Ch10 MEMORIA VIRTUALE

L'eseguibile necessita di andare in memoria per essere eseguito. Solitamente non viene usato tutto il programma in quanto pezzi del programma potrebbero non essere utili, come ad esempio:

- Gestioni errori
- Funzioni che gestiscono casi molto particolari

Inoltre, non è detto che se tutto il programma serve, serva allo stesso tempo. Magari qualcosa mi serve all'inizio e non più dopo, quando mi servirà un'altra cosa che prima non mi serviva.

Vogliamo fare un **loading parziale** → a questo punto posso avere anche dei programmi più grossi della RAM o avere un maggior numero di processi dentro la CPU. Inoltre, essendo il loading parziale, impiego meno tempo per fare load.

Memoria virtuale: Separazione più netta tra memoria logica e fisica.

- Solo una parte del programma ha bisogno di essere in memoria per essere eseguito
    - Spazio di indirizzamento logico può essere più grande di quello fisico
        - Sharing
- Permette shared di spazio di indirizzamento tra più processi
- Creazione processo più efficiente
- Più processi possono andare in contemporanea
- Meno I/O necessari per caricare o swappare processi

Spazio di indirizzamento virtuale: modo logico per salvare percorsi in memoria

- Solitamente indirizzi partono da 0, contigui fino a fine spazio
- Memoria fisica organizzata in frame
- MMU mappa logico a fisico

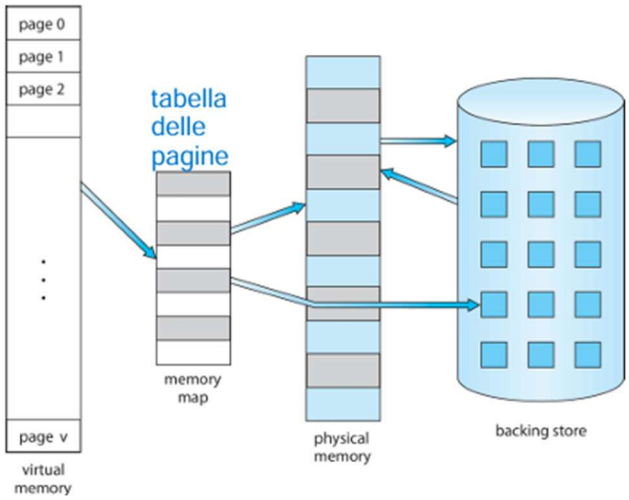
Memoria virtuale può essere **implementata con**:

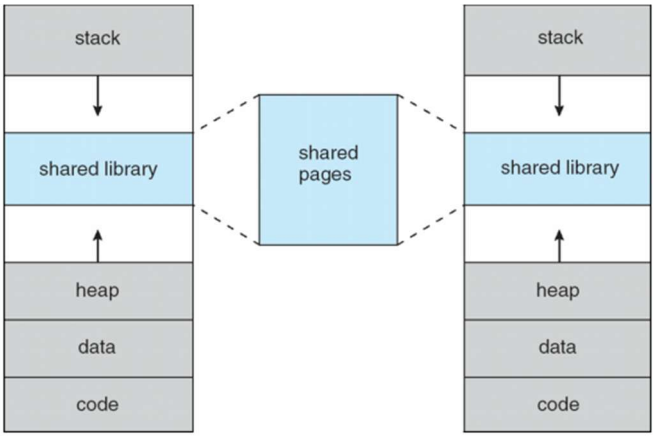
- **Demand Paging**
- **Demand Segmentation** → si rilassa l'idea di tagliare in modo uniforme, si taglia in base al contenuto

Se ci sono pagine dell'eseguibile che non stanno in RAM → **backing store** (disco)

Ram piccola, backing store molto grande, memoria virtuale molto grande.

Indirizzamento logico: Normalmente codice in linguaggio macchina e dati stanno ad indirizzi bassi.
Ci sono due parti di memoria: heap (malloc) e stack (chiamate a funzioni) che sono soggetti ad allocazioni e deallocazioni → sono messi uno in alto e uno in basso in modo tale da non darsi troppo fastidio tra loro. Inolte, nello heap ci potrebbero essere dei buchi. In più ci sono le shared library.



