Parte terza

Gestione della memoria

Scopo principale di un sistema di calcolo è eseguire programmi; durante l'esecuzione, i programmi e i dati cui essi accedono devono trovarsi almeno parzialmente in memoria centrale.

Per migliorare l'utilizzo della CPU e la velocità con cui essa risponde ai suoi utenti, il calcolatore deve tenere in memoria parecchi processi. Esistono molti metodi di gestione della memoria e l'efficacia di ciascun algoritmo dipende dalla situazione. La scelta di un metodo di gestione della memoria, per un sistema specifico, dipende da molti fattori, in particolar modo dall'*architettura* del sistema; ogni algoritmo richiede infatti uno specifico supporto hardware.

Capitolo 8

Memoria centrale

OBIETTIVI

- **Descrizione dettagliata delle diverse organizzazioni dei dispositivi per la gestione della memoria.**
- **Analisi delle diverse tecniche di gestione della memoria, comprese paginazione e segmentazione.**
- **Descrizione accurata di Intel Pentium, che supporta sia la segmentazione pura sia la segmentazione con paginazione.**

Nel Capitolo 5 si spiega come sia possibile condividere la CPU tra un insieme di processi. Uno dei risultati dello scheduling della CPU consiste nella possibilità di migliorare sia l'utilizzo della CPU sia la rapidità con cui il calcolatore risponde ai propri utenti; per ottenere questo aumento delle prestazioni occorre tenere in memoria parecchi processi: la memoria deve, cioè, essere *condivisa*.

In questo capitolo si presentano i diversi metodi di gestione della memoria. Gli algoritmi di gestione della memoria variano dal metodo iniziale sulla nuda macchina ai metodi di paginazione e segmentazione. Ogni metodo presenta vantaggi e svantaggi. La scelta di un metodo specifico di gestione della memoria dipende da molti fattori, in particolar modo dall'architettura del sistema, infatti molti algoritmi richiedono specifiche caratteristiche dell'architettura; progetti recenti integrano in modo molto stretto hardware e il sistema operativo.

8.1 Introduzione

Come abbiamo visto nel Capitolo 1, la memoria è fondamentale nelle operazioni di un moderno sistema di calcolo; consiste in un ampio vettore di parole o byte, ciascuno con il proprio indirizzo. La CPU preleva le istruzioni dalla memoria sulla base del contenuto del contatore di programma; tali istruzioni possono determinare ulteriori letture (*load*) e scritture (*store*) in specifici indirizzi di memoria.

Un tipico ciclo d'esecuzione di un'istruzione, per esempio, prevede che l'istruzione sia prelevata dalla memoria; decodificata (e ciò può comportare il prelievo di operandi dalla memoria) ed eseguita sugli eventuali operandi; i risultati si possono salvare in memoria. La memoria *vede* soltanto un flusso d'indirizzi di memoria, e non *sa* come sono generati (contatore di programma, indicizzazione, riferimenti indiretti, indirizzamenti immediati e così

via), oppure a che cosa servano (istruzioni o dati). Di conseguenza, è possibile ignorare *come* un programma genera un indirizzo di memoria, e prestare attenzione solo alla sequenza degli indirizzi di memoria generati dal programma in esecuzione.

L'analisi che sviluppiamo nel seguito comprende una panoramica degli aspetti basilari dell'architettura, dagli spazi di indirizi logici e fisici agli indirizzi fisici reali, offrendo una loro analisi comparativa. Seguono temi quali caricamento dinamico, collegamento e condivisione di librerie.

8.1.1 Dispositivi essenziali

La memoria centrale e i registri incorporati nel processore sono le sole aree di memorizzazione a cui la CPU può accedere direttamente. Vi sono istruzioni macchina che accettano gli indirizzi di memoria come argomenti, ma nessuna accetta gli indirizzi del disco. Pertanto, qualsiasi istruzione in esecuzione, e tutti i dati utilizzati dalle istruzioni, devono risiedere in uno di questi dispositivi per la memorizzazione ad accesso diretto. I dati che non sono in memoria devono essere caricati prima che la CPU possa operare su di loro.

I registri incorporati nella CPU sono accessibili, in genere, nell'arco di un ciclo dell'orologio di sistema. Molte CPU sono capaci di decodificare istruzioni ed effettuare semplici operazioni sui contenuti dei registri alla velocità di una o più operazioni per ciclo. Ciò non vale per la memoria centrale, cui si accede attraverso una transazione sul bus della memoria. Nei casi in cui l'accesso alla memoria richieda molti cicli d'orologio, il processore entra necessariamente in **stallo** (*stall*), poiché manca dei dati richiesti per completare l'istruzione che sta eseguendo. Questa situazione è intollerabile, perché gli accessi alla memoria sono frequenti. Il rimedio consiste nell'interposizione di una memoria veloce tra CPU e memoria centrale. Un buffer di memoria, detto **cache**, è in grado di conciliare le differenti velocità (Paragrafo 1.8.3).

Non basta prestare attenzione alle velocità relative di accesso alla memoria fisica: è anche necessario proteggere il sistema operativo dall'accesso dei processi utenti, e salvaguardare i processi utenti l'uno dall'altro. Tale protezione deve essere messa in atto a livello dei dispositivi; come si vedrà lungo tutto il capitolo, essa può essere realizzata con meccanismi diversi. In questo paragrafo evidenziamo una delle possibili implementazioni.

Innanzitutto, bisogna assicurarsi che ciascun processo abbia uno spazio di memoria separato. A tal fine, occorre poter determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi. Si può implementare il meccanismo di protezione tramite due registri, detti **registri base** e **registri limi-**

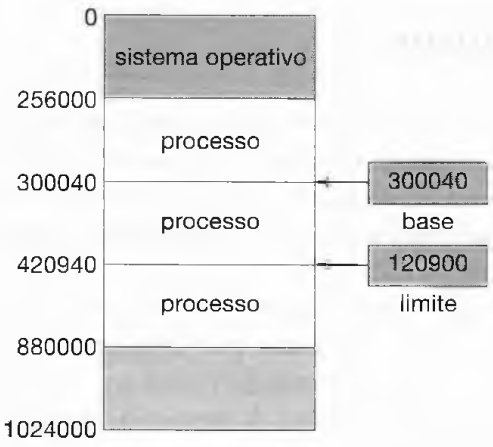


Figura 8.1 I registri base e limite definiscono lo spazio degli indirizzi logici.

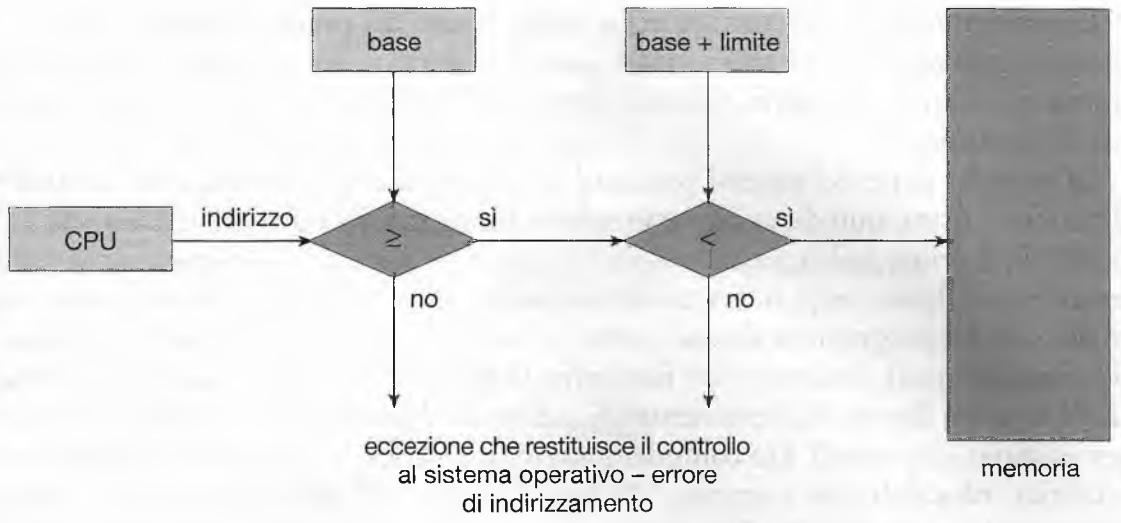


Figura 8.2 Protezione hardware degli indirizzi tramite registri base e limite.

te, come illustrato nella Figura 8.1. Il registro base contiene il più piccolo indirizzo legale della memoria fisica; il registro limite determina la dimensione dell'intervallo ammesso. Ad esempio, se i registri base e limite contengono rispettivamente i valori 300040 e 120900, al programma si consente l'accesso alle locazioni di memoria di indirizzi compresi tra 300040 e 420939, estremi inclusi.

Per mettere in atto il meccanismo di protezione, la CPU confronta *ciascun* indirizzo generato in modalità utente con i valori contenuti nei due registri. Qualsiasi tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di memoria riservata ad altri utenti comporta l'invio di un segnale di eccezione che restituisce il controllo al sistema operativo che, a sua volta, interpreta l'evento come un errore fatale (Figura 8.2). Questo schema impedisce a qualsiasi programma utente di alterare (accidentalmente o intenzionalmente) il codice o le strutture dati, sia del sistema operativo sia degli altri utenti.

Solo il sistema operativo può caricare i registri base e limite, grazie a una speciale istruzione privilegiata. Dal momento che le istruzioni privilegiate possono essere eseguite unicamente nella modalità di sistema, e poiché solo il sistema operativo può essere eseguito in tale modalità, tale schema gli consente di modificare il valore di questi registri, ma impedisce la medesima operazione ai programmi utenti.

Grazie all'esecuzione nella modalità di sistema, il sistema operativo ha la possibilità di accedere indiscriminatamente sia alla memoria a esso riservata sia a quella riservata agli utenti. Questo privilegio consente al sistema di caricare i programmi utenti nelle aree di memoria a loro riservate; di generare copie del contenuto di queste regioni di memoria (*dump*) a scopi diagnostici, qualora si verifichino errori; di modificare i parametri delle chiamate di sistema, e così via.

8.1.2 Associazione degli indirizzi

In genere un programma risiede in un disco in forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria e inserito all'interno di un processo. Secondo il tipo di gestione della memoria adoperato, durante la sua esecuzione, il processo si può trasferire dalla memoria al disco e viceversa. L'insieme dei processi presenti nei dischi e che attendono d'essere trasferiti in memoria per essere eseguiti forma la **coda d'ingresso** (*input queue*).

La procedura normale consiste nello scegliere uno dei processi appartenenti alla coda d'ingresso e nel caricarlo in memoria. Il processo durante l'esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina, si dichiara disponibile il suo spazio di memoria.

La maggior parte dei sistemi consente ai processi utenti di risiedere in qualsiasi parte della memoria fisica, quindi, anche se lo spazio d'indirizzi del calcolatore comincia all'indirizzo 00000, il primo indirizzo del processo utente non deve necessariamente essere 00000. Quest'assetto influisce sugli indirizzi che un programma utente può usare. Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari stadi, alcuni dei quali possono essere facoltativi (Figura 8.3), in cui gli indirizzi sono rappresentabili in modi diversi. Generalmente gli indirizzi del programma sorgente sono simbolici (per esempio, contatore). Un compilatore di solito **associa** (*bind*) questi indirizzi simbolici a indirizzi rilocabili (per esempio, "14 byte dall'inizio di questo modulo"). L'editor dei collegamenti (*linkage editor*), o il caricatore (*loader*), fa corrispondere a sua volta questi indirizzi rilocabili a indirizzi assoluti (per esempio, 74014). Ogni associazione rappresenta una corrispondenza da uno spazio d'indirizzi a un altro.

Generalmente, l'associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi fase del seguente percorso.

- **Compilazione.** Se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare **codice assoluto**. Se, per esempio, è noto a priori che un processo utente inizia alla locazione *r*, anche il codice generato dal compilatore comincia da quella locazione. Se, in un momento successivo, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice. I programmi per MS-DOS nel formato identificato dall'estensione .COM sono collegati al tempo di compilazione.
- **Caricamento.** Se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare **codice rilocabile**. In questo caso si ritarda l'associazione finale degli indirizzi alla fase del caricamento. Se l'indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato.
- **Esecuzione.** Se durante l'esecuzione il processo può essere spostato da un segmento di memoria a un altro, si deve ritardare l'associazione degli indirizzi fino alla fase d'esecuzione. Per realizzare questo schema sono necessarie specifiche caratteristiche dell'architettura; questo argomento è trattato nel Paragrafo 8.1.3. La maggior parte dei sistemi operativi d'uso generale impiega questo metodo.

Una gran parte di questo capitolo è dedicata alla spiegazione di come i vari tipi di associazione degli indirizzi si possano realizzare efficacemente in un calcolatore e, inoltre, alla discussione delle caratteristiche dell'architettura appropriate alla realizzazione di queste funzioni.

8.1.3 Spazi di indirizzi logici e fisici a confronto

Un indirizzo generato dalla CPU di solito si indica come **indirizzo logico**, mentre un indirizzo visto dall'unità di memoria, cioè caricato nel **registro dell'indirizzo di memoria** (*memory address register*, MAR) di solito si indica come **indirizzo fisico**.

I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Con i metodi di associazione nella fase d'esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. In questo caso ci si riferisce,

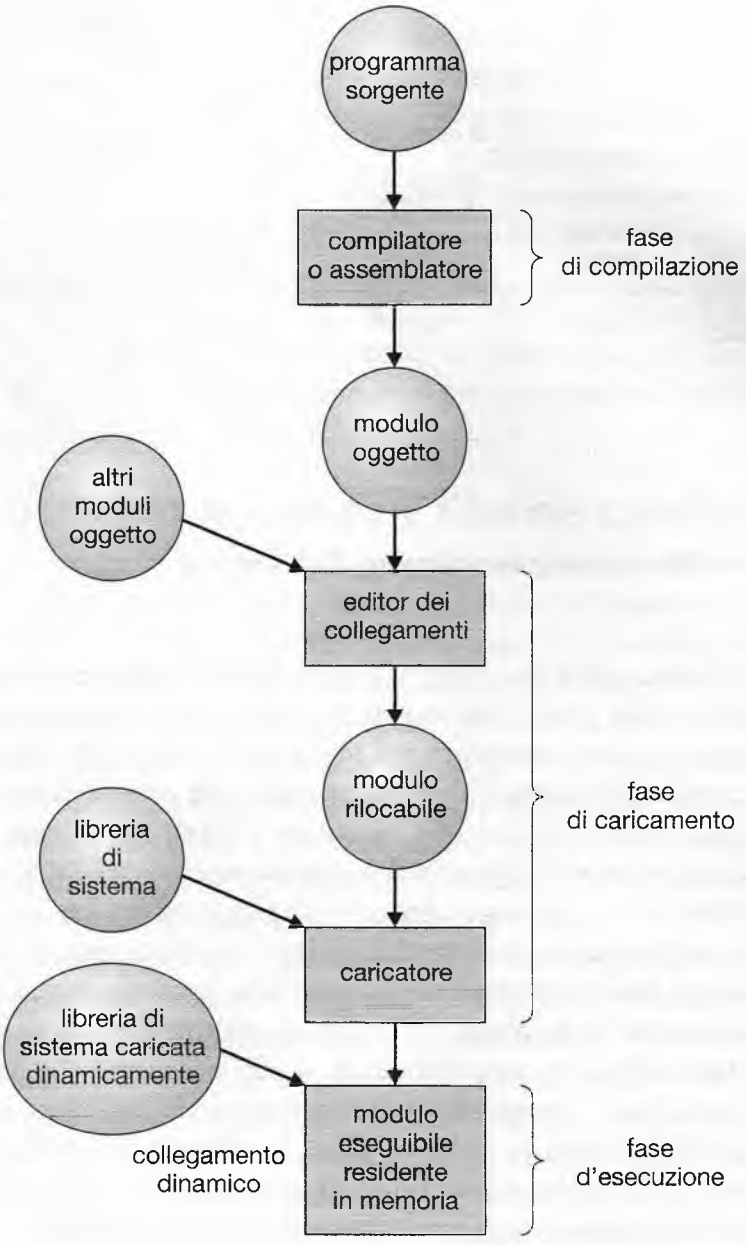


Figura 8.3 Fasi di elaborazione per un programma utente.

di solito, agli indirizzi logici col termine **indirizzi virtuali**; in questo testo si usano tali termini in modo intercambiabile. L'insieme di tutti gli indirizzi logici generati da un programma è lo **spazio degli indirizzi logici**; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è lo **spazio degli indirizzi fisici**. Quindi, con lo schema di associazione degli indirizzi nella fase d'esecuzione, lo spazio degli indirizzi logici differisce dallo spazio degli indirizzi fisici.

L'associazione nella fase d'esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto **unità di gestione della memoria** (*memory-management unit*, MMU). Come si illustra nei Paragrafi dal 8.3 al 8.7, si può scegliere tra diversi metodi di realizzazione di tale associazione; di seguito s'illustra un semplice schema di associazione degli indirizzi che impiega una MMU; si tratta di una generalizzazione dello schema con registro di base descritto nel Paragrafo 8.1.1.

