**Introduzione**

La memoria RAM è una risorsa che va gestita attentamente. La parte del SO che gestisce la memoria è chiamata gestore della memoria. Il suo compito è gestire con efficienza la memoria: tener traccia di quali parti di memoria sono in uso, allocare memoria ai processi secondo le loro necessità e liberarla a lavoro terminato.

**Nessuna astrazione di memoria**

Il modo più semplice per gestire la memoria è senza forme di astrazione. Ogni programma vede semplicemente la memoria fisica, così com’è (numero di celle e dimensione del contenuto).

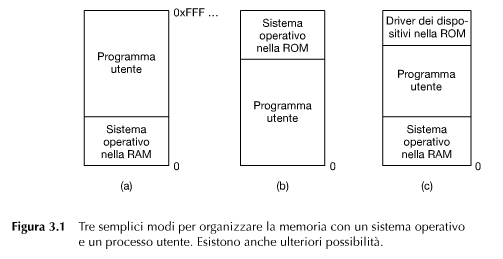
I primi mainframe (prima del 1960), i primi minicomputer (prima del 1970) e i primi PC (prima del 1980) non avevano astrazione della memoria.

In queste condizioni non era possibile avere due programmi eseguiti in memoria nel medesimo istante: i due programmi si sarebbero sovrapposti.

Anche con il solo modello di memoria fisica esistono diverse possibilità. Nella Figura 3.1 sono illustrate tre varianti. Il sistema operativo può trovarsi sul fondo della memoria nella RAM (Figura 3.1(a)), in cima alla memoria nella ROM (Figura 3.1(b)), i driver dei dispositivi possono essere in cima alla memoria nella ROM e il resto del sistema nella RAM subito sotto

(Figura 3.1(c)).

Il terzo modello è stato usato sui primi PC, dove la parte del sistema operativo nella ROM è chiamata BIOS (Basic Input Output System). I modelli (a) e (c) hanno lo svantaggio che un difetto nel programma utente può cancellare il sistema operativo, con conseguenze disastrose.



**Esecuzione di molteplici programmi senza astrazione della memoria**

Anche senza l’astrazione di memoria, è possibile fornire all’utente l’idea di avere in esecuzione più programmi contemporaneamente.

Il sistema operativo salva l’intero contenuto della memoria in un file sul disco, carica ed esegue il programma successivo. Finché in memoria vi è un solo programma per volta, non vi sono conflitti. Questo concetto è chiamato swapping.

Con l’aggiunta di hardware speciale, è possibile eseguire molteplici programmi contemporaneamente, anche senza swapping.

La mancanza di una astrazione della memoria è tuttora abituale nei sistemi integrati, smart card, radio, lavatrici, ecc… Il software indirizza la memoria in modo assoluto e funziona solo perché i programmi sono noti in anticipo e gli utenti non sono liberi di eseguire il proprio software.

**Un’astrazione della memoria: gli spazi degli indirizzi**

Esporre la memoria fisica ai processi presenta tanti grossi inconvenienti. Punto primo, se i programmi utente possono indirizzare ogni byte della memoria, allora possono facilmente spazzar via il sistema operativo; punto secondo, con questo modello è difficile che siano eseguiti contemporaneamente molteplici programmi.

**Nozione di spazio degli indirizzi**

Per permettere a molteplici applicazioni di risiedere in memoria contemporaneamente senza interferire l’un l’altra devono essere risolti due problemi: la protezione e il riposizionamento.

Una soluzione primitiva è stata usata sull’IBM 360: etichettare grossi pezzi di memoria con una chiave di protezione e confrontare la chiave del processo in esecuzione con quella di ogni parola di memoria prelevata. Questo approccio tuttavia non risolve il secondo problema, sebbene si possano riposizionare i programmi quando vengono caricati, ma si tratta di una soluzione lenta e complicata.

Una soluzione migliore è inventare una nuova astrazione per la memoria: lo spazio degli indirizzi. Così come il concetto di processo crea una specie di CPU astratta per eseguire i programmi, lo spazio degli indirizzi crea una specie di memoria astratta per farci vivere i programmi.

Uno spazio degli indirizzi è l’insieme degli indirizzi che un processo può usare per indirizzare la memoria. Ogni processo ha il suo spazio degli indirizzi personale, indipendente da quello appartenente ad altri processi.

Il concetto di spazio degli indirizzi è molto generale ed è appropriato in molti contesti (numeri di telefono, porte I/O, indirizzi IPv4, domini internet).

Più difficile è come dare a ogni programma il suo spazio degli indirizzi: l’indirizzo 28 in un programma significa una locazione fisica diversa dall’indirizzo 28 in un altro programma.

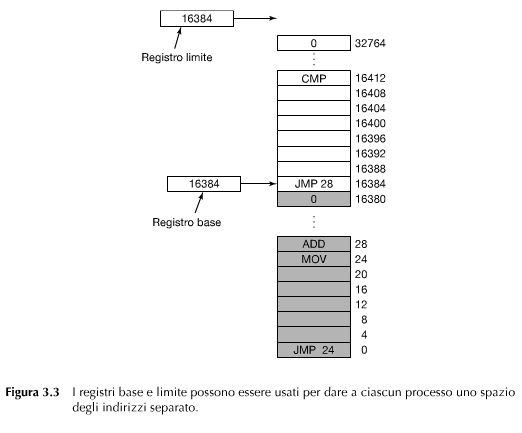
**Registri base e registri limite**

Questa soluzione si basa su una versione particolarmente semplice della rilocazione dinamica. Quello che fa è mappare lo spazio degli indirizzi di ogni processo su di una parte diversa di memoria fisica in un modo semplice.

La soluzione classica è di dotare ogni CPU con due registri hardware speciali, solitamente chiamati registro base e registro limite. Quando sono usati questi registri, i programmi sono caricati in posizioni di memoria consecutive dovunque vi sia spazio e senza riposizionamento durante il caricamento.

Al momento dell’esecuzione di un processo, il registro base è caricato con l’indirizzo fisico dove comincia il suo programma in memoria e il registro limite è caricato con la lunghezza del programma.

Ogni volta che un processo consulta la memoria, sia per prelevare un’istruzione sia per leggere o scrivere una parola dati, prima di inviare l’indirizzo sul bus di memoria l’hardware della CPU aggiunge automaticamente il valore di base all’indirizzo generato tramite il processo. Contemporaneamente controlla se l’indirizzo offerto sia uguale o maggiore del valore nel registro limite, nel cui caso è generato un errore l’accesso viene interrotto.



In molte implementazioni i registri limite e base sono protetti in modo che solo il sistema operativo possa modificarli.

Uno svantaggio della rilocazione per mezzo dei registri limite e base è la necessità di eseguire una somma e un confronto con ogni riferimento della memoria.

I confronti possono avvenire velocemente, ma le somme sono lente a causa del tempo di propagazione, a meno che non siano usati circuiti speciali per la somma.

**Swapping**

Se la memoria fisica del computer è abbastanza ampia da contenere tutti i processi, gli schemi descritti finora più o meno funzioneranno. Nella realtà tuttavia la quantità totale di RAM necessaria per tutti i processi è spesso molto di più di quanto possa stare in memoria.

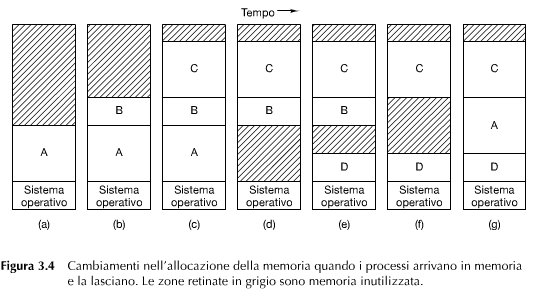
Per gestire tale problematica sono possibili due approcci.

La strategia più semplice, chiamata swapping, consiste nel prendere ciascun processo nella sua totalità, eseguirlo per un certo tempo, quindi rimetterlo nuovamente su disco. I processi inattivi vengono per la maggior parte archiviati su disco, così non portano via memoria mentre non sono in esecuzione.

L’altra strategia, chiamata memoria virtuale, consente ai programmi di essere eseguiti anche quando sono solo parzialmente nella memoria principale.

Il funzionamento di un sistema di swapping è illustrato nella Figura 3.4.

All’inizio solo il processo A è in memoria. Quindi vengono creati o viene fatto lo swapping dal disco dei processi B e C. Nella Figura 3.4(d) viene fatto lo swapping su disco di A. Quindi arriva D ed esce B. Alla fine A ritorna nuovamente. Poiché ora A è in una posizione diversa, i suoi indirizzi devono essere rilocati dal software quando viene fatto lo swapping o dall’hardware durante l’esecuzione del programma.

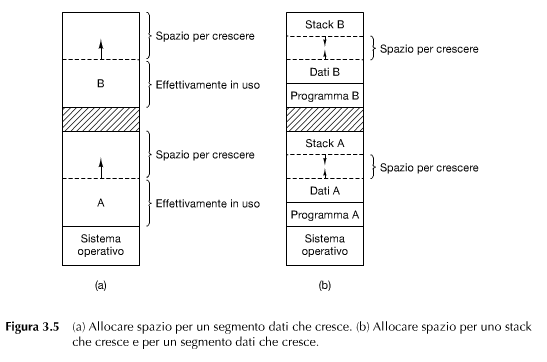


Quando lo swapping crea molteplici spazi vuoti nella memoria, è possibile combinarli tutti in un unico spazio vuoto spostando tutti i processi il più in basso possibile. Questa tecnica è conosciuta come memory compaction. Di solito non viene fatta perché richiede molto tempo di CPU.

Un punto da evidenziare è quanta memoria dovrebbe essere allocata per un processo quando viene creato o quando ne viene fatto lo swapping dal disco.

Se i processi hanno una dimensione fissa, l’allocazione è semplice: il sistema operativo conosce esattamente di quanto ha bisogno, alloca ciò che è necessario.

Se i processi crescono di dimensione, per esempio allocando dinamicamente memoria dallo heap, sorge un problema. Se vi è spazio adiacente al processo, esso può essere allocato e il processo può crescere in tale spazio. D’altra parte, se il processo ha un altro processo adiacente, bisognerà trovare uno spazio di memoria sufficientemente largo ad ospitarlo. Una soluzione è allocare un po’ di memoria extra per prevenire la sua crescita.



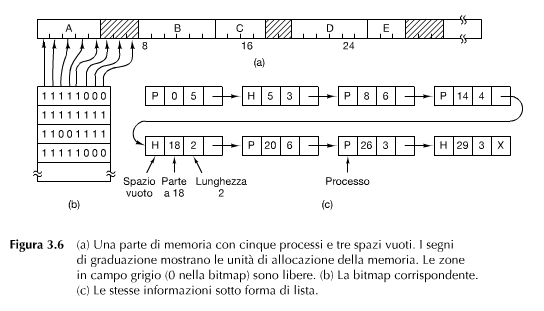
**Gestione della memoria libera**

Quando la memoria è assegnata dinamicamente, il sistema operativo deve gestirla.

Ci sono due modalità per tener traccia dell’utilizzo della memoria: bitmap e liste.

**Gestione della memoria con bitmap**

La memoria, con una bitmap, è divisa in unità di allocazione piccole come qualche parola o grandi come molti kilobyte. A ogni unità di allocazione corrisponde un bit della bitmap, che è 0 se l’unità è libera e 1 se è utilizzata. La Figura 3.6 mostra parte della memoria e la sua corrispondente bitmap.



In questo sistema è cruciale la scelta della dimensione dell’unità di allocazione. Più è piccola la dimensione dell’unità di allocazione, maggiore è la quantità di memoria occupata dalla bitmap; più è grande la dimensione dell’unità di allocazione, minore è la dimensione della bitmap ma aumenta la quantità non banale di memoria che potrebbe essere sprecata nell’ultima unità del processo.

**Gestione della memoria con liste collegate**

Un altro sistema per tenere traccia della memoria è mantenere delle liste di segmenti di memoria allocati e liberi, in cui un segmento o contiene un processo o è uno spazio vuoto fra due processi. La memoria della Figura 3.6(a) è rappresentata nella Figura 3.6(c) come una lista collegata di segmenti. Ogni voce della lista specifica uno spazio vuoto (H) o un processo (P), l’indirizzo da cui parte, la lunghezza e il puntatore alla voce successiva.

