Allocazione non contigua

Paginazione

Allocazione non contigua - paginazione

- Le partizioni fisse/variabili non sono efficienti nell'uso della memoria e soffrono di frammentazione interna/esterna
- I moderni sistemi operativi usano la **paginazione**
  - Permette di allocare ai processi spazio in memoria in celle **NON** contigue
  - Permette di tenere in memoria solo una parte del programma
  - Necessita di hardware adeguato e il codice deve essere rilocabile dinamicamente

Paginazione

Memoria fisica

- La memoria principale viene divisa in blocchi chiamati **frame**
- Ogni frame ha la stessa dimensione (potenza di 2, da 512 byte a 16MB)
- Al processo vengono allocati frame non contigui

Memoria logica

- La memoria principale viene divisa in blocchi chiamate **pagine**
- La memoria logica è uno spazio contiguo
- Se un processo ha dimensione *n* pagine, trovo *n* frame liberi
- Ho frammentazione interna per ultimo frame

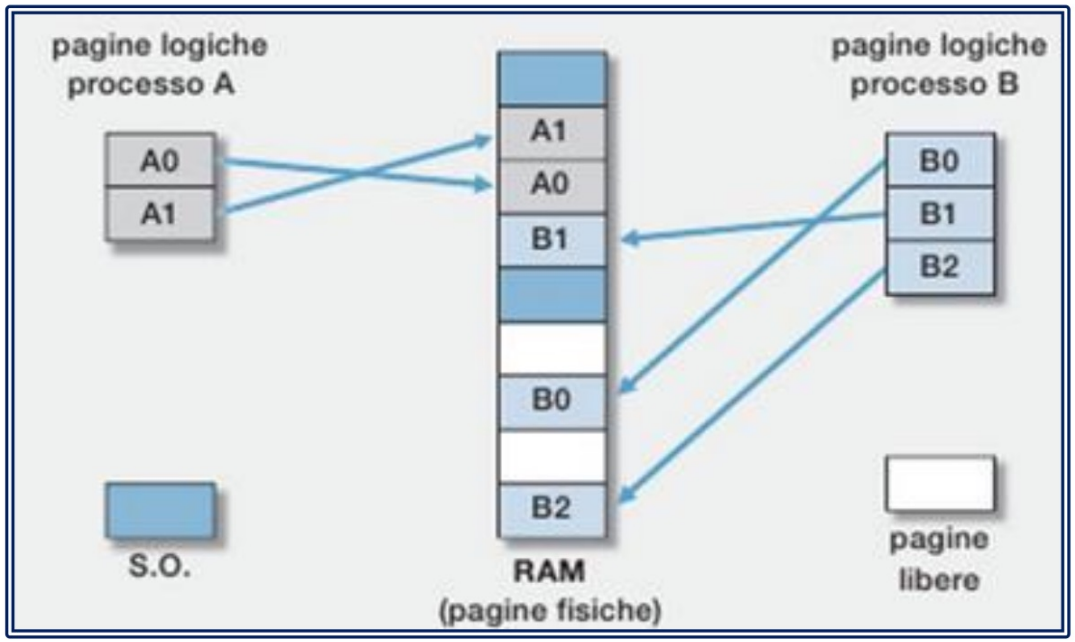


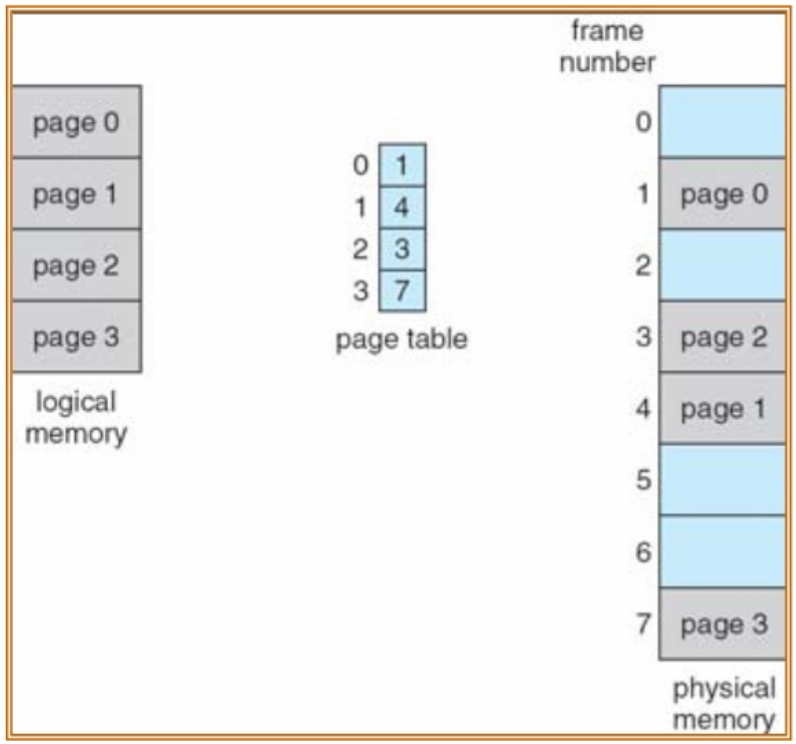
Tabella delle pagine (1)

Memoria fisica

- La memoria principale viene divisa in blocchi chiamati **frame**
- Ogni frame ha la stessa dimensione (potenza di 2, da 512 byte a 16MB)
- Al processo vengono allocati frame non contigui

Memoria logica

- La memoria principale viene divisa in blocchi chiamate **pagine**
- La memoria logica è uno spazio contiguo
- Se un processo ha dimensione *n* pagine, trovo *n* frame liberi
- Ho frammentazione interna per ultimo frame



Il sistema operativo usa una tabella delle pagine per tenere traccia delle pagine associate al processo (il PCB di un processo contiene un puntatore alla tabella delle pagine)

Il numero della pagina serve come indice di accesso

Al numero di pagina corrisponde l'indirizzo base in memoria

Possibile condividere codice

Tabella delle pagine (2)

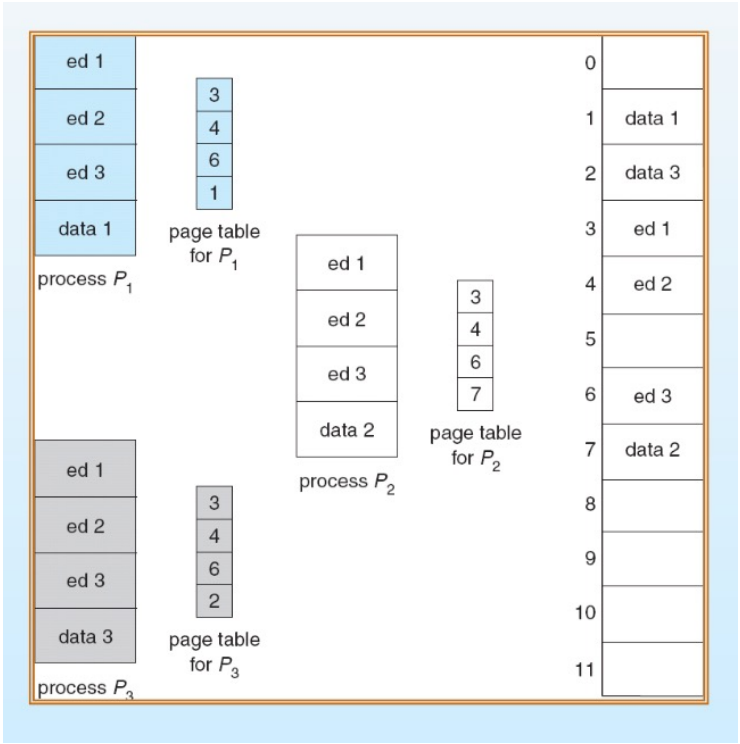
Memoria fisica

- La memoria principale viene divisa in blocchi chiamati **frame**
- Ogni frame ha la stessa dimensione (potenza di 2, da 512 byte a 16MB)
- Al processo vengono allocati frame non contigui

Memoria logica

- La memoria principale viene divisa in blocchi chiamate **pagine**
- La memoria logica è uno spazio contiguo
- Se un processo ha dimensione *n* pagine, trovo *n* frame liberi
- Ho frammentazione interna per ultimo frame

(attenzione!!!! Libro di testo figure sbagliate pg. 268 e 270)



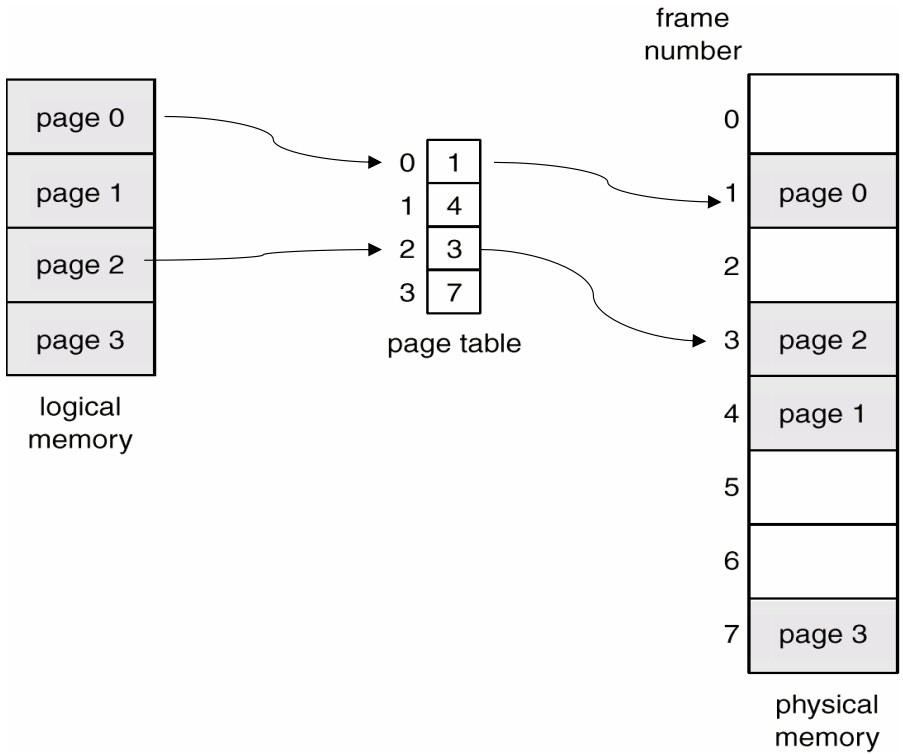
Il sistema operativo usa una tabella delle pagine per tenere traccia delle pagine associate al processo