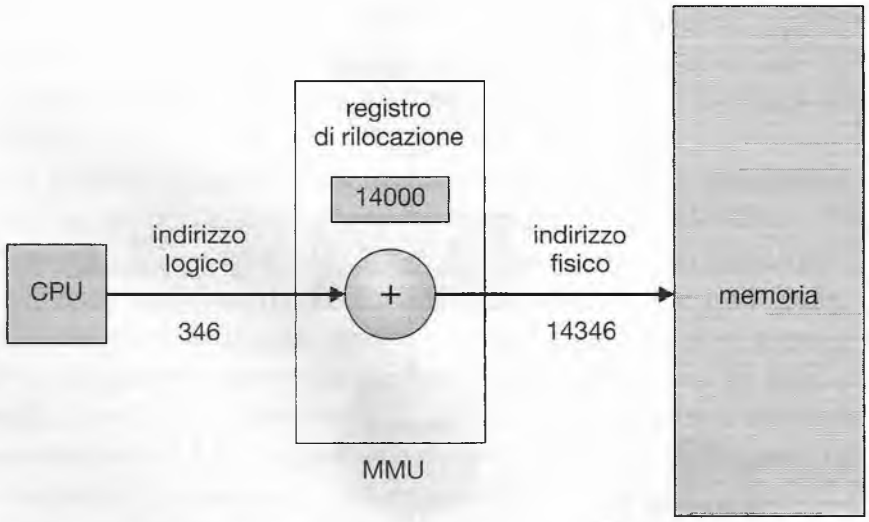


Figura 8.4 Rilocazione dinamica tramite un registro di rilocazione.

Com'è illustrato nella Figura 8.4, il registro di base è ora denominato **registro di rilocazione**: quando un processo utente genera un indirizzo, prima dell'invio all'unità di memoria, si *somma* a tale indirizzo il valore contenuto nel registro di rilocazione. Ad esempio, se il registro di rilocazione contiene il valore 14000, un tentativo da parte dell'utente di accedere alla locazione 0 è dinamicamente rilocato alla locazione 14000; un accesso alla locazione 346 corrisponde alla locazione 14346. Quando il sistema operativo MS-DOS, eseguito sulla famiglia di CPU Intel 80*x*86, carica ed esegue processi impiega quattro registri di rilocazione.

Il programma utente non considera mai gli indirizzi fisici *reali*. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò semplicemente come un numero. Solo quando assume il ruolo di un indirizzo di memoria (magari in una load o una store indiretta), si riloca il numero sulla base del contenuto del registro di rilocazione. Il programma utente tratta indirizzi *logici*, l'architettura del sistema converte gli indirizzi logici in indirizzi *fisici*. Il collegamento nella fase d'esecuzione è trattato nel Paragrafo 8.1.2. La locazione finale di un riferimento a un indirizzo di memoria non è determinata finché non si compie effettivamente il riferimento.

In questo caso esistono due diversi tipi di indirizzi: gli indirizzi logici (nell'intervallo da 0 a *max*) e gli indirizzi fisici (nell'intervallo da $r + 0$ a $r + max$ per un valore di base r). L'utente genera solo indirizzi logici e *pensa* che il processo sia eseguito nelle posizioni da 0 a *max*. Il programma utente fornisce indirizzi logici che, prima d'essere usati, si devono far corrispondere a indirizzi fisici.

Il concetto di *spazio d'indirizzi logici* associato a uno *spazio d'indirizzi fisici* separato è fondamentale per una corretta gestione della memoria.

8.1.4 Caricamento dinamico

Per migliorare l'utilizzo della memoria si può ricorrere al **caricamento dinamico** (*dynamic loading*), mediante il quale si carica una procedura solo quando viene richiamata; tutte le procedure si tengono in memoria secondaria in un formato di caricamento rilocabile. Si carica il programma principale in memoria e quando, durante l'esecuzione, una procedura deve richiamarne un'altra, si controlla innanzitutto che sia stata caricata, altrimenti si richiama il caricatore di collegamento rilocabile per caricare in memoria la procedura richiesta e ag-

giornare le tabelle degli indirizzi del programma in modo che registrino questo cambiamento. A questo punto il controllo passa alla procedura appena caricata.

Il vantaggio dato dal caricamento dinamico consiste nel fatto che una procedura che non si adopera non viene caricata. Questo metodo è utile soprattutto quando servono grandi quantità di codice per gestire casi non frequenti, per esempio le procedure di gestione degli errori. In questi casi, anche se un programma può avere dimensioni totali elevate, la parte effettivamente usata, e quindi effettivamente caricata, può essere molto più piccola.

Il caricamento dinamico non richiede un intervento particolare del sistema operativo. Spetta agli utenti progettare i programmi in modo da trarre vantaggio da un metodo di questo tipo. Il sistema operativo può tuttavia aiutare il programmatore fornendo librerie di procedure che realizzano il caricamento dinamico.

8.1.5 Collegamento dinamico e librerie condivise

La Figura 8.3 mostra anche le **librerie collegate dinamicamente**. Alcuni sistemi operativi consentono solo il **collegamento statico**, in cui le librerie di sistema del linguaggio sono trattate come qualsiasi altro modulo oggetto e combinate dal caricatore nell'immagine binaria del programma. Il concetto di collegamento dinamico è analogo a quello di caricamento dinamico. Invece di differire il caricamento di una procedura fino al momento dell'esecuzione, si differisce il collegamento. Questa caratteristica si usa soprattutto con le librerie di sistema, per esempio le librerie di procedure del linguaggio. Senza questo strumento tutti i programmi di un sistema dovrebbero disporre all'interno dell'immagine eseguibile di una copia della libreria di linguaggio (o almeno delle procedure cui il programma fa riferimento). Tutto ciò richiede spazio nei dischi e in memoria centrale.

Con il collegamento dinamico, invece, per ogni riferimento a una procedura di libreria s'inserisce all'interno dell'immagine eseguibile una piccola porzione di codice di riferimento (*stub*), che indica come localizzare la giusta procedura di libreria residente in memoria o come caricare la libreria se la procedura non è già presente. Durante l'esecuzione, il codice di riferimento controlla se la procedura richiesta è già in memoria, altrimenti provvede a caricarla; in ogni caso tale codice sostituisce se stesso con l'indirizzo della procedura, che viene poi eseguita. In questo modo, quando si raggiunge nuovamente quel segmento del codice, si esegue direttamente la procedura di libreria, senza costi aggiuntivi per il collegamento dinamico. Con questo metodo tutti i processi che usano una libreria del linguaggio si limitano a eseguire la stessa copia del codice della libreria.

Questa caratteristica si può estendere anche agli aggiornamenti delle librerie, per esempio, la correzione di errori. Una libreria si può sostituire con una nuova versione, e tutti i programmi che fanno riferimento a quella libreria usano automaticamente quella. Senza il collegamento dinamico tutti i programmi di questo tipo devono subire una nuova fase di collegamento per accedere alla nuova libreria. Affinché i programmi non eseguano accidentalmente nuove versioni di librerie incompatibili, sia nel programma sia nella libreria si inserisce un'informazione relativa alla versione. È possibile caricare in memoria più di una versione della stessa libreria, ciascun programma si serve dell'informazione sulla versione per decidere quale copia debba usare. Se le modifiche sono di piccola entità, il numero di versione resta invariato; se l'entità delle modifiche diviene rilevante, si aumenta anche il numero di versione. Perciò, solo i programmi compilati con la nuova versione della libreria subiscono gli effetti delle modifiche incompatibili incorporate nella libreria stessa. I programmi collegati prima dell'installazione della nuova libreria continuano ad avvalersi della vecchia libreria. Questo sistema è noto anche con il nome di **librerie condivise**.

A differenza del caricamento dinamico, il collegamento dinamico richiede generalmente l'assistenza del sistema operativo. Se i processi presenti in memoria sono protetti l'uno dall'altro, il sistema operativo è l'unica entità che può controllare se la procedura richiesta da un processo è nello spazio di memoria di un altro processo, o che può consentire l'accesso di più processi agli stessi indirizzi di memoria. Questo concetto è sviluppato nel contesto della paginazione, nel Paragrafo 8.4.4.

8.2 Avvicendamento dei processi (swapping)

Per essere eseguito, un processo deve trovarsi in memoria centrale, ma si può trasferire temporaneamente in **memoria ausiliaria** (*backing store*) da cui si riporta in memoria centrale al momento di riprenderne l'esecuzione. Si consideri, per esempio, un ambiente di multiprogrammazione con un algoritmo circolare (*round-robin*) per lo scheduling della CPU. Trascorso un quanto di tempo, il gestore di memoria scarica dalla memoria il processo appena terminato e carica un altro processo nello spazio di memoria appena liberato; questo procedimento è illustrato nella Figura 8.5 e si chiama **avvicendamento dei processi in memoria** – o, più brevemente, **avvicendamento** o **scambio** (*swapping*). Nel frattempo lo scheduler della CPU assegna un quanto di tempo a un altro processo presente in memoria. Quando esaurisce il suo quanto di tempo, ciascun processo viene scambiato con un altro processo. In teoria il gestore della memoria può avvicendare i processi in modo sufficientemente rapido da far sì che alcuni siano presenti in memoria, pronti per essere eseguiti, quando lo scheduler della CPU voglia riassegnare la CPU stessa. Anche il quanto di tempo deve essere sufficientemente lungo da permettere che un processo, prima d'essere sostituito, esegua quantità ragionevole di calcolo.

Una variante di questo criterio d'avvicendamento dei processi s'impiega per gli algoritmi di scheduling basati sulle priorità. Se si presenta un processo con priorità maggiore, il gestore della memoria può scaricare dalla memoria centrale il processo con priorità inferio-

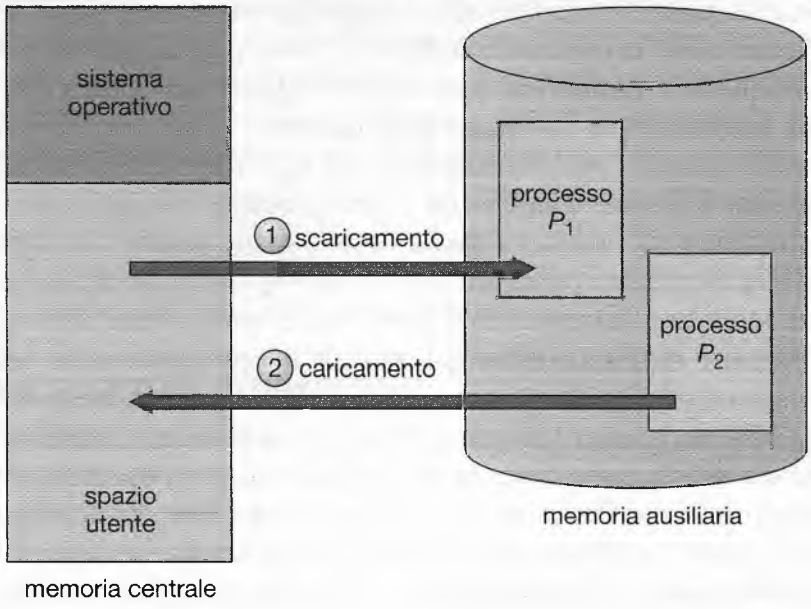


Figura 8.5 Avvicendamento di due processi con un disco come memoria ausiliaria.

re per fare spazio all'esecuzione del processo con priorità maggiore. Quando il processo con priorità maggiore termina, si può ricaricare in memoria quello con priorità minore e continuare la sua esecuzione (questo tipo d'avvicendamento è talvolta chiamato **roll out, roll in**).

Normalmente, un processo che è stato scaricato dalla memoria si deve ricaricare nello spazio di memoria occupato in precedenza. Questa limitazione è dovuta al metodo di associazione degli indirizzi. Se l'associazione degli indirizzi logici agli indirizzi fisici della memoria si effettua nella fase di assemblaggio o di caricamento, il processo non può essere ricaricato in posizioni diverse. Se l'associazione degli indirizzi logici agli indirizzi fisici della memoria si compie nella fase d'esecuzione, un processo può essere riversato in uno spazio di memoria diverso, poiché gli indirizzi fisici si calcolano nella fase d'esecuzione.

L'avvicendamento dei processi richiede una memoria ausiliaria. Tale memoria ausiliaria deve essere abbastanza ampia da contenere le copie di tutte le immagini di memoria di tutti i processi utenti, e deve permettere un accesso diretto a dette immagini di memoria. Il sistema mantiene una **coda dei processi pronti** (*ready queue*) formata da tutti i processi pronti per l'esecuzione, le cui immagini di memoria si trovano in memoria ausiliaria o in memoria. Quando lo scheduler della CPU decide di eseguire un processo, richiama il dispatcher, che controlla se il primo processo della coda si trova in memoria. Se non si trova in memoria, e in questa non c'è spazio libero, il dispatcher scarica un processo dalla memoria e vi carica il processo richiesto dallo scheduler della CPU, quindi ricarica normalmente i registri e trasferisce il controllo al processo selezionato.

In un siffatto sistema d'avvicendamento, il tempo di cambio di contesto *(context-switch time)* è abbastanza elevato. Per avere un'idea delle sue dimensioni si pensi a un processo utente di 100 MB e a una memoria ausiliaria costituita da un disco ordinario con velocità di trasferimento di 50 MB al secondo. Il trasferimento effettivo del processo di 100 MB da e in memoria richiede:

100 MB /50 MB al secondo = 2 secondi

Ipotizzando che la latenza media sia di 8 millisecondi, l'operazione richiede 2008 millisecondi. Poiché l'avvicendamento richiede lo scaricamento e il caricamento di un processo, il tempo totale è di circa 4016 millisecondi.

Occorre notare che la maggior parte del tempo d'avvicendamento è data dal tempo di trasferimento. Il tempo di trasferimento totale è direttamente proporzionale alla *quantità* di memoria interessata. In un calcolatore con 4 GB di memoria centrale e un sistema operativo residente di 1 GB, la massima dimensione possibile per un processo utente è di 3 GB. Tuttavia possono essere presenti molti processi utenti con dimensione minore, per esempio 100 MB. Lo scaricamento di un processo di 100 MB può concludersi in 2 secondi, mentre per lo scaricamento di 3 GB sono necessari 60 secondi. Perciò sarebbe utile sapere esattamente quanta memoria *sia* effettivamente usata da un processo utente e non solo quanta questo *potrebbe* teoricamente usarne, poiché in questo caso è necessario scaricare solo quanto è effettivamente utilizzato, riducendo il tempo d'avvicendamento. Affinché questo metodo risulti efficace, l'utente deve tenere informato il sistema su tutte le modifiche apportate ai requisiti di memoria; un processo con requisiti di memoria dinamici deve impiegare chiamate di sistema (`request memory` e `release memory`) per informare il sistema operativo delle modifiche da apportare alla memoria.

L'avvicendamento dei processi è soggetto ad altri vincoli. Per scaricare un processo dalla memoria è necessario essere certi che sia completamente inattivo. Particolare importanza

ha qualsiasi I/O pendente: mentre si vuole scaricare un processo per liberare la memoria, tale processo può essere nell'attesa del completamento di un'operazione di I/O. Tuttavia, se un dispositivo di I/O accede in modo asincrono alle aree di I/O della memoria (*buffer*) utente, il processo non è scaricabile. Si supponga che l'operazione di I/O sia stata accodata, perché il dispositivo era occupato. Se il processo P_2 s'avvicendasse al processo P_1, l'operazione di I/O potrebbe tentare di usare la memoria che attualmente appartiene al processo P_2. Questo problema si può risolvere in due modi: non scaricando dalla memoria un processo con operazioni di I/O pendenti, oppure eseguendo operazioni di I/O solo in aree di memoria per l'I/O del sistema operativo. In questo modo i trasferimenti fra tali aree del sistema operativo e la memoria assegnata al processo possono avvenire solo quando il processo è presente in memoria centrale.

L'ipotesi che l'avvicendamento dei processi richieda pochi o nessun movimento delle testine dei dischi merita ulteriori spiegazioni (tale argomento è trattato nel Capitolo 12, che illustra la struttura della memoria secondaria). Affinché il suo uso sia quanto più veloce è possibile, generalmente si assegna l'area d'avvicendamento in una porzione di disco separata da quella riservata al file system.

Attualmente l'avvicendamento semplice si usa in pochi sistemi; richiede infatti un elevato tempo di gestione, e consente un tempo di esecuzione troppo breve per essere considerato una soluzione ragionevole al problema di gestione della memoria. Versioni modificate dell'avvicendamento dei processi si trovano comunque in molti sistemi.

Una sua forma modificata era, per esempio, usata in molte versioni di UNIX: normalmente era disabilitato e si avviava solo nel caso in cui molti processi si fossero trovati in esecuzione superando una quantità limite di memoria. L'avvicendamento dei processi sarebbe stato di nuovo sospeso qualora il carico del sistema fosse diminuito. La gestione della memoria in UNIX è descritta ampiamente nel Paragrafo 21.7.

