Memoria virtuale

Pietro Braione

Reti e Sistemi Operativi – Anno accademico 2021-2022

Obiettivi

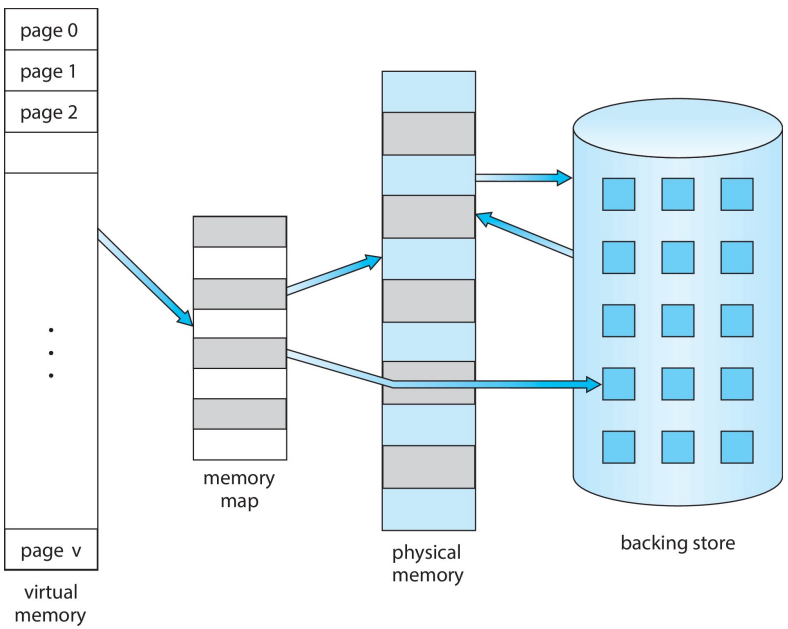
- Definire la memoria virtuale e descriverne i benefici
- Illustrare come le pagine vengono caricate in memoria usando la paginazione su richiesta
- Applicare gli algoritmi FIFO, ottimale e LRU per la sostituzione delle pagine
- Descrivere il working set di un processo e spiegare come è correlato alla località di un programma

Motivazioni

- Abbiamo detto che un processo deve avere la sua immagine completamente in memoria per essere eseguito
- In realtà nella pratica si verificano le seguenti cose:
  - Raramente la memoria di un processo viene usata integralmente
  - Raramente viene usata tutta nello stesso istante
- Consideriamo la possibilità di poter eseguire un programma anche se la sua immagine è caricata in memoria solo in parte; questo ha alcuni vantaggi:
  - Il processo potrebbe avere un'immagine più grande della memoria fisica disponibile
  - Se ogni processo ha bisogno di meno memoria fisica, si può aumentare il numero di processi in memoria (migliore utilizzo del processore senza aumento nei tempi di risposta o turnaround)
  - Più processi in memoria può significare meno I/O necessario per lo swapping

Memoria virtuale

- La memoria virtuale è la completa separazione tra memoria logica e memoria fisica di un programma
- Un processo può essere eseguito anche se solo una parte di esso è in memoria fisica
- Lo spazio di indirizzamento logico può essere molto più grande dello spazio di indirizzamento fisico
- Due possibili implementazioni:
  - Paginazione su richiesta
  - Segmentazione su richiesta (**non ne parleremo**)



