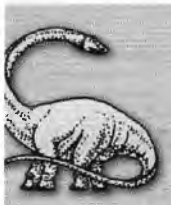
Capitolo 9

# Memoria virtuale



**OBIETTIVI**

- **Descrizione dei vantaggi derivanti dalla memoria virtuale.**
- **Definizione dei concetti di paginazione su richiesta, algoritmi di sostituzione di pagina e allocazione dei frame.**

Nel Capitolo 8 sono state esaminate le strategie di gestione della memoria impiegate nei calcolatori. Hanno tutte lo stesso scopo: tenere contemporaneamente più processi in memoria per permettere la multiprogrammazione; tuttavia esse tendono a richiedere che l'intero processo si trovi in memoria prima di essere eseguito.

La **memoria virtuale** è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria. Il vantaggio principale offerto da questa tecnica è quello di permettere che i programmi siano più grandi della memoria fisica; inoltre la memoria virtuale astrae la memoria centrale in un vettore di memorizzazione molto grande e uniforme, separando la memoria logica, com'è vista dall'utente, da quella fisica. Questa tecnica libera i programmatori da quel che riguarda i limiti della memoria. La memoria virtuale permette inoltre ai processi di condividere facilmente file e spazi d'indirizzi, e fornisce un meccanismo efficiente per la creazione dei processi. La memoria virtuale è però difficile da realizzare e, s'è usata scorrettamente, può ridurre di molto le prestazioni del sistema. In questo capitolo si esamina la memoria virtuale nella forma della paginazione su richiesta e se ne valutano complessità e costi.

## 9.1 Introduzione

Gli algoritmi di gestione della memoria delineati nel Capitolo 8 sono necessari perché, per l'attivazione di un processo, le istruzioni da eseguire si devono trovare all'interno della memoria fisica. Il primo metodo per far fronte a tale requisito consiste nel collocare l'intero spazio d'indirizzi logici del processo relativo in memoria fisica. Il caricamento dinamico può aiutare ad attenuare gli effetti di tale limitazione, ma richiede generalmente particolari precauzioni e un ulteriore impegno dei programmatori.

La condizione che le istruzioni debbano essere nella memoria fisica sembra tanto necessaria quanto ragionevole, ma purtroppo riduce le dimensioni dei programmi a valori strettamente correlati alle dimensioni della memoria fisica. In effetti, da un esame dei pro-

grammi reali risulta che in molti casi non è necessario avere in memoria l'intero programma; si considerino ad esempio le seguenti situazioni.

- Spesso i programmi dispongono di codice per la gestione di condizioni d'errore insolite. Poiché questi errori sono rari, se non inesistenti, anche i relativi segmenti di codice non si eseguono quasi mai.
- Spesso a array, liste e tabelle si assegna più memoria di quanta sia effettivamente necessaria. Un array si può dichiarare di 100 per 100 elementi, anche se raramente contiene più di 10 per 10 elementi. Una tabella dei simboli di un assemblatore può avere spazio per 3000 simboli, anche se un programma medio ne ha meno di 200.
- Alcune opzioni e caratteristiche di un programma sono utilizzabili solo di rado.

Anche nei casi in cui è necessario disporre di tutto il programma è possibile che non serva tutto in una volta.

La possibilità di eseguire un programma che si trova solo parzialmente in memoria può essere vantaggiosa per i seguenti motivi.

- Un programma non è più vincolato alla quantità di memoria fisica disponibile. Gli utenti possono scrivere programmi per uno **spazio degli indirizzi virtuali** molto grande, semplificando così le operazioni di programmazione.
- Poiché ogni utente impiega meno memoria fisica, si possono eseguire molti più programmi contemporaneamente, ottenendo un corrispondente aumento dell'utilizzo e della produttività della CPU senza aumentare il tempo di risposta o di completamento.
- Per caricare (o scaricare) ogni programma utente in memoria sono necessarie meno operazioni di I/O, quindi ogni programma utente è eseguito più rapidamente.

La possibilità di eseguire un programma che non si trovi completamente in memoria apporterebbe quindi vantaggi sia al sistema sia all'utente.

La **memoria virtuale** si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica. Questa separazione permette di offrire ai programmatori una memoria virtuale molto ampia, anche se la memoria fisica disponibile è più piccola, com'è illustrato nella Figura 9.1. La memoria virtuale facilita la programmazione, poiché il programmatore non deve preoccuparsi della quantità di memoria fisica disponibile o di quale codice si debba inserire nelle sezioni sovrapponibili, ma può concentrarsi sul problema da risolvere.

L'espressione **spazio degli indirizzi virtuali** si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico – per esempio, l'indirizzo 0 – e si estende alla memoria contigua, come evidenziato dalla Figura 9.2. Come si ricorderà dal Capitolo 8, è tuttavia possibile organizzare la memoria fisica in frame di pagine; in questo caso i frame delle pagine fisiche assegnati ai processi possono non essere contigui. Spetta all'unità di gestione della memoria (MMU) associare in memoria le pagine logiche alle pagine fisiche.

Si noti come, nella Figura 9.2, allo heap sia lasciato sufficiente spazio per crescere verso l'alto nello spazio di memoria, poiché esso ospita la memoria allocata dinamicamente. In modo analogo, consentiamo alla pila di svilupparsi verso il basso nella memoria, a causa di ripetute chiamate di funzione. Lo spazio vuoto ben visibile (o buco) che separa lo heap dalla pila è parte dello spazio degli indirizzi virtuali, ma richiede pagine fisiche realmente esistenti solo nel caso che lo heap o la pila crescano. Qualora contenga buchi, lo spazio degli indirizzi virtuali si definisce sparso. Un simile spazio degli indirizzi è doppiamente utile, poiché consente di riempire i buchi grazie all'espansione dei segmenti heap o pila, e di col-

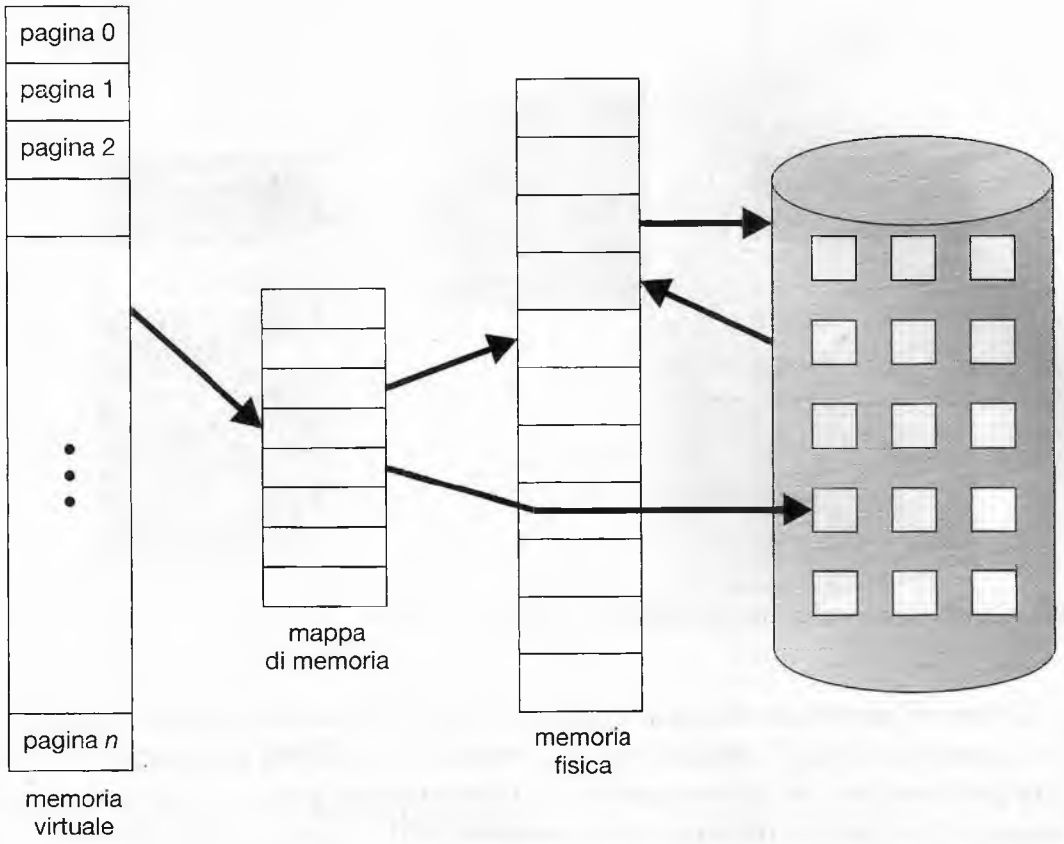



**Figura 9.1** Schema che mostra una memoria virtuale più grande di quella fisica.

legare dinamicamente delle librerie (o altri oggetti condivisi) durante l'esecuzione del programma.

Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre, per due o più processi, il vantaggio di condividere i file e la memoria, mediante la condivisione delle pagine (Paragrafo 8.4.4). Ciò comporta i seguenti vantaggi.

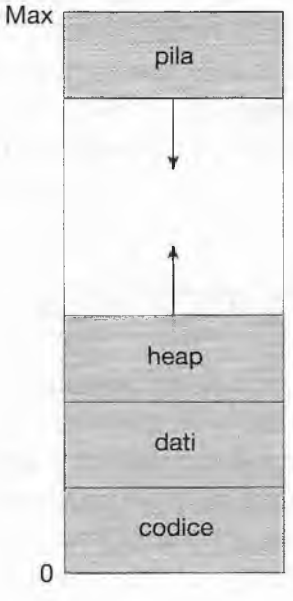



**Figura 9.2** Spazio degli indirizzi virtuali.

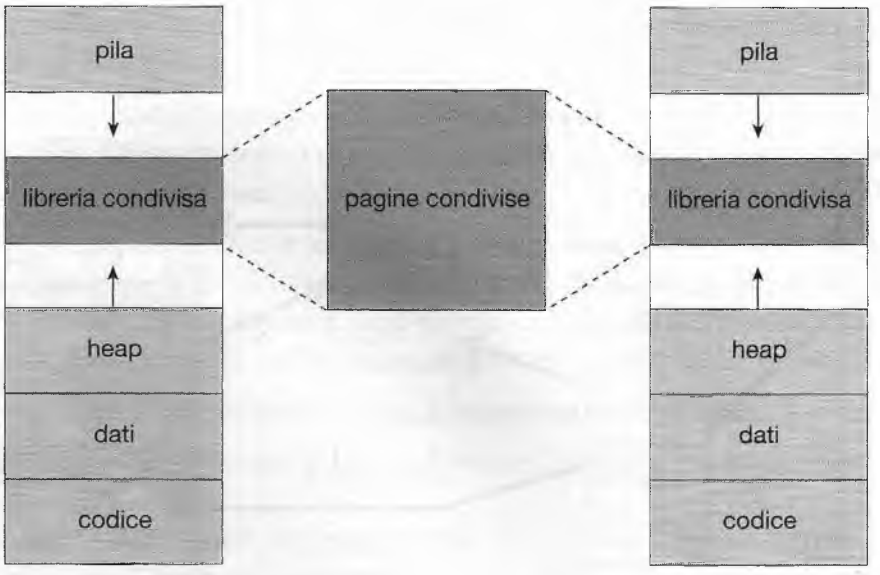



**Figura 9.3** Condivisione delle librerie tramite la memoria virtuale.

- Le librerie di sistema sono condivisibili da diversi processi associando l'oggetto condiviso a uno spazio degli indirizzi virtuali, procedimento detto **mappatura**. Benché ciascun processo veda le librerie condivise come parte del proprio spazio degli indirizzi virtuali, le pagine che ospitano effettivamente le librerie nella memoria fisica sono in condivisione tra tutti i processi (Figura 9.3). In genere le librerie si associano allo spazio di ogni processo a loro collegato, in modalità di sola lettura.
- In maniera analoga, la memoria virtuale rende i processi in grado di condividere la memoria. Come si rammenterà dal Capitolo 3, due o più processi possono comunicare condividendo memoria. La memoria virtuale permette a un processo di creare una regione di memoria condivisibile da un altro processo. I processi che condividono questa regione la considerano parte del proprio spazio degli indirizzi virtuali, malgrado le pagine fisiche siano, in realtà, condivise, come illustrato dalla Figura 9.3.
- La memoria virtuale può consentire, per mezzo della chiamata di sistema `fork()`, che le pagine siano condivise durante la creazione di un processo, così da velocizzare la generazione dei processi.

Approfondiremo questi e altri vantaggi offerti dalla memoria virtuale nel corso di questo capitolo. In primo luogo, però, ci soffermeremo sulla memoria virtuale realizzata attraverso la paginazione su richiesta.

## 9.2 Paginazione su richiesta

Si consideri il caricamento in memoria di un eseguibile residente su disco. Una possibilità è quella di caricare l'intero programma nella memoria fisica al momento dell'esecuzione. Il problema, però, è che all'inizio non è detto che serva avere tutto il programma in memoria: se il programma, per esempio, fornisce all'avvio una lista di opzioni all'utente, è inutile caricare il codice per l'esecuzione di *tutte* le opzioni previste, senza tener conto di quella effettivamente scelta dall'utente. Una strategia alternativa consiste nel caricare le pagine nel momento in cui servono realmente; si tratta di una tecnica, detta **paginazione su richiesta**, comunemente adottata dai sistemi con memoria virtuale. Secondo questo schema, le pagine

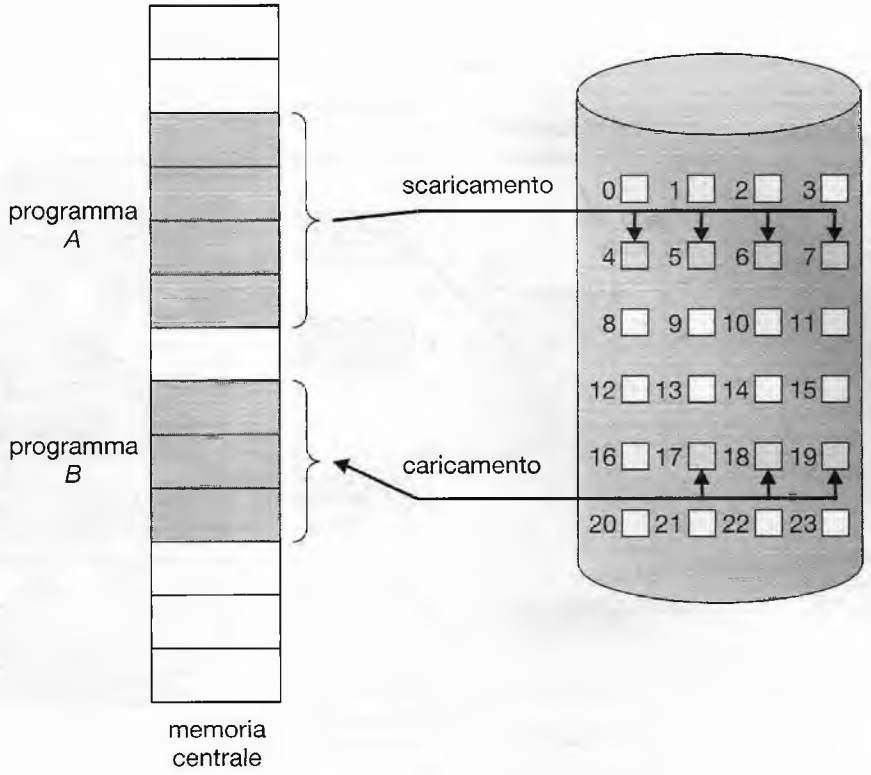



**Figura 9.4** Trasferimento di una memoria paginata nello spazio contiguo di un disco.

sono caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica.

Un sistema di paginazione su richiesta è analogo a un sistema paginato con avvicendamento dei processi in memoria; si veda la Figura 9.4. I processi risiedono in memoria secondaria, generalmente costituita di uno o più dischi. Per eseguire un processo occorre caricarlo in memoria. Tuttavia, anziché caricare in memoria l'intero processo, si può seguire un criterio d'avvicendamento "pigro" (*lazy swapping*): non si carica mai in memoria una pagina che non sia necessaria. Poiché stiamo considerando un processo come una sequenza di pagine, invece che come un unico ampio spazio d'indirizzi contiguo, l'uso del termine *avvicendamento dei processi* non è appropriato: non si manipolano interi processi ma singole pagine di processi. Nell'ambito della paginazione su richiesta, il modulo del sistema operativo che si occupa della sostituzione delle pagine si chiama **paginatore** (*pager*).

## 9.2.1 Concetti fondamentali

Quando un processo sta per essere caricato in memoria, il paginatore ipotizza quali pagine saranno usate, prima che il processo sia nuovamente scaricato dalla memoria. Anziché caricare in memoria tutto il processo, il paginatore trasferisce in memoria solo le pagine che ritiene necessarie. In questo modo è possibile evitare il trasferimento in memoria di pagine che non sono effettivamente usate, riducendo il tempo d'avvicendamento e la quantità di memoria fisica richiesta.

Con tale schema è necessario che l'architettura disponga di un qualche meccanismo che consenta di distinguere le pagine presenti in memoria da quelle nei dischi. A tal fine è utilizzabile lo schema basato sul bit di validità, descritto nel Paragrafo 8.5. In questo caso, però, il bit impostato come "valido" significa che la pagina corrispondente è valida ed è presente in memoria; il bit impostato come "non valido" indica che la pagina non è valida (cioè

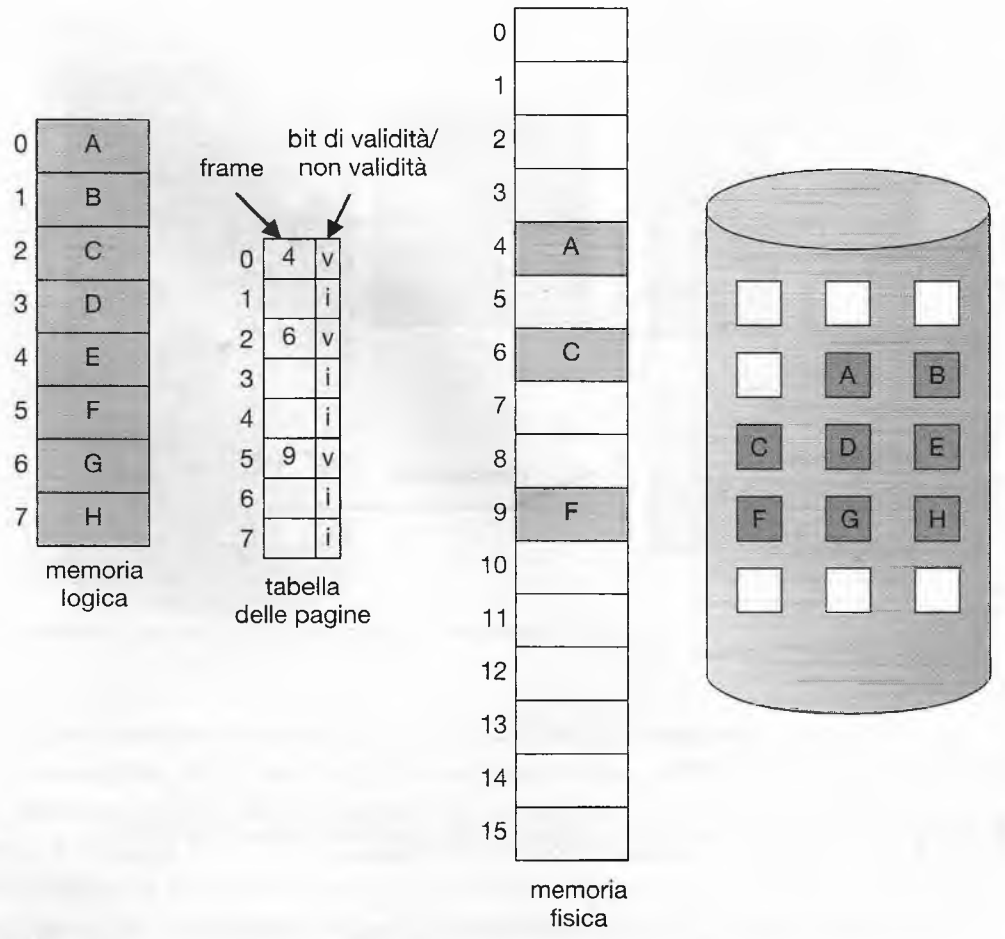



**Figura 9.5** Tabella delle pagine quando alcune pagine non si trovano nella memoria centrale.

non appartiene allo spazio d'indirizzi logici del processo) oppure è valida ma è attualmente nel disco. L'elemento della tabella delle pagine di una pagina caricata in memoria s'imposta come al solito, mentre l'elemento della tabella delle pagine corrispondente a una pagina che attualmente non è in memoria è semplicemente contrassegnato come non valido o contiene l'indirizzo che consente di trovare la pagina nei dischi. Tale situazione è illustrata nella Figura 9.5.

Occorre notare che indicare una pagina come non valida non sortisce alcun effetto se il processo non tenta mai di accedervi. Quindi, se l'ipotesi del paginatore è esatta e si caricano tutte e solo le pagine che servono effettivamente, il processo è eseguito proprio come se fossero state caricate tutte le pagine. Durante l'esecuzione, il processo accede alle pagine **residenti in memoria**, e l'esecuzione procede come di consueto.

Se il processo tenta l'accesso a una pagina che non era stata caricata in memoria, l'accesso a una pagina contrassegnata come non valida causa un'**eccezione di pagina mancante** (*page fault trap*). L'architettura di paginazione, traducendo l'indirizzo attraverso la tabella delle pagine, nota che il bit è non valido e invia un segnale di eccezione al sistema operativo; tale eccezione è dovuta a un "insuccesso" del sistema operativo nella scelta delle pagine da caricare in memoria. La procedura di gestione dell'eccezione di pagina mancante (Figura 9.6) è chiara, e corrisponde ai passi seguenti.

1. Si controlla una tabella interna per questo processo; in genere tale tabella è conservata insieme al blocco di controllo di processo (PCB), allo scopo di stabilire se il riferimento fosse un accesso alla memoria valido o non valido.

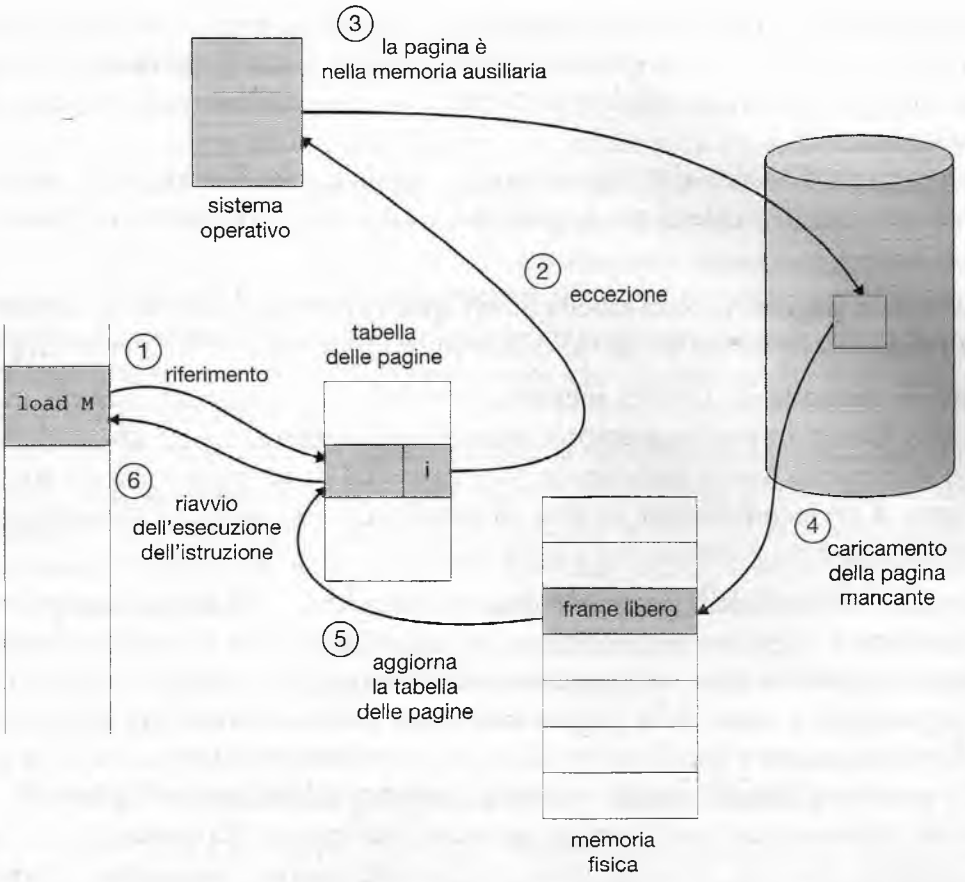



**Figura 9.6** Fasi di gestione di un'eccezione di pagina mancante.

2. Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua l'inserimento.
3. Si individua un frame libero, ad esempio prelevandone uno dalla lista dei frame liberi.
4. Si programma un'operazione sui dischi per trasferire la pagina desiderata nel frame appena assegnato.
5. Quando la lettura dal disco è completata, si modificano la tabella interna, conservata con il processo, e la tabella delle pagine per indicare che la pagina si trova attualmente in memoria.
6. Si riavvia l'istruzione interrotta dal segnale di eccezione. A questo punto il processo può accedere alla pagina come se questa fosse già presente in memoria.

È addirittura possibile avviare l'esecuzione di un processo *senza* pagine in memoria. Quando il sistema operativo carica nel contatore di programma l'indirizzo della prima istruzione del processo, che è in una pagina non residente in memoria, il processo accusa un'assenza di pagina. Una volta portata la pagina in memoria, il processo continua l'esecuzione, subendo assenze di pagine fino a che tutte le pagine necessarie non si trovino effettivamente in memoria; a questo punto si può eseguire il processo senza ulteriori richieste. Lo schema descritto è una **paginazione su richiesta pura**, vale a dire che una pagina non si trasferisce in memoria finché non sia richiesta.

In teoria alcuni programmi possono accedere a più pagine di memoria all'esecuzione di ogni istruzione (una pagina per l'istruzione e molte per i dati), causando più possibili eccezioni di pagine mancanti per ogni istruzione. In un caso simile le prestazioni del sistema

sarebbero inaccettabili. Un'analisi dei processi in esecuzione mostra che questo comportamento è molto improbabile. I programmi tendono ad avere una **località dei riferimenti**, descritta nel Paragrafo 9.6.1, quindi le prestazioni della paginazione su richiesta rientrano in un campo ragionevole.