I primi PC, avendo un'architettura troppo semplice per adottare metodi avanzati di gestione della memoria, eseguivano più processi di grandi dimensioni tramite una versione modificata dell'avvicendamento. Un primo esempio importante è dato dal sistema operativo Windows 3.1 di Microsoft che consente l'esecuzione concorrente di più processi presenti in memoria. Se si carica un nuovo processo, ma lo spazio disponibile in memoria centrale è insufficiente, si trasferisce nel disco un processo già presente in memoria centrale. Non si tratta di vero e proprio avvicendamento dei processi, poiché è l'utente, e non lo scheduler, che decide il momento in cui scaricare un processo in favore di un altro; inoltre, ciascun processo scaricato resta in tale stato fino a quando sarà selezionato, per l'esecuzione, dall'utente. Le versioni successive dei sistemi operativi Microsoft si avvantaggiano delle caratteristiche avanzate delle MMU, attualmente disponibili anche nei PC. Nel Paragrafo 8.4 e nel Capitolo 9 – dedicato alla memoria virtuale – vengono illustrate queste caratteristiche.

8.3 Allocazione contigua della memoria

La memoria centrale deve contenere sia il sistema operativo sia i vari processi utenti; perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente. In questo paragrafo si tratta l'allocazione contigua della memoria.

La memoria centrale di solito si divide in due partizioni, una per il sistema operativo residente e una per i processi utenti. Il sistema operativo si può collocare sia in memoria bassa sia in memoria alta. Il fattore che incide in modo decisivo su tale scelta è generalmente la

posizione del vettore delle interruzioni. Poiché si trova spesso in memoria bassa, i programmatori collocano di solito anche il sistema operativo in memoria bassa. Per questo motivo prendiamo in considerazione solo la situazione in cui il sistema operativo risiede in quest'area di memoria.

Generalmente si vuole che più processi utenti risiedano contemporaneamente in memoria centrale. Perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda d'ingresso che attendono di essere caricati in memoria. Con l'**allocazione contigua della memoria**, ciascun processo è contenuto in una singola sezione contigua di memoria.

8.3.1 Rilocazione e protezione della memoria

Prima di trattare l'allocazione della memoria, dobbiamo soffermarci sulle questioni relative alla rilocazione e alla protezione della memoria. Tale protezione si può realizzare usando un registro di rilocazione, come si descrive nel Paragrafo 8.1.3, con un registro limite, come si descrive nel Paragrafo 8.1.1. Il registro di rilocazione contiene il valore dell'indirizzo fisico minore; il registro limite contiene l'intervallo di indirizzi logici, per esempio, *rilocazione* = 100.040 e *limite* = 74.600. Con i registri di rilocazione e limite, ogni indirizzo logico deve essere minore del contenuto del registro limite; la MMU fa corrispondere *dinamicamente* l'indirizzo fisico all'indirizzo logico sommando a quest'ultimo il valore contenuto nel registro di rilocazione (Figura 8.6).

Quando lo scheduler della CPU seleziona un processo per l'esecuzione, il dispatcher, durante l'esecuzione del cambio di contesto, carica il registro di rilocazione e il registro limite con i valori corretti. Poiché si confronta ogni indirizzo generato dalla CPU con i valori contenuti in questi registri, si possono proteggere il sistema operativo, i programmi e i dati di altri utenti da tentativi di modifiche da parte del processo in esecuzione.

Lo schema con registro di rilocazione consente al sistema operativo di cambiare dinamicamente le proprie dimensioni. Tale flessibilità è utile in molte situazioni; il sistema operativo, per esempio, contiene codice e spazio di memoria per i driver dei dispositivi; se uno di questi, o un altro servizio del sistema operativo, non è comunemente usato, è inutile tenerne in memoria codice e dati, poiché lo spazio occupato si potrebbe usare per altri scopi. Talvolta questo codice si chiama codice **transiente** del sistema operativo, poiché s'inserisce secondo le necessità; l'uso di tale codice cambia le dimensioni del sistema operativo durante l'esecuzione del programma.

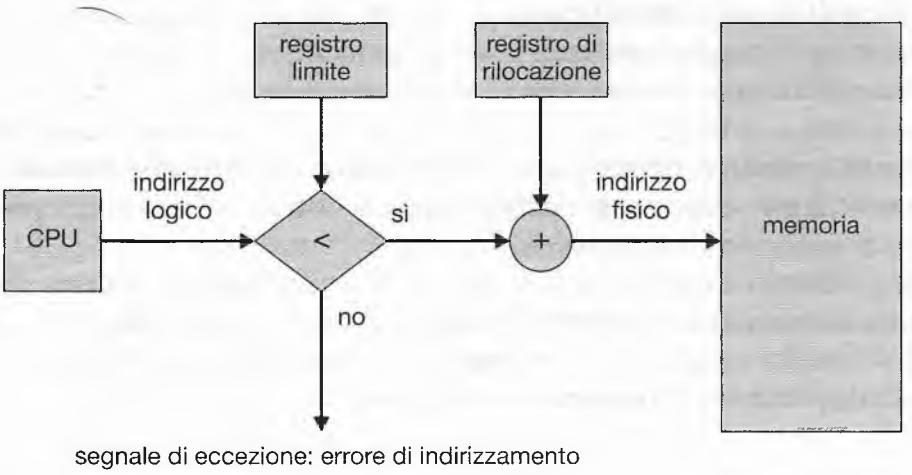


Figura 8.6 Registri di rilocazione e limite.

8.3.2 Allocazione della memoria

Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in **partizioni** di dimensione fissa. Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato dal numero di partizioni. Con il **metodo delle partizioni multiple** quando una partizione è libera può essere occupata da un processo presente nella coda d'ingresso; terminato il processo, la partizione diviene nuovamente disponibile per un altro processo. Originariamente questo metodo, detto mft, si usava nel sistema operativo IBM OS/360, ma attualmente non è più in uso. Il metodo descritto di seguito, detto mvt, è una generalizzazione del metodo con partizioni fisse e si usa soprattutto in ambienti d'elaborazione a lotti. Si noti, tuttavia, che molte idee che lo riguardano sono applicabili agli ambienti a partizione del tempo d'elaborazione nei quali si fa uso della segmentazione semplice per la gestione della memoria (Paragrafo 8.6).

Nello schema a partizione fissa il sistema operativo conserva una tabella in cui sono indicate le partizioni di memoria disponibili e quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utenti; si tratta di un grande blocco di memoria disponibile, un **buco** (*hole*). Infine, come avrete modo di vedere, la memoria contiene una serie di buchi di diverse dimensioni.

Quando entrano nel sistema, i processi vengono inseriti in una coda d'ingresso. Per determinare a quali processi si debba assegnare la memoria, il sistema operativo tiene conto dei requisiti di memoria di ciascun processo e della quantità di spazio di memoria disponibile. Quando a un processo si assegna dello spazio, il processo stesso viene caricato in memoria e può quindi competere per il controllo della CPU. Al termine, rilascia la memoria che gli era stata assegnata, e il sistema operativo può impiegarla per un altro processo presente nella coda d'ingresso.

In ogni dato istante è sempre disponibile una lista delle dimensioni dei blocchi liberi e della coda d'ingresso. Il sistema operativo può ordinare la coda d'ingresso secondo un algoritmo di scheduling. La memoria si assegna ai processi della coda finché si possono soddisfare i requisiti di memoria del processo successivo, cioè finché esiste un blocco di memoria (o buco) disponibile, sufficientemente grande da accogliere quel processo. Il sistema operativo può quindi attendere che si renda disponibile un blocco sufficientemente grande, oppure può scorrere la coda d'ingresso per verificare se sia possibile soddisfare le richieste di memoria più limitate di qualche altro processo.

In generale, è sempre presente un *insieme* di buchi di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un buco di dimensioni sufficienti per contenerlo. Se è troppo grande, il buco viene diviso in due parti: si assegna una parte al processo in arrivo e si riporta l'altra nell'insieme dei buchi. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nell'insieme dei buchi; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formarne uno più grande. A questo punto il sistema deve controllare se vi siano processi nell'attesa di spazio di memoria, e se la memoria appena liberata e ricombinata possa soddisfare le richieste di qualcuno fra tali processi.

Questa procedura è una particolare istanza del più generale problema di **allocazione dinamica della memoria**, che consiste nel soddisfare una richiesta di dimensione n data una lista di buchi liberi. Le soluzioni sono numerose. I criteri più usati per scegliere un buco libero tra quelli disponibili nell'insieme sono i seguenti.

- **First-fit.** Si assegna il *primo* buco abbastanza grande. La ricerca può cominciare sia dall'inizio dell'insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi.
- **Best-fit.** Si assegna il *più piccolo* buco in grado di contenere il processo. Si deve compiere la ricerca in tutta la lista, sempre che questa non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più piccole.
- **Worst-fit.** Si assegna il buco *più grande*. Anche in questo caso si deve esaminare tutta la lista, sempre che non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più grandi, che possono essere più utili delle parti più piccole ottenute col criterio precedente.

Con l'uso di simulazioni si è dimostrato che sia first-fit sia best-fit sono migliori rispetto a worst-fit in termini di risparmio di tempo e di utilizzo di memoria. D'altra parte nessuno dei due è chiaramente migliore dell'altro per quel che riguarda l'utilizzo della memoria ma, in genere, first-fit è più veloce.

8.3.3 Frammentazione

Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di **frammentazione esterna**: quando si caricano e si rimuovono i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna se lo spazio di memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria è frammentata in tanti piccoli buchi. Questo problema di frammentazione può essere molto grave; nel caso peggiore può verificarsi un blocco di memoria libera, sprecata, tra ogni coppia di processi. Se tutti questi piccoli pezzi di memoria costituissero in un unico blocco libero di grandi dimensioni, si potrebbero eseguire molti più processi.

Sia che si adotti l'uno o l'altro, l'impiego di un determinato criterio può influire sulla quantità di frammentazione: in alcuni sistemi dà migliori risultati la scelta del primo buco abbastanza grande, in altri dà migliori risultati la scelta del più piccolo tra i buchi abbastanza grandi; inoltre è necessario sapere qual è l'estremità assegnata di un blocco libero (se la parte inutilizzata è quella in alto o quella in basso). A prescindere dal tipo di algoritmo usato, la frammentazione esterna è un problema.

La sua gravità dipende dalla quantità totale di memoria e dalla dimensione media dei processi. L'analisi statistica dell'algoritmo che segue il criterio di scelta del primo buco abbastanza grande, per esempio, rivela che, pur con una certa ottimizzazione, per n blocchi assegnati, si perdono altri 0,5 n blocchi a causa della frammentazione, ciò significa che potrebbe essere inutilizzabile un terzo della memoria. Questa caratteristica è nota con il nome di **regola del 50 per cento**.

La frammentazione può essere sia interna sia esterna. Si consideri il metodo d'allocazione con più partizioni con un buco di 18.464 byte. Supponendo che il processo successivo richieda 18.462 byte, assegnando esattamente il blocco richiesto rimane un buco di 2 byte. Il carico necessario per tener traccia di questo buco è sostanzialmente più grande del buco stesso. Il metodo generale prevede di suddividere la memoria fisica in blocchi di dimensione fissa, che costituiscono le unità d'allocazione. Con questo metodo la memoria assegnata può essere leggermente maggiore della memoria richiesta. La **frammentazione interna** consiste nella differenza tra questi due numeri; la memoria è interna a una partizione, ma non è in uso.

Una soluzione al problema della frammentazione esterna è data dalla **compattazione**. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco. La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è fatta nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si compie nella fase d'esecuzione. Se gli indirizzi sono rilocati dinamicamente, la rilocazione richiede solo lo spostamento del programma e dei dati, e quindi la modifica del registro di rilocazione in modo che rifletta il nuovo indirizzo di base. Quando è possibile eseguire la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria, mentre tutti i buchi vengono spostati nell'altra direzione formando un grosso buco di memoria. Questo metodo può essere assai oneroso.

Un'altra possibile soluzione del problema della frammentazione esterna è data dal consentire la non contiguità dello spazio degli indirizzi logici di un processo, permettendo così di assegnare la memoria fisica ai processi dovunque essa sia disponibile. Due tecniche complementari conseguono questo risultato: la paginazione (Paragrafo 8.4) e la segmentazione (Paragrafo 8.6). Queste tecniche si possono anche combinare (Paragrafo 8.7).

8.4 Paginazione

La paginazione è un metodo di gestione della memoria che permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. Elimina il gravoso problema della sistemazione di blocchi di memoria di diverse dimensioni in memoria ausiliaria, una questione che riguarda la maggior parte dei metodi di gestione della memoria analizzati. Il problema insorge perché, quando alcuni frammenti di codice o dati residenti in memoria centrale devono essere scaricati, si deve trovare lo spazio necessario in memoria ausiliaria. I problemi di frammentazione relativi alla memoria centrale valgono anche per la memoria ausiliaria, con la differenza che in questo caso l'accesso è molto più lento, quindi è impossibile eseguire la compattazione. Grazie ai vantaggi offerti rispetto ai metodi precedenti, la paginazione nelle sue varie forme è comunemente usata in molti sistemi operativi.

Tradizionalmente, l'architettura del sistema offre specifiche caratteristiche per la gestione della paginazione. Recenti progetti (soprattutto le CPU a 64 bit) prevedono che il sistema di paginazione sia realizzato integrando strettamente l'architettura e il sistema operativo.

8.4.1 Metodo di base

Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di dimensione costante, detti anche **frame** o **pagine fisiche**, e nel suddividere la memoria logica in blocchi di pari dimensione, detti **pagine**. Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria, divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria.

L'architettura d'ausilio alla paginazione è illustrata nella Figura 8.7; ogni indirizzo generato dalla CPU è diviso in due parti: un **numero di pagina** (*p*), e uno **scostamento** (*offset*) **di pagina** (*d*). Il numero di pagina serve come indice per la **tabella delle pagine**, contenente l'indirizzo di base in memoria fisica di ogni pagina. Questo indirizzo di base si combina con lo scostamento di pagina per definire l'indirizzo della memoria fisica, che s'invia all'unità di memoria. La Figura 8.8 illustra il modello di paginazione della memoria.

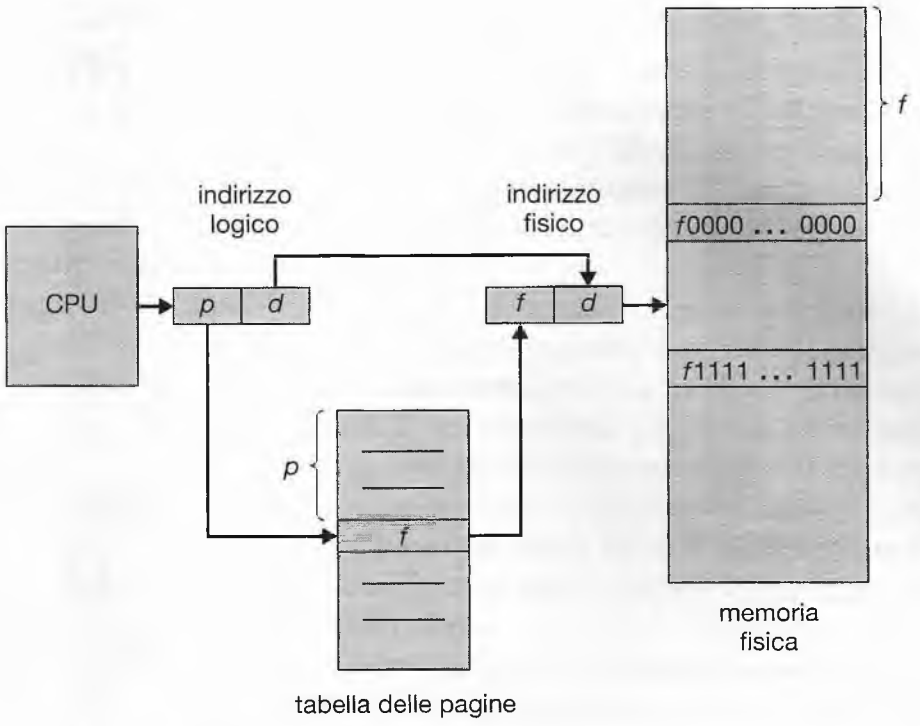


Figura 8.7 Architettura di paginazione.

