema, che verrà eseguito nella fase di *bootstrap*.

La regione *SuperBlock*, anch'essa della dimensione di un blocco, descrive l'allocatione del file system; in particolare contiene i limiti delle 4 regioni, il puntatore alla lista dei blocchi liberi e il puntatore alla lista degli i-node liberi.

L'estensione dell'area *DataBlocks* è tipicamente molto maggiore delle altre, in quanto essa rappresenta la zona del disco effettivamente disponibile per la memorizzazione dei file; in particolare, essa contiene i blocchi effettivamente allocati ed i

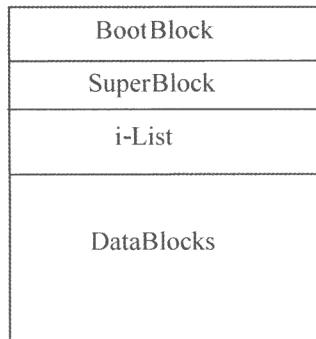


Figura 7.13 Organizzazione fisica del file system Unix.

blocchi liberi. Questi ultimi sono organizzati in una lista collegata, il cui indirizzo è contenuto nel *SuperBlock*.

Infine, la *i-List* contiene il vettore di tutti i descrittori (*i-node*) dei file, direttori e dispositivi presenti nel file system. Ogni *i-node* viene quindi riferito mediante l'indice *i-number* che lo identifica univocamente.

L'*i-node* contiene gli attributi associati al file, tra i quali è importante citare:

- *Tipo*: indica se il file è ordinario, directory o file speciale (cioè, rappresenta un dispositivo).
- *Proprietario e Gruppo*: indicano rispettivamente chi è l'utente proprietario del file e qual è il gruppo di appartenenza del proprietario.
- *Dimensione*: è il numero di blocchi occupati dal file in memoria di massa.
- *Data*: indica la data dell'ultima modifica effettuata sul file;
- *Link*: il numero dei nomi che riferiscono il file; se il file ha un solo nome, questo valore è 1.
- *I bit di protezione*: è l'insieme dei 12 bit (i 9 bit dei permessi, SUID, SGID, e STicky bit) che esprime la politica di protezione da applicare al file (vedi paragrafo 7.7.2).
- *Vettore di Indirizzamento*: è costituito da un insieme di indirizzi (ad esempio, 13 puntatori) che consente l'indirizzamento dei blocchi di dati sui quali è allocato il file.

In particolare, il metodo di allocazione utilizzato in Unix è ad indice, a *più livelli* di indirizzamento. Infatti, gli elementi del vettore di indirizzamento rappresentano puntatori a blocchi appartenenti alla regione *DataBlocks*. Nell'ipotesi che il vettore contenga 13 indirizzi (come nelle prime versioni del sistema operativo) i primi 10 elementi del vettore riferiscono direttamente blocchi di dati impiegati per l'allocazione del file (indirizzamento *diretto*); i rimanenti elementi (dall'11-simo al 13-simo) vengono utilizzati per riferire blocchi che contengono a loro volta indirizzi (indirizzamento *indiretto*). In particolare: l'11-simo indirizzo punta a un blocco contenente, a sua volta, indirizzi di blocchi dati (primo livello di *indirettezza*), il 12-simo indirizzo punta a un blocco contenente a sua volta indirizzi di blocchi che contengono ancora

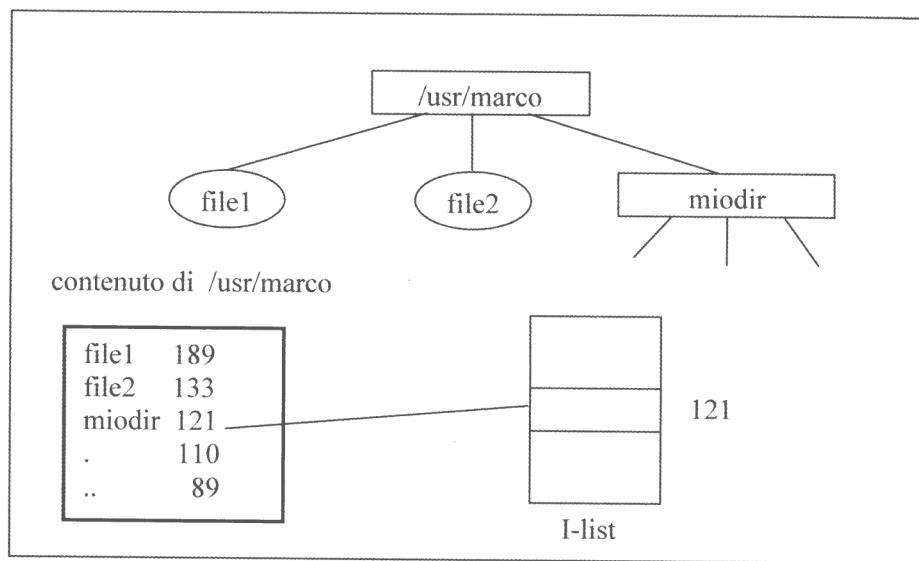


Figura 7.14 Realizzazione del direttorio.

indirizzi di blocchi di dati (secondo livello di *indirezione*), e i 13-simo consente l’accesso ai blocchi di dati mediante tre livelli di indirezione.

Questo sistema di allocazione fornisce la possibilità di indirizzare file di dimensione elevata. Ad esempio, se la dimensione del singolo blocco è di 512 byte, e se riserviamo 32 bit per ogni indirizzo, ogni blocco è in grado di contenere 128 indirizzi. Se l’i-node contiene 13 indirizzi, il numero di blocchi di dati utilizzabili per allocare un file è dato dal numero di blocchi indirizzabili direttamente (10, cioè 5 KB), più quelli accessibili indirettamente ($128 \times 512 + 128 \times 128 \times 512 + 128 \times 128 \times 128 \times 512$ B). La dimensione massima del file è quindi dell’ordine del gigabyte (esattamente: $5\text{KB}+64\text{KB}+8\text{MB}+1\text{GB}=1.056.837\text{ KB}$).

Il Direttorio Il direttorio è rappresentato nel *file system* da un file, il cui contenuto ne descrive la struttura logica. Infatti, ogni file-direttorio (vedi figura 7.14) contiene un insieme di record logici ognuno dei quali rappresenta un elemento (file o directory) appartenente al direttorio considerato. In particolare, ogni record contiene la coppia di informazioni *<nome relativo, i-number>* associate al file che esso rappresenta.

7.6.4 Strutture dati del kernel per l’accesso a file

In questo paragrafo vengono descritte le strutture dati gestite dal kernel di Unix per realizzare le operazioni di accesso al file system. A questo scopo è importante introdurre preliminarmente alcuni concetti generali riguardanti le modalità di accesso previste nel sistema.

Ogni file è organizzato come una sequenza di byte (o *stream*): il *record logico* (vedi paragrafo 6.1) è quindi il byte. Il file può essere acceduto in varie modalità: ad esempio: lettura, scrittura e scrittura in aggiunta (*append*). Ogni sessione di accesso ad un file F deve essere preceduta dall'operazione di *apertura*, mediante la quale vengono aggiornate le strutture dati del kernel con le informazioni relative ad F.

Il metodo di accesso adottato nel sistema Unix è quello sequenziale: ad ogni file aperto è associato un *I/O pointer*, che indica implicitamente il prossimo elemento a cui accedere (in lettura o scrittura). Ogni lettura/scrittura di un record logico del file provoca un avanzamento dell'I/O pointer all'elemento successivo nella sequenza. Pertanto il funzionamento delle primitive di accesso ai file è del tutto simile a quello delle operazioni astratte *readnext/writenext* citate nel capitolo 6.

I meccanismi di accesso sono realizzati all'interno del kernel il quale mantiene alcune strutture dati specifiche per la gestione dei file. Per comprendere appieno come viene realizzato l'accesso ai file è quindi importante analizzare le caratteristiche di tali strutture dati.

A livello globale il kernel mantiene la *Tabella dei File Aperti di Sistema* (che nel seguito verrà denotata con la sigla *TFAS*): questa struttura dati contiene un elemento per ogni file aperto nel sistema. Più precisamente, viene allocato un elemento nella *TFAS* per ogni operazione di apertura di file: ciò significa che se due processi aprono separatamente lo stesso file, nella *TFAS* vi saranno due elementi distinti, nonostante il file acceduto dai due processi sia lo stesso. Tra le informazioni contenute nell'elemento della *TFAS*, vi è l'*I/O pointer*, che indica il prossimo byte da leggere/scrivere nel file aperto. Inoltre, l'elemento della *TFAS* contiene un riferimento all'i-node del file aperto, che il sistema copia e mantiene in memoria centrale per tutta la sessione di accesso al file. In particolare, gli i-node dei file aperti sono inseriti all'interno di un'altra struttura dati globale: la *Tabella dei File Attivi* (che riferiremo nel seguito con la sigla *TFAT*). Oltre alla *TFAS* e alla *TFAT*, a ogni processo è associata una *tabella dei file aperti* di dimensione limitata (tipicamente, 20 elementi), nella quale ogni elemento della tabella rappresenta un file aperto dal processo. La *tabella dei file aperti del processo* (nel seguito verrà indicata con la sigla *TFAP*) è organizzata a vettore: pertanto ogni elemento è individuato da un indice intero, che prende il nome di *file descriptor*. Il sistema apre automaticamente i file speciali di standard input, standard output e standard error: per questo motivo i primi tre elementi della *TFAP* sono sempre automaticamente inizializzati per rappresentare standard input (*file descriptor 0*), standard output (*file descriptor 1*) e standard error (*file descriptor 2*). Ogni elemento della *TFAP* contiene un riferimento all'elemento corrispondente nella *TFAS*. La *TFAP* è una struttura accessibile soltanto dal kernel e pertanto è collocata nella User Structure del processo (vedi paragrafo 7.4.2).

La figura 7.15 mostra come le tre tabelle sono collegate tra loro. Nella figura è possibile osservare come, a partire dal file descriptor *fd*, si possa ricavare l'indirizzo del prossimo record logico da leggere/scrivere sul dispositivo di memorizzazione di massa utilizzando le informazioni distribuite nelle tre strutture dati. Come già visto, infatti, l'i-node (memorizzato nella *TFAT*) contiene il vettore di indirizzi che descrive l'allocazione del file sul disco; conoscendo inoltre il valore dell'I/O pointer, si può quindi calcolare l'indirizzo fisico (<blocco, offset>) del prossimo byte da leggere/scrivere sul disco.

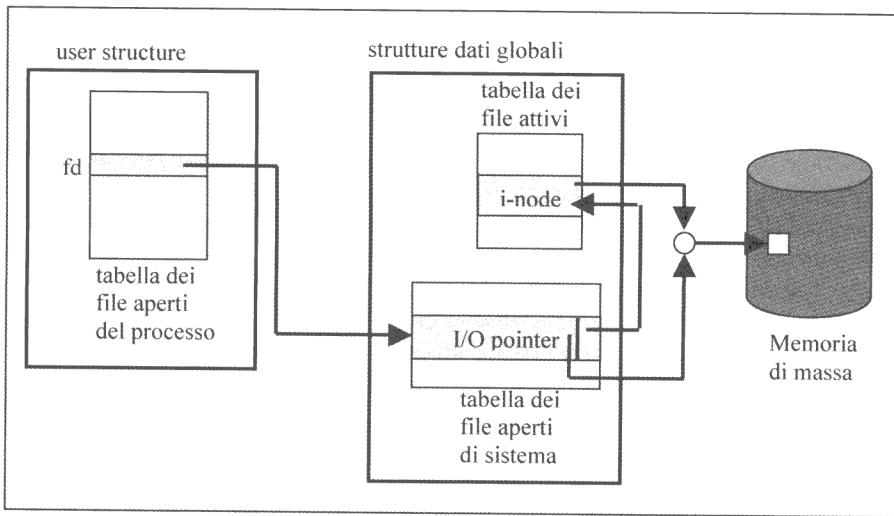


Figura 7.15 Strutture dati per l'acceso a file.

L'operazione di apertura di un file da parte di un processo P determina sulle strutture dati del kernel i seguenti effetti: viene inserito un nuovo elemento (individuato da un file descriptor) nella prima posizione libera della TFAP associata a P ; viene inserito un nuovo elemento nella tabella dei file aperti di sistema; se il file non è già in uso, l'i-node del file aperto viene copiato dalla i-list (in memoria secondaria) alla TFAT.

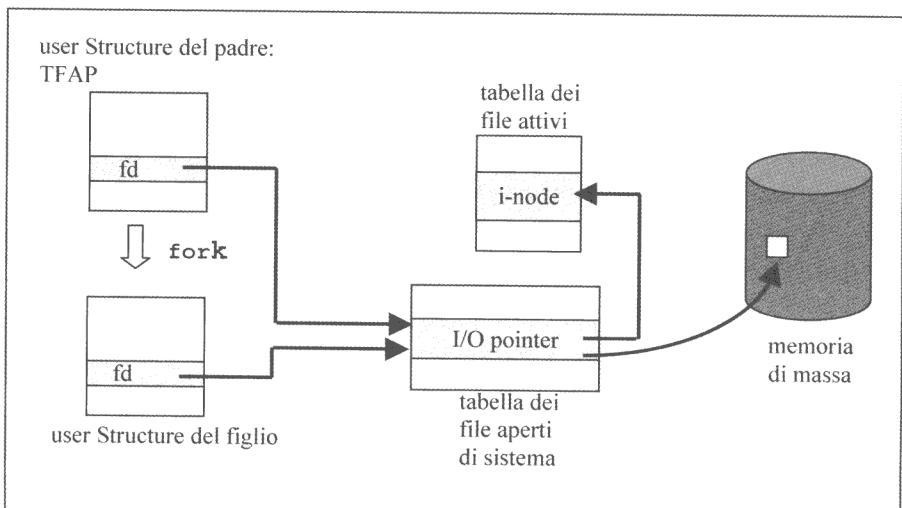


Figura 7.16 Condivisione di I/O pointer tra padre e figlio.

Ricordando quanto detto a proposito del meccanismo di creazione dei processi mediante `fork()` (vedi paragrafo 7.4.3), è interessante osservare che, poiché ogni nuovo processo eredita dal padre una copia della *User Structure*, esso eredita quindi anche la tabella dei file aperti di processo: pertanto se il padre ha aperto un file prima della chiamata `fork`, il figlio ne eredita la entry corrispondente nella *TFAP*, e pertanto condivide lo stesso elemento della *TFAS* con il padre. Questa situazione, illustrata in figura 7.16, è l'unico caso in cui due processi che accedono allo stesso file, ne condividono anche l'*I/O pointer*.

7.6.5 System Call per l'accesso a file

Le principali operazioni di accesso a file sono effettuabili dai processi mediante particolari system call che verranno illustrate brevemente nel seguito.

L'apertura di un file si può effettuare attraverso la primitiva:

```
int open(char nomefile[], int mode, int prot);
```

Il parametro `nomefile` è il nome del file da aprire; il parametro `mode`, invece, esprime il modo di accesso: ad esempio va specificato 0 se si vuole leggere il file, 1 se si vuole scrivere il file, ecc.. Nell'header file `<fcntl.h>` sono definite alcune costanti che esprimono il modo di apertura in maniera simbolica; per esempio `O_RDONLY` per lettura, `O_WRONLY` per scrittura, `O_APPEND` per scrittura in aggiunta ecc. Alcuni modi di apertura possono essere combinati tra di loro mediante l'operatore logico “”: per esempio è possibile specificare come valore del parametro `mode` la composizione `O_WRONLY|O_CREAT`, per specificare che il file da scrivere, se non esiste, va creato. Il parametro `prot` è opzionale: nel caso in cui l'apertura provochi la creazione di un nuovo file (ad esempio, se `mode` è `O_CREAT`) esso esprime i 12 bit di protezione associati al file.

Se la `open` ha successo, il file viene aperto nel modo richiesto, e l'*I/O pointer* posizionato sul primo record logico, tranne nel caso di apertura in aggiunta.

Il valore restituito da una chiamata a `open` è il *file descriptor* associato al file aperto, oppure il valore -1 in caso di errore. Alcuni esempi di uso della `open` sono mostrati in figura 7.17.

Al termine di una sessione di accesso ad un file è necessario effettuare l'operazione di *chiusura*, mediante la primitiva:

```
int close(int fd);
```

L'unico parametro `fd` è il file descriptor del file da chiudere. La chiusura determina la memorizzazione del file sul disco e l'aggiornamento delle strutture del kernel. In particolare, viene eliminato l'elemento di indice `fd` dalla *TFAP*, e vengono eventualmente eliminati gli elementi corrispondenti dalla *TFAS* e dalla *TFAT*.

L'accesso vero e proprio al file, una volta aperto, viene effettuato con le system call `read` e `write`: la prima serve a leggere, la seconda a scrivere il file. Entrambe le system call richiedono che il file venga specificato mediante il file descriptor. Entrambe, inoltre, agiscono sequenzialmente sul file, a partire dalla posizione corrente (indicata dall'*I/O pointer*), spostando l'*I/O pointer* avanti di tanti byte quanti sono i byte letti o scritti.

```
#include <fcntl.h>
...
int main()
{
    int fd1, fd2, fd3;
    /* apertura di un file in modalità di lettura: */
    fd1=open("/home/giovanni/ff.txt", O_RDONLY);
    if (fd1<0) perror("open fallita");
    ...
    /* apertura di un file in modalità di scrittura: */
    fd2=open("f2.new", O_WRONLY);
    if (fd2<0) /* f2.new non esiste: */ {
        perror("open in scrittura fallita:");
        /* creazione del nuovo file f2.new: */
        fd2=open("f2.new", O_WRONLY|O_CREAT, 0777);
    }
    /*OMOGENEITA': apertura dispositivo di output:*/
    fd3=open("/dev/prn", O_WRONLY);
    ...
}
```

Figura 7.17 Uso della system call open.

La sintassi di `read` è la seguente:

```
int read(int fd, char *buf, int n);
```

dove, il parametro `fd` è il file descriptor del file, `buf` è l'area in cui trasferire i byte letti ed `n` è il numero di byte da leggere. La `read` restituisce un intero di valore minore o uguale a `n` che, se positivo, rappresenta il numero di byte effettivamente letti: se tale valore è uguale a zero, significa quindi che il file è finito. La `read` può restituire anche un valore negativo: questo indica una situazione di insuccesso, che può essere dovuta a varie cause (ad esempio, il file non è stato aperto, oppure è stato aperto in una modalità diversa dalla lettura). Ad esempio, la chiamata `read(fd, B, N)`, se eseguita con successo, produce i seguenti effetti: a partire dal valore corrente dell'I/O pointer, vengono letti (al più) `N` bytes dal file `fd` e memorizzati all'indirizzo `B`; di conseguenza, l'*I/O pointer* viene spostato avanti di `N` bytes.

La `write` ha una sintassi analoga alla `read`:

```
int write(int fd, char *buf, int n);
```

in cui il parametro `fd` è il file descriptor del file, `buf` è l'area da cui trasferire i byte da scrivere nel file ed `n` è il numero di byte da scrivere. La `write` restituisce un intero che, se positivo, rappresenta il numero di byte effettivamente scritti; se il valore restituito è negativo, esso indica invece una situazione di insuccesso. Un esempio di utilizzo delle primitive di accesso a file è mostrato nel programma di figura 7.18 nel quale viene realizzato un meccanismo di copia di file, analogo a quello realizzato dal comando `cp`.

L'insieme delle system call Unix per l'accesso e la gestione di file prevede, oltre che le system call di base appena presentate, altre system call che, per esigenze di sintesi non descriveremo in questo testo.

```

/* realizzazione comando cp (copia argv[1] in argv[2])
sintassi: cp origine destinazione */

#include <fcntl.h>
#include <stdio.h>
#define BUFDIM 1000
#define perm 0777

int main (int argc, char **argv)
{
    int status;
    int infile, outfile, nread;
    char buffer[BUFDIM];
    if (argc != 3) {
        printf (" errore \n"); exit (1);
    }
    if ((infile=open(argv[1], O_RDONLY)) < 0) {
        printf("Errore in apertura file origine!");
        exit(1);
    }
    if ((outfile=open(argv[2], O_CREAT, perm)) < 0) {
        printf("Errore in apertura file destinazione!");
        close (infile); exit(1);
    }
    while((nread=read(infile, buffer, BUFDIM)) > 0 )
        if (write(outfile, buffer, nread)< nread) {
            close(infile); close(outfile);
            exit(1);
        }
    close(infile); close(outfile);
    exit(0);
}

```

Figura 7.18 Copia di file.

Ad esempio, esistono system call per la cancellazione ed il linking di file, system call per impostare/modificare la politica di protezione associata a un file, system call specifiche per la gestione dei direttori. Per una trattazione approfondita di queste primitive il lettore interessato può consultare [87].

7.7 Interazione tra processi

Nel capitolo 3 è stato mostrato che i processi possono interagire per cooperare o competere; nel primo caso i processi interagiscono comunicando fra di loro allo scopo di perseguire obiettivi comuni, mentre la competizione si verifica quando due o più processi (eventualmente indipendenti) hanno bisogno della stessa risorsa e per operare su di essa è necessario imporre dei vincoli di sincronizzazione (ad esempio, la mutua esclusione).

Poiché i processi Unix aderiscono al modello ad ambiente locale, ogni processo

riferisce un proprio spazio di indirizzamento privato: pertanto esso non ha alcuna possibilità di condividere variabili con altri processi. Perciò, in Unix l'unica forma di interazione tra processi è la cooperazione che, mediante i meccanismi di comunicazione, permette lo scambio di informazioni tra i processi interagenti, mentre la sincronizzazione consente l'impostazione di vincoli temporali sull'esecuzione dei processi.

Gli strumenti d'interazione disponibili in Unix consentono la sincronizzazione e la comunicazione tra processi: tali meccanismi si basano su astrazioni realizzate dal kernel del sistema operativo (come, ad esempio, il canale di comunicazione tra processi) esterne allo spazio di indirizzamento dei processi interagenti ed accessibili da essi mediante specifiche chiamate di sistema operativo. In particolare, la sincronizzazione può avvenire attraverso lo scambio di eventi (utilizzando i *segnali*), mentre la comunicazione avviene mediante lo scambio di messaggi attraverso un canale (utilizzando *pipe* e *socket*). Nel seguito, i principali strumenti di interazione tra processi Unix (segnali e pipe) verranno descritti con maggiore dettaglio e corredati da alcuni esempi di utilizzo. Poiché le socket vengono tipicamente impiegate per la comunicazione tra processi in ambito distribuito, l'analisi delle *socket*, invece, è rinviata all'appendice A, dove verranno forniti alcuni cenni sull'interazione tra processi nelle reti di calcolatori.

7.7.1 Sincronizzazione: i segnali

In Unix la sincronizzazione avviene mediante i segnali, meccanismi offerti dal sistema operativo che consentono la notifica di *eventi* asincroni tra processi.

Il segnale è un evento inviato da un processo mittente a uno o più processi destinatari. Il segnale provoca nel processo destinatario una reazione del tutto analoga a quella provocata nel sistema di elaborazione da un'interruzione hardware: il processo viene interrotto per permettere l'eventuale esecuzione di una routine di gestione dell'evento. Per questo motivo il segnale viene spesso presentato come un'interruzione "software". Come per i messaggi, i segnali hanno quindi un mittente e uno (o più) destinatari. Dal punto di vista del mittente il segnale potrebbe essere assimilato a un messaggio privo di contenuto. Tuttavia, differente è il punto di vista del destinatario: diversamente dallo scambio di messaggi (in cui il destinatario, per ricevere un messaggio deve eseguire esplicitamente l'operazione *receive*, vedi capitolo 3), la ricezione di un segnale provoca una reazione immediata nel processo destinatario, qualunque istruzione esso stia eseguendo.

Pur essendo strumenti generali per esprimere la sincronizzazione tra processi, i segnali vengono largamente usati dal sistema operativo per notificare ai processi di utente il verificarsi di eccezioni: ad esempio, in caso di violazione dei limiti di memoria, il *kernel* del sistema operativo invia al processo responsabile dell'eccezione un segnale che causa istantaneamente la terminazione del processo destinatario.

In generale, quando un processo riceve un segnale, può comportarsi in tre modi diversi:

1. gestire il segnale con una funzione *handler* definita dal programmatore,
2. eseguire un'azione predefinita dal sistema operativo (azione di *default*),
3. ignorare il segnale.

```

#define SIGHUP 1      /* Hangup (POSIX). Exit */
#define SIGINT 2      /* Interrupt (ANSI). Core dump */
#define SIGQUIT 3     /* Quit (POSIX). Core dump */
#define SIGILL 4      /* Ill. instr. (ANSI). Core dump */

...
#define SIGKILL 9     /* Kill, unblockable (POSIX). Exit */
#define SIGUSR1 10    /* User-def signal 1 (POSIX). Exit */
#define SIGSEGV 11    /* Seg. violation (ANSI). Core dump */
#define SIGUSR2 12    /* User-def signal 2 (POSIX). Exit */
#define SIGPIPE 13    /* Broken pipe (POSIX). Exit */
#define SIGALRM 14    /* Alarm clock (POSIX). Exit */
#define SIGTERM 15    /* Termination (ANSI). Exit */

...
#define SIGCHLD 17    /* Child st. changed (POSIX). Ignore */
#define SIGCONT 18    /* Continue (POSIX). Ignore */
#define SIGSTOP 19    /* Stop, unblockable (POSIX). Stop */

```

Figura 7.19 Contenuto del file signal.h.

Nei primi due casi, il processo destinatario reagisce in modo asincrono al segnale. In particolare, l'esecuzione del programma viene interrotta per eseguire l'azione associata (*handler o default*). Una volta terminata l'azione associata al segnale, il processo, se non è terminato, riprende l'esecuzione dall'istruzione successiva all'ultima eseguita prima dell'interruzione.

Per ogni versione di Unix esistono diversi segnali (in Linux, ad esempio, sono definiti 32 segnali), ognuno identificato da un intero: ogni segnale, infatti, è associato a un particolare evento e prevede una specifica azione di default. La lista dei segnali definiti nel sistema è memorizzata nel file di sistema `signal.h` (vedi figura 7.19).

Come si può notare in figura 7.19, nell'header file `signal.h`, non solo vengono elencati i segnali disponibili, ma ad ognuno di essi viene associato un nome simbolico (tramite la direttiva `define`), con il quale è possibile riferirlo; per ogni segnale, inoltre, il commento nella figura descrive sinteticamente l'evento al quale è associato e l'azione di default. È possibile notare, a questo proposito, che vi sono due segnali “user-defined”: `SIGUSR1` e `SIGUSR2`. Questi segnali, a differenza di tutti gli altri, non sono associati ad alcun evento, e vengono tipicamente impiegati dai processi di utente per realizzare specifiche politiche di sincronizzazione. Va anche osservato che esistono alcuni segnali che non sono né intercettabili mediante `handler`, né ignorabili (ad esempio il segnale `SIGKILL`, che è *unblockable*, vedi figura 7.19). Per questi segnali, associati ad eventi particolarmente prioritari (ad esempio la terminazione forzata, o *killing*, di un processo) l'unica reazione possibile del destinatario è quella prescritta dall'azione di *default* (ad esempio, per `SIGKILL`, la terminazione del processo).

7.7.2 System Call per l'uso dei segnali

Ogni potenziale destinatario di un segnale può configurare la sua eventuale reazione al ricevimento di tale evento utilizzando la system call `signal`, la cui sintassi è descritta come segue:

```
void (* signal(int sig, void (*func)()))(int);
```

dove il parametro *sig* è l'intero (o il nome simbolico) che individua il segnale da gestire, mentre *func* è il puntatore alla funzione che esprime l'azione da associare al segnale; più precisamente *func* può puntare alla routine di gestione dell'interruzione (*handler*), valere **SIG_IGN** (nel caso in cui il segnale vada ignorato), oppure valere **SIG_DFL** (nel caso in cui debba essere eseguita l'azione di default). La system call *signal* restituisce un puntatore al precedente gestore del segnale, oppure la costante **SIG_ERR**, nel caso di errore.

Dopo l'invocazione a *signal*, quindi, in caso di ricezione del segnale specificato dal primo parametro, il processo reagirà nel modo specificato dal secondo parametro. Si noti che la funzione *handler* deve prevedere un parametro di tipo *int*: questo parametro, al momento dell'attivazione dell'*handler* da parte del kernel viene attualizzato con il numero del segnale effettivamente ricevuto.

Un esempio di uso della system call *signal* è mostrato in figura 7.20: il processo mediante la *signal* imposta la propria reazione al segnale **SIGUSR1**, associan-
do ad esso l'*handler* gestore; in questo modo, se successivamente il processo riceverà tale segnale, verrà eseguita la funzione gestore, che, facendo uso del parametro *signum*, provocherà la stampa dell'identificatore del segnale.

```
#include <signal.h>
void gestore(int signum)
{
    printf("Ricevuto segnale %d\n", signum);
}

main()
{
    ...
    signal(SIGUSR1, gestore);
    /* da qui in poi il processo reagira' a */
    /* SIGUSR1 eseguendo gestore */
    ...
    signal(SIGUSR1, SIG_IGN);
    /* USR1 ignorato: da qui in poi il processo */
    /* non reagira' a SIGUSR1 */
    ...
}
```

Figura 7.20 Esempio di uso di *signal*.

Benché le realizzazioni di Unix più recenti (ad esempio BSD, SystemV r.3 e seguenti) prevedano che l'azione rimanga installata anche dopo la ricezione del segnale, in qualche versione (SystemV, prime versioni), invece, l'associazione *segnale/handler* non è persistente: questo significa che, dopo l'attivazione dell'*handler* viene ripristinata automaticamente l'azione di default. In questi casi, per riagganciare il segnale all'*handler*, è necessario re-installarlo mediante una nuova *signal* all'interno del gestore:

```

void gestore(int s)
{
    signal(SIGUSR1, gestore);
    ...
}

int main()
{
    ...>>>>
    signal(SIGUSR1, gestore);
    ...
}

```

Le associazioni tra segnali e azioni vengono registrate nella *User Area* del processo (vedi paragrafo 7.4.2).

Ricordando che una `fork` copia la *User Area* del padre nella *User Area* del figlio e che padre e figlio condividono lo stesso codice, ne consegue che il figlio eredita dal padre le informazioni relative alla gestione dei segnali. Pertanto:

- ogni nuovo processo ignora gli stessi segnali ignorati dal padre,
- ogni nuovo processo gestisce con le stesse funzioni gli stessi segnali gestiti dal padre,
- i segnali a default del figlio sono gli stessi del padre.

Dal momento che padre e figlio hanno *User Area* distinte, possiamo anche osservare che eventuali signal del figlio non hanno effetto sulla gestione dei segnali del padre.

Ricordiamo inoltre che una `exec1` sostituisce codice e dati del processo che la chiama, mantenendo invece la *User Area*, tranne le informazioni legate al codice del processo, come ad esempio le funzioni di gestione dei segnali. Un processo, quindi, dopo una chiamata ad `exec1`, non può mantenere l'associazione segnale-handler: pertanto, per i segnali gestiti mediante handler dopo `exec1` viene ripristinata l'azione di default. Viceversa i segnali ignorati prima di `exec1`, rimangono ignorati; analogamente, i segnali trattati con l'azione di default vengono gestiti allo stesso modo anche dopo `exec1`.

I processi possono inviare segnali ad altri processi con la system call `kill`, secondo la seguente sintassi:

```
int kill(int pid, int sig);
```

I due parametri di `kill` rappresentano rispettivamente il destinatario (o i destinatari) del segnale (parametro `pid`) e l'intero che individua il segnale da gestire (parametro `sig`). In particolare, se `pid` è maggiore di zero esso rappresenta il *pid* dell'unico processo destinatario. Se, invece, `pid` ha valore zero, il segnale viene inviato a tutti i processi della gerarchia (*gruppo*) del processo mittente.

