10. Memoria Virtuale

- Concetti di base
- Paginazione su richiesta (Demand Paging)
- Sostituzione delle pagine
- Algoritmi di Sostituzione
- Allocazione dello spazio in Memoria Principale
- Thrashing
- Altre considerazioni
- Casi reali

10.1 Memoria Virtuale: introduzione

- I metodi di gestione della MP tentano di tenere in RAM più processi possibile per aumentare la multiprogrammazione

- Tuttavia, per una certa quantità di RAM disponibile in un sistema, il numero di processi che possono stare in MP dipende dalla loro dimensione.

- La **Memoria Virtuale (MV)** è un insieme di tecniche per permettere l'esecuzione di processi in cui codice e/o dati **non sono caricati completamente** in Memoria Primaria

10.1 Memoria Virtuale: introduzione

- la MV funziona perché i programmi non hanno bisogno di essere caricati interamente in MP per essere eseguiti.
  Ad esempio:
  - il codice per trattare condizioni di errore potrebbe non essere mai usato, in una data esecuzione del programma
  - array, liste, tabelle sono spesso dichiarate di dimensioni maggiori di quanto poi effettivamente usato
  - alcune opzioni di programma sono raramente usate
  - le librerie dinamiche vengono caricate in RAM solo se e solo quando sono effettivamente usate

10.1 Memoria Virtuale: introduzione

- L'idea di fondo della Memoria Virtuale è quindi la seguente:

- **carichiamo in MP un pezzo di programma solo *se* questo deve effettivamente essere eseguito, e solo *quando* deve essere eseguito.**

- **Analogamente, carichiamo in MP solo la porzione di strutture dati che vengono usate in una certa fase dell'esecuzione del programma.**

10.1 Memoria Virtuale: Vantaggi

- L'ovvia conseguenza di questo modo di procedere è che si possono eseguire programmi più grandi della MP. Più formalmente:
  possiamo eseguire un processo che sfrutta uno spazio di indirizzamento logico maggiore di quello fisico.
- **possiamo avere contemporaneamente in esecuzione processi che, assieme, occupano più spazio della MP disponibile**
- Come conseguenza del punto precedente, aumenta il grado di multiprogrammazione, e quindi il throughput della CPU
- I programmi partono più velocemente, perché non dobbiamo caricarli completamente in memoria primaria

10.1 Memoria Virtuale: Svantaggi

- Naturalmente, vi sono anche degli inconvenienti, che analizzeremo in dettaglio più avanti:

1. innanzi tutto, si verifica normalmente un aumento di traffico tra la RAM e l'Hard disk.
2. L'esecuzione del singolo programma può richiedere più tempo che se la memoria virtuale non fosse implementata
3. In alcune situazioni particolari, le prestazioni complessive del sistema possono degradare drasticamente, un fenomeno noto come **thrashing**.

10.2.1 Paginazione su Richiesta (Demand Paging)

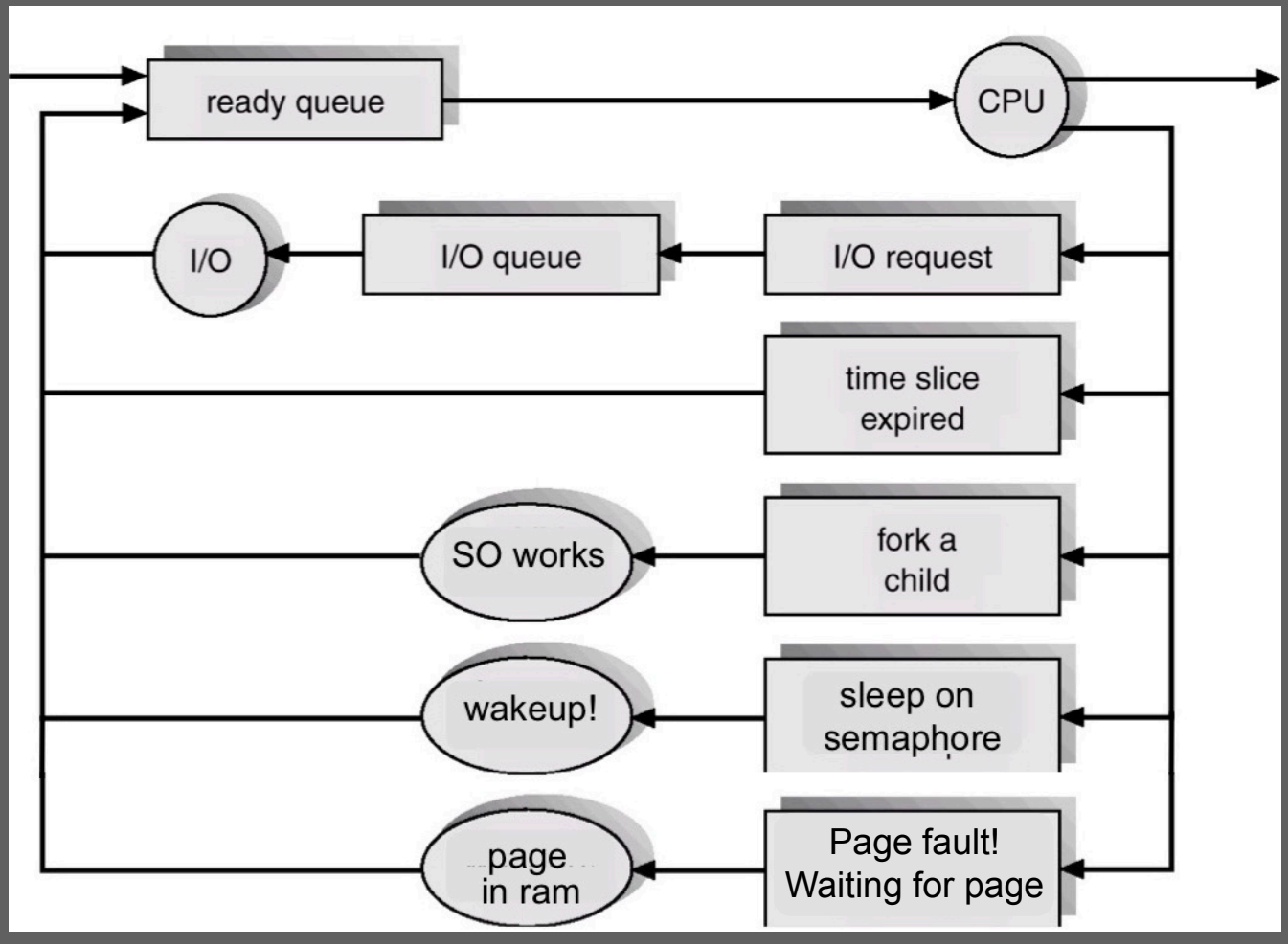
- Idea di base: portare una pagina in MP solo al momento del primo indirizzamento di una locazione (un dato, una istruzione) appartenente alla pagina stessa.
- Quando la CPU esegue un'istruzione che indirizza una locazione di RAM in una pagina diversa da quella che contiene l'istruzione in esecuzione, e la pagina non è in MP, si dice che il processo ha generato un **page fault**
- il SO deve 1) sospendere il processo, 2) portare in memoria la pagina mancante e, 3) quando sarà il suo turno, far ripartire il processo dal punto in cui era stato sospeso.

10.2.1 Demand Paging

- Più in dettaglio, quando manca la pagina riferita:
  - il processo viene tolto dalla CPU e messo in uno stato di "**waiting for page**"
  - Un modulo del SO detto **pager** inizia il caricamento della pagina mancante dalla MS in un frame libero della MP
  - In attesa di completare il caricamento, la CPU viene assegnata ad un altro processo.
  - Quando la pagina è in MP, il processo corrispondente è rimesso in Coda di Ready: riprenderà l'esecuzione dall'istruzione che aveva causato il problema quando sarà scelto dallo scheduler.

3.2.1 Code di scheduling

- Ora, nel **Diagramma di accodamento** del capitolo 3, il caso *wait for an interrupt* lo possiamo anche associare ad un processo *waiting for page* (fig. 3.4 modificata).



10.2.1 Demand Paging

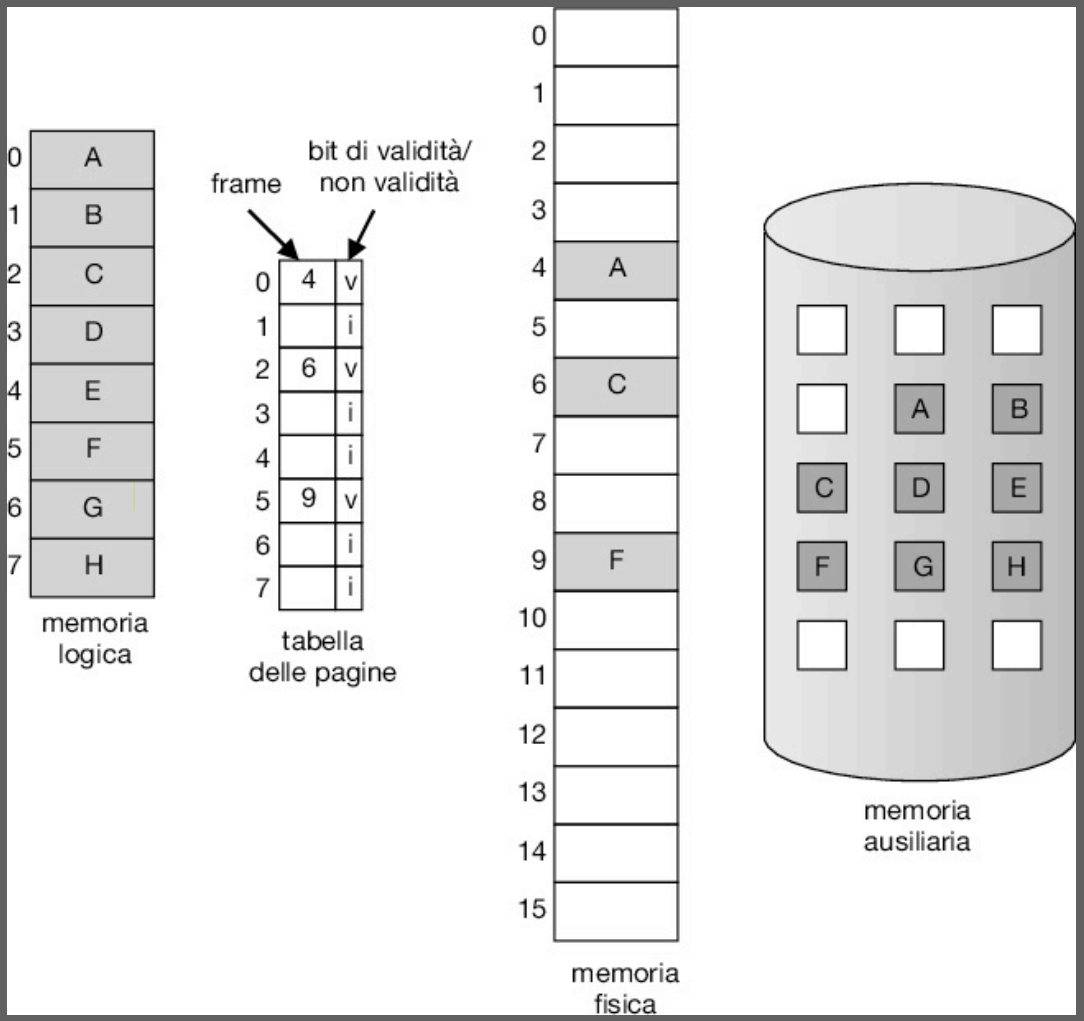
- Ma come fa la CPU a sapere se una pagina non è caricata in RAM? Può usare un **bit di validità** della pagina associato ad ogni entry della PT, che dice se la pagina associata a quella entry è effettivamente in RAM o no.