La dimensione di una pagina, così come quella di un frame, è definita dall'architettura del calcolatore ed è, in genere, una potenza di 2 compresa tra 512 byte e 16 MB. La scelta di una potenza di 2 come dimensione della pagina facilita notevolmente la traduzione di un indirizzo logico nei corrispondenti numero di pagina e scostamento di pagina. Se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagina è di 2^n unità di indiriz-

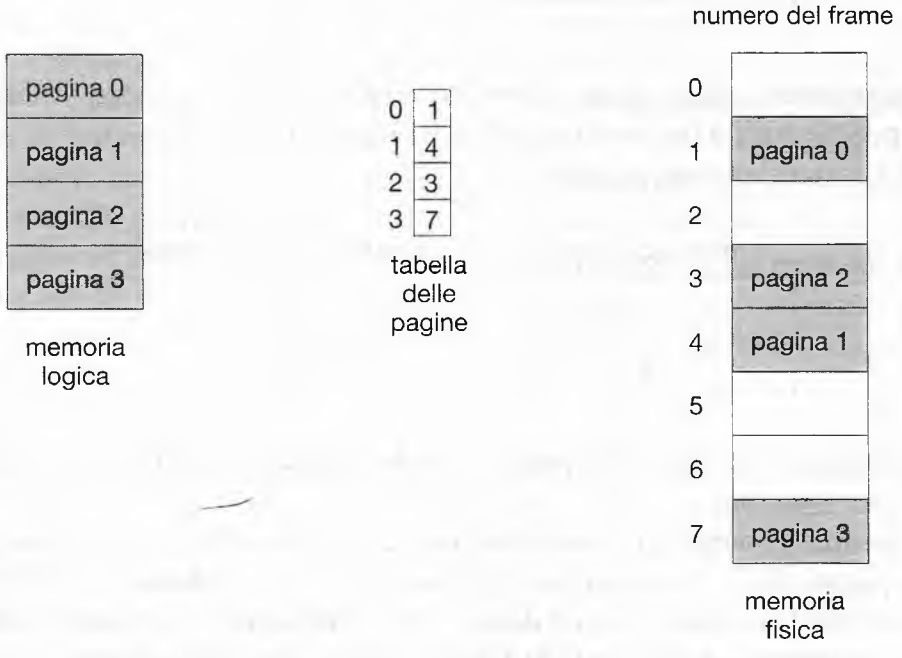


Figura 8.8 Modello di paginazione di memoria logica e memoria fisica.

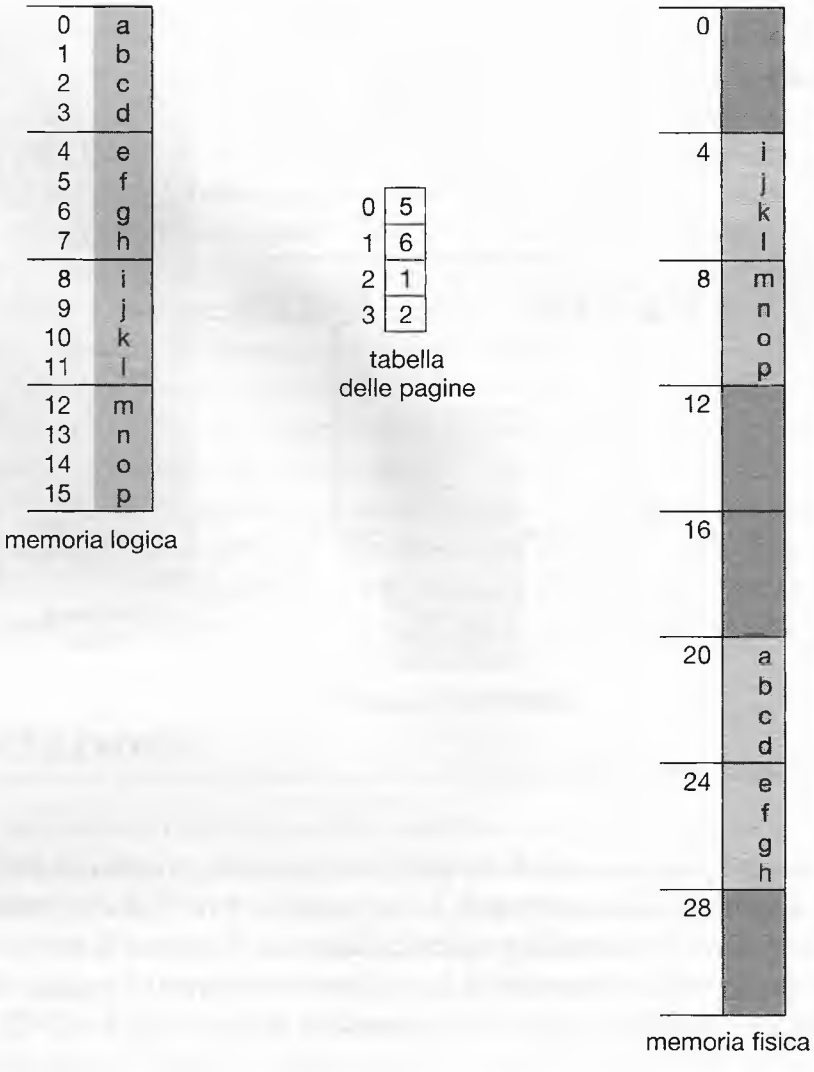


Figura 8.9 Esempio di paginazione per una memoria di 32 byte con pagine di 4 byte.

zamento (byte o parole), allora gli $m - n$ bit più significativi di un indirizzo logico indicano il numero di pagina, e gli n bit meno significativi indicano lo scostamento di pagina. L'indirizzo logico ha quindi la forma seguente:

numero di pagina	scostamento di pagina
p	d
$m - n$	n

dove p è un indice della tabella delle pagine e d è lo scostamento all'interno della pagina indicata da p.

Come esempio concreto, anche se minimo, si consideri la memoria illustrata nella Figura 8.9; con pagine di 4 byte e una memoria fisica di 32 byte (8 pagine), si mostra come si faccia corrispondere la memoria vista dall'utente alla memoria fisica. L'indirizzo logico 0 è la pagina 0 con scostamento 0. Secondo la tabella delle pagine, la pagina 0 si trova nel frame 5. Quindi all'indirizzo logico 0 corrisponde l'indirizzo fisico 20 (= (5 × 4) + 0). All'indirizzo logico 3 (pagina 0, scostamento 3) corrisponde l'indirizzo fisico 23 (= (5 × 4) + 3). Per

quel che riguarda l'indirizzo logico 4 (pagina 1, scostamento 0), secondo la tabella delle pagine, alla pagina 1 corrisponde il frame 6, quindi, all'indirizzo logico 4 corrisponde l'indirizzo fisico 24 (= (6 × 4) + 0). All'indirizzo logico 13 corrisponde l'indirizzo fisico 9.

Il lettore può aver notato che la paginazione non è altro che una forma di rilocazione dinamica: a ogni indirizzo logico l'architettura di paginazione fa corrispondere un indirizzo fisico. L'uso della tabella delle pagine è simile all'uso di una tabella di registri base (o di rilocazione), uno per ciascun frame.

Con la paginazione si può evitare la frammentazione esterna: *qualsiasi* frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia si può avere la frammentazione interna. I frame si assegnano come unità. Poiché in generale lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle pagine, l'*ultimo* frame assegnato può non essere completamente pieno. Se, per esempio, le pagine sono di 2048 byte, un processo di 72.766 byte necessita di 35 pagine più 1086 byte. Si assegnano 36 frame, quindi si ha una frammentazione interna di 2048 – 1086 = 962 byte. Il caso peggiore si ha con un processo che necessita di n pagine più un byte: si assegnano $n + 1$ frame, quindi si ha una frammentazione interna di quasi un intero frame.

Se la dimensione del processo è indipendente dalla dimensione della pagina, ci si deve aspettare una frammentazione interna media di mezza pagina per processo. Questa considerazione suggerisce che conviene usare pagine di piccole dimensioni; tuttavia, a ogni elemento della tabella delle pagine è associato un carico che si può ridurre aumentando le dimensioni delle pagine. Inoltre, con un maggior numero di dati da trasferire, l'I/O su disco è più efficiente (Capitolo 12). Generalmente la dimensione delle pagine cresce col passare del tempo, come i processi, gli insiemi di dati e la memoria centrale; attualmente la dimensione tipica delle pagine è compresa tra 4 KB e 8 KB; in alcuni sistemi può essere anche maggiore. Alcune CPU e alcuni nuclei di sistemi operativi gestiscono anche pagine di diverse dimensioni; il sistema Solaris ad esempio usa pagine di 4 KB o 8 MB, secondo il tipo dei dati memorizzati nelle pagine. Sono in fase di studio e progettazione sistemi di paginazione che consentono la variazione dinamica della dimensione delle pagine.

Ciascun elemento della tabella delle pagine di solito è lungo 4 byte, ma anche questa dimensione può variare; un elemento di 32 bit può puntare a uno dei 2^{32} frame; quindi se un frame è di 4 KB, un sistema con elementi di 4 byte può accedere a 2^{44} byte (o 64 TB) di memoria fisica.

Quando si deve eseguire un processo, si esamina la sua dimensione espressa in pagine. Poiché ogni pagina del processo necessita di un frame, se il processo richiede n pagine, devono essere disponibili almeno n frame che, se ci sono, si assegnano al processo stesso. Si carica la prima pagina del processo in uno dei frame assegnati e s'inserisce il numero del frame nella tabella delle pagine relativa al processo in questione. La pagina successiva si carica in un altro frame e, anche in questo caso, s'inserisce il numero del frame nella tabella delle pagine, e così via (Figura 8.10).

Un aspetto importante della paginazione è la netta distinzione tra la memoria vista dall'utente e l'effettiva memoria fisica: il programma utente *vede* la memoria come un unico spazio contiguo, contenente solo il programma stesso; in realtà, il programma utente è sparso in una memoria fisica contenente anche altri programmi. La differenza tra la memoria vista dall'utente e la memoria fisica è colmata dall'architettura di traduzione degli indirizzi, che fa corrispondere gli indirizzi fisici agli indirizzi logici generati dai processi utenti. Queste trasformazioni non sono visibili agli utenti e sono controllate dal sistema operativo. Si noti che

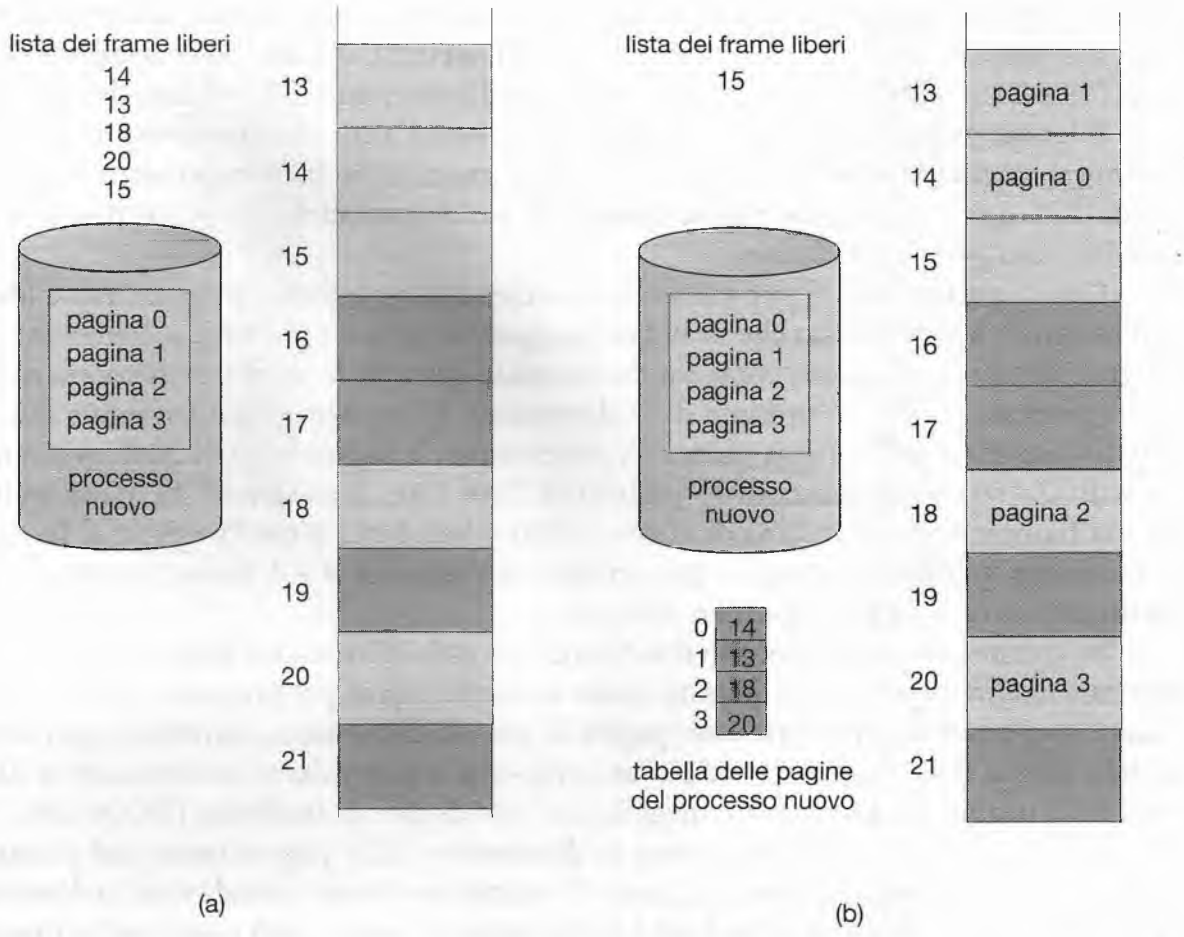


Figura 8.10 Frame liberi; (a) prima e (b) dopo l'allocazione.

un processo utente, per definizione, non può accedere alle zone di memoria che non gli appartengono. Non ha modo di accedere alla memoria oltre quel che è previsto dalla sua tabella delle pagine; tale tabella riguarda soltanto le pagine che appartengono al processo.

Poiché il sistema operativo gestisce la memoria fisica, deve essere informato dei relativi particolari di allocazione: quali frame sono assegnati, quali sono disponibili, il loro numero totale, e così via. In genere queste informazioni sono contenute in una struttura dati chiamata **tabella dei frame**, contenente un elemento per ogni frame, indicante se sia libero oppure assegnato e, se è assegnato, a quale pagina di quale processo o di quali processi.

Inoltre, il sistema operativo deve sapere che i processi utenti operano nello spazio utente, e tutti gli indirizzi logici si devono far corrispondere a indirizzi fisici. Se un utente usa una chiamata di sistema con un indirizzo di memoria come parametro, per esempio per eseguire un'operazione di I/O all'indirizzo specificato, si deve tradurre tale indirizzo nell'indirizzo fisico corretto. Il sistema operativo conserva una copia della tabella delle pagine per ciascun processo, così come conserva una copia dei valori contenuti nel contatore di programma e nei registri. Questa copia si usa per tradurre gli indirizzi logici in indirizzi fisici ogni volta che il sistema operativo deve associare esplicitamente un indirizzo fisico a un indirizzo logico. La stessa copia è usata anche dal dispatcher della CPU per impostare l'architettura di paginazione quando a un processo sta per essere assegnata la CPU. La paginazione fa quindi aumentare la durata dei cambi di contesto.

8.4.2 Architettura di paginazione

Ogni sistema operativo segue metodi propri per memorizzare le tabelle delle pagine. La maggior parte dei sistemi impiega una tabella delle pagine per ciascun processo. Il PCB contiene, insieme col valore di altri registri, come il registro delle istruzioni, un puntatore alla tabella delle pagine. Per avviare un processo, il dispatcher ricarica i registri utente e imposta i corretti valori della tabella delle pagine fisiche, usando la tabella delle pagine presente in memoria e relativa al processo.

L'architettura d'ausilio alla tabella delle pagine si può realizzare in modi diversi. Nel caso più semplice, si usa uno specifico insieme di **registri**. Per garantire un'efficiente traduzione degli indirizzi di paginazione, questi registri devono essere costruiti in modo da operare a una velocità molto elevata. Tale efficienza è determinante, poiché ogni accesso alla memoria passa attraverso il sistema di paginazione. Il dispatcher della CPU ricarica questi registri proprio come ricarica gli altri, ma le istruzioni di caricamento e modifica dei registri della tabella delle pagine sono privilegiate, quindi soltanto il sistema operativo può modificare la mappa della memoria. Il DEC PDP-11 è un esempio di tale tipo di architettura. Un indirizzo è costituito di 16 bit e la dimensione di una pagina è di 8 KB; la tabella delle pagine consiste quindi di otto elementi che sono mantenuti in registri veloci.

L'uso di registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, nell'ordine, per esempio, di 256 elementi. La maggior parte dei calcolatori contemporanei usa comunque tabelle molto grandi, per esempio di un milione di elementi, quindi non si possono impiegare i registri veloci per realizzare la tabella delle pagine; quest'ultima si mantiene in memoria principale e un **registro di base della tabella delle pagine** (*page-table base register*, PTBR) punta alla tabella stessa. Il cambio delle tabelle delle pagine richiede soltanto di modificare questo registro, riducendo considerevolmente il tempo dei cambi di contesto.

Questo metodo presenta un problema connesso al tempo necessario di accesso a una locazione della memoria utente. Per accedere alla locazione *i*, occorre far riferimento alla tabella delle pagine usando il valore contenuto nel PTBR aumentato del numero di pagina relativo a *i*, perciò si deve accedere alla memoria. Si ottiene il numero del frame che, associato allo scostamento di pagina, produce l'indirizzo cercato; a questo punto è possibile accedere alla posizione di memoria desiderata. Con questo metodo, per accedere a un byte occorrono *due* accessi alla memoria (uno per l'elemento della tabella delle pagine e uno per il byte stesso), quindi l'accesso alla memoria è rallentato di un fattore 2. Nella maggior parte dei casi un tale ritardo è intollerabile; sarebbe più conveniente ricorrere all'avvicendamento dei processi!

