LEZIONE 4

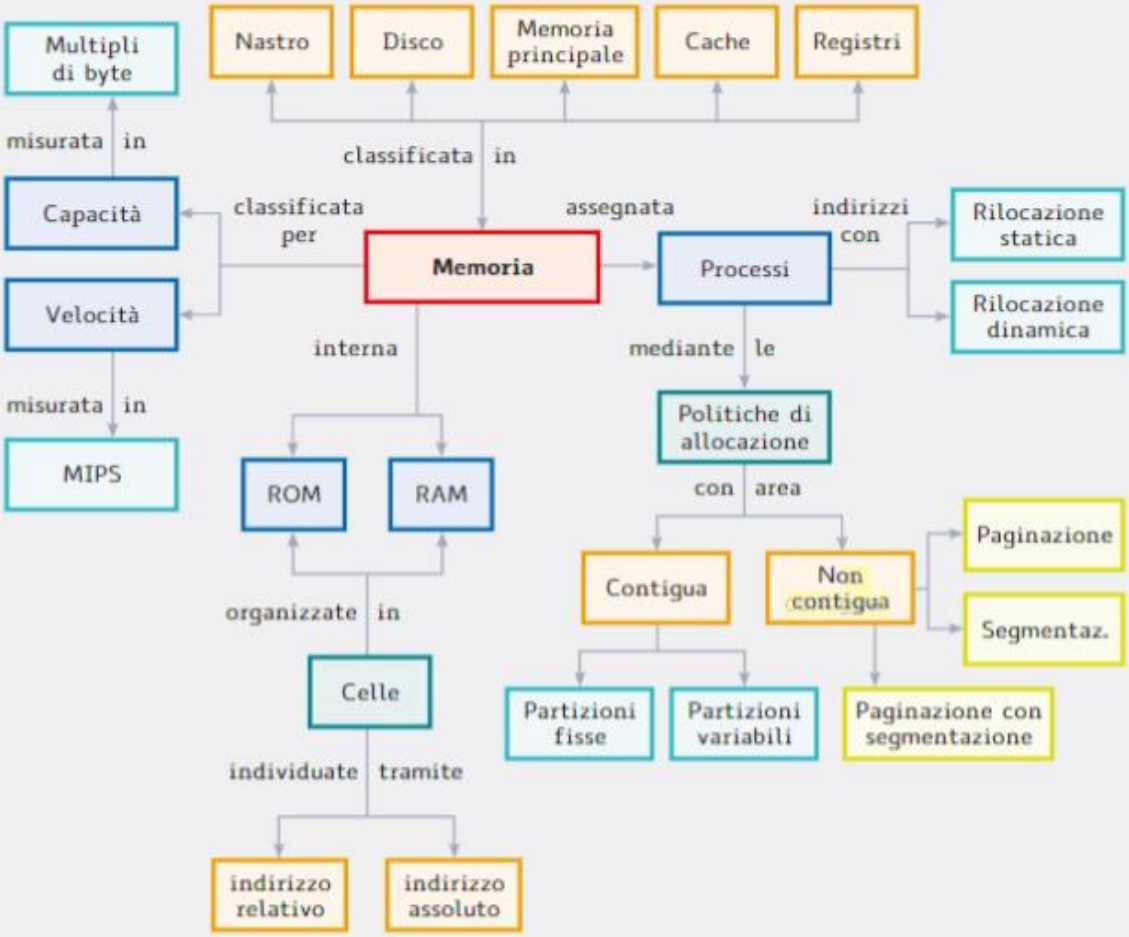
4 La gestione della memoria

IN QUESTA LEZIONE IMPAREREMO...

- la classificazione delle memorie
- i meccanismi di caricamento del programma in memoria
- le tecniche di virtualizzazione della memoria

MAPPA CONCETTUALE

didattica inclusiva



Introduzione

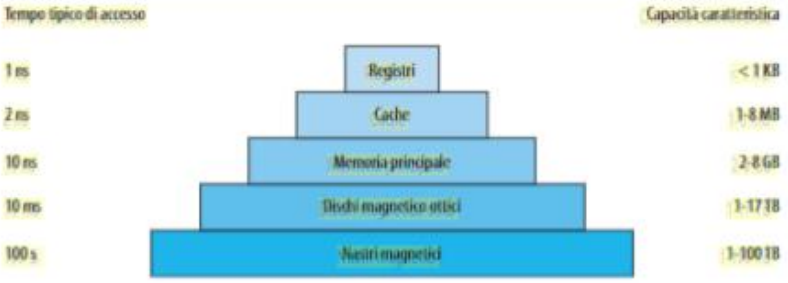
La memoria centrale (RAM) è una risorsa importante per i calcolatori moderni che deve essere gestita al meglio, anche se rispetto al passato la sue dimensioni oggi sono molto aumentate, dell'ordine dei gigabyte.
La memoria centrale è sempre stata una risorsa limitata:
- inizio anni '80: 128 kB;
- inizio anni '90: 1 MB;
- inizio anni 2000: 128 MB;
- attualmente: 4-8 GB.

Ricordiamo una celebre frase di Bill Gates, fondatore di Microsoft, pronunciata nel 1985, che si rivelò profondamente errata: *"Chi mai avrà bisogno di più di 640 K di memoria centrale?"*.
In sostanza si può affermare che la memoria centrale è una risorsa che "non basta mai"!

Ogni utente desidera avere una "memoria infinita", veloce, poco costosa e possibilmente non volatile: naturalmente questi desideri sono contraddittori e quindi non possono essere esauditi tutti contemporaneamente.

Il SO utilizza delle tecniche di gestione della RAM, cercando di "renderla infinita" (memoria virtuale) così da poter sempre esaudire ogni richiesta di spazio effettuata dell'utente.

Nel calcolatore sono presenti diversi tipi di memoria, classificati in base alle loro caratteristiche fondamentali, cioè velocità e capacità:
- nastro: molto capiente, magnetico, sequenziale (memoria di back-up);
- disco: capiente, lento, non volatile ed economico (memoria secondaria);
- memoria principale: volatile, mediamente grande, veloce e costosa;
- cache: volatile, veloce, piccola e costosa;
- registri: all'interno del processore, estremamente veloci e ridotti ad alcuni byte.



In questa Lezione analizzeremo quella parte di SO che gestisce in particolare la memoria principale: il "memory manager" (gestore della memoria) che spesso utilizza anche la memoria disco per svolgere le sue funzioni.
La memoria centrale consiste in un ampio vettore di parole di memoria (o byte), ognuna delle quali ha un proprio indirizzo: la CPU preleva istruzioni e dati direttamente da essa per caricarli nei propri registri, in particolare carica l'istruzione presente nella posizione indicata dal program counter.
Ogni istruzione può a sua volta generare nuovi accessi alla memoria e quindi l'evoluzione di un programma è un susseguirsi di caricamenti (load) e archiviazioni (store) di istruzioni e di dati.

I compiti del gestore della memoria sono sostanzialmente tre:
- sapere sempre quali parti della memoria centrale sono in uso e quali sono libere;
- scegliere quale parte di memoria allocare ai processi che la necessitano e quindi deallocarla;
- gestire lo swapping tra la memoria principale e il disco quando la memoria principale non è sufficientemente grande per mantenere tutti i processi.

Caricamento del programma

Il programma eseguibile, in formato binario, risiede in un file su una memoria permanente, tipicamente un hard disk (memoria secondaria).

> Il problema fondamentale che il gestore della memoria deve risolvere è trasformare il programma eseguibile (su memoria di massa) in un processo in esecuzione (in memoria di lavoro).

I programmi che "stanno per diventare processi", cioè per i quali è già stata fatta la richiesta di caricamento in memoria centrale, vengono messi in una coda di entrata, dalla quale ne verrà selezionato uno (o più) da caricare da parte del loader e quindi da collocare nella lista dei processi pronti (RL).

