**LABORATORIO DI PROGRAMMAZIONE 2**
**Corso di laurea in matematica**

**Sistemi Operativi : La gestione della memoria**

Marco Lapegna
Dipartimento di Matematica e Applicazioni
Universita' degli Studi di Napoli Federico II

wpage.unina.it/lapegna

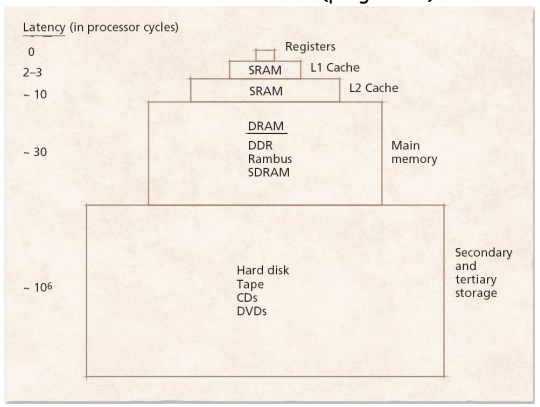


## I livelli di memoria

E' il supporto del calcolatore per
conservare dati e istruzioni (programmi)

*Veloce e cara*

*Lenta e economica*






## La memoria centrale

- La memoria centrale e' costituita da locazioni di memoria ad accesso casuale (RAM) individuate da indirizzi
- La CPU puo' accedere alla memoria centrale in un ordine qualunque
- la memoria centrale e' il piu' basso livello di memoria accessibille direttamente dalla CPU
- volatile: perde il contenuto in mancanza di tensione eletterica
- il Bandwidth e' la quantita' di dati che possono essere trasferiti alla CPU per unita' di tempo



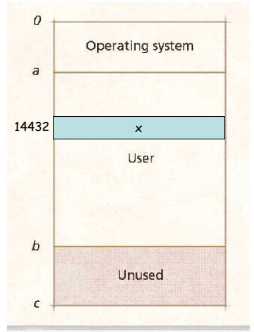
## osservazione

La memoria e' un insieme sequenziale di locazioni con un proprio indirizzo.

Ogni programma risiede in una porzione ben definita della memoria,

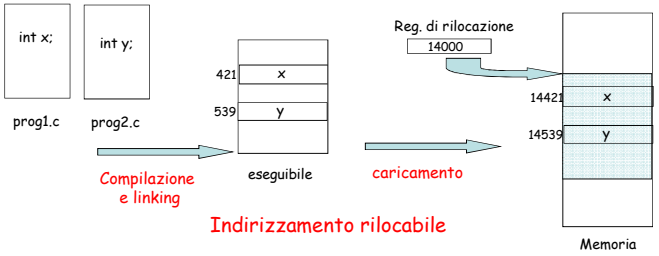
Ogni variabile del programma sara' individuata da un indirizzo

Quando vengono definiti tali indirizzi?



## Generazione degli indirizzi

- L'associazione (***binding***) di istruzioni e dati a indirizzi di memoria può avvenire durante la fase di…
  - **Esecuzione**: se il processo può essere spostato *a run-time* da un segmento di memoria all'altro, il binding viene rimandato fino al momento dell'esecuzione. È necessario un opportuno supporto hardware per mappare gli indirizzi (registri di rilocazione).

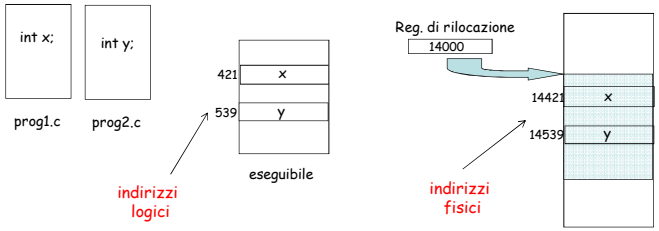



Indirizzamento rilocabile



## Spazio di indirizzi logici e fisici

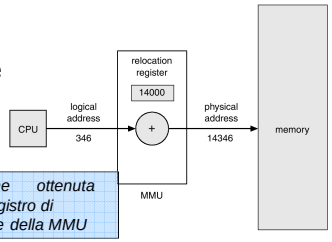
- Il concetto di uno *spazio di indirizzi logici* nettamente distinto dallo *spazio degli indirizzi fisici* è centrale per la gestione della memoria.
  - ***Indirizzi logici*** — generati dalla CPU; chiamati anche *indirizzi virtuali*.
  - ***Indirizzi fisici*** — indirizzi utilizzati per accedere alla memoria.






## Memory–Management Unit (MMU)

- L'operazione di far corrispondere indirizzi logici e fisici nel caso di indirizzamento rilocabile, e' un'operazione frequentissima e viene affidata, per efficienza, ad uno specifico dispositivo hardware (MMU)
- Nello schema MMU, il valore contenuto nel registro di rilocazione viene sommato ad ogni indirizzo generato dai processi utente nel momento stesso in cui l'indirizzo viene inviato alla memoria.
- Il programma utente *ragiona* in termini di indirizzi virtuali, e non è conscio della loro mappatura fisica.



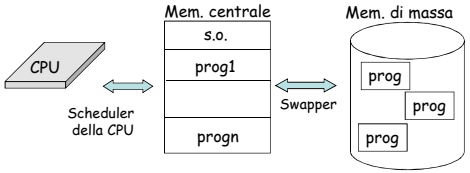

*Rilocazione ottenuta tramite registro di rilocazione della MMU*



## La multiprogrammazione

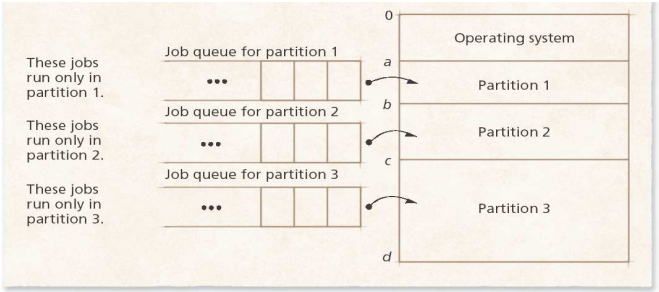
A causa delle continue interruzioni dovute all I/O, un uso efficiente della CPU impone che non solo i programmi in esecuzione, ma anche quelli che stanno per essere eseguiti, devono risiedere in memoria centrale

Lo scheduler di medio livello (swapper) decide quale programma deve competere per l'uso della CPU, e quindi deve risiedere in memoria centrale



## Multiprogrammazione a partizione fissa

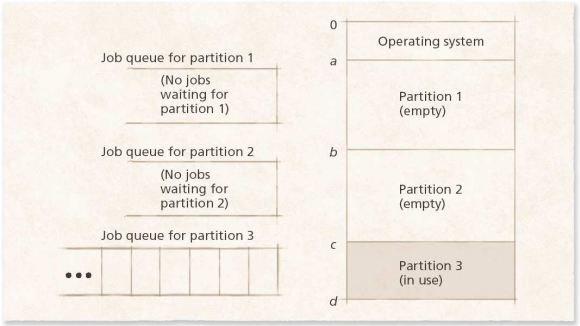
- Ogni processo dispone di un blocco di memoria di dimensione fissata
- Differenti code di job per differenti tipi di programmi
- Piu' registri di limitazione garantiscono la protezione del s.o. e degli altri programmi






## Problema

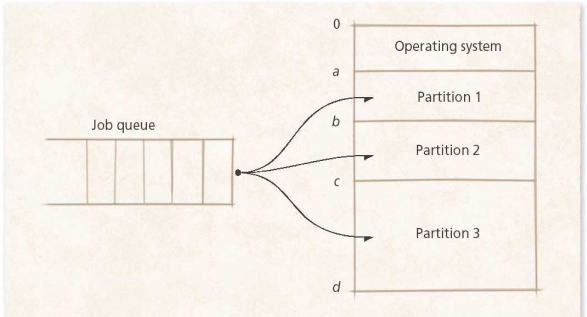
- Spreco di risorse: molte partizioni inutilizzate mentre molti processi aspettano in coda
- Es. 3 partizioni di cui 2 inutilizzate






