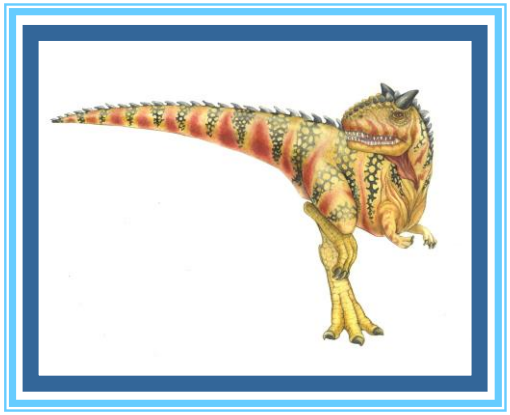
La memoria virtuale



Obiettivi

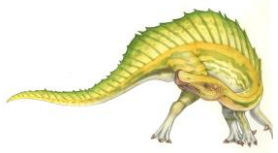
- Definire la memoria virtuale e descriverne i benefici
- Illustrare come le pagine vengono caricate in memoria utilizzando la paginazione su richiesta
- Definire gli algoritmi per la sostituzione delle pagine
- Descrivere il concetto di working set, correlato alla località dei programmi
- Descrivere come Linux e Windows gestiscono la memoria virtuale

Sommario

- Background
- Paginazione su richiesta
- Copy–on–Write
- Sostituzione delle pagine
- Allocazione dei frame
- Thrashing
- Allocazione di memoria del kernel
- Altre considerazioni
- Memoria virtuale in Linux e Windows

Background – 1

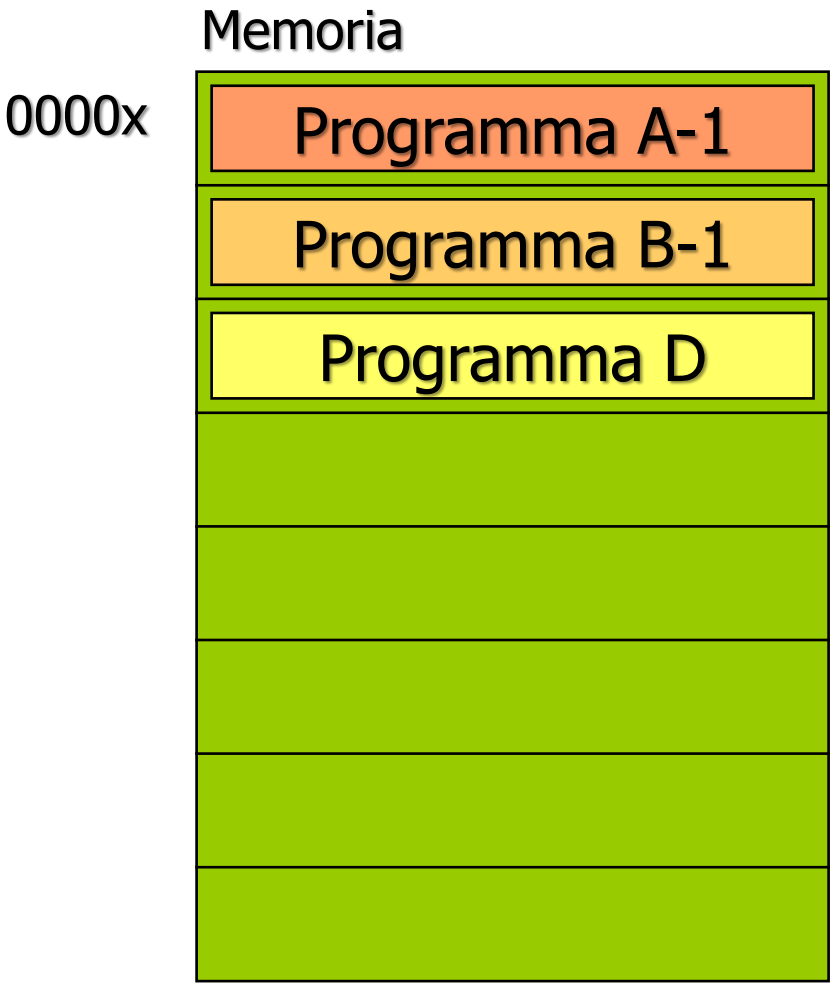
- Il codice, per poter essere eseguito, deve risiedere nella memoria centrale anche se, molto raramente, "tutto" il codice relativo ad un dato programma è necessario (e viene effettivamente "raggiunto") durante la sua esecuzione
  - Porzioni di codice correlate alla gestione di condizioni di errore insolite o percorsi nel flusso raramente seguiti
  - Array, liste e tabelle sovradimensionati rispetto all'utilizzo reale standard
- Anche nei casi in cui tutto il programma deve risiedere completamente in memoria è possibile che non serva tutto in una volta

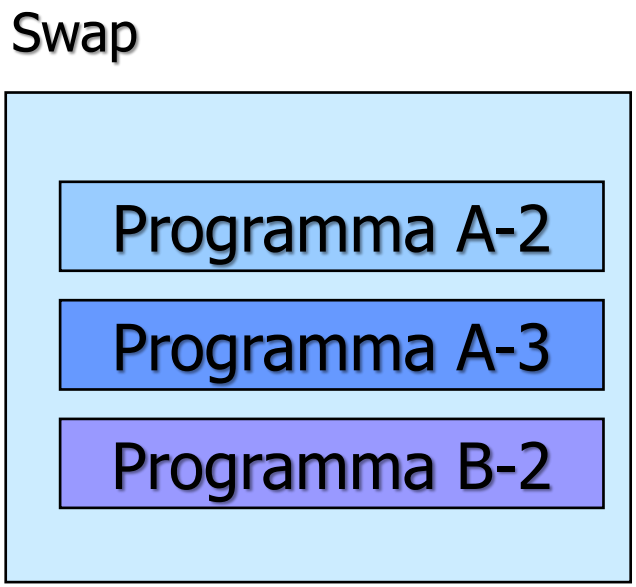


Background – 2

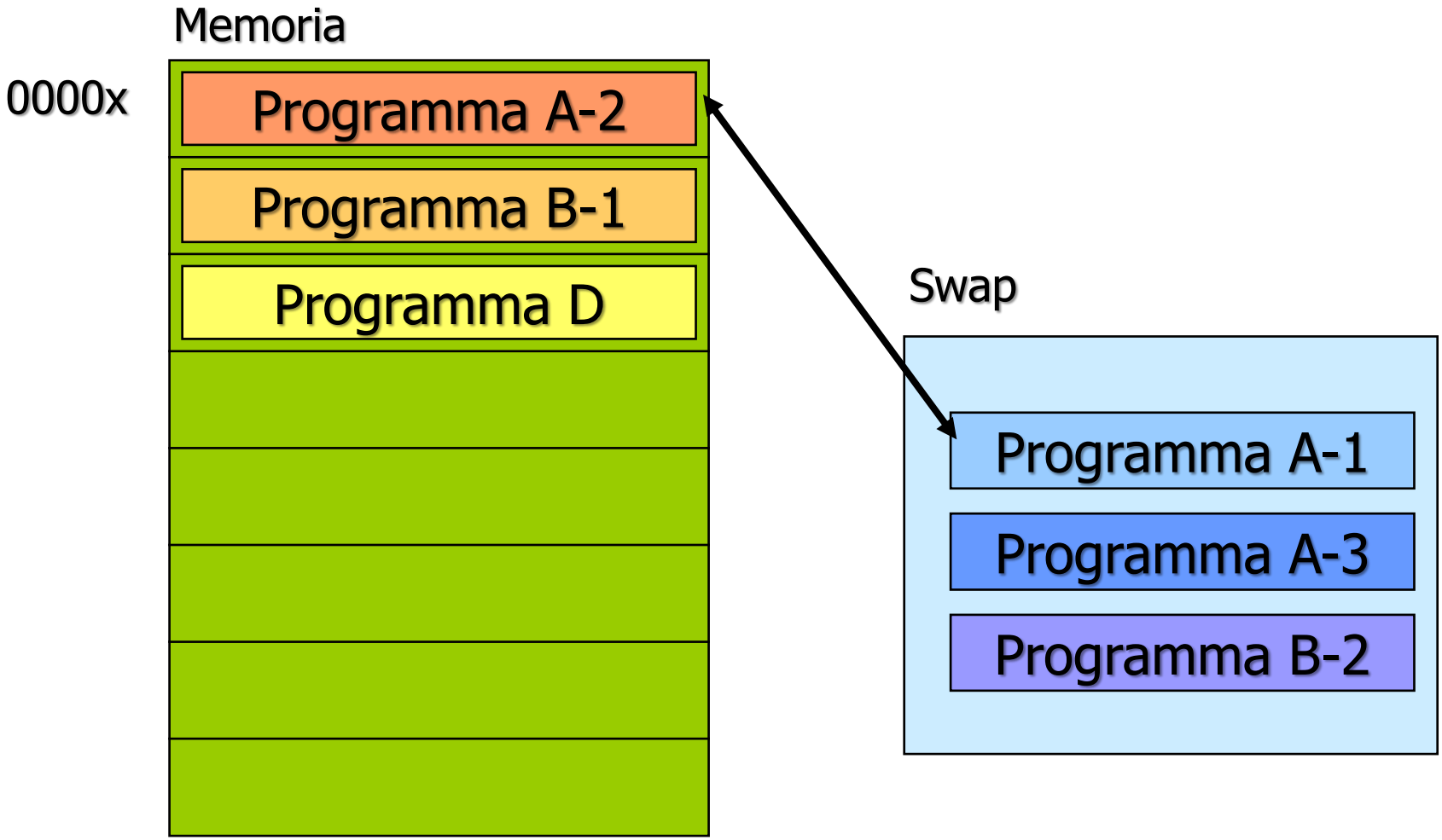
- Vantaggi della presenza di "porzioni" di programmi in memoria:
  - I programmi non sono più vincolati alla quantità di memoria fisica disponibile
  - Poiché ogni programma impiega meno memoria fisica, aumenta il grado di multiprogrammazione e la produttività del sistema, senza impatto sul tempo di risposta o di turnaround
  - Per caricare/scaricare ogni porzione di programma (piuttosto che l'intero processo) in/dalla memoria di massa è necessario un minor numero di operazioni di ingresso/uscita
    - ⇨ Aumentano le prestazioni del sistema

Background – 3





Background – 3



Background – 4

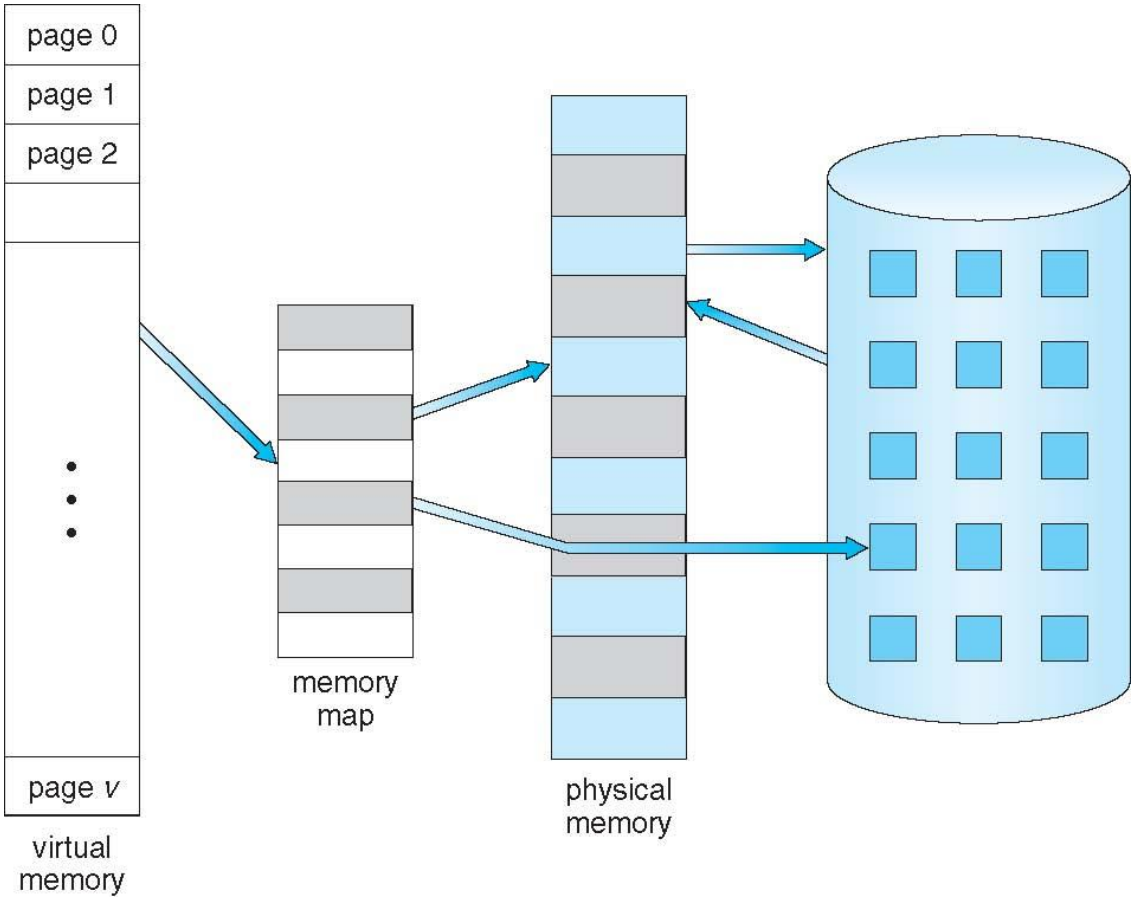
- Memoria virtuale — separazione della memoria logica dell'utente dalla memoria fisica
  - Solo parte del programma si trova in memoria per l'esecuzione
  - Lo spazio degli indirizzi logici può essere molto più grande dello spazio fisico (disponibile in RAM)
  - Porzioni di spazio fisico possono essere condivise da più processi
  - Creazione di nuovi processi più efficiente
- La memoria virtuale può essere implementata per mezzo di paginazione su richiesta

Background – 5

- L'espressione spazio degli indirizzi virtuali si riferisce alla collocazione dei processi in memoria dal punto di vista logico
  - Il processo inizia in corrispondenza dell'indirizzo logico 0 e si estende alla memoria contigua fino all'indirizzo Max
  - Viceversa, la memoria fisica allocata al processo è composta da pagine sparse (nella RAM e sulla backing store)
  - È la MMU che si occupa di effettuare il mapping da indirizzi logici a indirizzi fisici

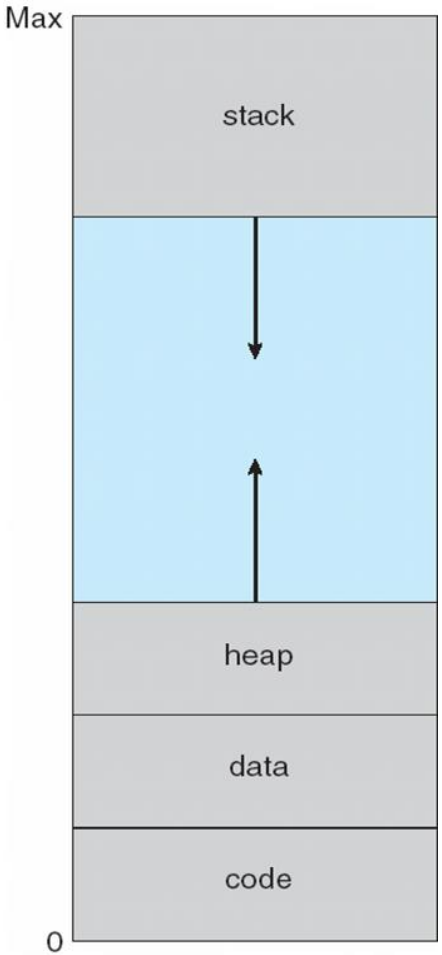
Background – 6

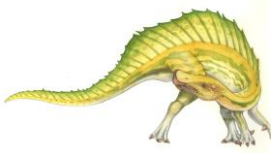
- Separazione memoria virtuale/memoria fisica
  - ⇨Memoria virtuale molto più ampia della memoria fisica



Spazio degli indirizzi virtuali – 1

- Lo stack serve per conservare i parametri/le variabili locali ai vari sottoprogrammi; è allocato a partire dall'indirizzo massimo e cresce (verso il basso) a causa delle ripetute chiamate di funzione
- Lo heap ospita la memoria allocata dinamicamente e cresce verso indirizzi alti
  - Il "buco" che separa lo heap dalla pila è parte dello spazio degli indirizzi virtuali del processo, ma si richiede l'allocazione di pagine fisiche solo se viene progressivamente "riempito"

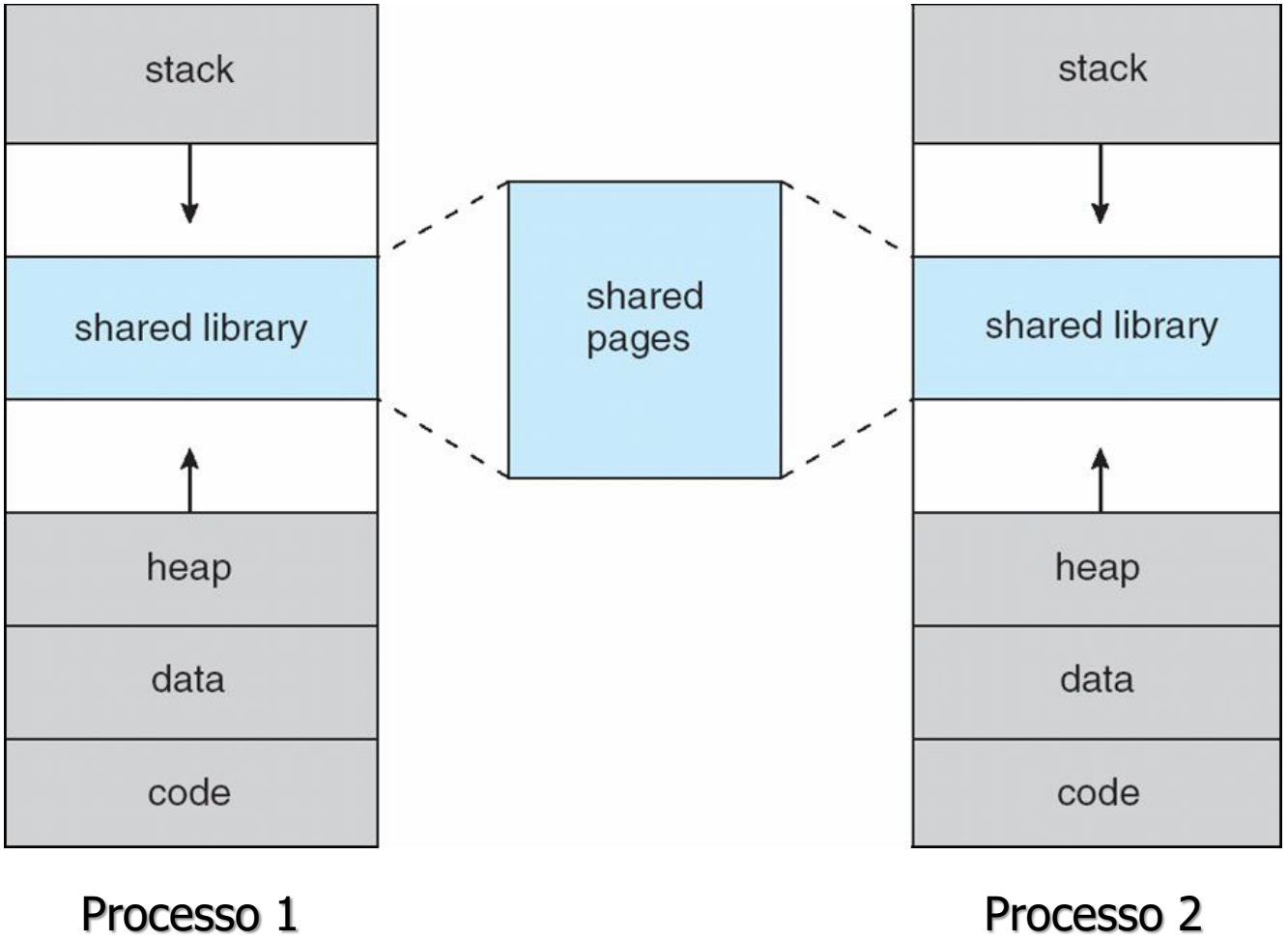




Spazio degli indirizzi virtuali – 2

- Uno spazio virtuale contenente buchi si definisce sparso
  - ⇨ Utile non solo per le possibili espansioni di heap e stack, ma anche per il collegamento dinamico di librerie durante l'esecuzione dei programmi
- La memoria virtuale facilita la condivisione di file e memoria, mediante condivisione delle pagine fisiche
  - Le librerie di sistema sono condivisibili mediante mappatura delle pagine logiche di più processi sugli stessi frame
    - Ogni processo vede le librerie come parte del proprio spazio di indirizzi virtuali, ma le pagine che le ospitano effettivamente nella memoria fisica – accessibili in sola lettura – sono in condivisione fra tutti i processi che utilizzano le librerie
  - Mediante la condivisione delle pagine in fase di creazione, i nuovi processi possono essere generati più rapidamente