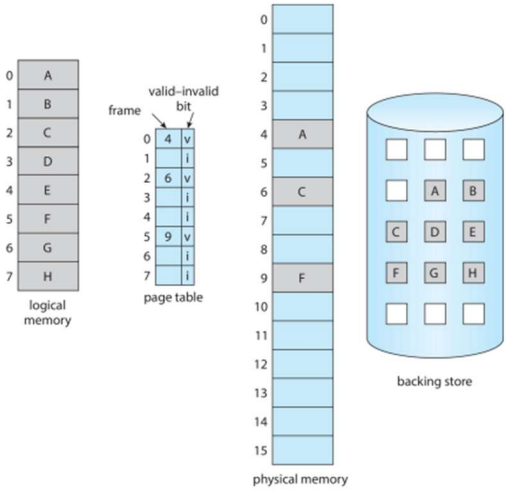


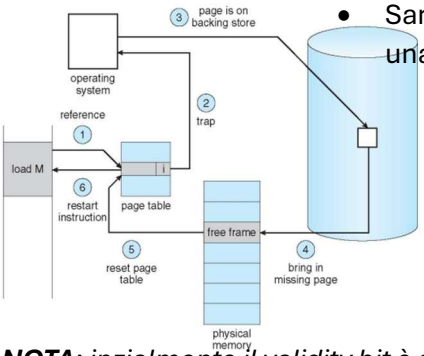
DEMAND PAGING

NOTA: si può portare l'intero processo in memoria al load time
NOTA: si possono portare solo le cose necessarie → **es caso estremo**: alla partenza non porto nulla eccetto l'indirizzo della prima istruzione (partenza del main) → *quando la eseguo, prendo la prima pagina del main e la porto in memoria; quando la uso, allora prendo la seconda pagina e la porto dentro → dunque si ha una risposta molto più rapida alla partenza con un I/O iniziale limitato, meno memoria necessaria, più utenti possibili.*

Scelta pagine: come faccio a motivare le pagine che servono (devo capire cosa mettere dentro e cosa fuori) → Dunque, nel meccanismo di esecuzione serve un meccanismo che si occupa della traduzione logica-fiscia(MMU) e un meccanismo che si occupi di portare dentro ciò che serve (sfruttando il validity bit):

- Il **validity bit** qui **ha 3 significati**
  - **Valido** → c'è → MMU fa traduzione
  - Non valido, PAGE FAULT, TRAP → errore di pagina o pagina mancante → invoco gestore interrupt:
    - In un'altra tabella, verifica se:
      - Veramente indirizzo **non valido** →Non c'è proprio → quando ci si trova nel vuoto
        - ABORT → SEGMENTATION FAULT
      - Sarebbe **valido, se fosse in RAM**, è una pagina a cui manca il frame
        - Cerca frame libero
        - Prendi la pagina da backing store o nella partizione di swap o nel file eseguibile(file system) (ovunque essa sia)
        - Tabella della pagina diventa valid, scrivo V, faccio ripartire l'istruzione che ora trova validity bit=Valid





***NOTA**: inzialmente il validity bit è settato come invalid per tutte le entry*

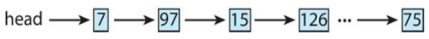
NOTA: se la prima istruzione avesse già bisogno di più pagine → multiple page faults *(grazie al principio di località dei riferimenti, dolore diminuisce)*

NOTA: serve un supporto HW per il demand paging:

- Serve avere il bit valid/invalid nella page table
- Memoria secondaria per swap
- Restart delle istruzioni efficiente

GESTIONE DEI FRAME LIBERI

Per gestire i frame liberi → **free frame list**: lista di frame liberi



Alcune volte, servono dei frame azzerati → zero frame → liberi è già messi a 0 → **zero-fill-on-demand.**

Quanto costa un **page fault** e a cosa sono disposto per gestirlo? PEGGIOR CASO

1. Trap del sistema operativo
2. Salvare registri utente e stato processo
3. Gestione capisce che è un page fault
4. Gestore page fault → determina che era un accesso legale e determina dove si trova la pagina sul disco
5. Issue la lettura dal disco al free frame (testa della lista di free frame)
    - Aspetta il tempo di richiesta read
    - Aspetta per latency time del dispositivo
    - Fai partire trasferimento e aspetti
6. Gestore del page fault è in waiting, alloco CPU ad un altro processo utente B
7. I/O del gestore del page fault è finito → interrupt
8. Salva registri e stato processo B
9. Determinare se interrupt era da disco
10. Correggere page table e altre tabelle per mostrare che ora la pagina è in memoria
11. Aspettare la CPU che riallochi il procrsso
12. Restore dei registri utenti, stato processo e nuova page table e riparte l'istruzione interrotta