## Soluzione

Multiprogrammazione con partizione fissa e indirizzamento rilocabile

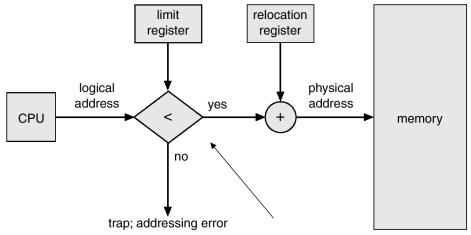



Un job puo' essere sistemato in una qualunque partizione libera di dimensione sufficiente



## Protezione

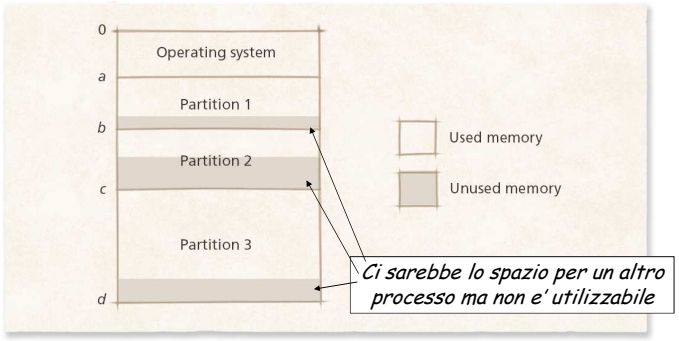
- Si puo' impiegare uno schema basato su registri base e limite
- Il registro di rilocazione (base) contiene il valore del più piccolo indirizzo fisico di memoria allocata al processo;
- il registro limite contiene l'intervallo degli indirizzi logici: ciascun indirizzo logico deve essere inferiore al valore del registro limite.




*A carico del MMU*

## Problema

I processi non fanno uso interamente della partizione a loro assegnata




Frammentazione interna



## Soluzione

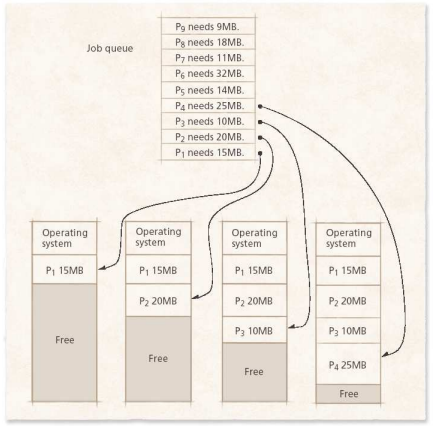
Partizioni fisse:

Proc grandi per la partizione

Proc piccoli per la partizione

Frammentazione interna

partizioni variabili

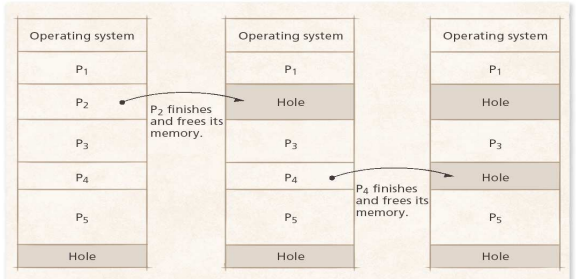





## Multiprogrammazione a partizioni variabili

Con la multiprogrammazione a partizioni variabili si alloca ad un processo uno spazio della dimensione del processo stesso

non c'e' il problema della frammentazione interna, MA.....

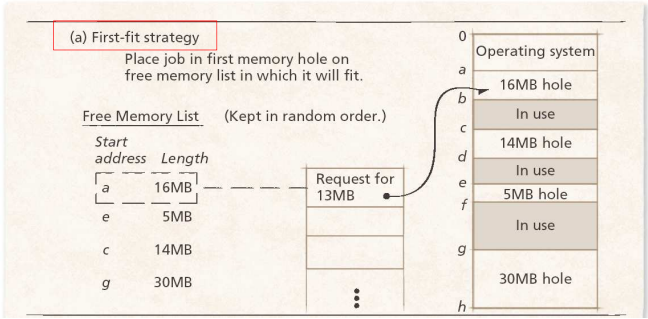



Quando i processi finiscono si creano dei buchi (holes) in memoria
FRAMMENTAZIONE ESTERNA



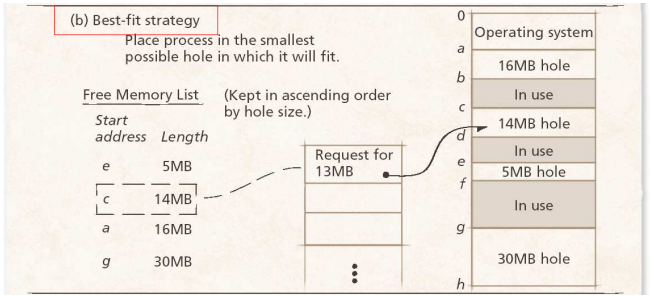
## Riallocazione della memoria

- **First-fit**: ai nuovi processi viene allocato il **primo buco** grande abbastanza.
  - Facile da implementare e con basso overhead



## Riallocazione della memoria

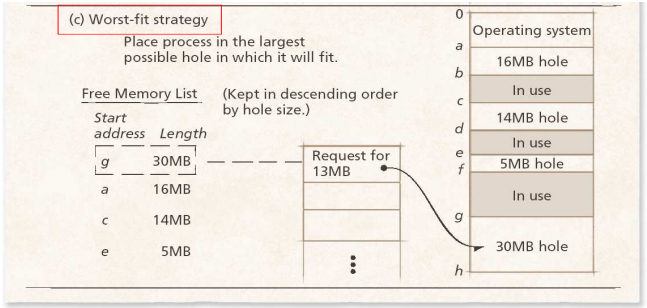
- **Best-fit**: ai nuovi processi viene allocato il buco **più piccolo** capace di contenere il processo.
  - È necessario scandire tutta la lista dei buchi (maggiore overhead)
  - Si produce il più piccolo buco residuo.






## Riallocazione della memoria

- **Worst-fit**: ai nuovi processi viene allocato il buco **più grande**.
  - È ancora necessario ricercare in tutta la lista.
  - Si produce il più grande buco residuo (utile per successive allocazioni)






## problema

I meccanismi di riallocazione
- First fit
- Best fit
- Worst fit

Non risolvono il problema della frammentazione esterna

ANZI, con l'andare del tempo producono buchi inutilizzati di dimensione sempre piu' piccola

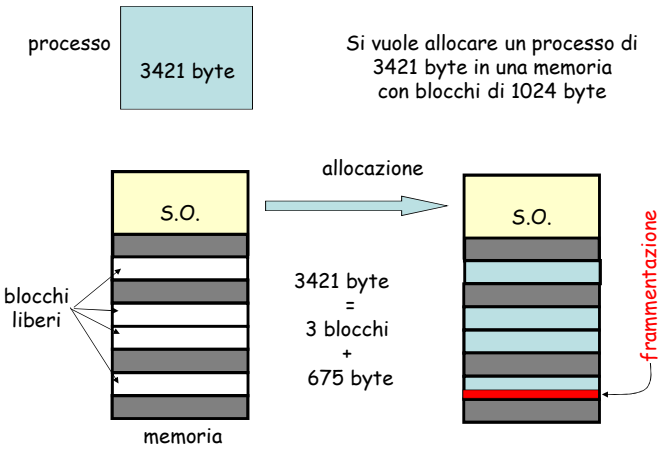
Approcci alternativi



## Allocazione non contigua

- Un'altra soluzione alla frammentazione esterna è ottenuta consentendo la **non contiguità degli indirizzi fisici**, permettendo così di allocare la memoria fisica ai processi ovunque essa sia disponibile.
- Si divide la memoria fisica in blocchi di dimensione fissa chiamati **blocchi** (o frame)
- Si divide il processo in blocchi della stessa dimensione chiamati **pagine**.
- Per eseguire un processo di *n* pagine, è necessario trovare *n* frame liberi prima di caricare il programma.
- Si ha solo frammentazione interna (relativa all'ultimo frame).

