Ascolta

3.5 Problemi di progettazione dei sistemi di paginazione

Nei paragrafi precedenti abbiamo spiegato come funziona la paginazione fornendo un'infarinatura delle basi degli algoritmi di sostituzione della pagine. Per progettare un sistema ben funzionante c'è però molto altro da sapere. Nei paragrafi seguenti analizzeremo le altre questioni che i progettisti di sistemi operativi devono considerare attentamente al fine di ottenere buone prestazioni da un sistema di paginazione.

3.5.1 Politiche di allocazione globali e locali a confronto

Nei paragrafi precedenti abbiamo discusso parecchi algoritmi per la scelta della pagina da sostituire quando accade un page fault. Un'importante questione riguardo a questa scelta (che finora abbiamo attentamente nascosto sotto il tappeto) è come la memoria dovrebbe essere allocata fra i processi eseguibili concorrenti.

Diamo uno sguardo alla **Figura 3.22**(a). In questa figura i processi *A*, *B* e *C* sono l'insieme dei processi eseguibili. Supponiamo che *A* ottenga un page fault; l'algoritmo di sostituzione delle pagine dovrebbe cercare di trovare la pagina usata meno recentemente considerando solo le sei pagine attualmente allocate ad *A,* o dovrebbe considerare tutte le pagine in memoria? Se guardasse solo alle pagine di *A*, la pagina con l'età inferiore sarebbe *A5* e avremmo la situazione della **Figura 3.22**(b).

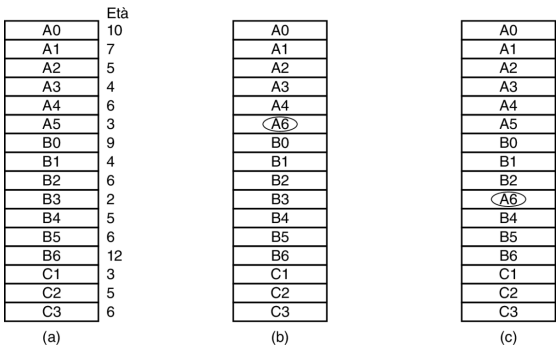

Figura 3.22 Sostituzione locale e globale delle pagine a confronto. (a) Configurazione originale. (b) Sostituzione locale delle pagine. (c) Sostituzione globale delle pagine.

Se invece la pagina da rimuovere fosse quella con il valore minore, indipendentemente dal processo al quale appartiene, la scelta cadrebbe su *B3* e avremmo la situazione della **Figura 3.22**(c). L'algoritmo della **Figura 3.22**(b) è detto "algoritmo di sostituzione **locale** delle pagine", mentre quello della **Figura 3.22**(c) è detto "algoritmo di sostituzione **globale** delle pagine". Gli algoritmi locali assegnano a ogni processo una frazione fissa della memoria. Gli algoritmi globali assegnano dinamicamente frame fra i processi eseguibili. In questo modo il numero di frame assegnato a ogni processo varia nel tempo.

Gli algoritmi globali funzionano generalmente meglio, specie quando la dimensione del set di lavoro può variare durante la vita di un processo. Se è usato un algoritmo locale e il set di lavoro aumenta si ottiene del thrashing, anche in presenza di frame liberi a sufficienza. Se il set di lavoro si restringe, gli algoritmi locali sprecano memoria. Se è utilizzato un algoritmo globale, il sistema deve continuamente decidere quanti frame assegnare a ciascun processo. Un metodo è monitorare la dimensione del set di lavoro come indicato dai bit di aging, ma non è detto che questo approccio prevenga il thrashing. Il set di lavoro può cambiare dimensione in millisecondi, mentre i bit di aging sono una misura molto grezza distribuita su un numero di cicli del clock.

?

Un altro approccio è avere un algoritmo per allocare i frame ai processi; una possibilità è determinare periodicamente il numero di processi in esecuzione e assegnare a ciascuno una parte uguale. Così, con 12.416 frame disponibili (cioè non del sistema operativo) e 10 processi, ogni processo avrà 1.241 frame. I sei restanti vanno in un pool da usare quando si verificano dei page fault.