Paginazione su richiesta

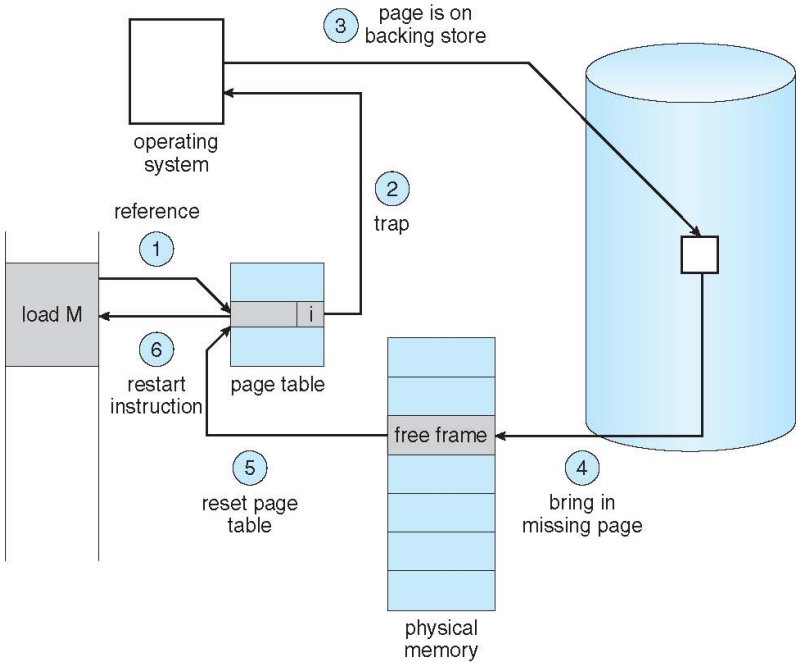
- La **paginazione su richiesta** (demand paging) è basata sull'hardware di paginazione
- Idea: non portare l'intero processo in memoria alla creazione, ma solo le pagine che vengono volta per volta usate
  - Una pagina viene caricata in memoria quando viene utilizzata durante l'esecuzione del programma
  - Il resto del processo risiede nella backing store
- Simile allo swapping con paginazione, ma questo individua le pagine da scaricare/caricare in maniera predittiva, mentre la paginazione su richiesta lo fa in base all'uso

Supporto hardware alla paginazione su richiesta

- L'hardware di paginazione deve fornire un opportuno supporto alla paginazione su richiesta
- La tabella delle pagine deve possedere il bit di validazione per indicare se una certa pagina è non valida oppure assente dalla memoria
- Una volta che il programma tenta un accesso ad una pagina non valida la MMU genera un'interruzione di **page fault**
- Il sistema operativo a questo punto gestisce il page fault per portare la pagina in memoria
- L'hardware inoltre deve permettere la riesecuzione dell'istruzione interrotta da un page fault in maniera trasparente al programma
- Infine occorre un'area opportuna (**swap space**) nella backing store

Gestione di un page fault

- Il sistema operativo decide se la pagina che ha generato il fault è invalida o non in memoria
  - Nel primo caso: abort del processo
  - Nel secondo caso: caricamento pagina da backing store
- Il sistema operativo trova un frame libero
- Quindi schedula l'operazione di caricamento dalla backing store
- Possibile cambio di contesto nell'attesa
- Al ripristino, aggiorna la tabella delle pagine con il frame caricato (e setta il bit di validità)
- Quindi ritorna dall'interruzione di page fault, e il processore riesegue l'istruzione che era stata interrotta

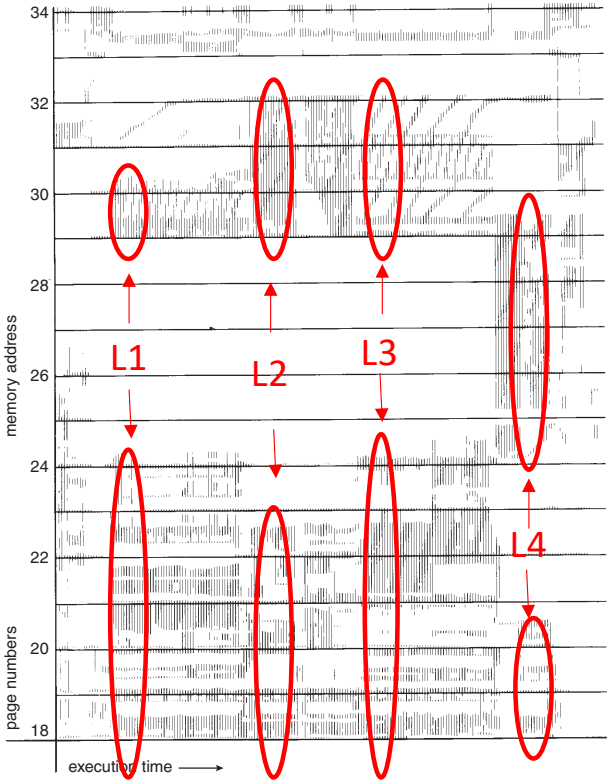


Commenti

- Se alla creazione di un processo nessuna pagina è in memoria e le pagine vengono caricate solo quando servono si parla di paginazione su richiesta **pura**
- La riesecuzione dell'istruzione interrotta da un page fault può essere difficile quando l'istruzione modifica più parole di memoria attraverso più pagine (in particolare nei processori CISC)
- La paginazione su richiesta è efficiente se i page fault sono pochi rispetto alle istruzioni eseguite, cosa di regola garantita dal fenomeno della **località dei riferimenti**