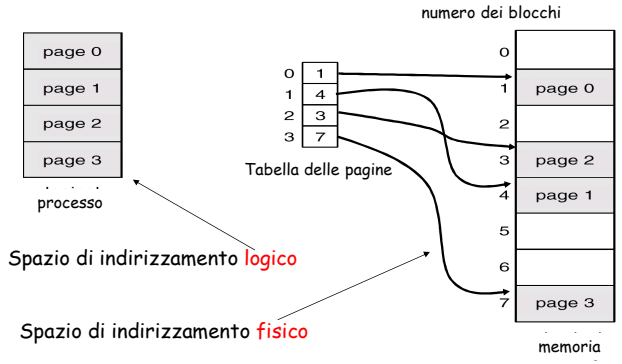
## Esempio:





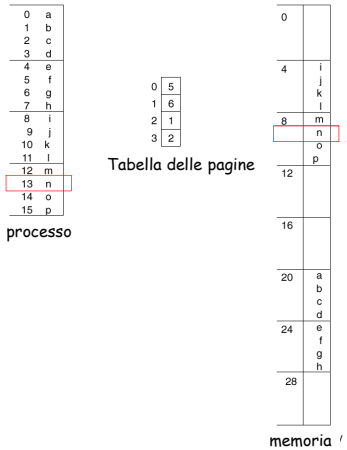

## Tabella delle pagine

- Il s.o. conserva in memoria una tabella delle pagine per ogni processo che contiene gli indirizzi dei blocchi utilizzati dalle pagine di quel processo in memoria






## Esempio: memoria di 32 byte e frame di 4 byte




L'indirizzo logico 13 e' nella posizione 1 della pagina 3

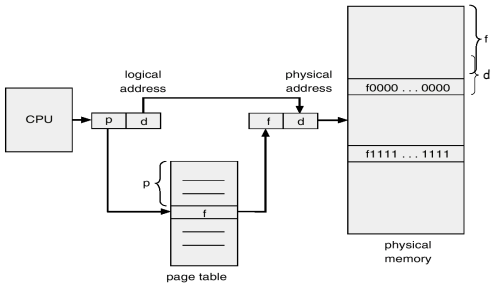
La pagina 3 e' nel blocco di memoria 2 (indirizzo 8)

L'indirizzo logico 13 corrisponde all'indirizzo fisico 8+1 = 9



## Schema di traduzione degli indirizzi nel MMU

- Ogni indirizzo generato dalla CPU (indirizzo logico) viene suddiviso in:
  - ***Numero di pagina*** **( *p* )** — impiegato come indice nella tabella delle pagine per trovare l'indirizzo del blocco
  - ***Scostamento nella pagina*** **( *d* )** — sommato con l'indirizzo del blocco per definire l'indirizzo fisico di memoria



## Due osservazioni

**1:** Il numero dei blocchi e la dimensioni dei blocchi sono potenze di 2

Esempio: m= 16 bit a disposizione per gli indirizzi
m-n = 8 bit per il numero di pagina p
n= 8 bit per lo scostamento d

| p | d |
|---|---|

$2^8$ = 256 blocchi $2^8$ = 256 byte per ogni blocco

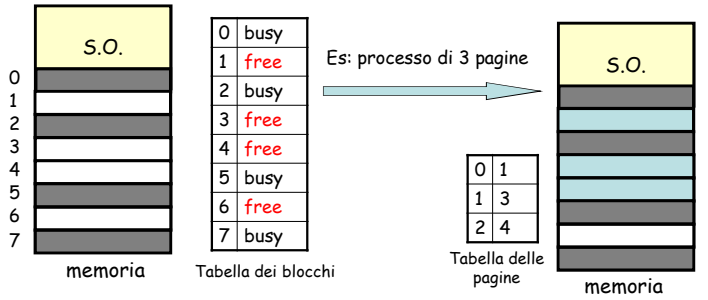
In generale:
$2^{m-n}$ blocchi di $2^n$ byte

**2:** La paginazione e' una forma di rilocazione dinamica e la protezione dei blocchi avviene mediante i valori di p e d analogamente ai registri base e limit



## Tabella dei blocchi

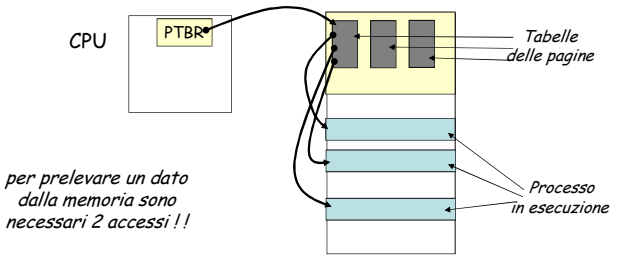
- Il sistema operativo tiene traccia sull'uso dei blocchi di memoria (libero, occupato e a chi assegnato) mediante una tabella dei blocchi
- Quando un nuovo processo richiede memoria viene consultata la tabella dei blocchi e viene creata la tabella delle pagine del nuovo processo






## Supporto hw per la paginazione

- Le tabella delle pagine (una per ogni processo) risiedono in memoria centrale.
- un registro della CPU identifica una tabella in memoria
  - ***Page-Table Base Register*** (PTBR) punta all'inizio della tabella.
- Al cambio di contesto il dispatcher aggiorna solo tale registro



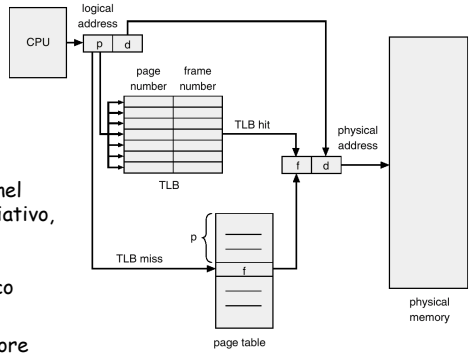



## Supporto hw alla paginazione

- Il problema dei due accessi alla memoria può essere risolto con dei ***registri associativi*** (altrimenti detti *translation look–aside buffer*, TLB)
- Tali registri sono caricati dal dispatcher al momento del cambio di contesto e attraverso i quali si effettua una ricerca parallela veloce su una piccola tabella (senza scorrerla tutta)
- Tali registri , realizzati nella CPU o nella MMU, sono molto veloci ma economicamente costosi

- Vengono usati per memorizzare un sottinsieme della tabella delle pagine

## Uso dei registri associativi




- Se p si trova nel registro associativo, si estrae il corrispondente numero di blocco
- Altrimenti, occore fare un riferimento in memoria alla tabella delle pagine.



## Effective access time (EAT)

- Esempio
- Tempo di accesso alla memoria = 100 nsec
- Tempo di accesso alla TLB = 20 nsec

Tempo di accesso (EAT) = 120 nsec se TLB hit / 220 nsec se TLB miss

- Se percentuale di successi (TLB ratio) = 80%

  Tempo di accesso = 0.8 x 120 + 0.2 x 220 = 140 nsec

- In generale
  - $\alpha$ = tempo acc alla TLB
  - $\beta$ = tempo acc alla memoria
  - $\varepsilon$ = TLB ratio

  ⇨ $EAT = \varepsilon(\alpha+\beta) + (1-\varepsilon)(\alpha+2\beta)$



## Background

- La memoria e' di solito gestita mediante tecniche di allocazione non contigua (paginazione)
- Motivazione principale : Riduzione della frammentazione

Ci sono altri vantaggi ad usare tecniche di allocazione non contigua?