Nella figura 7.21 è mostrato un esempio di uso delle primitive `signal` e `kill` in una applicazione costituita da due processi: padre e figlio. In particolare, il programma, genera due processi (padre e figlio). Entrambi i processi gestiscono il segnale `SIGUSR1` mediante la funzione `gestore`: il figlio, infatti, eredita l'impostazione del padre effettuata mediante `signal` prima della chiamata a `fork`. Una volta attivi entrambi i processi, il padre invia ripetutamente il segnale `SIGUSR1` al figlio.

```
#include <stdio.h>
#include <signal.h>
int cont=0;

void gestore(int signo)
{
    printf ("Pid %d: ricevuti n.%d segnali %d\n",
            getpid(), cont++, signo);
}

int main ()
{
    int pid;
    signal(SIGUSR1, gestore);
    pid=fork();
    if (pid==0) /* figlio */
        for (;;);
    else /* padre */
        for(;;) kill(pid, SIGUSR1);
}
```

Figura 7.21 Sincronizzazione tra padre e figlio mediante segnali.

La reazione del figlio, esplicitata nella funzione gestore, consiste nella stampa di un messaggio.

Oltre alla system call `kill`, esistono altre primitive che implicitamente provocano l'invio di segnali: tra queste ricordiamo `sleep` e `alarm` che causano l'invio del segnale `SIGALRM` al processo che le invoca dopo un dato intervallo di tempo.

7.7.3 Comunicazione: pipe

I processi Unix possono comunicare mediante scambio di messaggi utilizzando le *pipe*. A livello astratto, la pipe è un canale per la comunicazione tra processi.

La comunicazione attraverso la pipe avviene in modo *indiretto*, cioè senza naming esplicito; più precisamente, quindi, essa realizza il concetto di *mailbox* (o *porta*, vedi capitolo 3). La capacità della pipe è *limitata*: essa, infatti, è in grado di gestire l'accodamento di un numero limitato di messaggi che vengono trattati in modo FIFO. Il limite è stabilito dalla dimensione della pipe, espressa dalla costante di sistema `BUFSIZ` (ad esempio 4096 bytes). La comunicazione mediante pipe è inoltre *unidirezionale*, poiché al canale che essa realizza si può accedere in lettura (o ricezione) da un solo estremo e in scrittura (o trasmissione) dall'altro estremo. Attraverso la stessa pipe, inoltre, più processi possono spedire messaggi e più processi possono ricevere messaggi: per questo la pipe è classificata come un canale di comunicazione *da-molti-a-molti*.

Il sistema operativo realizza la pipe in assoluta omogeneità con i file: come si vedrà nel seguito, infatti, la pipe viene riferita e utilizzata mediante gli strumenti tipicamente impiegati per i file. In particolare, ogni lato di accesso alla pipe è rappresentato da un *file descriptor* (vedi paragrafo 7.6.4). Grazie a questa caratteristica, quindi,

i processi potranno depositare e prelevare messaggi dalla pipe mediante le primitive di lettura e scrittura di file (come `read` e `write`, vedi paragrafo 7.6.5).

Per creare una pipe si utilizza la system call:

```
int pipe(int fd[2]);
```

in cui il parametro `fd` è il puntatore a un vettore di 2 file descriptor, che vengono inizializzati dalla system call. In caso di successo, infatti, l'intero `fd[0]` rappresenta il lato di lettura della pipe e `fd[1]` il lato di scrittura della pipe. La system call `pipe` restituisce un valore negativo, in caso di fallimento, oppure zero, se viene eseguita con successo.

Ogni lato di accesso alla pipe, quindi è rappresentato da un file descriptor. In particolare, (vedi figura 7.22) un processo mittente, per esprimere l'operazione *send*, utilizza la system call `write` sul file descriptor `fd[1]`; analogamente un destinatario effettua l'operazione *receive* mediante la system call `read` sul file descriptor `fd[0]`.

I processi che possono comunicare attraverso una stessa pipe sono tutti i discendenti del processo che ha creato la pipe (compreso il creatore stesso): infatti, poiché la pipe è riferibile dalla coppia di file descriptor (`fd[0]`, `fd[1]`) appartenenti allo spazio di indirizzamento del processo creatore, ogni processo originato dal creatore (direttamente o indirettamente) eredita una copia di (`fd[0]`, `fd[1]`) e una copia della tabella dei file aperti del processo (vedi paragrafo 7.6.4). Pertanto ogni discendente del creatore può riferire e accedere a entrambi i lati di accesso al canale.

La pipe ha capacità limitata: pertanto, come nel problema produttore/consumatore (vedi capitolo 3) è necessario sincronizzare i processi in caso di canale pieno e vuoto; in questo caso la sincronizzazione è implicitamente fornita dalle primitive di comunicazione (`read` e `write`) e prevede che, in caso di pipe vuota, un processo destinatario attenda il prossimo messaggio; analogamente, in caso di pipe piena, un processo mittente si sospende in attesa di spazio libero. In questi due casi particolari, quindi, le system call `read` e `write` sono sospensive per il processo che le invoca.

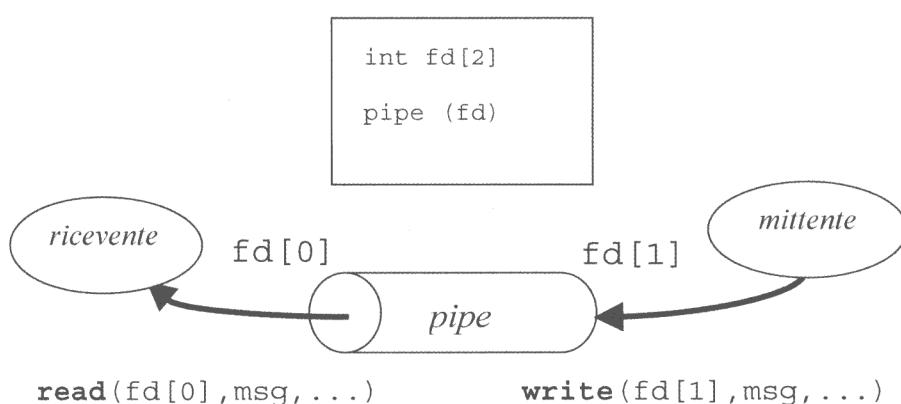


Figura 7.22 La pipe.

La principale limitazione della pipe è che essa consente la comunicazione soltanto a processi appartenenti alla stessa gerarchia; se tuttavia si ha la necessità di mettere in comunicazione processi non legati da relazioni di parentela, in molte realizzazioni del sistema Unix è possibile utilizzare strumenti alternativi alle pipe come le *fifo* (o *named pipe*) e le *socket* (vedi appendice A).

7.8 I Threads nel sistema Linux

Una delle principali differenze del sistema Linux rispetto a Unix consiste nella possibilità di utilizzare i *thread*. A differenza di Unix, infatti, Linux realizza due tipi di processo: il processo “pesante”, e il *thread*.

Il primo tipo di processo è equivalente in tutto al processo Unix (vedi paragrafo 7.4). Come è stato mostrato, esso è caratterizzato da uno spazio di indirizzamento privato e pertanto non ha alcuna possibilità di condividere variabili con altri processi. Le primitive di gestione (creazione, terminazione, ecc.) di questo tipo di processo in ambiente Linux sono le stesse già trattate nel paragrafo 7.3 (*fork*, *wait*, *exit* ecc.). Il *thread*, invece, è un processo “leggero” che ha la possibilità di condividere uno spazio di indirizzi (o parte di esso) con il padre e gli altri processi della gerarchia a cui appartiene.

Nel sistema operativo Linux il *thread* è realizzato a livello kernel ed è, quindi, l’unità di *scheduling*. Concettualmente, *thread* e processi in Linux coesistono. Tuttavia, a livello concreto tutto è *thread*: infatti il processo tradizionale Unix non è altro che un particolare *thread* che non condivide memoria con gli altri.

La system call nativa di Linux per la generazione di *thread* è la *clone*, mediante la quale il programmatore può specificare quali parti dello spazio di indirizzamento mettere in condivisione tra padre e figlio. La *fork*, quindi, viene realizzata mediante una *clone* nella quale viene specificato che nessuna parte dello spazio di indirizzamento è condivisa.

7.8.1 Gestione di thread secondo lo standard POSIX: la libreria *pthreads*

La *clone* è una system call specifica di Linux e quindi non è portabile. Volendo impiegare i *thread* in ambiente Linux e contemporaneamente mantenere garanzie di portabilità delle applicazioni, è più conveniente utilizzare strumenti standard, come quelli offerti dallo standard POSIX. In particolare la libreria *pthreads*, definita nell’ambito dello standard POSIX, offre un insieme sufficientemente ricco di primitive per la programmazione di applicazioni *multi-threaded*. Perciò nel seguito verranno descritte ed esemplificate alcune primitive fondamentali di *pthreads*, con lo scopo di presentare alcuni esempi concreti di sincronizzazione tra *thread*. Tra le varie realizzazioni della libreria *pthreads* in ambiente Linux, in questo testo abbiamo utilizzato a titolo esemplificativo la *LinuxThreads* [44] principalmente perché questa realizzazione è integrata nella libreria *glibc*, sulla quale sono basate le versioni Linux più recenti. Pertanto in queste versioni *LinuxThreads* è parte integrante del sistema operativo.

Caratteristiche dei threads POSIX/Linux Ogni programma in esecuzione nel sistema Linux è rappresentato da uno o più *threads*, ognuno dei quali è individuato

univocamente da un indentificatore (Thread IDentifier, o TID). A questo scopo, la libreria introduce il tipo `pthread_t`⁵ per riferire i thread all'interno di programmi concurrenti.

Lo standard POSIX prevede che i thread vengano creati all'interno di un processo (*task*). Questa caratteristica non è mantenuta nell'implementazione LinuxThreads, perché il sistema operativo non realizza il concetto di task: in particolare, l'esecuzione di un programma determina la creazione di un thread iniziale che esegue il codice specificato all'interno del `main`⁶. Il thread iniziale può essere il capostipite di una gerarchia che viene a formarsi attraverso la generazione di nuovi thread; questa gerarchia è un insieme di thread che condividono uno spazio di indirizzi e quindi concettualmente equivale al task.

Creazione/terminazione di thread La creazione di ogni nuovo *thread* viene effettuata mediante la chiamata della primitiva `pthread_create`, che osserva la sintassi:

```
int pthread_create(pthread_t* T, pthread_attr_t* A,
                  void* (*cod) (void* arg));
```

dove:

- `T`: è il puntatore alla variabile che raccoglierà il TID del nuovo thread;
- `A`: può essere usato per specificare eventuali attributi da associare al thread (ad esempio la priorità del thread), oppure `NULL`.
- `cod`: è il puntatore alla funzione che contiene il codice del nuovo thread;
- `arg`: è il puntatore all'eventuale vettore contenente i valori dei parametri da passare alla funzione codice;

La primitiva `pthread_create` restituisce 0 in caso di successo, altrimenti un codice di errore. Ogni nuovo *thread* esegue concorrentemente con il padre e condivide con esso le variabili globali del programma nel quale è definito.

Un thread può terminare chiamando:

```
void pthread_exit(void *retval);
```

dove `retval` è il puntatore alla variabile che contiene il valore eventualmente restituito dal thread.

Un thread padre può sospendersi in attesa della terminazione di un thread figlio con:

```
int pthread_join(pthread_t th, void **retval);
```

dove `th` è il TID del particolare thread da attendere e `retval` è il puntatore alla variabile dove verrà memorizzato il valore eventualmente restituito dal thread (con `pthread_exit`).

L'uso delle primitive di gestione dei *thread* è esemplificato nel programma riportato di seguito. Nell'esempio, il processo iniziale crea due thread figli che eseguono

⁵ Tipi e funzioni definiti nella libreria `pthread` sono utilizzabili previa inclusione dell'header file `<pthread.h>`.

⁶ Anche in questo capitolo faremo implicitamente riferimento a programmi scritti nel linguaggio C.

entrambi lo stesso codice (contenuto nella funzione `codice_T`): i tre thread (padre e due figli) eseguono concorrentemente e condividono la variabile globale `MSG`. Il padre, una volta generati i figli (identificati rispettivamente dalle variabili `th1` e `th2`), si pone in attesa di essi con `pthread_join` e poi termina.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/*variabili condivise: */
char MSG[]="Ciao!";

void *codice_T (void * arg)
{
    int i;
    for (i=0 ; i < 5 ; i++) {
        printf ("Thread %s: %s\n", (char*)arg, MSG);
        sleep (1); /* sospensione per 1 secondo */
    }
    pthread_exit (0);
}

int main ()
{
    pthread_t th1, th2;
    int retcode;
    /* creazione primo thread: */
    if (pthread_create(&th1,NULL,codice_T,"1") < 0) {
        fprintf (stderr, "Errore di creazione thread 1\n");
        exit (1);
    }
    /* creazione secondo thread: */
    if (pthread_create(&th2,NULL,codice_T,"2") < 0) {
        fprintf (stderr, "Errore di creazione thread 2\n");
        exit (1);
    }
    retcode = pthread_join (th1, NULL);
    if (retcode != 0)
        fprintf (stderr, "join fallito %d\n", retcode);
    else printf("terminato il thread 1\n");
    retcode = pthread_join (th2, NULL);
    if (retcode != 0)
        fprintf (stderr, "join fallito %d\n", retcode);
    else printf("terminato il thread 2\n");
    return 0;
}
```

7.8.2 Sincronizzazione tra thread Linux

Per la risoluzione di problemi di sincronizzazione tra thread la libreria `pthread` offre solo due strumenti di sincronizzazione: i *mutex* (o *semafori binari*) e le *variabili condizione*, che andremo ad analizzare ed esemplificare nel seguito.

Va comunque aggiunto che – nonostante il pthread non implementi il semaforo (vedi paragrafo 3.4) – la libreria LinuxThreads realizza tale strumento conformemente a quanto stabilito dallo standard POSIX 1003.1b. Per motivi di spazio, tuttavia, il semaforo di LinuxThreads non verrà trattato; il lettore interessato potrà approfondire l'argomento consultando la documentazione specifica sulla libreria [47].

Mutex Per risolvere problemi di mutua esclusione la libreria pthread definisce il *mutex*, astrazione del tutto simile al concetto di semaforo binario (vedi paragrafo 3.4): il mutex è un particolare semaforo il cui valore intero (lo *stato* del mutex) può essere 0 oppure 1. Nel primo caso si dice che il mutex è *occupato*, mentre nel secondo caso il mutex è *libero*.

Nella libreria pthreads il mutex è definito dal tipo `pthread_mutex_t` che implicitamente rappresenta:

- lo stato del mutex;
- la coda di processi nella quale verranno sospesi i processi in attesa che il mutex sia libero.

La definizione di un mutex M, quindi, viene effettuata mediante:

```
pthread_mutex_t M;
```

Una volta definito un mutex, mediante l'inizializzazione si attribuisce un valore iniziale al suo stato (*libero* o *occupato*). La primitiva da usare a questo scopo è `pthread_mutex_init`, la cui sintassi è:

```
int pthread_mutex_init(pthread_mutex_t* M,
                      const pthread_mutexattr_t* attr)
```

dove M individua il mutex da inizializzare e attr punta a una struttura che contiene gli attributi del mutex; se il valore di attr è NULL, il mutex viene inizializzato a *libero* (che è il valore di default).

Sul mutex sono possibili soltanto 2 operazioni: *locking* e *unlocking*, che sono concettualmente equivalenti rispettivamente alle operazioni *wait* e *signal* dei semafori (vedi paragrafo 3.4). In particolare, la primitiva `pthread_mutex_lock` è la realizzazione della wait per il mutex e prevede la sintassi:

```
int pthread_mutex_lock(pthread_mutex_t* M)
```

dove M rappresenta il mutex. Se M è *occupato* (cioè il suo stato è 0), il thread chiamante si sospende nella coda associata al mutex; altrimenti *occupa* M (cioè, porta lo stato di M a 0).

Analogamente, la primitiva:

```
int pthread_mutex_unlock(pthread_mutex_t* M);
```

in cui il parametro M rappresenta il mutex, è la realizzazione della *signal*. L'effetto di questa primitiva, infatti, dipende dallo stato della coda di processi associata al mutex: se vi sono processi in attesa del mutex, ne risveglia uno; altrimenti libera il mutex.

Un mutex viene tipicamente impiegato per garantire che gli accessi a una risorsa avvengano in modo mutuamente esclusivo. Un esempio di utilizzo in questo ambito è mostrato nel seguente frammento di codice: due threads accedono alternativamente

alla stessa variabile intera CONT per incrementarne il valore; la variabile mutex M viene utilizzata per garantire che le sezioni critiche di accesso a CONT vengano eseguite in modo mutuamente esclusivo.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX 100
pthread_mutex_t M; /* def.mutex condiviso tra threads */
int CONT=0; /* variabile condivisa */

void *thread1(void * arg)
{
    int k=1;
    while(k) {
        pthread_mutex_lock(&M); /*prologo sez. critica */
        CONT++;
        k=(CONT>=MAX?0:1);
        pthread_mutex_unlock(&M); /*epilogo sez. critica */
    }
    pthread_exit(0);
}

void *thread2(void * arg)
{
    int k=1;
    while(k) {
        pthread_mutex_lock(&M); /*prologo sez. critica */
        CONT++;
        k=(CONT>=MAX?0:1);
        pthread_mutex_unlock(&M); /*epilogo sez. critica */
    }
    pthread_exit (0);
}

int main ()
{
    pthread_t th1, th2;
    /* il mutex e' inizialmente libero: */
    pthread_mutex_init (&M, NULL);
    if (pthread_create(&th1, NULL, thread1, NULL) < 0) {
        fprintf (stderr, "create error for thread 1\n");
        exit (1);
    }
    if (pthread_create(&th2, NULL, thread2,NULL) < 0){
        fprintf (stderr, "create error for thread 2\n");
        exit (1);
    }
    pthread_join (th1, NULL);
    pthread_join (th2, NULL);
}
```

Variabili condizione La variabile condizione (*condition*) è uno strumento di sincronizzazione che permette ai threads di sospendere la propria esecuzione in attesa che sia soddisfatta una condizione logica. Analogamente al semaforo, ad ogni variabile *condizione* viene associata una coda nella quale i threads possono sospendersi volontariamente utilizzando un'apposita primitiva (*wait*). A differenza del semaforo (e quindi anche del mutex), però, la variabile condizione non ha uno stato: essa rappresenta soltanto una coda di thread. Tale meccanismo viene utilizzato anche in Java (vedi appendice B, paragrafo B.3.2).

Le operazioni fondamentali sulle *condition* sono la sospensione (*wait*) ed il risveglio (*signal*) di thread.

La libreria pthread realizza la variabile condizione mediante il tipo di dato `pthread_cond_t`; ad esempio la definizione:

```
pthread_cond_t C;
```

crea la variabile condizione C.

Una volta definita, una *condition* deve essere inizializzata, ad esempio mediante la primitiva:

```
int pthread_cond_init(pthread_cond_t* C,
                      pthread_attr_t* attr);
```

dove `cond` è la *condition* da inizializzare; il parametro `attr` è l'indirizzo della struttura che contiene eventuali *attributi*⁷ specificati per la *condition* (se NULL, viene inizializzata a default).

Un thread può sospendersi su una *condition*, se la condizione logica associata ad essa non è verificata. Ad esempio, nel problema dei produttori e consumatori (vedi capitolo 3) è necessario che un produttore si sospenda se il buffer dei messaggi è pieno.

Usando i pthread questa situazione si può esprimere associando alla condizione di buffer pieno una condition, come nell'esempio che segue:

```
/*variabili globali:*/
pthread_cond_t C;
int bufferpieno=0;
...
/* codice produttore:*/
...
if (bufferpieno) <sospensione sulla condition C>;
<inserimento messaggio nel buffer>;
```

Va osservato che la verifica della condizione è una sezione critica: nell'esempio precedente, infatti, la variabile booleana `bufferpieno` è condivisa tra tutti i produttori e i consumatori e vi si accede quindi in modo mutuamente esclusivo. Per questo motivo ad ogni variabile condizione viene associato un mutex il cui ruolo è quello di garantire la mutua esclusione nell'accesso alla sezione critica per la verifica della condizione. L'esempio precedente deve essere quindi modificato introducendo un mutex `M` e aggiungendo un prologo e un epilogo rispettivamente prima e dopo la

⁷ Linux non implementa gli attributi delle condition, pertanto il valore del secondo parametro non è rilevante.

verifica (ed eventuale sospensione del thread) della condizione per garantire la mutua esclusione nell'accesso a buffer pieno:

```
/* variabili globali: */
pthread_cond_t C;
pthread_mutex_t M; /* per mutua esclusione su condizione */
int bufferpieno=0;
...
/* codice produttore: */
...
pthread_mutex_lock(&M);
if (bufferpieno) <sospensione sulla cond. C>;
<inserimento messaggio e aggiornamento del buffer>;
pthread_mutex_unlock(&M);
```

Tenendo conto di queste considerazioni, la libreria pthread prevede implicitamente che congiuntamente all'uso di una *condition* venga sempre introdotto un *mutex*. Questo spiega perché la primitiva di sospensione richiede come parametro aggiuntivo (oltre alla *condition* su cui deve operare) una variabile di tipo mutex. In particolare, la *primitiva di sospensione* è realizzata dalla funzione:

```
int pthread_cond_wait(pthread_cond_t * C, pthread_mutex_t * M);
```

dove C è la variabile condizione e M è il mutex associato ad essa. La chiamata di questa primitiva da parte di un thread T provoca due effetti: T viene sospeso nella coda associata a C, e il mutex M viene liberato. Al successivo risveglio di T il thread rioccupa il mutex M automaticamente.

Un thread che si sospende con `pthread_cond_wait` libera il mutex M associato alla condition, per poi rioccuparlo successivamente, quando verrà risvegliato.

Il risveglio di un thread sospeso su una variabile condizione può essere ottenuto mediante la funzione:

```
int pthread_cond_signal(pthread_cond_t * C)
```

dove C rappresenta la *condition*.

Gli effetti di una chiamata a `pthread_cond_signal` sono i seguenti:

- se esistono *thread* sospesi nella coda associata a C, ne viene risvegliato il primo.
- se non vi sono *thread* sospesi sulla condizione, la signal non ha alcun effetto.

La *condition* permette di realizzare politiche di sincronizzazione più sofisticate rispetto al *mutex*, come vedremo nel paragrafo successivo.

Per esemplificare l'uso della *condition*, si consideri il caso di una risorsa che può essere usata contemporaneamente da, al massimo, MAX thread⁸. Realizziamo una politica di controllo degli accessi mediante variabili condizione. A questo scopo, introduciamo la *condition* PIENO, sulla quale verranno sospesi i thread che vogliono

⁸ Un problema di questo tipo va affrontato, per esempio, nei sistemi operativi multiprogrammati, quando è fissato un limite massimo per il numero di processi caricati in memoria. Al raggiungimento di tale limite, il caricamento di un nuovo processo deve essere ritardato fino a quando uno dei processi caricati non termina.

accedere alla risorsa nel caso di capacità esaurita. Sia M il mutex associato alla condizione PIENO. Introduciamo inoltre la variabile intera non negativa N_in per rappresentare lo stato della risorsa, cioè il numero di thread che stanno usando la risorsa:

```
#define MAX 100
/*variabili globali: */
int N_in=0; /* numero thread che usano la risorsa */
pthread_cond_t PIENO;
pthread_mutex_t M; /* Mutex associato alla cond. PIENO */

void codice_thread()
{
    /* Fase di entrata: */
    pthread_mutex_lock(&M);
    /* controlla la condizione di accesso: */
    if (N_in==MAX) pthread_cond_wait(&PIENO, &M);
    /* aggiorna lo stato della risorsa */
    N_in++;
    pthread_mutex_unlock(&M);

    <uso della risorsa>

    /*Fase di Uscita: */
    pthread_mutex_lock(&M);
    /* aggiorna lo stato della risorsa */
    N_in--;
    pthread_cond_signal(&PIENO);
    pthread_mutex_unlock(&M);
}
```

Figura 7.23 Struttura di un thread che utilizza una risorsa a capacità limitata.

```
#define MAX 100
/*variabili globali: */
int N_in=0; /* numero thread che usano la risorsa */
pthread_cond_t PIENO;
pthread_mutex_t M; /* Mutex associato alla cond. PIENO */
```

Nella figura 7.23 viene mostrata la struttura del codice di un generico thread che vuole utilizzare la risorsa. Esso deve prevedere una fase di entrata, nella quale viene controllato lo stato della risorsa: in caso di risorsa “piena” il thread si sospende sulla condition PIENO. Se, invece, la risorsa può essere usata dal thread, il numero dei thread N_in viene incrementato. La fase di entrata è una sezione critica, perché prevede l’accesso e l’aggiornamento della variabile N_in che rappresenta lo stato della risorsa; pertanto, per garantire la mutua esclusione, essa viene preceduta da una wait sul mutex M (pthread_mutex_lock) e terminata da una signal (pthread_mutex_unlock).

Terminato l’uso della risorsa, il thread deve eseguire una fase di uscita, rappresentata ancora da una sezione critica nella quale viene decrementato il valore di

`N_in` ed eventualmente risvegliato (mediante `pthread_cond_signal`) un processo sospeso sulla condition `PIENO`. Come per la fase di ingresso, anche in questo caso la mutua esclusione sulla fase di uscita è garantita da una `wait` iniziale sul mutex `M` (`pthread_mutex_lock`) e da una `signal` (`pthread_mutex_unlock`) finale.

7.8.3 Un esempio di sincronizzazione tra thread

Si vuole risolvere il classico problema del produttore e consumatore, introdotto nel paragrafo 3.3. Nel caso più generale, più produttori e consumatori possono utilizzare un buffer in grado di contenere, al massimo, N messaggi. Richiamando quanto detto nel capitolo 3, i vincoli nell'accesso al buffer sono due: il produttore non può inserire un messaggio nel buffer pieno; il consumatore non può prelevare un messaggio dal buffer vuoto.

Supponendo, ad esempio, che i messaggi siano dei valori interi, realizziamo il buffer come un vettore di interi che viene gestito in modo circolare. Per consentire una gestione corretta del buffer vengono inoltre associate al buffer le seguenti informazioni:

- il numero degli messaggi contenuti (`cont`) ;
- il puntatore alla prima posizione libera (`writepos`) ;
- il puntatore al primo elemento occupato (`readpos`) .

Il buffer è una risorsa cui accedere in modo mutuamente esclusivo: perciò è necessario associare ad esso un mutex `M` per il controllo della mutua esclusione nell'accesso al buffer.

Oltre al vincolo di mutua esclusione, i thread produttori e consumatori necessitano di sincronizzazione in caso di buffer pieno e di buffer vuoto. Pertanto si associano al buffer due condition: la condition `PIENO` per la sospensione dei produttori se il buffer è pieno e la condition `VUOTO` per la sospensione dei consumatori se il buffer è vuoto.

Per realizzare la “risorsa buffer”, il vettore di messaggi e tutte le informazioni ad esso associate vengono aggregate all'interno di una struttura del tipo `prodcons`:

```
typedef struct
{
    int buffer[BUFFER_SIZE];
    pthread_mutex_t M;
    int readpos, writepos;
    int cont;
    pthread_cond_t PIENO;
    pthread_cond_t VUOTO;
} prodcons;
```

Per gestire risorse del tipo `prodcons`, definiamo inoltre tre funzioni: *inizializzazione* della risorsa, *inserimento* di un messaggio e *estrazione* di un messaggio. L'inizializzazione è l'operazione mediante la quale si attribuiscono dei valori iniziali alle informazioni associate al buffer. La funzione seguente realizza l'operazione di inizializzazione sulla risorsa buffer puntata dal parametro `b`:

```
/* Inizializza il buffer */
void init (prodcons *b)
{
    pthread_mutex_init (&b->M, NULL);
    pthread_cond_init (&b->PIENO, NULL);
    pthread_cond_init (&b->VUOTO, NULL);
    b->cont=0;
    b->readpos = 0;
    b->writepos = 0;
}
```

Ogni produttore può effettuare l'inserimento di un nuovo messaggio (MSG) nel buffer (b) mediante l'operazione di inserimento, che può essere realizzata come segue:

```
void inserisci (prodcons *b, int MSG)
{
    pthread_mutex_lock (&b->M);
    while (b->cont==BUFFER_SIZE) /* il buffer e' pieno? */
        pthread_cond_wait (&b->PIENO, &b->M);
    /* scrivi MSG e aggiorna lo stato del buffer */
    b->buffer[b->writepos] = MSG;
    b->cont++;
    b->writepos++;
    /* la gestione e' circolare */
    if (b->writepos >= BUFFER_SIZE)
        b->writepos = 0;
    /* risveglia un eventuale thread consumatore sospeso */
    pthread_cond_signal (&b->VUOTO);
    pthread_mutex_unlock (&b->M);
}
```

Ogni consumatore può effettuare l'estrazione di un messaggio MSG dal buffer b mediante l'operazione di estrazione, che può essere realizzata come segue:

```
int estrai (prodcons *b)
{
    int MSG;
    pthread_mutex_lock (&b->M);
    while (b->cont==0) /* il buffer e' vuoto? */
        pthread_cond_wait (&b->VUOTO, &b->M);
    /* Leggi il messaggio e aggiorna lo stato del buffer*/
    MSG = b->buffer[b->readpos];
    b->cont--;
    b->readpos++;
    if (b->readpos >= BUFFER_SIZE)
        b->readpos = 0;
    /* Risveglia un eventuale thread produttore*/
    pthread_cond_signal (&b->PIENO);
    pthread_mutex_unlock (&b->M);
    return MSG;
}
```

Nel seguente frammento di codice viene proposto l'esempio di un programma che genera 2 thread: un produttore ed un consumatore che accedono al buffer B mediante le operazioni `inserisci` ed `estrai`. In particolare il produttore inserisce sequenzialmente nel buffer B `max` interi; il consumatore estraе iterativamente un intero alla volta per stamparlo sullo standard output fino a che non viene estratto il valore -1.

```
#include <pthreads.h>
#define OVER (-1)
#define max 2000
#define BUFFER_SIZE 100

typedef struct {
    int buffer[BUFFER_SIZE];
    pthread_mutex_t M;
    int readpos, writepos, cont;
    pthread_cond_t PIENO, VUOTO;
} prodcons;

void init (prodcons *b);
void inserisci (prodcons *b, int MSG);
int estrai (prodcons *b);

prodcons B;

void *produttore (void *arg) {
    int n;
    for (n = 0; n < max; n++) {
        printf ("Thread produttore %d --->\n", n);
        inserisci (&B, n);
    }
    inserisci (&B, OVER);
    return NULL;
}

void *consumatore (void *arg)
{
    int d;
    while (1) {
        d = estrai (&B);
        if (d == OVER) break;
        printf("Thread consumatore: --> %d\n", d);
    }
    return NULL;
}

int main ()
{
    pthread_t th_a, th_b;
    init (&B);
    /* Creazione threads: */
}
```

```

pthread_create(&th_a, NULL, produttore, 0);
pthread_create(&th_b, NULL, consumatore, 0);
/* Attesa teminazione threads creati: */
pthread_join(th_a, NULL);
pthread_join(th_b, NULL);
return 0;
}

```

7.9 Sommario

I concetti generali trattati nella prima parte del testo sono stati esemplificati in questo capitolo prendendo in esame la famiglia di sistemi operativi Unix. A questa famiglia, diventata attualmente uno standard di fatto per workstation e mainframe, appartiene anche il sistema operativo Linux, la più popolare realizzazione di Unix per personal computer: Linux è un sistema di pubblico dominio alla cui realizzazione concorrono da più di un decennio centinaia di appassionati sviluppatori in tutto il mondo.