I meccanismi d'ausilio alla paginazione su richiesta che l'architettura del calcolatore deve offrire sono quelli richiesti per la paginazione e l'avvicendamento dei processi in memoria:

- **tabella delle pagine.** Questa tabella ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione;
- **memoria secondaria.** Questa memoria conserva le pagine non presenti in memoria centrale. Generalmente la memoria secondaria è costituita da un disco ad alta velocità detto dispositivo d'avvicendamento; la sezione del disco usata a questo scopo si chiama **area d'avvicendamento**, o **area di scambio** (*swap space*). L'allocazione dell'area d'avvicendamento è trattata nel Capitolo 12.

Uno dei requisiti cruciali della paginazione su richiesta è la possibilità di rieseguire una qualunque istruzione a seguito di un'eccezione di pagina mancante o assenza di pagina (*page fault*). Avendo salvato lo stato del processo interrotto (registri, codici di condizione, contatore di programma) a causa della pagina mancante, occorrerà riavviare il processo esattamente dallo stesso punto e con lo stesso stato, eccezion fatta per la presenza della pagina desiderata in memoria. Nella maggior parte dei casi questa situazione è piuttosto comune: un'assenza di pagina si può verificare per qualsiasi riferimento alla memoria. Se l'assenza di pagina si presenta durante la fase di prelievo di un'istruzione, l'esecuzione si può riavviare prelevando nuovamente tale istruzione. Se si verifica durante il prelievo di un operando, l'istruzione deve essere di nuovo prelevata e decodificata, quindi si può prelevare l'operando.

Come caso limite si consideri un'istruzione a tre indirizzi, come ad esempio la somma (`ADD`) del contenuto di `A` al contenuto di `B`, con risultato posto in `C`. I passi necessari per eseguire l'istruzione sono i seguenti:

1. prelievo e decodifica dell'istruzione (`ADD`);
2. prelievo del contenuto di `A`;
3. prelievo del contenuto di `B`;
4. addizione del contenuto di `A` al contenuto di `B`;
5. memorizzazione della somma in `C`.

Se l'assenza di pagina avviene al momento della memorizzazione in `C`, poiché `C` si trova in una pagina che non è in memoria, occorre prelevare la pagina desiderata, caricarla in memoria, correggere la tabella delle pagine e riavviare l'istruzione. Il riavvio dell'istruzione richiede una nuova operazione di prelievo della stessa, con nuova decodifica e nuovo prelievo dei due operandi; infine occorre ripetere l'addizione. In ogni modo il lavoro da ripetere non è molto, meno di un'istruzione completa, e la ripetizione è necessaria solo nel caso si verifichi un'assenza di pagina.

La difficoltà maggiore si presenta quando un'istruzione può modificare parecchie locazioni diverse. Si consideri, ad esempio, l'istruzione `MVC` (*move character*) del sistema IBM 360/370: quest'istruzione può spostare una sequenza di byte (fino a 256) da una locazione a un'altra con possibilità di sovrapposizione. Se una delle sequenze (quella d'origine o quella di destinazione) esce dal confine di una pagina, e se lo spostamento è stato effettuato solo in parte, si può verificare un'assenza di pagina. Inoltre, se le sequenze d'origine e di destina-

zione si sovrappongono, è probabile che la sequenza d'origine sia stata modificata, in tal caso non è possibile limitarsi a riavviare l'istruzione.

Il problema si può risolvere in due modi. In una delle due soluzioni il microcodice computa e tenta di accedere alle estremità delle due sequenze di byte. Un'eventuale assenza di pagina si può verificare solo in questa fase, prima che si apporti qualsiasi modifica. A questo punto si può compiere lo spostamento perché tutte le pagine interessate si trovano in memoria. L'altra soluzione si serve di registri temporanei per conservare i valori delle locazioni sovrascritte. Nel caso di un'assenza di pagina, si riscrivono tutti i vecchi valori in memoria prima che sia emesso il segnale di eccezione di pagina mancante. Questa operazione riporta la memoria allo stato in cui si trovava prima che l'istruzione fosse avviata, perciò si può ripetere la sua esecuzione.

Sebbene non si tratti certo dell'unico problema da affrontare per estendere un'architettura esistente con la funzionalità della paginazione su richiesta, illustra alcune delle difficoltà da superare. Il sistema di paginazione si colloca tra la CPU e la memoria di un calcolatore e deve essere completamente trasparente al processo utente. L'opinione comune che la paginazione si possa aggiungere a qualsiasi sistema è vera per gli ambienti senza paginazione su richiesta, nei quali un'eccezione di pagina mancante rappresenta un errore fatale, ma è falsa nei casi in cui un'eccezione di pagina mancante implica la necessità di caricare in memoria un'altra pagina e quindi riavviare il processo.

### 9.2.2 Prestazioni della paginazione su richiesta

La paginazione su richiesta può avere un effetto rilevante sulle prestazioni di un calcolatore. Il motivo si può comprendere calcolando il **tempo d'accesso effettivo** per una memoria con paginazione su richiesta. Attualmente, nella maggior parte dei calcolatori il tempo d'accesso alla memoria, che si denota *ma*, varia da 10 a 200 nanosecondi. Finché non si verifichino assenze di pagine, il tempo d'accesso effettivo è uguale al tempo d'accesso alla memoria. Se però si verifica un'assenza di pagina, occorre prima leggere dal disco la pagina interessata e quindi accedere alla parola della memoria desiderata.

Supponendo che $p$ sia la probabilità che si verifichi un'assenza di pagina ($0 \leq p \leq 1$), è probabile che $p$ sia molto vicina allo zero, cioè che ci siano solo poche assenze di pagine. Il **tempo d'accesso effettivo** è dato dalla seguente espressione:

$$\textit{tempo d'accesso effettivo} = (1 - p) \times ma + p \times \textit{tempo di gestione dell'assenza di pagina}$$

Per calcolare il tempo d'accesso effettivo occorre conoscere il tempo necessario alla gestione di un'assenza di pagina. Alla presenza di un'assenza di pagina si esegue la seguente sequenza:

1. segnale d'eccezione al sistema operativo;
2. salvataggio dei registri utente e dello stato del processo;
3. verifica che l'interruzione sia dovuta o meno a una pagina mancante;
4. controllo della correttezza del riferimento alla pagina e determinazione della locazione della pagina nel disco;
5. lettura dal disco e trasferimento in un frame libero:
   a) attesa nella coda relativa a questo dispositivo finché la richiesta di lettura non sia servita;
   b) attesa del tempo di posizionamento e latenza del dispositivo;
   c) inizio del trasferimento della pagina in un frame libero;

6. durante l'attesa, allocazione della CPU a un altro processo utente (scheduling della CPU, facoltativo);
7. ricezione di un'interruzione dal controllore del disco (I/O completato);
8. salvataggio dei registri e dello stato dell'altro processo utente (se è stato eseguito il passo 6);
9. verifica della provenienza dell'interruzione dal disco;
10. aggiornamento della tabella delle pagine e di altre tabelle per segnalare che la pagina richiesta è attualmente presente in memoria;
11. attesa che la CPU sia nuovamente assegnata a questo processo;
12. recupero dei registri utente, dello stato del processo e della nuova tabella delle pagine, quindi ripresa dell'istruzione interrotta.

Non sempre sono necessari tutti i passi sopra elencati. Nel passo 6, ad esempio, si ipotizza che la CPU sia assegnata a un altro processo durante un'operazione di I/O. Tale possibilità permette la multiprogrammazione per mantenere occupata la CPU, ma una volta completato il trasferimento di I/O implica un dispendio di tempo per riprendere la procedura di servizio dell'eccezione di pagina mancante.

In ogni caso, il tempo di servizio dell'eccezione di pagina mancante comporta tre operazioni principali:

1. servizio del segnale di eccezione di pagina mancante;
2. lettura della pagina;
3. riavvio del processo.

La prima e la terza operazione si possono realizzare, per mezzo di un'accurata codifica, in parecchie centinaia di istruzioni. Ciascuna di queste operazioni può richiedere da 1 a 100 microsecondi. D'altra parte, il tempo di cambio di pagina è probabilmente vicino a 8 millisecondi. Un disco ha in genere un tempo di latenza di 3 millisecondi, un tempo di posizionamento di 5 millisecondi e un tempo di trasferimento di 0,05 millisecondi, quindi il tempo totale della paginazione è dell'ordine di 8 millisecondi, compresi i tempi d'esecuzione del codice relativo e delle operazioni dei dispositivi fisici coinvolti. Nel calcolo si è considerato solo il tempo di servizio del dispositivo. Se una coda di processi è in attesa del dispositivo (altri processi che hanno accusato un'assenza di pagina) è necessario considerare anche il tempo di accodamento del dispositivo, poiché occorre attendere che il dispositivo di paginazione sia libero per servire la richiesta, quindi il tempo d'avvicendamento aumenta ulteriormente.

Considerando un tempo medio di servizio dell'eccezione di pagina mancante di 8 millisecondi e un tempo d'accesso alla memoria di 200 nanosecondi, il tempo effettivo d'accesso in nanosecondi è il seguente:

$$\begin{aligned} \textit{tempo d'accesso effettivo} &= (1 - p) \times 200 + p \text{ (8 millisecondi)} \\ &= (1 - p) \times 200 + p \times 8.000.000 \\ &= 200 + 7.999.800 \times p \end{aligned}$$

Il tempo d'accesso effettivo è direttamente proporzionale alla **frequenza delle assenze di pagine** (*page-fault rate*). Se un accesso su 1000 accusa un'assenza di pagina, il tempo d'accesso effettivo è di 8,2 microsecondi. Impiegando la paginazione su richiesta, il calcolatore è ral-

lentato di un fattore pari a 40. Se si desidera un rallentamento inferiore al 10 per cento, occorre che valgano le seguenti condizioni:

$$220 > 200 + 7.999.800 \times p$$
$$20 > 7.999.800 \times p$$
$$p < 0{,}00000025$$

Quindi, per mantenere a un livello ragionevole il rallentamento dovuto alla paginazione, si può permettere meno di un'assenza di pagina ogni 399.990 accessi alla memoria. In un sistema con paginazione su richiesta, è cioè importante tenere bassa la frequenza delle assenze di pagine, altrimenti il tempo effettivo d'accesso aumenta, rallentando molto l'esecuzione del processo.

Un altro aspetto della paginazione su richiesta è la gestione e l'uso generale dell'area d'avvicendamento. L'I/O di un disco relativo all'area d'avvicendamento è generalmente più rapido di quello relativo al file system, poiché essa è composta di blocchi assai più grandi e non s'impiegano ricerche di file e metodi d'allocazione indiretta (Capitolo 12). Perciò il sistema può migliorare l'efficienza della paginazione copiando tutta l'immagine di un file nell'area d'avvicendamento all'avvio del processo e di lì eseguire la paginazione su richiesta. Un'altra possibilità consiste nel richiedere inizialmente le pagine al file system, ma scrivere le pagine nell'area d'avvicendamento al momento della sostituzione. Questo metodo assicura che si leggano sempre dal file system solo le pagine necessarie, ma che tutta la paginazione successiva sia fatta dall'area d'avvicendamento.

Quando s'impiegano file binari, alcuni sistemi tentano di limitare tale area: le pagine richieste per questi file si prelevano direttamente dal file system; tuttavia, quando è richiesta una sostituzione di pagine, i frame possono semplicemente essere sovrascritti, dato che non sono mai stati modificati, e le pagine, se è necessario, possono ancora essere lette dal file system. Seguendo questo criterio, lo stesso file system funziona da memoria ausiliaria (*backing store*). L'area d'avvicendamento si deve in ogni caso usare per le pagine che non sono relative ai file; queste comprendono la **pila** (*stack*) e lo **heap** di un processo. Questa tecnica che sembra essere un buon compromesso si usa in diversi sistemi tra cui Solaris e UNIX BSD.

## 9.3 Copiatura su scrittura

Nel Paragrafo 9.2 si è visto come un processo possa cominciare rapidamente l'esecuzione richiedendo solo la pagina contenente la prima istruzione. La generazione dei processi tramite `fork()`, però, può inizialmente evitare la paginazione su richiesta per mezzo di una tecnica simile alla condivisione delle pagine (Paragrafo 8.4.4), che garantisce la celere generazione dei processi riuscendo anche a minimizzare il numero di pagine allocate al nuovo processo.

Si ricordi che la chiamata di sistema `fork()` crea un processo figlio come duplicato del genitore. Nella sua versione originale la `fork()` creava per il figlio una copia dello spazio d'indirizzi del genitore, duplicando le pagine appartenenti al processo genitore. Considerando che molti processi figli eseguono subito dopo la loro creazione la chiamata di sistema `exec()`, questa operazione di copiatura risulta inutile. In alternativa, si può impiegare una tecnica nota come **copiatura su scrittura** (*copy-on-write*), il cui funzionamento si fonda sulla condivisone iniziale delle pagine da parte dei processi genitori e dei processi figli. Le pagine condivise si contrassegnano come pagine da copiare su scrittura, a significare che, se un processo (genitore o figlio) scrive su una pagina condivisa, il sistema deve creare una co-

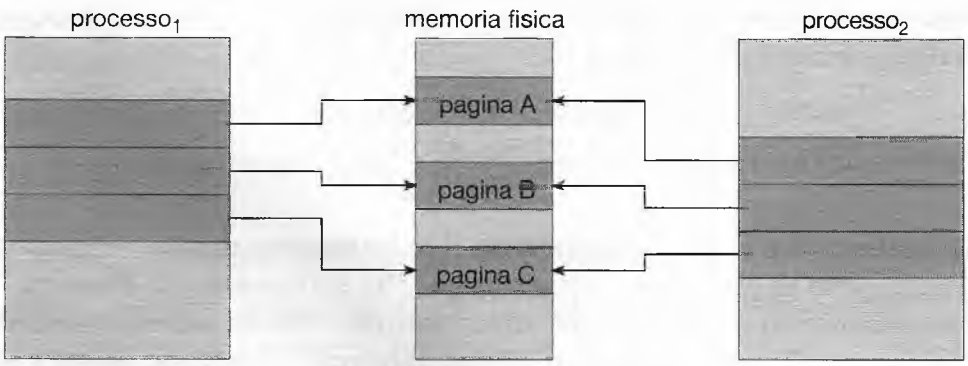



**Figura 9.7** Prima della modifica alla pagina C da parte del processo$_1$.

pia di tale pagina. La copia su scrittura è illustrata rispettivamente nella Figura 9.7 e nella Figura 9.8, che mostrano il contenuto della memoria fisica prima e dopo che il processo 1 abbia modificato la pagina C.

Si consideri ad esempio un processo figlio che cerchi di modificare una pagina contenente parti della pila; il sistema operativo considera questa una pagina da copiare su scrittura e ne crea una copia nello spazio degli indirizzi del processo figlio. Il processo figlio modifica la sua copia della pagina e non la pagina appartenente al processo genitore. È chiaro che, adoperando la tecnica di copiatura su scrittura, si copiano soltanto le pagine modificate da uno dei due processi, mentre tutte le altre sono condivisibili dai processi genitore e figlio. Si noti inoltre che soltanto le pagine modificabili si devono contrassegnare come da copiare su scrittura, mentre quelle che non si possono modificare (ad esempio, le pagine contenenti codice eseguibile) sono condivisibili dai processi genitore e figlio. La tecnica di copiatura su scrittura è piuttosto comune e si usa in diversi sistemi operativi, tra i quali Windows XP, Linux e Solaris.

Quando è necessaria la duplicazione di una pagina secondo la tecnica di copiatura su scrittura, è importante capire da dove si attingerà la pagina libera necessaria. Molti sistemi operativi forniscono, per queste richieste, un gruppo (*pool*) di pagine libere, che di solito si assegnano quando la pila o il cosiddetto *heap* di un processo devono espandersi, oppure proprio per gestire pagine da copiare su scrittura. L'allocazione di queste pagine di solito avviene secondo una tecnica nota come **azzeramento su richiesta** (*zero-fill-on-demand*); prima dell'allocazione si riempiono di zeri le pagine, cancellandone in questo modo tutto il contenuto precedente.

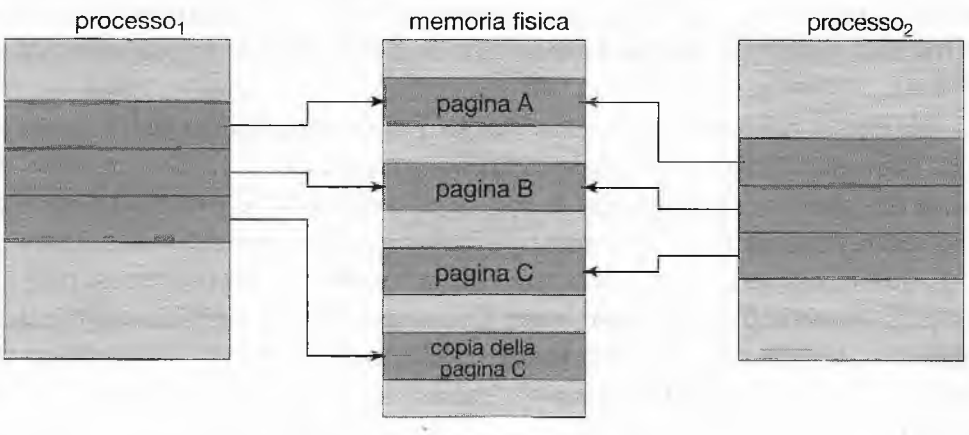



**Figura 9.8** Dopo la modifica alla pagina C da parte del processo$_1$.

Diverse versioni di UNIX (compreso Solaris e Linux) offrono anche una variante della chiamata di sistema `fork()` – detta `vfork()` (per *virtual memory fork*). La `vfork()` offre un'alternativa all'uso della `fork()` con copiatura su scrittura. Con la `vfork()` il processo genitore viene sospeso e il processo figlio usa lo spazio d'indirizzi del genitore. Poiché la `vfork()` non usa la copiatura su scrittura, se il processo figlio modifica qualche pagina dello spazio d'indirizzi del genitore, le pagine modificate saranno visibili al processo genitore non appena riprenderà il controllo. Per assicurarsi che il processo figlio non modifichi lo spazio d'indirizzi del genitore è quindi necessaria molta attenzione nell'uso di `vfork()`. La chiamata di sistema `vfork()` è adatta al caso in cui il processo figlio esegua una `exec()` immediatamente dopo la sua creazione. Poiché non richiede alcuna copiatura delle pagine, la `vfork()` è un metodo di creazione dei processi molto efficiente, in alcuni casi impiegato per realizzare le interfacce degli interpreti dei comandi in UNIX.

## 9.4 Sostituzione delle pagine

Nelle descrizioni fatte finora, la frequenza (*rate*) delle assenze di pagine non è stata un problema grave, giacché ogni pagina poteva essere assente al massimo una volta, e precisamente la prima volta in cui si effettuava un riferimento a essa. Tale rappresentazione tuttavia non è molto precisa. Se un processo di 10 pagine ne impiega effettivamente solo la metà, la paginazione su richiesta fa risparmiare l'I/O necessario per caricare le cinque pagine che non sono mai usate. Il grado di multiprogrammazione potrebbe essere aumentato eseguendo il doppio dei processi. Quindi, disponendo di 40 frame, si potrebbero eseguire otto processi anziché i quattro che si eseguirebbero se ciascuno di loro richiedesse 10 blocchi di memoria, cinque dei quali non sarebbero mai usati.

Aumentando il grado di multiprogrammazione, si **sovrassegna** la memoria. Eseguendo sei processi, ciascuno dei quali è formato da 10 pagine, di cui solo cinque sono effettivamente usate, s'incrementerebbero l'utilizzo e la produttività della CPU e si risparmierebbero 10 frame. Tuttavia è possibile che ciascuno di questi processi, per un insieme particolare di dati, abbia improvvisamente necessità di impiegare tutte le 10 pagine, perciò sarebbero necessari 60 frame, mentre ne sono disponibili solo 40.

Si consideri inoltre che la memoria del sistema non si usa solo per contenere pagine di programmi: le aree di memoria per l'I/O impegnano una rilevante quantità di memoria. Ciò può aumentare le difficoltà agli algoritmi di allocazione della memoria. Decidere quanta memoria assegnare all'I/O e quanta alle pagine dei programmi è un problema rilevante. Alcuni sistemi riservano una quota fissa di memoria per l'I/O, altri permettono sia ai processi utenti sia al sottosistema di I/O di competere per tutta la memoria del sistema.

La **sovrallocazione** (*over-allocation*) si può illustrare come segue. Durante l'esecuzione di un processo utente si verifica un'assenza di pagina. Il sistema operativo determina la locazione del disco in cui risiede la pagina desiderata, ma poi scopre che la lista dei frame liberi è *vuota*: tutta la memoria è in uso; si veda a questo proposito la Figura 9.9.

A questo punto il sistema operativo può scegliere tra diverse possibilità, ad esempio può terminare il processo utente. Tuttavia, la paginazione su richiesta è un tentativo che il sistema operativo fa per migliorare l'utilizzo e la produttività del sistema di calcolo. Gli utenti non devono sapere che i loro processi sono eseguiti su un sistema paginato. La paginazione deve essere logicamente trasparente per l'utente, quindi la terminazione del processo non costituisce la scelta migliore.

na causa un'eccezione di pagina mancante. In quel momento, la pagina viene riportata in memoria e può sostituire un'altra pagina del processo.

Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali: occorre sviluppare un **algoritmo di allocazione dei frame** e un **algoritmo di sostituzione delle pagine**. Se sono presenti più processi in memoria, occorre decidere quanti frame vadano assegnati a ciascun processo. Inoltre, quando è richiesta una sostituzione di pagina, occorre selezionare i frame da sostituire. La progettazione di algoritmi idonei a risolvere questi problemi è un compito importante, poiché l'I/O nei dischi è piuttosto oneroso. Anche miglioramenti minimi ai metodi di paginazione su richiesta ne apportano notevoli alle prestazioni del sistema.

Esistono molti algoritmi di sostituzione delle pagine; probabilmente ogni sistema operativo ha il proprio schema di sostituzione. È quindi necessario stabilire un criterio per selezionare un algoritmo di sostituzione particolare; generalmente si sceglie quello con la minima **frequenza delle assenze di pagine** (*page-fault rate*).

Un algoritmo si valuta effettuandone l'esecuzione su una particolare successione di riferimenti alla memoria e calcolando il numero di assenze di pagine. La successione dei riferimenti alla memoria è detta, appunto, **successione dei riferimenti**. Queste successioni si possono generare artificialmente (ad esempio con un generatore di numeri casuali), oppure analizzando un dato sistema e registrando l'indirizzo di ciascun riferimento alla memoria. Quest'ultima opzione permette di ottenere un numero elevato di dati, dell'ordine di un milione di indirizzi al secondo. Per ridurre questa quantità di dati occorre notare due fatti.

Innanzitutto, di una pagina di date dimensioni, generalmente fissate dall'architettura del sistema, si considera solo il numero della pagina anziché l'intero indirizzo. In secondo luogo, quando si fa riferimento a una pagina $p$, i riferimenti alla stessa pagina *immediatamente* successivi al primo non accusano assenze di pagine: dopo il primo riferimento, la pagina $p$ è presente in memoria.

Esaminando un processo si potrebbe ad esempio registrare la seguente successione di indirizzi:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

che, a 100 byte per pagina, si riduce alla seguente successione di riferimenti:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Per stabilire il numero di assenze di pagine relativo a una particolare successione di riferimenti e a un particolare algoritmo di sostituzione delle pagine, occorre conoscere anche il numero dei frame disponibili. Naturalmente, aumentando il numero di quest'ultimi diminuisce il numero di assenze di pagine. Per la successione dei riferimenti precedentemente esaminata, ad esempio, dati tre o più blocchi di memoria si possono verificare tre sole assenze di pagine: una per il primo riferimento di ogni pagina. D'altra parte, se si dispone di un solo frame è necessaria una sostituzione per ogni riferimento, con il risultato di 11 assenze di pagine. In generale è prevista una curva simile a quella della Figura 9.11. Aumentando il numero dei frame, il numero di assenze di pagine diminuisce fino al livello minimo. Naturalmente aggiungendo memoria fisica il numero dei frame aumenta.

Per illustrare gli algoritmi di sostituzione delle pagine s'impiega la seguente successione di riferimenti

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

per una memoria con tre frame.

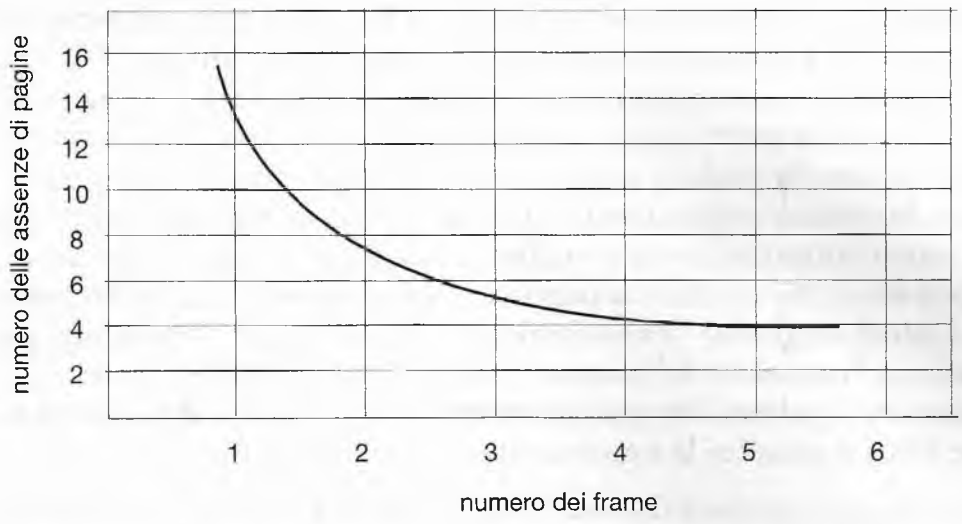



**Figura 9.11** Grafico che illustra il numero di assenze di pagine rispetto al numero dei frame.

## 9.4.2 Sostituzione delle pagine secondo l'ordine d'arrivo (FIFO)

L'algoritmo di sostituzione delle pagine più semplice è un algoritmo FIFO. Questo algoritmo associa a ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo. Occorre notare che non è strettamente necessario registrare l'istante in cui si carica una pagina in memoria; infatti si possono strutturare secondo una coda FIFO tutte le pagine presenti in memoria. In questo caso si sostituisce la pagina che si trova nel primo elemento della coda. Quando si carica una pagina in memoria, la si inserisce nell'ultimo elemento della coda.