Le tre **attività maggiori** sono:

1. Servire interrupt
2. Lettura page
3. Restart del processo

Calcolo:

- Page Fault Rate $0 \le p \le 1$
    - if $p = 0$ no page faults
    - if $p = 1$, every reference is a fault

Effective Access Time (EAT)

EAT = (1 – *p*) x memory access
+ *p* (page fault overhead
+ swap page out
+ swap page in)

memory acess: tempo effettivo di accesso con la page table

Page fault overhead: tutto ciò che hai fatto prima di arrivare su disco (accesso tlb andando male, accesso page fault andato male)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
    - 220 > 200 + 7,999,800 x p
      20 > 7,999,800 x p
    - p < .0000025
    - < one page fault in every 400,000 memory accesses

Un page fault costa parecchio, quindi bisogna trovare strategia per ridurre il costo di ciascun page fault o diminuire il numero di page fault

→ OTTIMIZZAZIONE DEL DEMAND PAGING

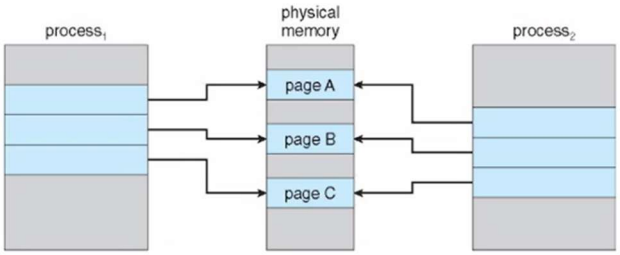
Ottimizzazione del demand paging

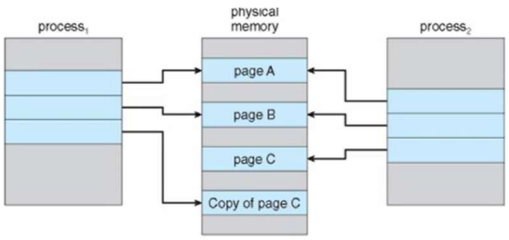
Due opzioni:

- Fare in modo che il trasferimento dei dati tra disco e memoria costi meno tempo
    - I/O legato allo spazio di swap può essere migliorato rendendo più efficiente lo spazio a disco
        - Allocando in modo migliore la parte del disco dedicata a swap
        - Gestione di tale disco efficiente (no file system standard
- Risparmiare numero di copie
    - Una volta che si è deciso chela parte di disco dedicata a swap permette accessi più efficienti rispetto al file system, conviene copiare all'inizio tutto il file da file system nella partizione di swap e poi usare questa
    - Se le pagine su disco non vengono cambiate, si fa solo swap-in e non swap-out delle pagine su disco
    - Alcune cose vengono copiate e altre no
        - **Anonymous memory**: pagine non associate con un file (es: stack e heap)

COW: Copy-on-Write

Permette a padre e figli di **condividere le stesse pagine in memoria**. Se uno dei due vuole scrivere su una di queste pagine, deve fare una copia e modificare la copia.





Nel caso in cui servano pagine e frame liberi per ottenere spazio in memoria, vengono gestiti dei pool (insiemi di pagine già azzerate) → zero-fill-on-demand.

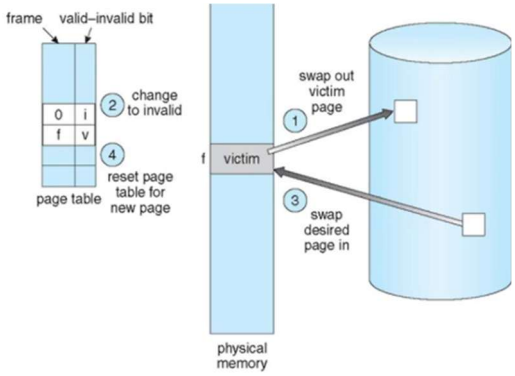
Nel caso in cui **non ci siano free-frame**: occorre adottare una politica di rimpiazzamento/sostituzione.
Bisogna scegliere quale pagina portare fuori → algoritmi che ottimizzino la scelta.