Dopo alcuni cenni storici e introduttivi, è stata descritta l'organizzazione del sistema, ed il ruolo delle principali componenti, sottolineando che quasi tutte le realizzazioni adottano un'organizzazione basata su kernel monolitico. Sono state successivamente analizzate le caratteristiche del processo Unix, che deriva dal modello ad ambiente locale, sia da un punto di vista realizzativo che dal punto di vista del programmatore di sistema, mostrando ed esemplificando le principali system call per la gestione di processi.

I processi tradizionali Unix non possono condividere variabili, e pertanto l'interazione tra di essi può avvenire soltanto mediante scambio di messaggi o invio di segnali. A questo proposito sono stati descritti i principali strumenti di comunicazione e sincronizzazione disponibili nel sistema, come *pipe* e *segnali*.

Alcune realizzazioni di Unix realizzano il concetto di *thread*: in particolare, nel paragrafo 7.8 sono state descritte le proprietà dei thread in ambiente Linux, sottolineandone sia gli aspetti realizzativi che gli aspetti di utilizzo. A questo proposito, è stata presentata la libreria *pthreads*, definita nell'ambito dello standard POSIX e disponibile anche in ambiente Linux. Come è stato evidenziato, questa libreria fornisce un insieme di strumenti sufficientemente ricco per lo sviluppo di applicazioni multi-threaded; in particolare, i problemi di sincronizzazione tra thread possono essere affrontati con *mutex* e *variabili condizione*, come mostrato dagli esempi.

Il capitolo ha inoltre presentato le caratteristiche di Unix relative ad altri importanti aspetti come la gestione dei file e la gestione della memoria centrale.

Esemplificando in un caso reale alcuni tra i concetti generali presentati nei primi capitoli, la trattazione ha pertanto sottolineato gli aspetti rilevanti a questo scopo e ha invece soltanto accennato o omesso altri aspetti non essenziali (come ad esempio lo *shell*), che potranno essere eventualmente approfonditi dal lettore consultando le fonti bibliografiche.

7.10 Note bibliografiche

Panoramiche su Unix simili a quella presentata in questo libro, ma meno orientate alla programmazione di sistema sono presenti nei classici testi [96] e [80]. Un'analisi più completa e approfondita delle caratteristiche di Unix, focalizzata soprattutto sugli aspetti di programmazione di sistema è presentata invece in [87]. Gli aspetti inerenti all'organizzazione interna del kernel di Unix e di Linux sono trattati in modo esteso rispettivamente in [58] e [12]. Riguardo a Linux, il software di installazione, tutta la documentazione ed il codice sorgente possono essere scaricati dal sito ufficiale [60]. Le modalità di uso di Unix mediante il linguaggio di comandi previsto dallo shell, non affrontate in questo capitolo, possono essere approfondite consultando il testo [11].

Il Sistema Operativo Windows

In questo capitolo daremo un breve panoramica sul sistema operativo Microsoft Windows™. Si tratta di uno dei sistemi operativi più diffusi al mondo: sicuramente il più diffuso sistema operativo sui Personal Computer per l'utenza casalinga e professionale, possiede un'ottima diffusione anche nel mercato dei server. Esistono diverse versioni di tale sistema operativo, con caratteristiche anche molto differenti. In questo capitolo ci limiteremo a descrivere la struttura e le funzionalità fornite dai sistemi Windows NT, Windows 2000 e Windows XP.

Le informazioni contenute in questo capitolo sono state ricavate dalla documentazione tecnica fornita dalla Microsoft e dall'esperienza diretta di programmazione. Alcune informazioni sulla struttura interna sono state ricavate da terze parti e non sono verificabili direttamente in quanto, come noto, Windows è un prodotto commerciale e non è possibile divulgare la sua struttura interna, né accedere al codice sorgente del sistema operativo.

8.1 Struttura generale

Data la grande diversità tra i vari sistemi operativi che vanno sotto il nome di Windows, è bene fare un breve inciso sull'evoluzione storica di tali sistemi e sulle relazioni che li legano l'uno all'altro.

8.1.1 Evoluzione storica

Il sistema operativo Windows fu proposto all'inizio degli anni '80 come evoluzione del sistema operativo MS-DOS. Quest'ultimo era un sistema operativo mono-programmato e mono-utente che forniva delle funzionalità di input/output di base verso l'hardware. MS-DOS è stato un sistema molto diffuso sui Personal Computer e con migliaia di applicazioni disponibili. Microsoft aveva dunque l'esigenza di fornire un sistema più avanzato senza perdere la grande base di utenza di MS-DOS. Per questo

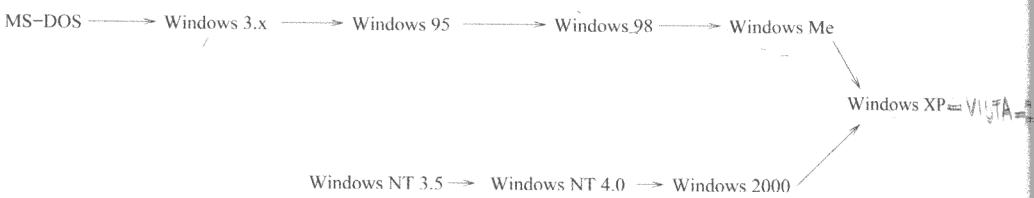


Figura 8.1 Evoluzione storica dei sistemi Microsoft Windows.

motivo, uno dei grandi temi fondamentali dello sviluppo dei sistemi operativi Microsoft è stata l'esigenza di mantenere la compatibilità con i sistemi precedenti. Le applicazioni che funzionavano su MS-DOS dovevano poter essere utilizzate anche sui sistemi successivi.

Il primo sistema Windows ad essere diffuso largamente sul mercato fu la versione 3.0. Esso supportava in maniera molto limitata la multi-programmazione; non esisteva ancora il concetto di protezione, né di multi-utenza; lo scheduler era non-preemptive e molto rozzo. Esso era pensato più come una interfaccia grafica che estendeva le funzionalità di MS-DOS piuttosto che come sistema operativo a se stante.

Parallelamente alla commercializzazione e allo sviluppo di Windows 3.0 e successive versioni, la Microsoft cominciò a sviluppare un nucleo di sistema operativo multi-programmato che avesse tutte le caratteristiche dei sistemi operativi allora in auge, come Unix e VMS. Il risultato di tale sforzo fu il rilascio del sistema Windows NT, rivolto principalmente al mercato professionale e al mercato dei server. Windows NT supporta la separazione degli spazi di indirizzamento, la multi-programmazione "vera", la gestione della memoria, la multi-utenza, la protezione delle risorse, il supporto al networking.

Per un certo periodo, la Microsoft continuò a supportare due diverse linee di prodotti: sistemi operativi per i personal computer e per l'utenza casalinga, quali Windows 95, Windows 98 e Windows Me; e sistemi operativi per l'utenza professionale e per il mercato dei server, quali Windows NT 3.5, NT 4.0 e Windows 2000. Recentemente, la Microsoft ha inteso semplificare il supporto e lo sviluppo dei propri sistemi operativi unificando le due linee di prodotti in un unico sistema operativo, Windows XP. Tale sistema operativo viene commercializzato in due versioni, Home e Professional, con alcune differenze nella tipologia di servizi supportati. Si tratta comunque dello stesso sistema operativo.

8.1.2 Sottosistemi e moduli del nucleo

Il sistema operativo Windows è strutturato in maniera modulare e stratificata. Come mostrato schematicamente in figura 8.2, possiamo distinguere almeno tre diversi strati di software all'interno del sistema operativo:

1. lo strato di astrazione della macchina fisica (HAL, *Hardware Abstraction Layer*);
2. il nucleo (*Kernel*);
3. l'esecutivo (*Executive*).

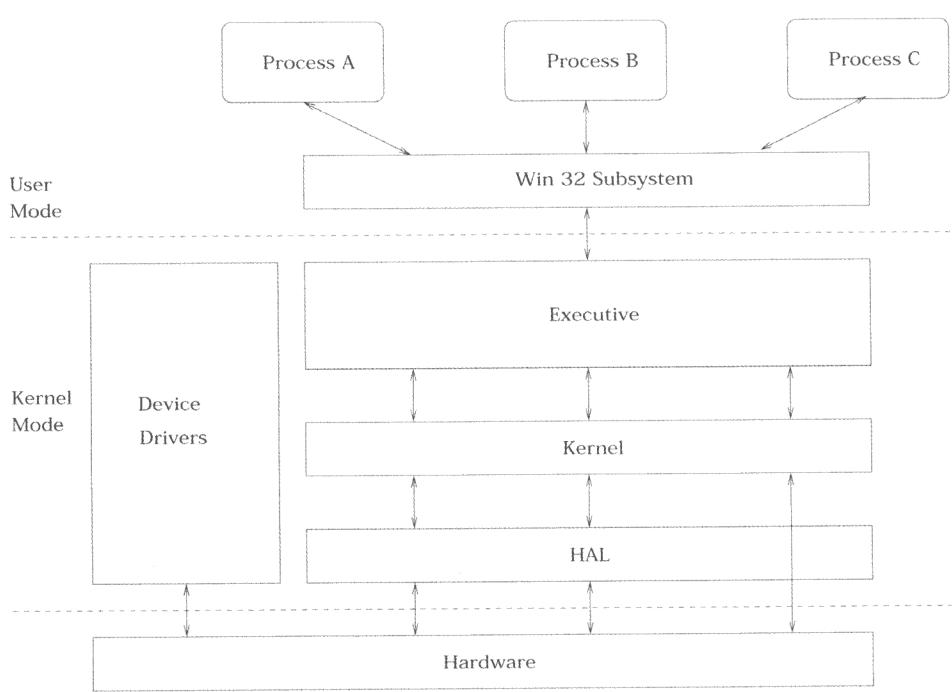


Figura 8.2 Struttura del sistema operativo Windows.

Tutti questi moduli software vengono eseguiti in modo *kernel*. Il sistema Windows inoltre fornisce alcuni componenti che lavorano in modalità utente, come ad esempio i *sottosistemi d'ambiente*. Si tratta di moduli e librerie software che traducono le richieste dei processi utente in opportune richieste di servizi all'executive. In questo modo i processi utente possono accedere alle funzionalità del sistema operativo.

Lo **HAL** è uno strato di software che astrae alcuni meccanismi di base dalla particolare macchina fisica. Ad esempio, il cambio di contesto e la gestione delle interruzioni a basso livello, sono servizi forniti dallo HAL. In questo modo, il nucleo è indipendente dalla macchina fisica e il sistema operativo può essere portato con facilità da una macchina fisica ad un'altra semplicemente cambiando lo HAL. In realtà, per ragioni di performance lo HAL può essere scavalcato. Quindi, alcuni componenti del sistema operativo, quali ad esempio il sottosistema grafico e alcuni driver di I/O, possono accedere direttamente all'hardware senza coinvolgere lo HAL.

Il **nucleo** è il cuore del sistema operativo e fornisce quattro servizi fondamentali: lo scheduling dei thread, la gestione delle interruzioni, la sincronizzazione a basso livello e la gestione del consumo energetico (*power management*). Questo componente del sistema, proprio per la sua importanza, non viene mai paginato (cioè le sue pagine non vengono mai trasferite su memoria di massa) e gira sempre in modo tale da non subire mai revoca (*non preemptive*). Questo obiettivo viene raggiunto mediante

una gestione prioritaria delle interruzioni. In particolare, lo HAL fornisce al nucleo 32 livelli di interrupt standard nei quali vengono mappati tutti gli interrupt della particolare macchina hardware su cui il sistema viene realizzato. In questo modo viene garantita la portabilità del sistema rendendo il nucleo e l'executive indipendenti dalla configurazione degli interrupt della macchina fisica.

Inoltre, per servire le interruzioni, il nucleo di Windows usa il seguente meccanismo: per ogni processore viene riservata una variabile (*IRQL - Interrupt Request Level*) contenente un valore intero compreso fra 0 e 31 (livello di priorità). Tutti gli interrupt che hanno un livello di priorità minore o uguale a IRQL vengono mascherati (mantenuti pendenti fino a quando il livello IRQL non viene opportunamente abbassato). In questo modo, quando il nucleo sta girando può sollevare il livello IRQL ad un valore superiore a quello relativo all'interrupt software utilizzato per attivare un thread (*dispatch interrupt*). Soltanto quando il nucleo termina, abbassando il livello IRQL il *dispatch interrupt* può essere servito e quindi un nuovo thread attivato. Tale meccanismo può essere fornito direttamente dal processore oppure simulato dallo HAL.

In tabella 8.1 vengono riportati i 32 livelli di interruzioni previsti da Windows.

Livelli di interruzione	Tipi di interruzione
31	Bus error
30	Power down
29	Inter-processor interrupt
28	Timer
12-27	HW interrupts
4-11	Deferred procedure call and dispatcher
3	Debugger
0-2	SW interrupts

Tabella 8.1 Livelli di interruzione in Windows.

Il nucleo utilizza una tabella (*Interrupt Dispatch Table*) per collegare ogni livello di interruzione alla corrispondente routine di servizio.

Dei 32 livelli, 8 sono utilizzati direttamente dal nucleo (i livelli 4-11, vedi tabella 8.1). Fra questi, oltre al *dispatch interrupt* utilizzato per attivare un thread, è da notare il meccanismo delle chiamate di procedura differite (DPC, *Deferred Procedure Call*). Ad esempio, quando un handler di interruzione relativo a un driver di I/O deve eseguire una certa procedura potenzialmente lunga e la cui esecuzione non è urgente, invece di eseguire tale procedura può generare un'interruzione di tipo DPC, la cui routine di servizio sia costituita dalla precedente procedura. In questo modo, la procedura non viene eseguita immediatamente, ma soltanto quando il livello di interruzione IRQL viene abbassato dal nucleo ad un livello tale da permetterne l'esecuzione. In pratica, è lo stesso meccanismo visto precedentemente ed usato per l'attivazione dei thread. Quando lo scheduler decide di dover mandare in esecuzione un nuovo thread, genera una interruzione di tipo *dispatch interrupt*, che ha un livello di priorità piuttosto basso. Come è stato mostrato, tale interruzione non viene servita

fino a quando il livello IRQL non viene sufficientemente abbassato, (cioè poco prima di rientrare in modalità utente). A quel punto, l'attivazione del nuovo thread viene effettivamente eseguita. Lo schedulatore verrà descritto con maggiori dettagli più avanti nel capitolo.

Lo strato **Executive** fornisce un insieme di servizi per la modalità utente, quali la gestione degli oggetti di sistema, della memoria virtuale, del file system, dell'I/O, la creazione dei processi e dei thread, il controllo della sicurezza e la comunicazione inter-processo tramite le chiamate di procedura locali (LPC, *Local Procedure Call*). Daremo una panoramica su alcune di tali funzionalità nel resto del capitolo.

Come già visto, i processi utente possono accedere alle funzionalità del sistema operativo tramite i **sottosistemi d'ambiente**. Tali moduli girano in modalità utente e interfacciano i processi utente con l'executive.

Il più importante di tali sottosistemi, sempre presente su ogni sistema Windows, è il sottosistema Win32, che fornisce una interfaccia (API) ai processi utenti mappata uno a uno sui servizi dell'executive.

Windows fornisce anche altri sottosistemi per supportare la compatibilità con altri sistemi operativi. Ad esempio, è possibile eseguire applicazioni MS-DOS grazie al sottosistema omonimo, pur con qualche importante restrizione. Il sottosistema MS-DOS fornisce una macchina virtuale per cui programmi eseguibili MS-DOS possono eseguire come un processo utente di Windows. Purtroppo, l'astrazione di macchina virtuale MS-DOS non è perfetta e alcune applicazioni MS-DOS non funzionano correttamente in ambiente Windows. Infatti, molte applicazioni MS-DOS furono scritte pensando di avere a disposizione l'intera macchina fisica e quindi accedono direttamente all'hardware del sistema con dei trucchi non facilmente simulabili in ambiente Windows.

Altri sottosistemi di rilievo sono l'ambiente Win16, che permette l'esecuzione di applicazioni scritte per Windows 3.x e Windows 95; l'ambiente POSIX, che fornisce ai processi utente un'interfaccia compatibile con lo standard POSIX; l'ambiente OS/2, che fornisce un'interfaccia compatibile con il sistema operativo OS/2.

8.1.3 Struttura a microkernel

Notiamo che la struttura interna di Windows presenta una marcata somiglianza con la struttura a microkernel (vedi capitolo 1, paragrafo 1.4.2). In effetti, inizialmente gli sviluppatori di Windows seguirono il modello a microkernel, con un nucleo molto ridotto che forniva le funzionalità elementari a moduli di più alto livello operanti a livello utente. Successivamente, per motivi di efficienza, tutti i moduli di sistema operativo furono portati all'interno del nucleo. Inoltre, sia per ragioni di efficienza che per ragioni commerciali, anche il sottosistema di gestione grafica delle finestre viene eseguito in spazio kernel. Inoltre, sempre per ragioni di efficienza, alcuni moduli "rompono" la struttura del microkernel accedendo direttamente alla macchina fisica. È il caso dei sottosistemi grafici e di gestione del suono. Infatti, alcune delle applicazioni più diffuse e richieste sui Personal Computer sono i videogiochi che hanno necessità di alte performance grafiche e sonore. Per questo motivo, Microsoft ha sviluppato i sottosistemi DirectX e DirectSound che accedono direttamente alla macchina fisica.

8.1.4 Kernel Objects

Prima di proseguire con la descrizione delle funzionalità di Windows, è opportuno spiegare in dettaglio il concetto di oggetto di nucleo (*kernel object*) all'interno del sistema Windows.

Quasi tutte le risorse che Windows mette a disposizione del programmatore sono implementate internamente secondo il paradigma di programmazione orientata agli oggetti. Sono oggetti i processi, i thread, i semafori, i file, le directory e così via. Gli oggetti di nucleo risiedono nella memoria del nucleo. Il programmatore non può in nessun caso accedere direttamente a tali oggetti, ma può manipolarli indirettamente e in maniera sicura tramite le funzioni di interfaccia dell'executive di Windows. Tipicamente, i nomi delle funzioni per creare un oggetto di nucleo cominciano per `Create`, mentre le funzioni per creare un riferimento a un oggetto esistente cominciano per `Open`. Ad esempio, nel caso in cui vogliamo usare un semaforo di sincronizzazione tra due processi, il primo dovrà creare il semaforo usando la chiamata di sistema `CreateSemaphore(...)`, mentre il secondo potrà aprire lo stesso semaforo chiamando la `OpenSemaphore(...)`.

Tutti i tipi di oggetto discendono da un tipo base da cui ereditano le caratteristiche. Tali caratteristiche di base sono:

- un contatore di riferimenti;
- un sistema di identificazione univoco;
- gli attributi di protezione.

Il contatore di riferimenti serve a tener traccia di quanti processi stanno effettivamente accedendo all'oggetto in questione. Quando un processo crea l'oggetto, il contatore viene inizializzato a 1. Ogni volta che un nuovo processo dichiara di usare l'oggetto (tramite una chiamata a una funzione `Open`), il contatore viene incrementato di 1. Ogni volta che un processo dichiara di non voler più usare l'oggetto (tramite una chiamata alla funzione `CloseHandle()`) il contatore viene decrementato di 1. Soltanto quando il contatore torna al valore 0, il sistema operativo può distruggere l'oggetto, liberando la memoria.

Un processo può riferire un oggetto di kernel attraverso un riferimento (*handle*). Ogni volta che un processo crea un oggetto, o apre un oggetto esistente, ottiene un riferimento di tipo `HANDLE` all'oggetto. Ogni processo possiede una tabella privata di handle a oggetti aperti. Dal punto di vista logico, possiamo pensare a un handle come l'indice dell'oggetto all'interno della tabella. Il valore di un handle è dunque relativo ad ogni processo: se due processi aprono lo stesso oggetto, possono ottenere due handle diversi. Il valore numerico di un handle di un processo non ha alcun significato per un altro processo. Facendo un parallelo fra Windows e Unix, possiamo dire che un handle è molto simile a un descrittore di file di Unix (vedi capitolo 7) pur essendo utilizzato in modo più generale.

8.1.5 Sicurezza e controllo degli accessi

Come ogni sistema operativo multi-utente, anche Windows, nelle versioni Windows NT, Windows 2000 e Windows XP, fornisce al possibilità di proteggere l'accesso da parte dei processi alle risorse del sistema. Una panoramica generale sui meccani-

smi di protezione è stata già presentata nel paragrafo 6.33 con riferimento alle risorse di tipo file o directory. A differenza di altri sistemi, Windows fornisce un controllo degli accessi ad un livello di granularità molto fine, non solo per file e directories, ma per qualunque oggetto di nucleo. Il meccanismo è estremamente flessibile, ma anche molto complicato: in questo paragrafo ci limiteremo ad accennare al suo funzionamento generale e rimandiamo a [100] per maggiori dettagli sull'argomento.

Tutti gli oggetti di nucleo possiedono degli attributi di sicurezza che ne regolano l'accesso e le operazioni possibili da parte dei processi utente. Quasi tutte le funzioni usate per creare oggetti di nucleo richiedono un puntatore a una struttura `SECURITY_ATTRIBUTES`, descritta qui di seguito:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;
    LPVOID lpSecurityDescriptor;
    BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Tale struttura contiene tre campi. Il campo `nLength` contiene la dimensione in byte della struttura `SECURITY_ATTRIBUTES`. Il campo `bInheritHandle` serve per permettere la condivisione di un oggetto di kernel tra due processi, come spiegato nella sezione successiva. Infine, il campo `lpSecurityDescriptor` è, a sua volta, un puntatore a una lista che contiene i permessi di accesso, in lettura e scrittura, dei vari processi e utenti nel sistema. Tale lista implementa il meccanismo di protezione secondo lo schema delle *liste di controllo degli accessi* (*Access Control List* già viste nel sesto capitolo, paragrafo 6.3.3). È possibile assegnare al campo `lpSecurityDescriptor` il valore `NULL`: in questo modo vengono impostati i valori di protezione di default. Per maggiori dettagli relativi al descrittore di sicurezza rimandiamo a un manuale di programmazione Win32.

Condivisione di oggetti fra più processi Per permettere a due o più processi di comunicare e sincronizzarsi fra loro, è possibile condividere oggetti di nucleo fra più processi. Ci sono tre diverse possibilità per fare in modo che due processi possano accedere allo stesso oggetto di nucleo: in base al nome simbolico, tramite ereditarietà, e tramite la chiamata di sistema `DuplicateHandle()`.

Quasi tutti gli oggetti possiedono un nome simbolico (una stringa) che li identifica univocamente all'interno del sistema. Tale stringa può essere usata per fare in modo che due processi possano condividere uno stesso oggetto di nucleo. Ad esempio, qui di seguito riportiamo l'intestazione in linguaggio C della chiamata di sistema per creare un semaforo:

```
HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);
```

Il primo parametro (`psa`) di tipo `PSECURITY_ATTRIBUTES` serve per specificare gli attributi di sicurezza, come spiegato precedentemente. Il secondo (`lInitialCount`) denota il valore iniziale del semaforo. Il terzo parametro (`lMaximum-`

`Count`) rappresenta il massimo valore che il semaforo può assumere. L'ultimo parametro è un puntatore a una stringa di caratteri (`PCTSTR, pointer to character string`), che deve essere univoca all'interno del sistema e serve per identificare il semaforo. Una volta creato l'oggetto semaforo, un altro processo può accedervi utilizzando la seguente chiamata di sistema:

```
HANDLE OpenSemaphore(
    ...
    PCTSTR pszName);
```

e passando, fra i vari parametri, la stessa stringa con cui il semaforo è stato creato.

Un altro metodo per condividere oggetti di nucleo è legato alla relazione *padre-figlio* fra due processi. Un oggetto creato da un processo PA può essere *ereditato* dai tutti i processi creati da PA. Per far questo è necessario porre a TRUE il campo `bInheritHandle` della struttura `SECURITY_ATTRIBUTES`. Ad esempio, un processo può creare un semaforo come segue:

```
SECURITY_ATTRIBUTES s;
s.nLength = sizeof(s);
s.lpSecurityDescriptor = NULL;
s.bInheritedHandle = TRUE;

HANDLE hSem = CreateSemaphore(&s, 0, 10, NULL);
```

Il semaforo così creato è “ereditabile” dai processi figli. Infatti, la struttura `s` che viene passata come primo parametro alla chiamata `CreateSemaphore`, ha il campo `bInheritedHandle` al valore TRUE. Inoltre, l'oggetto è “anonimo” in quanto non viene specificato un nome (valore NULL passato come ultimo parametro). Il semaforo ha valore iniziale 0 e può assumere 10 come massimo valore.

Quando viene creato un nuovo processo, è possibile specificare fra i suoi parametri che gli oggetti ereditabili vengano effettivamente ereditati. Riportiamo qui di seguito l'intestazione della primitiva `CreateProcess()`:

```
BOOL CreateProcess (
    PCTSTR pszName,
    PTSTR pszCommandLine,
    ...
    BOOL bInheritedHandles,
    ...);
```

Tra i tanti parametri, oltre al nome del processo (`pszName`), abbiamo messo in evidenza il parametro `bInheritedHandles`, il quale, se posto a TRUE, specifica che gli handle ereditabili vanno effettivamente ereditati. La `CreateProcess`, tra le altre cose, inizializza la tabella degli handle per il nuovo processo copiando esattamente nello stesso posto gli handle ereditabili. Inoltre, il contatore dei riferimenti di ciascuno degli oggetti ereditabili viene incrementato di 1. Naturalmente, il nuovo processo (il cui nome viene passato tramite il parametro `pszName`) deve essere a conoscenza che alcuni handle sono ereditati. Per conoscere l'esatto valore di tali handle, è possibile, ad esempio, passarli come parametri della linea di comando (parametro `pszCommandLine`).

L'ultimo modo che consente a più threads di condividere oggetti di nucleo è tramite la primitiva `DuplicateHandle()`. Tramite questa primitiva, è possibile per un processo PA copiare un handle dalla propria tabella nella tabella di un processo PB già esistente. Naturalmente, il contatore di riferimento dell'oggetto viene incrementato di 1. Il processo PB non viene notificato dell'operazione avvenuta. Quindi, se si intende usare la `DuplicateHandle()`, è necessario usare anche un meccanismo di comunicazione interprocesso per notificare l'avvenuta operazione di duplicazione.

8.2 Gestione dei processi e dei thread

In Windows, il concetto di processo e il concetto di thread sono nettamente separati. Un processo in Windows è un tipo di oggetto di nucleo che possiede delle caratteristiche aggiuntive:

- un identificatore di processo unico nel sistema;
- uno spazio di indirizzamento privato;
- uno o più thread di esecuzione;
- una directory corrente;
- varie tabelle contenenti le risorse del processo (ad esempio la tabella degli handle aperti).

Un processo è una entità che definisce uno spazio di indirizzamento e ha bisogno di almeno un thread per eseguire. Quando tutti i thread del processo sono terminati, il processo termina.

Un thread è una entità concorrente e schedulabile. Anche i thread sono descritti da oggetti di nucleo. Ad un thread viene associato un identificatore univoco nel sistema, una funzione da eseguire, un "contesto" (ovvero l'insieme dei registri del processore), un puntatore allo stack. Tutti i thread appartenenti allo stesso processo condividono il suo spazio di indirizzamento. Quando un processo viene creato, viene automaticamente creato il suo thread "primario".

Windows supporta due tipi di applicazione. Le applicazioni grafiche (o GUI-based, *Graphical User Interface*) e le applicazioni testo (o CUI, *Console User Interface*).

Per questioni di spazio, non descriveremo qui processi di tipo GUI, e al solito rimandiamo a un manuale di programmazione Win32 per maggiori dettagli. Ci concentriamo invece sui processi di tipo CUI, i quali devono contenere la seguente funzione `main`:

```
int __cdecl main(
    int argc,
    char *argv[],
    char *envp[]);
```

L'intestazione di tale funzione è analoga a quella dettata dallo standard ANSI C. In particolare, il parametro `argc` indica il numero di argomenti passati sulla linea di comando; il parametro `argv[]` è un vettore di `argc` stringhe contenente tutti i parametri passati sulla linea di comando; il parametro `envp[]` è un vettore di stringhe contenente tutte le variabili di ambiente e il loro valore.