Questo metodo può sembrare equo, ma ha poco senso assegnare parti uguali di memoria a un processo di 10 KB e a uno di 300 KB. Le pagine possono invece essere assegnate in proporzione alla dimensione totale del processo, con un processo di 300 KB che si prende 30 volte più pagine di un processo di 10 KB. È probabilmente una cosa saggia dare a ogni processo un limite minimo, in modo che possa essere eseguito indipendentemente da quanto sia piccolo. Su alcune macchine, per esempio, una singola operazione a due operandi può richiedere fino a sei pagine, poiché l'istruzione stessa, l'operando di origine e l'operando di destinazione potrebbero essere tutti a cavallo dei limiti delle pagine. Allocando sole cinque pagine, programmi contenenti queste istruzioni non possono essere eseguiti.

Se è usato un algoritmo globale è possibile avviare ogni processo con un certo numero di pagine proporzionale alla sua dimensione, ma quando l'allocazione deve essere aggiornata dinamicamente durante l'esecuzione. Un modo per gestire l'allocazione è usare l'algoritmo **PFF** (**Page Fault Frequency**), che indica quando aumentare o diminuire l'allocazione delle pagine di un processo, ma non dice nulla riguardo alla pagina da sostituire in caso di page fault. Controlla solo la dimensione del set di allocazione.

Per un'ampia classe di algoritmi di sostituzione delle pagine, incluso l'LRU, si sa che la frequenza dei page fault decresce al crescere delle pagine assegnate. Tale proprietà è illustrata nella **Figura 3.23**.

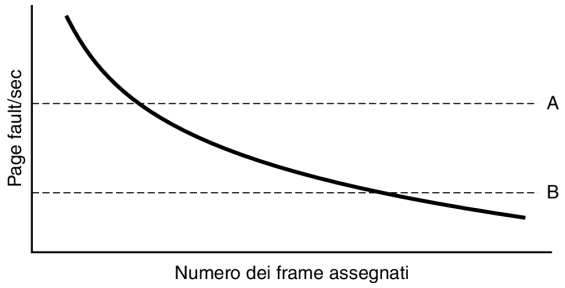

Figura 3.23 La frequenza dei page fault come funzione del numero di frame assegnati.

Misurare la frequenza dei page fault è semplice: basta contare il numero di page fault per secondo, magari tenendo una media mobile dei secondi appena trascorsi. Un sistema semplice per farlo è aggiungere il numero di page fault durante il secondo immediatamente antecedente alla media attuale di esecuzione e dividere per due. La linea tratteggiata *A* corrisponde a una frequenza di page fault troppo alta; per diminuire la frequenza dei page fault vengono dati più frame al processo che genera gli errori. La linea tratteggiata *B* corrisponde a una percentuale di page fault così bassa da supporre che il processo abbia troppa memoria, e in questo caso possono essergli tolti dei frame. Il PFF cerca quindi di tenere entro limiti accettabili la frequenza di paginazione per ciascun processo.

È importante notare che alcuni algoritmi di sostituzione delle pagine possono funzionare con una politica di sostituzione sia locale sia globale. Per esempio, il FIFO può sostituire le pagine vecchie in tutta la memoria (algoritmo globale) o le vecchie pagine di proprietà del processo attuale (algoritmo locale). In modo analogo, l'LRU o le sue varianti possono sostituire le ultime pagine meno usate recentemente in tutta la memoria (algoritmo globale) o solo quelle di proprietà del processo attuale (algoritmo locale). In alcuni casi, la scelta tra locale e globale è indipendente dall'algoritmo.

Per altri algoritmi di sostituzione delle pagine, invece, ha senso solo una strategia locale. In particolare, gli algoritmi working set e WSClock fanno riferimento a un processo specifico e devono essere applicati in quel contesto: non esiste un set di lavoro di tutta la macchina, e cercando di usare l'unione di tutti i set di lavoro si perderebbe la proprietà della località e la cosa non funzionerebbe.

3.5.2 Controllo del carico

Può succedere che nel sistema si verifichi il thrashing anche con il miglior algoritmo di sostituzione delle pagine e l'assegnazione ottimale dei frame. Infatti, se la combinazione dei set di lavoro di tutti i processi supera la capacità della memoria ci si deve aspettare del thrashing. Un sintomo di questa situazione è che l'algoritmo PFF indica che alcuni processi hanno bisogno di più memoria, ma nessun processo ha bisogno di meno memoria. In questo caso non è possibile dare più memoria ai processi che ne hanno bisogno senza ledere altri processi. L'unica soluzione reale è di sbarazzarsi temporaneamente di alcuni processi.