Pace-replacement → rimpiazzamento di pagina: previene la sovra-allocazione di memoria:

- Quando stiamo gestendo un page fault e si è deciso che occorre un frame libero per caricare una pagina, bisogna fare page replacement.
    - Occorre un modify/dirty bit: bit associato alla pagina che dice se la pagina è stata modificata rispetto all'ultima volta che è stata portata da disco in memoria
        - Se non settato → c'è già una copia uguale su disco (perché non è stato modificato)

Politica di sostituzione base: gestione di un page fault ha trovato la pagina sul disco e deve trovare un frame libero

- Frame c'è → usalo
- Frame non c'è → seleziona un frame vittima
    - Manda la pagina che lo ha in backing store, ora libero
        - Se la pagina che lo occupa ha dirty bit a 0, non serve copiarla su disco
        - Se la pagina che lo occupa ha dirty bit a 1, bisogna fare 2 trasferimenti:
            - Copia su disco della pagina vittima
            - Caricamento della nuova pagina

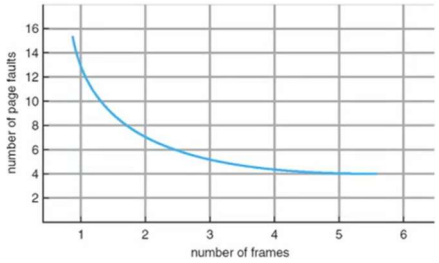


ALGORITMI DI SOSTITUZIONE PAGINE

Usati per individuare la vittima

- Quanti frame associo ad un processo?
- Quale frame selezionare come vittima?

Obiettivo: ottenere il nr di page fault più basso possibile

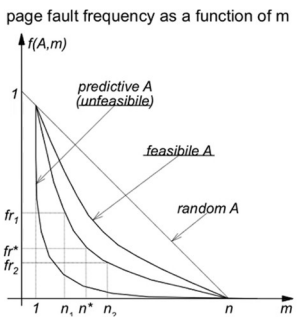


Più frame si mettono, meno page fault ci saranno in quanto sarà più facile vedere la pagina in memoria.

Page Fault Frequency (empirical probability)

$$f(A,m) = \sum_{\forall w} p(w) \frac{F(A,m,w)}{len(w)}$$

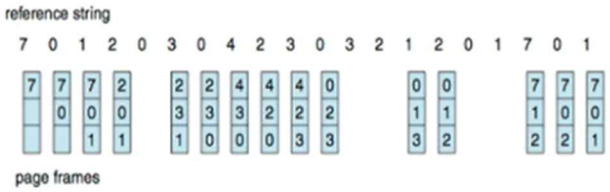
- *A* page replacement algorithm under evaluation
- *w* a given reference string
- *p*(*w*) probability of reference string *w*
- *len*(*w*) length of reference string *w*
- *m* number of available page frames
- *F*(*A*,*m*,*w*) number of page faults generated with the given reference string (*w*) using algorithm *A* on a system with *m* page frames.



Se si riesce ad avere un frame per ogni pagina → page fault tendono a zero, se invece c'è un solo frame: caso peggiore di pagefault → sempre pagefault.

FIFO → first in, first out

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 3 frames (3 pages can be in memory at a time per process)

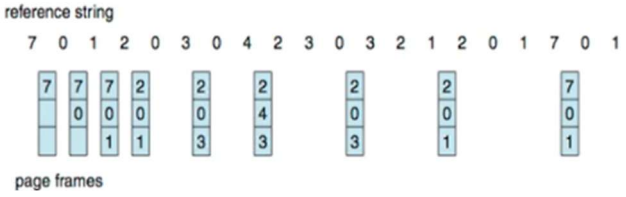


Alla quarta iterazione, la testa è 7, quindi sarà sostituito con 2 e la testa diventa 0. All'iterazioni 6 ho buttato fuori lo 0 che mi serve nella sette → shit.
→ 15 page fault

Se una pagina si trova in memoria da più tempo, è la vittima.
La stringa di riferimento è soggetta all'**anomalia di Belady** → aumentando il nr di frame, non è detto che migliora.