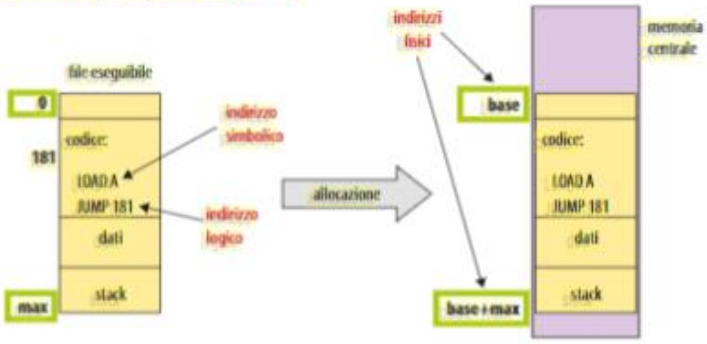
> ⚠ È opportuno fare una precisazione: durante la generazione del file eseguibile il compilatore e il linker generano all'interno del programma dei collegamenti tra istruzioni e indirizzi senza sapere dove il programma o i dati saranno caricati in memoria.

L'assegnazione degli indirizzi è quindi incompleta: vengono cioè generati degli indirizzi relativi (indirizzo logico) e all'atto del caricamento vero e proprio questi vengono trasformati in indirizzi assoluti (indirizzo fisico), quello utilizzato nella memoria RAM).

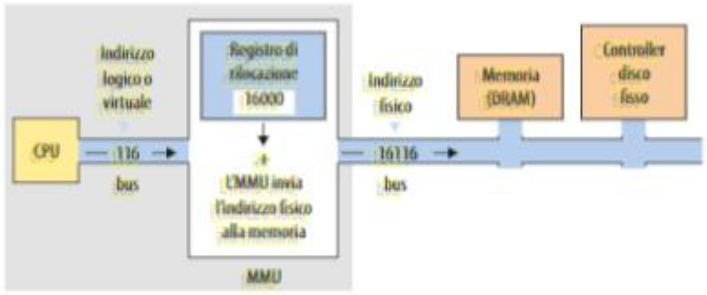
> Un codice che ha tali caratteristiche si chiama codice rilocabile.

Vediamo un semplice esempio di funzionamento in due possibili situazioni rilocazione statica e rilocazione dinamica.
Ogni processo dispone di un proprio spazio di indirizzamento logico [0, max], che viene allocato nella memoria fisica: il compilatore genera gli indirizzi ipotizzando che il programma venga caricato nella memoria a partire dall'indirizzo 0, e in base a questo valore, vengono generati tutti gli indirizzi e i collegamenti dati/istruzioni, secondo due modalità di rilocazione:

- rilocazione statica: all'atto del caricamento in memoria viene individuato l'indirizzo iniziale, indirizzo di base, e viene sommato a tutti i riferimenti presenti nel programma (offset);
- rilocazione dinamica: il programma viene caricato in una zona libera di memoria e, solo in fase di esecuzione, viene inserito in un apposito registro, chiamato registro base (o di rilocazione RL), il valore dell'indirizzo effettivo della prima locazione di memoria centrale; quindi durante l'esecuzione, istruzione per istruzione, si calcola l'indirizzo assoluto sommando a ogni indirizzo relativo il valore di contenuto del registro base RL.



Il dispositivo hardware che è in grado di associare gli indirizzi virtuali agli indirizzi fisici è il Memory Management Unit (o MMU).
Nello schema a lato si può osservare come il valore contenuto nel registro di rilocazione viene sommato a ogni indirizzo generato dai processi utente nel momento stesso in cui l'indirizzo viene inviato alla memoria.



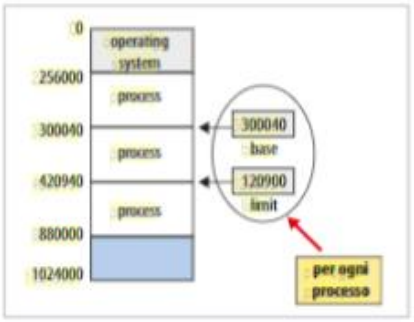
La formula generale è quindi:

indirizzo fisico = indirizzo logico + offset

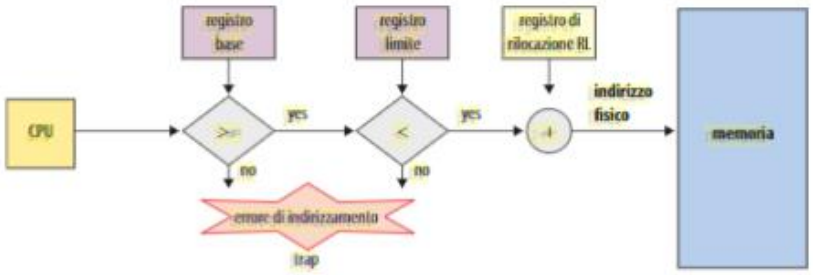
Gli indirizzi logici assumono quindi valori che vanno da 0 a un valore massimo (maxind), mentre i corrispondenti indirizzi fisici vanno da (RL + 0) a (RL + maxind):

Il passaggio dall'indirizzo logico all'indirizzo fisico si definisce address binding.

Il registro di rilocazione RL contiene quindi l'indirizzo fisico più piccolo: è anche presente un ulteriore registro, il registro limite che contiene il massimo indirizzo utilizzabile dagli indirizzi logici.
Nel caso di locazione contigua, questi registri vengono utilizzati per effettuare il controllo di protezione della memoria per garantire che i processi non accedano senza permesso ad aree riservate ad altri processi.



Dato che è impossibile controllare a priori quali indirizzi di memoria un programma referenzierà questo controllo deve essere fatto durante l'esecuzione del processo:



Binding e linking

Il calcolo degli indirizzi logici viene effettuato nella fase di linking dei programmi; il passaggio dall'indirizzo logico a quello fisico avviene nella fase di binding che può essere fatta in momenti diversi:

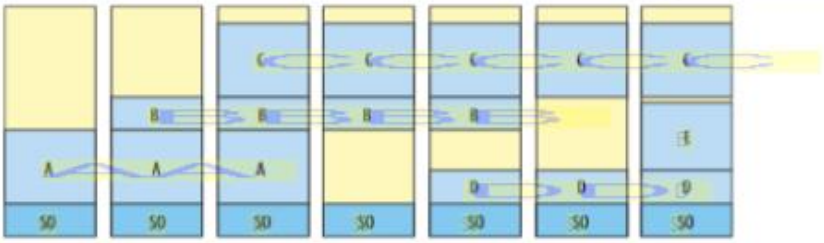
- compile time: durante la compilazione, è però necessario che le locazioni di memoria siano note a priori (indirizzamento assoluto);
- load time: nel momento del loading del programma (rilocazione statica);
- execution time: durante l'esecuzione, soprattutto se si hanno librerie dinamiche (rilocazione dinamica); necessita di hardware di supporto come registri base e limite, descritti in precedenza.

Loading

Sappiamo che per essere eseguito un programma deve risiedere in memoria centrale, ma per poterlo fare è necessario che lo spazio libero in memoria sia sufficientemente grande da poterlo contenere e inoltre che questo spazio sia contiguo.
Possiamo subito fare alcune semplici osservazioni:

- oggi i programmi hanno dimensioni notevoli sicuramente superiori alla dimensione della memoria RAM: è impensabile che il programmatore scriva un programma cercando di risparmiare spazio, come si faceva negli anni Ottanta!

- nei sistemi multiprogrammati il continuo caricamento e scaricamento dei programmi produce una frammentazione della memoria creando delle regioni libere di dimensioni spesso ridotte: magari la somma dello spazio totale libero sarebbe sufficiente a contenere il programma, ma le zone non sono contigue e il SO dovrebbe effettuare un'operazione di compattamento della memoria spostando tutti i programmi già caricati (ed effettuando per ciascuno una rilocazione dinamica, molto costosa in termini di tempo, quindi di efficienza);



- va però osservato che non tutte le istruzioni che sono scritte in un programma devono contemporaneamente essere presenti in memoria anche perché magari buona parte di queste non verranno mai eseguite (si pensi a tutte le opzioni "inutilizzate" da programmi tipo Word o Excel!).

Sapendo che l'obiettivo è quello di ottimizzare l'utilizzo della memoria, elenchiamo alcune possibili soluzioni.

