

34. Perché le routine di servizio dell'interrupt sono associate a priorità mentre le procedure normali non hanno priorità?
35. L'architettura IA-64 contiene un numero di registri insolitamente elevato, 64. La scelta di averne così tanti è legata in qualche modo all'uso dell'attribuzione di predicit? Se sì, in che modo? Se no, allora perché ce n'è così tanti?
36. Nel testo analizziamo il concetto delle LOAD speculative, però non si fa nessuna menzione a istruzioni di STORE speculative. Perché no? Perché sono essenzialmente analoghe alle LOAD speculative o c'è un'altra ragione per non parlarne?
37. Quando si vogliono collegare due reti locali vi si inserisce in mezzo un computer chiamato *bridge*. Ogni pacchetto trasmesso in ciascuna delle due reti causa un interrupt sul bridge, affinché possa stabilire se deve inoltrare il pacchetto da una parte all'altra. Si supponga che ci vogliono 250  $\mu$ s per gestire l'interrupt e l'ispezione di ogni pacchetto, mentre il suo inoltro è a carico dell'hardware DMA e non aggrava il carico della CPU. Se tutti i pacchetti sono di 1 KB, qual è il massimo tasso di trasferimento dati che il bridge può tollerare su ciascuna delle due reti prima di cominciare a perdere pacchetti?
38. Nella Figura 5.40 il puntatore al record d'attivazione punta alla prima variabile locale. Di che informazione ha bisogno il programma per effettuare il ritorno da una procedura?
39. Si scriva una subroutine in linguaggio assemblativo per convertire in ASCII un intero binario con segno.
40. Si scriva una subroutine in linguaggio assemblativo per convertire una formula infissa in notazione polacca inversa.
41. Le torri di Hanoi non sono il solo problema ricorsivo popolare tra gli informatici. Un altro problema tra i più amati è il calcolo di  $n!$ . Si scriva una procedura in un linguaggio assemblativo per il calcolo di  $n!$ .
42. Se non si è convinti che la ricorsione è alle volte indispensabile si provi a programmare le torri di Hanoi senza usare la ricorsione e senza simulare la soluzione ricorsiva mantenendo lo stack?

## CAPITOLO 6

**Livello macchina  
del sistema operativo**

Il “motivo conduttore” di questo libro è un calcolatore moderno visto come una serie di livelli, ciascuno responsabile di nuove funzionalità rispetto a quelli sottostanti. Abbiamo già incontrato il livello logico digitale, il livello di microarchitettura e quello di architettura dell'insieme d'istruzioni. È ora tempo di salire di un altro livello, nel dominio del sistema operativo.

Un **sistema operativo** è un programma che, dal punto di vista del programmatore, aggiunge una moltitudine di nuove istruzioni e caratteristiche di più alto livello rispetto a quelle fornite dal livello ISA. In genere il sistema operativo è implementato per la maggior parte via software, ma non esiste alcuna argomentazione teorica che ne vietи l'implementazione in hardware, come succede normalmente per i microprogrammi (quando presenti). Per brevità, ci riferiremo al livello che implementa il sistema operativo con il termine di livello OSM (Operating System Machine). Si veda al proposito la Figura 6.1.

Nonostante entrambi i livelli OSM e ISA siano astratti (nel senso che non sono veri livelli hardware), sono sostanzialmente differenti. L'insieme d'istruzioni del livello OSM contiene tutte le istruzioni disponibili ai programmatore di applicazioni, pressoché tutte le istruzioni del livello ISA, così come l'insieme delle nuove istruzioni aggiunte dal sistema operativo, dette **chiamate di sistema (system call)**: una chiamata di sistema invoca un predeterminato servizio del sistema operativo, a tutti gli effetti una sua istruzione. Una chiamata di sistema tipica è la lettura di dati da un file. Nel prosieguo del testo indichiamo i nomi delle chiamate di sistema con un carattere che si distingue da quello del resto.

Il livello OSM è sempre interpretato. Quando un utente esegue un'istruzione OSM, come la lettura di dati da file, il sistema operativo svolge l'istruzione passo dopo passo, proprio come un microprogramma svolgerebbe un'istruzione ADD. D'altra parte, quando un programma esegue un'istruzione di livello ISA, questa viene calcolata direttamente

dal sottostante livello microarchitetturale, senza alcun intervento da parte del sistema operativo.

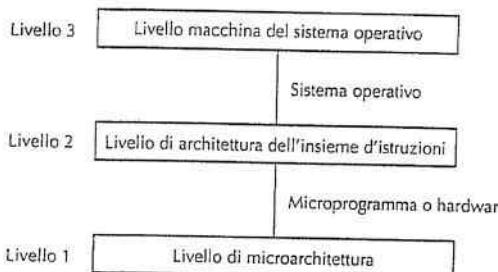


Figura 6.1 Livello macchina del sistema operativo.

In questo libro non possiamo che fornire un'introduzione molto succinta alla materia dei sistemi operativi, ma ci soffermeremo su tre argomenti importanti: il primo è la memoria virtuale, una tecnica messa a disposizione da molti sistemi operativi moderni per far sembrare che la macchina abbia più memoria di quanta ne ha davvero; il secondo è l'I/O su file, un concetto più ad alto livello delle istruzioni di I/O, analizzato nel capitolo precedente; il terzo argomento è il calcolo parallelo, ovvero il modo in cui i processi vengono eseguiti, come comunicano e si sincronizzano. Il concetto di processo è decisamente importante, e ne daremo una descrizione dettagliata nel corso del capitolo. Per il momento potete pensare a un processo come a un programma in esecuzione unito alle sue informazioni di stato (memoria, registri, program counter, stato dell'I/O e così via). Dopo aver analizzato questi principi in generale, vedremo come si applicano ai sistemi operativi di due delle nostre macchine paradigmatiche, cioè il Core i7 (Windows 7) e l'OMAP4430 ARM (Linux). L'ATmega168 non ha sistema operativo poiché è usato di norma all'interno di sistemi integrati.

## 6.1 Memoria virtuale

All'inizio dell'era informatica, le memorie dei computer erano costose e poco capienti. L'IBM 650, il calcolatore scientifico più avanzato del suo tempo (fine anni '50), aveva solo 2000 parole di memoria. Uno dei primi compilatori ALGOL 60 venne scritto per un computer con soltanto 1024 parole di memoria. Uno dei primi sistemi multiutente a condivisione di tempo girava abbastanza bene su di un PDP-1 con una memoria di 4096 parole di 18 bit, usata sia dal sistema operativo sia dai programmi degli utenti. In quel periodo, i programmatore impiegavano molto tempo nel tentativo di "assottigliare" i programmi perché entrassero in memorie minuscole. Alle volte si rendeva indispensabile usare un algoritmo molto più lento rispetto a un altro per il solo motivo che quello

più veloce era troppo "grande", ovvero nessun programma che lo usasse avrebbe potuto essere contenuto nella memoria del calcolatore.

La soluzione tradizionale a questo problema era l'uso di una memoria secondaria, per esempio di un disco. Il programmatore divideva il programma in un certo numero di pezzi, detti *overlay* ("ricoprimento"), ciascuno dei quali poteva entrare in memoria. Al fine di eseguire il programma si recuperava il primo overlay e lo si eseguiva per un po'. Al termine della sua esecuzione il primo pezzo leggeva l'overlay successivo e lo mandava in esecuzione e così via. Stava al programmatore ripartire il programma in parti, decidere dove memorizzarle in memoria secondaria, organizzare il trasferimento degli overlay tra la memoria principale e la secondaria, e quindi gestire l'intero procedimento di overlay senza ricevere alcun aiuto dal calcolatore.

Pur se usata per molti anni, questa tecnica richiedeva troppo lavoro da dedicare alla gestione degli overlay. Nel 1961 un gruppo di ricercatori di Manchester, in Inghilterra, propose un metodo per eseguire il processo di overlay automaticamente, in modo tale che il programmatore non avrebbe dovuto neanche accorgersi della sua presenza (Fotheringham, 1961).

Questo metodo, oggi chiamato **memoria virtuale**, presentava il vantaggio evidente di liberare il programmatore da un grosso lavoro di noiosa preparazione. Inizialmente utilizzata nell'ambito di progetti di ricerca sulla progettazione di sistemi di calcolo, a partire dagli anni '70 la memoria virtuale era divenuta disponibile su quasi tutti i calcolatori. Oggi anche i computer su chip singolo, tra cui il Core i7 e la CPU ARM OMAP4430, dispongono di sistemi di memoria virtuale molto sofisticati, come avremo modo di osservare nel corso di questo capitolo.

### 6.1.1 Paginazione

L'idea proposta dal gruppo di Manchester era quella di separare i concetti di spazio degli indirizzi e di locazioni di memoria. Si consideri, per esempio, un computer tipico di quel momento storico, dotato plausibilmente d'istruzioni con campi d'indirizzo di 16 bit e di 4096 parole di memoria. Un programma per questo computer avrebbe potuto indirizzare 65536 parole di memoria, poiché esistono  $65536 (2^{16})$  stringhe di 16 bit, ciascuna corrispondente a un diverso indirizzo di memoria. Si tenga presente che il numero di parole indirizzabili dipende esclusivamente dal numero di bit nell'indirizzo e non ha nulla a che fare con il numero di parole di memoria effettivamente disponibili. Lo spazio degli indirizzi di questo computer è fatto dai numeri 0, 1, 2, ..., 65535 perché è quello l'insieme degli indirizzi possibili. Il computer può avere comunque meno di 65536 parole di memoria.

Prima dell'invenzione della memoria virtuale, si guardava diversamente agli indirizzi minori di 4096 e a quelli uguali o maggiori di 4096. Anche se non venivano chiamati esplicitamente così, il primo gruppo d'indirizzi era considerato spazio utile degli indirizzi, il secondo spazio inutile degli indirizzi (inutile perché non aveva corrispondenza con gli effettivi indirizzi di memoria). Non si distingueva tra spazio degli indirizzi e locazioni di memoria, perché l'hardware imponeva una corrispondenza uno a uno tra di loro.

L'idea di separare lo spazio degli indirizzi e gli indirizzi di memoria è la seguente: in ogni istante sono direttamente accessibili 4096 parole di memoria, ma non c'è bisogno che corrispondano agli indirizzi di memoria tra 0 e 4095. Per esempio, potremmo "comunicare" al computer che d'ora in poi, ogniqualvolta ci si riferisce all'indirizzo 4096, occorre usare la parola di memoria all'indirizzo 0. Quando si referencia l'indirizzo 4097, si deve usare la parola di memoria all'indirizzo 1; se si referencia l'indirizzo 8191, bisogna usare la parola di memoria all'indirizzo 4095 e così via. In altri termini, abbiamo definito una corrispondenza tra lo spazio degli indirizzi e le effettive locazioni di memoria, illustrata nella Figura 6.2.

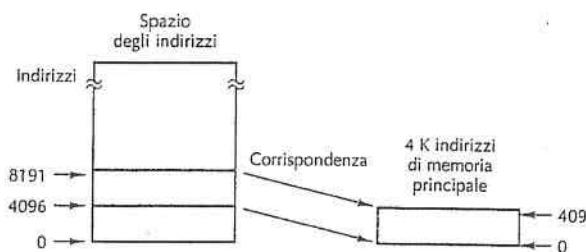


Figura 6.2 Corrispondenza tra gli indirizzi virtuali da 4096 a 8191 e gli indirizzi di memoria principale da 0 a 4095.

Secondo questa corrispondenza tra lo spazio degli indirizzi e le reali locazioni di memoria, una macchina con 4 KB di memoria, ma senza memoria virtuale, presenta una semplice corrispondenza fissa tra gli indirizzi compresi tra 0 e 4095 e le 4096 parole di memoria. Una domanda interessante è: "che cosa succede se un programma salta a un indirizzo tra 8192 e 12287?". Su di una macchina priva di memoria virtuale il programma causerebbe una trap che visualizzerebbe sullo schermo un messaggio adeguatamente severo, per esempio "memoria referenziata inesistente", e terminerebbe il programma. Su di una macchina con memoria virtuale si verificherebbero i seguenti avvenimenti:

1. il contenuto della memoria principale verrebbe salvato su disco;
2. verrebbero localizzate su disco le parole dalla 8192 alla 12287;
3. le parole dalla 8192 alla 12287 verrebbero caricate in memoria;
4. si modificherebbe la mappa degli indirizzi per mappare gli indirizzi da 8192 a 12287 nelle locazioni di memoria da 0 a 4095;
5. l'esecuzione continuerebbe come se non fosse successo nulla di insolito.

Questa tecnica di overlay automatico si chiama **paginazione** e le parti di programma lette dal disco si chiamano **pagine**.

È possibile anche una modalità più sofisticata di corrispondenza tra lo spazio degli indirizzi e le reali locazioni di memoria. Per evitare una possibile confusione, ci riferi-

remo agli indirizzi utilizzabili dal programma con il termine di **spazio degli indirizzi virtuali**, mentre ci riferiremo alle locazioni effettive, fisicamente cablate in memoria come allo **spazio degli indirizzi fisici**. Una mappa di memoria o tabella delle pagine specifica qual è l'indirizzo fisico corrispondente a ogni indirizzo virtuale. Facciamo qui l'ipotesi che ci sia spazio sufficiente sul disco per memorizzare l'intero spazio degli indirizzi virtuali (o quanto meno la sua porzione in uso).

I programmi vengono scritti come se ci fosse memoria sufficiente per l'intero spazio degli indirizzi virtuali, anche se non è così. I programmi possono caricare o memorizzare ogni parola nello spazio degli indirizzi virtuali, o saltare a qualsiasi istruzione situata ovunque nello spazio degli indirizzi virtuali, senza curarsi del fatto che in realtà non c'è sufficiente memoria fisica. In effetti, il programmatore può scrivere programmi senza neanche sapere che esiste la memoria virtuale, semplicemente pensando che il calcolatore abbia una grande memoria.

Questo concetto è di cruciale importanza e sarà poi messo a confronto con la segmentazione, che invece richiede che il programmatore sia consapevole dell'esistenza dei segmenti. Lo sottolineiamo ancora una volta: la paginazione dà l'illusione al programmatore di una memoria grande, continua e lineare, delle stesse dimensioni dello spazio degli indirizzi virtuali. In realtà la memoria principale disponibile potrebbe essere più piccola (o più grande) dello spazio degli indirizzi virtuali. La simulazione di questa grande memoria principale mediante la paginazione non può essere individuata dal programma (se non eseguendo test sui tempi di esecuzione). Ogniqualvolta viene referenziato un indirizzo, l'istruzione o la parola di dati corrispondente sono percepite come presenti. Dal momento che il programmatore può lavorare come se la paginazione non esistesse, il meccanismo di paginazione si dice **trasparente**.

L'idea che un programmatore possa usare alcune caratteristiche inesistenti senza doversi preoccupare di come funzionano non è, dopo tutto, un'assoluta novità. L'insieme d'istruzioni a livello ISA include spesso un'istruzione MUL, anche se la microarchitettura sottostante manca di un moltiplicatore hardware. L'illusione che la macchina possa moltiplicare è sorretta dal microcodice. Allo stesso modo, la memoria virtuale messa a disposizione dal sistema operativo può fornire l'illusione che gli indirizzi virtuali si riflettano nella memoria reale, anche se non è così. Solo i programmatori del sistema operativo (e gli studenti della materia) devono sapere su che cosa si regge questa illusione.

## 6.1.2 Implementazione della paginazione

Un requisito fondamentale per la memoria virtuale è disporre di un disco che contenga l'intero programma e tutti i dati. Il disco può essere meccanico a rotazione oppure allo stato solido. Per il resto del libro parleremo per semplicità di "disco" o "disco fisso", lasciando sottointeso che sono inclusi anche le unità allo stato solido. Concettualmente è più semplice pensare alla copia del programma su disco come all'originale, e ai pezzi trasferiti di quando in quando in memoria centrale come alle copie, piuttosto che il contrario. Naturalmente è importante mantenere l'originale aggiornato: quando vengono effettuati cambiamenti sulla copia in memoria centrale questi dovrebbero essere propagati all'originale (prima o poi).

Lo spazio degli indirizzi virtuali è suddiviso in un certo numero di pagine della stessa dimensione. Le dimensioni di pagina più in voga vanno dai 512 ai 64 KB, anche se occasionalmente vengono usate pagine di 4 MB. La dimensione di pagina è sempre una potenza di 2, per esempio  $2^k$ , di modo che tutti gli indirizzi possano essere rappresentati con  $k$  bit. allo stesso modo, anche lo spazio degli indirizzi fisici è suddiviso in porzioni che hanno le dimensioni di una pagina, di modo che ogni pezzo di memoria principale possa contenere esattamente una pagina. Questi pezzi di memoria principale che servono a contenere le pagine si dicono **blocchi di memoria (page frame)**. La memoria principale della Figura 6.2 contiene un solo blocco di memoria, ma una memoria reale ne contiene in genere alcune migliaia.

La Figura 6.3(a) illustra un modo di suddividere in pagine di 4 KB i primi 64 KB del spazio degli indirizzi virtuali. In realtà qui stiamo parlando di 64 KB di memoria e di pagine di 4 K d'indirizzi; un indirizzo potrebbe essere lungo un byte, ma anche una parola (nei computer in cui indirizzi consecutivi individuano parole consecutive). La memoria virtuale della Figura 6.3 potrebbe essere implementata per mezzo di una tabella delle pagine grande quanto il numero di pagine presenti nello spazio degli indirizzi virtuali. Per semplicità, nella figura mostriamo solo i primi 16 elementi della tabella. Quando un programma cerca di accedere a una parola di memoria nei primi 64 KB del suo spazio degli indirizzi virtuali (o per fare il fetch di un'istruzione o di un dato, o per salvare alcuni dati) per prima cosa genera un indirizzo virtuale compreso tra 0 e 65532 (nell'ipotesi che gli indirizzi di parola siano divisibili per 4). La generazione di questo indirizzo può avvenire tramite una qualsiasi tecnica usuale d'indirizzamento: indicativo, indiretto e così via.

La Figura 6.3(b) mostra una memoria fisica che consiste in otto blocchi di memoria di 4 KB. Questa memoria potrebbe essere limitata a 32 KB perché (1) è tutto ciò di cui dispone la macchina (potrebbe bastare a un processore integrato in una lavatrice o in un forno a microonde) oppure (2) perché il resto della memoria è stato allocato ad altri programmi.

Si consideri ora il meccanismo di corrispondenza tra un indirizzo virtuale di 32 bit e un indirizzo fisico della memoria principale. Dopo tutto la sola cosa comprensibile alla memoria sono gli indirizzi di memoria principale, non gli indirizzi virtuali, perciò bisogna essere in grado di fornirglieli. Ogni computer dotato di memoria virtuale è provvisto di un dispositivo atto alla corrispondenza tra indirizzi virtuali e fisici: si chiama MMU (*Memory Management Unit*, “unità di gestione della memoria”) e può risiedere nel chip della CPU oppure in un chip separato che lavora a stretto contatto con la CPU. La MMU del nostro esempio mappa indirizzi virtuali di 32 bit in indirizzi fisici di 15 bit, perciò ha bisogno di registri d’ingresso di 32 bit e di registri d’uscita di 15 bit.

Si veda la Figura 6.4 per capire come funziona la MMU. Quando viene inviato un indirizzo virtuale di 32 bit alla MMU, questa divide l'indirizzo in un numero di pagina virtuale di 20 bit e in un offset di 12 bit all'interno della pagina (si ricordi che le pagine del nostro esempio sono di 4 KB). Il numero di pagina virtuale è usato per indicizzare la tabella delle pagine al fine di trovare l'elemento corrispondente alla pagina referenziata. Nel caso della Figura 6.4 il numero di pagina virtuale è 3 e così viene selezionato il terzo elemento della tabella delle pagine.

Pagina	Indirizzi virtuali	
~	~	~
15	61440 - 65535	
14	57344 - 61439	
13	53248 - 57343	
12	49152 - 53247	
11	45056 - 49151	
10	40960 - 45055	
9	36864 - 40959	
8	32768 - 36863	
7	28672 - 32767	
6	24576 - 28671	
5	20480 - 24575	
4	16384 - 20479	
3	12288 - 16383	
2	8192 - 12287	
1	4096 - 8191	
0	0 - 4095	
		Primi 32 K di memoria principale
		Blocco di memoria
		Indirizzi fisici
		7 28672 - 32767
		6 24576 - 28671
		5 20480 - 24575
		4 16384 - 20479
		3 12288 - 16383
		2 8192 - 12287
		1 4096 - 8191
		0 0 - 4095

**Figura 6.3** (a) Primi 64 KB degli indirizzi virtuali suddivisi in 16 pagine da 4 KB. (b) Memoria principale di 32 KB suddivisa in otto blocchi di memoria di 4 KB.

Per prima cosa la MMU verifica nella tabella delle pagine se la pagina referenziata è presente in memoria centrale. Ovviamente, con  $2^{20}$  pagine virtuali e soli otto blocchi di memoria, non tutte le pagine virtuali possono trovarsi contemporaneamente in memoria. Nello svolgere questa verifica la MMU esamina il bit **presente/assente** nell'elemento della tabella delle pagine. Nel nostro esempio il bit è a 1, a significare che la pagina si trova correntemente in memoria.

Il passo successivo è il recupero del valore del blocco di memoria contenuto nell'elemento selezionato (6 in questo caso) e la sua copia nei 3 bit più significativi del registro d'uscita di 15 bit. Ci vogliono esattamente tre bit perché ci sono otto blocchi nella memoria fisica. Contemporaneamente a questa operazione si effettua la copia dei 12 bit meno significativi dell'indirizzo virtuale (il campo offset di pagina) all'interno dei 12 bit meno significativi del registro d'uscita, come mostrato nella figura. Questo indirizzo di 15 bit è ora pronto per essere inviato alla cache o alla memoria per la sua ricerca.

La Figura 6.5 mostra una possibile corrispondenza tra le pagine virtuali e i blocchi di memoria fisici. La pagina virtuale 0 si trova nel blocco di memoria 1 e la pagina virtuale 1 nel blocco di memoria 0. La pagina virtuale 2 non si trova in memoria. La pagina virtuale 3 si trova nel blocco di memoria 2. La pagina virtuale 4 non si trova in memoria. La pagina virtuale 5 si trova nel blocco di memoria 6 e così via.

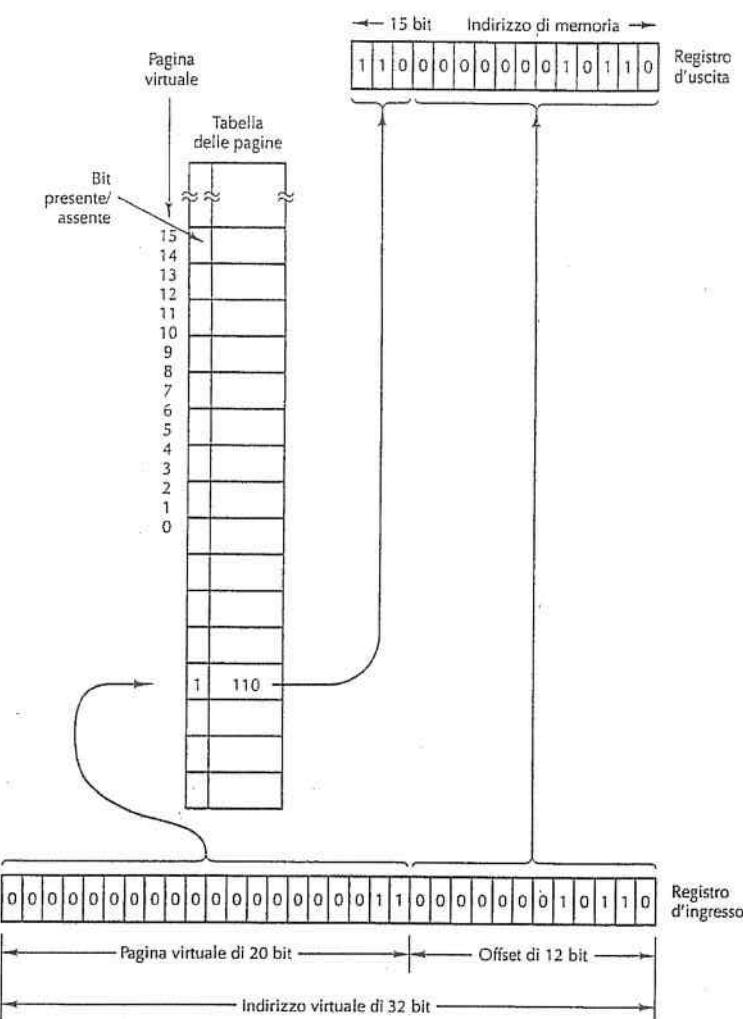


Figura 6.4 Formazione di un indirizzo fisico a partire da un indirizzo virtuale.

### 6.1.3 Paginazione a richiesta e working set

Nella precedente trattazione abbiamo supposto che la pagina virtuale referenziata si trovasse in memoria principale, ma questa assunzione non è sempre vera, perché non c'è spazio sufficiente in memoria principale per ospitare tutte le pagine virtuali. Quando si fa riferimento a un indirizzo contenuto in una pagina che non è presente in memoria si

solleva un errore di pagina (*page fault*). Dopo l'occorrenza di un errore di pagina è necessario che il sistema operativo legga dal disco la pagina richiesta, inserisca la sua nuova posizione all'interno della memoria fisica nella tabella delle pagine e ripeta l'istruzione che ha causato l'errore.

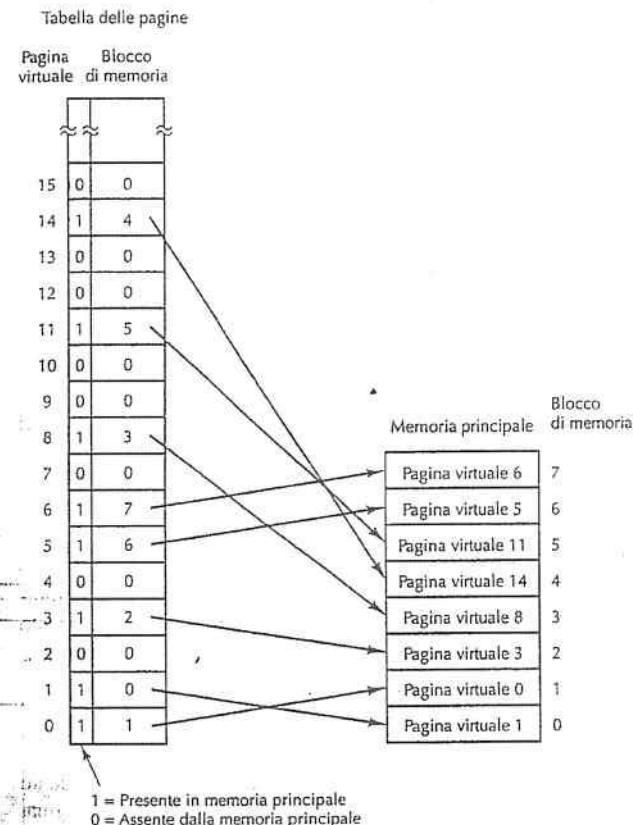


Figura 6.5 Assegnamento di 16 pagine virtuali in una memoria principale composta di otto blocchi.

Su una macchina con memoria virtuale è possibile lanciare l'esecuzione di un programma anche se la memoria principale non contiene alcuna sua istruzione. Bisogna semplicemente indicare nella tabella delle pagine che ogni pagina virtuale del programma si trova in memoria secondaria e non in memoria principale. Quando la CPU prova a effettuare il fetch della prima istruzione si verifica immediatamente un errore di pagina, che causa il caricamento in memoria della pagina contenente la prima istruzione e il suo

ingresso nella tabella delle pagine. Dopodiché può cominciare l'esecuzione della prima istruzione; se questa contiene due indirizzi su pagine differenti, e diverse dalla pagina dell'istruzione, allora si verificheranno altri due errori di pagina e dovranno essere recuperate altre due pagine prima che l'istruzione possa essere finalmente eseguita. L'istruzione successiva può a sua volta causare altri errori di pagina e così via.

Questo funzionamento della memoria virtuale si chiama **paginazione a richiesta**, in analogia al ben noto algoritmo di allattamento a richiesta dei bambini: il neonato viene allattato quando piange (piuttosto che in base a una precisa tabella oraria). Nella paginazione a richiesta una pagina è portata in memoria principale solo quando viene effettuata una richiesta, non prima.

L'interrogativo circa l'utilità della paginazione a richiesta è rilevante solo nella fase iniziale dell'esecuzione di un programma. Dopo un certo tempo d'esecuzione, le pagine necessarie saranno già state radunate in memoria. Se, tuttavia, il computer è usato a condivisione di tempo e i processi si alternano ogni 100 ms, ciascun programma verrà fatto ripartire molte volte nel corso della sua esecuzione. Poiché c'è una mappa di memoria per ogni programma e questa deve essere sostituita ognqualvolta due programmi si danno il cambio, come succede per esempio in un computer a condivisione di tempo, il problema se reiterato diventa critico.

L'approccio alternativo si basa sull'osservazione che molti programmi non referenziano il loro spazio degli indirizzi in modo uniforme, bensì i riferimenti tendono a raggrupparsi su di un piccolo numero di pagine. Questo concetto è noto come **principio di località**. Un riferimento alla memoria può servire al fetch di un'istruzione o di dati, o alla memorizzazione di dati; per ogni istante di tempo  $t$  si può definire l'insieme delle pagine usate dai  $k$  riferimenti a memoria più recenti. Denning (1968) ha chiamato questo insieme **working set** ("insieme di lavoro").

Visto che il working set in genere cambia lentamente nel tempo, è possibile indovinare con un buon margine di certezza quali saranno le pagine necessarie a un programma nel momento in cui viene fatto ripartire, basandosi sul suo working set al momento in cui è stato fermato. Così le pagine possono essere caricate in anticipo prima di far partire il programma (se ci stanno in memoria).

#### 6.1.4 Politica di sostituzione delle pagine

In teoria il working set, l'insieme delle pagine usate attivamente e frequentemente da un programma, può essere tenuto in memoria per ridurre il numero di errori di pagina. Tuttavia, i programmatori raramente conoscono le pagine del working set, perciò l'insieme deve essere costruito dinamicamente dal sistema operativo. Quando un programma riferenzia una pagina che non si trova in memoria principale questa deve essere recuperata dal disco. D'altra parte, per farle spazio sarà opportuno in genere spostare un'altra pagina sul disco e si rende quindi necessario un algoritmo per decidere quale pagina rimuovere.