Librerie condivise e memoria virtuale

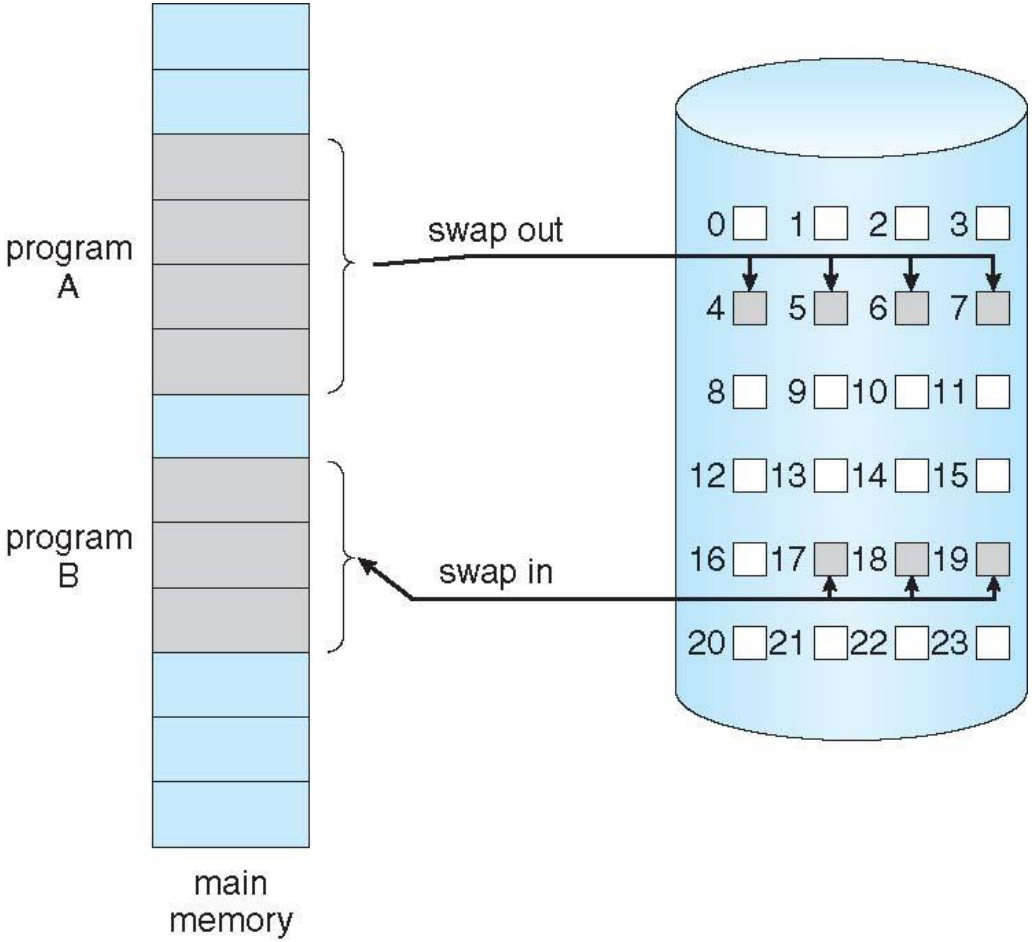


Paginazione su richiesta – 1

- I processi possono essere spostati in RAM per intero durante la fase di loading o si può invece decidere di caricare una particolare pagina in memoria solo quando è necessario (quando si fa riferimento a quella pagina)
  - Minor numero di operazioni di I/O, nessuna inutile
  - Minor utilizzo/spreco di memoria
  - Minor tempo di risposta
  - Più programmi utente presenti in memoria allo stesso tempo
- Simile ai sistemi di paginazione basati sullo swapping
- *Swapper pigro* – Si sposta una pagina in memoria solo quando è necessaria:
  - Richiesta di una pagina ⇨ si fa un riferimento alla pagina
    - Riferimento non valido ⇨ abort
    - Pagina non in memoria ⇨ trasferimento in memoria
- Il modulo del SO che si occupa della paginazione su richiesta è il *pager*

Paginazione su richiesta – 2

Swapper tradizionale in un sistema paginato con avvicendamento: trasferimento di una memoria paginata nello spazio contiguo di un disco



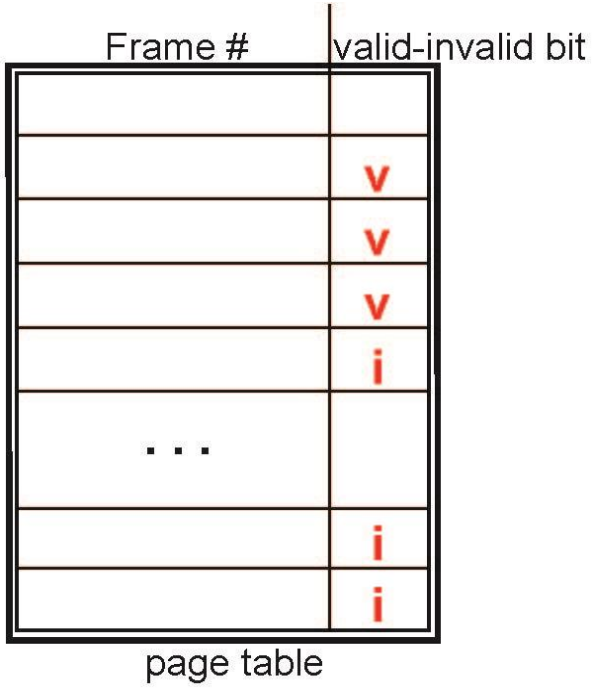
Invece, lo swapper pigro utilizza un criterio di avvicendamento (relativamente a singole pagine, non a processi) che non si attiva se non in caso di necessità

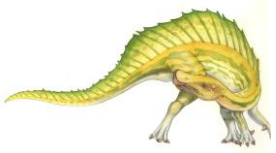
Paginazione su richiesta – 3

- Quando un processo sta per essere caricato in memoria, il pager ipotizza quali pagine saranno usate, prima che il processo venga nuovamente scaricato dalla memoria
- Anziché caricare in memoria tutto il processo, il pager trasferisce in memoria solo le pagine che ritiene necessarie
- Occorre una MMU con funzionalità ulteriori; quando si fa un riferimento ad una pagina:
  - Se la pagina richiesta è già presente in memoria, l'esecuzione procede come di consueto
  - Viceversa, la pagina richiesta deve essere localizzata sulla memoria di massa e trasferita in memoria centrale
    - Senza alterare il flusso del programma
    - Senza che il programmatore debba intervenire sul codice
  - ⇨ Occorre un meccanismo per distinguere le pagine presenti in memoria da quelle residenti sulla memoria secondaria
  - ⇨ Utilizzo opportuno del bit di validità presente nella tabella delle pagine

Bit di validità – 1

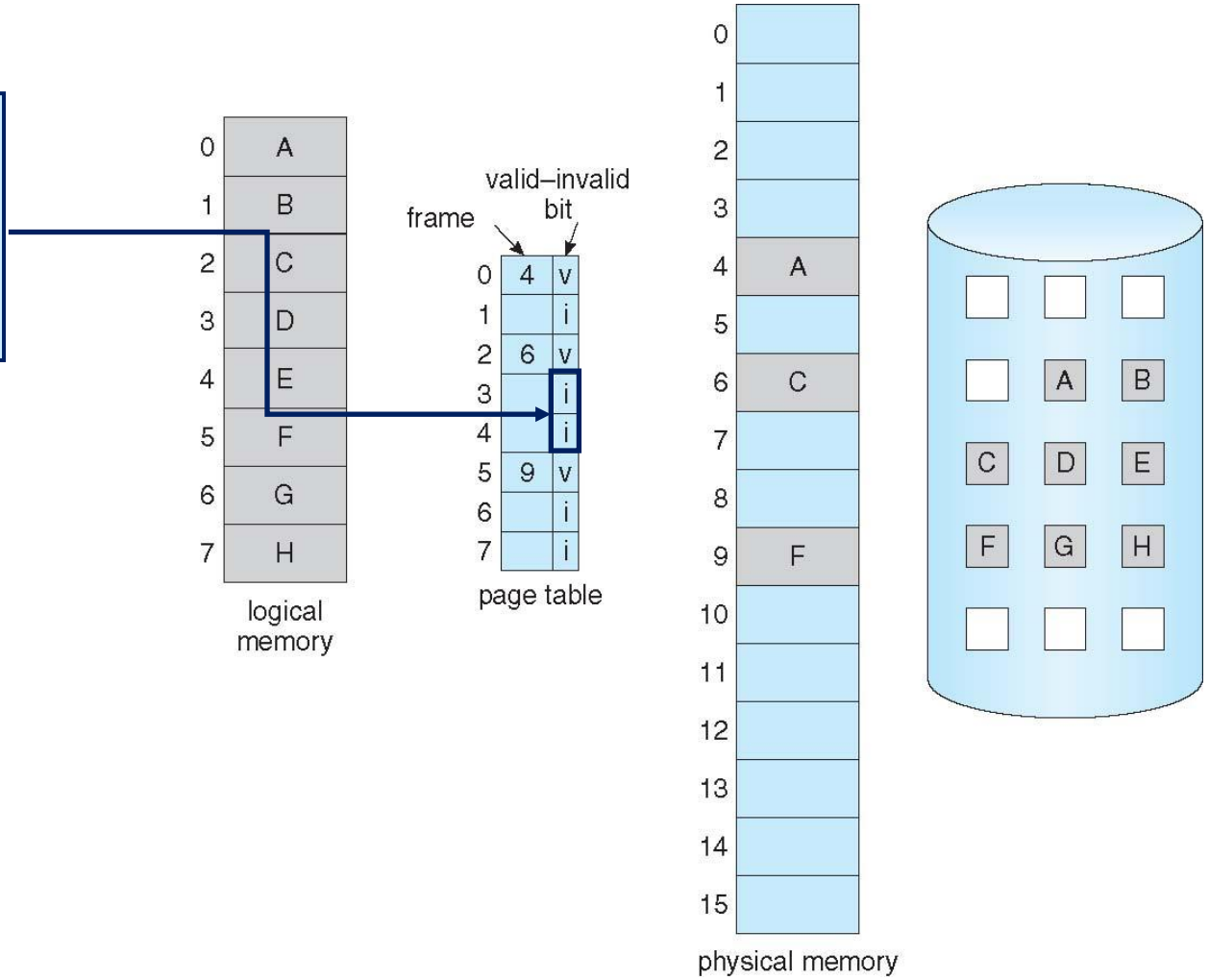
- Bit di validità:
  - v ⇨ pagina in memoria
  - i ⇨ pagina non valida o non residente in memoria
- Inizialmente il bit di validità viene posto ad "i" per tutte le pagine
- In fase di traduzione degli indirizzi, se il bit di validità vale "i" si verifica un *page fault*





Bit di validità – 2

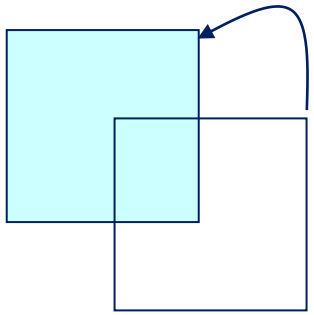
L'elemento della tabella delle pagine potrebbe contenere l'indirizzo per reperire la pagina sulla memoria secondaria



Istantanea di una tabella delle pagine con pagine non allocate in memoria principale

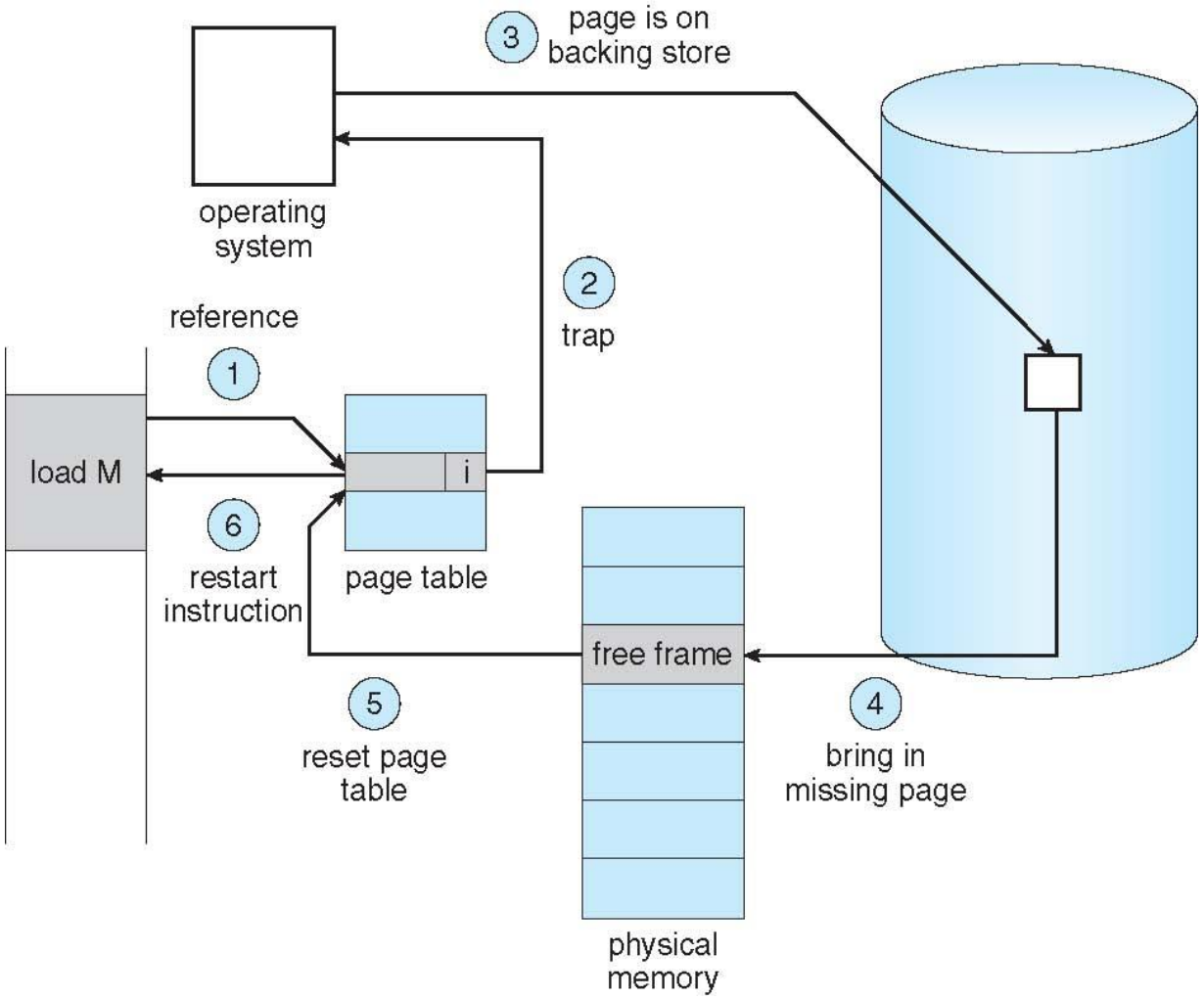
Page fault – 1

- Il primo riferimento ad una pagina causa una trap al sistema operativo ⇨ page fault
- Il SO consulta il PTLR (conservato nel PCB) per decidere se si tratta di…
  - riferimento non valido ⇨ abort
  - pagina non in memoria



- Nel secondo caso…
  - Seleziona un frame vuoto
  - Sposta la pagina nel frame (schedulando un'operazione di I/O)
  - Aggiorna le tabelle delle pagine (bit di validità = v) e dei frame (marcando come occupato il frame utilizzato)
  - Riavvia l'istruzione che era stata interrotta

Page fault – 2



Fasi di gestione di un'eccezione di pagina mancante

Paginazione pura

- È possibile avviare l'esecuzione di un processo senza pagine in memoria
- Quando il SO carica nel contatore di programma l'indirizzo della prima istruzione del processo, si verifica un page fault
- Il processo continua la propria esecuzione, provocando page fault, fino ad ottenere il caricamento in memoria di tutte le pagine "necessarie"

Ancora sulla paginazione su richiesta – 1

- In teoria, alcuni programmi possono accedere a diverse pagine di memoria all'atto dell'esecuzione di una singola istruzione (una pagina per l'istruzione e molte per i dati) provocando più page fault
  - Si consideri la fase di fetch e di decodifica di un'istruzione che somma due numeri residenti in memoria e, quindi, salva il risultato calcolato (`c=a+b`)
    - Potrebbero essere necessari fino a quattro page fault
  - Situazione improbabile, dato che i programmi tendono a mantenere un comportamento che preserva la località dei riferimenti
- Hardware di supporto alla paginazione su richiesta
  - Tabella delle pagine con bit di validità
  - Spazio di swap in memoria secondaria
  - Riesecuzione di un'istruzione che ha causato un page fault

Ancora sulla paginazione su richiesta – 2

- Un'istruzione interrotta a causa di un page fault deve essere riavviata
- Nella somma fra due numeri, se il page fault avviene all'atto del salvataggio in memoria del risultato in `c`, occorre…
  - Prelevare la pagina desiderata e caricarla in memoria
  - Correggere la tabella delle pagine e dei frame
  - Riavviare l'istruzione:
    - Nuova operazione di fetch, nuova decodifica e nuovo prelievo degli operandi
    - Calcolo dell'addizione
- La difficoltà maggiore si presenta quando un'istruzione può modificare interi blocchi di memoria, specialmente in caso di (parziale) sovrapposizione

Lista dei frame liberi – 1

- Quando si verifica un page fault, il SO deve portare la pagina desiderata dalla memoria secondaria nella memoria principale
- La maggior parte dei sistemi operativi mantiene un elenco di frame liberi

- I frame liberi devono essere allocati anche quando lo stack o lo heap di un processo si espandono