Il modello di località

- Il **modello di località** dell'esecuzione dei processi assicura l'efficienza della paginazione su richiesta
- In un certo momento un processo opera su una certa **località**, ossia su un certo sottoinsieme di pagine
- I processi nel tempo «migrano» da una località all'altra
- Località diverse possono sovrapporsi
- Se ogni processo riesce a tenere in memoria, in ogni istante, la sua località, i page fault sono pochi, e sono localizzati agli istanti in cui il processo migra di località



Gestione dei frame liberi

- Gestiti attraverso una lista dei frame liberi mantenuta dall'OS
- I frame vanno azzerati (zero-fill) prima di essere assegnati ad un processo per evitare leak di informazioni
- Man mano che i processi richiedono frame la lista si riduce: se si azzera, o scende sotto una certa dimensione minima, occorre ripopolarla (ne parleremo più avanti)

Prestazioni della paginazione su richiesta

- Se definiamo le seguenti costanti:
  - *p* = probabilità di un page fault (% page fault per istruzioni eseguite)
  - *ma* = tempo medio di accesso alla memoria
  - *pf* = tempo medio di gestione del page fault
- Allora il tempo medio di accesso effettivo è:

$$EAT = (1 - p)\, ma + p\, pf = ma + p\, (pf - ma) \approx ma + p\, pf$$

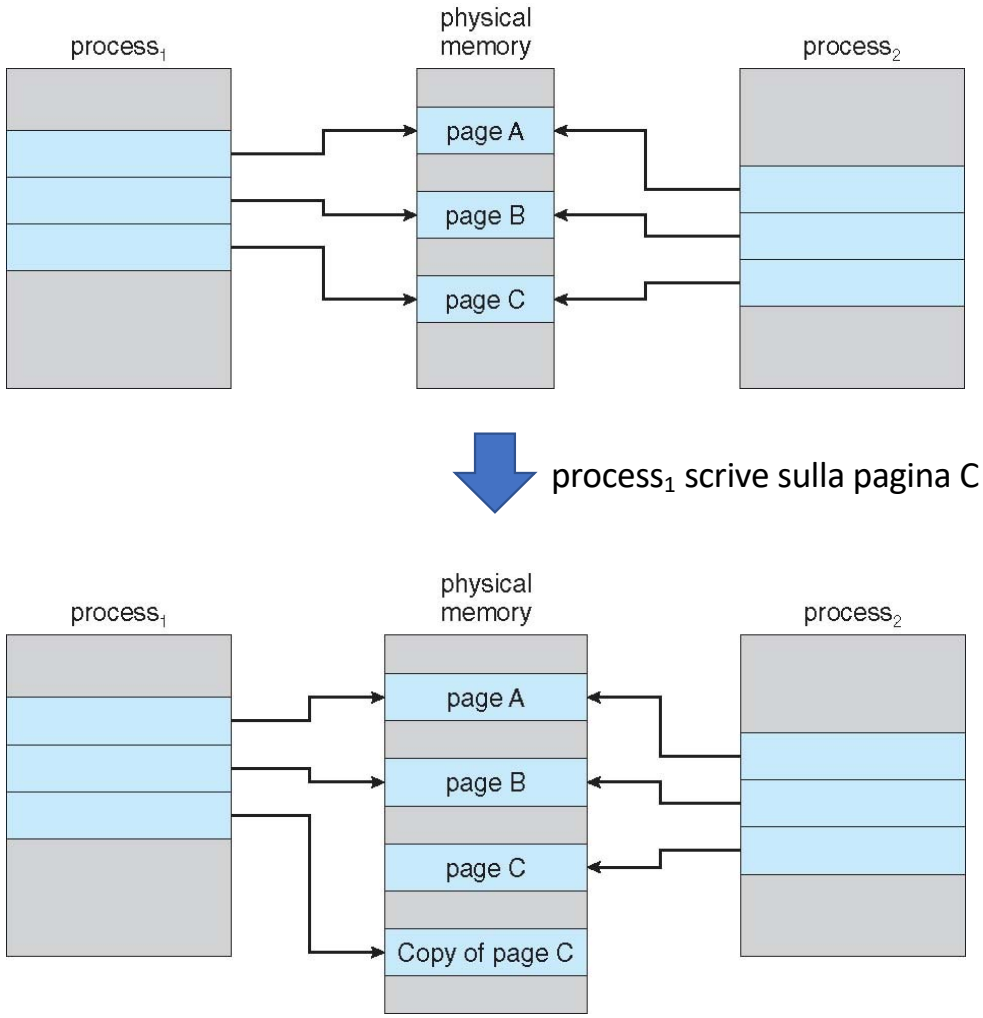
- Notare che *pf* è composto da servizio eccezione + caricamento pagina da backing store + ripristino processo, ma è dominato dal tempo di caricamento pagina da backing store
- Ad esempio, per *pf* = 8 msec e *ma* = 200 nsec, se vogliamo un rallentamento massimo del 10% occorre che i page fault siano meno di uno ogni 400.000 accessi a memoria