Nella successione di riferimenti adottata, i nostri tre frame sono inizialmente vuoti. I primi tre riferimenti (7, 0, 1) accusano ciascuno un'assenza di pagina con conseguente caricamento delle relative pagine nei frame vuoti. Il riferimento successivo (2) causa la sostituzione della pagina 7, perché essa è stata caricata per prima in memoria. Siccome 0 è il riferimento successivo e si trova già in memoria, per questo riferimento non ha luogo alcuna assenza di pagina. Il primo riferimento a 3 causa la sostituzione della pagina 0, che era la prima fra le tre pagine in memoria (0, 1, e 2) da caricare. A causa di questa sostituzione il riferimento successivo, a 0, accuserà un'assenza di pagina. La pagina 1 è allora sostituita dalla pagina 0. Questo processo prosegue come è illustrato nella Figura 9.12. Le pagine presenti nei tre frame sono indicate ogni volta che si verifica un'assenza di pagina. Complessivamente si hanno 15 assenze di pagine.

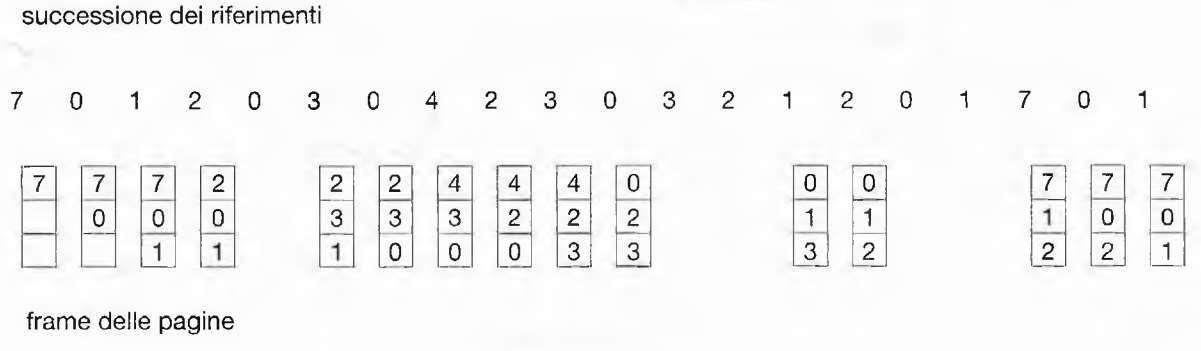



**Figura 9.12** Algoritmo di sostituzione delle pagine FIFO.

L'algoritmo FIFO di sostituzione delle pagine è facile da capire e da programmare; tuttavia la sue prestazioni non sono sempre buone. La pagina sostituita potrebbe essere un modulo di inizializzazione usato molto tempo prima e che non serve più, ma potrebbe anche contenere una variabile molto usata, inizializzata precedentemente, e ancora in uso.

Occorre notare che anche se si sceglie una pagina da sostituire che è in uso attivo, tutto continua a funzionare correttamente. Dopo aver rimosso una pagina attiva per inserirne una nuova, quasi immediatamente si verifica un'eccezione di pagina mancante per riprendere la pagina attiva. Per riportare la pagina attiva in memoria è necessario sostituire un'altra pagina. Quindi, scegliendo una sostituzione errata, aumenta la frequenza di pagine mancanti che rallenta l'esecuzione del processo, ma non vengono causati errori.

Per illustrare i problemi che possono insorgere con l'uso dell'algoritmo di sostituzione delle pagine FIFO, si consideri la seguente successione di riferimenti:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Nella Figura 9.13 è illustrata la curva delle assenze di pagine in funzione del numero dei frame disponibili. Occorre notare che il numero delle assenze di pagine (10) per quattro frame è *maggiore* del numero delle assenze di pagine (9) per tre frame. Questo inatteso risultato è noto col nome di **anomalia di Belady**, e riflette il fatto che, con alcuni algoritmi di sostituzione delle pagine, la frequenza delle assenze di pagine può *aumentare* con l'aumentare del numero dei frame assegnati. A prima vista sembra logico supporre che fornendo più memoria a un processo le prestazioni di quest'ultimo migliorino. Si è invece notato che questo presupposto non sempre è vero; l'anomalia di Belady ne è la prova.

## 9.4.3 Sostituzione ottimale delle pagine

In seguito alla scoperta dell'anomalia di Belady, la ricerca si è diretta verso un **algoritmo ottimale di sostituzione delle pagine**. Tale algoritmo è quello che fra tutti gli algoritmi presenta la minima frequenza di assenze di pagine e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato OPT o MIN. È semplicemente:

*si sostituisce la pagina che non si userà per il periodo di tempo più lungo.*

L'uso di quest'algoritmo di sostituzione delle pagine assicura la frequenza di assenze di pagine più bassa possibile per un numero fisso di frame.

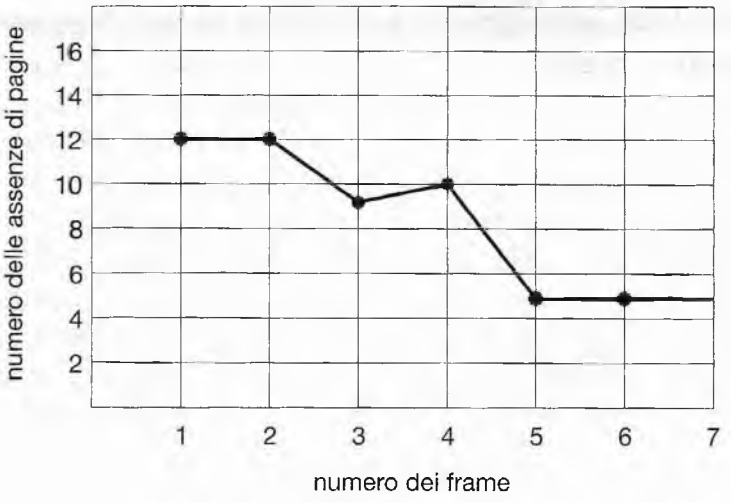



**Figura 9.13** Curva delle assenze di pagine per sostituzione FIFO su una successione di riferimenti.

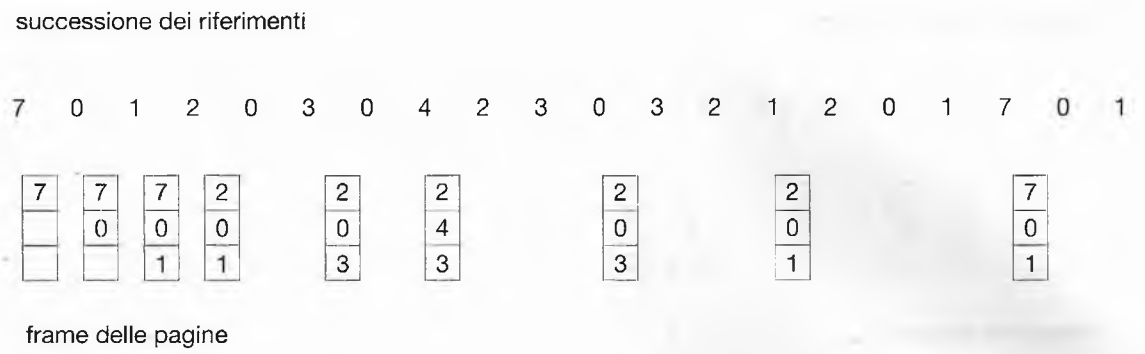



**Figura 9.14** Algoritmo ottimale di sostituzione delle pagine.

Ad esempio, nella successione dei riferimenti considerata, l'algoritmo ottimale di sostituzione delle pagine produce nove assenze di pagine, come è mostrato nella Figura 9.14. I primi tre riferimenti causano assenze di pagine che riempiono i tre blocchi di memoria vuoti. Il riferimento alla pagina 2 determina la sostituzione della pagina 7, perché la 7 non è usata fino al riferimento 18, mentre la pagina 0 viene usata al 5 e la pagina 1 al 14. Il riferimento alla pagina 3 causa la sostituzione della pagina, poiché la pagina 1 è l'ultima delle tre pagine in memoria cui si fa nuovamente riferimento. Con sole nove assenze di pagine, la sostituzione ottimale risulta assai migliore di quella ottenuta con un algoritmo FIFO, dove le assenze di pagine erano 15. Ignorando le prime tre assenze di pagine, che si verificano con tutti gli algoritmi, la sostituzione ottimale è due volte migliore rispetto all'algoritmo FIFO; nessun algoritmo di sostituzione può gestire questa successione di riferimenti a tre blocchi di memoria con meno di nove assenze di pagine.

Sfortunatamente l'algoritmo ottimale di sostituzione delle pagine è difficile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti. Una situazione analoga si è riscontrata con l'algoritmo SJF di scheduling della CPU, nel Paragrafo 5.3.2. Quindi, l'algoritmo ottimale si impiega soprattutto per studi comparativi. Ad esempio, può risultare abbastanza utile sapere che, sebbene un algoritmo nuovo non sia ottimale, nel peggiore dei casi le sue prestazioni sono inferiori del 12,3 per cento rispetto a quelle dell'algoritmo ottimale, e mediamente questa percentuale è del 4,7 per cento.

### 9.4.4 Sostituzione delle pagine usate meno recentemente (LRU)

Se l'algoritmo ottimale non è realizzabile, è forse possibile realizzarne un'approssimazione. La distinzione fondamentale tra gli algoritmi FIFO e OPT, oltre quella di guardare avanti o indietro nel tempo, consiste nel fatto che l'algoritmo FIFO impiega l'istante in cui una pagina è stata caricata in memoria, mentre l'algoritmo OPT impiega l'istante in cui una pagina è *usata*. Usando come approssimazione di un futuro vicino un passato recente, si sostituisce la pagina che *non è stata usata* per il periodo più lungo. Il metodo appena descritto è noto come **algoritmo LRU** (*least recently used*).

La sostituzione LRU associa a ogni pagina l'istante in cui è stata usata per l'ultima volta. Quando occorre sostituire una pagina, l'algoritmo LRU sceglie quella che non è stata usata per il periodo più lungo. Questa strategia costituisce l'algoritmo ottimale di sostituzione delle pagine con ricerca all'indietro nel tempo, anziché in avanti. Infatti, supponendo che $S^R$ sia la successione inversa di una successione di riferimenti $S$, la frequenza di assenze di pagine per l'algoritmo OPT su $S$ è uguale a quella per l'algoritmo LRU su $S^R$. Allo stesso modo, la frequenza di assenze di pagine per l'algoritmo LRU su $S$ è uguale a quella per l'algoritmo OPT su $S^R$.

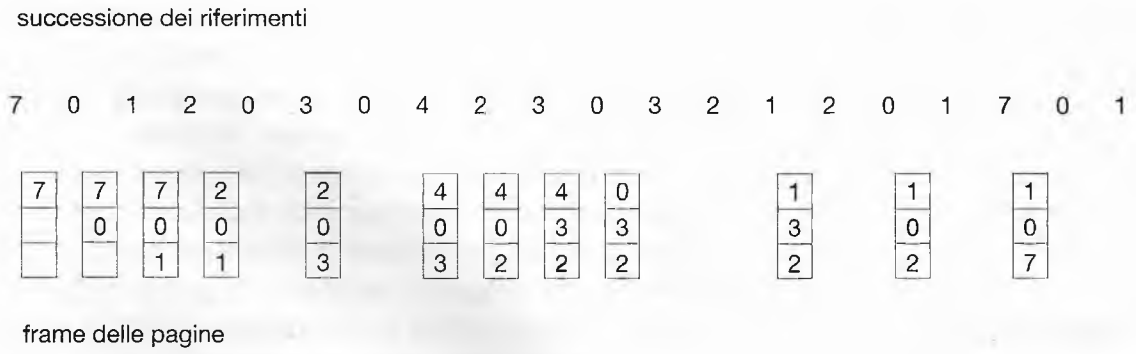



**Figura 9.15** Algoritmo di sostituzione delle pagine LRU.

Il risultato dell'applicazione dell'algoritmo LRU alla successione dei riferimenti dell'esempio è illustrato nella Figura 9.15. L'algoritmo LRU produce 12 assenze di pagine. Occorre notare che le prime cinque assenze di pagine sono le stesse della sostituzione ottimale. Quando si presenta il riferimento alla pagina 4, però, l'algoritmo LRU trova che, fra i tre blocchi di memoria, quello usato meno recentemente è della pagina 2. Quindi, l'algoritmo LRU sostituisce la pagina 2 senza sapere che sta per essere usata. Quando si verifica l'assenza della pagina 2, l'algoritmo LRU sostituisce la pagina 3, poiché, fra le tre pagine in memoria (0, 3, 4), la pagina 3 è quella usata meno recentemente. Nonostante questi problemi, la sostituzione LRU, con 12 assenze di pagine, è in ogni modo migliore della sostituzione FIFO, con 15 assenze di pagine.

Il criterio LRU si usa spesso come algoritmo di sostituzione delle pagine ed è considerato valido. Il problema principale riguarda la realizzazione della sostituzione stessa. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza da parte dell'architettura del sistema di calcolo. Il problema consiste nel determinare un ordine per i frame definito secondo il momento dell'ultimo uso. Si possono realizzare le due seguenti soluzioni.

- **Contatori.** Nel caso più semplice, a ogni elemento della tabella delle pagine si associa un campo del momento d'uso, e alla CPU si aggiunge un contatore che si incrementa a ogni riferimento alla memoria. Ogni volta che si fa un riferimento a una pagina, si copia il contenuto del registro contatore nel campo del momento d'uso nella tabella relativa a quella specifica pagina. In questo modo è sempre possibile conoscere il momento in cui è stato fatto l'ultimo riferimento a ogni pagina. Si sostituisce la pagina con il valore associato più piccolo. Questo schema implica una ricerca all'interno della tabella delle pagine per individuare la pagina usata meno recentemente (LRU), e una scrittura in memoria (nel campo del momento d'uso della tabella delle pagine) per ogni accesso alla memoria. I riferimenti temporali si devono mantenere anche quando, a seguito dello scheduling della CPU, si modificano le tabelle delle pagine. Occorre infine considerare il superamento della capacità del contatore (*overflow*).
- **Pila.** Un altro metodo per la realizzazione della sostituzione delle pagine LRU prevede la presenza di una pila dei numeri delle pagine. Ogni volta che si fa un riferimento a una pagina, la si estrae dalla pila e la si colloca in cima a quest'ultima. In questo modo, in cima alla pila si trova sempre la pagina usata per ultima, mentre in fondo si trova la pagina usata meno recentemente, com'è illustrato dalla Figura 9.16. Poiché alcuni elementi si devono estrarre dal centro della pila, la migliore realizzazione si ottiene usando una lista doppiamente concatenata, con un puntatore all'elemento iniziale e uno a quello finale. Per estrarre una pagina dalla pila e collocarla in cima, nel caso peggiore è necessario modificare sei puntatori. Ogni aggiornamento è un po' più costoso, ma per una sostituzione non si deve compiere alcuna ricerca; il puntatore dell'elemento di co-

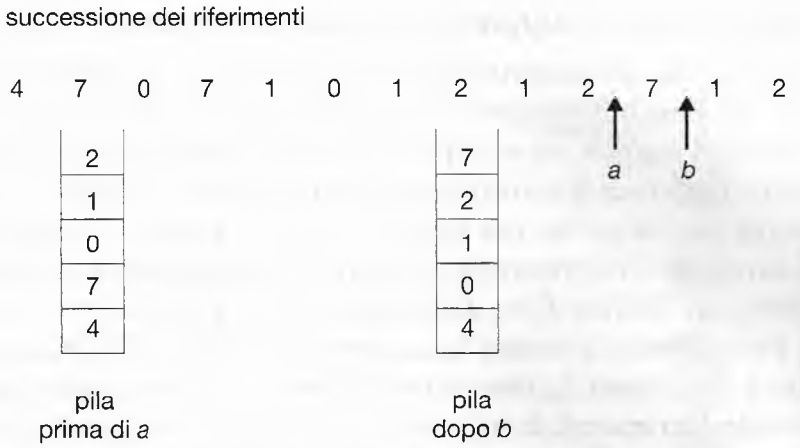



**Figura 9.16** Uso di una pila per registrare i più recenti riferimenti alle pagine.

da punta al fondo della pila, vale a dire la pagina usata meno recentemente. Questo metodo è adatto soprattutto alle realizzazioni programmate (o microprogrammate) della sostituzione delle pagine LRU.

Né la sostituzione ottimale né quella LRU sono soggette all'anomalia di Belady. Esiste una classe di algoritmi di sostituzione delle pagine, chiamati **algoritmi a pila**, che non presenta l'anomalia di Belady. Un algoritmo a pila è un algoritmo per il quale è possibile mostrare che l'insieme delle pagine in memoria per $n$ frame è sempre un *sottoinsieme* dell'insieme delle pagine che dovrebbero essere in memoria per $n + 1$ frame. Per la sostituzione LRU, l'insieme di pagine in memoria è costituito delle $n$ pagine cui si è fatto riferimento più recentemente. Se il numero dei frame è aumentato, queste $n$ pagine continuano a essere quelle cui si è fatto riferimento più recentemente e quindi restano in memoria.

Oltre i registri TLB standard, senza l'ausilio dell'architettura sarebbe inconcepibile anche la realizzazione della sostituzione LRU. L'aggiornamento dei campi del contatore o della pila si deve effettuare per *ogni* riferimento alla memoria. Se per ogni riferimento si dovesse adoperare un segnale d'interruzione per permettere alle procedure del sistema operativo di modificare tali strutture dati, tutti i riferimenti alla memoria sarebbero rallentati di un fattore almeno pari a 10, quindi anche tutti i processi utenti sarebbero rallentati di un uguale fattore. Pochi sistemi possono permettersi un tale sovraccarico per la gestione della memoria.

### 9.4.5 Sostituzione delle pagine per approssimazione a LRU

Sono pochi i sistemi di calcolo che dispongono di un'architettura adatta a una vera sostituzione LRU delle pagine. Nei sistemi che non offrono tali caratteristiche specifiche si devono impiegare altri algoritmi di sostituzione delle pagine, ad esempio l'algoritmo FIFO. Molti sistemi tuttavia possono fornire un aiuto: un **bit di riferimento**. Il bit di riferimento a una pagina è impostato automaticamente dall'architettura del sistema ogni volta che si fa un riferimento a quella pagina, che sia una lettura o una scrittura su qualsiasi byte della pagina. I bit di riferimento sono associati a ciascun elemento della tabella delle pagine.

Inizialmente, il sistema operativo azzera tutti i bit. Quando s'inizia l'esecuzione di un processo utente, l'architettura del sistema imposta a 1 il bit associato a ciascuna pagina cui si fa riferimento. Dopo qualche tempo è possibile stabilire quali pagine sono state usate semplicemente esaminando i bit di riferimento. Non è però possibile conoscere l'*ordine* d'uso. È questa l'informazione alla base di molti algoritmi per la sostituzione delle pagine che approssimano LRU.

### 9.4.5.1 Algoritmo con bit supplementari di riferimento

Ulteriori informazioni sull'ordinamento si possono ottenere registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella in memoria una serie di bit per ogni pagina. A intervalli regolari, ad esempio di 100 millisecondi, un segnale d'interruzione del timer del sistema trasferisce il controllo al sistema operativo. Questo sposta il bit di riferimento per ciascuna pagina nel bit più significativo della sequenza, traslando gli altri bit a destra di 1 bit e scartando il bit meno significativo. Questi registri a scorrimento, ad esempio di 8 bit, contengono l'ordine d'uso delle pagine relativo agli ultimi otto periodi di tempo. Se il registro a scorrimento contiene la successione di bit 00000000, significa che la pagina associata non è stata usata da otto periodi di tempo; a una pagina usata almeno una volta per ogni periodo corrisponde la successione 11111111 nel registro a scorrimento. Una pagina cui corrisponde la successione 11000100 nel relativo registro, è stata usata più recentemente di quanto non lo sia stata una cui è associata la successione 01110111. Interpretando queste successioni di bit come interi senza segno, la pagina cui è associato il numero minore è la pagina LRU, e può essere sostituita. In ogni caso l'unicità dei numeri non è garantita. Si possono sostituire (o scaricare dalla memoria all'area d'avvicendamento) tutte le pagine con il valore minore, oppure si può ricorrere a una selezione FIFO.

Il numero dei bit può ovviamente essere variato: si stabilisce secondo l'architettura disponibile per accelerarne al massimo la modifica. Nel caso limite tale numero si riduce a zero, lasciando soltanto il bit di riferimento e definendo un algoritmo noto come **algoritmo di sostituzione delle pagine con seconda chance.**

### 9.4.5.2 Algoritmo con seconda chance

L'algoritmo di base per la sostituzione con seconda chance è un algoritmo di sostituzione di tipo FIFO. Dopo aver selezionato una pagina si controlla il bit di riferimento, se il suo valore è 0, si sostituisce la pagina; se il bit di riferimento è impostato a 1, si dà una seconda chance alla pagina e la selezione passa alla successiva pagina FIFO. Quando una pagina riceve la seconda chance, si azzera il suo bit di riferimento e si aggiorna il suo istante d'arrivo al momento attuale. In questo modo, una pagina cui si offre una seconda chance non viene mai sostituita fiché tutte le altre pagine non siano state sostituite, oppure non sia stata offerta loro una seconda chance. Inoltre, se una pagina è usata abbastanza spesso, in modo che il suo bit di riferimento sia sempre impostato a 1, non viene mai sostituita.

Un metodo per realizzare l'algoritmo con seconda chance, detto anche a orologio (*clock*), è basato sull'uso di una coda circolare, in cui un puntatore indica qual è la prima pagina da sostituire. Quando serve un frame, si fa avanzare il puntatore finché non si trovi in corrispondenza di una pagina con il bit di riferimento 0; a ogni passo si azzera il bit di riferimento appena esaminato (Figura 9.17). Una volta trovata una pagina "vittima", la si sostituisce e si inserisce la nuova pagina nella coda circolare nella posizione corrispondente. Si noti che nel caso peggiore, quando tutti i bit sono impostati a 1, il puntatore percorre un ciclo su tutta la coda, dando a ogni pagina una seconda chance. Prima di selezionare la pagina da sostituire, azzera tutti i bit di riferimento. Se tutti i bit sono a 1, la sostituzione con seconda chance si riduce a una sostituzione FIFO.

### 9.4.5.3 Algoritmo con seconda chance migliorato

L'algoritmo con seconda chance descritto precedentemente si può migliorare considerando i bit di riferimento e di modifica (si veda il Paragrafo 9.4.1) come una coppia ordinata, con cui si possono ottenere le seguenti quattro classi:

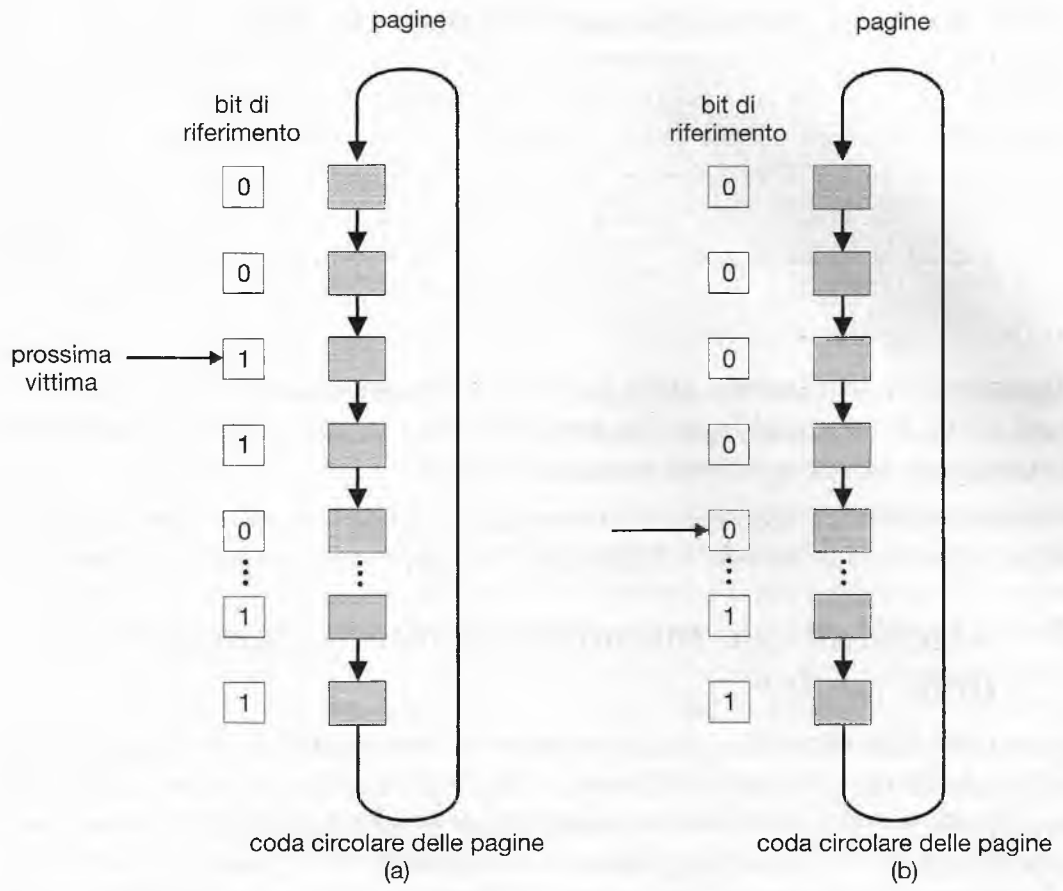



**Figura 9.17** Algoritmo di sostituzione delle pagine con seconda chance (orologio).

1. (0, 0) né recentemente usato né modificato – migliore pagina da sostituire;
2. (0, 1) non usato recentemente, ma modificato – la pagina non così buona poiché prima di essere sostituita deve essere scritta in memoria secondaria;
3. (1, 0) usato recentemente ma non modificato – probabilmente la pagina sarà presto ancora usata;
4. (1, 1) usato recentemente e modificato – probabilmente la pagina sarà presto ancora usata e dovrà essere scritta in memoria secondaria prima di essere sostituita.

Ogni pagina rientra in una di queste quattro classi. Alla richiesta di una sostituzione di pagina, si usa lo stesso schema impiegato nell'algoritmo a orologio, ma anziché controllare se la pagina puntata ha il bit di riferimento impostato a 1, si esaminano le classi cui la pagina appartiene e si sostituisce la prima pagina che si trova nella classe minima non vuota. Si noti che la coda circolare deve essere scandita più volte prima di trovare una pagina da sostituire.