Lista dei frame liberi – 2

- Il sistema operativo assegna frame liberi utilizzando una tecnica nota come *zero–fill–on–demand* – il contenuto dei frame viene azzerato prima dell'allocazione
  - La non eliminazione dei contenuti precedenti avrebbe infatti un impatto devastante sulla sicurezza del sistema
- All'avvio di un sistema, tutta la memoria disponibile viene inserita nell'elenco dei frame liberi, dalla quale attinge, per esempio, la paginazione su richiesta
  - Utilizzo di tecniche per evitare lo svuotamento della lista dei frame liberi fondamentale per la corretta operatività del sistema di calcolo

Prestazioni della paginazione su richiesta – 1

- La paginazione su richiesta può avere un effetto rilevante sulle prestazioni del sistema di calcolo
- La gestione di un page fault comporta i seguenti passi:
  1) Trap al SO
  2) Salvataggio dei registri e dello stato del processo
  3) Verifica che l'interruzione sia dovuta o meno ad un page fault
  4) Controllo della correttezza del riferimento alla pagina e localizzazione della pagina sulla memoria di massa
  5) Lettura dalla memoria di massa e trasferimento
     a) Attesa nella coda al dispositivo di I/O (con eventuale attesa dovuta a posizionamento e latenza in caso di disco magnetico)
     b) Trasferimento della pagina in un frame libero
  6) Durante l'attesa: allocazione della CPU ad altro processo utente
  7) Ricezione dell'interrupt dal dispositivo di I/O (I/O completato)
  8) Context switch (se è stato eseguito il passo 6))
  9) Verifica della provenienza dell'interruzione
  10) Aggiornamento della tabella delle pagine (e dei frame) per segnalare la presenza in memoria della pagina richiesta
  11) Attesa nella ready queue
  12) Context switch di accesso alla CPU

Prestazioni della paginazione su richiesta – 2

- In ogni caso, il tempo di servizio dell'eccezione di page fault è costituito da tre componenti principali
  1) Servizio del segnale di eccezione di page fault
  2) Lettura della pagina dalla memoria di massa
  3) Riavvio del processo
- Se ben codificate, le operazioni di servizio constano di poche centinaia di istruzioni (1–100μsec), mentre il tempo di accesso alla memoria di massa costituisce il contributo più significativo
- **Page Fault Rate:** $0 \le p \le 1$
  - se p=0 non si hanno page fault
  - se p=1, ciascun riferimento è un fault
- Tempo medio di accesso (EAT):

$$EAT = (1-p) \times t[\text{accesso\_alla\_memoria}] + p \times (t[\text{servizio\_page\_fault}] + t[\text{swap\_out\_pagina}] + t[\text{swap\_in\_pagina}] + t[\text{riavvio\_del\_processo}])$$

Il 50% delle volte che una pagina viene rimpiazzata ha subito delle modifiche e deve essere sottoposta a swap out

Esempio di paginazione su richiesta

- Tempo di accesso alla memoria pari a 200νsec
- Tempo medio di servizio dei page fault pari a 8msec (=8000000νsec)

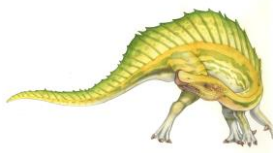
$$EAT = (1-p)\times 200 + p\times(8000000)$$

$$= 200 + 7999800p \quad \text{(in } \nu\text{sec)}$$

- Nota: Se un accesso su 1000 causa un page fault, EAT=8.2μsec ⇨ con la paginazione su richiesta l'accesso in memoria viene rallentato di un fattore 40
- Se si desidera un rallentamento inferiore al 10%:

$$p < 0.00000025$$

si può avere un page fault ogni 399990 accessi alla memoria



Ottimizzare la paginazione su richiesta – 1

- I/O dallo spazio di swap più rapido che da file system (anche relativamente allo stesso device)
  - Memoria allocata in blocchi più grandi, minor tempo di gestione rispetto all'attraversamento del file system
- Interi processi caricati da subito nell'area di swap
  - Attività di paginazione dall'area di swap (vecchie versioni di BSD UNIX)
- Viceversa, processi inizialmente caricati dal file system, ma con successiva attività di paginazione dalla sola area di swap (Solaris e BSD)
  - Pagine non modificate, quando soggette a sostituzione, non vengono copiate nell'area di swap
  - Sono invece soggette a swap
    - Le pagine non associate a file (come lo stack e lo heap), che costituiscono la memoria anonima
    - Pagine modificate in RAM e non ancora riscritte sul file system

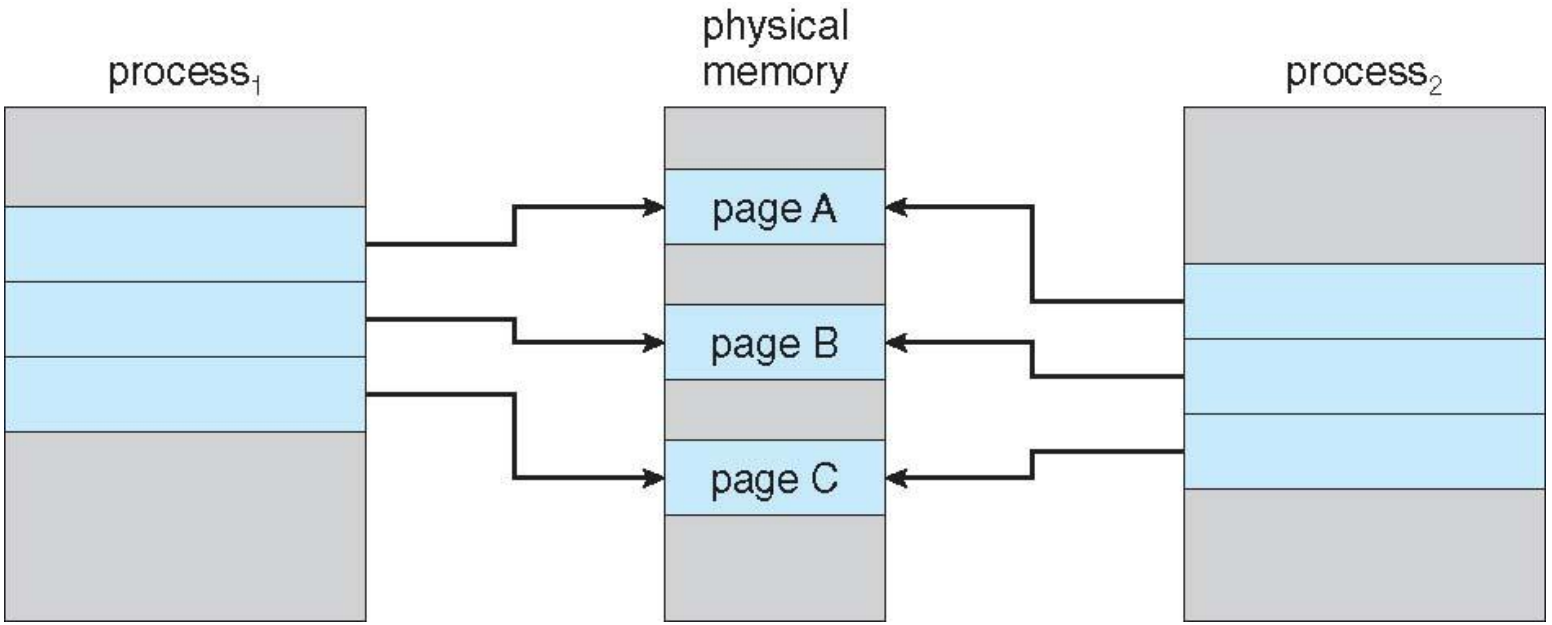
Ottimizzare la paginazione su richiesta – 2

- I sistemi mobili normalmente non supportano lo swapping ma, in caso di carenza di memoria, richiedono pagine al file system e recuperano pagine di sola lettura dalle applicazioni
- In iOS, non vengono mai riprese ad un'applicazione le pagine di memoria anonima, almeno che l'applicazione non sia terminata o abbia rilasciato la memoria volontariamente

Copy–on–Write – 1

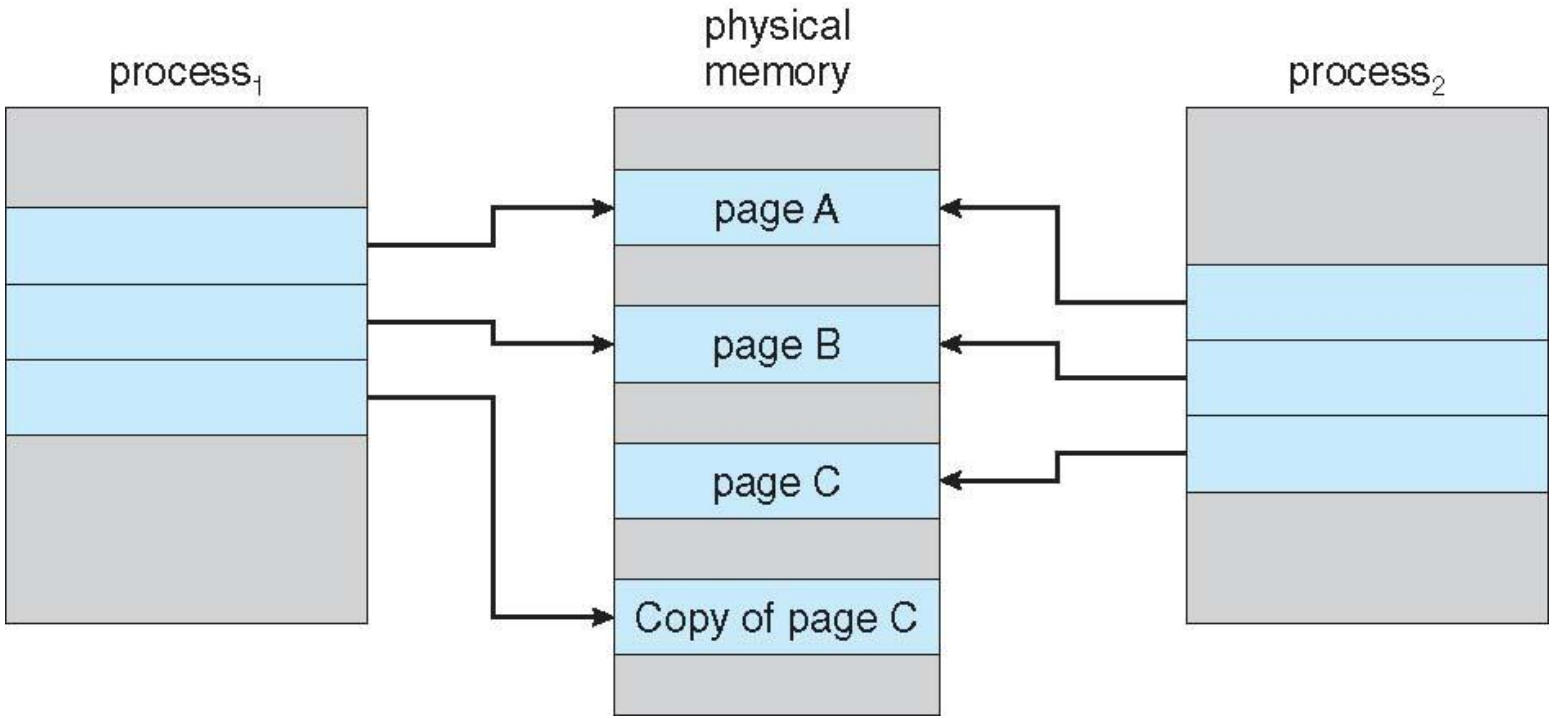
- Il processo di copiatura su scrittura, *Copy–on–Write*, *COW*, permette alla coppia di processi padre–figlio di condividere, inizialmente, le stesse pagine di memoria
- Se uno dei due processi modifica una pagina condivisa, e solo in quel caso, viene creata una copia della pagina
- La tecnica COW garantisce una modalità di creazione dei processi più efficiente, grazie alla copia delle sole pagine (condivise) modificate
- L'allocazione delle nuove pagine per effetto di un'operazione di copia avviene mediante una tecnica nota come azzeramento su richiesta
  - Prima dell'allocazione, il contenuto delle pagine viene azzerato

Copy–on–Write – 2



Prima della modifica della pagina C da parte del processo$_1$

Copy–on–Write – 3



Dopo la modifica della pagina C da parte del processo$_1$

Copy–on–Write – 4

- La **`vfork()`**, una variante della system call **`fork()`**, fa sì che il padre si sospenda in attesa del figlio e che il processo figlio usi lo spazio di indirizzi del genitore
- Poiché la **`vfork()`** non utilizza la copiatura su scrittura, se il processo figlio modifica qualche pagina dello spazio di indirizzi del genitore, le pagine modificate saranno visibili al processo genitore non appena riprenderà il controllo
  - Adatta al caso in cui il processo figlio esegua una **`exec()`** immediatamente dopo la sua creazione
  - Molto efficiente per la creazione di nuovi processi
  - Utilizzata per creare shell in UNIX

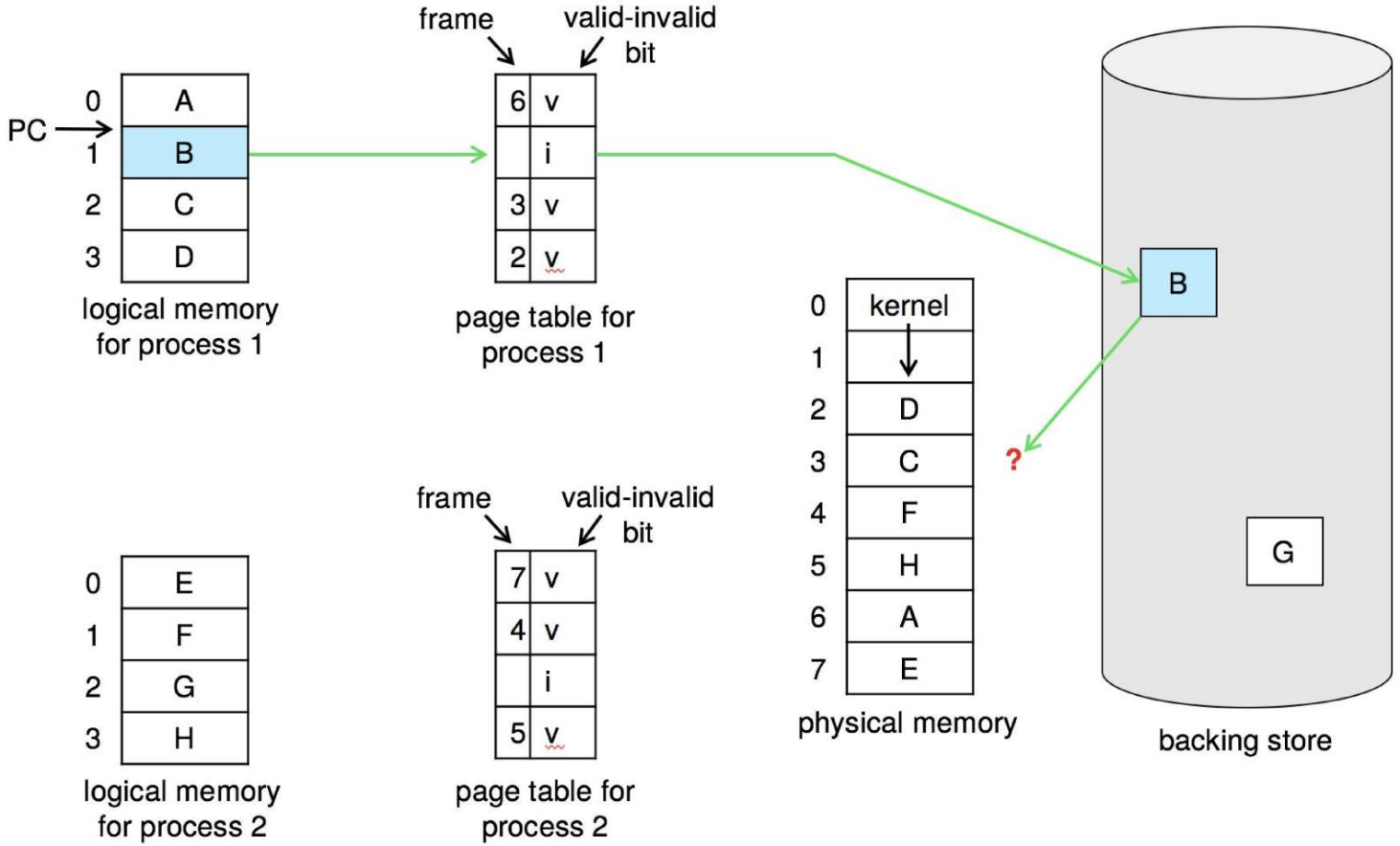
Cosa accade quando non ci sono frame liberi?

- Sovrallocazione: durante l'esecuzione di un processo utente si verifica un page fault; una nuova pagina deve essere caricata in memoria… ma non vi sono frame liberi
- Sostituzione di pagina – si trova una pagina in memoria che non risulti attualmente utilizzata e si sposta sulla memoria ausiliaria
  - Scelta di un algoritmo di selezione
  - Prestazioni: è richiesto un metodo che produca il minimo numero di page fault
- La stessa pagina può essere riportata in memoria più volte

Sostituzione delle pagine – 1

- La sovrallocazione della memoria si verifica quando è richiesta più memoria di quella effettivamente disponibile (page fault con assenza di frame liberi)
- Si previene la sovrallocazione della memoria modificando le routine di servizio del page fault, includendo la sostituzione delle pagine
- Si impiega un bit di modifica, detto anche dirty bit, per ridurre il sovraccarico dei trasferimenti di pagine: solo le pagine modificate vengono riscritte sulla memoria di massa
- La sostituzione delle pagine completa la separazione fra memoria logica e memoria fisica — una grande memoria virtuale può essere fornita ad un sistema con poca memoria fisica