Il numero della pagina serve come indice di accesso

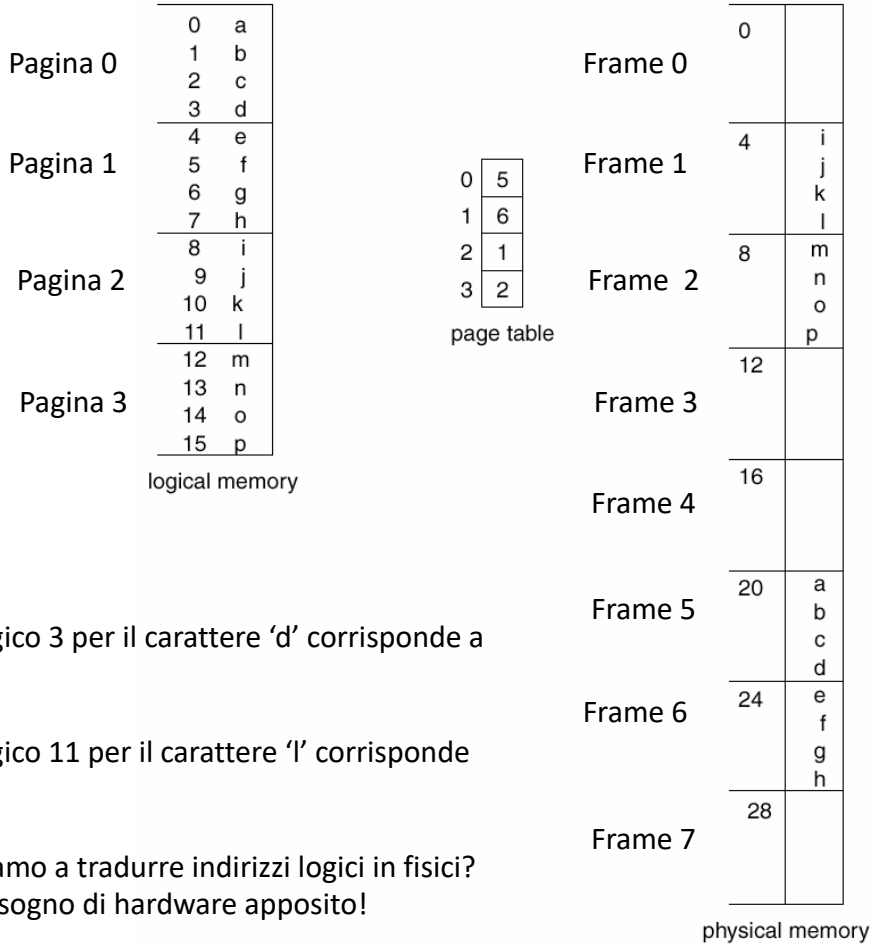
Al numero di pagina corrisponde l'indirizzo base in memoria

Possibile condividere codice

Esempio traduzione indirizzi logici a fisici



Memoria 32 byte, pagine di 4 byte



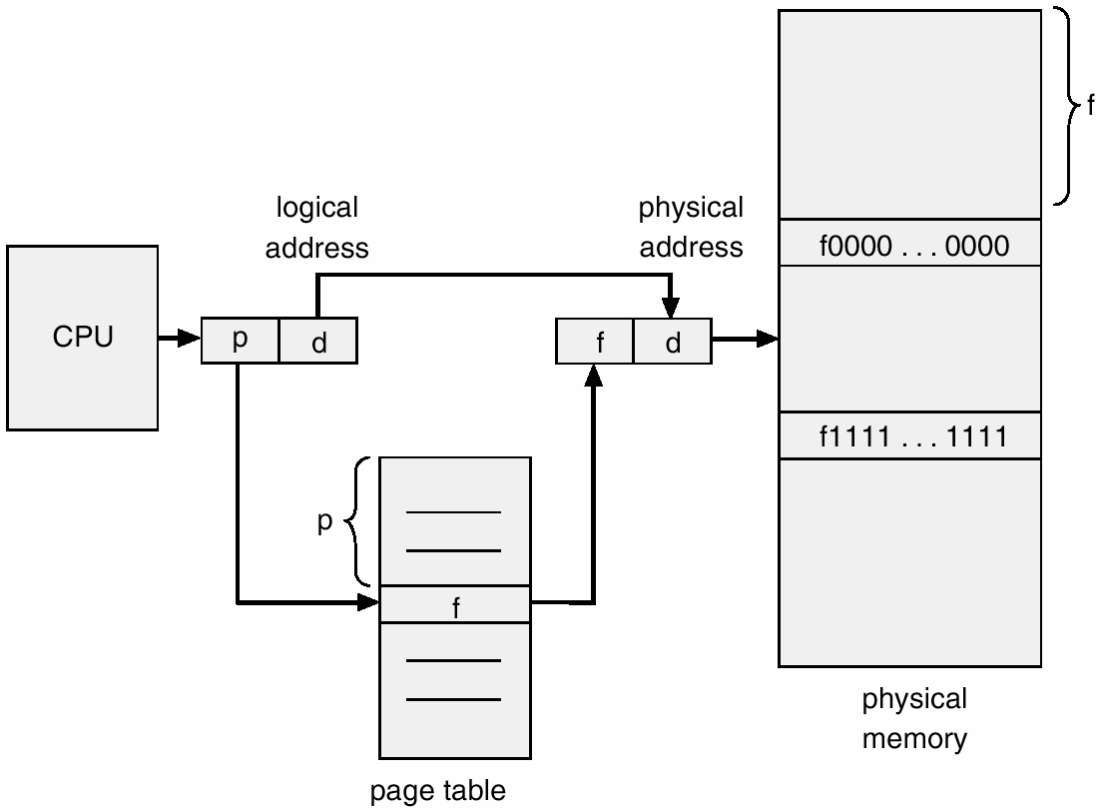
Indirizzo logico 3 per il carattere ‘d’ corrisponde a fisico 23

Indirizzo logico 11 per il carattere ‘l’ corrisponde a fisico 7

Come facciamo a tradurre indirizzi logici in fisici? Abbiamo bisogno di hardware apposito!

MMU e paginazione

- L'indirizzo logico generato dalla CPU viene suddiviso in
  - **numero di pagina** usato come indice nella tabella delle pagine che contiene il frame associato
  - **scostamento di pagina** all'interno della pagina e permette di definire l'indirizzo fisico in memoria
- La dimensione di una pagina è sempre una potenza di 2 per semplificare la traduzione degli indirizzi

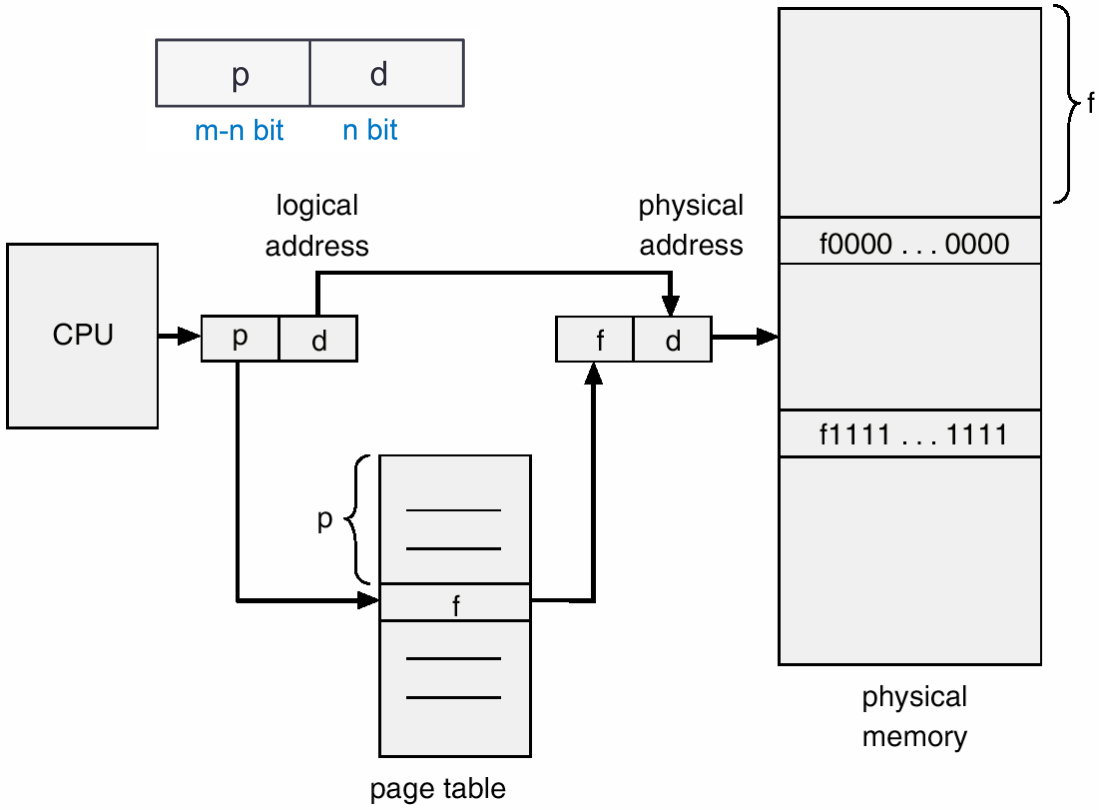


La tabella delle pagine è in memoria principale

MMU e paginazione

- L'indirizzo logico generato dalla CPU viene suddiviso in
  - **numero di pagina** usato come indice nella tabella delle pagine che contiene il frame associato
  - **scostamento di pagina** all'interno della pagina e permette di definire l'indirizzo fisico in memoria
- La dimensione di una pagina è sempre una potenza di 2 per semplificare la traduzione degli indirizzi
- Attenzione! La tabella delle pagine risiede in memoria RAM
  - si deve fare un accesso alla RAM per recuperare il frame
  - oppure hardware apposito che limita gli accessi alla RAM

Se lo spazio di indirizzamento è 2^m pagine e ogni pagina occupa 2^n allora **m-n** bit indicano la pagina, **n** bit lo scostamento
Per esempio indirizzo a 8 bit e 2 bit per numero pagina
m 8-6= 2 quindi 2^2 pagine 4 pagine
n 2^6 = 64 byte per pagina



Altro supporto hardware

- Vengono usati dei registri associativi detti TLB (translation look-aside buffer) per memorizzare una parte della tabella delle pagine
- Context switch: sono caricati dal dispatcher
- Sono nella MMU o nella CPU, molto veloci e costosi
- Se p nel registro associativo bene, altrimenti si fa un riferimento alla tabella delle pagine in memoria
- Su questo ci sono dei calcoli sui tempi di accesso basati su TLB miss e hit ratio