Effettuare questa scelta a caso è quasi sicuramente una cattiva idea. Se capitasse di scegliere la pagina appena caricata a seguito dell'errore, un altro errore di pagina si verificherebbe non appena venisse richiesto il fetch dell'istruzione successiva. Gran parte dei sistemi operativi cerca di predire quale sia la pagina di memoria meno utile.

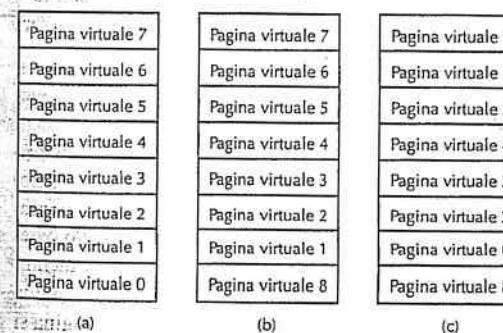
ossia la cui mancanza produrrebbe meno effetti deleteri sull'esecuzione del programma. Un modo di agire è predire quando avverrà il successivo riferimento a ciascuna pagina e rimuovere quella per cui la predizione è più lontana nel futuro. In altre parole, si cerca di scegliere una pagina che non verrà usata per molto tempo invece di rimuoverne una che sarà richiesta a breve.

Un algoritmo molto diffuso, detto **LRU** (*Least Recently Used*), estromette la pagina usata meno recentemente perché a questa corrisponde la massima probabilità di non far parte del working set corrente. Nonostante LRU funzioni mediamente bene, ci sono situazioni patologiche, come quella che ora descriviamo, in cui fallisce miseramente.

Si immagini un programma che esegue un grosso ciclo che si estende per nove pagine virtuali e che gira su di una macchina con spazio in memoria fisica per sole otto pagine. La Figura 6.6(a) mostra la memoria principale nel momento in cui il programma raggiunge la pagina 7. Quando si tenta di fare il fetch di un'istruzione della pagina virtuale 8 si verifica un errore di pagina e si pone il problema di quale pagina estromettere. L'algoritmo LRU sceglie la pagina virtuale 0 perché è quella usata meno recentemente. La pagina virtuale 0 viene rimossa e la pagina virtuale 8 la rimpiazza, come illustrato nella Figura 6.6(b).

Dopo aver eseguito le istruzioni della pagina virtuale 8, il programma salta all'inizio del ciclo, alla pagina virtuale 0. Anche questo passo causa un errore di pagina. La pagina virtuale 0, quella appena estromessa, viene riportata in memoria. L'algoritmo LRU sceglie di rimuovere ora la pagina 1, conducendo alla situazione della Figura 6.6(c). Il programma si intrattiene per un po' nella pagina 0, quindi cerca di recuperare un'istruzione della pagina virtuale 1, causando un errore di pagina. Bisogna riportare indietro la pagina 1 ed estromettere la pagina 2.

Dovrebbe essere chiaro che in questo caso l'algoritmo LRU opera sempre la scelta peggiore (esistono anche altri algoritmi che falliscono in condizioni analoghe). Se però la memoria centrale disponibile eccede la dimensione del working set, allora LRU tende a minimizzare il numero di errori di pagina.



**Figura 6.6** Fallimento dell'algoritmo LRU

Un altro algoritmo di sostituzione di pagina è **FIFO** (*First-In First-Out*), che rimuove la pagina caricata meno recentemente, indipendentemente dal momento in cui sia stata referenziata. Ogni pagina è associata a un contatore, inizialmente posto a 0. Dopo la gestione di un errore di pagina, il contatore di ciascuna pagina presente in memoria viene incrementato di un'unità e il contatore della pagina appena immessa è posto a 0. Quando si rende necessario scegliere un'pagina da estromettere, si seleziona quella con contatore più alto, cioè la pagina che ha assistito a più errori di pagina e che quindi è stata caricata prima di qualsiasi altra pagina presente in memoria, il che la rende (si spera) probabilmente meno utile delle altre.

Se il working set è più grande del numero di blocchi di memoria disponibili, non esiste alcun algoritmo, a meno che non abbia doti divinatorie, che possa dare buoni risultati e si verificheranno frequenti errori di pagina. Se un programma genera frequentemente e continuamente errori di pagina, allora è detto **thrashing** ("sconfitto, battuto"). Viceversa, se un programma usa una grande porzione dello spazio degli indirizzi virtuali ma ha un working set piccolo, che cambia lentamente nel tempo e che sta tutto in memoria principale, allora darà ben pochi problemi. Questa osservazione è vera anche per quei programmi che, nel corso della loro vita, usano un numero di parole della memoria virtuale centinaia di volte maggiore del numero di parole disponibili in memoria principale.

Se la pagina che sta per essere estromessa non è stata mai modificata dal momento in cui è stata caricata (cosa molto probabile se la pagina contiene soltanto istruzioni), non è necessario riscriverla su disco, perché questo ne contiene una copia fedele. Se invece è stata modificata, la copia su disco non è più accurata e la pagina deve essere riscritta.

Se esistesse un modo per stabilire quando una pagina è rimasta invariata dal suo caricamento (e in tal caso la pagina si dice "pulita", "intatta") o quando ha subito dei cambiamenti (e allora si dice "sporca"), si eviterebbero tutte le riscritture di pagine pulite, risparmiando molto tempo. Molti calcolatori dispongono nella MMU di 1 bit per pagina, detto appunto **dirty bit**, posto a 0 quando una pagina viene caricata e asserito dal microprogramma o dall'hardware quando è fatta oggetto di salvataggi (cioè quando è stata "sporcata"). Il sistema operativo può capire se una pagina è pulita o sporca osservando il suo dirty bit, e così decidere se va riscritta o meno.

### 6.1.5 Dimensione di pagina e frammentazione

Nell'eventualità in cui il programma e i dati dell'utente riempiano esattamente un numero intero di pagine, quando si trovano in memoria non causeranno alcuno spreco di spazio, mentre in caso contrario l'ultima pagina conterrà dello spazio inutilizzato. Per esempio, se il programma e i dati richiedono 26.000 byte su una macchina con pagine di 4096 byte, le prime sei pagine verranno riempite per un totale di  $6 \times 4096 = 24.576$  byte, mentre l'ultima pagina conterrà  $26.000 - 24.576 = 1424$  byte. Dal momento che in ogni pagina c'è spazio per 4096 byte, 2672 byte verranno sprecati. Ogni volta che la settima pagina si trova in memoria questi byte "vuoti" ne occuperanno una parte senza però svolgere alcuna funzione. Il problema legato a questo tipo di spreco di byte si chiama **frammentazione interna** (perché lo spazio inutilizzato si trova all'interno di una pagina).

Se la dimensione di pagina è di  $n$  byte, la quantità media di spazio sprecato nell'ultima pagina per frammentazione interna sarà di  $n/2$  byte, il che sembrerebbe suggerire l'adozione di dimensioni di pagina più piccole per minimizzare lo spreco. D'altro canto a una dimensione di pagina piccola corrisponde un gran numero di pagine e quindi un tabella delle pagine più grande. Se la tabella delle pagine è implementata in hardware, le sue dimensioni richiedono un maggior numero di registri per immagazzinarla, il che eleva i costi di produzione del calcolatore. Inoltre, anche i tempi di caricamento e salvataggio dei registri aumentano ogni volta che un programma viene fatto partire o viene fermato.

D'altra parte, le pagine piccole fanno un uso poco efficiente della larghezza di banda del disco: visto che bisogna aspettare sempre almeno 10 ms prima che un trasferimento possa cominciare (tempo di ricerca più ritardo rotazionale), è evidente che i trasferimenti sono tanto più efficienti quanto più byte trasferiscono. Trasferire 8 KB alla velocità di 100 MB/s aggiunge solo 70 µs rispetto al trasferimento di 1 KB.

Tuttavia, le pagine piccole presentano il vantaggio di causare meno thrashing, essendo il working set costituito da un gran numero di piccole regioni separate dello spazio degli indirizzi virtuali. Si consideri per esempio una matrice  $A$  di  $10.000 \times 10.000$  elementi, memorizzata come  $A[1,1], A[2,1], A[3,1]$  e così via, in parole consecutive di 8 byte. L'ordinamento per colonna implica che gli elementi della riga 1,  $A[1,1], A[1,2], A[1,3]$  e così via, si trovano distanti 80.000 byte. Se un programma svolgesse i propri calcoli su tutti gli elementi di questa riga, dovrebbe utilizzare 10.000 regioni, ciascuna separata dalla successiva da 79.992 byte. Se la dimensione di pagina è di 8 KB ciò richiede una disponibilità totale di 80 MB per contenere tutte le pagine in uso.

La dimensione di pagina di 1 KB richiederebbe invece soli 10 MB di RAM per contenere tutte le pagine. Se la memoria a disposizione è di 32 MB, il programma andrebbe in thrashing con pagine di 8 KB, ma non con pagine di 1 KB. Tutto considerato, la tendenza attuale si orienta verso dimensioni di pagina grandi. Nella pratica, il minimo al giorno d'oggi è 4 KB.

### 6.1.6 Segmentazione

La memoria virtuale vista fin qui è unidimensionale perché gli indirizzi virtuali vanno da 0 a un certo indirizzo massimo, in ordine progressivo. In molte situazioni può essere decisamente preferibile disporre di due o più spazi degli indirizzi virtuali separati piuttosto che di uno solo. Per fare un esempio, un compilatore potrebbe avere molte tabelle che si riempiono man mano che la compilazione procede, tra cui:

1. la tabella dei simboli per contenere i nomi e gli attributi delle variabili;
2. il codice sorgente, memorizzato per la visualizzazione del listato;
3. una tabella per contenere tutte le costanti intere e in virgola mobile utilizzate;
4. l'albero sintattico del programma;
5. lo stack, utilizzato per le chiamate di procedura del compilatore.

Ciascuna di queste tabelle cresce continuamente durante la compilazione. L'ultima addirittura cresce e decresce in modi imprevedibili. In una memoria unidimensionale

queste cinque tabelle sarebbero state allocate in unità contigue dello spazio degli indirizzi virtuali, come illustrato nella Figura 6.7.



Figura 6.7 In uno spazio degli indirizzi unidimensionale due tabelle in espansione possono collidere.

Se un programma ha un numero eccezionalmente elevato di variabili, la parte d'indirizzi allocata per la tabella dei simboli si potrebbe riempire, nonostante ci sia molto spazio libero nelle altre tabelle. Naturalmente il compilatore potrebbe cavarsela con un messaggio di errore comunicando l'impossibilità di proseguire la compilazione per la presenza di troppe variabili, ma non è un comportamento molto leale, specie considerando tutto lo spazio libero delle altre tabelle.

Un'altra possibilità prevede che il compilatore si comporti come Robin Hood, sottraendo spazio alle tabelle che ne hanno troppo per ridistribuirlo a quelle che ne sono povere. Questo rimescolamento è possibile, ma è un po' come gestire manualmente gli overlay: se va bene è una seccatura, se va male diventa un grosso carico di lavoro tedioso e poco remunerativo.

C'è bisogno di un modo per liberare il programmatore dalla gestione dell'espansione e della contrattura delle tabelle, analogamente al modo in cui la memoria virtuale elimina la preoccupazione di dover organizzare i programmi in overlay.

Una soluzione semplice è prevedere molti spazi degli indirizzi completamente indipendenti, detti segmenti. Ogni segmento consiste in una sequenza lineare d'indirizzi, che va da 0 a un valore massimo. La lunghezza di ciascun segmento può variare a sua volta da 0 a un massimo consentito: segmenti diversi potrebbero avere, e in genere hanno, lunghezze diverse. Per di più la lunghezza dei segmenti potrebbe cambiare durante l'esecuzione: la lunghezza del segmento dello stack potrebbe essere incrementata dopo ogni push e decrementata dopo ogni pop.

Poiché ciascun segmento costituisce uno spazio separato d'indirizzi, i diversi segmenti possono crescere o decrescere indipendentemente, senza influenzarsi a vicenda. Se uno stack ha bisogno di più spazio per crescere all'interno del proprio segmento, allora può prenderselo perché non c'è nient'altro nel suo spazio degli indirizzi con cui possa collidere.

Ovviamente può sempre succedere che un segmento venga riempito completamente, ma accade molto raramente, perché la sua dimensione è di solito molto grande. Per specificare un indirizzo di una memoria segmentata o bidimensionale, il programma deve fornire un indirizzo composto da due parti: un numero di segmento e un indirizzo all'interno del segmento. La Figura 6.8 illustra una memoria segmentata usata per le tabelle del compilatore illustrate in precedenza.

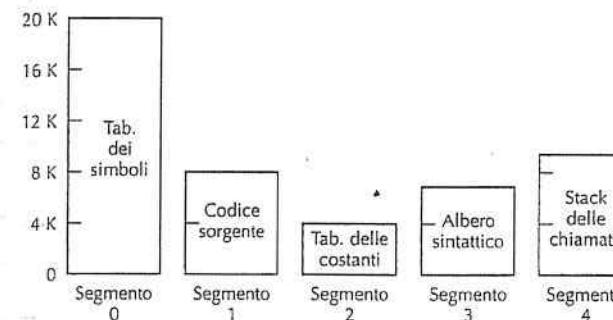


Figura 6.8 Una memoria segmentata consente a ciascuna tabella di crescere o decrescere indipendentemente dalle altre.

Sottolineiamo il fatto che un segmento è un'entità *logica* della cui esistenza il programmatore è consci e quindi la usa come una singola entità logica. Un segmento potrebbe contenere una procedura, un array, uno stack o una raccolta di variabili e in genere contiene oggetti di tipo omogeneo.

Una memoria segmentata presenta anche altri vantaggi oltre a semplificare la gestione delle strutture dati con dimensione variabile. Se ogni procedura occupa un segmento separato e inizia all'indirizzo 0, il collegamento delle procedure compilate separatamente risulta grandemente agevolato. Quando tutte le procedure di un programma sono state compilate e collegate, la chiamata di una procedura del segmento *n* userà l'indirizzo (*n*, 0) per indirizzare la parola 0 (il punto d'ingresso).

Se poi la procedura del segmento *n* viene modificata e ricompilata, non c'è bisogno di modificare le altre procedure (perché il loro indirizzo iniziale non è stato modificato), neanche nel caso in cui la nuova versione sia più grande della vecchia. In una memoria unidimensionale le procedure sono di norma impacchettate una di seguito all'altra, senza spazio tra di loro, perciò la modifica delle dimensioni di una procedura può causare la variazione dell'indirizzo iniziale di altre procedure indipendenti. La cosa implica

la modifica di tutte le procedure che richiamano una di quelle spostate, per memorizzare il loro nuovo indirizzo iniziale. Questo procedimento diventa costoso su programmi composti da centinaia di procedure.

La segmentazione facilita anche la condivisione di dati o procedure tra più programmi. Se un computer ha molti programmi in esecuzione parallela (vera o simulata) che usano tutti certe procedure di libreria, fornire a ciascuno una propria copia privata delle procedure risulterebbe un gran spreco di memoria principale. Se invece ogni procedura occupa un segmento separato, possono essere condivise facilmente, risparmiando così lo spazio di memoria richiesto da copie multiple.

Si è detto che ogni segmento forma un'entità logica visibile al programmatore, come una procedura, un array o uno stack, dunque è possibile definire modalità di protezione diverse su segmenti diversi. Un segmento di procedura potrebbe essere specificato come solo eseguibile, proibendone accessi in lettura o scrittura. Un array di numeri in virgola mobile potrebbe essere impostato come accessibile in lettura/scrittura, ma non in esecuzione, così che verrebbero rilevati e negati eventuali tentativi di salto verso di esso. Questi tipi di protezione sono di grande aiuto nello scoprire errori di programmazione.

È importante rendersi conto del perché la protezione è concepibile in una memoria segmentata, ma non in una memoria paginata (cioè lineare). In una memoria segmentata l'utente conosce il contenuto di ogni segmento e di norma nessun segmento contiene sia una procedura sia uno stack, ma l'una o l'altro. Il fatto che ciascun segmento contenga un solo tipo di oggetto rende possibile l'attribuzione di una protezione appropriata al tipo in questione. La Figura 6.9 mette a confronto paginazione e segmentazione.

Domanda	Paginazione	Segmentazione
Il programmatore deve esserne consci?	No	Sì
Quanti spazi lineari d'indirizzi ci sono?	1	Molti
Lo spazio degli indirizzi virtuali può eccedere la dimensione della memoria?	Sì	Sì
È facile gestire le tabelle a dimensione variabile?	No	Sì
Perché è stata inventata questa tecnica?	Per simulare memorie grandi	Per fornire molteplici spazi d'indirizzi

Figura 6.9 Confronto tra paginazione e segmentazione.

Il contenuto di una pagina è, in un certo senso, accidentale. Il programmatore non ha alcun sentore della paginazione. Anche se in linea di principio sarebbe possibile usare una manciata di bit per ogni elemento della tabella delle pagine al fine di specificare il tipo di accesso consentito, in tal caso il programmatore dovrebbe tener traccia, nel proprio spazio degli indirizzi, dei punti di transizione tra pagine. Questo non va bene perché è proprio il genere di amministrazione che la paginazione vuole eliminare. L'utente di una memoria segmentata ha l'illusione che tutti i segmenti si trovino in memoria allo stesso tempo, perciò può referenziarli senza curarsi della loro gestione.

### 6.1.7 Implementazione della segmentazione

La segmentazione può essere implementata in due modi diversi: con lo swapping ("scambio") o con la paginazione. Nel primo schema, in ogni istante la memoria contiene un certo numero di segmenti. Se viene fatto un riferimento a un segmento che non si trova correntemente in memoria, allora viene caricato. Se non c'è abbastanza spazio per accoglierlo, bisogna prima scrivere uno o più segmenti su disco (a meno che esista già una copia pulita su disco, nel qual caso la copia in memoria viene semplicemente abbandonata). In un certo qual modo lo swapping di segmenti non è dissimile dalla paginazione su richiesta: i segmenti sono caricati una volta richiesti.

Tuttavia, l'implementazione della segmentazione differisce dalla paginazione in modo essenziale: le pagine hanno dimensione fissa, i segmenti no. La Figura 6.10(a) mostra un esempio di memoria fisica che contiene inizialmente cinque segmenti. Si consideri ora che cosa succede quando il segmento 1 viene estromesso e viene caricato al suo posto il segmento 7, che è più piccolo. Ci troviamo nella configurazione di memoria della Figura 6.10(b). Lo spazio tra il segmento 7 e 2 è area non utilizzata, cioè una lacuna. Poi il segmento 4 è rimpiazzato dal segmento 5 (Figura 6.10(c)) e il segmento 3 dal 6 (Figura 6.10(d)). Dopo un po' di tempo la memoria si troverà divisa in segmenti e lacune. Questo fenomeno si chiama frammentazione esterna (perché lo spazio sprecato è al di fuori dei segmenti). Alle volte ci si riferisce alla frammentazione esterna con il termine di checkerboarding, cioè "comportamento a scacchiera".

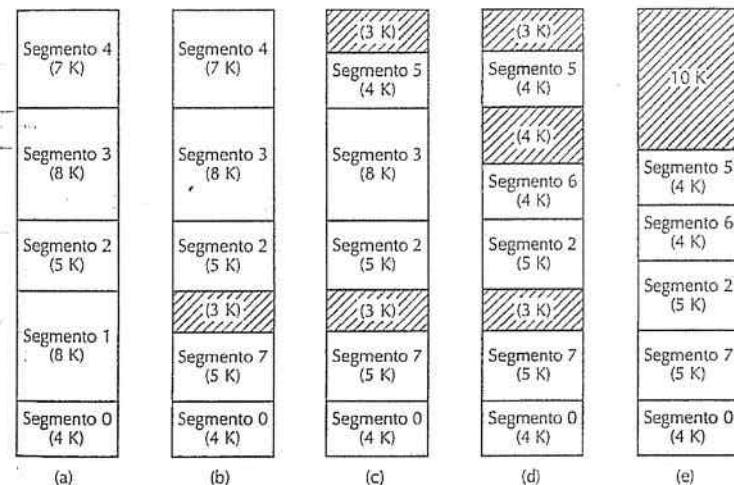


Figura 6.10 (a)-(d) Incremento della frammentazione esterna. (e) Rimozione della frammentazione esterna mediante compattamento.

Si consideri ciò che si verificherebbe se il programma accedesse al segmento 3 nel momento in cui la memoria è affetta da frammentazione esterna, come nella Figura 6.10(d). Lo spazio globalmente libero è 10 K, più di quanto serva al segmento 3, ma poiché è distribuito in frammenti piccoli e inutili, il segmento 3 non può essere caricato. Bisognerebbe rimuovere un altro segmento.

C'è un modo per evitare la frammentazione esterna: ogni volta che appare una lacuna, si può spostare il segmento che segue la lacuna in direzione della locazione di memoria 0, eliminando così quella lacuna, ma creandone una grande alla fine del segmento. In alternativa si potrebbe aspettare fino al momento in cui la frammentazione esterna diventi grave (per esempio quando eccede una certa percentuale della memoria totale) prima di eseguire un compattamento (spostando le lacune lontano dai segmenti). La Figura 6.10(e) mostra la memoria della Figura 6.10(d) dopo compattamento. Lo scopo del compattamento della memoria (a volte chiamato *garbage collection*, cioè "raccolta della spazzatura") è di compattare tutte le piccole e inutilizzabili lacune in una sola, in cui poter caricare uno o più segmenti. Il compattamento presenta l'inconveniente ovvio di portar via del tempo ed eseguirlo dopo la creazione di ogni singola lacuna, consumando troppe risorse.

Se il tempo di compattamento è così lungo da risultare inaccettabile, si rende necessario trovare un algoritmo per determinare quale lacuna usare per caricare un particolare segmento. Tale algoritmo richiede il mantenimento di una lista degli indirizzi e della dimensione delle lacune. Un algoritmo diffuso, chiamato *best fit* ("miglior corrispondenza"), sceglie la lacuna più piccola che può contenere il segmento, nel tentativo di far corrispondere un segmento a ogni lacuna per evitare di frammentarne una che potrebbe servire per ospitare un segmento grande.

Un altro algoritmo molto usato, *first fit* ("prima corrispondenza"), scandisce circolarmente la lista delle lacune e sceglie la prima in grado di ospitare il segmento. Naturalmente così facendo impiega meno tempo di quanto sia necessario alla scansione dell'intera lista operata da *best fit*. Sorprende invece sapere che *first fit* è un algoritmo migliore di *best fit* anche in termini di prestazioni complessive, dal momento che *best fit* tende a generare moltissime lacune totalmente inservibili (Knuth, 1997).

*First fit* e *best fit* tendono a diminuire la dimensione media delle lacune. Ogni volta che un segmento viene caricato in una lacuna più grande di lui, il che capita praticamente sempre (le corrispondenze esatte sono rare), la lacuna viene divisa in due parti: una parte è occupata dal segmento, e l'altra costituisce una nuova lacuna, più piccola di quella originaria. A meno di un processo di compensazione che ricreia lacune più grandi a partire da quelle piccole, alla lunga sia *first fit* sia *best fit* tendono a creare in memoria molte lacune inutilizzabili.

Un meccanismo di compensazione è il seguente. Quando viene rimosso un segmento dalla memoria e uno o entrambi i suoi vicini sono lacune e non segmenti, le lacune adiacenti possono saldarsi in una sola. Se venisse rimosso il segmento 5 dalla Figura 6.10(d), le lacune circostanti e i 4 KB usati dal segmento verrebbero fusi in una sola lacuna di 11 KB.

All'inizio di questo paragrafo si è detto che ci sono due modi di implementare la segmentazione: con lo swapping o con la paginazione. La trattazione ha fin qui riguardato lo swapping. Secondo questo schema i segmenti vengono spostati dalla memoria al

disco su richiesta. L'altro modo di implementare la segmentazione è di suddividere ogni segmento in un numero prefissato di pagine e svolgere la loro paginazione su richiesta. Secondo questo schema alcune delle pagine del segmento potranno trovarsi in memoria, altre sul disco. La paginazione dei segmenti richiede una tabella delle pagine per ogni segmento. Non essendo un segmento altro che uno spazio lineare degli indirizzi, tutte le tecniche viste fin qui per la paginazione si applicano direttamente a ogni segmento. L'unica novità è la presenza di una tabella delle pagine per ciascun segmento.

Uno dei primi sistemi operativi a combinare la segmentazione con la paginazione fu MULTICS (*MULTplexed Information and Computing Service*), nato come sforzo congiunto di M.I.T., dei Laboratori Bell e della General Electric (Corbató e Vyssotsky, 1965; Organick, 1972). Gli indirizzi di MULTICS erano formati da due parti: un numero di segmento e un indirizzo all'interno del segmento. C'era un segmento descrittore per ogni processo, contenente a sua volta un descrittore per ciascun segmento. Quando l'hardware riceveva un indirizzo virtuale, il numero di segmento era usato come indice nel segmento descrittore per localizzare il descrittore del segmento cui accedere, come mostrato nella Figura 6.11. Il descrittore puntava alla tabella delle pagine, rendendo possibile la paginazione di ogni segmento nelle modalità usuali. Al fine di incrementare le prestazioni, una memoria associativa hardware di 16 bit era usata per contenere le combinazioni segmento/pagina usate più di recente, in modo da potervi accedere velocemente. Anche se MULTICS è oggi estinto, ha avuto una vita molto lunga, dal 1965 al 30 ottobre 2000, data in cui l'ultimo sistema MULTICS è stato spento. Pochi altri sistemi operativi hanno resistito per 35 anni. Inoltre, il suo spirito gli è sopravvissuto e le memorie virtuali di tutte le CPU Intel a partire dal 386 sono state modellate sul suo esempio. La storia di MULTICS, insieme ad altri aspetti del sistema, è descritta sul sito [www.multicians.org](http://www.multicians.org).

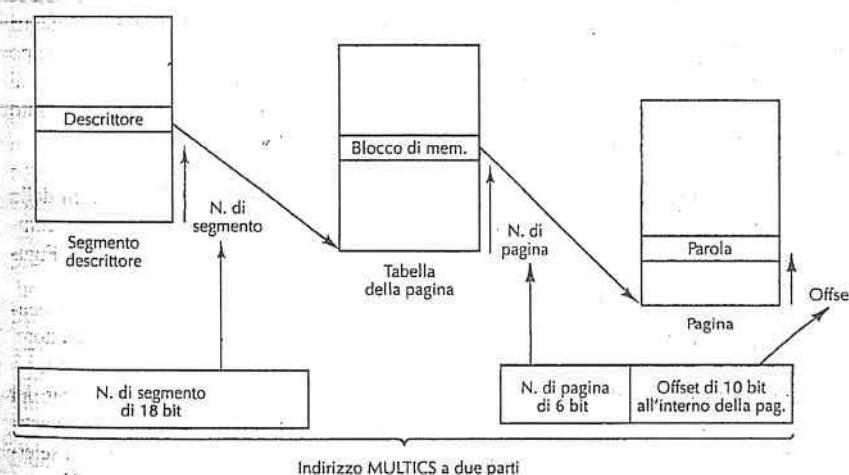


Figura 6.11 Conversione di un indirizzo MULTICS a due parti in un indirizzo fisico.

### 6.1.8 Memoria virtuale del Core i7

Il Core i7 è fornito di un sistema sofisticato di memoria virtuale che supporta la paginazione a richiesta, la segmentazione pura e la segmentazione paginata. Nel nucleo della memoria virtuale del Core i7 ci sono due tabelle: LDT (*Local Descriptor Table*, “tabella dei descrittori locali”) e GDT (*Global Descriptor Table*, “tabella dei descrittori globali”). Ciascun programma ha la propria LDT, ma l'unica GDT è condivisa da tutti i programmi. LDT descrive i segmenti locali a ogni programma, tra cui il suo codice, i dati, lo stack e così via, mentre GDT descrive i segmenti di sistema, compreso il sistema operativo stesso.

Come già descritto nel Capitolo 5, per accedere a un segmento un programma Core i7 deve caricare un selettori per quel segmento in uno dei suoi registri di segmento. Durante l'esecuzione, CS contiene il selettori per il segmento di codice, DS quello per il segmento di dati e così via. Ogni selettori è un numero di 16 bit, come illustrato nella Figura 6.12.

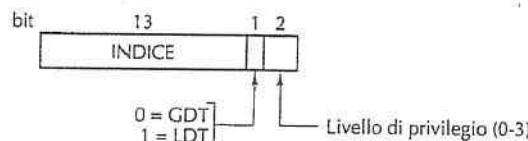


Figura 6.12 Selettori del Core i7.

Un bit del selettori specifica se il segmento è locale oppure globale (cioè se si trova in LDT o in GDT). Altri 13 bit specificano il numero dell'elemento di LDT o di GDT, perciò entrambe le tabelle possono contenere 8 KB ( $2^{13}$ ) descrittori di segmenti. Gli altri 2 bit riguardano la protezione e verranno trattati in seguito. Il descrittore 0 non è valido e causa una trap se viene usato. È possibile caricarlo senza problemi in un registro di segmento per indicare che il registro non è al momento disponibile, ma se viene usato provoca comunque una trap.

Nell'istante in cui un selettori viene caricato in un registro di segmento, il descrittore corrispondente è recuperato da LDT o GDT e memorizzato nei registri interni della MMU, dove può essere referenziato velocemente. Un descrittore è fatto di 8 byte, compreso l'indirizzo della base del segmento, la sua dimensione e altre informazioni (Figura 6.13).

Il formato del selettori è stato concepito in modo intelligente per facilitare la localizzazione del descrittore. Prima si sceglie tra LDT e GDT, in base al bit 2 del selettori, quindi si copia il selettori in un registro di lavoro della MMU e ne vengono azzerati i 3 bit meno significativi, il che equivale a moltiplicare per otto il numero di 13 bit del selettori. Infine gli si somma l'indirizzo della tabella LDT o GDT (tenuto nei registri interni della MMU) per ottenere un puntatore diretto al descrittore. Per esempio, il selettor 72 fa riferimento all'elemento 9 di GDT, che si trova all'indirizzo GDT + 72.