La soluzione più semplice è piuttosto cruda: uccidere qualche processo. Spesso i sistemi operativi dispongono di uno speciale processo detto **OOM** (**Out of Memory killer**) che si attiva quando il sistema è a corto di memoria. Analizza tutti i processi in esecuzione e sceglie una vittima sacrificale, liberandone le risorse per continuare a far funzionare il sistema. Specificamente, il killer OOM esamina tutti i processi assegnando dei voti per indicare quanto siano "cattivi". Per esempio, usare moltissima memoria fa aumentare il voto di "cattiveria" di un processo, mentre i processi importanti (come quelli root e del sistema) hanno voti bassi. Inoltre, il killer OOM cerca di ridurre al minimo i processi da uccidere, pur liberando memoria a sufficienza. Dopo aver considerato tutti i processi, ucciderà quelli con i voti più alti.

Un modo notevolmente meno violento per ridurre il numero dei processi che si contendono la memoria consiste nello spostarne qualcuno in memoria non volatile liberando tutte le pagine che detiene. Per esempio, si può spostare un processo in memoria non volatile e spartire i suoi frame fra gli altri processi in thrashing. Se il thrashing termina, il sistema può continuare così per un po'. In caso contrario si dovrà spostare un altro processo e proseguire finché il thrashing finisce. Quindi lo scambio può essere necessario anche con la paginazione, solo che lo scopo è ridurre la potenziale richiesta di memoria e non recuperare pagine. Ne consegue che scambio e paginazione non si contraddicono a vicenda.

Lo scambio dei processi per alleviare il carico di memoria è una reminiscenza dello scheduling a due livelli, in cui alcuni processi sono messi in memoria non volatile ed è usato uno scheduler a breve termine per schedulare i restanti. Chiaramente le due idee possono essere combinate, spostando solo il numero di processi sufficiente a ottenere una frequenza di page fault accettabile. Alcuni processi sono periodicamente prelevati dalla memoria non volatile e altri vi vengono spostati.

Un altro fattore da considerare è il grado di multiprogrammazione: quando il numero di processi nella memoria principale è troppo basso, la CPU può rimanere inattiva per periodi di tempo considerevoli. Nel decidere di quale processo eseguire lo scambio su disco, questo aspetto porta a considerare non solo la sua dimensione e la frequenza della paginazione, ma anche le sue caratteristiche (se è CPU bound o I/O bound) e quelle dei restanti processi.

Per concludere questo paragrafo è bene ricordare che esistono altre possibilità oltre a uccidere e spostare i processi. Ad esempio, un'altra soluzione comune consiste nel ridurre l'occupazione della memoria usando compattamento e compressione. In realtà, ridurre l'impronta nella memoria di un sistema è una priorità piuttosto alta per i progettisti di sistemi operativi di tutto il mondo. Una tecnica ingegnosa di uso comune è la **deduplicazione**, o **same page merging** (**unione nella stessa pagina**). L'idea è semplice: analizzare periodicamente la memoria per vedere se due pagine (magari anche in processi diversi) hanno esattamente lo stesso contenuto; in caso affermativo, anziché mantenere lo stesso contenuto in due frame fisici, il sistema operativo elimina uno dei due doppioni e modifica la mappatura della tabella delle pagine in modo da configurare due pagine virtuali che puntano allo stesso frame. Il frame è condiviso copy- on-write: appena un processo cerca di scrivere sulla pagina, viene generata una copia supplementare in modo che l'operazione di scrittura non coinvolga l'altra pagina. Qualcuno ha chiamato questa operazione "de-deduplicazione"; non è sbagliato, ma se proprio si vogliono inventare parole nuove varrebbe la pena fare qualche sforzo per non partorire altri mostri.

3.5.3 Policy di pulizia

Una questione correlata all'argomento del controllo del carico è quella della pulizia. L'aging funziona al meglio quando c'è abbondanza di frame di pagina liberi che possono essere richiesti quando avvengono dei page fault. Se tutti i frame sono pieni, e in più modificati, prima di poter fare entrare una nuova pagina è necessario scriverne una vecchia in memoria non volatile. Per garantire una certa abbondanza di frame di pagina, i sistemi di paginazione dispongono solitamente di un processo in background detto **paging daemon** (daemon di paginazione) che è a riposo per la maggior parte del tempo ma viene risvegliato periodicamente per ispezionare lo stato della memoria. Se i frame liberi sono troppo pochi, inizia a scegliere le pagine da sfrattare usando un algoritmo di sostituzione delle pagine. Se le pagine sono state modificate dopo il loro caricamento, vengono scritte su memoria non volatile.