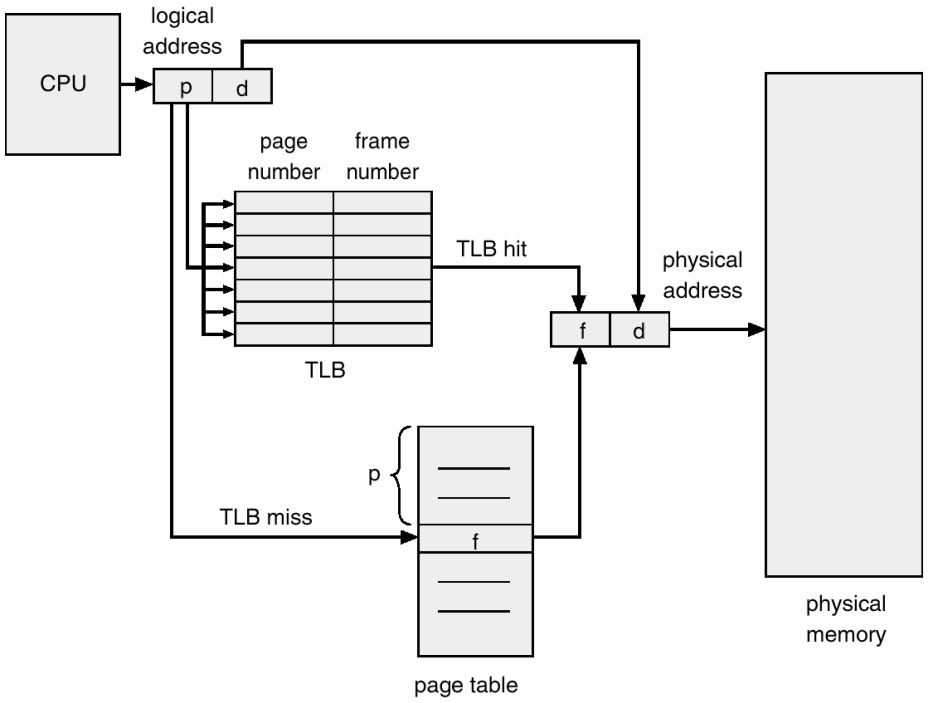
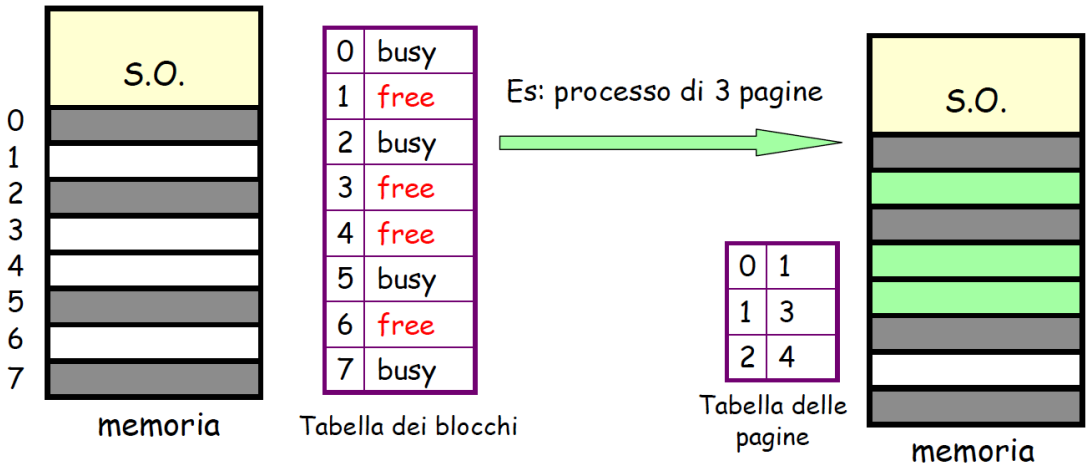


Tabella dei Blocchi

- SO mantiene anche una tabella dei blocchi (frame, blocchi fisici)
- La tabella dei frame contiene un elemento per ogni frame e indica se questo è libero, allocato e (se allocato) a quale pagina di quale processo o processi
- Quando si deve eseguire un processo, SO esamina la sua dimensione espressa in pagine
- SO consulta tabella dei blocchi e crea la tabella delle pagine per il nuovo processo e inizia a caricare il primo blocco di memoria libero per la prima pagina nella tabella delle pagine



Bit di validità

Serve un bit di presenza/validità nella tabella delle pagine associato da ogni elemento

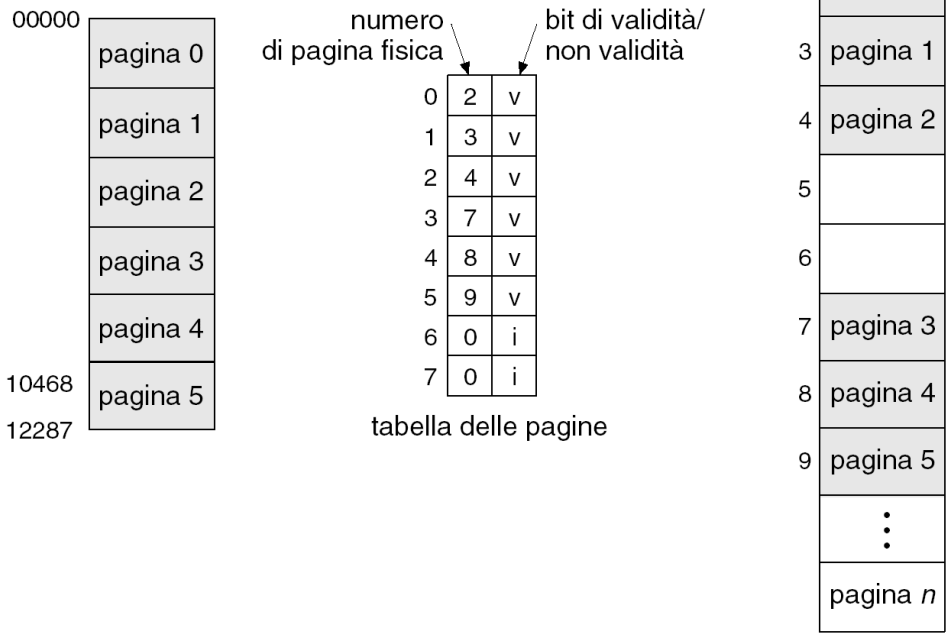
- Valido, bit a 1: indica che il frame è nello spazio di indirizzi (pagina caricata in RAM)
- Non valido, bit a 0: indica che il frame non è nello spazio di indirizzi (pagina non in RAM)

Un bit anche per lettura scrittura per indicare se la pagina è in sola lettura etc.

Altri bits:

Bit usata / non usata

Dirty bit: bit modificata nel caso in cui scarico la pagina dalla memoria



Memoria virtuale

- Abbiamo detto che con la paginazione **non** tutte le pagine del processo vengono caricate in memoria principale
- In memoria secondaria le altre pagine, richiamo Swapping e scheduler breve termine
- Il bit **assente/presente** serve per tenere traccia di quali pagine sono in memoria principale
- In questo modo si realizza il concetto di **memoria virtuale**
  - Uso la memoria secondaria come estensione della memoria principale
  - I processi credono di avere molta memoria a disposizione
  - Si tengono in esecuzione più processi

Memoria virtuale e paginazione su richiesta

Lo spazio logico degli indirizzi è più grande dello spazio fisico

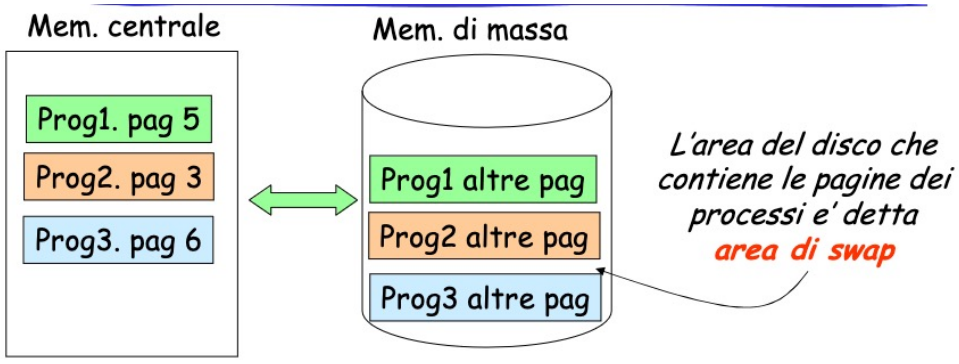
Un maggior numeo di processi concorre alla CPU

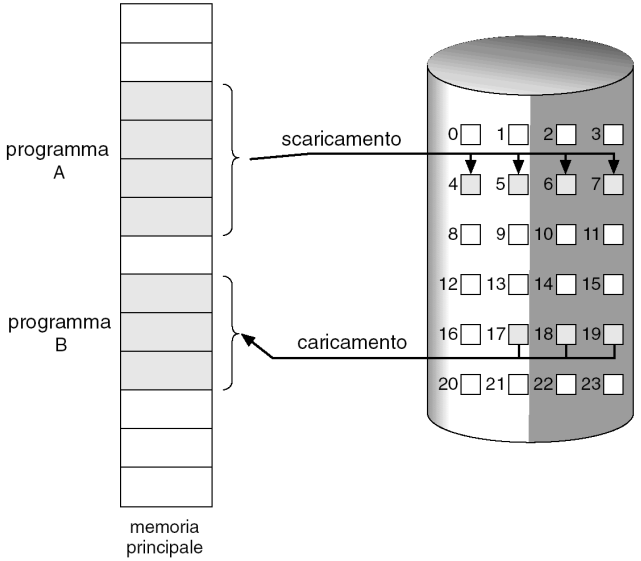
Paginazione su richiesta: È una tecnica analoga all'avvicendamento dei processi (swapping) ma vale solo per le pagine e non per intero processo

Quali pagine carico in memoria? Cosa succede se devo accedere ad una pagina non in memoria? Località spaziale e temporale dei programmi…

Bit assente/presente nella tabella delle pagine

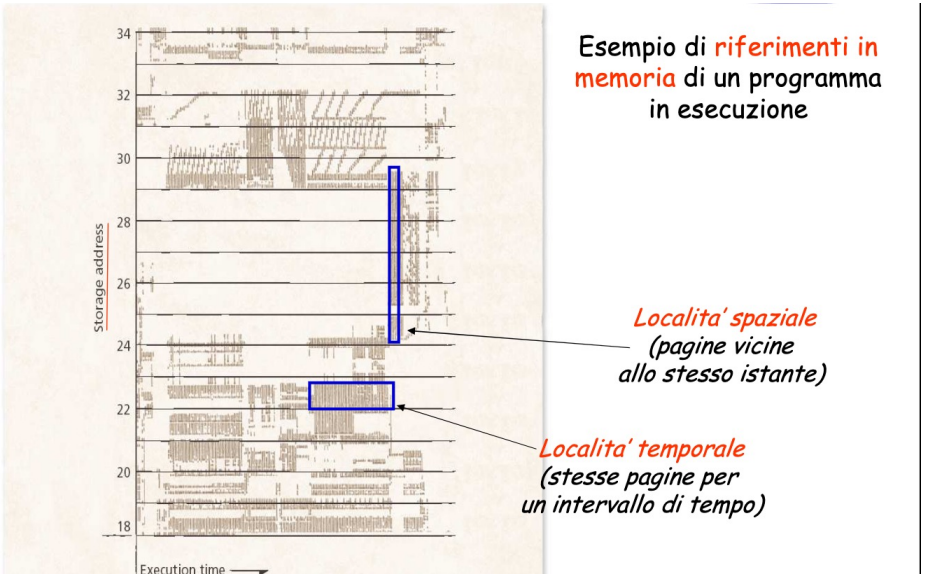
In fase di traduzione indirizzi, se bit a 0 … **page fault!**