1 Swapping: nei sistemi multiprogrammati è possibile effettuare lo scaricamento dei processi temporaneamente inattivi dalla RAM a disco per liberare spazio in caso di necessità: questa operazione prende il nome di swapping e genera un insieme di operazioni che rallentano notevolmente il SO provocando un calo di prestazioni. Nello specifico lo swapping si compone di quattro fasi:
   - *identificare* i processi inattivi presenti in memoria (per esempio in stato di attesa);
   - *salvare sulla memoria temporanea* i loro dati (dati globali, heap, stack);
   - rimuoverli dalla memoria centrale (*scaricamento*);
   - caricare nello spazio appena liberato il processo che deve essere eseguito (*caricamento*).
2 Caricamento dinamico: si possono caricare in memoria solo parti fondamentali del programma e lasciare su disco moduli che solo al momento di un'effettiva richiesta saranno caricati successivamente in memoria: questa tecnica si chiama caricamento dinamico e viene molto usata soprattutto nel collegamento alle librerie di sistema (DLL, Dynamic Link Library).
3 Overlay: nel caso di processi con dimensione maggiore della memoria centrale è possibile effettuare a priori già un frazionamento del programma a opera del programmatore, individuando le parti che possono essere tra loro "alternative" e che verranno caricate in memoria nella stessa zona proprio alternativamente. La tecnica si chiama overlay, che sta a indicare che nella stessa zona di memoria vengono sovrapposte più sezioni di programma, naturalmente una alla volta.
4 Partizionamento: molti problemi nascono dalla differenza di dimensione tra i programmi; la soluzione ottimale sarebbe quella di riservare a ogni processo una dimensione fissa di memoria per semplificare le operazioni di swapping e ridurre la frammentazione: a tal fine viene utilizzata la tecnica di partizionamento della memoria.

Queste soluzioni sono implementate nelle tecniche di allocazione.

Tecniche di allocazione della memoria centrale

Esistono molte tecniche per l'allocazione del codice e dei dati dei processi in memoria centrale, riportiamo di seguito la loro classificazione in base al tipo di sistema:

a monoprogrammato
- allocazione contigua
  - a partizione singola

b multiprogrammato
- allocazione contigua
  - a partizioni fisse
  - a partizioni variabili (o dinamiche)

- allocazione non contigua (memoria virtuale)
  - paginazione
  - segmentazione
  - segmentazione con paginazione

> ⚠ Nella tecnica di allocazione continua a partizione singola la parte di memoria disponibile per l'allocazione dei processi di utente non è partizionata e, quindi, solo un processo alla volta può essere allocato in memoria: non c'è multiprogrammazione.

Tutte le altre tecniche sono state progettate per far evolvere più processi contemporaneamente: le analizziamo dettagliatamente in questa Lezione.

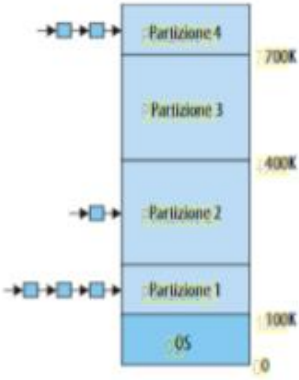
Allocazione della memoria: il partizionamento

Il sistema più semplice per allocare la memoria centrale nei sistemi multiprogrammati è quello di suddividerla in partizioni e assegnare una partizione a un processo indipendentemente dalla sua dimensione, partendo dalla prima (che viene generalmente riservata al sistema operativo) e cercando di riempire contiguamente tutte le partizioni.
Naturalmente la scelta della dimensione della partizione è fondamentale per l'efficienza del sistema in quanto effettuare partizioni troppo piccole potrebbe provocare problemi di frammentazione, mentre segmenti troppo grandi rischiano di sprecare memoria con i processi di piccole dimensioni e di aumentare le richieste di swapping per mancanza di memoria.
Di seguito vengono presi in esame due diversi schemi di partizionamento.

Schema a partizione fissa

Nello schema a partizione fissa la dimensione della partizione viene definita all'atto dell'inizializzazione del sistema, quindi staticamente, e viene creata una tabella dove si memorizza lo stato delle partizioni, indicando quali sono libere e quali occupate (e da chi sono occupate).

> ⚠ Le partizioni possono anche essere di dimensione diversa tra loro e possono essere stabilite dall'operatore all'avvio del SO.



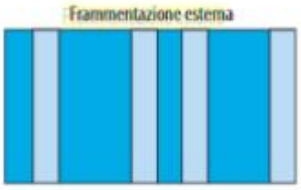
Per ogni partizione viene successivamente gestita una coda dei processi in attesa in base alle loro dimensioni (come nell'esempio della figura a lato): un nuovo job viene aggiunto alla coda relativa alla partizione più piccola che è in grado di contenerlo.
Nella partizione fissa si possono verificare due problemi:
- il problema della frammentazione interna, che si presenta quando le singole partizioni sono di grandi dimensioni;
- il problema della frammentazione esterna, che si presenta quando le singole partizioni sono di piccole dimensioni.

Frammentazione interna

Supponiamo che ogni processo occupi un'intera partizione e che venga schedulato un job "piccolo": se ogni partizione piccola ma sufficientemente grande per contenerlo ha la coda piena e invece la coda di una partizione grande è vuota, il job piccolo viene assegnato alla partizione grande. Come si vede nella figura a lato potremmo sprecare quasi tutta la partizione grande per un job molto piccolo.

Frammentazione esterna

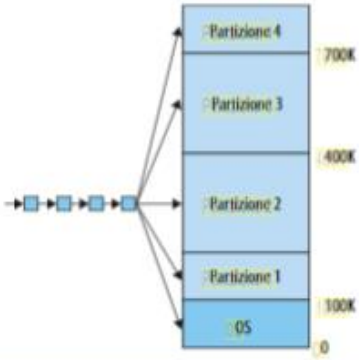
La dimensione di un processo può essere più grande di una qualunque partizione esistente anche se, nel complesso, la memoria associata a tutte le partizioni è sufficiente per contenerla.



> ⚠ Il grado di multiprogrammazione è limitato al numero di partizioni: non posso eseguire processi se non sono in grado di allocare una partizione.

Il problema della frammentazione interna si può risolvere mediante l'uso di una singola coda di ingresso; quando si libera una partizione, si possono avere due strategie di assegnazione ai job:

- si percorre la coda a partire dalla testa fino a individuare un job di dimensioni tali da poter essere contenuto;
- si percorre tutta la coda individuando il job più grande che può essere contenuto nella partizione che è libera: in questo modo, però, vengono discriminati i job di dimensione ridotta.



Schema a partizione variabile o dinamico

Nello schema a partizione variabile è possibile modificare dinamicamente sia il numero sia la dimensione di ogni singola partizione, per esempio unendo blocchi contigui precedentemente distinti o suddividendone uno particolarmente sovradimensionato, a seconda delle richieste di spazio all'atto del caricamento dei processi.

> ⚠ È anche possibile creare blocchi di dimensione specifica per il nuovo processo direttamente al momento del suo caricamento: avremo quindi tante partizioni quanti sono i processi attivi.

In questo caso il lavoro del SO è più complesso, in quanto lo spazio disponibile per i nuovi processi continua a variare e quindi deve decidere continuamente dove caricare i nuovi processi che attendono nella coda di entrata: utilizza gli algoritmi di allocazione dinamica della memoria centrale.
Il problema dell'allocazione dinamica della memoria centrale ha come strategia risolutiva quattro possibili alternative:

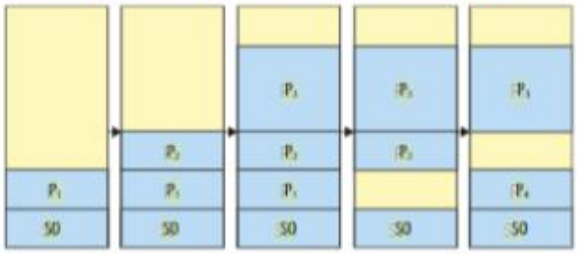
1. first-fit: il gestore della memoria scandisce la tabella dei segmenti finché trova la prima zona libera abbastanza grande per contenere il programma; è molto veloce perché la ricerca finisce subito al primo matching;
2. next-fit: il gestore della memoria individua la prima zona libera sufficientemente grande a partire dall'ultimo buco utilizzato;
3. best-fit: il gestore della memoria scandisce tutto l'elenco dei segmenti e sceglie la zona libera più piccola sufficientemente grande da contenere il processo; è un metodo più lento del first-fit e spreca più memoria perché lascia zone di memoria troppo piccole per essere utilizzate;
4. worst-fit: per risolvere il problema precedente sceglie la zona libera più grande ma, statisticamente, si è verificato che questo è il peggior algoritmo di allocazione.

> ⚠ Le simulazioni hanno dimostrato che le strategie migliori sono le prime due, in particolar modo la first-fit, che è la più veloce.