La soluzione tipica a questo problema consiste nell'impiego di una speciale, piccola cache di ricerca veloce, detta TLB (*translation look-aside buffer*). La TLB è una memoria associativa ad alta velocità in cui ogni suo elemento consiste di due parti: una chiave, o un indicatore (*tag*) e un valore. Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. La ricerca è molto rapida, ma le memorie associative sono molto costose. Il numero degli elementi in una TLB è piccolo, spesso è compreso tra 64 e 1024.

La TLB si usa insieme con le tabelle delle pagine nel modo seguente: la TLB contiene una piccola parte degli elementi della tabella delle pagine; quando la CPU genera un indirizzo logico, si presenta il suo numero di pagina alla TLB; se tale numero è presente, il corrispondente numero del frame è immediatamente disponibile e si usa per accedere alla memoria. Tutta l'operazione può richiedere un tempo inferiore al 10 per cento in più di quanto sarebbe richiesto per un riferimento alla memoria senza paginazione.

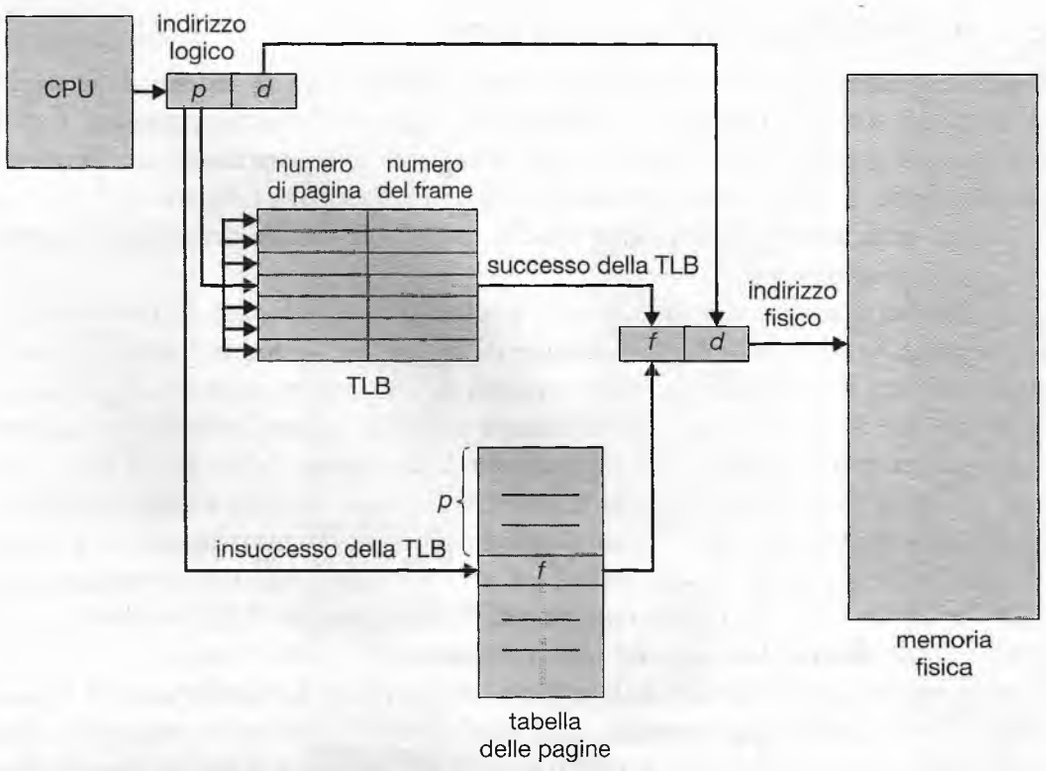


Figura 8.11 Architettura di paginazione con TLB.

Se nella TLB non è presente il numero di pagina, situazione nota come **insuccesso della TLB** (*TLB miss*), si deve consultare la tabella delle pagine in memoria. Il numero del frame così ottenuto si può eventualmente usare per accedere alla memoria (Figura 8.11). Inoltre, inserendo i numeri della pagina e del frame nella TLB, al riferimento successivo la ricerca sarà molto più rapida. Se la TLB è già piena d'elementi, il sistema operativo deve sceglierne uno per sostituirlo. I criteri di sostituzione variano dalla scelta dell'elemento usato meno recentemente (LRU) alla scelta casuale. Inoltre alcune TLB consentono che certi elementi siano **vincolati** (*wired down*), cioè non si possano rimuovere dalla TLB; in genere si vincolano gli elementi per il codice del kernel.

Alcune TLB memorizzano gli **identificatori dello spazio d'indirizzi** (*address-space identifier*, ASID) in ciascun elemento della TLB. Un ASID identifica in modo univoco ciascun processo e si usa per fornire al processo corrispondente la protezione del suo spazio d'indirizzi. Quando tenta di trovare i valori corrispondenti ai numeri delle pagine virtuali, la TLB assicura che l'ASID per il processo attualmente in esecuzione corrisponda all'ASID associato alla pagina virtuale. La mancata corrispondenza dell'ASID viene trattata come un insuccesso della TLB. Inoltre, per fornire la protezione dello spazio d'indirizzi, l'ASID consente che la TLB contenga nello stesso istante elementi di diversi processi. Se la TLB non permette l'uso di ASID distinti, ogni volta che si seleziona una nuova tabella delle pagine, per esempio a ogni cambio di contesto, si deve **cancellare** (*flush*) la TLB, in modo da assicurare che il successivo processo in esecuzione non faccia uso di errate informazioni di traduzione. Potrebbero altrimenti esserci vecchi elementi della TLB contenenti indirizzi virtuali validi, ma con indirizzi fisici corrispondenti sbagliati o non validi, lasciati in sospeso dal precedente processo.

La percentuale di volte che un numero di pagina si trova nella TLB è detta **tasso di successi** (*hit ratio*). Un tasso di successi dell'80 per cento significa che il numero di pagina desiderato si trova nella TLB nell'80 per cento dei casi. Se la ricerca nella TLB richiede 20 nano-

secondi e sono necessari 100 nanosecondi per accedere alla memoria, allora, supponendo che il numero di pagina si trovi nella TLB, un accesso alla memoria richiede 120 nanosecondi. Se, invece, il numero non è contenuto nella TLB (20 nanosecondi), occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero del frame (100 nanosecondi), quindi accedere al byte desiderato in memoria (100 nanosecondi); in totale sono necessari 220 nanosecondi. Per calcolare il **tempo effettivo d'accesso alla memoria** occorre tener conto della probabilità dei due casi:

$$\textit{tempo effettivo d'accesso} = 0{,}80 \times 120 + 0{,}20 \times 220 = 140 \text{ nanosecondi}$$

In questo esempio si verifica un rallentamento del 40 per cento nel tempo d'accesso alla memoria (da 100 a 140 nanosecondi).

Per un tasso di successi del 98 per cento si ottiene il seguente risultato:

$$\textit{tempo effettivo d'accesso} = 0{,}98 \times 120 + 0{,}02 \times 220 = 122 \text{ nanosecondi}$$

Aumentando il tasso di successi, il rallentamento del tempo d'accesso alla memoria scende al 22 per cento. L'impatto del tasso di successi nelle TLB è ulteriormente analizzato nel Capitolo 9.

8.4.3 Protezione

In un ambiente paginato, la protezione della memoria è assicurata dai bit di protezione associati a ogni frame; normalmente tali bit si trovano nella tabella delle pagine.

Un bit può determinare se una pagina si può leggere e scrivere oppure soltanto leggere. Tutti i riferimenti alla memoria passano attraverso la tabella delle pagine per trovare il numero correto del frame; quindi mentre si calcola l'indirizzo fisico, si possono controllare i bit di protezione per verificare che non si scriva in una pagina di sola lettura. Un tale tentativo causerebbe l'invio di un segnale di eccezione al sistema operativo, si avrebbe cioè una violazione della protezione della memoria.

Questo metodo si può facilmente estendere per fornire un livello di protezione più perfezionato. Si può progettare un'architettura che fornisca la protezione di sola lettura, di sola scrittura o di sola esecuzione. In alternativa, con bit di protezione distinti per ogni tipo d'accesso, si può ottenere una qualsiasi combinazione di tali tipi d'accesso; i tentativi illegali causano l'invio di un segnale di eccezione al sistema operativo.

Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto **bit di validità**. Tale bit, impostato a *valido*, indica che la pagina corrispondente è nello spazio d'indirizzi logici del processo, quindi è una pagina valida; impostato a *non valido*, indica che la pagina non è nello spazio d'indirizzi logici del processo. Il bit di validità consente quindi di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso un segnale di eccezione. Il sistema operativo concede o revoca la possibilità d'accesso a una pagina impostando in modo appropriato tale bit.

Per esempio, supponiamo che in un sistema con uno spazio di indirizzi di 14 bit (da 0 a 16.383) si possa avere un programma che deve usare soltanto gli indirizzi da 0 a 10.468 Con una dimensione delle pagine di 2 KB si ha la situazione mostrata nella Figura 8.12. Gli indirizzi nelle pagine 0, 1, 2, 3, 4 e 5 sono tradotti normalmente tramite la tabella delle pagine. D'altra parte, ogni tentativo di generare un indirizzo nelle pagine 6 o 7 trova non valido il bit di validità; in questo caso il calcolatore invia un segnale di eccezione al sistema operativo (riferimento di pagina non valido).

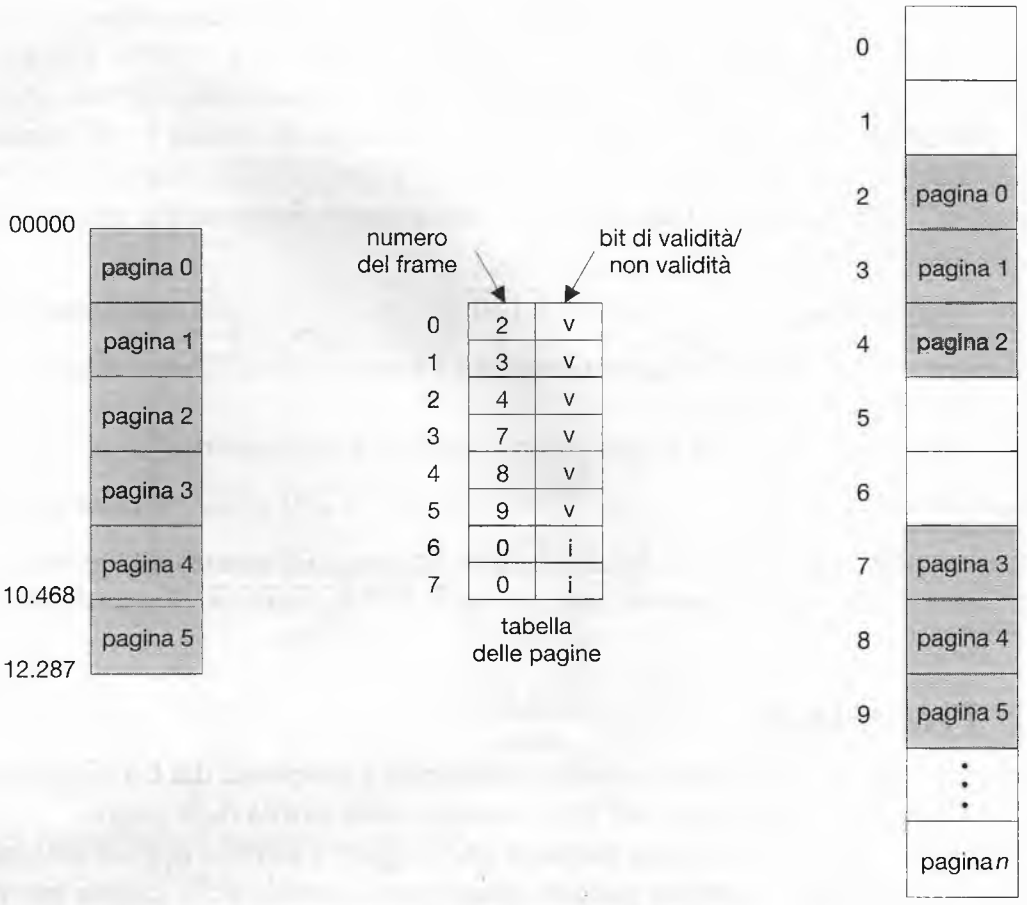


Figura 8.12 Bit di validità (v) o non validità (i) in una tabella delle pagine.

Questo schema ha creto un problema: poiché il programma si estende solo fino all'indirizzo 10.468, ogni riferimento oltre tale indirizzo è illegale; i riferimenti alla pagina 5 sono tuttavia classificati come validi, e ciò rende validi gli accessi sino all'indirizzo 12.287; solo gli indirizzi da 12.288 a 16.383 sono non validi. Tale problema è dovuto alla dimensione delle pagine di 2 KB e riflette la frammentazione interna della paginazione.

Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, infatti molti processi utilizzano solo una piccola frazione dello spazio d'indirizzi di cui dispongono. In questi casi è un inutile spreco creare una tabella di pagine con elementi per ogni pagina dell'intervallo di indirizzi, poiché una gran parte di questa tabella resta inutilizzata e occupa prezioso spazio di memoria. Alcune architetture dispongono di registri, detti **registri di lunghezza della tabella delle pagine** (*page-table length register*, PTLR), per indicare le dimensioni della tabella. Questo valore si controlla rispetto a ogni indirizzo logico per verificare che quest'ultimo si trovi nell'intervallo valido per il processo. Un errore causa l'invio di un segnale di eccezione al sistema operativo.

8.4.4 Pagine condivise

Un altro vantaggio della paginazione consiste nella possibilità di *condividere* codice comune. Questa considerazione è importante soprattutto in un ambiente a partizione del tempo. Si consideri un sistema con 40 utenti, ciascuno dei quali usa un elaboratore di testi. Se tale programma è formato da 150 KB di codice e 50 KB di spazio di dati, per gestire i 40 utenti

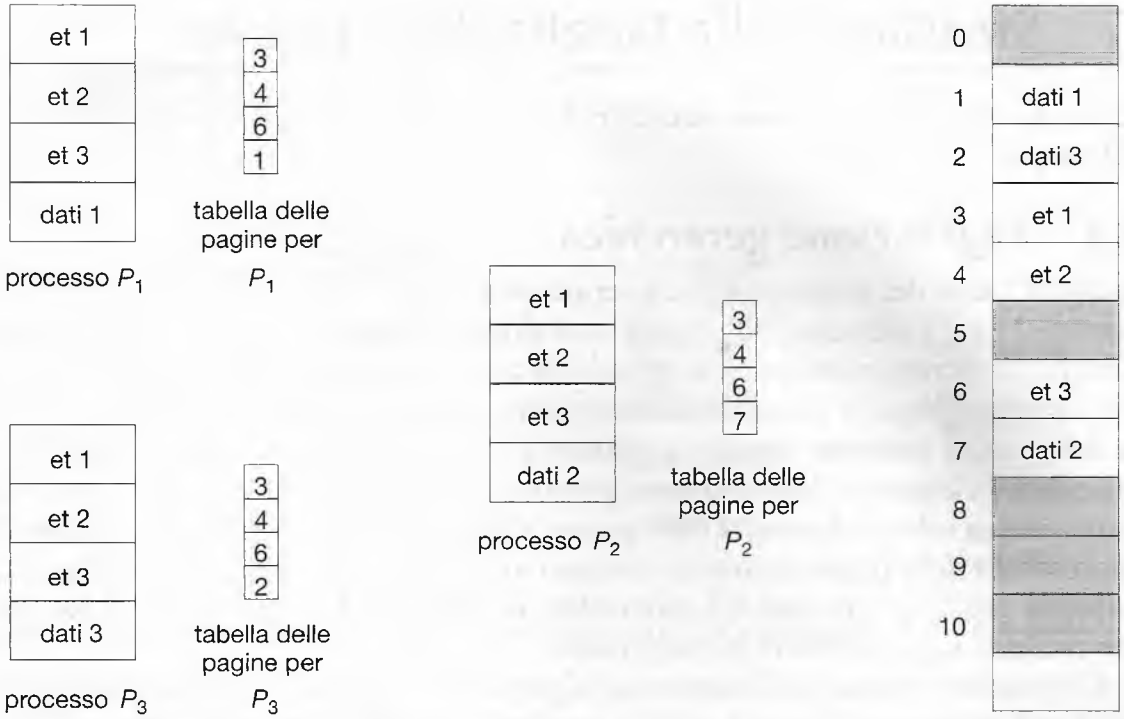


Figura 8.13 Condivisione di codice in un ambiente paginato.

sono necessari 8000 KB. Se però il codice è rientrante, può essere condiviso, come mostra la Figura 8.13: un elaboratore di testi, di tre pagine di 50 KB ciascuna (l'ampia dimensione delle pagine ha lo scopo di rendere più chiara la rappresentazione), condiviso da tre processi, ciascuno dei quali dispone della propria pagina di dati.

Il **codice rientrante**, detto anche **codice puro**, è un codice non automodificante: non cambia durante l'esecuzione. Quindi, due o più processi possono eseguire lo stesso codice nello stesso momento. Ciascun processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari alla propria esecuzione. I dati per due differenti processi variano, ovviamente, per ciascun processo.