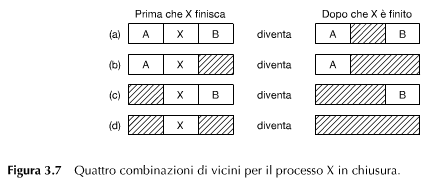
In questo esempio la lista dei segmenti è in ordine di indirizzo. Fare questo ordinamento ha il vantaggio che quando il processo finisce o ne viene fatto lo swapping, l’aggiornamento della lista è molto semplice.

Un processo che finisce ha normalmente due vicini. Questi possono essere sia processi sia spazi vuoti, il che porta alle quattro combinazioni della Figura 3.7(d).

Nella Figura 3.7(a) aggiornare la lista porta a rimpiazzare una P con una H.

Nella Figura 3.7(b) e 3.7(c) due voci sono fuse insieme e la lista si accorcia di una voce.

Nella Figura 3.7(d) tre voci sono fuse insieme e due elementi sono rimossi dalla lista.



Quando la lista è ordinata per indirizzo, molti algoritmi possono essere usati per allocare la memoria per un processo creato o di cui viene fatto lo swapping dal disco.

Troviamo i seguenti algoritmi:

- *first fit* (il primo va bene): il gestore della memoria scorre finché uno spazio vuoto abbastanza grande;

- *next fit* (il prossimo va bene): lavora allo stesso modo del precedente, ma tiene traccia di ogni posto dove ha trovato uno spazio adatto. La volta dopo che è interpellato, cerca nella lista a partire dal punto dove era rimasto l’ultima volta, non riparte dal principio.

- *best fit*: cerca all’interno della lista, dall’inizio alla fine, prendendosi lo spazio più piccolo ma che sia comunque sufficiente;

- *worst fit*: come il precedente però si prende lo spazio disponibile più grande;

- *quick fit*: mantiene liste divise per alcune delle più comuni dimensioni richieste;

Per tutti questi algoritmi si assume che il gestore della memoria conosca quanta memoria occorre allocare.

**Memoria virtuale**

Mentre da un lato i registri base e limite possono essere utilizzati per creare l’astrazione dello spazio di indirizzamento, dall’altro sorge un nuovo problema: la gestione del software bloat (bloatware).

La dimensione del software aumenta molto più velocemente della dimensione della memoria.

Sorge la necessità di eseguire programmi che eccedono in dimensioni rispetto alla memoria, oltre al conseguente bisogno di disporre di sistemi in grado di supportare molteplici programmi eseguiti contemporaneamente, che stiano in memoria, ma che complessivamente siano più grandi della stessa.

Lo swapping non è un’opzione interessante, poiché i dispositivi di I/O sono troppo lenti rispetto alla velocità della memoria.

Il problema dei programmi più grandi della memoria è presente fin dalle origini dell’informatica. Negli anni ’60 fu adottata una soluzione che divideva i programmi in pezzi piccoli, chiamati overlay. All’avvio di un programma, tutto ciò che veniva caricato nella memoria era il gestore degli overlay, che caricava e avviava subito l’overlay 0. Al termine, veniva indicato al gestore degli overlay di caricare l’overlay 1 sopra l’overlay 0 in memoria (se c’era spazio per farlo) oppure sovrascrivendo l’overlay 0 (in mancanza di spazio). Gli overlay erano tenuti sul disco e portati dentro e fuori la memoria dal gestore degli overlay.

Il lavoro di swapping era fatto dal sistema operativo, ma la suddivisione del programma in pezzi era fatto manualmente dal programmatore. Suddividere grandi programmi in piccoli pezzi modulari richiedeva molto tempo, era noioso e suscettibile di errori. Pochi programmatori lo facevano bene.

Per risolvere, si è cercato di delegare questo compito al computer. Il metodo che fu escogitato è noto come memoria virtuale. L’idea alla base della memoria virtuale è che ogni programma ha il proprio spazio degli indirizzi personale, suddiviso in pezzi chiamati pagine. Ogni pagina è un intervallo di indirizzi contigui. Queste pagine sono mappate sulla memoria fisica, ma non tutte le pagine devono stare nella memoria fisica per eseguire il programma.

Quando il programma fa riferimento a una parte del proprio spazio degli indirizzi che è nella memoria fisica, l’hardware esegue il necessario mappaggio diretto; quando fa riferimento a una parte del proprio spazio degli indirizzi non presente in memoria fisica, il sistema operativo è allertato di andare a prendere il pezzo mancante e rieseguire l’istruzione fallita.

La memoria virtuale funziona bene in sistemi con multiprogrammazione, con bit e pezzi di molti programmi contemporaneamente in memoria. Mentre un programma è in attesa che un suo pezzo sia letto, la CPU può essere assegnata a un altro processo.

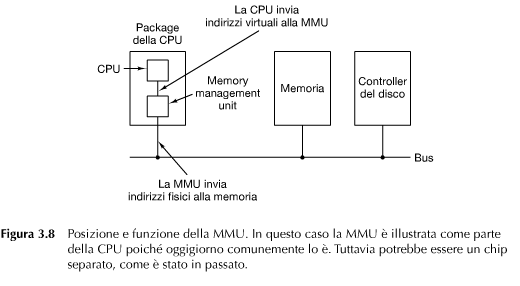
**Paginazione**

La maggior parte dei sistemi di memoria virtuale usa una tecnica chiamata paginazione o paging. Su qualsiasi computer i programmi referenziano un insieme di indirizzi di memoria.

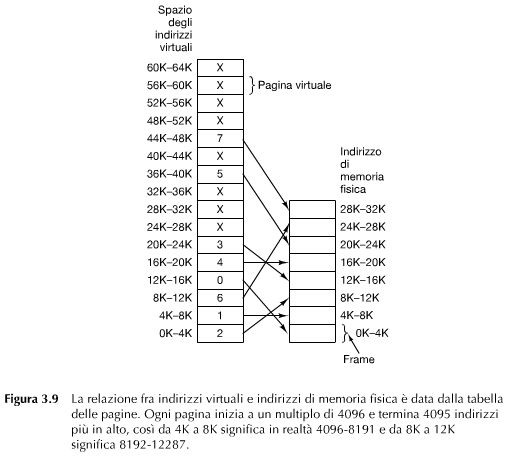
Gli indirizzi possono essere generati usando l’indicizzazione, i registri base, i registri segmento o in altri modi.

Questi indirizzi generati dal programma sono chiamati indirizzi virtuali e formano lo spazio virtuale degli indirizzi.

Quando è usata la memoria virtuale gli indirizzi virtuali vanno direttamente a una MMU (memory management unit) che mappa gli indirizzi virtuali sugli indirizzi di memoria fisica, come illustrato nella Figura 3.8.



Un esempio del funzionamento di questo mappaggio è illustrato nella Figura 3.9. In questo esempio abbiamo un computer che genera indirizzi di 16 bit, da 0 a 64K-1. Questi sono gli indirizzi virtuali. Tale computer, però, ha solo 32 KB di memoria fisica. Così, sebbene possano essere scritti programmi di 64 KB, questi non possono essere caricati nella loro interezza nella memoria ed essere eseguiti. Una copia completa dell’immagine del nucleo del programma, fino a 64 KB, deve comunque essere presente sul disco, in modo che possano esserne caricati dei pezzi secondo necessità.



Lo spazio degli indirizzi virtuali è suddiviso in unità di dimensione fissa, chiamate pagine. Le unità corrispondenti nella memoria fisica sono chiamate frame o page frame. Le pagine e i frame sono generalmente della stessa dimensione. I trasferimenti tra la RAM e il disco sono sempre di pagine intere.

La notazione della Figura 3.9 è la seguente. L’intervallo segnato 0K-4K significa che gli indirizzi fisici o virtuali in quella pagina vanno da 0 a 4095. L’intervallo 4K-8K si riferisce agli indirizzi da 4096 al 8191 e così via.

Nell’hardware effettivo, un bit presente/assente tiene traccia di quali pagine sono presenti fisicamente in memoria.

Quando una pagina virtuale non è nella memoria, la MMU si accorge che il riferimento nell’istruzione non è in memoria e causa un trap della CPU verso il sistema operativo. Questo trap è chiamato errore di pagina (page fault).

Il sistema operativo preleva un frame poco utilizzato e ne scrive il suo contenuto sul disco (se non è già presente). Prende poi la pagina appena referenziata e la mette nel frame appena liberato, cambia la mappa e riavvia l’istruzione che era in trap.

All’interno della MMU c’è una tabella delle pagine che permette di traslare un indirizzo virtuale in uno fisico.

Il numero di pagina è usato come indice della tabella che porta al numero di frame corrispondente alla pagina virtuale. Se il bit presente/assente è 0, avviene un trap al sistema operativo; se il bit è 1, il numero di frame trovato nella tabella delle pagine viene copiato nei tre bit più significativi del registro di output, insieme all’offset di 12 bit che è copiato senza modifiche dall’indirizzo virtuale in arrivo. Insieme formano un indirizzo fisico di 15 bit. Il registro di output è poi messo sul bus di memoria come indirizzo fisico di memoria.

**Tabelle delle pagine**

L’indirizzo virtuale è diviso in numero di pagina virtuale (i bit più significativi) e un offset (i bit meno significativi).

Il numero di pagina virtuale è utilizzato come un indice nella tabella delle pagine per trovare la voce per quella pagina virtuale. Il numero di frame (se esiste) è rintracciato dalla voce della tabella delle pagine. Il numero di frame è allegato alla parte finale più significativa dell’offset, rimpiazzando il numero di pagina virtuale, per formare un indirizzo fisico che può essere inviato alla memoria. In questo modo lo scopo della tabella delle pagine è di mappare le pagine virtuali sui frame delle pagine.

**Struttura di una voce della tabella delle pagine**

Anche se la struttura fisica del record dipende dalla macchina, il tipo di informazioni presenti è generalmente lo stesso.

Nella Figura 3.11 abbiamo un campione di una voce della tabella delle pagine.

La lunghezza della voce usuale è di 32 bit e contiene:

- *numero del frame*;

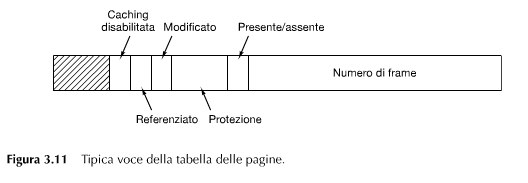
- *bit presente/assente*: se questo bit è 1, la voce è valida e può essere utilizzata; se è 0, la pagina virtuale cui appartiene la voce non è effettivamente in memoria;

- *bit di protezione*: specificano quali tipi di accesso sono consentiti (lettura, scrittura, esecuzione);

- *bit modificato*: tiene traccia della modifica della pagina; se la pagine è stata modificata deve essere riscritta sul disco, in caso contrario quella sul disco è ancora valida;

- *bit referenziato:* tiene traccia dell’uso della pagina; serve per aiutare il sistema operativo a scegliere quale pagina scaricare quando si verifica un page fault; le pagine inutilizzate sono le migliori candidate;

- *bit caching disabilitata*: consente di disabilitare la cache per la pagina.



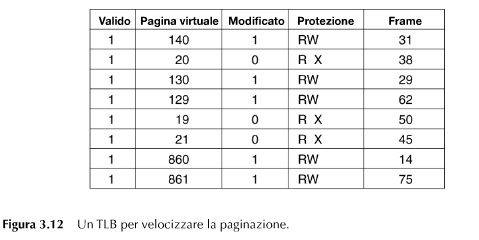
**Translation lookaside buffer**

Esaminiamo alcuni schemi largamente implementati per velocizzare la paginazione e per

gestire grandi spazi degli indirizzi virtuali, partendo dalla prima questione. Il punto di partenza della maggior parte delle tecniche di ottimizzazione è che la tabella delle pagine sta nella memoria.

Osservando che la maggior parte dei programmi rende a fare un gran numero di riferimenti a un piccolo numero di pagine e non il contrario, solo una piccola parte delle voci della tabella delle pagine viene letta frequentemente; il resto è poco utilizzato.