- se si tenta di fare riferimento ad una pagina non in MP il suo bit di validità sarà a 0, e verrà generata una trap detta **page fault** che fa intervenire il SO e partire il meccanismo descritto nella slide precedente

- quando la pagina mancante è arrivata in MP, il suo bit di validità è messo a 1 e la PT aggiornata: il processo può ripartire dall'istruzione causa del page fault.

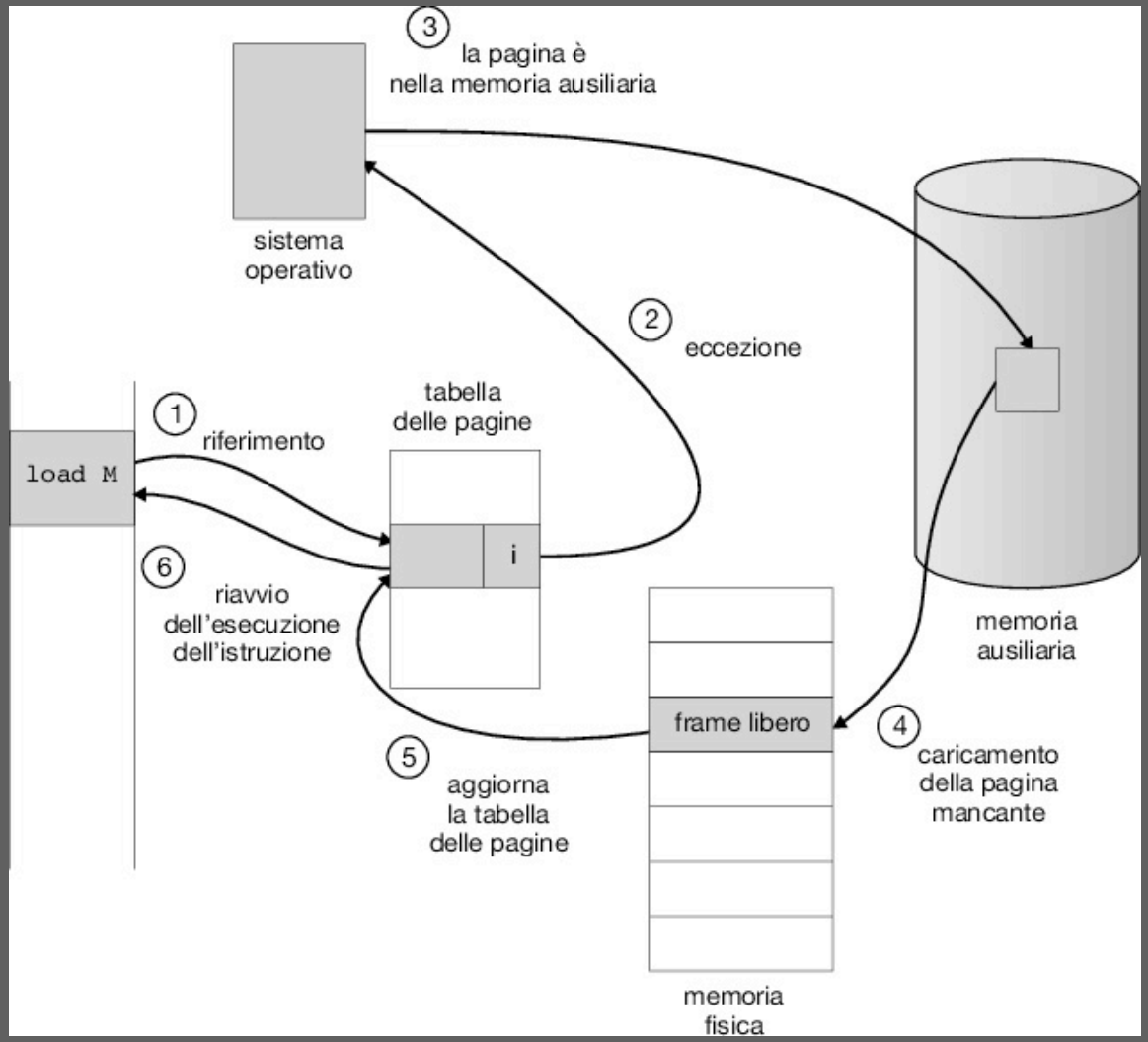
10.2.1 Demand Paging

- Tabella delle pagine: alcune non sono in RAM (fig. 10.4):



10.2.1 Demand Paging

- Fasi di gestione di un page fault (fig. 10.5):



10.2.1 Demand Paging

- Notate che un processo può addirittura essere fatto partire senza nessuna delle sue pagine in MP.

- Alla prima istruzione indirizzata dal PC (inizializzato dal SO) si genera un page fault: il Program Counter punta all'indirizzo di una pagina del processo non in MP:
  - Questo schema è detto **Pure Demand Paging**

- In alternativa, il SO può caricare inizialmente in MP *almeno* la pagina che contiene la prima istruzione da eseguire.

10.2.1 Demand Paging: supporto Hardware necessario

- Per implementare la memoria virtuale è necessario un certo supporto hardware. In particolare:
- la tabella delle pagine deve avere il bit di validità testabile dall'hardware per generare il page fault
- Le istruzioni devono essere ri-eseguibili dopo un page fault, oppure l'hardware della CPU deve controllare che tutti gli operandi di una istruzione siano presenti in MP prima di poter eseguire l'istruzione
- quindi: mentre la *paginazione* può essere aggiunta a qualsiasi sistema, la *paginazione su richiesta,* e più in generale la memoria virtuale, richiedono HW specifico

10.2.3 Prestazioni della paginazione su richiesta

- Supponiamo di dover leggere un dato in MP.
- **ma** = tempo di accesso in MP se il dato è presente (ad esempio, **ma** ≈100÷200 nanosec)
- **p** = probabilità di un page fault
- **eat** (**effective access time**) =
  = [(1-**p**) × **ma**] +
  [**p** × tempo di gestione del page fault]

10.2.3 Prestazioni della paginazione su richiesta

- l'elenco completo dei passi necessari a gestire un page fault lo trovate sul testo, ma le tre operazioni principali da compiere sono:
  1. gestire il page fault (1 ÷ 100 μ sec)
  2. recuperare la pagina mancante dalla MS (≈ 8 millisecondi. Questo valore varia da sistema a sistema, ma se la memoria secondaria è su Hard Disk, il valore è sempre dell'ordine di qualche millisecondo)
  3. riavviare il processo (1 ÷ 100 μ sec)
- Evidentemente, i punti 1 e 3 sono trascurabili rispetto a 2

10.2.3 Prestazioni del paginazione su richiesta

- Ponendo **ma** = 200 nanosec, abbiamo (valori espressi in nanosecondi):
- **eat** = (1-**p**) × 200 + **p** × 8.000.000 =
  = 200 + **p** × 7.999.800
- Se **p** = 0.001 (un page fault ogni 1000 accessi), si ha: **eat** = 8.199,8 (circa 8,2 microsecondi)
- L'esecuzione rallenta di più di 40 volte!!!

10.2.3 Prestazioni della paginazione su richiesta

- Se volessimo un degrado massimo del 10%, allora **p** deve essere tale che

  eat = $220 > 200 + 8 \times 10^6 \times \mathbf{p}$

  $20 > 8 \times 10^6 \times \mathbf{p}$

  $\mathbf{p} < 2{,}5 \times 10^{-6}$

- ossia, non più di 1 page fault ogni 400.000 accessi in memoria.

- **Ma 400.000 accessi in RAM sono pochi o tanti?**
  Se un processo esegue in tutto un milione di istruzioni, quanti accessi in RAM genera, approssimativamente?