La funzione `main()` viene associata al thread primario creato insieme al processo. Successivamente, altri thread possono essere creati tramite la chiamata di sistema `CreateThread`:

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    DWORD cbStack,
    PTHREAD_START_ROUTINE pfnStart,
    PVOID pvParam,
    DWORD fdwCreate,
    PDWORD pdwThreadID);
```

dove:

- Il parametro `psa`, come detto in precedenza, è un puntatore a una struttura contenente i parametri di sicurezza.
- Il parametro `cbStack` viene usato per definire un limite massimo alla dimensione dello stack per il thread. Se viene passato 0, viene utilizzato il valore di default (solitamente pari a 1 Mb).
- Il parametro `pfnStart` è il puntatore alla funzione che verrà eseguita dal thread. La funzione deve avere il seguente prototipo:

```
DWORD WINAPI MyThread(PVOID param);
```

- Il parametro `pvParam` viene copiato nel parametro `param` del thread.
- Il parametro `fdwCreate` può assumere il valore `CREATE_SUSPENDED`, nel qual caso il thread creato viene posto nello stato di sospensione, oppure il valore 0, nel qual caso il thread viene posto nello stato di pronto.
- Infine, il parametro `pdwThreadID` è un puntatore a una variabile che dopo la chiamata a `CreateThread` conterrà l'identificatore del thread. La funzione restituisce uno handle all'oggetto thread appena creato.

Un oggetto thread possiede un contatore interno, detto *suspend counter*. Se tale contatore è pari a 0, allora il thread viene posto nello stato di pronto, altrimenti il thread è sospeso. Quindi, se il thread viene creato con il parametro `fdwCreate` pari a `CREATE_SUSPENDED`, il suo *suspend counter* viene inizialmente posto a 1, altrimenti vale 0. Il *suspend counter* di un thread può essere decrementato tramite la primitiva:

```
VOID ResumeThread(HANDLE hTh);
```

dove `hTh` è l'handle del thread a cui decrementare il *suspend counter*.

Il contatore può essere incrementato tramite la seguente primitiva:

```
VOID SuspendThread(HANDLE hTh);
```

dove `hTh` è lo handle del thread da sospendere. Alternativamente, un thread può autosospendersi per un certo tempo tramite la primitiva `Sleep`. Tale primitiva sospende il thread per un numero di millisecondi pari al suo argomento:

```
VOID Sleep(DWORD dwMilliseconds);
```

dove `dwMilliseconds` può valere:

- 0: in tal caso, il thread chiamante rilascia il tempo rimanente nel quanto di esecuzione corrente al prossimo thread pronto con uguale priorità (vedi paragrafo 8.2.1); nel caso in cui non ci siano altri thread di pari priorità, la primitiva ritorna immediatamente¹;
- INFINITE: in tal caso, il thread chiamante viene sospeso indefinitamente, ponendo il suo *suspend counter* a 1; potrà essere risvegliato da una chiamata a `ResumeThread()`;
- un numero intero di millisecondi; in tal caso, il thread verrà sospeso per un intervallo di tempo minimo pari al numero di millisecondi specificato.

Creazione di un thread e librerie run-time La primitiva `CreateThread()` non fa alcuna assunzione sulla struttura del programma e in particolare non fa assunzioni sulla libreria a run-time utilizzata dal compilatore. Quindi, se si usa direttamente la `CreateThread()` possono nascere alcune inconsistenze a livello di programma C. Ad esempio, il compilatore C assume che ci sia un solo thread, e quindi crea un'unica variabile `errno` per tutto il programma. La variabile `errno` viene utilizzata dalla libreria run-time per memorizzare il codice dell'ultimo errore causato dal programma. Nel caso in cui si vogliano creare altri thread, è quindi opportuno prendere alcuni accorgimenti: per prima cosa bisogna collegare la libreria run-time multi-thread del linguaggio C che duplica alcune strutture dati globali in modo che ogni thread abbia la sua variabile `errno`. Inoltre, invece di usare la chiamata di funzione `CreateThread()`, bisogna usare la funzione di libreria C `_beginthreadex()` che crea le strutture globali interne alla libreria run-time e poi invoca la `CreateThread()`. La funzione `_beginthreadex()` ha dei parametri analoghi a quelli della `CreateThread()`. Rimandiamo alla documentazione Microsoft per maggiori dettagli su tali funzioni.

Nel seguito del capitolo, per semplicità di esposizione, ci limiteremo ad usare la primitiva `CreateThread()`. Il lettore però è avvisato: in programmi reali, è sempre opportuno utilizzare la funzione `_beginthreadex()`.

8.2.1 Lo Schedulatore

Lo schedulatore di Windows è basato su una politica mista che è una via di mezzo fra una politica prioritaria e la politica round robin. I precisi dettagli interni del funzionamento dello schedulatore non sono stati divulgati dalla Microsoft e la sua implementazione può variare da una versione all'altra di Windows. In questa sezione ci limiteremo a descrivere i principi generali di funzionamento.

A ogni thread è associata una priorità assoluta tra 0 e 31 che viene calcolata come somma di due componenti: una “classe” di priorità assegnata al processo, più una priorità relativa per il thread. Windows supporta 6 classi, in ordine decrescente di priorità:

Real-time: i thread dei processi in questa classe hanno precedenza sui thread di ogni altro processo nel sistema, e possono avere precedenza su alcune componenti del

¹ La chiamata `Sleep(0)` si comporta allo stesso modo del metodo `yield()` di Java; vedi il paragrafo B.2.4 per maggiori dettagli.

sistema operativo. Questa classe di funzionamento va usata con cautela, poiché una applicazione non corretta che esegua secondo priorità real-time può bloccare l'intero sistema.

High: i thread dei processi in questa classe hanno precedenza su tutti i thread delle classi inferiori. Tale classe serve per implementare applicazioni critiche che debbono andare in esecuzione con tempo di risposta minimo. Per esempio, il "Task Manager", l'applicazione che permette di sospendere altre applicazioni bloccate nel sistema, esegue in questa classe di priorità.

Above Normal: i thread dei processi in questa classe hanno bisogno di tempi di risposta abbastanza brevi e quindi hanno una priorità leggermente più alta rispetto al funzionamento normale.

Normal: rappresenta la classe più usata per la maggioranza delle applicazioni Windows.

Below Normal: i thread dei processi appartenenti a questa classe hanno priorità inferiore al normale. Di solito, tali processi rappresentano attività che possono essere eseguite con priorità molto bassa.

Idle: i thread appartenenti a processi di questa classe eseguono quando il sistema non ha altro da fare.

Per ogni classe di priorità viene assegnata una priorità "nominale". I valori numerici di ciascuna di queste priorità nominali non vengono riportati nella documentazione di Windows in quanto tali valori sono passibili di cambiamento da una versione all'altra del sistema operativo. Per questo motivo il funzionamento di una applicazione non dovrebbe mai dipendere sui valori numerici assoluti delle priorità, quanto piuttosto sui valori relativi. Nella tabella 8.2 riportiamo le priorità nominali per ciascuna delle classi di priorità sopra descritte nel caso del sistema Windows 2000.

Priority class	Nominal Priority
Real-Time	24
High	13
Above Normal	10
Normal	8
Below Normal	6
Idle	4

Tabella 8.2 Priorità nominali in Windows 2000.

A ogni thread può essere assegnata una priorità relativa fra le seguenti:

Time critical: se il thread appartiene a un processo della classe Real-Time, allora la priorità del thread è pari a 31, altrimenti la priorità del thread è pari a 15.

Highest: al thread viene assegnata una priorità pari a 2 valori sopra la priorità nominale del processo a cui appartiene.

Above Normal: al thread viene assegnata una priorità pari a 1 valore sopra la priorità nominale del processo a cui appartiene.

Normal: al thread viene assegnata una priorità pari alla priorità nominale del processo a cui appartiene.

Below Normal: al thread viene assegnata una priorità pari a 1 valore sotto la priorità nominale del processo a cui appartiene.

Lowest: al thread viene assegnata una priorità pari a 2 valori sotto la priorità nominale del processo a cui appartiene.

Idle: se il thread appartiene a un processo della classe Real-Time, allora la priorità del thread è pari a 16, altrimenti è pari a 1.

Quindi, ad esempio, se un thread con priorità “Highest” appartiene a un processo della classe di priorità “Above Normal”, gli viene assegnata una priorità pari a 12 (nel sistema Windows 2000). Tale priorità viene anche chiamata “priorità base” del thread.

Per ognuno dei possibili livelli di priorità, lo schedulatore di Windows mantiene una coda che viene gestita in round robin. Il sistema definisce un quanto temporale, di solito pari a 10 millisecondi. Lo schedulatore manda in esecuzione il primo thread della coda a priorità più alta fra tutte le code che posseggono almeno un thread pronto. Il thread selezionato può eseguire fino a quando:

- il quanto temporale termina; in tal caso, il thread viene reinserito in fondo alla coda relativa alla sua priorità;
- il thread si sospende in attesa di un evento esterno o interno;
- un thread a più alta priorità è stato attivato.

In tutti e tre i casi, lo schedulatore viene invocato nuovamente per scegliere un nuovo thread da eseguire.

Un ulteriore meccanismo, chiamato *dynamic priority boost*, permette di ridurre il tempo di risposta dei thread interattivi. Se un thread sospeso viene riattivato, gli viene inizialmente assegnata una priorità più alta di due valori rispetto alla sua priorità base. Dopo che il thread esegue per un quanto temporale, la sua priorità viene decrementata di 1. In ogni caso, comunque, la priorità del thread non può mai essere inferiore alla sua priorità base.

Per chiarire il funzionamento di tale meccanismo, facciamo un esempio pratico. Supponiamo che un thread bloccato in attesa di input dall’utente abbia una priorità base pari a 8. Nel momento in cui l’utente sveglia il processo cliccando con il mouse su un elemento dell’interfaccia grafica, il thread viene sbloccato e inserito nella coda pronti relativa alla priorità 10. Supponiamo che per processare la richiesta il thread impieghi 40 millisecondi. Quindi, dopo un quanto temporale (pari a 10 millisecondi) la sua priorità viene decrementata a 9 e il thread viene inserito nella coda a priorità inferiore. Quindi, adesso il thread deve cedere il passo a tutti i thread a priorità maggiore di 9. Quando il thread ritorna in esecuzione, esegue per un altro quanto e al termine la sua priorità viene ulteriormente decrementata a 8. Da questo momento in poi, la sua priorità è pari alla sua priorità di base e quindi non verrà ulteriormente decrementata.

Per impostare la classe di priorità di un processo, si può usare la seguente chiamata di sistema:

```
BOOL SetPriorityClass(
    HANDLE hProcess,
    DWORD fdwPriority);
```

passando come parametri lo handle del processo (`hProcess`) e il valore della classe di priorità (`fdwPriority`). La chiamata restituisce un booleano per indicare l'esito della chiamata stessa. Analogamente, per leggere il valore della classe di priorità di un processo si può usare la seguente chiamata

```
DWORD GetPriorityClass(HANDLE hProcess);
```

che restituisce la priorità del processo il cui handle è passato come parametro.

Per impostare e leggere la priorità di un thread, si possono usare le seguenti chiamate di sistema:

```
BOOL SetThreadPriority(
    HANDLE hThread,
    int nPriority);
int GetThreadPriority(HANDLE hThread);
```

dove i parametri hanno gli stessi significati di quelli visti per le chiamate relative ai processi.

8.3 Sincronizzazione fra thread

In Windows distinguiamo due tipi di sincronizzazione fra thread. Un primo insieme di meccanismi di sincronizzazione può essere usato come meccanismo “leggero” per sincronizzare due thread appartenenti allo stesso processo e consiste nell’usare la famiglia di funzioni `Interlocked`. Tali meccanismi sono molto semplici e di basso livello, ma molto efficienti soprattutto nell’uso su piattaforme multiprocessore.

Per sincronizzazioni complesse oppure nel caso in cui si vogliano sincronizzare due thread appartenenti a processi diversi, è necessario usare gli opportuni oggetti di nucleo. Alcuni oggetti di nucleo sono stati appositamente definiti per scopi di sincronizzazione, come i `Semaphore` e i `Mutex`. Comunque, come vedremo, molti altri oggetti di nucleo, come i `Process` e i `Thread` possiedono delle funzionalità di sincronizzazione.

8.3.1 La famiglia di funzioni `Interlocked`

Nel capitolo 3 (paragrafo 3.2.1) è stata mostrata una soluzione al problema della mutua esclusione adatta soprattutto per la realizzazione di brevi sezioni critiche e, in particolare, in ambienti multielaboratore e che prevedono che un thread, che tenta di entrare in una sezione critica occupata, resti in attesa attiva. Questa soluzione è stata mostrata mediante la realizzazione delle due funzioni `LOCK` e `UNLOCK` (vedi paragrafo 3.2.1). Tale realizzazione prevede l’uso di una particolare istruzione macchina, come ad esempio la `TSL` (*Test and Set Lock*) presente su molti processori.

Questo meccanismo viene indicato, spesso, con il nome di meccanismo degli *spinlock*, o anche degli *spinsemaphore*, per indicare proprio questa caratteristica relativa all’attesa attiva in cui un thread rimane quando trova la sezione critica occupata.

Un’altra istruzione che può essere usata al posto della `TSL` per questo scopo, e che è spesso prevista insieme, o in alternativa, alla `TSL`, è l’istruzione `XCH` (*Exchange*).

ge) che in maniera atomica scambia il contenuto di un registro del processore con una locazione di memoria:

```
XCH R0 , x
```

In questo caso, il contenuto del registro R0 viene scambiato con la locazione di memoria x.

Windows mette a disposizione, a livello utente, alcune funzioni che forniscono le stesse funzionalità delle precedenti istruzioni hardware e che possono essere utilizzate per realizzare il meccanismo degli *spinlock* qualora il processore non fornisca tali istruzioni direttamente a livello hardware.

Ad esempio viene fornita la seguente funzione `InterlockedExchange()`, funzionalmente equivalente all'istruzione XCH:

```
LONG InterlockedExchange(
    PLONG p1Target,
    LONG lValue);
```

che, in maniera atomica, assegna alla locazione di memoria puntata da p1Target il valore lValue e contemporaneamente ritorna il valore precedentemente contenuto in p1Target.

Usando questa funzione, si può implementare il meccanismo degli spinlock, del tutto equivalente alle due funzioni LOCK e UNLOCK viste nel paragrafo 3.2.1, nel seguente modo:

```
void SpinLock(PLONG pSpinVar) {
    while (InterlockedExchange(pSpinVar, TRUE) == TRUE);
}

void SpinUnlock(PLONG pSpinVar) {
    *pSpinVar = TRUE;
}
```

La funzione `InterlockedExchange()` viene implementata direttamente se il processore fornisce nativamente l'istruzione XCH, altrimenti viene simulata opportunamente.

Windows mette a disposizione un'intera famiglia di funzioni `Interlocked`, come ad esempio le seguenti:

```
LONG InterlockedExchangeAdd(
    PLONG p1Addend,
    LONG lIncrement);

LONG InterlockedCompareExchange(
    PLONG p1Destination,
    LONG lExchange,
    LONG lComparand);
```

La prima incrementa in maniera atomica la locazione di memoria puntata da p1Addend della quantità lIncrement e ritorna il vecchio valore.

La `InterlockedCompareExchange()` confronta il contenuto della locazione puntata dal puntatore `p1Destination` con `lComparand`: se sono uguali, copia il valore di `lExchange` nella locazione puntata da `p1Destination` e ritorna il suo valore. Altrimenti, ritorna il contenuto di `p1Destination`.

Microsoft fornisce anche il concetto di sezione critica tramite il tipo di dato `CRITICAL_SECTION` e le due funzioni `EnterCriticalSection()` per entrare nella sezione critica, e `LeaveCriticalSection()` per rilasciarla. Ad esempio, il seguente frammento di codice implementa una sezione critica:

```
CRITICAL_SECTION cs;
...
InitializeCriticalSection(&cs);
...
EnterCriticalSection(&cs);
<Sezione Critica>
LeaveCriticalSection(&cs);
...
```

Se un thread si trova dentro una sezione critica (cioè ha effettuato l'operazione `EnterCriticalSection(&cs)` ma non ha ancora effettuato l'operazione `LeaveCriticalSection()`), e un altro thread tenta di accedere alla stessa sezione critica, il secondo thread viene sospeso in attesa che la sezione critica venga liberata. Questo implica un cambio di contesto fra thread. Ancora una volta, è importante sottolineare che tale meccanismo è disponibile soltanto per la sincronizzazione fra thread appartenenti allo stesso processo, in quanto richiede la definizione di una variabile `cs` in una zona di memoria a comune fra tutti i thread che vogliono accedere alla sezione critica.

Inoltre, se un thread si trova dentro una sezione critica definita dalla variabile `cs`, esegue un'altra volta l'operazione `EnterCriticalSection(&cs)`, esso non si blocca: comunque, per liberare la sezione critica il numero di chiamate alla funzione `LeaveCriticalSection(&cs)` deve corrispondere al numero di chiamate alla `EnterCriticalSection(&cs)`.

Come anticipato, queste funzioni sono molto efficienti. Nella maggior parte delle implementazioni, esse non richiedono un salto nello spazio kernel. Esse possono essere molto utili per realizzare sezioni critiche molto brevi oppure per proteggere alcune semplici operazioni (come l'incremento di una variabile) da interferenze.

Comunque, esse forniscono meccanismi atomici di livello molto basso, e diventa complicato realizzare dei meccanismi di sincronizzazioni più complessi basandosi solo su tali funzioni. Nel prossimo paragrafo, invece, ci occuperemo di meccanismi di sincronizzazione realizzati tramite oggetti di nucleo.

8.3.2 Oggetti di sincronizzazione

La maggior parte degli oggetti di nucleo, descritti precedentemente, possono essere usati per scopi di sincronizzazione. Un oggetto di nucleo (a parte poche eccezioni) può trovarsi nello stato “segnalato” (*signaled*) o “non-segnalato” (*non-signaled*). Un thread può mettersi in attesa che un certo oggetto diventi segnalato tramite la seguente funzione:

```
DWORD WaitForSingleObject(
    HANDLE hObject,
    DWORD dwMilliseconds);
```

Se l'oggetto identificato dallo `HANDLE hObject` è nello stato segnalato, allora la funzione ritorna immediatamente. Altrimenti, il thread chiamante si blocca in attesa che:

- l'oggetto identificato da `hObject` diventi segnalato; in questo caso la funzione ritorna la costante `WAIT_OBJECT_0`;
- oppure che sia trascorso un numero di millisecondi pari a `dwMilliseconds` sia passato; in questo caso la funzione ritorna la costante `WAIT_TIMEOUT`.

Quindi, il secondo parametro serve a limitare il tempo di attesa (*time-out*) per cui il thread può rimanere bloccato sull'oggetto di sincronizzazione. Se il parametro `dwMilliseconds` è impostato a `INFINITE`, il thread può bloccarsi indefinitivamente.

La regola perché un oggetto passi dallo stato segnalato allo stato non-segnalato dipende dal tipo di oggetto. Nel seguito, riportiamo il comportamento di alcuni importanti oggetti di nucleo.

Thread e processi Se un thread è stato creato ma non ha ancora terminato la sua esecuzione, il corrispondente oggetto di nucleo è nello stato non-segnalato. Non appena il thread termina, l'oggetto corrispondente passa nello stato segnalato, e i thread bloccati in attesa su di esso con la `WaitForSingleObject()` vengono svegliati. Tale meccanismo può essere appunto utilizzato per attendere la terminazione di un thread.

Come esempio di utilizzo di questo meccanismo, in figura 8.3 viene mostrato un semplice programma in cui il thread primario, rappresentato dalla funzione `main()`, crea un altro thread e si mette in attesa della sua terminazione.

Il `main()` invoca la primitiva `CreateThread()`, specificando come parametri dei valori di default (`NULL` e `0`) per i parametri di sicurezza e per la dimensione dello stack, rispettivamente. Il corpo del thread è la funzione `mythread`, il cui indirizzo viene passato come terzo parametro. Il quarto parametro, che rappresenta il puntatore al parametro da passare al thread, vale `0`: questo significa che il thread non potrà usare il parametro `param` pena il sollevarsi di una eccezione di page fault. Il quinto parametro vale `0`, e questo vuol dire che il thread viene posto immediatamente nello stato di pronto. Infine, l'ultimo parametro è un valore di ritorno: alla variabile `theId` verrà assegnato l'identificatore del thread creato.

Al ritorno della primitiva `CreateThread()`, l'oggetto di nucleo thread è stato creato ed è nello stato *non segnalato*. Il thread corrispondente viene posto nello stato di pronto. Subito dopo aver creato il nuovo thread, il thread primario si mette in attesa della sua terminazione invocando la primitiva `WaitForSingleObject()` con parametro `INFINITE`: in questo caso, non viene settato alcun time-out.

Il nuovo thread, dopo aver inviato un messaggio sullo *standard output* (nel caso il programma sia stato lanciato dalla linea di comando il messaggio viene visualizzato sullo schermo), invoca la primitiva `Sleep()`. Quindi, stampa un altro messaggio e ritorna. Quando la funzione `mythread` ritorna, il thread termina. A quel punto,

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

DWORD WINAPI mythread(PVOID *param)
{
    printf("sono il thread 1!\n");
    Sleep(1000);
    printf("Thread 1: prima di uscire\n");
    return 0;
}

int main(int argc, char* argv[])
{
    HANDLE hTh;
    DWORD thId;

    printf("prima della creazione\n");
    hTh = CreateThread(NULL, 0, mythread, 0, 0, &thId);
    printf("dopo la creazione, in attesa\n");
    WaitForSingleObject(hTh, INFINITE);
    printf("OK!\n");
    Sleep(3000);
    return 0;
}
```

Figura 8.3 Esempio in cui il thread main crea un altro thread e aspetta la sua terminazione.

l’oggetto di nucleo corrispondente diventa *segnalato*. Quindi, il thread primario si sblocca e può continuare la sua esecuzione.

Gli oggetti di tipo processo si comportano esattamente nello stesso modo. È quindi possibile utilizzare lo stesso meccanismo per attendere la terminazione di un processo.

Semafori generici È possibile creare un semaforo tramite la chiamata di sistema `CreateSemaphore()` già vista precedentemente (paragrafo 8.1.4).

Fra i processi che devono sincronizzarsi, uno crea l’oggetto semaforico che dovrà essere utilizzato per la sincronizzazione, mentre tutti gli altri si limitano ad aprire l’oggetto tramite la primitiva `OpenSemaphore()`.

Un semaforo si trova nello stato *segnalato* se il contatore è maggiore di zero, e non-*segnalato* se il contatore è uguale a 0. Se applicata a un semaforo, la chiamata di sistema `WaitForSingleObject()` controlla il valore del contatore: se è maggiore di zero, lo decrementa di una unità e ritorna immediatamente. Notare che se in seguito a tale decremento il contatore diventa zero, l’oggetto passa nello stato *non-segnalato*. Se il contatore è uguale a zero, blocca il thread chiamante. La funzione duale è la `ReleaseSemaphore()`, la cui intestazione è la seguente:

```
LONG ReleaseSemaphore (
    HANDLE hSem,
    LONG lReleaseCount,
    LPLONG lPreviousCount );
```

dove:

- hSem è lo handle del semaforo sul quale si vuole operare;
- lReleaseCount è di quanto si vuole incrementare il valore del contatore;
- lPreviousCount è il valore del contatore precedente alla chiamata della primitiva.

Tale funzione incrementa il valore del contatore di lReleaseCount. Se qualche thread era bloccato sul semaforo, esso viene svegliato e il contatore decrementato di 1. Il numero di thread sbloccati con una operazione di rilascio dipende dal valore del contatore e del parametro lReleaseCount. Ad esempio se il valore del contatore è 0 e ci sono 3 thread bloccati, la seguente operazione:

```
ReleaseSemaphore(hMySem, 2, 0);
```

sblocca due thread. Alla fine dell'operazione il valore del contatore è ancora 0: infatti, il valore viene incrementato di 2 unità dalla ReleaseSemaphore, ma viene decrementato di 1 unità da ciascuno dei due thread risvegliati.

Per fare un esempio di uso dei semafori generici di Windows, nel seguito riportiamo il codice di una applicazione del tipo *Produttore/Consumatore* (vedi Capitolo 3).

```
#include "stdafx.h"
#include <windows.h>
#include <process.h>
#include <stdio.h>

class MailBox {
    int nelem;
    int *array;
    int primo;
    int ultimo;
    HANDLE sempieno;
    HANDLE semvuoto;
public:
    MailBox(int max) : nelem(max), primo(0), ultimo(0)
    {
        array = new int[max];
        sempieno = CreateSemaphore(0, max, max, 0);
        semvuoto = CreateSemaphore(0, 0, max, 0);
    }
    void inserisci(int elem)
    {
        WaitForSingleObject(sempieno, INFINITE);
        array[primo]=elem;
        primo = (primo+1)%nelem;
        ReleaseSemaphore(semvuoto, 1, 0);
    }
    int estrai()
    {
        int ret;
        WaitForSingleObject(semvuoto, INFINITE);
```

```
ret=array[ultimo];
ultimo = (ultimo+1)%nelem;
ReleaseSemaphore(sem pieno, 1, 0);
return ret;
}
~MailBox()
{
    CloseHandle(sem pieno);
    CloseHandle(sem vuoto);
    delete array;
}
};

MailBox mbox(5);

DWORD WINAPI produttore(PVOID arg)
{
    int i;
    for(i=0; i<20; i++) {
        printf("Produttore: inserisco %d\n", i);
        mbox.inserisci(i);
    }
    return 0;
}

DWORD WINAPI consumatore(PVOID arg)
{
    int i,r;
    for(i=0; i<20; i++) {
        r = mbox.estrai();
        printf("Consumatore: estraggo %d\n", r);
    }
    return 0;
}

int main(int argc, char* argv[])
{
    HANDLE hCons, hProd;
    DWORD tCons, tProd;
    printf("Inizializzazione!\n");

    hProd = CreateThread(0, 0, produttore, 0, 0, &tProd);
    hCons = CreateThread(0, 0, consumatore, 0, 0, &tCons);

    WaitForSingleObject(hProd, INFINITE);
    WaitForSingleObject(hCons, INFINITE);

    printf("Finito!\n");
    Sleep(4000);
    return 0;
}
```

Nell'esempio, ci sono 3 thread: il thread primario (funzione main) si occupa di inizializzare le strutture dati e creare gli altri due thread, produttore e consumatore. Il thread produttore manda dei dati di tipo intero a una struttura dati mbox di tipo MailBox mentre il thread consumatore riceve degli interi dalla stessa struttura dati.

Il codice del programma è stato scritto in linguaggio C++ ([91]). La struttura dati mbox è un oggetto della classe MailBox. Si tratta di una coda circolare max posizioni. La classe include due semafori di sincronizzazione, il semaforo sempiero, inizializzato a max, e il semaforo semvuoto, inizializzato a 0.

Se il produttore produce a un ritmo più veloce di quanto il consumatore riesca a prelevare, allora dopo un po' la struttura dati si riempie e il produttore si blocca in attesa che il consumatore liberi almeno una posizione. Viceversa, se il consumatore preleva i dati a un ritmo superiore rispetto a quanto il produttore sia in grado di produrre, presto la coda si svuoterà e il consumatore si bloccherà in attesa che il produttore inserisca almeno un dato.

Il produttore esegue un chiamata alla funzione membro inserisci() la quale, per prima cosa effettua una WaitForSingleObject() sul semaforo sempiero. Se la chiamata va a buon fine (cioè il thread non si blocca), allora c'è uno spazio libero nel buffer e quindi è possibile effettuare l'inserimento. Altrimenti il thread si blocca in attesa che venga effettuata una ReleaseSemaphore sul semaforo sempiero. Dopo aver inserito il dato, il produttore invoca la ReleaseSemaphore sul semaforo semvuoto, segnalando che adesso c'è una posizione libera in meno.

Il consumatore si comporta in maniera duale, invocando la funzione membro estrai().

Semafori di mutua esclusione Il sistema operativo Windows fornisce anche i semafori di mutua esclusione, o mutex. Per creare un mutex, è necessario invocare la seguente funzione:

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpSecAtt,
    BOOL bInitialOwner,
    LPCTSTR lpName);
```

Come al solito, è possibile condividere l'oggetto appena creato con un altro processo, specificando una stringa contenente il nome unico che si vuole assegnare all'oggetto. Un altro processo a questo punto potrà aprire l'oggetto con la seguente primitiva:

```
HANDLE OpenMutex(
    DWORD dwDesiredAccess,
    BOOL bInherit,
    LPCTSTR lpName);
```

Un oggetto di mutex può essere dinamicamente associato a un thread che ne diventa il "proprietario" (*owner*). Inizialmente un oggetto di mutex è segnalato e non ha alcun proprietario. Se un thread effettua una WaitForSingleObject() su un

mutex che si trova nello stato segnalato, allora il mutex passa nello stato non-segnalato e il thread che ha effettuato l'operazione diventa il proprietario del mutex. Se un thread effettua una operazione di `WaitForSingleObject()` su un mutex che si trovi nello stato non-segnalato, si hanno due casi: se il thread proprietario del mutex è diverso dal thread chiamante, allora il thread chiamante si blocca; altrimenti il thread continua l'esecuzione.

Per rilasciare il mutex, si può usare la seguente primitiva:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

La funzione può essere invocata con successo soltanto dal proprietario del mutex, altrimenti la chiamata fallisce e ritorna con FALSE. Se la chiamata ha successo, e almeno un thread è bloccato sul mutex, allora il nuovo thread viene sbloccato e diventa il nuovo proprietario del mutex, mentre il mutex rimane nello stato non-segnalato. Se invece nessun thread è bloccato sul mutex in corrispondenza della chiamata alla `ReleaseMutex()`, il mutex diventa segnalato e senza proprietario.

8.4 Gestione della memoria virtuale

In Windows, ogni processo ha a disposizione uno spazio virtuale di memoria di 4 gigabyte, corrispondenti a 32-bit di indirizzamento. Di norma, tale memoria virtuale è divisa in due parti: la parte bassa, cioè i primi due gigabyte sono effettivamente disponibili per il processo, mentre i due gigabyte superiori sono riservati al sistema operativo e il processo non vi può accedere direttamente. I 2 gigabyte accessibili al processo sono ulteriormente suddivisi in 3 zone: la zona da 0x00000000 a 0x0000FFFF è riservata e serve per intercettare errori di programmazione come l'uso di puntatori non inizializzati; la zona da 0x00010000 a 0x7FEFFFFF è disponibile per il codice e i dati del processo; la zona da 0x7FFF0000 a 0x7FFFFFFF è di nuovo inaccessibile e serve a identificare errori di programmazione.

Tale memoria virtuale è divisa in pagine di dimensione fissa. La dimensione effettiva della pagina dipende dalla macchina fisica su cui sta funzionando il sistema operativo, e può andare dai 4 kilobyte delle piattaforme Intel e PowerPC, agli 8 kilobyte delle piattaforme Alpha. Per ogni processo, il gestore della memoria virtuale gestisce una tabella delle pagine. Data la grandezza dello spazio virtuale gestito da un processo, tale tabella viene implementata tramite una struttura a due livelli: un direttorio delle pagine e un insieme di tabelle delle pagine di dimensione ridotte (vedi capitolo 4, figura 4.34). Il direttorio contiene 1024 elementi, ognuno di 4 byte, chiamati PDE (*Page Directory Entry*). Ciascuno di tali elementi punta a una tabella delle pagine contenente 1024 elementi di 4 byte detti PTE (*Page Table Entry*) mostrati in figura 8.5. Infine, ogni PTE punta a un frame della memoria fisica. Quindi, un indirizzo virtuale viene diviso in 3 parti, come mostrato in figura 8.4: la prima parte identifica l'elemento nella tabella del direttorio; la seconda parte rappresenta l'indice nella tabella delle pagine; la terza parte è l'offset all'interno della pagina. Poiché la dimensione totale di tutte le tabelle delle pagine è di 4 megabyte, alcune di tali tabelle possono essere salvate su disco. Notare che nel caso in cui le pagine siano di 4 kilobyte, ogni tabella delle pagine occupa esattamente un frame fisico.

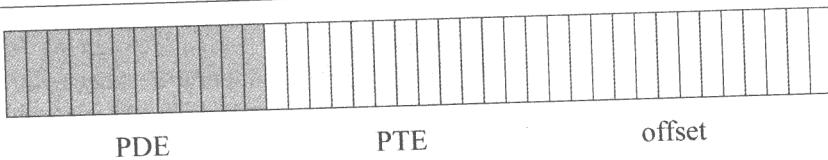


Figura 8.4 Struttura di un indirizzo virtuale in Windows.



Figura 8.5 Elemento della tabella delle pagine in Windows.

Il PTE viene diviso in 4 parti. Una prima parte descrive le proprietà di sicurezza della pagina in questione (ad esempio se accessibile sono in lettura o in scrittura, e da quali processi). Segue l'indirizzo fisico del frame a cui viene associata la pagina; quindi, la terza parte specifica su quale file su disco la pagina si trova quando non è in memoria; infine, l'ultima parte codifica lo stato della pagina. Una pagina può essere:

- *valida*, se è presente in memoria;
- *azzerata*, se è stata appena inizializzata ed è quindi pronta per essere utilizzata;
- *libera*, se non viene usata dal processo;
- *in attesa*, se deve essere caricata dal gestore della memoria;
- *modificata*, se è stata modificata dal processo ma non ancora salvata su disco;
- *difettosa* se presenta degli errori fisici in lettura o scrittura.

Il gestore della memoria virtuale implementa una tecnica di tipo *paginazione a domanda* con *resident set* di dimensioni variabili e con rimpiazzamento locale (vedi capitolo 4). Inoltre, Windows permette a processi con livelli di privilegio particolari di bloccare alcune pagine in memoria per ragioni di efficienza (ad esempio processi in tempo reale). Infine, tramite un meccanismo piuttosto complesso, è possibile permettere la condivisione di memoria fra più processi, permettendo la condivisione di alcune pagine.

8.5 File system

Storicamente, a partire da MS-DOS fino alla versione Windows Me, i sistemi Microsoft usavano il sistema FAT (vedi paragrafo 6.4.1). Il file system utilizzato dalle ultime versioni di Windows (in particolare Windows NT, Windows 2000 e Win-

dows XP) è chiamato NTFS e presenta prestazioni notevolmente superiori. In questa sezione ci limiteremo a descrivere il sistema NTFS.

Come nella maggior parte dei file system, NTFS supporta il concetto di partizioni logiche. Ogni partizione logica rappresenta un *volume*. Un volume può occupare un disco intero, parte di un disco, oppure anche più di un disco. Ogni volume è diviso in blocchi fondamentali, detti *cluster*, che rappresentano l'unità elementare di allocazione dei dati. La dimensione di un cluster è stabilita al momento della formattazione dell'unità di volume, e può variare. La dimensione minima è pari a un settore sul disco. Più grande è il cluster, più veloce è l'accesso al disco, ma maggiore è la frammentazione. Tipicamente, per volumi di dimensioni superiori ai 2 Gb, comunemente usati nei moderni sistemi, un cluster ha dimensione pari a 4 kilobyte.

Un file viene visto come un aggregato di *attributi*. Alcuni attributi rappresentano delle informazioni elementari associati ad ogni file, come il suo nome, il suo percorso assoluto (*absolute path*), la sua dimensione, i permessi in accesso e statistiche di vario tipo. Un attributo particolare è il contenuto stesso del file.

Ogni file in NTFS viene descritto da un opportuna struttura memorizzata nel *Master File Table* (MFT). Gli attributi di piccole dimensioni, come ad esempio la dimensione del file, sono memorizzati direttamente nella MFT e sono detti *attributi residenti*, mentre gli attributi di grandi dimensioni, come i dati contenuti nel file, sono memorizzati in una serie di cluster sul disco, detti *estensioni contigue*. Un puntatore ad ogni estensione contigua del file viene memorizzato nel MFT. Ogni file è identificato univocamente da un identificatore a 64 bit, di cui i primi 48 sono l'indice nella MFT contenente i dati relativi al file, e i 16 bit seguenti rappresentano un numero di sequenza. Ogni volta che un elemento della tabella viene riutilizzato per memorizzare un nuovo file, il numero di sequenza viene incrementato. In questo modo, è possibile per NTFS eseguire dei controlli di coerenza sul disco e scoprire, ad esempio, riferimenti a file ormai cancellati.

I file sono organizzati in *directory*, secondo la classica struttura ad albero propria dei sistemi operativi moderni. È possibile comunque, creare dei riferimenti simbolici a dei file, rendendo possibili strutture più complesse quali grafi aciclici.

Tutti i dati che rappresentano strutture del sistema operativo sono memorizzati in certi file particolari. Alcune di queste strutture sono ad esempio il MFT stesso, il file di *bitmap* che contiene l'indicazione di quali cluster sono liberi e quali sono occupati ecc.

NTFS è un file system *transazionale*. Infatti, tutte le operazioni sulle strutture dati interne del file system sono chiamate *transazioni* e assumono una forma particolare. Esiste un file di *log* che registra la sequenza di operazioni che vengono effettuate sulle strutture dati. Una transazione si compone di 3 fasi: la prima fase consiste nello scrivere sul file di log il tipo di operazione da compiere e tutti i suoi parametri; la seconda fase consiste nell'eseguire l'operazione vera e propria; nella terza fase, si scrive sul file di log che l'operazione è stata completata. L'uso delle transazioni è molto utile durante eventuali malfunzionamenti del sistema, come ad esempio cali di tensione o arresto critico. In tal caso, al momento del ripristino è possibile capire se una data operazione sul disco è stata portata a termine con successo oppure è stata interrotta lasciando qualche struttura dati in stato inconsistente. Nel secondo caso, è pos-

sibile perfino operare un recupero dell'operazione e quindi ripristinare la consistenza dei dati sul disco.

Il file di log potrebbe però crescere indefinitamente: per evitare ciò, a intervalli regolari di tempo (tipicamente ogni 5 secondi) un record speciale viene scritto sul file. Tutti i record precedenti tale record speciale possono essere eliminati dopo un po' di tempo. Il log è un file speciale di dimensione fisse e viene creato al momento della formattazione del volume.

8.6 Sommario

In questo capitolo è stata presentata una panoramica molto sintetica del sistema operativo Windows, e in particolare delle versioni NT, 2000 e XP. Come prima cosa è stata presentata l'evoluzione storica di tale sistema operativo nelle sue varie versioni.

La struttura interna del sistema, presentata nel paragrafo 8.1.2, è stata pensata per essere scalabile da sistemi mono processore a sistemi multi processore, e per essere espandibile e portabile su diverse macchine fisiche. Sono state discusse le somiglianze fra i sistemi operativi a micro kernel e la struttura di Windows nel paragrafo 8.1.3.

Quindi sono stati presentati gli oggetti di nucleo, e tra questi, gli oggetti "attivi" come i processi e i thread, e gli oggetti di sincronizzazione, tra cui i semafori e i mutex. Infine, è stata discussa l'organizzazione della memoria virtuale e quella del file system.

Naturalmente, questo breve capitolo ha il duplice scopo di mostrare come i concetti descritti nei primi sei capitoli del libro sono stati applicati nella progettazione e nella implementazione di Windows, e di invogliare il lettore ad approfondire gli argomenti trattati.

8.7 Note bibliografiche

La fonte principale di informazioni sul sistema Windows è naturalmente la documentazione fornita dalla Microsoft. L'insieme della documentazione va sotto il nome di *MSDN (Microsoft Developer Network)*, ed è disponibile anche on-line presso [65].

Una panoramica sulla struttura originale di Windows NT si trova in [82] e [83]. Sebbene si tratti di un testo un po' datato, bisogna tenere presente che la maggior parte della struttura di Windows NT è rimasta invariata anche nelle successive versioni.

Esistono tantissimi manuali di programmazione per Windows. In particolare raccomandiamo [75] per la sua chiarezza espositiva e [37] per la sua completezza. Ottimi manuali per la programmazione di driver per Windows sono [16] e [26], nei quali vengono forniti molti dettagli della struttura interna del sistema. C'è però da sottolineare che, data l'evoluzione rapida dello sviluppo del sistema Windows, alcuni dettagli interni di tali manuali potrebbero rilevarsi superati.

Infine, una buona trattazione sui problemi di gestione della sicurezza in Windows si trova in [100].

A

Elementi di sincronizzazione in ambiente distribuito

In questa appendice viene estesa la trattazione relativa alle interazioni tra processi al caso distribuito, con specifica introduzione agli strumenti offerti dal sistema operativo Unix per la realizzazione di applicazioni in una rete di calcolatori.

A.1 Introduzione alle reti

La grande diffusione delle nuove tecnologie nel settore dell'informazione ci ha ormai abituati a disporre dell'accesso a reti locali e geografiche per usufruire di servizi remoti come, ad esempio, e-mail, world wide web, sistemi di trasferimento di file tra nodi della rete e così via.

Per la realizzazione di questi servizi vengono utilizzate tecnologie hardware e software che consentono la comunicazione tra processi in una *rete di calcolatori*. Una rete può essere definita come l'insieme delle risorse (hardware e software) che vengono utilizzate per stabilire la connessione tra diversi elaboratori.

L'idea di collegare vari calcolatori risale ai primi anni '70, quando furono realizzate le prime reti per collegare *mainframe* a terminali remoti, e si è evoluta fino ad oggi, fino a determinare l'attuale diffusione di *Internet*, una rete di reti utilizzata da milioni di utenti in tutto il mondo.

Tutte le reti di calcolatori della prima generazione erano costituite da sistemi chiusi (o *sistemi proprietari*), composti da componenti provenienti da un unico fornitore. Questa caratteristica ha in breve tempo messo in evidenza seri problemi derivanti dalle difficoltà di integrare all'interno della stessa rete nodi costituiti da apparati diversi e di adattare le reti esistenti alla rapida evoluzione della tecnologia.

Per questi motivi, dalla fine degli anni 70 si cominciò a studiare e definire modelli di realizzazione di calcolatori aderenti a un modello di *sistema aperto*, cioè reti composte da componenti hardware e software eterogenei, eventualmente prodotti da costruttori diversi.

A questo scopo è stato necessario stabilire degli *standard* che definissero le caratteristiche funzionali dei componenti hardware e software della rete, per garantire la caratteristica di *interoperabilità* nel sistema, cioè la capacità di interazione tra nodi architetturalmente diversi, mediante l'adozione di interfacce comuni.

Il più importante risultato di quest'opera di standardizzazione è stata la definizione del modello OSI (*Open System Interconnection*), definito dall'*International Standards Organization* (ISO) [49], che propone un modello astratto per l'architettura di una rete aperta organizzato a strati, nel quale ogni strato realizza un sottoinsieme delle funzionalità necessarie a garantire la comunicazione nel sistema. Come si vedrà nel paragrafo A.1.1, ogni livello può essere realizzato da un particolare *protocollo*.

Nel paragrafo seguente verranno forniti alcuni cenni relativi alle architetture delle reti di calcolatori, facendo riferimento al modello OSI e ai protocolli specifici della rete Internet.

A.1.1 Il modello OSI e la rete Internet

Il modello OSI propone un modello di riferimento (*reference model*) stratificato per l'architettura di una rete. In particolare, esso suddivide le funzionalità necessarie alla rete in sette strati, numerati dal basso verso l'alto da 1 a 7, come illustrato dalla figura A.1. Ogni strato (o *livello*) rappresenta un modulo del sistema che offre alcuni servizi al livello superiore tramite una interfaccia standard; questi servizi vengono realizzati sfruttando i servizi offerti dal livello inferiore.

Ogni strato è concretamente rappresentato da un *protocollo di comunicazione* che realizza la comunicazione tra livelli dello strato superiore.

In generale, è possibile classificare i protocolli di comunicazione in due categorie:

- protocolli con connessione (*connection-oriented*);
- protocolli senza connessione (*connection-less*).

Nel primo caso, i servizi di comunicazione erogati dal protocollo utilizzano un *canale virtuale* (o *connessione*) che collega logicamente i due interlocutori, attraverso il quale avviene il trasferimento dei dati. In questo caso, quindi, prima di avviare lo scambio di informazioni tra i due sistemi, è necessario instaurare la connessione: una volta effettuata questa operazione, ogni informazione potrà essere trasferita attraverso il canale senza dover indicare esplicitamente il nome del destinatario. Al termine della comunicazione il canale andrà chiuso. Questa modalità di comunicazione, assimilabile a quella utilizzata nel sistema telefonico, è *affidabile*: infatti, poiché la gestione delle informazioni all'interno del canale segue una politica FI-FI, ciò garantisce che le informazioni trasferite tra mittente e destinatario vengano sempre recapitate al destinatario, esattamente nello stesso ordine con cui sono state inviate.

Invece nei protocolli di comunicazione *connection-less*, assimilabili al meccanismo utilizzato nel sistema postale, la comunicazione non usa un canale virtuale: pertanto ad ogni invio di dati è necessario specificare esplicitamente la destinazione del messaggio. L'assenza del canale rende la comunicazione più efficiente ma, in genere, meno affidabile.

OSI

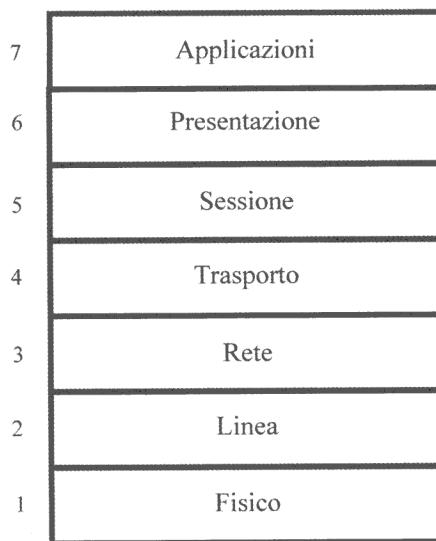


Figura A.1 Il modello OSI.

Nel seguito, verranno analizzate sinteticamente le caratteristiche dei diversi livelli del modello OSI.

Il ruolo dello strato *Fisico* (livello 1) è di attivare, mantenere e disattivare la connessione fisica fra due entità di livello 2. A questo scopo, il primo livello stabilisce le modalità di invio dei singoli bit sul mezzo fisico di trasmissione, mediante la specifica delle caratteristiche meccaniche (ad esempio, il tipo dei connettori) ed elettriche (ad esempio, il voltaggio dei segnali elettrici) associate al mezzo fisico trasmittivo.

Il livello di *Linea* (livello 2) ha il compito di attivare, mantenere e disattivare la connessione fisica fra due entità di strato 3, realizzando un collegamento affidabile diretto (punto-punto) tra due nodi di rete.

Il livello di *Rete* (livello 3) deve occuparsi dell'instradamento delle unità d'informazione (dette *pacchetti*), scegliendo la strada da percorrere attraverso la rete. Per realizzare questo compito è necessario adottare un modo univoco per l'individuazione dei nodi destinatari: a questo livello vengono quindi fissate le regole per l'indirizzamento dei nodi.

Il livello di *Trasporto* (livello 4) ha il compito specifico di assicurare il trasferimento dei dati tra strati di sessione di un nodo mittente e uno destinatario (end-to-end) senza che vi siano errori o duplicazioni. Tipicamente esso suddivide/riassembla in pacchetti i blocchi di informazioni da spedire/ricevere, tenendo conto dei requisiti del protocollo di rete.

Il livello di *Sessione* (livello 5) suddivide il dialogo fra le applicazioni in unità logiche (dette *sessions*). Una sessione è un collegamento logico e diretto tra due inter-

locutori, attraverso il quale avviene il dialogo. Ogni sessione viene esplicitamente aperta e, al termine del dialogo, essa viene chiusa. A questo scopo, il livello di sessione fornisce i meccanismi che permettono la chiusura *ordinata* (*soft*) del dialogo, con la garanzia che tutti i dati trasmessi siano arrivati a destinazione.

Poiché è possibile integrare nella stessa rete elaboratori architetturalmente differenti, nodi diversi possono adottare tecniche di rappresentazione dei dati diverse; è compito dello strato di *Presentazione* (livello 6) effettuare le opportune conversioni di formato sui messaggi in modo da compensare eventuali differenze di rappresentazione dei dati in arrivo e/o in partenza.

Infine, lo strato di *Applicazione* (livello 7) rappresenta il programma applicativo che necessita dell'interazione con altre applicazioni remote.

Il modello OSI è un modello di riferimento: esso stabilisce i compiti delegati ai protocolli dei vari strati e come questi devono interagire con gli altri strati; non viene specificato, invece, come devono essere realizzati i protocolli. Per ogni livello, quindi, si può pensare di avere diversi protocolli possibili (purché conformi al modello OSI). È importante osservare, tuttavia, che per ottenere un'unica rete geografica universale, gli strati di rete e di trasporto devono essere unici. Per questo motivo, le commissioni OSI hanno fatto un'eccezione e hanno definito due specifici protocolli, rispettivamente di rete (ISO 8473) e di trasporto (ISO 8073), che devono essere adattati da tutti i computer se si vuole ottenere la rete aperta universale secondo il modello OSI.

Tuttavia, mentre il modello di riferimento OSI è stato universalmente adottato come modo di organizzare le architetture delle reti, i protocolli di rete e di trasporto di OSI non hanno avuto successo; il motivo di ciò è dovuto alla grande diffusione di Internet e dei suoi protocolli TCP (*Transmission Control Protocol*, di livello 4) ed IP (*Internet Protocol*, di livello 3) incompatibili ed in concorrenza con quelli di OSI¹.

In figura A.2 l'architettura di Internet è confrontata con il modello OSI: come si può osservare, a differenza del modello OSI, la rete Internet si basa su 4 livelli.

Il livello più basso (*interfaccia di trasmissione*) assolve agli stessi compiti dei livelli 1 e 2 del modello OSI. Il secondo livello, corrispondente allo strato di rete del modello OSI, accoglie il protocollo di rete IP: poiché esso è un protocollo senza connessione, ciò implica che due messaggi inviati dallo stesso nodo mittente allo stesso destinatario potranno essere instradati su cammini fisici diversi. IP permette lo scambio di messaggi tra nodi appartenenti anche a reti diverse, definendo un sistema di indirizzamento universale dei nodi, nel quale ogni indirizzo, basato su 32 bit, associa al nodo che esso rappresenta la rete a cui appartiene; questa modalità di indirizzamento consente inoltre la strutturazione gerarchica del sistema in reti e sottoreti.

Al terzo livello (corrispondente allo strato di trasporto del modello OSI) si possono utilizzare i protocolli di trasporto TCP o UDP (*User Datagram Protocol*), a seconda dei requisiti imposti dalle applicazioni.

¹ Per questo motivo l'architettura di Internet è spesso indicata con il nome TCP/IP.

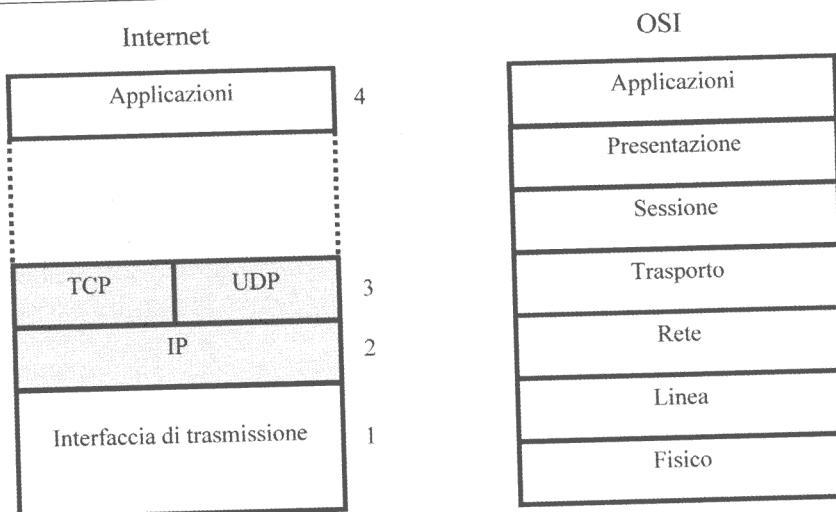


Figura A.2 Architettura di Internet.

Il TCP è un protocollo con connessione che consente la trasmissione affidabile di dati tra mittente e destinatario attraverso il canale virtuale. È importante evidenziare che le informazioni da trasferire attraverso il canale vengono trattate come una sequenza non strutturata di byte: questo implica (come si vedrà nel paragrafo A.2.4) che i *confini* dei messaggi non vengano mantenuti. Ciò significa che più messaggi spediti in sequenza dal mittente vengano ricevuti dal destinatario come un'unica sequenza di byte, nella quale non è previsto alcun separatore tra un messaggio e il successivo, lasciando al destinatario l'onere di riconoscere ogni messaggio in base a convenzioni concordate con il mittente (ad esempio, fissando una dimensione costante).

A differenza del TCP, il protocollo UDP è invece senza connessione e fornisce un servizio di trasporto per i messaggi non affidabile; inoltre, diversamente dal TCP i confini dei messaggi inviati vengono mantenuti fino alla destinazione.

Il quarto livello della rete *Internet* è rappresentato dalle *applicazioni*: l'architettura prevede infatti che le applicazioni possano interfacciarsi direttamente al livello di trasporto, risolvendo autonomamente i problemi peculiari dei livelli di presentazione e di sessione. Nel paragrafo seguente, si vedrà come queste problematiche potranno essere parzialmente affrontate in ambiente Unix mediante l'uso delle socket.

A.2 Sviluppo delle applicazioni di rete

Nel capitolo 3 sono state trattate le problematiche relative alle interazioni tra processi, introducendo le principali tecniche di sincronizzazione per processi concorrenti. Nei capitoli 7 e 8 tali tecniche sono state esemplificate mediante gli strumenti disponibili nelle famiglie di sistemi operativi *Unix* e *Windows*.

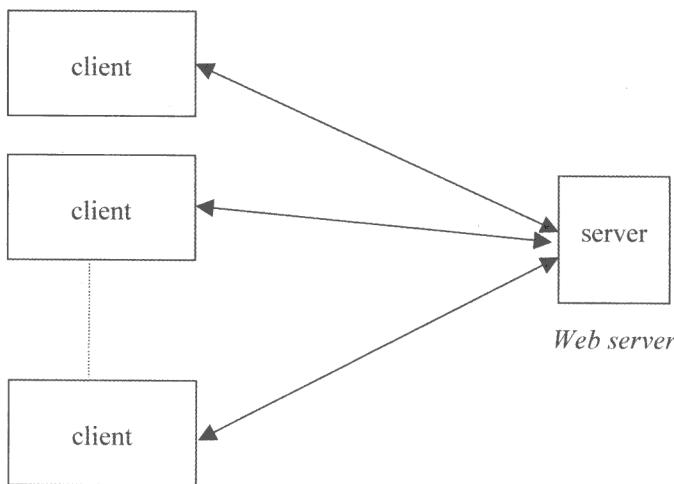


Figura A.3 Schema cliente-servitore.

In questa appendice si vuole estendere la trattazione relativa all’interazione tra i processi al caso distribuito, introducendo in particolare, gli strumenti offerti dai sistemi operativi Unix e Linux per la realizzazione di applicazioni all’interno di una rete.

A.2.1 Il modello cliente-servitore

La maggior parte delle applicazioni costituite da un insieme di processi distribuiti su diversi nodi di una rete (le *applicazioni di rete*) sono strutturate secondo lo schema *cliente-servitore* (*client-server*): nell’insieme di processi cooperanti è possibile individuare un processo servitore, che è dedicato a realizzare uno o più servizi, e un insieme di processi clienti, i quali rappresentano i potenziali fruitori dei servizi offerti dal servitore (vedi figura A.3).

Ad esempio, un sito web può offrire un insieme di servizi realizzati da un WEB server con il quale possono interagire più clienti (i *browsers* WEB, come *Internet Explorer*, *Netscape* ecc.); in questo caso, i servizi offerti potranno essere nel caso più semplice, l’invio di pagine *html*, oppure servizi più complessi, come interrogazioni di database, esecuzione di particolari funzioni e altro ancora. Il servizio www si basa sul protocollo *http*, che è un protocollo di livello applicativo che si appoggia sul protocollo TCP.

In generale, server e client risiedono in nodi diversi, e pertanto le informazioni scambiate tra clienti e servitore devono transitare attraverso la rete.

A.2.2 Le socket di Unix BSD

In questo paragrafo verranno forniti alcuni cenni sugli strumenti disponibili nei sistemi operativi della famiglia Unix/Linux per la realizzazione dell’interazione tra processi (ad esempio, client e servitori), all’interno di una rete.

Richiamando quanto detto nel capitolo 7, i principali meccanismi di interazione tra processi Unix sono rappresentati da *segnali* e *pipe*. I primi consentono la sincronizzazione tra processi sullo stesso nodo senza scambio di informazioni: l'unica informazione trasferita da un segnale è l'intero che lo identifica. Le pipe, invece, consentono lo scambio di messaggi tra processi appartenenti alla stessa gerarchia e residenti sullo stesso nodo. Questi strumenti, quindi, non sono idonei per esprimere la comunicazione tra processi all'interno di una rete.

Per questo motivo Unix BSD ha introdotto (dalla versione 4.2) un'interfaccia di programmazione (API) specifica per lo sviluppo di applicazioni di rete. Questa API, rappresentata da una libreria di funzioni C, è basata sul concetto di *socket*.

Una socket rappresenta il mezzo di comunicazione mediante il quale un processo può scambiare messaggi con altri processi.

Come si vedrà, le socket sono caratterizzate da vari attributi che ne differenziano le modalità di uso.

Indipendentemente dal valore degli attributi, gli aspetti peculiari della comunicazione tra processi mediante socket sono i seguenti:

- *eterogeneità*: la comunicazione può avvenire fra processi che risiedono in architetture diverse;
- *trasparenza*: la comunicazione fra processi avviene con le stesse modalità, indipendentemente dalla localizzazione fisica dei processi comunicanti;
- *indipendenza dalla rete*: l'interfaccia di comunicazione è indipendente dall'architettura della rete;
- *compatibilità con i file*: le socket sono rappresentate all'interno dei programmi in modo omogeneo ai file; questa caratteristica rende possibile l'applicazione dei tradizionali meccanismi Unix di ridirezione e piping dei comandi anche a canali di comunicazione tra processi remoti.

Ogni socket è definita in un *dominio* di comunicazione che stabilisce implicitamente l'*ambiente* nel quale avviene la comunicazione, le caratteristiche *semantiche* della comunicazione, e le convenzioni di *denotazione* dei processi comunicanti.

In particolare, l'ambiente di comunicazione, può coincidere con una rete, la cui architettura è definita da un particolare insieme di protocolli, ma può essere anche un unico sistema Unix nel quale i processi comunicanti non devono necessariamente appartenere a una stessa gerarchia.

Per quello che riguarda gli aspetti semanticici, la scelta di un particolare dominio può influire su caratteristiche come, ad esempio, l'affidabilità della comunicazione oppure la possibilità di eseguire delle comunicazioni uno-a-molti (*multicast*).

Riguardo alle modalità di *denotazione*, esse stabiliscono le convenzioni da utilizzare per indicare una *socket*: ad esempio, in una rete esse consistono nelle regole di formato da adottare per esprimere l'indirizzo dei nodi.

La prima scelta da compiere nel progetto di una applicazione distribuita riguarda, quindi, il dominio della comunicazione, specificabile mediante un nome standard. Ad esempio, se l'applicazione utilizza la rete internet viene scelto il dominio di comunicazione PF_INET (più precisamente, esso denota l'Internet Protocol Version 4); se, invece, i processi comunicanti eseguono su uno stesso nodo Unix, viene scelto il dominio PF_UNIX.

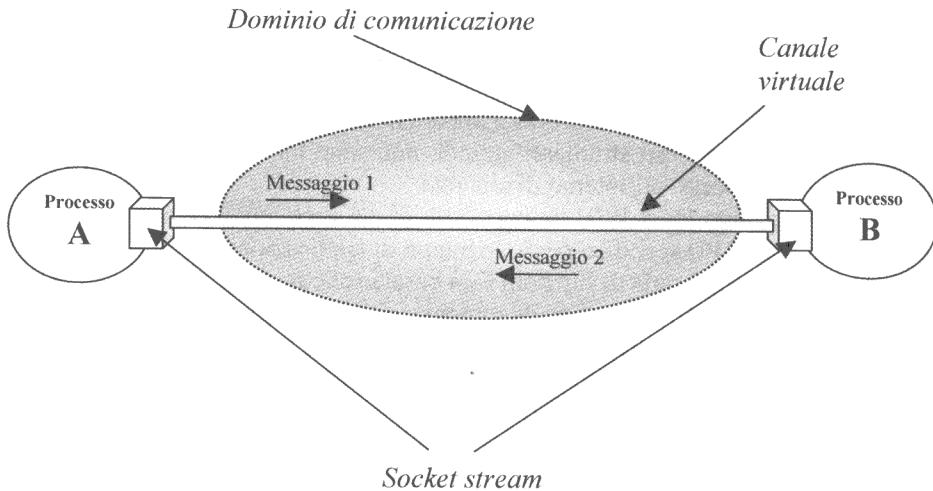


Figura A.4 Comunicazione tra due processi mediante socket stream.

In generale, all'interno di un particolare dominio di comunicazione sono disponibili vari *tipi* di *socket*. Il *tipo* denota lo stile di comunicazione: per esempio, mediante il tipo è possibile scegliere tra una comunicazione basata su un canale virtuale (*connection-oriented*) oppure una comunicazione senza canale (*connection-less*). In particolare i tipi di socket più comuni sono *stream* e *datagram*².

Le socket stream permettono una comunicazione attraverso un canale virtuale bidirezionale, mediante il quale i messaggi vengono recapitati al destinatario nello stesso ordine in cui sono stati spediti. La comunicazione mediante stream è affidabile, poiché il canale garantisce che ogni messaggio venga consegnato senza errori al destinatario (al limite in un tempo infinito). Questa proprietà solleva il programmatore dal dover gestire esplicitamente i problemi derivanti da traffico, come i malfunzionamenti e guasti che possono perturbare il corretto trasporto dei messaggi in una rete reale. In questo caso, quindi, la socket rappresenta un punto terminale del canale virtuale (vedi figura A.4).

Come si può osservare in figura, il canale virtuale consente una comunicazione simmetrica uno-a-uno. La presenza del canale, come si vedrà nel prossimo paragrafo, richiede che prima di iniziare lo scambio dei messaggi il canale venga preliminarmente creato mediante l'operazione di connessione (system call `connect`, vedi paragrafo A.2.3); analogamente, al termine di una sessione di comunicazione il canale dovrà essere chiuso.

La socket datagram, invece, rappresenta un punto di accesso alla rete mediante il quale un processo può comunicare con altri processi nella stessa rete. Socket di que-

² Va aggiunto che, oltre a stream e datagram, esistono altri tipi di socket, come *seqpacket* e *raw*, che, per motivi di spazio, non verranno trattati in questo testo. Il lettore interessato può approfondire questo argomento consultando [89].

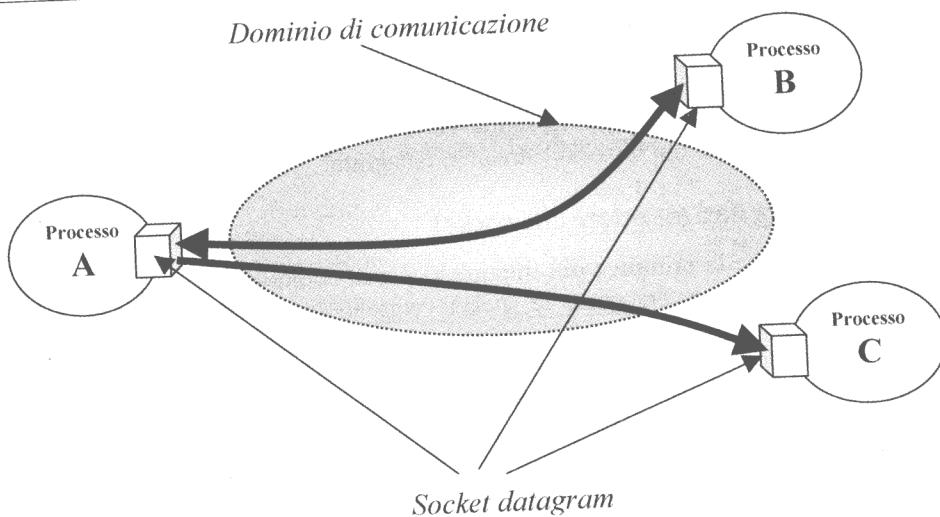


Figura A.5 Comunicazione tra processi mediante socket datagram.

sto tipo consentono una comunicazione senza connessione: in questo caso, quindi, i messaggi non utilizzano un canale virtuale e la stessa socket può essere utilizzata per lo scambio di messaggi con più di un processo (vedi figura A.5), permettendo quindi schemi di comunicazione asimmetrici.

L'assenza del canale virtuale non garantisce che i messaggi da un mittente M a un destinatario D vengano consegnati a D nello stesso ordine con cui sono stati spediti, né che ogni messaggio spedito arrivi sicuramente a destinazione.

Il tipo, quindi, stabilisce le caratteristiche della comunicazione; inoltre, se la comunicazione avviene attraverso una rete, il tipo della socket determina implicitamente i protocolli da utilizzare nel dominio prescelto. A questo proposito, nella tabella A.1, per ogni dominio (PF_UNIX o PF_INET) vengono indicati i tipi di socket utilizzabili ed eventualmente i protocolli di rete da adottare (nel caso del dominio PF_INET).

Tipo socket	PF_UNIX	PF_INET
Stream socket	Possibile	TCP
Datagram socket	Possibile	UDP
Raw socket	No	IP
Seq-pack socket	No	No

Tabella A.1 Associazione tra domini, tipi e protocolli.

Come si può notare in tabella, nel caso di Internet (dominio PF_INET), le socket stream sono supportate dal protocollo di trasporto TCP, mentre le datagram utilizzano il protocollo UDP.

Pertanto, facendo riferimento all'architettura di Internet (vedi paragrafo A.1.1), le socket possono essere collocate tra il livello di trasporto (TCP/UDP) e le applicazioni.

Nel seguito verranno illustrate le principali system call per la comunicazione tra processi mediante socket: a questo scopo verrà fatto implicitamente riferimento al dominio PF_INET e ai tipi di socket stream o datagram.

A.2.3 Strutture dati associate alle socket

Prima di esaminare le primitive più importanti per la comunicazione mediante socket, è necessario descrivere come le socket vengono rappresentate all'interno dei programmi C che le utilizzano.

Ogni processo che intende comunicare con altri processi deve creare una socket; grazie alla caratteristica di omogeneità delle risorse in ambiente Unix, ogni socket viene rappresentata localmente in modo omogeneo al file, mediante un *file descriptor*.

Inoltre è necessario che ogni socket possa essere individuata univocamente dai processi del dominio attraverso la specifica del suo indirizzo.

A questo scopo, l'indirizzo di ogni socket viene rappresentato da una struttura dati del tipo `sockaddr`, definita come segue:

```
struct sockaddr {
    sa_family_t sa_family;      /*dominio*/
    char sa_data[14];           /*indirizzo*/
};
```

L'indirizzo di una socket dipende dal dominio, poiché deve uniformarsi alle sue specifiche convenzioni di denotazione.

In particolare, per il dominio *Internet* (PF_INET) ogni socket è individuabile univocamente nella rete mediante una coppia di informazioni:

- l'indirizzo IP del nodo in cui risiede il processo proprietario della socket;
- il numero della porta a cui è associata la socket: ad ogni socket, infatti, deve essere attribuito un numero di porta³.

Nel caso del dominio PF_INET, quindi, la struttura si specializza come segue (vedi standard Posix.1g [46]):

```
struct sockaddr_in {
    sa_family_t      sin_family;     /*dominio*/
    in_port_t        sin_port;       /*porta*/
    struct in_addr   sin_addr;      /*ind. nodo*/
    char            sin_zero[8];    /*non usato*/
};
```

³ Ogni nodo Internet dispone di 2^{16} porte, ognuna individuata da un intero positivo; è necessario tenere presente che in UNIX le prime 1024 porte sono riservate a processi dell'utente *root*, e vengono tipicamente utilizzate per servizi standard, noti a priori. Alcuni esempi: la porta 25 è associata al servizio di posta elettronica, la porta 80 è associata al servizio http sul quale si basa www.

dove il tipo del campo `sin_addr`, che rappresenta l'indirizzo del nodo, è dichiarato nel modo seguente:

```
struct in_addr uint32_t s_addr ;
```

I tipi di dato non primitivi utilizzati nelle precedenti definizioni sono descritti nella tabella A.2, in cui vengono anche indicati i file header in cui sono contenute le rispettive dichiarazioni.

Tipo	Descrizione	Header file
<code>sa_family_t</code>	Tipo associato al dominio	<code><sys/types.h></code>
<code>in_port_t</code>	Tipo associato alla porta (di solito <code>unsigned int</code> a 16 bit)	<code><netinet/in.h></code>
<code>uint32_t</code>	<code>unsigned int</code> a 32 bit	<code><sys/types.h></code>
<code>sockaddr</code>	Tipo generico associato all'indirizzo di una socket	<code><sys/socket.h></code>
<code>sockaddr_in</code>	Tipo specifico per l'indirizzo di una socket socket nel dominio internet (IP v.4)	<code><sys/socket.h></code>

Tabella A.2 Descrizione dei tipi di dato.

A.2.4 System Call per l'uso delle socket

L'interfaccia di programmazione Unix BSD per la comunicazione tra processi offre un'insieme di system call C, che fanno riferimento alla socket come astrazione di comunicazione; queste funzioni sono visibili mediante l'inclusione dell'header file `<sys/socket.h>`.

Creazione e binding di socket Per creare una socket si deve utilizzare la system call:

```
int socket(int domain, int type, int protocol);
```

dove:

- il parametro `domain` rappresenta il dominio; ad esempio, la costante `PF_INET` indica il dominio Internet (IP v.4);
- `type` rappresenta il tipo della socket; in particolare, le costanti `SOCK_STREAM` e `SOCK_DGRAM` denotano rispettivamente socket stream e datagram;
- `protocol` indica il protocollo da utilizzare tra quelli associati al tipo (se si indica 0, si sceglie il protocollo di default).

La chiamata a `socket` crea una nuova socket con le caratteristiche specificate dai parametri e restituisce un intero che rappresenta il *socket descriptor* (cioè il file descriptor associato alla socket); pertanto la creazione di una socket alloca un nuovo elemento nella tabella dei file aperti del processo creatore.

Ad esempio, se si desidera creare una socket datagram nel dominio Internet, si può effettuare la seguente chiamata:

```
int sd;
sd= socket(PF_INET, SOCK_DGRAM, 0);
```

La socket creata mediante la chiamata a `socket` è visibile soltanto localmente al processo che l'ha creata: per poter consentire la visibilità della socket anche agli altri processi del dominio è necessario collegare la socket ad un indirizzo globale. Ciò è reso possibile mediante l'operazione di *binding*, effettuabile con la system call `bind`:

```
int bind(int sd, struct sockaddr *ind, int lun);
```

dove:

- il parametro `sd` rappresenta il socket descriptor associato alla socket;
- `ind` è il puntatore alla struttura che contiene l'indirizzo della socket (nel dominio Internet essa è una struttura del tipo `sockaddr_in`, vedi paragrafo A.2.3);
- `lun` è la lunghezza in byte della struttura puntata da `ind`.

Una volta compiuta l'operazione di binding, la socket è pronta per mettere in comunicazione il processo che la possiede con altri processi del dominio.

A seconda del tipo della socket, le modalità di uso e le primitive da utilizzare si differenziano: per questo motivo, nel seguito tratteremo separatamente le socket stream e le socket datagram.

System Call per l'uso di socket stream Le socket stream consentono l'interazione tra processi secondo il modello *connection-oriented*: i messaggi scambiati tra due processi viaggiano quindi attraverso un canale virtuale che garantisce una comunicazione affidabile (vedi figura A.4).

Prima di avviare una comunicazione tra due processi è quindi preliminarmente necessario instaurare il canale virtuale tra di essi.

In questo contesto, facendo riferimento al modello *client-server*, indicheremo come *client* il processo che prende l'iniziativa nella comunicazione; nel caso di comunicazione mediante canale virtuale esso coincide con il processo che richiede la creazione del canale virtuale. L'altro processo svolge invece il ruolo di *server* e, una volta ricevuta una richiesta di connessione, esso predispone un canale virtuale per la comunicazione con il cliente che ha inoltrato la richiesta.

Passiamo ora ad analizzare come concretamente si può realizzare la creazione del canale virtuale tra un processo cliente ed un processo servitore.

Il cliente, dopo aver creato la socket dedicata allo scambio di messaggi con il server, ne richiede la connessione ad una socket remota (presso il server) mediante la primitiva `connect`:

```
int connect(int sd, struct sockaddr *ind, int lun);
```

dove:

- il parametro `sd` rappresenta il socket descriptor associato alla socket locale;

- `ind` è il puntatore alla struttura che contiene l'indirizzo della socket remota del server;
- `lun` è la lunghezza in byte della struttura puntata da `ind`.

La chiamata a `connect` provoca la sospensione del processo fino a che il canale virtuale non è stato creato; se questa operazione ha successo il valore restituito da `connect` è 0. In caso di fallimento della connessione (ad esempio perché l'indirizzo della socket remota non è corretto), la `connect` restituisce un valore negativo.

Mediante la `connect` il client richiede al server il collegamento mediante canale virtuale: affinché ciò sia possibile, è necessario che il processo server sia disponibile a ricevere richieste di questo tipo. Per questo motivo il server, dopo aver creato una socket *stream* e aver effettuato su di essa l'operazione di *binding*, la dedica alla ricezione di eventuali richieste di connessione mediante la system call `listen`:

```
int listen(int sd, int dim)
```

dove:

- il parametro `sd` rappresenta il socket descriptor associato alla socket;
- `dim` è la dimensione di una coda che viene associata automaticamente alla socket, nella quale vengono depositate le richieste di connessione.

In questo modo la socket `sd` assume il ruolo di *socket di ascolto*, la cui funzione è soltanto quella di raccogliere le richieste di connessione dai clienti remoti.

Ogni richiesta di connessione contenuta nella coda associata alla socket di ascolto può essere servita dal processo server mediante la primitiva `accept`:

```
int accept(int sd, struct sockaddr *ind, int *lun);
```

dove:

- il parametro `sd` rappresenta il socket descriptor associato alla socket di ascolto;
- `ind` è il puntatore alla struttura nella quale viene memorizzato l'indirizzo della socket remota del client.
- `lun` è il puntatore a una variabile nella quale viene memorizzata la lunghezza dell'indirizzo del client.

Se la coda delle richieste contiene almeno un messaggio, la `accept` estrae da essa la prima richiesta e crea una nuova socket che rappresenta il terminale del canale virtuale di comunicazione dal lato server, restituendone il socket descriptor; l'effetto di tale chiamata sarà pertanto l'effettiva creazione del canale virtuale tra client e server ed il risveglio del processo client, sospeso sulla chiamata a `connect`.

Invece, se la coda delle richieste di connessione associata a `sd` è vuota, la `accept` determina la sospensione del processo server, finché non viene ricevuta una nuova richiesta di connessione.

Riassumendo, se due processi (client e server) intendono comunicare mediante socket *stream*, la necessaria predisposizione del canale virtuale per lo scambio dei messaggi si realizza attraverso il protocollo di sincronizzazione mostrato in figura A.6.

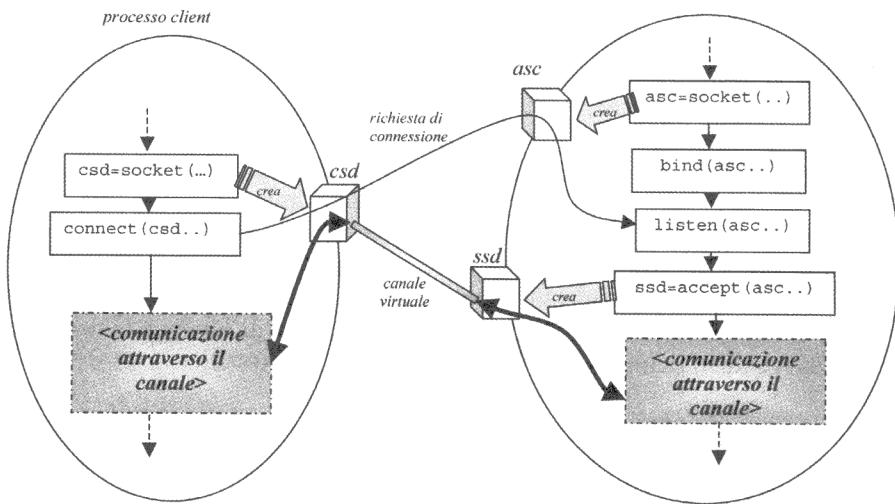


Figura A.6 Protocollo di creazione del canale virtuale.

Una volta creato il canale virtuale i due processi agli estremi di esso (client e server) possono scambiarsi messaggi in entrambe le direzioni.

In assoluta omogeneità con i file, analogamente al caso delle *pipe* (vedi 7.6.3), le primitive di comunicazione sono le stesse disponibili per l'accesso ai file: *write* e *read*⁴.

In particolare, la spedizione di un messaggio viene richiesta mediante la system call *write* (vedi 7.5.5), alla quale si fornisce il socket descriptor associato alla socket locale. Ad esempio, facendo riferimento alla figura A.4, il server può spedire un messaggio al client nel modo seguente:

```

int asc, ssd;
char msg[6]=34Ciao!34;

<creazione socket e apertura del canale>;
write(ssd, msg, 6);

```

È importante notare che, nel caso di comunicazione mediante stream, poiché lo schema di comunicazione è punto-a-punto, non è necessario ad ogni invio di messaggio specificare l'indirizzo del processo destinatario; è sufficiente indicare soltanto il terminale locale del canale virtuale.

Analogamente, il processo client, destinatario del messaggio, può effettuare la ricezione come segue:

⁴ Oltre a *read/write* sono disponibili anche le primitive di comunicazione *sendv* e *recv*, che non verranno trattate per motivi di spazio. Consultare [89] per un eventuale approfondimento.

```

int csd;
char msg[6];
...
<creazione socket e apertura del canale>;
read(ssd, msg, 6);
...

```

Il default prevede che le socket siano *bloccanti*: questo significa che se un destinatario esegue una `read` quando non vi è alcuna informazione all'interno del canale, il processo viene sospeso, in attesa del primo messaggio. Va inoltre ricordato che il protocollo TCP non prevede alcun separatore tra un messaggio e il successivo (vedi paragrafo A.1.1): questo implica che il contenuto del canale è visto come una sequenza non strutturata di byte. È necessario, quindi, che i due processi mittente e destinatario si accordino sulle caratteristiche dei messaggi, per esempio assumendo messaggi di lunghezza costante prefissata.

Al termine di una sessione di comunicazione, la connessione può essere chiusa mediante la primitiva `shutdown()`, la cui sintassi è la seguente:

```
int shutdown (int sd, int modo);
```

dove:

- il parametro `sd` rappresenta il socket descriptor associato al terminale del canale;
- `modo` esprime la modalità di chiusura; infatti è possibile chiudere il canale soltanto in una direzione (valore 0 per la ricezione, valore 1 per la trasmissione), oppure in entrambe (valore di modo uguale a 2).

Se la chiusura avviene in entrambe le direzioni, la socket `sd` viene eliminata, e l'elemento corrispondente nella tabella dei file aperti viene liberato.

Il seguente programma mostra un semplice esempio di due processi client e server che si scambiano messaggi attraverso socket stream.

```

Processo client:
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

struct sockaddr_in *D, *server;
char msg[2000];
int sd, l;

int main()
{
    sd=socket(AF_INET, SOCK_STREAM, 0);
    <inizializzazione indirizzo server>;
    /* richiesta di connessione*/
    connect(sd,&server,1);

    <preparazione messaggio msg>;

```

```

        write(sd, msg, 2000);           /* invio messaggio */
        read(sd, ris, 2000);          /* ricezione risposta */

        shutdown(sd, 2);              /* chiusura connessione */
    }

Processo server:
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

struct sockaddr_in *M, *mio;
char msg[BUFFERSIZE], ris[2000];
int asc, l, sd, addrlen;
int main()
{
    asc=socket(AF_INET,SOCK_STREAM,0);

    <inizializz. mio indirizzo>;
    l=sizeof(struct sockaddr_in);
    bind(asc,&mio,l);             /* pubblicazione indirizzo */
    listen(asc, 100);            /* creazione della coda richieste */

    sd=accept(asc, M, &addrlen);   /* apertura canale */
    read (sd, msg, 2000);         /* ricezione messaggio */

    <calcolo risposta ris>;

    write (sd, ris, 2000);        /* invio risposta */
    ...
}

```

System Call per l'uso di socket datagram Passiamo ora ad analizzare gli strumenti e le modalità di comunicazione per socket di tipo datagram.

In questo caso, poiché la comunicazione è senza connessione, ogni processo, dopo aver creato la socket ed effettuato l'operazione di binding, può utilizzarla per la comunicazione con altri processi.

A questo scopo le system call specifiche di spedizione e ricezione dei messaggi sono `sendto` e `recvfrom`.

In particolare, la primitiva di spedizione osserva la seguente sintassi:

```
int sendto(int sd, char *msg, int lun, int flag,
           struct sockaddr *D, int lund);
```

dove:

- il parametro `sd` rappresenta il socket descriptor associato alla socket locale;
- `msg` è il puntatore al messaggio da inviare;
- `lun` è la lunghezza del messaggio;
- `flag` è un intero mediante il quale è possibile specificare opzioni particolari sul trasporto del messaggio;

- D è il puntatore alla struttura che contiene l'indirizzo della socket (nel dominio Internet essa è una struttura del tipo `sockaddr_in`, vedi paragrafo A.2.1) del destinatario;
- lunD è la lunghezza della struttura puntata da D.

La primitiva di ricezione prevede una sintassi analoga:

```
int recvfrom(int sd, char *msg, int lun, int flag,
             struct sockaddr *M, int *lunM);
```

dove:

- il parametro `sd` rappresenta il socket descriptor associato alla socket locale;
- `msg` è il puntatore alla variabile a cui assegnare il messaggio ricevuto;
- `lun` è la lunghezza del messaggio;
- `flag` è un intero mediante il quale è possibile specificare opzioni particolari sul trasporto del messaggio;
- D è il puntatore alla struttura nella quale viene memorizzato l'indirizzo della socket del mittente;
- `lun` è la lunghezza della struttura puntata da D.

La `recvfrom` può bloccare il processo destinatario se il messaggio non è ancora disponibile; a differenza della comunicazione mediante canale virtuale, in questo caso, i confini dei messaggi vengono mantenuti e il processo quindi viene sospeso fino a che l'intero messaggio non gli è stato recapitato.

Come si può osservare, a differenza del caso di comunicazione con connessione, le primitive di comunicazione prevedono parametri specifici per gli indirizzi dei partner nella comunicazione: nell'invio è necessario specificare l'indirizzo del destinatario; nella ricezione è previsto che, assieme al messaggio la `recvfrom` fornisca anche l'indirizzo del mittente.

Per chiudere una socket datagram è possibile usare ancora la system call `shutdown`, oppure la system call `close` (vedi 7.6.5).

Nel seguente esempio viene presentata una semplice applicazione di due processi che dialogano mediante socket datagram.

Processo client:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

struct sockaddr_in *D, *mio;
char msg[2000], ris[BUFFERSIZE];
int sd, l, addrlen;
int main()
{
    sd=socket(AF_INET, SOCK_DGRAM, 0);
    <inizializz. mio indirizzo>;
    l=sizeof(struct sockaddr_in);
    bind (sd,&mio,l);
```

```
/* invio messaggio al server: */

sendto (sd, msg, 2000, 0, D,1);
/* ricezione risposta: */
recvfrom (sd, ris, BUFFERSIZE,0, D, &addrlen);
...
close(sd);
}

Processo server:
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

struct sockaddr_in *M, *mio;
char msg[BUFFERSIZE], ris[2000];
int sd, l, addrlen;
int main()
{
    sd=socket(AF_INET,SOCK_DGRAM,0);
    <inizializz. mio indirizzo>;
    l=sizeof(struct sockaddr_in);
    bind (sd,&mio,1);
    addrlen=1;
    /* ricezione messaggio:*/
    recvfrom (sd, msg, BUFFERSIZE,0, M, &addrlen);
    <calcolo risposta ris>;
    /*invio risposta: */
    sendto (sd, ris, 2000, 0, M, addrlen);
    ...
    close(sd);
}
```

A.3 Un esempio: esecuzione remota di comandi

A conclusione della trattazione di questa appendice, introduciamo ora un esempio completo di utilizzo delle socket stream nel dominio Internet. In particolare, si vuole mostrare una possibile realizzazione di un'applicazione client-server che consenta ad ogni processo client l'esecuzione remota (cioè, sul nodo del server) di semplici comandi, e la visualizzazione locale (cioè sul nodo del client) dell'output prodotto dai comandi eseguiti.

Come si è visto nel paragrafo A.2, la prima scelta importante da compiere riguarda il tipo di socket da usare per la comunicazione tra i processi. In questo caso, poiché si desidera visualizzare su dispositivi locali l'output di comandi remoti, è necessario scegliere socket di tipo stream, in quanto si è visto che esse possono essere utilizzate con gli stessi strumenti disponibili per l'accesso a file. In questo modo, quindi, sfruttando i tradizionali meccanismi di ridirezione di Unix, possiamo convogliare l'output dei comandi eseguiti dal server sul canale virtuale, trasferendo quindi tali informazioni al client in modo completamente trasparente.

Processo cliente Ogni processo cliente dovrà prendere l'iniziativa nell'interazione con il server, richiedendo la creazione della connessione e successivamente inviando la propria richiesta. Tale richiesta è rappresentata da un messaggio contenente il nome di un comando (per semplicità, senza argomenti).

Successivamente il cliente si pone in attesa della risposta, che è rappresentata dall'output del comando eseguito dal server: ogni byte ricevuto attraverso il canale viene copiato dal client sul dispositivo di standard output.

Pertanto il programma eseguito dal client potrebbe essere strutturato come segue:

```
/*processo client: */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    int sock, retval, i;
    char mess[10], ris[1000];
    /*indirizzo socket remota*/
    struct sockaddr_in rem_ind;

    /*Preparazione indirizzo del server*/
    rem_ind.sin_family = PF_INET;           /*dominio*/
    /* ad esempio: se l'indirizzo internet del
    rem_ind.sin_addr.s_addr=inet_addr("34137.204.57.115"34);
    rem_ind.sin_port = 22375;                /*porta del server*/

    /*Preparazione del messaggio*/
    strcpy(mess, argv[1]);

    /*Creazione della socket*/
    sock=socket(PF_INET, SOCK_STREAM, 0);
    connect(sock, &rem_ind, sizeof(struct sockaddr_in));
    write(sock,mess, 10);                  /*invio messaggio*/
    while (i=read(sock,ris, 1)>0)          /*ric. risposta*/
        write(1,ris,i);                   /*stampa su stdout*/

    /*chiusura collegamento*/
    shutdown(sock,2);
}
```

Processo server Ogni processo server ha tipicamente una struttura ciclica: ad ogni iterazione viene servita una particolare richiesta di connessione.

Il processo server può, in generale, servire le richieste in modo sequenziale (viene servita una richiesta per volta) o concorrente (vengono servite più richieste in parallelo); per semplicità, in questo esempio si è scelto di realizzare il server in modo sequenziale, affidando a un nuovo processo alla volta l'esecuzione di un comando richiesto da un particolare cliente.

Prima di entrare nel ciclo di servizio, il server deve compiere le operazioni preliminari per consentire l'interazione con i clienti: in particolare, esso deve inizialmente creare la socket di ascolto e pubblicarne l'indirizzo mediante l'operazione di binding; successivamente, con la primitiva listen associerà alla socket di ascolto una coda nella quale verranno inserite le eventuali richieste di connessione. A questo punto, il server è pronto per servire ciclicamente ogni nuova richiesta da parte di clienti remoti. In particolare, per ogni richiesta, dopo aver instaurato una nuova connessione dedicata ad essa (mediante accept), viene creato un nuovo processo che riceve dal canale il nome del comando da eseguire e, successivamente, dopo aver opportunamente ridirezionato la socket di comunicazione sullo standard output, passa ad eseguire il comando mediante una system call della famiglia exec (vedi paragrafo 7.4.3). Al termine dell'esecuzione del comando, il processo server chiude la socket e passa al servizio di nuove richieste.

Quindi, il codice del programma eseguito dal server potrebbe essere il seguente:

```
/*processo server:*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

int main()
{
    char comando[20];
    int newsock, sock, figlio, status, i;
    struct sockaddr_in mio_ind;
    /* Creazione socket di ascolto */
    sock=socket(PF_INET, SOCK_STREAM, 0);

    /* Preparazione indirizzo socket : */
    mio_ind.sin_family=PF_INET;
    mio_ind.sin_addr.s_addr = INADDR_ANY;
    mio_ind.sin_port=22375;
    /* Binding:*/
    if(bind(sock, &mio_ind, sizeof(struct sockaddr_in))<0) {
        perror(34bind34);
        exit(1);
    }
    /* sock riceverà le richieste di connessione: */
    /*creaz. coda richieste connessione*/
    listen(sock, 5);

    for (;;) {                      /* ciclo di servizio */
        /* estrazione nuova richiesta dalla coda: */
        newsock=accept(sock,(struct sockaddr_in *) 0, 0);
        if ((figlio=fork())==0) { /* figlio */
            close(sock);
            /* ricez. comando*/
            read(newsock, comando, 10);
            /*ridirezione dell'output sulla socket: */
            if (execve(comando, NULL, NULL) < 0)
                perror(execve);
        }
    }
}
```

```

        close(1);
        dup(newsock);
        /* esecuzione comando: */
        if((i=execlp(comando, comando, (char )0))<0) {
            write(1,"errore", 7);
            exit(-1);
        }
    } /* figlio */
    else { /* padre */
        wait(&status);           /* attesa figlio */
        shutdown(newsock, 1); /* chiusura socket: */
        close(newsock);
    }
}
close(sock);
}

```

A.4 Sommario

In questa appendice sono state introdotte alcune problematiche relative all’interazione tra processi in un sistema distribuito. Preliminarmente sono stati forniti alcuni concetti di base, introducendo il modello di riferimento per l’architettura delle reti definito dal modello OSI e i protocolli specifici della rete Internet: TCP, UDP e IP.

Successivamente sono stati analizzati i meccanismi di base offerti dal sistema Unix per l’interazione tra processi in ambito distribuito, incentrati sul concetto di socket. La socket rappresenta il mezzo di comunicazione che ogni processo può impiegare per la comunicazione con altri processi; pur essendo disponibili in generale vari tipi di socket, in questa sede la trattazione è stata limitata ai tipi stream e datagram.

In particolare, sono stati presentate le system call specifiche per l’utilizzo di questi tipi di socket ed infine, è stato presentato un esempio completo relativo a una semplice applicazione client-server basata su socket di tipo stream.

A.5 Note bibliografiche

L’inevitabile sintesi della trattazione relativa ai concetti di base delle reti può essere compensata consultando [94]. Per approfondimenti relativi al modello OSI si rimanda al documento ufficiale [49]. Maggiori dettagli relativi all’architettura TCP/IP, potranno essere ottenuti consultando [88].

Per maggiori dettagli sull’uso delle socket si rimanda il lettore interessato al testo [89].

B

Multithreading in Java

Java [34] è un linguaggio originariamente sviluppato presso la Sun Microsystems con lo scopo di fornire uno strumento linguistico per la programmazione di dispositivi di largo consumo (*consumer devices*) come ad esempio i telefoni cellulari. Successivamente, con lo sviluppo del web, Java ha mostrato tutte la proprie potenzialità come linguaggio da utilizzare per lo sviluppo di applicazioni web.

Java è un linguaggio concorrente nel quale cioè sono presenti come costrutti linguistici molti dei concetti esaminati nei precedenti capitoli del testo, in particolare i concetti di thread, di applicazioni multithreading, di interazioni e di condivisione di dati fra threads.

Lo scopo di questa appendice è duplice: da un lato esemplificare i precedenti concetti utilizzando uno strumento linguistico adatto a questo scopo, dall'altro fornire un raccordo fra i contenuti del testo e quelli di un secondo volume relativo all'approfondimento degli argomenti dei sistemi operativi e, più in particolare di quelli relativi alle architetture di sistemi più complesse, tipiche dei sistemi distribuiti, e alle problematiche inerenti la programmazione di sistema, tipiche della programmazione concorrente e distribuita.

Chiaramente quest'appendice non vuole essere una introduzione a Java, argomento questo che esula gli scopi del testo. Viene quindi data per scontata una generica conoscenza del linguaggio oltre che dei principi della programmazione orientata agli oggetti. In particolare viene data per scontata una generale conoscenza dei principali concetti della programmazione *object oriented* (classi, oggetti, encapsulamento, meccanismi di astrazione, ereditarietà, polimorfismo, gestione delle eccezioni, garbage collection), e di come tali concetti sono stati calati nel linguaggio Java. Per approfondire i principi del paradigma orientato agli oggetti si può fare riferimento a [10]. Per quanto riguarda un buon tutorial su Java si può fare riferimento a [15] che è anche disponibile su web sul sito della Sun [52].

B.1 Ambiente di sviluppo e ambiente di esecuzione

Java non è soltanto un linguaggio di programmazione, ma una tecnologia che consiste di un ambiente per lo sviluppo di nuove applicazioni (*Java Development Environment*) e dell'ambiente per la loro esecuzione (*Java Run-Time Environment*), spesso indicato anche come *Java Platform*.

L'ambiente di sviluppo consiste nel linguaggio Java e nel relativo compilatore che traduce il programma in un linguaggio intermedio (*bytecode*) indipendente dalla particolare architettura della macchina su cui deve girare il programma e dal particolare sistema operativo presente sulla stessa macchina. L'ambiente di esecuzione consiste in una libreria di packages Java contenenti alcune classi di utilità offerte al programmatore già tradotte in linguaggio intermedio (Java API), nel linker-loader per collegare il codice prodotto dal compilatore e quello delle classi di libreria utilizzate e caricare il risultato in memoria, e nell'interprete del linguaggio intermedio necessario per eseguire il programma sulla specifica macchina. Tale interprete implementa la *Java Virtual Machine (JVM)* che, come dice il nome, rappresenta una macchina astratta indipendente dalla specifica macchina ospite su cui il programma deve girare.

Un programma Java è costituito da un insieme di classi. Una di queste deve contenere il metodo `main()` (un metodo corrisponde a una funzione in C o in C++). Il metodo `main()` deve essere definito come:

```
public static void main (String args[])
```

Il file con il codice sorgente contenente la classe in cui compare il metodo `main()` deve avere lo stesso nome di questa classe ed avere l'estensione `.java` (per esempio: `esempio.java`). Il compilatore traduce il programma sorgente nel linguaggio intermedio producendo un file con lo stesso nome ed estensione `.class(esempio.class)`.

La libreria Java API, come già detto, consiste in un insieme di packages contenenti classi già tradotte in linguaggio intermedio (il package standard del linguaggio: `java.lang`, il package con le classi di supporto per la grafica: `java.awt`, il package con le classi per la gestione dell'I/O: `java.io`, ed altri ancora). Un elenco completo di tutti i packages si può trovare sul sito [52].

La *Java Virtual Machine* contiene il linker-loader per caricare le classi da eseguire e l'interprete che implementa sulla macchina ospite la macchina astratta il cui linguaggio macchina coincide con il *bytecode* (vedi figura B.1).

Il concetto di macchina virtuale Java rappresenta un'astrazione che può essere implementata in vari modi. Il più ovvio e più spesso utilizzato è proprio quello di realizzare un interprete del *bytecode* sopra la macchina ospite sfruttando le funzionalità del sistema operativo che su essa opera, ma in modo tale da garantire la completa indipendenza dei programmi Java dalla specifica architettura e dallo specifico sistema operativo (vedi figura B.2). Si può anche ipotizzare una successiva fase di compilazione che traduca il *bytecode* direttamente sulla macchina ospite o, addirittura, di realizzare in hardware la macchina che interpreti direttamente il *bytecode*.

Come già detto, Java è un linguaggio concorrente che consente la definizione di più threads all'interno dello stesso programma. Ciò ovviamente implica che la *Java Virtual Machine* è una macchina multiprogrammata, nel senso che fornisce il sup-

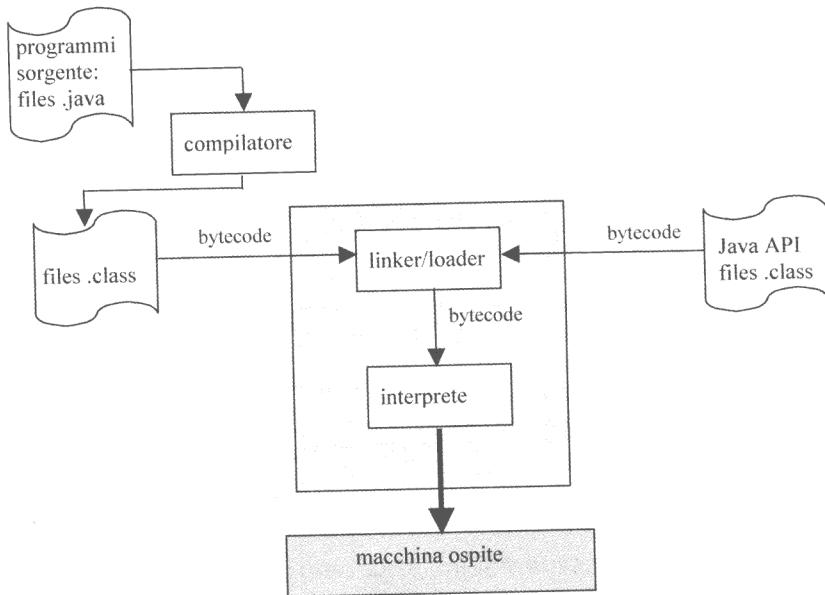


Figura B.1 Java Virtual Machine.

porto all'esecuzione concorrente di più threads. Nel secondo capitolo (paragrafo 2.10) è stato messo in evidenza che non tutti i sistemi operativi multiprogrammati forniscono il supporto ai threads. Ci sono casi in cui la gestione dei threads può avvenire a *livello utente*, in altri avviene a *livello di nucleo*. La realizzazione della Java Virtual Machine astrae da questi dettagli implementando i threads Java a livello utente o a livello di nucleo in base alle caratteristiche del sistema operativo ospite, ma rende i programmi del tutto indipendenti da questi dettagli.