La soluzione escogitata è equipaggiare i computer di un piccolo dispositivo hardware per mappare gli indirizzi virtuali sugli indirizzi fisici senza passare dalla tabella delle pagine.  
Il dispositivo, chiamato TLB (translation lookaside buffer) o qualche volta anche memoria associativa, è illustrato nella Figura 3.12.



Si trova abitualmente all’interno della MMU e consiste di un numero ridotto di voci, otto in questo caso, ma raramente più di 256. Ciascuna voce contiene informazioni riguardo una pagina, tra cui il numero di pagina virtuale, un bit impostato quando la pagina viene modificata, il codice dei permessi e il frame fisico in cui si trova la pagina. Questi campi hanno una corrispondenza uno-a-uno con i campi nella tabella delle pagine, eccetto che per il numero di pagina virtuale, che non è necessario nella tabella delle pagine. Un altro bit indica se la voce è valida (cioè in uso) o no.

Analizziamo adesso come funziona il TLB. Quando un indirizzo virtuale viene presentato alla MMU per la traduzione, l’hardware prima guarda se il suo numero di pagina virtuale è presente nel TLB confrontandolo simultaneamente (cioè in parallelo) con tutte le voci.  
Se trova un riscontro valido il frame è preso direttamente dal TLB, senza andare alla tabella delle pagine; altrimenti la MMU rileva la TLB miss (fallimento di TLB) e fa una normale ricerca

sulla tabella delle pagine. Quindi sfratta una delle voci dal TLB e la rimpiazza con la voce della tabella delle pagine appena cercata. In questo modo se quella pagina è riusata a breve, la seconda volta si avrà una TLB hit (successo di TLB) invece che una TLB miss.

Quando una voce è eliminata dal TLB, il bit modificato è copiato di nuovo nella voce della tabella delle pagine nella memoria.

**Gestione software del TLB**

La tabelle delle pagine e la TLB possono essere gestiti dall’hardware (le uniche trap al SO avvengono quando una pagina non è in memoria) oppure dal software.

Molte macchine moderne eseguono quasi tutta questa gestione delle pagine tramite software. Su queste macchine, le voci del TLB sono caricate esplicitamente dal sistema operativo. Quando si verifica una TLB miss, invece di avere la MMU che va alla tabella delle pagine per cercare e prelevare il necessario riferimento alla pagina, viene generato soltanto un errore di TLB e il problema passa al sistema operativo. Il sistema deve trovare la pagina, rimuovere una voce dal TLB, inserirne una nuova e riavviare l’istruzione in errore.

Il modo consueto di trattare una TLB miss, sia hardware sia software, è andare alla tabella delle pagine e compiere le operazioni di indicizzazione per localizzare la pagina referenziata.  
Il problema che deriva da questa ricerca nel software è che le pagine contenenti la tabella delle  
pagine potrebbero non essere nel TLB, il che causerebbe ulteriori errori TLB durante il processo. Questi errori sono riducibili se le pagine che contengono la tabella delle pagine siano sempre mantenute nel TLB. Un modo è riservare uno spazio di posizioni fisse nella TLB in modo che queste righe non possano essere rimosse quando accade un TLB miss.

Con la gestione TLB via software è fondamentale capire la differenza fra due tipi di miss.

Una soft miss avviene quando la pagina di riferimento non è nel TLB ma è nella memoria. Tutto ciò di cui si ha bisogno in questo caso è che il TLB sia aggiornato. Nessun accesso al disco è necessario, e quindi può essere completata in pochi nanosecondi.

Una hard miss avviene quando la pagina stessa non è in memoria (e ovviamente nemmeno nel TLB). Per prendere la pagina serve un accesso al disco, il che comporta parecchi millisecondi. Una hard miss è milioni di volte più lenta di una soft miss.

**Tabelle delle pagine per grandi memorie**

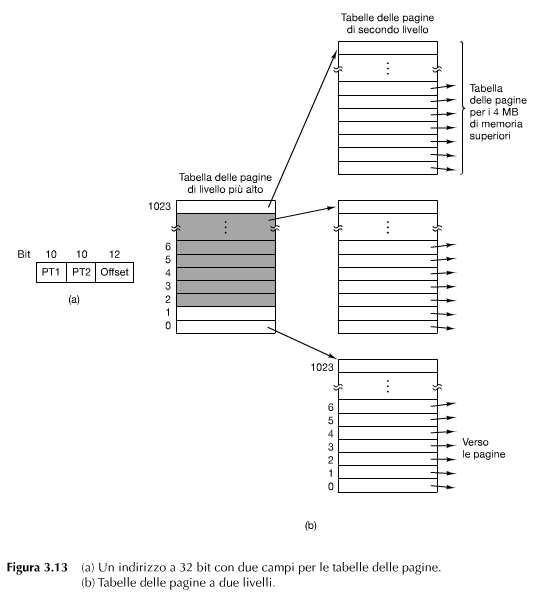
I TLB sono usati per velocizzare la traduzione dell’indirizzo virtuale nell’indirizzo fisico.

L’altro problema da affrontare è come gestire spazi degli indirizzi virtuali molto grandi.

Esistono due approcci.

**Tabelle delle pagine multilivello**

Consideriamo innanzitutto le tabelle delle pagine multilivello. Un semplice esempio è illustrato nella Figura 3.13.



Nella Figura 3.13(a) abbiamo un indirizzo virtuale a 32 bit partizionato in un campo *PT1* a 10 bit, un campo *PT2* a 10 bit e un campo *Offset* a 12 bit. Poiché gli offset sono 12 bit, le pagine sono 4 KB e ve n’è un totale di 220.

Il segreto del metodo della tabella delle pagine multilivello sta nell’evitare di mantenere tutte le tabelle delle pagine in memoria per tutto il tempo. In particolare quelle non necessarie dovrebbero essere messe da parte.

Nella Figura 3.13(b) vediamo come lavora in questo esempio la tabella delle pagine a due livelli. Sulla sinistra abbiamo la tabella delle pagine di livello più alto, con 1024 voci, corrispondenti ai 10 bit del campo *PT1*. Quando viene presentato un indirizzo virtuale alla MMU, prima estrae il campo *PT1* e usa questo valore come indice nella tabella delle pagine di livello più alto.   
La voce localizzata tramite l’indicizzazione nella tabella delle pagine di livello superiore produce l’indirizzo o il numero di frame di una tabella delle pagine di secondo livello. Il campo *PT2* adesso è usato come un indice nella tabella delle pagine di secondo livello selezionata per cercare il numero di frame della pagina stessa.

Se quella pagina non è in memoria, il bit Presente/Assente nella voce della tabella delle pagine sarà zero, provocando un page fault.

Se la pagina è nella memoria, il numero di frame preso dalla tabella delle pagine di secondo livello è combinato con l’offset per costruire l’indirizzo fisico. Questo indirizzo è messo nel bus e inviato alla memoria.

La cosa interessante da notare è che, sebbene lo spazio degli indirizzi contenga più di un milione di pagine, sono effettivamente necessarie solo quattro tabelle delle pagine: la tabella di livello più alto e le tabelle di secondo livello da O a 4 MB (per il testo del programma), da 4 MB a 8 MB (per i dati) e i 4 MB in alto (per lo stack).

I bit Presente/Assente in 1021 voci della tabella delle pagine di livello più alto sono impostati a 0, forzando un page fault nel caso qualcuno provasse mai ad accedervi. Se accadesse, il sistema operativo noterebbe che il processo starebbe provando a far riferimento a memoria cui non dovrebbe e intraprenderebbe un’azione adeguata, come mandargli un segnale o terminarlo.

Il sistema della tabella delle pagine a due livelli della Figura 3.13 può essere esteso a tre, quattro o più livelli. I livelli aggiuntivi danno maggior flessibilità.

**Tabelle delle pagine invertite NO**

Le tabelle delle pagine multilivello lavorano bene fino a 32 bit di spazio di indirizzi virtuali. Con un computer da 64 bit la tabella delle pagine diventerebbe enorme (più di 30 milioni).

Poiché lo spazio di indirizzamento è 264 byte, se si utilizzano pagine da 4 KB, la tabella delle pagine ha 252 righe. Se ogni riga occupa 8 byte, la tabella è di oltre 30 PB.

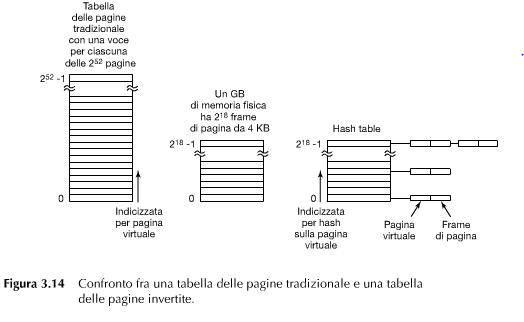
Un’alternativa al continuo aumento della gerarchia della paginazione è nota con il termine tabelle delle pagine invertite. In questa progettazione c’è una sola voce per frame nella memoria reale, piuttosto che una voce per pagina dello spazio virtuale degli indirizzi.

Per esempio, con indirizzi virtuali con 64 bit, una pagina da 4 KB e 1 GB di RAM, una tabella delle pagine invertire richiede solo 262.144 voci. La voce tiene traccia di ciò che è memorizzato nel frame (processo, pagina virtuale).

Sebbene le tabelle delle pagine invertire risparmino una gran quantità di spazio, quando lo spazio virtuale degli indirizzi è molto superiore rispetto alla memoria fisica, esse presentano

un grosso svantaggio: la traduzione da virtuale a fisica diventa molto difficile. Quando il  
processo *n* referenzia la pagina virtuale *p*, l’hardware non può più trovare la pagina fisica usando *p* come indice nella tabella delle pagine. Deve cercare invece la voce (*n*, *p*) nell’intera tabella delle pagine invertire. Per di più questa ricerca deve essere eseguita per ogni riferimento alla memoria e non solo in caso di page fault.

La soluzione è data dall’impiego del TLB. Se il TLB può contenere tutte le pagine più usate, la traduzione può avvenire velocemente come con le normali tabelle delle pagine. Nel caso di una TLB miss deve essere comunque fatta una ricerca sulla tabella delle pagine invertite. Un modo possibile di realizzare questa ricerca è di avere una hash table sull’indirizzo virtuale.



**Algoritmi di sostituzione delle pagine**

Quando si verifica un page fault, per far spazio alla pagina entrante il sistema operativo deve

scegliere una pagina da rimuovere dalla memoria.

Se la pagina da rimuovere è stata modificata mentre era in memoria, deve essere riscritta sul disco, per mantenere la copia aggiornata. Se tuttavia la pagina non è stata cambiata la copia su disco è già aggiornata e non c’è bisogno di riscrittura.

Nonostante sarebbe possibile prendere una pagina a caso da rimuovere a ogni page fault, le prestazioni del sistema sono migliori se si sceglie una pagina non particolarmente utilizzata. Se si prendesse da rimuovere una pagina usata frequentemente, con ogni probabilità dovrebbe essere richiamata a breve, con il risultato di un ulteriore sovraccarico.

Vale la pena sottolineare che il problema della “sostituzione delle pagine” si verifica anche quando occorre sostituire i dati nella cache, quindi i procedimenti che verranno mostrati hanno una valenza generale.

**L’algoritmo ottimo di sostituzione delle pagine**

Il miglior algoritmo di sostituzione delle pagine è semplice da descrivere ma impossibile da implementare.

Funziona così. Al momento del page fault, nella memoria si trova un insieme di pagine. Una di loro sta per essere referenziata in un’istruzione molto vicina. Le altre pagine non saranno referenziate prima di 10, 100 0 forse 1000 istruzioni.

Se ciascuna pagina fosse etichettata con il numero di istruzioni da eseguire prima che quella pagina sia referenziata per la prima volta, l’algoritmo di sostituzione ottimale indica che deve essere rimossa la pagina con il più alto numero di etichetta. Se una pagina non sarà usata per 8 milioni di istruzioni e un’altra pagina per 6 milioni di istruzioni, rimuovere la prima allontana il page fault che ricaricherebbe la pagina il più in là possibile.