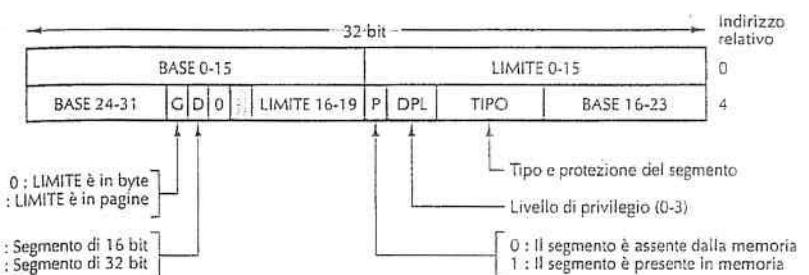


Figura 6.13 Descrittore di un segmento di codice nel Core i7. I segmenti di dati sono leggermente diversi.

Seguiamo passo dopo passo il meccanismo di conversione di una coppia (selettor, offset) in un indirizzo fisico.

Non appena stabilito il registro di segmento da utilizzare, l'hardware può trovare il descrittore completo corrispondente al selettori memorizzato nel registro interno. Se il segmento non esiste (selettori 0), oppure se non si trova correntemente in memoria (P vale 0), si verifica una trap. La prima eventualità è un errore di programmazione, la seconda richiede l'intervento del sistema operativo per recuperare il segmento.

Dopo di ciò, l'hardware verifica se l'offset eccede la fine del segmento e, in caso affermativo, si ha un'altra trap. Secondo logica, il descrittore dovrebbe contenere semplicemente un campo di 32 bit per specificare la dimensione del segmento, ma essendo solo 20 bit disponibili, si usa uno schema differente. Se il bit G di granularità vale 0, allora il campo LIMITE rappresenta la dimensione esatta del segmento, fino a un massimo di 1 MB. Se invece G vale 1, il campo LIMITE restituisce la dimensione del segmento in pagine invece che in byte. La dimensione di pagina nel Core i7 non è mai minore di 4 KB, perciò bastano 20 bit per segmenti grandi fino a  $2^{32}$  byte.

Ipotizziamo che il segmento si trovi in memoria e che l'offset sia corretto. In tal caso il Core i7 somma il campo BASE di 32 bit del descrittore all'offset, per formare quello che è detto un indirizzo lineare, mostrato nella Figura 6.14. Il campo BASE è suddiviso in tre parti, sparpagliate all'interno del descrittore per motivi di compatibilità con l'i80286, il cui campo BASE è di soli 24 bit. La lunghezza del campo BASE permette a un segmento di cominciare in una locazione qualsiasi dello spazio degli indirizzi lineari di 32 bit. Se la paginazione è disabilitata (da un bit del registro globale di controllo) l'indirizzo lineare è interpretato come indirizzo fisico e viene quindi inviato alla memoria per la sua lettura o scrittura. Senza la paginazione abbiamo dunque uno schema di segmentazione pura, dove il descrittore di un segmento ne individua l'indirizzo base. È consentito ai segmenti di sovrapporsi accidentalmente, forse perché la verifica che siano tutti disgiunti richiederebbe troppo tempo e comporterebbe eccessive complicazioni.

D'altra parte, se la paginazione è abilitata, l'indirizzo lineare è interpretato come un indirizzo virtuale e viene trasformato nell'indirizzo fisico tramite le tabelle delle pagine, come già visto. La sola complicazione qui è che, con indirizzi virtuali di 32 bit e pagine

di 4 KB, è possibile avere segmenti da un milione di pagine, perciò viene usata una corrispondenza a due livelli per ridurre la dimensione della tabella delle pagine nel caso di segmenti piccoli.

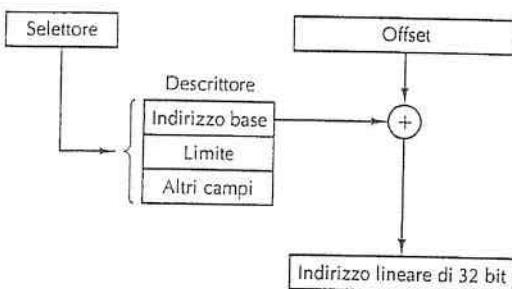


Figura 6.14 Conversione di una coppia (selettore, offset) in un indirizzo lineare.

Ogni programma in esecuzione ha una **directory delle pagine** costituita da 1024 elementi di 32 bit e localizzata presso un indirizzo puntato da un registro globale. Ogni elemento di questa directory punta a una **tabella delle pagine** costituita anch'essa di 1024 elementi di 32 bit. Gli elementi della tabella delle pagine puntano ai **blocchi di memoria**. Lo schema è quello della Figura 6.15.

La Figura 6.15(a) raffigura un indirizzo lineare suddiviso in tre campi: **DIR**, **PAGINA** e **OFF**. Il campo **DIR** è usato per primo per indicizzare la directory delle pagine al fine di trovare il puntatore alla tabella delle pagine appropriata. Poi viene usato il campo **PAGINA** come indice nella tabella delle pagine per trovare l'indirizzo del blocco di memoria. Infine **OFF** è sommato all'indirizzo del blocco di memoria per formare l'indirizzo fisico del byte o della parola cui accedere.

Gli elementi della tabella delle pagine sono tutti di 32 bit, di cui 20 per il numero del blocco di memoria. I bit restanti contengono bit di accesso, un dirty bit (asserito dall'hardware a beneficio del sistema operativo se il blocco è stato modificato dopo il caricamento), bit di protezione e altri bit.

Ogni tabella delle pagine contiene elementi per 1024 blocchi di memoria di 4 KB, perciò una sola tabella delle pagine raggiunge 4 MB di memoria. Ai segmenti più piccoli di 4 MB basta una directory delle pagine con un solo elemento, cioè il puntatore alla sua unica tabella delle pagine. Così facendo l'informazione accessoria per i segmenti piccoli è di sole due pagine, invece dei milioni di pagine che sarebbero richieste in una tabella delle pagine a un solo livello.

Per evitare di accedere ripetutamente alla memoria, la MMU del Core i7 ha un hardware speciale per la ricerca veloce delle combinazioni **DIR–PAGINA** usate più di recente, e le mappa negli indirizzi fisici dei blocchi di memoria corrispondenti. I passi illustrati nella Figura 6.15 vengono svolti effettivamente solo quando la combinazione corrente non sia stata usata di recente.

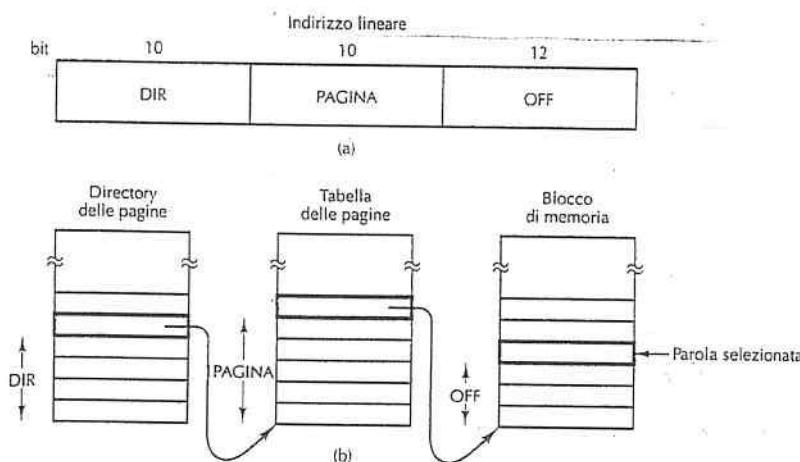


Figura 6.15 Corrispondenza tra un indirizzo lineare e un indirizzo fisico.

Non c'è alcuna ragione per avere un valore diverso da 0 nel campo **BASE** del descrittore quando si usa la paginazione. Ciò è dovuto alla semplice osservazione che l'effetto del campo **BASE** è di accedere a un elemento nel mezzo della directory delle pagine che si trova a un certo offset dall'elemento iniziale, invece che all'elemento iniziale stesso. Le uniche ragioni che giustificano l'esistenza del campo **BASE** sono consentire la segmentazione pura (non paginata) e garantire la retrocompatibilità con il vecchio 80286, privo di paginazione.

Vale la pena menzionare il fatto che, se un'applicazione non ha bisogno della segmentazione e richiede un singolo spazio degli indirizzi di 32 bit paginato, è facile soddisfare la sua richiesta. Basta impostare tutti i registri di segmento allo stesso selettore, il cui descrittore ha **BASE = 0** e **LIMITE = massimo**. Se si usa un solo spazio degli indirizzi, l'offset dell'istruzione diviene l'indirizzo lineare e si ottiene in effetti la paginazione tradizionale.

Abbiamo così concluso la trattazione della memoria virtuale del Core i7. Abbiamo visto solo una piccola parte (usata comunemente) del sistema di memoria virtuale del Core i7. Il lettore interessato può approfondire le sue conoscenze con la documentazione del Core i7 e scoprire anche le estensioni a 64 bit degli indirizzi virtuali e il supporto per gli spazi degli indirizzi fisici virtualizzati. Prima di abbandonare l'argomento, resta da aggiungere qualche osservazione a proposito della protezione, essendo questa una materia intimamente legata alla memoria virtuale. Il Core i7 supporta quattro livelli di protezione, da 0 (massimi privilegi) a 3, illustrati nella Figura 6.16. In ogni momento un programma in esecuzione si trova a un certo livello di protezione, indicato da un campo di 2 bit nel registro hardware **PSW (Program Status Word)**, contenente i codici di condizione e svariati altri bit di stato. Inoltre, ogni segmento del sistema si trova a un certo livello di protezione.

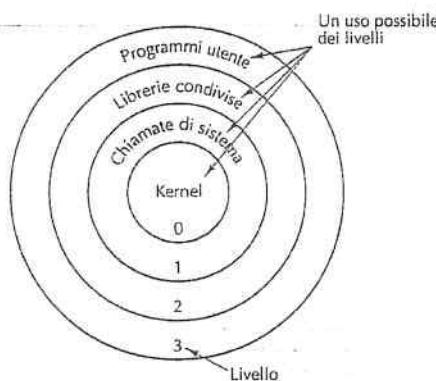


Figura 6.16 Protezione nel Core i7.

Fintantoché un programma si limita a usare segmenti al suo stesso livello, non ci sono problemi. Sono consentiti i tentativi di accesso a dati che si trovano a un livello più alto, mentre i tentativi di accesso a dati a livello più basso sono vietati e causano una trap. È consentito richiamare procedure a livelli diversi (più alti o più bassi), benché in modo severamente controllato. Un’istruzione *CALL* che voglia effettuare una chiamata tra livelli deve contenere un selettore invece di un indirizzo. Questo selettore designa un descrittore che si chiama *call gate* (“accesso alle chiamate”) e che restituisce l’indirizzo della procedura da invocare. Così facendo si nega la possibilità di saltare nel mezzo di un segmento di codice di livello diverso; è possibile accedere solo ai punti d’ingresso ufficiali.

Un uso possibile di questo meccanismo è suggerito dalla Figura 6.16. A livello 0 troviamo il kernel del sistema operativo, che gestisce l’I/O, la memoria e altre questioni critiche. A livello 1 troviamo il gestore delle chiamate di sistema. I programmi utente possono richiamare procedure a questo livello perché svolgono chiamate di sistema, ma solo quelle appartenenti a una lista specifica di procedure protette. Il livello 2 contiene le procedure di libreria, eventualmente condivise tra molti programmi in esecuzione. I programmi utente le possono richiamare, ma non modificare. Infine i programmi utente girano a livello 3, cui è associata la protezione minima. Al pari dello schema di gestione della memoria, anche il sistema di protezione del Core i7 è ispirato fortemente a quello di MULTICS.

Le trap e gli interrupt usano un procedimento simile a quello dei call gate: anch’essi referenziano descrittori, invece d’indirizzi assoluti, e questi descrittori puntano alle specifiche procedure da eseguire. Il campo *TIPO* della Figura 6.13 opera la distinzione tra segmenti di codice e segmenti di dati, oltre che tra vari tipi di call gate.

### 6.1.9 Memoria virtuale della CPU ARM OMAP4430

La CPU ARM OMAP4430 è una macchina a 32 bit che supporta una memoria virtuale paginata basata su indirizzi virtuali di 32 bit che vengono tradotti in indirizzi fisici

di 32 bit. Una CPU ARM può dunque supportare fino a  $2^{32}$  byte (4 GB) di memoria fisica. Sono supportate quattro dimensioni di pagina: 4 KB, 64 KB, 1 MB e 16 MB. Le corrispondenze indotte da queste quattro dimensioni di pagina sono illustrate nella Figura 6.17.

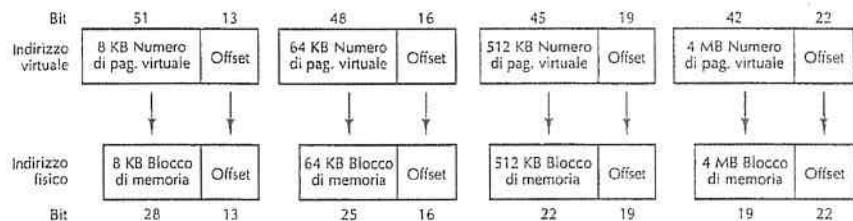


Figura 6.17 Corrispondenze tra indirizzi virtuali e fisici nella CPU ARM OMAP4430.

La CPU ARM OMAP4430 utilizza una struttura di tabella delle pagine simile a quella del Core i7. La mappatura della tabella delle pagine per una pagina degli indirizzi virtuali di 4 KB è mostrata nella Figura 6.18(a). La tabella dei descrittori di primo livello è indicizzata dai 12 bit più significativi dell’indirizzo virtuale. Una sua entry indica l’indirizzo fisico della tabella dei descrittori di secondo livello. Questo indirizzo, combinato con i successivi 8 bit dell’indirizzo virtuale, fornisce l’indirizzo del descrittore di pagina. Il descrittore di pagina contiene l’indirizzo del blocco di pagina fisica e altre informazioni sui permessi di accesso alla pagina.

La mappatura della memoria virtuale della CPU ARM OMAP4430 contempla quattro dimensioni di pagina. Le dimensioni di 1 e di 16 MB vengono mappate con un descrittore di pagina collocato nella tabella dei descrittori di primo livello. Infatti, non c’è bisogno delle tabelle di secondo livello, perché, in ognuna di queste, tutte le entry punterebbero alla stessa grande pagina fisica. I descrittori delle pagine di 64 KB sono posti nella tabella dei descrittori di secondo livello. Visto che ogni voce della tabella dei descrittori di secondo livello mappa 4 KB di pagina di indirizzi virtuali in una pagina degli indirizzi fisici di 4 KB, per le pagine da 64 KB sono richiesti 16 descrittori identici nella pagina dei descrittori di secondo livello. Perché dunque un programmatore di sistemi dovrebbe dichiarare una dimensione di pagina di 64 KB quando questo richiederebbe lo stesso spazio che si utilizza per mappare una più flessibile pagina di 4 KB? Perché, come vedremo tra poco, le pagine di 64 KB richiedono meno voci nella tabella TLB, che è una risorsa critica per ottenere buone prestazioni.

Niente rallenta un programma più del collo di bottiglia dovuto alla memoria. Osservando attentamente la Figura 6.18, noterete probabilmente che per ogni accesso del programma alla memoria sono richiesti due accessi aggiuntivi per la traduzione degli indirizzi virtuali renderebbe un programma eccessivamente lento. Per evitare questo collo di bottiglia la CPU ARM OMAP4430 incorpora una tabella hardware chiamata

TLB ("Translation Lookaside Buffer") che permette di stabilire rapidamente la corrispondenza tra numeri di pagina virtuale e numeri di blocco di pagina fisica. Per una dimensione di pagina di 4KB ci sono  $2^{20}$  numeri di pagina virtuale, ovvero più di un milione. Ovviamente non possono essere tutti mappati nella tabella e la TLB conterrà solo i numeri di pagina virtuale più recentemente usati.

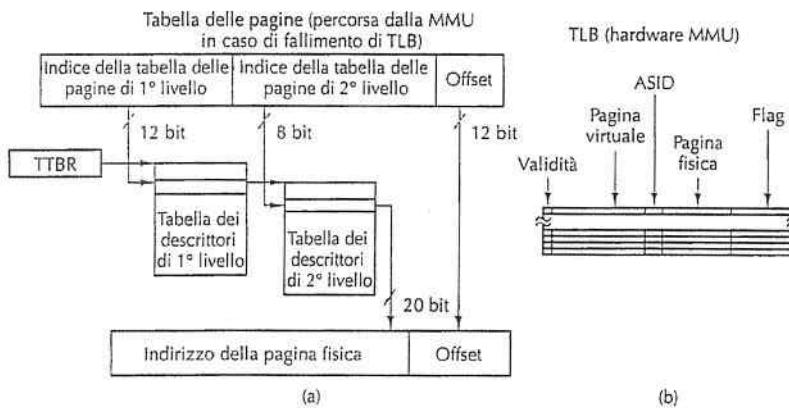


Figura 6.18 Strutture dati utilizzata dalla CPU ARM OMAP4430 per la traduzione degli indirizzi virtuali. (a) Tabella di traduzione degli indirizzi. (b) TLB.

La tracciatura delle istruzioni e dei dati avviene separatamente: la TLB contiene per ciascuna delle due categorie i 128 numeri di pagina virtuale usati più di recente. Ogni elemento di TLB contiene un numero di pagina virtuale e il numero del blocco di memoria fisico corrispondente. Alla ricezione di un numero di processo, detto **identificatore dello spazio degli indirizzi** (ASID), e di un indirizzo virtuale all'interno di quello spazio di indirizzi, la MMU usa una circuiteria speciale per confrontare il numero di pagina virtuale in esso contenuto con tutti gli elementi di TLB simultaneamente, al fine di trovare quel contesto. Se l'esito del confronto è positivo, si forma un indirizzo fisico di 32 bit, componendo il numero del blocco di memoria estratto da TLB con l'offset preso dall'indirizzo virtuale; in questa fase si impostano anche alcuni flag, compresi i bit di protezione. La TLB è illustrata nella Figura 6.18(b).

Se invece non si trova alcuna corrispondenza, si verifica un **fallimento di TLB**, che dà il via a un "cammino" hardware attraverso le tabelle delle pagine. Quando si trova il nuovo descrittore della pagina fisica nella tabella delle pagine si verifica se la pagina è in memoria e, se lo è, viene caricata nella TLB la corrispondente traduzione dell'indirizzo. Se la pagina non è in memoria viene avviata la gestione del page fault. Poiché la TLB è formata da un numero ridotto di voci, è probabile che si debba rimuovere una voce esistente per far spazio a una nuova voce. I futuri accessi alla pagina rimossa dovranno ancora percorrere le tabelle delle pagine per ottenere un indirizzo. Se si acce-

de a molte pagine troppo velocemente, la TLB non riuscirà a svolgere il suo compito e la maggior parte degli accessi alla memoria richiederanno un overhead del 200% per la traduzione degli indirizzi.

È interessante confrontare i sistemi di memoria virtuale di Core i7 e OMAP4430. Il Core i7 supporta la segmentazione pura, la paginazione pura e la segmentazione paginata. La CPU ARM OMAP4430 dispone solo della paginazione. Sia il Core i7 che l'OMAP4430 utilizzano l'hardware per percorrere la tabella delle pagine al fine di ricaricare la TLB in caso di fallimento di accesso. Altre architetture, come SPARC e MIPS, in caso di fallimento di TLB cedono il controllo al sistema operativo. Queste ultime architetture definiscono istruzioni con privilegi speciali per la manipolazione della TLB, in modo che il sistema operativo possa percorrere la tabella delle pagine e caricare nella TLB i valori necessari per tradurre gli indirizzi.

### 6.1.10 Memoria virtuale e caching

La memoria virtuale con paginazione su richiesta e il caching possono sembrare a un primo sguardo completamente diversi, tuttavia sono molto simili dal punto di vista concettuale. Con la memoria virtuale il programma è mantenuto su disco e suddiviso in pagine di dimensione fissa; solo alcune di queste pagine si trovano in memoria principale. Se il programma usa prevalentemente le pagine in memoria si verificano pochi errori di pagina e il programma viene eseguito velocemente. Con il caching, il programma è contenuto tutto in memoria e suddiviso in blocchi di cache di dimensione prefissata, alcuni dei quali si trovano nella cache. Se il programma usa prevalentemente i blocchi di cache, allora ci saranno pochi fallimenti di cache e l'esecuzione procederà velocemente. Dal punto di vista concettuale i due meccanismi sono la stessa cosa, ma operano a due diversi livelli gerarchici.

Naturalmente la memoria virtuale e il caching presentano anche alcune differenze. Una è che i fallimenti di cache sono gestiti dall'hardware, mentre gli errori di pagina sono gestiti dal sistema operativo. Inoltre i blocchi di cache sono in genere molto più piccoli delle pagine (per esempio 64 byte contro 8 KB). Si aggiunga che la corrispondenza tra pagine virtuali e blocchi di memoria è diversa: le tabelle delle pagine sono organizzate per essere indicizzate dai bit più significativi dell'indirizzo virtuale, mentre le cache sono indicizzate dai bit meno significativi degli indirizzi di memoria. Nonostante ciò, è importante capire che queste sono differenze di carattere implementativo, il concetto sottostante è molto simile.

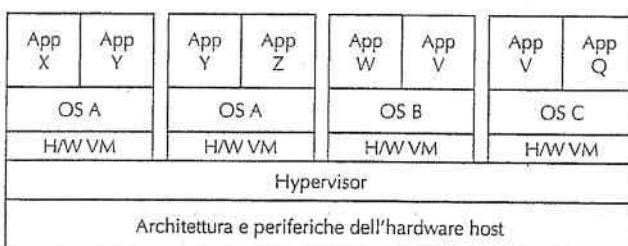
## 6.2 Virtualizzazione hardware

Tradizionalmente le architetture hardware sono state progettate con la prospettiva di dover eseguire un solo sistema operativo alla volta. La proliferazione di risorse condivise, come i servizi cloud, trae beneficio dalla possibilità di eseguire più sistemi operativi contemporaneamente. Per esempio, i servizi di hosting di Internet forniscono ai clienti paganti un sistema completo sul quale possono essere installati servizi web. Installare un nuovo computer nella sala macchine ogni volta che arriva un nuovo clien-

te avrebbe costi proibitivi. I servizi di hosting utilizzano piuttosto la virtualizzazione, per supportare l'esecuzione di diversi sistemi completi, compresi i sistemi operativi, sullo stesso server. Solo quando i server esistenti raggiungono un carico eccessivo il servizio di hosting deve installare una nuova macchina fisica.

Anche se esistono approcci software alla virtualizzazione, questi di solito rallentano il sistema virtuale e richiedono modifiche al sistema operativo o l'utilizzo di complessi analizzatori di codice per riscrivere i programmi in tempo reale. Questi svantaggi hanno suggerito ai progettisti l'idea di dover supportare direttamente la **virtualizzazione hardware**.

Questa tecnica, come mostrato nella Figura 6.19, è una combinazione di supporti hardware e software che permette la simultanea esecuzione di diversi sistemi operativi su una singola macchina fisica. Ogni macchina virtuale eseguita sulla macchina fisica appare all'utente come un sistema completo e indipendente. L'**hypervisor** è un componente software, simile al kernel di un sistema operativo, che crea e gestisce le istanze di macchine virtuali. L'hardware fornisce gli eventi visibili al software che sono necessari all'hypervisor per implementare le politiche di gestione (policy) per la CPU, la memoria di massa e i dispositivi di I/O.



**Figura 6.19** La virtualizzazione hardware permette diversi sistemi operativi in esecuzione simultanea sulla stessa hardware. L'hypervisor implementa la condivisione della memoria e dei dispositivi di I/O della macchina host.

La coesistenza di diverse macchine virtuali sulla stessa macchina fisica (il computer host), ognuna delle quali esegue un sistema operativo potenzialmente diverso, offre numerosi benefici. Nei sistemi server la virtualizzazione dà agli amministratori di sistema la capacità di installare più macchine virtuali sulla stessa macchina fisica e di spostare le macchine virtuali tra diversi server per distribuire al meglio il carico totale. Le macchine virtuali danno inoltre all'amministratore di sistema la possibilità di un controllo capillare sull'accesso all'I/O. Per esempio, la larghezza di banda di un'interfaccia di rete virtualizzata può essere suddivisa basandosi sul livello dei servizi utenti. Ai singoli utenti la virtualizzazione è in grado di offrire la possibilità di eseguire diversi sistemi operativi in contemporanea.

Per l'implementazione della virtualizzazione hardware tutte le istruzioni dell'architettura devono accedere soltanto alle risorse della macchina virtuale corrente. Per la

maggior parte delle istruzioni si tratta di una richiesta banale. Per esempio, le istruzioni aritmetiche hanno soltanto bisogno di accedere al banco dei registri che può essere virtualizzato copiando i registri della macchina virtuale nel banco dei registri del processore host al cambio di contesto della macchina virtuale.

La virtualizzazione delle istruzioni per l'accesso alla memoria (per esempio delle istruzioni di caricamento e memorizzazione) è leggermente più complicata, perché queste istruzioni devono effettivamente accedere alla memoria fisica allocata alla macchina virtuale correntemente in esecuzione. Di solito un processore che supporta la virtualizzazione hardware offre funzionalità aggiuntive per la mappatura delle pagine che permettono la mappatura di pagine di memoria fisica della macchina virtuale in pagine di memoria fisica della macchina fisica.

Infine, le istruzioni di I/O (incluso l'I/O memory/mapped) non accedono direttamente ai dispositivi fisici di I/O, perché le politiche di virtualizzazione prevedono una suddivisione dell'accesso ai dispositivi. Questo controllo capillare dell'I/O viene tipicamente implementato mediante interrupt che vengono inviati all'hypervisor ogni volta che una macchina virtuale tenta l'accesso a un dispositivo di I/O. A questo punto l'hypervisor può implementare le politiche di accesso alle risorse a sua discrezione. Di solito, viene supportato un sottoinsieme dei dispositivi di I/O e ci si aspetta che il sistema operativo in esecuzione sulla macchina virtuale, chiamato sistema operativo ospite (guest), utilizzi questi dispositivi supportati.

### 6.2.1 Virtualizzazione hardware nel Core i7

La virtualizzazione hardware nel Core i7 è supportata grazie alle estensioni VMX (*virtual machine extensions*), una combinazione di estensioni di istruzioni, memoria e interrupt che permette una gestione efficace delle macchine virtuali. Con VMX, la virtualizzazione della memoria è implementata per mezzo del sistema EPT (*Extended Page Table*) che viene abilitato con la virtualizzazione hardware. La tabella EPT traduce gli indirizzi fisici delle pagine della macchina virtuale negli indirizzi fisici delle pagine della macchina host. Questa mappatura è realizzata con una struttura multilivello di tabelle di pagina che viene attraversata in caso di fallimento di accesso alla TLB. L'hypervisor è il manutentore di questa tabella e può quindi implementare la politica di condivisione della memoria fisica desiderata.

La virtualizzazione delle operazioni di I/O, sia per l'I/O memory/mapped che per le istruzioni di I/O, è implementata per mezzo del supporto esteso agli interrupt definito dalla VMCS (*Virtual-Machine Control Structure*). Ogni volta che una macchina virtuale accede a un dispositivo di I/O viene inviato un interrupt all'hypervisor. Quando l'hypervisor riceve l'interrupt può realizzare l'operazione di I/O via software, adottando le politiche necessarie che permettono la condivisione dei dispositivi tra le macchine virtuali.

## 6.3 Istruzioni di I/O a livello OSM

L'insieme d'istruzioni del livello ISA è completamente diverso dall'insieme d'istruzioni della microarchitettura. Differiscono sia le operazioni disponibili ai due livelli, sia i

formati delle istruzioni. L'esistenza di poche istruzioni sostanzialmente invariate da un livello all'altro va considerata come puramente accidentale.

L'insieme d'istruzioni del livello OSM contiene invece molte delle istruzioni del livello ISA, più un numero limitato d'istruzioni importanti, meno una manciata d'istruzioni potenzialmente nocive. L'input/output è uno dei campi in cui i due livelli differiscono maggiormente e la ragione è semplice: un utente che fosse in grado di eseguire le vere istruzioni di I/O del livello ISA potrebbe leggere dati confidenziali memorizzati in qualsiasi locazione del sistema, potrebbe scrivere sui terminali degli altri utenti e, in generale, costituirebbe una minaccia per la sicurezza del sistema. In secondo luogo, nessun programmatore dotato di senso vorrebbe usare l'I/O del livello ISA, perché è estremamente noioso e complicato. Si tratta di impostare campi e bit in vari registri dei dispositivi, attendere il completamento delle operazioni e quindi verificarne l'esito. Un esempio di ciò sono i bit del registro di dispositivo dei dischi, usati in genere per rilevare gli errori seguenti (tra i tanti possibili):

1. il braccio del disco non è riuscito a portare a termine la ricerca;
2. il buffer specifica locazioni di memoria inesistenti;
3. l'I/O del disco è cominciato prima che venisse completata un'altra attività di I/O già iniziata;
4. errore di temporizzazione della lettura;
5. riferimento a un disco inesistente;
6. riferimento a un cilindro inesistente;
7. riferimento a un settore inesistente;
8. errore di checksum del dato letto;
9. errore di verifica dei dati dopo scrittura.

Quando si verifica uno di questi errori viene asserito il bit corrispondente nel registro di dispositivo. Sono pochi gli utenti che vogliono preoccuparsi di tener traccia di tutti questi bit di errore e delle altre informazioni di stato.

### 6.3.1 File

Un modo di organizzare l'I/O virtuale è di usare l'astrazione chiamata file. Nella sua forma più semplice, un file consiste in una sequenza di byte scritti su di un dispositivo di I/O. Se il dispositivo di I/O è una memoria di massa, quale un disco, il file può essere letto successivamente; se non si tratta di una memoria di massa (ma per esempio di una stampante) chiaramente il file non può più essere letto. Un disco può contenere molti file e ogni file può essere di natura diversa: un'immagine, un foglio di calcolo oppure il testo del capitolo di un libro. File diversi hanno lunghezze e proprietà distinte. L'astrazione del file permette un'organizzazione semplice dell'I/O.