Sostituzione delle pagine – 2

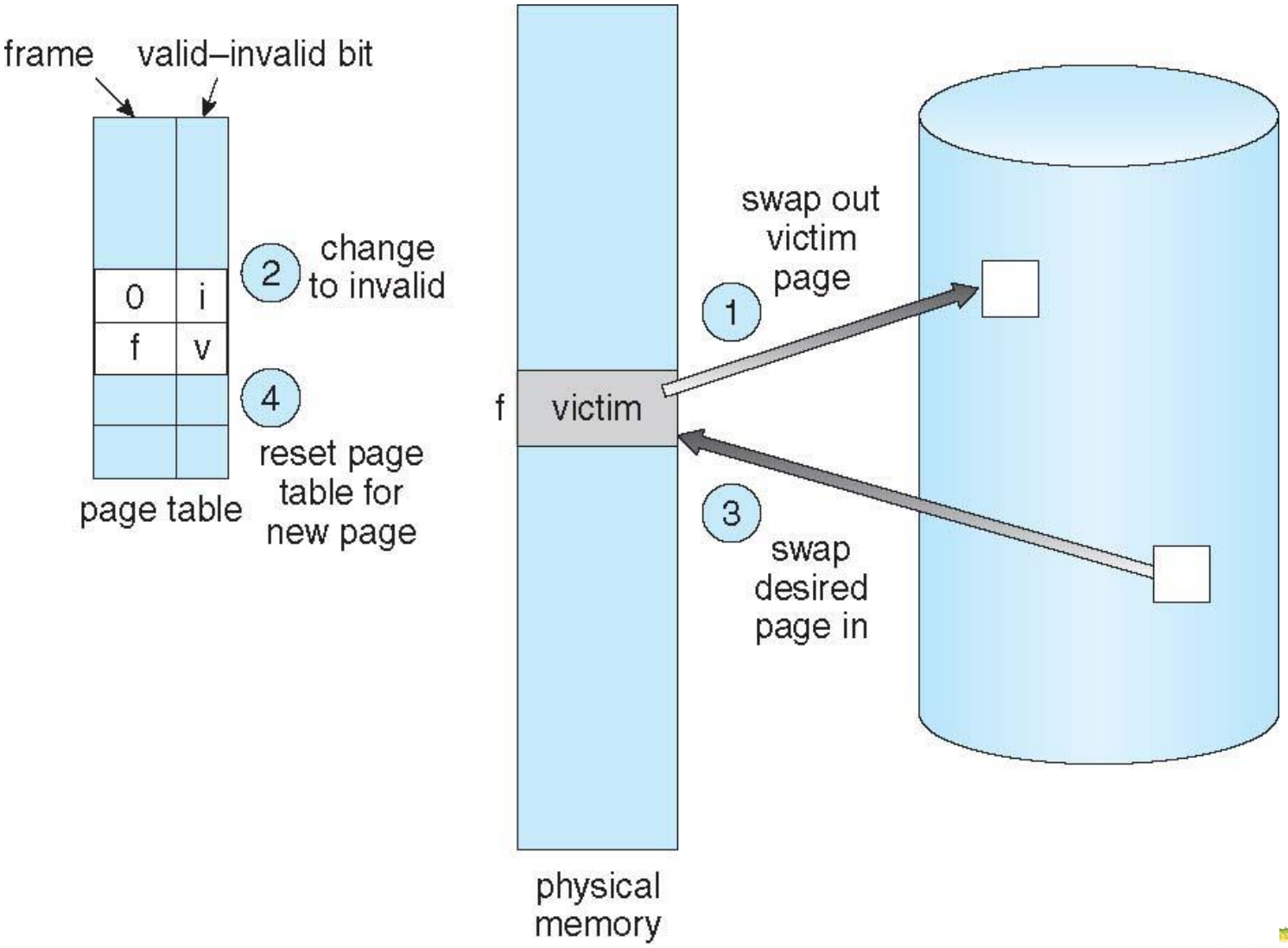


Sostituzione delle pagine – 3

1. Individuazione della pagina richiesta sulla memoria di massa
2. Individuazione di 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 sulla memoria secondaria; le tabelle delle pagine e dei frame vengono modificate conformemente
3. Lettura della pagina richiesta nel frame appena liberato; modifica delle tabelle delle pagine e dei frame
4. Riavvio del processo utente (riavviando l'istruzione che ha provocato la trap)

- Nota: Potenzialmente possono essere necessari due trasferimenti di pagina per la gestione del page fault (con relativo aumento dell'EAT)

Sostituzione delle pagine – 4

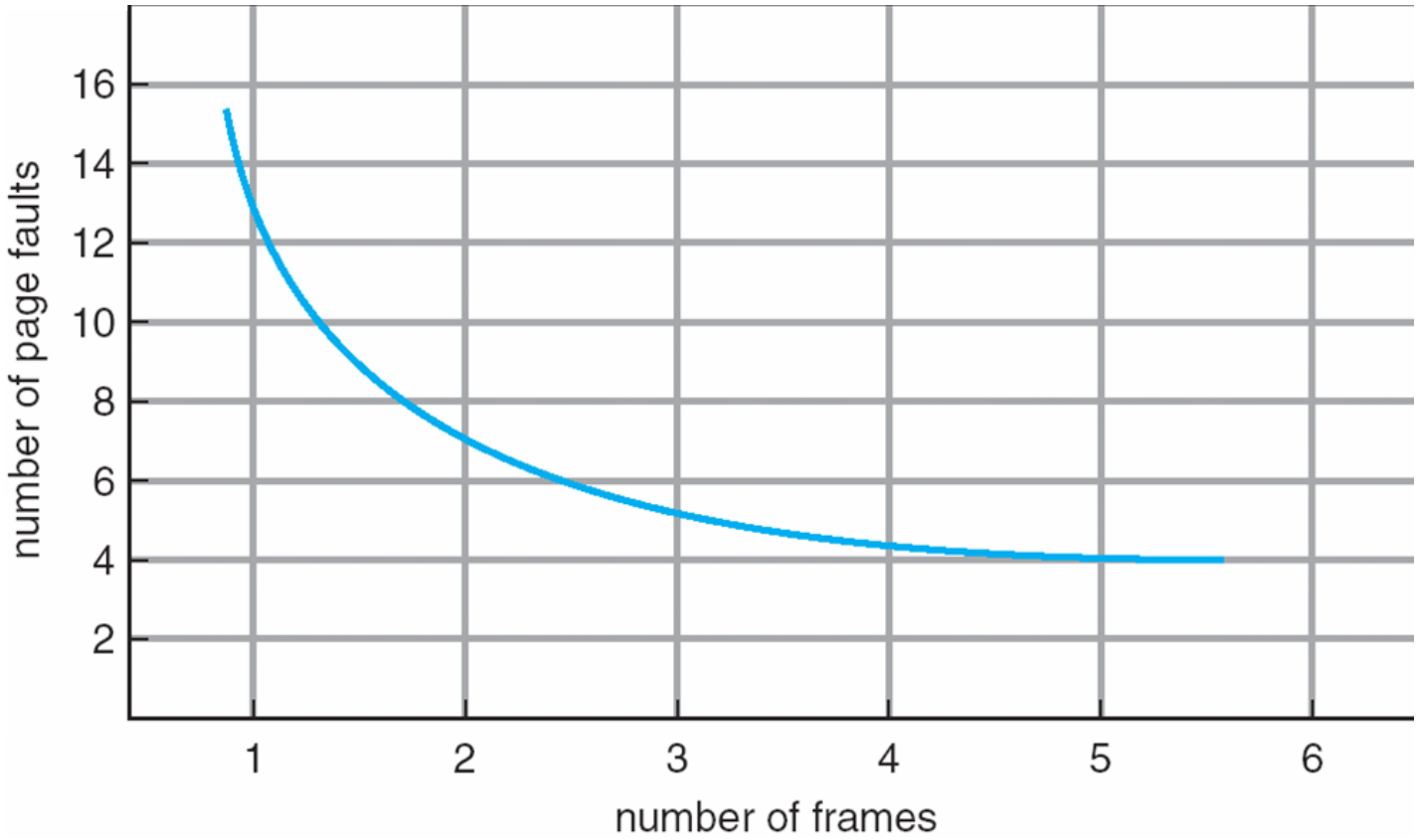


Algoritmi per la gestione della paginazione su richiesta

- Gli algoritmi per l'allocazione dei frame determinano quanti frame devono essere assegnati a ciascun processo
- Gli algoritmi per la sostituzione delle pagine mirano a minimizzare la frequenza di page fault, scegliendo quali frame sostituire quando necessario
- Si valutano gli algoritmi eseguendoli su una particolare stringa di riferimenti a memoria (detta *reference string*) e contando il numero di page fault su tale stringa (nei sistemi reali, si generano circa 1 milione di indirizzi al secondo)
  - La stringa è costituita dai soli numeri di pagina, non da indirizzi completi
  - Accessi multipli alla stessa pagina non provocano page fault
  - I risultati ottenuti dipendono significativamente dal numero di frame a disposizione
- In tutti gli esempi seguenti, la reference string è:

7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1

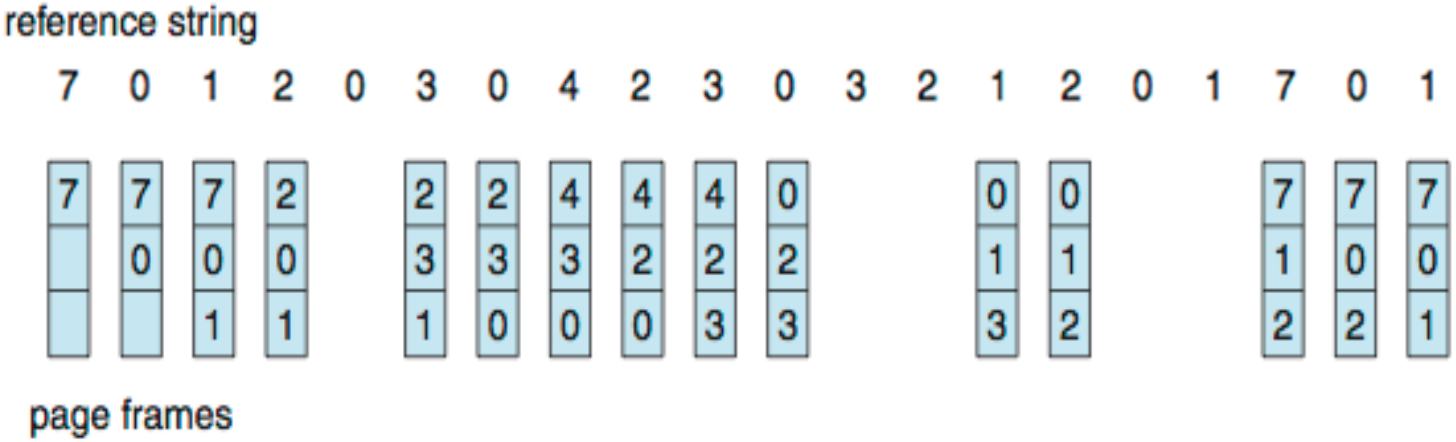
Numero di page fault in funzione del numero di frame



Algoritmo First–In–First–Out – 1

- Implementato mediante una coda FIFO relativa a tutte le pagine residenti in memoria
  - Si sostituisce la pagina che si trova in testa alla coda
- Stringa dei riferimenti:

  7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1
- 3 frame (3 pagine per ciascun processo possono trovarsi contemporaneamente in memoria)



⇨ 15 page fault

Algoritmo First–In–First–Out – 2

- Stringa dei riferimenti: 1 2 3 4 1 2 5 1 2 3 4 5
- 3 frame

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

9 page fault

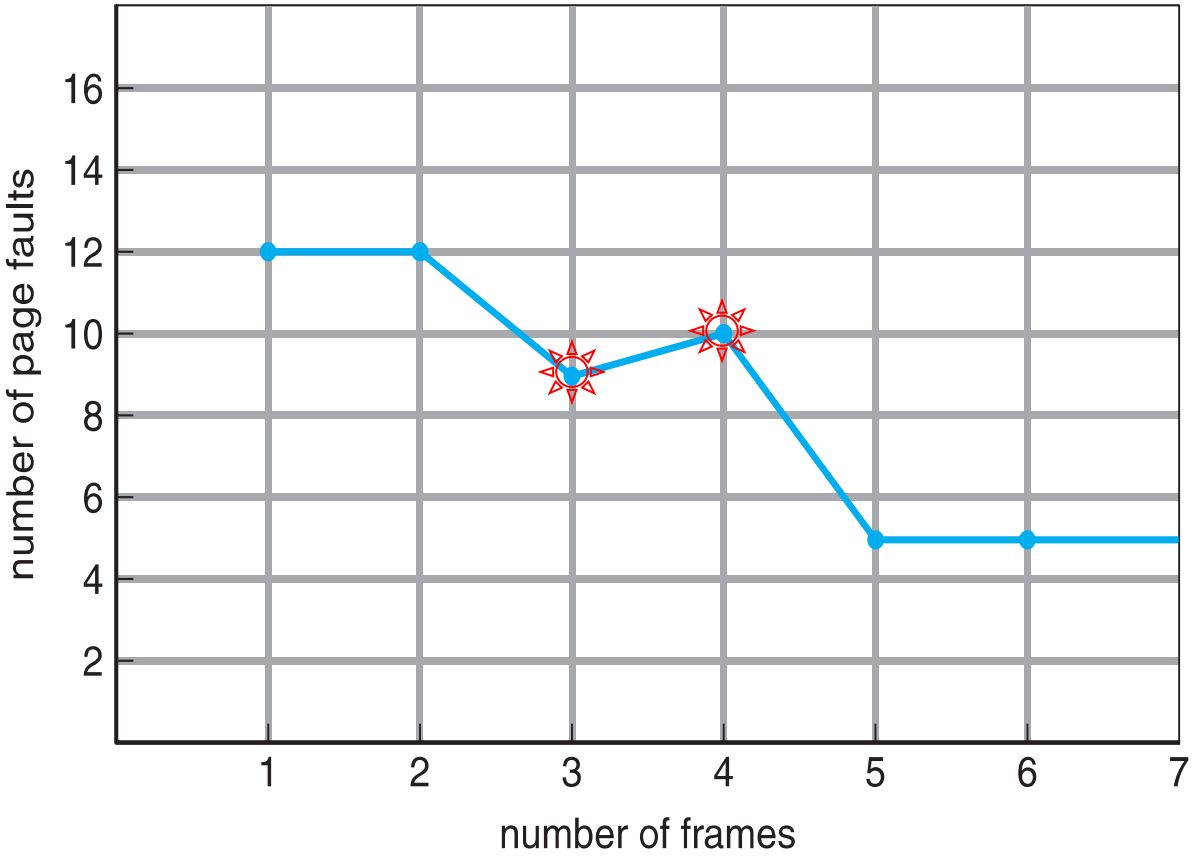
- 4 frame

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

10 page fault

- Aumentando il numero dei frame aumenta il numero di page fault ⇨ *Anomalia di Belady*

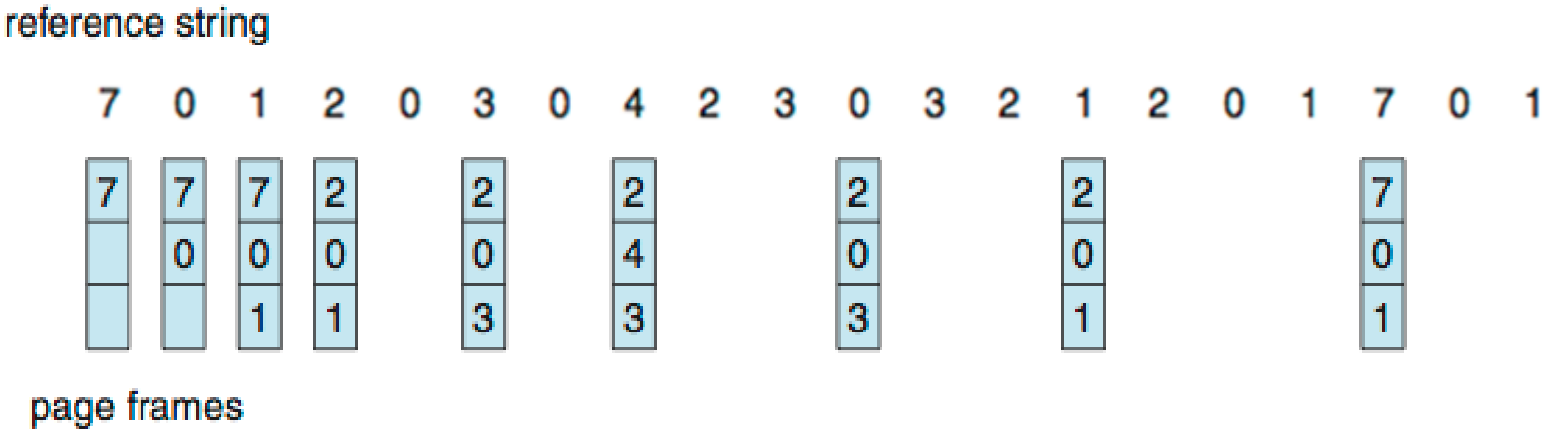
Algoritmo First–In–First–Out – 3



Curva dei page fault per sostituzione FIFO e anomalia di Belady

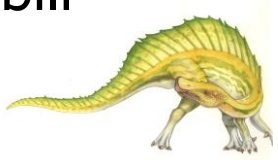
Algoritmo ottimo – 1

- Sostituire la pagina che non verrà usata per il periodo di tempo più lungo



⇨ 9 page fault: è il minimo ottenibile per la stringa considerata

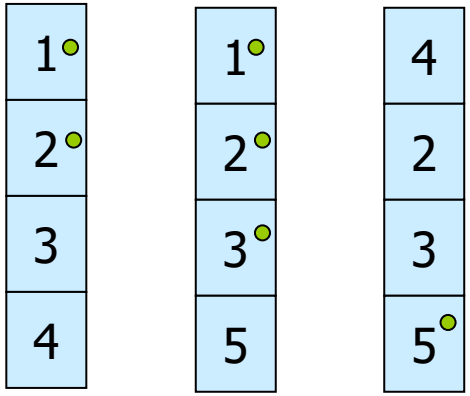
- Come si può conoscere l'identità della pagina se non conosciamo il futuro?
- Di solo interesse teorico: viene impiegato per misurare le prestazioni (comparative) di algoritmi implementabili



Algoritmo ottimo – 2

- Esempio: 4 frame, con stringa dei riferimenti

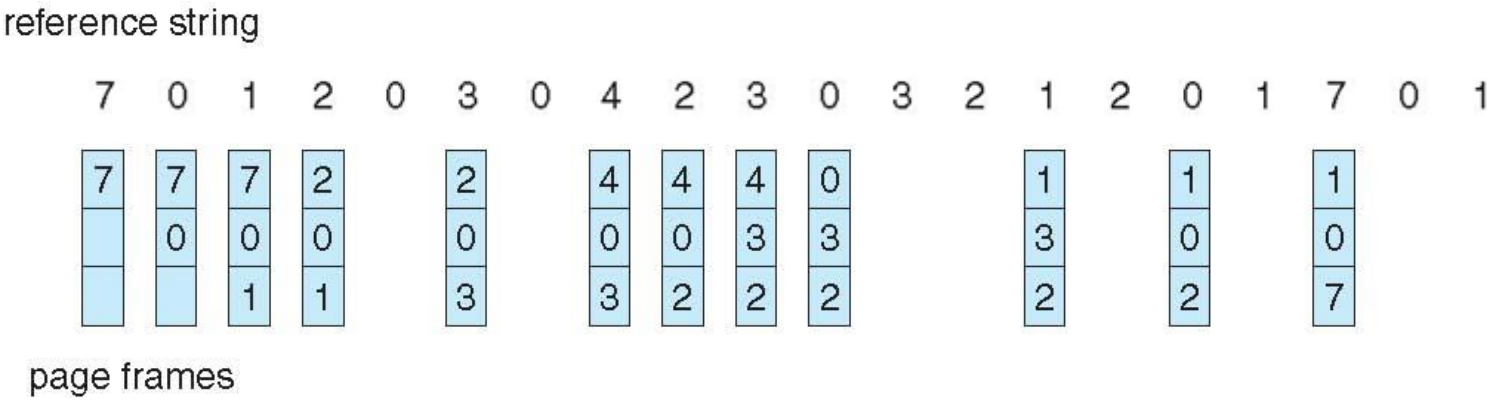
  1 2 3 4 1 2 5 1 2 3 4 5



6 page fault

Algoritmo Least Recently Used (LRU) – 1

- Si usa la conoscenza passata (acquisita) piuttosto che basarsi sul futuro
- Si rimpiazza la pagina che non è stata utilizzata per più tempo
  - ⇨ Occorre associare a ciascuna pagina il tempo di ultimo accesso



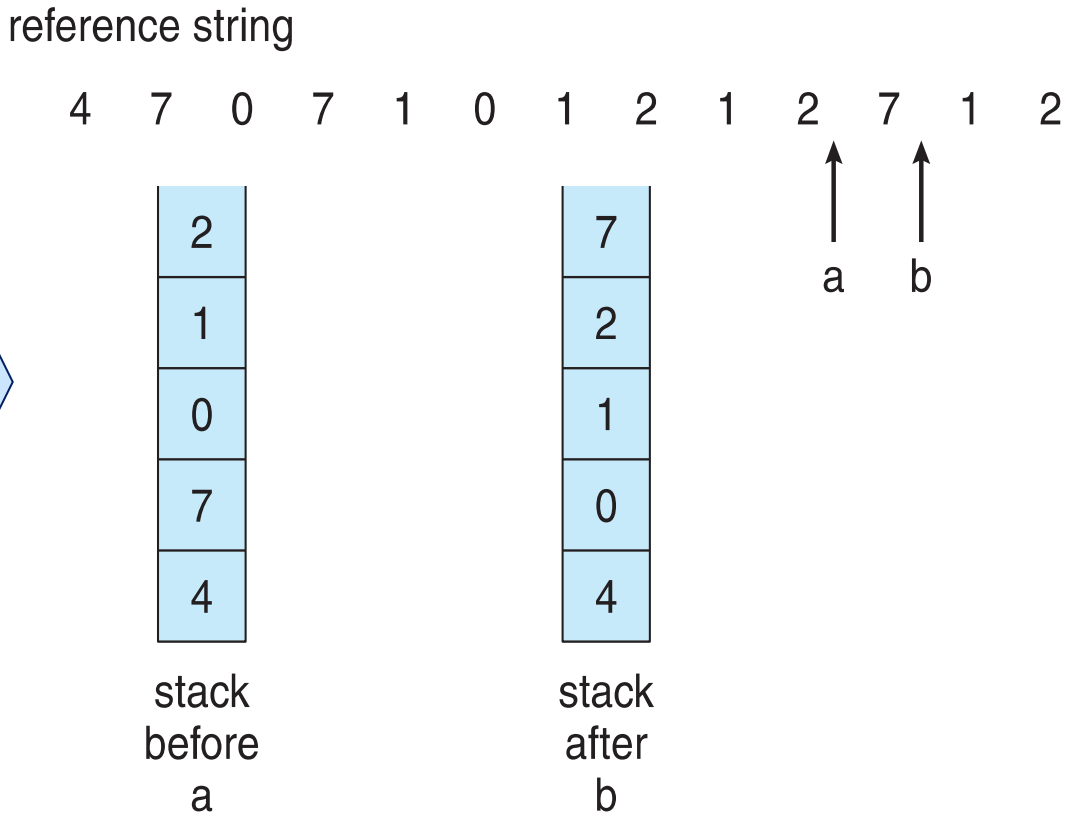
  - ⇨ 12 page fault: migliore di FIFO ma peggiore di OPT
- LRU è un buon algoritmo, comunemente utilizzato nei sistemi reali, ma…
- Come implementarlo?