In memoria fisica è presente una sola copia dell'elaboratore di testi: la tabella delle pagine di ogni utente fa corrispondere gli stessi frame contenenti l'elaboratore di testi, mentre le pagine dei dati si fanno corrispondere a frame diversi. Quindi per gestire 40 utenti sono sufficienti una copia dell'elaboratore di testi (150 KB) e 40 copie dei 50 KB di spazio di dati per ciascun utente; per un totale di 2150 KB, invece di 8000 KB; il risparmio è notevole.

Si possono condividere anche altri programmi d'uso frequente: compilatori, interfacce a finestre, librerie a collegamento dinamico, sistemi di basi di dati e così via. Per essere condivisibile, il codice deve essere rientrante. La natura di sola lettura del codice condiviso non si può affidare alla sola correttezza intrinseca del codice stesso, ma deve essere fatta rispettare dal sistema operativo. Tale condivisione della memoria tra processi di un sistema è simile al modo in cui i thread condividono lo spazio d'indirizzi di un task (Capitolo 4). Inoltre con riferimento al Capitolo 3, dove si descrive la memoria condivisa come un metodo di comunicazione tra processi, alcuni sistemi operativi realizzano la memoria condivisa impiegando le pagine condivise.

Oltre a permettere che più processi condividano le stesse pagine fisiche, l'organizzazione della memoria in pagine offre numerosi altri vantaggi; ne vedremo alcuni nel Capitolo 9.

8.5 Struttura della tabella delle pagine

In questo paragrafo si descrivono alcune tra le tecniche più comuni per strutturare la tabella delle pagine.

8.5.1 Paginazione gerarchica

La maggior parte dei moderni calcolatori dispone di uno spazio d'indirizzi logici molto grande (da 2^{32} a 2^{64} elementi). In un ambiente di questo tipo la stessa tabella delle pagine finirebbe per diventare eccessivamente grande. Si consideri, per esempio, un sistema con uno spazio d'indirizzi logici a 32 bit. Se la dimensione di ciascuna pagina è di 4 KB (2^{12}), la tabella delle pagine potrebbe arrivare a contenere fino a 1 milione di elementi ($2^{32}/2^{12}$). Se ogni elemento consiste di 4 byte, ciascun processo potrebbe richiedere fino a 4 MB di spazio fisico d'indirizzi solo per la tabella delle pagine. Chiaramente, sarebbe meglio evitare di collocare la tabella delle pagine in modo contiguo in memoria centrale. Una semplice soluzione a questo problema consiste nel suddividere la tabella delle pagine in parti più piccole; questo risultato si può ottenere in molti modi.

Un metodo consiste nell'adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata (Figura 8.14). Si consideri il precedente esempio di macchina a 32 bit con dimensione delle pagine di 4 KB. Si suddivide ciascun indirizzo logico in un numero di pagina di 20 bit e in uno scostamento di pagina di 12 bit. Paginando la tabella delle

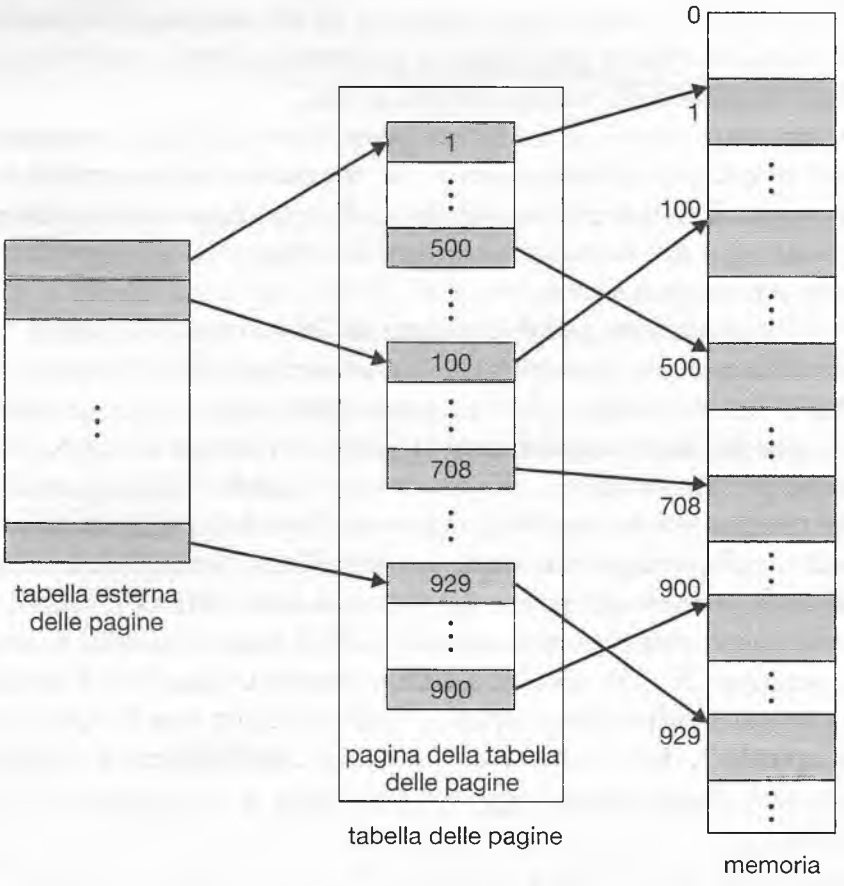


Figura 8.14 Schema di una tabella delle pagine a due livelli.

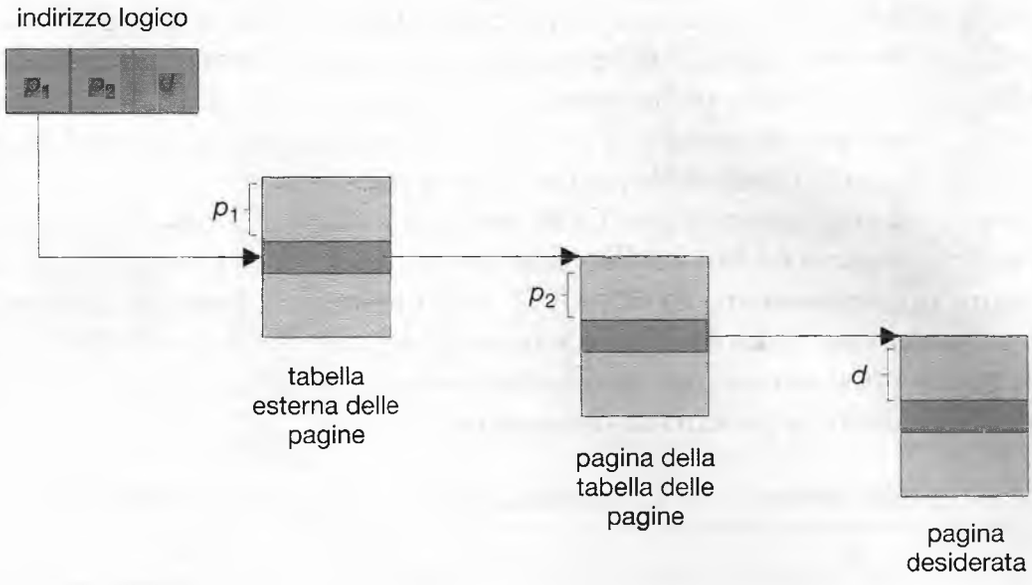


Figura 8.15 Traduzione degli indirizzi per un'architettura a 32 bit con paginazione a due livelli.

pagine, anche il numero di pagina è a sua volta suddiviso in un numero di pagina di 10 bit e uno scostamento di pagina di 10 bit. Quindi, l'indirizzo logico è:

numero di pagina		scostamento di pagina
p_1	p_2	d
10	10	12

dove p_1 è un indice della tabella delle pagine di primo livello, o tabella esterna delle pagine, e p_2 è lo scostamento all'interno della pagina indicata dalla tabella esterna delle pagine. Il metodo di traduzione degli indirizzi seguito da questa architettura è mostrato nella Figura 8.15. Poiché la traduzione degli indirizzi si svolge dalla tabella esterna delle pagine verso l'interno, questo metodo è anche noto come **tabella delle pagine ad associazione diretta** (*forward-mapped page table*).

Anche l'architettura VAX ha una variante della paginazione a due livelli. Il VAX è una macchina a 32 bit con pagine di 512 byte. Lo spazio d'indirizzi logici di un processo è suddiviso in quattro sezioni uguali, ciascuna di 2^{30} byte; ogni sezione rappresenta una parte differente dello spazio d'indirizzi logici di un processo. I 2 bit più significativi dell'indirizzo logico identificano la sezione appropriata, i successivi 21 bit rappresentano il numero logico di pagina all'interno di tale sezione e gli ultimi 9 bit lo scostamento nella pagina richiesta. Suddividendo in questo modo la tabella delle pagine, il sistema operativo può lasciare inutilizzate le diverse parti fino al momento in cui un processo ne fa richiesta. Nell'architettura VAX un indirizzo ha quindi la forma seguente:

sezione	pagina	scostamento
s	p	d
2	21	9

dove s denota il numero della sezione, p è un indice per la tabella delle pagine e d è lo scostamento all'interno della pagina. Anche quando si usa questo schema, la dimensione di una tabella delle pagine a un livello per un processo in un sistema VAX che usa una sezione è ancora 2^{21} bit × 4 byte per elemento = 8 MB. Per ridurre ulteriormente l'uso della memoria centrale, il VAX pagina le tabelle delle pagine dei processi utenti.

Lo schema di paginazione a due livelli non è più adatto nel caso di sistemi con uno spazio di indirizzi logici a 64 bit. Per illustrare questo aspetto, si supponga che la dimensione delle pagine di questo sistema sia di 4 KB (2^{12}). In questo caso, la tabella delle pagine conterrà fino a 2^{52} elementi. Adottando uno schema di paginazione a due livelli, le tabelle interne delle pagine possono occupare convenientemente una pagina, o contenere 2^{10} elementi di 4 byte. Gli indirizzi si presentano come segue:

pagina esterna	pagina interna	scostamento
p_1	p_2	d
42	10	12

La tabella esterna delle pagine consiste di 2^{42} elementi, o 2^{44} byte. La soluzione ovvia per evitare una tabella tanto grande consiste nel suddividere la tabella in parti più piccole. Questo metodo si adotta anche in alcune CPU a 32 bit allo scopo di fornire una maggiore flessibilità ed efficienza.

La tabella esterna delle pagine si può suddividere in vari modi. Si può paginare la tabella esterna delle pagine, ottenendo uno schema di paginazione a tre livelli. Si supponga che la tabella esterna delle pagine sia costituita di pagine di dimensione ordinaria (2^{10} elementi, o 2^{12} byte); uno spazio d'indirizzi a 64 bit è ancora enorme:

seconda pagina esterna	pagina esterna	pagina interna	scostamento
p_1	p_2	p_3	d
32	10	10	12

La tabella esterna delle pagine è ancora di 2^{34} byte.

Il passo successivo sarebbe uno schema di paginazione a quattro livelli, in cui si pagina anche la tabella esterna di secondo livello delle pagine. L'UltraSPARC a 64 bit richiederebbe sette livelli di paginazione – con un numero proibitivo di accessi alla memoria – per tradurre ciascun indirizzo logico. Da questo esempio è possibile capire perché, per le architetture a 64 bit, le tabelle delle pagine gerarchiche sono in genere considerate inappropriate.

8.5.2 Tabella delle pagine di tipo hash

Un metodo di gestione molto comune degli spazi d'indirizzi relativi ad architetture oltre i 32 bit consiste nell'impiego di una **tabella delle pagine di tipo hash**, in cui l'argomento della funzione hash è il numero della pagina virtuale. Per la gestione delle collisioni, ogni elemento della tabella hash contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa locazione. Ciascun elemento è composto da tre campi: (1) il nume-

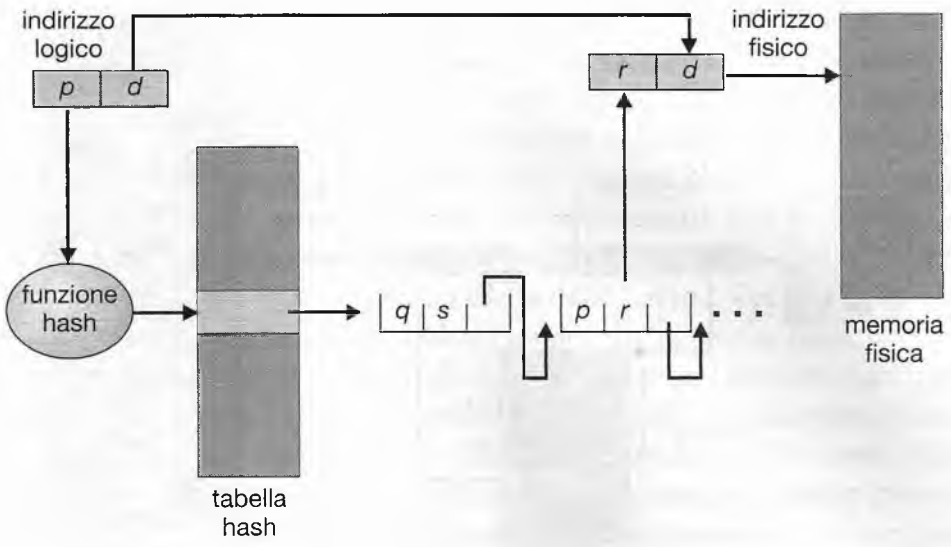


Figura 8.16 Tabella delle pagine di tipo hash.

ro della pagina virtuale; (2) l'indirizzo del frame (pagina fisica) corrispondente alla pagina virtuale; (3) un puntatore al successivo elemento della lista.

L'algoritmo opera come segue: si applica la funzione hash al numero della pagina virtuale contenuto nell'indirizzo virtuale, identificando un elemento della tabella. Si confronta il numero di pagina virtuale con il campo (1) del primo elemento della lista concatenata corrispondente. Se i valori coincidono, si usa l'indirizzo del relativo frame (campo 2) per generare l'indirizzo fisico desiderato. Altrimenti, l'algoritmo esamina allo stesso modo gli elementi successivi della lista concatenata (Figura 8.16). Le tabelle delle pagine di tipo hash sono particolarmente utili per gli spazi d'indirizzi **sparsi**, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio d'indirizzi.

Per questo schema è stata proposta una variante, adatta a spazi di indirizzamento a 64 bit. Si tratta della **tabella delle pagine a gruppi** (*clustered page table*), simile alla tabella hash; ciascun elemento della tabella delle pagine contiene però i riferimenti alle pagine fisiche corrispondenti a un gruppo di pagine virtuali contigue (per esempio 16). In questo modo si riduce lo spazio di memoria richiesto. Inoltre, poiché lo spazio degli indirizzi di molti programmi non è arbitrariamente sparso su pagine isolate ma distribuito per *raffiche* di riferimenti, le tabelle delle pagine a gruppi sfruttano bene questa caratteristica.

8.5.3 Tabella delle pagine invertita

Generalmente, si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, oppure un elemento per ogni indirizzo virtuale a prescindere dalla validità di quest'ultimo. Questa è una rappresentazione naturale della tabella, poiché i processi fanno riferimento alle pagine tramite gli indirizzi virtuali delle pagine stesse, che il sistema operativo deve poi tradurre in indirizzi di memoria fisica. Poiché la tabella è ordinata per indirizzi virtuali, il sistema operativo può calcolare in che punto della tabella si trova l'elemento dell'indirizzo fisico associato, e usare direttamente tale valore. Uno degli inconvenienti insiti in questo metodo è costituito dalla dimensione di ciascuna tabella delle pagine, che può contenere milioni di elementi e occupare grandi quantità di memoria fisica, necessaria proprio per sapere com'è impiegata la rimanente memoria fisica.

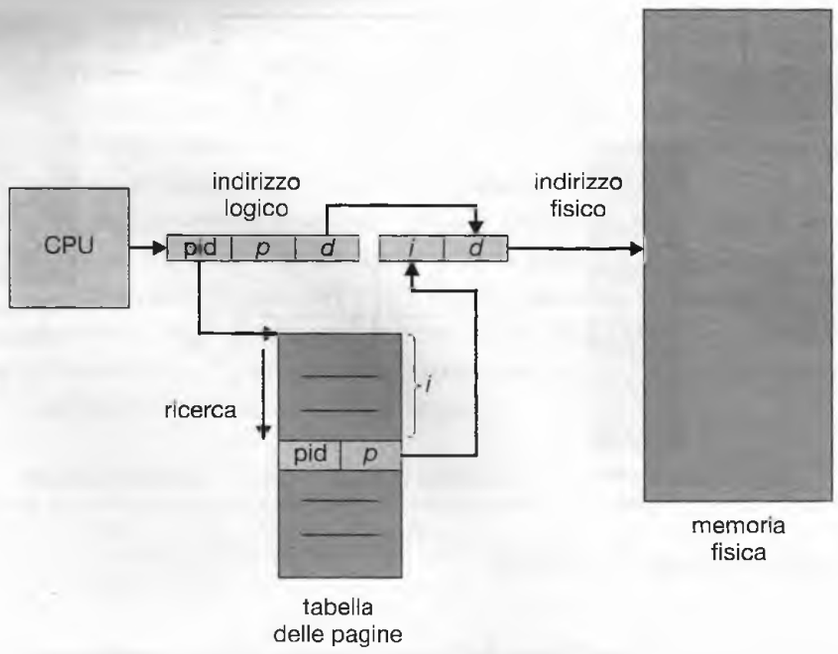


Figura 8.17 Tabella delle pagine invertita.