Località spaziale e temporale

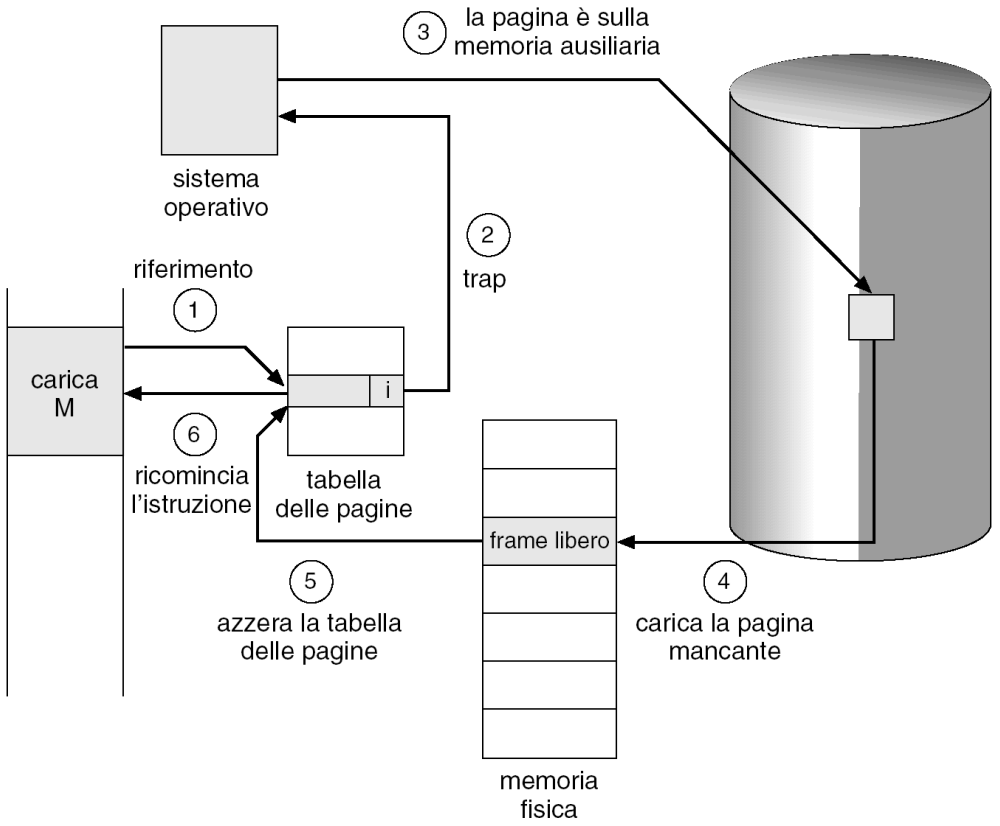
- Località spaziale: processi che usano pagine vicine
- Località temporale: processi che usano le stesse pagine per un intervallo di tempo



Gestione Page Fault

Si esamina una tabella del processo per decidere:

1. pagina non valida: bit validità a 0, il processo viene terminato da SO che esegue una sorta di abort o simili
2. pagina non in memoria: bit presente a 0, si procede così
   a) Si cerca un frame libero sufficientemnte grande
   b) Si carica dal disco la pagina nel frame individuato
   c) Si modificano la tabella interna al processo (con i vari bit) e la tabella dei blocchi del SO
   d) Si riavvia l'istruzione interrotta

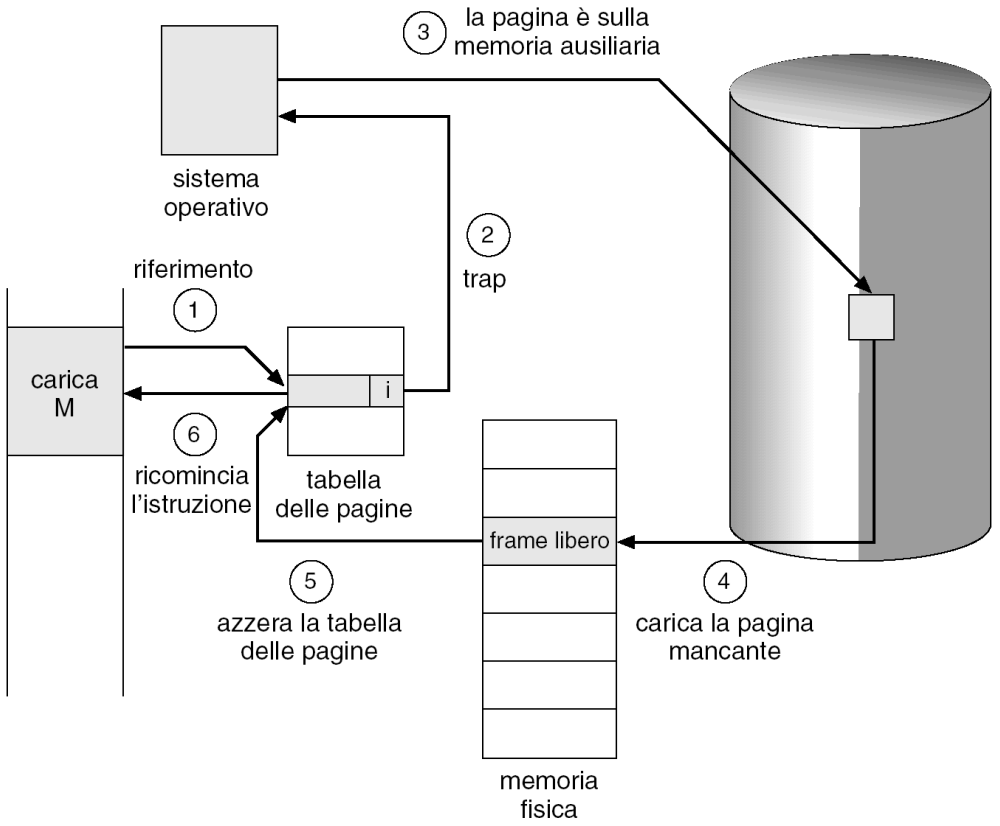


[Spiegazione chiara e semplice su internet]

Gestione Page Fault

Si esamina una tabella del processo per decidere:

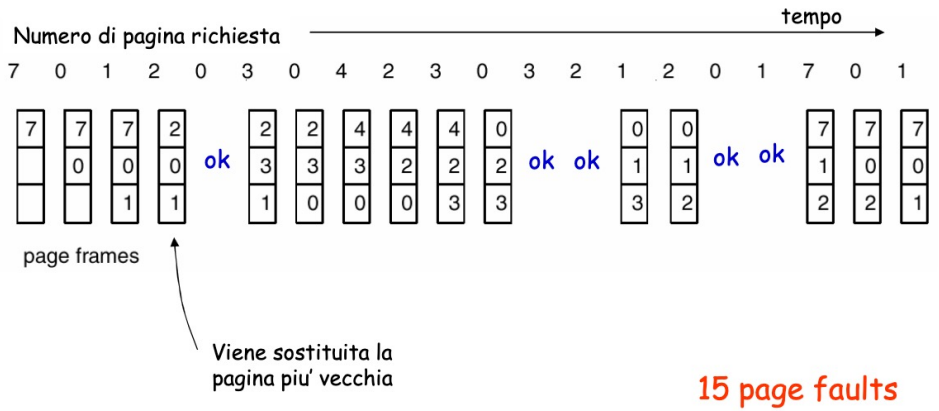
1. pagina non valida: bit validità a 0, il processo viene terminato da SO che esegue una sorta di abort o simili
2. pagina non in memoria: bit presente a 0, si procede così
   a) Si cerca un frame libero sufficientemnte grande
   b) Si carica dal disco la pagina nel frame individuato
   c) Si modificano la tabella interna al processo (con i vari bit) e la tabella dei blocchi del SO
   d) Si riavvia l'istruzione interrotta



E se non ci sono frame liberi? Ovvio, servono algoritmi per sostituire le pagine ;)

FIFO

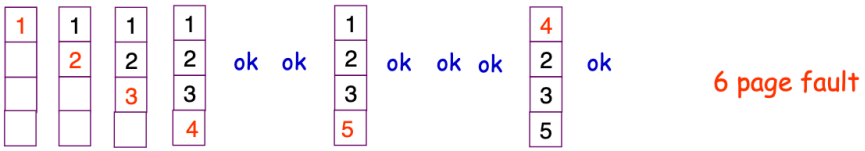
- First in First Out
  - Sostituisco la pagina in memoria da più tempo
  - Facile da implementare ma…
  - … rischio di sostituire pagine molto utilizzate e per questo presenti in memoria da molto tempo
  - Soffre anomali di Belady, più frame in memoria più page fault



15 page faults

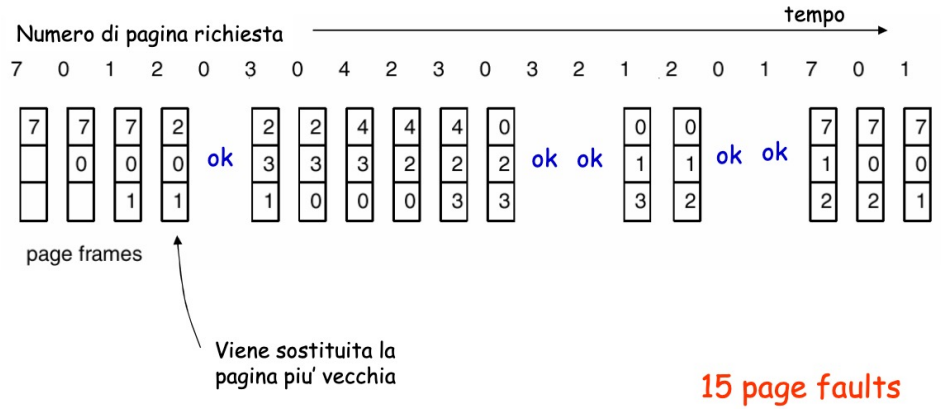
OPT

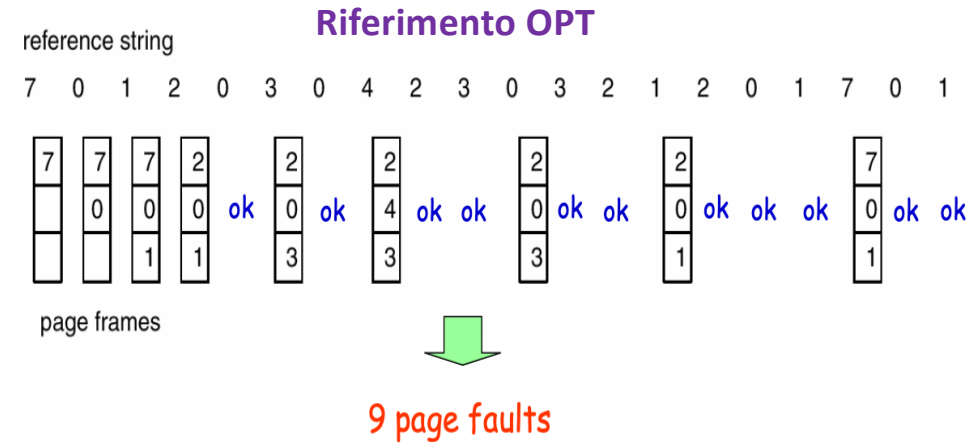
- Algoritmo ottimo
- Viene sostituita la pagina che non verrà usata per il periodo di tempi più lungo
- Minimizza il numero di page fault
- Come conosco questa pagina?
  - È solo un algoritmo teorico, si impiega per misurare le prestazioni comparitive degli algoritmi con valenza pratica