Algoritmo LRU – 2

- Implementazione con contatore
  - Ciascuna pagina ha un contatore; ogni volta che si fa riferimento alla pagina si copia l'orologio nel contatore
  - Quando si deve rimuovere una pagina, si analizzano i contatori per scegliere quale pagina cambiare
    - ⇨ Necessaria la ricerca lineare sulla tabella delle pagine (e l'aggiornamento del contatore per ogni accesso in memoria – possibile overflow)
- Implementazione con stack
  - Si mantiene uno stack di numeri di pagina in forma di lista doppiamente concatenata
  - Pagina referenziata:
    - ⇨ Si sposta in cima allo stack
    - ⇨ È necessario aggiornare al più 6 puntatori
  - Non è necessario fare ricerche per la scelta della pagina, ma mantenere la pila aggiornata ha un costo

Algoritmo LRU – 3

Utilizzo dello stack per registrare i riferimenti alle pagine più recenti



Nel caso specifico, si modificano: il puntatore alla testa dello stack, il puntatore indietro di 2, il puntatore in avanti di 0, il puntatore indietro di 4, entrambi i puntatori di 7

Algoritmo LRU – 4

- **Esempio:** 4 frame, con stringa dei riferimenti

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

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

8 page fault

Algoritmi a pila

- OPT e LRU sono algoritmi "a pila" che non soffrono dell'anomalia di Belady
- Per un algoritmo a pila è possibile mostrare che l'insieme delle pagine in memoria per n frame è un sottoinsieme dell'insieme della pagine contenute in memoria per $n+1$ frame
  - Esempio: per la sostituzione LRU, l'insieme di pagine in memoria, in ogni istante, è costituito dalle n pagine cui si è fatto riferimento più recentemente, che restano tali (con l'aggiunta di una) se si utilizzano $n+1$ frame

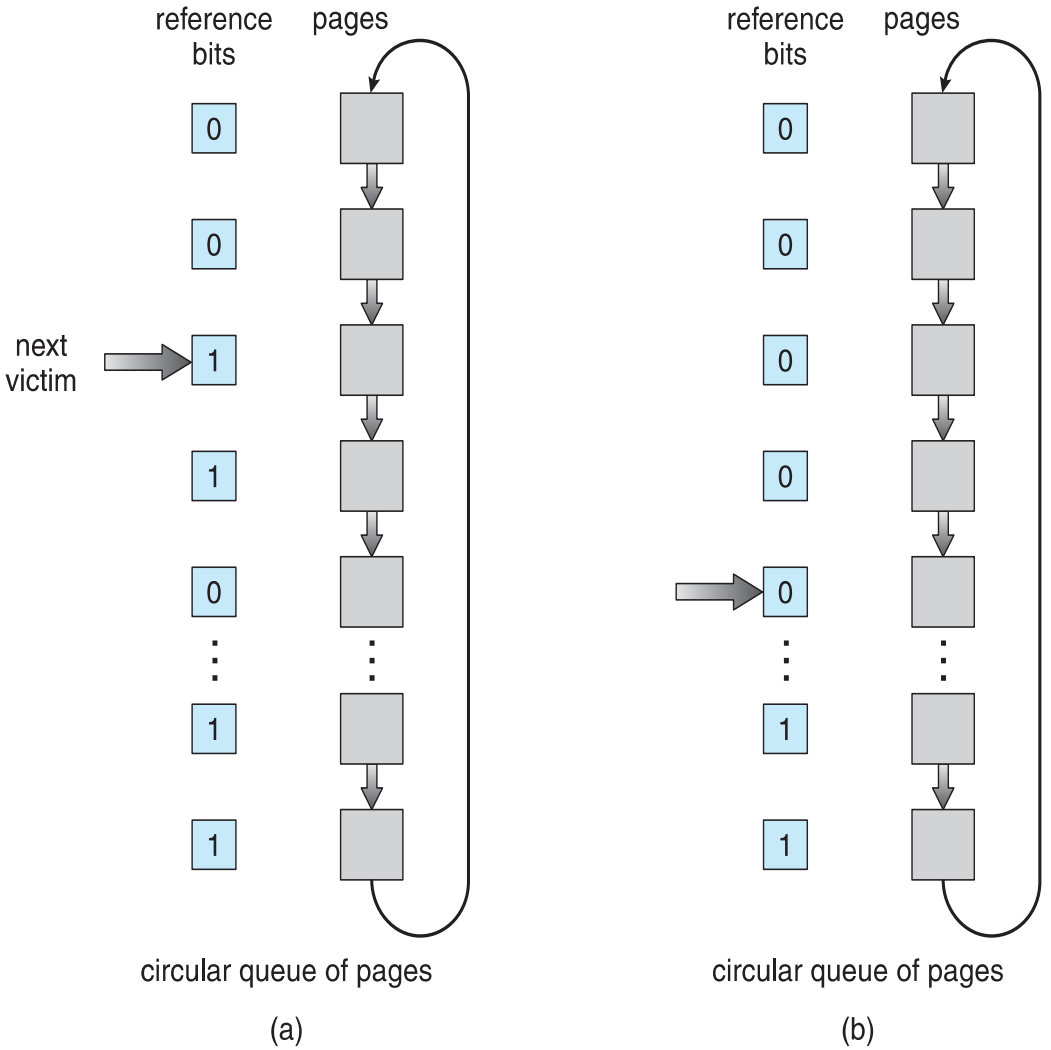
Algoritmi di approssimazione a LRU – 1

- LRU presuppone la presenza di hardware dedicato e rimane, comunque, un algoritmo lento
- Bit di riferimento
  - A ciascuna pagina si associa un bit, inizialmente = 0
  - Quando si fa riferimento alla pagina si pone il bit a 1
  - Si rimpiazza la pagina il cui bit di riferimento vale 0 (se esiste)
  - Tuttavia… non si conosce l'ordine di accesso alle pagine
- Alternativamente…
  - Registrazione dei bit di riferimento ad intervalli regolari (100 msec) in un registro a scorrimento
  - p_i con registro di scorrimento 11000100 acceduta più recentemente di p_j con valore 01100101
  - La pagina LRU è quella cui corrisponde l'intero unsigned più piccolo

Algoritmi di approssimazione a LRU – 2

- Algoritmo Seconda Chance (o Clock)
  - Algoritmo di base: FIFO
  - È necessario un bit di riferimento
  - Quando la pagina riceve una seconda chance, il bit di riferimento viene azzerato ed il tempo di arrivo viene aggiornato al tempo attuale
  - Se la pagina da rimpiazzare (in ordine di clock) ha il bit di riferimento a 0, viene effettivamente rimpiazzata
  - Se la pagina da rimpiazzare (in ordine di clock) ha il bit di riferimento a 1, allora:
    - 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

Algoritmo Seconda Chance



Algoritmi di approssimazione a LRU – 3

- Algoritmo Seconda Chance migliorato
  - Si considera la coppia (bit_riferimento, bit_modifica)
  - Si ottengono quattro classi:
    - (0,0), non recentemente usata né modificata – rappresenta la migliore scelta per la sostituzione
    - (0,1), non usata recentemente, ma modificata – deve essere salvata su memoria di massa
    - (1,0), usata recentemente, ma non modificata – probabilmente verrà di nuovo acceduta a breve
    - (1,1), usata recentemente e modificata – verrà probabilmente acceduta di nuovo in breve termine e deve essere salvata su memoria di massa prima di essere sostituita
  - Si sostituisce la prima pagina che si trova nella classe minima non vuota
  - La coda circolare deve essere scandita più volte prima di reperire la pagina da sostituire

Algoritmi con conteggio

- Si mantiene un contatore del numero di riferimenti che sono stati fatti a ciascuna pagina
- Algoritmo LFU (*Least Frequently Used*): si rimpiazza la pagina col valore più basso del contatore
- Algoritmo MFU (*Most Frequently Used*): è basato sulla assunzione che la pagina col valore più basso del contatore è stata spostata recentemente in memoria e quindi deve ancora essere impiegata
- L'implementazione è molto costosa; le prestazioni (rispetto all'algoritmo ottimo) scadenti

Qualche trucco… – 1

- Si mantiene sempre un pool di frame liberi:
  - Quando si verifica un page fault, si seleziona un frame vittima, ma prima di trascriverlo in memoria secondaria, si procede alla copia della pagina richiesta in un frame del pool
  - ⇨Il processo può essere riattivato rapidamente, senza attendere la fine dell'operazione di salvataggio del frame vittima sulla memoria di massa
- Alternativamente:
  - Quando si verifica un page fault, prima di accedere alla memoria di massa, si controlla se la pagina richiesta è ancora presente nel pool dei frame liberi (occorre mantenere informazione sul numero di pagina/processo)
  - ⇨Non è necessaria nessuna operazione di I/O
  - ⇨Utile quando una pagina, selezionata come vittima, viene rapidamente reintegrata

Qualche trucco… – 2

- Si mantiene una lista delle pagine modificate e, ogni volta che il dispositivo di paginazione è inattivo:
  - Si sceglie una pagina modificata, la si salva sulla memoria secondaria e si reimposta il relativo bit di modifica
  - ⇨ Aumenta la probabilità che, al momento della selezione di una vittima, questa non abbia subito modifiche e non debba essere trascritta su memoria di massa

Esempio 1

- Un processo in esecuzione genera i seguenti riferimenti ad indirizzi logici (per il fetch di istruzioni e l'accesso a dati):

5130 2070 4100 1035 5132 2070 6150 5134 7200 6150 5136 6152 5138 2070

Sapendo che il sistema operativo implementa la memoria virtuale paginata con pagine da 1KB si trasformi la sequenza di indirizzi in una sequenza di numeri di pagine logiche, quindi si calcoli il numero di page fault applicando gli algoritmi di sostituzione FIFO, OPT ed LRU supponendo che al processo vengano allocati quattro frame.

Esempio 1 (cont.)

- **Soluzione**

  Con pagine da 1KB, la stringa di riferimenti da considerare risulta

5130	2070	4100	1035	5132	2070	6150	5134	7200	6150	5136	6152	5138	2070
5	2	4	1	5	2	6	5	7	6	5	6	5	2

FIFO

5°	6	6	6°	6
2°	2	5	5°	5
4	4	4	7	7
1	1	1	1	2

8 page fault

OPT

5°	5°	5°
2°	2	2°
4	6	6°
1	1	7

6 page fault

LRU

5°	5°	5°
2°	2	2°
4	6	6°
1	1	7

6 page fault

Esempio 2

- Si consideri il seguente frammento di programma in linguaggio C:

```
{
   ...
   int a[1000], b[1000];

   for (i=0; i<999; i++)
        a[i]=a[i]+b[i];
   ...
}
```

- Si supponga che un intero occupi una parola (2 byte) e si calcoli il numero di page fault generati dal programma se si possono allocare, in ogni momento…
  - …2 pagine contenenti 100 parole per ognuno dei vettori **`a`** e **`b`**;
  - …3 pagine contenenti 100 parole per ognuno dei vettori **`a`** e **`b`**;
  - …2 pagine contenenti 500 parole per ognuno dei vettori **`a`** e **`b`**.
- Si utilizzi la strategia FIFO; si assuma infine che il codice e la variabile **`i`** siano collocati in un'altra pagina e che nessun accesso a tali entità provochi un page fault.

Esempio 2 (cont.)

- Soluzione

```
{
   ...
   int a[1000], b[1000];

   for (i=0; i<999; i++)
        a[i]=a[i]+b[i];

   ...
}
```

La pagina che contiene il codice e la variabile **`i`** è residente in memoria:

- Per 2 pagine contenenti 100 parole per ognuno dei vettori **`a`** e **`b`** si hanno 20 page fault;
- Per 3 pagine contenenti 100 parole per ognuno dei vettori **`a`** e **`b`** si hanno 20 page fault;
- Per 2 pagine contenenti 500 parole per ognuno dei vettori **`a`** e **`b`** si hanno 4 page fault.

Esempio 3

- Si consideri la seguente stringa di riferimenti alla memoria di un processo in un sistema con memoria virtuale:

7 10 15 19 23 10 7 5 19 15 14 17 19 15 16 18 17 14 23

  - Illustrare il comportamento dell'algoritmo LRU di sostituzione delle pagine per una memoria fisica di 5 blocchi e calcolare il numero di page fault.
  - Illustrare il comportamento dell'algoritmo Clock di sostituzione delle pagine per una memoria fisica di 5 blocchi e calcolare il numero di page fault.

Esempio 3 (cont.)

- Soluzione

7 10 15 19 23 10 7 5 19 15 14 17 19 15 16 18 17 14 23

LRU

7	7	7	7	17	17	17	17	17
10	10	10	14	14	14	18	18	18
15	5	5	5	5	16	16	16	16
19	19	19	19	19	19	19	14	14
23	23	15	15	15	15	15	15	23

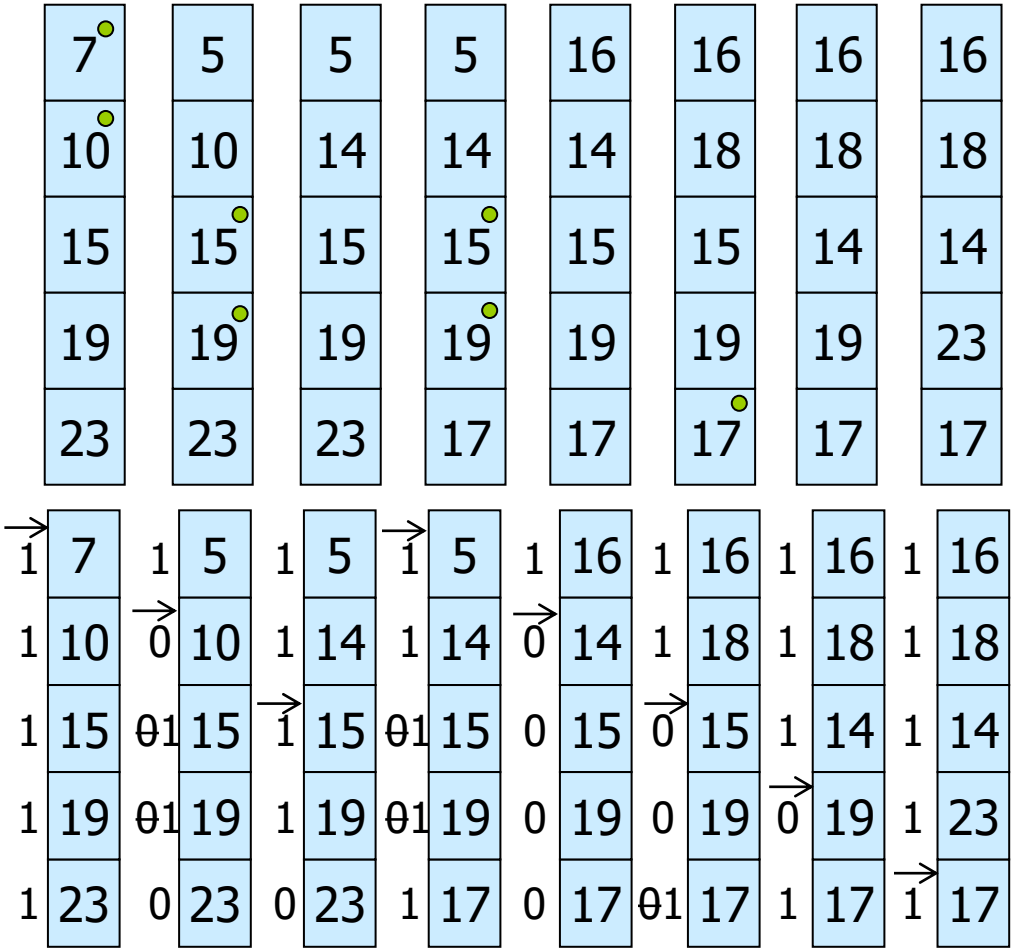
13 page fault

Esempio 3 (cont.)