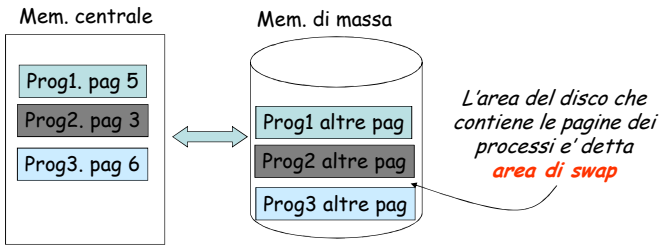
## Esempio:

- Memoria 32 MB
- S.O. 8 MB
- 3 processi di 8 MB
- Pagine di 2 MB


Altri processi non possono utilizzare la memoria

Soluzione:
Tenere in memoria centrale solo la pagine correntemente in esecuzione di ogni processo, mentre le altre possono risiedere in memoria di massa

## La memoria virtuale




La memoria virtuale e' lo spazio di indirizzamento effettivamente utilizzabile dall'utente

1. Lo spazio logico degli indirizzi e' più grande dello spazio fisico
2. Un maggior numero di processi puo' concorrere all'uso della CPU



## Esempi:

Intel Pentium :
pagine da $2^{12}$ B = 4KB
indirizzamento a 32 bit
} $2^{20}$ pagine

memoria totale = $2^{32}$ B ~ 4GB

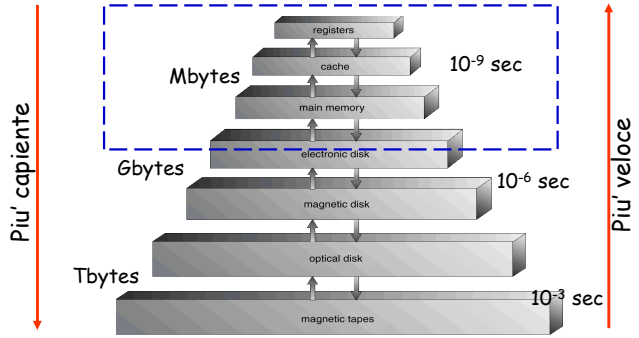
*Spazio effettivamente indirizzabile (la memoria centrale puo' essere molto piu' piccola !!)*

IBM PowerPC 970:
pagine da $2^{24}$ B = 16MB
indirizzamento a 64 bit
} $2^{40}$ pagine

memoria totale = $2^{64}$ B ~ $16 \times 10^{18}$B



## Memoria gerarchica



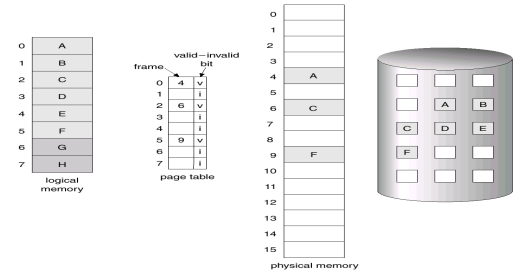

Direttamente accessibile dalla CPU

Se un riferimento alla memoria virtuale e' ad una pagina presente sul disco, la pagina deve essere portata in memoria centrale



## Page fault

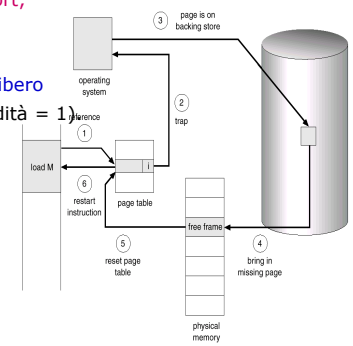
- Un bit di validità viene associato a ciascun elemento della tabella delle pagine (1 ⇒ in memoria, 0 ⇒ non in memoria).
- Inizialmente il bit di validità viene posto a 0 per tutte le pagine.
- Esempio




- In fase di traduzione degli indirizzi, se il bit di validità vale 0 ⇒ si ha un **page fault**.

## Gestione del Page fault

1. Viene effettuato un riferimento ad una pagina,
2. Se la pagina non e' in memoria viene mandata una interruzione al SO che consulta una tabella per decidere se si tratta di…
   1. riferimento non valido ⇒ abort;
   2. pagina non in memoria.
3. Si accede alla pagina sul disco.
4. Si sposta la pagina in un frame libero
5. Si aggiorna la tabella (bit di validità = 1)
6. Viene riavviata l'istruzione che era stata interrotta.






## Avvicendamento delle pagine

Non sempre e' presente in memoria un frame libero

3. Individuazione della pagina richiesta su disco.
4. Si sposta la pagina in un frame libero
   A. Se esiste un frame libero, viene utilizzato.
   B. Altrimenti viene utilizzato un algoritmo di sostituzione per selezionare un frame vittima.
   C. La pagina vittima viene scritta sul disco e la pagina richiesta viene letta nel frame appena liberato;
5. modifica delle tabelle delle pagine e dei frame.
6. Riavvio del processo utente.



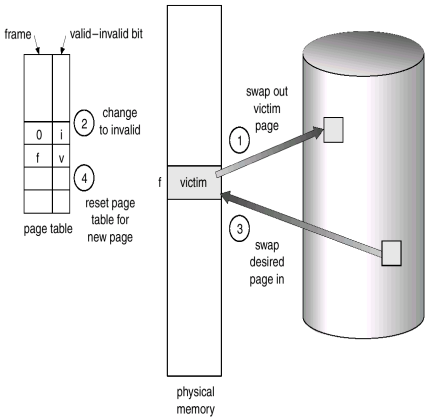
## Avvicendamento delle pagine

C.1 La pagina vittima viene scritta sul disco

C.2 il bit di validita' della pagina vittima viene posto = 0

C.3 la pagina richiesta viene letta nel frame appena liberato

C.4 il bit di validita' della pagina richiesta viene posto = 1






## Paginazione su richiesta

- porta una pagina in memoria solo quando è richiesta:
  - Vantaggi:
    - in memoria ci sono tutte e sole le pagine necessarie
    - Maggior grado di multiprogrammazione
  - Svantaggi
    - possibilita' di notevole overhead

## Prestazioni della paginazione su richiesta

- *Page Fault Rate*: $0 \le p \le 1.0$ e' il rapporto tra riferimenti in memoria e page fault
  - se $p = 0$ non si hanno page fault;
  - se $p = 1$, ciascun riferimento è un fault.

- Tempo medio di accesso (EAT):

EAT = (1 – *p*) x t[accesso alla memoria] + *p* x t[page fault]

I/O su disco { t[swap out di pagina] + t[swap in di pagina] + overhead di restart

è richiesto un metodo che produca il minimo page fault rate.



## Esempio di paginazione su richiesta

- Tempo di accesso alla memoria = 1 µsec
- Tempo di page fault = 10 msec = 10000 µsec
- p=0.5 (il 50% dei riferimenti produce un page fault)

EAT = 0.5 x 1 + 0.5 x 10000 ~ 5000 (in µsec)

- Se un accesso su 1000 causa un page fault (p=0.001),
  EAT = 0.999 x 1 + 0.001 x 10000 ~ 11 µsec ⇒
  l'accesso in memoria viene rallentato di un fattore 11.

- Se si desidera un rallentamento inferiore al 10%:
  EAT = (1-p) x 1 + p x 10000 < 1.1 µsec ⇒
  $9999p < 0.1 \Rightarrow p < 0.00001\ldots$
  può essere permesso al piu'
  un page fault ogni 100000 accessi in memoria.