10.2.3 Prestazioni della paginazione su richiesta

- il numero di page fault deve quindi essere molto molto basso altrimenti il tempo medio di esecuzione dei processi aumenta in modo inaccettabile, e il throughput del sistema peggiora anziché migliorare.
- Si può ovviamente agire anche sul tempo di gestione del page fault. Ad esempio, usando pagine di grandi dimensioni, si hanno in media meno page fault (perché?)
- E poi si può ottimizzare l'accesso alla MS, anche se più di tanto non si può fare, visto che dobbiamo comunque usare l'hard disk (ma si veda anche la parte sulle memorie a stato solido del capitolo 11)

10.2.3 L'area di swap

- Per funzionare, la memoria virtuale ha bisogno di una opportuna porzione dell'HD detta **area di Swap**
- All'installazione, sull'HD quest'area viene riservata come porzione del disco ad uso esclusivo del SO.
- L'area di swap è gestita con meccanismi più semplici ed efficienti di quelli usati per il File System
- in particolare, le pagine dei processi non sono scritte dentro a dei file, in modo da evitare l'uso dei file descriptor, e spesso si usano blocchi più grandi con allocazione solo contigua (capiremo meglio questo punto nella parte sulla gestione della memoria secondaria)

10.2.3 L'area di swap

- Il modo più semplice in cui viene usata l'area di swap è il seguente: all'avvio di un processo, il suo eseguibile viene copiato interamente nell'area di swap:
  - il tempo di avvio aumenta
  - abbiamo bisogni di aree di swap grandi
  - ma migliora il tempo di gestione dei page fault, perché ci vuole meno tempo per recuperare le pagine in MS una volta che sono nell'area di swap, perché non si deve più passare attraverso il file system

10.2.3 L'area di swap

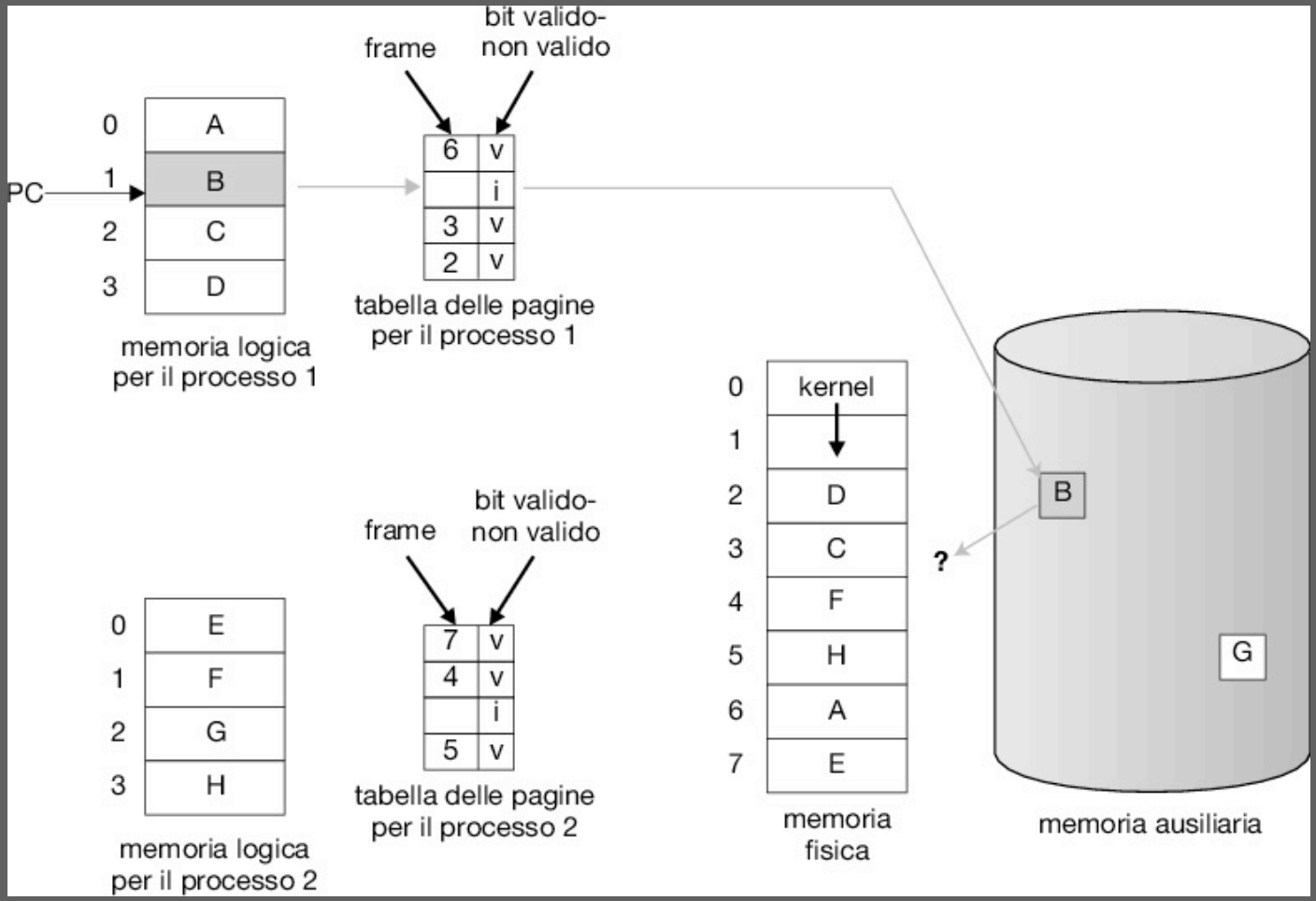
- In alternativa, dobbiamo prelevare le pagine dell'eseguibile, o di un eventuale file di dati direttamente dal file system:
  - può essere necessario se dobbiamo limitare le dimensioni dell'area di swap
  - l'avvio dei processi è più veloce
  - ma la loro esecuzione può risultare più lenta
- Ma, notate, la swap area non sembra essere molto utile se viene usata solo per copiarvi dentro gli eseguibili (ed eventuali dati in input) prima di far partire i processi...

10.2.3 L'area di swap

- **l'area di swap viene usata soprattutto per liberare spazio in memoria primaria e ospitare pagine mancanti che devono essere caricate in RAM perché la loro assenza ha provocato un page fault**
- Infatti, se ci fosse sempre un frame libero, una pagina sarebbe caricata in MP solo la prima volta in cui quella pagina viene indirizzata
- Ma l'idea della memoria virtuale è proprio di:
  - poter eseguire un processo più grande della memoria primaria disponibile.
  - Eseguire contemporaneamente processi che, assieme, occupano più spazio di quello disponibile in RAM

10.2.3 L'area di swap

- Due processi occupano più dello spazio disponibile in memoria principale (fig. 10.9)



10.2.3 L'area di swap

- Dunque, se si verifica un page fault e tutti i frame della RAM sono occupati, occorre liberarne uno rimuovendo la pagina che ospita, che prende il nome di **pagina vittima**.

- Se la pagina vittima contiene dati modificati e fa parte dello stack o della heap di un processo, la pagina va salvata nell'area di swap, in modo che possa essere recuperata quando il processo a cui appartiene vi farà riferimento

- Le pagine di codice non devono essere salvate, tanto ce n'è comunque una copia nel file system, ma se erano state copiate inizialmente nell'area di swap potranno poi essere recuperate più velocemente se riferite di nuovo

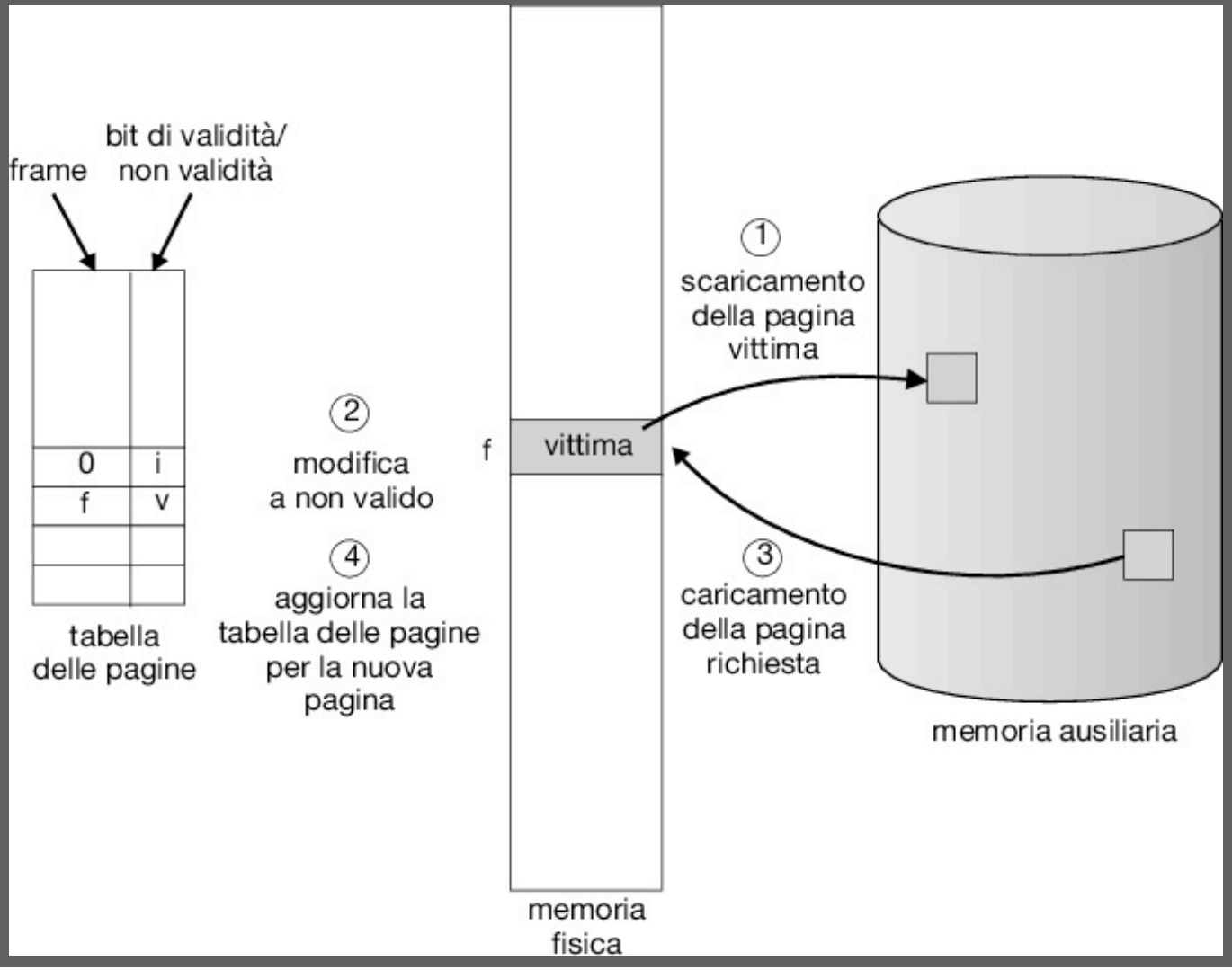
10.4 Sostituzione delle pagine

- Dunque, se si verifica un page fault e non c'è alcun frame libero in RAM:
  - il SO sceglie una pagina "**vittima**" da rimuovere;
  - salva la pagina vittima nella swap area (se necessario);
  - carica nel frame liberato la pagina mancante.