Ottimizzazioni della paginazione su richiesta

- L'I/O relativo allo swap space è di norma più rapido che non l'I/O relativo al file system:
  - Ottimizzazione: copiare il codice nell'area di swap all'avvio del processo e da lì usare il demand paging
  - Altra possibilità: copiare il codice dal file system all'avvio, ma solo le pagine necessarie, e fare page out dei frame contenenti codice nell'area di swap
- Alcuni sistemi cercano di limitare l'uso dell'area di swap:
  - Ottimizzazione opposta alla precedente: scartare i frame (read-only) contenenti codice e rileggerli dal file system invece di farne page out
  - In tal caso l'area di swap si usa solo per le pagine relative a stack, heap e variabili globali (la cosiddetta **memoria anonima**)
  - Tecnica usata in Linux e BSD
- Nei sistemi mobili vengono riprese dai processi solo le pagine read-only, non la memoria anonima (nessun page out, page in solo da file system)

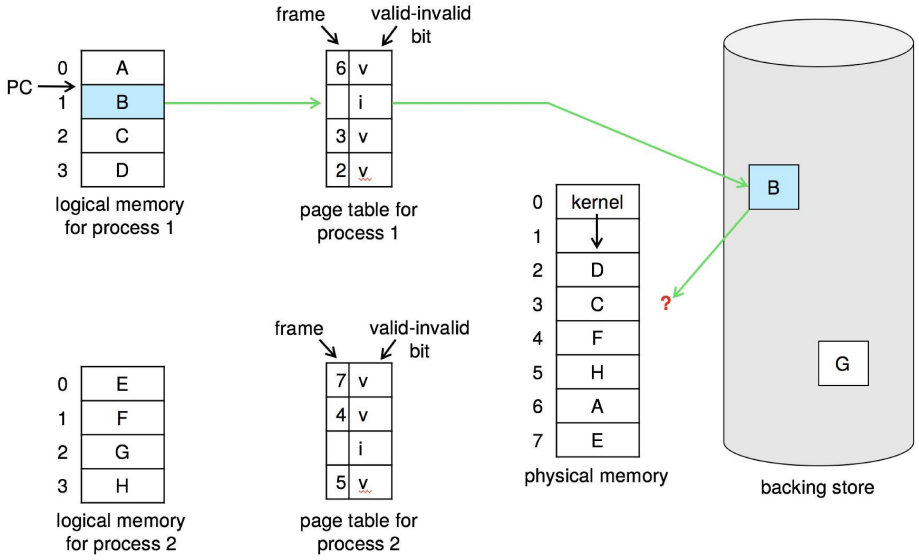
Copy-on-write

- **Copy-on-write** (**COW**) permette ad un processo figlio di condividere tutte (non solo quelle read-only) le pagine con il processo padre
- Se uno dei due processi modifica una pagina condivisa, la pagina viene copiata
- Utile per la `fork()`