La frammentazione della memoria

Come possiamo vedere nella figura di pagina seguente, che simula lo stato della RAM in seguito all'esecuzione di una sequenza di processi, il partizionamento dinamico è generalmente soggetto a frammentazione esterna:

per esempio la terminazione dei processi P1 e P2 e il successivo caricamento del processo P4 hanno provocato la formazione di due frammenti di dimensioni modeste all'interno della RAM.

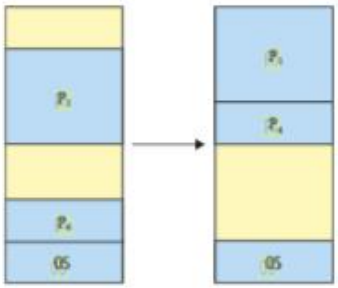


Una possibile soluzione è quella di spostare periodicamente i processi all'interno della RAM per fare in modo di "ricompattare le aree di memoria libere" (memory compaction) ottenendo quindi un'intera zona da riutilizzare.

Questa procedura è simile a quella utilizzata sui dischi rigidi (defrag) per togliere la deframmentazione.

ESEMPIO

Applicando la memory compaction nel caso sopra descritto, questa porterebbe ad avere la situazione riportata nella figura seguente:

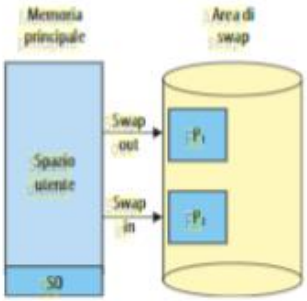


È però importante notare che l'operazione di compattazione della memoria è assai onerosa dal punto di vista computazionale (per esempio, per ricopiare una memoria di 1 Kb si impiega un secondo, dato che la velocità tipica della copia è di 1 byte/ms e durante questo arco di tempo nessun processo può ovviamente essere eseguito.

Potrebbe essere un problema di difficile soluzione soprattutto nei sistemi interattivi, dove provocherebbe il decadimento delle prestazioni.

Area di swap

Per ottimizzare l'utilizzo della memoria generalmente il partizionamento dinamico fa uso del disco come area di appoggio (backing store o swap) per i processi che per un qualsiasi motivo non sono in grado di continuare e passano nello stato di wait: per esempio, un processo che effettua una chiamata di I/O viene bloccato e la sua immagine trasferita interamente su swap (swap out) e quando la chiamata di I/O è completata, e quindi il processo può riprendere la sua normale evoluzione, l'immagine del processo viene nuovamente trasferita in memoria centrale (swap in) per riprendere l'esecuzione.



Una variante dello swapping, usata per algoritmi di scheduling (a medio termine) a priorità è detta "roll out, roll in", dove processi a bassa priorità vengono "swappati" per permettere il ripristino dei processi a priorità maggiore.

Memoria virtuale: introduzione

Entrambe le tecniche descritte in precedenza determinano problemi di frammentazione della memoria lasciando dei blocchi di memoria liberi tra quelli occupati, blocchi che, se fossero uniti e contigui, potrebbero ospitare un altro processo.
Man mano che si utilizza il sistema, lo spazio perso diventa considerevole, provocando un lento ma inesorabile declino delle prestazioni. La tecnica della compattazione, ovvero la fusione di tutti i blocchi liberi in uno solo, non sempre può essere applicata e in ogni caso è piuttosto onerosa in termini di computazione.

> I moderni sistemi operativi introducono quindi il concetto di memoria virtuale, realizzata con le tecniche di paginazione, segmentazione e tecniche ibride che uniscono i vantaggi di entrambe le soluzioni.

Il principale motivo dell'introduzione di queste tecniche è quello di ridurre la frammentazione della memoria, cioè della quantità di memoria che risulta non utilizzabile.

L'introduzione di una tecnica di allocazione non contigua può portare a un altro fondamentale vantaggio: caricare in memoria solo le parti di programma che effettivamente sono utili per l'evoluzione del processo in quel dato istante e liberare le altre parti per poter caricare nuovi task.

L'idea di base è quella di fare in modo che il SO mantenga in memoria principale solo le parti del programma in uso e trattenga il resto su disco (idea, tra l'altro, già utilizzata nelle tecniche di swapping e di caricamento dinamico, ma realizzata in modo diverso).

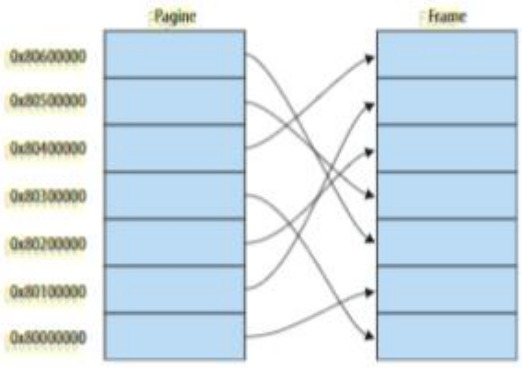
> Statisticamente nei programmi vale il principio di località: quando viene eseguita un'istruzione, la probabilità più alta è che subito dopo ne venga eseguita una a essa vicina: quindi è sufficiente avere caricato un "pezzo" di codice mentre il resto può rimanere sulla memoria di massa.

Mediante questa tecnica è quindi possibile recuperare altro spazio della memoria centrale per massimizzare il numero di processo in esecuzione contemporanea.
Inoltre caricando un pezzo di programma per volta è quindi possibile mandare in esecuzione programmi di qualunque dimensione: con una macchina a 32 bit ogni processo può indirizzare 4 GB e con questa tecnica può in effetti "credere" di averla tutta a sua disposizione, anche se praticamente gliene viene dato un "pezzo alla volta".

Memoria virtuale: paginazione

Il primo metodo che descriviamo per risolvere i problemi appena descritti è la paginazione.
Gli obiettivi della paginazione sono i seguenti:

- mantenere in memoria solo le parti necessarie;
- gestire ogni volta solo piccole porzioni di memoria;
- non sprecare spazio evitando la frammentazione;
- ridurre la frammentazione interna a valori trascurabili;
- poter utilizzare porzioni di memoria non contigue per lo stesso programma;
- non porre vincoli al programmatore (come nelle overlay).



Nella paginazione sia il programma sia la memoria centrale vengono suddivisi in pagine di dimensione fissa:

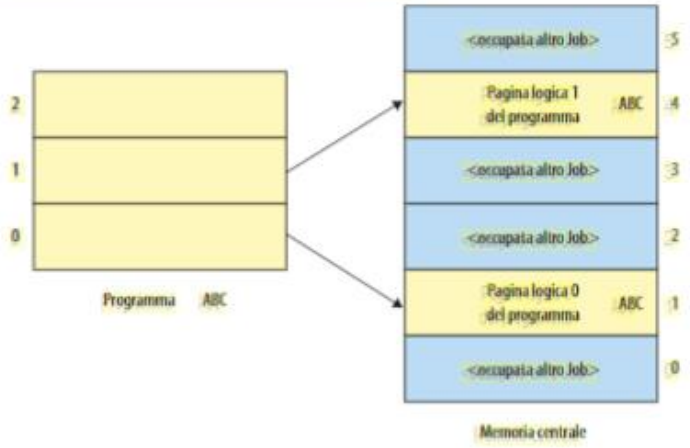
- la memoria fisica in blocchi chiamati frame o pagine fisiche;
- il programma in blocchi di uguale dimensione detti pagine (o pagine logiche).

Naturalmente il numero di pagine logiche (la cui somma costituisce la dimensione del programma) può essere diverso dal numero di pagine fisiche (la cui somma è la dimensione della memoria):

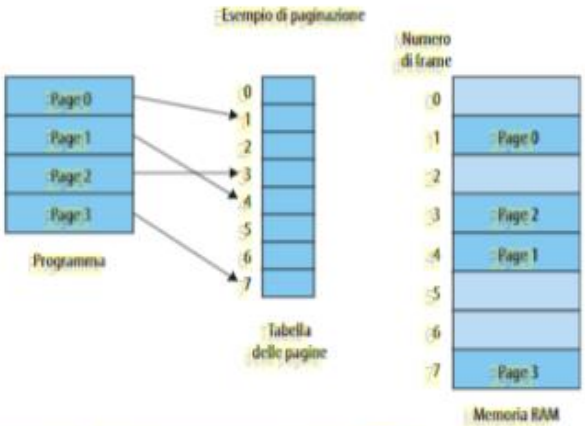
- se il numero delle pagine logiche è minore del numero delle pagine fisiche (libere) il programma potrebbe anche essere caricato tutto in memoria;
- se il numero delle pagine logiche è maggiore del numero delle pagine fisiche (libere) il programma verrà caricato in memoria parzialmente.