FIFO vs OPT

- FIFO/OPT
- 15/9 1.66 66% più lento di OPT





LRU (Least Recently Used)

Scarica le pagine non usate da più tempo

Meglio di FIFO ma maggiore overhead

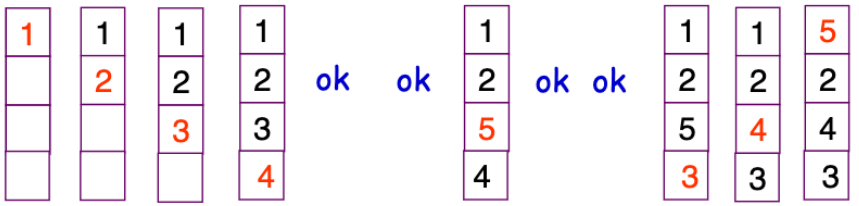
Favorisce I processi che soddisfano il principio di località temporale

8/6 = 1.33 33% più lento di OPT

12/9 = 1.33 33% più lento di OPT

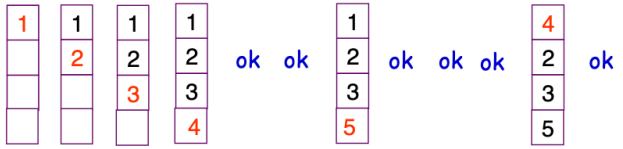
- Esempio: 4 frame con stringa di riferimenti

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

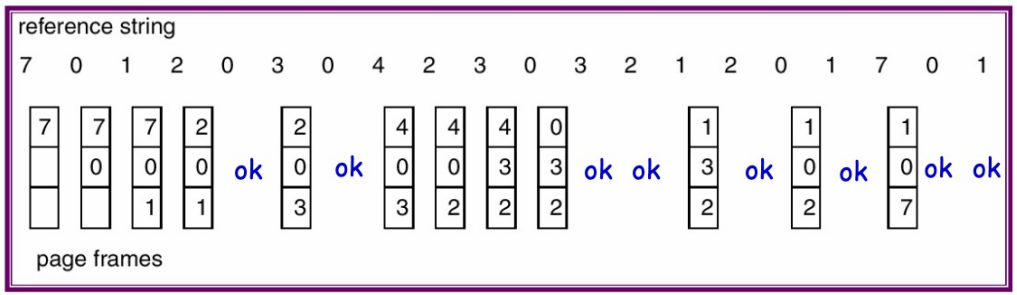


8 page fault

Riferimento OPT

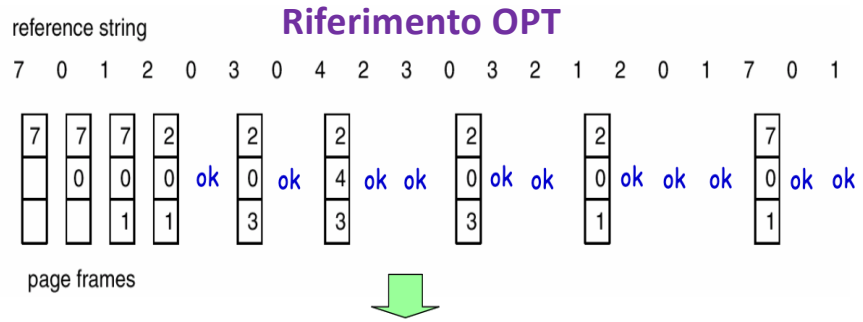


6 page fault



12fault

Riferimento OPT



9 page faults

LFU Least Frequently Used

Viene sostituita la pagina meno frequentemente utilizzata

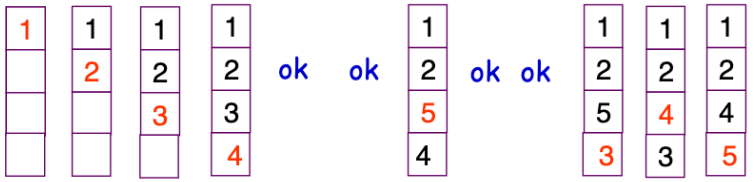
Una pagina poco referenziata lo sarà anche in futuro

Più efficiente di FIFO

Rischio di sostituire pagine da poco presenti in memoria e per questo poco referenziate (non tiene conto della località temporale)

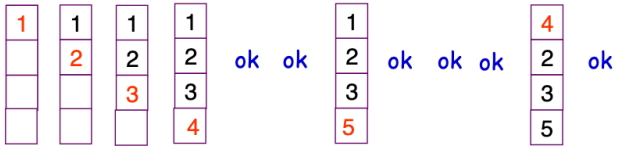
LFU/OPT 8/6 1.33 33% più lento di OPT

- Esempio: 4 frame con stringa di riferimenti
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



8 page fault

Riferimento OPT



6 page fault

NRU – Not recently used

Anche conosciuto come **algoritmo seconda chance** o **algoritmo dell'orologio**

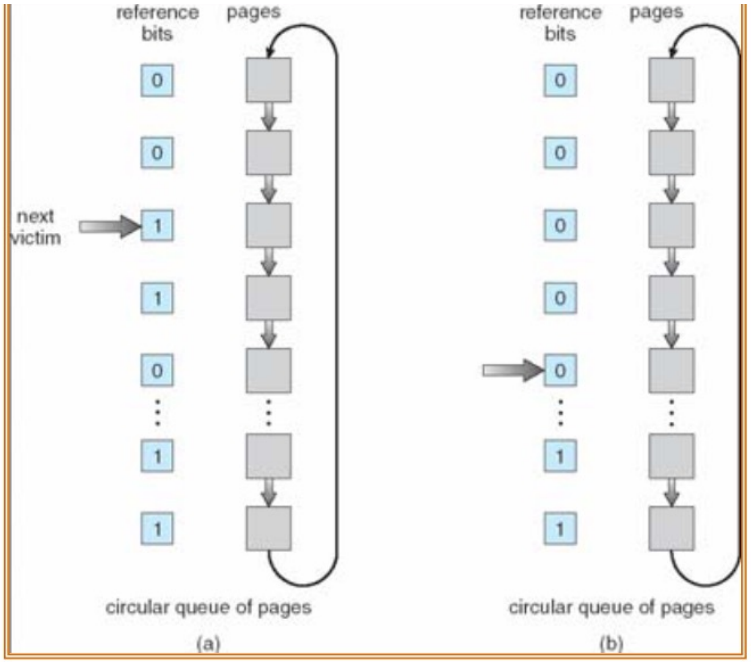
Algoritmo di base: FIFO

1. È necessario un bit di riferimento
2. Quando la pagina riceve una seconda chance, il bit di riferimento viene azzerato ed il tempo di arrivo viene aggiornato al tempo attuale
3. Se la pagina da rimpiazzare (in ordine di clock) ha il bit di riferimento a 0, viene effettivamente rimpiazzata

Se la pagina da rimpiazzare (in ordine di clock) ha il bit di riferimento a 1, allora:

1. Si pone il bit di riferimento a 0
2. Si lascia la pagina in memoria
3. Si rimpiazza la pagina successiva (in ordine di clock), in base alle stesse regole

Libro pg. 273 non chiaro



NRU Not recently used migliorato

Si considera la coppia (bit_riferimento, bit_modifica)
Si ottengono quattro classi:

1. (0,0), non recentemente usata né modificata− rappresenta la migliore scelta per la sostituzione
2. (0,1), non usata recentemente, ma modificata− deve essere salvata su memoria di massa
3. (1,0), usata recentemente, ma non modificata− probabilmente verrà di nuovo acceduta a breve
4. (1,1), usata recentemente e modificata− verrà probabilmente acceduta di nuovo in breve termine e deve essere salvata su memoria di massa prima di essere sostituita

Si sostituisce la prima pagina che si trova nella classe minima non vuota
La coda circolare deve essere scandita più volte prima di reperire la pagina da sostituire

Libro pg. 273 non chiaro

Algoritmi con conteggio

Si mantiene un contatore del numero di riferimenti che
sono stati fatti a ciascuna pagina

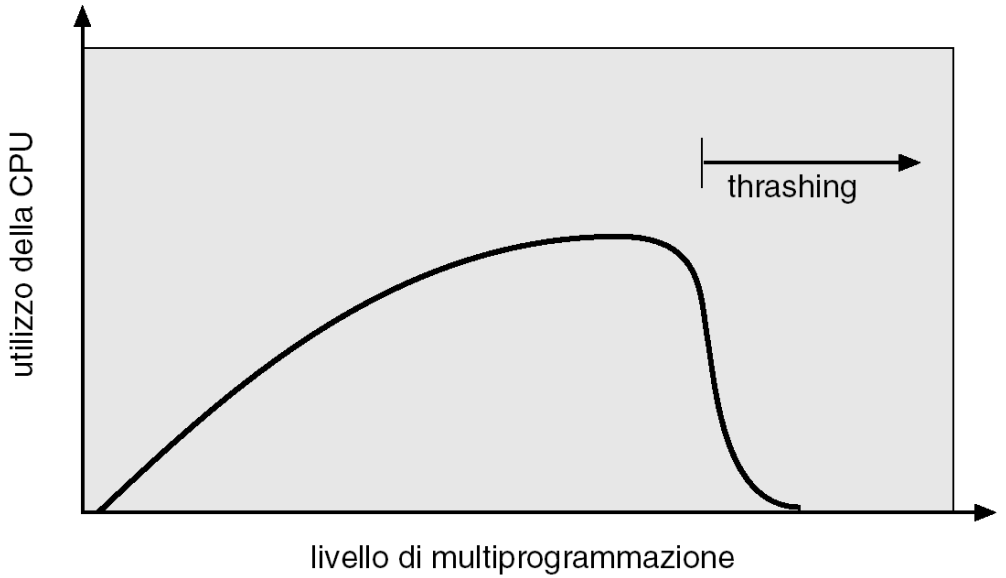
Algoritmo LFU (Least Frequently Used): si rimpiazza la pagina col valore più basso del contatore