In ogni caso, i precedenti contenuti della pagina vengono ricordati. Nel caso una delle pagine sfrattate sia nuovamente richiesta prima che il suo frame sia sovrascritto, può essere ripristinata togliendola dal gruppo dei frame liberi. In termini di prestazioni, avere a disposizione una buona quantità di frame di pagina conviene rispetto a occupare tutta la memoria e cercare di trovare un frame libero solo quando è necessario. Come minimo, il paging daemon garantisce che tutti i frame liberi siano puliti, e quindi non debbano essere scritti in memoria non volatile in fretta e furia quando servono.

Un modo per implementare questa policy della pulizia è un clock a due lancette, con quella anteriore controllata dal paging daemon. Quando punta a una pagina sporca, essa viene scritta nella memoria non volatile e la lancetta anteriore avanza. Quando punta a una pagina pulita, avanza senza fare altro. La lancetta posteriore è utilizzata per la sostituzione di pagine, come nel normale algoritmo clock, ma con maggiori probabilità di incontrare una pagina pulita grazie al lavoro del paging daemon.

3.5.4 Dimensione delle pagine

La dimensione delle pagine è spesso un parametro che può essere scelto dal sistema operativo. Ad esempio, anche se l'hardware è stato progettato con pagine di 4096 byte, il sistema operativo può facilmente considerare le coppie 0 e 1, 2 e 3, 4 e 5 e così via, come pagine da 8 KB, allocando due frame da 4096 byte consecutivi.

Per determinare la migliore dimensione delle pagine occorre bilanciare parecchi fattori in concorrenza fra loro, quindi non esiste un risultato adatto a qualsiasi situazione. Per cominciare, ci sono due fattori a favore delle pagine di piccole dimensioni. Un qualunque segmento di testo, dati o dello stack non riempie un numero intero di pagine. In media, metà della pagina finale è vuota e lo spazio vuoto è sprecato; si parla in questo caso di **frammentazione interna**. Con n segmenti in memoria e una dimensione di pagina di p byte, $np/2$ byte sono sprecati dalla frammentazione interna. Questo ragionamento va a favore di pagine di dimensione ridotta.

Un'altra argomentazione a favore delle pagine piccole diventa evidente se pensiamo a un programma che consiste di otto fasi sequenziali di 4 KB ciascuna. Con pagine da 32 KB, il programma deve allocare ogni volta 32 KB. Con pagine da 16 KB ha bisogno di soli 16 KB. Con pagine da 4 KB o meno, richiede solo 4 KB in ogni momento. In generale, pagine di grandi dimensioni generano più spazio inutilizzato in memoria rispetto a pagine di piccole dimensioni.

Le pagine piccole però comportano che i programmi richiedano molte pagine, ossia una tabella delle pagine estesa. Un programma da 32 KB richiede solo quattro pagine da 8 KB, ma se le pagine sono da 512 byte ce ne vorranno 64. I trasferimenti da e verso il disco o l'SSD sono generalmente di una pagina per volta. Se la memoria non volatile non è un SSD ma un disco magnetico, la maggior parte del tempo impiegato sarà il ritardo per la ricerca e la rotazione, e ciò significa che trasferire una pagina piccola potrebbe richiedere quasi lo stesso tempo di una pagina grande. Caricare 64 pagine da 512 byte richiede 64 × 10 ms, mentre per caricare 4 pagine da 8 KB servono solo 4 × 12 ms.

Aspetto forse ancora più importante, le pagine piccole occupano molto spazio prezioso nel TLB. Supponiamo che il programma utilizzi 1 MB di memoria con un set di lavoro di 64 KB. Se le pagine hanno una dimensione di 4 KB, il programma occuperà almeno 16 voci nel TLB; se le pagine sono di 2 MB, sarà sufficiente una singola voce nel TLB (in teoria, potrebbe capitare che si vogliano però separare dati e istruzioni). Poiché le voci nel TLB sono poche e fondamentali per migliorare le prestazioni, quando è possibile conviene utilizzare pagine di grandi dimensioni. Per trovare un equilibrio tra tutti questi compromessi, i sistemi operativi utilizzano a volte pagine di dimensioni diverse per le diverse parti del sistema, per esempio pagine grandi per il kernel e più piccole per i processi utente. In effetti, alcuni sistemi operativi fanno salti mortali per usare pagine grandi, anche spostando di qua e di là la memoria di un processo per trovare o creare intervalli di memoria contigua adatti a una pagina grande; questa caratteristica è detta a volte **THP** (**Transparent Huge Pages**, pagine enormi trasparenti).