Non necessariamente le pagine fisiche devono essere contigue e quindi le pagine logiche possono trovarsi "mischiate" nella memoria centrale: naturalmente affinché un processo possa sfruttare questa tecnica di gestione condizione necessaria è che il codice sia rilocabile dinamicamente.

Un beneficio importante è che con la paginazione, per poter eseguire un programma all'atto del caricamento iniziale, è sufficiente che venga caricata in memoria la prima pagina, quella cioè che contiene la prima istruzione da eseguire: è quindi sufficiente che nella memoria fisica sia libera una sola pagina.
Nella figura seguente si vede come il nuovo processo non viene completamente caricato in RAM, ma solo due pagine logiche trovano spazio libero e tale spazio non è contiguo.

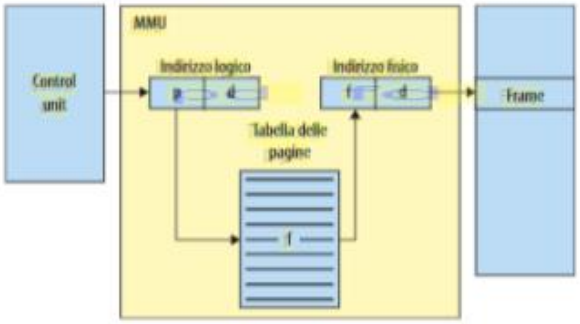


Il SO cura una tabella delle pagine dove il numero stesso di pagina fisica viene utilizzato come indice: nella tabella sono memorizzati gli indirizzi fisici iniziali di tutti i frame presenti nella memoria fisica oltre alle indicazioni generali, cioè se la pagina è occupata o libera, ed eventualmente l'ID del processo che la occupa.
Inoltre, anche a ogni processo viene associata una tabella delle pagine specifica dove sono indicati quali segmenti di codice sono caricati in RAM e quali su disco.
La gestione delle pagine rientra tra i compiti del SO mentre la traduzione da indirizzo virtuale a indirizzo fisico viene effettuata da MMU, precedentemente già descritto, dove:



- lo spazio degli indirizzi logici rappresenta l'intero insieme degli indirizzi generabili dalla CPU;
- lo spazio degli indirizzi fisici rappresenta l'intero insieme degli indirizzi visibili dalla memoria.

Per ottenere un indirizzo di memoria dobbiamo ora avere a disposizione due elementi, ovvero il numero di pagina (p) e lo spiazzamento nella pagina o offset (d): la somma di questi due elementi permette di ricavare l'indirizzo fisico desiderato (binding tra indirizzo logico e fisico).



Dal numero di pagina la MMU preleva dalla tabella delle pagine l'indirizzo fisico (f) al quale deve essere sommato l'offset (d).
La paginazione permette la condivisione del codice: se il codice è rientrante, cioè si separa il codice eseguibile dal record di attivazione, una sola copia di codice read-only può essere condivisa tra più processi; casi tipici sono gli editor, lo shell, i compilatori, per i quali tutti i processi che lo utilizzano condividono il codice nelle stesse locazioni logiche mantenendo, naturalmente, una copia separata dei propri dati.

Il SO determina, tramite le politiche fetch policy, quando una pagina deve essere portata nella RAM.
Le fetch policy si basano su due concetti fondamentali:

1. demand page: la pagina è caricata in memoria solo quando viene fatto un riferimento a una sua locazione, in modo da lasciare il maggior numero di free frame;
2. prepaging: si cerca di sfruttare il principio di località spaziale caricando, oltre alla pagina desiderata, alcune pagine adiacenti però dello stesso processo; questa tecnica però può risultare inefficace se si caricano troppe pagine inutili.

Se è necessario caricare una pagina e non ci sono frame liberi siamo in presenza di una sovrallocazione.

Protezione della memoria

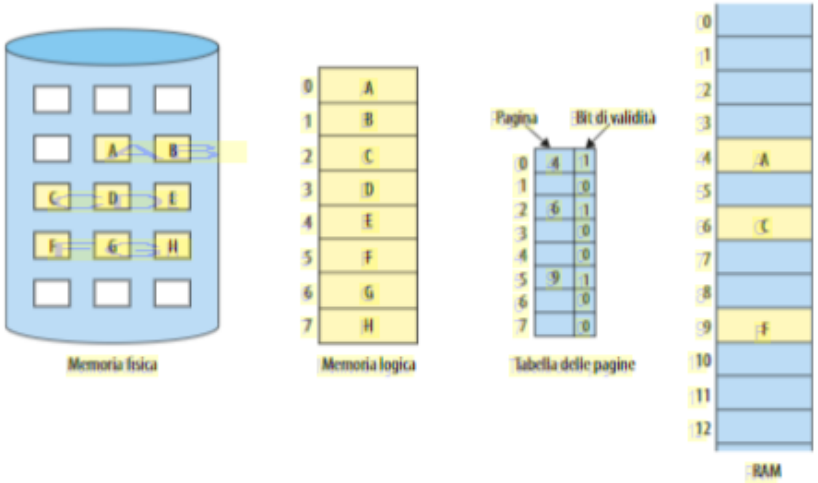
Nelle tabelle delle pagine, a ogni frame vengono associati alcuni bit di protezione che possono indicare se l'accesso al frame è consentito in sola lettura, in lettura-scrittura oppure in sola esecuzione.
I tentativi di accesso errati (o illegali) generano una segnalazione al sistema operativo (trap) che agirà di conseguenza, generalmente sospendendo il processo che sta facendo un'operazione non corretta.

Un apposito bit, inoltre, indica che la pagina associata è nello spazio logico del processo e quindi per esso è legale accedervi (valid) oppure, in caso contrario, non è nello spazio logico del processo (invalid) e questo sta quindi effettuando una violazione di indirizzi (segment violation).

Page fault

Sono anche presenti dei bit di validità che indicano se la pagina appartiene o meno allo spazio logico del processo; se durante l'esecuzione di un programma viene fatto riferimento a un'istruzione che non è presente in alcuna pagina tra quelle caricate in memoria, la MMU determina un'eccezione della CPU al SO detta page fault trap.

A ogni pagina logica viene aggiunto nella tabella delle pagine un bit di validità, che a seconda dello stato della pagina cambia di valore (1 = pagina in memoria, 0 = pagina non in memoria).



Quindi quando una pagina viene caricata in memoria, il suo bit di validità viene posto a 1 per indicare la sua presenza in memoria: nella fase di traduzione dall'indirizzo virtuale all'indirizzo fisico si consulta la tabella delle pagine e se il bit di validità è uguale a 0 si ha un page fault.
Il SO quindi deve provvedere a caricare quella pagina:

- se sono presenti in memoria frame liberi, la pagina viene caricata immediatamente;
- se tutte le pagine fisiche sono occupate, è necessario "fare spazio", cioè si sceglie un frame di pagina scegliendo la più adatta, si salva il suo contenuto su disco e solo in quel momento è possibile caricare la pagina nel frame appena liberato (procedura di page replacement).

In entrambi i casi, naturalmente, si deve cambiare la mappa delle pagine e solo allora il SO riparte con l'istruzione bloccata.

Località dei programmi

Si è osservato che ogni programma, in una certa fase di esecuzione:

- usa solo un **sottoinsieme** delle sue pagine logiche;
- il sottoinsieme delle pagine **effettivamente utilizzate** varia lentamente nel tempo.

Questi fatti sono dovuti alla natura e alla tipologia delle istruzioni, nonché alla strutturazione dei programmi: abbiamo due possibili cause per le quali possiamo definire il "principio di località spazio-temporale":

- località spaziale: esiste un'alta probabilità di accesso a locazioni vicine nello spazio logico/virtuale a locazioni appena accedute, come, per esempio, a elementi di un vettore, alle istruzioni "adiacenti" in un codice sequenziale ecc.;
- località temporale: esiste un'alta probabilità di accesso a locazioni accedute di recente, come, per esempio, nel caso di cicli, funzioni ricorsive ecc.