## Paginatore

- Il modulo del S.O. Incaricato dell'avvicendamento delle pagine e' chiamato PAGINATORE

- Esistono molti algoritmi alla base dei paginatori con l'obiettivo comune di minimizzare la frequenza di page fault.

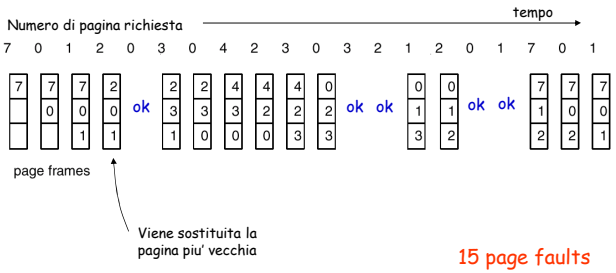
ALGORITMI DI AVVICENDAMENTO



## Algoritmo First–In–First–Out (FIFO)

- Viene sostituita la pagina presente in memoria da piu' tempo

- Esempio: processo di 8 pagine (0,..,7)
- 3 frame (3 pagine per ciascun processo possono trovarsi in memoria contemporaneamente).




15 page faults

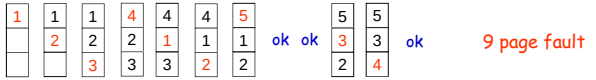
## Algoritmo FIFO

- Idea di base: una pagina presente in memoria da molto tempo non verra' piu' refernziata
- Vantaggi:
  - Facile da implementare (coda FIFO)
  - Basso overhead
- Svantaggi
  - Rischio di sostituire pagine molto utilizzate e per questo presenti in memoria da molto tempo
  - Anomalia di Belady



## Anomalia di Belady

- Si potrebbe pensare che con l'algoritmo FIFO aumentando il numero di frame si ha un minor numero di page faults
- Esempio: sequenza: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frame




9 page fault

- 4 frame

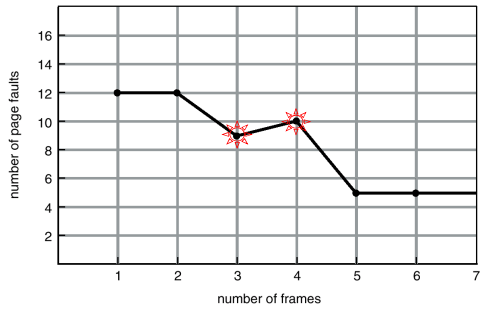

10 page fault

più frame ⇒ più page fault



## Anomalia di Belady

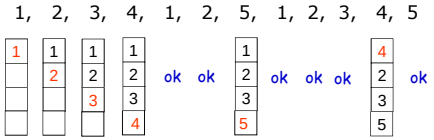
Richiesta alle pagine 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
al variare del numero di frame disponibili






## Algoritmo ottimo (OPT)

- Viene sostituita la pagina che non verrà usata per il periodo di tempo più lungo.
- Esempio: 4 frame, con stringa dei riferimenti

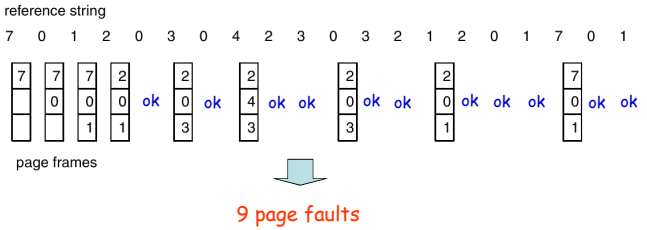



6 page fault

- Si puo' dimostrare che minimizza il numero di page faults
- Problema: Come si può conoscere l'identità della pagina?
- Di solo interesse teorico: viene impiegato per misurare le prestazioni (comparative) degli algoritmi con valenza pratica
- Esempio: FIFO/OPT = 10/6 = 1.67 ➔ FIFO 67% piu' lento di OPT

## Algoritmo ottimo

**ESEMPIO**




9 page faults

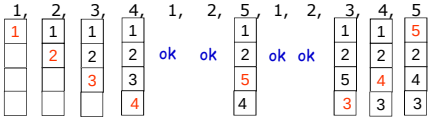
Con la stessa sequenza l'algoritmo FIFO produce 15 page faults

FIFO/OPT = 15/9 = 1.66 ➔ FIFO 66% piu' lento di OPT



## Algoritmo Least–Recently–Used (LRU)

- Viene sostituita la pagina che non e' referenziata da piu' tempo
- Esempio: 4 frame con stringa di riferimenti

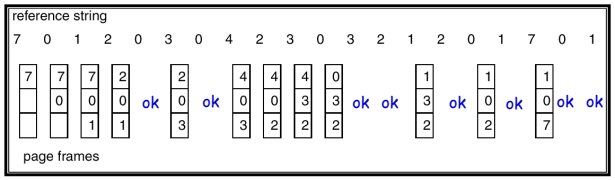



8 page fault

- LRU/OPT = 8/6 = 1.33 ➔ LRU 33% piu' lento di OPT
- migliori prestazioni rispetto all'algoritmo FIFO ma con un maggiore overhead
- Favorisce i processi che esibiscono il principio di localita' temporale



## ESEMPIO algoritmo LRU




**12**fault

LRU/OPT = 12/9 = 1.33 = LRU 33% piu' lento di OPT



## Implementazione dell'algoritmo LRU

- Implementazione con registro
  - Ciascuna pagina ha un registro; ogni volta che si fa riferimento alla pagina si copia l'orologio nel registro.
  - Quando si deve rimuovere una pagina, si analizzano i registri per scegliere quale pagina cambiare.
  - Overhead per la ricerca della pagina "piu' vecchia" non refernziata
- Implementazione con lista :
  - Le pagine in memoria sono organizzate secondo una lista; ogni volta che si fa riferimento alla pagina, la si sposta in testa alla coda
  - Quando si deve rimuovere una pagina, si elimina la pagina in coda alla lista
  - Overhead per la gestione della lista.

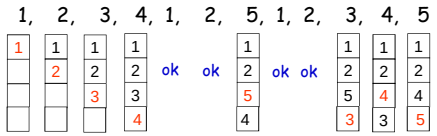
## Algoritmo LRU

- Idea di base: una pagina presente in memoria che non e' referenziata da molto tempo non verra' piu' refernziata
- Vantaggi:
  - Piu' efficiente dell'algoritmo FIFO
- Svantaggi
  - Difficile da implementare
  - Alto overhead



## Algoritmo Least–Frequently–Used (LFU)

- Viene sostituita la pagina meno frequentemente utilizzata
- Esempio: 4 frame con stringa di riferimenti




- LFU/OPT = 8/6 = 1.33 ➔ LFU 33% piu' lento di OPT
- migliori prestazioni rispetto all'algoritmo FIFO ma con un maggiore overhead



## Algoritmo LFU

- Idea di base: una pagina presente in memoria che non e' molto referenziata sara' poco referenziata anche in futuro
- Vantaggi:
  - Piu' efficiente dell'algoritmo FIFO
- Svantaggi
  - Rischio di sostituire pagine da poco presenti in memoria e per questo poco referenziate rispetto ad altre (NO localita' temporale)
  - Alto overhead



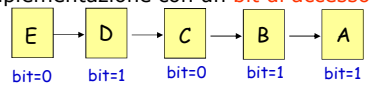
## variante dell'algoritmo FIFO: seconda chance

Il principale difetto dell'algoritmo FIFO e' che sostituisce pagine presenti in memoria da molto tempo
e che potrebbero essere molto utilizzate