- Soluzione

7 10 15 19 23 10 7 5 19 15 14 17 19 15 16 18 17 14 23

Clock



12 page fault

Allocazione dei frame – 1

- Ciascun processo richiede un numero minimo di pagine fisiche
  - Il numero minimo di frame necessari per ogni processo è definito dall'architettura, il massimo dalla disponibilità di memoria
  - Infatti, quando si verifica un page fault prima che sia stata completata l'esecuzione di un'istruzione, quest'ultima deve essere riavviata
    - I frame disponibili devono essere in numero sufficiente per contenere tutte le pagine cui ogni singola istruzione può far riferimento

Allocazione dei frame – 2

- Esempio 1 — nel computer IBM System/370 (1970), l'istruzione **SS MOVE** (da memoria a memoria, muove blocchi di 256 byte) può utilizzare fino a 6 frame:
  - L'istruzione occupa 6 byte, quindi può dividersi su 2 pagine
  - 2 pagine per gestire from
  - 2 pagine per gestire to
- Esempio 2 — nei processori Intel, l'operazione **MOVE** consente di spostare dati solo da registro a registro o tra registro e memoria (ma non da memoria a memoria) per limitare il numero minimo di frame per ogni processo
- Si hanno due schemi principali di allocazione:
  - Allocazione statica
  - Allocazione a priorità

Allocazione statica

- **Allocazione uniforme** — I frame si distribuiscono equamente fra i processi presenti nel sistema; per esempio, se si hanno a disposizione 100 frame per 5 processi, si assegnano 20 pagine a ciascun processo
  - Alternativa: si mantiene un pool di frame liberi (esempio: 18 frame allocati a ciascun processo, 10 frame liberi)
- **Allocazione proporzionale** — Si allocano frame sulla base della dimensione del processo
  - s_i = dimensione del processo p_i (in pagine)
  - $S = \Sigma\ s_i$
  - m = numero complessivo dei frame
  - a_i = allocazione per il processo p_i
    $= (s_i/S)\times m$

ESEMPIO

$m = 64$

$s_1=10$

$s_2=127$

$a_1=(10/137)\times 64\approx 5$

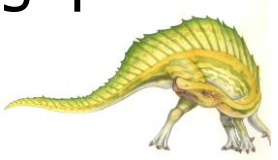
$a_2=(127/137)\times 64\approx 59$

Allocazione con priorità

- Si impiega uno schema di allocazione proporzionale basato sulla priorità piuttosto che sulla dimensione (o basato su una combinazione di entrambe)
- Inoltre, se il processo P_i genera un page fault…
  - Si seleziona per la sostituzione un frame di un processo con priorità minore
  - Si seleziona per la sostituzione uno dei suoi frame

Sostituzione locale e globale

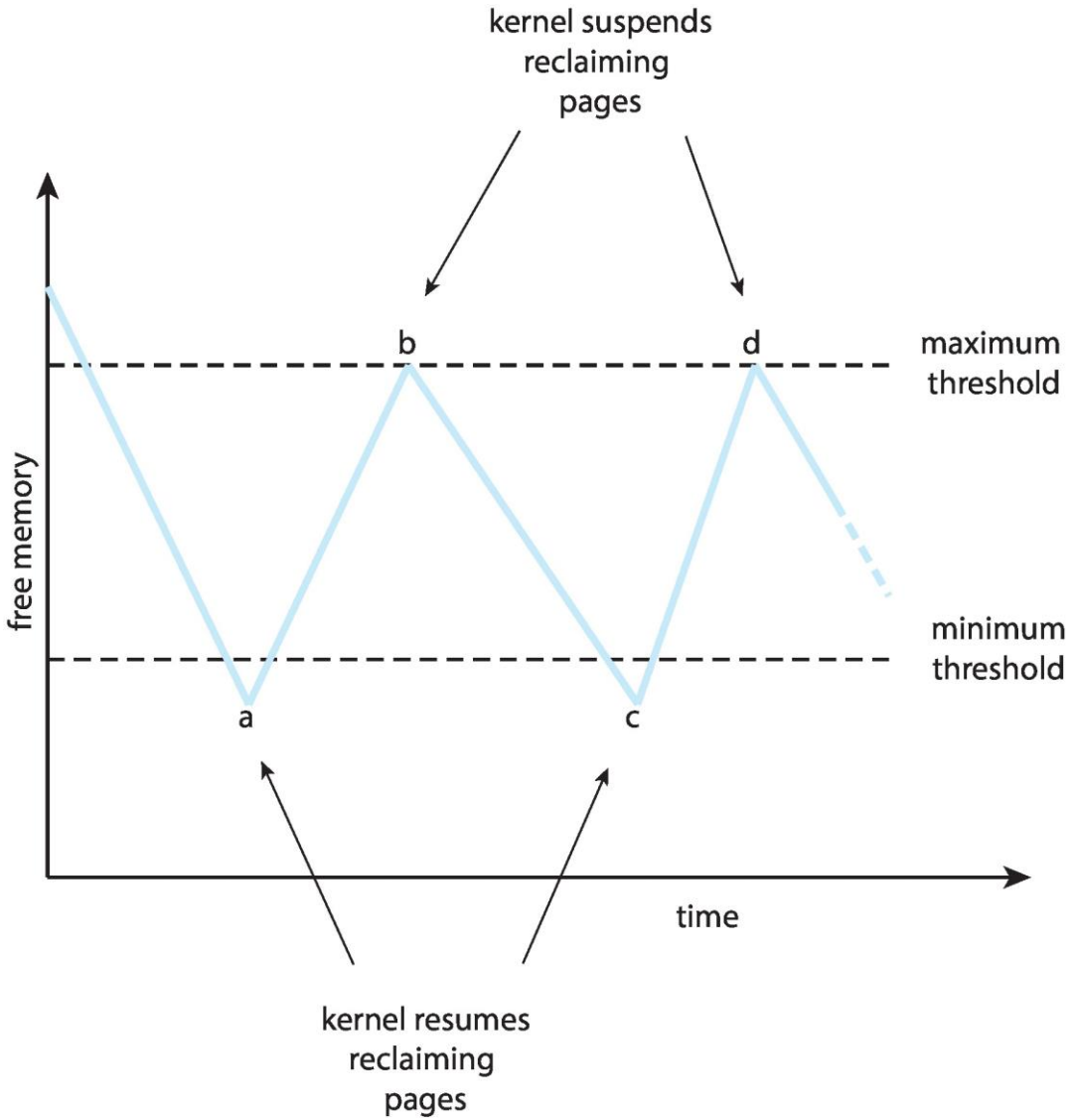
- Sostituzione globale — il SO seleziona il frame da sostituire dall'insieme di tutti i frame; il processo può selezionare un frame di un altro processo
  - ⇨Un processo non può controllare la propria frequenza di page fault
  - ⇨Il tempo di esecuzione di ciascun processo può variare in modo significativo (in base al carico del sistema)
- Sostituzione locale — il SO, per ciascun processo, seleziona i frame solo dal relativo insieme di frame ad esso allocati
  - ⇨Non rende disponibili a processi che ne facciano richiesta pagine di altri processi scarsamente utilizzate
  - ⇨Possibile un utilizzo non efficiente della memoria
- La sostituzione globale garantisce maggior throughput
  - ⇨Normalmente implementata nei SO più diffusi



Recupero di pagine – 1

- Strategia per implementare la politica di sostituzione globale delle pagine
- Tutte le richieste di memoria sono soddisfatte dall'elenco dei frame liberi, evitando di attendere che l'elenco scenda a zero prima di iniziare a selezionare le pagine per la sostituzione
- La sostituzione della pagina viene attivata quando il numero di frame liberi scende al di sotto di una soglia predeterminata
- Si tenta di garantire che ci sia sempre sufficiente memoria libera per soddisfare nuove richieste
- La routine del kernel che recupera le pagine è detta reaper (mietitrice) e può applicare una qualsiasi strategia per la selezione delle vittime

Recupero di pagine – 2



Recupero di pagine – 3

- Per recuperare pagine, in genere, il reaper utilizza LRU (o Clock), ma può diventare "più aggressivo", passando a FIFO, quando non riesce a mantenere la lista dei frame liberi al di sopra della soglia minima
- In LINUX, quando la quantità di memoria libera scende a livelli molto bassi, la routine OOM (*Out–Of–Memory killer*) seleziona un processo da eliminare, recuperando tutta la sua memoria
  - Ogni processo è dotato del proprio punteggio OOM (dipendente dalla sua dimensione): più è alto più è facile che venga selezionato per l'eliminazione
  - Esempio: l'OOM del processo con pid 2500 è visibile in **`/proc/2500/oom_score`**

Page fault principali e secondari

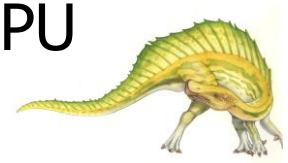
- Un page fault principale si verifica quando si fa riferimento ad una pagina che non è in memoria
  - Per risolverlo occorre caricare la pagina desiderata e aggiornare la tabella delle pagine
- I page fault secondari si verificano invece quando un processo non ha una mappatura logica per una pagina, anche se la pagina è in memoria
  - Riferimenti a librerie condivise
  - La pagina è sempre presente nella lista dei frame liberi
    - In entrambi i casi occorre solo aggiornare la tabella delle pagine ⇨ tempo di servizio ridotto
- In Linux, usare il comando

```
ps -eo min_flt, maj_flt, cmd
```

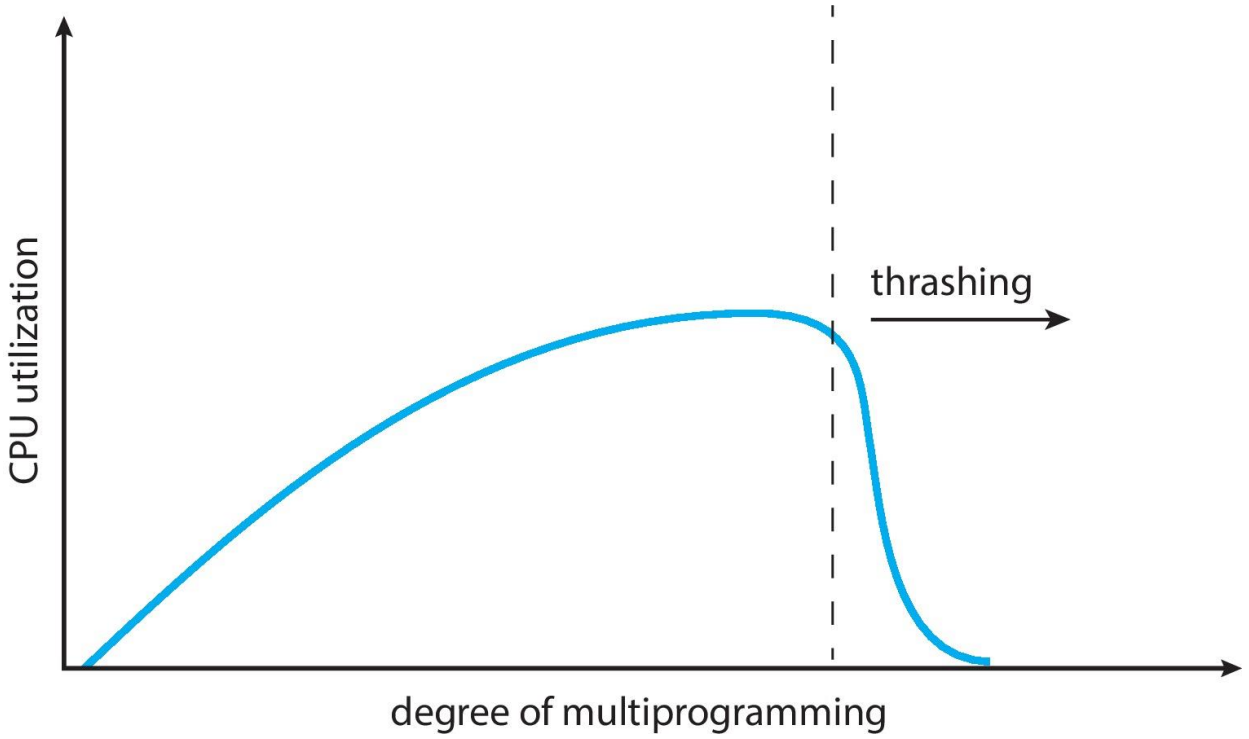
che restituisce il numero di page fault secondari, principali ed il nome del comando che li ha provocati

Thrashing – 1

- Se un processo non ha abbastanza pagine fisiche a disposizione, la frequenza di page fault è molto alta
  - Si effettua un page fault per caricare una pagina in memoria e si rimpiazza un frame esistente…
  - …ma presto il contenuto di quel frame sarà di nuovo necessario
  - Conseguenze:
    - Basso impiego della CPU
    - Il SO crede di dover aumentare il grado di multiprogram-mazione
    - Si aggiunge un altro processo al sistema!
- Thrashing ≡ un processo è costantemente occupato a spostare pagine dalla backing store alla memoria e viceversa e non esegue calcoli all'interno della CPU



Thrashing – 2



La paginazione degenera ⇨ thrashing

Thrashing – 3

- Più in dettaglio, si produce la seguente situazione:
  - La memoria fisica libera è insufficiente a contenere i frame necessari all'esecuzione dei processi, che genera-no numerosi page fault rallentando considerevolmente la propria velocità
  - Il SO deduce erroneamente che sia necessario aumenta-re il grado di multiprogrammazione (dato che la CPU rimane per la maggior parte del tempo inattiva a causa dell'intensa attività di I/O)
  - Vengono avviati nuovi processi che, per mancanza di frame liberi, innescheranno, a loro volta, un'attività di paginazione molto intensa
  - In breve, le prestazioni del sistema collassano: la CPU non viene utilizzata, mentre si formano lunghe code di accesso al dispositivo di I/O

Thrashing – 4

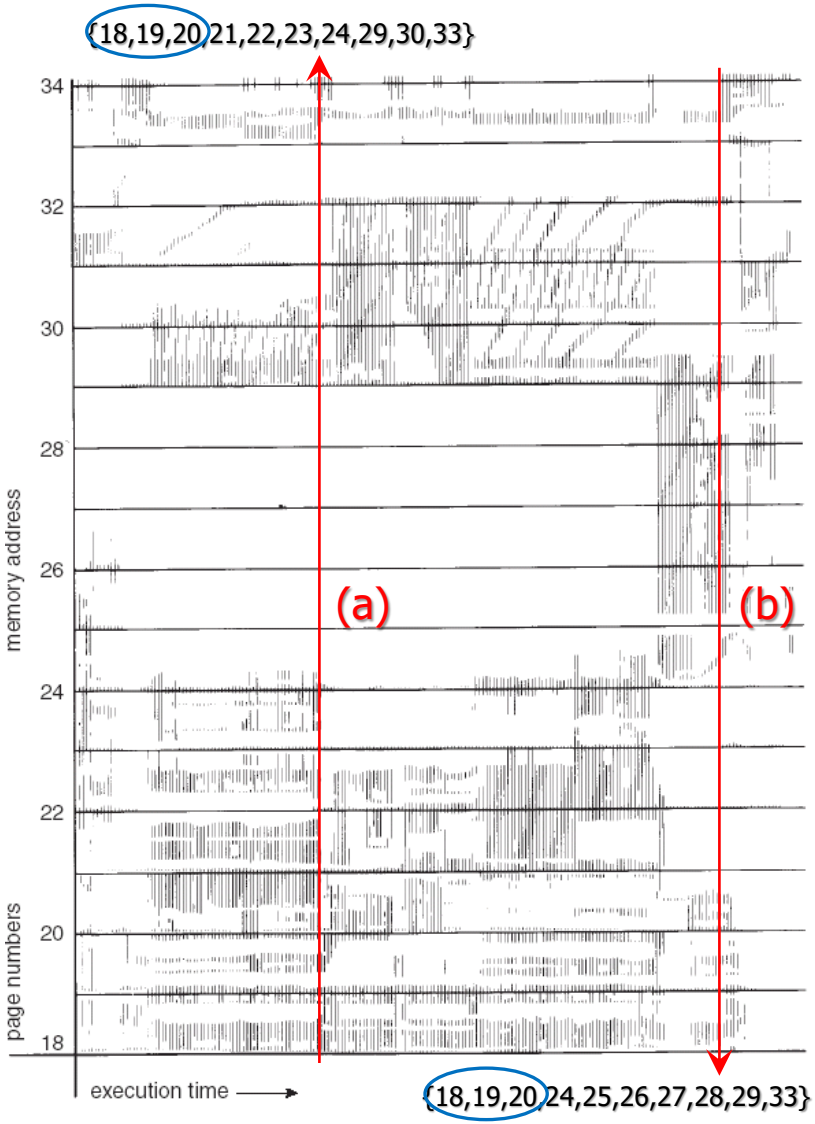
- In altre parole… anche se lo spostamento di frame rappresenta un'operazione standard del funzionamento del sistema di calcolo, è possibile che si verifichi un paging eccessivo
- Il thrashing è estremamente dannoso per le prestazioni, perché l'inattività della CPU ed i carichi di I/O che si generano possono avere il sopravvento sul normale funzionamento del sistema
- In casi estremi, il sistema potrebbe non funzionare utilmente, spendendo tutte le sue risorse spostando le pagine da e verso la memoria

Thrashing – 5

- Gli effetti del thrashing possono essere limitati utilizzando un algoritmo di sostituzione locale o per priorità
  - Con la sostituzione locale, se l'attività di paginazione di un processo degenera in thrashing, il processo non può sottrarre frame ad altri processi, provocandone l'ulteriore degenerazione
  - Tuttavia, i processi in thrashing rimangono nella coda del dispositivo di paginazione per la maggior parte del tempo, rallentando comunque l'attività di paginazione dei processi non in thrashing
- Per evitare il thrashing occorre fornire al processo tutte le pagine che fanno parte della sua **località attuale**

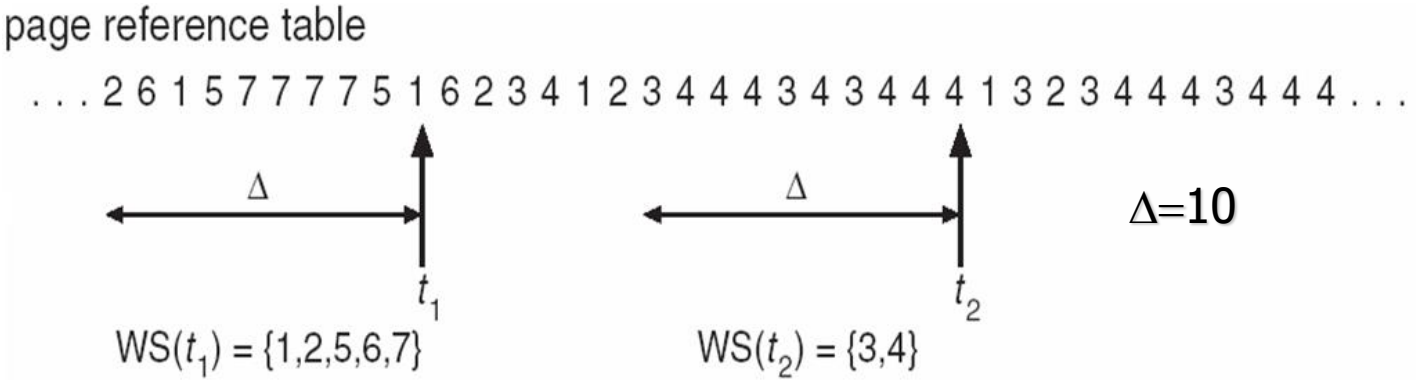
Pattern di località nei riferimenti alla memoria

- Le località hanno una connotazione spazio–temporale
- Esempio
  - Quando viene invocato un sottoprogramma, si definisce una nuova località: vengono fatti riferimenti alle sue istruzioni, alle sue variabili locali ed a un sottoinsieme delle variabili globali
  - Quando il sottoprogramma termina, il processo lascia la località corrispondente
  - Località definite dalla struttura del programma e dei dati
    - I processi migrano da una località all'altra
    - Le località possono essere parzialmente sovrapposte



Modello working–set

- $\Delta \equiv$ "finestra scorrevole" $\equiv$ un numero fissato di riferimenti alla memoria (per esempio, 10.000 riferimenti alla memoria)
- WS_i (working–set del processo P_i) = numero di pagine referenziate nel più recente Δ (varia nel tempo)
  - se Δ è troppo piccolo non comprende tutta la località
  - se Δ è troppo grande comprenderà più località
  - se $\Delta = \infty$ ⇨ comprende l'intero processo
- $D = \Sigma_i WS_i \equiv$ numero totale di pagine richieste
- se $D > m$ (numero totale dei frame) ⇨ Thrashing
- **Politica:** se $D > m$, occorre sospendere un processo e sottoporlo a swapping

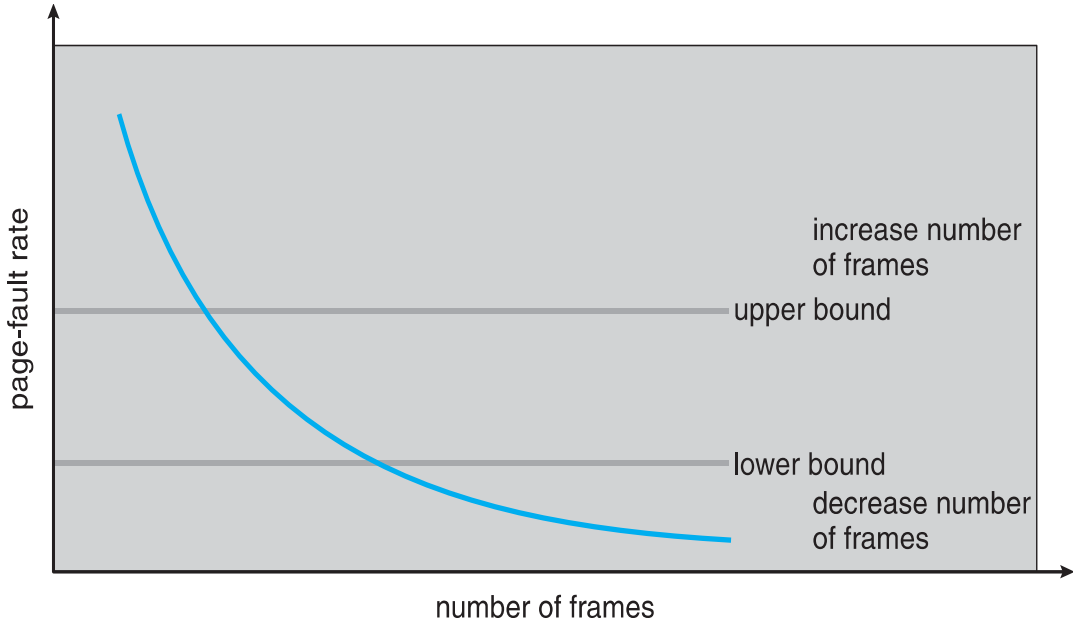


Come stabilire il working–set?