Dal punto di vista del sistema operativo un file è semplicemente una sequenza di byte. Ogni ulteriore strutturazione riguarda i programmi applicativi. L'I/O dei file è svolto tramite chiamate di sistema per l'apertura, lettura, scrittura e chiusura di file.

Prima di poter essere letti i file devono essere aperti. Il procedimento di apertura di un file consente al sistema operativo di localizzarlo su disco e di caricare in memoria le informazioni necessarie per accedervi.

La chiamata di sistema per la lettura deve avere almeno i seguenti parametri:

1. un'indicazione di quale file leggere tra quelli aperti;
2. un puntatore a un buffer in memoria per immagazzinare i dati letti;
3. il numero di byte da leggere.

La chiamata read pone i dati richiesti nel buffer e in genere restituisce il numero di byte letti effettivamente (tale numero potrebbe essere inferiore al numero di byte richiesti: non è possibile leggere 2000 byte da un file di 1000 byte).

A ogni file aperto è associato un puntatore che indica il successivo byte da leggere. Dopo una lettura il puntatore è incrementato secondo il numero di byte letti, perciò letture consecutive leggono blocchi di dati consecutivi del file. In genere è possibile assegnare a questo puntatore un valore specifico, di modo da accedere direttamente a ogni locazione del file. Quando un programma ha finito di leggere un file può chiuderlo per informare il sistema operativo che non gli serve più, consentendogli così di liberare lo spazio occupato nella tabella delle informazioni sul file.

I mainframe vengono ancora utilizzati (specialmente per gestire siti di commercio elettronico di grandi dimensioni) e alcuni di questi eseguono sistemi operativi tradizionali (anche se la maggior parte utilizza Linux). I sistemi operativi dei mainframe hanno una nozione più sofisticata di file e vale la pena dare un veloce sguardo a questo modello per mostrare che il metodo UNIX non è l'unico possibile. Per questi sistemi tradizionali un file è una sequenza di record logici, ciascuno dotato di una struttura ben definita. Un record logico, per esempio, potrebbe essere una struttura di cinque oggetti: due stringhe di caratteri, "nome" e "supervisore", due interi, "dipartimento" e "ufficio" e un valore booleano, "diSessoFemminile". Alcuni sistemi operativi fanno una distinzione tra file contenenti record con la stessa struttura e file contenenti record di tipo diverso.

L'istruzione elementare di input legge il record successivo del file specificato e lo mette in memoria principale all'indirizzo indicato, come illustrato nella Figura 6.20. Per svolgere questa operazione, l'istruzione virtuale deve sapere quale file leggere e dove riportare il record in memoria. Spesso si danno opzioni per leggere un determinato record, specificato o per posizione o tramite una chiave.

L'istruzione elementare di output scrive un record logico dalla memoria in un file. Scritture sequenziali consecutive producono nel file record logici consecutivi.

### 6.3.2 Implementazione delle istruzioni di I/O a livello OSM

Per comprendere l'implementazione delle istruzioni virtuali di I/O è necessario esaminare le modalità di organizzazione e memorizzazione dei file. Una questione basilare che devono affrontare tutti i file system (la parte del sistema operativo dedicata alla gestione dei file) è l'allocazione di memoria di massa. L'unità di allocazione (a volte chiamata blocco) può essere un singolo settore del disco, ma spesso consiste in un blocco di settori consecutivi.

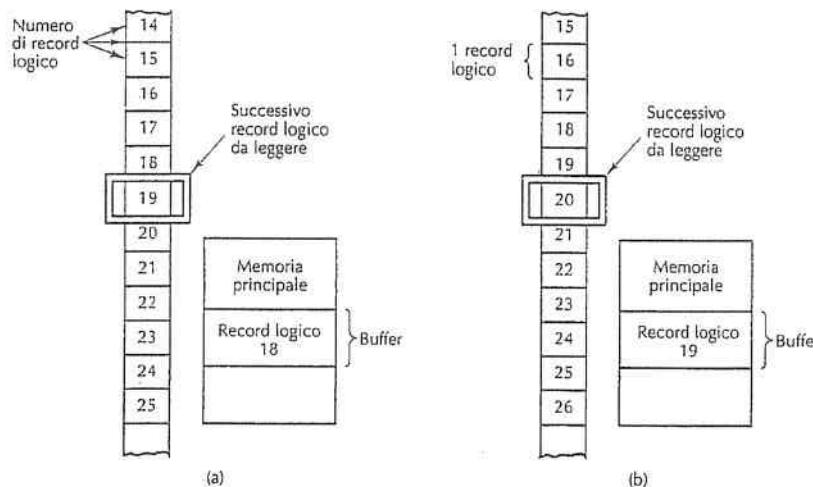


Figura 6.20 Lettura di un file costituito di record logici. (a) Prima della lettura del record 19.  
(b) Dopo la lettura del record 19.

Un'altra proprietà fondamentale dell'implementazione di un file system è la memorizzazione dei file in unità di allocazione consecutive o meno. La Figura 6.21 raffigura un semplice disco la cui superficie contiene 5 tracce di 12 settori ciascuna. La Figura 6.21(a) mostra uno schema di allocazione in cui i file occupano settori consecutivi, lo schema comune dei CD-ROM. La Figura 6.21(b) mostra invece uno schema in cui il settore è ancora l'unità di allocazione, ma dove un file non è vincolato a occupare settori consecutivi. Questo è lo schema utilizzato di norma nei dischi.

Esiste una differenza importante tra la nozione di file dei programmati di applicazioni e quella del sistema operativo. Il programmatore vede il file come una sequenza di byte o di record logici. Il sistema operativo vede il file come una collezione ordinata, non necessariamente consecutiva, di unità di allocazione su disco.

Per poter soddisfare la richiesta del byte o del record logico alla locazione  $n$  di un file, il sistema operativo deve sapere come localizzare i dati. Se il file è allocato in maniera consecutiva, al sistema operativo basta riconoscere l'inizio del file per calcolare la posizione del byte o del record logico richiesto.

In caso contrario, non è possibile calcolare la posizione di un byte o di un record logico arbitrario all'interno del file, a partire dalla sua sola posizione iniziale. Ci sarà dunque bisogno di una tabella contenente le unità di allocazione e i loro indirizzi su disco: la tabella degli indici. Quest'ultima può essere organizzata come una lista d'indirizzi di blocchi del disco (come in UNIX), come una lista di sequenze di blocchi consecutivi (utilizzata in Windows 7) o come una lista di record logici, specificandone l'indirizzo e l'offset su disco. Alle volte i record logici sono associati a una chiave e i programmi possono fare riferimento a un record attraverso la sua chiave, piuttosto che

attraverso il suo numero. In questo caso è necessario il secondo tipo di organizzazione, che aggiunge a ogni elemento della lista la chiave del record oltre alla sua posizione su disco. Questa organizzazione è comune nei mainframe.

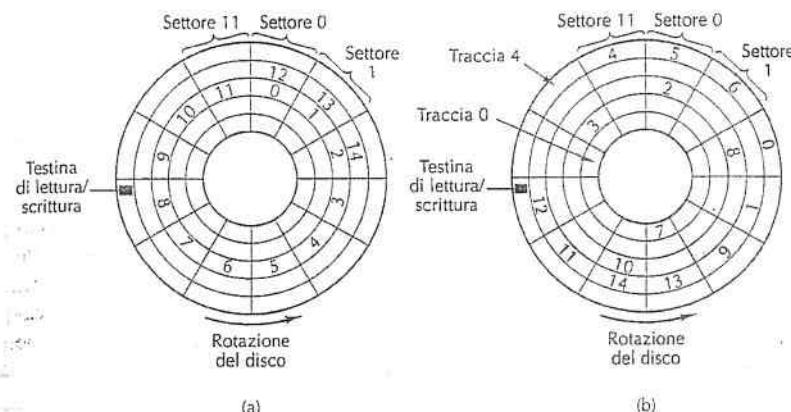


Figura 6.21 Strategie di allocazione su disco. (a) Un file in settori consecutivi. (b) Un file in settori non consecutivi.

Un altro metodo per localizzare le unità di allocazione di un file è l'organizzazione del file come una lista concatenata. Ogni unità di allocazione contiene l'indirizzo dell'unità che le succede. Per implementare in modo efficiente questo schema è necessario mantenere in memoria principale una tabella con gli indirizzi di tutte le unità successive. Per esempio, se il disco contiene 64 KB di unità d'allocazione, il sistema operativo dovrebbe gestire in memoria una tabella di 64 KB di elementi, ciascuno contenente l'indice del suo successore. Se un file occupa le unità 4, 52 e 19, l'elemento 4 della tabella dovrebbe contenere un 52, l'elemento 52 un 19, e l'elemento 19 conterebbe un codice speciale (per esempio 0 o -1) a indicare la fine del file. Era questo il funzionamento del file system di MS-DOS e di Windows 95/98: Le più recenti versioni di Windows (2000, XP, Vista e 7) supportano questo schema, ma possiedono anche un proprio file system nativo più simile a quello di UNIX.

Fin qui abbiamo considerato file allocati consecutivamente o meno, ma non abbiamo chiarito perché si utilizzano entrambe le soluzioni. La prima consente un'amministrazione semplice dei blocchi, ma è di difficile implementazione quando non è nota a priori la dimensione massima di file. Se un file comincia al settore  $j$  e cresce nei settori consecutivi può succedere che, se non ha abbastanza spazio per espandersi, finisce per collidere con un altro file nel settore  $k$ . Questa evenienza non crea problemi se il file non è allocato consecutivamente, poiché i blocchi successivi possono essere messi altrove sul disco. Se un disco contiene svariati file in espansione, senza che sia nota la loro

dimensione finale, è pressoché impossibile memorizzarli in modo consecutivo. Il trasferimento di un file esistente è spesso possibile, ma tuttavia molto costoso.

D'altro canto, se la dimensione massima dei file è nota a priori, la situazione cambia. Per esempio il programma di masterizzazione di un CD-ROM può preallocare per ogni file una sequenza consecutiva di settori della sua lunghezza. Così i file di lunghezza 1200, 700, 200 e 900 possono essere salvati sul CD-ROM rispettivamente a partire dai settori 0, 1200, 1900 e 3900 (non prendiamo in considerazione qui la tabella dei contenuti). È facile accedere a qualsiasi locazione di un file perché il suo primo settore è noto.

Per poter allocare su disco spazio sufficiente per memorizzare i file nuovi, il sistema operativo deve tener traccia dei blocchi liberi e di quelli occupati da file. Nel caso del CDROM il calcolo viene effettuato all'inizio una volta per tutte, ma nel caso di un disco i file vengono salvati e cancellati di continuo. È possibile mantenere una lista di tutte le lacune, ovvero le successioni di unità di allocazione libere e contigue. Questa lista prende il nome di lista delle locazioni libere (*free list*). La Figura 6.22(a) illustra la lista delle locazioni libere per il disco della Figura 6.21(b), con un settore per unità di allocazione.

In alternativa è possibile mantenere una bit map con un bit per ogni unità di allocazione, come mostrato nella Figura 6.22(b). I bit sono posti a 1 se l'unità di allocazione corrispondente è già occupata e azzerati in caso contrario.

Traccia	Settore	Numero di settori nella lacuna
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

Traccia	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

Figura 6.22 Due modi di tener traccia dei settori disponibili. (a) Con una lista delle locazioni libere. (b) Con una bit map.

La prima soluzione presenta il vantaggio di semplificare la ricerca di uno spazio libero di una data dimensione, anche se presenta lo svantaggio di richiedere una struttura dati di lunghezza variabile: la lunghezza della lista varia alla creazione o alla cancellazione di file, una caratteristica poco gradita. La bit map ha il vantaggio di avere una dimensione costante e in più richiede la sola modifica di un bit per cambiare lo stato di allocazione di un'unità, da libera a occupata o viceversa. Risulta invece difficile la ricerca di un blocco di una data lunghezza. Entrambe le tecniche richiedono l'aggiornamento della lista o della tabella all'atto di allocazione o di rimozione di un file.

Prima di terminare la trattazione dell'implementazione del file system è bene analizzare la dimensione dell'unità di allocazione, che chiama in gioco molti fattori diversi. Innanzitutto va detto che il tempo di ricerca e il ritardo rotazionale del disco predomina sui tempi di accesso. Dopo aver investito 5-10 ms per raggiungere l'inizio di un'unità di allocazione è decisamente preferibile leggere 8 KB (circa 80 µs) che 1 KB (circa 10 µs), poiché leggere 8 KB in tranches di 1 KB richiede otto ricerche su disco. L'efficienza del trasferimento gioca a favore delle unità di allocazione grandi. Con i dischi allo stato solido che diventano più comuni e meno costosi, questo argomento cesserà di valere, perché per questi dispositivi non esiste un tempo di accesso.

A favore delle unità di allocazione grande c'è anche il fatto che avere unità di allocazione piccole vuol dire averne molte, il che a sua volta implica indici di file lunghi o grandi tabelle di liste concatenate in memoria. Come nota storica, ricordiamo che MS-DOS iniziò con unità di allocazione che erano di un settore da 512 byte e con numeri di 16 bit utilizzati per identificare i settori. Quando i dischi hanno superato i 65.536 settori, l'unico modo per utilizzare tutto lo spazio del disco e continuare a usare numeri da 16 bit per identificare le unità di allocazione era utilizzare unità di allocazione sempre più grandi. La prima versione di Windows 95 presentava lo stesso problema, poi risolto in una versione successiva con indirizzi di 32 bit. Windows 98 supportava entrambe le dimensioni.

A favore di unità di allocazione piccole è invece la considerazione che ben pochi file occupano esattamente un numero intero di unità di allocazione. Di conseguenza quasi tutti i file sprecano dello spazio nell'ultima unità che occupano e, se un file è molto più grande dell'unità di allocazione, in media il suo spreco è dell'ordine della metà dell'unità di allocazione stessa. Più è grande l'unità di allocazione, maggiore è lo spreco: se i file sono in media molto più piccoli dell'unità di allocazione, verrà sprecato gran parte dello spazio su disco.

Per fare un esempio, in una partizione di 2 GB MS-DOS o della prima versione di Windows 95, in cui le unità di allocazione erano quindi di 32 KB, un file di 100 caratteri sprecava 32.668 byte di spazio del disco. L'efficienza della memorizzazione gioca a favore delle unità di allocazione piccole. A causa del prezzo sempre più basso dei dischi di grandi dimensioni, l'efficienza in termini di tempo tende oggi a essere il fattore più importante. Le unità di allocazione tendono di conseguenza a crescere e lo spreco di spazio sul disco viene semplicemente accettato.

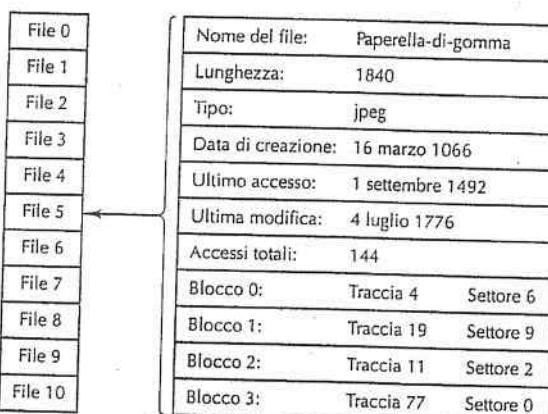
### 6.3.3 Istruzioni per la gestione di directory

Ai primordi dell'era informatica dati e programmi venivano conservati su schede perforate archiviate negli schedari dei propri uffici. Questa soluzione risultò sempre meno agevole al crescere del numero e della dimensione dei dati e dei programmi e condusse infine all'idea di usare una memoria secondaria (per esempio un disco) come alternativa agli schedari per la memorizzazione dei dati e dei programmi. Le informazioni accessibili direttamente dal computer senza bisogno dell'intervento umano si dicono *in linea* (*online*) in contrapposizione alle informazioni *non in linea* (*offline*), le quali richiedono la mediazione di un operatore perché il computer possa accedervi (per esempio tramite l'inserimento di un CD-ROM, di una chiavetta USB o di una scheda SD).

L'informazione in linea è memorizzata nei file ed è resa accessibile ai programmi per mezzo delle istruzioni di I/O. In realtà l'informazione in linea ha bisogno d'istruzioni ulteriori per il suo mantenimento, per il raggruppamento in unità agevoli e per la protezione da tentativi di utilizzo non autorizzati.

I sistemi operativi sono soliti organizzare i file raggruppandoli in directory (cartelle). La Figura 6.23 raffigura un esempio di organizzazione per directory. Tutti i sistemi mettono a disposizione almeno le chiamate di sistema per assicurare le seguenti funzionalità:

1. creare un file e inserirlo in una directory;
2. cancellare un file da una directory;
3. rinominare un file;
4. cambiare lo stato di protezione di un file.



File 0	Nome del file: Paperella-di-gomma
File 1	Lunghezza: 1840
File 2	Tipo: jpeg
File 3	Data di creazione: 16 marzo 1066
File 4	Ultimo accesso: 1 settembre 1492
File 5	Ultima modifica: 4 luglio 1776
File 6	Accessi totali: 144
File 7	Blocco 0: Traccia 4 Settore 6
File 8	Blocco 1: Traccia 19 Settore 9
File 9	Blocco 2: Traccia 11 Settore 2
File 10	Blocco 3: Traccia 77 Settore 0

Figura 6.23 Esempio di directory di file utente e un suo tipico elemento.

Sono possibili diversi schemi di protezione. Uno è far sì che l'utente specifichi una password per ciascuno dei propri file: quando un programma desidera accedere al file deve fornire la password e attendere la sua convalida da parte del sistema operativo. Un altro metodo di protezione richiede a ciascun utente di specificare esplicitamente, per ogni file, una lista di utenti: solo i programmi di quegli utenti potranno accedere al file.

Negli sistemi operativi moderni gli utenti possono gestire più di una directory. Ogni directory è a sua volta un file e può far parte così di un'altra directory, dando origine a un albero di directory. Questa molteplicità è particolarmente utile quando si lavora su più progetti allo stesso tempo: ogni directory può raggruppare tutti i file che afferiscono a un dato progetto, evitando le interferenze tra file di progetti diversi. Inoltre le directory agevolano la condivisione di file tra utenti membri di uno stesso gruppo.

## 6.4 Istruzioni per il calcolo parallelo a livello OSM

Alcuni tipi di calcolo possono essere programmati per essere svolti con maggior profitto da due o più processi cooperanti che girano in parallelo (cioè simultaneamente, su processori diversi) piuttosto che da un solo processo. Altri calcoli possono essere divisi in parti da far eseguire parallelamente, per ridurre il tempo dell'intera computazione. Nei paragrafi successivi illustreremo alcune istruzioni virtuali necessarie per consentire a più processi di collaborare in parallelo.

Le leggi della fisica forniscono un'altra ragione a favore dell'interesse attualmente rivolto al calcolo parallelo. Secondo la teoria della relatività speciale di Einstein è impossibile trasmettere un segnale elettrico a una velocità superiore a quella della luce, che è di circa 30 cm/ns (300.000 km/s) nel vuoto, e un po' inferiore nei fili di rame e nelle fibre ottiche. Questo limite ha implicazioni importanti sulla strutturazione dei computer. Per esempio se la CPU richiede dati alla memoria principale e questa si trova a 30 cm, ci vorrà almeno 1 ns perché la richiesta giunga alla memoria e ancora 1 ns perché la risposta giunga alla CPU. Perciò i computer che vorranno lavorare in tempi sotto il nanosecondo dovranno essere molto miniaturizzati o, in alternativa, essere dotati di molte CPU. Un computer con mille CPU da 1 ns può raggiungere (in linea teorica) la stessa potenza di calcolo di una CPU con cicli di 0,001 ns, ed è molto più facile e più economico da costruire. Il calcolo parallelo sarà discusso in dettaglio nel Capitolo 8.

Su un calcolatore con più di una CPU è possibile assegnare ciascun processo cooperativo a una CPU, facendo sì che progrediscano tutti simultaneamente. Se invece c'è una sola CPU è possibile simulare l'effetto dell'elaborazione parallela alternando l'esecuzione dei diversi processi per brevi lassi di tempo. In altre parole, il processore è condizionato dai diversi processi.

La Figura 6.24 mostra la differenza tra il vero calcolo parallelo, possibile con più processori, e quello simulato, attuato su di un solo processore fisico. Anche quando il parallelismo è solo simulato risulta utile guardare a ogni processo come se disponesse di un processore virtuale dedicato. Nella simulazione si verificano gli stessi problemi di comunicazione riscontrabili nel vero calcolo parallelo. In entrambi i casi, effettuare il debug dei problemi è molto difficile.

### 6.4.1 Creazione dei processi

Ogni programma in esecuzione fa parte di un processo. Un processo è caratterizzato da uno stato e da uno spazio degli indirizzi tramite cui accedere ai dati e al programma. Lo stato comprende almeno il program counter, una PSW, un puntatore allo stack e dei registri d'uso generale.

Quasi tutti i sistemi operativi moderni consentono la creazione e la terminazione dinamica dei processi. Esiste perciò una chiamata di sistema per la creazione di un processo, fondamentale per realizzare il calcolo parallelo. Questa chiamata di sistema può limitarsi a clonare il processo chiamante oppure gli può consentire di specificare stato iniziale, programma, dati e indirizzo iniziale del nuovo processo.

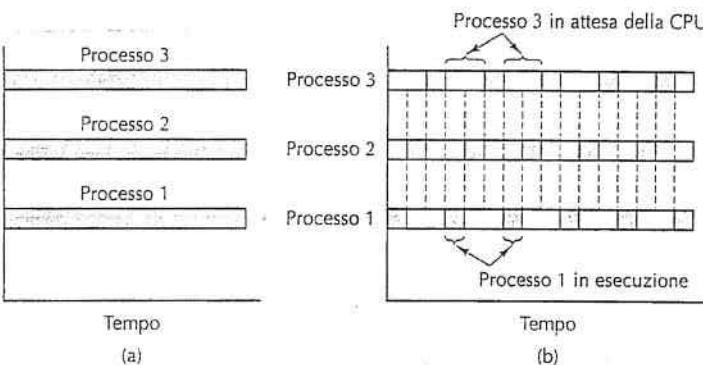


Figura 6.24 (a) Calcolo genuinamente parallelo su CPU multiple. (b) Calcolo parallelo simulato tramite l'alternanza della singola CPU all'interno di un insieme di tre processi.

In alcuni casi il processo creatore (genitore) mantiene controllo parziale o anche totale del processo creato (figlio). A tal fine esistono istruzioni virtuali perché il processo genitore possa fermare, far ripartire, esaminare e porre fine al processo figlio. In altri casi il genitore ha meno controllo sul figlio: una volta creato il processo figlio, il genitore non ha modo di farlo fermarsi, ripartire, farsi esaminare o terminare la propria esecuzione. I due processi procedono nelle rispettive attività in modo indipendente l'uno dall'altro.

#### 6.4.2 Corsa critica

In molti casi i processi paralleli hanno bisogno di comunicare e di sincronizzarsi al fine di poter svolgere il proprio compito. Qui ci avvaliamo di un esempio dettagliato per trattare la sincronizzazione tra processi ed esaminare alcune delle difficoltà che comporta; nel paragrafo successivo forniremo le soluzioni a queste problematiche.

Si consideri la situazione di due processi indipendenti, il processo 1 e il processo 2, che comunicano attraverso un buffer condiviso in memoria principale; per comodità chiamiamo il primo produttore e il secondo consumatore. Il produttore calcola i numeri primi e li ripone nel buffer uno per volta. Il consumatore li rimuove dal buffer uno alla volta e li visualizza sullo schermo.

Questi due processi girano in parallelo con velocità diverse. Se il produttore si rende conto che il buffer è pieno allora va a dormire, ovvero sospende temporaneamente le sue operazioni in attesa di un segnale dal consumatore. Successivamente il consumatore, dopo aver rimosso un numero dal buffer, invia il segnale al produttore per svegliarlo, cioè per farlo ripartire. Analogamente, il consumatore va a dormire quando il buffer è vuoto. Il produttore lo sveglierà non appena avrà inserito un numero nel buffer vuoto.

In questo esempio usiamo un buffer circolare per la comunicazione tra processi. I puntatori *in* e *out* sono usati nel modo seguente: *in* punta alla parola libera successiva (dove il produttore inserirà il prossimo numero primo), *out* punta al numero che sarà

rimosso per primo dal consumatore. Il buffer è vuoto quando *in* = *out*, come nella Figura 6.25(a). La Figura 6.25(b) mostra la situazione dopo che il produttore ha generato alcuni numeri primi. La Figura 6.25(c) mostra la situazione dopo che il consumatore ha rimosso e stampato alcuni di quei numeri primi. Le Figure 6.25(d)-(f) illustrano una possibile evoluzione del buffer a seguito dell'attività dei due processi. La cima del buffer è logicamente contigua con la sua base, ossia il buffer si riavvolge in modo circolare. Se si verifica una raffica di inserimenti tali che *in* punti alla parola precedente di quella puntata da *out* (per esempio *in* = 52 e *out* = 53) allora il buffer è pieno. L'ultima parola libera non è utilizzata, perché altrimenti non ci sarebbe alcun modo per capire se *in* = *out* indica il buffer vuoto o il buffer pieno.

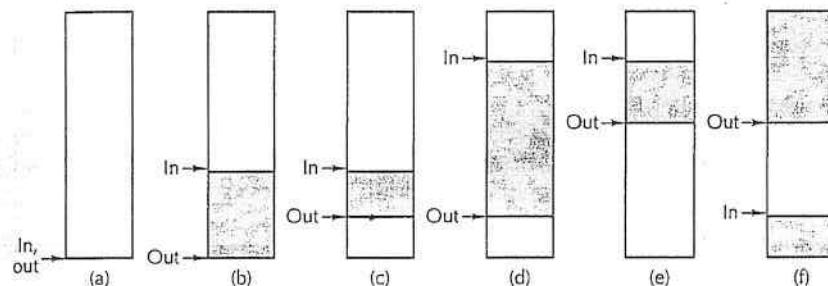


Figura 6.25 Uso di un buffer circolare.

La Figura 6.26 mostra una semplice implementazione in Java del problema produttore-consumatore. Questa soluzione usa tre classi: *m*, *producer* e *consumer*. La classe principale *m* contiene alcune definizioni di costanti, i puntatori al buffer *in* e *out* e il buffer stesso, che in questo esempio può contenere 100 primi da *buffer[0]* a *buffer[99]*. Le classi *producer* e *consumer* svolgono il lavoro effettivo.

Il codice si avvale dei thread Java per simulare il calcolo parallelo. Le classi *producer* e *consumer* sono istanziate rispettivamente nelle variabili *p* e *c*; sono entrambe classi-derivate dalla classe base *Thread* provvista del metodo *run*, che serve a contenere il codice del thread. L'invocazione del metodo *start* di un oggetto derivato da *Thread* provoca l'avvio di un nuovo thread (l'esecuzione del suo metodo *run*).

Un thread somiglia a un processo, con la differenza che tutti i thread di un programma Java accedono allo stesso spazio degli indirizzi; questa caratteristica permette loro di condividere un buffer. Se un calcolatore ha due o più CPU è possibile assegnare ciascun thread a una diversa CPU e ottenere quindi il vero parallelismo. Se c'è una sola CPU, i thread la condividono secondo lo schema di condivisione del tempo. Continueremo a far riferimento ai processi produttore e consumatore (visto che il nostro interesse è per i processi paralleli) anche se Java supporta solo i thread paralleli e non i processi veramente paralleli.

```

public class m {
    final public static int BUF_SIZE = 100;           // il buffer va da 0 a 99
    final public static long MAX_PRIME = 100000000000L; // valore d'arresto
    public static int in = 0, out = 0;                 // puntatori ai dati
    public static long buffer[] = new long[BUF_SIZE]; // memorizza i numeri primi
    public static producer p;                         // nome del produttore
    public static consumer c;                        // nome del consumatore

    public static void main(String args[]) {
        p = new producer();
        c = new consumer();
        p.start();
        c.start();
    }

    // funzione di servizio per l'incremento circolare di in e out
    public static int next(int k) { if (k < BUF_SIZE - 1) return(k+1); else return(0); }
}

class producer extends Thread {
    public void run() {
        long prime = 2;

        while (prime < m.MAX_PRIME) {
            prime = next_prime(prime);
            if (m.next(m.in) == m.out) suspend();
            m.buffer[m.in] = prime;
            m.in = m.next(m.in);
            if (m.next(m.out) == m.in) m.c.resume();
        }
    }

    private long next_prime(long prime){ ... }
}

class consumer extends Thread {
    public void run() {
        long emirp = 2;

        while (emirp < m.MAX_PRIME) {
            if (m.in == m.out) suspend();
            emirp = m.buffer[m.out];
            m.out = m.next(m.out);
            if (m.out == m.next(m.next(m.in))) m.p.resume();
            System.out.println(emirp);
        }
    }
}

```

Figura 6.26 Calcolo parallelo in situazione facale di corsa critica.

La funzione `next` serve a incrementare `in` e `out` in modo agevole, senza dover controllare ogni volta il verificarsi di un riavvolgimento circolare. Se il parametro di `next` è minore o uguale a 98 la funzione restituisce l'intero successivo. Se invece il parametro vale 99 abbiamo raggiunto la fine del buffer e la funzione restituisce 0.

C'è bisogno poi di un modo per consentire ai processi di dormire quando non possono continuare l'esecuzione. I progettisti di Java hanno previsto questa necessità e hanno incluso i metodi `suspend` (sonno) e `resume` (risveglio) nella classe `Thread` sin dalla prima versione di Java. Se ne fa uso nella Figura 6.26.

Veniamo quindi al codice del produttore e del consumatore vero e proprio. All'inizio il produttore genera un nuovo numero primo nell'istruzione P1. Si noti qui l'uso di `m.MAX_PRIME`: il prefisso `m.` indica che intendiamo il valore `MAX_PRIME` definito nella classe `m`. Per le stesse ragioni `m.` è anche il prefisso di `in`, `out`, `buffer` e `next`.