La località spazio-temporale è statisticamente ben verificata dalla maggior parte dei programmi: è stato calcolato che il 90% del tempo è speso sul 10% codice, dato che la maggioranza delle istruzioni appartiene a cicli interni con corpo composto da poche istruzioni iterate numerose volte.

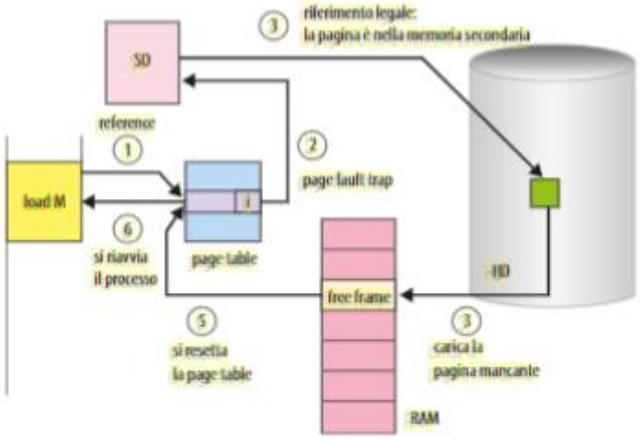
Anche per i dati è stato verificato che il loro utilizzo soddisfa il principio di località spazio-temporale.

Di questo principio se ne tiene conto negli algoritmi di sostituzione di seguito descritti.

Trattamento del page fault

Quando il kernel riceve l'interruzione dovuta al page fault esegue questa sequenza di operazioni:

1 riferimento a una pagina;
2 pagina non presente: viene generato un page fault trap;
3 verifica del motivo del page fault (mediante una tabella interna al kernel):
  – riferimento illegale (violazione delle politiche di protezione): provoca la terminazione del processo (abort),
  – riferimento legale: la pagina è in memoria secondaria e deve essere caricata su un frame libero;
4 individua un frame libero ed effettua il caricamento della pagina;
5 aggiorna la tabella delle pagine;
6 riprende l'esecuzione del processo: viene mandata in esecuzione l'istruzione interrotta dal page fault.



È necessario provvedere alla sostituzione di una pagina presente in memoria (frame vittima) per "fare posto" alla nuova pagina da caricare.

Replacement policy: la scelta della pagina da scaricare

La scelta della pagina "vittima" viene effettuata mediante le strategie di replacement policy che sostanzialmente affrontano nella progettazione i tre aspetti seguenti:

1 Resident set management: è necessario stabilire il numero di frame da allocare per ogni processo e definire se rimpiazzare le pagine scegliendole solo tra quelle del processo che ha determinato il page fault oppure anche tra le altre pagine di altri processi che sono presenti in quel momento in memoria;
2 Page replacement algorithm: l'obiettivo è quello di attenere il minimo numero di page fault e, quindi, è di particolare importanza la scelta delle pagine "vittime" che dovranno lasciare posto in memoria alle nuove pagine (i principali algoritmi sono descritti di seguito);
3 Optimal Page Replacement algorithm: oltre a individuare la pagina vittima sarebbe opportuno individuare la "migliore pagina vittima", cioè quella che potrebbe non essere più necessaria e quindi non più caricata in memoria oppure quella che verrà riferita più in là nel tempo, cioè "il più tardi possibile": questo desiderio teorico di ottimizzazione non è generalmente realizzabile anche perché potrebbe richiedere più "dispendio di tempo" in calcolo statistico rispetto al benefici che se ne ricaverebbe, dato che ci si deve basare su una stima analizzando le frequenze degli accessi alle pagine usate nelle precedenti esecuzioni del processo.

La procedura per la sostituzione della pagina vittima Pvitt con la pagina Pnew da caricare è la seguente:

1 individuazione della vittima Pvitt;
2 salvataggio di Pvitt su disco;
3 caricamento di Pnew nel frame liberato;
4 aggiornamento tabelle;
5 ripresa del processo.

Dirty bit

In generale, quindi, la sostituzione di una pagina richiede due trasferimenti di dati da/verso il disco:
- per scaricare la vittima Pvitt;
- per caricare la pagina nuova Pnew.

È però possibile che la frame vittima non sia stata modificata rispetto alla copia residente sul disco e, quindi, non sia necessario salvarla, risparmiando così un'operazione di trasferimento.

ESEMPIO

I casi in cui la pagina esistente non ha subito modifiche sono:
- le pagine di codice (read-only);
- le pagine contenenti dati che non sono stati modificati durante la permanenza in memoria.

In questi casi la copia della vittima sul disco può essere evitata.

Per sfruttare questa situazione e rendere più efficiente il trattamento del page fault in caso di sovrallocazione si introduce in ogni elemento della tabella delle pagine un bit di modifica M (dirty bit):
- se è settato a 1 significa che la pagina ha subito almeno un aggiornamento da quando è caricata in memoria;
- se è 0: la pagina non è stata modificata.

> ⚠ Qualunque sia l'algoritmo di sostituzione questo, prima di scaricare la pagina Pvitt, esamina il bit di modifica della vittima ed esegue lo swap-out solo se il dirty bit è settato.

Page replacement algorithm (algoritmi di sostituzione)

Questi algoritmi effettuano la scelta delle pagine "vittime" che andranno sovrascritte in memoria dalle nuove pagine da caricare con l'obiettivo di minimizzare il n° page fault: per il principio di località dei riferimenti è meglio non scegliere pagine usate frequentemente di recente: probabilmente occorrerà ancora utilizzarle nel prossimo futuro e dovrebbero, a breve, essere riportate in memoria.

> ⚠ La finalità di ogni algoritmo di sostituzione delle pagine è sostituire quelle pagine la cui probabilità di essere accedute a breve termine è bassa.

Le principali strategie (politiche) di sostituzione che individuano la pagina vittima sono riportate di seguito.

1. **FIFO Page Replacement Algorithm**
   Con questa politica viene sostituita la pagina che occupa da più tempo la memoria, indipendentemente dal suo uso: per implementare questa politica è quindi necessario memorizzare un time stamping che permetta di costruire la cronologia dei caricamenti in memoria oppure realizzare una lista che tenga memoria della sequenza con la quale le pagine sono state caricate.

   > ⚠ Questa tecnica, però, non tiene conto del principio di località dei riferimenti, solo della sua "anzianità".

2. **Least Recently Used (LRU)**
   Questa tecnica si basa sul principio di località dei riferimenti: viene sostituita la pagina che è stata usata meno recentemente, cioè che "da più tempo risulta inutilizzata".
   Per implementare questa politica è necessario registrare la sequenza degli accessi alle pagine in memoria:
   - memorizzando un time-of-use come per la FIFO in un campo che rappresenta l'istante in cui è avvenuto l'ultimo accesso alla pagina;

- introducendo una struttura dati tipo stack in cui ogni elemento rappresenta una pagina: quando si accede a una pagina se ne modifica la posizione ponendola al top: quindi il bottom contiene la pagina vittima.

> ⚠ In quest'ultimo caso il costo di ricerca della pagina è pressocché nullo, mentre non lo è il costo di gestione.

3. **Not Recently Used (NRU) (LRU approssimato o algoritmo dell'orologio)**
Esiste una semplice ottimizzazione della tecnica FIFO che ovvia al problema dello *swap* anche delle pagine molto utilizzate semplificando l'algoritmo LRU: al posto della sequenza degli accessi viene aggiunto un bit di uso R, nella tabella che tiene traccia dell'età delle pagine, associato a ciascuna pagina caricata, gestito come segue:
- al momento del caricamento è inizializzato a 0;
- quando la pagina viene acceduta (riferita), viene settato a 1;
- tutti i bit di uso vengono resettati periodicamente.

> ⚠ In questo modo, le pagine utilizzate di frequente hanno alta probabilità di rimanere in memoria principale: si ricerca la pagina vittima solo tra quelle che hanno il bit R = 0 applicando l'algoritmo FIFO.

Esiste una versione modificata di questo algoritmo che, oltre al controllo del bit d'uso R, si avvale anche del bit di modifica M (dirty bit) prima descritto, ottenendo le seguenti combinazioni:
0) R=0, M=0 pagina non riferita e non modificata;
1) R=0, M=1 pagina non riferita ma modificata;
2) R=1, M=0 pagina riferita ma non modificata;
3) R=1, M=1 pagina riferita e modificata.