- Notiamo le somiglianze con il concetto di swapping di processi. Ma qui ci limitiamo spostare dalla RAM all'hard disk e viceversa solo pezzi dei processi, una pagina alla volta.

10.4.1 Sostituzione di pagina

- Sostituzione di una pagina in caso di page fault (fig. 10.10):



10.4.1 Sostituzione di pagina

- Dunque, se la pagina vittima non è stata modificata da quando era arrivata in RAM ne abbiamo già una copia in memoria secondaria, e possiamo evitare di salvarla.
- Ma se la pagina vittima va salvata, il tempo di gestione del page fault raddoppia:
  (1) salva la pagina vittima, (2) carica la nuova pagina
- Un **dirty bit** associato ad ogni entry della PT che ci dica se la pagina relativa è stata modificata può aiutarci: viene settato a 1 dall'hardware della CPU la prima volta che una pagina in RAM viene acceduta in scrittura.
- Così dobbiamo salvare in memoria secondaria solo le pagine vittima che hanno il dirty bit settato a 1.

10.4.1 Sostituzione di pagina

- Notate che solo pagine di dati/stack/heap possono essere modificate, e quindi avere il dirty bit a 1.
- Se una pagina di dati con il dirty bit a 1 viene scelta come pagina vittima, va salvata nell'area di swap per non perdere le modifiche effettuate sui dati che contiene.
- Le pagine di codice sono accedute solo in lettura, e non vanno salvate nell'area di swap se scelte come pagine vittima.
- Inoltre, se il codice era stato inizialmente copiato nell'area di swap, riportare in RAM le pagine di codice rimosse risulta molto più veloce che prelevarle dall'eseguibile di cui fanno parte memorizzato nel file system

10.4.1 Sostituzione di pagina

- La sostituzione delle pagine è essenziale per permettere l'esecuzione di programmi più grandi della MP disponibile!
- Nascono però due problemi:
  - **Scelta della pagina da sostituire**: quale pagina scegliamo come vittima?
  - **Allocazione dei frame**: quanti ne diamo a ciascun processo? (su cui torneremo più avanti)
- Il tipo di soluzione adottato per questi problemi può influenzare moltissimo il tempo di esecuzione dei processi

10.4.1 Algoritmi di sostituzione delle pagine

- Se una pagina vittima appena rimossa viene di nuovo indirizzata dal processo a cui appartiene si genera un page fault: la pagina deve essere ricaricata in memoria principale: **abbiamo sprecato un sacco di lavoro!!!**
- Viceversa, se scegliamo una pagina vittima che non verrà mai più indirizzata non dovremo più preoccuparci di ricaricarla in memoria principale
- **Un buon algoritmo di sostituzione MINIMIZZA IL NUMERO DI PAGE FAULT**
- Nota: in letteratura, a volte questi algoritmi sono detti "algoritmi di *rimpiazzamento* delle pagine"

10.4.1 Algoritmi di sostituzione

- Come possiamo valutare la bontà di diversi algoritmi di sostituzione delle pagine?
- Possiamo usare delle **sequenze di riferimenti** in memoria principale:
  - generate in modo casuale, oppure
  - generate dall'esecuzione di programmi reali
- Notate che non ci interessa l'indirizzo preciso generato dalla esecuzione di una istruzione, ma solo il numero della pagina indirizzata (possiamo cioè ignorare l'offset)

10.4.1 Algoritmi di sostituzione

- Ad esempio, una sequenza di riferimento potrebbe essere la seguente:
  - 10, 7, 4, 5, 6, 1, 10, 4, ...

- Ossia, durante l'esecuzione delle istruzioni di un processo, la CPU ha generato una sequenza di indirizzi logici che indirizzano (qualcosa contenuto nel)la pagina 10, poi la pagina 7, la 4, e così via.

10.4.1 Algoritmi di sostituzione

- ma quanti page fault genera questa sequenza di riferimento?
  - 10, 7, 4, 5, 6, 1, 10, 4
- Dipende dal numero di frame disponibili. Se (per ipotesi un po' estrema) abbiamo una memoria con un solo frame disponibile (ad esempio tutti gli altri sono occupati dal codice del SO), la sequenza provoca 8 page fault.
- e quanti page fault genera questa sequenza, sempre con un solo frame a disposizione?
  - 10, 7, 7, 7, 4, 5, 5, 5, 5, 6, 1, 10, 10, 10, 10, 4

10.4.1 Algoritmi di sostituzione

- Sempre 8 page fault. Infatti, dopo aver caricato in MP la pagina 7 (ad esempio), tutti i successivi riferimenti **consecutivi** alla stessa pagina non provocano ulteriori page fault.
- Quindi, le sequenze:

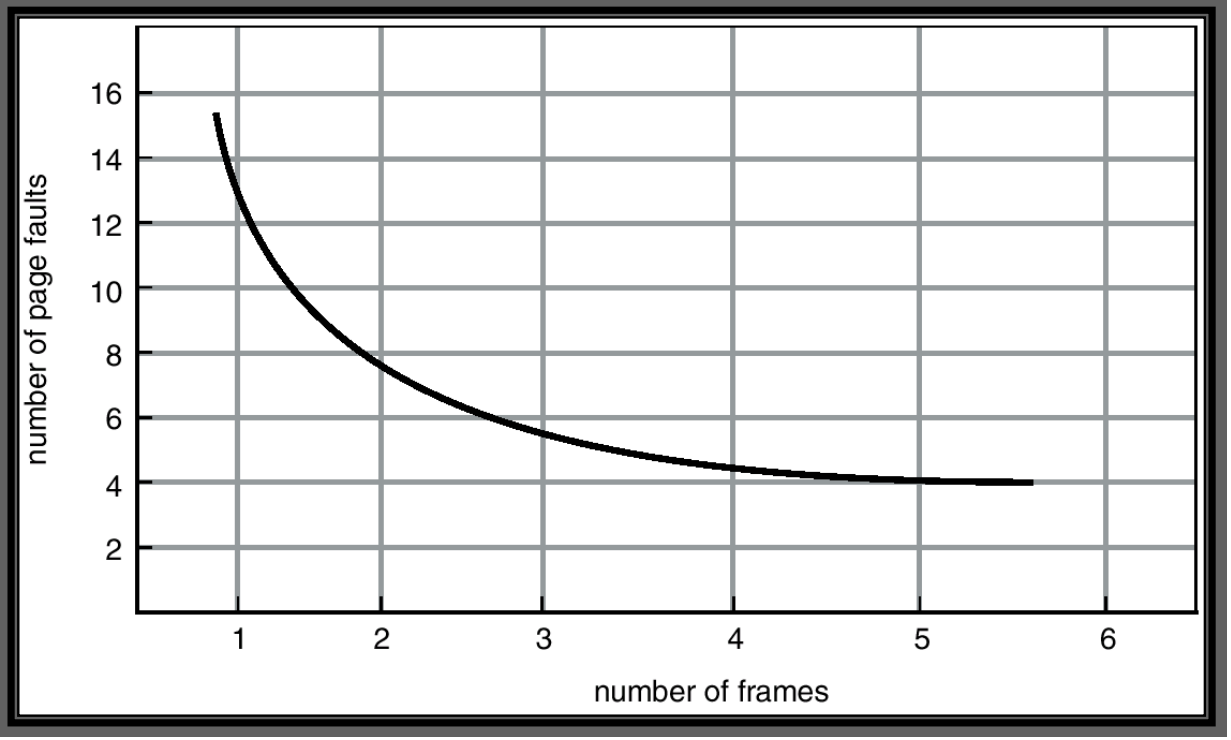
  "10, 7, 4, 5, 6, 1, 10, 4" e

  "10, 7, 7, 7, 4, 5, 5, 5, 5, 6, 1, 10, 10, 10, 10, 4"

  sono da considerarsi equivalenti per li nostro scopo di valutare la bontà di un algoritmo di sostituzione

10.4.1 Algoritmi di sostituzione

- Il numero di page fault per una data sequenza e algoritmo di sostituzione dipende anche dal numero di frame disponibili.
- Intuitivamente, maggiore è il numero di frame disponibili e minore è il numero di page fault (ma non sempre, come vedremo...)
- Fig. 10.11



10.4.1 Algoritmi di sostituzione

- Nel seguito assumeremo implicitamente che ad ogni processo è assegnato un certo numero prestabilito di frame, e una pagina vittima è selezionata tra le pagine del processo stesso. Si parla allora di:

  sostituzione LOCALE delle pagine

- Assumeremo anche uno schema di paginazione su richiesta **puro:** quando un processo parte nessuna sua pagina è in RAM, e dunque il primo riferimento a qualsiasi sua pagina genera un page fault