OPTIONAL Algorithm → previsione su futuro

Simuliamo una stringa già rilevata e facciamo si che l'algoritmo veda già il futuro.
Vittima: pagina che per più tempo non mi servirà



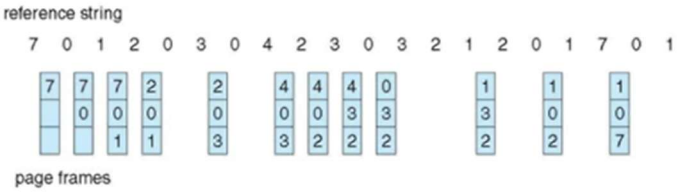
Alla quarta iterazione vede che 7 sarà quello usato tra più tempo e quindi butta fuori il 7. Etc.

L'algoritmo predittivo non è effettivamente realizzabile perché si può solo conoscere la storia passata e non le pagine che devono ancora arrivare (tranne nel caso di programmi molto semplici -senza if, jump, etc.).

LRU → Least Recently Used

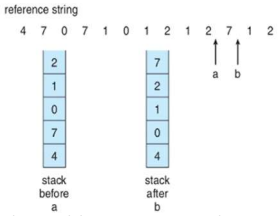
Per avvicinarsi all'algoritmo ottimo, usando il passato, si suppone che la storia passata si replichi nel futuro e quindi si può dire che è plausibile che la pagina a cui farò accesso più lontano nel futuro sarà quella a cui ho fatto accesso più lontano nel passato → ripetitività degli accessi. **→ la vittima è la pagina meno usata di recente.**

- Occorre conoscere la pagina che non viene acceduta da più tempo



→12 page fault

Questo algoritmo ha un costo eccessivo in quanto ogni volta che si fa accesso ad una pagina, si deve modificare il tempo di ultimo accesso e nel caso in cui c'è un page fault bisogna cercare la pagina a cui non si fa accesso da più tempo.
Esiste una versione **con implementazione a Stack** che migliora la ricerca: ma è abbastanza costoso.



A causa dell'elevato costo, si usa un algoritmo approssimato in cui ci si accontenta di sacrificare una pagina abbastanza lontana nel tempo anche se non è la più lontana. → serve un supporto HW → reference bit.

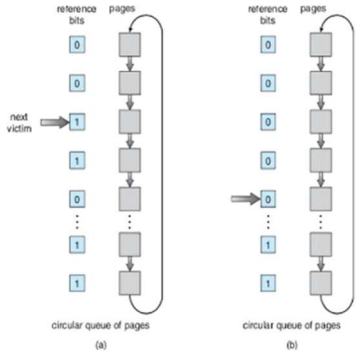
- *Reference bit a 1 → ho fatto accesso di recente*
- *Reference bit a 0 → non ho fatto accesso di recente*

Se siamo in grado di azzerare in certi istanti i reference bit, allora esiste un modo per determinare se una pagina ha fatto accesso abbastanza di recente o se non è usato da abbastanza tempo.

Algoritmo di **seconda opportunità**

Algoritmo di base FIFO + reference bit. La potenziale vittima viene guardata solo se ha reference bit a 0.

- Nel caso in cui abbia reference bit a 1: viene salvata e viene messo il reference bit a 0 → quindi se non si fa accesso in tempi brevi a questa pagina, probabilmente la prossima volta sarà vittima



Si possono fare delle varianti tenendo conto di bit di riferimento e bit di modifica cioè cercare se c'è una pagina che abbia sia reference che modified bit a 0 in modo tale da evitare il salvataggio.

Esistono algoritmi alternativi che invece di vedere l'ultimo accesso nel passato, guardano la frequenza di accessi in un certo intervallo. → una pagina mi serve se nell'ultimo tempo di riferimento l'ho usata tante volte, non mi serve se l'ho usata poche volte.

- **LFU** → Least Frequently Used → vittima: pagina usata meno frequentemente
- **MFU** → Most Frequently Used → vittima: pagina usata più frequentemente

Oltre alla scelta della vittima, **ci sono altri aspetti**:

Algoritmi di Page-Buffering

Cercano di gestire un pool di pagine libere → zona intermedia di pagine che sono vittime non ancora spostate su backing store → *cercano di ovviare al problema della vittima errata (ho scelto come vittima una pagina che mi serve poco dopo).*
Pool: insieme di pagine libere, sempre disponibili. L'obbiettivo è che ci sia sempre almeno una pagina libera nel pool. Dunque, la gestione del page fault non deve cercare una vittima.