La differenza principale tra questo algoritmo e il più semplice algoritmo a orologio è che nel primo si dà la preferenza alle pagine modificate, al fine di ridurre il numero di I/O richiesti.

### 9.4.6 Sostituzione delle pagine basata su conteggio

Esistono molti altri algoritmi che si possono usare per la sostituzione delle pagine. Ad esempio, si potrebbe usare un contatore del numero dei riferimenti fatti a ciascuna pagina, e sviluppare i due seguenti schemi.

- **Algoritmo di sostituzione delle pagine meno frequentemente usate** (*least frequently used*, LFU); richiede che si sostituisca la pagina con il conteggio più basso. La ragione di questa scelta è che una pagina usata attivamente deve avere un conteggio di riferimento alto. Il punto debole di questo algoritmo è rappresentato dai casi in cui una pagina è usata molto intensamente durante la fase iniziale di un processo, ma poi non viene più usata. Poiché è stata usata intensamente il suo conteggio è alto, quindi rimane in memoria anche se non è più necessaria. Una soluzione può essere quella di spostare i conteggi a destra di un bit a intervalli regolari, formando un conteggio per l'uso medio con esponente decrescente.
- **Algoritmo di sostituzione delle pagine più frequentemente usate** (*most frequently used*, MFU); è basato sul fatto che, probabilmente, la pagina con il contatore più basso è stata appena inserita e non è stata ancora usata.

Le sostituzioni MFU e LFU non sono molto comuni, poiché la realizzazione di questi algoritmi è abbastanza onerosa; inoltre, tali algoritmi non approssimano bene la sostituzione OPT.

### 9.4.7 Algoritmi con memorizzazione transitoria delle pagine

Oltre a uno specifico algoritmo, per la sostituzione delle pagine si usano spesso anche altre procedure; ad esempio, i sistemi hanno generalmente un gruppo di frame liberi (*pool of free frames*). Quando si verifica un'assenza di pagina, si sceglie innanzi tutto un frame vittima, ma prima che sia scritta in memoria secondaria, si trasferisce la pagina richiesta in un frame libero del gruppo. Questa procedura permette al processo di ricominciare al più presto, senza attendere che la pagina vittima sia scritta in memoria secondaria. Quando nel seguito si scrive la vittima in memoria secondaria, si aggiunge il suo frame al gruppo dei frame liberi.

Quest'idea si può estendere conservando una lista delle pagine modificate: ogniqualvolta il dispositivo di paginazione è inattivo, si sceglie una pagina modificata, la si scrive nel disco e si reimposta il suo bit di modifica. Questo schema aumenta la probabilità che, al momento della selezione per la sostituzione, la pagina non abbia subito modifiche e non debba essere scritta in memoria secondaria.

Anche un'altra modifica prevede l'uso di un gruppo di frame liberi ma, in questo caso, per ricordare quale pagina era contenuta in ciascun frame. Poiché quando si scrive il contenuto di un frame in un disco tale contenuto non cambia, se è necessaria, la vecchia pagina è ancora utilizzabile direttamente dal gruppo dei frame liberi, prima che quel sia riusato quel frame. In questo caso non è necessario alcun I/O. Se si verifica un'assenza di pagina occorre controllare se la pagina richiesta si trova nel gruppo dei frame liberi; se non c'è si deve individuare un frame libero e trasferirvi la pagina.

Questa tecnica, insieme con l'algoritmo di sostituzione FIFO, è usata dal sistema VAX/VMS. Quando l'algoritmo FIFO sostituisce per errore una pagina ancora in uso, la si ricupera rapidamente dal gruppo dei frame liberi senza ricorrere a operazioni di I/O. Il gruppo dei frame liberi offre protezione contro l'algoritmo di sostituzione FIFO, relativamente povero, ma semplice. Questo metodo è necessario poiché le prime versioni del VAX non disponevano del bit di riferimento correttamente realizzato.

Alcune versioni di UNIX adottano questo algoritmo insieme a quello con seconda chance. In effetti, si tratta di un'utile integrazione a qualunque algoritmo di sostituzion, al fine di ridurre il prezzo pagato per l'eventuale errata esclusione di una pagina.

### 9.4.8 Applicazioni e sostituzione della pagina

In taluni casi, le applicazioni che accedono ai dati tramite la memoria virtuale del sistema operativo non conseguono prestazioni migliori di quelle che il sistema, senza impiegare alcun buffer, potrebbe offrire. Si pensi, quale esempio tipico, a una base di dati che gestisce la memoria e il buffer dell'I/O in modo autonomo. Applicazioni come questa capiscono il funzionamento della memoria e del disco che occupano meglio di quanto possa fare un sistema operativo, che applica algoritmi adatti a un uso generale. Se il sistema operativo adotta un buffer per l'I/O, e così pure fa l'applicazione, la quantità di memoria necessaria per l'I/O sarà inutilmente raddoppiata.

Un altro esempio proviene dagli archivi di dati, che effettuano spesso impegnative letture sequenziali del disco, seguite da calcoli e scritture. L'algoritmo LRU eliminerebbe le pagine vecchie per conservare le nuove, mentre in questo caso è ragionevole attendersi la lettura delle pagine vecchie in luogo di quelle nuove (dato che l'applicazione esegue periodicamente la lettura sequenziale). In queste circostanze l'algoritmo MFU sarebbe più efficiente di LRU.

Per risolvere tali problemi, alcuni sistemi operativi permettono a certi programmi di utilizzare una partizione del disco come un array sequenziale di blocchi logici, senza ricorrere alle strutture di dati del file system. Un simile array è anche detto **disco di basso livello** (*raw disk*), e il relativo I/O è denominato I/O **di basso livello** (*raw I/O*). Il disco di basso livello salta tutti i servizi del file system, come la paginazione su richiesta dei file in ingresso e in uscita, i lock dei file, il prefetching, l'allocazione dello spazio, i nomi dei file e le directory. Si noti come, sebbene alcune applicazioni siano più efficienti nel gestire i propri servizi specifici di memorizzazione sul disco di basso livello, quasi tutte hanno una resa migliore quando operano con i servizi regolari del file system.

## 9.5 Allocazione dei frame

Torniamo al concetto di allocazione. A questo punto occorre stabilire un criterio per l'allocazione della memoria libera ai diversi processi. Come esempio, è possibile considerare un caso in cui 93 frame liberi si debbano assegnare a due processi.

Il caso più semplice di memoria virtuale è il sistema con utente singolo. Si consideri un sistema monoutente che disponga di 128 KB di memoria, con pagine di 1 KB. Complessivamente sono presenti 128 frame. Il sistema operativo può occupare 35 KB, lasciando 93 frame per il processo utente. In condizioni di paginazione su richiesta pura, tutti i 93 blocchi di memoria sono inizialmente posti nella lista dei frame liberi. Quando comincia l'esecuzione, il processo utente genera una sequenza di eccezioni di pagine mancanti. Le prime 93 pagine assenti ricevono i frame liberi dalla lista. Una volta esaurita quest'ultima, per stabilire quale tra le 93 pagine presenti in memoria si debba sostituire con la novantaquattresima, si può usare un algoritmo di sostituzione delle pagine. Terminato il processo, si reinseriscono i 93 frame nella lista dei frame liberi.

Questa strategia è semplice, ma può subire molte variazioni. Si può richiedere che il sistema operativo assegni tutto lo spazio richiesto dalle proprie strutture dati attingendo dalla lista dei frame liberi. Quando questo spazio è inutilizzato dal sistema operativo può essere sfruttato per la paginazione utente. Un'altra variante prevede di riservare sempre tre frame liberi, in modo che quando si verifica un'assenza di pagina sia disponibile un frame libero in cui trasferire la pagina richiesta. Mentre ha luogo il trasferimento della pagina, si può fare una sostituzione, la pagina coinvolta viene poi scritta nel disco mentre il processo utente

continua l'esecuzione. Sono possibili anche altre varianti, ma la strategia di base è chiara: al processo utente si assegna qualsiasi frame libero.

### 9.5.1 Numero minimo di frame

Le strategie di allocazione dei frame sono soggette a parecchi vincoli. Non si possono assegnare più frame di quanti siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre è necessario assegnare almeno un numero minimo di frame. Naturalmente, col diminuire del numero dei frame allocati a ciascun processo aumenta la frequenza delle assenze di pagine, con conseguente rallentamento dell'esecuzione del processo. Esaminiamo quest'ultimo requisito in maggiore dettaglio.

Una delle ragioni per allocare sempre un numero minimo di frame è legata alle prestazioni. Ovviamente, al decrescere del numero dei frame allocati a ciascun processo aumenta la frequenza di mancanza di pagina, con conseguente ritardo dell'esecuzione dei processi. Inoltre va ricordato che, quando si verifica un'assenza di pagina prima che sia stata completata l'esecuzione di un'istruzione, quest'ultima deve essere riavviata. Di conseguenza, i frame disponibili devono essere in numero sufficiente per contenere tutte le pagine cui ogni singola istruzione può far riferimento.

Si consideri, ad esempio, un calcolatore in cui tutte le istruzioni di riferimento alla memoria hanno solo un indirizzo di memoria; in questo caso occorre almeno un frame per l'istruzione e uno per il riferimento alla memoria. Inoltre se è ammesso un indirizzamento indiretto a un livello (come nel caso di un'istruzione `load` presente nella pagina 16 che può far riferimento a un indirizzo della pagina 0, che costituisce a sua volta un riferimento indiretto alla pagina 23) la paginazione richiede allora almeno tre blocchi di memoria per ogni processo. Si consideri che cosa accadrebbe nel caso di un processo che disponga di due soli frame.

Il numero minimo di frame è definito dall'hardware del calcolatore. Ad esempio, L'istruzione di spostamento del PDP-11, per alcune modalità di indirizzamento, è costituita di più di una parola, quindi la stessa istruzione può stare a cavallo tra due pagine. Inoltre, ciascuno dei suoi due operandi può essere un riferimento indiretto, per un totale di sei frame. Un altro esempio è dato dall'istruzione `MVC` di IBM 370. Poiché l'istruzione è da memoria a memoria, può occupare 6 byte e stare a cavallo tra due pagine. Anche la sequenza di caratteri da spostare e l'area su cui effettuare lo spostamento possono essere a cavallo tra due pagine; questa situazione richiede sei frame. In effetti, la situazione peggiore si presenta quando l'istruzione `MVC` è l'operando di un'istruzione `EXECUTE` che sta a cavallo di un limite di pagina; in questo caso occorrono otto frame.

Il caso peggiore si può presentare nelle architetture di calcolatori che permettono riferimenti indiretti a più livelli (ad esempio quando ogni parola di 16 bit può contenere un indirizzo di 15 bit più un indicatore indiretto di 1 bit). In teoria, una semplice istruzione di caricamento può far riferimento a un indirizzo indiretto che a sua volta può far riferimento a un indirizzo indiretto (su un'altra pagina) anch'esso facente riferimento a un indirizzo indiretto su un'altra pagina ancora, e così via, finché tutte le pagine della memoria virtuale siano state chiamate in causa. Quindi, nel caso peggiore, tutta la memoria virtuale si deve trovare in memoria fisica. Per superare questa difficoltà occorre porre un limite al livello dei riferimenti indiretti, ad esempio limitando un'istruzione a un massimo di 16 livelli. Quando si verifica il riferimento indiretto di primo livello, si imposta un contatore al valore 16, per decrementarlo a ciascun livello successivo relativo a questa istruzione. Se il contatore si riduce a 0 si verifica un segnale di eccezione (livello di riferimenti indiretti eccessivo). Tale limite riduce a 17 il numero massimo dei riferimenti alla memoria per ogni istruzione, richiedendo un pari numero di frame.

Il numero minimo di frame per ciascun processo è definito dall'architettura, mentre il numero massimo è definito dalla quantità di memoria fisica disponibile. Rimane ancora aperta la questione della scelta dell'allocazione dei frame.

## 9.5.2 Algoritmi di allocazione

Il modo più semplice per suddividere $m$ frame tra $n$ processi è quello per cui a ciascuno si dà una parte uguale, $m/n$ frame. Dati 93 frame e cinque processi, ogni processo riceve 18 frame. I tre frame lasciati liberi si potrebbero usare come gruppo dei frame liberi. Questo schema è chiamato **allocazione uniforme**.

Un'alternativa consiste nel riconoscere che diversi processi hanno bisogno di quantità di memoria diverse. Si consideri un sistema con frame di 1 KB. Se un piccolo processo utente di 10 KB e una base di dati interattiva di 127 KB sono gli unici due processi in esecuzione su un sistema con 62 frame liberi, non ha senso allocare a ciascun processo 31 frame. Al processo utente non ne servono più di 10, quindi gli altri 21 sarebbero semplicemente sprecati.

Per risolvere questo problema è possibile ricorrere all'**allocazione proporzionale**, secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione. Si supponga che $s_i$ sia la dimensione della memoria virtuale per il processo $p_i$. Si definisce la seguente quantità:

$$S = \Sigma s_i.$$

Quindi, se il numero totale dei frame disponibili è $m$, al processo $p_i$ si assegnano $a_i$ frame, dove $a_i$ è approssimativamente

$$a_i = s_i/S \times m.$$

Naturalmente è necessario scegliere ciascun $a_i$ in modo che sia un intero maggiore del numero minimo di frame richiesti dalla struttura della serie di istruzioni di macchina e in modo che la somma di tutti gli $a_i$ non sia maggiore di $m$.

Usando l'allocazione proporzionale, per suddividere 62 frame tra due processi, uno di 10 e uno di 127 pagine, si assegnano rispettivamente 4 e 57 frame, infatti:

$$10/137 \times 62 \approx 4$$
$$127/137 \times 62 \approx 57.$$

In questo modo entrambi i processi condividono i frame disponibili secondo le rispettive necessità, e non in modo uniforme.

Sia nell'allocazione uniforme sia in quella proporzionale, l'allocazione a ogni processo può variare rispetto al livello di multiprogrammazione. Se tale livello aumenta, ciascun processo perde alcuni frame per fornire la memoria necessaria per il nuovo processo. D'altra parte, se il livello di multiprogrammazione diminuisce, i frame allocati al processo allontanato si possono distribuire tra quelli che restano.

Occorre notare che sia con l'allocazione uniforme sia con l'allocazione proporzionale, un processo a priorità elevata è trattato come un processo a bassa priorità anche se, per definizione, si vorrebbe che al processo con elevata priorità fosse allocata più memoria per accelerarne l'esecuzione, a discapito dei processi a bassa priorità. Un soluzione prevede l'uso di uno schema di allocazione proporzionale in cui il rapporto dei frame non dipende dalle dimensioni relative dei processi, ma dalle priorità degli stessi oppure da una combinazione di dimensioni e priorità.

### 9.5.3 Allocazione globale e allocazione locale

Un altro importante fattore che riguarda il modo in cui si assegnano i frame ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione delle pagine si possono classificare in due categorie generali: **sostituzione globale** e **sostituzione locale**. La sostituzione globale permette che per un processo si scelga un frame per la sostituzione dall'insieme di tutti i frame, anche se quel frame è correntemente allocato a un altro processo; un processo può dunque sottrarre un frame a un altro processo. La sostituzione locale richiede invece che per ogni processo si scelga un frame solo dal proprio insieme di frame.

Si consideri ad esempio uno schema di allocazione che, per una sostituzione a favore dei processi ad alta priorità, permetta di sottrarre frame ai processi a bassa priorità. Per un processo si può stabilire una sostituzione che attinga tra i suoi frame oppure tra quelli di qualsiasi processo con priorità minore. Questo metodo permette a un processo ad alta priorità di aumentare il proprio livello di allocazione dei frame a discapito del processo a bassa priorità.

Con la strategia di sostituzione locale, il numero di blocchi di memoria assegnati a un processo non cambia. Con la sostituzione globale, invece, può accadere che per un certo processo si selezionino solo frame allocati ad altri processi, aumentando così il numero di frame assegnati a quel processo, purché per altri non si scelgano per la sostituzione i *propri* frame.

L'algoritmo di sostituzione globale risente di un problema: un processo non può controllare la propria frequenza di assenze di pagine (*page-fault rate*), infatti l'insieme di pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi. Quindi, lo stesso processo può comportarsi in modi piuttosto diversi, ad esempio impiegando 0,5 secondi per un'esecuzione e 10,3 secondi per quella successiva, a causa di circostanze esterne. Con l'algoritmo di sostituzione locale questo problema non si presenta. Infatti l'insieme di pagine in memoria per un processo subisce l'effetto del comportamento di paginazione di quel solo processo. Dal canto suo, la sostituzione locale può limitare un processo, non rendendogli disponibili altre pagine di memoria meno usate. Generalmente, la sostituzione globale genera una maggiore produttività del sistema, e perciò è il metodo più usato.

### 9.5.4 Accesso non uniforme alla memoria

Fino a questo momento trattando il tema della memoria virtuale abbiamo assunto che le diverse parti della memoria centrale funzionassero tutte allo stesso modo, o almeno che vi si potesse accedere con le stesse modalità. In molti sistemi informatici non è così. Spesso in sistemi con processori multipli (Paragrafo 1.3.2) un certo processore può accedere ad alcune regioni della memoria più rapidamente rispetto ad altre. Tali differenze nelle prestazioni sono causate dalla modalità di interconnessione tra processori e memoria all'interno del sistema. Frequentemente un tale sistema è costituito da diverse schede madri, ognuna contenente più processori e una parte di memoria. Le schede sono connesse in vari modi, a partire dai bus di sistema fino a connessioni di rete ad alta velocità come InfiniBand. Come forse ci si aspetta, i processori di una particolare scheda possono accedere alla memoria della scheda stessa in meno tempo rispetto a quello necessario per accedere ad altre schede del sistema. I sistemi nei quali i tempi di accesso alla memoria variano in modo significativo sono generalmente detti **sistemi con accesso non uniforme alla memoria** (*non-uniform memory access*, NUMA) e, senza eccezioni, sono più lenti dei sistemi nei quali memoria e processori risiedono sulla stessa scheda madre.

Le decisioni su quali frame di pagina memorizzare e dove memorizzarli possono condizionare in modo significativo le prestazioni nei sistemi NUMA. Se, in sistemi del genere,

ignorassimo le diversità nei tempi di accesso alla memoria, i processori potrebbero dover aspettare molto più a lungo per accedere alla memoria rispetto al caso in cui gli algoritmi di allocazione della memoria siano modificati per tenere in conto il NUMA. Analoghe modifiche devono essere apportate anche al sistema di scheduling. L'obiettivo di questi cambiamenti è quello di allocare i frame di memoria "il più vicino possibile" al processore sul quale il processo è in esecuzione, dove per "vicino" si intende "con latenza minima", ovvero, di solito, sulla stessa scheda della CPU.

I cambiamenti negli algoritmi consistono nel fatto che lo scheduler tiene traccia dell'ultimo processore sul quale ciascun processo è stato eseguito. Se lo scheduler prova a pianificare ciascun processo sul suo processore precedente, e se il sistema di gestione della memoria prova ad allocare frame per il processo vicino al processore sul quale sta per essere mandato in esecuzione, si otterrà un incremento dei successi di cache e una diminuzione del tempo di accesso alla memoria.

La questione diventa ancora più complicata con l'aggiunta dei thread. Ad esempio, un processo con molti thread in esecuzione potrebbe vedere quei thread pianificati su differenti schede del sistema. Come viene allocata la memoria in questo caso? Solaris risolve il problema creando una entità **lgroup** nel kernel. Ogni lgroup raccoglie i processori e la memoria vicini fra loro. In realtà gli lgroup sono ordinati gerarchicamente sulla base del periodo di latenza tra i gruppi. Solaris tenta di pianificare tutti i thread di un processo e di allocare tutta la memoria di un processo nell'ambito di un solo lgroup. Se una tale soluzione non è possibile, per il resto delle risorse necessarie vengono utilizzati gli lgroup più vicini, in modo da minimizzare la latenza complessiva della memoria e massimizzare il grado di successo della cache del processore.

## 9.6 Paginazione degenere (thrashing)

Se il numero dei frame allocati a un processo con priorità bassa diviene inferiore al numero minimo richiesto dall'architettura del calcolatore, occorre sospendere l'esecuzione del processo, e quindi togliere le pagine restanti, liberando tutti i frame allocati. Questa operazione introduce un livello intermedio di scheduling per la gestione dell'entrata e dell'uscita dei processi in memoria centrale.

Infatti, si consideri un qualsiasi processo che non disponga di un numero di frame "sufficiente". Anche se tecnicamente si può ridurre al valore minimo il numero dei frame allocati, esiste un certo (in generale grande) numero di pagine in uso attivo. Se non dispone di questo numero di frame, il processo accusa immediatamente un'assenza di pagina. A questo punto si deve sostituire qualche pagina; ma, poiché tutte le sue pagine sono in uso attivo, si deve sostituire una pagina che sarà immediatamente necessaria, e di conseguenza si verificano subito parecchie assenze di pagine. Il processo continua a subire assenze di pagine, facendo sostituire pagine che saranno immediatamente trattate come assenti e dovranno essere riprese.

Questa intensa quanto degenere paginazione (nota come *thrashing*) si verifica quando si spende più tempo per la paginazione che per l'esecuzione dei processi.

### 9.6.1 Cause della paginazione degenere

La degenerazione dell'attività di paginazione causa parecchi problemi di prestazioni. Si consideri il seguente scenario, basato sul comportamento effettivo dei primi sistemi di paginazione.

Il sistema operativo vigila sull'utilizzo della CPU. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione

delle pagine globale, che sostituisce le pagine senza tener conto del processo al quale appartengono. Per ora si ipotizzi che un processo entri in una nuova fase d'esecuzione e richieda più frame; se ciò si verifica si ha una serie di assenze di pagine, cui segue la sottrazione di nuove pagine ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch'essi delle assenze di pagine, con conseguente sottrazione di pagine ad altri processi. Per effettuare il caricamento e lo scaricamento delle pagine per questi processi si deve usare il dispositivo di paginazione. Mentre si mettono i processi in coda per il dispositivo di paginazione, la coda dei processi pronti per l'esecuzione si svuota, quindi l'utilizzo della CPU diminuisce.

Lo scheduler della CPU rileva questa riduzione dell'utilizzo della CPU e *aumenta* il grado di multiprogrammazione. Si tenta di avviare il nuovo processo sottraendo pagine ai processi in esecuzione, causando ulteriori assenze di pagine e allungando la coda per il dispositivo di paginazione. L'utilizzo della CPU scende ulteriormente, e lo scheduler della CPU tenta di aumentare ancora il grado di multiprogrammazione. L'attività di paginazione è degenerata in una situazione patologica che fa precipitare la produttività del sistema. La frequenza delle assenze di pagine aumenta in modo impressionante, e di conseguenza aumenta il tempo effettivo d'accesso alla memoria. I processi non svolgono alcun lavoro, poiché si sta spendendo tutto il tempo nell'attività di paginazione.

Questo fenomeno è illustrato nella Figura 9.18, in cui si riporta l'utilizzo della CPU in funzione del grado di multiprogrammazione. Aumentando il grado di multiprogrammazione aumenta anche l'utilizzo della CPU, anche se più lentamente, fino a raggiungere un massimo. Se a questo punto si aumenta ulteriormente il grado di multiprogrammazione, l'attività di paginazione degenera e fa crollare l'utilizzo della CPU. In questa situazione, per aumentare l'utilizzo della CPU occorre *ridurre* il grado di multiprogrammazione.

Gli effetti di questa situazione si possono limitare usando un **algoritmo di sostituzione locale**, o **algoritmo di sostituzione per priorità**. Con la sostituzione locale, se un processo ricade nell'attività di paginazione degenere, non può sottrarre frame a un altro processo e quindi provocarne a sua volta la degenerazione. Le pagine si sostituiscono tenendo conto del processo di cui fanno parte. Tuttavia, se i processi la cui attività di paginazione degenera rimangono nella coda d'attesa del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio di un'eccezione di pagina mancante aumenta a causa dell'allungamento medio della coda d'attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo d'accesso al dispositivo di paginazione aumenta anche per gli altri processi.