IDEA:
dare una seconda opportunita' alle pagine prima di essere rimosse

Se la pagina e' utilizzata la si lascia in memoria

- Implementazione con un bit di accesso

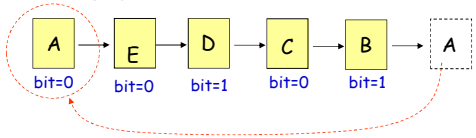



Coda FIFO

## variante dell'algoritmo FIFO: seconda chance

Quando una pagina viene referenziata si pone bit=1

Quando le pagine raggiungono la testa della lista
- Se bit=1 si sposta la pagina in fondo alla lista e si pone bit=0
- Se bit=0 la pagina viene rimossa






## Altre considerazioni

- Struttura dei programmi
  - Pagine di 4096 byte (4x1024)
  - A[ ][ ] = new int[1024][1024];
  - Ciascuna riga viene memorizzata in una pagina (1024 pagine)
  - Programma 1

```
for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
            A[i][j] = 0;
```

accesso per colonne
1024 x 1024 page fault

  - Programma 2

```
for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
            A[i][j] = 0;
```

Accesso per righe
1024 page fault

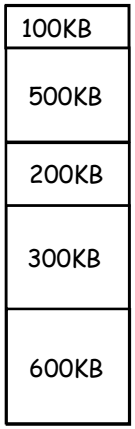


## Esercizi

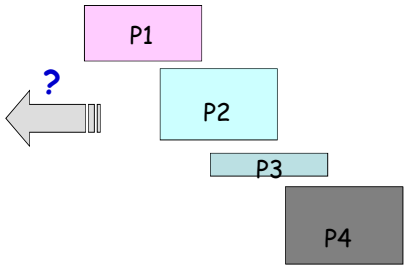
ESERCIZI



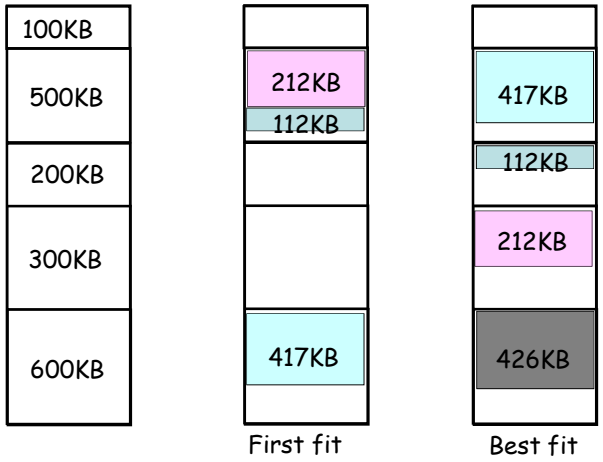
## Esercizio 1




Date le seguenti sezioni di memoria libera : 100KB, 500KB, 200KB, 300KB e 600KB, descrivere come sono disposti dagli algoritmi di *first-fit* e *best-fit* i processi di 212KB, 417KB, 112KB e 426KB (in ordine).



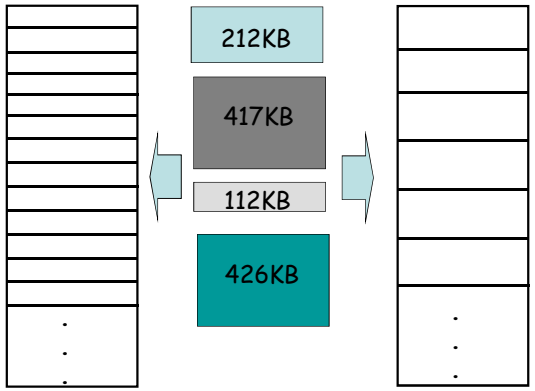
## soluzione





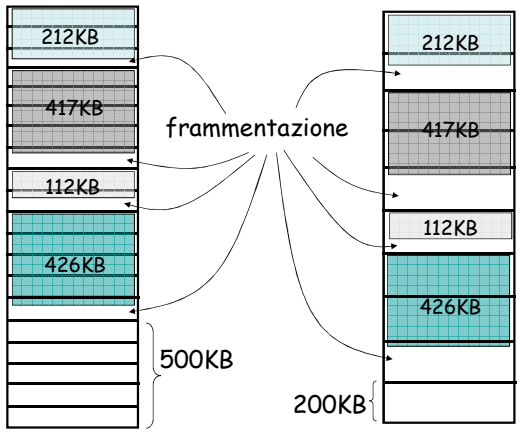

## Esercizio 2

Sia una memoria di 2000KB che viene gestita mediante la paginazione. Si descriva l'allocazione della memoria, relativamente a processi di dimensioni 212KB, 417KB, 112KB e 426KB nel caso di pagine da 100KB e 200KB,






## soluzione






## Esercizio 3

- Un sistema ha indirizzi di 16 bit e pagine da 1024B. Determinare:
  - quanti elementi sono necessari al piu' per la tabella delle pagine

## soluzione

- Con 16 bit e' possibile indirizzare $2^{16}$ byte
- Ogni pagina e' di $2^{10}$ byte

$2^{16}/2^{10} = 2^6 = 64$ pagine

- Supponendo che la tabella delle pagine contenga i seguenti riferimenti, calcolare gli indirizzi fisici ai seguenti indirizzi logici
  a) < 3, 126 >
  b) < 4, 723 >
  c) < 2, 59 >
  d) < 1, 865 >

| pagina | Indirizzo |
|---|---|
| 0 | 5119 |
| 1 | 1023 |
| 2 | 9215 |
| 3 | 0 |
| 4 | 12287 |



## Soluzione

| pagina | Indirizzo |
|---|---|
| 0 | 5119 |
| 1 | 1023 |
| 2 | 9215 |
| 3 | 0 |
| 4 | 12287 |

a) < 3, 126 >
b) < 4, 723 >
c) < 2, 59 >
d) < 1, 865 >

a) 0 + 126 = 126
b) 12287 + 723 = 13010
c) 9215 + 59 = 9274
d) 1023 + 865 = 1888



## Esercizio 4

Data la seguente tabella dei segmenti:

| Segmento | Base | Lunghezza |
|---|---|---|
| 0 | 219 | 600 |
| 1 | 2300 | 14 |
| 2 | 90 | 100 |
| 3 | 1327 | 580 |
| 4 | 1952 | 96 |

Calcolate gli indirizzi fisici corrispondenti ai seguenti indirizzi logici:
a) < 0, 430 >
b) < 1, 10 >
c) < 2, 500 >
d) < 3, 400 >
e) < 4, 112 >



## Soluzione

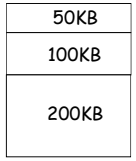
| Segmento | Base | Lunghezza |
|---|---|---|
| 0 | 219 | 600 |
| 1 | 2300 | 14 |
| 2 | 90 | 100 |
| 3 | 1327 | 580 |
| 4 | 1952 | 96 |

indirizzi fisici corrispondenti agli indirizzi logici:
a) < 0, 430 > ➔ 219 + 430 = 649
b) < 1, 10 > ➔ 2300 + 10 = 2310
c) < 2, 500 > ➔ 500 > 100 *errore di segmentazione*
d) < 3, 400 > ➔ 1327 + 400 = 1727
e) < 4, 112 > ➔ 112 > 96 *errore di segmentazione*

## Esercizio 5

Un programma ha il segmento di testo di 200KB, quello dei dati di 100KB e uno stack di 50KB.

Se la memoria è organizzata a pagine da 16KB:
1. Quanto è grande la tabella delle pagine del processo?
2. Quanta memoria può al più essere gestita se si hanno a disposizione 32 bit per l'indirizzamento?




*Ogni segmento del processo e' paginato*



## soluzione

Risulta:
- Testo 200KB ➔ 200/16 = 12.5 = 13 pagine
- Dati 100KB ➔ 100/16 = 6.25 = 7 pagine
- Stack 50KB ➔ 50/16 = 3.125 = 4 pagine

Per un totale di **24 pagine**

Per indirizzare pagine da 16KB ($16K=2^{14}$), c'è bisogno di 14 bit.
Restano 18 bit per generare i numeri delle pagine ➔ 256K pagine

Quindi $2^{32}$ = 4GB di memoria complessiva.