Va rimpiazzata la pagina appartenente alla classe più bassa: tra tutte le pagine non usate di recente (bit di uso = 0), ne viene scelta una che non è stata aggiornata (dirty bit = 0).

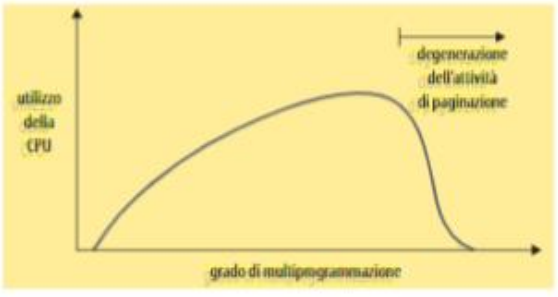
4. **Least Frequently Used (LFU) e Most Frequently Used (MFU)**
Queste due tecniche si basano su algoritmi chiamati "di conteggio" in quanto effettuano il conteggio del numero di riferimenti fatti a ciascuna pagina:
- LFU (Least Frequently Used): sostituisce la pagina con il minor numero di riferimenti: si basa sull'idea che una pagina molto usata ha un conteggio alto, mentre una pagina che serve poco avrà un conteggio basso;
- MFU (Most Frequently Used): sostituisce la pagina con il maggior numero di riferimenti: si basa sul principio che una pagina con contatore basso è stata probabilmente caricata da poco, quindi è utile mantenerla.

Per implementare queste politiche è necessario associare un contatore degli accessi a ogni pagina, in modo da individuare come vittima quella con il minimo valore del contatore.

> ⚠ In questo caso è previsto sia un costo di ricerca sia un costo di gestione.

Working set

Essendo la memoria secondaria molto più lenta (centinaia o migliaia di volte) rispetto alla memoria principale, lo swap dei processi e il rimpiazzamento delle pagine causa un considerevole rallentamento del sistema, che potrebbe anche essere impegnato quasi esclusivamente in operazioni di I/O diventando presto inutilizzabile e poco o per nulla reattivo ai comandi dell'utente: questo fenomeno è chiamato thrashing.
Il grafico a lato riporta come può degenerare l'attività di paginazione fino al thrashing.

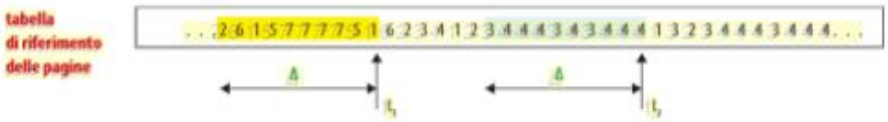


Per prevenire il thrashing bisogna assicurarsi che un processo abbia sempre a disposizione il numero dei frame necessari (resident set) per evolvere, e il modello di riferimento chiamato del working set (WS) cerca di definire tale valore in base al modello di località tempo-spaziale.
In questa tecnica viene definito un parametro Δ per definire la dimensione della "finestra di lavoro", cioè la dimensione del working set, procedendo con l'analisi degli ultimi Δ riferimenti alle pagine: tali pagine, indifferentemente che siano state accedute in lettura o scrittura, fanno parte del working set.

⚠ Il working set così individuato è un'approssimazione della "dimensione" dell'area di località del programma.

ESEMPIO

Vediamo un esempio di come individuare il valore per il WS: a parità Δ otteniamo due risultati diversi:



$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$

$WS(t_w)$ è l'insieme di pagine indirizzate dal processo P nei più recenti Δ riferimenti: Δ definisce la "finestra" del working set.

⚠ L'idea è quella di mantenere il working set di ogni processo aggiornandolo dinamicamente, in base al principio di località temporale, così che all'istante t vengono mantenute le pagine usate dal processo nell'ultima finestra WS(t).

È quindi importante individuare un corretto valore per Δ in quanto in base a esso si influenza la precisione dell'algoritmo:
- se è troppo basso il working set non include l'intera località e non riesce a contenere il numero di page fault;
- se è troppo alto è possibile che più località si sovrappongano e vengano caricate pagine non necessarie.

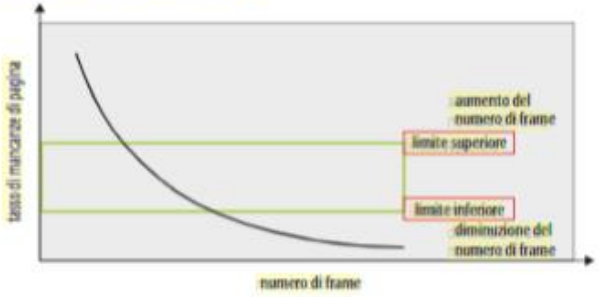
È possibile calcolare la richiesta totale di frame D di ogni processo una volta che è noto il working set con la seguente formula:

$$D = \sum_{i=1}^{n} WSS_i$$

dove con WSSi viene indicato il numero di pagine referenziate dal processo P durante gli ultimi Δ riferimenti.
Indicato con m un numero dei frame liberi, in base al valore di D abbiamo queste possibili situazioni:
- se D < m il SO assegna a ogni processo il numero di frame WSSi e solo se rimangono sufficienti frame liberi è possibile avviare un nuovo processo;
- se D > m il SO individua uno (o più) processi da sospendere (swapping) per assicurare l'evoluzione degli altri, garantendo che la paginazione non degeneri in thrashing.

⚠ Esiste anche un metodo più diretto (e più semplice) per controllare il trashing: quello di stabilire a priori un valore accettabile per la frequenza di mancanze di pagina e di tenere costantemente sotto controllo il valore attuale che si verifica durante il run time: se il tasso attuale è troppo basso, si deallocano alcuni frame del processo, mentre se il tasso è troppo alto, si allocano nuovi frame al processo.

Una variante di questo metodo consiste nel tenere due valori per il tasso di mancanza di pagina, un limite inferiore e uno superiore, come riportato nel disegno seguente:

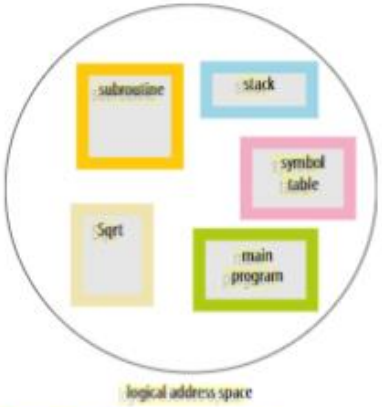


Conclusioni

Il vantaggio della paginazione è che si elimina completamente il problema della frammentazione esterna della memoria in quanto le dimensioni di pagine e frame coincidono e, inoltre, possono essere assegnate anche in modo non contiguo.
Come svantaggio si ha, invece, che si viene a creare una separazione tra vista utente e vista fisica della memoria.

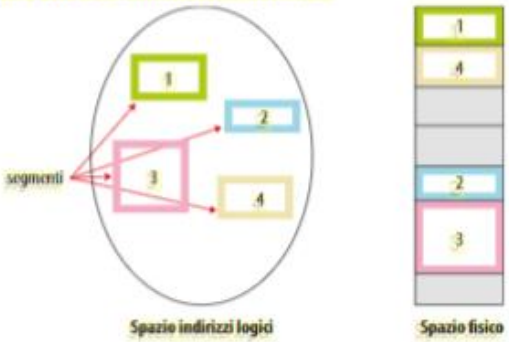
Memoria virtuale: segmentazione

Nella paginazione il programma viene suddiviso in pagine a prescindere dal suo contenuto, cioè, per esempio, senza distinguere l'area del codice da quella dei dati: questo impedisce di fatto che si possa condividere il codice su due istanze dello stesso programma, obbligando a caricare in memoria più volte lo stesso codice. Per ovviare a questo problema si utilizza la segmentazione: è uno schema di gestione della memoria centrale che mantiene la separazione tra memoria logica e fisica come nella paginazione, ma suddivide quest'ultima in segmenti di dimensione variabile. L'idea di base è quella di suddividere la memoria centrale nello stesso modo in cui sono divisi logicamente i programmi, cioè in entità con diverse funzioni, e quindi associare a ogni modulo software un segmento di memoria che può contenere una procedura, un array, uno stack o un insieme di variabili: un segmento è tipizzato ed è caricato in un frame di medesima dimensione.