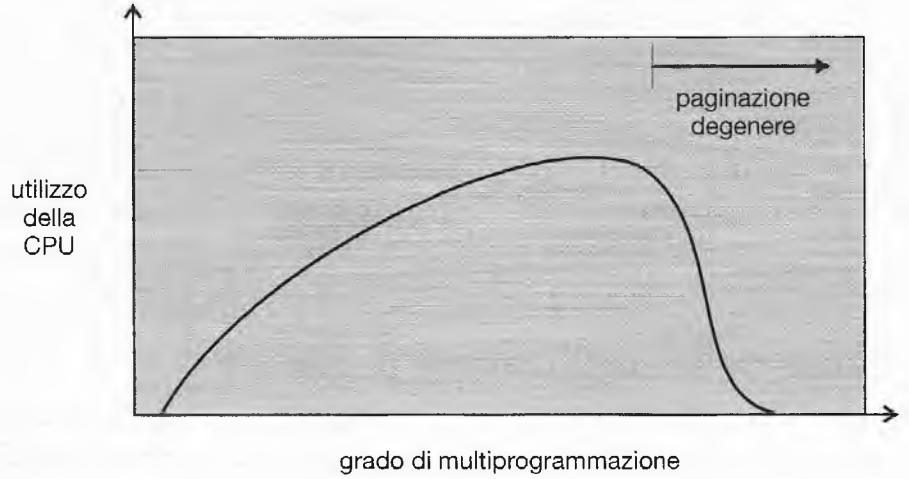



**Figura 9.18** Paginazione degenere (*thrashing*).

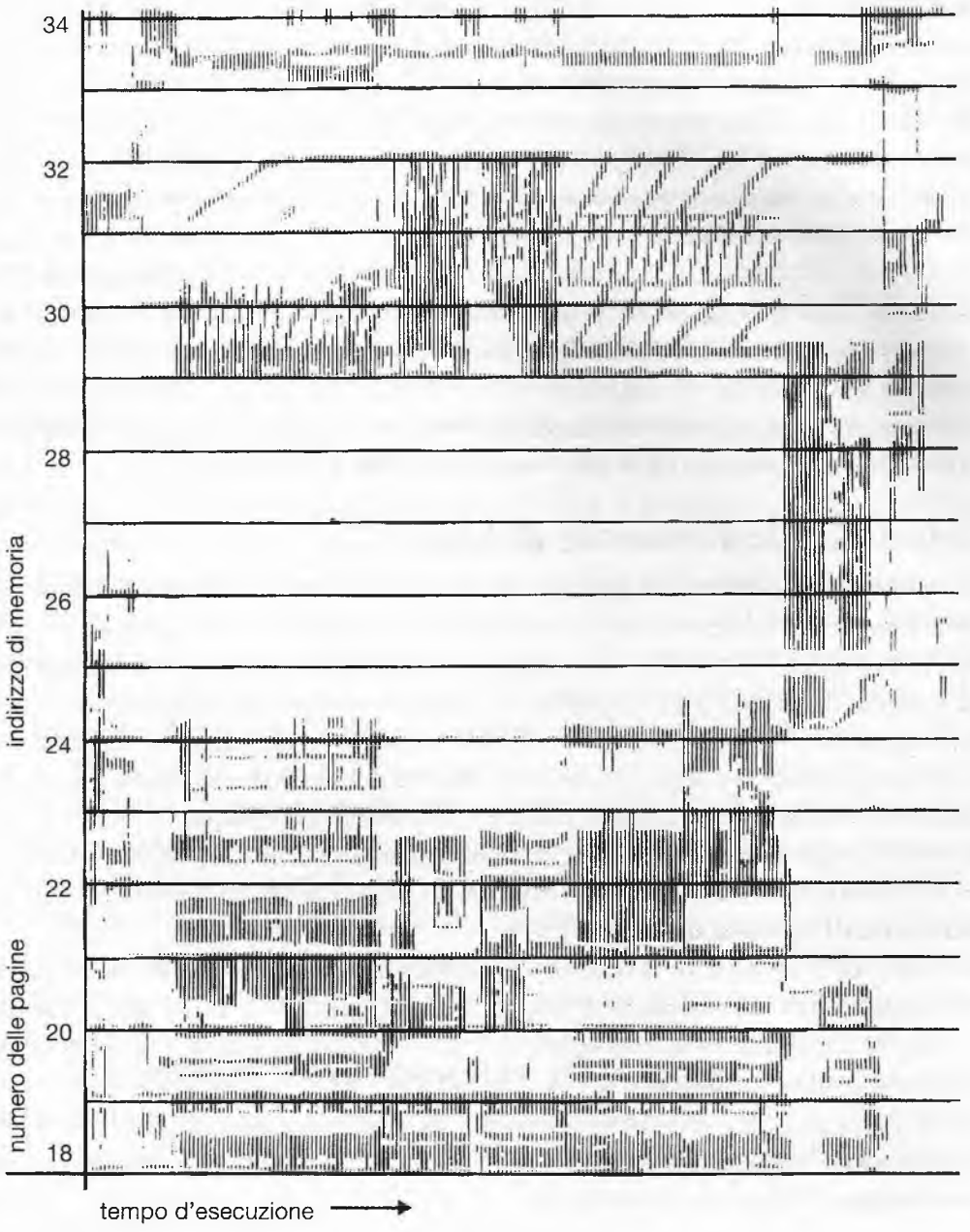



**Figura 9.19** Località dei riferimenti alla memoria.

Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame "servano" a un processo si impiegano diverse tecniche. Il modello dell'insieme di lavoro (*working-set*), trattato nel Paragrafo 9.6.2, comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo metodo definisce il **modello di località** d'esecuzione del processo.

Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine usate attivamente, com'è illustrato nella Figura 9.19. Generalmente un programma è formato di parecchie località diverse, che sono sovrapponibili.

Ad esempio, quando s'invoca una procedura, essa definisce una nuova località. In questa località si fanno riferimenti alla memoria per le istruzioni della procedura, per le sue variabili locali e per un sottoinsieme delle variabili globali. Quando la procedura termina, il

processo lascia questa località, poiché le variabili locali e le istruzioni della procedura non sono più usate attivamente. Ritorneremo più avanti al concetto di località.

Quindi, le località sono definite dalla struttura del programma e dalle relative strutture dati. Il modello di località sostiene che tutti i programmi mostrino questa struttura di base di riferimenti alla memoria. Si noti che il modello di località è il principio non dichiarato sottostante all'analisi fin qui svolta sul caching. Se gli accessi ai vari tipi di dati fossero casuali, anziché strutturati in località, il caching sarebbe inutile.

Si supponga di allocare a un processo un numero di frame sufficiente per sistemare le sue località attuali. Finché tutte queste pagine non si trovano in memoria, si verificano le assenze delle pagine relative a tali località; quindi, finché le località non vengano modificate, non hanno luogo altre assenze di pagine. Se si assegnano meno frame rispetto alla dimensione della località attuale, la paginazione del processo degenera, poiché non si possono tenere in memoria tutte le pagine che il processo sta usando attivamente.

## 9.6.2 Modello dell'insieme di lavoro

Come già accennato, il **modello dell'insieme di lavoro** (*working-set model*) è basato sull'ipotesi di località. Questo modello usa un parametro, $\Delta$, per definire la **finestra dell'insieme di lavoro**. L'idea consiste nell'esaminare i più recenti $\Delta$ riferimenti alle pagine. L'insieme di pagine nei più recenti $\Delta$ riferimenti è l'*insieme di lavoro*; si veda a questo proposito la Figura 9.20. Se una pagina è in uso attivo si trova nell'insieme di lavoro; se non è più usata esce dall'insieme di lavoro $\Delta$ unità di tempo dopo il suo ultimo riferimento. Quindi, l'insieme di lavoro non è altro che un'approssimazione della località del programma.

Ad esempio, data la successione di riferimenti alla memoria mostrata nella Figura 9.20, se $\Delta = 10$ riferimenti alla memoria, l'insieme di lavoro all'istante $t_1$ è {1, 2, 5, 6, 7}. All'istante $t_2$ l'insieme di lavoro è diventato {3, 4}.

La precisione dell'insieme di lavoro dipende dalla scelta del valore di $\Delta$. Se $\Delta$ è troppo piccolo non include l'intera località, se è troppo grande può sovrapporre più località. Al limite, se $\Delta$ è infinito l'insieme di lavoro coincide con l'insieme di pagine cui il processo fa riferimento durante la sua esecuzione.

La caratteristica più importante dell'insieme di lavoro è la sua dimensione. Calcolando la dimensione dell'insieme di lavoro, $WSS_i$, per ciascun processo $p_i$ del sistema, si può determinare la richiesta totale di frame, cioè $D$:

$$D = \Sigma\, WSS_i.$$

Ogni processo usa attivamente le pagine del proprio insieme di lavoro. Quindi, il processo $i$ necessita di $WSS_i$ frame. Se la richiesta totale è maggiore del numero totale di frame liberi ($D > m$), la paginazione degenera, poiché alcuni processi non dispongono di un numero sufficiente di frame.

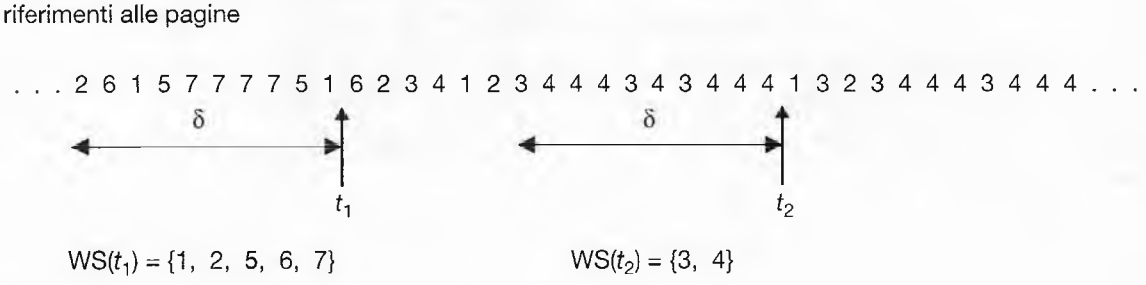



**Figura 9.20** Modello dell'insieme di lavoro.

Una volta scelto $D$, l'uso del modello dell'insieme di lavoro è abbastanza semplice. Il sistema operativo controlla l'insieme di lavoro di ogni processo e gli assegna un numero di frame sufficiente, rispetto alle dimensioni del suo insieme di lavoro. Se i frame ancora liberi sono in numero sufficiente, si può iniziare un altro processo. Se la somma delle dimensioni degli insiemi di lavoro aumenta, superando il numero totale dei frame disponibili, il sistema operativo individua un processo da sospendere. Scrive in memoria secondaria le pagine di quel processo e assegna i propri frame ad altri processi. Il processo sospeso può essere ripreso successivamente.

Questa strategia impedisce la paginazione degenere, mantenendo il grado di multiprogrammazione più alto possibile, quindi ottimizza l'utilizzo della CPU.

Poiché la finestra dell'insieme di lavoro è una finestra dinamica, la difficoltà insita in questo modello consiste nel tener traccia degli elementi che compongono l'insieme di lavoro stesso. A ogni riferimento alla memoria, a un'estremità appare un riferimento nuovo e il riferimento più vecchio fuoriesce dall'altra estremità. Una pagina si trova nell'insieme di lavoro se esiste un riferimento a essa in qualsiasi punto della finestra dell'insieme di lavoro.

Si supponga, ad esempio, che $\Delta$ sia pari a 10.000 riferimenti e che sia possibile ottenere un segnale d'interruzione dal timer ogni 5000 riferimenti. Quando si verifica uno di tali segnali d'interruzione, i valori dei bit di riferimento di ciascuna pagina vengono copiati e poi azzerati. Così, quando si verifica un'assenza di pagina è possibile esaminare il bit di riferimento corrente e i 2 bit in memoria per stabilire se una pagina sia stata usata entro gli ultimi 10.000-15.000 riferimenti. Se lo è stata, almeno uno di questi bit è attivo. Se non lo è stata, questi bit sono tutti inattivi. Le pagine con almeno un bit attivo si considerano appartenenti all'insieme di lavoro. Occorre notare che questo schema non è del tutto preciso, poiché non è possibile stabilire dove si è verificato un riferimento entro un intervallo di 5000. L'incertezza si può ridurre aumentando il numero dei bit cronologici e la frequenza dei segnali d'interruzione, ad esempio, 10 bit e un'interruzione ogni 1000 riferimenti. Tuttavia, il costo per servire questi segnali d'interruzione più frequenti aumenta in modo corrispondente.

### 9.6.3 Frequenza delle assenze di pagine

Il modello dell'insieme di lavoro riscuote un discreto successo, e la sua conoscenza può servire per la prepaginazione (Paragrafo 9.9.1), ma appare un modo alquanto goffo per controllare la degenerazione della paginazione. La strategia basata sulla **frequenza delle assenze di pagine** (*page fault frequency*, PFF) è più diretta.

Il problema specifico è la prevenzione della paginazione degenere. La frequenza delle assenze di pagine in tale situazione è alta, ed è proprio questa che si deve controllare. Se la frequenza delle assenze di pagine è eccessiva, significa che il processo necessita di più frame. Analogamente, se la frequenza delle assenze di pagine è molto bassa, il processo potrebbe disporre di troppi frame. Si può fissare un limite inferiore e un limite superiore per la frequenza desiderata delle assenze di pagine, com'è illustrato nella Figura 9.21. Se la frequenza effettiva delle assenze di pagine per un processo oltrepassa il limite superiore, occorre allocare a quel processo un altro frame; se la frequenza scende sotto il limite inferiore, si sottrae un frame a quel processo. Quindi, per prevenire la paginazione degenere, si può misurare e controllare direttamente la frequenza delle assenze di pagine.

Come nel caso dell'insieme di lavoro, può essere necessaria la sospensione di un processo. Se la frequenza delle assenze di pagine aumenta e non ci sono frame disponibili, occorre selezionare un processo e sospenderlo. I frame liberati si distribuiscono ai processi con elevate frequenze di assenze di pagine.

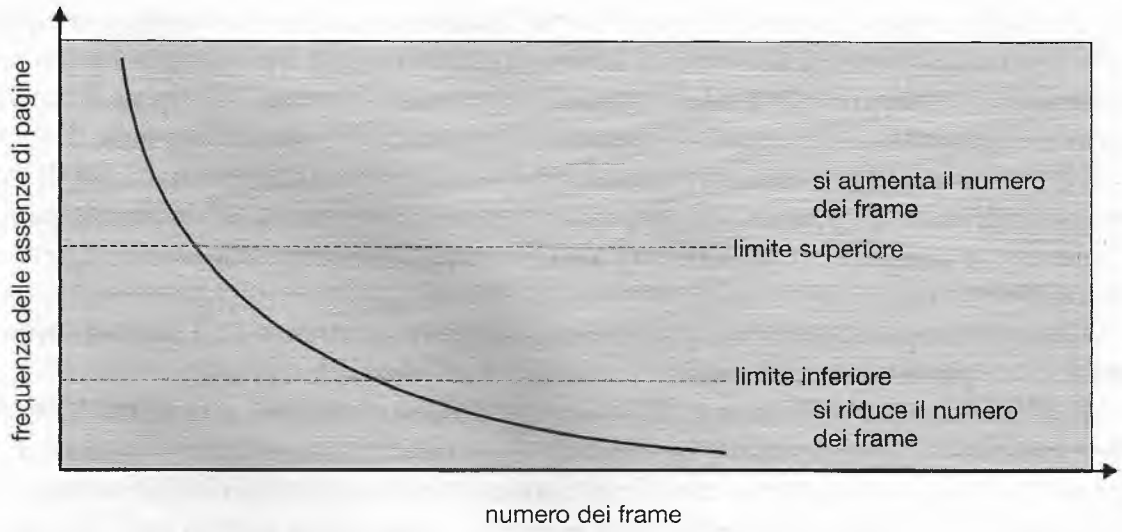



**Figura 9.21** Frequenza delle assenze di pagine.

## INSIEME DI LAVORO E FREQUENZA DELLE PAGINE MANCANTI

Vi è una relazione diretta tra l'insieme di lavoro di un processo e la frequenza di pagine mancanti. Come mostra la Figura 9.20, l'insieme di lavoro di un processo cambia nel corso del tempo, mentre i riferimenti ai dati e alcune parti del codice sono dislocati dall'una all'altra posizione.

Assumendo memoria sufficiente a contenere l'insieme di lavoro di un processo (ossia, a evitare che la paginazione di un processo degeneri), l'andamento delle pagine mancanti oscillerà, in un dato periodo di tempo, tra picchi e valli. Questa tendenza generale è illustrata dalla Figura 9.22.

Un picco nella frequenza di pagine mancanti si verifica alla richiesta di paginazione relativa a una nuova località. Tuttavia, una volta che l'insieme di lavoro interessato sia in memoria, la frequenza di pagine mancanti precipita. Quando il processo entra in un nuovo insieme di lavoro, la frequenza si impenna ancora una volta verso un picco; quando il nuovo insieme di lavoro è caricato in memoria, la frequenza crolla nuovamente. L'intervallo di tempo tra l'inizio di un picco e quello del picco successivo descrive la transizione da un insieme di lavoro a un altro.

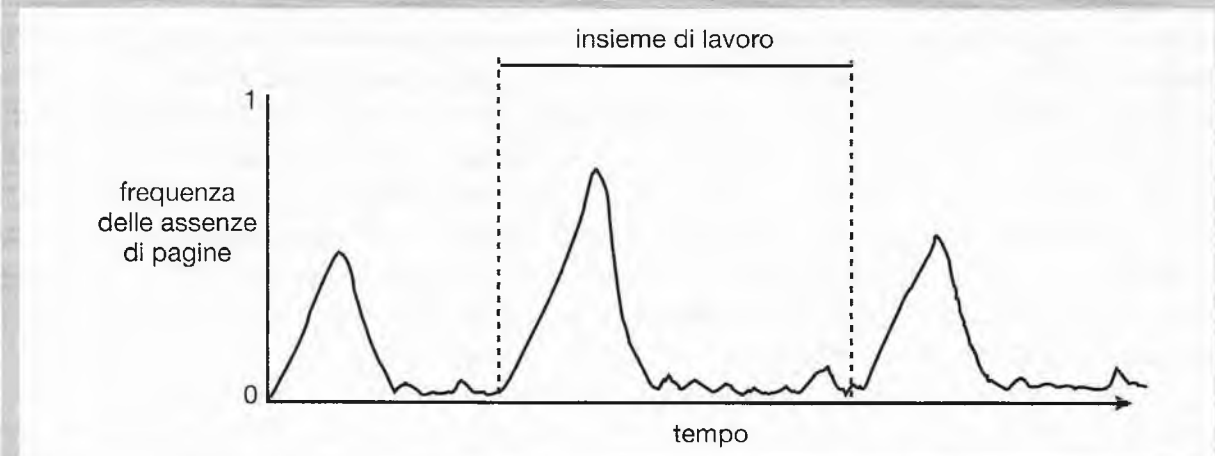



**Figura 9.22** Frequenza delle assenze di pagine nel periodo di tempo considerato.

# 9.7 File mappati in memoria

Si consideri la lettura sequenziale di un file sul disco per mezzo delle consuete chiamate di sistema: `open()`, `read()` e `write()`. Ciascun accesso al file richiede una chiamata di sistema e un accesso al disco. In alternativa, possiamo avvalerci delle tecniche di memoria virtuale analizzate sin qui per equiparare l'I/O dei file all'accesso ordinario alla memoria. Grazie a questa soluzione, nota come **mappatura dei file in memoria**, una parte dello spazio degli indirizzi virtuali può essere associata logicamente al file. Come vedremo, ciò comporta un incremento significativo delle prestazioni in fase di I/O.

## 9.7.1 Meccanismo di base

La mappatura di un file in memoria si realizza associando un blocco del disco a una o più pagine residenti in memoria. L'accesso iniziale al file avviene tramite una normale richiesta di paginazione, che causa un errore di pagina mancante. Tuttavia, una porzione del file, che è pari a una pagina, è caricata dal file system in una pagina fisica (alcuni sistemi possono decidere di caricare porzioni più grandi di memoria). Ogni successiva lettura e scrittura del file è gestita come accesso ordinario alla memoria, semplificando così l'accesso al file e il suo utilizzo, in quanto si permette al sistema di manipolare i file attraverso la memoria anziché appesantire il sistema stesso con le chiamate `read()` e `write()`. Allo stesso modo, dal momento che il file I/O è creato nella memoria (invece di utilizzare chiamate di sistema che fanno uso di I/O da disco) anche l'accesso al file è molto più veloce.

Si osservi come le scritture sul file mappato in memoria non si traducano necessariamente e immediatamente in scritture sincrone sul file del disco. Alcuni sistemi scelgono di aggiornare il file fisico quando il sistema operativo esegue un controllo periodico per l'accertamento di eventuali modifiche alle pagine. Quando il file viene chiuso, tutti i dati mappati in memoria sono scritti nuovamente su disco e rimossi dalla memoria virtuale del processo.

Alcuni sistemi operativi prevedono un'apposita chiamata di sistema per la mappatura dei dati; le chiamate ordinarie sono riservate a tutte le altre operazioni di I/O. Altri sistemi possono mappare un file in memoria, anche in assenza di un'esplicita richiesta in tal senso. Prendiamo per esempio Solaris. Se si dichiara che un file deve essere mappato in memoria tramite la chiamata di sistema `mmap()`, Solaris opera la mappatura del file nello spazio degli indirizzi del processo. Se si apre un file, e vi si accede con le chiamate di sistema ordinarie, quali `open()`, `read()` e `write()`, Solaris continua a mappare in memoria il file; tuttavia, il file è mappato nello spazio degli indirizzi del kernel. A prescindere da come si apre il file, dunque, Solaris considera tutto l'I/O relativo ai file come mappato in memoria, sfruttando per l'acceso ai file l'efficiente sottosistema della memoria.

Per consentire la condivisione dei dati, più processi possono essere autorizzati a mappare contemporaneamente un file in memoria. Quanto scritto da uno di questi processi modifica i dati nella memoria virtuale e risulta visibile a tutti gli altri processi che pure mappano il file. L'analisi sulla memoria virtuale svolta fin qui dovrebbe aver chiarito come la condivisione delle sezioni di memoria interessate dalla mappatura abbia luogo: la memoria virtuale di ciascun processo che partecipa alla condivisione punta alla stessa pagina della memoria fisica – la pagina che ospita una copia del blocco del disco. Tale situazione è illustrata nella Figura 9.23. Le chiamate di sistema per la mappatura in memoria possono inoltre offrire la funzionalità della copia su scrittura, che consente ai processi di condividere un file in modalità di sola lettura trattenendo una copia dei dati che modificano. Affinché l'accesso ai

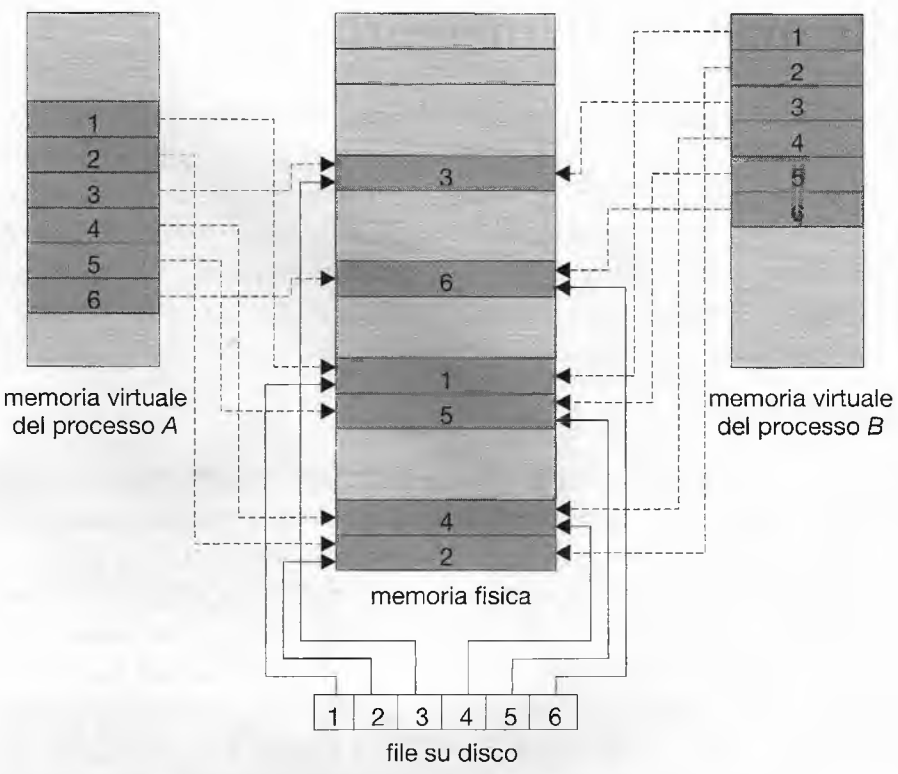



**Figura 9.23** File mappati in memoria.

dati condivisi sia coordinato, i processi interessati potrebbero usare uno dei meccanismi per la mutua esclusione, descritti nel Capitolo 6.

Per molti versi, la condivisione dei file mappati in memoria mostra analogie con la memoria condivisa, trattata nel Paragrafo 3.4.1. Non tutti i sistemi adottano il medesimo meccanismo in entrambi i casi; UNIX e Linux, per esempio, gestiscono la mappatura della memoria con la chiamata di sistema `mmap()`, mentre, per la memoria condivisa, ricorrono alle chiamate `shmget()` e `shmat()` dello standard POSIX (Paragrafo 3.5.1). Nei sistemi Windows NT, 2000 e XP, però, la condivisione della memoria trova concreta applicazione mediante la mappatura dei file in memoria. La comunicazione fra processi si ottiene in questi sistemi mappando in memoria uno stesso file negli spazi degli indirizzi virtuali dei processi coinvolti. Il file mappato in memoria funge da area di memoria condivisa tra i processi comunicanti (Figura 9.24). Nel paragrafo successivo vedremo come la API Win32 fornisca gli strumenti per condividere memoria tramite mappatura dei file in memoria.

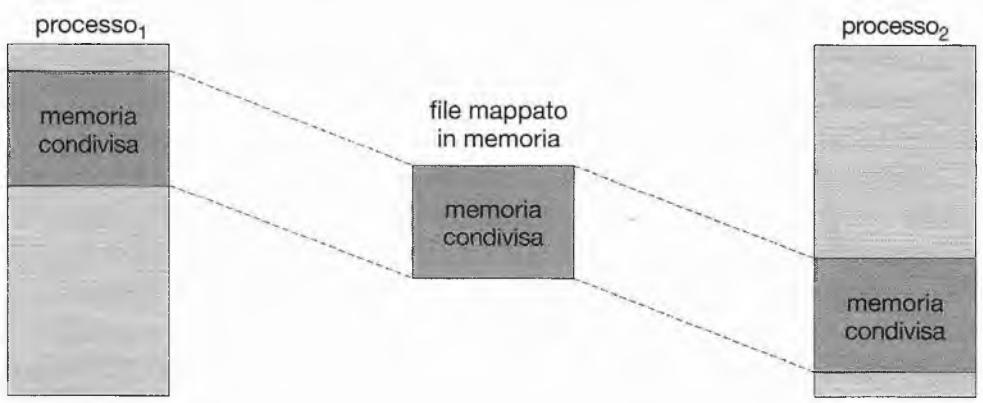



**Figura 9.24** Condivisione della memoria in Windows tramite I/O mappato in memoria.

### 9.7.2 Memoria condivisa nella API Win32

La prassi generale per la configurazione di una regione di memoria condivisa nella API Win32 consiste, dapprima, nel **mappare il file** interessato dall'operazione, per poi stabilire una vista (*view*) del file mappato nello spazio degli indirizzi virtuali di un processo. Un secondo processo, a questo punto, può aprire e creare una sua vista del file mappato nel proprio spazio degli indirizzi virtuali. Il file mappato è l'oggetto condiviso tramite il quale può svolgersi la comunicazione tra i processi.

Analizziamo ora queste fasi più da vicino. Nel nostro esempio, il processo produttore crea dapprima un oggetto condiviso, sfruttando le funzionalità per la mappatura in memoria disponibili nella API Win32. Il produttore scrive quindi un messaggio nella memoria condivisa. Il processo consumatore accede a sua volta alla mappatura del file, e legge il messaggio scritto dal produttore.

Per costruire un file mappato in memoria, il processo apre in primo luogo il file da mappare con la funzione `CreateFile()`, che restituisce un riferimento `HANDLE` al file. Il processo allora crea una mappatura di questo riferimento, usando la funzione `CreateFileMapping()`. Una volta stabilita la mappatura del file, il processo genera nel proprio spazio degli indirizzi virtuali una vista del file mappato, con la funzione `MapViewOfFile()`. La vista costituisce la porzione di file che risiederà nello spazio degli indirizzi virtuali del processo; il file può essere mappato solo in parte o per intero. Il programma mostrato nella Figura 9.25 illustra questa procedura (molti controlli sugli errori sono stati omessi per esigenze di brevità del codice).