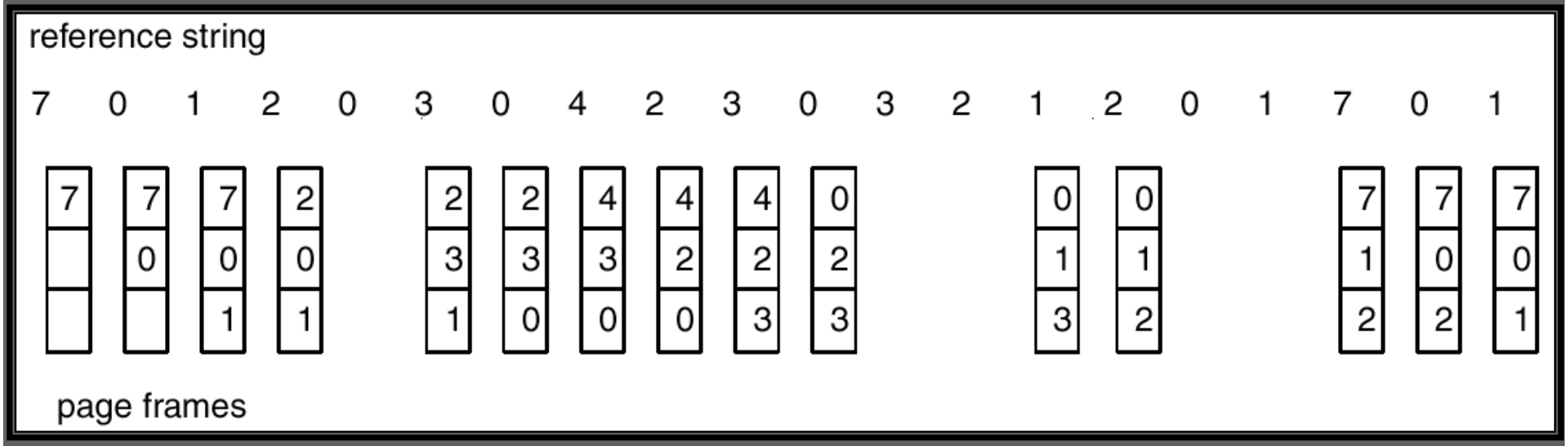
10.4.2 Sostituzione delle pagine secondo l'ordine d'arrivo (FIFO)

- Questo algoritmo sceglie come pagina vittima quella arrivata da più tempo in memoria principale
- E' un algoritmo facile da implementare, ma non fornisce necessariamente buone prestazioni, infatti:
  - se la pagina vittima contiene del codice di inizializzazione del processo usato solo all'inizio allora va bene, non servirà più e possiamo rimuoverla dalla MP.
  - ma se la pagina vittima contiene una variabile inizializzata all'inizio e usata per tutta l'esecuzione del codice, oppure una procedura richiamata spesso, non conviene rimuoverla dalla MP

10.4.2 Algoritmo FIFO: esempio

- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

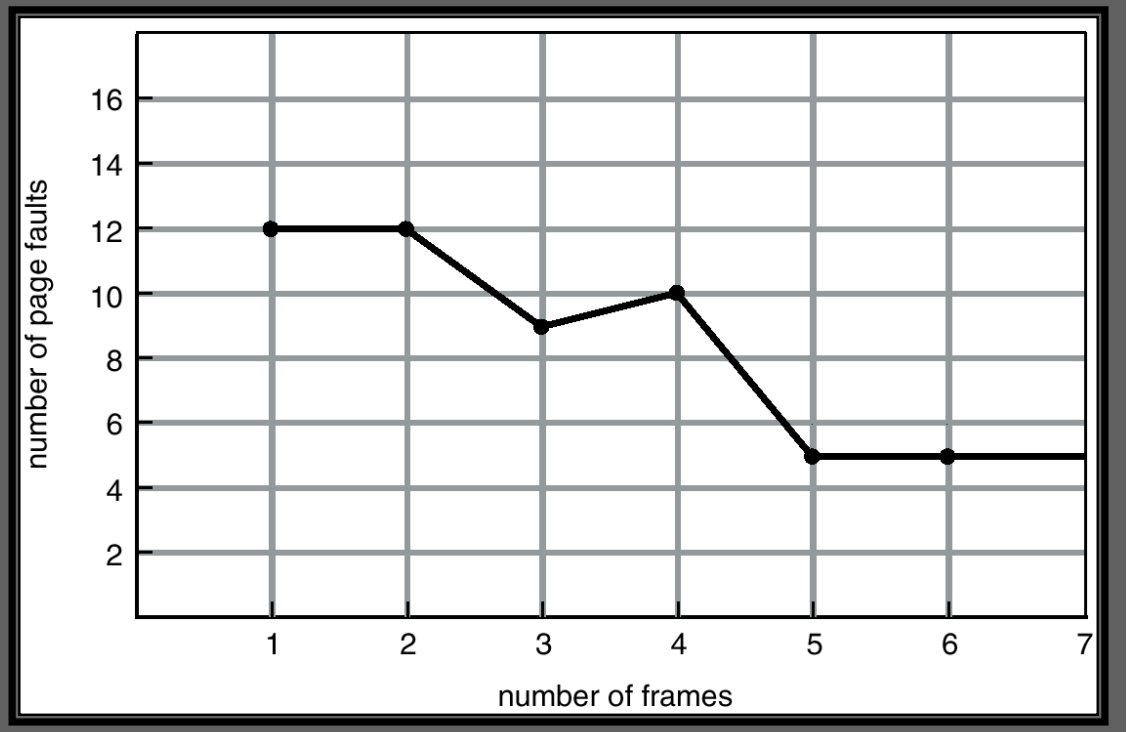
  usata con 3 frame produce 15 page fault (fig. 10.12):



10.4.2 FIFO: l'anomalia di Belady

- FIFO soffre della cosiddetta **Anomalia di Belady:** usando più frame, il numero di page fault può aumentare!!!

- stringa di riferimenti:
  1 2 3 4 1 2 5 1 2 3 4 5
  (fig. 10.13):



- N.B.: succede solo per alcune stringhe di riferimento, non per qualsiasi stringa

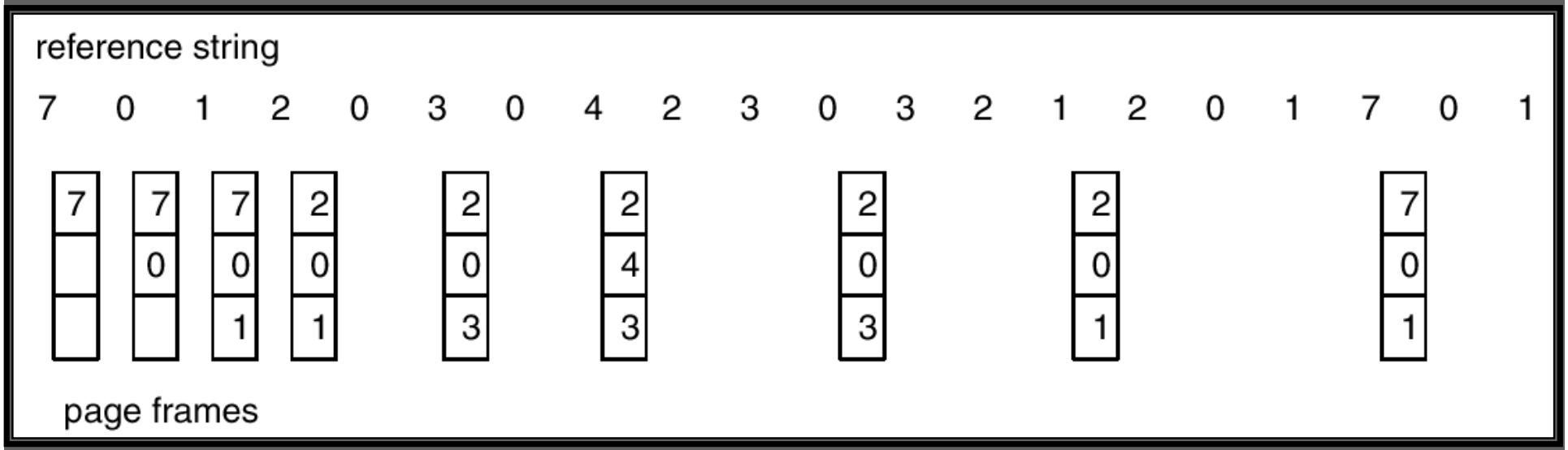
10.4.3 Sostituzione ottimale delle pagine

- E' un algoritmo che non soffre dell'anomalia di Belady e che, soprattutto:
  - produce il numero minimo di page fault per un certo numero di frame disponibili
- **Algoritmo OPT (o MIN): la pagina vittima è quella che sarà usata più in là nel tempo.**
- Ma ovviamente OPT non è implementabile, ed è usato solo come termine di paragone per misurare le prestazioni di altri algoritmi

10.4.3 Sostituzione ottimale delle pagine: esempio

- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

usata con 3 frame produce 9 page fault (fig. 10.14):