In seguito il produttore controlla (in P2) se `in` è arretrato di una posizione rispetto a `out`. In tal caso (per esempio `in = 62, out = 63`) il buffer è pieno e il produttore va a dormire invocando `suspend` in P2. Se il buffer non è pieno il nuovo primo può essere inserito nel buffer (P3) e `in` incrementato (P4). Se in P5 il nuovo valore di `in` eccede di 1 `out` (per esempio `in = 17 e out = 16`), dovevano essere necessariamente uguali prima dell'incremento di `in`. Il produttore conclude che il buffer era precedentemente vuoto e che il consumatore sta ancora dormendo, dunque chiama la `resume` per svegliarlo (P5). Infine il produttore comincia a preoccuparsi del successivo valore da produrre.

Il programma del consumatore è strutturato in modo simile. All'inizio effettua un test (C1) per verificare se il buffer è vuoto; in caso affermativo non c'è lavoro da svolgere e il consumatore può andare a dormire. In caso negativo, il consumatore rimuove il successivo numero da stampare (C2) e incrementa `out` (C3). A questo punto, se `out` è due posizioni oltre `in` (C4) vuol dire che precedentemente distavano l'un l'altro una sola posizione; poiché `in = out - 1` è la condizione di "buffer pieno", il produttore deve essere ancora in fase di sonno; quindi il consumatore lo risveglia chiamando la `resume`. Infine stampa il numero (C5) e ricomincia lo stesso ciclo.

Sfortunatamente questa soluzione contiene un difetto fatale, illustrato nella Figura 6.27. Si ricordi che i due processi girano in modo asincrono ed, eventualmente, a diverse velocità. Si consideri il caso della Figura 6.27(a), dove il buffer contiene un solo elemento nel posto 21, con `in = 22` e `out = 21`. Se il produttore si trova all'istruzione P1 in cerca di un numero primo e il consumatore è occupato in C5 a stampare l'elemento di posizione 20, allora il consumatore termina la stampa del numero, effettua il test di C1 e in C2, preleva dal buffer l'ultimo numero presente, quindi incrementa `out`. In questo istante `in` e `out` valgono entrambi 22. Poi il consumatore stampa il numero e torna a C1, dove legge `in` e `out` dalla memoria per effettuarne il confronto, come mostrato nella Figura 6.27(b).

In questo preciso istante, dopo che il consumatore ha prelevato `in` e `out` dalla memoria e prima di averli confrontati, il produttore trova il numero primo successivo, lo inserisce nel buffer con l'istruzione P3 e incrementa `in` in P4. In altre parole, `in` vale ora `out + 1` a indicare la presenza di un elemento nel buffer e il produttore ne deduce (erroneamente) che il consumatore è sospeso, perciò gli invia un segnale di risveglio (cioè chiama `resume`), come raffigurato dalla Figura 6.27(c). Invece il consumatore è ancora sveglio e così il segnale di risveglio va perso. Il produttore inizia la ricerca di un altro numero primo.

Adesso il consumatore continua la propria attività: ha già recuperato `in` e `out` dalla memoria prima dell'inserimento nel buffer dell'ultimo numero trovato dal produttore. Poiché entrambi i puntatori valgono 22 il consumatore va a dormire. Successivamente il produttore trova un nuovo numero primo, osserva che `in = 24` e `out = 22`, e conclude che ci sono due numeri nel buffer (vero) e che il consumatore è sveglio (falso). Il produttore prosegue la sua esecuzione e arriva il momento in cui riempie il buffer e va a dormire. Ora entrambi i processi sono dormienti e lo rimarranno per sempre.

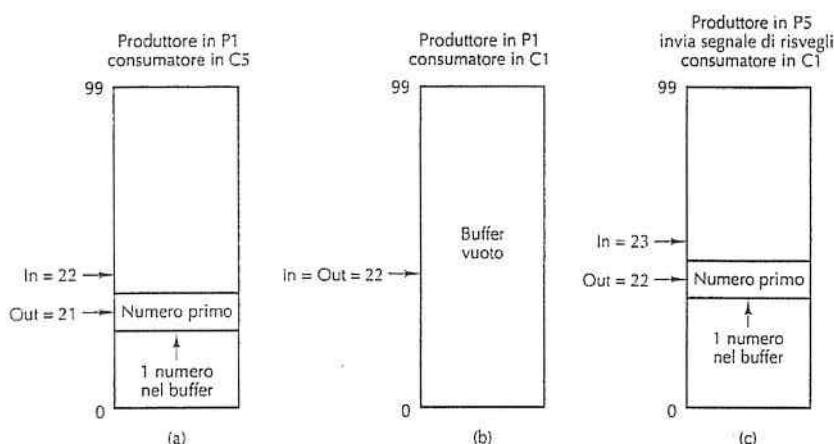


Figura 6.27 Insuccesso del meccanismo di comunicazione tra produttore e consumatore.

Il problema qui è stato originato dal fatto che, nel lasso di tempo tra la lettura di *in* e *out* da parte del consumatore e il suo assopimento, il produttore è intervenuto di soppiatto, ha scoperto che  $in = out + 1$ , ha pensato che il consumatore stesse dormendo (non ancora vero) e gli ha spedito un segnale di sveglia che è andato perso proprio perché il consumatore era ancora sveglio. Questa problematica è nota con il nome di **corsa critica**, perché il successo del metodo dipende dal vincitore della corsa al test di *in* e *out* dopo l'incremento di *out*.

Il problema delle corse critiche è ben noto ed è così serio che, svariati anni dopo l'introduzione di Java, Sun ha cambiato l'interfaccia della classe Thread rinnegando l'uso di suspend e resume perché conducevano spesso a condizioni di corsa critica. La soluzione alternativa proposta si basa sul linguaggio, ma visto che ci interessiamo qui ai sistemi operativi, prendiamo in considerazione un'ulteriore soluzione fornita da molti di loro, compresi UNIX e Windows 7.

### 6.4.3 Sincronizzazione dei processi tramite semafori

È possibile risolvere le situazioni di corsa critica in almeno due modi. Una soluzione consiste nel dotare ciascun processo di un "bit risveglio pendente". Ogni volta che viene spedito un segnale di risveglio a un processo ancora attivo il suo bit risveglio pendente viene asserito. Se un processo va a dormire con questo bit asserito, la sua esecuzione riprende immediatamente e il bit viene azzerato. Il bit risveglio pendente serve a conservare un segnale di risveglio superfluo per un uso futuro.

Anche se questo metodo risolve la corsa critica tra due processi, fallisce però nel caso generale di comunicazione tra  $n$  processi, perché bisognerebbe salvare  $n - 1$  bit di risvegli pendenti per contare fino a  $n - 1$ , ma si tratta evidentemente di una soluzione astrusa.

Dijkstra (1968b) ha proposto una soluzione più generale al problema della sincronizzazione tra processi paralleli che prevede l'uso di variabili intere (non negative) chiamate semafori. Il sistema operativo fornisce due chiamate di sistema, up e down, per agire sui semafori: la prima per incrementare un semaforo di 1, la seconda per decrementarlo di 1.

Se si effettua un'operazione down di un semaforo che ha valore positivo, questo viene decrementato di 1 e il processo corrispondente continua. Invece se il semaforo vale 0 la down non può andare a buon fine; il processo corrispondente è messo a dormire e resta in questo stato finché un altro processo esegue una up su quel semaforo. In genere i processi dormienti sono posti in una coda al fine di poterne tener traccia.

L'istruzione up su un semaforo controlla se il semaforo è a 0, nel qual caso lo incrementa di un'unità e consente al processo dormiente di completare l'operazione down che ha causato la sua sospensione; il risultato di queste operazioni è che il semaforo vale ancora 0, ma entrambi i processi possono ora continuare la loro esecuzione. Un'istruzione up su di un semaforo non nullo, lo incrementa di 1. Un semaforo serve essenzialmente a contare i segnali di risveglio per uso futuro, così che non vadano persi. Una proprietà fondamentale delle istruzioni sui semafori è che, una volta che un processo ha intrapreso un'operazione su di un semaforo, nessun altro può accedervi finché il primo non ha completato l'istruzione o finché non resta sospeso nel tentativo di effettuare una down su 0. La Figura 6.28 riassume le proprietà essenziali delle chiamate di sistema up e down.

Come già menzionato, Java possiede una soluzione basata sul linguaggio per trattare le corse critiche: occupandoci qui di sistemi operativi, cerchiamo un modo di rappresentare in Java i semafori anche se non fanno parte del linguaggio o delle classi standard. Lo facciamo ipotizzando l'esistenza di due metodi nativi, up e down, che svolgono i compiti delle chiamate di sistema up e down (rispettivamente) e che accettano in ingresso parametri interi. Questi metodi ci consentono di esprimere l'uso dei semafori in Java.

Istruzione	Semaforo = 0	Semaforo > 0
Up	Semaforo = semaforo + 1; se un altro processo era stato arrestato nel tentativo di completare un'istruzione down su questo semaforo, adesso può completare la down e proseguire la propria esecuzione	Semaforo = semaforo + 1
Down	Il processo resta sospeso finché un altro processo non esegue una up su questo semaforo	Semaforo = semaforo - 1

Figura 6.28 Risultato delle operazioni su semafori.

La Figura 6.29 mostra come eliminare la corsa critica tramite l'uso dei semafori. Aggiungiamo alla classe `m` i due semafori `available` (disponibile), che inizialmente vale 100, e `filled` (riempito), posto inizialmente a 0. Come prima, il produttore comincia l'esecuzione da P1 e il consumatore parte da C1. L'esecuzione di down su `filled` interrompe immediatamente il processo consumatore. Quando il produttore ha trovato il numero primo invoca down con parametro `available`, assegnandogli il valore 99. All'istruzione P5 chiama `up` con parametro `filled`, assegnandogli il valore 1 e liberando così

il consumatore dall'attesa; questi può ora completare la sua chiamata down sospesa. In questo istante `filled` vale 0 ed entrambi i processi sono in esecuzione.

```

public class m {
    final public static int BUF_SIZE = 100;
    final public static long MAX_PRIME = 10000000000L;
    public static int in = 0, out = 0;
    public static long buffer[] = new long[BUF_SIZE];
    public static producer p;
    public static consumer c;
    public static int filled = 0, available = 100;
}

public static void main(String args[]) {
    p = new producer();
    c = new consumer();
    p.start();
    c.start();
}

// funzione di servizio per l'aggiornamento circolare di in e out
public static int next(int k) { if (k < BUF_SIZE - 1) return(k+1); else return(0); }

class producer extends Thread {
    native void up(int s); native void down(int s);
    public void run() {
        long prime = 2;

        while (prime < m.MAX_PRIME) {
            prime = next_prime(prime);
            down(m.available);
            m.buffer[m.in] = prime;
            m.in = m.next(m.in);
            up(m.filled);
        }
    }

    private long next_prime(long prime){ ... }

    // classe produttore
    // metodi sui semafori
    // codice del produttore
    // variabile di lavoro
}
// istruzione P1
// istruzione P2
// istruzione P3
// istruzione P4
// istruzione P5

// funzione che calcola il numero primo successivo

class consumer extends Thread {
    native void up(int s); native void down(int s);
    public void run() {
        long emirp = 2;

        while (emirp < m.MAX_PRIME) {
            down(m.filled);
            emirp = m.buffer[m.out];
            m.out = m.next(m.out);
            up(m.available);
            System.out.println(emirp);
        }
    }

    // classe consumatore
    // metodi sui semafori
    // codice del consumatore
    // variabile di lavoro
}
// istruzione C1
// istruzione C2
// istruzione C3
// istruzione C4
// istruzione C5

```

Figura 6.29 Calcolo parallelo con i semafori.

Torniamo alla precedente situazione di corsa critica in una situazione in cui `in = 22`, `out = 21`, il produttore si trova in P1 e il consumatore in C5. Dunque il consumatore completa l'istruzione e torna a C1, dove invoca `down su filled`, che passa così da 1 a 0. Poi

il consumatore preleva l'ultimo numero disponibile nel buffer, chiama `up su available`, che arriva a 100, stampa a schermo il valore e torna nuovamente in C1. Un attimo prima che il consumatore chiami `down`, il produttore trova il successivo numero primo ed esegue in rapida successione le istruzioni P2, P3 e P4.

Adesso `filled` vale 0. Il produttore è sul punto di eseguire una `up su filled` e il consumatore una `down` sullo stesso parametro. Se il consumatore esegue per primo l'istruzione rimarrà sospeso finché il produttore non lo libererà (chiamando la `up`). Se invece parte per primo il produttore, questi porrà il semaforo a 1 e il consumatore non verrà sospeso. In entrambi i casi non viene perso alcun segnale di risveglio; era proprio l'obiettivo che ci eravamo proposti introducendo i semafori.

La proprietà essenziale delle operazioni sui semafori è che sono indivisibili: una volta iniziata un'operazione su di un semaforo, nessun altro processo può usare il semaforo finché il primo non abbia completato l'operazione o non rimanga sospeso. Per di più, l'uso dei semafori scongiura la perdita dei segnali di risveglio. L'istruzione `if` della Figura 6.26 non è indivisibile: è possibile che un processo spedisca un segnale di risveglio tra la valutazione della condizione e l'esecuzione dell'istruzione selezionata.

Il problema della sincronizzazione di processi è stato risolto dichiarando indivisibili le chiamate di sistema `up` e `down`, richiamate dai metodi `up` e `down`. Perché ciò accada, il sistema operativo deve proibire che un semaforo venga usato da più di un processo per volta, o può almeno limitarsi a interrompere l'esecuzione di ogni altro processo finché quello impiegato nella chiamata di sistema non la porti a compimento. Sui sistemi monoprocesso i semafori sono implementati, qualche volta, mediante la disattivazione degli interrupt durante le operazioni sui semafori. Sui sistemi multiprocessore questo stratagemma non funziona.

La sincronizzazione mediante semafori funziona per un numero arbitrario di processi. Ci possono essere allo stesso tempo molti processi dormienti, impegnati nel tentativo di completare una chiamata di sistema `down` sullo stesso semaforo. Quando un altro processo effettua una `up` su quel semaforo, uno dei processi in attesa può completare finalmente la `down` e riprendere l'esecuzione. Il valore del semaforo resta 0 e gli altri processi restano in attesa.

Usiamo un'analogia per chiarire la natura dei semafori. Immaginate un torneo con 20 squadre di pallavolo suddiviso in 10 partite (i processi), ciascuna disputata sul proprio campo, e una cassa (il semaforo) contenente i palloni. Sfortunatamente ci sono solo 7 palloni disponibili (il semaforo assume valori tra 0 e 7). L'inserimento di un pallone nella cassa equivale a una `up` perché incrementa il valore del semaforo; l'estrazione di un pallone corrisponde a una `down` poiché ne decremente il valore.

All'inizio del torneo ogni campo invia un giocatore al canestro per appropriarsi di un pallone. Sette di loro riusciranno nell'impresa (completano la `down`), tre sono costretti ad aspettare un pallone (cioè non riescono a completare la `down`). Le loro partite sono temporaneamente sospese; prima o poi una delle altre partite finirà e il rispettivo pallone verrà riportato nel canestro (esecuzione di una `up`). Grazie a questa operazione uno dei tre giocatori in attesa potrà accaparrarsi la palla (completando la sua `down` incompiuta) e dar via alla propria partita. Gli altri due giocatori restano in attesa che vengano riportati nel cesto altri palloni; solo allora le ultime due partite potranno essere disputate.

## 6.5 Sistemi operativi di esempio

In questo paragrafo procediamo con la trattazione dei nostri sistemi d'esempio, il Core i7 e l'OMAP4430. Per ciascuno analizzeremo uno dei sistemi operativi utilizzati sul processore. Per il Core i7 faremo riferimento a Windows e per OMAP4430 a UNIX. Cominciamo da quest'ultimo perché è più semplice e per molti versi più elegante. Inoltre questo ordine è più sensato e naturale dal momento che UNIX è stato progettato e implementato molto tempo prima di Windows 7 e lo ha assai influenzato.

### 6.5.1 Introduzione

In questo paragrafo introdurremo brevemente i due sistemi di esempio, UNIX e Windows 7, focalizzandoci sulla loro storia, la loro struttura e le chiamate di sistema.

#### UNIX

UNIX venne sviluppato presso i laboratori Bell nei primi anni '70. La prima versione, scritta da Ken Thompson in linguaggio assemblativo del minicomputer PDP-7, venne presto seguita da una versione per il PDP-11 (scritta nell'allora nuovo linguaggio C, ideato e sviluppato da Dennis Ritchie). Nel 1974 Ritchie e Thompson pubblicarono un articolo su UNIX che è rimasto una pietra miliare del settore (Ritchie e Thompson, 1974). In ragione del lavoro descritto in quell'articolo furono successivamente insigniti del prestigioso ACM Turing Award (Ritchie, 1984; Thompson, 1984). La pubblicazione dell'articolo stimolò molte università a richiedere una copia di UNIX ai laboratori Bell. Essendo questi di proprietà della AT&T, ai tempi un'azienda monopolista cui non era concesso fare affari nel mondo dell'informatica, la richiesta venne accolta senza difficoltà e la licenza di UNIX venne ceduta alle università per una modica cifra.

Per una di quelle coincidenze che spesso fanno la storia, il PDP-11 era il computer prediletto dai dipartimenti di informatica di quasi tutte le università e i sistemi operativi di cui era equipaggiato erano considerati terribili tanto dai professori quanto dagli studenti. UNIX colmò velocemente questa lacuna, specialmente perché era distribuito con tutti i codici sorgenti e così chiunque poteva armeggiarci liberamente, cosa che fecero in molti.

Una delle prime università a procurarsi UNIX fu la University of California, a Berkeley. Avendo a disposizione tutto il codice sorgente, i ricercatori di Berkeley poterono modificare il sistema in modo sostanziale. Tra i risultati più importanti ci fu l'implementazione sul minicomputer VAX, l'aggiunta della memoria virtuale paginata, l'estensione dei nomi di file da 14 a 255 caratteri e l'inclusione del protocollo di rete TCP/IP, usato ancor oggi in Internet (soprattutto perché si trovava nello UNIX di Berkeley all'avvento delle reti).

Mentre Berkeley procedeva nelle sue modifiche, la stessa AT&T continuava a sviluppare UNIX, giungendo al System III nel 1982 e al System V nel 1984. Alla fine degli anni '80 circolavano due versioni di UNIX diverse e abbastanza incompatibili: UNIX Berkeley e System V. Questa scissione del mondo di UNIX, unita al fatto che mancavano gli standard per i formati dei programmi binari, inibì fortemente il successo commerciale di UNIX perché nessun fornitore di software avrebbe potuto scrivere e confezionare programmi UNIX che funzionassero su ogni sistema UNIX (a differenza di MS-DOS). Dopo molti litigi, il comitato per gli standard IEEE creò uno standard chiamato **POSIX** (*Portable Operating System-IX*, "sistema operativo portabile-IX"), noto anche con il nome di P1003 (il suo numero di standard IEEE). Successivamente POSIX è divenuto uno standard internazionale.

Lo standard è diviso in molte sezioni, ciascuna dedicata a una parte differente di UNIX. La prima sezione, P1003.1, definisce le chiamate di sistema; la seconda sezione, P1003.2, definisce le funzionalità basilari e così via; P1003.1 definisce circa 60 chiamate di sistema che devono essere fornite da tutti i sistemi conformi allo standard. Si tratta delle chiamate elementari per la lettura e scrittura di file, per la creazione di nuovi processi e così via. Praticamente tutti i sistemi UNIX oggi in commercio supportano le chiamate di sistema P1003.1 e in genere molti ne supportano anche altre, in special modo quelle definite da System V e/o da UNIX Berkeley. Queste chiamate di sistema addizionali possono raggiungere facilmente le 200 unità.

Nel 1987 uno degli autori di questo libro (Tanenbaum) distribuì il codice sorgente di MINIX, una piccola versione di UNIX, per uso accademico (Tanenbaum, 1987). Linus Torvald, dopo aver studiato MINIX presso la sua università di Helsinki e dopo averlo installato sul suo PC di casa, decise di scrivere un proprio clone di MINIX, battezzato Linux e divenuto da allora molto popolare.

Molti sistemi operativi in esecuzione su piattaforme ARM sono basati su Linux. MINIX e Linux sono entrambi conformi allo standard POSIX e tutto quanto verrà detto in questo capitolo su UNIX si applica anche a loro, se non specificato diversamente.

La Figura 6.30 fornisce una prima suddivisione in categorie delle chiamate di sistema di Linux. Le categorie più numerose e importanti comprendono le chiamate di sistema per la gestione dei file e delle directory. Linux è prevalentemente conforme a POSIX P1003.1, anche se in alcuni casi gli sviluppatori si sono allontanati dalle specifiche. In generale, non è comunque difficile fare in modo che programmi conformi a POSIX girino su sistemi Linux.

Categoria	Alcuni esempi
Gestione dei file	Apertura, lettura, scrittura, chiusura e lock di file
Gestione delle directory	Creazione e cancellazione delle directory; trasferimento di file
Gestione dei processi	Generazione, terminazione, tracing dei processi e invio di segnali tra loro
Gestione della memoria	Condivisione di memoria tra processi; protezione di pagine
Lettura e scrittura di parametri	Consultazione dell'ID di un utente, di un gruppo, di un processo; assegnamento della priorità
Data e ora	Impostazione dell'ora di accesso a file; timer d'intervallo; profilo esecutivo
Rete	Imposta/accetta connessioni; invia/riceve messaggi
Miscellanea	Creazione di utenti; manipolazione delle quote su disco; riavvio del sistema

Figura 6.30 Suddivisione approssimativa delle chiamate di sistema UNIX.

La categoria delle chiamate di rete è invece dovuta soprattutto a UNIX Berkeley, che ha introdotto il concetto di *socket*, ossia il punto terminale di una connessione di rete, modellata sull'esempio delle prese a muro per le connessioni telefoniche. Un processo UNIX può creare una socket, collegarsi a essa e stabilire una connessione a una socket di un'altra macchina, grazie alla quale può scambiare dati in modo bidirezionale, avvalendosi in genere del protocollo TCP/IP. Una frazione rilevante dei server di Internet funzionano con UNIX proprio perché la sua tecnologia di rete esiste da decenni ed è ormai stabile e matura.

È difficile fare delle asserzioni generali sulla struttura del sistema operativo UNIX perché ne esistono molte implementazioni, ciascuna a suo modo diversa dalle altre. Ciò nondimeno, la Figura 6.31 vale per quasi tutte le implementazioni. Alla sua base si trova il livello dei driver di dispositivo che mette il sistema operativo al riparo dal contatto con l'hardware vero e proprio. In un primo momento i driver di dispositivo venivano scritti come entità indipendenti e separate dagli altri driver, causando un'inutile reiterazione degli sforzi, visto che molti driver devono affrontare problematiche analoghe, quali il controllo del flusso, la gestione degli errori, le priorità, la separazione dei dati dal controllo e così via. Questa osservazione suggerì a Dennis Ritchie lo sviluppo del modello *stream* ("flusso") per la scrittura di driver modulari. Uno stream permette di stabilire una connessione bidirezionale tra un processo dell'utente e un dispositivo hardware e di inserire uno o più moduli lungo il percorso. Il processo dell'utente invia dati lungo lo stream e questi vengono elaborati o trasformati da ciascun modulo finché non giungono all'hardware. I dati provenienti dai dispositivi subiscono l'elaborazione inversa.

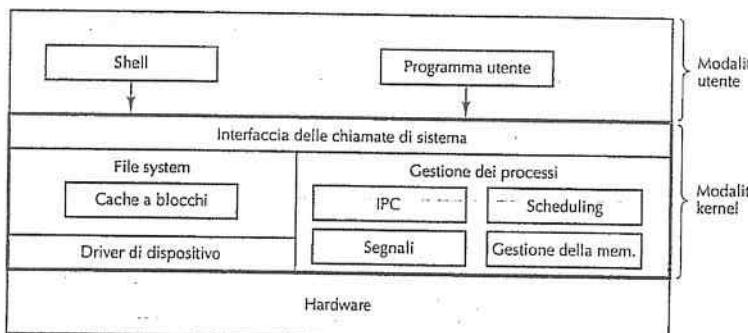


Figura 6.31 Struttura di un comune sistema UNIX.

Al di sopra dei driver di dispositivo c'è il file system che gestisce i nomi dei file, le directory, l'allocazione dei blocchi del disco, la protezione e altro ancora. Parte del file system è la *cache a blocchi* che memorizza i blocchi letti dal disco più di recente in caso dovessero tornare utili a breve. Nel corso degli anni sono stati utilizzati molti file system diversi, tra cui il fast file system di Berkeley (McKusick et al., 1984) e file system a struttura logaritmica (Rosenblum e Ousterhout, 1991; Seltzer et al., 1993).

Un'altra parte del kernel UNIX riguarda la gestione dei processi che, tra le altre cose, si occupa della IPC (*InterProcess Communication*, "comunicazione tra processi"). La IPC mette a disposizione svariati meccanismi che consentono ai processi di comunicare tra loro e di sincronizzarsi per evitare le corse critiche. Il codice per la gestione dei processi gestisce anche lo scheduling dei processi in base alle loro priorità. Anche i segnali, che costituiscono una forma di interrupt software (asincrono), sono gestiti a questo livello, così come la memoria. La gran parte dei sistemi UNIX supporta la memoria virtuale con paginazione su richiesta e alle volte fornisce anche delle caratteristiche in più, come la possibilità di condividere regioni dello spazio degli indirizzi tra più processi.

Sin dalla sua nascita, UNIX fu pensato come un sistema dalle dimensioni contenute al fine di incrementarne affidabilità e prestazioni. La prima versione di UNIX era tutta testuale e usava terminali che potevano visualizzare 24 o 25 righe di 80 caratteri ASCII. L'interfaccia utente era gestita dalla shell ("conchiglia, involucro"), un programma che girava a livello utente e offriva un'interfaccia a linea di comando. Poiché la shell non faceva parte del kernel era facile aggiungere a UNIX nuove shell, e con il passare del tempo ne sono state realizzate molte altre, sempre più sofisticate.

In seguito all'introduzione dei primi terminali grafici, venne sviluppato presso il M.I.T. un sistema a finestre per UNIX chiamato X Windows, poi raffinato mediante l'aggiunta di Motif, una GUI (*Graphical User Interface*, "interfaccia grafica per l'utente") più elaborata. Queste interfacce si sono con il tempo evolute in veri e propri ambienti desktop con un'accattivante gestione a finestre, dotati di strumenti per aumentare la produttività e di programmi di utilità. Esempi di questi ambienti desktop sono GNOME e KDE. Quasi tutto il codice di X Windows e delle GUI che lo accompagnano girano a livello utente, fuori dal kernel, perché si attengono alla filosofia di UNIX che vuole il kernel di dimensioni contenute.

## Windows 7

Al suo lancio nel 1981, il primo PC IBM era equipaggiato con MS-DOS 1.0, un sistema operativo a 16 bit, a singolo utente e orientato alla linea di comando. Questo sistema operativo era costituito da 8 KB di codice residente in memoria. Due anni dopo fece la sua comparsa MS-DOS 2.0, un sistema molto più potente da 24 KB; conteneva un elaboratore di linea di comando (la shell) più un certo numero di caratteristiche prese in prestito da UNIX. Nel 1984 IBM lanciò sul mercato i PC/AT basati sul 286 ed equipaggiati con MS-DOS 3.0, che aveva ormai raggiunto i 36 KB. Con il passare degli anni MS-DOS continuò ad acquisire nuove funzionalità, ma restava pur sempre un sistema orientato alla linea di comando.

Ispirata dal successo dei Macintosh di Apple, Microsoft decise di dotare MS-DOS di un'interfaccia grafica per l'utente che chiamò Windows. Le prime tre versioni di Windows, culminate nella versione 3.x, non erano veri sistemi operativi, bensì interfacce grafiche per l'utente montate su MS-DOS, che manteneva ancora il controllo della macchina. Tutti i programmi giravano nello stesso spazio degli indirizzi e così un baco in uno di loro poteva arrestare malamente l'intero sistema.

La distribuzione di Windows 95 nel 1995 mantenne MS-DOS, anche se introduceva la nuova versione 7.0. Windows 95 e MS-DOS 7.0 assommavano insieme quasi tutte le

caratteristiche di un sistema operativo completo, compresa la memoria virtuale, la gestione dei processi e la multiprogrammazione. Tuttavia Windows 95 non era un programma a 32 bit effettivi, ma conteneva grosse porzioni di codice a 16 bit (affiancate al codice a 32 bit) e si appoggiava ancora sul file system di MS-DOS, ereditandone quasi tutti i limiti. I soli cambiamenti di una certa importanza furono l'aggiunta del supporto dei nomi di file lunghi, a estendere gli 8 + 3 caratteri consentiti in MS-DOS, e la possibilità di avere più di 65.536 blocchi su un disco.

MS-DOS sopravvisse anche al lancio di Windows 98 nel 1998 (anche se era arrivato nel frattempo alla versione 7.1) e continuò a eseguire codice a 16 bit. Nonostante la migrazione di qualche altra funzionalità dalla parte MS-DOS verso Windows e l'organizzazione dei dischi che rendeva standard il supporto dei dischi grandi, al di là delle apparenze, Windows 98 non era poi così diverso da Windows 95. La differenza principale stava nell'interfaccia per l'utente che integrava in modo più stretto il desktop, Internet e la riproduzione video. Fu questa integrazione ad attirare l'attenzione del Dipartimento di Giustizia degli Stati Uniti che citò Microsoft in giudizio per aver assunto una posizione illegale di monopolio. A Windows 98 succedette per breve tempo Windows Millennium Edition (ME) che ne costituiva una versione leggermente migliorata.

Parallelamente a questi sviluppi, Microsoft stava lavorando alacremente al completamento di un nuovo sistema operativo a 32 bit, per la cui scrittura era partita da zero, e che chiamò Windows New Technology o Windows NT. In un primo momento venne presentato come il sistema operativo che avrebbe rimpiazzato tutti gli altri nei PC basati su Intel (così come i PowerPC MIPS), ma per qualche ragione ebbe una diffusione lenta e venne più tardi ricollocato nelle fasce alte di mercato, dove conquistò una propria nicchia. La seconda versione di NT prese il nome di Windows 2000 e divenne la versione di punta, anche nel mercato dei desktop. XP è il successore di Windows 2000, ma si tratta in questo caso di cambiamenti secondari (una migliore compatibilità con i sistemi precedenti e alcune nuove funzioni). Nel 2007 venne rilasciato Windows Vista, in cui vennero implementate numerose migliorie grafiche e aggiunte diverse applicazioni per gli utenti, come Media Center. L'adozione di Vista venne rallentata dalle sue scarse prestazioni e dalle alte richieste in termini di risorse. Solo due anni dopo vide la luce Windows 7, che per molti aspetti non è altro che una versione di Vista risistemata. Windows 7 funziona meglio sull'hardware più datato e richiede molte meno risorse.