- Si legge la pagina da disco nel free frame
- Si seleziona una vittima che viene selezionata e aggiunta al free frame
  - Quando ritenuto opportuno, i frame che precedentemente erano stati individuati come vittime, potranno essere salvati su disco.
- **Idea**: quando una pagina è scelta come vittima e aggiunta nel free pool frame, mantengo le informazioni che ci sono
  - Se faccio nuovamente riferimento a questa pagina → la riprendo senza dover andare sul disco (e senza averla salvata precedentemente su disco)

***Application** & page replacement*

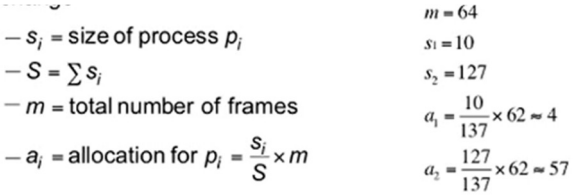
Se l'applicazione ha la possibilità di determinare la politica di assegnazione delle pagine, le cose potrebbero andare meglio in quanto l'applicazione, a differenza del sistema operativo, dovrebbe avere più chiaro cosa si sta facendo e in che modo.
Se l'applicazione si assume la responsabilità di partecipare al lavoro si rendere efficiente l'allocazione di pagine in memoria o di dati in memoria, allora il sistema operativo non fa certe azioni, altrimenti si rischierebbe di duplicare alcune azioni.

***Quanti frame** allocare per processo*

Ci sono istruzioni che hanno bisogno di più pagine e non solo 1. (es: possono servire 6 pagine)

2 schemi di allocazione:

- **Allocazione fissa**
  - EQUAL: Ad ogni processo attribuisco lo stesso numero di frame
    - 100 frames, 5 processi → 20 frames ciascuno.
  - PROPORZIONALE: conoscendo la dimensione del suo spazio di indirizzamento, provo ad attribuire RAM in modo proporzionale allo spazio del processo.



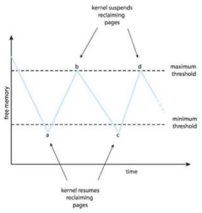
- **Allocazione prioritaria:**

Ricerca della vittima:

- **Rimpiazzamento globale**
  - Quando c'è bisogno di una sostituzione pagina, si cerca una vittima da ciascun processo
    - Un processo può rubare frame ad altri processi
- **Rimpiazzamento locale** (nel singolo processo)
  - Un processo può attingere frame solo tra quelli a lui dedicati
    - Potrebbe portare ad un sotto-utilizzo della memoria in quando magari si sono dati troppi frame ad un processo che effettivamente non ne ha bisogno e troppo pochi ad uno che ne ha bisogno

Riutilizzo/sfruttamento/**reclaming di pagine**

Per implementare una strategia di rimpiazzamento globale, tutte le richieste di memoria vengono gestite da una lista di free frames (che abbiamo supposto non si svuoti). La **ricerca** e il consecutivo rimpiazzo viene **attivata quando la lista scende sotto una soglia** (non 0 ma pochi) che continua finché la free list non arriva ad una certa soglia.

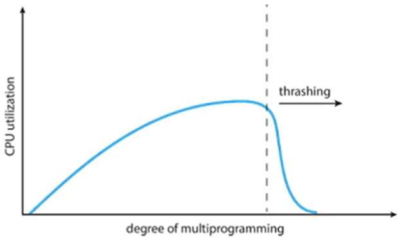


NUMA: Non-Uniform Memory Access

Nei sistemi multicore, tendenzialmente, più core (sullo stesso HW) hanno accesso ad una memoria condivisa. Globalmente la RAM può essere vista come se fosse partizionata in parti per ogni processore (non è così ma la si può immaginare così) accessibili da ogni processore a velocità differente. In questo contesto sarebbe opportuno che, quando un processo che girà su una cpu ha bisogno di frame, li prenda dalla ram più efficiente per lui → bisogna tenere conto del principio di località (es: Solaris crea degli lgroups che fanno in modo che ci sia una parentela tra processi e cpu su cui vengono eseguiti)

TRASHING:

Se l'**insieme dei processi non ha abbastanza frame,** allora continuerà a generare tanti page fault → frequenza di page fault alta → perdo prestazioni.