Per risolvere questo problema si può fare uso della **tabella delle pagine invertita**. Una tabella delle pagine invertita ha un elemento per ogni pagina reale (o frame). Ciascun elemento è quindi costituito dell'indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi, nel sistema esiste una sola tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica. Nella Figura 8.17 sono mostrate le operazioni di una tabella delle pagine invertita; si confronti questa figura con la Figura 8.7, che illustra il modo di operare per una tabella delle pagine ordinaria. Le tabelle invertite richiedono spesso la memorizzazione di un identificatore dello spazio d'indirizzi (Paragrafo 8.4.2) in ciascun elemento della tabella delle pagine, perché essa contiene di solito molti spazi d'indirizzi diversi associati alla memoria fisica; l'identificatore garantisce che una data pagina logica relativa a un certo processo sia associata alla pagina fisica corrispondente. Esempi di architetture che usano le tabelle delle pagine invertite sono l'UltraSPARC a 64 bit e la PowerPC.

Per illustrare questo metodo è possibile descrivere una versione semplificata della tabella delle pagine invertita dell'IBM RT. Ciascun indirizzo virtuale è una tripla del tipo seguente:

<*id-processo*, *numero di pagina*, *scostamento*>

Ogni elemento della tabella delle pagine invertita è una coppia <*id-processo*, *numero di pagina*> dove l'*id-processo* assume il ruolo di identificatore dello spazio d'indirizzi. Quando si fa un riferimento alla memoria, si presenta una parte dell'indirizzo virtuale, formato da <*id-processo*, *numero di pagina*>, al sottosistema di memoria. Quindi si cerca una corrispondenza nella tabella delle pagine invertita. Se si trova tale corrispondenza, per esempio sull'elemento *i*, si genera l'indirizzo fisico <*i*, *scostamento*>. In caso contrario è stato tentato un accesso illegale a un indirizzo.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella quando si fa riferi-

mento a una pagina. Poiché la tabella delle pagine invertita è ordinata per indirizzi fisici, mentre le ricerche si fanno per indirizzi virtuali, per trovare una corrispondenza occorre esaminare tutta la tabella; questa ricerca richiede molto tempo. Per limitare l'entità del problema si può impiegare una tabella hash (come si descrive nel Paragrafo 8.5.2), che riduce la ricerca a un solo, o a pochi, elementi della tabella delle pagine. Naturalmente, ogni accesso alla tabella hash aggiunge al procedimento un riferimento alla memoria, quindi un riferimento alla memoria virtuale richiede almeno due letture della memoria reale: una per l'elemento della tabella hash e l'altro per la tabella delle pagine. Per migliorare le prestazioni, la ricerca si effettua prima nella TLB, quindi si consulta la tabella hash.

Nei sistemi che adottano le tabelle delle pagine invertite, l'implementazione della memoria condivisa è difficoltosa. Difatti, la condivisione si realizza solitamente tramite indirizzi virtuali multipli (uno per ogni processo che partecipa alla condivisione) associati a un unico indirizzo fisico. Il metodo è pero inutilizzabile in presenza di tabelle invertite, perché, essendovi un solo elemento indicante la pagina virtuale corrispondente a ogni pagina fisica, questa non può avere più di un indirizzo virtuale associato. Una semplice tecnica per superare il problema consiste nel porre nella tabella delle pagine una sola associazione fra un indirizzo virtuale e l'indirizzo fisico condiviso; ciò comporta un errore dovuto all'assenza della pagina per ogni riferimento agli indirizzi virtuali non associati (*page fault*).

8.6 Segmentazione

Un aspetto importante della gestione della memoria, inevitabile alla presenza della paginazione, è quello della separazione tra la visione della memoria dell'utente e l'effettiva memoria fisica. Lo spazio d'indirizzi *visto* dall'utente non coincide con l'effettiva memoria fisica, ma lo si fa corrispondere alla memoria fisica. I metodi che stabiliscono questa corrispondenza consentono di separare la memoria logica dalla memoria fisica.

8.6.1 Metodo di base

Ci si potrebbe chiedere se l'utente può considerare la memoria come un vettore lineare di byte, alcuni dei quali contengono istruzioni e altri dati. Molti risponderebbero di no. Gli utenti la vedono piuttosto come un insieme di segmenti di dimensione variabile non necessariamente ordinati (Figura 8.18).

La tipica struttura di un programma con cui i programmatori hanno familiarità è costituita di una parte principale e di un gruppo di procedure, funzioni o moduli, insieme con diverse strutture dati come tabelle, matrici, pile, variabili e così via. Ciascuno di questi moduli o elementi di dati si identifica con un nome: "tabella dei simboli", "funzione `sqrt()`", "programma principale", indipendentemente dagli indirizzi che questi elementi occupano in memoria. Non è necessario preoccuparsi del fatto che la tabella dei simboli sia memorizzata prima o dopo la funzione `sqrt()`. Ciascuno di questi segmenti ha una lunghezza variabile, definita intrinsecamente dallo scopo che il segmento stesso ha all'interno del programma. Gli elementi che si trovano all'interno di un segmento sono identificati dal loro scostamento, misurato dall'inizio del segmento: la prima istruzione del programma, il settimo elemento della tabella dei simboli, la quinta istruzione della funzione `sqrt()`, e così via.

La **segmentazione** è uno schema di gestione della memoria che consente di gestire questa rappresentazione della memoria dal punto di vista dell'utente. Uno spazio d'indirizzi

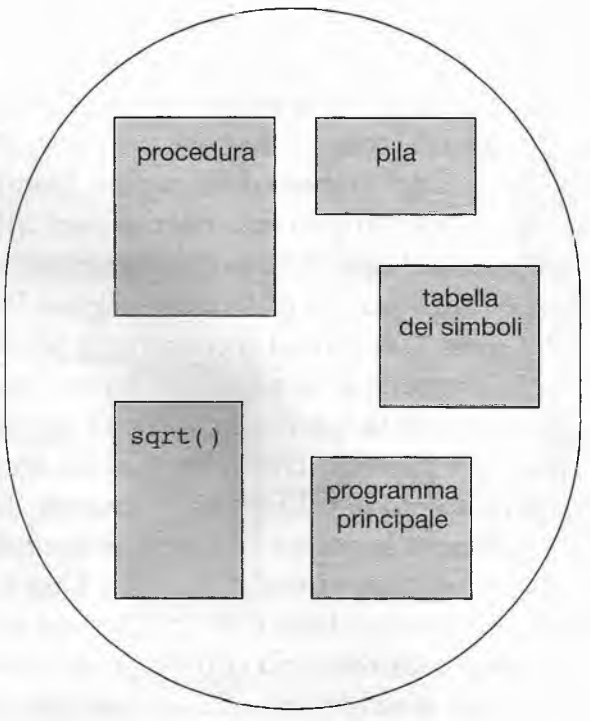


Figura 8.18 Un programma dal punto di vista dell'utente.

logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indirizzi specificano sia il nome sia lo scostamento all'interno del segmento, quindi l'utente fornisce ogni indirizzo come una coppia ordinata di valori: un nome di segmento e uno scostamento. Questo schema contrasta con la paginazione, in cui l'utente fornisce un indirizzo singolo, che l'architettura di paginazione suddivide in un numero di pagina e uno scostamento, non visibili dal programmatore.

Per semplicità i segmenti sono numerati, e ogni riferimento si compie per mezzo di un numero anziché di un nome; quindi un indirizzo logico è una *coppia*

<numero di segmento, scostamento>

Normalmente il programma utente è stato compilato, e il compilatore struttura automaticamente i segmenti secondo il programma sorgente. Un compilatore per il linguaggio C può creare segmenti distinti per i seguenti elementi di un programma:

1. il codice;
2. le variabili globali;
3. lo heap, da cui si alloca la memoria;
4. le pile usate da ciascun thread;
5. la libreria standard del C.

Alle librerie collegate dal linker al momento della compilazione possono essere assegnati dei nuovi segmenti. Il caricatore preleva questi segmenti e assegna loro i numeri di segmento.

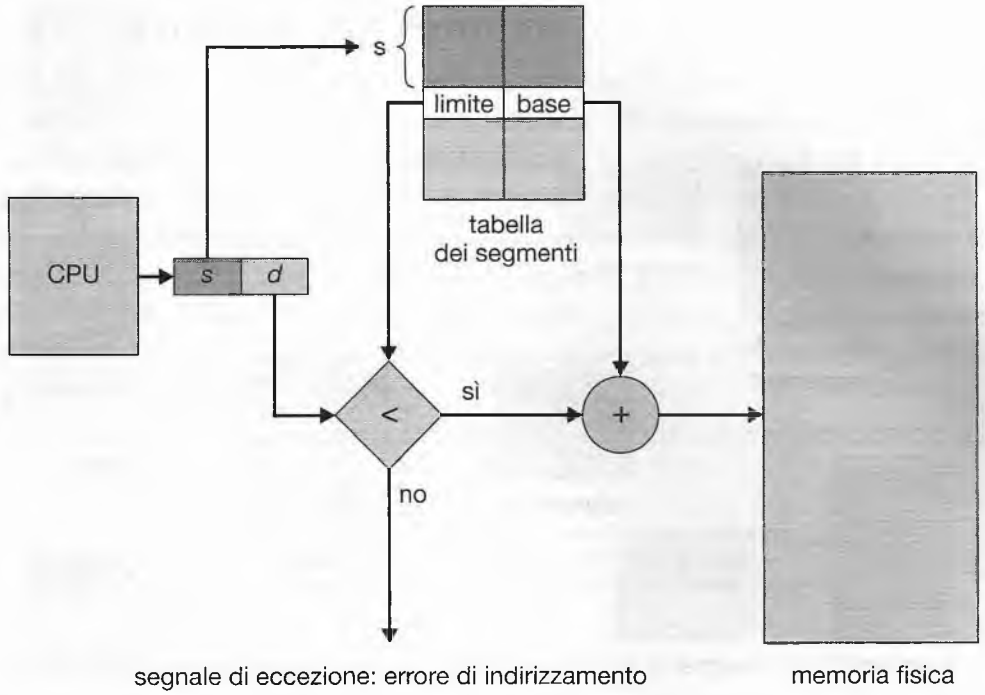


Figura 8.19 Architettura di segmentazione.

8.6.2 Architettura di segmentazione

Sebbene l'utente possa far riferimento agli oggetti del programma per mezzo di un indirizzo bidimensionale, la memoria fisica è in ogni caso una sequenza di byte unidimensionale. Per questo motivo occorre tradurre gli indirizzi bidimensionali definiti dall'utente negli indirizzi fisici unidimensionali. Questa operazione si compie tramite una **tabella dei segmenti**; ogni suo elemento è una coppia ordinata: la *base del segmento* e il *limite del segmento*. La base del segmento contiene l'indirizzo fisico iniziale della memoria dove il segmento risiede, mentre il limite del segmento contiene la lunghezza del segmento.

L'uso della tabella dei segmenti è illustrato nella Figura 8.19. Un indirizzo logico è formato da due parti: un numero di segmento *s* e uno scostamento in tale segmento *d*. Il numero di segmento si usa come indice per la tabella dei segmenti; lo scostamento *d* dell'indirizzo logico deve essere compreso tra 0 e il limite del segmento, altrimenti s'invia un segnale di eccezione al sistema operativo (tentativo di indirizzamento logico oltre la fine del segmento). Se tale condizione è rispettata, si somma lo scostamento alla base del segmento per produrre l'indirizzo della memoria fisica dove si trova il byte desiderato. Quindi la tabella dei segmenti è fondamentalmente un vettore di coppie di registri di base e limite.

Come esempio si può considerare la situazione illustrata nella Figura 8.20. Sono dati cinque segmenti numerati da 0 a 4, memorizzati in memoria fisica. La tabella dei segmenti ha un elemento distinto per ogni segmento, indicante l'indirizzo iniziale del segmento in memoria fisica (la base) e la lunghezza di quel segmento (il limite). Per esempio, il segmento 2 è lungo 400 byte e inizia alla locazione 4300, quindi un riferimento al byte 53 del segmento 2 si fa corrispondere alla locazione 4300 + 53 = 4353. Un riferimento al segmento 3, byte 852, si fa corrispondere alla locazione 3200 (la base del segmento 3) + 852 = 4052. Un riferimento al byte 1222 del segmento 0 causa l'invio di un segnale di eccezione al sistema operativo, poiché questo segmento è lungo 1000 byte.

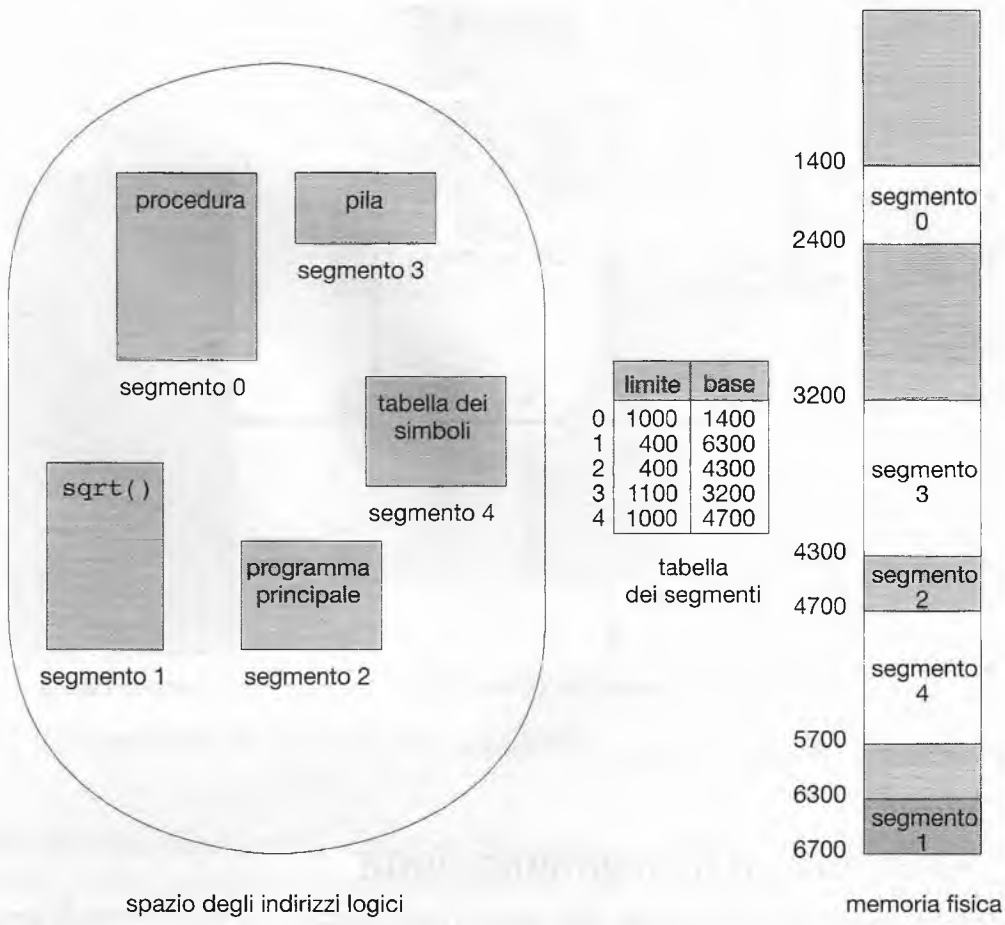


Figura 8.20 Esempio di segmentazione.

8.7 Un esempio: Pentium Intel

Paginazione e segmentazione presentano vantaggi e svantaggi, tanto che alcune architetture dispongono di entrambe le tecniche. In questo paragrafo prendiamo in esame l'architettura del Pentium Intel che utilizza, oltre alla segmentazione pura, la segmentazione mista a paginazione. Non ci inoltreremo in una trattazione completa della gestione della memoria del Pentium, ma ci limiteremo a illustrare i concetti fondamentali su cui è basata. La nostra analisi si concluderà con una panoramica della traduzione degli indirizzi di Linux nei sistemi Pentium.

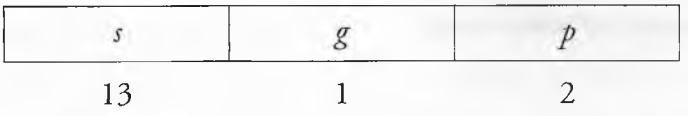
In questi sistemi la CPU genera indirizzi logici, che confluiscono nell'unità di segmentazione. Questa produce un indirizzo lineare per ogni indirizzo logico. L'indirizzo lineare passa quindi all'unità di paginazione, la quale, a sua volta, genera l'indirizzo fisico all'interno della memoria centrale. Così, le unità di segmentazione e di paginazione formano un equivalente dell'unità di gestione della memoria (MMU). Il modello è rappresentato nella Figura 8.21.

Figura 8.21 Traduzione degli indirizzi logici in indirizzi fisici in Pentium.