- Problema: la finestra del working–set è "mobile", con riferimenti che entrano ed escono dal working–set
- Si approssima con un interrupt del timer e un bit di riferimento
- Esempio: $\Delta = 10.000$ unità di tempo
  - Il timer emette un interrupt ogni 5000 unità di tempo
  - Si tengono in memoria 2 bit per ogni pagina
  - Quando si ha un interrupt del timer, si copiano in memoria i valori di tutti i bit di riferimento e si pongono a 0
  - Se uno dei bit è 1 ⇨ pagina nello working–set (con riferimento negli ultimi 10.000–15.000 accessi)
- Questo approccio non è completamente accurato
- Miglioramento: 10 bit e interrupt ogni 1000 unità di tempo

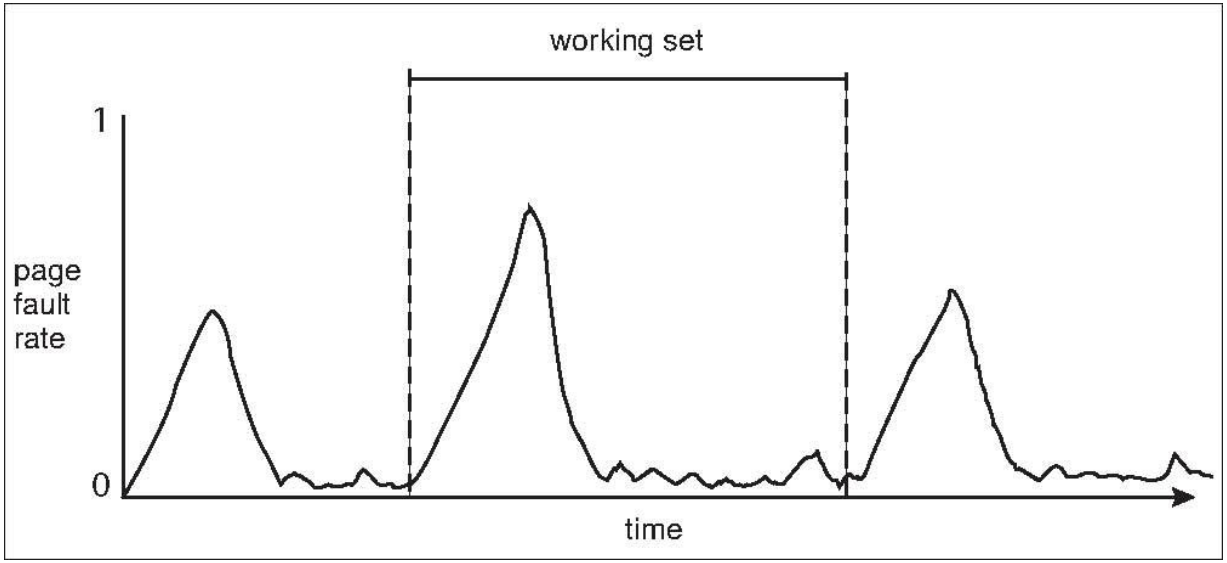
Frequenza di page fault

- Approccio più diretto e intuitivo rispetto all'utilizzo del working–set (che è invece utile per la prepaginazione)
- Si stabilisce una frequenza di page fault "accettabile" e si utilizza una politica di sostituzione locale
  - Se la frequenza effettiva è troppo bassa, il processo rilascia dei frame
  - Se la frequenza è troppo elevata, il processo acquisisce nuovi frame



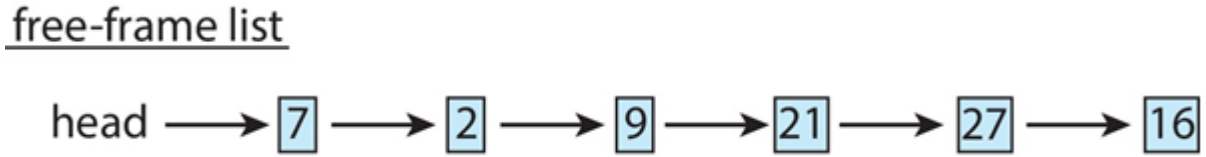
Working–set e frequenza di page fault

- Vi è comunque una relazione diretta tra working–set di un processo e frequenza di page fault
- Il working–set varia nel tempo
- Quando si verifica un cambio di località del processo si ha un picco nel grafico della frequenza di page fault



Compressione della memoria – 1

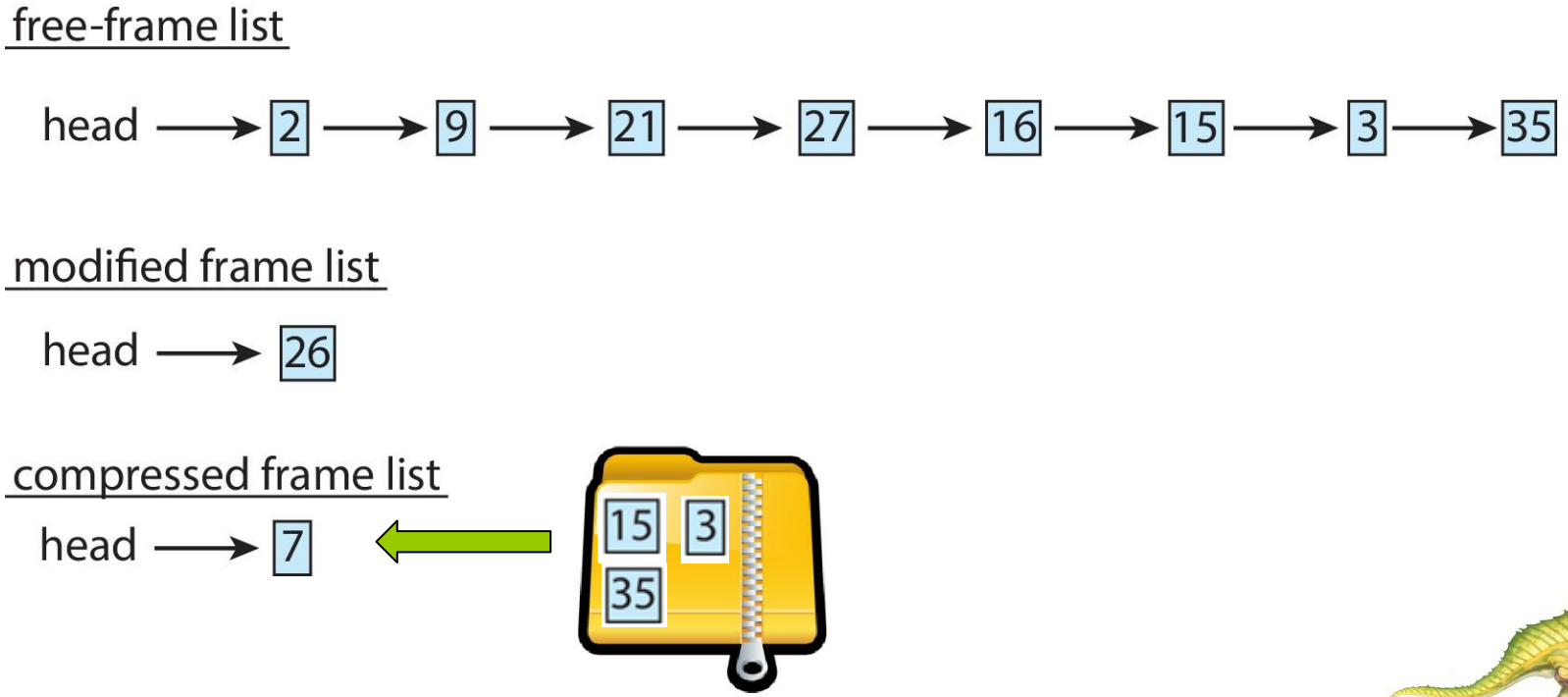
- Piuttosto che paginare i frame modificati per creare spazio, si comprimono diversi frame in un singolo frame, consentendo al sistema di ridurre l'utilizzo della memoria senza ricorrere al paging
- Si consideri la seguente lista di frame liberi composta da 6 frame



- Supponiamo che il numero di frame liberi scenda al di sotto di una certa soglia, che innesca la sostituzione delle pagine
- L'algoritmo di sostituzione (ad esempio, un algoritmo di approssimazione a LRU) seleziona quattro frame, 15, 3, 35 e 26 da inserire nell'elenco dei frame liberi, preventivamente inserendoli nella lista dei frame modificati

Compressione della memoria – 2

- I frame modificati vengono quindi scritti nell'aria di avvicendamento, rendendo i frame disponibili all'uso
- Una strategia alternativa consiste nel comprimere un certo numero di frame (per esempio 3) da memorizzare in un unico frame



Compressione della memoria – 3

- Compressione della memoria
  - Fondamentale per i sistemi mobili, che non utilizzano lo swapping (iOS e Android)
  - Supportata anche da Windows 11 e MacOS
  - Per MacOS le prestazioni ottenute con la compressione sono migliori rispetto al paging anche in presenza di supporti veloci come gli SSD
- Rapporto di compressione e algoritmi utilizzati sono "antitetici"
  - ⇨ Rapporti di compressione più elevati, algoritmi di compressione computazionalmente più costosi

Allocazione di memoria al kernel

- Il kernel, per allocare la propria memoria, attinge ad una riserva di memoria libera diversa dalla lista dei frame usata per soddisfare i processi utente
  - Il kernel richiede memoria per strutture dati dalle dimensioni variabili, di solito molto più piccole di un frame
    - Uso oculato della memoria per evitare gli sprechi: in molti SO, codice e dati del kernel non sono soggetti a paginazione
  - Parti della memoria del kernel devono essere contigue perché alcuni dispositivi accedono direttamente alla memoria fisica senza l'interfaccia della memoria virtuale (per esempio, i device di I/O)

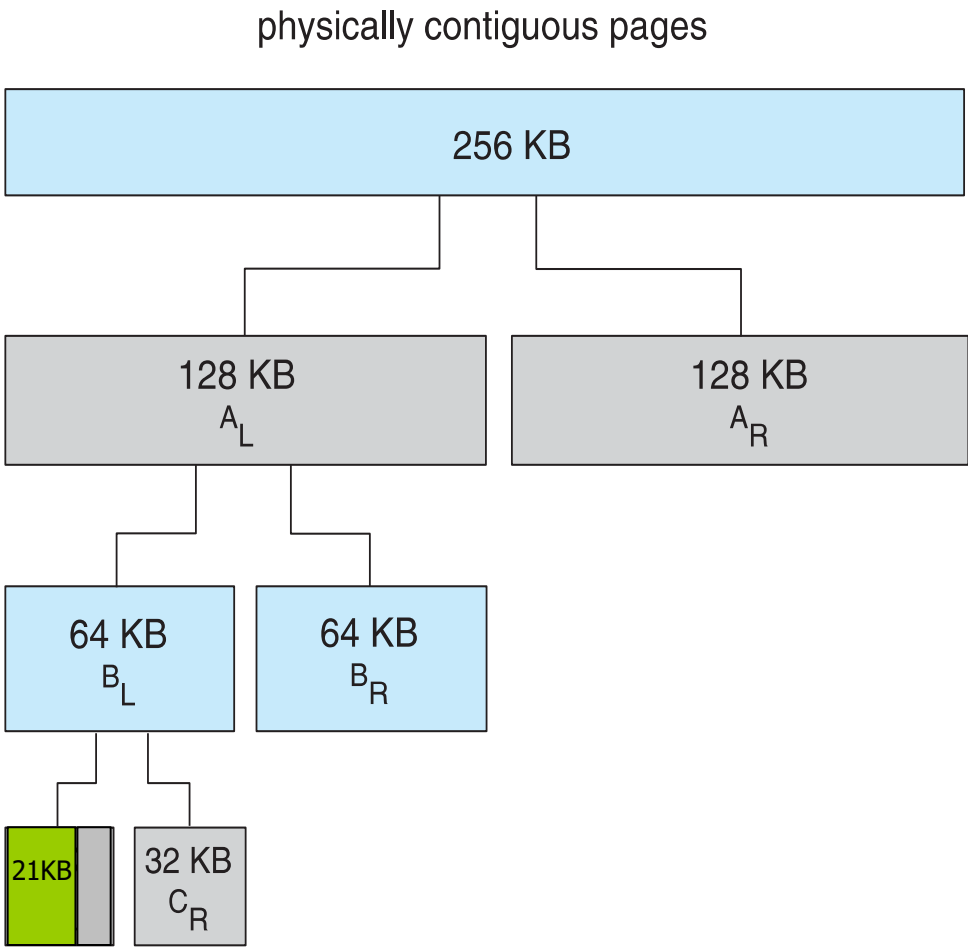
Sistema buddy – 1

- Utilizza un segmento di dimensione fissa per l'allocazione della memoria, composto da pagine fisicamente contigue
- La memoria viene allocata attraverso un allocatore–potenza–di–2
  - Alloca memoria in blocchi di dimensione pari a potenze del 2
  - La quantità richiesta viene arrotondata alla più piccola potenza del 2 che la contiene
  - Quando si richiede meno memoria di quella che costituisce il segmento corrente, questo viene diviso in due segmenti gemelli di identica dimensione
    - Il procedimento continua fino ad ottenere il segmento minimale per l'allocazione richiesta

Sistema buddy – 2

- Per esempio, si assuma di avere a disposizione un segmento da 256KB e che il kernel richieda invece 21KB
  - Si ottengono dapprima due segmenti A_L e A_R, ciascuno di 128KB
    - Un'ulteriore divisione produce B_L e B_R di 64KB
      - Infine, si ottengono C_L e C_R da 32KB ciascuno – uno dei quali viene utilizzato per soddisfare la richiesta
- Quando si rilascia memoria, il sistema buddy offre il vantaggio di poter congiungere rapidamente segmenti adiacenti a formare segmenti di memoria contigua più lunghi (tecnica nota come *coalescing* – fusione)
- Tuttavia, l'arrotondamento della dimensione del blocco alla potenza del 2 più piccola in grado di contenerlo può generare frammentazione interna

Sistema buddy – 3



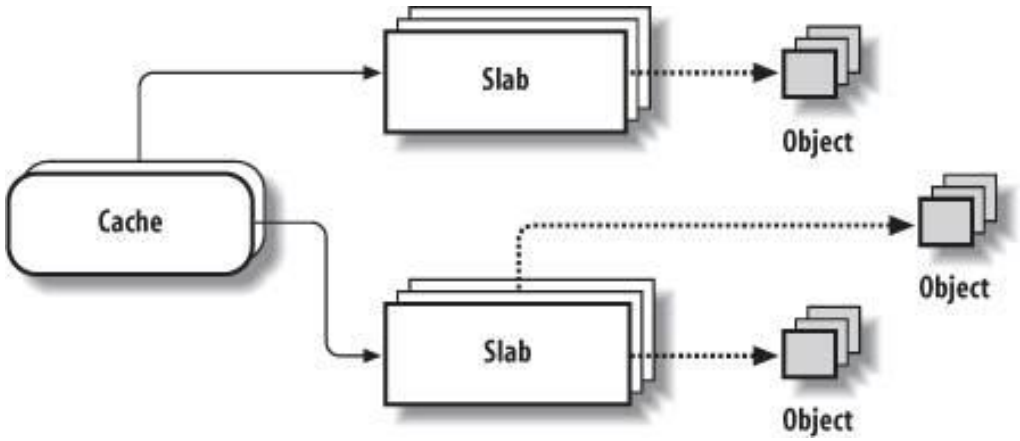
Allocatore–potenza–di–2

Allocazione a lastre – 1

- Una lastra – *slab* – è composta da uno o più frame fisicamente contigui
- Una cache è una lista di una o più lastre
- Vi è una cache per ciascuna categoria di strutture dati del kernel
  - Una cache dedicata ai PCB, una ai descrittori di file, una ai semafori, etc.
  - Ciascuna slab è popolata da oggetti – istanze della relativa struttura dati
- Quando si crea una cache, tutti gli oggetti ad essa relativi sono dichiarati "liberi"
  - Quando una struttura dati del kernel deve essere allocata, si sceglie dalla cache opportuna un qualunque oggetto libero e lo si marca come "in uso"