I processi lavorano con un modello di località → un processo ad un certo istante cerca di lavorare su certe pagine. Si va in trashing se la sommatoria delle località di tutti i processi (pagine su cui stanno lavorando ad un certo punto i processi) sono più della memoria disponibile. Σ size of locality > total memory size

Working-Set Model (provo a prevenire)

Modello che prova ad utilizzare il principio di località degli accessi.
Località degli accessi: pagine su cui lavoro ora e su cui lavorerò nell'immediato futuro (pagine su cui un processo a lavorato negli ultimi Δ istanti -sperando che le usi ancora)

- **Δ** = working-set window → intervallo di tempo
- **Working-set** → pagine su cui si è lavorato durante Δ
- **WSS** → working set size → nr di pagine su cui ha lavorato un processo durante Δ
- **D**= numero di frame richiesti decisi sommando i WSS di tutti i processi.

- Δ troppo piccolo → WS non comprende la località
- Δ troppo grande → WS comprende troppe cose

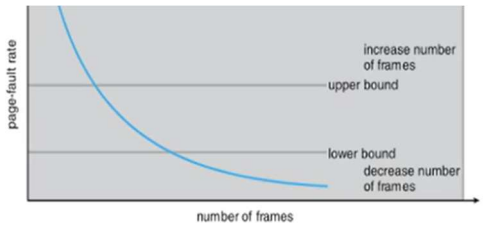
Dopo ogni accesso, bisogna allineare il working set all'insieme dei frame residenti → troppo costoso. Quindi, si cerca una **strategia che approssimi questa funzione**. Se Δ è 10000, allora **ogni 5000 istanti** eseguo:

- Per ogni pagina salvo il valore del reference bit da qualche parte
- Per ogni pagina azzero il reference bit
- Durante i prossimi 5000 istanti se si accede alla pagina, si mette ad 1 il reference bit
- Alla fine di questi 5000 istanti
  - Vedendo i reference bit si discriminano le pagine su cui ho fatto accesso negli ultimi 5000 istanti e quelle a cui non ho fatto accesso negli ultimi 5000 istanti
  - Vedendo il valore che mi ero salvato posso vedere se si era fatto accesso negli altri 5000 istanti precedenti → dunque ho coperto 10000 istanti (Δ).

Page-Fault Frequency (guardo effetti e reagisco)

Stabilisce il **numero di frames in base ai page-fault**:

- Se si hanno troppi page fault → aumenta numero di frame
- Se ha pochi page fault → diminuisci numero di frame



Quando avviene un page fault, misuro la distanza in tempo dall'ultimo page fault τ. Definisco una costante c (in realtà ne sono due):

- **τ < c** → frequenza di page fault è alta
  - **aggiungo un frame** al resident set e ci inserisco la pagina che ha causato page fault
- **τ ≥ c** → frequenze di page fault bassa
  - **sostituzione** pagina (metto dentro quella cha ha causato page fault e tolgo le pagine a cui non abbiamo fatto accesso a partire dall'ultimo page fault – pagine con reference bit a 0)

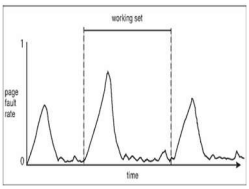
Tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Stringa	**6**	**4**	**4**	**3**	**2**	**4**	**4**	**4**	**1**	**3**	**3**	**2**	**4**	**4**	**5**	**1**	**2**	**2**	**2**	**6**	**1**	**2**	**5**	**4**
Frame 0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	5	5	5	5	6	6	6	6	6
Frame 1	-	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	2	2	2	2	2
Frame 2	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1
Frame 3	-	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-	-	-	5	5
Frame 4	-	-	-	-	-	-	-	-	1	1	1	1	1	1	-	1	1	1	1	-	-	-	-	4
Fault	*	*		*	*				*						*	*				*			*	*
Ref Bit								0						0					0					

NOTA: il reference bit viene azzerato ad ogni sostituzione pagina e viene settato ad 1 se avviene accesso a pagina.

NOTA2: quando t=15, 6 e 1 (che hanno reference bit a 0) vengono tolte

C=3
Page referenced
victim

Il working set (pagine su cui ha lavorato il programma) e le frequenze di page fault hanno un andamento non uniforme. Si alternano momenti con tanti page fault e momenti con bassi page fault.



Oltre alla memoria da allocare ai processi utente, bisogna allocare della memoria a kernel e sistema operativo.