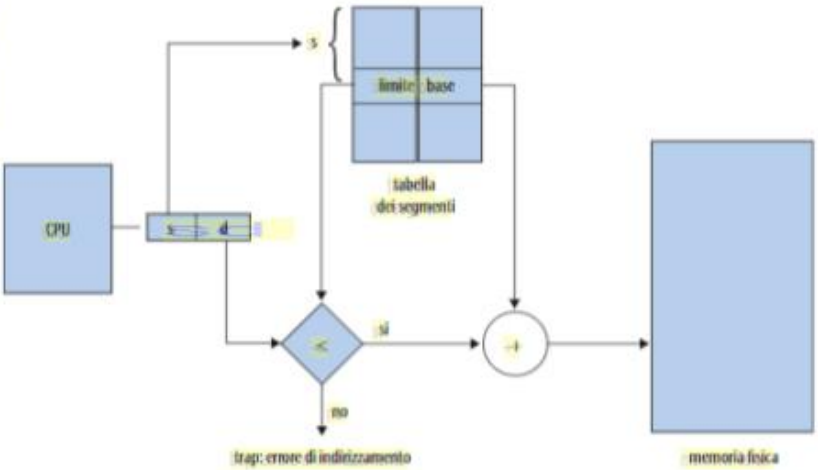


Nella segmentazione la memoria centrale fisica è divisa in segmenti fisici (frame) di dimensioni diverse, mentre lo spazio di indirizzamento del processo è diviso in segmenti logici (segmenti).

A ogni segmento viene associato:
- un numero che permette di individuarlo;
- la sua lunghezza, dato che sono tutti di dimensione variabile.



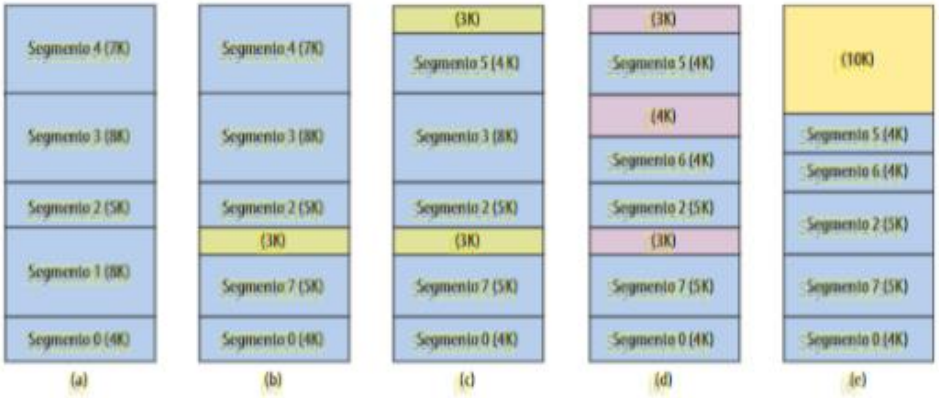
Vengono tenute una o più tabelle dei segmenti dove il numero di segmento funge da indice e contiene anche l'*indirizzo iniziale di memoria centrale* (base) e il *limite del segmento* (limite) cioè il valore massimo che lo scostamento associato può assumere: per individuare l'indirizzo fisico assoluto è quindi sufficiente sommare il valore base allo scostamento.



I segmenti di un processo possono essere caricati in frame non contigui in memoria centrale fisica, mentre quelli non caricati sono conservati nell'area di swap.

Con la segmentazione si ottengono i seguenti vantaggi:
- rispetto alla paginazione, la gestione di strutture dati dinamiche, che per loro natura crescono o diminuiscono, è semplificata;
- viene facilitata la possibilità di condivisione di segmenti tra alcuni processi;
- si semplifica il linking di procedure compilate separatamente;
- ogni segmento può avere un diverso tipo di protezione.

Un problema caratteristico della segmentazione è il frazionamento della memoria: questo avviene nel caso in cui i segmenti vengano continuamente caricati e sostituiti in memoria generando zone inutilizzate (frammentazione esterna chiamata anche checkerboarding).



Situazione iniziale: prima allocazione (a)
Situazione intermedia: caricamento e sostituzione di segmenti (b –c)
Situazione intermedia: frammentazione eccessiva (d)
Situazione finale: ricompattazione (e)

L'hardware dedicato per il supporto alla segmentazione è anche in questo caso la MMU, stavolta orientata alla gestione dei segmenti: si applicano le tecniche già descritte di first-fit o best-fit per l'allocazione dei segmenti.

Conclusioni

Con la segmentazione si ottengono i seguenti vantaggi:
- rispetto alla paginazione, la gestione di strutture dati dinamiche, che per loro natura crescono o diminuiscono, è semplificata;
- viene facilitata la possibilità di condivisione di segmenti tra alcuni processi;
- si semplifica il linking di procedure compilate separatamente;
- ogni segmento può avere un diverso tipo di protezione;
- viene mantenuta una consistenza tra vista utente e vista fisica della memoria.

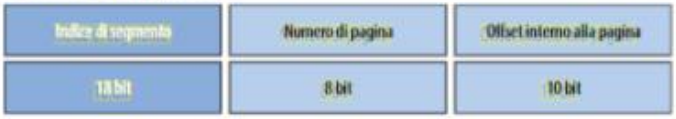
Gli svantaggi sono sostanzialmente dovuti al fatto che:
- è richiesta l'allocazione (dinamica) dei segmenti;
- si è soggetti a una potenziale frammentazione esterna.

Segmentazione con paginazione

Questa tecnica è stata proposta per sfruttare contemporaneamente i vantaggi della paginazione e della segmentazione:
- dalla paginazione si prende l'identificazione dei frame liberi, la scelta del frame libero in cui caricare una pagina, senza alcuna frammentazione;
- dalla segmentazione si prende la condivisione di porzioni di memoria, la verifica degli accessi e delle operazioni.

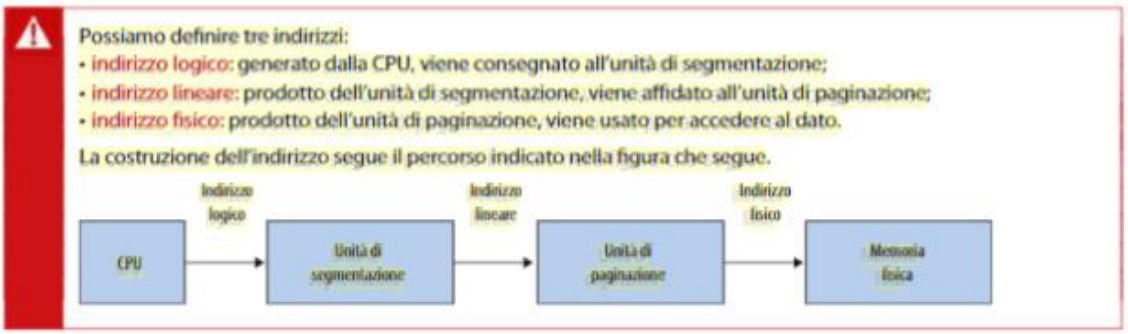
La soluzione utilizzata consiste nel "paginare i segmenti".

Dato che la memoria centrale fisica è divisa in pagine fisiche (*frame*) di dimensione fissa, mentre lo spazio di indirizzamento del processo è suddiviso in segmenti logici (*segmenti*) di dimensioni diverse, ciascuno suddiviso in pagine logiche, introduciamo la paginazione logica dei segmenti, dove le pagine logiche hanno la stessa dimensione dei frame:
- a ogni segmento è assegnato un numero e ogni segmento è suddiviso in più pagine;
- ogni segmento possiede una sua tabella delle pagine;
- l'indirizzo logico è composto quindi da tre componenti:
  - il numero di segmento;
  - il numero di pagina;
  - lo spiazzamento all'interno della pagina;



- l'indirizzo fisico invece è analogo a quello della paginazione, cioè composto dal numero di frame e l'offset nel frame.

La gestione degli indirizzi è effettuata in modo completamente automatico dal SO e dalla MMU che effettua la traduzione dell'indirizzo logico in quello fisico.



Possiamo definire tre indirizzi:
- indirizzo logico: generato dalla CPU, viene consegnato all'unità di segmentazione;
- indirizzo lineare: prodotto dell'unità di segmentazione, viene affidato all'unità di paginazione;
- indirizzo fisico: prodotto dell'unità di paginazione, viene usato per accedere al dato.

La costruzione dell'indirizzo segue il percorso indicato nella figura che segue.