10.4.4 Sostituzione delle pagine LRU (Least Recently Used)

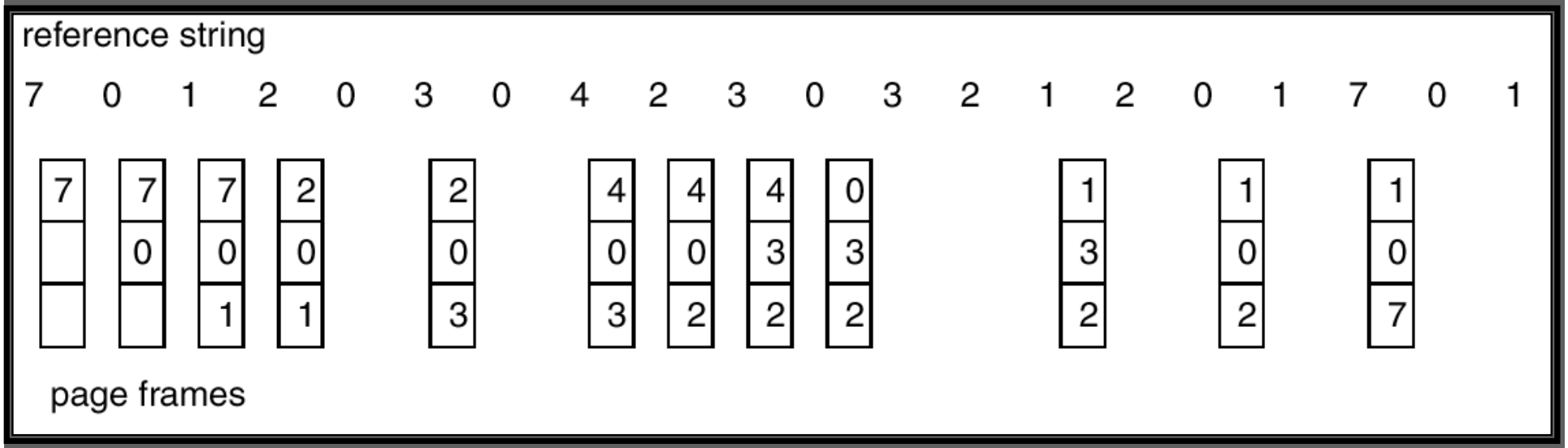
- Cerca di approssimare OPT sostituendo la pagina che **non è stata usata da più tempo**
- Rispetto a OPT, guarda all'indietro nel tempo anziché che in avanti: almeno il passato lo conosciamo!
- LRU è un buon algoritmo di sostituzione, non soffre dell'anomalia di belady e si avvicina più a OPT che a FIFO

ma è difficile da implementare in modo efficiente

10.4.4 Sostituzione delle pagine LRU: esempio

- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

  usata con 3 frame produce 12 page fault (fig. 10.15):



10.4.5 Approssimazioni di LRU

- Per poter implementare l'algoritmo di sostituzione LRU ci sarebbe bisogno di un supporto hardware da parte del processore che non è disponibile nelle CPU moderne

- Molti processori forniscono tuttavia un semplice supporto HW che permette di implementare una approssimazione più che accettabile di LRU:

- **Reference bit**: un bit associato ad ogni pagina nella Page Table di un processo.

10.4.5 Approssimazioni di LRU

- Quando un processo parte, i reference bit delle sue pagine sono tutti inizializzati a 0 dal sistema.
- Quando viene indirizzata una pagina (in lettura o in scrittura) l'hardware mette a 1 il reference bit di quella pagina.
- Quindi, in ogni istante, possiamo sapere quali pagine di un processo sono state usate e quali no.
- Non possiamo sapere in quale ordine è stato fatto riferimento alle pagine usate,
  ma sappiamo che sono state riferite più di recente delle pagine con il reference bit ancora a 0

10.4.5.2 Approssimazione di LRU: Algoritmo della Seconda Chance

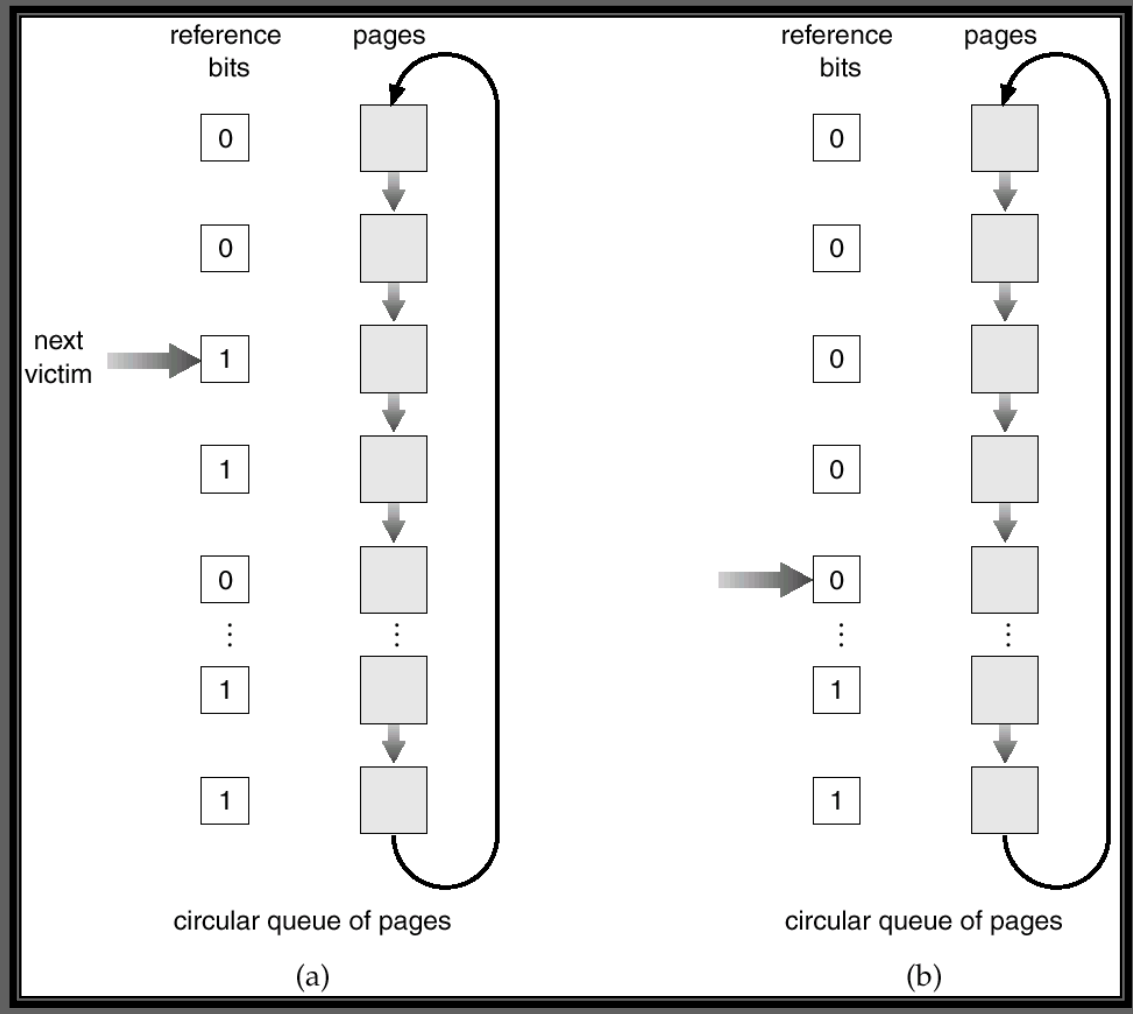
- Si parte da un algoritmo FIFO: in caso di page fault il SO esamina la pagina entrata in RAM da più tempo
- **se il reference bit della pagina è a 0**, allora quella diventa la pagina vittima: è da molto tempo in RAM e non è stata riferita di recente
- **Se il reference bit della pagina è a 1**, il SO le da una **seconda chance:**
  - Il suo reference bit viene azzerato, e la pagina viene trattata come se fosse appena entrata in RAM.
  - Il SO passa ad esaminare la successiva pagina della coda FIFO

10.4.5.2 Algoritmo della Seconda Chance

- Nel caso peggiore, l'algoritmo trova che tutte le pagine della coda hanno il reference bit a 1
- Allora, una volta percorso l'elenco di tutte le pagine in RAM, il SO ritorna alla prima pagina esaminata (che era anche quella entrata in RAM da più tempo) e la sceglie come vittima
- In questo caso l'algoritmo della seconda chance si riduce all'algoritmo FIFO.
- Notiamo che se una pagina viene riferita abbastanza spesso, il suo reference bit sarà quasi sempre a 1, e difficilmente verrà scelta come pagina vittima

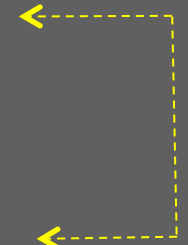
10.4.5.2 Algoritmo della Seconda Chance

- Algoritmo della seconda chance implementato con una coda circolare (fig. 10.17):
- Se “next victim” non viene riferita prima di una seconda chiamata dell’algoritmo, il suo reference bit resta a 0, quindi non è stata riferita di recente, e diviene quindi una buona candidata alla sostituzione



10.4.5.3 Algoritmo della Seconda Chance migliorato

- Se l'HW fornisce sia il reference che il dirty bit, le pagine possono essere raggruppate in 4 classi:
  - **(0, 0) né usata di recente né modificata**: ottima da rimpiazzare
  - **(1, 0) usata di recente ma non modificata**: meno buona da rimpiazzare
  - **(0, 1) non usata di recente, ma modificata:** ancora meno buona, dobbiamo salvarla in MS
  - **(1, 1) usata di recente e modificata**: la peggiore candidata al rimpiazzamento.



10.4.5.3 Algoritmo della Seconda Chance migliorato

- La tecnica usata è poi la stessa della seconda chance, ma si sceglie come pagina vittima la prima pagina che si trova nella classe migliore non vuota

- Questo algoritmo è usato in molti sistemi Unix e in MacOs.

10.4.7 Tecniche aggiuntive

- Il SO gestisce un pool di frame liberi: al page fault la pagina vittima, se ha il dirty bit a 1, viene trasferita in uno dei frame del pool (anziché salvata in MS)
- In questo modo, il processo che ha generato page fault può partire prima, senza aspettare il salvataggio della pagina.
- Inoltre, se la pagina vittima viene riferita subito dopo, è ancora in RAM!
- A "tempo perso" il SO può salvare sull'area di swap qualche pagina del pool. In questo modo aumentano le probabilità che il prossimo page fault potrà essere gestito in modo più efficiente, cioè che ci sia un frame libero nel pool.

10.5 Allocazione dei Frame

- In un sistema multiprogrammato, come distribuiamo i frame disponibili fra i processi?
  - lo stesso numero per tutti?
  - in proporzione alle dimensioni di ogni processo?
  - in proporzione alla priorità dei processi?
  - dovremmo tenere alcuni frame liberi per eventuali nuovi processi che entrano nel sistema?