Il solo problema di questo algoritmo è che è irrealizzabile. Al momento del page fault, il sistema operativo non ha modo di sapere quando ciascuna delle pagine sarà referenziata la volta successiva.

**Not recently used (NRU)**

Al fine di consentire al sistema operativo la raccolta di statistiche utili sull’uso delle pagine, molti computer con memoria virtuale hanno due bit di stato, R e M, associati a ciascuna pagina.

R viene impostato quando si fa riferimento alla pagina (in lettura o scrittura). M è impostato

quando la pagina è modificata. I bit sono contenuti in ciascuna voce della tabella delle pagine e devono essere aggiornati a ogni riferimento alla memoria.

All’avvio di un processo, entrambi i bit di pagina per tutte le pagine sono impostati a 0 dal sistema operativo. Ad ogni impulso del clock, il bit R è ripulito, per contraddistinguere le pagine che non sono state recentemente referenziate da quelle che lo sono state. Quando avviene un page fault, il sistema operativo ispeziona tutte le pagine e le divide in 4 categorie basate sui valori attuali dei loro bit R e M:

- classe 0: non referenziato, non modificato;

- classe 1 : non referenziato, modificato;

- classe 2: referenziato, non modificato;

- classe 3: referenziato, modificato.

Sebbene le pagine di classe 1 sembrino, a prima vista, impossibili, si verificano quando un interrupt del clock azzera il bit R di una pagina di classe 3. Gli interrupt del clock non azzerano il bit M perché questa informazione è necessaria per conoscere se la pagina deve essere riscritta su disco o meno. Azzerare R ma non M produce una pagina di classe 1.

L’algoritmo NRU (not recently used) rimuove una pagina a caso dalla classe non vuota con il numero più basso. In questo algoritmo è chiara l’idea che è meglio rimuovere una pagina modificata che non è stata referenziata nemmeno una volta nell’ultimo intervallo del clock piuttosto che una pagina pulita usata frequentemente.

NRU è facilmente comprensibile, discretamente efficiente da implementare e fornisce prestazioni che, per quanto non ottimali, possono risultare adeguate.

**First-in, first-out (FIFO)**

Il sistema operativo tiene una lista delle pagine attualmente in memoria, con l’arrivo più recente in coda e il più vecchio in testa.

A un page fault, la pagina di testa è rimossa e la nuova aggiunta in coda all’elenco.

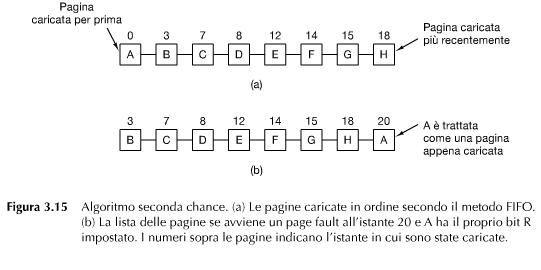
Poiché la gestione FIFO non tiene conto dell’utilizzo della pagina ma solo dell’ordine di caricamento in memoria è raramente utilizzata.

**Seconda chance**

Una semplice modifica a FIFO evita il problema di gettare una pagina usata di frequente controllando il bit R della pagina più vecchia. Se è 0, la pagina è vecchia e inutilizzata e viene così sostituita immediatamente. Se R è 1, il bit viene azzerato, la pagina è messa in fondo all’elenco e il suo tempo di caricamento aggiornato come se fosse appena arrivata in memoria.

Poi la ricerca continua.

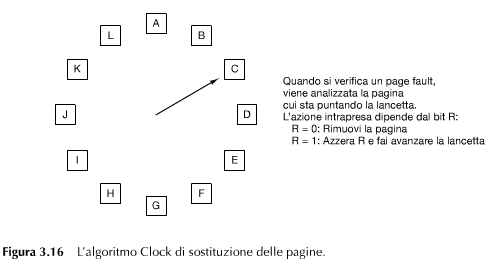
L’operazione di questo algoritmo è illustrata nella Figura 3.15(a) dove vediamo le pagine da A a H mantenute in una lista collegata e ordinate in base al tempo in cui sono arrivare nella memoria. Supponiamo che si verifichi un page fault all’istante 20. La pagina più vecchia è A, arrivata al momento 0, all’inizio del processo. Se A ha il bit R azzerato, allora è rimossa dalla memoria. D’altro canto, se il bit R è impostato, A è messa in fondo alla lista e il suo “momento di caricamento” è reimpostato all’attuale (20).



Sebbene questo algoritmo sia ragionevole, è inefficiente poiché fa scorrere di continuo le pagine lungo la sua lista.

**Clock**

Un approccio migliore è di tenere tutti i frame su una lista circolare a forma di orologio, come mostrato nella Figura 3.16. La lancetta indica la pagina più vecchia. Quando avviene un page fault, la pagina indicata dalla lancetta viene controllata. Se il suo bit R è 0, allora viene sfrattata, la nuova pagina inserita al suo posto nell’orologio e la lancetta spostata in avanti di una posizione. Se R è 1, viene azzerato e la lancetta avanzata alla pagina successiva. Questo processo è ripetuto finché non viene trovata una pagina con R = 0.



**Least recently used (LRU)**

Una buona approssimazione dell’algoritmo ottimale si basa sull’osservazione che le pagine usate più frequentemente nelle ultime istruzioni lo saranno anche nelle successive.

Al contrario quelle inutilizzate da molto tempo lo resteranno ancora per molto.

Questa strategia si chiama paginazione LRU (least recently used). Sebbene questo algoritmo sia teoricamente fattibile, non è economico. Per implementare pienamente l’LRU è necessario mantenere una lista collegata di tutte le pagine in memoria, con davanti quelle più usate e dietro quelle meno usate. La difficoltà sta nel fatto che l’elenco deve essere aggiornato a ogni riferimento alla memoria. Trovare una pagina in memoria, cancellarla e poi portarla davanti è un’operazione che costa del tempo. Tuttavia sono possibili due realizzazioni hardware: contatore a 64 bit o matrice binaria.

**LRU con matrice binaria**

Per un macchina con *n* pagine fisiche (frame), l’hardware LRU può mantenere una matrice di

*n x n* bit, inizialmente tutti a zero.

Ogni volta che la pagina *k* viene referenziata, l’hardware prima imposta tutti i bit della riga *k* a 1, poi imposta tutti i bit della colonna *k* a 0.

La riga il cui valore binario è più basso indica il frame meno recentemente utilizzato.

**Not Frequently Used (NFU)**

Sebbene entrambi i precedenti algoritmi siano (in linea di principio) realizzabili, sono poche, per non dire nessuna, le macchine con l’hardware necessario.

Occorre invece una soluzione che si possa implementare via software.

Una possibilità è l’algoritmo NFU (not frequently used). Richiede un contatore software associato a ogni pagina e inizialmente impostato a zero. Il sistema operativo fa la scansione di tutte le pagine in memoria, a ogni interrupt del clock. Per ciascuna pagina viene sommato il bit R (che è 0 o 1) al contatore. I contatori tengono traccia di quante volte ciascuna pagina è stata referenziata.

Quando avviene un page fault la scelta di quale pagina sostituire cade su quella con il contatore più basso.

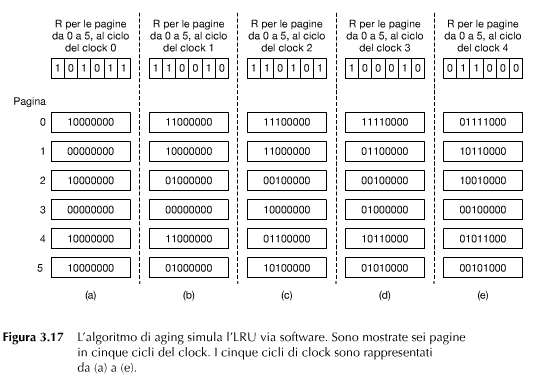
Il problema principale dell’NFU è che non si dimentica mai nulla. Per esempio, nel caso di un compilatore a molteplici passaggi, le pagine usate frequentemente durante il passaggio 1 avrebbero ancora un contatore alto pure nei passaggi a seguire. Infatti se accade che il passaggio 1 è quello con il tempo di esecuzione più alto fra tutti i passaggi, le pagine contenenti il codice per i passaggi seguenti potrebbero avere sempre conteggi inferiori alle pagine del passaggio 1.

Di conseguenza il sistema operativo rimuoverebbe pagine utili invece di pagine non più utilizzate.

**Aging**

Per fortuna una piccola modifica all’NFU lo rende in grado di simulare abbastanza bene l’algoritmo LRU. La modifica consta di due parti. Innanzitutto, i contatori sono tutti spostati a destra di 1 bit prima che sia aggiunto il bit R. In secondo luogo, il bit R è aggiunto al bit più a sinistra piuttosto che a quello più a destra.

La Figura 3.17 illustra come lavora questo algoritmo modificato, conosciuto come aging.



Supponiamo che dopo il primo ciclo di clock i bit R delle pagine da 0 a 5 abbiano rispettivamente i valori 1, 0, 1, 0, 1 e 1 (la pagina 0 è 1, la pagina 1 è 0, la pagina 2 è 1, e così via). In altre parole, fra il ciclo 0 e il ciclo 1 del clock sono state referenziate le pagine 0, 2, 4 e 5, impostando i loro bit R a 1, mentre gli altri restano a 0. Dopo che i sei corrispondenti contatori sono stati fatti scalare e il bit R è stato inserito alla loro sinistra, i loro valori sono quelli illustrati nella Figura 3.17(a). Le quattro colonne restanti mostrano i sei contatori dopo i successivi quattro cicli del clock.

Quando avviene un page fault, viene rimossa quella con il contatore più basso.

Questo algoritmo si differenzia dall’LRU in due modi:

- utilizzando un solo bit per intervallo di clock sarà possibile registrare al massimo un riferimento per pagina anche se ce ne fossero di più nel medesimo intervallo di tempo;

- nell’algoritmo di aging i contatori sono codificati con un numero finito di bit, questo limita l’orizzonte di analisi a quel numero di cicli di clock.

**Working set**

I processi, nella forma più pura di paginazione, sono avviati senza nessuna delle loro pagine in memoria. Appena la CPU prova a prelevare la prima istruzione, genera un page fault, facendo sì che il sistema operativo prelevi la pagina con la prima istruzione. Rapidamente seguono altri page fault per le variabili globali e lo stack. Dopo un certo tempo, il processo ha la maggior parte delle pagine di cui ha bisogno ed è sistemato per essere eseguito con relativamente pochi page fault. Questa strategia è detta demand paging poiché le pagine sono caricate a richiesta (on demand) e non in anticipo.

Nella maggior parte dei casi vale un principio di località di riferimento, che significa che durante qualunque fase dell’esecuzione il processo referenzia solo una frazione relativamente piccola delle sue pagine: il working set. Se l’intero working set è in memoria, il processo sarà eseguito senza causare molti page fault, finché si sposterà in un’altra fase dell’esecuzione.

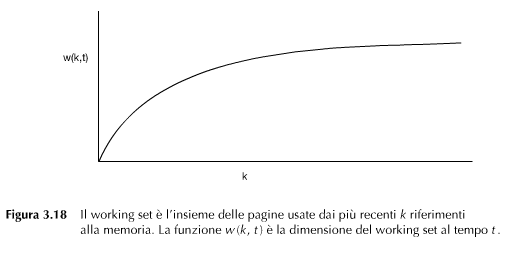
Se la memoria disponibile è troppo piccola per contenere l’intero working set, il processo provocherà molti page fault e sarà lento, poiché eseguire un’istruzione richiede pochi nanosecondi, mentre leggere la pagina dal disco richiede 10 millisecondi.

Un programma che causa page fault ogni poche istruzioni è detto essere thrashing.

Molti sistemi di paging cercano di assicurarsi che il working set del processo sia in memoria prima di eseguirlo (Working Set Model).

Caricare le pagine prima di lasciar eseguire i processi è anche detto prepaginazione o prepaging. Notate che il working set cambia nel tempo. È risaputo da tempo che molti programmi non referenziano il loro spazio degli indirizzi uniformemente, ma che i riferimenti tendono a raggrupparsi su un numero ristretto di pagine.