## Esercizio 8

Considerate un sistema di paginazione con la tabella delle pagine nella memoria centrale.

1. Se il riferimento alla memoria richiede 200ns, dite quanto tempo richiede un riferimento alla memoria paginata.

2. Se si aggiunge TLB e il 75% di tutti i riferimenti alla tabella delle pagine si trova in quest'ultimo, dite quant'è il tempo medio di accesso alla memoria, nei casi di:
- tempo di reperimento nel TLB = 0;
- tempo di reperimento nel TLB = 20ns.

3. In quest'ultimo caso, quanto dovrebbe essere l'hit ratio del TLB, affinché si abbia un tempo di accesso effettivo di 240ns?



## soluzione

Per accede ad un elemento in memoria, bisogna
- accedere alla tabella delle pagine (200ms)
- accedere all'elemento (200ms)

} 400ns

In generale $EAT = (\alpha + \beta)\varepsilon + (2\beta + \alpha)(1 - \varepsilon)$

- $\varepsilon$ =0.75, $\beta$ = 200ns, $\alpha$ = 0 =>
  $EAT = 200ns * 0,75 + 400ns * 0,25 = 250ns$

- $\varepsilon$ =0.75, $\beta$ = 200ns, $\alpha$ = 20ns =>
  $EAT = 220ns * 0,75 + 420ns * 0,25 = 270ns$

$220\varepsilon + 420(1 - \varepsilon) = 240$ ➔ $\varepsilon = 90\%$

## Algoritmi di avvicendamento

- FIFO (viene sostituita la pagina in memoria presente da piu' tempo)
- LRU (viene sostituita la pagina che non e' usata da piu' tempo)
- LFU (viene sostituita la pagina meno frequentemente utilizzata)
- Seconda chance/CLOCK (varianti di FIFO)

Per tutti obiettivo comune:
Minimizzare il numero di page faults



## Esercizio 1

- Con riferimento ad un ambiente di gestione della memoria virtuale con paginazione su richiesta, si consideri un processo caratterizzato dalla seguente stringa di riferimenti

1 0 3 5 6 9 1 19 15 18 9 15 1 3 5 1 9 19 3

- Si illustri il comportamento degli algoritmi FIFO e LRU nel caso siano assegnati al processo 5 frame. Si calcoli il numero di page faults



## Soluzione FIFO

| 1 | 0 | 3 | 5 | 6 | 9 | 1 | 19 | 15 | 18 | 9 | 15 | 1 | 3 | 5 | 1 | 9 | 19 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 9 | 9 | 9 | 9 | 9 | | | | 3 | 3 | 3 | 3 | 3 | |
| | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | | | | 1 | 5 | 5 | 5 | 5 | |
| | | 3 | 3 | 3 | 3 | 3 | 19 | 19 | 19 | ok | ok | ok | 19 | 19 | 1 | 1 | 1 | ok |
| | | | 5 | 5 | 5 | 5 | 5 | 15 | 15 | | | | 15 | 15 | 15 | 9 | 9 | |
| | | | | 6 | 6 | 6 | 6 | 6 | 18 | | | | 18 | 18 | 18 | 18 | 19 | |

15 page faults



## Soluzione LRU

| 1 | 0 | 3 | 5 | 6 | 9 | 1 | 19 | 15 | 18 | 9 | 15 | 1 | 3 | 5 | 1 | 9 | 19 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 9 | 9 | 9 | 9 | 9 | | | | 9 | 9 | | | 9 | |
| | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | | | | 1 | 1 | | | 1 | |
| | | 3 | 3 | 3 | 3 | 3 | 19 | 19 | 19 | ok | ok | ok | 3 | 3 | ok | ok | 3 | ok |
| | | | 5 | 5 | 5 | 5 | 5 | 15 | 15 | | | | 15 | 15 | | | 19 | |
| | | | | 6 | 6 | 6 | 6 | 6 | 18 | | | | 18 | 5 | | | 5 | |

13 page faults

## Esercizio 2

- Supponiamo che il sistema di paginazione usato dal S.O. assegni 3 frame da 512B a ciascun processo e si consideri il seguente programma

```
...
#define  N   512
int A[N], C[N];
int i,j;
...
for(i=1; i<=2; i++)
for(j=0; j<N/2; j++)
A[i*j] = A[2*i] + C[2*j];
...
```

1. si determini la stringa dei riferimenti alle pagine della memoria supponendo che un intero si rappresentato con 2B
2. si determini la tabella delle pagine, al termine dell'esecuzione della procedura, nel caso che l'algoritmo di avvicendamento sia (i) FIFO, (ii) LRU



## Soluzione

- Gli array A e C occupano 1024B ciascuno, suddivisi in 2 pagine da 512B come segue:
  - A[0..255] alla pagina 0, A[256..511] alla pagina 1
  - C[0..255] alla pagina 2, C[256..511] alla pagina 3
- Riferimenti generati dalla CPU per eseguire A[i*j] = A[2*i] + C[2*j];
  - i=1
    - j=0 0 2 0
    - ...
    - j=127 0 2 0
    - j=128 0 3 0
    - ...
    - j=255 0 3 0
  - i=2
    - j=0 0 2 0
    - ...
    - j=127 0 2 0
    - j=128 0 3 1
    - ...
    - j=255 0 3 1

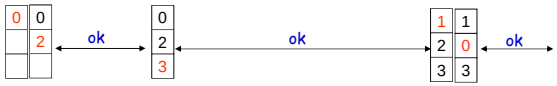
..... 0 2 0 ..... ..... 0 3 0 ..... ..... 0 2 0 ..... ..... 0 3 1 .....

128 volte 128 volte 128 volte 128 volte

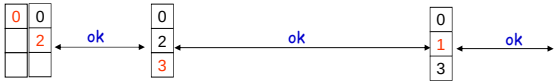


## soluzione

..... 0 2 0 ..... ..... 0 3 0 ..... ..... 0 2 0 ..... ..... 0 3 1 .....




FIFO ➔ 5 page fault




LRU ➔ 4 page fault



## Esercizio 3

- Nei due casi precedenti:
  - FIFO = 5 page fault
  - LRU = 4 page fault

qual'e' il tempo di accesso effettivo della paginazione su richiesta se il tempo medio di servizio e'

- 25 millisecondi ($25*10^{-3}$ sec) in caso di page fault
- 100 microsecondi ($100*10^{-6}$ sec) in caso di pagina presente in memoria?

## Soluzione

- Complessivamente l'istruzione A[i*j] = A[2*i] + C[2*j] viene eseguita 512 volte
- Ogni volta vengono fatti 3 accessi
- Totale di 1536 riferimenti in memoria.

- EAT = (1 – *p*) x t[accesso alla memoria] + *p* x t[page fault]

FIFO ➔ p=5/1536 = 0.0032

EAT = 0.9968 x 100*10$^{-6}$ + 0.0032 x 25000*10$^{-6}$
= 179.68 *10$^{-6}$ sec

LRU ➔ p=4/1536 = 0.0026

EAT = 0.9974 x 100*10$^{-6}$ + 0.0026 x 25000*10$^{-6}$
= 164.74 *10$^{-6}$ sec



## Esercizio 4

- Considerate la seguente successione di riferimenti di pagine:

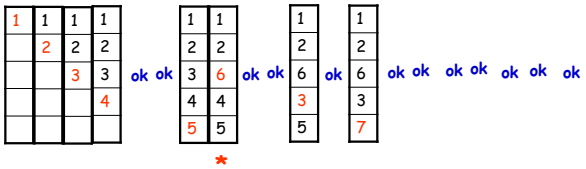
  1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 3, 2, 7, 6, 3, 2, 1, 2, 3, 6.