B.2 I threads in Java

Ogni programma Java contiene almeno un singolo thread, corrispondente all'esecuzione del metodo `main()` sulla JVM. È possibile però creare dinamicamente ulteriori threads attivando le loro esecuzioni concorrentemente all'interno del programma. I threads Java sono oggetti che derivano dalla classe `Thread` fornita dal package `java.lang`.

Esistono due diversi modi in Java per creare nuovi threads:

- Il primo consiste nel derivare nuove classi dalla classe `Thread` mediante la normale tecnica di estensione tipica dell'ereditarietà,
- Il secondo nel definire una nuova classe che implementa l'interfaccia `Runnable` (anch'essa offerta dal package `java.lang`).

Esaminiamo adesso il primo dei due metodi e, successivamente l'altro.

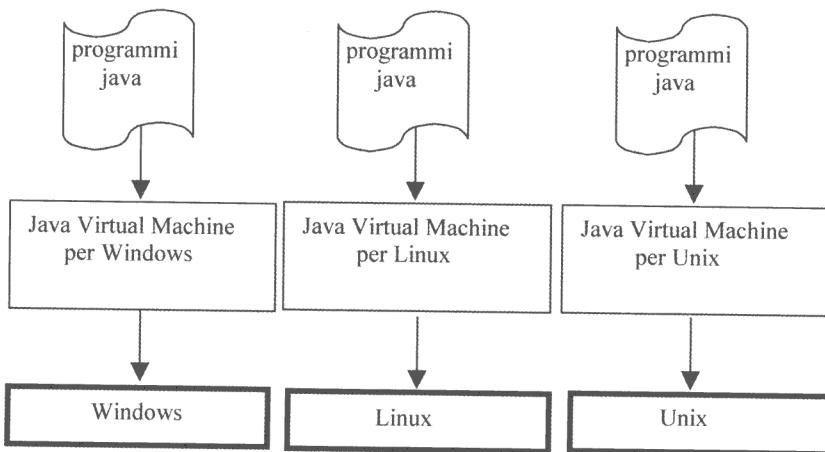


Figura B.2 Indipendenza di java dalla macchina ospite.

B.2.1 Creazione di threads mediante estensione della classe Thread

La classe di libreria `Thread` definisce e implementa i threads Java fornendo un insieme di metodi necessari per il controllo delle loro esecuzioni. Uno di questi metodi, il metodo `run()`, definisce ciò che ogni oggetto della classe eseguirà come thread separato (una qualunque sequenza di statements Java), concorrentemente con gli altri threads del programma. La classe `Thread` implementa un thread generico che, per default, non fa rigorosamente niente, cioè l'implementazione del suo metodo `run()` è vuota. Per creare quindi qualcosa di utile si può definire una sottoclasse di `Thread` ridefinendone (*override*) il metodo `run()` in modo tale da fargli eseguire ciò che è richiesto dal programma. Nella figura B.3 viene illustrato questo approccio. La classe `AltriThreads` (estensione di `Thread`) implementa i nuovi threads ridefinendo il metodo `run()`. La successiva classe è quella che fornisce il `main` nel quale viene creato il thread `t1` come oggetto derivato dalla classe `Thread`. Da notare che la creazione dell'oggetto `t1` nel primo statement del metodo `main` si limita a creare un thread vuoto, cioè senza nessuna risorsa allocata al thread (senza un processore virtuale in grado di eseguirlo). Come vedremo in un successivo paragrafo esaminando il grafo di stato di un thread Java, per attivare il thread assegnandogli un processore virtuale, e quindi per farlo transitare in stato pronto per l'esecuzione, è necessario eseguire il metodo `start()` come indicato nel secondo statement del `main`. È il metodo `start()` che invoca il metodo `run()` attivando l'esecuzione del nuovo thread. Il metodo `run()` non può essere chiamato direttamente ma solo tramite lo `start()`.

Nell'esempio di figura B.3 la JVM gestisce due thread concorrenti: il thread principale associato al `main` e il thread `t1`, creato dinamicamente dal precedente con l'esecuzione dello statement `t1.start()` che lancia, in concorrenza, l'esecuzione del metodo `run()` del nuovo thread.

```

class AltriThreads extends Thread {
    public void run() {
        <corpo del programma eseguito>
        <da ogni thread di questa classe>;
    }
}

public class PrimoEsempioConDueThreads {
    public static void main (String[] args) {
        AltriThreads t1 = new AltriThreads();
        t1.start();
        <resto del programma eseguito dal thread main>;
    }
}

```

Figura B.3 Creazione di un thread mediante estensione della classe Thread.

B.2.2 Creazione di threads mediante implementazione dell'interfaccia Runnable

Un diverso modo di fornire il metodo `run()` ad un thread, rispetto a quello visto precedentemente, è quello di definire una nuova classe che implementa l'interfaccia `Runnable` che è così definita:

```
public interface Runnable public abstract void run();
```

da cui ovviamente risulta che ogni classe che implementa questa interfaccia deve definire il metodo `run()`. Nella successiva figura B.4 viene illustrato questo approccio.

La nuova classe che implementa `Runnable` non estende `Thread` e quindi, in questo caso, non ha accesso al metodo `start()` necessario per attivare un nuovo thread. È quindi obbligatorio anche ora creare un oggetto della classe `Thread` (`t1` nell'esempio di figura, nel secondo statement del metodo `main`). Il criterio con cui viene specificato il metodo `run()` che il nuovo thread dovrà eseguire è quello di creare un oggetto `Runnable`, come nel primo statement del `main` nel quale l'oggetto esecutore è creato come istanza della classe `AltriThreads` che implementa il nuovo metodo `run()`. Tale oggetto è poi passato come parametro al costruttore del thread `t1` nel secondo statement del `main`. In questo modo, quando `t1` è attivato mediante il metodo `start()` (terzo statement), inizia l'esecuzione del metodo `run()` dell'oggetto `Runnable`.

La necessità di avere in Java anche questo secondo criterio di creazione di un thread è dovuto all'impossibilità di derivazione multipla di una classe. Per cui, ad esempio, se una classe è già derivata da un'altra non è possibile estendere contemporaneamente anche la classe `Thread`. In questo caso si può allora consentire alla classe derivata di implementare contemporaneamente l'interfaccia `Runnable`. Negli esempi che seguono, per semplicità, faremo riferimento esclusivamente al primo modo (illustrato in figura B.3).

```

class AltriThreads implements Runnable {
    public void run() {
        <corpo del programma eseguito>
        <da ogni thread di questa classe>;
    }
}

public class SecondoEsempioConDueThreads {
    public static void main (String[] args) {
        Runnable esecutore= new AltriThreads();
        Threads t1 = new Threads(esecutore);
        t1.start();
        <resto del programma eseguito dal thread main>;
    }
}

```

Figura B.4 Creazione di un thread mediante implementazione di `Runnable`.

B.2.3 Grafo di stato dei threads Java

Gli stati possibili nei quali può trovarsi un thread sono quattro come indicato nella figura B.5. Lo stato *nuovo* (*new*) è quello in cui si trova un thread dopo che è stato creato un oggetto per il thread mediante lo statement `new`. Quando il thread si trova in questo stato non può eseguire poiché non è stato ancora attivato. L'attivazione avviene invocando il metodo `start()` che, a sua volta, chiama il metodo `run()` per iniziare l'esecuzione del thread. Lo stato in cui si trova il thread dopo la sua attivazione viene indicato come *runnable*. Quando termina l'esecuzione del metodo `run()` il thread commuta il proprio stato in *terminato* (*dead*). Durante la sua esecuzione il thread può sospendersi per varie cause entrando in stato *bloccato* (*blocked*). Una prima causa di blocco è dovuta a una operazione di I/O che implica la sospensione del thread fino al termine dell'operazione. Un'altra causa di sospensione corrisponde all'invocazione del metodo `sleep` a cui viene passato un parametro che corrisponde al numero di millisecondi che devono trascorrere prima che il thread diventi di nuovo *runnable*. In pratica tale metodo consente ad un thread di ritardare la propria esecuzione per un periodo di tempo programmato. Un'ulteriore causa di blocco corrisponde all'esecuzione del metodo `wait` per attendere che una condizione necessaria per la prosecuzione del thread sia verificata. Vedremo in dettaglio nei successivi paragrafi l'uso di questo metodo.

Per ciascuna delle cause di sospensione di un thread esiste un evento la cui occorrenza rende di nuovo attivo il thread commutandolo in stato *runnable*. Ad esempio se il thread si era sospeso per aver attivato un'operazione di I/O, tale evento corrisponde alla terminazione dell'operazione. Analogamente, se il thread si era spontaneamente sospeso invocando `sleep`, l'evento corrisponde al completamento del periodo di attesa. Se, infine, il thread si era sospeso eseguendo una `wait` l'evento corrisponde all'esecuzione del metodo `notify` (o `notifyAll`) da parte di un altro thread, come vedremo nel seguito.

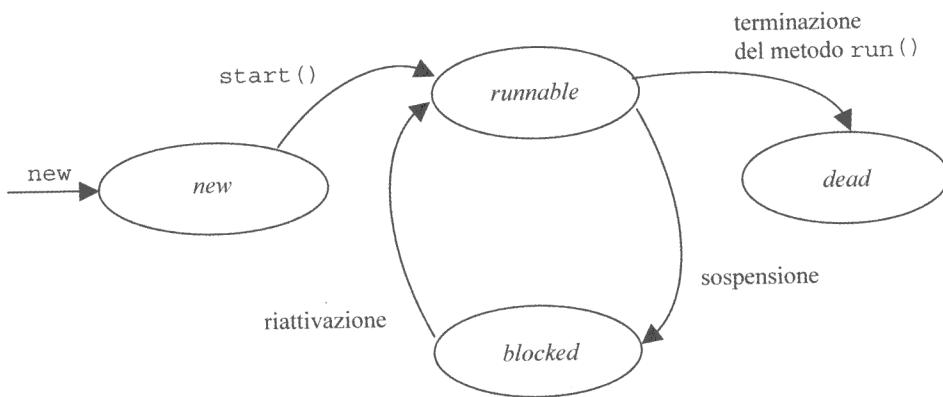


Figura B.5 Grafo di stato di un thread.

Lo stato *runnable* identifica che il thread è attivo e quindi che la JVM lo può eseguire. Però, come indicato nel capitolo 2, lo stato attivo (vedi figure 2.2 e 2.3) viene suddiviso nei due stati di *pronto* e di *esecuzione* per tenere conto che la macchina reale ha un numero di processori fisici inferiore a quello dei processori virtuali. Quindi, tenendo conto di questa osservazione, il grafo di stato di un thread corrisponde esattamente al grafo a 5 stati illustrato nel capitolo 2 (vedi figura 2.4).

È possibile verificare lo stato di un thread, almeno in parte, mediante il metodo `isAlive` che restituisce un valore booleano. In particolare restituisce il valore `false` se il thread è terminato e `true` in caso contrario.

B.2.4 Priorità e algoritmi di scheduling dei threads Java

All'interno della JVM i threads sono schedulati mediante un algoritmo che opera su base prioritaria con diritto di prelazione (*preemptive priority scheduling*) con priorità fisse.

Ogni thread ha una propria priorità rappresentata mediante un valore intero compreso fra due valori limite: `MIN_PRIORITY` e `MAX_PRIORITY` (due costanti definite all'interno della classe `Thread`). A intero maggiore corrisponde una maggiore priorità. Ogni thread all'atto della creazione, eredita la priorità dal thread che lo ha creato. È però possibile modificare tale valore mediante il metodo `setPriority`.

L'algoritmo di scheduling sceglie fra tutti i threads pronti per l'esecuzione il thread con la priorità più alta e, nel caso di threads con la stessa priorità, in subordine effettua la scelta in ordine FIFO.

La JVM esegue l'algoritmo di scheduling in uno dei seguenti due casi:

1. quando il thread correntemente in esecuzione esce dallo stato *runnable* (perché si sospende o termina);
2. nel caso in cui diventi *runnable* un thread a priorità più alta (*preemption*) rispetto a quella del thread in esecuzione.

```

public void run() {
    while (true) {
        <sequenza di statements eseguiti dal thread...>;
        <...fra quando entra in esecuzione e...>;
        <...quando cede il controllo della CPU>;
        <(approssimazione di un quanto di tempo)>;
        Thread.yield();
    }
}

```

Figura B.6 Uso del metodo `yield()`.

La JVM non fornisce nessuna indicazione rispetto ad un'eventuale assegnazione della CPU *a quanti di tempo* (*time slice*) tipica dei sistemi operativi che adottano algoritmi di scheduling *round robin* (vedi capitolo 2). Se la JVM è implementata su sistemi che adottano questo criterio (ad esempio Windows), i threads di uguale priorità vengono gestiti con tecnica Round Robin e non semplicemente FIFO. Quindi, in questi casi, il thread in esecuzione mantiene il controllo della CPU fino a quando si verifica uno dei due eventi indicati precedentemente o, al massimo, fino alla scadenza del quanto di tempo assegnato.

Se la JVM viene implementata su un sistema che non adotta la tecnica a partizione di tempo è comunque possibile simulare a programma tale comportamento mediante il metodo `yield()` che il thread in esecuzione può invocare per cedere volontariamente la CPU e richiamare lo scheduler al fine di assegnarla ad un altro thread della stessa priorità (vedi figura B.6).

L'uso del metodo `yield()` visto precedentemente viene spesso indicato con il termine di *cooperative multithreading*.

La trasparenza della JVM rispetto alla gestione dei quanti di tempo ha costituito un potenziale problema per quanto riguarda la portabilità di alcune applicazioni su sistemi operativi che adottano diversi criteri di schedulazione.

B.3 Sincronizzazione in Java

Come è stato mostrato nel capitolo 2 (paragrafo 2.10), i threads di un'applicazione condividono lo *spazio di indirizzamento*. è quindi naturale che le soluzioni ai problemi di interazione tra threads siano strutturate seguendo il *modello ad ambiente globale* (vedi capitolo 3 - paragrafo 3.1). Tale modello, in particolare, prevede che ogni tipo di interazione tra threads avvenga tramite la memoria comune: quindi, nel caso di applicazioni Java realizzate seguendo il paradigma *object-oriented*, tramite oggetti comuni.

Sempre nel paragrafo 3.1 è stato mostrato che le interazioni possono essere di due tipi diversi: di tipo *competitivo*, per l'accesso a oggetti comuni, e di tipo *cooperativo*, per lo scambio di informazioni. Per risolvere i problemi inerenti i due tipi di interazione sono necessari due diversi tipi di meccanismi di sincronizzazione, indicati con i nomi di *sincronizzazione indiretta* (o di *mutua esclusione*) e di *sincroniz-*

zazione diretta rispettivamente. In Java tali meccanismi sono stati introdotti a livello di linguaggio e sono supportati dalla JVM. Essi corrispondono, rispettivamente, al meccanismo degli “*object-locks*” e al meccanismo “*wait-notify*”.

B.3.1 Accessi esclusivi a un oggetto e sezioni critiche

Nel capitolo 3 (paragrafo 3.2) è stato messo in evidenza che quando due o più processi (o threads nel caso di Java) operano concorrentemente su variabili comuni possono nascere problemi di *corse critiche*, e cioè la possibilità che, per certi rapporti di velocità fra le esecuzioni di due threads, le operazioni eseguite sulla variabili comuni da uno di essi siano eseguite in concorrenza con operazioni sulle stesse variabili da parte dell’altro, generando interferenze e quindi errori. Per questo motivo è necessario imporre la mutua esclusione fra le esecuzioni di queste operazioni che, come è stato indicato nel capitolo 3, vengono spesso riferite col nome di *sezioni critiche*.

Per risolvere questo problema in Java ad ogni oggetto, a qualunque classe appartenga, viene associato dalla JVM un meccanismo di mutua esclusione (*lock*) analogo a un semaforo binario (come, ad esempio, il semaforo `mutex` visto nel paragrafo 3.4.1). Tale meccanismo è nascosto all’interno del supporto fornito dalla JVM e non è quindi direttamente disponibile al programmatore. È però possibile denotare alcune sezioni di codice che operano su un oggetto, come sezioni critiche identificandole con la parola chiave `synchronized`. È poi compito del compilatore garantire che tali sezioni critiche siano eseguite in mutua esclusione inserendo in testa alla sezione critica un prologo, il cui scopo è garantire l’acquisizione del *lock* associato all’oggetto (se libero, altrimenti il thread che lo esegue viene sospeso) e, alla fine della sezione critica un epilogo per rilasciare il *lock* (vedi paragrafo 3.4.1 per una dettagliata illustrazione del prologo e dell’epilogo).

Per esempio, con riferimento ad un oggetto `x` è possibile definire un blocco di statements come una sezione critica nel seguente modo, noto col termine di blocco sincronizzato (*synchronized block*):

```
synchronized (oggetto x) <sequenza di statements>;
```

Nella figura B.7 viene riportato l’esempio del metodo `M` che più threads possono invocare ma che, al proprio interno, contiene un blocco sincronizzato (*<sezione di codice critica>*) che viene quindi eseguito in mutua esclusione.

```
Object mutexLock = new Object();
...
...
public void M() {
    <sezione di codice non critica>;
    synchronized (mutexLock) {
        <sezione di codice critica>;
    }
    <sezione di codice non critica>;
}
```

Figura B.7 Esempio di blocco sincronizzato.

In questo esempio l'oggetto `mutexLock` viene usato esclusivamente (come indicato dal suo nome) per sfuggire il suo *lock* al fine di garantire che la sezione critica di codice all'interno del metodo `M` sia eseguita in mutua esclusione. In particolare quando un thread invoca `M` può eseguire la prima parte del metodo (<sezione di codice non critica>) senza nessun vincolo, anche in concorrenza con altri threads che a loro volta abbiano invocato `M`. Quando, però, un thread tenta di eseguire il blocco sincronizzato può proseguire soltanto se il *lock* associato a `mutexLock` è libero, altrimenti il thread viene sospeso dalla JVM in attesa che il *lock* si liberi. Se il *lock* è libero il thread prosegue ed occupa (atomicamente) il *lock* disabilitando altri threads ad entrare a loro volta nella sezione critica. Quando il thread termina l'esecuzione del blocco sincronizzato, se non ci sono altri threads in attesa, il *lock* viene reso libero altrimenti la JVM sceglie arbitrariamente uno dei threads in attesa abilitandolo ad occupare nuovamente il *lock* e ad eseguire, a sua volta, la sezione critica.

Il fatto che ad ogni oggetto sia implicitamente associato un *lock* implica che ad esso è anche associato un insieme di threads (eventualmente vuoto) che avendo tentato di eseguire un blocco sincronizzato controllato dal *lock* di tale oggetto ed avendolo trovato occupato, sono stati sospesi in attesa che il *lock* venga liberato. Questo insieme di threads viene anche indicato come *entry set* (vedi figura B.8).

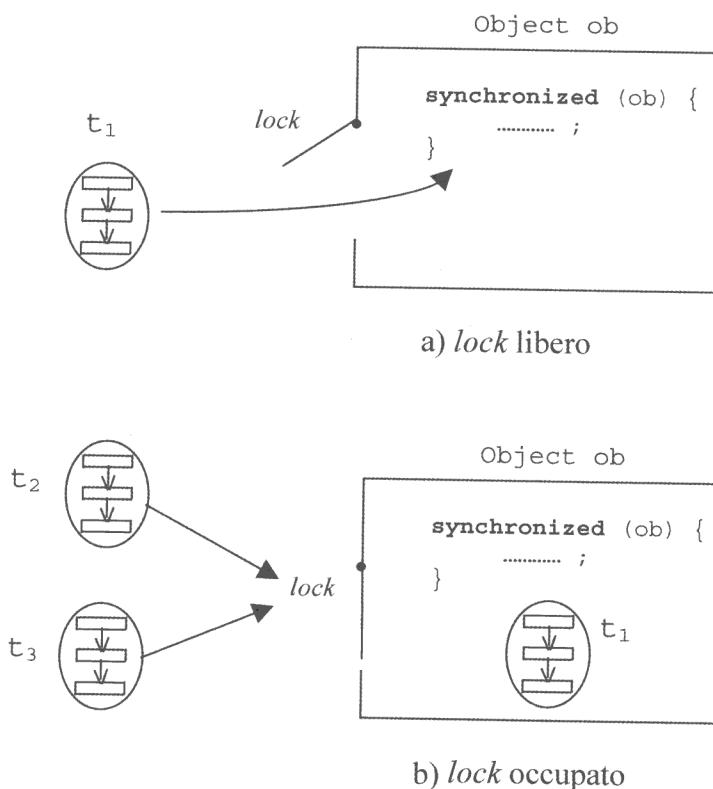


Figura B.8 Entry set di un oggetto.

```

public class IntVar {
    private int i=0;
    public synchronized void incrementa() {
        i++;
    }
    public synchronized void decrementa() {
        i--;
    }
}

```

Figura B.9 Metodi synchronized.

Nella parte a) della figura viene rappresentato il thread t_1 che tenta di eseguire un blocco sincronizzato controllato dal *lock* dell'oggetto *ob* e, avendolo trovato libero, può iniziare la sua esecuzione occupando il *lock* che viene chiuso. Successivamente (parte b della figura) altri threads (t_2 e t_3) tentano di eseguire lo stesso blocco sincronizzato ma trovando il *lock* chiuso si sospendono entrando quindi a far parte dell'*entry set* di *ob*.

Il blocco sincronizzato non costituisce l'unica modalità per definire sezioni critiche in Java. Esiste anche la possibilità di definire la mutua esclusione fra metodi di una classe. In questo caso è sufficiente che tali metodi siano caratterizzati dalla parola chiave *synchronized*. Quando uno di tali metodi viene invocato per operare su un oggetto della classe, l'esecuzione del metodo viene garantita in mutua esclusione sfruttando il *lock* dell'oggetto. Ad esempio in figura B.9 viene fornito il codice di una classe che permette di definire variabili intere sulle quali più threads possano eseguire le operazioni di *incrementa* e *decrementa* in maniera mutuamente esclusiva.

I due modi di definire sezioni critiche: il blocco sincronizzato e il metodo sincronizzato non sono fra loro indipendenti. Un metodo dichiarato *synchronized* corrisponde ad un metodo il cui corpo coincide con un blocco sincronizzato controllato dall'oggetto su cui il metodo viene eseguito (vedi figura B.10 nella quale i due metodi della figura B.9 sono riscritti come blocchi sincronizzati).

Possiamo adesso ricapitolare i meccanismi offerti da Java per garantire la mutua esclusione di sezioni critiche:

- la sincronizzazione viene implementata mediante il meccanismo interno dei *lock* (semplici semafori di mutua esclusione) offerto dalla JVM (e non visibile direttamente al programmatore) che associa un *lock* ad ogni oggetto;
- il meccanismo linguistico principale è quello indicato precedentemente col termine di *synchronized block*, mediante il quale un qualunque blocco di codice può essere sincronizzato in modo tale da garantirne l'esecuzione in maniera mutuamente esclusiva rispetto ad altre esecuzioni dello stesso blocco o di altri blocchi sincronizzati sullo stesso oggetto. L'altro meccanismo, quello dei metodi *synchronized*, corrisponde allo stesso metodo non dichiarato *synchronized* ma il cui corpo è un unico blocco sincronizzato mediante il *lock* dell'oggetto su cui il metodo viene eseguito;

```

public class IntVar {
    private int i = 0;
    public void incrementa() {
        synchronized (this) {
            i++;
        }
    }
    public void decrementa() {
        synchronized (this) {
            i--;
        }
    }
}

```

Figura B.10 Equivalenza tra blocchi sincronizzati e metodi synchronized.

- un metodo sincronizzato può invocare un altro metodo sincronizzato sullo stesso oggetto senza bloccarsi al fine di evitare condizioni di blocco critico;
- due diversi metodi, uno sincronizzato ed uno non sincronizzato, possono essere eseguiti concorrentemente sullo stesso oggetto.

B.3.2 Sincronizzazione diretta: metodi wait e notify

Il secondo meccanismo linguistico offerto da Java per consentire il coordinamento delle esecuzioni dei threads è relativo alla sincronizzazione diretta. A questo fine, la JVM associa implicitamente ad ogni oggetto, oltre al meccanismo dei locks visto precedentemente, anche una coda di threads, inizialmente vuota, nota col nome di *wait set*. I threads entrano ed escono da questa coda utilizzando i due metodi *wait()* e *notify()* offerti dalla classe *Object*, da cui tutte le classi di Java sono derivate. Questi due metodi vengono utilizzati con le seguenti regole:

- i due metodi *wait()* e *notify()* possono essere invocati da un thread esclusivamente all'interno di un blocco sincronizzato o di un metodo sincronizzato e cioè soltanto quando il thread detiene il lock relativo ad un oggetto;
- l'esecuzione del metodo *wait()* risulta nelle seguenti azioni:
 - il lock sull'oggetto viene rilasciato,
 - l'esecuzione del thread viene sospesa,
 - il thread viene inserito nel *wait set* relativo all'oggetto;
- l'esecuzione del metodo *notify()* risulta nelle seguenti azioni:
 - se il *wait set* relativo all'oggetto è vuoto non viene eseguita nessuna azione, altrimenti la JVM sceglie arbitrariamente un thread (indichiamolo con *t*) estraendolo dal *wait set* e lo inserisce nell'*entry set* in modo tale che, riattivando la sua esecuzione, questo possa riacquisire il *lock* e riprendere l'esecuzione dall'istruzione successiva alla *wait* con cui si era sospeso,
 - *t*, dovendo riacquisire il *lock* per riprendere l'esecuzione, deve comunque attendere che il thread che ha invocato la *notify()* rilasci a sua volta il *lock* che detiene (azione che avviene alla fine del blocco o del metodo sincronizzato in

- cui si trova la `notify`). Inoltre, poiché la `notify` inserisce `t` nell'*entry set*, per motivi di competizione può accadere che all'atto del rilascio del *lock* questo venga acquisito da un thread `t'` prima che `t` riesca a sua volta ad acquisirlo,
- una volta che `t` ha riacquisito il *lock*, la sua esecuzione riprende dall'istruzione successiva alla `wait` con cui si era sospeso;
 - oltre al metodo `notify()` viene offerto anche il metodo `notifyAll()` che risulta nelle stesse azioni del metodo `notify()`, ma con la differenza che vengono coinvolti tutti i threads contenuti nel *wait set*;
 - anche il metodo `wait()` ha delle alternative, in particolare quella che prevede la specifica di un tempo massimo da passare nel *wait set* prima di essere risvegliato automaticamente (*time-out*);
 - i due metodi `notify()` e `notifyAll()` non provocano il rilascio del *lock* da parte di chi li invoca per cui, come indicato precedentemente, i threads risvegliati devono attendere che il thread in esecuzione rilasci il *lock* terminando l'esecuzione del blocco, o del metodo, sincronizzato al cui interno si trova la `notify`, per poter riacquisire il *lock* e ripartire dall'istruzione successiva alla `wait`.

Per illustrare l'uso di questo meccanismo vedremo adesso come può essere risolto il problema della comunicazione tra threads (problema del *produttore/consumatore*) introdotto nel terzo capitolo (paragrafo 3.3). Supponiamo, per prima cosa, di risolvere il problema nel caso più semplice (vedi figura 3.3) di un solo thread *produttore* e un solo thread *consumatore* che si scambiano messaggi tramite un'area condivisa (*buffer*) in grado di contenere un solo messaggio e che il tipo dei messaggi, per semplicità, sia il tipo intero.

Nella successiva figura B.11 viene presentata la classe `Mailbox` a cui dovrà appartenere l'oggetto `buffer` da utilizzare come area condivisa tra i due threads.

```

public class Mailbox {
    private int contenuto;
    private boolean pieno = false;
    public synchronized int preleva () {
        while (pieno == false) {
            wait();
        }
        pieno = false;
        notify();
        return contenuto;
    }
    public synchronized void deposita (int valore) {
        while (pieno == true) {
            wait();
        }
        contenuto = valore;
        pieno = true;
        notify();
    }
}

```

Figura B.11 Mailbox unitaria.

La variabile intera `contenuto` rappresenta l'area di memoria condivisa mentre l'indicatore booleano `pieno` serve per registrare lo stato di tale area contenente un messaggio già depositato dal produttore e non ancora prelevato dal consumatore (valore `true`), oppure (valore `false`) quando ancora nessun messaggio è stato depositato o, se già depositato è stato anche prelevato. I metodi `deposita()` e `preleva()` sono sincronizzati in quanto operano su un oggetto condiviso. Se il produttore invoca `deposita` quando il `contenuto` è pieno deve sospendersi eseguendo `wait` in attesa che il consumatore prelevi il messaggio già presente in `contenuto`. Altrimenti il nuovo valore depositato andrebbe a sovrascrivere quello ancora presente in `contenuto`. Viceversa, quando il `contenuto` non è pieno, il messaggio viene inserito nel `contenuto` ponendo l'indicatore `pieno` a `true` per evitare ulteriori depositi e viene eseguita la `notify` per risvegliare il consumatore qualora questo sia in attesa che il `contenuto` sia pieno. L'altro metodo (`preleva`) è praticamente il duale del precedente. Da notare che quando un thread si sospende sulla `wait` (per esempio il produttore all'interno del metodo `deposita`) deve attendere che l'altro thread lo risvegli con la `notify` eseguita alla fine dell'altro metodo (nell'esempio eseguita dal consumatore alla fine del metodo `preleva` e viceversa). Quando un thread viene risvegliato, in base alle regole viste precedentemente, deve attendere che il thread in esecuzione rilasci il `lock`, alla fine del metodo in cui è presente la `notify`, quindi riacquisisce il `lock` e riprende l'esecuzione dopo la `wait`. Poiché questa è all'interno di un ciclo `while`, il thread risvegliato valuta di nuovo la condizione del `while` che adesso sarà sicuramente falsa. In questo caso particolare al posto del `while` avremmo potuto usare, più semplicemente, uno statement `if`. Se però generalizziamo il problema, ad esempio ipotizzando che siano presenti più threads produttori e/o più threads consumatori, allora l'uso del `while` diventa necessario. Per capire ciò, e contestualmente anche la necessità del metodo `notifyAll` in alternativa al semplice `notify`, ipotizziamo che un oggetto `buffer` della precedente classe sia condiviso tra molti produttori e molti consumatori. In questo caso (quando ad esempio, il `buffer` è vuoto) può accadere che siano contemporaneamente bloccati alcuni thread consumatori che – avendo invocato `preleva` e non essendoci niente da prelevare – si sono sospesi entrando nel `wait set` di `buffer`. Supponiamo anche che alcuni produttori siano presenti nell'`entry set` in attesa di acquisire il `lock` ed eseguire il metodo `deposita`. Appena il primo fra questi acquisisce il `lock`, esegue `deposita` riempiendo il `buffer` e con la `notify` sveglia uno dei consumatori prelevandolo dal `wait set` ed inserendolo nell'`entry set`. Quando tale produttore termina l'esecuzione di `deposita` rilascia il `lock` e quindi uno dei threads presenti nell'`entry set` viene abilitato a proseguire. Se per caso viene scelto ancora un produttore, quest'ultimo troverà il `buffer` pieno e quindi si sospenderà. Da questo momento in poi abbiamo nel `wait set` sia threads consumatori sospesi perché hanno trovato il `buffer` vuoto sia produttori che hanno trovato il `buffer` pieno. Quando un consumatore riuscirà ad acquisire il `lock`, potrà prelevare il messaggio contenuto nel `buffer` ed eseguire la `notify` per svegliare un produttore. Però, per le regole viste precedentemente, la `notify` sceglie in maniera casuale uno dei threads presenti nel `wait set` da risvegliare e allora può accadere che invece di un produttore venga scelto un consumatore che, evidentemente non può essere in grado di proseguire essendo vuoto il `buf-`

```

public class Mailbox {
    private int[] contenuto;
    private int contatore, testa, coda;

    public mailbox(){
        contenuto = new int[N];
        contatore = 0;
        testa = 0;
        coda = 0;
    }
    public synchronized int preleva (){
        int elemento;
        while (contatore == 0){
            wait();
        }
        elemento = contenuto[testa];
        testa = (testa + 1)%N;
        --contatore;
        notifyAll();
        return elemento;
    }
    public synchronized void deposita (int valore){
        while (contatore == N) {
            wait();
        }
        contenuto[coda] = valore;
        coda = (coda + 1)%N;
        ++contatore;
        notifyAll();
    }
}

```

Figura B.12 Buffer circolare.

fer. In questo caso, per garantire la correttezza della soluzione, è quindi necessario sostituire alla `notify` la `notifyAll` che risvegli tutti i threads presenti nel *wait set*. Questi, uno alla volta, riacquisiranno il *lock*, ma necessariamente dovranno rivelare la condizione per verificare se possono proseguire o se debbano sospendersi di nuovo. È questo il motivo per cui è necessario inserire la `wait` all'interno di un ciclo nel quale il thread rimane fino a quando non trovi la condizione falsa.