A ogni istante di tempo *t* esiste un insieme che consiste di tutte le pagine usare dai *k* riferimenti alla memoria più recenti. Questo insieme *w(k, t)* è il working set. La Figura 3.18 rappresenta la dimensione del working set come una funzione di *k*.



*w(k, t)* è una funzione monotona non decrescente di *k*. Il limite di *w(k, t)* quando *k* diventa grande è finito, poiché un programma non può referenziare più pagine di quante ne contiene il suo spazio degli indirizzi e pochi programmi utilizzeranno ogni singola pagina.

Il fatto che la maggior parte dei programmi acceda casualmente a un piccolo numero di pagine, ma che quell’insieme cambi lentamente nel tempo, spiega la crescita rapida iniziale della curva e poi il lento crescere con *k* grande.

A causa del suo andamento asintotico, i contenuti del working set non sono sensibili al valore di *k* scelto. In altre parole esiste un ampio insieme di valori *k* per cui il working set non varia.

La prepaginazione consiste nel caricamento di queste pagine prima che il processo sia riavviato.

Per implementare il modello a working set serve che il sistema operativo tenga traccia di quali pagine sono nel working set. Avere questa informazione porta inoltre immediatamente a un possibile algoritmo di sostituzione delle pagine: quando accade un page fault, cerca una pagina al di fuori del working set e la rimuove.

Naturalmente, avere una definizione operativa di ciò che è il working set non significa che vi sia un metodo efficiente per calcolarlo durante l’esecuzione dei programmi. Potremmo immaginare un registro a scorrimento di lunghezza *k*, con ogni riferimento alla memoria che scorre il registro di una posizione a sinistra e inserisce il numero dell’ultima pagina referenziata sulla destra. L’insieme di tutti i *k* numeri di pagine referenziate è il working set. In teoria, a ogni page fault il registro a scorrimento dovrebbe essere letto e riordinato. I duplicati di pagina dovrebbero essere rimossi. Il risultato sarebbe il working set. Tuttavia, mantenere ed elaborare il registro a scorrimento a ogni page fault potrebbero essere due attività proibitive in termini di costo; di conseguenza questa tecnica non è mai usata.

Sono usate invece varie sue approssimazioni. Una tra le più comuni è quella di eliminare l’idea di contare a ritroso *k* riferimenti alla memoria ma di considerare l’insieme delle pagine utilizzate negli ultimi T ms del tempo di esecuzione. Con questa definizione è più facile lavorare.

Notate che per ogni processo conta solo il suo tempo di esecuzione, chiamato il suo tempo virtuale attuale (current virtual time). Con questa approssimazione il working set di un processo è l’insieme delle pagine referenziate negli ultimi T secondi del suo tempo virtuale.

Andiamo adesso ad analizzare un algoritmo di sostituzione delle pagine che si basa sul working set. L’idea base è quella di trovare una pagina che non sia nel working set e rimuoverla.

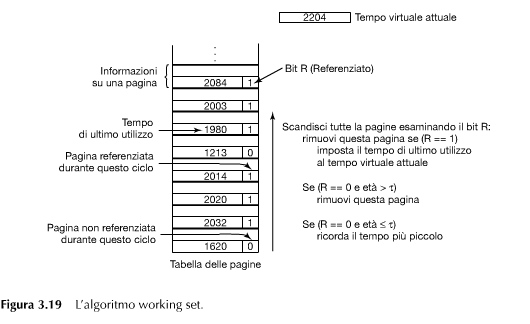
Poiché solo le pagine in memoria sono da considerare candidate alla rimozione, le pagine assenti dalla memoria sono ignorate da questo algoritmo. Ogni voce della tabella delle pagine contiene: il tempo (h) dell’ultima volta che la pagina è stata usata, il bit R se è stata referenziata nell’ultimo ciclo di clock e il bit M se la pagina è stata modificata.

A ogni page fault viene fatta la scansione della tabella delle pagine per cercare una pagina da rimuovere. Quando viene elaborata ciascuna voce, viene esaminato il bit R. Se è 1, il tempo virtuale attuale è scritto nel campo time of last use (tempo di ultimo utilizzo) nella tabella delle pagine, a indicare che la pagina era in uso al momento del page fault. Poiché la pagina è stata referenziata durante l’ultimo ciclo del clock, è chiaramente nel working set e non è una candidata alla rimozione.  
Se R è 0, la pagina non è stata referenziata durante il ciclo del clock attuale e può essere candidata alla rimozione. Per vedere se rimuoverla o meno, è calcolata la sua età (che è il   
tempo virtuale attuale meno il suo tempo di ultimo utilizzo) e confrontata con T. Se l’età è   
maggiore di T, la pagina non è più nel working set e la nuova pagina la sostituisce. La scansione riprende aggiornando le voci restanti.

Tuttavia, se R è 0, ma l’età è minore o uguale a T, la pagina rimane nel working set. La pagina al momento è risparmiata, ma la pagina con l’età maggiore (il valore inferiore di tempo di ultimo utilizzo) viene contrassegnata. Se la scansione dell’intera tabella non trova alcuna pagina da rimuovere significa che tutte appartengono al working set. In questo caso, se vengono trovate una o più pagine con R = 0, viene rimossa quella contrassegnata con l’età maggiore.

Nel peggiore dei casi, tutte le pagine sono state referenziate nell’ultimo ciclo del clock

(da cui R = 1), per cui ne viene presa una a caso per la rimozione, preferibilmente, se esiste una pagina pulita.



**WSClock**

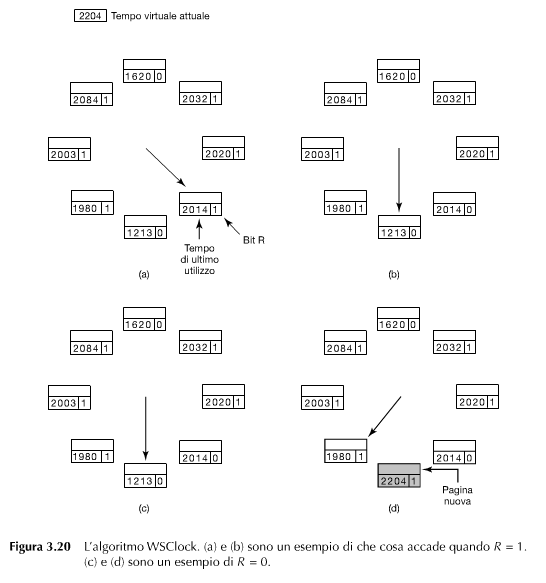
Poiché l’intera tabella delle pagine deve essere scandita a ogni page fault al fine di trovare una pagina adatta alla rimozione, l’algoritmo base del working set è piuttosto lento.

Un algoritmo migliore, basato sull’algoritmo Clock, ma che usa anche le informazioni del working set, è chiamato WSClock. Grazie alla sua semplicità di implementazione e alle buone prestazioni, nella pratica è diffusamente usato.

La struttura dati necessaria è una lista circolare di frame, come nell’algoritmo Clock e come illustrato nella Figura 3.20(a). Inizialmente questo elenco è vuoto. Quando è caricata la prima pagina, essa viene aggiunta all’elenco.

Ogni voce contiene il campo tempo di ultimo utilizzo (time of last use) dall’algoritmo base del working set, così come il bit R (mostrato) e il bit M (non mostrato). Come nell’algoritmo clock, a ogni page fault quella indicata dalla lancetta dell’orologio è esaminata per prima. Se il bit R è 1, la pagina è stata usata nel ciclo del clock, quindi non è la candidata ideale alla rimozione. Il bit

R è impostato a 0, la lancetta avanza alla pagina successiva e l’algoritmo rieseguito per la nuova pagina. La situazione dopo questa sequenza è mostrata nella Figura 3.20(b).



Consideriamo adesso che cosa accade se la pagina indicata ha R = 0, come mostrato nella Figura 3.20(c).

Se l’età è maggiore di T e la pagina è pulita, non è nel working set e ne esiste una copia valida su disco. Il frame è semplicemente reclamato e la nuova pagina messa in quella posizione, come nella Figura 3.20(d). Dall’altra parte, se la pagina è sporca, poiché non ne esiste una copia valida su disco non può essere richiesta immediatamente. Per evitare un cambio di processo viene schedulata la scrittura su disco, ma viene avanzata la lancetta e l’algoritmo procede con la pagina successiva.

Che cosa accade se la lancetta fa un giro dell’orologio e torna al punto di partenza? Ci sono da considerare due situazioni:  
1) è stata schedulata almeno una scrittura;  
2) non ci sono scritture schedulate.

Nel primo caso, la lancetta continua a scorrere, alla ricerca di una pagina pulita. La prima pagina pulita incontrata viene rimossa.

Nel secondo caso, tutte le pagine sono nel working set, altrimenti almeno una scrittura sarebbe stata schedulata. Scarseggiando di ulteriori informazioni, la cosa più semplice da fare è richiamare una pagina pulita qualunque e usarla. In mancanza di pagine pulite viene scelta come vittima la pagina attuale e scritta sul disco.

**Riepilogo degli algoritmi di sostituzione delle pagine**

La lista degli algoritmi è nella Figura 3.21.

L’algoritmo ottimale sfratta la pagina che sarà referenziata il più tardi possibile. Peccato che non vi sia modo per determinare quale essa sia, così questo algoritmo non può essere utilizzato.

L’algoritmo NRU suddivide le pagine in quattro classi che dipendono dallo stato dei bit R e M. Viene scelta una pagina a caso fra le pagine della classe con la numerazione inferiore. È un algoritmo semplice da implementare, ma molto grezzo. Ne esistono di migliori.

Il FIFO tiene traccia dell’ordine in cui sono caricate le pagine in memoria, tenendole in una lista collegata. Diventa banale il rimuovere la pagina più vecchia, ma quella pagina potrebbe essere ancora in uso, quindi il FIFO è una pessima scelta.

L’algoritmo seconda chance è una modifica del FIFO che controlla se la pagina è in uso prima di rimuoverla. Se lo è, la pagina è risparmiata. Questa modifica incrementa enormemente le prestazioni.

L’algoritmo clock è semplicemente una diversa implementazione del seconda chance. Ha le stesse proprietà prestazionali, ma impiega leggermente meno tempo a eseguire l’algoritmo.

L’LRU è un algoritmo eccellente, ma non può essere implementato senza un hardware speciale.

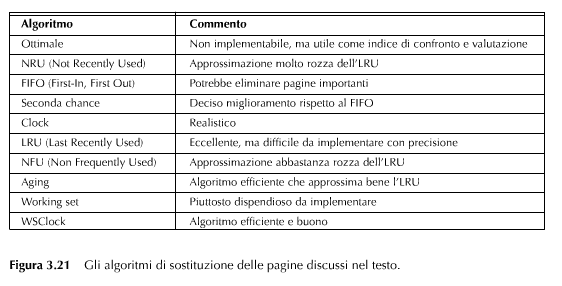
L’NFU è un tentativo grezzo di avvicinarsi all’LRU.

L’aging è un’approssimazione dell’LRU decisamente migliore e può essere efficacemente implementata.

L’algoritmo del working set fornisce prestazioni ragionevoli, ma è in un certo modo costoso da implementare.

Il WSClock è una variante che fornisce non solo buone prestazioni, ma anche efficienza nell’implementazione.

Tutto sommato, i due algoritmi migliori sono l’aging e il WSClock. Sono rispettivamente basati sull’LRU e sul working set. Danno entrambi buone prestazioni di paginazione e possono essere implementati efficientemente.



**Problemi di progettazione dei sistemi di paginazione**

I progettisti di sistemi operativi devono considerare altri aspetti al fine di ottenere buone prestazioni dai sistemi di paging:

- politiche di allocazione globale e locale;

- controllo del carico;

- dimensioni della pagina;

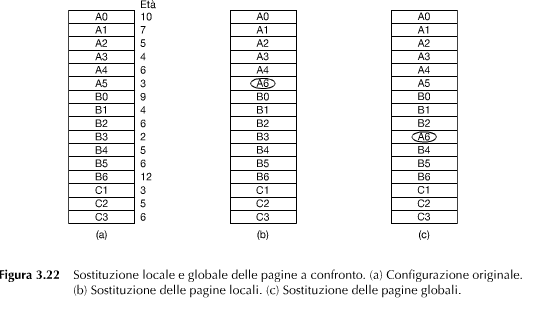
- dati e istruzioni su spazi separati;

- pagine condivise.