Windows 7 è commercializzato in sei diverse versioni. Tre di queste sono destinate agli utenti home dei diversi paesi, due agli utenti business e l'ultima combina tutte le funzioni delle altre. Queste versioni sono pressoché identiche: le differenze risiedono nella loro destinazione d'uso, nelle funzionalità più avanzate e nelle ottimizzazioni che sono incluse. Noi ci concentreremo sugli aspetti centrali del sistema, senza fare ulteriori distinzioni tra le varie versioni.

Prima di addentrarci nell'interfaccia che Windows 7 presenta ai programmati diamo un breve sguardo alla sua struttura interna, illustrata nella Figura 6.32. La struttura di Windows 7 è costituita da un certo numero di moduli organizzati a livelli e che contribuiscono a implementare il sistema operativo. A ogni modulo corrisponde una funzione particolare e un'interfaccia ben definita con gli altri moduli. Quasi tutti i moduli sono scritti in C, anche se una parte dell'interfaccia del dispositivo grafico è scritta in C++ e una piccola porzione dei livelli più bassi è scritta in linguaggio assemblativo.

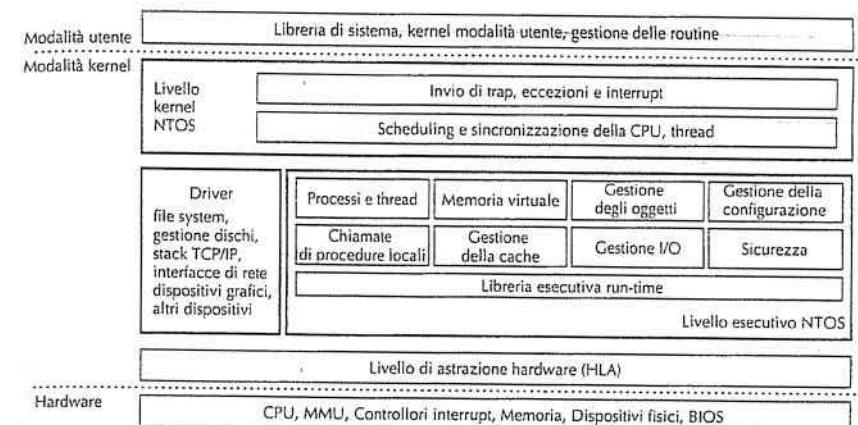


Figura 6.32 La struttura di Windows 7.

Alla base della struttura c'è un sottile **livello di astrazione hardware (hardware abstraction layer)** che ha lo scopo di presentare al resto del sistema operativo dispositivi hardware astratti e privi degli spigli e delle idiosincrasie di cui abbonda l'hardware reale. Tra i dispositivi modellati ci sono le cache non residenti nel chip, i timer, i bus di I/O, i controlleri di interrupt e i DMA. La loro presentazione in forma idealizzata al resto del sistema operativo facilita la portabilità di Windows 7 su altre piattaforme hardware, dal momento che tutte le modifiche da effettuare sono concentrate in un unico posto.

Sopra il livello HAL il codice è diviso in due parti principali: il livello esecutivo NTOS e i driver, che includono il file system, il networking e la grafica. Sopra questi si trova il livello kernel. Tutto questo codice viene eseguito in modalità kernel protetta.

L'esecutivo gestisce le astrazioni fondamentali di Windows 7, tra cui i thread, i processi, la memoria virtuale, gli oggetti kernel e le configurazioni. Nello stesso livello ci sono anche i gestori per le chiamate di procedure locali, la cache dei file, l'I/O e il software per la sicurezza.

Il livello kernel gestisce trap, eccezioni, scheduling e sincronizzazione. Fuori dal kernel troviamo i programmi utente e la libreria di sistema usata per interfacciarsi al sistema operativo. Al contrario di UNIX, Microsoft non vuole incoraggiare i programmati a effettuare chiamate di sistema direttamente, ma preferisce che questi chiamino procedure della libreria. Per garantire conformità tra diverse versioni di Windows (per esempio XP, Vista e 7), Microsoft ha definito un insieme di chiamate detto **API di Win32 (Application Programming Interface, "interfaccia per la programmazione delle applicazioni")**. Si tratta di procedure di libreria che alle volte si avvalgono delle chiamate di sistema per eseguire il compito loro richiesto, alle altre operano direttamente nella procedura di libreria all'interno dello spazio utente. Anche se dalla definizione di Win32 sono state aggiunte diverse chiamate, noi ci concentreremo sul nucleo centrale delle chiamate di libreria. In seguito all'adattamento di Windows alle macchine a 64 bit,

Microsoft ha cambiato il nome di Win32 per includere la versione a 64 bit, ma ai nostri fini basterà l'analisi della prima versione.

La filosofia delle API di Win32 è completamente diversa da quella di UNIX. Nel caso di UNIX le chiamate di sistema sono tutte pubbliche e formano un'interfaccia minimale: l'eliminazione di una sola di loro ridurrebbe la funzionalità del sistema operativo. Al contrario Win32 fornisce un'interfaccia ridondante (ci sono spesso tre o quattro modi diversi di fare la stessa cosa) che comprende anche alcune funzioni che non dovrebbero essere (e che infatti non sono) chiamate di sistema, per esempio una chiamata API per la copia di un file intero.

Molte chiamate API di Win32 servono a creare oggetti del kernel, come file, processi, thread, pipe e così via, e come risultato restituiscono al chiamante l'handle ("impugnatura") dell'oggetto. L'handle può poi essere usato per svolgere le operazioni definite sull'oggetto. L'handle appartiene al processo che ha creato l'oggetto corrispondente e non può essere passato direttamente a un altro processo perché ne faccia uso (nello stesso modo in cui i descrittori di file UNIX non possono essere passati ad altri processi perché ne facciano uso). Tuttavia, in determinate circostanze è possibile duplicare un handle e passarne la copia a un altro processo in modo protetto, consentendogli l'accesso a un oggetto che non gli appartiene. Un handle può essere accompagnato da un descrittore di sicurezza che specifica in dettaglio quali processi possono operare su di esso, e con quali operazioni.

Qualche volta si sente parlare di Windows 7 come di un sistema orientato agli oggetti, perché l'unico modo di manipolare gli oggetti del kernel è di invocare metodi (funzioni API) sui loro handle, che vengono restituiti quando gli oggetti sono creati. D'altra parte, mancano a Windows 7 alcune delle proprietà fondamentali dei sistemi orientati agli oggetti quali l'ereditarietà e il polimorfismo.

### 6.5.2 Esempi di memoria virtuale

In questo paragrafo affrontiamo la memoria virtuale di UNIX e di Windows 7 che, dal punto di vista del programmatore, sono molto simili.

#### Memoria virtuale di UNIX

Tutti i processi UNIX hanno tre segmenti: codice, dati e stack (Figura 6.33). Nelle macchine dotate di un solo spazio lineare d'indirizzi, in genere si pone il codice alla base della memoria, lo si fa seguire dai dati e si alloca lo stack in cima alla memoria stessa. La dimensione del codice è fissa, mentre i dati e lo stack sono liberi di crescere in direzioni opposte. Questo modello è molto semplice da implementare su qualsiasi macchina ed è quello usato dalle varianti di Linux utilizzate sulle CPU ARM OMAP4430.

Inoltre, se la macchina consente la paginazione, l'intero spazio degli indirizzi può essere paginato senza che il programmatore neanche se ne accorga. L'unica cosa che il programmatore sa è che può prendersi più spazio per i suoi programmi di quanto ce ne sia in memoria fisica. Per consentire comunque la condivisione di tempo tra un numero arbitrario di processi, i sistemi UNIX che non dispongono della paginazione, in genere fanno lo swap tra memoria e disco di interi processi per volta.

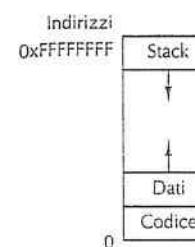


Figura 6.33 Spazio degli indirizzi di un processo UNIX.

Per quanto riguarda la memoria virtuale (con paginazione a richiesta) dello UNIX Berkeley non resta da aggiungere altro. Per il System V (e quindi Linux) invece va aggiunto che esso comprende diverse funzionalità che consentono agli utenti una gestione molto più sofisticata della loro memoria virtuale. Di particolare importanza è la possibilità che un processo mappi un file (o una sua parte) nel proprio spazio degli indirizzi. Per esempio, se un file di 12 KB è mappato all'indirizzo virtuale 144 KB, la lettura della parola all'indirizzo 144 KB equivale alla lettura della prima parola del file. Grazie a questo meccanismo è possibile leggere e scrivere file senza ricorrere alle chiamate di sistema. Dato che la dimensione di un file può eccedere quella dello spazio degli indirizzi virtuali, è possibile mapparne in memoria anche solo una parte. L'operazione comincia con l'apertura del file che restituisce il suo descrittore *fd*, usato successivamente per identificare il file corrispondente da mappare. Quindi il processo effettua la chiamata

```
paddr = mmap(indirizzo_virtuale, lunghezza, protezione, flag, fd, file_offset)
```

che mappa *lunghezza* byte, a partire dalla posizione *offset* all'interno del file, negli indirizzi virtuali che cominciano a *indirizzo\_virtuale*. In alternativa è possibile impostare il parametro *flag* in modo che sia il sistema a scegliere un indirizzo virtuale, restituito nella variabile *paddr*. La regione mappata deve comprendere un numero intero di pagine e deve essere allineata alle loro estremità. Il parametro *protezione* può specificare una combinazione qualsiasi di permessi in lettura, scrittura o esecuzione. La corrispondenza può essere poi rimossa con la chiamata *unmap*.

Più processi possono mappare lo stesso file contemporaneamente. La condivisione può avvenire secondo due opzioni. La prima prevede che tutte le pagine siano condivise, perciò le scritture effettuate da un processo sono visibili a tutti gli altri. Questa opzione mette a disposizione un percorso per la comunicazione tra processi con larghezza di banda elevata. L'altra opzione prevede la condivisione delle pagine fintanto che non siano state modificate; non appena un processo tenta di scrivere in una pagina si verifica un errore di protezione, a seguito del quale il sistema operativo fornisce al processo una copia privata della pagina su cui scrivere. Questo schema, noto con il nome di *copy dopo scrittura* (*copy on write*), torna utile quando si vuol dare a ogni processo l'illusione di essere l'unico a mappare un certo file. In questo modello la condivisione è un'ottimizzazione, e non parte della semantica.

### Memoria virtuale di Windows 7

In Windows 7 ogni processo utente ha il proprio spazio degli indirizzi virtuali. Nella versione a 32 bit di Windows 7, gli indirizzi virtuali sono lunghi 32 bit, così ogni processo ha uno spazio degli indirizzi virtuali di 4 GB. I primi 2 GB sono disponibili per il codice e i dati, i successivi 2 GB permettono l'accesso (limitato) alla memoria del kernel, fatta eccezione per le versioni Server di Windows, in cui la suddivisione è di 3 GB per l'utente e 1 GB per il kernel. Lo spazio degli indirizzi virtuali è paginato a richiesta, con una dimensione di pagina fissa (4 KB nel Core i7). Lo spazio degli indirizzi nella versione a 64 bit di Windows 7 è simile. Tuttavia, lo spazio di codice e dati è formato dagli 8 terabyte più bassi dello spazio degli indirizzi virtuali.

Le pagine virtuali si possono trovare in uno dei tre stati seguenti: libero, riservato o impegnato. Una **pagina libera** è una pagina non attualmente in uso; un riferimento a essa causa un errore di pagina. All'avvio di un processo tutte le sue pagine sono libere finché non comincia la corrispondenza tra programma e dati nel suo spazio degli indirizzi. Una volta che una parte del codice o dei dati è mappata in una pagina, questa si dice **impegnata (committed)**. Un riferimento a una pagina impegnata viene mappato per mezzo dell'hardware della memoria virtuale e ha successo solo se la pagina si trova in memoria principale. In caso contrario, si verifica un errore di pagina e il sistema operativo si preoccupa di caricare la pagina dal disco. Una pagina virtuale può anche essere **riservata** per indicare che non è disponibile alla corrispondenza, a meno che la riserva venga rimossa esplicitamente. Le pagine riservate vengono usate quando una sequenza di pagine consecutive può essere necessaria in futuro, come nel caso dello stack. Oltre a questi attributi, una pagina può anche essere leggibile, scrivibile o eseguibile. I primi e gli ultimi 64 KB di memoria sono sempre liberi al fine di intercettare gli errori di puntamento (i puntatori non inizializzati valgono in genere 0 o -1).

A ciascuna pagina impegnata corrisponde su disco una **pagina ombra (shadow page)** dove viene conservata quando non è presente in memoria principale. Le pagine libere o riservate non hanno una pagina ombra, perciò i riferimenti a esse causano errori di pagina (il sistema non può caricare una pagina dal disco se questa non esiste). Le pagine ombra sono organizzate su disco in uno o più file di paginazione. Il sistema operativo si preoccupa di tener traccia della corrispondenza tra le pagine virtuali e le parti dei file di paginazione. Nel caso del testo del programma (con permesso di sola esecuzione) è il file binario eseguibile a contenere le pagine ombra; le pagine di dati sono invece contenute in file speciali.

Windows 7, al pari di System V, permette la corrispondenza diretta tra file e regioni dello spazio degli indirizzi virtuali (cioè in sequenze di pagine consecutive). Una volta mappato nello spazio degli indirizzi, un file può essere letto o scritto tramite le comuni operazioni di riferimento alla memoria.

I file mappati in memoria sono implementati allo stesso modo delle pagine impegnate, con la sola differenza che le pagine ombra si trovano in questo caso nei file stessi invece che nel file di paginazione. Di conseguenza, è possibile che la versione di un file mappato in memoria sia diversa da quella su disco (a causa di scritture recenti nello spazio degli indirizzi virtuali). Tuttavia, quando il file viene estromesso dalla memoria o a seguito di una richiesta esplicita di *flush*, la versione su disco viene aggiornata dalla memoria.

Windows 7 permette espressamente la corrispondenza contemporanea di un file da parte di due o più processi, anche a indirizzi virtuali differenti. Grazie alla lettura e scrittura di parole di memoria, i processi possono comunicare tra loro e scambiarsi dati con una larghezza di banda molto elevata, poiché non è richiesta alcuna operazione di copia. Processi diversi possono avere permessi d'accesso diversi. Poiché tutti i processi che mappano lo stesso file condividono le stesse pagine, i cambiamenti effettuati da uno di loro sono visibili immediatamente a tutti gli altri, anche se non è ancora stata aggiornata la copia su disco.

Le API di Win32 contengono diverse funzioni per consentire a un processo la gestione esplicita della propria memoria virtuale; le più importanti sono elencate nella Figura 6.34. Tutte queste funzioni operano su regioni costituite da una singola pagina o da una sequenza di due o più pagine consecutive nello spazio degli indirizzi virtuali.

Funzione API	Significato
VirtualAlloc	Riserva o impegnava una pagina
VirtualFree	Libera o disimpegnava una pagina
VirtualProtect	Cambia la protezione in lettura/scrittura/esecuzione di una regione
VirtualQuery	Esamina lo stato di una regione
VirtualLock	Rende una regione residente in memoria (cioè ne disabilita la paginazione)
VirtualUnlock	Rende una regione paginabile nel modo consueto
CreateFileMapping	Crea un oggetto di mappatura file e gli assegna un nome (se specificato)
MapViewOfFile	Colloca (una parte di) un file nello spazio degli indirizzi
UnmapViewOfFile	Rimuove un file mappato dallo spazio degli indirizzi
OpenFileMapping	Apre un oggetto di mappatura file precedentemente creato

Figura 6.34 Le più importanti chiamate API di Windows XP per la gestione della memoria virtuale.

Le prime quattro funzioni API sono autoesplicative. Con le altre due, un processo può costringere la paginazione a tenere un certo numero di pagine in memoria o può rimuovere questo vincolo. La cosa può essere utile alle applicazioni in tempo reale, ma è consentita solo ai programmi dell'amministratore di sistema. Inoltre, il sistema operativo stabilisce dei vincoli cui devono attenersi i processi perché non diventino troppo avidi. Seppur non elencate nella Figura 6.34, Windows 7 dispone di funzioni API per consentire a un processo di accedere alla memoria virtuale di un altro processo per il quale ha ricevuto il controllo (cioè di cui ha un handle).

Le ultime quattro funzioni API elencate servono alla gestione dei file mappati in memoria. La mappatura di un file comincia con la creazione di un oggetto di mappatura file mediante *CreateFileMapping*. La funzione restituisce un handle dell'oggetto di mappatura file e, se richiesto, lo associa a un nome nel file system perché possa essere usato da un altro processo. Le due funzioni seguenti servono rispettivamente alla mappatura e alla rimozione di file dalla memoria. Un file mappato è un file su disco (o una sua parte) che può essere letto o scritto accedendo solamente allo spazio degli indirizzi

virtuali, senza esplicite operazioni di I/O. L'ultima funzione può essere invocata da un processo per mappare un file già mappato da un altro processo. In questo modo, due o più processi possono condividere regioni dei rispettivi spazi degli indirizzi.

Queste sono le funzioni API basilari attorno alle quali è costruito il sistema di gestione della memoria. Per fare un esempio, esistono funzioni API per l'allocazione o la rimozione di strutture dati in uno o più *heap* (“mucchio, cumulo”). Gli heap sono usati per la memorizzazione di quelle strutture dati che vengono create e distrutte dinamicamente e che non sono oggetto di *garbage collection* (l'operazione di rimozione automatica delle strutture dati inutilizzate a cura del sistema), perciò sta al software dell'utente liberare i blocchi di memoria virtuale che non sono più in uso. La gestione dello heap in Windows 7 è simile alla funzione *malloc* dei sistemi UNIX, con la differenza che in Windows 7 ci possono essere più heap gestiti indipendentemente.

### 6.5.3 Esempi di I/O a livello OS

L'obiettivo principale del sistema operativo è quello di fornire servizi ai programmi utente, soprattutto servizi di I/O come la scrittura e la lettura di file. Sia UNIX sia Windows 7 offrono un ampio ventaglio di servizi di I/O ai programmi utente. Se è vero che esiste una chiamata Windows 7 equivalente a ogni chiamata UNIX, non vale l'inverso, perché Windows 7 possiede molte più chiamate e ogni sua chiamata è molto più complessa della controparte UNIX.

#### I/O virtuale di UNIX

Buona parte del successo di UNIX risiede nella sua semplicità che, a sua volta, è una conseguenza diretta dell'organizzazione del file system. Un file ordinario è una sequenza lineare di byte che comincia alla posizione 0 e prosegue fino a un massimo di  $2^{64} - 1$  byte. Il sistema operativo non impone alcuna struttura ai file, anche se gli utenti interpretano i file di testo ASCII come una successione di righe, ciascuna completata da un carattere di fine linea.

A ogni file aperto è associato un puntatore al byte successivo da leggere o in cui scrivere. Le chiamate di sistema *read* e *write* leggono o scrivono dati a partire dalla posizione nel file indicata dal puntatore. Entrambe le chiamate implicano l'avanzamento del puntatore di un numero di posizioni uguale al numero di byte trasferiti. È tuttavia possibile l'accesso diretto a una posizione del file fornendone il valore al puntatore.

Oltre ai file ordinari, UNIX supporta anche file speciali, usati per accedere ai dispositivi di I/O. In genere ogni dispositivo di I/O è associato a uno o più file speciali. Un programma può utilizzare il dispositivo di I/O attraverso la lettura o scrittura su questi file. È questo il modo in cui vengono gestiti dischi, stampanti, terminali e molti altri dispositivi.

La Figura 6.35 elenca le chiamate di UNIX più importanti relative al file system. La chiamata *creat* (*CREATE* sarebbe sbagliato!) è utilizzabile per creare un nuovo file, anche se non è più necessaria perché nelle ultime versioni anche la funzione *open* è in grado di farlo. *unlink* cancella un file, nell'ipotesi che questo si trovi in una sola directory.

Chiamata di sistema	Significato
<i>creat(name, mode)</i>	Crea un file (mode specifica la modalità di protezione)
<i>unlink(name)</i>	Cancella un file (assumendo che ci sia un solo link a esso)
<i>open(name, mode)</i>	Apre o crea un file e restituisce il suo descrittore
<i>close(fd)</i>	Chiude un file
<i>read(fd, buffer, count)</i>	Legge count byte in un buffer
<i>write(fd, buffer, count)</i>	Scrive count byte da un buffer
<i>lseek(fd, offset, w)</i>	Sposta il puntatore del file secondo offset e w
<i>stat(name, buffer)</i>	Restituisce le informazioni sul file
<i>chmod(name, mode)</i>	Cambia la modalità di protezione del file
<i>fentl(fd, cmd, ...)</i>	Esegue varie operazioni di controllo come il lock di (parte di) un file

Figura 6.35 Chiamate principali per il file system di UNIX.

La chiamata *open* si usa per aprire file esistenti (e per crearne di nuovi). L'indicatore *mode* specifica la modalità d'apertura (in lettura, scrittura e così via). La chiamata restituisce come risultato il **descrittore di file**, un piccolo intero che serve a identificare il file nelle chiamate successive.

L'I/O propriamente detto si svolge con le chiamate *read* e *write*, le quali dispongono di un descrittore del file da usare, un buffer per i dati da leggere o contenente i dati da scrivere, più un valore *count* che stabilisce la quantità di dati da trasmettere. *lseek* serve a spostare il puntatore al file, in modo da realizzare l'accesso diretto alle sue posizioni.

La chiamata *stat* serve per ottenere informazioni sui file, quali la dimensione, la data dell'ultimo accesso, il proprietario e altro ancora. *chmod* modifica la modalità di protezione del file, per esempio consentendo o negando la lettura del file agli utenti che non ne sono proprietari. Infine *fentl* svolge svariate operazioni, tra cui l'aggiunta o la rimozione di un blocco che indica se il file è stato riservato o meno per l'accesso esclusivo da parte di un processo.

La Figura 6.36 illustra il funzionamento delle principali chiamate di I/O; il codice raffigurato è ridotto, in quanto non comprende il pur necessario controllo degli errori. Prima di entrare nel ciclo, il programma apre un file esistente, *data*, e ne crea uno nuovo, *newf*; le due chiamate prendono in ingresso come secondo parametro i bit di protezione che specificano la modalità dei due file, rispettivamente in lettura e in scrittura, e restituiscono a loro volta due descrittori di file, *infid* e *outfd*. Se una delle due chiamate non dovesse andare a buon fine, restituirebbe un descrittore di file negativo, a indicare l'insuccesso.

La procedura *read* ha tre parametri: un descrittore di file, un buffer e un numero di byte. La *read* prova a leggere dal file il numero di byte specificato e a porlo nel buffer, ma il numero di byte effettivamente letto è restituito in *count* e sarà minore di *bytes* se il file è troppo corto. La chiamata *write* ripone i byte appena letti nel file di output. Il ciclo prosegue finché non termina la lettura di tutto il file di input, al che il ciclo termine e i file vengono chiusi.

```

/* Apertura dei descrittori di file. */
infd = open("data", 0);
outfd = creat("new!", ProtectionBits);

/* Ciclo di copia. */
do {
    count = read(infd, buffer, bytes);
    if (count > 0) write(outfd, buffer, count);
} while (count > 0);

/* Chiusura dei file. */
close(infd);
close(outfd);

```

**Figura 6.36** Frammento di programma per la copia di un file con chiamate di sistema UNIX. Usiamo il C perché mette bene in evidenza le chiamate di sistema di livello più basso (a differenza di Java, che le nasconde).

I descrittori di file di UNIX sono dei numeri interi (in genere minori di 20): i descrittori 0, 1 e 2 sono speciali e riservati rispettivamente a **standard input**, **standard output** e **standard error**, che puntano di norma alla tastiera, allo schermo e ancora allo schermo, ma che possono essere rediretti<sup>1</sup> su file specificati dall'utente. Molti programmi di UNIX prendono dati in ingresso dallo standard input e scrivono in uscita sullo standard output; spesso sono chiamati **filtri**.

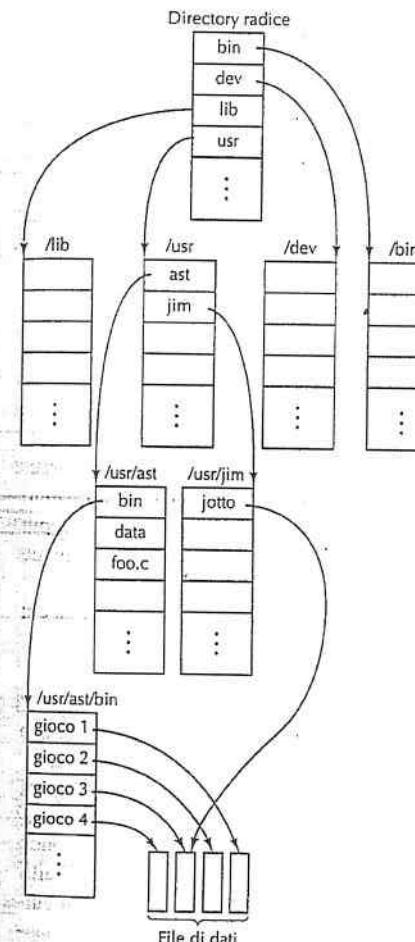
Intimamente legato al file system è il directory system. Ogni utente può possedere molte directory, che a loro volta possono contenere file o altre sottodirectory. I sistemi UNIX sono configurati spesso come mostrato dalla Figura 6.37, con una directory principale, la **directory radice** (*root directory*), che contiene le sottodirectory *bin* (per i programmi di esecuzione frequente), *dev* (per i file speciali di dispositivi di I/O), *lib* (per le librerie) e *usr* (per le directory degli utenti). Nell'esempio la directory *usr* contiene le sottodirectory *ast* e *jim*; la prima contiene a sua volta due file, *data* e *foo.c*, e una sottodirectory, *bin*, che contiene quattro giochi.

Quando sono presenti diversi dischi (o diverse partizioni dello stesso disco), questi possono essere montati (*mount*) sull'albero dei nomi in modo che tutti i file di tutti i dischi appaiano nella stessa gerarchia di directory e che siano tutti raggiungibili dalla directory radice.

Un file è rintracciabile tramite il suo **percorso** (*path*) a partire dalla radice. Un percorso contiene la lista di tutte le directory attraversate dalla radice fino al file, separate da un carattere barra. Per esempio, il percorso assoluto del file *gioco2* è */usr/ast/bin/gioco2*. Si dice **percorso assoluto** un percorso di file che comincia dalla radice.

A ciascun programma in esecuzione è associata una **directory di lavoro** (*working directory*). I percorsi dei file possono essere specificati relativamente alla directory di lavoro, nel qual caso cominciano con una barra, perché siano distinguibili dai percorsi

assoluti. Parliamo in tal caso di **percorsi relativi**: se la directory di lavoro è */usr/ast*, *gioco3*, il file è raggiungibile usando il percorso relativo *bin/gioco3*. Un utente può creare un **collegamento** (*link*) a un file di un altro utente grazie alla chiamata di sistema *link*. Nell'esempio precedente */usr/ast/bin/gioco3* e */usr/jim/jotto* puntano allo stesso file. Non sono permessi collegamenti a directory per evitare la formazione di cicli nel directory system. Le chiamate *open* e *creat* accettano come parametro sia percorsi relativi sia assoluti.



**Figura 6.37** Parte del directory system di un tipico sistema UNIX.

<sup>1</sup> Per redirezione di un puntatore (descrittore di file) si intende l'attribuzione a esso di una nuova locazione (direzione) a cui puntare (N.d.T.).

Le principali chiamate di sistema per la gestione delle directory UNIX sono elencate nella Figura 6.38. `mkdir` crea una nuova directory e `rmdir` rimuove una directory (vuota) già esistente. Le successive tre chiamate si usano per leggere gli elementi di una directory: la prima apre la directory, la seconda legge i suoi elementi, la terza chiude la directory. `chdir` cambia la directory di lavoro.

Chiamata di sistema	Significato
<code>mkdir(name, mode)</code>	Crea una directory nuova
<code>rmdir(name)</code>	Elimina una directory vuota
<code>opendir(name)</code>	Apre una directory per la lettura
<code>readdir(dirpointer)</code>	Legge l'elemento successivo di una directory
<code>closedir(dirpointer)</code>	Chiude una directory
<code>chdir(dirname)</code>	Cambia la directory di lavoro in <i>dirname</i>
<code>link(name1, name2)</code>	Crea un elemento di directory <i>name2</i> che punta a <i>name1</i>
<code>unlink(name)</code>	Rimuove <i>name</i> dalla sua directory

Figura 6.38 Chiamate principali di UNIX per la gestione delle directory.

`link` crea in una directory un nuovo elemento che punta a un file esistente. Per esempio, l'elemento `/usr/jim/jotto` potrebbe essere stato creato dalla chiamata

```
link("/usr/ast/bin/gioco3", "/usr/jim/jotto")
```

o da una chiamata equivalente con, in ingresso, percorsi relativi dipendenti dalla directory di lavoro del programma che effettua la chiamata. `unlink` rimuove un elemento dalla directory: se il file ha un solo collegamento viene cancellato, ma se ne ha di più viene mantenuto. Non ha alcuna importanza se il collegamento rimosso era l'originale o una copia.

Una volta creato, un collegamento ha tutti i diritti e le proprietà dell'originale, da cui è indistinguibile. La chiamata

```
unlink("/usr/ast/bin/gioco3")
```

rende *gioco3* accessibile d'ora in poi solo attraverso il percorso `/usr/jim/jotto`. In questo modo è possibile usare `link` e `unlink` per "trasferire" file da una directory all'altra.