In figura B.12 l'esempio viene generalizzato supponendo che l'area condivisa sia in grado di contenere non uno soltanto, ma fino ad N messaggi contemporaneamente. In questo caso la variabile `contenuto` diventa un array di N elementi i quali vengono gestiti con tecnica FIFO mediante gli indici `testa` e `coda` (come è stato mostrato nel capitolo 3). Inoltre viene aggiunta la variabile `contatore` che è destinata a contenere il numero di elementi pieni dell'array. La condizione buffer vuoto coincide, in questo caso, con (`contatore == 0`) e, analogamente, la condizione buffer pieno con (`contatore == N`).

```

public class Semaforo {
    private int valore;
    public Semaforo(int v) {
        valore = v;
    }
    public synchronized void P() {
        while (valore == 0) {
            wait();
        }
        valore--;
    }
    public synchronized void V() {
        ++valore;
        notify();
    }
}

```

Figura B.13 Semaforo Java.

Come ultimo esempio, in figura B.13 viene riportato il codice della classe `semaforo` che implementa oggetti il cui comportamento corrisponde esattamente a quello di un semaforo introdotto nel terzo capitolo (paragrafo 3.4). Nell'esempio i metodi di accesso ad oggetti di tipo semaforo sono stati indicati con gli identificatori `P` e `V` e non con `wait` e `signal` come indicato nel paragrafo 3.4. Ciò per evitare la confusione che potrebbe nascere col metodo `wait` di Java. Del resto, i nomi `P` al posto di `wait` e `V` al posto di `signal`, sono quelli originariamente previsti da Dijkstra [31] quando ha introdotto il meccanismo semaforico per la prima volta.

Come ultima considerazione è necessario evidenziare che, per semplicità, in tutti i precedenti esempi il metodo `wait` è stato utilizzato senza fare riferimento ad eventuali eccezioni che tale metodo può sollevare. Il suo corretto uso prevede, viceversa, che sia sempre utilizzato all'interno di un blocco `try ... catch (InterruptedException e) {...}` poiché la definizione di tale metodo è la seguente:

```
public final void wait() throws InterruptedException;
```

Ciò significa che il codice che invoca il metodo `wait` può ricevere, come conseguenza di questa invocazione, l'eccezione `InterruptedException` che, quindi, deve essere gestita all'interno da un ramo `catch` appartenente a un blocco `try {...} catch(...){...}` al cui interno compare la chiamata di `wait`.

Infatti la specifica di Java prevede che, se un thread `ta` è sospeso, avendo invocato la `wait`, e prima di essere riattivato viene interrotto da un altro thread `tb` mediante il metodo `interrupt`, `ta` viene risvegliato e riceve l'eccezione `InterruptedException` che deve quindi gestire.

B.4 Sommario

Lo scopo di questa appendice è stato quello di esemplificare alcuni dei concetti di sistema visti nei precedenti capitoli mediante gli strumenti linguistici che Java mette a disposizione del programmatore. In particolare, data per scontata una generica conoscenza del paradigma di programmazione ad oggetti, e di Java in modo specifico, sono stati presentati il meccanismo multithreading di Java e i meccanismi di sincronizzazione previsti nel linguaggio per garantire la soluzione sia a problemi di competizione (sincronizzazione indiretta) che di cooperazione (sincronizzazione indiretta) tra threads.

B.5 Note bibliografiche

Per approfondire i principi del paradigma orientato agli oggetti si può fare riferimento a [10]. Per quanto riguarda un buon tutorial su Java si può fare riferimento a [15], che è anche disponibile sul sito web della Sun [52]. Altri testi di riferimento sono [43] e [44]. In [57] si può trovare una trattazione completa della concorrenza in Java.

Bibliografia

- [1] M. Accetta, R. Baron, W. Bolosky, D.B. Golub, R. Rashid, A. Tavenian e M. Young, "Mach: A New Kernel Foundation for Unix Development", *Proceedings of the Summer 1986 USENIX Conference*, pp. 93-112, Giugno 1986.
- [2] P. Ancilotti, M. Boari, *Principi e Tecniche di programmazione concorrente*, Utet Libreria, 1990.
- [3] G.R. Andrews, F.B. Schneider, *Concepts and Notations for Concurrent Programming*, ACM Computing Surveys, Volume 15, n. 1, 1983.
- [4] Apple Computer Inc., *Inside Macintosh*, Volume VI, Addison-Wesley, Reading, MA, 1991.
- [5] AT&T, S.V. Earhart (Editor), *UNIX Programmer's Manual*, Holt Rinehart Winston, New York, NY, 1986.
- [6] M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, N.Y., 1987.
- [7] L.A. Belady, "A Study of Replacement Algorithms for A Virtual Storage Computer", *IBM System Journal*, Volume 5, Numero 2, pp. 78-101, 1066.
- [8] 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*, Volume 12, Numero 6, pp. 349-353, 1969.
- [9] A.D. Birrel, B.J. Nelson, "Implementing Remote Procedure Call", *ACM Transactions on Computer Systems*, Volume 5, n. 1, 1987.
- [10] G. Booch, *Object Oriented Analysis and Design*, Second Edition, Benjamin/Cummings, Redwood City, CA, 1994.
- [11] S.R. Bourne, *UNIX System V*, Addison-Wesley, 1990.
- [12] D.P. Bovet, M. Cesati: *Understanding the Linux Kernel*, O'Reilly, 2001.
- [13] F.P. Brooks, *The Mythical Man-Month: Essay on Software Engineering*, Addison-Wesley, 1975.
- [14] G.C. Buttazzo, *Sistemi in Tempo Reale*, Pitagora Editrice, Bologna, 2001.
- [15] M. Campione e K., Walrath, *The Java Tutorial: Object-Oriented Programming for the Internet*, Second Edition, Sun Microsystems Press, Upper Saddle, NJ, 1998.
- [16] C. Cant, *Writing Windows WDM device drivers : covers NT 4, Win 98, and Win 2000*, Lawrence, KS : R&D Books, 1999.

- [17] R. Card, E. Dumas, F. Mevel, *The Linux Kernel Book*, John Wiley and Sons Inc., 1997.
- [18] W.R. Carr e J.L. Hennessy, "WSClock - A Simple and Effective Algorithm for Virtual Memory Management", *Proceedings of the Eighth Symposium on Operative Systems Principles*, pp. 87-95,
- [19] D. Clarke, D. Merusi, *System Software Programming: The Way Things Work*, Prentice-Hall, Upper Saddle River, NJ, 1998.
- [20] E.G. Coffmann, M.J. Elphick, A. Soshani, "System Deadlocks", *ACM Computing Surveys*, Volume 3, n. 2, 1971.
- [21] F.J. Corbató e V. Ayssotsky, "Introduction and overview of the MULTICS system", *Proceedings of the AFIPS Fall Joint Computer Conference*, 1965, pp. 185-196.
- [22] P. Corsini, G. Frosini, B. Lazzerini, *Architettura dei Calcolatori*, McGraw-Hill, 1997.
- [23] Computer System Research Group - University of California at Berkeley, *BSD UNIX Reference Manuals*, USENIX Association, 1986.
- [24] H. Custer, *Inside Windows NT*, Microsoft press, Redmont, Ma, 1993.
- [25] <http://www.debian.org>
- [26] E. Dekker, J. Newcomer, *Developing Windows NT Device Drivers: A Programmer's Handbook*, Reading MA, Addison Wesley 2000.
- [27] J.B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems", *Journal of the ACM*, Volume 12, Numero 4, pp. 589-602, 1965.
- [28] P.J. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, Volume 11, Numero 5, pp. 323-333, 1968.
- [29] P.J. Denning, "Virtual Memory", *ACM Computing Surveys*, Settembre 1970.
- [30] Digital Equipment Corporation, *Vax11 Software Handbook*, Maynard, Mass. 1978.
- [31] E.W Dijkstra, "Cooperating Sequential Processes", *Technical Report EDW-123*, Technological University, Eindhoven, Olanda, 1965.
- [32] E.W. Dijkstra, "The Structure of the THE Multiprogramming System", *Communications of the ACM*, Volume 8, Number 9, pp. 341-346, Maggio 1968.
- [33] J.I. Egan, T.J. Teixeira, *Writing a UNIX device driver*, Second Edition, John Wiley, NJ, 1992.
- [34] J. Gosling, B. Joy e G. Steele, *The java Language Specification*, Addison-Wesley, Reading, MA 1996.
- [35] D. Grosshans, *File Systems: Design and Implementation*, Prentice Hall, 1986.
- [36] A.N. Habermann, "Synchronization of Communicating Processes", *Communications of the ACM*, Volume 15, n. 3 1972.
- [37] Johnson Hart, *Win32 System Programming*, Addison-Wesley, settembre 2000.
- [38] J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Second edition, Morgan Kaufmann Publishers, Palo Alto (CA), 1996.
- [39] C.A.R Hoare, "Towards theory of parallel programming", in *Hoare and Perrot (Eds), Operating System Techniques*, Academic Press, New York, 1972.

- [40] C.A.R Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Volume 17, n. 10, 1974.
- [41] R.C. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys*, Volume 4, n. 3, 1972.
- [42] J.J. Horning, B. Randell, "Process structuring", *ACM Computing Surveys*, Volume 5, n. 1, 1973.
- [43] C. Horstmann e G. Cornell, *Core java 1.2, Volume 1: Fundamentals*, Prentice-Hall, Upper Saddle River, NJ, 1998.
- [44] C. Horstmann e G. Cornell, *Core java 1.2, Volume 2: Advanced Features*, Prentice-Hall, Upper Saddle River, NJ, 1998.
- [45] D.J. Howarth, R.B. Payne e F.H. Sumner, "The Manchester University Atlas Operating System, part II: User's Description", *Computer Journal*, Volume 4, Numero 3, pp. 226-229, 1961.
- [46] standards.ieee.org/catalog/olis/posix.html
- [47] http://pauillac.inria.fr/xleroy/linuxthreads/
- [48] http://www.intel.com/products/desktop/processors/pentium4/
- [49] International Standards Organization: "Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model", Report n. ISO/IEC 7498-1, 1994.
- [50] B. Jacob e T. Mudge, "Virtual Memory: Issues of Implementation" *Computer*, Giugno 1968.
- [51] B. Jacob e T. Mudge, "Virtual Memory in Contemporary Microprocessor", *IEEE Micro*, Agosto 1998.
- [52] http://java.sun.com/docs/books/tutorial/
- [53] L.J. Kenah e S.F. Bate, *VAX/VMS Internals and Data Structures*, Maynard, MA, Digital Press, 1984.
- [54] T. Kilburn, D.J. Howarth, R.B. Payne e F.H. Sumner, "The Manchester University Atlas Operating System, part I: Internal Organization", *Computer Journal*, Volume 4, Numero 3, pp. 222-225, 1961.
- [55] S. Kleiman, D. Shah, B. Smallders, *Programming with Threads*, Upper Saddle River, NJ: Prentice Hall, 1996.
- [56] D. Knuth, *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, seconda edizione, Addison-Wesley, Reading, MA, 1997.
- [57] D. Lea, *Concurrent Programming in Java: Design principles and Patterns*, Addison-Wesley, Readings, Ms, 1997.
- [58] S.J. Leffler, McKusick e al., *The design and the Implementation of the 4.3 BSD Unix Operating System*, Addison Wesley, 1989.
- [59] B. Lewis, D. Berg, *Threads Primer*, Upper Saddle River, NJ, Prentice Hall, 1996.
- [60] http://www.linux.org
- [61] http://www.linuxworks.com/
- [62] P. Massiglia (editor), *The RAID Book: A Storage System Technology Handbook*, St. Peter, MN: The RAID Advisory Board, 1997.
- [63] C. Mee, E. Daniel, *Magnetic Storage Handbook*, New York: McGraw-Hill, 1996.

- [64] M. Milenkovic, *Operating Systems: Concepts and Design*, McGraw-Hill, New York, 1992.
- [65] <http://msdn.microsoft.com/>
- [66] <http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html>
- [67] P. Norton, *Inside the IBM PC, Revised and Enlarged*, Brady Brooks, New York, NY, 1986.
- [68] W. Oney, *Programming the Microsoft Windows driver model*, Washington: Microsoft Press, 1999.
- [69] E.I. Organick, *The Multics System: An Examination of its Structure*, MIT Press, Cambridge, MA, 1972.
- [70] G. Pajari, *Writing UNIX device drivers*, Reading, Addison-Wesley, 1992.
- [71] D.L. Parnas, "A Technique for Software Module Specification with Examples", *Communications of the ACM*, Volume 15, Numero 5, Maggio 1972.
- [72] D.L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, Volume 5, Numero 12, Dicembre 1972.
- [73] D.A. Patterson, J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, San Mateo (CA), 1998.
- [74] G.L. Peterson, *Myths About The Mutual Exclusion Problem*, Information Processing letters, Volume 12, n. 3, 1981.
- [75] Charles Petzold, *Programming Windows*, Microsoft Press, Novembre 1998.
- [76] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones. "Mach: A System Software kernel", *Proceedings of the 34th Computer Society International Conference COMPCON 89*, February 1989.
- [77] <http://www.redhat.com>
- [78] W. Schwaderer, A. Wilson, *Understanding I/O Subsystem*, Milpitas, CA, Adaptec Press, 1996.
- [79] A. Silberschatz, P. Galvin, G. Gagne, *Operating Systems Concepts*, Sixth Edition, John Wiley and Sons Inc., 2001.
- [80] A. Silberschatz, P.B. Galvin, G. Gagne, *Sistemi operativi*, Addison Wesley, 2002.
- [81] A. Silberschatz, P. Galvin, G. Gagne, *Operating Systems Concepts with Java*, John Wiley and Sons Inc., 2003.
- [82] D. Solomon, "The WINDOWS NT Kernel Architecture", IEEE Computer, Ottobre 1998.
- [83] D.A. Solomon, *Indide Windows NT*, Second Edition, Microsoft Press, Redmond, WA, 1998.
- [84] W. Stallings, *Computer organization and architecture*, 5th edition, Upper Saddle River NJ, Prentice Hall, 2000.
- [85] W. Stallings, *Operating Systems*, 4th edition, Upper Saddle River NJ, Prentice Hall, 2001.
- [86] J.S. Stankovic e K. Ramamrithan, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems", *Operating System Review*, Luglio 1989.
- [87] W.R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.

- [88] R.W. Stevens, *TCP/IP Illustrated, vol. 1, vol. 2, vol. 3*, Addison-Wesley, 1996.
- [89] R.W. Stevens, *UNIX Network Programming - Networking APIs: Sockets and XTI*, Volume 1, Second Edition, Prentice Hall, 1998.
- [90] C. Strachey, "Time Sharing in Large Fast Computers", *Proceedings of the International Conference on Information Processing*, pp. 336-341, Giugno 1959.
- [91] Bjarne Stroustrup, *The C++ Programming Language, Third Edition*, Addison-Wesley, Giugno 1997.
- [92] <http://wwws.sun.com/software/solaris/>
- [93] A.S. Tanenbaum e R. Van Renesse, "Distributed Operating Systems", *ACM Computing Surveys*, Volume 17, Numero 4, pp. 419-470, Dicembre 1985.
- [94] A.S. Tanenbaum, Computer Networks, Prentice-Hall, 1988.
- [95] A.S. Tanenbaum, A.S. Woodhull, *Operating systems: Design and Implementation*, Prentice-Hall, 1997.
- [96] A.S. Tanenbaum, *Modern Operating Systems, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ, 2001.
- [97] A.S. Tanenbaum e M. Van Steen, *Distributed Systems: Principles and Paradigms*, Prentice-Hall, Englewood Cliffs, N.Y., 2002.
- [98] The Linux Documentation Project, <http://tldp.org/guides.html>
- [99] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
- [100] Chris Weber e Gary Bahadur, *Windows XP Professional Security*, McGraw-Hill, Ottobre 2002.
- [101] <http://www.windriver.com/products/vxworks5/>

Indice analitico

A

accept, 299, 306
ACL, 205, 206, 233
alarm, 246
algoritmo, 50
– clock, 147, 148
– rimpiazzamento, 136, 144-146, 148
ambiente globale, 75, 316
ambiente locale, 76, 90
attesa attiva, 80, 84, 171
attesa circolare, 96, 97

B

background, 66
batch, 8-10, 64-66
– multiprogrammati, 9-14
bind, 298
bit di flag, 168, 169, 172-173
bit di modifica, 136, 142, 145
bit di presenza, 136, 142, 144, 150, 153
bit di uso, 142, 146, 148
bit SUID, 205, 234
blocco
– fisico, 195
– sincronizzato, 317
blocco critico, 6, 91-101
– condizioni per, 96
– prevenzione dinamica, 98
– prevenzione statica, 97
– ripristino, 100
buffer, 80, 161-166
buffering, 161
bufferizzazione, 158, 162
bytecode, 50, 310

C

C-list, 205
cache, 23
– accesso associativo, 24
– bit di validità, 24
– elemento, 24
caricamento
– a domanda, 107, 120, 121, 136
– modulo di, 109-111, 113-115, 120
caricatore
– rilocante, 111, 115, 116, 122
catena circolare, 93
check point, 100
chiamata di procedura remota, *vedi* RPC
chiamata di sistema, *vedi* primitiva
chiave, 203
cilindro, 182
client server, *vedi* cliente-servitore
cliente-servitore, 90, 292
close, 158, 303
compattamento, 118, 140, 208
comunicazione
– asimmetrica, 88, 89
– canale di, 87
– diretta, 88
– indiretta, 89
– simmetrica, 89
concorrenza, 19
connect, 295, 298, 299
contatore delle istruzioni, 4, 21
contestato
– cambio di, 12, 16, 29, 43, 57, 263, 271
controllo di programma, 170-171, 190
controllore, 156, 159, 161, 163, 166-169, 171, 172, 175-177, 179, 180, 182

cooperative multithreading, 316
cycle stealing, 32, 177

D

database, 202
deadline, 17
deadlock, *vedi* blocco critico
deferred procedure call, *vedi* DPC
descrittore
– di segmento, 133, 136
– di sicurezza, 267
– dispositivo, 174-178, 180
device driver, 159, 166, 174, 177
direct memory access, *vedi* DMA
directory, *vedi* direttorio
direttorio, 196, 230, 236
– corrente, 222
– padre, 231
– radice, 197, 231
– Unix, 227
diritti di accesso, 117, 133, 136, 141
disco virtuale, 188, 189
dispatch interrupt, 264
DMA, 30-31, 177
– gestione, 177-178
DPC, 264
dummy process, *vedi* processo fittizio
dynamic priority boost, 273

E

eccezione, 163-164, 166
embedded system, 18
entry set, 318
`exec`, 204, 306
`exec1`, 227, 228
executive, 262, 265
`exit`, 226, 227, 248

F

file, 193, 194, 230
– accesso a indice, 201-203
– accesso diretto, 194, 202, 207-298
– accesso sequenziale, 194, 201
– apertura, 200
– append, 200
– attributi, 196
– chiusura, 201
– descrittore, 196, 199, 233-235
– diritti, 204
– linking, 197, 199
– operazioni, 198

– protezione, 196, 203-206
– puntatore, 201
file system, 193, 195-197, 230, 283
– gestione, 198
– struttura logica, 195
– transazionale, 284
foreground, 66
`fork`, 224-225, 239, 245, 248
frame, 137
frame table, 140
frammentazione, 117, 118, 127, 128, 137, 208
– esterna, 126, 128, 149
– interna, 123, 124, 126, 140

H

HAL, 262-264
handle, 266, 268, 269, 270, 279
hardware abstraction layer, *vedi* HAL

I

i-node, 232, 235
indice, 203
indirizzo
– fisico, 109, 114
– virtuale, 109, 110, 114
instruction pointer, *vedi* contatore
delle istruzioni
inter process communication, *vedi* IPC
interleaving, 59
interprete dei comandi, 3, 218
– Unix, 218, 219, 220
interrupt, *vedi* interruzione
interrupt request level, 264
interruzione, 12, 26-30, 151, 158, 159, 167, 168, 171-173, 179, 181, 263-265
– bit di abilitazione, 168
– esterna, 61
– gestione, 171-174
– handler di, 26, 167
– interna, 61
– ritorno da, 61, 176, 179
– segnale di, 159, 169, 179
interruzione software, *vedi* segnale
IPC, 87

J

java
– platform, 310
java virtual machine, *vedi* JVM
JCL, job control language, 8
JVM, 50, 310, 315, 316

K

kernel, vedi nucleo
kill, 245

L

linker, 108, 109, 115, 120
Linux, 216-217, 220, 248, 253, 259
LinuxThreads, 248, 249, 250
lista di capability, vedi C-list
lista di controllo degli accessi, vedi ACL
listen, 299, 306
loader, vedi caricatore
lock, 317
lock, 84

M

macchina virtuale, 3, 6-7, 15
malfunzionamento, 163, 164, 169, 179
master file table, vedi MFT
memoria
 – condivisione di, 105, 127, 130, 133, 142
 – protezione di, 105, 114, 117, 127, 130, 132, 149
memoria virtuale, 107-109, 217, 282
memory management unit, vedi MMU
memory mapping, 201
MFT, 284
microkernel, 43, 44, 218, 265
MMU, 103, 106, 114, 116, 120, 122, 128, 129, 130-132, 137, 139, 140, 154
monitor, 76
multiprogrammazione, 5, 10, 12, 13, 14, 16, 17, 30, 36, 229
 – grado di, 12, 36, 37, 67
multitasking, vedi multiprogrammazione
multithreading, 34, 68, 70, 220
mutua esclusione, 77-80, 82-84, 96, 316

N

notify, 314, 320-324
notifyAll, 314, 321, 322
NTFS, 284
nucleo
 – oggetti di, 266-269, 277
nucleo, 40, 42, 50, 60-63, 217, 262, 263
 – oggetti di, 266-269

O

offset, 130, 132, 138, 149, 152
open, 158, 239

overhead, 9, 12, 46, 64, 120
overlay, 120

P

page fault, 143-151, 153, 163
page table length register, vedi PTLR
pagedaemon, 229, 230
pagina, 137
 – fisica, 137, 140
 – rimpiazzamento, 143
 – virtuale, 137, 139, 140
paginazione, 121, 139-142, 217, 229, 230
 – a domanda, 142, 148, 151, 283
 – a livelli, 152
pagine
 – rimpiazzamento, 146
partizione
 – logica, 284
partizioni
 – fisse, 122-127
 – multiple, 127
 – rilocabili, 129
 – variabili, 122-127
PDE, 282
pipe, 242, 246-248
 – named, 248
POSIX, 216, 217, 248, 249
pre-paging, 148
primitiva, 6, 57, 58, 62, 181
priorità, 17, 26, 272
process manager, vedi nucleo
processi
 – interazione, 58-60
processo, 220, 221, 222, 269-271, 277-278
 – code di, 55-56
 – competizione, 59
 – concorrente, 58
 – contesto, 55
 – cooperazione, 59
 – CPU-bound, 66, 155
 – creazione, 57, 224-225
 – definizione, 50-51
 – descrittore, 54-55
 – esterno, 169, 170, 172, 173
 – figlio, 57
 – fittizio, 56
 – I/O-bound, 66, 155
 – immagine, 107-114, 222
 – leggero, vedi thread
 – padre, 57
 – priorità, 52, 65

- revoca, 52, 83
- riattivazione, 52
- sequenziale, 50
- sincronizzazione, 60
- sospensione, 52
- stato, 51-54
 - transizione di, 51
- storia, 51
- tabella dei, 54
- terminazione, 57, 225-227
- Unix, 220

produttore-consumatore, 59, 74, 80

program counter, *vedi* contatore delle istruzioni

program status word, 55

programma, 50

- istanze del, 51
- traccia di esecuzione, 51

protezione, 3, 5, 6, 266

- dominio di, 204
- matrice di, 205
- Unix, 232

PTE, 282

pthreads, 248-256

PTLR, 141

R

RAID, 187, 190

read, 158, 161-163, 166, 179, 239, 240, 247, 300, 301

real time, 7-18

receive, 76, 86-88, 91

record

- logico, 237

record logici, 194

record logico, 194

recv, 300

recvfrom, 302, 303

regione non salva, 98

registro

- controllo, 168, 169, 176
- dati, 168, 169, 171
- stato, 168, 169, 176

registro base, 114, 115

registro limite, 114

remote procedure call, *vedi* RPC

revoca, 106

rilocazione, 37, 103

- dinamica, 106, 109-115, 116-121, 128, 129, 150, 151, 161
- funzione di, 109, 113, 119, 120, 135
- statica, 109-115, 117, 120, 124

- rimpiazzamento, 135, 144
- globale, 148
- locale, 283
- risorse consumabili, 94
- risorse riusabili, 92
- round robin, 16, 64, 65, 179, 229, 316
- RPC, 79, 100

S

scambio di messaggi, 76

scheduling, 9, 53, 63-67

- algoritmi di, 13
- algoritmo, 105
- del disco, 165, 184-187
- FCFS, 64
- in java, 315
- long term, 66
- medium term, 66, 67, 134
- non preemptive, 64
- preemptive, 64
- short term, 56, 57, 63, 134
- windows, 271

scostamento, *vedi* offset

seek, 165-168

segment fault, 135-136, 150

segmentazione, 129, 132, 217, 229

- a domanda, 134-137
- paginata, 149

segmento

- rimpiazzamento, 135

segnalet, 242-246

semaforo, 76, 82-86, 250, 251, 253, 278, 281

- binario, 84, 250, 251

send, 76, 86-91, 100

sendto, 302

sendv, 300

setPriority, 315

settore, 165, 182-185, 189

sezione critica, 77, 274, 317, 318

shell, *vedi* interprete dei comandi

shutdown, 303

shutdown(), 301

signal, 76, 82-86, 92, 100

sincronizzazione, 286

- competizione, 74
- cooperazione, 73
- in java, 316
- oggetti di, 276

sistema di programmazione, 8

sistemi batch, *vedi* batch

sleep, 246, 314

socket, 242, 248, 292-296
 – datagram, 294, 295, 302
 – descrittore, 297
 – stream, 294, 295, 298, 304
socket, 297
 sottosistemi d'ambiente, 263
 spazio degli indirizzi, 109
 spinlock, 274
 spinsemaphore, 274
 spooling, 10, 161, 164
 stack pointer, 55
 stallo, *vedi* blocco critico
 stand-by, 169
 starvation, 65, 80, 84
 stato
 – diagramma degli, 220
 stato zombie, 221, 226
 sticky bit, 234
 supervisor call, *vedi* primitiva
SVC, *vedi* primitiva
 swap area, 105, 111, 112, 124, 134, 135, 137, 142-146
 swap-in, 67, 221
 swap-out, 67, 221, 222, 230
 swapping, 54
 synchronized, 317, 319
 system call, *vedi* primitiva

T

tabella
 – dei codici, 223
 – dei processi, 223, 224
 – dei segmenti, 131-133, 139
 – delle pagine, 138, 139, 142, 143, 150, 153
 – delle pagine fisiche, *vedi* frame table
 – descrittori di file, 198
 – file aperti, 200, 201, 237
 – interrupt dispatch, 264

task, 68
 tempo reale, *vedi* real time
 test and set lock, *vedi* TSL
 trashing, 146, 148
 thread
 – creazione, 249, 271, 312, 313
 – gestione dei, 68-70
 – package, 269
 – stato, 314, 315
 time sharing, 217
 time-sharing, 14-16
 TLB, 24, 131, 139
 traccia, 165, 181
 translation lookaside buffer, *vedi* TLB
TSL, 79, 274

U

unlock, 79, 80, 84

V

variabile condizione, 235, 254
 volume, 284

W

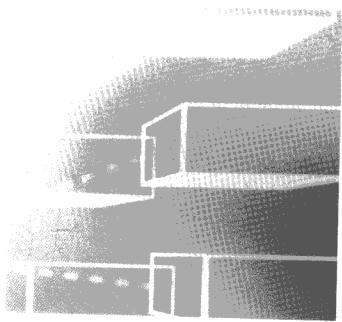
wait, 76, 82, 83, 84, 92, 93, 101, 226, 248, 253, 314, 320-324
wait set, 320
 working set, 146, 148, 149
write, 158, 163, 166, 168, 239, 240, 247, 300

X

XCH, 274

Y

yield, 316



Paolo Ancilotti
Maurelio Boari
Anna Ciampolini
Giuseppe Lipari

Sistemi operativi

Paolo Ancilotti è professore ordinario di Sistemi operativi presso la Scuola Superiore di Studi Universitari Sant'Anna di Pisa.

Maurelio Boari è professore ordinario di Calcolatori elettronici presso l'Università degli Studi di Bologna.

Anna Ciampolini è professore associato di Sistemi operativi presso l'Università degli Studi di Bologna.

Giuseppe Lipari è ricercatore di Sistemi di elaborazione delle informazioni presso la Scuola Superiore di Studi Universitari Sant'Anna di Pisa.

Questo testo, frutto della lunga esperienza di ricerca e di insegnamento degli Autori, è concepito per un primo insegnamento di Sistemi operativi previsto dal Nuovo Ordinamento della Laurea Triennale.

Vengono trattati gli argomenti di base che si ritiene debbono far parte del bagaglio culturale di chi si laurea in Ingegneria informatica o in Informatica, esaminando i concetti comuni a sistemi operativi diversi, illustrandoli tramite casi di studio (Unix, Linux e Windows) ed esempi.

Il testo fa riferimento a una tradizionale architettura monoelaboratore e a nessun linguaggio di programmazione in particolare, bensì utilizza uno pseudo linguaggio "C-like".

Corredano il testo due Appendici sulle problematiche di sincronizzazione in sistemi distribuiti e sul linguaggio Java.

All'indirizzo web www.ateneonline.it/ancilotti è disponibile materiale di supporto per i docenti.

€ 29,00

- www.mcgraw-hill.it
- www.ateneonline.it
- www.hyperbook.it

ISBN 88-386-6069-7

9 788838 660696

SOGGETTO LIBRERIA BIBLIOTECARIO DI ATENEO