L'invocazione a `CreateFileMapping()` genera un **oggetto condiviso con nome**, chiamato `SharedObject`. Il processo consumatore utilizzerà questo segmento di memoria condivisa per comunicare, creando una mappatura del medesimo oggetto con nome. Il produttore, quindi, crea nel proprio spazio degli indirizzi virtuali una vista del file mappato in memoria. Passando agli ultimi tre parametri il valore 0, si richiede che la porzione mappata sia l'intero file; si potrebbero tuttavia anche passare valori che specifichino l'inizio e la dimensione della porzione da mappare. (Si osservi, a questo proposito, che non è detto che l'intero file sia caricato in memoria al momento in cui se ne richiede la mappatura: è possibile che il caricamento avvenga tramite paginazione su richiesta, trasferendo perciò di volta in volta le pagine a cui si accede.) La funzione `MapViewOfFile()` restituisce un puntatore all'oggetto condiviso; ogni accesso a questa locazione di memoria è dunque un accesso al file mappato in memoria. In questo caso, nella memoria condivisa il processo produttore scriverà il messaggio `"Messaggio nella memoria condivisa"`.

La Figura 9.26 contiene un programma che dimostra come il processo consumatore stabilisca una vista dell'oggetto condiviso con nome. Il codice è un po' più semplice di quello della Figura 9.25, poiché il processo non ha che da mappare l'oggetto condiviso con nome già esistente. Come il processo produttore, anche il processo consumatore deve creare una vista del file mappato. Il consumatore, quindi, legge dalla memoria condivisa il messaggio scritto dal processo produttore.

Infine, entrambi i processi eliminano la vista del file mappato chiamando `UnmapViewOfFile()`. Alla fine del capitolo il lettore troverà un esercizio di programmazione per la API Win32 incentrato sulla condivisione della memoria tramite mappatura dei file.

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

int main(int argc, char *argv[])
{
  HANDLE hFile, hMapFile;
  LPVOID lpMapAddress;

  hFile = CreateFile("temp.txt", // nome del file
    GENERIC_READ | GENERIC_WRITE, // accesso in lettura/scrittura
    0, // nessuna condivisione del file
    NULL, // sicurezza di default
    OPEN_ALWAYS, // apre il file sia se esistente sia se nuovo
    FILE_ATTRIBUTE_NORMAL, // attributi del file ordinari
    NULL); // niente template del file

  hMapFile = CreateFileMapping(hFile, // riferimento al file
    NULL, // sicurezza di default
    PAGE_READWRITE, // accesso in lettura/scrittura alle pagine mappate
    0, // mappa l'intero file
    0,
    TEXT("OggettoCondiviso")); // oggetto condiviso con nome

  lpMapAddress = MapViewOfFile(hMapFile, // rif. all'oggetto mappato
    FILE_MAP_ALL_ACCESS, // accesso in lettura/scrittura
    0, // vista dell'intero file
    0,
    0);

  // scrive nella memoria condivisa
  sprintf(lpMapAddress, "Messaggio nella memoria condivisa");

  UnmapViewOfFile(lpMapAddress);
  CloseHandle(hFile);
  CloseHandle(hMapFile);
}
```

**Figura 9.25** Produttore che scrive nella memoria condivisa tramite la API Win32.

## 9.7.3 Mappatura in memoria dell'I/O

Per quanto riguarda l'I/O, come descritto nel Paragrafo 1.2.1, ogni controllore è dotato di registri contenenti istruzioni e i dati in via di trasferimento. Solitamente esistono istruzioni espressamente dedicate all'I/O che consentono il trasferimento dei dati fra questi registri e la memoria del sistema. Le architetture di molti elaboratori, per rendere più agevole l'accesso ai dispositivi dell'I/O, forniscono la **mappatura in memoria dell'I/O**. In questo caso, gli in-

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

int main(int argc, char *argv[])
{
  HANDLE hMapFile;
  LPVOID lpMapAddress;

  hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, // accesso R/W
    FALSE, // nessuna ereditarietà
    TEXT("OggettoCondiviso")); // nome dell'oggetto file mappato

  lpMapAddress = MapViewOfFile(hMapFile, // rif. all'oggetto mappato
    FILE_MAP_ALL_ACCESS, // accesso R/W
    0, // vista dell'intero file
    0,
    0);

  // lettura dalla memoria condivisa
  printf("Messaggio letto: %s", lpMapAddress);

  UnmapViewOfFile(lpMapAddress);
  CloseHandle(hMapFile);
}
```

**Figura 9.26** Consumatore che legge dalla memoria condivisa tramite la API Win32.

dirizzi di memoria compresi in certi intervalli devono essere riservati alla mappatura dei registri dei dispositivi. Le operazioni in lettura e scrittura su questi indirizzi di memoria fanno sì che i dati siano trasferiti fra i registri dei dispositivi e la memoria, o viceversa. Questo metodo è adatto a dispositivi che abbiano rapidi tempi di risposta, quali i controllori video. Nel PC IBM, a ciascuna posizione sullo schermo corrisponde una locazione mappata in memoria; visualizzare un testo sullo schermo è quasi altrettanto facile che scrivere il testo nelle relative locazioni mappate in memoria.

La mappatura in memoria dell'I/O, inoltre, è conveniente per altri dispositivi, come le porte seriali e parallele usate per connettere i modem e le stampanti all'elaboratore. La CPU invia dati tramite tali dispositivi leggendo e scrivendo alcuni registri all'interno del dispositivo, detti **porte di I/O**. Per inviare una lunga sequenza di byte attraverso una porta seriale mappata in memoria, la CPU scrive un byte nel registro dei dati e imposta un bit nel registro di controllo per indicare che il byte è disponibile. Il dispositivo preleva il byte di dati e cancella il bit del registro di controllo per segnalare che è pronto a ricevere un nuovo byte; a questo punto, la CPU può trasferire il byte successivo. Se la CPU sottopone il bit di controllo a *polling* e cioè esegue costantemente un ciclo per rilevare quando il dispositivo divenga pronto, si parla di **I/O programmato** (*programmed I/O*, PIO). Se la CPU, invece, riceve dal dispositivo un'interruzione non appena è pronto per il byte successivo, si parla di trasferimento dati **guidato dalle interruzioni**.

# 9.8 Allocazione di memoria del kernel

Quando un processo eseguito in modalità utente necessita di memoria aggiuntiva, le pagine sono allocate dalla lista dei frame disponibili che il kernel mantiene. Per formare questa lista, si applica in genere uno degli algoritmi di sostituzione delle pagine esaminati nel Paragrafo 9.4; molto verosimilmente, la lista conterrà pagine non utilizzate sparse per tutta la memoria, come abbiamo visto in precedenza. Va inoltre ricordato che, se un processo utente richiede un solo byte di memoria, si ottiene frammentazione interna, poiché al processo viene garantito un intero frame.

Il kernel, tuttavia, per allocare la propria memoria, attinge spesso a una riserva di memoria libera differente dalla lista usata per soddisfare i processi ordinari in modalità utente. Questo avviene principalmente per due motivi.

1. Il kernel richiede memoria per strutture dati dalle dimensioni variabili; alcune di loro corrispondono a meno di una pagina. Deve quindi fare un uso oculato della memoria, tentando di contenere al minimo gli sprechi dovuti alla frammentazione. Questo fattore è di particolare rilevanza, se si considera che, in molti sistemi operativi, il codice o i dati del kernel non sono soggetti a paginazione.
2. Le pagine allocate ai processi in modalità utente non devono necessariamente essere contigue nella memoria fisica. Alcuni dispositivi, però, interagiscono direttamente con la memoria fisica, senza il vantaggio dell'interfaccia della memoria virtuale; di conseguenza, possono richiedere memoria che risieda in pagine fisicamente contigue.

Nei paragrafi successivi esaminiamo due strategie per la gestione della memoria libera assegnata ai processi del kernel: il cosiddetto "sistema buddy" e l'allocazione a lastre.

## 9.8.1 Sistema buddy

Il "sistema buddy" (*sistema gemellare*) utilizza un segmento di grandezza fissa per l'allocazione della memoria, contenente pagine fisicamente contigue. La memoria è assegnata mediante un cosiddetto **allocatore-potenza-di-2**, che alloca memoria in unità di dimensioni pari a potenze di 2 (4 KB, 8 KB, 16 KB, e via di seguito). Le differenti quantità richieste sono arrotondate alla successiva potenza di 2. Ad esempio, una richiesta di 11 KB viene soddisfatta con un segmento di 16 KB.

Consideriamo un esempio semplice e ipotizziamo che la grandezza di un segmento di memoria sia inizialmente di 256 KB e che il kernel richieda 21 KB di memoria. In primo luogo il segmento è suddiviso in due *buddy* ("gemelli"), che chiameremo $A_S$ e $A_D$, ciascuno dei quali misura 128 KB. Uno di questi è ulteriormente dimezzato in 2 buddy da 64 KB, diciamo $B_S$ e $B_D$. Poiché la minima potenza di 2 che superi 21 KB è pari a 32 KB, occorre suddividere ancora $B_S$, oppure $B_D$, in due buddy la cui dimensione è 32 KB; chiamiamoli $C_S$ e $C_D$. Uno di loro è il segmento scelto per soddisfare la richiesta di 21 KB. Lo schema è illustrati nella Figura 9.27, dove $C_S$ è il segmento allocato per la richiesta di 21 KB.

Questo sistema offre il vantaggio di poter congiungere rapidamente buddy adiacenti per formare segmenti più grandi tramite una tecnica nota come **fusione** (*coalescing*). Nella Figura 9.27, per esempio, quando il kernel rilascia l'unità $C_S$ che gli era stata allocata, il sistema può fondere $C_S$ e $C_D$ in un segmento di 64 KB. Questo segmento, $B_S$, può a sua volta fondersi con il proprio gemello $B_D$, costituendo un segmento di 128 KB. Con l'ultima operazione di fusione si può ritornare al segmento originale di 256 KB.

L'ovvio inconveniente di questo sistema è che l'arrotondamento per eccesso a una potenza di 2 può facilmente generare frammentazione all'interno dei segmenti allocati. Una ri-

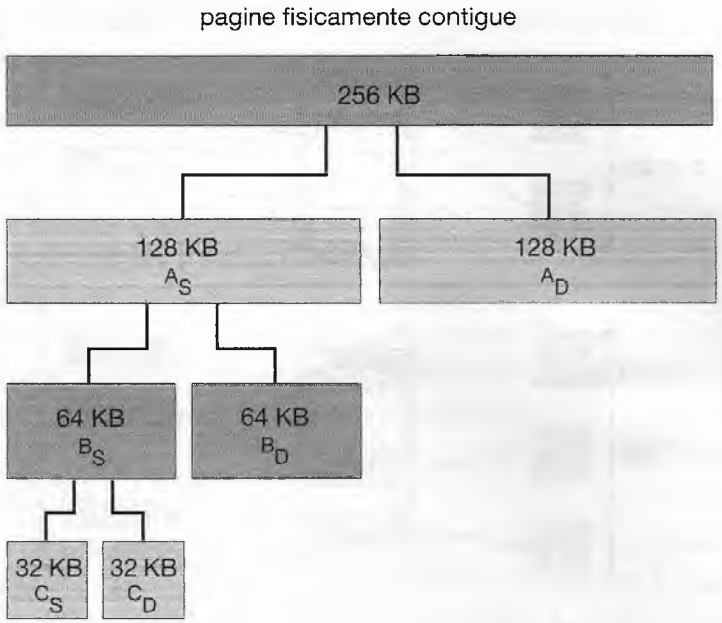



**Figura 9.27** Sistema di allocazione buddy.

chiesta di 33 KB, per esempio, può essere soddisfatta solo con un segmento di 64 KB. Proprio per effetto della frammentazione interna, dunque, risulta impossibile garantire che lo spreco dell'unità allocata resterà al di sotto del 50%. Nel paragrafo successivo presentiamo una tecnica di allocazione della memoria priva dello spreco dovuto alla frammentazione.

## 9.8.2 Allocazione a lastre

Una seconda strategia per assegnare la memoria del kernel è detta **allocazione a lastre** (*slab allocation*). Una **lastra** è composta da una o più pagine fisicamente contigue. Una **cache** consiste di una o più lastre. Vi è una sola cache per ciascuna categoria di struttura dati del kernel: una cache dedicata alla struttura dati che rappresenta i descrittori dei processi, una dedicata agli oggetti che rappresentano i file, un'altra per i semafori, e così via. Ogni cache è popolata da **oggetti**, istanze della struttura dati del kernel rappresentata dalla cache. La cache che rappresenta i semafori, per esempio, memorizza istanze di oggetti dei semafori; quella che rappresenta i descrittori dei processi memorizza istanze di descrittori dei processi, e così via. La relazione fra lastre, cache e oggetti è illustrata nella Figura 9.28; essa mostra due oggetti del kernel che misurano 3 KB e tre oggetti che misurano 7 KB, memorizzati nelle rispettive cache.

L'algoritmo di allocazione delle lastre utilizza le cache per memorizzare oggetti del kernel. Quando si crea una cache, un certo numero di oggetti, inizialmente dichiarati liberi, viene assegnato alla cache. Questo numero dipende dalla grandezza della lastra associata alla cache. Per esempio, una lastra di 12 KB (formata da tre pagine contigue di 4 KB) potrebbe contenere sei oggetti di 2 KB ciascuno. Al principio, tutti gli oggetti nella cache sono contrassegnati come **liberi**. Quando una struttura dati del kernel ha bisogno di un oggetto, per soddisfare la richiesta l'allocatore può selezionare dalla cache qualunque oggetto libero; l'oggetto tratto dalla cache è quindi contrassegnato come **usato**.

Consideriamo una situazione in cui il kernel richieda all'allocatore delle lastre la memoria per un oggetto rappresentante un descrittore dei processi. Nei sistemi Linux, un descrittore dei processi ha tipo `struct task_struct`, che richiede circa 1,7 KB di memo-

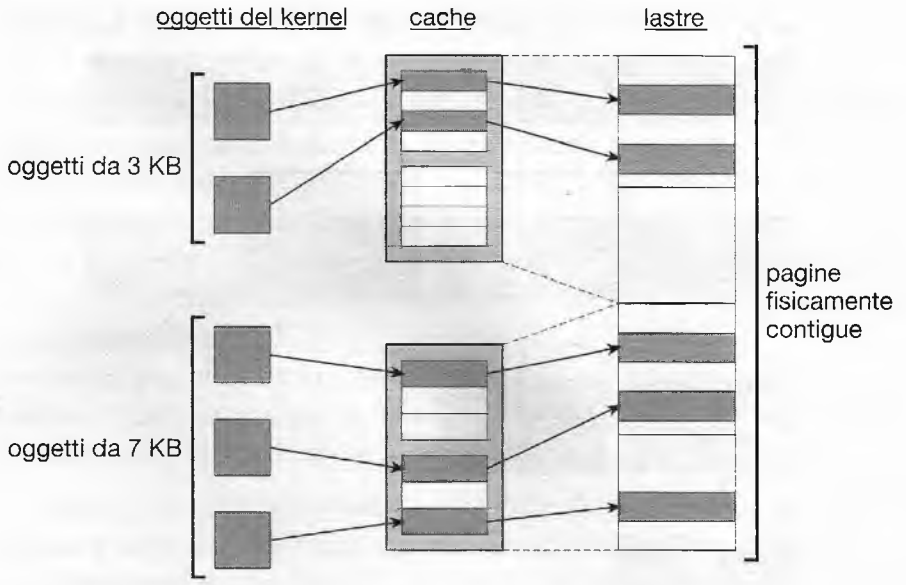



**Figura 9.28** Allocazione a lastre (*slab*).

ria. Quando il kernel di Linux crea un nuovo task, richiede alla propria cache la memoria necessaria per l'oggetto di tipo `struct task_struct`. La cache darà corso alla richiesta impiegando un oggetto di questo tipo che sia già stato allocato in una lastra e rechi il contrassegno "libero".

In Linux una lastra può essere in uno dei seguenti stati.

1. **Piena**. Tutti gli oggetti della lastra sono contrassegnati come usati.
2. **Vuota**. Tutti gli oggetti della lastra sono contrassegnati come liberi.
3. **Parzialmente occupata**. La lastra contiene oggetti sia usati sia liberi.

L'allocatore delle lastre, per soddisfare una richiesta, tenta in primo luogo di estrarre un oggetto libero da una lastra parzialmente occupata; se non ne esistono, assegna un oggetto libero da una lastra vuota; in mancanza di lastre vuote disponibili, crea una nuova lastra da pagine fisiche contigue e la alloca a una cache; da tale lastra si attinge la memoria da allocare all'oggetto.

L'allocatore delle lastre offre, essenzialmente, due vantaggi.

1. Annulla lo spreco di memoria derivante da frammentazione. La frammentazione cessa di costituire un problema, poiché ogni struttura dati del kernel ha una cache associata; ciascuna delle cache è composta da un numero variabile di lastre, suddivise in spezzoni di grandezza pari a quella degli oggetti rappresentati. Pertanto, quando il kernel esige memoria per un oggetto, l'allocatore delle lastre restituisce la quantità esatta di memoria necessaria per rappresentare l'oggetto.
2. Le richieste di memoria possono essere soddisfatte rapidamente. La tecnica di allocazione delle lastre si rivela particolarmente efficace quando, nella gestione della memoria, gli oggetti sono frequentemente allocati e deallocati, come spesso accade con le richieste del kernel. In termini di tempo, allocare e deallocare memoria può essere un processo dispendioso. Tuttavia, gli oggetti sono creati in anticipo e possono dunque essere allocati rapidamente dalla cache. Inoltre, quando il kernel rilascia un oggetto di cui non ha più bisogno, questo è dichiarato libero e restituito alla propria cache, rendendosi così immediatamente disponibile ad altre richieste del kernel.

L'allocatore delle lastre ha fatto la sua prima apparizione nel kernel di Solaris 2.4. Per la sua natura generale è ora applicato da Solaris anche ad alcune richieste di memoria in modalità utente. Linux adottava, originariamente, il sistema buddy; tuttavia, a partire dalla versione 2.2, il kernel di Linux include l'allocazione a lastre.

## 9.9 Altre considerazioni

Le due scelte fondamentali nella progettazione dei sistemi di paginazione sono la definizione dell'algoritmo di sostituzione e della politica di allocazione, già analizzate in questo capitolo. Si devono però fare anche molte altre considerazioni.

### 9.9.1 Prepaginazione

Una caratteristica ovvia per un sistema di paginazione su richiesta pura consiste nell'alto numero di assenze di pagine che si verificano all'avvio di un processo. Questa situazione è dovuta al tentativo di portare la località iniziale in memoria. La stessa situazione si può presentare anche in altri momenti. Ad esempio, quando si riavvia un processo scaricato nell'area d'avvicendamento, tutte le sue pagine si trovano nel disco e ognuna di loro deve essere reinserita in memoria tramite la gestione di un'eccezione di pagina mancante. La **prepaginazione** rappresenta un tentativo di prevenire un così alto livello di paginazione iniziale. Alcuni sistemi operativi, e in particolare Solaris, applicano la prepaginazione ai frame dei file di dimensioni modeste.

In un sistema che usi il modello dell'insieme di lavoro, ad esempio, a ogni processo si associa una lista delle pagine contenute nel suo insieme di lavoro. Se occorre sospendere un processo a causa di un'attesa di I/O oppure dell'assenza di frame liberi, si memorizza il suo insieme di lavoro. Al momento di riprendere l'esecuzione del processo (perché l'I/O è terminato o un numero sufficiente di frame liberi è divenuto disponibile), al completamento dell'I/O oppure quando si raggiunge il numero di frame sufficiente, prima di riavviare il processo, si riporta in memoria il suo intero insieme di lavoro.

In alcuni casi la prepaginazione può essere vantaggiosa. La questione riguarda semplicemente il suo costo, che deve essere inferiore al costo per l'assistenza delle corrispondenti mancanze di pagina. Può accadere che molte pagine trasferite in memoria dalla prepaginazione non siano usate.

Si supponga che siano prepaginate $s$ pagine e sia effettivamente usata una frazione $\alpha s$ di queste $s$ pagine ($0 \leq \alpha \leq 1$). Occorre sapere se il costo delle $\alpha s$ eccezioni di pagine mancanti risparmiate sia maggiore o minore del costo di prepaginazione di $(1 - \alpha)s$ pagine non necessarie. Se il parametro $\alpha$ è prossimo allo 0, la prepaginazione non è conveniente; se $\alpha$ è prossimo a 1, la prepaginazione certamente lo è.

### 9.9.2 Dimensione delle pagine

È raro che chi progetta un sistema operativo per un calcolatore esistente possa scegliere le dimensioni delle pagine. Tuttavia, se si devono progettare nuovi calcolatori, occorre stabilire quali siano le dimensioni migliori per le pagine. Come s'intuisce non esiste un'unica dimensione migliore, ma più fattori sono a sostegno delle diverse dimensioni. Le dimensioni delle pagine sono invariabilmente potenze di 2, in genere comprese tra 4096 ($2^{12}$) e 4.194.304 ($2^{22}$) byte.

Un fattore da considerare nella scelta delle dimensioni di una pagina è la dimensione della tabella delle pagine. Per un dato spazio di memoria virtuale, diminuendo la dimensio-

ne delle pagine aumenta il numero delle stesse e quindi la dimensione della tabella delle pagine. Per una memoria virtuale di 4 MB ($2^{22}$), ci sarebbero 4096 pagine di 1024 byte, ma solo 512 pagine di 8192 byte. Poiché ogni processo attivo deve avere la propria copia della tabella delle pagine, conviene che le pagine siano ampie.

D'altra parte, la memoria è utilizzata meglio se le pagine sono piccole. Se a un processo si assegna una porzione della memoria che comincia alla locazione 00000 e continua fino alla quantità di cui necessita, è molto probabile che il processo non termini esattamente sul limite di una pagina, lasciando inutilizzata (le pagine sono unità di allocazione) una parte della pagina finale (frammentazione interna). Supponendo che le dimensioni del processo e delle pagine siano indipendenti è probabile che, in media, metà dell'ultima pagina di ogni processo sia sprecata. Questa perdita è di soli 256 byte per una pagina di 512 byte, ma di 4096 byte per una pagina di 8192 byte. Quindi, per ridurre la frammentazione interna occorrono pagine di piccole dimensioni.

Un altro problema è dovuto al tempo richiesto per leggere o scrivere una pagina. Il tempo di I/O è dato dalla somma dei tempi di posizionamento, latenza e trasferimento. Il tempo di trasferimento è proporzionale alla quantità trasferita, ossia alla dimensione delle pagine; tutto ciò farebbe supporre che siano preferibili pagine piccole. Come però vedremo nel Paragrafo 12.1.1, il tempo di trasferimento è piccolo se è confrontato con il tempo di latenza e il tempo di posizionamento. A una velocità di trasferimento di 2 MB al secondo, per trasferire 512 byte s'impiegano 0,2 millisecondi. D'altra parte, il tempo di latenza è di circa 8 millisecondi e quello di posizionamento 20 millisecondi. Perciò, del tempo totale di I/O (28,2 millisecondi), l'1 per cento è attribuibile al trasferimento effettivo. Raddoppiando le dimensioni delle pagine, il tempo di I/O aumenta solo fino a 28,4 millisecondi. S'impiegano 28,4 millisecondi per leggere una sola pagina di 1024 byte, ma 56,4 millisecondi per leggere la stessa quantità di byte su due pagine di 512 byte l'una. Quindi, per ridurre il tempo di I/O occorre avere pagine di dimensioni maggiori.

Tuttavia, con pagine di piccole dimensioni si riduce l'I/O totale, poiché si migliorano le caratteristiche di località. Pagine di piccole dimensioni permettono di corrispondere con maggior precisione alla località del programma. Si consideri, ad esempio, un processo di 200 KB, dei quali solo la metà (100 KB) sono effettivamente usati durante l'esecuzione. Se si dispone di una sola ampia pagina, occorre inserirla tutta, sicché vengono trasferiti e assegnati 200 KB. Disponendo di pagine di 1 byte, si possono invece inserire i soli 100 KB effettivamente usati, con trasferimento e allocazione di quei soli 100 KB. Con pagine di piccole dimensioni è possibile avere una migliore **risoluzione**, che permette di isolare solo la memoria effettivamente necessaria. Se le dimensioni delle pagine sono maggiori, occorre assegnare e trasferire non solo quanto è necessario, ma tutto quel che è presente nella pagina, a prescindere dal fatto che sia o meno necessario. Quindi dimensioni più piccole dànno come risultato una minore attività di I/O e una minore memoria totale assegnata.

D'altra parte occorre notare che con pagine di 1 byte si verifica un'assenza di pagina per *ciascun* byte. Un processo di 200 KB che usa solo metà di tale memoria accusa una sola assenza di pagina con una pagina di 200 KB, ma 102.400 assenze di pagine con le pagine di 1 byte. Ciascuna assenza di pagina causa un rilevante sovraccarico necessario a elaborare il segnale di eccezione, salvare i registri, sostituire una pagina, mettere in coda nell'attesa del dispositivo di paginazione e aggiornare le tabelle. Per ridurre il numero delle assenze di pagine al minimo sono necessarie pagine di grandi dimensioni.

Occorre considerare altri fattori, come la relazione tra la dimensione delle pagine e quella dei settori del mezzo di paginazione. Non esiste una risposta ottimale al problema considerato. Alcuni fattori (frammentazione interna, località) sono a favore delle piccole dimensioni, mentre altri (dimensione delle tabelle, tempo di I/O) sono a favore delle grandi

dimensioni. La tendenza è storicamente verso l'incremento delle dimensioni delle pagine. Nella prima edizione di questo testo (1983) si considerava un valore di 4096 byte come limite superiore alla dimensione delle pagine. Nel 1990 tale dimensione delle pagine era la più comune. I sistemi moderni possono impiegare pagine di dimensioni assai maggiori, come vedremo nel paragrafo successivo.

### 9.9.3 Portata della TLB

Il **tasso di successi** (*hit ratio*) di una TLB – si veda in proposito anche il Capitolo 8 – si riferisce alla percentuale di traduzioni di indirizzi virtuali risolte dalla TLB anziché dalla tabella delle pagine. Il tasso di successi è evidentemente proporzionale al numero di elementi della TLB. Tuttavia, la memoria associativa che si usa per costruire le TLB è costosa e consuma molta energia.