**Politiche di allocazione globali e locali a confronto**

Il problema principale associato a questa scelta è la quantità di memoria che dovrebbe essere ripartita tra processi eseguibili.

Diamo uno sguardo alla Figura 3.22(a).



In questa figura i processi A, B e C sono l’insieme dei processi eseguibili. Supponiamo che A dia un page fault. L’algoritmo di sostituzione delle pagine dovrebbe cercare di trovare la pagina usata meno recentemente considerando solo le sei pagine attualmente allocate ad A o dovrebbe considerare tutte le pagine in memoria?

Se guardasse solo alle pagine di A, la pagina con l’età inferiore sarebbe A5, così avremmo la situazione della Figura 3.22(b).

D’altro canto se la pagina da rimuovere fosse fra quelle con il valore minore senza guardare a quale appartenesse sarebbe scelta B3 e avremmo la situazione della Figura 3.22(c).

L’algoritmo della Figura 3.22(b) è detto “algoritmo di sostituzione delle pagine locali”,

mentre quello della Figura 3.22(c) è detto “algoritmo di sostituzione delle pagine globali”.

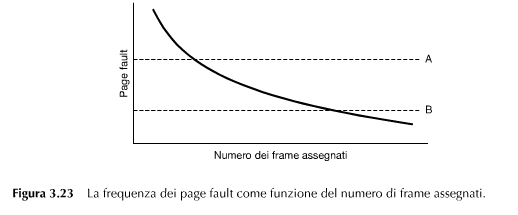
Gli algoritmi globali funzionano meglio quando la dimensione del working set può variare durante la vita di un processo.

Se è usato un algoritmo locale e il working set aumenta, ne risulta del thrashing, anche in presenza di frame liberi. Se il working set si restringe, gli algoritmi locali sprecano memoria.

Se è utilizzato un algoritmo globale, il sistema deve continuamente decidere quanti frame assegnare a ciascun processo.

Un modo è di allocare dinamicamente le pagine di un processo (partendo da un valore iniziale proporzionale alla sua dimensione) facendo in modo che la PFF (page fault frequency) sia contenuta in un certo intervallo.

La proprietà PFF è illustrata nella Figura 3.23.



Per misurare la frequenza dei page fault basta contare il numero di page fault per secondo.

La linea tratteggiata marcata A corrisponde a una frequenza di page fault troppo alta, quindi

per diminuire la frequenza dei page fault al processo in errore sono dati più frame. La linea

tratteggiata marcata B corrisponde a una percentuale di page fault così bassa da supporre

che il processo abbia troppa memoria. In questo caso possono essergli tolti dei frame. Il PFF

cerca così di tenere la frequenza di paginazione per ciascun processo all’interno di limiti

accettabili.

È importante notare che alcuni algoritmi di sostituzione delle pagine possono funzionare con una politica di sostituzione sia locale sia globale.

FIFO, LRU e tutte le approssimazioni dell’LRU possono lavorare sia con politiche di sostituzione delle pagine locali sia globali.

Working set e WSClock possono lavorare esclusivamente con politiche di sostituzione delle pagine locali.

**Controllo del carico**

Può succedere che nel sistema si verifichi del sovraccarico anche con il miglior algoritmo di

sostituzione delle pagine e l’assegnazione ottimale dei frame. Infatti qualora la combinazione

dei working set di tutti i processi superi la capacità di memoria ci si deve aspettare del thrashing.

Un sintomo di questa situazione è che l’algoritmo PFF indica che alcuni processi hanno bisogno di più memoria, ma nessun processo ha bisogno di meno memoria.

L’unica soluzione reale è di sbarazzarsi temporaneamente di alcuni processi. Un buon metodo per ridurre il numero di processi in competizione fra loro per la memoria è di scaricarne alcuni sul disco e liberare tutte le pagine che stavano utilizzando.

Nel decidere quale processo mantenere in memoria e quale mandare su disco occorre considerare il grado di multiprogrammazione, la dimensione del processo, la frequenza di paginazione (PFF) e il tipo di processo (I/O bound o CPU bound).

**Dimensione delle pagine**

La dimensione delle pagine è un parametro scelto dal sistema operativo.

Anche se l’hardware è stato progettato per lavorare con frame di k byte, il sistema operativo può scegliere di lavorare con multipli di k.

Determinare la migliore dimensione delle pagine richiede il bilanciamento di parecchi

fattori in concorrenza fra loro.

Innanzitutto ci sono due fattori a favore delle pagine con dimensioni piccole.

Un segmento di testo del programma, dati o dello stack scelto a caso, non è contenuto in un numero intero di pagine. Lo spazio in più è sprecato. Questo spreco è detto frammentazione interna.

Un altro argomento a favore di dimensioni di pagina piccole è che, in generale, una dimensione di pagina grande fa sì che vi sia nella memoria più spazio inutilizzato.

D’altra parte, le pagine piccole comportano che il programma richieda molte pagine, ossia una tabella delle pagine estesa. I trasferimenti da e verso il disco sono generalmente di una pagina per volta, con la maggior parte del tempo impiegato per la ricerca e il ritardo della rotazione; trasferire pagine piccole dunque richiede quasi tanto tempo quanto trasferire una pagina grande.

Caricare 64 pagine di 512 byte richiede 64 x 10 = 640 ms, mentre per caricare 4 pagine da 8 KB servono solo 4 x 12 = 48 ms.

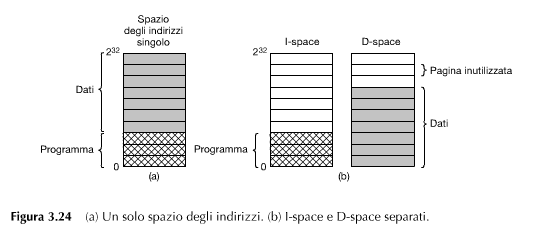
Su alcune macchine la tabella delle pagine deve essere caricata nei registri hardware ogni volta che la CPU passa da un processo a un altro. In questo caso è meglio avere pagine grandi poiché si riduce il tempo necessario per caricare i registri. Gli attuali computer utilizzano pagine da 1, 4 o 8 KB.

Con il crescere delle dimensioni delle memorie anche la dimensione delle pagine tende a crescere anche se in modo non lineare. Quadruplicando la memoria RAM raramente si raddoppia la dimensione della pagina.

**Istruzioni separate e spazi dei dati**

La maggior parte dei computer ha un solo spazio degli indirizzi contenente sia i programmi

sia i dati, come illustrato nella Figura 3.24(a).



Se questo spazio degli indirizzi è abbastanza grande, tutto funziona bene. Quando è troppo piccolo costringe i programmi ad adattarsi nell’intero spazio di indirizzi.

Una soluzione è di avere spazi degli indirizzi separati per le istruzioni e i dati, come mostrato nella Figura 3.24(b). In un computer con una simile progettazione entrambi gli spazi degli indirizzi possono essere paginati, indipendentemente l’uno dall’altro. Ognuno ha la sua tabella delle pagine, con il suo mappaggio delle pagine virtuali nei frame fisici.

**Pagine condivise**

Un altro problema che si presenta durante la progettazione è la condivisione.

In un grande sistema a multiprogrammazione è normale che molti utenti eseguano lo stesso programma nel medesimo istante. Per evitare due copie della stessa pagina in memoria contemporaneamente è ovviamente più efficiente condividere le pagine. Il problema è che non tutte le pagine sono condivisibili. In particolare le pagine di sola lettura (read-only), come il testo dei programmi, possono essere condivise, ma non le pagine dei dati.

Quando due o più processi condividono del codice, si verifica un problema con le pagine condivise. Supponiamo che i processi A e B stiano entrambi eseguendo un editor e condividendo le sue pagine. Se lo scheduler decide di rimuovere A dalla memoria, sfrattando tutte le sue pagine e riempiendo i frame vuoti con qualche altro programma si avrà che, per riportarle di nuovo in memoria, B darà luogo a un gran numero di page fault.

Analogamente, quando A termina, è fondamentale che sia in grado di scoprire se le pagine sono ancora in uso, in modo che il loro spazio su disco non sia liberato accidentalmente.

Cercare tutte le tabelle delle pagine per vedere se una pagina è condivisa è troppo dispendioso; così, per tenere traccia delle pagine condivise, sono necessarie delle strutture dati speciali.

Condividere dati invece che codice è più complesso, ma non impossibile.

**Librerie condivise**

La condivisione può essere fatta con una granularità diversa dalle singole pagine.

Se un programma è avviato due volte, molti sistemi operativi condivideranno automaticamente

tutte le pagine testo in modo che ve ne sia in memoria una sola copia.

Le pagine di testo sono sempre in sola lettura, per cui in questo caso non c’è alcun problema. A seconda del sistema operativo, ciascun processo può avere la sua personale copia privata delle pagine dei dati oppure esse sono condivisibili e contrassegnabili come di sola lettura. Se un qualche processo modifica una pagina di dati, ne viene fatta una copia privata per lui, ossia è applicato il “copy on write”.

Dopo la

fase di traduzione, il linker esegue il collegamento dei programmi oggetto. In essi qualsiasi riferimento esterno non definito (chiamata a funzione di sistema come printf()) è ricercato nelle librerie e incluso nell’eseguibile binario.

Al termine del collegamento il linker scrive sul disco un programma binario eseguibile che contiene tutte le funzioni (dalle librerie sono incluse solo quelle necessarie).

Supponiamo che i normali programmi usino 20-50 MB sia di grafica sia di funzioni

di interfaccia utente. Collegare centinaia di programmi con tutte queste librerie sprecherebbe

una quantità enorme di spazio quando fossero caricate. Ecco il perché delle librerie condivise.

Si utilizzano librerie condivise, dette librerie a collegamento (DLL, “dynamic link library” in Windows).

Quando un programma è collegato con librerie condivise, invece di includere la funzione effettivamente richiamata, il linker include una piccola routine che rimanda alla funzione richiamata al momento dell’esecuzione. Così facendo, se un altro programma ha già caricato la libreria condivisa non serve caricarla nuovamente, che è il punto fondamentale. Si noti che quando una libreria condivisa è caricata o utilizzata, non viene letta in memoria l’intera libreria in un sol colpo.

Oltre a creare file eseguibili più piccoli e risparmiare spazio nella memoria, le librerie condivise hanno un altro vantaggio: se una funzione contenuta nella libreria condivisa viene aggiornata per correggere un errore, non serve ricompilare il programma che la richiama. I vecchi eseguibili binari continuano a funzionare.

Le librerie condivise tuttavia hanno un piccolo problema: non consentono la rilocazione del codice al volo, necessaria quando diversi processi accedono alla stessa libreria.

Esistono due possibili soluzioni:

- copy-on-write delle pagine per ciascun processo (ma così si perdono i vantaggi della condivisione);

- compilare le librerie con uno speciale flag che permette al compilatore di non produrre le istruzioni che utilizzino indirizzi assoluti (codice indipendente dalla posizione).

**Politica di ripulitura**

La paginazione lavora al meglio quando c’è abbondanza di frame liberi che possono essere richiamati se si verificano dei page fault. Per assicurare un rifornimento abbondante di frame liberi, molti sistemi di paginazione dispongono di un processo in background, chiamato paging daemon (demone della paginazione), che resta dormiente per la maggior parte del tempo, ma che periodicamente si risveglia per verificare lo stato della memoria.

Se la quantità di frame liberi è troppo bassa, il paging daemon comincia a selezionare pagine da eliminare usando un qualche algoritmo di sostituzione delle pagine. Se queste pagine sono state modificate da quando erano state caricate, sono scritte su disco.

Mantenere a portata di mano una scorta di frame garantisce prestazioni migliori rispetto a

usare tutta la memoria e poi cercare di ritrovare il frame al momento opportuno. Il paging

daemon assicura perlomeno che tutti i frame liberi siano puliti, così da non doverli scrivere

in tutta fretta sul disco quando sono necessari.

Un modo per implementare questa politica di ripulitura è con un clock a doppia passata.