10.5 Allocazione dei Frame

- **Allocazione uniforme**: lo stesso numero di frame a tutti i processi:
  - *n* frame, *p* processi, *n/p* frame a ciascun processo

- **allocazione proporzionale:** tiene conto del fatto che i processi hanno dimensioni diverse
  - se le dimensioni in pagine di tre processi sono:
    $P_1 = 4$, $P_2 = 6$, $P_3 = 12$

    e ci sono 11 frame disponibili, l'allocazione sarà:
    $P_1 = 2$, $P_2 = 3$, $P_3 = 6$.

10.5 Allocazione dei Frame

- **allocazione proporzionale in base alla priorità:** tiene conto del fatto che i processi hanno priorità diverse
- Il numero di frame assegnati a ciascun processo dipende dalla priorità relativa dei processi (ed eventualmente anche dalle loro dimensioni)
- Quale che sia lo schema adottato, è chiaro che il numero di frame allocati per processo cambia anche col variare del grado di multiprogrammazione

10.5.3 Allocazione globale e locale

- In quale gruppo di pagine possiamo/dobbiamo scegliere la vittima da rimuovere dalla MP?

- **Allocazione globale**: scegliamo la vittima fra tutte le pagine in memoria principale (di solito, escluse quelle del SO)

  – notate: probabilmente verrà portata via una pagina ad un processo diverso da quello che ha generato il page fault

- **Allocazione locale**: scegliamo la vittima fra le pagine del processo che ha generato page fault

  – il numero di frame per processo rimane costante

10.5.3 Allocazione globale e locale

- **Problemi dell'allocazione globale**:
  - il turnaround di un processo è fortemente influenzato dal comportamento degli altri processi con cui convive, e potrebbe variare tantissimo da un'esecuzione all'altra
- **Problema con l'allocazione locale**:
  - se si danno troppe (relativamente) pagine ad un processo, si può peggiorare il throughput del sistema perché gli altri processi genereranno più page fault

10.5.3 Allocazione globale e locale

- Si è visto sperimentalmente che l'allocazione globale fornisce in genere un throughput maggiore e riesce a gestire la multiprogrammazione in maniera più flessibile.
- l'allocazione globale è di solito preferita per sistemi time sharing, in cui molti utenti possono usare contemporaneamente il sistema
- i sistemi Windows usano l'allocazione locale, mentre Linux e Solaris usano l'allocazione globale delle pagine

10.6 Thrashing (attività di paginazione degenere...)

- Consideriamo un sistema in cui, in un certo momento, ogni processo ha a disposizione pochi frame, ossia ogni processo ha in RAM un piccolo numero di pagine rispetto al numero totale di pagine di cui è composto.

- Supponiamo anche di adottare una allocazione globale dei frame (attenzione, però: il problema del thrashing si presenta anche nel caso di allocazione locale dei frame)

10.6 Thrashing

- Avendo poche pagine in RAM, ogni processo ha un'alta probabilità di generare un page fault.
- Quando un processo genera page fault, una pagina vittima viene tolta dalla MP, probabilmente ad un altro processo.
- Quest'altro processo ora ha in RAM ancora meno pagine di prima, e quindi si alza ancora la probabilità che possa generare un page fault nell'immediato futuro.
- Si innesca così un circolo vizioso per cui tutti i processi generano continuamente page fault, si rubano i frame l'un l'altro e inducono la generazione di ulteriori page fault.
- Questo fenomeno si chiama **THRASHING**

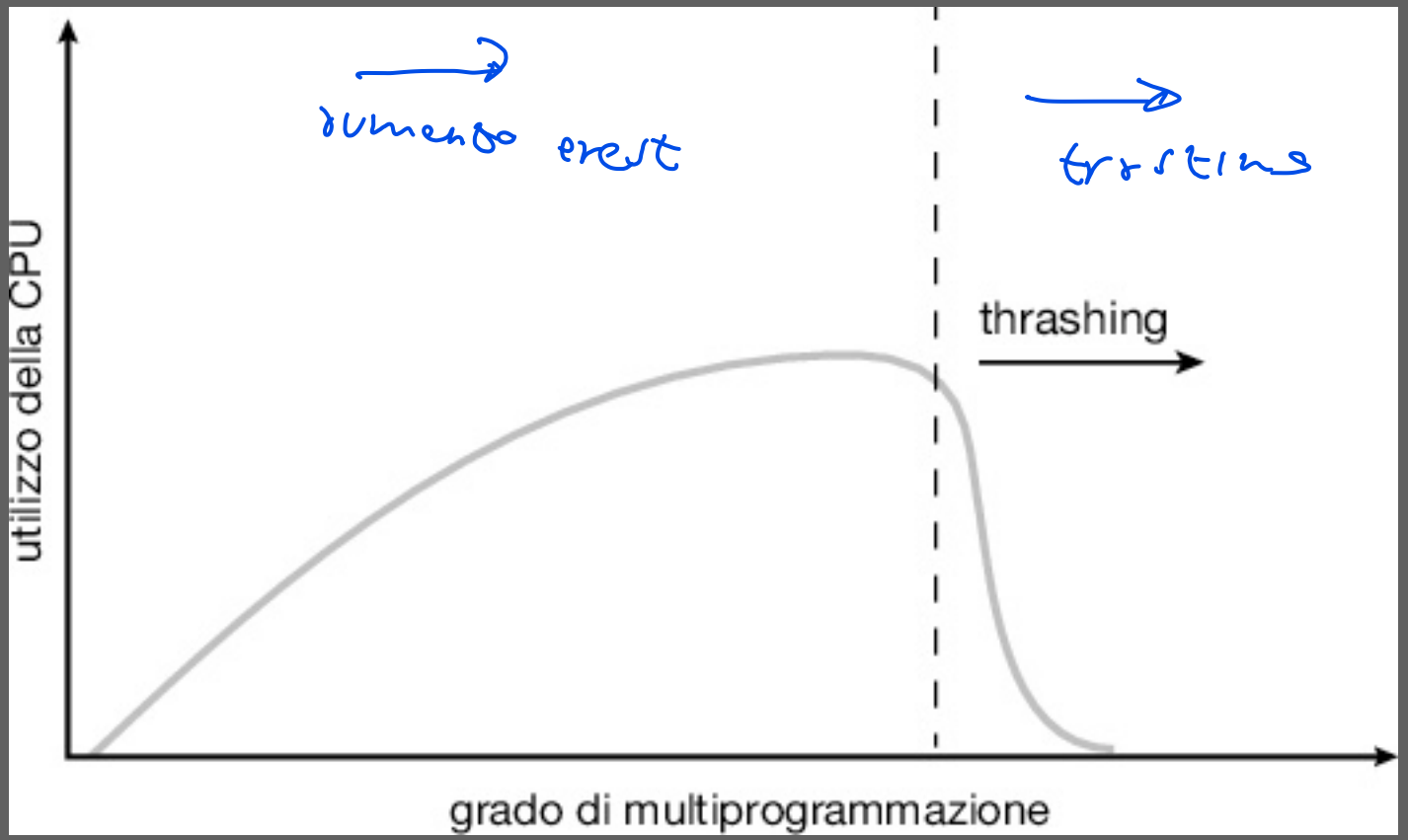
10.6 Thrashing

- Intuitivamente, il thrashing si verifica quando si tenta di aumentare troppo il grado di multiprogrammazione, in modo da sfruttare al massimo il tempo di CPU e incrementare il throughput del sistema.

- Oltre una certa soglia però, i processi passano più tempo ad aspettare che venga gestito il page fault che hanno generato piuttosto che a portare avanti il loro lavoro, e il livello di utilizzazione della CPU, e quindi il throughput, crollano verticalmente!

10.6 Thrashing

- Relazione tra grado di multiprogrammazione e utilizzo della CPU: il fenomeno del thrashing (fig. 10.20):



10.6.1 Cause del thrashing

- Se il livello di utilizzo della CPU di un sistema è troppo basso, lo si può alzare aumentando in grado di permettendo a più utenti di connettersi, e/o di lanciare un maggior numero di processi
- in questo modo però, i nuovi processi incominciano a sottrarre pagine ai processi già presenti, per “farsi un pò di spazio”...
- Fino ad un certo punto l’aumento di processi è ben tollerato dal sistema, poiché ciascun processo ha comunque una quantità sufficiente di frame a disposizione da poter girare senza generare troppi page fault.

10.6.1 Cause del thrashing

- ma se si esagera, ci si può avvicinare alla soglia del thrashing: molti processi incominciano a generare dei page fault e come conseguenza vengono tolti dalla RQ e messi in una coda di wait in attesa della pagina mancante

- Risultato? La RQ si svuota, e il livello di utilizzo della CPU scende.

- Beh, ma se la CPU è sotto-utilizzata, si può lanciare qualche altro processo, o permettere a qualche altro utente di collegarsi... E la situazione non fa che peggiorare!

10.6.1 Cause del thrashing

- Nei moderni sistemi time-sharing, il ciclo perverso è spesso innescato dagli utenti, che lanciano altri programmi senza attendere la fine di quelli già in esecuzione, sperando così di aumentare la percentuale del tempo di CPU globale che riescono ad usare a loro vantaggio...

10.6.1 Come combattere il thrashing

- In definitiva quindi, il thrashing è una sorta di "ingolfamento" del sistema: Vogliamo sfruttarlo al meglio "iniettando" più e più processi nel sistema, fino ad arrivare ad un punto in cui i processi si ostacolano a vicenda.