Algoritmo MFU (Most Frequently Used): è basato sulla assunzione che la pagina col valore più basso del contatore è stata spostata recentemente in memoria e quindi deve ancora essere impiegata

L'implementazione è molto costosa; le prestazioni (rispetto all'algoritmo ottimo) scadenti

Trashing

- Quante pagine carico quando il processo viene lanciato? Si carica un numero minimo di frame
  - Dipende da architettura (instruction set)
  - Numero minimo che permette esecuzione istruzione senza generare eccezione pagina mancante
  - Per esempio, istruzione macchina MOV ha bisogno di 6 frame
- Trashing: un processo spende più tempo nella paginazione che nell'esecuzione
- Per evitare il trashing bisogna assicurare che un processo abbia sempre il numero di frame necessari

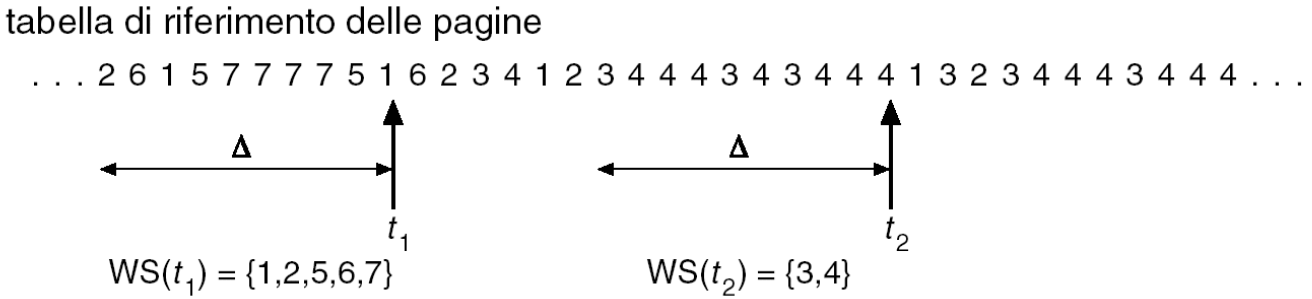


Working set

- Cerca di definire un modello di località temporale
- Il working set di un processo W(t, **w**) è l'insieme delle pagine referenziate dal processo nell'intervallo di tempo t-w, t=> **Δ**
- **Δ** intervallo di osservazione fissato
- T istante di osservazione

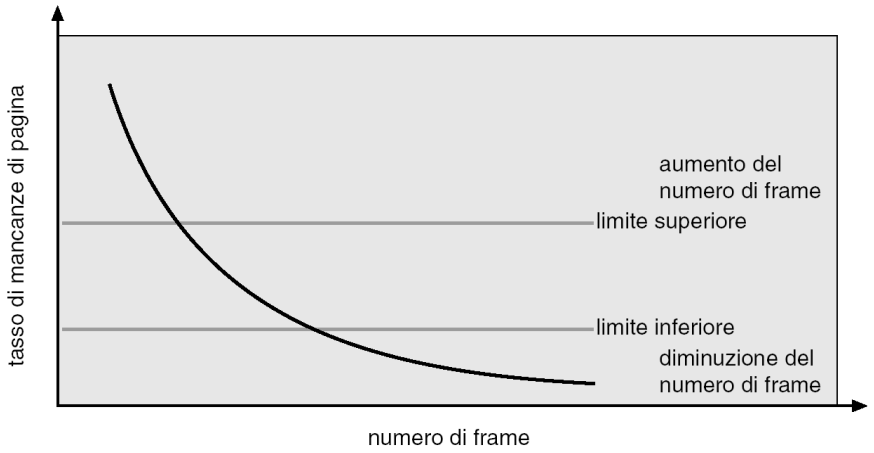
Δ troppo piccolo: avvicendamento di pagine utili (trashing) perché non include intera lotcalità e rischia di aumentare il numero di page fault (ovvero le pagine non sono caricate in memoria principale)

Δ troppo grande rischio di tenere in memoria pagine poco utilizzate, si sovrappongono località diverse… tengono in memoria principale pagine non necessarie



Variante… frequenza mancanze di pagina

- Metodo più diretto per controllare il trashing
- Stabilire un tasso accettabile per la frequenza di mancanze di pagina
- Se troppo basso, si deallocano alcuni frame del processo
- Se troppo alto, si allocano nuovi frame al processo



Altri metodi di paginazione

- Quando lo spazio degli indirizzi logici è troppo grande lo schema descritto diventa inefficiente
- Esempio indirizzamento a 32 o 64 bit
- Altre soluzioni:
  - Paginazione gerarchica o a livelli
  - Tabella di pagine di tipo hash
  - Tabella delle pagine invertita

Allocazione non contigua

Segmentazione

Segmentazione

Non divido la memoria in blocchi tutti uguali come le pagine ma in blocchi di dimensioni tra loro diverse

Rispecchia meglio la visione del programma da parte dell'utente

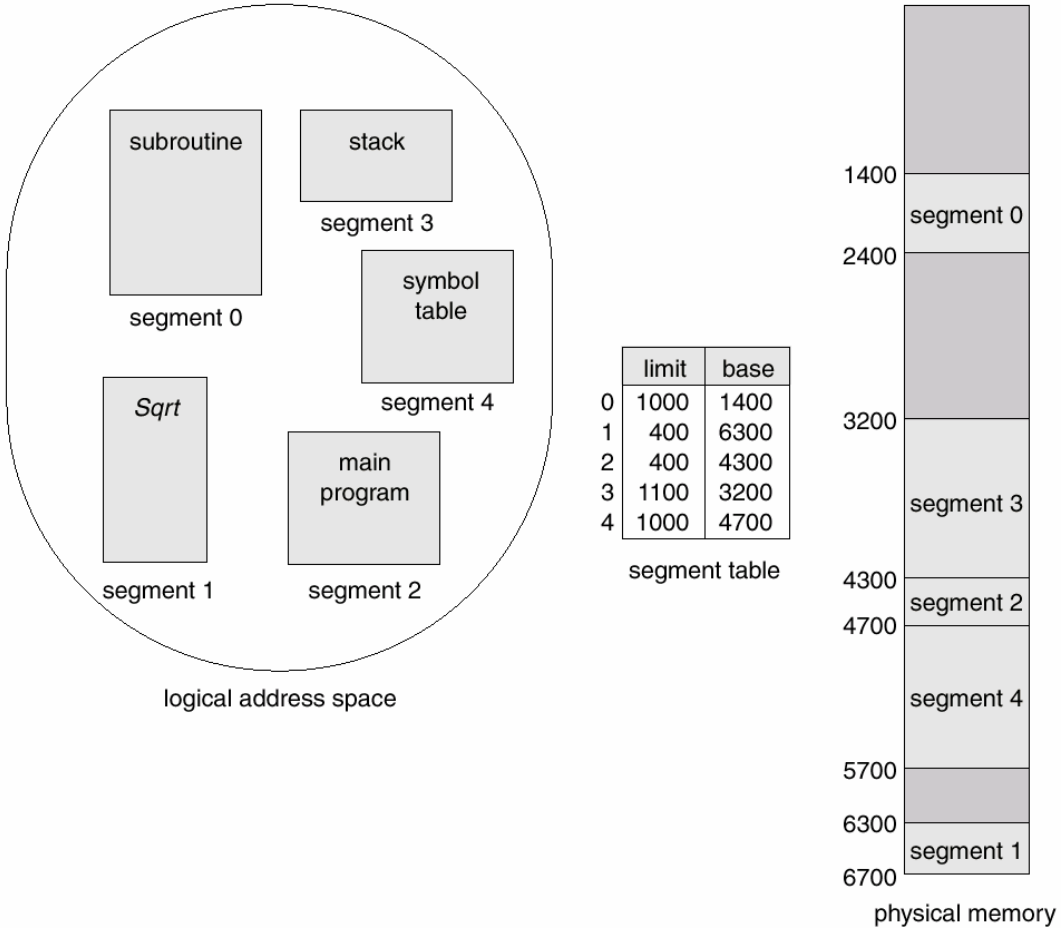
Ogni entità viene caricata dal loader separatamente in aree di dimensione variabile dette **segmenti**

I **segmenti** hanno dimensione variabili e sono allocati in modo non contiguo nella memoria

Lo spazio di indirizzamento logico è un insieme di segmenti di dimensione variabile

Ogni indirizzo logico consiste di due parti

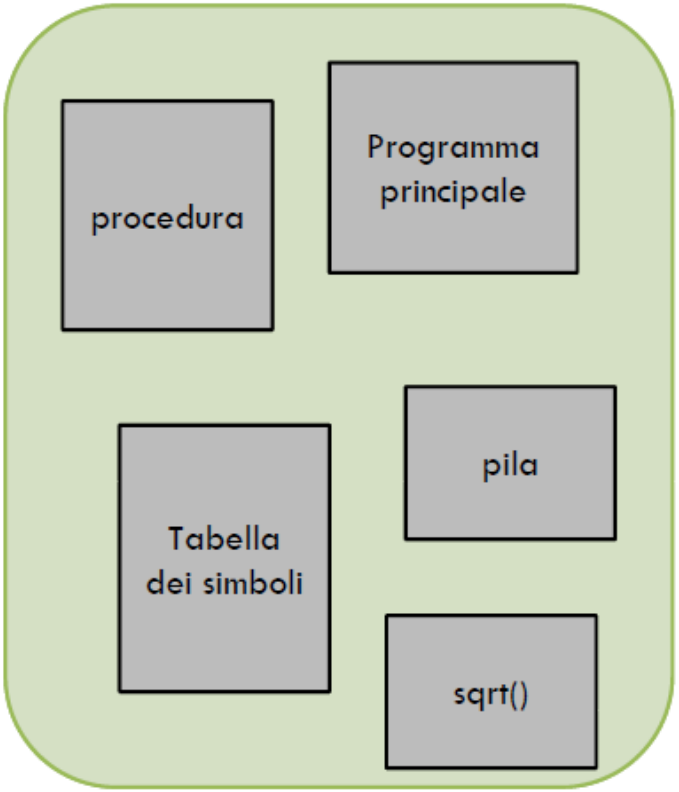
Numero del segmento + offset nel segmento



Visione memoria del processo

Per esempio, un compilatore C crea i segmenti:

- Il codice
- Variabili globali
- Heap, da cui si alloca la memoria
- Variabili locali statiche di ogni funzione o procedura
- Librerie standard del C