La prima passata è controllata dal paging daemon. Quando punta a una pagina sporca, la pagina è riscritta sul disco e la prima passata va oltre. Quando punta una pagina pulita va semplicemente oltre.

La seconda passata è usata per la sostituzione delle pagine, come nell’algoritmo clock standard, solo che adesso, grazie al lavoro della prima passata, la probabilità che la seconda passata trovi una pagina pulita è maggiore.

**Swapping su disco**

Quando una pagina viene rimossa dalla memoria non sappiamo dove è memorizzata sul disco. L’algoritmo più semplice (adottato da molti sistemi UNIX) per allocare su disco lo spazio delle pagine e avere una partizione speciale per lo swapping (area di swap) o ancora meglio su un disco separato (per bilanciare il carico di I/O).

All’avvio del sistema questa partizione di swap è vuota ed è rappresentata in memoria come una singola voce, con il suo inizio e la sua dimensione. Nello swapping statico ad ogni processo corrisponde sul disco uno spazio identico a quello occupato in memoria. Quando i processi terminano, il loro spazio su disco viene liberato. Nella tabella dei processi è memorizzato soltanto l’indirizzo di inizio sul disco della sua area di swap.

Nella Figura 3.28(a) vediamo le pagine 0, 3, 4 e 6 in memoria, mentre le pagine 1, 2, 5 e 7 sono sul disco. L’area di swap sul disco è grande quanto il processo (otto pagine), ogni pagina ha una posizione fissa che viene scritta quando la pagina è rimossa dalla memoria. Le pagine vengono memorizzate in modo contiguo in funzione del loro numero di pagina virtuale. Tutte le pagine in memoria hanno una copia ombra (o shadow) su disco, che potrebbe non essere aggiornata se la pagina in memoria è stata modificata.

Questo semplice modello presenta però un problema: i processi possono aumentare di dimensione dopo essere stati avviati. Sebbene il testo programma sia generalmente

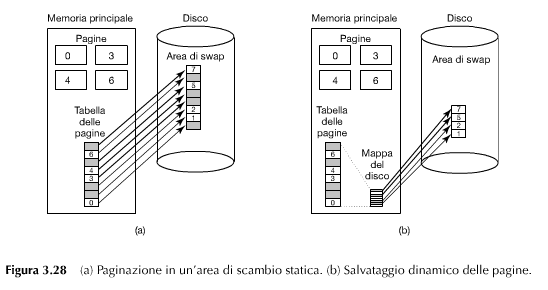
terminato, l’area dei dati può talvolta crescere, così come lo stack.

Possiamo risolvere riservando aree di swap separate per il testo, i dati e lo stack e sovradimensionando tale spazio per permettere la crescita dei dati e dello stack.

Oppure possiamo non allocare niente in anticipo e allocare lo spazio su disco per ciascuna pagina quando ne venga fatto lo swapping su disco, deallocandola quando viene riportata in memoria. In questo modo i processi in memoria non sono vincolati a un dato spazio di swap. Questa tecnica è nota come swapping dinamico. Lo svantaggio è che per tener traccia di ogni pagina su disco è necessario avere un indirizzo del disco in memoria. In altre parole, deve esserci una tabella per ogni processo che indica per ogni pagina su disco dove essa sia.

Lo swapping dinamico è illustrato nella Figura 3.28(b). Nella figura le pagine non hanno un indirizzo su disco fisso. Quando viene fatto lo swapping su disco di una pagina viene scelta

“al volo” una pagina libera sul disco e viene aggiornata la mappa del disco.



**Separazione fra politica e meccanismo**

Uno strumento importante per gestire la complessità di qualunque sistema è di separare la

politica dal meccanismo. Questa teoria può essere applicata alla gestione della memoria,

poiché nella maggior parte dei computer il gestore della memoria è eseguito come un processo

a livello utente.

Il sistema della gestione della memoria è suddiviso in tre parti:

1) un gestore a basso livello dell’MMU;

2) un gestore dei page fault è parte del kernel;

3) un pager esterno eseguito nello spazio utente.

Una volta che il processo inizia a essere eseguito può generare un page fault. Il gestore degli errori capisce di quale pagina virtuale c’è bisogno e invia un messaggio al pager esterno, avvertendolo del problema. Il pager esterno legge quindi la pagina necessaria dal disco e la copia in una porzione del suo spazio degli indirizzi. Poi indica al gestore degli errori dove si trova la pagina. Questo toglie il mappaggio dallo spazio degli indirizzi del pager esterno e richiede al gestore dell’MMU di mettere la pagina nello spazio degli indirizzi dell’utente, al posto giusto.

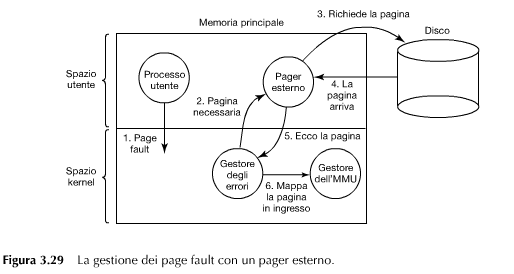
A questo punto il processo utente può essere riavviato.

Questa implementazione lascia aperta la questione su dove è messo l’algoritmo di sostituzione

delle pagine.

La cosa migliore sarebbe averlo nel pager esterno, ma questo metodo comporta alcuni problemi, primo fra tutti che il pager esterno non ha accesso ai bit R e M di tutte le pagine. Così occorre inventare qualche meccanismo per passare queste informazioni al pager esterno.

Altrimenti l’algoritmo di sostituzione della pagine deve andare nel kernel. In quest’ultimo caso, il gestore degli errori avvisa il pager esterno di quale pagina ha selezionato per la rimozione e fornisce i dati. Il vantaggio principale di questa implementazione è di ottenere codice più flessibile. Lo svantaggio principale è il sovraccarico dovuto al continuo passaggio da modalità kernel a modalità utente.



**Segmentazione**

La memoria virtuale illustrata finora è monodimensionale poiché gli indirizzi virtuali vanno da 0 a un certo indirizzo massimo, uno dopo l’altro. Per molti problemi, avere due o più spazi degli indirizzi virtuali può essere molto meglio che averne uno. Per esempio, un compilatore ha tante tabelle costruite man mano che procede la compilazione, includendo eventualmente:

1. il testo sorgente salvato per la stampa dei listati (sui sistemi batch);

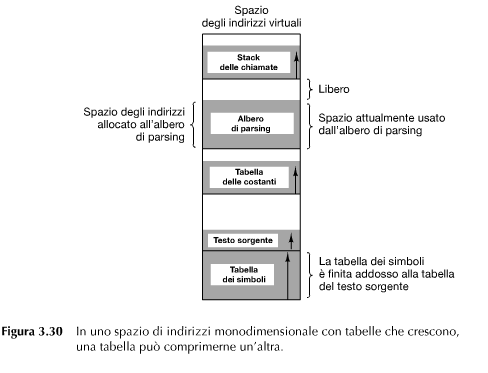
2. la tabella dei simboli, contenente i nomi e gli attributi delle variabili;

3. la tabella contenente tutte le costanti intere e a virgola mobile utilizzate;

4. l’albero di parsing, contenente l’analisi statica del programma;

5. lo stack usato per le chiamate di procedura nel compilatore.

Con il procedere della compilazione ognuna delle prime quattro tabelle cresce continuamente. Durante la compilazione l’ultima cresce e si comprime in modi imprevedibili. In una memoria a una dimensione queste cinque tabelle dovrebbero essere allocare in parti contigue dello spazio virtuale degli indirizzi, come nella Figura 3.30.



Consideriamo che cosa accade se un programma ha un numero di variabili molto più grande del

solito e una normale quantità di tutto il resto. Il pezzo dello spazio degli indirizzi allocato per la tabella dei simboli può riempirsi, ma può esservi molto spazio nelle altre tabelle. Quello che occorre è un modo per liberare il programmatore dal dover gestire l’espansione e la contrazione delle tabelle, nello stesso modo in cui la memoria virtuale elimina il fastidio di organizzare i programmi in overlay.

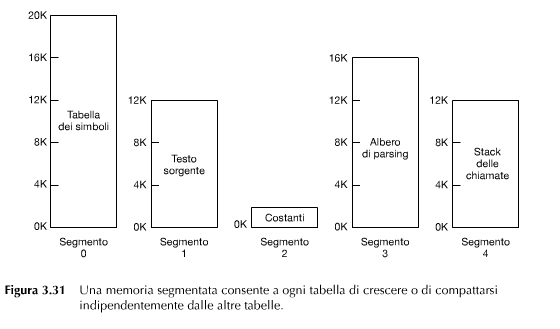
Una soluzione diretta molto generica è fornire la macchina di molti spazi degli indirizzi completamente indipendenti, chiamati segmenti. Ogni segmento consiste di una sequenza lineare di indirizzi, da 0 a un certo massimo. La lunghezza di ciascun segmento può essere qualsiasi, da 0 al massimo consentito. Segmenti diversi possono avere, come solitamente accade, lunghezze diverse. Inoltre la lunghezza dei segmenti può variare durante l’esecuzione. La lunghezza di un segmento di stack può essere aumentata ogni volta che viene inserito qualcosa nello stack e diminuita ogni volta che qualcosa ne è estratto.

Poiché ogni segmento costituisce uno spazio degli indirizzi separato, segmenti diversi possono crescere o compattarsi indipendentemente, senza influenzarsi l’un l’altro.

Per specificare un indirizzo in questa memoria segmentata o bidimensionale, il programma deve fornire un indirizzo a due parti: un numero di segmento e un indirizzo all’interno del segmento.

La Figura 3.31 illustra una memoria segmentata utilizzata per le tabelle del compilatore discusse

in precedenza. Sono mostrati cinque segmenti indipendenti.



È bene sottolineare che il segmento è una entità logica, di cui il programmatore è a conoscenza e che utilizza come accade per gli array, le procedure, etc...

Oltre a semplificare il trattamento delle strutture di dati che crescono o si compattano, una memoria segmentata ha anche altri vantaggi.

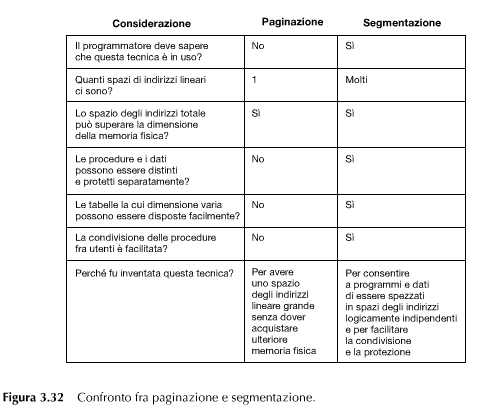
Se ogni procedura occupa un segmento separato, con indirizzo 0 al suo punto di partenza, il linking di procedure compilate separatamente è estremamente semplificato.

La segmentazione facilita inoltre la condivisione di procedure o di dati fra molti processi. Un esempio comune è offerto dalle librerie condivise. Esse possono essere messe in un segmento e condivise da molteplici processi, eliminando la necessità di averla nello spazio degli indirizzi di ogni processo.

I segmenti possono avere diversi tipi di protezione a seconda del loro contenuto (una procedura può essere eseguita e non scritta, un array utilizzato ma non eseguito). Per questo è importante inserire in un segmento un solo tipo di oggetto. Questo tipo di protezione è utile per bloccare gli errori di programmazione.

Mentre il programmatore sa cosa c’è dentro un segmento, il contenuto delle pagine e la loro gestione è totalmente invisibile al programmatore.

Paginazione e segmentazione sono messe a confronto nella Figura 3.32.



**Implementazione della segmentazione pura**

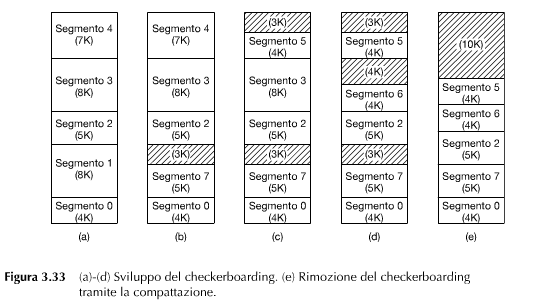
L’implementazione della segmentazione differisce dalla paginazione per un motivo fondamentale: le pagine sono di dimensioni fisse, i segmenti no.