In alcune macchine, è necessario che il sistema operativo carichi nei registri hardware la tabella delle pagine ogni volta che la CPU passa da un processo a un altro. Su queste macchine, una dimensione delle pagine ridotta significa che il tempo richiesto per caricare i registri delle pagine aumenta con il ridursi della dimensione delle pagine. Inoltre, anche lo spazio occupato dalla tabella delle pagine aumenta al diminuire della dimensione delle pagine.

Quest'ultimo punto può essere analizzato da un punto di vista matematico. Supponiamo che la dimensione media del processo sia di s byte e la dimensione della pagina sia di p byte. Supponiamo inoltre che ogni voce della tabella delle pagine richieda e byte. Il numero di pagine necessario per processo è all'incirca s/p, che occupa se/p byte di spazio nella tabella delle pagine. La memoria sprecata dalla frammentazione interna nell'ultima pagina del processo è $p/2$. L'overhead totale dovuto alla tabella delle pagine e alla perdita a causa della frammentazione interna è dato dalla somma dei due termini seguenti:

$$\text{overhead} = se/p + p/2$$

Il primo termine (la dimensione della tabella delle pagine) è grande quando la dimensione della pagina è piccola. Il secondo termine (la frammentazione interna) è grande quando la dimensione della pagina è grande. L'ottimo sta da qualche parte nel mezzo. Prendendo la derivata prima rispetto a p e uguagliandola a 0, abbiamo l'equazione

$$-se/p^2 + 1/2 = 0$$

Da questa equazione si può derivare la formula che dà la dimensione di pagina ottimale (considerando solo la memoria sprecata da frammentazione e dimensione della tabella delle pagine). Il risultato è:

$$p = \sqrt{2se}$$

Con s = 1 MB ed e = 8 byte per voce di tabella delle pagine, la dimensione ottimale delle pagine è di 4 KB. I computer disponibili in commercio hanno usato dimensioni di pagina che vanno dai 512 byte ai 64 KB. Un valore tipico è 1 KB, ma attualmente 4 KB è più comune.

3.5.5 Istruzioni separate e spazi dei dati

La maggior parte dei computer ha un solo spazio degli indirizzi contenente sia i programmi sia i dati, come illustrato nella **Figura 3.24**(a). Se questo spazio degli indirizzi è abbastanza grande, tutto funziona bene. Spesso tuttavia è troppo piccolo, e costringe i programmatori a fare i salti mortali per riuscire a farci stare tutto.

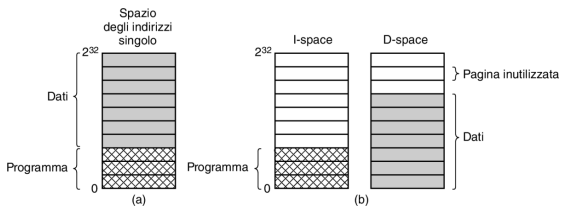

Figura 3.24 (a) Un solo spazio degli indirizzi. (b) I-space e D-space separati.

Una soluzione, adottata la prima volta sul PDP-11 a 16 bit, è separare lo spazio degli indirizzi delle istruzioni (testo del programma) da quello dei dati, chiamandoli rispettivamente **I-space** e **D-space** come mostrato nella **Figura 3.24**(b). Ogni spazio degli indirizzi va da 0 a un certo massimo, tipicamente $2^{16} - 1$ o $2^{32} - 1$. Il *linker* deve sapere quando sono utilizzati I-space e D-space separatamente, poiché in questo caso i dati sono riposizionati all'indirizzo virtuale 0 anziché partire dopo il programma.

In un computer così progettato, entrambi gli spazi degli indirizzi possono essere paginati indipendentemente l'uno dall'altro. Ognuno ha la sua tabella delle pagine, con la sua mappatura delle pagine virtuali sui frame fisici. Quando l'hardware vuole prelevare un'istruzione sa che deve usare l'I-space e la tabella delle pagine dell'I-space. Analogamente, i riferimenti ai dati devono passare attraverso la tabella delle pagine del D-space. A parte questa distinzione, gli I-space e D-space separati non creano altre complicazioni particolari, e lo spazio degli indirizzi disponibile raddoppia.