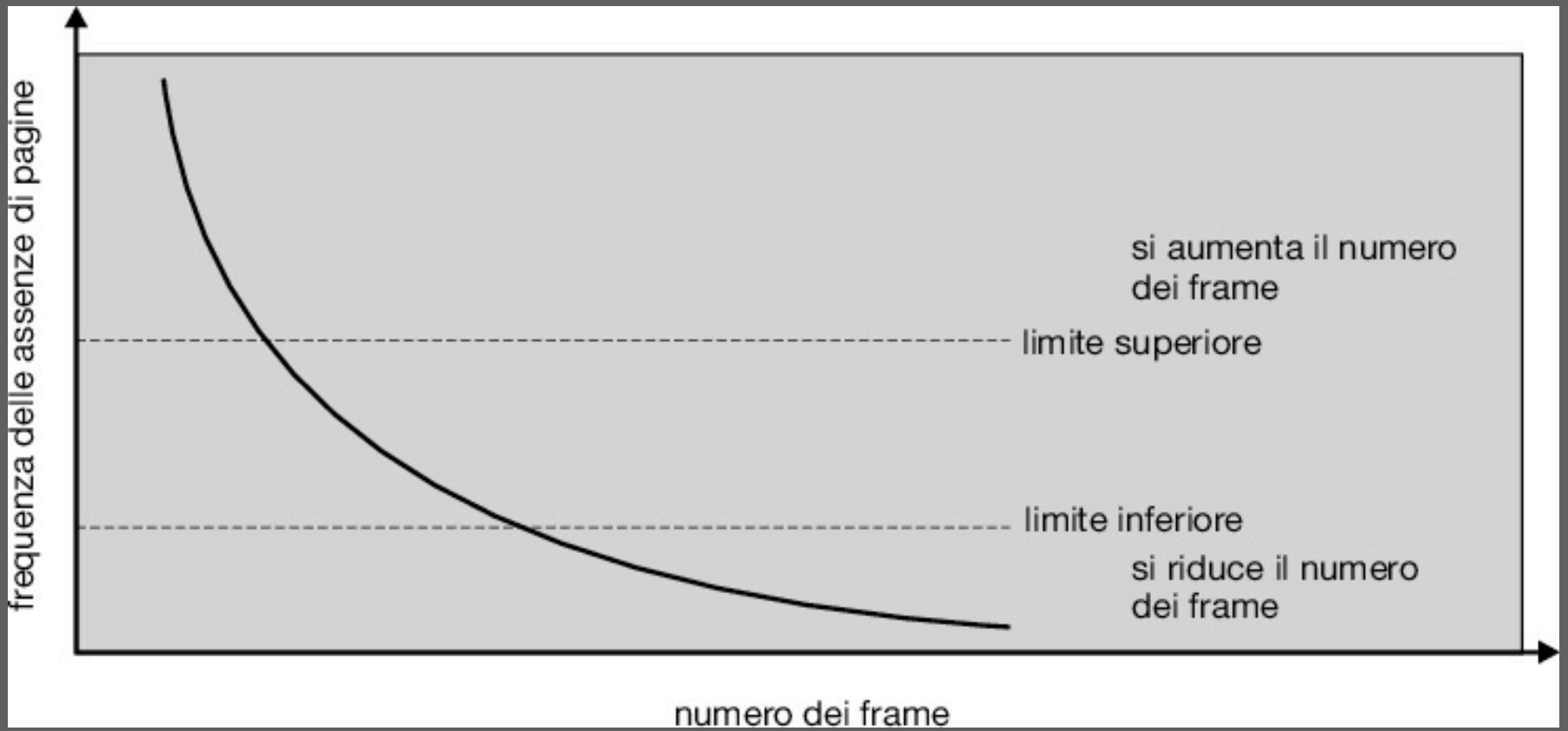
- La soluzione giusta sarebbe di **diminuire il grado di multiprogrammazione temporaneamente**, in modo che i processi non rimossi dalla MP abbiamo il tempo di terminare correttamente prima di far (ri)partire gli altri.

10.6.3 Gestire la frequenza dei page fault

- Stabiliamo (ad esempio in base ad osservazioni sperimentali) quando la frequenza di page-fault e' "accettabile" rispetto alle prestazioni che vogliamo ottenere dal sistema.
  - Se la frequenza osservata è troppo bassa, possiamo togliere ai processi qualche frame, e aumentare il grado di multiprogrammazione
  - Se la frequenza osservata è troppo alta, diminuiamo il grado di multiprogrammazione e ridistribuiamo i frame liberati fra i processi non rimossi

10.6.3 Gestire la frequenza dei page fault

- prevenzione del thrashing attraverso il monitoraggio della frequenza di page fault (fig. 10.23):



10.6 Come prevenire il thrashing

- Usare una politica di sostituzione locale mitiga il problema, perché i processi non si possono rubare le pagine l'un l'altro.
- Se anche un singolo processo va in thrashing, non danneggia gli altri (in realtà un pochino si, perché farà uso pesante dell'I/O su disco)
- se però diamo troppo pochi frame a ogni processo per aumentare il grado di multiprogrammazione, può accadere che tutti vadano in thrashing lo stesso
- **Ma alla fine, la miglior prevenzione del trashing è dotare il sistema di sufficiente memoria principale!**

10.9.2 Altre considerazioni: dimensione delle pagine

- Sono sempre potenze di 2; ma quanto dovrebbero essere grandi? Pagine piccole implicano:
  - PT grandi
  - meno frammentazione interna
  - peggiori prestazioni nell'uso dell'HD perché seek e latenza del disco sono costanti, (ci torniamo nel cap. 11)
  - pagine piccole producono in genere più page fault (pensate a pagine da un byte con pure demand paging...)
- Per pagine grandi valgono le considerazioni opposte

10.9.2 Altre considerazioni: dimensione delle pagine

- a causa della diminuzione del costo della RAM e dell'aumento degli spazi di indirizzamento fisico disponibili, la tendenza è a usare pagine sempre più grandi.
- Negli anni '80 si consideravano 4 Kbyte come la massima dimensione accettabile di una pagina, mentre ora è un valore normale, spesso superato.
- Inoltre, l'aumento costante della quantità di RAM disponibile nei sistemi moderni ha fatto si che nel tempo, il problema del thrashing, e più in generale le problematiche legate alla memoria virtuale, abbiano un impatto inferiore sulle prestazioni dei sistemi.

10.9.5 Struttura dei programmi

- Nei programmi, il modo di usare i dati influisce enormemente sul numero di page fault
- ad esempio gli array bidimensionali sono allocati per riga: se scriviamo del codice che accede gli elementi per colonne, aumentiamo il rischio di page fault (supponiamo che una pagina contenga esattamente una riga dell'array: che succede se viene allocato un solo frame per contenere parte dell'array?)
- le hash table forniscono pessime prestazioni con la memoria virtuale, perché anche dati concettualmente contigui vengono memorizzati in modo sparpagliato.

10.9.5 Struttura dei programmi

- char A[1024][1024];
- Ogni riga e' memorizzata in una pagina, e all' array è assegnato un solo frame (= 1024 byte), in RAM. L'array è memorizzato per righe: (A[0][0], A[0][1], A[0][2], A[0][3],… A[0][1023], A[1][0], …)

Programma 1: **for** (j = 0; j < 1024; j++)
for (i = 0; i < 1024; i++) A[i][j] = '0';

Programma 2: **for** (i = 0; i < 1024; i++)
for (j = 0; j < 1024; j++) A [i][j] = '0';

- Quale programma genera meno page faults? Quanti?
- Quale programma genera più page faults? Quanti?

10.10.2 Windows 10

- In Windows 10 viene implementata la **demand paging with clustering**: quando una pagina viene caricata in memoria, vengono caricate anche alcune pagine adiacenti, che si presume potrebbero essere usate a breve.
- Alla creazione di un processo, a questo vengono assegnati due numeri:
  - **L'insieme di lavoro minimo**: minimo numero di pagine che il SO garantisce di allocare in RAM per quel processo (di solito, 50)
  - **L'insieme di lavoro massimo**: massimo numero di pagine che il SO allocherà in RAM per quel processo (di solito, 345)

10.10.2 Windows 10

- Il SO mantiene anche una lista di frame liberi, con associato un numero minimo di frame presenti nella lista.
- Se un processo P genera un page fault, e P non ha raggiunto il suo insieme di lavoro massimo, allora la pagina mancante viene portata in RAM assegnandogli un frame libero.
- Se invece P ha già raggiunto il suo insieme di lavoro massimo, viene scelta una pagina vittima tra quelle di P (quindi, sostituzione locale delle pagine)

10.10.2 Windows 10

- Se si raggiunge il limite minimo di frame liberi in RAM, viene innescata una procedura per liberare spazio.
- Ciascun processo che ha in RAM un numero di pagine superiore al suo insieme di lavoro minimo si vede rimosse dalla RAM tutte le pagine in eccesso.
- Per decidere quali pagine di ciascun processo rimuovere, nei sistemi con processore Intel viene usato l'algoritmo della seconda chance.

10.10.3 Solaris

- Solaris usa una normale paginazione su richiesta, assegnando un frame libero in caso di page fault.
- Il parametro **lostfree**, associato all'elenco dei frame liberi, pari di solito ad 1/64 del numero di frame in cui è suddivisa la RAM, indica l'eventuale mancanza di frame liberi.
- Ogni 1/4 di secondo, il SO controlla se il numero di frame liberi in RAM è inferiore a lostfree. In tal caso, viene attivato il processo **pageout**.

10.10.3 Solaris

- Pageout funziona in due fasi, applicando una variante dell'algoritmo della seconda chance.
- In una prima fase scandisce tutte le pagine allocate in RAM azzerandone il bit di riferimento.
- Nella seconda fase riscandisce tutte le pagine, e quelle il cui bit di riferimento è ancora a 0 vengono tutte considerate riutilizzabili: quelle che hanno il dirty bit a 1 vengono salvate prima di essere effettivamente riutilizzate.
- Se un processo fa riferimento ad una pagina "riutilizzabile" in attesa di essere salvata, questa pagina viene semplicemente riassegnata a quel processo.

10.10.3 Solaris

- In base a vari parametri del SO, può cambiare il tempo che intercorre tra le due scansioni effettuate da pageout, ma in ogni caso tale valore è dell'ordine di qualche secondo.
- Se pageout non riesce a mantenere la quantità di frame liberi ad un livello accettabile (stabilito da alcuni parametri del sistema), è possibile che si stia verificando il fenomeno del trashing.
- Il SO allora può allora decidere di rimuovere tutte le pagine di un processo, scegliendo tra i processi che sono rimasti inattivi per più tempo.