La Figura 3.33(a) mostra un esempio di memoria fisica contenente in principio cinque segmenti. Consideriamo adesso che cosa accade se il segmento 1 è rimosso e il segmento 7, che è più piccolo, viene messo al suo posto. Arriviamo alla configurazione di memoria della

Figura 3.33(b). Fra il segmento 7 e il segmento 2 c’è dell’area inutilizzata. Poi il segmento 4 è sostituito dal segmento 5, come nella Figura 3.33(c) e il segmento 3 è rimpiazzato dal segmento

6, come nella Figura 3.33(d). Dopo che il sistema è stato in esecuzione per un po’, la memoria sarà fatta a pezzetti, qualcuno contenente segmenti e altri degli spazi vuoti. Questo fenomeno, chiamato checkerboarding o frammentazione esterna, spreca la memoria negli spazi vuoti.

Può essere risolto tramite la compattazione, come mostrato nella Figura 3.33(e).



**Segmentazione con la paginazione: MULTICS**

Se i segmenti sono grandi, può essere difficile, se non impossibile, tenerli per intero in memoria principale. Questo porta all’idea di paginarli, in modo che siano in giro solo quelle pagine effettivamente necessarie. Molti sistemi hanno supportato la paginazione dei segmenti.

Il sistema operativo MULTICS forniva ciascun programma di una memoria virtuale sino a

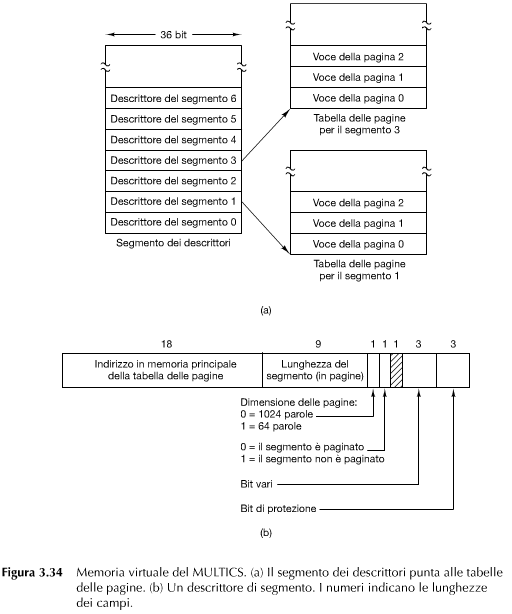
218 segmenti (più di 250.000), ognuno dei quali poteva essere lungo fino a 65.536 (36 bit) parole. Ciascun segmento è paginato, combinando i vantaggi della paginazione (la dimensione delle pagine uniforme senza tenere l’intero segmento in memoria, ma solo la parte che deve essere usata) con i vantaggi della segmentazione (facilità di programmazione, modularità, protezione e condivisione).

Ogni programma MULTICS ha una tabella dei segmenti, con un descrittore per segmento. Poiché nella tabella ci sono potenzialmente più di un quarto di milione di voci, la stessa tabella dei segmenti è segmentata e paginata.

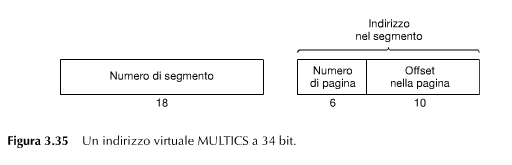
Un descrittore di segmento contiene un’indicazione se il segmento sia nella memoria principale o meno. Se una qualunque parte del segmento è in memoria, il segmento è considerato in memoria e la sua tabella delle pagine sarà in memoria. Se il segmento è in memoria, il suo descrittore contiene un puntatore a 18 bit alla sua tabella delle pagine, come nella

Figura 3.34(a). Poiché gli indirizzi fisici sono a 24 bit e le pagine sono allineate su limiti di 64 byte (il che implica che i 6 bit meno significativi degli indirizzi della pagina siano 000000), nel descrittore servono solo 18 bit per salvare un indirizzo della tabella delle pagine. Il descrittore contiene anche la dimensione del segmento, i bit di protezione e pochi altri oggetti.

La Figura 3.34(b) illustra un descrittore di segmento del MULTICS. L’indirizzo del segmento nella memoria secondaria non è nel descrittore del segmento ma in un’altra tabella usata dal gestore dei segment fault.



Nel MULTICS un indirizzo consiste di due parti: il segmento e l’indirizzo nel segmento. L’indirizzo nel segmento è ulteriormente suddiviso in un numero di pagina e in una parola nella pagina, come illustrato nella Figura 3.35.



Quando avviene un riferimento alla memoria viene messo in atto l’algoritmo descritto di seguito, illustrato nella Figura 3.36 .

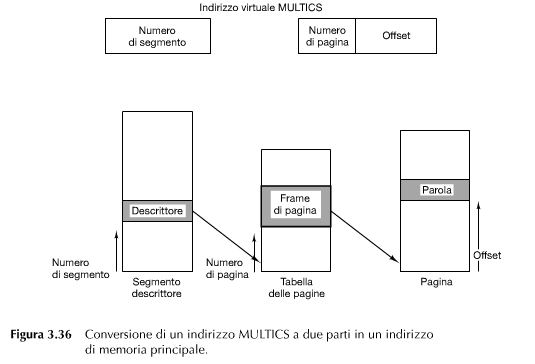
1) Il numero del segmento è usato per trovare il descrittore del segmento.

2) Si controlla se la tabella delle pagine del segmento è in memoria. Se lo è, viene localizzata. Se non lo è avviene un segment fault. Se c’è la violazione della protezione, avviene un trap.

3) Viene esaminata la voce della tabella delle pagine per la pagina virtuale richiesta. Se la pagina stessa non è in memoria viene generato un page fault. Se lo è, viene estratto, dalla voce della tabella delle pagine, l’indirizzo dell’inizio della pagina nella memoria principale.

4) All’inizio della pagina è aggiunto l’offset, per ottenere l’indirizzo nella memoria principale in cui è localizzata la parola.

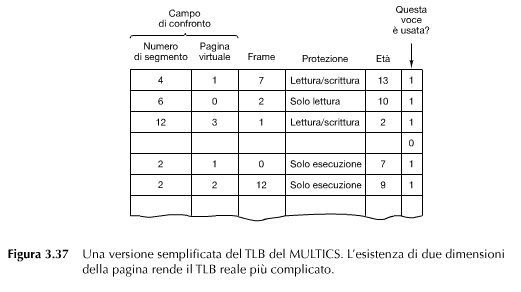
5) La lettura o il salvataggio hanno infine luogo.



Ripetere tutti questi passaggi ad ogni riferimento di memoria può rallentare l’intero sistema.

In realtà l’hardware del MULTICS contiene un TLB ad alta velocità a 16 parole, illustrato nella Figura 3.37, che può ricercare in un sol colpo una determinata chiave in tutte le sue voci.

Ad ogni riferimento di memoria, l’hardware di indirizzamento prima controlla per vedere se l’indirizzo virtuale è nel TLB. Se è così, prende direttamente dal TLB il numero di frame e forma l’indirizzo effettivo della parola referenziata, senza dover guardare nel segmento del descrittore o nella tabella delle pagine.



**Segmentazione con la paginazione: l’Intel x86**

La memoria virtuale sul Pentium è simile a quella sul MULTICS.

Mentre il MULTICS aveva 256.000 segmenti indipendenti, ciascuno con massimo 64.000

parole a 36 bit, l’x86 ha 16.000 segmenti indipendenti, ognuno contenente fino a un miliardo

di parole a 32 bit. Sebbene vi siano meno segmenti, la dimensione dello stesso è più grande.

La memoria virtuale del Pentium è fondata su due tabelle:

- LDT (local descriptor table, tabella dei descrittori locali);

- GDT (global descriptor table, tabella dei descrittori globali).

Ciascun programma ha la propria LDT, ma vi è una sola GDT, condivisa da tutti i programmi sul computer.

La LDT descrive i segmenti locali a ciascun programma, inclusi il suo codice, i dati, lo stack e così via, mentre la GDT descrive i segmenti di sistema, compreso il sistema operativo stesso.

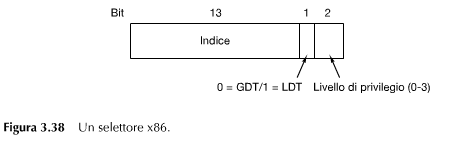
Per accedere a un segmento, un programma x86 carica prima un selettore per quel segmento stesso all’interno di uno dei sei registri segmento della macchina. Durante l’esecuzione, il registro CS mantiene il selettore per il segmento del codice e il registro DS tiene il selettore per il segmento dei dati.

Ogni selettore è un numero a 16 bit, come illustrato nella Figura 3.38:

- 1 bit indica se il segmento è locale o globale, cioè se sta nella LDT (0) o nella GDT (1);

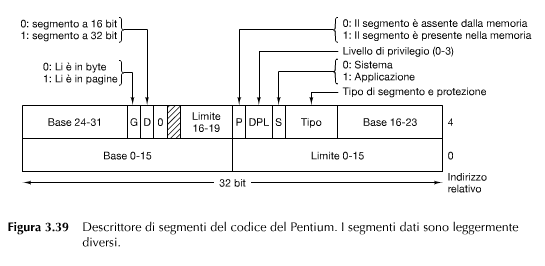
- 13 bit specificano la riga nella LDT o nella GDT;

- 2 bit sono di protezione.



Al momento in cui un selettore è caricato in un registro segmento, il suo descrittore corrispondente è richiamato dalla LDT o dalla GDT e salvato nei registri dei microprogrammi.

Come rappresentato nella Figura 3.39, un descrittore consiste di 8 byte, incluso l’indirizzo di base del segmento, la dimensione e altre informazioni.



Andiamo ora a tracciare i passi tramite cui una coppia <selettore, offset> è convertita in

un indirizzo fisico.

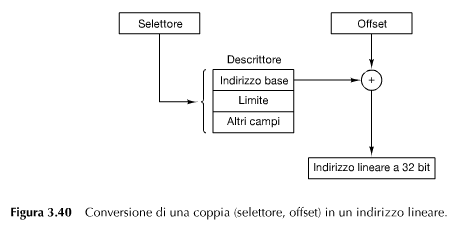
Appena il microprogramma sa quale registro segmento è stato usato, può trovare il descrittore completo corrispondente a quel selettore nei suoi registri interni.

Se il segmento non esiste (selettore 0) o è stato messo fuori dalla paginazione, si verifica un trap. L’hardware usa poi il campo Limite (Limit) per controllare se l’offset sia oltre la fine del segmento, caso in cui si verifica nuovamente un trap.

Logicamente, nel descrittore dovrebbe esservi un campo a 32 bit che fornisce la dimensione del segmento, ma ci sono solamente 20 bit disponibili, così è utilizzato uno schema diverso. Se il campo Gbit (Granularity bit) è 0, il campo Limite è l’esatta dimensione del segmento, fino a 1 MB. Se è 1, il campo Limite fornisce la dimensione del segmento in pagine invece che byte.

Poiché la dimensione della pagina nel Pentium è 4 KB, 20 bit sono sufficienti per segmenti sino a 232 byte.

Dando per assunto che il segmento sia in memoria e l’offset all’interno dell’intervallo, il Pentium aggiunge al campo Base (32 bit) del descrittore l’offset per formare quello che è chiamato indirizzo lineare, come illustrato nella Figura 3.40.



Se la paginazione è disabilitata (tramite un bit nel registro di controllo globale), l’indirizzo lineare è interpretato come un indirizzo fisico e inviato alla memoria per la lettura o la scrittura. D’altra parte, se la paginazione è attivata, l’indirizzo lineare è interpretato come un indirizzo virtuale e mappato sull’indirizzo fisico usando le tabelle delle pagine.

Ogni programma in esecuzione ha una directory delle pagine composta di 1024 voci a 32 bit.

Si trova a un indirizzo puntato da un registro globale. Ogni voce di questa directory punta

a una tabella delle pagine a sua volta contenente 1024 voci a 32 bit. Le voci della tabella delle

pagine puntano ai frame delle pagine. Questo schema è illustrato nella Figura 3.41.