Oggi gli spazi degli indirizzi sono ampi, ma un tempo la loro dimensione era un problema molto serio. Gli I-space e D-space separati sono comunque utilizzati a tutt'oggi, ma non per i normali spazi degli indirizzi bensì per suddividere la cache di primo livello (L1): la memoria nella cache L1 è a tutt'oggi molto scarsa. Infatti, osservando alcuni processori scopriamo che il TLB è a sua volta partizionato in L1 e L2 e che il TLB L1 è ulteriormente suddiviso in un TLB per le istruzioni e un TLB per i dati.

3.5.6 Pagine condivise

Un altro problema progettuale è la condivisione. In un grande sistema multiprogrammato è normale che molti utenti eseguano lo stesso programma nel medesimo istante. Anche un singolo utente potrebbe eseguire più programmi che utilizzano la stessa libreria. Per evitare che due copie della stessa pagina siano in memoria contemporaneamente è ovviamente più efficiente condividere le pagine. Il problema è che non tutte le pagine sono condivisibili. In particolare, è possibile condividere le pagine di sola lettura (read-only), come il testo dei programmi, ma non le pagine dei dati.

Se sono supportati I-space e D-space separati è relativamente semplice condividere i programmi, facendo in modo che due o più processi usino la stessa tabella delle pagine per i loro I-space, ma tabelle delle pagine diverse per i loro D-space. In genere, in un'implementazione che supporta la condivisione in questo modo, le tabelle delle pagine sono strutture dati indipendenti dalla tabella dei processi. Nella tabella dei processi di ogni processo ci sono quindi due puntatori: uno alla tabella delle pagine dell'I-space e uno alla tabella delle pagine del D-space, come illustrato nella **Figura 3.25**. Quando lo scheduler sceglie un processo da eseguire, usa questi puntatori per determinare le tabelle delle pagine corrette e imposta l'MMU usandole. I processi possono condividere i programmi (e talvolta le librerie) anche senza I-space e D-space separati, ma il meccanismo è più complicato.

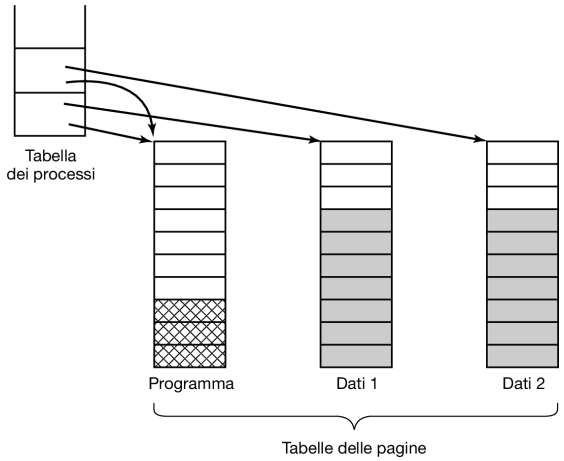

Figura 3.25 Due processi che condividono lo stesso programma condividendo le sue tabelle delle pagine.

Quando due o più processi condividono del codice, si verifica un problema con le pagine condivise. Supponiamo che i processi *A* e *B* stiano entrambi eseguendo un editor e condividano le sue pagine. Se lo scheduler decide di rimuovere *A* dalla memoria, lo sfratto di tutte le sue pagine per riempire i frame vuoti con qualche altro programma obbligherà *B* a generare un gran numero di page fault per riportarle in memoria.

Analogamente, quando *A* termina, è fondamentale poter scoprire se le pagine sono ancora in uso, in modo che il loro spazio su memoria non volatile non sia liberato accidentalmente. Cercare in tutte le tabelle delle pagine per vedere se una pagina è condivisa è troppo dispendioso, quindi per tenere traccia delle pagine condivise servono strutture dati speciali, specialmente se l'unità di condivisione è la singola pagina (o una serie di pagine) e non un'intera tabella delle pagine.

La condivisone dei dati è più complessa di quella del codice, ma non impossibile. In particolare, in UNIX, dopo una chiamata di sistema fork, genitore e figlio devono condividere sia il testo del programma sia i dati. In un sistema paginato, spesso si dà a ciascuno di questi processi una propria tabella delle pagine e si fa in modo che entrambi puntino allo stesso set di pagine. Di conseguenza non viene creata alcuna copia delle pagine al momento della fork, ma tutte le pagine dati sono mappate in entrambi i processi come READ-ONLY.