A ogni file (e alle directory, che sono esse stesse dei file) è associata una stringa di bit che specifica chi può accedervi, ed è costituita da tre campi **RWX**: il primo controlla i permessi di lettura (R), scrittura (W) ed esecuzione (X) del proprietario, il secondo controlla i permessi degli altri utenti dello stesso gruppo, il terzo riguarda tutti gli altri utenti. Così **RWX R-X --X** vuol dire che il proprietario può leggere, scrivere ed eseguire il file (se si tratta di un programma eseguibile, altrimenti la X sarebbe assente), mentre gli altri utenti del suo gruppo possono solo leggerlo o eseguirlo e gli estranei possono solo eseguirlo. Con questi permessi gli estranei possono usare il programma, ma non rubarlo (copiarlo), poiché non hanno il permesso in lettura. L'attribuzione di un gruppo

agli utenti è fatta dall'amministratore di sistema, chiamato **in genere superuser**, che ha anche il potere di scavalcare il meccanismo di protezione e può leggere, scrivere o eseguire qualunque file.

Veniamo ora brevemente all'implementazione dei file e delle directory UNIX (per una trattazione completa si veda Vahalia, 1996). A ogni file (e quindi a ogni directory) è associato un blocco d'informazioni di 64 byte detto **i-node**, che contiene le informazioni sul proprietario e sui permessi del file, sulla locazione dei dati e altre cose simili. Gli i-node dei file sono conservati in sequenza all'inizio del disco o all'inizio di un gruppo di cilindri, se il disco è così suddiviso. UNIX è in grado di calcolare l'indirizzo su disco di un i-node a partire dal suo numero nella sequenza.

Gli elementi delle directory consistono in due parti: un nome di file e un numero di i-node. Quando un programma esegue

```
open("foo.c", 0)
```

il sistema cerca il file di nome "foo.c" nella directory di lavoro al fine di localizzarne il numero di i-node. Una volta trovato il numero può accedere all'i-node e a tutte le informazioni sul file.

Se viene specificato un percorso più lungo, i passi precedenti vengono ripetuti finché non sia stato attraversato l'intero percorso. Per esempio, per localizzare il numero di i-node di `/usr/ast/data`, il sistema cerca prima l'elemento *usr* nella directory radice. Quando ha trovato l'i-node di *usr* può leggere il file corrispondente (ripetiamo: in UNIX le directory sono file) e cercare al suo interno l'elemento *ast*, arrivando a localizzare il numero di i-node del file `/usr/ast`. La lettura di `/usr/ast` consente al sistema di trovare l'elemento *data* e quindi il numero di i-node di `/usr/ast/data`. Grazie a quel numero è possibile trovare tutto quanto riguarda il file suddetto.

Il formato, i contenuti e l'organizzazione degli i-node variano leggermente da sistema a sistema (specie se il sistema svolge attività di rete), ma almeno le voci seguenti si trovano in tutte le versioni di i-node:

1. il tipo del file, i 9 bit di protezione **RWX** e qualche altro bit;
2. il numero di collegamenti al file (numero di elementi di directory che lo punta);
3. l'identità del proprietario;
4. il gruppo del proprietario;
5. la lunghezza del file in byte;
6. tredici indirizzi su disco;
7. la data dell'ultimo accesso in lettura;
8. la data dell'ultimo accesso in scrittura;
9. la data dell'ultima modifica all'i-node.

Riguardo il tipo, un file può essere un file ordinario, una directory o uno dei due tipi di file speciali che si usano per distinguere i dispositivi di I/O strutturati a blocchi da quelli non strutturati. Abbiamo già analizzato il numero di collegamenti e l'identificativo del proprietario. La lunghezza del file è data da un intero di 32 bit che specifica l'ultima

locazione del file con un contenuto informativo: è del tutto lecito creare un file, fare una `lseek` alla posizione 1.000.000 e scrivere un byte, nel qual caso la lunghezza del file è di 1.000.001 byte. Va detto però che l'eventuale memorizzazione del file non richiederebbe alcuno spazio per i byte "mancanti".

I primi 10 indirizzi del disco puntano a blocchi di dati. Se la dimensione di un blocco è 1024 byte, in questo modo è possibile gestire file fino alla dimensione di 10.240 byte. L'indirizzo 11 punta a un blocco del disco che si chiama **blocco indiretto**, contenente diversi indirizzi del disco. Con un blocco di 1024 byte e indirizzi del disco di 32 bit, il blocco indiretto conterebbe 256 indirizzi del disco. In questo modo è possibile gestire file fino alla dimensione di  $10.240 + 256 \times 1024 = 272.384$  byte. Nel caso dei file ancora più grandi si usa l'indirizzo 12 che punta a un blocco contenente 256 blocchi indiretti, riuscendo a gestire così file di  $272.384 + 256 \times 256 \times 1024 = 67.381.248$  byte. Se anche questo schema con **blocco indiretto doppio** non è sufficiente per contenere il file, si usa l'indirizzo 13 per puntare a un **blocco indiretto triplo** contenente 256 indirizzi di blocchi indiretti doppi. Usando congiuntamente gli indirizzi dei blocchi diretti, indiretti, indiretti doppi e tripli si possono indirizzare 16.843.018 blocchi, raggiungendo il limite teorico di dimensione per i file di 17.247.250.432 byte. Se gli indirizzi del disco sono di 64 bit invece che di 32 bit e i blocchi sono di 4 KB, i file possono essere davvero molto, molto grandi. In realtà, essendo i puntatori di file di 32 bit, il limite pratico è di 4.294.967.295 byte. I blocchi liberi del disco sono mantenuti in una lista concatenata da cui vengono rimossi uno alla volta quando necessari. Di conseguenza, i blocchi di ciascun file si trovano sparpagliati alla rinfusa per tutto il disco.

Al fine di rendere le operazioni di I/O più efficienti, all'apertura di un file il suo inode viene copiato (per ragioni di comodità) in una tabella in memoria principale e lì mantenuto finché il file non viene chiuso. Inoltre viene tenuta in memoria anche una raccolta dei blocchi referenziati di recente; poiché i file vengono letti in modo sequenziale nella stragrande maggioranza dei casi, succede spesso che un riferimento alla locazione di un file si trovi all'interno dello stesso blocco del riferimento precedente. Motivato da questa considerazione, il sistema cerca di leggere anche il blocco *successivo* a quello correntemente in lettura prima che venga effettivamente referenziato, in modo da accelerare l'esecuzione. Tutti questi meccanismi di ottimizzazione sono nascosti all'utente: quando un utente effettua una `read`, il suo programma resta sospeso finché non siano disponibili nel buffer i dati da lui richiesti.

Alla luce di queste nozioni, possiamo ora capire come funziona l'I/O su file. La `open` induce il sistema a cercare le directory nel percorso specificato: se la ricerca ha successo, l'i-node viene caricato nella tabella interna. Le chiamate `read` e `write` richiedono al sistema il calcolo del numero di blocco a partire dalla posizione corrente nel file. Gli indirizzi su disco dei primi 10 blocchi si trovano sempre in memoria centrale (nell'i-node); i blocchi di numero maggiore richiedono la lettura preventiva di uno o più blocchi indiretti. `lseek` si limita a spostare il puntatore alla posizione corrente senza effettuare alcun I/O.

È ora semplice comprendere anche `link` e `unlink`: `link` cerca all'interno del primo argomento il numero di i-node, quindi crea un elemento di directory nel secondo argomento, assegnandogli lo stesso numero di i-node. Infine incrementa di un'unità il nume-

ro di collegamenti nell'i-node. `unlink` rimuove un elemento da una directory e decrementa il numero di collegamenti nell'i-node. Se il numero arriva a zero, il file viene cancellato e i suoi blocchi resi disponibili tramite la loro inserzione nella lista dei blocchi liberi.

### I/O di Windows 7

Windows 7 supporta molti file system, tra cui i più importanti sono NTFS (*NT File System*) e FAT (*File Allocation Table*, "tabella di allocazione dei file"). NTFS è stato sviluppato specificamente per NT; FAT è il vecchio file system di MS-DOS, usato anche da Windows 95/98 (anche se con l'introduzione dei nomi di file lunghi). Poiché FAT è ormai obsoleto, eccezion fatta per le chiavette USB e le schede di memoria per fotocamere, ci limitiamo allo studio del primo.

I nomi di file in NTFS possono essere lunghi un massimo di 255 caratteri e utilizzano Unicode per consentire agli utenti di quei paesi la cui lingua non deriva dal latino (per esempio i giapponesi, gli indiani o gli israeliani), di attribuire ai file nomi nella loro lingua madre. Per la precisione, Windows 7 usa Unicode estesamente al proprio interno; fin da Windows 2000 il sistema viene distribuito in un unico codice binario e le differenze linguistiche, che si manifestano nei menu, nei messaggi di errore e così via, sono gestite tramite file di configurazione differenziati per nazione. NTFS è un sistema completamente *case sensitive* (cioè, per esempio, *pippo* è diverso da *PIPPO*) ma sfortunatamente le API di Win32 lo sono solo parzialmente per i nomi di file e non lo sono affatto per i nomi di directory, così questa proprietà viene a mancare ai programmi che usano Win32.

Come per UNIX, un file Windows 7 è semplicemente una sequenza lineare di byte, che però può raggiungere i  $2^{64} - 1$  byte. I puntatori di file esistono, come in UNIX, ma sono di 64 bit invece che di 32, in modo da poter gestire file della massima lunghezza possibile. Le chiamate di funzioni API di Win32 per la manipolazione dei file e delle directory sono abbastanza simili alla loro controparte UNIX, a parte il fatto che hanno spesso più parametri e che si avvalgono di un modello di sicurezza diverso. L'apertura di un file restituisce un handle, poi usato per la lettura o la scrittura del file. A differenza di UNIX, però, gli handle non sono piccoli interi, perché devono servire a identificare potenzialmente milioni di oggetti kernel.

Le funzioni principali delle API di Win32 per la gestione dei file sono elencate nella Figura 6.39.

Veniamo ora alle chiamate di sistema. `CreateFile` genera un file nuovo e restituisce il suo handle; si usa anche per aprire i file già esistenti, dato che non esiste una funzione `open` specifica. Nella figura non abbiamo incluso i parametri delle funzioni API di Windows 7 perché sono molto voluminosi. Solo per fare un esempio, i parametri della `CreateFile` sono sette:

1. un puntatore al nome del file da creare o da aprire;
2. dei flag per indicare se il file può essere letto, scritto, o entrambe le cose;
3. dei flag che specificano se il file può essere aperto contemporaneamente da più processi;

4. un puntatore al descrittore di sicurezza che stabilisce chi può accedere al file;
5. dei flag che indicano il comportamento da assumere se il file esiste/non esiste;
6. dei flag che riguardano gli attributi del file, come il suo essere un file di archivio, compresso e così via;
7. l'handle di un file da cui clonare gli attributi per l'attribuzione al nuovo file.

Funzione API	UNIX	Significato
CreateFile	open	Crea un file o apre un file esistente; restituisce l'handle
DeleteFile	unlink	Cancella da una directory la voce relativa a un file esistente
CloseHandle	close	Chiude un file
ReadFile	read	Legge dati da un file
WriteFile	write	Scrive dati in un file
SetFilePointer	lseek	Assegna al puntatore del file una determinata posizione
GetFileAttributes	stat	Restituisce le informazioni sul file
LockFile	fcntl	Mette in lock una regione del file per garantire la mutua esclusione
UnlockFile	fcntl	Toglie il lock da una regione di un file precedentemente riservata

Figura 6.39 Funzioni principali delle API di Win32 per l'I/O di file. La seconda colonna riporta la funzione di UNIX più simile.

Le successive sei funzioni API della Figura 6.39 sono abbastanza simili alle corrispondenti chiamate di UNIX. Si noti tuttavia che l'I/O di Windows 7 è in linea di principio asincrono, anche se un processo può aspettare il completamento. Le ultime due consentono di porre o di togliere il lock su una regione di un file per garantirne l'accesso esclusivo da parte di un processo (ovvero la mutua esclusione tra processi).

A partire da queste funzioni API è possibile scrivere una procedura per la copia di un file analoga a quella della Figura 6.36. La Figura 6.40 mostra una siffatta procedura (senza controllo degli errori), concepita sul modello della versione UNIX. Nella pratica non c'è alcun bisogno di usare il codice della figura, perché esiste la funzione API `CopyFile` che chiama una procedura di libreria il cui codice è simile a quello indicato nell'esempio.

Windows 7 supporta un file system gerarchico simile a quello di UNIX, anche se il carattere separatore dei componenti di un nome è \ invece di /, un retaggio di MS-DOS. Esiste il concetto di directory di lavoro e i nomi di percorso possono essere assoluti o relativi. C'è comunque una differenza sostanziale rispetto a UNIX: UNIX permette di "montare" (mount) insieme i file system di dischi o di macchine diversi in un unico albero dei nomi a partire da una radice comune, nascondendo così la struttura dei dischi al software. Le versioni più vecchie di Windows (pre Windows 2000) non avevano questa proprietà, perciò i nomi di file assoluti devono cominciare con la lettera del drive (l'unità) che indica il disco logico in cui si trova il file, per esempio in `C:\windows\`.