Per il kernel ci deve essere un free-memory-pool da cui il kernel potrebbe fare richiesta di èagine per allocazione continua (es: tabella delle pagine)

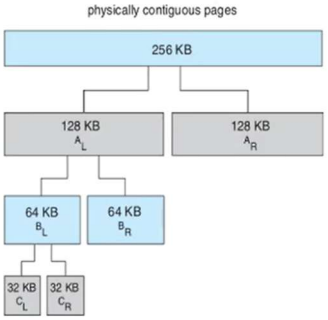
Allocatori per il kernel

Spesso allocati in un memory pool dedicato al kernel

1.Buddy System

Prova ad allocare in modo continuo tentando di ridurre il problema della frammentazione esterna. Cerca di sfruttare alcuni vantaggi della paginazione:

- Alloca continuo di tagli abbastanza uniformi
  → **si arrotonda alla potenza di 2** (per eccesso) di ciò che serve.
- Nota: si accetta frammentazione interna e si limita frammentazione esterna
- Se si ottiene un pezzo più grosso di quello necessario, si prova a limitare i danni
  - Se ho chunk da 256KB e mi serve 21KB:
    - Divido in 128(AL) e 128(BL)
    - Divido AL in 64BL e 64BR
    - Divido sx in 32CL e 32CR
    - Occupo solamente CL



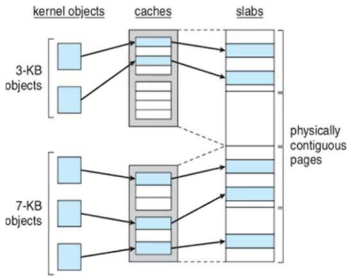
For example, assume 256KB chunk available, kernel requests 21KB
- Split into A_L and A_R of 128KB each
  - One further divided into B_L and B_R of 64KB
    - One further into C_L and C_R of 32KB each – one used to satisfy request

2.Slab Allocator

Per un tipo di struttura dati del kernel si cerca di allocare una cache:

- **Cache**: 1+ slab
- **Slab**: 1+ pagine continue

Quando si crea una cache la si riempire di oggetti liberi che vengono allocati quando si usano per una struttura dati del sistema.



Nota: I primi accessi di memoria generano page-fault, dunque può essere utile prepaginare.

Prepaginare: portare in frame alcune pagine di cui il processo avrà bisogno, prima che si faccia riferimento ad esse.

- C'è il rischio di prepaginare qualcosa che non serve
- Bisogna far in modo che "*a*" tenda ad 1

Può esser fatto:

- Alla startup
- Mentre sto protando dentro una pagina, porto dentro anche la vicina perché probabilmente sarà usata.

Assume *s* pages are prepaged and *a* of the pages is used
- Is cost of ***s * a*** save pages faults > or < than the cost of prepaging
  s * (1- a) unnecessary pages?
- ***a*** near zero ⇒ prepaging loses

Dimensione delle pagine:

Sempre potenze di 2, tra 4kB e 4MB

- Frammentazione (meglio pagine piccole)
- Dimensione tabella delle pagine (meglio pagine grandi)
- I/O Overhead (meglio pagine grandi)
- Numero di page fault (meglio pagine grandi)
- Località di accessi (± dipende dal programma in base al fatto se il programma ha tanti salti in altre pagine o meno)
- Dimensione TLB ed efficacia (meglio pagine grandi)

Copertura della TLB:

Dipende da: ■ TLB Reach = (TLB Size) X (Page Size)

- Nr righe (pagina --- frame)
- Dimensione della pagina
  - Se aumenta però, aumenta frammentazione

Avere pagine di dimensione multipla diminuisce frammentazione ma può creare complicazioni nella gestione delle pagine (possiamo avere sia pagine grandi che piccole).

TLBreach: quante pagine riesce a vedere senza fare accesso in memoria.

Nota: una matrice è allocata per righe → se devo fare un programma che deve percorrere una matrice conviene fare in modo che vada per righe altrimenti molti page fault.

- Program 2

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

128 page faults

Program structure

- `int[128,128] data;`
- Each row is stored in one page
- Program 1

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

128 x 128 = 16,384 page faults

I/O Interlock → quando una pagina è coinvolta in un'operazione di I/O, la pagina va bloccata e non si può fare rimpiazzamento di questa pagina → PINNING.