Finché entrambi i processi leggono solo i loro dati, senza modificarli, questa situazione può continuare. Appena un processo aggiorna una parola di memoria, la violazione della protezione di sola lettura causa una trap al sistema operativo. Viene quindi creata una copia della pagina che ha generato il page fault, in modo che ogni processo ne abbia una copia privata. Entrambe le copie sono impostate adesso come READ/WRITE, così le successive scritture su una qualsiasi delle due vengono eseguite senza trap. Questa strategia comporta che le pagine che non vengono mai modificate (incluse tutte le pagine del programma) non vengano mai copiate: è necessario copiare solo le pagine dati effettivamente modificate. Tale approccio, chiamato **copy on write** (copia in caso di scrittura), migliora le prestazioni riducendo le copie.

3.5.7 Librerie condivise

La condivisione può essere a una granularità diversa dalle singole pagine. Se un programma è avviato due volte, molti sistemi operativi condivideranno automaticamente tutte le pagine del testo in modo che ve ne sia in memoria una sola copia. Le pagine di testo sono sempre in sola lettura, per cui in questo caso non c'è alcun problema. A seconda del sistema operativo, ciascun processo può avere la sua copia privata delle pagine dei dati, oppure esse sono condivisibili e marcate come di sola lettura. Se un qualche processo modifica una pagina di dati, ne viene creata una copia privata a suo uso, ossia è applicato il "copy on write".

Nei sistemi moderni esistono molte grandi librerie usate da tanti processi, come ad esempio quelle di I/O e grafiche. Statisticamente, collegare tutte queste librerie a ogni programma eseguibile su memoria non volatile le renderebbe ancor più ingombranti di quanto già siano.

Una tecnica comune è quindi quella di usare librerie condivise (chiamate in Windows **DLL** o **Dynamic Link Library**, librerie a collegamento dinamico). Per chiarire l'idea di una libreria condivisa, consideriamo prima il collegamento tradizionale. Quando viene eseguito il linking di un programma, nel comando inviato al linker sono indicati uno o più file oggetto ed eventualmente alcune librerie, come nel comando UNIX

```
ld *.o -lc -lm
```

che collega tutti i file *.*o* (oggetto) nella directory corrente ed esegue poi la scansione di due librerie, */usr/lib/libc.a* e */usr/lib/libm.a*. Tutte le funzioni chiamate nei file oggetto ma non presenti (per esempio *printf*) sono chiamate **riferimenti esterni non definiti** e sono ricercate nelle librerie. Se trovate, sono incluse nel binario eseguibile. Ogni funzione da esse chiamata ma non ancora presente diventa un riferimento esterno non definito. Per esempio, *printf* ha bisogno di *write*, così se *write* non è già inclusa, il linker la cercherà e una volta trovata la includerà. Quando il linker ha finito, viene scritto su memoria non volatile un file binario eseguibile contenente tutte le funzioni necessarie. Le funzioni delle librerie che non sono state chiamate sono escluse; quando il programma viene caricato in memoria ed eseguito ha tutte le funzioni necessarie, quelle non necessarie sono escluse.

Supponiamo che i normali programmi usino 20-50 MB tra grafica e funzioni di interfaccia utente. Collegare staticamente centinaia di programmi con tutte queste librerie sprecherebbe una quantità significativa di spazio in memoria non volatile e di spazio nella RAM quando fossero caricate, poiché il sistema non avrebbe modo di sapere che può condividerle. E qui entrano in scena le librerie condivise. Quando un programma è collegato con librerie condivise (leggermente diverse da quelle statiche), invece di includere la funzione effettivamente chiamata, il linker include una piccola routine stub che si lega alla funzione chiamata al momento dell'esecuzione. A seconda del sistema e dei particolari della configurazione, le librerie condivise possono essere caricate quando viene caricato il programma o quando le funzioni che contengono vengono richieste per la prima volta. Naturalmente, se un altro programma ha già caricato la libreria condivisa non serve caricarla nuovamente: il punto fondamentale è proprio questo. Notate che quando una libreria condivisa è caricata o utilizzata, non viene letta in memoria tutta insieme; le pagine vengono paginate in memoria man mano che sono necessarie, e le funzioni non chiamate non sono portate nella RAM.

Oltre a creare file eseguibili più piccoli e risparmiare spazio nella memoria, le librerie condivise hanno un altro vantaggio: se una funzione contenuta nella libreria condivisa viene aggiornata per correggere un errore, non serve ricompilare il programma che la richiama. I vecchi binari eseguibili continuano a funzionare. Questa caratteristica è particolarmente importante per il software commerciale, il cui codice sorgente non è distribuito ai clienti. Per esempio, se Microsoft trova e corregge un errore di sicurezza in qualche DLL, Windows Update scaricherà la nuova DLL per sostituire la vecchia, e i programmi useranno automaticamente la nuova versione alla prossima esecuzione.