Architettura segmentazione

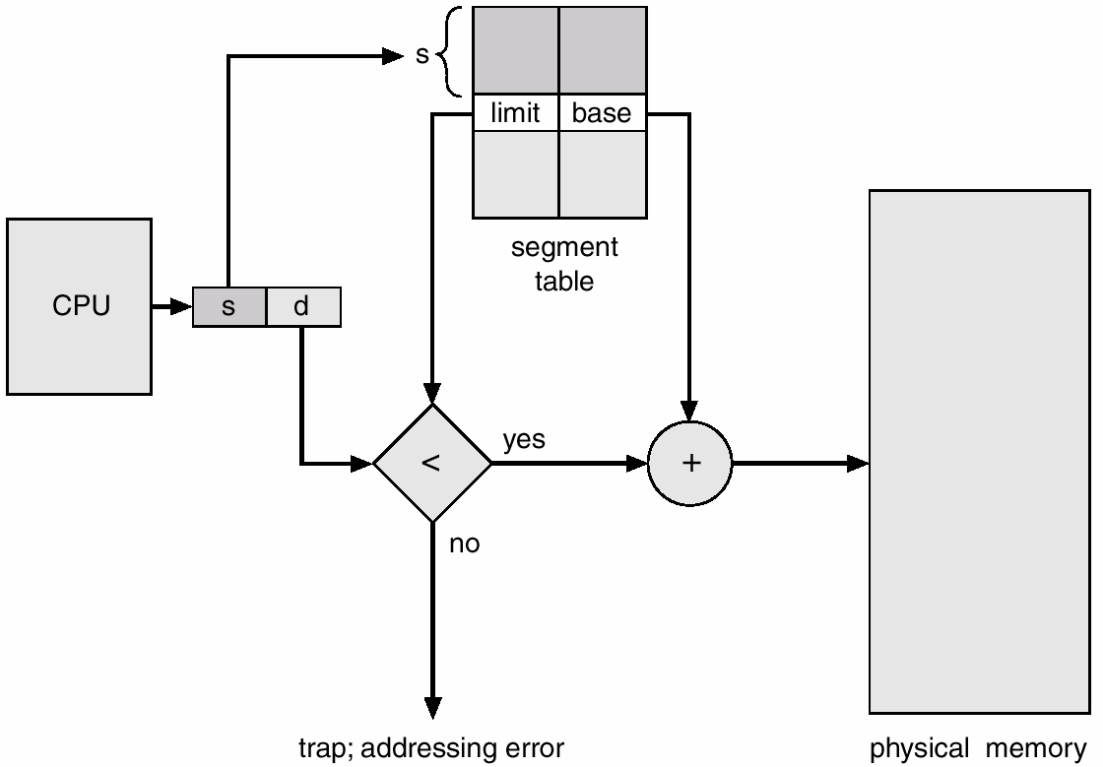
Tabella dei segmenti mappa indirizzi fisici che contiene

1. Base: indirizzo fisico del segmento
2. Limite: specifica la lunghezza del segmento

Dato un indirizzo logico <s,d> num segmento e scostamento

1. Si usa s come indice della tabella per ricavaare l'indirizzo base
2. Si verifca che d< limite (protezione)
3. Indirizzo fisico: base+d

Ogni indirizzo logico consiste di due parti
Numero del segmento + offset nel segmento



Altro esempio

Ogni processo ha la sua tabella dei segmenti

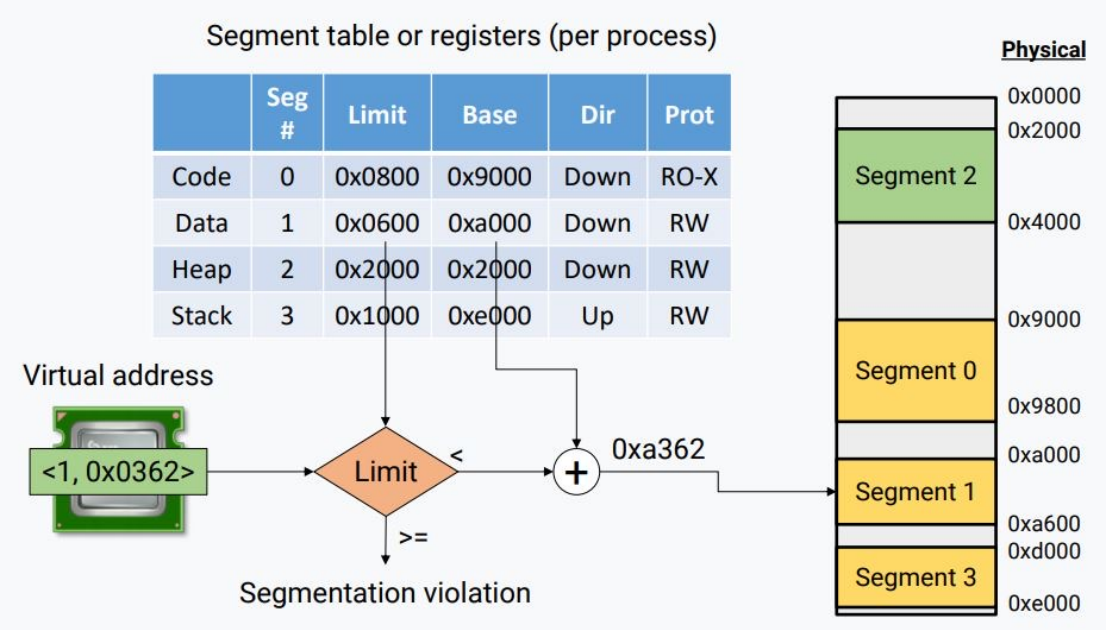
Base: inizio indirizzo fisico

Limit: lunghezza segmento

Stack e heap possono contrarsi indipendentemente

MA…

Frammentazione esterna! Come allocazione contigua a partizioni variabili



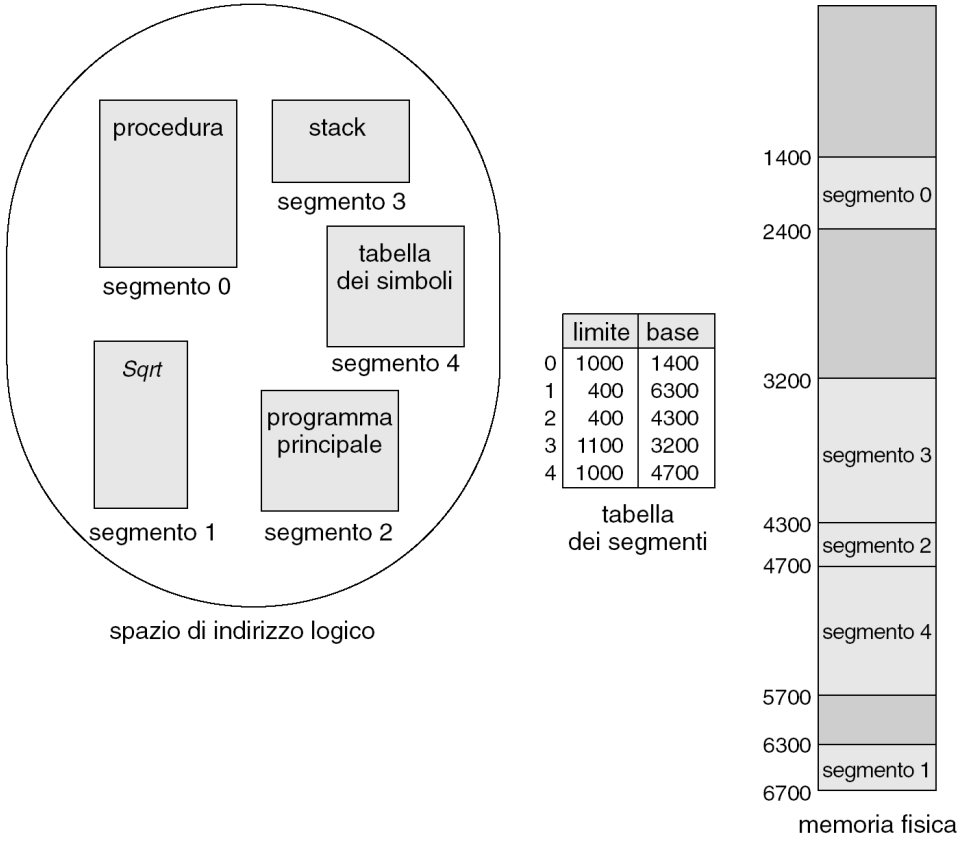
Indirizzo logico non valido - esempio

Indirizzo logico 2,53?

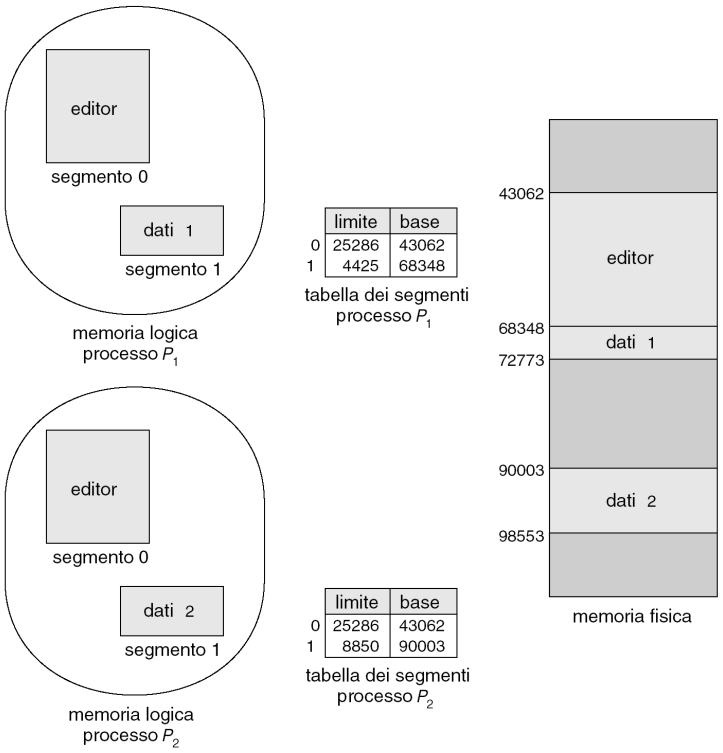
4353

Indirizzo logico 0, 1222

eccezione



Condivisione dei segmenti

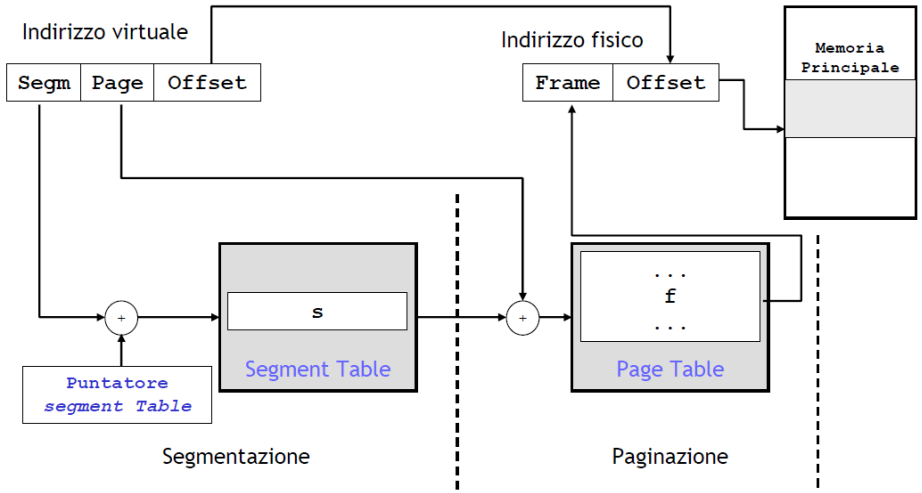


- Come per la paginazione, posso condividere il codice puntando allo stesso numero del segmento
- Protezione: analoga alla paginazione
- I segmenti variano di dimensione quindi l'allocazione è dinamica
- Posso combinare la paginazione e la segmentazione