Allocazione a lastre – 2

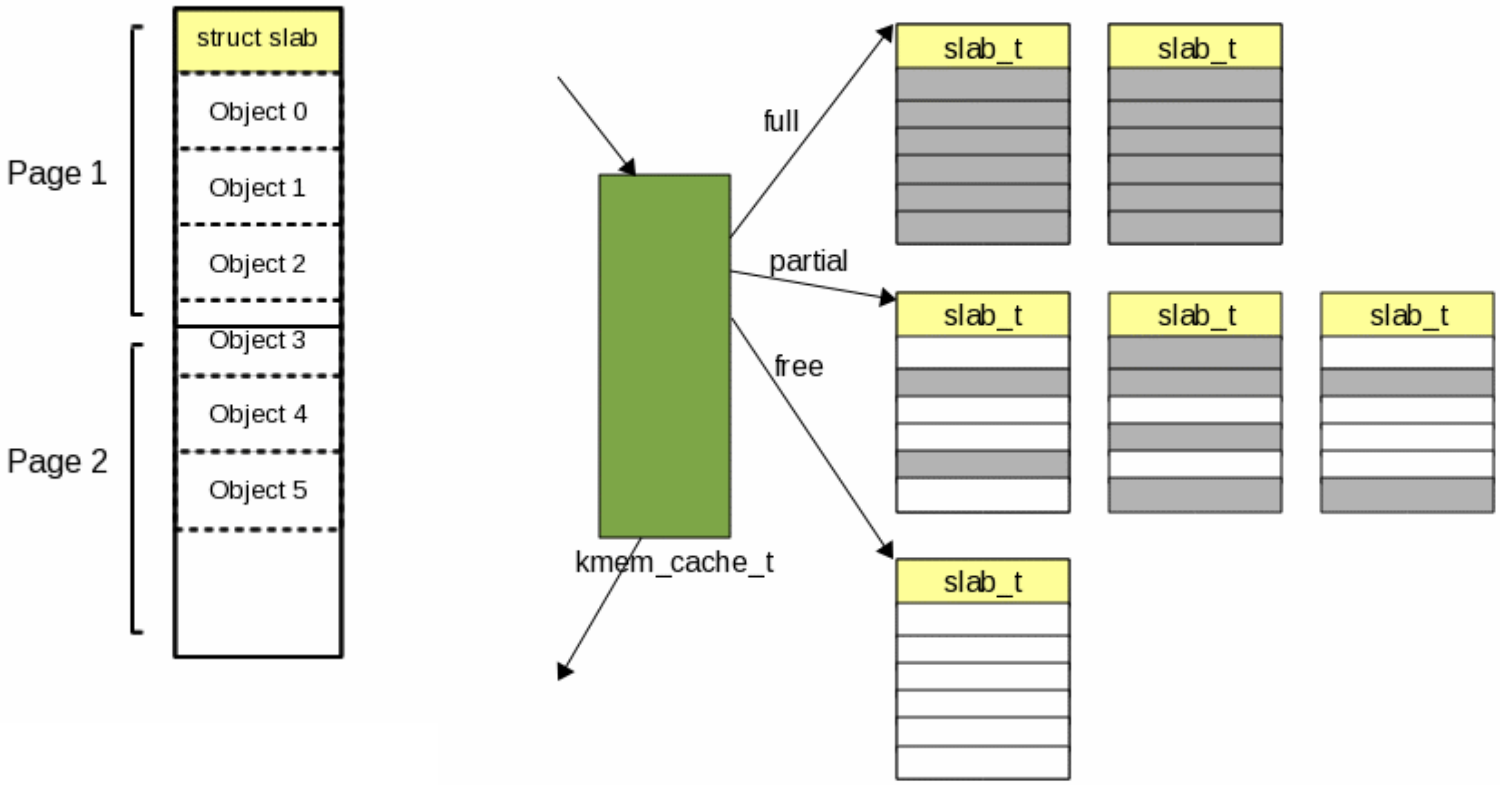
- Ovvero…
  - Lo slab allocator è organizzato in "cache"
  - Ciascuna cache è un "magazzino" di oggetti dello stesso tipo (in particolare della stessa dimensione)
  - Ciascuna cache è costituita da una o più slab
  - Ciascuna slab è un insieme di frame fisicamente contigui
  - In ogni slab sono memorizzati un certo numero di oggetti



Allocazione a lastre – 3

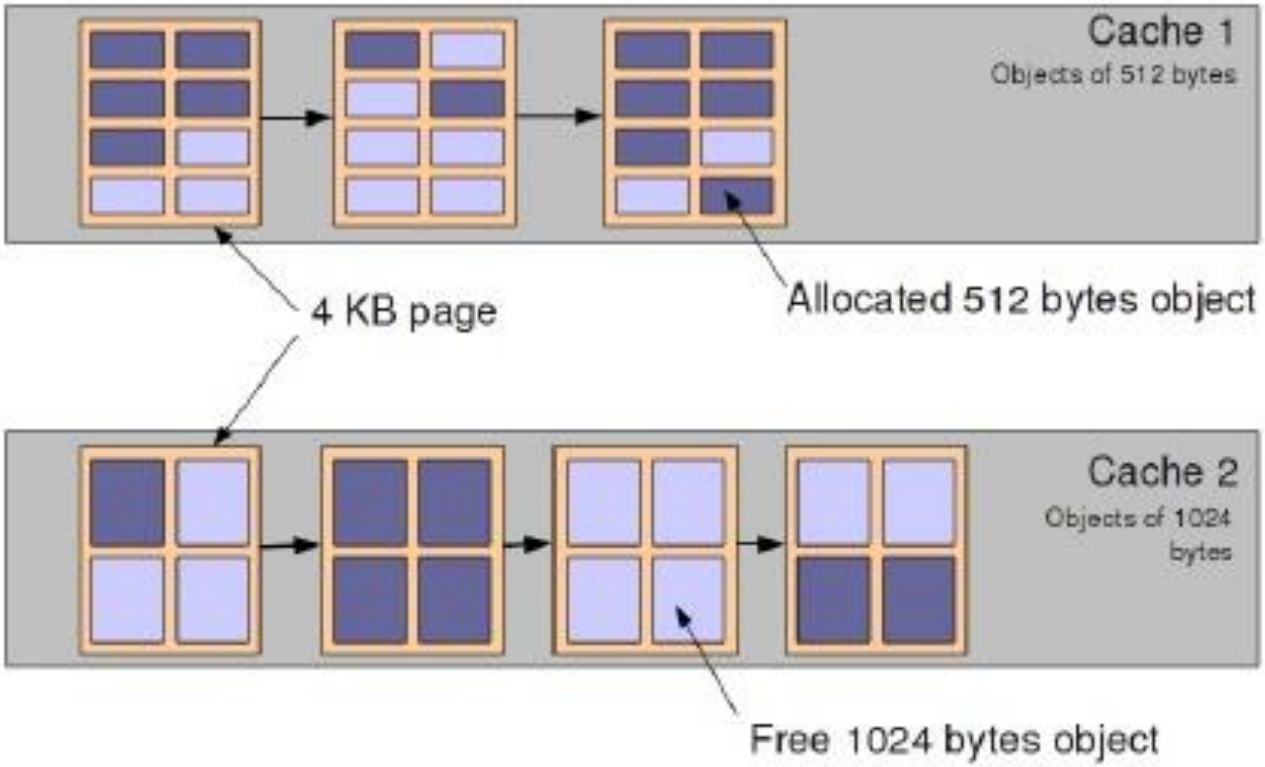
- Stati di una slab
  - Piena – tutti gli oggetti contrassegnati come usati
  - Vuota – tutti gli oggetti contrassegnati come liberi
  - Parzialmente occupata – la lastra contiene oggetti sia usati sia liberi
- Quando una lastra diventa piena, un nuovo oggetto viene allocato in una slab completamente vuota
- Se non vi sono lastre vuote, si procede all'allocazione di una nuova lastra (cache grow) e la si appende in fondo alla cache opportuna
- Si annulla lo spreco di memoria derivante dalla frammentazione interna
- Le richieste di memoria vengono soddisfatte rapidamente (dato che gli oggetti sono preallocati)

Allocazione a lastre – 4

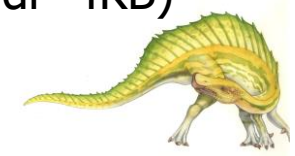


Allocazione a lastre – 5

"*Caching is important because the cost of constructing an object can be significantly higher than the cost of allocating memory for it.*"
Jeff Bonwick (inventore di ZFS e dell'allocatore SLAB, 1965)



Esempio: una cache da 12KB (formata da tre slab contigue di 4KB) potrebbe contenere 6 oggetti da 2KB ciascuno o 24 oggetti da 512B

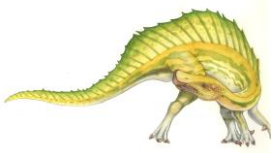


Riduzione della frammentazione interna

- Si può calcolare la dimensione di ciascuna slab in modo da ridurre al minimo la memoria interna sprecata
- Ad esempio, se una cache deve contenere oggetti lunghi 592 byte e la pagina è di 4096 byte:

dim. slab#	oggetti	mem. Persa
1 pf	6	544 B (13%)
2 pf	13	496 B (6%)
3 pf	20	448 B (3.6%)
4 pf	27	400 B (2.4%)

- È possibile sfruttare parte dei byte "sprecati" entro la slab per memorizzare strutture dati necessarie allo slab allocator (ad esempio, lo stato libero/occupato degli oggetti)



Case–study: allocazione a lastre in Linux – 1

- Il descrittore dei task è di tipo **`struct task_struct`** ed occupa approssimativamente 1.7KB di memoria
- Nuovo task ⇨ si alloca una nuova struttura dalla cache
  - Si utilizzerà un elemento libero esistente di tipo **`struct task_struct`**
- All'atto della richiesta di un nuovo elemento, lo slab allocator:
  1. Utilizza una struttura libera in una lastra parzialmente occupata
  2. Se non ve ne sono, seleziona una struttura da una lastra vuota
  3. Se non ci sono lastre vuote, ne crea una nuova (aumentando la dimensione della cache)

Case–study: allocazione a lastre in Linux – 2

- L'allocatore a lastre ha fatto la sua prima apparizione nel kernel di Solaris 2.4, ma è attualmente comune per l'allocazione sia di memoria kernel che utente in diversi SO
- Linux adottava originariamente il sistema buddy ma, dalla versione 2.2, il kernel di Linux include l'allocazione a lastre
- Le distribuzioni recenti includono due allocatori di memoria del kernel, detti SLOB e SLUB
  - SLOB (Simple List Of Blocks), progettato per sistemi con poca memoria (per es., embedded), mantiene tre liste, relative ad oggetti piccoli, medi e grandi
  - SLUB è la nuova versione dell'allocatore a lastre (che ha sostituito SLAB) ottimizzata per sistemi multiprocessore

Altre considerazioni – Prepaging

- Per ridurre il gran numero di page fault necessari allo startup del processo in regime di paginazione pura...
- Prepaging
  - Si portano in memoria tutte o alcune delle pagine necessarie al processo, prima che vengano referenziate; ad esempio, si può memorizzare il working–set al momento della sospensione per I/O per poi riprendere tutte le pagine che gli appartengono
  - Tuttavia... se le pagine precaricate non vengono utilizzate, si spreca I/O e memoria
  - Siano s le pagine prepaginate e sia α la percentuale di esse effettivamente utilizzata
    - Il costo $s \times \alpha$ dei page fault evitati è maggiore o minore del costo di prepaging $s \times (1-\alpha)$ relativo al caricamento di pagine inutilizzate?
    - Per α vicino a zero ⇨ il prepaging non è conveniente

Altre considerazioni – Prepaging

- Prepaging
  - Difficile prepaginare eseguibili perché non si conoscono le parti di codice che verranno raggiunte in un particolare run
  - Facile con i file, che vengono normalmente acceduti in maniera sequenziale
  - La chiamata di sistema **`readahead`** di Linux precarica il contenuto di un file in memoria perché gli accessi successivi avvengano da memoria e non dal file system

Altre considerazioni – Dimensione delle pagine

- Criteri per la determinazione della dimensione delle pagine:
  - Frammentazione interna
  - Risoluzione (isolare solo la memoria necessaria)
  - Località (si riduce l'I/O totale)
    - ⇨pagine di piccole dimensioni
  - Dimensione della tabella delle pagine
  - Dimensione ed efficienza del TLB
  - Sovraccarico (overhead) di I/O
  - Numero di page fault
    - ⇨pagine di grandi dimensioni
- Dimensione delle pagine sempre rappresentata da una potenza di 2, attualmente nell'intervallo da 2^{12} (4KB) a 2^{22} (4MB)
- In media, cresce nel tempo

Altre considerazioni – TLB reach

- **TLB Reach:** quantità di memoria accessibile via TLB, (dim. TLB)×(dim. pagina)
- Idealmente, il working–set di ogni processo dovrebbe essere contenuto nel TLB
  - Altrimenti si verificano molti riferimenti alla tabella delle pagine ed il tempo di esecuzione diviene proibitivo
- Aumentare la dimensione delle pagine
  - Potrebbe portare ad un incremento della frammentazione, dato che non tutte le applicazioni richiedono pagine grandi
- Prevedere pagine di diverse dimensioni
  - Permette l'utilizzo di pagine grandi alle applicazioni che lo richiedono, senza aumento della frammentazione
  - Gestione via SO del TLB

Altre considerazioni – Struttura dei programmi

- `int data[128][128];`
- Ciascuna riga viene memorizzata in una pagina
  - Programma 1

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

  128×128 = 16384 page fault

  - Programma 2

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

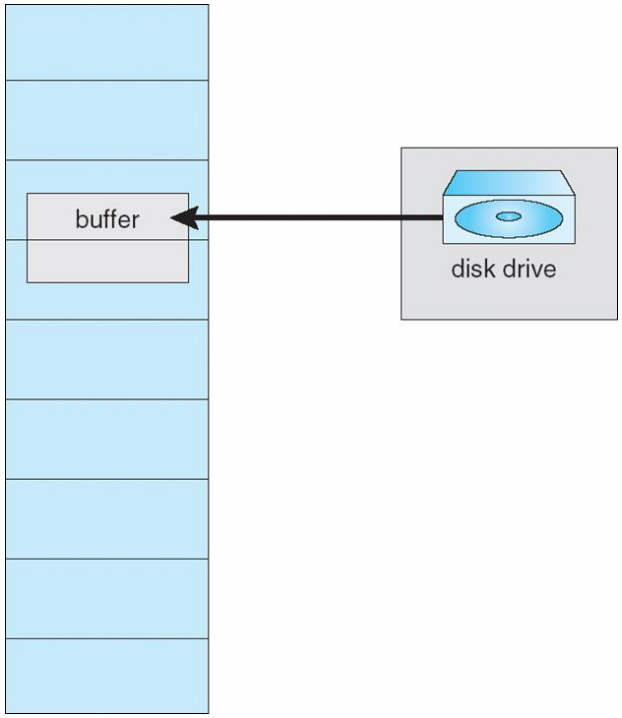
  128 page fault

Altre considerazioni – Struttura dei programmi

- La scelta di strutture dati opportune e della modalità di programmazione può aumentare la località dei processi e ridurre la frequenza dei page fault e la dimensione del working–set
- La separazione di codice e dati garantisce la presenza di pagine mai modificate (codice rientrante) e quindi facilmente sostituibili
- Il caricatore può ottimizzare il numero di page fault dovuti a riferimenti incrociati:
  - non collocando procedure lungo i limiti di pagina
  - "impaccando", nella stessa pagina, procedure che si invocano a vicenda
- Scelta del linguaggio di programmazione
  - Esempio: C/C++, che fanno uso di puntatori, tendono a distribuire casualmente gli accessi alla memoria

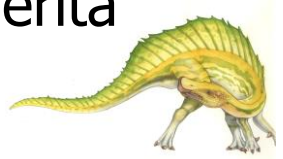
Altre considerazioni – Interlock di I/O

- Talvolta, occorre permettere ad alcune pagine di rimanere bloccate (pinned) in memoria
- Esempio: Le pagine utilizzate per copiare un file da un dispositivo di I/O devono essere bloccate, per non essere selezionate come vittime da un algoritmo di sostituzione



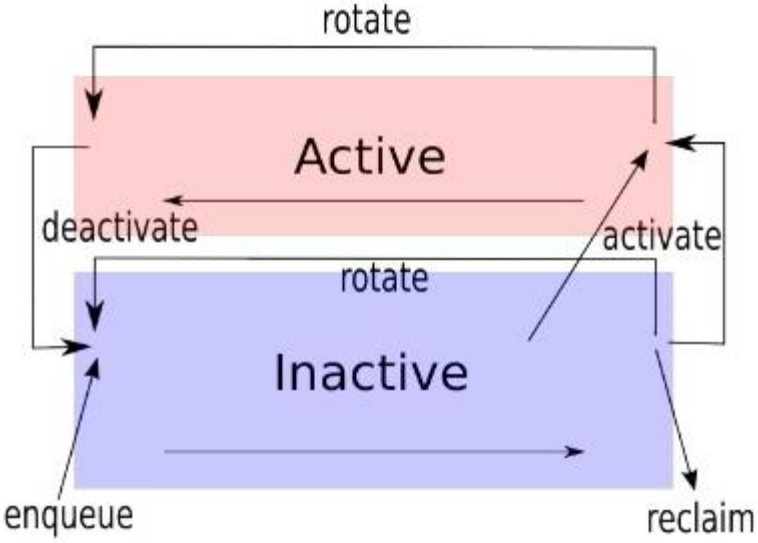
Esempio: Linux – 1

- Impiega la paginazione su richiesta allocando pagine da una lista di frame liberi
  - Politica di sostituzione globale con algoritmo Clock
  - Si mantengono due liste: **`active_list`**, che contiene le pagine in uso, e **`inactive_list`**, per le pagine non riferite di recente, la cui memoria può essere recuperata
- Ogni pagina ha (almeno) un bit di accesso che viene posto a 1 quando si fa riferimento alla pagina
  - Quando una pagina viene allocata per la prima volta, il suo bit viene impostato a 1 e la si pone in fondo alla **`active_list`**, ugualmente quando viene riferita
  - Periodicamente, i bit dei processi attivi vengono resettati
  - La pagina utilizzata meno di recente si sposterà all'inizio dell'**`active_list`** e da lì potrà essere spostata in fondo all'**`inactive_list`**, da cui verrà recuperata, se riferita



Esempio: Linux – 2

- Le liste vengono mantenute bilanciate
  - Quando l'**`active_list`** diventa molto più grande della **`inactive_list`**, le pagine in testa all'**`active_list`** passano all'**`inactive_list`**, dove diventano idonee per il recupero di spazio di memoria
- Il daemon **`kswapd`** del kernel si risveglia periodicamente e controlla la quantità di memoria disponibile
  - Se è sotto soglia, **`kswapd`** inizia la scansione delle pagine nella **`inactive_list`** e le recupera per la lista dei frame liberi



Esempio: Windows

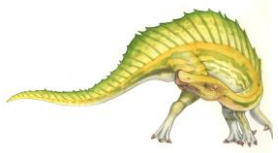
- Impiega la paginazione su richiesta via clustering
  - Con il clustering si gestiscono le assenze di pagina caricando in memoria non solo la pagina richiesta, ma più pagine ad essa (logicamente) adiacenti
- Alla sua creazione, un processo riceve le dimensioni minima e massima del working–set (50–345)
  - Il working–set minimo è il minimo numero di pagine caricate in memoria per un processo (assegnazione garantita dal SO)
  - Ad un processo possono essere assegnate al più tante pagine fisiche quanto vale la dimensione del suo working–set massimo
- Quando la quantità di memoria libera nel sistema scende sotto una data soglia, si effettua un procedimento di ridimensionamento automatico del working–set, per riportare la quantità di memoria libera a livelli "accettabili"
  - Si rimuovono le pagine di quei processi che ne possiedono in eccesso rispetto alla dimensione del minimo working–set

Esercizio 1

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

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

  - Si illustri il comportamento dell'algoritmo LRU nel caso in cui al processo vengano assegnati 4 blocchi fisici, supponendo un regime iniziale di paginazione pura;
  - Come supporto alla trasformazione degli indirizzi viene usato un TLB: l'accesso alla memoria centrale richiede 100νsec e la ricerca nella tabella associativa 5νsec; si calcoli il tempo medio di accesso ai dati supponendo un hit ratio del 90% e che la pagina acceduta sia attualmente presente in memoria.



Esercizio 2

- Supponendo che lo spazio degli indirizzi virtuali sia costituito da otto pagine, mentre la memoria fisica può allocarne effettivamente quattro, si stabilisca quanti page fault hanno luogo utilizzando una strategia di sostituzione LRU, relativamente alla stringa di riferimenti alle pagine

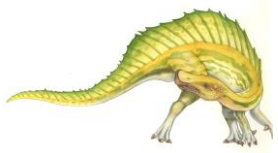
  0 2 1 3 4 6 3 7 4 7 3 3 5 5 3 1 1 1 1 7 2 3 4 1

Esercizio 3

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

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

  si illustri il comportamento dell'algoritmo LRU nel caso che al processo siano assegnati 5 blocchi fisici; si calcoli il numero di page fault, supponendo un regime iniziale di paginazione pura.



Fine del Capitolo 10