Un parametro simile, detto **portata della TLB** (*TLB reach*), esprime la quantità di memoria accessibile dalla TLB, ed è dato semplicemente dal numero di elementi (quindi è correlato al tasso di successi) moltiplicato per la dimensione delle pagine. Idealmente, la TLB dovrebbe contenere l'insieme di lavoro di un processo; altrimenti, la necessità di ricorrere alla tabella delle pagine per la traduzione dei riferimenti alla memoria farà sì che il processo impieghi in quest'operazione assai più tempo di quello richiesto dalla sola TLB. Se si raddoppia il numero di elementi della TLB, se ne raddoppia la portata; per alcune applicazioni che comportano un uso intensivo della memoria ciò potrebbe rivelarsi ancora insufficiente per la memorizzazione dell'insieme di lavoro.

Un altro metodo per aumentare la portata della TLB consiste nell'aumentare la dimensione delle pagine oppure nell'impiegare diverse dimensioni delle pagine. Se si aumenta la dimensione delle pagine, per esempio da 8 KB a 32 KB, la portata della TLB si quadruplica. Quest'aumento potrebbe però condurre a una maggiore frammentazione della memoria relativamente alle applicazioni che non richiedono pagine così grandi. UltraSPARC II è un esempio di architettura che consente diverse dimensioni delle pagine: 8 KB, 64 KB, 512 KB e 4 MB. Di queste possibili dimensioni il sistema operativo Solaris impiega sia quella di 8 KB sia quella di 4 MB. Con una TLB a 64 elementi, la portata della TLB per Solaris varia da 512 KB, con tutte le pagine di 8 KB, a 256 MB, con tutte le pagine di 4 MB. Per la maggior parte delle applicazioni una dimensione delle pagina di 8 KB è sufficiente, sebbene Solaris associ i primi 4 MB del codice e dei dati del kernel a due pagine di 4 MB. Il sistema operativo Solaris permette anche alle applicazioni, come ad esempio i sistemi di gestione delle basi di dati, di trarre vantaggio dalle grandi pagine di 4 MB. Per la maggior parte delle applicazioni è sufficiente la dimensione di 8 KB, sebbene Solaris mappi i primi 4 MB del codice del kernel e dei dati in due pagine da 4 MB. Solaris inoltre consente alle applicazioni – come le basi di dati – di trarre vantaggio dalla dimensione ampia della pagina da 4 MB.

L'uso di diverse dimensioni delle pagine richiede però che la gestione della TLB sia svolta dal sistema operativo e non direttamente dall'architettura. Ad esempio, uno dei campi degli elementi della TLB deve indicare la dimensione della pagina fisica cui il contenuto di ciascun elemento fa riferimento. La gestione della TLB svolta dal sistema operativo e non esclusivamente dall'architettura comporta una penalizzazione delle prestazioni. Tuttavia, i vantaggi dovuti all'aumento del tasso di successi e della portata della TLB superano gli svantaggi che derivano dalla riduzione della rapidità di traduzione degli indirizzi. I recenti sviluppi indicano infatti un'evoluzione verso TLB gestite dal sistema operativo e verso l'uso di pagine di diverse dimensioni. Le architetture UltraSPARC, MIPS, e Alpha adottano TLB che si gestiscono tramite il sistema operativo, mentre le architetture PowerPC e Pentium gestiscono le TLB direttamente, senza l'intervento del sistema operativo.

### 9.9.4 Tabella delle pagine invertita

Nel Paragrafo 8.5.3 si è introdotto il concetto di tabella delle pagine invertita come sistema di gestione delle pagine che consente di ridurre la quantità di memoria fisica necessaria per tener traccia della corrispondenza tra gli indirizzi virtuali e gli indirizzi fisici. Tale riduzione si ottiene tramite una tabella con un elemento per pagina fisica, indicizzato dalla coppia <*id-processo, numero di pagina*>.

Poiché contiene informazioni su quale pagina di memoria virtuale è memorizzata in ciascuna pagina fisica, la tabella delle pagine invertita riduce la quantità di memoria fisica necessaria a memorizzare tali informazioni. Tuttavia essa non contiene le informazioni complete sullo spazio degli indirizzi logici di un processo, richieste se una pagina a cui si è fatto riferimento non è correntemente presente in memoria; la paginazione su richiesta richiede tali informazioni per elaborare le eccezioni di pagine mancanti. Per disporre di tali informazioni è necessaria una tabella delle pagine esterna per ciascun processo. Queste tabelle sono simili alle ordinarie tabelle delle pagine di ciascun processo e contengono le informazioni relative alla locazione di ciascuna pagina virtuale.

Contrariamente a quel che potrebbe apparire, l'uso delle tabelle esterne delle pagine non è in contrasto con l'utilità della tabella delle pagine invertita; infatti a loro si fa riferimento solo nel caso di un'assenza di pagina; quindi non è necessario che siano immediatamente disponibili ed esse stesse sono paginate dentro e fuori dalla memoria come è necessario. Sfortunatamente, in questo modo si può verificare un'assenza di qualche pagina dello stesso gestore della memoria virtuale; in tal caso si verifica un'altra assenza di pagina quando esso carica in memoria la tabella esterna delle pagine per individuare la pagina virtuale in memoria ausiliaria (*backing store*). Questo caso particolare richiede un'accurata gestione da parte del kernel del sistema operativo e un ritardo nell'elaborazione della ricerca della pagina.

### 9.9.5 Struttura dei programmi

La paginazione su richiesta deve essere trasparente per il programma utente; spesso, l'utente non è neanche a conoscenza della natura paginata della memoria. In altri casi, però, le prestazioni del sistema si possono addirittura migliorare se l'utente (o il compilatore) è consapevole della sottostante presenza della paginazione su richiesta.

Come esempio limite, ma esplicativo, ipotizziamo pagine di 128 parole. Si consideri il seguente frammento di programma scritto in C la cui funzione è inizializzare a 0 ciascun elemento di una matrice di 128 × 128 elementi. È tipico il seguente codice:

```
int i, j;
int[128][128] data;

for (j = 0; j < 128; j++)
     for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

Occorre notare che l'array è memorizzato per riga, vale a dire che è disposto in memoria secondo l'ordine `data[0][0]`, `data[0][1]`, ..., `data[0][127]`, `data[1][0]`, `data[1][1]`, ..., `data[127][127]`. In pagine di 128 parole, ogni riga occupa una pagina, quindi il frammento di codice precedente azzera una parola per pagina, poi un'altra parola per pagina, e così via. Se il sistema operativo assegna meno di 128 frame a tutto il programma, la sua esecuzione causa 128 × 128 = 16.384 assenze di pagine. D'altra parte, cambiando il codice in

```
int i, j;
int[128][128] data;

for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

si azzerano tutte le parole di una pagina prima che si inizi la pagina successiva, riducendo a 128 il numero di assenze di pagine.

Un'attenta scelta delle strutture dati e delle strutture di programmazione può aumentare la località e quindi ridurre la frequenza delle assenze di pagine e il numero di pagine dell'insieme di lavoro. Una buona località è quella di una pila, poiché l'accesso avviene sempre alla sua parte superiore. Una tabella hash, invece, è progettata proprio per diffondere i riferimenti, causando una località non buona. Naturalmente, la località dei riferimenti rappresenta soltanto una misura dell'efficienza d'uso di una struttura dati. Altri fattori rilevanti sono rapidità di ricerca, numero totale dei riferimenti alla memoria e numero totale delle pagine coinvolte.

In uno stadio successivo, anche il compilatore e il caricatore possono avere un effetto notevole sulla paginazione. La separazione di codice e dati e la generazione di un codice rientrante significano che le pagine di codice si possono soltanto leggere e quindi non vengono modificate. Nel sostituire le pagine non modificate, non occorre scriverle in memoria ausiliaria. Il caricatore può evitare di collocare procedure lungo i limiti delle pagine, sistemando ogni procedura completamente all'interno di una pagina. Le procedure che si invocano a vicenda più volte si possono "impaccare" nella stessa pagina. Questa forma di impaccamento è una variante del problema del *bin-packing* della ricerca operativa: cercare di impaccare i segmenti di dimensione variabile in pagine di dimensione fissa, in modo da ridurre al minimo i riferimenti tra pagine diverse. Un metodo di questo tipo è utile soprattutto per pagine di grandi dimensioni.

Anche la scelta del linguaggio di programmazione può influire sulla paginazione. Con il linguaggio C e il C++, ad esempio, si fa spesso uso dei puntatori, che tendono a distribuire in modo casuale gli accessi alla memoria, diminuendo così la località di un processo. Alcuni studi hanno mostrato che anche i programmi scritti in linguaggi orientati agli oggetti tendono ad avere una scarsa località dei riferimenti.

### 9.9.6 Vincolo di I/O

Quando si usa la paginazione su richiesta, talvolta occorre permettere che alcune pagine si possano **vincolare** alla memoria (*locked in memory*). Una situazione di questo tipo si presenta quando l'I/O si esegue verso o dalla memoria utente (virtuale). Spesso il sistema di I/O comprende una specifica unità d'elaborazione; al controllore di un dispositivo di memorizzazione USB, ad esempio, generalmente si indica il numero di byte da trasferire e un indirizzo di memoria per il buffer (si veda a questo proposito la Figura 9.29). Completato il trasferimento, la CPU riceve un segnale d'interruzione.

Occorre essere certi che non si verifichi la seguente successione di eventi: un processo emette una richiesta di I/O ed è messo in coda per il relativo dispositivo. Nel frattempo si assegna la CPU ad altri processi che accusano assenze di pagine e, usando un algoritmo di sostituzione globale per uno di questi, si sostituisce la pagina contenente l'indirizzo di memoria per l'operazione di I/O attesa dal primo processo; la pagina viene scaricata dalla memoria. Qualche tempo dopo, quando la richiesta di I/O raggiunge la prima posizione della coda d'attesa per il dispositivo, l'operazione di I/O avviene all'indirizzo specificato, ma questo frame è ora impiegato per una pagina appartenente a un altro processo.

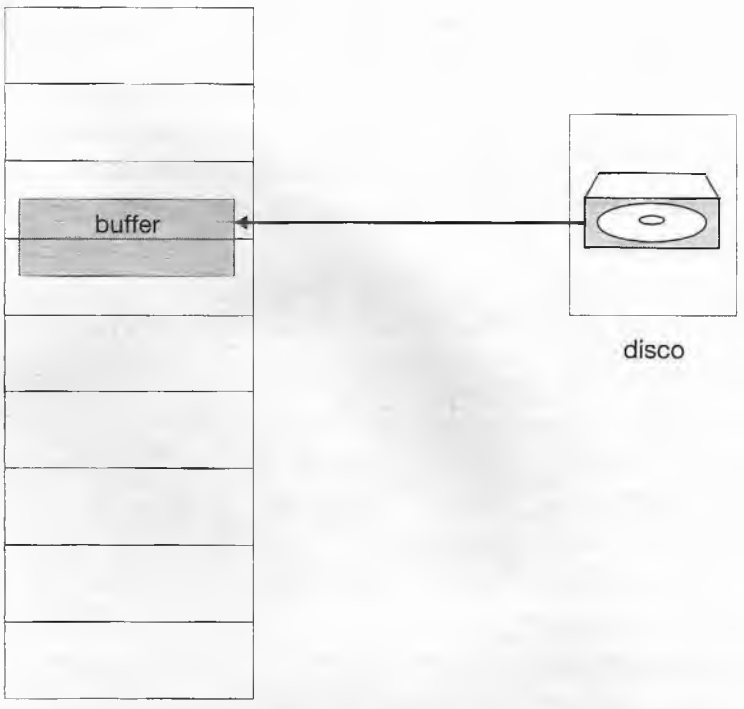



**Figura 9.29** Ragione per i cui i frame usati per l'I/O devono essere presenti in memoria.

Questo problema si può risolvere in due modi. Una soluzione prevede di non eseguire operazioni di I/O in memoria utente, ma di copiare i dati sempre tra la memoria di sistema e la memoria utente. In questo modo l'I/O avviene solo tra la memoria di sistema e il dispositivo di I/O. Per scrivere dati in un nastro, occorre prima copiarli in memoria di sistema, quindi trasferirli all'unità a nastro. Tale copia supplementare può causare un sovraccarico inaccettabile.

Un'altra soluzione consiste nel permettere che le pagine siano vincolate alla memoria. A ogni frame si associa un bit di vincolo (*lock bit*); se tale bit è attivato, la pagina contenuta in tale frame non può essere selezionata per la sostituzione. Seguendo questo metodo, per scrivere dati in un nastro occorre vincolare alla memoria le pagine contenenti tali dati, quindi il sistema può continuare come di consueto. Le pagine vincolate non si possono sostituire. Completato l'I/O, si rimuove il vincolo.

I bit di vincolo sono usati in varie situazioni. Spesso il kernel del sistema operativo, o una sua parte, è vincolato alla memoria. La maggior parte dei sistemi non può tollerare l'assenza di una pagina relativa al kernel.

Un altro uso del bit di vincolo riguarda la normale sostituzione di pagine. Si consideri la seguente successione d'eventi: un processo a bassa priorità subisce un'assenza di pagina. Selezionando un frame per la sostituzione, il sistema di paginazione carica in memoria la pagina necessaria. Pronto per continuare, il processo con priorità bassa entra nella coda dei processi pronti per l'esecuzione e attende l'allocazione della CPU. Giacché si tratta di un processo con bassa priorità, può non essere selezionato dallo scheduler della CPU per un certo tempo. Mentre il processo con priorità bassa attende, un processo ad alta priorità accusa un'assenza di pagina. Durante la ricerca per la sostituzione, il sistema di paginazione individua una pagina in memoria alla quale non sono stati fatti riferimenti o modifiche; si tratta della pagina che il processo con bassa priorità ha appena caricato. Questa pagina sembra una sostituzione perfetta: non è stata modificata, non è necessario scriverla in memoria secondaria e, apparentemente, non è stata usata da molto tempo.

Stabilire se la pagina del processo con bassa priorità si debba sostituire a vantaggio del processo con alta priorità è una questione di scelta di un criterio. Semplicemente, si ritarda il processo con bassa priorità a vantaggio di quello con priorità alta. D'altra parte, però, si spreca il lavoro fatto per trasferire in memoria la pagina del processo con bassa priorità. Per evitare che una pagina appena caricata sia sostituita prima che sia usata almeno una volta si può usare il bit di vincolo. Se una pagina è selezionata per la sostituzione, si attiva il suo bit di vincolo: tale bit rimane attivato finché si esegue nuovamente il processo che ha accusato l'assenza di pagina.

Tuttavia, l'uso dei bit di vincolo può essere pericoloso: se un bit non viene mai disattivato, ad esempio a causa di un baco del sistema operativo, il frame relativo alla pagina vincolata diventa inutilizzabile. Su un sistema a singolo utente, l'abuso di tale meccanismo può causare danni soltanto allo stesso utente. Ciò non si può consentire nei sistemi multiutente. Il sistema operativo Solaris, ad esempio, consente l'impiego di "suggerimenti" (*hint*) di vincolo delle pagine, che si possono però trascurare se l'insieme delle pagine libere diviene troppo piccolo o se un singolo processo richiede che troppe pagine siano vincolate alla memoria.

# 9.10 Esempi tra i sistemi operativi

In questo paragrafo si descrive la realizzazione della memoria virtuale nei sistemi operativi Windows XP e Solaris.

## 9.10.1 Windows XP

Il sistema operativo Windows XP realizza la memoria virtuale impiegando la paginazione su richiesta per gruppi di pagine (*demand paging with clustering*). Tale tecnica consiste nel gestire le assenze di pagine caricando in memoria, non solo la pagina richiesta, ma più pagine a essa adiacenti. Alla sua creazione, un processo riceve i valori dell'insieme di lavoro minimo e dell'insieme di lavoro massimo. L'**insieme di lavoro minimo** è il minimo numero di pagine caricate in memoria di un processo che il sistema garantisce di assegnare; se la memoria è sufficiente, però, il sistema potrebbe assegnare un numero di pagine pari al suo **insieme di lavoro massimo**. Per la maggior parte delle applicazioni la misura dell'insieme di lavoro minimo e di quello massimo va, rispettivamente, da 50 a 345 pagine. (In alcuni casi sarebbe anche possibile superare quest'ultimo valore). Il gestore della memoria virtuale mantiene una lista di pagine fisiche libere, con associato un valore di soglia che indica se è disponibile una quantità sufficiente di memoria libera oppure no. Se si verifica un'assenza di pagina per un processo che è sotto il suo insieme di lavoro massimo, il gestore della memoria virtuale assegna una pagina dalla lista delle pagine libere; se invece un processo è già al suo massimo e si verifica un'assenza di pagina, il gestore deve scegliere una pagina da sostituire usando un criterio di sostituzione locale.

Nel caso in cui la quantità di memoria libera scenda sotto la soglia, il gestore della memoria virtuale usa un metodo noto come **regolazione automatica dell'insieme di lavoro** (*automatic working-set trimming*) per riportare il valore sopra la soglia. Si tratta sostanzialmente di valutare il numero di pagine assegnate a ciascun processo; se a un processo sono state assegnate più pagine del suo insieme di lavoro minimo, il gestore della memoria virtuale rimuove pagine fino a raggiungere quel valore; a un processo che è al suo insieme di lavoro minimo, può assegnare altre pagine prendendole dalla lista delle pagine fisiche libere, non appena è disponibile una quantità sufficiente di memoria libera.

L'algoritmo impiegato per stabilire quale pagina rimuovere da un insieme di lavoro dipende dal tipo di unità d'elaborazione disponibile: nei sistemi monoprocessori 80*x*86, il si-

stema operativo Windows XP usa una variante dell'algoritmo a orologio, presentato nel Paragrafo 9.4.5.2; nei sistemi basati su CPU Alpha e nei sistemi multiprocessore *x*86, l'azzeramento del bit di riferimento può richiedere l'invalidamento dell'elemento corrispondente nella TLB delle altre unità d'elaborazione. Perciò anziché accettare quest'onere, nel sistema Windows XP si usa una variante dell'algoritmo FIFO illustrato nel Paragrafo 9.4.2.

## 9.10.2 Solaris

Il kernel del sistema operativo Solaris assegna una pagina a un thread ogni volta che si verifica un'assenza di pagina, prendendola dalla lista delle pagine libere che il kernel stesso mantiene. È quindi essenziale che il kernel riesca a mantenere una quantità sufficiente di memoria libera. Un parametro, *lotsfree*, associato alla lista delle pagine libere, rappresenta una soglia per l'inizio del processo di paginazione, e il suo è di solito fissato a 1/64 della dimensione della memoria fisica. Il kernel verifica, quattro volte al secondo, se la quantità di memoria libera è inferiore a *lotsfree*. Se il numero di pagine libere scende sotto *lotsfree*, si avvia un processo noto come **pageout**. Questo processo è simile all'algoritmo con seconda chance descritto nel Paragrafo 9.4.5.2, tranne per il fatto che usa due lancette per scorrere le pagine. Il suo funzionamento prevede che la prima lancetta scorra lungo tutte le pagine della memoria, azzerandone il bit di riferimento; in seguito, la seconda lancetta esamina il bit di riferimento delle pagine in memoria, ponendo le pagine con bit nullo in coda alla lista delle pagine libere, e scrivendone i contenuti su disco in caso di modifica. Solaris mantiene una lista cache di pagine liberate, ma non ancora soprascritte. La lista della pagine libere contiene frame dal contenuto non valido. Le pagine possono essere **richiamate** dalla lista cache nel caso in cui occorra accedervi prima che siano trasferite nella lista delle pagine libere.

Per controllare la frequenza di scansione delle pagine (chiamata anche *scanrate*) l'algoritmo pageout si serve di diversi parametri. Questa frequenza è espressa in pagine al secondo ed è compresa tra i valori *slowscan* e *fastscan*. Quando la memoria libera scende sotto *lotsfree*, la scansione delle pagine avviene alla frequenza *slowscan*, e sale fino a *fastscan* secondo la quantità di memoria libera disponibile. Il valore predefinito di *slowscan* è 100, mentre *fastscan* è di solito fissato a (*numero totale delle pagine fisiche*)/2 con un massimo di 8192 pagine al secondo. Questa variazione di frequenza è illustrata nella Figura 9.30 (con *fastscan* fissato al massimo).

La distanza (in pagine) tra le lancette dell'orologio è determinata dal parametro di sistema *handspread*. L'intervallo tra l'azzeramento di un bit da parte della lancetta anteriore e l'esame del suo valore da parte della lancetta posteriore dipende sia da *scanrate* sia da *handspread*. Se il valore di *scanrate* è pari a 100 pagine al secondo e quello di *handspread* è pari a 1024 pagine, possono passare 10 secondi tra la scrittura di un bit della lancetta anteriore e la sua verifica da parte di quella posteriore. Tuttavia, visti i requisiti imposti a un sistema di memoria, non sono rari valori di *scanrate* di diverse migliaia di pagine al secondo. Ciò significa che l'intervallo tra l'azzeramento e il controllo di un bit è spesso di pochi secondi.

Come si è descritto sopra, il processo pageout controlla la memoria quattro volte al secondo. Tuttavia, se la memoria libera scende sotto *desfree* (Figura 9.30) pageout sarà eseguito 100 volte al secondo con lo scopo di tenere una quantità di memoria libera almeno pari a *desfree*. Se il processo pageout non riesce a mantenere al valore *desfree* la quantità media di memoria libera calcolata in un intervallo di 30 secondi, il kernel intraprende l'avvicendamento dei processi, liberando, in questo caso, tutte le pagine assegnate a un processo. In generale, il kernel cerca i processi che sono rimasti inattivi per lunghi periodi. Infine, se il sistema non riesce a mantenere la quantità di memoria libera a *minfree*, invoca il processo pageout a ogni richiesta di una nuova pagina.

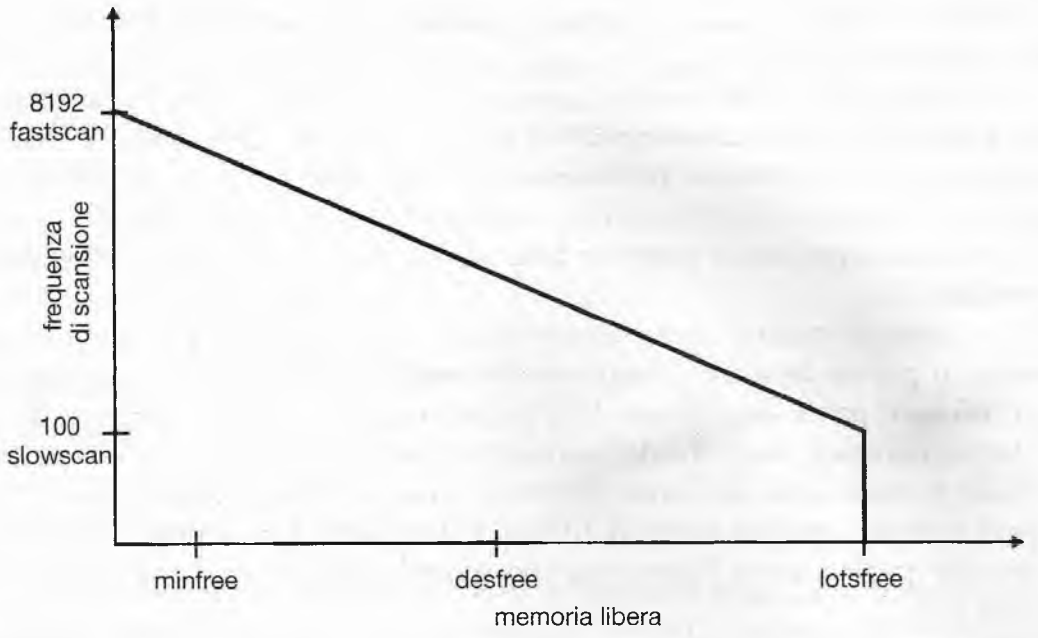



**Figura 9.30** Scansione delle pagine in Solaris.

Le versioni recenti del kernel di Solaris hanno portato alcuni miglioramenti all'algoritmo di paginazione. Uno è il riconoscimento delle pagine che appartengono a librerie condivise da più processi: anche se sono potenzialmente richiedibili per la scansione, sono ignorate durante il processo d'esame delle pagine. Un altro miglioramento riguarda la capacità di distinguere le pagine allocate ai processi da quelle allocate ai file ordinari. Si tratta del meccanismo di **paginazione con priorità** descritto nel Paragrafo 11.6.2.

# 9.11 Sommario

È auspicabile poter eseguire processi il cui spazio degli indirizzi logici superi quello disponibile per gli indirizzi fisici. La memoria virtuale è una tecnica che permette di associare grandi spazi degli indirizzi logici a quantità più ridotte di memoria fisica. La memoria virtuale è una tecnica che consente di eseguire processi molto grandi e di aumentare il grado di multiprogrammazione, incrementando l'utilizzo della CPU. Inoltre, grazie a tale tecnica, i programmatori di applicazioni non devono più preoccuparsi della disponibilità di memoria. In più, grazie alla memoria virtuale, processi distinti possono condividere librerie di sistema e memoria. La memoria virtuale consente anche l'utilizzo di un tipo efficiente di creazione di processo conosciuto con il nome di copiatura su scrittura (*copy-on-write*), in cui i processi genitore e figlio condividono pagine effettive di memoria.

La memoria virtuale è comunemente implementata tramite paginazione su richiesta, che trasferisce in memoria una pagina solo quando si incontra un riferimento alla pagina stessa; il primo riferimento produce un errore di pagina. Il kernel del sistema operativo consulta una tabella interna per stabilire la locazione della pagina in memoria ausiliaria, quindi individua un frame libero e vi trasferisce la pagina prelevandola dalla memoria ausiliaria. La tabella delle pagine viene aggiornata per riflettere tale modifica e si riavvia l'istruzione che aveva causato l'eccezione di pagina mancante. Questo metodo permette l'esecuzione di un processo anche se in memoria centrale non è interamente presente la sua immagine di me-

moria. Finché la frequenza delle assenze di pagine rimane ragionevolmente bassa, le prestazioni si considerano accettabili.

La paginazione su richiesta si può usare per ridurre il numero dei frame assegnati a un processo. Questo metodo può aumentare il grado di multiprogrammazione, permettendo che più processi siano disponibili per l'esecuzione in un dato momento e, almeno in teoria, può migliorare l'utilizzo della CPU. Inoltre, consente l'esecuzione di processi i cui requisiti di spazio di memoria superano la memoria fisica disponibile. Tali processi si eseguono in memoria virtuale.