8.7.1 Segmentazione in Pentium

Nell'architettura Pentium un segmento può raggiungere la dimensione massima di 4 GB; il numero massimo di segmenti per processo è pari a 16 KB. Lo spazio degli indirizzi logici di un processo è composto da due partizioni: la prima contiene fino a 8 KB segmenti riservati al processo; la seconda contiene fino a 8 KB segmenti condivisi fra tutti i processi. Le informazioni riguardanti la prima partizione sono contenute nella **tabella locale dei descrittori** (*local descriptor table*, LDT), quelle relative alla seconda partizione sono memorizzate nella **tabella globale dei descrittori** (*global descriptor table*, GDT). Ciascun elemento nella LDT e nella GDT è lungo 8 byte e contiene informazioni dettagliate riguardanti uno specifico segmento, oltre agli indirizzi base e limite.

Un indirizzo logico è una coppia (*selettore*, *scostamento*), dove il selettore è un numero di 16 bit:



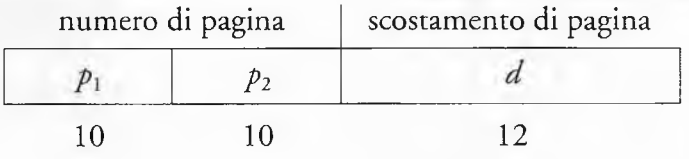
in cui *s* indica il numero del segmento, *g* indica se il segmento si trova nella GDT o nella LDT e *p* contiene informazioni relative alla protezione. Lo scostamento è un numero di 32 bit che indica la posizione del byte (o della parola) all'interno del segmento in questione.

La macchina ha sei registri di segmento che consentono a un processo di far riferimento contemporaneamente a sei segmenti; inoltre possiede sei registri di microprogramma di 8 byte per i corrispondenti descrittori prelevati dalla LDT o dalla GDT. Questa cache evita a Pentium di dover prelevare dalla memoria i descrittori per ogni riferimento alla memoria.

Un indirizzo lineare di Pentium è lungo 32 bit e si genera come segue. Il registro di segmento punta all'elemento appropriato all'interno della LDT o della GDT; le informazioni relative alla base e al limite di tale segmento si usano per generare un **indirizzo lineare**. Innanzi tutto si usa il valore del limite per controllare la validità dell'indirizzo; se non è valido, si ha un errore di riferimento alla memoria che causa l'invio di un segnale di eccezione e la restituzione del controllo al sistema operativo; altrimenti, si somma il valore dello scostamento al valore della base, ottenendo un indirizzo lineare di 32 bit. La Figura 8.22 illustra tale processo. Nel paragrafo successivo si considera come l'unità di paginazione trasforma questo indirizzo lineare in un indirizzo fisico.

8.7.2 Paginazione in Pentium

L'architettura Pentium prevede che le pagine abbiano una misura di 4 KB oppure di 4 MB. Per le prime, in Pentium vige uno schema di paginazione a due livelli che prevede la seguente scomposizione degli indirizzi lineari a 32 bit:



Lo schema di traduzione degli indirizzi per questa architettura, simile a quello rappresentato nella Figura 8.15, è mostrato in dettaglio nella Figura 8.23. I dieci bit più significativi puntano a un elemento nella tabella delle pagine più esterna, detta in Pentium **directory**

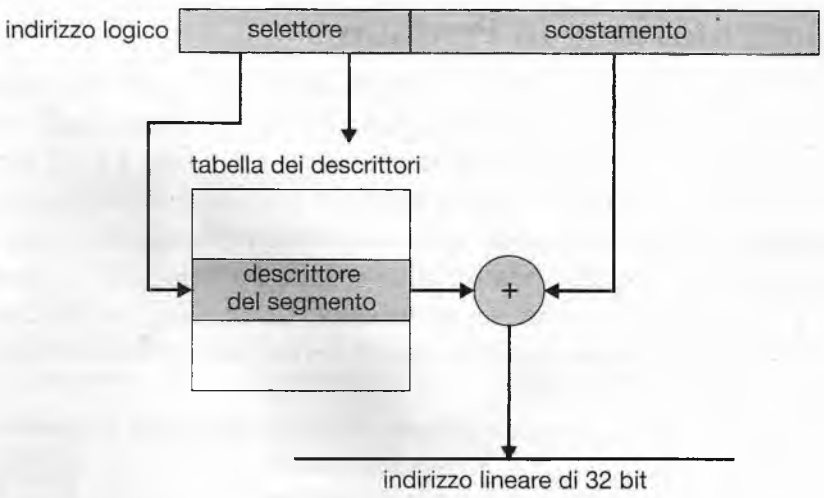


Figura 8.22 Segmentazione in Pentium Intel.

delle pagine. (Il registro `CR3` punta alla directory delle pagine del processo corrente.) Gli elementi della directory delle pagine puntano a una tabella delle pagine interne, indicizzata da dieci bit intermedi dell'indirizzo lineare. Infine, i bit meno significativi in posizione 0-11 contengono lo scostamento da applicare all'interno della pagina di 4 KB cui si fa riferimento nella tabella delle pagine.

Un elemento appartenente alla directory delle pagine è il flag `Page Size`; se impostato, indica che il frame non ha la dimensione standard di 4 MB, ma misura invece 4 KB. In questo caso, la directory di pagina punta direttamente al frame di 4 MB, scavalcando la tabella delle pagine interna; i 22 bit meno significativi nell'indirizzo lineare indicano lo scostamento nella pagina di 4 MB.

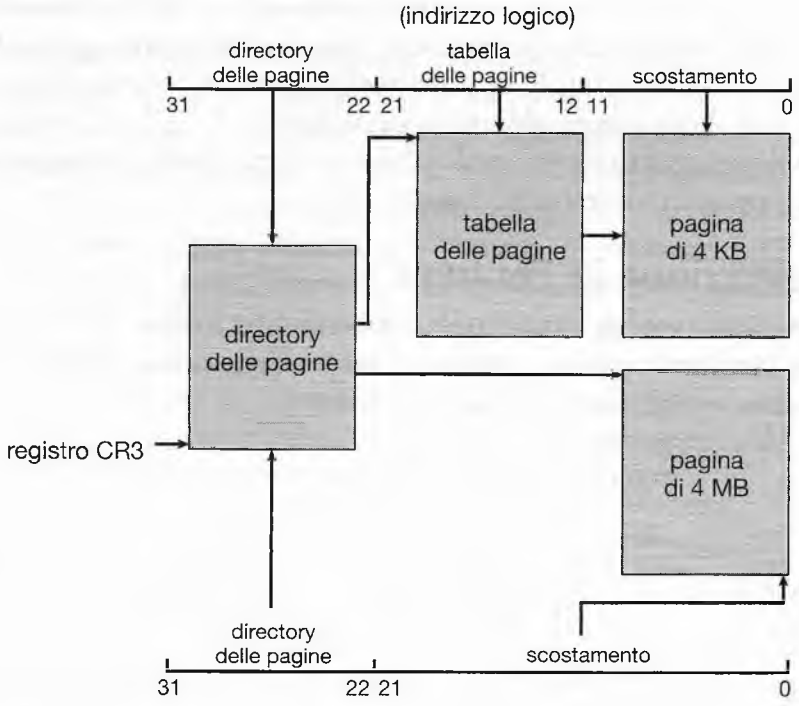


Figura 8.23 Paginazione nell'architettura Pentium.

Per migliorare l'efficienza d'uso della memoria fisica, le tabelle delle pagine del Pentium Intel possono essere trasferite sul disco. In questo caso, si ricorre a un bit in ciascun elemento della directory di pagina, per indicare se la tabella a cui l'elemento punta sia in memoria o sul disco. Nel secondo caso, il sistema operativo può usare i 31 bit rimanenti per specificare la collocazione della tabella sul disco; in questo modo, si può richiamare la tabella in memoria su richiesta.

8.7.3 Linux su sistemi Pentium

Si consideri, a scopo di esempio, il sistema operativo Linux montato sull'architettura Pentium Intel. Linux può essere eseguito dai processori più disparati, molti dei quali scarsamente predisposti alla segmentazione. Per questo motivo Linux non attribuisce un ruolo rilevante alla segmentazione, di cui fa un uso minimo. Su Pentium, Linux utilizza soltanto sei segmenti:

1. un segmento per il codice del kernel;
2. un segmento per i dati del kernel;
3. un segmento per il codice dell'utente;
4. un segmento per i dati dell'utente;
5. un segmento per lo stato del task (*task-state segment*, TSS);
6. un segmento di default per la tabella locale dei descrittori (LDT).

I segmenti per il codice e i dati dell'utente sono condivisi da tutti i processi eseguiti in modalità utente. Ciò è possibile perché tutti i processi usano il medesimo spazio degli indirizzi logici, e tutti i descrittori dei segmenti risiedono nella tabella globale dei descrittori (*global descriptor table*, GDT). Il TSS serve a memorizzare il contesto hardware dei processi durante il cambio di contesto. Il segmento LDT di default è solitamente condiviso da tutti i processi, ma inutilizzato; i processi che lo desiderino, tuttavia, possono creare e usare il proprio LDT privato.

Come s'è detto, i selettori dei segmenti comprendono un campo di due bit dedicato alla protezione. Pentium contempla pertanto 4 livelli di protezione, dei quali solo due sfruttati da Linux: modalità utente e kernel.

Sebbene Pentium sia dotato di uno schema di paginazione a due livelli, Linux è compatibile con un'intera gamma di piattaforme hardware, molte delle quali, essendo a 64 bit, rendono i due livelli poco consigliabili. Linux adotta pertanto una strategia di paginazione a tre livelli che funziona bene sia sulle architetture a 64 bit sia su quelle a 32 bit.

Un indirizzo lineare di Linux è composta da quattro parti:

directory globale	directory intermedia	tabella delle pagine	scostamento

La Figura 8.24 illustra il modello di paginazione a tre livelli di Linux.

Il numero di bit delle quattro parti dell'indirizzo lineare dipende dall'architettura. Tuttavia, come s'è osservato poc'anzi, Pentium prevede solo due livelli di paginazione. È naturale dunque chiedersi come Linux applichi il proprio schema a tre livelli a Pentium. La risposta è che, in questo caso, la lughezza intermedia del campo directory è di zero bit, ciò che porta a ignorare la directory intermedia.

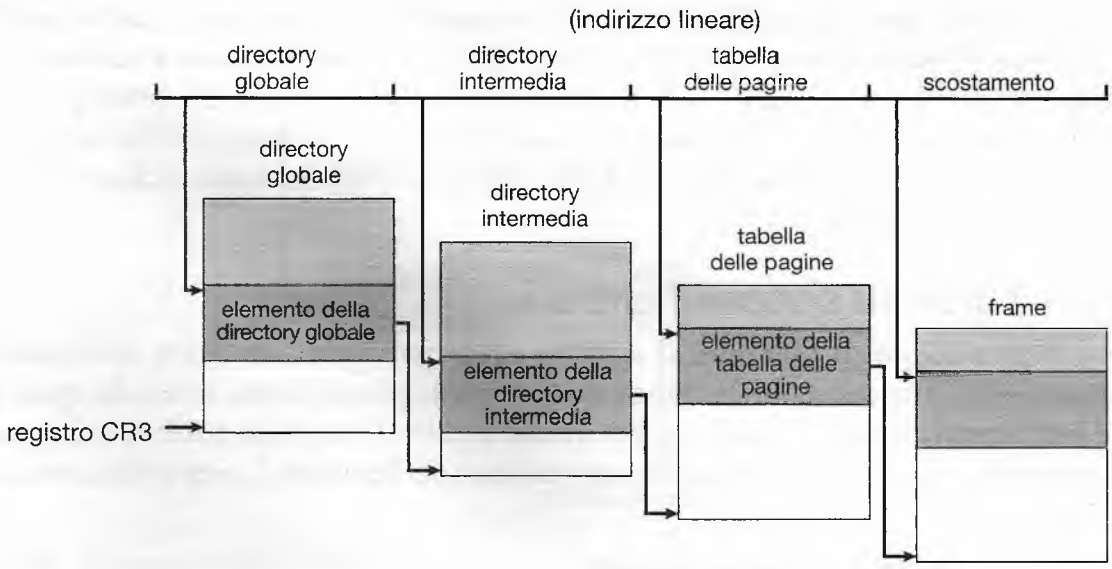


Figura 8.24 Paginazione a tre livelli di Linux.

Ogni task di Linux possiede il proprio insieme di tabelle delle pagine, come illustrato dalla Figura 8.23; il registro CR3 punta alla directory globale del task correntemente in esecuzione. Durante il cambio di contesto, il valore di CR3 è salvato nei segmenti TSS dei task coinvolti dal cambio.

8.8 Sommario

Gli algoritmi di gestione della memoria per sistemi operativi multiprogrammati variano da un metodo semplice, per sistema con singolo utente, fino alla segmentazione paginata. Il fattore più rilevante che incide maggiormente sulla scelta del metodo da seguire in un sistema particolare è costituito dall'architettura disponibile. È necessario controllare la validità di ogni indirizzo di memoria generato dalla CPU; inoltre tali indirizzi, se sono corretti, devono essere tradotti in un indirizzo fisico. Tali controlli per essere efficienti devono essere effettuati dall'architettura del sistema, che secondo le sue caratteristiche pone vincoli al sistema operativo.

Gli algoritmi di gestione della memoria analizzati (allocazione contigua, paginazione, segmentazione e combinazione di paginazione e segmentazione) differiscono per molti aspetti. Per confrontare i diversi metodi di gestione della memoria si possono considerare i seguenti elementi.

- **Architettura.** Un semplice registro di base oppure una coppia di registri di base e limite è sufficiente per i metodi con partizione singola e con più partizioni, mentre la paginazione e la segmentazione necessitano di tabelle di traduzione per definire la corrispondenza degli indirizzi.
- **Prestazioni.** Aumentando la complessità dell'algoritmo, aumenta anche il tempo necessario per tradurre un indirizzo logico in un indirizzo fisico. Nei sistemi più semplici è sufficiente fare un confronto o sommare un valore all'indirizzo logico; si tratta di operazioni rapide. Paginazione e segmentazione possono essere altrettanto rapide se

per realizzare la tabella s'impiegano registri veloci. Se però la tabella si trova in memoria, gli accessi alla memoria utente possono essere assai più lenti. Una TLB può limitare il calo delle prestazioni a un livello accettabile.

- **Frammentazione.** Un sistema multiprogrammato esegue le elaborazioni generalmente in modo più efficiente se ha un più elevato livello di multiprogrammazione. Per un dato gruppo di processi, il livello di multiprogrammazione si può aumentare solo compattando più processi in memoria. Per eseguire questo compito occorre ridurre lo spreco di memoria o la frammentazione. Sistemi con unità di allocazione di dimensione fissa, come lo schema con partizione singola e la paginazione, soffrono di frammentazione interna. Sistemi con unità di allocazione di dimensione variabile, come lo schema con più partizioni e la segmentazione, soffrono di frammentazione esterna.
- **Rilocazione.** Una soluzione al problema della frammentazione esterna è data dalla compattazione, che implica lo spostamento di un programma in memoria, senza che il programma stesso *si accorga* del cambiamento. Ciò richiede che gli indirizzi logici siano rilocati dinamicamente al momento dell'esecuzione. Se gli indirizzi si rilocano solo al momento del caricamento, non è possibile compattare la memoria.
- **Avvicendamento dei processi.** L'avvicendamento dei processi (*swapping*) si può incorporare in ogni algoritmo. I processi si copiano dalla memoria centrale alla memoria ausiliaria, e successivamente si ricopiano in memoria centrale a intervalli fissati dal sistema operativo, e generalmente stabiliti dai criteri di scheduling della CPU. Questo schema permette di inserire contemporaneamente in memoria più processi da eseguire.
- **Condivisione.** Un altro mezzo per aumentare il livello di multiprogrammazione è quello della condivisione del codice e dei dati tra diversi utenti. Poiché deve fornire piccoli pacchetti d'informazioni (pagine o segmenti) condivisibili, generalmente richiede l'uso della paginazione o della segmentazione. La condivisione permette di eseguire molti processi con una quantità di memoria limitata, ma i programmi e i dati condivisi si devono progettare con estrema cura.
- **Protezione.** Con la paginazione o la segmentazione, diverse sezioni di un programma utente si possono dichiarare di sola esecuzione, di sola lettura oppure di lettura e scrittura. Questa limitazione è necessaria per il codice e i dati condivisi, ed è utile, in genere, nei casi in cui sono richiesti semplici controlli nella fase d'esecuzione per l'individuazione degli errori di programmazione.

Esercizi pratici

8.1 Citate due differenze tra indirizzi logici e fisici.

8.2 Considerate un sistema nel quale un programma possa essere separato in due parti: codice e dati. Il processore sa se necessita di un'istruzione (prelievo di istruzione) o dati (prelievo o memorizzazione di dati). Perciò, vengono fornite due coppie di registri base e limite: una per le istruzioni e una per i dati. La coppia di registri base e limite per le istruzioni è automaticamente a sola lettura, di modo che i programmi possano essere condivisi tra i diversi utenti. Discutete i vantaggi e gli svantaggi di questo schema.