- Determinare il tempo di accesso effettivo della paginazione su richiesta per LRU con 5 blocchi, se:
  - il tempo medio di servizio di un page fault senza salvataggio della pagina avvicendata é di 80 millisecondi (80*10$^{-3}$ sec),
  - il tempo medio di servizio di un page fault con salvataggio della pagina avvicendata è di 140 millisecondi (140*10$^{-3}$ sec)
  - il tempo di accesso alla memoria è di 80 microsecondi (80*10$^{-6}$ sec),
- nell'ipotesi che l'accesso alle pagine 1, 2 , 3 sia sempre in scrittura.



## soluzione

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 3, 2, 7, 6, 3, 2, 1, 2, 3, 6




* Quando la pagina 6 prende il posto della pagina 3, quest'ultima deve essere salvata perche' l'accesso precedente era in scrittura

- 1 page fault con salvataggio della pagina sostituita
- 7 page fault senza salvataggio della pagina sostituita
- 12 accessi diretti senza page fault

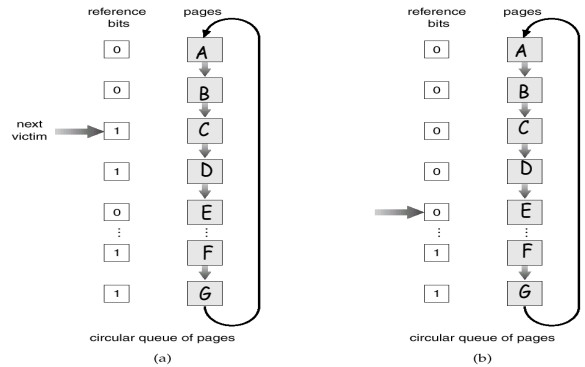
- EAT = (12/20 * 80 + 7/20 * 80000 + 1/20 * 140000 ) *10$^{-6}$ sec = 35.048 msec



## FIFO Seconda chance

- Implementazione mediante un bit di accesso. (bit=1 se referenziata)
- Si scorrono le pagine presenti nella lista:
- Se bit=1 :
  - Si pone il bit di riferimento a 0.
  - Si lascia la pagina in memoria.
  - Si rimpiazza la pagina successiva (in ordine di clock), in base alle stesse regole.
- Se bit=0
  - Si rimpiazza la pagina

## FIFO seconda chance




- pagina C candidata ad essere eliminata
- bit=1 ➔ C viene salvata e si esamina la pagina D
- bit=1 ➔ D viene salvata e si esamina la pagina E che viene eliminata



## Esercizio 5

- Data la seguente sequenza di riferimenti di pagine
  1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 3, 2, 7, 6, 3, 2, 1, 2, 3, 4

Determinare quanti page fault avvengono utilizzando il clock algorithm con 5 frame e lancetta dell'orologio posizionata sul primo elemento



## soluzione

| 1 | 2 | 3 | 4 | 2 | 1 | 5 | 6 | 2 | 1 | 3 | 2 | 7 | 6 | 3 | 2 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1* | 1* | 1 | 1 | 1* | 1* | 1* | 1* | 1* | 1* | 1* | 1* | 1* | 1* | 1 |
| | 2 | 2 | 2 | 2* | 2* | 2* | 2 | 2* | 2* | 2* | 2* | 2* | 2* | 2* | 2* | 2* | 2* | 2* | 2 |
| | | 3 | 3 | 3 | 3 | 3 | 6 | 6 | 6 | 6 | 6 | 6 | 6* | 6* | 6* | 6* | 6* | 6* | 6 |
| | | | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3* | 3* | 3* | 3* | 3* | 3 |
| | | | | | | 5 | 5 | 5 | 5 | 5 | 5 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 4 |
| | | | | ok | ok | | ok | ok | | ok | | ok | ok | ok | ok | ok | ok | | |
| (0) | | | | | | | (1) | | | (2) | | (3) | | | | | | | (4) |

9 P.F. !!

- (0) all'inizio clock = 1
- (1) pag 1 e 2 bit=1 ➔ si portano bit=0 e si avvicenda pagina 3 ➔ clock =4
- (2) clock = 5
- (3) clock = 1
- (4) pag 1, 2, 6, 3 bit=1 ➔ si portano bit=0 e si avvicenda pagina 7 ➔ clock =1



## Esercizio 8

- Supponiamo che il sistema di paginazione utilizzato dal sistema operativo assegni 3 frame (blocchi di memoria) da 512B a ciascun processo e che l'algoritmo di sostituzione delle pagine sia LRU.
- Prendiamo in considerazione il seguente programma:

```
...
#define N 512
int a[N];
int i;
...
for (i=0; i < N/2; i++)
a[i] = a[2*i] + a[N-i-1];
...
```

- Si risponda ai seguenti quesiti:
  - se la dimensione di un intero è 4B, qual è il numero di page faults ?
  - In tal caso, se il tempo medio di servizio di un page fault é di 25 millisecondi ($25*10^{-3}$ sec) ed il tempo di accesso alla memoria di 100 microsecondi ($100*10^{-6}$ sec), qual è il tempo di accesso medio della paginazione su richiesta?

## soluzione

Le pagine sono da 512B e ogni intero è 4B (128 elementi a pagina):

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| a[0] … a[127] | a[128] … a[255] | a[256] … a[383] | a[384] … a[511] |

for (i=0; i < N/2; i++) a[i] = a[2*i] + a[N-i-1];

| | | |
|---|---|---|
| i = 0,… 63 | a[2*i] -> 0<br>a[N-i-1] -> 3<br>a[i] -> 0 | 64 volte |
| i = 64,…127 | a[2*i] -> 1<br>a[N-i-1] -> 3<br>a[i] -> 0 | 64 volte |
| i = 128,…191 | a[2*i] -> 2<br>a[N-i-1] -> 2<br>a[i] -> 1 | 64 volte |
| i = 192,…255 | a[2*i] -> 3<br>a[N-i-1] -> 2<br>a[i] -> 1 | 64 volte |

Totale riferimenti = 768



## soluzione

| 0 | 3 | 0 | … | … | 1 | 3 | 0 | … | … | 2 | 2 | 1 | … | … | 3 | 2 | 1 | … | … |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 |
| | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | | | | | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | | | | ok | ok | ok | | ok | ok | ok | ok | | ok | | ok | ok | | ok | ok | ok | ok |

- 6 page fault
- 762 riferimenti validi

EAT = ( 6/768 * 25*10$^{-3}$ + 762/768 *100*10$^{-6}$ ) sec = 0.294*10$^{-3}$



## Esercizio 9

- In un s.o. con paginazione su richiesta occorrono:
  - 8 msec in caso di p.f. senza salvataggio della pagina avvicendata
  - 20 msec in caso di p.f. con salvataggio della pagina avvicendata
  - 100 nsec in caso di pagina presente in memoria
- Supponendo che il 70% delle volte e' necessario salvare la pagina avvicendata, determinare il massimo valore del p.f. rate p per ottenere un EAT al piu' di 200 nsec



## soluzione

EAT = p $T_{pf}$ + (1-p) $T_{am}$ < 0.2*10$^{-6}$

EAT = p ( 0.7 * 20 + 0.3 * 8) *10$^{-3}$ + ( 1-p ) * 0.1 *10$^{-6}$ =
1000p (1.4 + 2.4 )*10$^{-6}$ + ( 1-p ) * 0.1 *10$^{-6}$ =
3800p *10$^{-6}$ + 0.1 *10$^{-6}$ - 0.1 p * 10$^{-6}$ < 0.2*10$^{-6}$

3799.9p < 0.1

p < 0.1/3799.9 = 0.000026 = 2.6 10$^{-5}$
(26 p.f. ogni 10000 riferimenti)