Se i requisiti di spazio di memoria superano la memoria fisica, può essere necessaria la sostituzione di pagine presenti in memoria allo scopo di liberare frame per nuove pagine. Gli algoritmi usati per la sostituzione delle pagine sono diversi: la sostituzione di tipo FIFO è facile da programmare, ma soffre dell'anomalia di Belady; la sostituzione ottimale delle pagine richiede la conoscenza dei futuri riferimenti alla memoria; la sostituzione delle pagine LRU è quasi ottimale, ma può essere di difficile realizzazione. Quasi tutti gli algoritmi di sostituzione delle pagine, come l'algoritmo con seconda chance, sono approssimazioni della sostituzione LRU.

Oltre un algoritmo di sostituzione delle pagine, occorre un criterio di allocazione dei frame. L'allocazione può essere statica, indicando una sostituzione di pagine locale, oppure dinamica, con una sostituzione di pagine globale. Il modello dell'insieme di lavoro presuppone che i processi siano eseguiti in località. L'insieme di lavoro è l'insieme delle pagine nella località corrente. Di conseguenza, a ogni processo si possono allocare frame sufficienti al suo corrente insieme di lavoro. Se un processo non ha spazio di memoria sufficiente per il proprio insieme di lavoro, si ha una paginazione degenere (*thrashing*). Se a ogni processo si devono fornire frame sufficienti per evitare tale degenerazione, sono necessarie le attività d'avvicendamento (*swapping*) e scheduling dei processi.

La maggior parte dei sistemi operativi mette a disposizione degli strumenti per la mappatura in memoria dei file, ciò che permette di trattare l'I/O alla stregua degli accessi in memoria. La API Win32 implementa la condivisione della memoria tramite la mappatura in memoria di file.

Di solito, i processi del kernel richiedono l'allocazione di pagine fisicamente contigue. Il sistema buddy alloca memoria al kernel in segmenti di dimensioni pari a potenze di 2; ciò conduce facilmente a frammentazione. L'allocazione a lastre assegna le strutture dati del kernel a cache associate a lastre, le quali a loro volta sono costituite da una o più pagine fisiche contigue. Questa strategia non produce frammentazione e permette di servire rapidamente le richieste del kernel.

La corretta progettazione dei sistemi di paginazione non solo richiede la soluzione dei due problemi fondamentali della sostituzione delle pagine e dell'allocazione dei frame, ma porta anche a considerare questioni relative a dimensione delle pagine, I/O, gestione dei lock, prepaginazione, generazione dei processi, struttura dei programmi, e altro ancora.

## Esercizi pratici

9.1 In quali circostanze si verifica un'eccezione di pagina mancante? Descrivete le azioni che vengono intraprese dal sistema operativo in caso di eccezione di pagina mancante.

9.2 Considerate una successione di riferimenti alle pagine di memoria per un processo con $m$ frame (inizialmente tutti vuoti). La successione ha lunghezza $p$; in essa vi sono $n$ distinti numeri di pagina. Rispondete alle seguenti domande relative ad algoritmi di sostituzione delle pagine:

a. Qual è un limite inferiore del numero di pagine mancanti?
b. Qual è un limite superiore del numero di pagine mancanti?

**9.3** Quale delle seguenti tecniche e strutture impiegate nella programmazione si adatta a un ambiente di paginazione su richiesta? Quali invece non sono indicate? Argomentate le vostre risposte:

a. pila;
b. tabella hash dei simboli;
c. ricerca sequenziale;
d. ricerca binaria;
e. codice puro;
f. operazioni vettoriali;
g. indirezione (indirection).

**9.4** Considerate i seguenti algoritmi di sostituzione delle pagine e valutateli basandovi su di una scala a cinque valori da "pessimo" a "ottimo" a seconda della frequenza delle assenze di pagine. Separate gli algoritmi che soffrono dell'anomalia di Belady da quelli che non ne sono affetti:

a. sostituzione delle pagine usate meno recentemente (LRU);
b. sostituzione delle pagine secondo l'ordine d'arrivo (FIFO);
c. sostituzione ottimale;
d. sostituzione alla seconda chance.

**9.5** L'uso della memoria virtuale in un sistema operativo comporta una serie di costi e di benefici: elencateli. È possibile che i costi superino i benefici? In questo caso, quali misure possono essere introdotte per evitare che ciò accada?

**9.6** Un sistema operativo offre la memoria virtuale paginata, utilizzando un processore centrale con una durata di ciclo di 1 microsecondo. L'accesso a una pagina diversa da quella corrente richiede 1 ulteriore microsecondo. Le pagine hanno 1.000 parole e lo strumento di paginazione è un cilindro che ruota a 3.000 giri al minuto e trasferisce un milione di parole al secondo. Dal sistema si ottengono le seguenti misurazioni statistiche.

- L'1 per cento di tutte le istruzioni eseguite hanno avuto accesso a una pagina diversa dalla pagina corrente.
- L'80 per cento di queste istruzioni – che hanno cioè avuto accesso a un'altra pagina – hanno avuto accesso a una pagina già in memoria.
- Nel caso in cui sia stata richiesta una nuova pagina, la pagina sostituita è stata modificata nel 50 per cento dei casi.

Calcolate il tempo effettivo di esecuzione delle istruzioni di questo sistema, assumendo che il sistema stia eseguendo un unico processo e che il processore sia fermo durante i trasferimenti dal cilindro.

**9.7** Considerate un array bidimensionale A:

```
int A [] [] = new int[100][100];
```

dove `A[0][0]` si trova nella posizione 200 in un sistema di memoria paginato con pagine di dimensione 200. Un piccolo processo che manipola la matrice risiede alla pagina 0 (posizioni da 0 a 199); ogni prelievo di istruzioni partirà così dalla pagina 0.

Per tre frame di pagina, quanti errori per assenza di pagina vengono generati dai seguenti cicli di inizializzazione dell'array, utilizzando la sostituzione LRU e ipotizzando che un frame contenga il processo e gli altri due frame siamo inizialmente vuoti?

a.
```
for (int j = 0; j < 100; j++)
      for (int i = 0; i < 100; i++)
         A[i][j] = 0;
```

b.
```
for (int i = 0; i < 100; i++)
      for (int j = 0; j < 100; j++)
         A[i][j] = 0;
```

**9.8** Considerate la seguente successione di riferimenti a pagine di memoria:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Quante eccezioni di pagina mancante si verificherebbero per i seguenti algoritmi di sostituzione, assumendo uno, due, tre, quattro, cinque, sei e sette frame? Ricordate che tutti i frame sono inizialmente vuoti, per cui le vostre prime pagine uniche costeranno un'eccezione ciascuna.

- Sostituzione secondo LRU.
- Sostituzione secondo FIFO.
- Sostituzione ottimale.

**9.9** Supponete di voler utilizzare un algoritmo di paginazione che richiede un bit di riferimento (come nella sostituzione alla seconda chance o nel modello dell'insieme di lavoro) che non viene però fornito dall'hardware. Delineate un'ipotesi di simulazione per un bit di riferimento anche se non fornito dall'hardware, oppure spiegate perché non è possibile mettere in pratica una tale ipotesi. Se possibile, calcolate il costo di questo progetto.

**9.10** Avete progettato un nuovo algoritmo per la sostituzione delle pagine che pensate possa essere ottimale. In alcuni complicati test di controllo si verifica l'anomalia di Belady. L'algoritmo può essere considerato ottimale? Argomentate la vostra risposta.

**9.11** La segmentazione è simile alla paginazione, ma utilizza "pagine" di dimensione variabile. Definite due algoritmi di sostituzione dei segmenti basati sugli schemi di sostituzione delle pagine FIFO e LRU. Ricordate che, siccome i segmenti non hanno la stessa dimensione, il segmento scelto per essere sostituito può essere troppo piccolo per poter contenere abbastanza locazioni di memoria consecutive per il segmento richiesto. Considerate strategie per sistemi nei quali i segmenti non possono essere rilocati e per sistemi nei quali ciò è invece possibile.

**9.12** Considerate un sistema informatico a paginazione su richiesta nel quale il grado di multiprogrammazione sia attualmente fissato a quattro. Il sistema è stato recentemente sottoposto a misurazioni volte a determinare l'utilizzo del processore e del disco di paginazione, ottenendo come risultati una delle seguenti alternative. Per ognuno di questi casi, che cosa sta avvenendo? Il livello di multiprogrammazione può essere incrementato per migliorare l'utilizzo del processore? La paginazione può essere utile?

a. Utilizzo del processore 13 per cento; utilizzo del disco 97 per cento.
b. Utilizzo del processore 87 per cento; utilizzo del disco 3 per cento.
c. Utilizzo del processore 13 per cento; utilizzo del disco 3 per cento.

9.13 Considerate un sistema operativo per una macchina che utilizza registri base e limite, ma supponete di aver modificato la macchina di modo che metta a disposizione una tabella delle pagine. Si possono configurare le tabelle in modo da simulare registri base e limite? Come? Oppure, perché la cosa non è possibile?

## Esercizi

9.14 Supponete che un programma abbia appena fatto riferimento a un indirizzo nella memoria virtuale. Descrivete uno scenario nel quale si verifichi ognuno dei seguenti eventi. (Se non è possibile che si verifichi un tale scenario, spiegatene il motivo.)

- Insuccesso della TLB senza assenze di pagina.
- Insuccesso della TLB con assenze di pagina.
- Successo della TLB senza assenze di pagina.
- Successo della TLB con assenze di pagina.

9.15 Una visione semplificata degli stati di un thread è **Pronto** (*Ready*), **Esecuzione** (*Running*) e **Bloccato** (*Blocked*), dove un thread è pronto e in attesa di essere pianificato, oppure è in esecuzione nel processore, oppure è bloccato (ad esempio è in attesa di I/O), come illustrato nella Figura 9.31. Assumendo che un thread si trovi nello stato di Esecuzione, rispondete alle seguenti domande, argomentando le risposte.

a. Il thread cambierà di stato se incorrerà in una assenza di pagina? In caso affermativo, quale sarà il nuovo stato?
b. Il thread cambierà di stato se genererà un insuccesso della TBL che viene risolto nella tabella delle pagine? In caso affermativo, quale sarà il nuovo stato?
c. Il thread cambierà di stato se il riferimento all'indirizzo viene risolto nella tabella delle pagine? In caso affermativo, quale sarà il nuovo stato?

9.16 Considerate un sistema che utilizza la paginazione pura su richiesta.

a. Quando un processo inizia a essere eseguito, come caratterizzereste il tasso di assenze di pagina?
b. Una volta che l'insieme di lavoro di un processo viene caricato nella memoria, come caratterizzereste il tasso di assenze di pagina?
c. Supponete che un processo cambi la propria collocazione in memoria e che la dimensione del nuovo insieme di lavoro sia troppo grande per essere caricata nella memoria libera disponbile. Identificate alcune opzioni che i progettisti di sistemi potrebbero scegliere per gestire questa situazione.

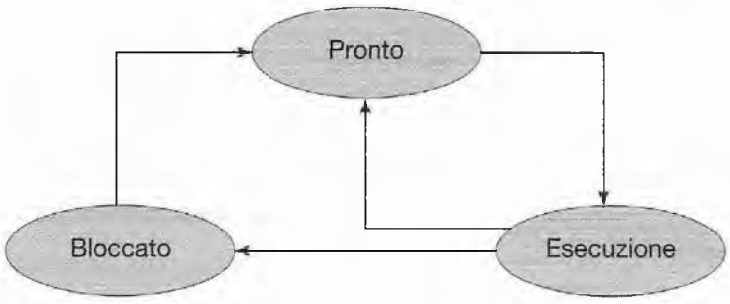



**Figura 9.31** Diagramma di stato del thread per l'Esercizio 9.15.

**9.17** Illustrate con un esempio il problema del riavvio dell'istruzione di rimozione del blocco (MVC) sull'IBM 360/70 quando le regioni di origine e di destinazione si sovrappongono.

**9.18** Analizzate i dispositivi fisici necessari alla paginazione su richiesta.

**9.19** Come descrivereste la funzionalità della copiatura su scrittura? A quali condizioni l'uso di tale funzionalità è vantaggioso? Quale dispositivo fisico è richiesto per implementarla?

**9.20** Un elaboratore fornisce ai propri utenti uno spazio di memoria virtuale di $2^{32}$ byte. L'elaboratore dispone di $2^{18}$ byte di memoria fisica. La memoria virtuale è implementata tramite paginazione, e la dimensione delle pagine è di 4096 byte. Un processo utente genera l'indirizzo virtuale 11123456. Spiegate in che modo il sistema determina la corrispondente locazione fisica, distinguendo fra operazioni software e hardware.

**9.21** Ipotizzate di avere una memoria paginata su richiesta. La tabella delle pagine è conservata in registri. Se un frame vuoto è disponibile o se la pagina sostituita non è modificata, per ovviare alla mancanza di una pagina sono necessari 8 millisecondi, mentre occorrono 20 millisecondi, qualora la pagina sostituita subisca modifiche. Il tempo di accesso alla memoria è pari a 100 nanosecondi.

Supponete che la pagina da sostituire subisca modifiche in ragione del 70 per cento del tempo. Per un tempo effettivo di accesso non superiore a 200 nanosecondi, qual è la frequenza massima tollerabile di pagine mancanti?

**9.22** Quando manca una pagina, il processo che ha richiesto la pagina deve bloccarsi mentre aspetta che la pagina venga portata dal disco alla memoria fisica. Posto che esista un processo con cinque thread a livello utente e che la rilocazione dei thread utente ai thread del kernel sia di molti a uno, se un thread utente incorre nell'assenza di pagina quando accede alla sua pila, anche gli altri thread utente appartenenti allo stesso processo sono coinvolti nell'assenza di pagina: devono quindi anch'essi aspettare che la pagina mancante venga portata alla memoria? Motivate la risposta.

**9.23** Considerate la tabella delle pagine per un sistema con indirizzi virtuali e fisici a 12 bit con pagine di 256 byte. La lista dei frame di pagina liberi è *D, E, F* (dove *D* è in testa alla lista, *E* al secondo posto, ed *F* è in coda).

| Pagina | Frame di pagina |
|---|---|
| 0 | – |
| 1 | 2 |
| 2 | C |
| 3 | A |
| 4 | – |
| 5 | 4 |
| 6 | 3 |
| 7 | – |
| 8 | B |
| 9 | 0 |

Convertite i seguenti indirizzi virtuali nei corrispondenti indirizzi fisici, in esadecimale. Tutti i numeri dati sono esadecimali. (Il trattino nella colonna dei frame di pagina indica che la pagina non è in memoria.)

- 9EF
- 111
- 700
- 0FF

**9.24** Si supponga di monitorare la velocità con cui si muove il puntatore nell'algoritmo a orologio (che indica la pagina candidata per la sostituzione). Che cosa si può inferire sul sistema sapendo che:

a. il puntatore si muove velocemente;
b. il puntatore si muove lentamente.

**9.25** Esaminate a quali condizioni l'algoritmo di sostituzione delle pagine meno frequentemente usate genera un numero inferiore di errori per pagine mancanti rispetto all'algoritmo di sostituzione delle pagine usate meno recentemente. Descrivete anche le circostanze nelle quali è vero il contrario.

**9.26** Considerate a quali condizioni l'algoritmo di sostituzione delle pagine più frequentemente usate genera un numero inferiore di errori per pagine mancanti rispetto all'algoritmo di sostituzione delle pagine usate meno recentemente. Descrivete anche le circostanze nelle quali è vero il contrario.

**9.27** Il sistema VAX/VMS utilizza un algoritmo di sostituzione FIFO per le pagine residenti, nonché un gruppo di frame liberi costituito dalle pagine recentemente usate. Per gestire il gruppo di frame, ricorrete al criterio che sostituisce le pagine meno frequentemente usate. Rispondete alle seguenti domande.

a. Se viene a mancare una pagina che non si trova nel gruppo di frame, come si genera lo spazio libero per la pagina appena richiesta?
b. Se si verifica la mancanza di una pagina che si trova nel gruppo di frame, come va impostata la pagina residente e in che modo deve essere gestito il gruppo di frame per far spazio alla pagina richiesta?
c. In che cosa degenera il sistema di paginazione se il numero delle pagine residenti è impostato a uno?
d. In che cosa degenera il sistema di paginazione se il numero delle pagine nel gruppo di frame è zero?

**9.28** Considerate un sistema con paginazione su richiesta con il seguente utilizzo temporale:

| | |
|---|---|
| utilizzo della CPU | 20 per cento |
| disco di paginazione | 97,7 per cento |
| altri dispositivi di I/O | 5 per cento |

Indicate, fra le seguenti operazioni, quelle che consentono (o è probabile che consentano) di migliorare l'utilizzo della CPU:

a. installazione di una CPU più veloce;
b. installazione di un disco di paginazione più grande;
c. aumento del grado di multiprogrammazione;

d. riduzione del grado di multiprogrammazione;
e. installazione di una maggiore quantità di memoria centrale;
f. installazione di un disco più veloce o di più controllori di unità con dischi multipli;
g. aggiunta della prepaginazione agli algoritmi di prelievo delle pagine;
h. aumento della dimensione delle pagine.

Motivate le risposte.

**9.29** Supponete che una macchina fornisca istruzioni per l'accesso alle locazioni di memoria attraverso il modello di indirizzamento indiretto a un livello. Quale sequenza di pagine mancanti si riscontra allorché tutte le pagine di un programma siano non residenti e la prima istruzione del programma sia un'operazione di caricamento indiretto dalla memoria? Che cosa succede se il sistema adopera una tecnica di allocazione dei frame per processo e soltanto due pagine siano allocate al processo in questione?

**9.30** Si supponga che il criterio di sostituzione (in un sistema paginato) consista nel controllo regolare delle pagine, una per volta, eliminando ogni pagina che non sia stata usata dopo l'ultimo controllo. Che cosa offre in più tale criterio, e che cosa in meno, se paragonato all'algoritmo di sostituzione LRU o con seconda chance?

**9.31** Un algoritmo di sostituzione delle pagine dovrebbe ridurre al minimo il numero delle assenze di pagine. Questa minimizzazione si può ottenere distribuendo in modo uniforme su tutta la memoria le pagine maggiormente usate, anziché lasciarle competere per un piccolo numero di frame. A ogni frame si può associare un contatore del numero delle pagine relative a quel frame. Quindi, per sostituire una pagina, si cerca il frame con il contatore più basso.

a. Definite un algoritmo di sostituzione delle pagine che si avvalga di questa idea di base. Affrontate in modo specifico i seguenti problemi:
   1. qual è il valore iniziale dei contatori;
   2. quando si incrementano i contatori;
   3. quando si decrementano i contatori;
   4. come si sceglie la pagina da sostituire.
b. Se sono disponibili quattro frame, dite quante assenze di pagine avvengono per l'algoritmo indicato, in relazione alla seguente successione di riferimenti:
   1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.
c. Calcolate il numero minimo di assenze di pagine per una strategia di sostituzione delle pagine ottimale per la successione di riferimenti del punto b), con quattro frame.

**9.32** Considerate un sistema di paginazione su richiesta con un disco di paginazione che abbia un tempo medio d'accesso e di trasferimento di 20 millisecondi. Gli indirizzi sono tradotti per mezzo di una tabella delle pagine che si trova in memoria centrale, con un tempo d'accesso di un microsecondo per ogni accesso alla memoria. Quindi, ogni riferimento alla memoria per mezzo della tabella delle pagine richiede due accessi. Per migliorare questo tempo, è stata aggiunta una memoria associativa che riduce il tempo d'accesso a un riferimento alla memoria, se l'elemento della tabella delle pagine si trova in memoria associativa.

Supponete che, per l'80 per cento degli accessi, l'elemento relativo si trovi in memoria associativa e che il 10 per cento dei restanti (cioè il 2 per cento del totale) causi un'assenza di pagina. Calcolate il tempo effettivo d'accesso alla memoria.

9.33 Qual è la causa della paginazione degenere? Come può il sistema accertarla? E, una volta rivelato questo problema, che cosa può fare per eliminarlo?

9.34 Chiarite se un processo possa avere due insiemi di lavoro, uno per rappresentare i dati e l'altro per rappresentare il codice.

9.35 Considerate il parametro $\Delta$ usato per definire la finestra dell'insieme di lavoro nell'ambito del modello omonimo. Impostando $\Delta$ a un valore basso, quale effetto ne deriva per la frequenza degli errori dovuti a pagine mancanti e per il numero di processi attivi (non sospesi) in esecuzione nel sistema? Qual è l'effetto quando $\Delta$ è impostato a un valore molto alto?

9.36 Ipotizzate di avere un segmento iniziale da 1024 KB allocato con il sistema buddy. Seguendo la Figura 9.27 come guida, tracciate l'albero che rappresenta l'allocazione di memoria derivante dalle richieste seguenti:

- richiesta di 240 byte;
- richiesta di 120 byte;
- richiesta di 60 byte;
- richiesta di 130 byte.

Modificate adesso l'albero in conformità ai seguenti rilasci di memoria; applicate la fusione ogni qual volta è possibile:

- rilascio di 240 byte;
- rilascio di 60 byte;
- rilascio di 120 byte.

9.37 Considerate un sistema in grado di gestire thread sia a livello utente sia a livello kernel. La rilocazione in questo sistema è di uno a uno (a ogni thread del kernel corrisponde un thread utente). Un processo a più thread consiste allora di (a) un insieme di lavoro per l'intero processo, oppure di (b) un insieme di lavoro per ciascun thread?

9.38 L'algoritmo di allocazione delle lastre (*slab*) riserva una cache a ciascun oggetto di tipo diverso. Assumendo di avere una cache per tipo di oggetto, spiegate perché il metodo si dimostra scarsamente applicabile a sistemi multiprocessore. Quale potrebbe essere la soluzione a tale problema di scalabilità?

9.39 Considerate un sistema che assegni ai propri processi pagine di dimensioni differenti. Quali vantaggi presenta tale schema di paginazione? Quali sono le modifiche da apportare al sistema di memoria virtuale per ottenere questa funzionalità?

## Problemi di programmazione

9.40 Scrivete un programma che codifichi gli algoritmi di sostituzione delle pagine FIFO e LRU descritti in questo capitolo. Generate una successione di riferimenti casuale, in cui i numeri delle pagine siano compresi tra 0 e 9. Applicate ciascun algoritmo a tale successione e registrate i rispettivi numeri delle assenze di pagine. Codificate gli algoritmi di sostituzione delle pagine in modo che il numero dei frame sia compreso tra 1 e 7. Supponete l'impiego della paginazione su richiesta.

9.41 I *numeri di Catalan* costituiscono una successione di interi $C_n$ che si incontra nei problemi di enumerazione degli alberi. I primi termini della successione, a partire da $n = 1$, sono 1, 2, 5, 14, 42, 132, .... Una formula esatta per $C_n$ è la seguente:

$$C_n = \frac{1}{(n+1)}\left(\frac{2n}{n}\right) = \frac{(2n)!}{(n+1)!n!}$$

Progettate due programmi che comunichino condividendo memoria, servendovi della API Win32, come accennato al Paragrafo 9.7.2. Il processo produttore dovrà generare la sequenza di numeri di Catalan e scriverla su un oggetto condiviso. Il processo consumatore, quindi, la leggerà, restituendo in uscita la sequenza dalla memoria condivisa. Al processo produttore sarà passato il parametro intero $n$ dalla riga di comando; se, per esempio, si passa il valore $n = 5$, il processo produttore dovrà generare i primi 5 termini della successione dei numeri di Catalan.

## 9.12 Note bibliografiche

La paginazione su richiesta è stata usata per la prima volta nel sistema operativo Atlas, realizzato per il calcolatore MUSE della Manchester University intorno al 1960 [Kilburn et al. 1961]. Un altro tra i primi sistemi di paginazione su richiesta è stato MULTICS, per il calcolatore GE 645 [Organick 1972].

[Belady et al. 1969] sono stati i primi ricercatori a osservare che la strategia di sostituzione FIFO poteva presentare l'anomalia che porta il nome di Belady. In [Mattson et al. 1970] si dimostra che gli algoritmi a pila non sono soggetti all'anomalia di Belady.

L'algoritmo di sostituzione ottimale è dovuto a [Belady 1966]. In [Mattson et al. 1970] si trova la dimostrazione che esso è ottimale. L'algoritmo ottimale di Belady si usa per l'allocazione statica; [Prieve e Fabry 1976] hanno proposto un algoritmo ottimale per situazioni in cui l'allocazione può variare.

L'algoritmo dell'orologio è trattato in [Carr e Hennessy 1981].

Il modello dell'insieme di lavoro è stato sviluppato da [Denning 1968]. Analisi relative al modello dell'insieme di lavoro sono presenti in [Denning 1980].

Lo schema di controllo della frequenza di assenze di pagine è stato sviluppato da [Wulf 1969], che ha applicato con successo la propria tecnica al calcolatore Burroughs B5500. [Gupta e Franklin 1978] fornisce un confronto delle prestazioni tra lo schema dell'insieme di lavoro e lo schema di sostituzione basato sulla frequenza delle assenze di pagine.

[Wilson et al. 1995] hanno introdotto diversi algoritmi per l'assegnazione dinamica della memoria. Varie problematiche sulla frammentazione della memoria sono esaminate da [Johnstone e Wilson 1998]. Gli allocatori di memoria con il sistema buddy sono stati descritti in [Knowlton 1965], [Peterson e Norman 1977], [Purdom Jr. e Stigler 1970]. [Bonwick 1994] ha trattato l'allocatore delle lastre e, insieme con Adams [2001] ha esteso la discussione ai processori multipli. Altri algoritmi del genere sono reperibili in [Stephenson 1983], [Bays 1977] e [Brent 1989]. Per una panoramica delle strategie di allocazione della memoria si può consultare [Wilson et al. 1995].

[Solomon e Russinovich 2000] descrivono come il sistema Windows 2000 implementi la memoria virtuale. [Mauro e McDougall 2001] trattano della memoria virtuale di Solaris. Le tecniche di memoria virtuale nei sistemi operativi Linux e BSD UNIX sono state descritte, rispettivamente, in [Bovet e Cesati 2001] e [McKusick et al. 1996]. Le caratteristiche dei sistemi con pagine di dimensioni diverse sono trattate da [Ganapathy e Schimmel 1998], e da [Navarro et al. 2002]. [Ortiz 2001] ha analizzato l'argomento della memoria virtuale integrata nei sistemi operativi in tempo reale.

[Jacob e Mudge 1998b] confrontano le implementazioni dei sistemi di memoria virtuale per le architetture MIPS, PowerPC e Pentium. Un articolo correlato, [Jacob e Mudge 1998a], descrive gli elementi dell'architettura di sistema necessari alla realizzazione della memoria virtuale in sei diversi sistemi, tra i quali l'UltraSPARC.