Le librerie condivise però hanno un piccolo problema da risolvere, illustrato nella **Figura 3.26**. Sono mostrati due processi che condividono una libreria di 20 KB di dimensione (supponendo che ogni rettangolo sia 4 KB). La libreria però è posizionata a indirizzi diversi per ciascun processo, presumibilmente perché i programmi stessi non hanno la medesima dimensione. Nel processo 1 la libreria inizia all'indirizzo 36K; nel processo 2 a 12K. Supponiamo che il primo passo della prima istruzione della libreria sia saltare all'indirizzo 16 nella libreria. Se questa non fosse condivisa, potrebbe essere riposizionata al volo durante il caricamento in modo che il salto (del processo 1) potrebbe essere all'indirizzo virtuale 36K + 16. Notate che l'indirizzo fisico nella RAM in cui è posta la libreria non ha importanza, poiché tutte le pagine sono mappate dagli indirizzi virtuali a quelli fisici tramite l'hardware MMU.

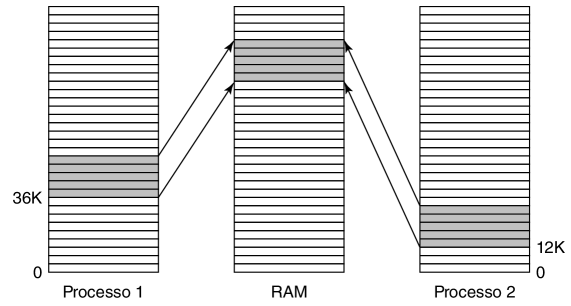

Figura 3.26 Una libreria condivisa usata da due processi.

Il riposizionamento al volo però non può funzionare, poiché la libreria è condivisa. Dopotutto quando il processo 2 chiama la prima funzione (all'indirizzo 12K), l'istruzione di salto deve portare a 12K + 16, non a 36K + 16. Questo è il piccolo problema. Un modo per risolverlo è di ricorrere al "copy on write" e creare nuove pagine per ciascun processo che condivide la libreria, riposizionandole al volo man mano che vengono create; naturalmente, però, questo schema contraddice lo scopo della condivisione della libreria.

Una soluzione migliore consiste nel compilare le librerie condivise con un contrassegno speciale che indichi al compilatore di non produrre istruzioni che usino indirizzi assoluti, ma solo istruzioni che usino indirizzi relativi. Per esempio, c'è quasi sempre un'istruzione che indica di saltare avanti (o indietro) di *n* byte (anziché un'istruzione che

indica un indirizzo specifico verso cui saltare). Questa istruzione funziona correttamente a prescindere dalla posizione della libreria condivisa nello spazio virtuale degli indirizzi. Evitando gli indirizzi assoluti, il problema è risolvibile. Il codice che utilizza solo offset relativi è detto **codice indipendente dalla posizione**.

3.5.8 File mappati

Le librerie condivise rappresentano in realtà un caso speciale di un concetto più generale, chiamata **memory-mapped file** (file mappati in memoria). L'idea è che un processo può inviare una chiamata di sistema per mappare un file all'interno di una porzione di un suo spazio degli indirizzi virtuali. Nella maggior parte delle implementazioni, al momento della mappatura non viene caricata alcuna pagina, ma appena le pagine vengono toccate sono paginate a richiesta una alla volta, utilizzando il file in memoria non volatile come archivio di appoggio. Quando il processo termina o elimina esplicitamente la mappatura, tutte le pagine modificate vengono riscritte sul file.

I file mappati forniscono un modello alternativo per l'I/O. Anziché eseguire letture e scritture, si può accedere al file come a un grande array di caratteri in memoria. In alcune situazioni i programmatori trovano che questo modello sia più comodo.

Se due o più processi mappano lo stesso file nel medesimo momento, possono comunicare attraverso la memoria condivisa. Le scritture eseguite da un processo nella memoria condivisa sono immediatamente visibili dall'altro. Questo meccanismo fornisce così un canale ad ampia banda di comunicazione fra i processi ed è spesso come tale (anche al punto da mappare un file provvisorio). Dovrebbe ora essere chiaro che, se i file mappati in memoria sono disponibili, le librerie condivise possono usare questo meccanismo.