Segmentazione e paginazione

- La segmentazione soffre, come allocazione contigua a partizioni variabili, del problema della frammentazione esterna
- IDEA: si possono paginare i segmenti mantenendo i vantaggi della segmentazione
- Queste due tecniche insieme vengono usate dalla maggior parte dei sistemi operativi
  - Diminuisco frammentazione esterna per paginazione, interna con segmentazione
  - Facile condividere codice e proteggere la memoria con segmentazione
- La tabella dei segmenti non contiene indirizzo segmento ma indirizzo della tabella delle pagine per quel segmento

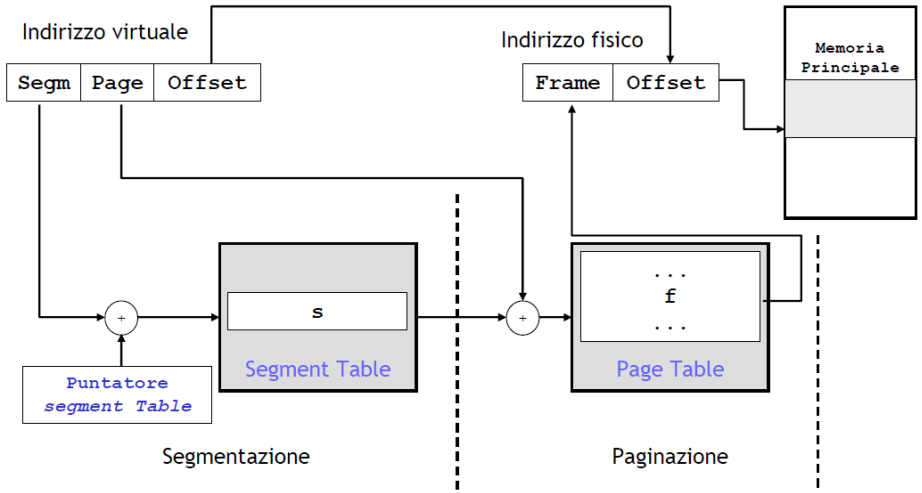
Segmentazione con paginazione



Quando un programma tenta di accedere a un indirizzo di memoria, specifica una tripla composta da segmento, pagina e offset:

1. **Segmento**: Identifica il segmento di memoria a cui si sta cercando di accedere.
2. **Pagina**: Dopo aver selezionato il segmento, la pagina indica quale pagina del segmento è necessario accedere.
3. **Offset**: Infine, l'offset è utilizzato per localizzare l'esatta posizione di memoria all'interno della pagina.

Segmentazione con paginazione



l sistema operativo consulta prima la **tabella dei segmenti** utilizzando l'identificatore del segmento per trovare l'indirizzo base della tabella delle pagine per quel segmento.

Successivamente, si accede alla **tabella delle pagine** del segmento, utilizzando l'identificatore della pagina per trovare l'indirizzo fisico della pagina nella memoria.

Infine, l'offset viene aggiunto a quest'indirizzo per ottenere l'indirizzo fisico esatto all'interno della memoria.

Gestione memoria ed efficienza programmi

- In alcuni casi anche un programma scritto in modo accurato può ridurre il numero di mancanze di pagina
- Si supponga di avere pagine da 128 parole
- Il seguente codice inizializza a 0 gli elementi di una matrice da 128x128

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

- In pagine da 128 parole ogni riga della matrice occupa una pagina
- Questo codice azzera una parola per pagina e poi passa alla successiva
- Se i sistema operativo assegna meno di 128 frame a tutto il programma si hanno 128x128=16384 assenze di pagina

Comandi mac

vm_stat Il comando vm_stat ti fornisce statistiche sulla memoria virtuale del sistema. Questo comando mostra informazioni sulla paginazione, sull'uso della memoria fisica e virtuale, e altre statistiche correlate alla memoria.

Pagesize

vmmap <pid_del_processo> visualizzerà una mappa della memoria virtuale del processo specifico, inclusi segmenti come stack, heap, librerie condivise e altre regioni di memoria allocate dal processo.

Linux comandi per memoria

vmstat

smem

pmap <PID_del_processo>

top

free -h

cat /proc/meminfo

cat /proc/*pid*/maps

Appendice A

CODICE RIENTRANTE

Codice rientrante

- Il codice è progettato in modo che una singola copia del codice in memoria possa essere condivisa ed eseguita contemporaneamente e senza risultati inaspettati da utenti multipli o processi separate
- Vitale per sistemi multitasking e thread-safe
- **Esempio**: un compilatore viene normalmente usato da più processi in un sistema multiutente. Effettuare una copia per ogni richiesta causerebbe una occupazione di spazio eccessiva. Le sue istruzioni di decodifica restano sempre le stesse, per cui in memoria ne viene tenuta una sola copia. Le tabelle di traduzione invece, che variano per ogni processo e non possono essere condivise, saranno in numero pari ai sorgenti da compilare.

Codice rientrante

Deve soddisfare i seguenti requisiti

1. Nessuna porzione del codice possa essere alterata durante l'esecuzione (codice *non automodificante*);
2. Il codice non deve richiamare nessuna routine che non sia a sua volta rientrante. Per esempio molte implementazioni delle funzioni malloc() e free() in C non lo sono (stesso discorso per *new* e *dispose* in Pascal) e non devono essere utilizzate. In caso sia necessario allocare memoria si potrebbero utilizzare le API di sistema (dopo averne consultato la documentazione per avere la certezza che siano rientranti), che, potendo in teoria essere chiamate in qualsiasi momento e in qualsiasi situazione (perlomeno in un sistema pienamente multitasking) offrono maggiori garanzie.
3. Il codice deve usare, se necessarie, solo variabili temporanee allocate sullo stack.
4. Il codice non deve modificare né variabili globali né aree di memoria condivisa né impiegare variabili locali statiche.

Codice rientrante allora

- Non modifica o dipende da variabili globali o statiche senza adeguata sincronizzazione, ma invece opera su dati locali o passati come argomenti.
- Accede a risorse condivise (come file o connessioni al database) in modo sicuro e controllato, assicurando che l'accesso sia atomico o che lo stato possa essere ripristinato dopo l'interruzione.
- L'esecuzione dell'operazione può essere ripetuta senza causare effetti collaterali indesiderati, garantendo lo stesso risultato ogni volta.
- Non si affida a dati specifici nello stack di chiamata, ma utilizza il proprio stack locale o parametri per mantenere lo stato.

Esempio

In questo esempio, la funzione incrementCounter modifica una variabile globale counter. Se questa funzione venisse chiamata contemporaneamente da più thread, potrebbero verificarsi delle condizioni di gara perché l'incremento non è un'operazione atomica (potrebbe essere interrotta tra la lettura del valore di counter e la scrittura del suo valore incrementato).

```
int counter = 0; // Variabile globale

void incrementCounter() {
    counter++;
}
```

CODICE NON RIENTRANTE

In questa versione rientrante della funzione, counter è passato come un puntatore a incrementCounter. Se ogni thread usa la propria variabile di contatore locale o si assicura che gli accessi alla variabile condivisa siano adeguatamente sincronizzati (per esempio, usando mutex), la funzione può essere chiamata in modo sicuro da più thread. Non ci sono variabili globali o statiche coinvolte, quindi non c'è rischio di condizioni di gara.

```
void incrementCounter(int *counter) {
    (*counter)++;
}
```

CODICE RIENTRANTE

Esempio

Questo codice non è rientrante per alcune ragioni:

La funzione f modifica una variabile globale g_var. Se f venisse chiamata da più thread contemporaneamente, potrebbero verificarsi condizioni di gara.

La funzione g chiama f, la quale ha un effetto collaterale sullo stato globale cambiando g_var. Questo contribuisce a rendere anche g non rientrante.

```c
#include <stdio.h>

int g_var = 1; // Global variable

int f() {
    g_var = g_var + 2; // Modifies the global variable
    return g_var;
}

int g() {
    return f() + 2; // Calls f() and further modifies g_var indirectly
}

int main() {
    int result = g();
    printf("Result: %d\n", result);
    printf("Global Variable g_var: %d\n", g_var);
    return 0;
}
```

CODICE NON RIENTRANTE

Esempio

Per creare una versione rientrante del codice dato, dobbiamo assicurarci che le funzioni non modifichino nessuna variabile globale e che ogni stato sia passato come parametro

```c
#include <stdio.h>

int g_var = 1; // Global variable

int f() {
    g_var = g_var + 2; // Modifies the global variable
    return g_var;
}

int g() {
    return f() + 2; // Calls f() and further modifies g_var indirectly
}

int main() {
    int result = g();
    printf("Result: %d\n", result);
    printf("Global Variable g_var: %d\n", g_var);
    return 0;
}
```

CODICE NON RIENTRANTE

In questo codice rientrante, sia f che g operano solo su parametri passati e non su variabili globali. Questo assicura che le funzioni possano essere chiamate in modo sicuro anche in contesti concorrenti come thread multipli, perché non c'è alcun stato condiviso che viene modificato.

```c
#include <stdio.h>

int f(int var) {
    return var + 2; // Non modifica alcuna variabile globale
}

int g(int var) {
    int local_result = f(var); // Passa la variabile locale a 'f'
    return local_result + 2;   // Aggiunge 2 al risultato locale
}

int main() {
    int g_var = 1; // Variabile locale in 'main'
    int risultato = g(g_var); // Passa la variabile locale a 'g'
    printf("Risultato: %d\n", risultato);
    // 'g_var' rimane invariato; stampa solo per dimostrare che non è cambiata
    printf("Variabile Locale g_var: %d\n", g_var);
    return 0;
}
```

CODICE RIENTRANTE