`system\foo.dll`. A partire da Windows 2000 è stata introdotta la possibilità di montare un file system nello stile di UNIX.

```
/* Apertura dei file per input e output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

/* Copia del file. */
do {
    s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
    if (s > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Chiusura dei file. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figura 6.40 Frammento di programma per la copia di un file con le funzioni API di Windows 7. Usiamo il C perché mette bene in evidenza le chiamate di sistema di livello più basso (a differenza di Java, che le nasconde).

La Figura 6.41 elenca le funzioni API più importanti per la gestione delle directory insieme alle corrispondenti funzioni UNIX. I loro nomi sono sufficientemente autoesplcativi.

Funzione API	UNIX	Significato
CreateDirectory	mkdir	Crea una directory nuova
RemoveDirectory	rmdir	Elimina una directory vuota
FindFirstFile	opendir	Prepara alla lettura degli elementi di una directory
FindNextFile	readdir	Legge l'elemento di directory successivo
MoveFile		Sposta un file da una directory in un'altra
SetCurrentDirectory	chdir	Cambia la directory di lavoro corrente

Figura 6.41 Principali funzioni API di Win32 per la gestione di directory. La seconda colonna indica la funzione di UNIX più simile, se ne esiste una.

Windows 7 è dotato di un meccanismo di sicurezza molto più elaborato di quello di molti sistemi UNIX. Benché esistano centinaia di funzioni API relative alla sicurezza, cerchiamo di fornire una breve descrizione dell'idea generale. Quando un utente effettua il login, il sistema operativo affida al suo processo iniziale un gettone di accesso (*access token*) che contiene il SID (*Security ID*) dell'utente, una lista dei gruppi di sicurezza cui egli appartiene, i privilegi speciali di cui dispone, il livello di integrità del processo, e poche altre informazioni. L'idea del gettone di accesso è di concentrare tutte le informa-

zioni di sicurezza in un solo posto facile da reperire. Tutti i processi creati a partire da quello iniziale ereditano lo stesso gettone di accesso.

Uno dei parametri d'ingresso che può essere passato all'atto della creazione di un oggetto è il suo descrittore di sicurezza, contenente una lista di controllo degli accessi, ACL (*Access Control List*). Ogni elemento della lista consente oppure vieta un certo insieme di operazioni sull'oggetto da parte di un certo SID o di un determinato gruppo. Per esempio un file potrebbe avere un descrittore di sicurezza secondo cui Marta non ha accesso al file, Marco può leggerlo soltanto, Linda può leggerlo o scrivervi e tutti i membri del gruppo XYZ possono soltanto leggere la lunghezza del file, nient'altro. Si può anche impostare per default il divieto di accesso a tutti quelli che non sono esplicitamente elencati.

Quando un processo cerca di eseguire un'operazione su un oggetto tramite un handle, il gestore della sicurezza prende il token di accesso del processo e controlla prima di tutto il livello di integrità nel token. Un processo non può mai ottenere un handle con permessi di scrittura per oggetti con un livello di integrità più alto. I livelli di integrità sono usati principalmente per restringere il campo d'azione del codice caricato da un browser Web nelle modifiche al sistema. Dopo il controllo del livello di integrità, il gestore della sicurezza scorre ordinatamente la lista delle voci di ACL. Non appena incontra un elemento corrispondente al SID o a gruppo del chiamante, il tipo di accesso ivi consentito è considerato come definitivo. Per questo motivo si usa elencare nella ACL prima gli elementi che negano l'accesso, poi quelli che lo consentono, così che un utente cui è stato negato espressamente l'accesso non riesca a rientrare dalla porta di servizio perché appartenente a un gruppo autorizzato. Il descrittore di sicurezza contiene anche informazioni per la certificazione degli accessi all'oggetto.

Diamo ora uno sguardo all'implementazione di file e directory in Windows 7. Ogni disco è suddiviso in volumi statici indipendenti che hanno la stessa funzione delle partitioni UNIX. Ogni volume contiene una bit map, file e directory, più altre strutture dati per la gestione delle informazioni. Ciascun volume è organizzato come una sequenza lineare di cluster ("raggruppamenti") di dimensione prefissata (compresa tra i 512 byte e i 64 KB) che dipende dalla dimensione del volume stesso. I riferimenti a un cluster avvengono tramite il suo offset dall'inizio del volume, per il quale si usa un numero di 64 bit.

La struttura dati principale di un volume è la MFT (*Master File Table*, "tabella principale di allocazione dei file") che contiene un elemento per ogni file o directory del volume, analogo all'i-node di UNIX. La stessa MFT è un file e perciò può essere posizionata ovunque nel volume. Questa proprietà è utile in caso di blocchi di disco difettosi all'inizio del volume, dove solitamente si trova la struttura MTF. I sistemi UNIX memorizzano di solito alcune informazioni chiave all'inizio di ogni volume e nel caso (alquanto improbabile) in cui uno di questi blocchi sia irrimediabilmente danneggiato l'intero volume deve essere riposizionato.

La MFT (illustrata nella Figura 6.42) comincia con un'intestazione contenente le informazioni relative al volume, come il puntatore alla directory di radice, il file di boot, il file dei blocchi corrotti, l'amministrazione della lista delle locazioni libere e così via. Di seguito si trovano gli elementi relativi ai file o alle directory che occupano 1 KB

ciascuno, tranne quando la dimensione del cluster è di 2 KB o più. Ogni elemento contiene i metadati (informazioni amministrative) che riguardano il file o la directory. Sono consentiti diversi formati, uno dei quali è presentato nella Figura 6.42.

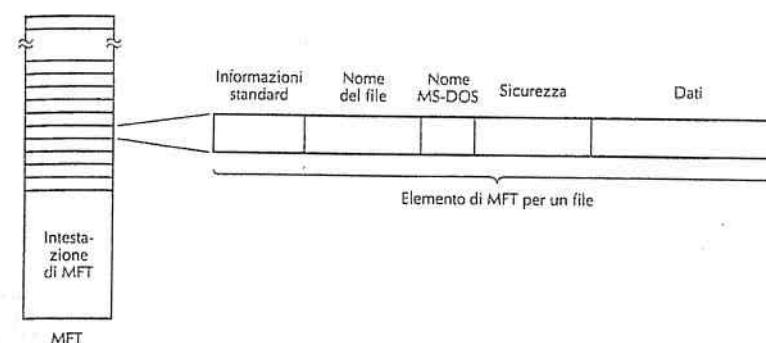


Figura 6.42 Tabella MFT di Windows XP.

Il campo delle informazioni standard contiene alcuni dati come le etichette temporali (*time stamp*) necessarie a POSIX, il computo del numero di collegamenti, i bit di sola lettura o di archivio e così via. Questo campo ha lunghezza fissa ed è sempre presente tra i metadati. Il campo del nome del file contiene un massimo di 255 caratteri Unicode. Affinché i file siano ancora accessibili dai programmi a 16 bit, viene fornito anche un campo per il nome MS-DOS costituito da otto caratteri alfanumerici eventualmente seguiti da un punto e da non più di tre caratteri alfanumerici per l'estensione. Se il vero nome del file rispetta già lo standard MS-DOS 8+3, non viene fornito un secondo nome MS-DOS.

Il campo successivo contiene le informazioni di sicurezza. Fino alla versione di Windows NT 4.0 questo campo conteneva il descrittore di file vero e proprio. A partire da Windows 2000 si è deciso di accentrare tutte le informazioni di sicurezza in un file unico e lasciare nel campo sicurezza un puntatore a una locazione del file.

Il campo dei dati nell'elemento di MFT contiene il file vero e proprio se questo è abbastanza piccolo, risparmiando così un accesso a disco. Questo stratagemma si chiama file immediato (Mullender e Tanenbaum, 1984). Per quanto riguarda i file più grandi, il campo dati contiene i puntatori ai cluster contenenti i dati oppure, più comunemente, a intere sequenze di cluster consecutivi; così con un solo identificativo di cluster e un parametro di lunghezza è possibile specificare una quantità di dati arbitraria. Se un elemento di MFT dovesse rivelarsi insufficiente a contenere tutte le informazioni richieste da un file è sempre possibile accorparlo a uno o più elementi successivi della tabella.

La dimensione massima dei file è di  $2^{64}$  byte. Per capire quanto grande sia  $2^{64}$  byte (cioè  $2^{67}$  bit), si immagini una sequenza di zeri e di uno che occupino ciascuno un millimetro: una sequenza di  $2^{67}$  bit suffatti ( $2^{64}$  byte) sarebbe lunga 15 anni luce, abbastanza

per oltrepassare i confini del sistema solare, raggiungere Alpha Centauri e tornare indietro.

Il file system NTFS presenta molte altre proprietà interessanti, tra cui il supporto a flussi multipli di dati per ogni file, la crittografia, la compressione dei dati e la tolleranza agli errori ottenuta per mezzo delle transazioni atomiche. Per informazioni ulteriori riguardo NTFS si consulti (Russinovich e Solomon 2005).

#### 6.5.4 Esempi di gestione dei processi

UNIX e Windows 7 prevedono entrambi la possibilità di suddividere un compito (*job*) tra più processi che possono girare in modo (pseudo)parallelo e comunicare con lo schema produttore-consutatore già illustrato. In questo paragrafo analizziamo la gestione dei processi dei due sistemi. Vedremo anche che entrambi supportano il parallelismo all'interno di un singolo processo mediante i thread.

##### Gestione dei processi in UNIX

Un processo UNIX può in ogni istante usare la chiamata di sistema `fork` per creare un sottoprocesso che è una sua esatta replica. Il processo originale è detto genitore e quello nuovo si chiama processo figlio. Appena dopo la `fork`, i due processi sono identici e condividono addirittura gli stessi descrittori di file. Di lì in poi ciascuno prosegue per la propria strada e svolge il proprio compito indipendentemente dall'altro.

Spesso il processo figlio manipola i descrittori di file ed esegue una chiamata di sistema `exec` che rimpiazza il suo programma e i suoi dati con il programma e i dati contenuti in un certo file eseguibile specificato come parametro della chiamata `exec`. Per esempio, quando un utente immette il comando `xyz` in un terminale, l'interprete a linea di comando (la shell) esegue una `fork` per creare un processo figlio che svolga il programma `xyz` a seguito di una `exec`.

I due processi girano in parallelo (con o senza `exec`) a meno che il genitore voglia attendere la terminazione del figlio prima di continuare. In questa evenienza, il genitore esegue una chiamata di sistema `wait` o `waitpid` che causa la sua sospensione fino al termine dell'esecuzione del figlio. Solo allora il genitore riprende l'esecuzione.

I processi possono eseguire quante `fork` vogliono, dando vita così a un albero di processi. La Figura 6.43 mostra il processo A che ha eseguito due `fork` e ha così generato i due figli B e C. Anche B ha eseguito due volte una `fork`, mentre C l'ha eseguita una volta sola, per un totale di sei processi.

In UNIX i processi possono comunicare tra loro attraverso una struttura chiamata pipe ("condutture"). Una pipe è una specie di buffer in cui un processo può scrivere un flusso di dati disponibili al prelievo da parte di un altro processo. I byte sono prelevati da una pipe sempre nell'ordine in cui sono stati inseriti, non è mai consentito l'accesso diretto a una certa posizione. Le pipe non preservano le estremità dei messaggi, perciò se un processo effettua quattro scritture da 128 byte e un altro processo effettua una lettura da 512 byte, il secondo riceverà tutti i dati in una volta sola, senza alcuna traccia del fatto che sono stati scritti in più operazioni.

System V e Solaris mettono a disposizione un'altra modalità di comunicazione tra processi che usa le code di messaggi (*message queue*). Un processo può usare la chia-

mata `msgget` per creare una nuova coda di messaggi o aprirne una esistente, può invocare `msgsnd` o `msgrcv` per spedire o ricevere un messaggio su una coda. I messaggi che passano in una coda differiscono per molti versi da quelli che attraversano le pipe. Innanzitutto vengono preservati gli estremi dei messaggi, mentre una pipe è solo un flusso di dati. Poi i messaggi possono avere priorità, così quelli urgenti possono sopravanzare quelli meno importanti. Infine, i messaggi hanno un tipo, che può essere specificato tramite la `msgrcv`.

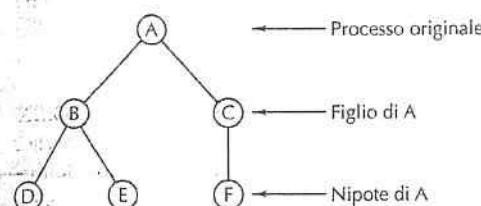


Figura 6.43 Albero di processi in UNIX.

Un altro meccanismo per la comunicazione tra processi è la condivisione di una o più regioni dei loro spazi degli indirizzi. UNIX gestisce la condivisione di memoria attraverso la corrispondenza simultanea tra alcune pagine e gli spazi degli indirizzi di tutti i processi. L'effetto di questa soluzione è che una scrittura operata in una regione condivisa è immediatamente visibile agli altri processi. Questo meccanismo garantisce una comunicazione con larghezza di banda molto elevata. Le chiamate di sistema coinvolte nella condivisione di memoria si chiamano `shmat` e `shmem`.

Un'altra caratteristica di System V e di Linux è il supporto dei semafori, il cui funzionamento ricalca in buona sostanza quanto visto nell'esempio del produttore-consutatore.

Un'ulteriore funzionalità offerta dai tutti i sistemi UNIX conformi a POSIX è la gestione di thread multipli all'interno di un singolo processo. I thread sono processi leggeri che condividono lo stesso spazio degli indirizzi e tutto ciò che gli è associato: i descrittori di file, le variabili d'ambiente e i timer esterni. Tuttavia, ogni thread ha il proprio program counter, i propri registri e il proprio stack. Quando un thread si blocca (per esempio perché si trova in attesa del completamento di un'operazione di I/O o di un altro evento) l'esecuzione può passare agli altri thread dello stesso processo. Due thread produttore-consutatore che operano all'interno dello stesso processo sono simili, ma non identici, a due processi costituiti ciascuno da un solo thread e che condividono un segmento di memoria come buffer. Le differenze stanno nel fatto che in quest'ultimo caso ogni processo ha i propri descrittori di file, variabili d'ambiente e così via, mentre nel primo caso questi enti sono condivisi. Abbiamo incontrato i thread Java nell'esempio del produttore-consutatore. Spesso il sistema esecutivo di Java si avvale dei thread del sistema operativo per i propri thread, ma può anche scegliere di non farlo.

I server web sono un buon esempio per illustrare l'utilità dei thread. Questi server possono contenere una cache delle pagine web usate più comunemente, così se arriva la richiesta di una pagina presente nella cache, questa può essere inviata immediatamente. In caso contrario, la pagina deve essere caricata dal disco e ciò comporta tempi maggiori (20 ms circa), durante i quali il processo resta bloccato e non può servire nuove richieste in ingresso, anche se si trattasse di richieste di pagine nella cache.

La soluzione consiste nel generare più thread all'interno del processo server, dove tutti condividono la stessa cache delle pagine web. Quando si blocca un thread, gli altri possono continuare a gestire le richieste in ingresso. In alternativa, sarebbe possibile moltiplicare il numero dei processi invece di quello dei thread, ma ciò richiederebbe probabilmente una replica della cache per ogni processo, con conseguente spreco di memoria.

Lo standard di UNIX per i thread si chiama `pthread` ed è definito da POSIX (P1003.1C). Sono previste chiamate per la gestione e per la sincronizzazione di thread, ma non è specificato se questi devono risiedere nello spazio del kernel o in quello dell'utente. Le chiamate più comuni per la gestione dei thread sono elencate nella Figura 6.44.

La prima chiamata, `pthread_create`, crea un nuovo thread, e dopo il suo completamento lo spazio degli indirizzi del chiamante contiene un thread in più in esecuzione. Quando un thread ha completato il proprio compito chiama `pthread_exit`, che ne provoca la terminazione. Un thread può attendere la terminazione di un altro thread tramite la chiamata `pthread_join`: se il thread che si vuole attendere è già terminato la chiamata si interrompe immediatamente, altrimenti il chiamante resta bloccato in attesa.

Chiamata di thread	Significato
<code>pthread_create</code>	Crea un nuovo thread nello spazio degli indirizzi del chiamante
<code>pthread_exit</code>	Provoca la terminazione del thread chiamante
<code>pthread_join</code>	Aspetta la terminazione di un thread
<code>pthread_mutex_init</code>	Crea un nuovo mutex
<code>pthread_mutex_destroy</code>	Distrugge un mutex
<code>pthread_mutex_lock</code>	Riserva un mutex
<code>pthread_mutex_unlock</code>	Rilascia un mutex
<code>pthread_cond_init</code>	Crea una variabile di condizione
<code>pthread_cond_destroy</code>	Distrugge una variabile di condizione
<code>pthread_cond_wait</code>	Attende su una variabile di condizione
<code>pthread_cond_signal</code>	Libera un thread in attesa su di una variabile di condizione

Figura 6.44 Principali chiamate POSIX per i thread.

I thread si sincronizzano grazie a particolari lock chiamati **mutex** e usati generalmente per sorvegliare l'uso di buffer condivisi da due thread. Per avere la certezza di essere l'unico ad accedere a una risorsa condivisa, un thread deve assicurarsi il mutex prima di modificare la risorsa e deve rimuoverlo non appena ha concluso il proprio accesso. Se

tutti i thread seguissero questo protocollo, evitare le corse critiche sarebbe possibile. I mutex somigliano a semafori binari, cioè a semafori che possono assumere solo i valori 0 e 1. Il nome "mutex" deriva dal fatto che sono usati per assicurare la mutua esclusione (*mutual exclusion*) da certe risorse.

Le chiamate `pthread_mutex_init` e `pthread_mutex_destroy` servono rispettivamente a creare e a distruggere un mutex. Un mutex può trovarsi in uno di due stati: riservato o non riservato. Per riservare un mutex non riservato, un thread usa `pthread_mutex_lock` che ha effetto immediato; viceversa, se il mutex è già riservato la chiamata diventa bloccante. Spetta al thread che aveva riservato il mutex precedentemente chiamare `pthread_mutex_unlock` per rilasciarlo non appena abbia completato le proprie operazioni sulla risorsa condivisa.

I mutex servono a salvaguardare gli accessi a breve termine alle variabili condivise, mentre per la sincronizzazione a lungo termine (come nell'attesa per un'unità a nastro) si usano le **variabili di condizione**. La loro creazione e distruzione avviene tramite le chiamate `pthread_cond_init` e `pthread_cond_destroy`.

L'uso di una variabile di condizione coinvolge due thread, uno in attesa sulla variabile e un altro che effettua la segnalazione per sopraggiunta disponibilità. Per esempio, se un thread realizza che l'unità a nastro che gli serve è occupata, chiama `pthread_cond_wait` su una variabile di condizione associata all'unità e precedentemente condivisa tra tutti i thread. Quando il thread che ha in uso l'unità completa la propria attività su di essa (non importa quanto tempo dopo), chiama `pthread_cond_signal` per liberare esattamente un thread in attesa su quella variabile di condizione (se ce n'è almeno uno). Se non c'è alcun thread in attesa il segnale va perso; le variabili di condizione non sono contatori come i semafori. Esistono poche altre operazioni definite sui thread, sui mutex e sulle variabili di condizione che però non tratteremo.

### Gestione dei processi in Windows 7

Windows 7 supporta esistenza, comunicazione e sincronizzazione di più processi. Ogni processo contiene almeno un thread. Processi e thread (che possono essere schedulati dal processo stesso) costituiscono un insieme di strumenti molto generale per la gestione del parallelismo sia su sistemi uniprocesso (macchine con CPU singola) sia multiprocesso (macchine con più CPU).

I processi sono creati mediante la funzione API `CreateProcess` che accetta 10 parametri, ciascuno dotato di molte opzioni. Si tratta evidentemente di un progetto molto più complesso dello schema di UNIX, la cui `fork` non ha parametri, mentre `exec` ne ha solo tre: il puntatore al nome del file da eseguire, l'array dei parametri individuati nella linea di comando e le stringhe d'ambiente. Con buona approssimazione, i 10 parametri di `CreateProcess` servono a contenere:

1. un puntatore al nome del file eseguibile;
2. la stessa linea di comando (prima dell'analisi di parsing);
3. un puntatore al descrittore di sicurezza del processo;
4. un puntatore al descrittore di sicurezza del thread iniziale;
5. un bit che determina se il nuovo processo eredita gli handle del processo creatore;

6. vari flag (per esempio la modalità d'errore, la priorità, il debugging, le console);
7. un puntatore alle stringhe d'ambiente;
8. un puntatore al nome della directory di lavoro del nuovo processo;
9. un puntatore alla struttura che descrive la finestra iniziale a schermo;
10. un puntatore alla struttura che serve per restituire 18 valori al chiamante.

Windows 7 non impone alcun rapporto di parentela o gerarchia tra processi: tutti i processi creati sono uguali. D'altra parte, poiché uno dei 18 parametri restituiti al chiamante è l'handle del nuovo processo (che gli consente un certo controllo sul processo neonato) risulta definita implicitamente una gerarchia in base alla relazione "il processo *x* possiede l'handle del processo *y*". Nonostante sia vietato il passaggio diretto di handle tra processi, c'è comunque un modo con cui un processo può adattare un handle per passarlo a un altro e così la gerarchia definita implicitamente può avere vita breve.

Ogni processo di Windows 7 all'atto della creazione contiene un solo thread, ma è possibile aggiungervene successivamente di nuovi. La creazione dei thread è più semplice di quella dei processi; `CreateThread` ha soli 6 parametri invece di 10: descrittore di sicurezza, dimensione dello stack, indirizzo iniziale, un parametro definito dall'utente, stato iniziale (pronto o bloccato) e ID. La creazione dei thread è svolta dal kernel che è dunque al corrente della loro esistenza (i thread non sono implementati interamente nello spazio dell'utente come avviene in altri sistemi).

Durante la propria attività di scheduling, il kernel guarda soltanto i thread eseguibili, senza prestare attenzione al processo a cui appartengono. Ciò significa che il kernel è sempre informato sullo stato (pronto o bloccato) di ogni thread. I thread sono oggetti del kernel e in quanto tali hanno un handle e un descrittore di sicurezza. Poiché è possibile il passaggio di handle tra processi, è altresì possibile per un processo controllare (o creare) i thread di un altro processo. È questa una caratteristica utile, per esempio, alla scrittura di debugger.

I processi possono comunicare in un numero svariato di modi: attraverso pipe, pipe con nome, *mailslot*, socket, chiamate di procedura a distanza e file condivisi. Alla creazione di una pipe si può scegliere tra due modalità di funzionamento: a byte e a messaggi. Le pipe a byte funzionano come quelle di UNIX. Le pipe a messaggi sono abbastanza simili, ma preservano le estremità dei messaggi: così quattro scritture di 128 byte ciascuna verranno ricevute come quattro messaggi di 128 byte e non come un solo messaggio di 512 byte, come accadrebbe con una pipe a byte. Esistono anche le pipe con nome, che mettono a disposizione entrambe le modalità. A differenza delle pipe ordinarie, le pipe con nome possono essere usate anche via rete.

Le socket sono come le pipe, se non che di norma collegano processi che risiedono su macchine diverse, ma possono essere usate anche per la comunicazione tra processi della stessa macchina. In genere non assicurano vantaggi particolari rispetto alla connessione di due macchine tramite una pipe, con o senza nome.

Le chiamate di procedura a distanza (*remote procedure call*) consentono al processo *A* di richiedere al processo *B* di chiamare per proprio conto una terza procedura nello spazio degli indirizzi di *B*, e di farsi restituire il risultato. Esistono diverse limitazioni sui

parametri; per esempio non ha alcun senso passare un puntatore a un altro processo. Piuttosto, gli oggetti puntati devono essere impacchettati e inviati al processo destinazione.

Infine i processi possono condividere la memoria attraverso la sovrapposizione di uno stesso file. Tutte le scritture effettuate da un processo compaiono anche nello spazio degli indirizzi dell'altro processo. Con questo meccanismo l'implementazione del buffer condiviso nell'esempio del produttore-consamatore è molto semplice.

Così come Windows 7 fornisce molti meccanismi di comunicazione tra processi, mette anche a disposizione un gran numero di tecniche di sincronizzazione, tra cui i semafori, i mutex, le sezioni critiche e gli eventi. Sono tutti meccanismi che operano a livello dei thread, non dei processi, perciò se un thread si blocca su un semaforo ciò non ha alcun effetto sugli (eventuali) altri thread dello stesso processo, che possono così continuare la propria esecuzione.

La funzione API `CreateSemaphore` crea un nuovo semaforo, con la possibilità di assegnargli un certo valore e un valore massimo. I semafori sono oggetti del kernel e hanno quindi un handle e un descrittore di sicurezza. È possibile duplicare l'handle di un semaforo attraverso la chiamata `DuplicateHandle` e passarlo a un altro processo affinché diversi processi si sincronizzino sullo stesso semaforo. Ai semafori può anche essere assegnato un nome nel momento della loro creazione, così che altri processi li possano aprire usandone il nome. Sono presenti le chiamate per le operazioni di up e down, anche se hanno nomi un po' peculiari: `ReleaseSemaphore` e `WaitForSingleObject` rispettivamente. Inoltre è possibile passare a `WaitForSingleObject` un timeout dopo il quale il thread chiamante viene liberato in ogni caso, anche se il semaforo resta a 0 (e sebbene l'uso del timer reintroduca le corse critiche).

Anche i mutex sono oggetti del kernel usati per la sincronizzazione, ma sono più semplici dei semafori perché non sono associati a contatori. Sono fondamentalmente dei lock cui corrispondono le funzioni API `WaitForSingleObject` (per riservarli) e `ReleaseMutex` (per rilasciarli). Al pari degli handle dei semafori, anche gli handle dei mutex possono essere duplicati e passati tra processi di modo che processi diversi accedano agli stessi mutex.

Il terzo meccanismo di sincronizzazione si basa sulle sezioni critiche, simili ai mutex. Diversamente da questi però, le sezioni critiche non sono oggetti del kernel, ma sono locali allo spazio degli indirizzi del thread che le ha generate. Di conseguenza non hanno handle né descrittori di sicurezza e non possono essere passate da un processo all'altro. Vengono riservate e rilasciate per mezzo delle chiamate `EnterCriticalSection` e `LeaveCriticalSection`. Sono funzioni molto più veloci delle analoghe definite sui mutex, perché vengono svolte interamente nello spazio dell'utente. Windows 7 offre anche variabili di condizione, lock leggeri di lettura e scrittura, operazioni libere da lock e altri meccanismi di sincronizzazione che operano soltanto tra thread dello stesso processo.

L'ultima forma di sincronizzazione usata ancora oggetti del kernel, che prendono il nome di eventi. Un thread può attendere che si verifichi un evento richiamando `WaitForSingleObject`, può liberare un thread in attesa su un evento per mezzo di `SetEvent`, o addirittura invocare `PulseEvent` per risvegliare tutti i thread in attesa su quell'evento. Anche gli eventi presentano molte varianti e consentono di scegliere tra diverse

opzioni. Windows utilizza gli eventi per la sincronizzazione quando un I/O asincrono viene completato, e per altri scopi.

È possibile attribuire un nome agli eventi, ai mutex e ai semafori e memorizzarli nel file system, proprio come avviene per le pipe con nome. Due o più processi possono poi sincronizzarsi tramite la semplice apertura dello stesso evento, mutex o semaforo, senza il bisogno che un processo crei l'oggetto e ne duplichhi l'handle per passarlo a tutti gli altri.

## 6.6 Riepilogo

Si può guardare al sistema operativo come al garante di alcune funzionalità architettoniche non presenti nel livello ISA. Prime fra tutte ci sono la memoria virtuale, le istruzioni virtuali di I/O e il supporto del calcolo parallelo.

La memoria virtuale è una funzionalità architettonica che ha lo scopo di fornire ai programmi spazi d'indirizzi più grandi della memoria fisica del sistema, insieme a un meccanismo flessibile e corretto per la protezione e la condivisione della memoria. Può essere implementata con la paginazione pura, con la segmentazione pura o con una combinazione delle due. Nel primo caso lo spazio degli indirizzi è suddiviso in pagine virtuali di uguale dimensione, alcune delle quali sono contenute in blocchi fisici di memoria. Un accesso a una pagina in memoria è tradotto dalla MMU nell'indirizzo fisico corretto, altrimenti il tentativo di accesso causa un errore di pagina. Il Core i7 e la CPU ARM OMAP4430 dispongono entrambi di MMU che gestiscono la memoria virtuale e la paginazione.

L'astrazione di I/O più importante presente a questo livello è il file. Un file consiste in una sequenza di byte o di record logici che può essere letta o scritta senza alcuna nozione del funzionamento dei dischi, delle unità a nastro o di altri dispositivi di I/O. L'accesso ai file può essere sequenziale o diretto (per chiave o per numero di record). Le directory sono utilizzate per raggruppare file. Questi possono essere memorizzati in settori consecutivi del disco o sparpagliati al suo interno. Nel secondo caso, che è il più frequente nei dischi normali, si fa uso di strutture dati per la localizzazione dei blocchi di un file. È possibile mantenere traccia delle locazioni libere del disco tramite una lista o una bit map.

Il calcolo parallelo viene spesso gestito e implementato tramite la simulazione di processi multipli che condividono a tempo la singola CPU. Le interazioni tra processi, se non controllate opportunamente, possono portare a corse critiche. Per scongiurare questo problema si introducono primitive di sincronizzazione, di cui un esempio è dato dai semafori. Grazie ai semafori è possibile risolvere problemi analoghi a quello del produttore-consumatore in maniera semplice ed elegante.

Windows 7 e UNIX sono due esempi di sistemi operativi molto sofisticati. Entrambi supportano paginazione, corrispondenza tra file in memoria e file system gerarchico, ed entrambi intendono i file come semplici sequenze di byte. Infine tutti e due i sistemi operativi forniscono la gestione dei processi e dei thread, nonché diverse modalità per la loro sincronizzazione.

## PROBLEMI

- Perché un sistema operativo interpreta solo alcune delle istruzioni del livello 3, mentre un microprogramma interpreta tutte le istruzioni del livello ISA?
- Una macchina ha uno spazio degli indirizzi virtuali di 32 bit indirizzabile al byte. La dimensione di pagina è di 4 KB. Quante pagine d'indirizzi virtuali esistono?
- È necessario che le pagine abbiano dimensioni pari a potenze di 2? Sarebbe possibile, in teoria, implementare pagine di 4000 byte? Se sì, si tratterebbe di una soluzione pratica?
- Una memoria virtuale ha dimensioni di pagina di 1024 parole, otto pagine virtuali e quattro blocchi fisici di memoria. La tabella delle pagine è la seguente:

Pagina virtuale	Blocco di memoria
0	3
1	1
2	assente dalla memoria
3	assente dalla memoria
4	2
5	assente dalla memoria
6	0
7	assente dalla memoria

- Elencare tutti gli indirizzi virtuali che causerebbero un errore di pagina.
- Quali sono gli indirizzi fisici corrispondenti a 0, 3728, 1023, 1024, 1025, 7800 e 4096?
- Un computer ha 16 pagine d'indirizzi virtuali e solo quattro blocchi di memoria. La memoria è inizialmente vuota. Se un programma accede alle pagine virtuali secondo l'ordine 0, 7, 2, 7, 5, 8, 9, 2, 4
  - Quali riferimenti causerebbero un errore di pagina se si usa LRU?
  - Quali riferimenti causerebbero un errore di pagina se si usa FIFO?
- Nel Paragrafo 6.1.4 è stato introdotto un algoritmo per l'implementazione della sostituzione delle pagine secondo la strategia FIFO. Se ne progetti una versione più efficiente. Suggerimento: è possibile limitarsi ad aggiornare il contatore della sola pagina appena caricata, lasciando invariato quello delle altre pagine.
- Nei sistemi paginati trattati nel capitolo, il gestore degli errori di pagina faceva parte del livello ISA e non era contenuto perciò nello spazio degli indirizzi di alcun programma del livello OSM. In realtà lo stesso gestore degli errori di pagina occupa alcune pagine e potrebbe venire esso stesso rimosso in determinate circostanze (per esempio se si adottasse la politica di sostituzione delle pagine FIFO). Che cosa accadrebbe se il gestore degli errori di pagina non si trovasse in memoria quando capita un errore di pagina? Come si potrebbe risolvere il problema?
- Non tutti i computer dispongono di un meccanismo che asserisce automaticamente un bit hardware quando viene effettuata una scrittura in una pagina. Nonostante è utile tener traccia delle pagine modificate al fine di evitare la scrittura su disco di tutte le pagine caricate (quando non servono più). Se in ogni pagina esistono alcuni bit hardware per l'abilitazione separata dei permessi in lettura, in scrittura e in esecuzione, come può il sistema operativo usare questi bit per stabilire se una pagina è "pulita" o se è stata modificata?
- Una memoria segmentata è a paginazione dei segmenti. Ogni indirizzo virtuale ha 2 bit per il numero di segmento; 2 bit per il numero di pagina e 11 bit per l'offset all'interno della pagina. La memoria principale contiene 32 KB, suddivisi in pagine di 2 KB. Per ogni segmento sono possibili solo le configurazioni seguenti: in sola lettura, in lettura/esecuzione, in lettura/scrittura o in lettura/scrittura/esecuzione. La tabella delle pagine e le modalità di protezione sono le seguenti:

Segmento 0		Segmento 1		Segmento 2		Segmento 3	
Sola lettura		Lettura/esecuzione		Lettura/scrrittura/esecuzione		Lettura/scrrittura	
Pagina virtuale	Blocco di memoria	Pagina virtuale	Blocco di memoria		Pagina virtuale	Blocco di memoria	
0	9	0	su disco	Tabella delle pagine non presente in memoria principale	0	14	
1	3	1	0		1	1	
2	su disco	2	15		2	6	
3	12	3	8		3	su disco	

Per ognuno dei seguenti accessi alla memoria virtuale si indichi l'indirizzo fisico che viene calcolato. Se si verifica un errore di pagina, specificarne il tipo.

Accesso	Segmento	Pagina	Offset nella pagina
1. fetch di dati	0	1	1
2. fetch di dati	1	1	10
3. fetch di dati	3	3	2047
4. memorizzazione di dati	0	1	4
5. memorizzazione di dati	3	1	2
6. memorizzazione di dati	3	0	14
7. salto a	1	3	100
8. fetch di dati	0	2	50
9. fetch di dati	2	0	5
10. salto a	3	0	60

10. Alcuni calcolatori consentono l'I/O direttamente nello spazio dell'utente. Per esempio, un programma potrebbe avviare un trasferimento dal disco verso un buffer di un processo dell'utente. Questa possibilità può causare problemi se la memoria virtuale è implementata con compattamento? Si argomenti la risposta.
11. I sistemi operativi che permettono la corrispondenza tra file in memoria richiedono che essa avvenga su multipli di pagine (e che rispetti le estremità delle pagine). Per esempio, se le pagine sono di 4 KB, un file può essere mappato a partire dall'indirizzo virtuale 4096, ma non dall'indirizzo 5000. Perché?
12. Quando nel Core i7 viene caricato un registro di segmento, il descrittore corrispondente viene rintracciato e caricato all'interno di una porzione invisibile del registro di segmento. Si dia una giustificazione per questa scelta dei progettisti Intel.
13. Un programma del Core i7 effettua un accesso al segmento locale 10 con offset 8000. Il campo BASE dell'elemento di LDT corrispondente al segmento 10 contiene il numero 10000. Quale elemento della directory delle pagine usa il Pentium 4? Qual è il numero di pagina? Qual è l'offset?
14. Si confrontino alcuni possibili algoritmi per l'estromissione dei segmenti in una memoria segmentata, ma non paginata.
15. Si confrontino la frammentazione interna e quella esterna. Che cosa si può fare per ridurla?
16. I supermercati sono abituati ad affrontare un problema molto simile a quello della sostituzione di pagine nei sistemi con memoria virtuale. I loro scaffali hanno una capienza fissa a fronte di un numero di prodotti sempre crescente. Se viene commercializzato un nuovo prodotto rivoluzionario, diciamo un cibo per cani molto protetico, sarà necessario fargli spazio estromettendo alcuni prodotti dall'inventario. Gli algoritmi disponibili sono i soliti LRU e FIFO. Quale sarebbe preferibile in questo contesto?

17. La cache e la paginazione sono per molti versi simili. In entrambi i casi ci sono due livelli di memoria (la cache e la memoria principale nel primo caso, la memoria e il disco nel secondo). In questo capitolo abbiamo enunciato alcune argomentazioni a favore delle pagine di disco grandi e altre a favore di quelle piccole. Valgono le stesse argomentazioni per la dimensione delle linee di cache?
18. Perché molti file system richiedono l'apertura preventiva di un file con una chiamata di sistema open prima della lettura?
19. Si confrontino il metodo che usa la bit map e quello che usa la lista delle lacune per la gestione dello spazio libero su di un disco con 800 cilindri, ciascuno costituito di 5 tracce di 32 settori. Quante lacune ci vorrebbero perché la lista relativa ecceda la dimensione della bit map? Si ipotizzi che l'unità di allocazione sia il settore e che una lacuna richieda un elemento di tabella di 32 bit.
20. Un terzo schema di allocazione delle lacune, che si aggiunge a best fit e first fit, è worst fit. Questo schema prevede che a un processo sia allocato lo spazio dato dalla più grande lacuna rimanente. Quale vantaggio si può ottenere dall'utilizzo di questo algoritmo?
21. Si descriva uno scopo della chiamata di sistema file open che non sia stato menzionato nel testo.
22. Al fine di prevedere le prestazioni del disco è utile avere un modello dell'allocazione dei dati. Si supponga che il disco sia uno spazio lineare degli indirizzi con svariati settori, costituito da una sequenza di blocchi di dati, quindi da una lacuna, da un'altra sequenza di blocchi di dati e così via. Se alcune misure empiriche dimostrano che le distribuzioni di probabilità dei dati e delle lacune sono uguali e attribuiscono probabilità  $2^{-i}$  alle sequenze di lunghezza  $i$ , qual è il valore atteso delle lacune in un disco?
23. In un certo calcolatore, un programma può creare tanti file quanti sono necessari, e tutti i suoi file possono crescere dinamicamente durante l'esecuzione, senza dare alcuna indicazione preventiva al sistema operativo circa la loro dimensione finale. Su un sistema siffatto i file verranno memorizzati in settori consecutivi? Si motivi la risposta.
24. Le statistiche hanno dimostrato che più della metà dei file ha dimensioni di pochi KB e che la stragrande maggioranza dei file è al di sotto di un certo numero di KB (attorno agli 8 KB). D'altro canto, il 10% dei file più voluminosi è responsabile del 95% dell'intera occupazione di spazio su disco. A partire da questi dati, che conclusione si può trarre circa la dimensione dei blocchi del disco?
25. Si consideri una possibile implementazione delle istruzioni sui semafori da parte di un sistema operativo: quando la CPU sta per effettuare una up o down su un semaforo (una variabile intera in memoria), il sistema imposta o maschera i bit di priorità della CPU in modo tale da disabilitare tutti gli interrupt. Quindi effettua il fetch del semaforo, lo modifica e salta in base al suo valore. Solo a questo punto riabilita gli interrupt. Il metodo funziona se:
  - c'è una sola CPU che effettua la commutazione tra processi ogni 100 ms
  - due CPU condividono una memoria comune in cui si trova il semaforo.
26. Nella descrizione dei semafori del paragrafo 6.3.3 viene affermato: "In genere i processi dormienti sono posti in una coda al fine di poterne tenere traccia." Quale vantaggio si ottiene utilizzando una coda per i processi in attesa al posto di svegliare un processo dormiente scelto a caso al momento dell'esecuzione di una up?
27. I produttori del Sistema Operativo Indistruttibile hanno ricevuto delle lamentele da parte di clienti a proposito dell'ultima distribuzione del sistema, in cui sono state introdotte le operazioni sui semafori. Secondo i clienti è immorale che un processo si blocchi (lo definiscono "poltrire"). Dato che l'azienda crede nel motto secondo cui il cliente ha sempre ragione, si è pensato di introdurre una terza operazione a supporto di up e down: l'operazione peek ("sbirciare"), che si limita a esaminare il valore di un semaforo senza modificarlo e senza sospendere l'esecuzione. In tal modo, se un programma crede sia immorale fermare la propria esecuzione, può cominciare con l'ispezionare il semaforo e stabilire se è sicuro effettuare una down. Questa soluzione funziona finché il semaforo è usato da tre o più processi? E se è usato da due processi?

28. Redigere una tabella che indichi, in funzione dei tempi sotto indicati (da 0 a 1000 ms), quale dei processi P1, P2 e P3 sia in esecuzione e quale sia bloccato. Tutti e tre i processi effettuano istruzioni up e down sullo stesso semaforo. Quando due processi sono bloccati viene eseguita una up, viene risvegliato il processo con nominativo più basso, ovvero P1 ha precedenza su P2 e P3 e così via. All'inizio tutti e tre i processi sono in esecuzione e il semaforo vale 1.
- Al tempo  $t = 100$  P1 effettua una down  
 Al tempo  $t = 200$  P1 effettua una down  
 Al tempo  $t = 300$  P2 effettua una up  
 Al tempo  $t = 400$  P3 effettua una down  
 Al tempo  $t = 500$  P1 effettua una down  
 Al tempo  $t = 600$  P2 effettua una up  
 Al tempo  $t = 700$  P2 effettua una down  
 Al tempo  $t = 800$  P1 effettua una up  
 Al tempo  $t = 900$  P1 effettua una up
29. Nei sistemi di prenotazioni aeree è necessario assicurare che durante l'accesso a un file da parte di un processo, nessun altro possa accedervi. In caso contrario due processi diversi, creati per conto di due compagnie aeree diverse, potrebbero prenotare inadvertitamente lo stesso posto dello stesso volo. Si progettino un metodo di sincronizzazione che usi i semafori per garantire l'accesso esclusivo dei processi ai file (ipotizzando che i processi obbediscano alle regole proposte).
30. Gli architetti degli elaboratori spesso forniscono un'istruzione TSL (Test and Set Lock, "esamina e riserva") per rendere possibile l'implementazione dei semafori sui computer con più CPU che condividono la stessa memoria. TSL X esamina la locazione X: se contiene il valore 0, viene posta a 1 all'interno di un solo, indivisibile ciclo di memoria, e l'istruzione successiva viene saltata. Se invece non vale 0, la TSL si comporta come una NOP. Grazie a TSL è possibile scrivere procedure lock e unlock con le seguenti proprietà: lock(x) verifica se x è riservata, in caso contrario la riserva e restituisce il controllo. Se x è riservata invece aspetta finché non venga rilasciata, dopo di che la riserva e restituisce il controllo. unlock libera una variabile riservata. Se tutti i processi riservano la tabella dei semafori prima di usarla, solo un processo per volta può manipolare le variabili e i puntatori, evitando così le corse critiche. Si scrivano le procedure lock e unlock in linguaggio assemblativo (dopo aver fatto le ipotesi ritenute ragionevoli).
31. Si mostrino i valori di in e out per un buffer circolare lungo 65 parole dopo ciascuna delle seguenti operazioni. Entrambi partono da 0.
- Inserimento di 22 parole.
  - Estrazione di 9 parole.
  - Inserimento di 40 parole.
  - Estrazione di 17 parole.
  - Inserimento di 12 parole.
  - Estrazione di 45 parole.
  - Inserimento di 8 parole.
  - Estrazione di 11 parole.
32. Se una versione di UNIX usa blocchi del disco di 2 KB e memorizza in ogni blocco indiretto (singolo, doppio o triplo) 512 indirizzi del disco, qual è la dimensione massima di file? (si considerino puntatori a file di 64 bit).
33. Si supponga che la chiamata di sistema di UNIX  
`unlink("/usr/ast/bin/gioco3")`  
 sia eseguita nel contesto della Figura 6.37. Si descrivano con cura i cambiamenti risultanti nel directory system.
34. Si immagini di dover implementare UNIX su di un microcomputer con pochissima memoria. Se, nonostante tutti gli sforzi adoperati, il sistema risulta ancora troppo grande, si rende necessario il sacrificio di una chiamata di sistema. La scelta ricade su pipe, usata per la creazione delle pipe che permettono la comunicazione di

- flussi di byte tra processi. È ancora possibile implementare la redirezione di I/O in qualche modo? E le pipeline? Si analizzino il problema e le sue possibili soluzioni.
35. I membri della commissione per l'Uguaglianza dei Descrittori di File stanno organizzando una protesta contro UNIX perché le sue chiamate che restituiscano descrittori di file, restituiscano sempre il numero più piccolo non ancora in uso. Di conseguenza i descrittori di file corrispondenti ai numeri grandi non vengono quasi mai usati. La loro proposta è di far sì che venga restituito il numero più piccolo non ancora usato dal programma, invece che quello più piccolo attualmente inutilizzato. Sostengono che si tratta di una richiesta banale da implementare, che non comporterà la modifica dei programmi esistenti e che renderà il sistema più equo. Che cosa ne pensate?
36. In Windows 7 è possibile generare una lista di controllo degli accessi affinché Roberta non abbia alcun accesso a un certo file e tutti gli altri utenti ne abbiano pieno accesso. Quale può essere l'implementazione di una lista siffatta?
37. Si descrivano due modi diversi di programmare il problema del produttore-consumatore usando i buffer condivisi e i semafori di Windows 7. Si rifletta in particolar modo sull'implementazione del buffer condiviso nelle due soluzioni proposte.
38. Capita frequentemente di usare la simulazione per valutare il comportamento degli algoritmi di sostituzione delle pagine. A tal fine, si scriva un simulatore per una memoria virtuale paginata con 64 pagine di 1 KB. Il simulatore dovrebbe mantenere in memoria una tabella con 64 elementi, uno per pagina, per indicare il numero della pagina fisica corrispondente a ogni pagina virtuale. Il simulatore dovrebbe leggere da un file gli indirizzi virtuali scritti in decimale, uno per riga. Se la pagina corrispondente è in memoria, è sufficiente annotare un successo di pagina (un page hit). Se non è in memoria, il simulatore chiama una procedura per la scelta della pagina da estromettere dalla memoria (cioè per la scelta dell'elemento della tabella da sovrascrivere) e registra un fallimento di pagina (un page miss). Nella realtà non avviene alcuno spostamento di pagina, ma solo la sua simulazione. Si generi un file d'indirizzi casuali e si valutino le prestazioni di LRU e di FIFO. Quindi si generi un file d'indirizzi in cui una frazione costante x degli indirizzi sia ottenuta sommando quattro byte agli indirizzi appena precedenti (per simulare il principio di località). Si eseguano dei test per valori diversi di x e si riferiscano i risultati ottenuti.
39. Il programma della Figura 6.26 ha una corsa critica fatale perché due thread accedono alle stesse variabili condivise in modo non controllato, senza usare semafori o altre tecniche di mutua esclusione. Si esegua il programma e si osservi quanto tempo passa prima del suo bloccaggio. Se non si osserva alcun bloccaggio, si incrementino le dimensioni della finestra di vulnerabilità inserendo qualche istruzione tra la modifica di m.in e m.out e la loro valutazione. Quante istruzioni bisogna aggiungere perché l'esecuzione si ferma, per esempio, una volta all'ora?
40. Si scriva un programma per UNIX o per Windows 7 che prende in input il nome di una directory e che stampa la lista dei file in essa contenuti, uno per riga. Il nome di ogni file va fatto seguire dalla stampa della sua dimensione. Si stampino i nomi dei file nell'ordine in cui si trovano nella directory. Gli elementi inutilizzati di una directory devono essere stampati con la dicitura (inutilizzato).