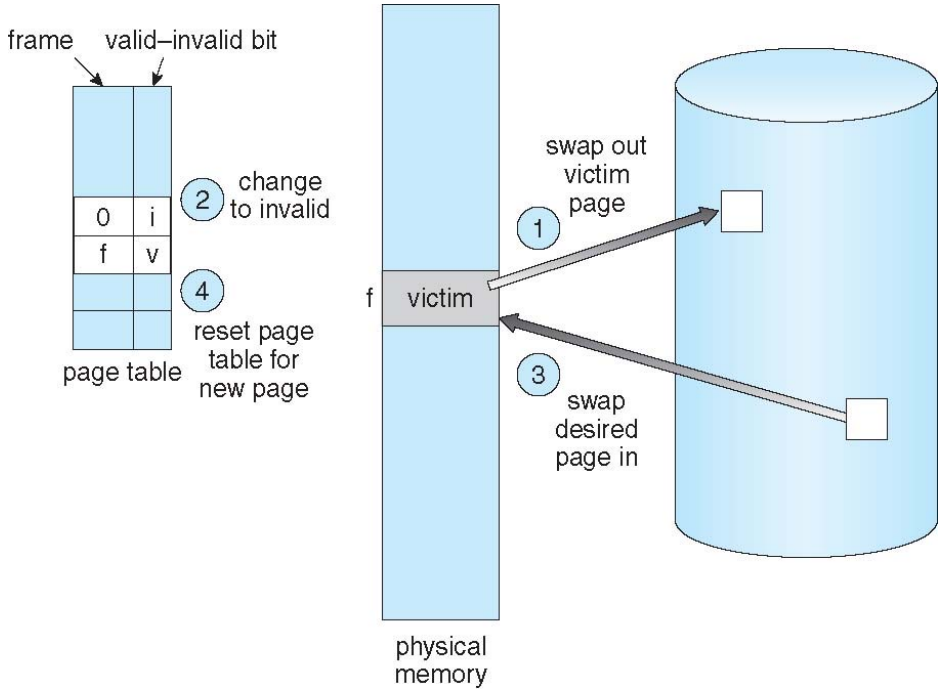
Sostituzione delle pagine (1)

- Cosa fare quando vengono esauriti i frame disponibili (**sovrallocazione** della memoria)?
  - Terminare il processo? Ma così perderemmo la trasparenza della memoria virtuale!
  - Fare swap out di un intero processo? Soluzione computazionalmente «pesante»
- Soluzione adottata: **sostituzione di pagina**, ossia liberare un frame attualmente inutilizzato



Sostituzione delle pagine (2)

- Routine di page fault aggiornata:
  - Si individua su disco la locazione della pagina richiesta
  - Si cerca un frame libero
  - Se esiste viene selezionato
  - Altrimenti:
    - Si usa l'algoritmo di sostituzione delle pagine per selezionare il frame vittima
    - Si effettua page out del contenuto del frame vittima
    - Si aggiornano le tabelle delle pagine e dei frame
  - Si effettua page in della pagina richiesta nel frame libero
  - Si aggiornano le tabelle delle pagine e dei frame



Sostituzione delle pagine: ottimizzazioni

- Una sostituzione di pagina richiede due trasferimenti di pagine dalla backing store
- Per tale motivo si desidera ridurre il più possibile le sostituzioni
- In primo luogo, non c'è bisogno di fare page out delle pagine read-only
- In secondo luogo, le entry della tabella delle pagine hanno di solito un **bit di modifica**, che l'hardware imposta a 1 se una certa pagina viene modificata dopo il suo caricamento: le pagine con il bit di modifica a 0 non necessitano di page out

Algoritmi di allocazione frame e sostituzione pagine

- Sono due politiche importantissime perché la memoria virtuale funzioni in maniera adeguata
- L'**algoritmo di allocazione frame** determina quanti frame assegnare ad ogni processo
- L'**algoritmo di sostituzione pagine** determina qual è il frame vittima quando è necessaria una sostituzione di pagina

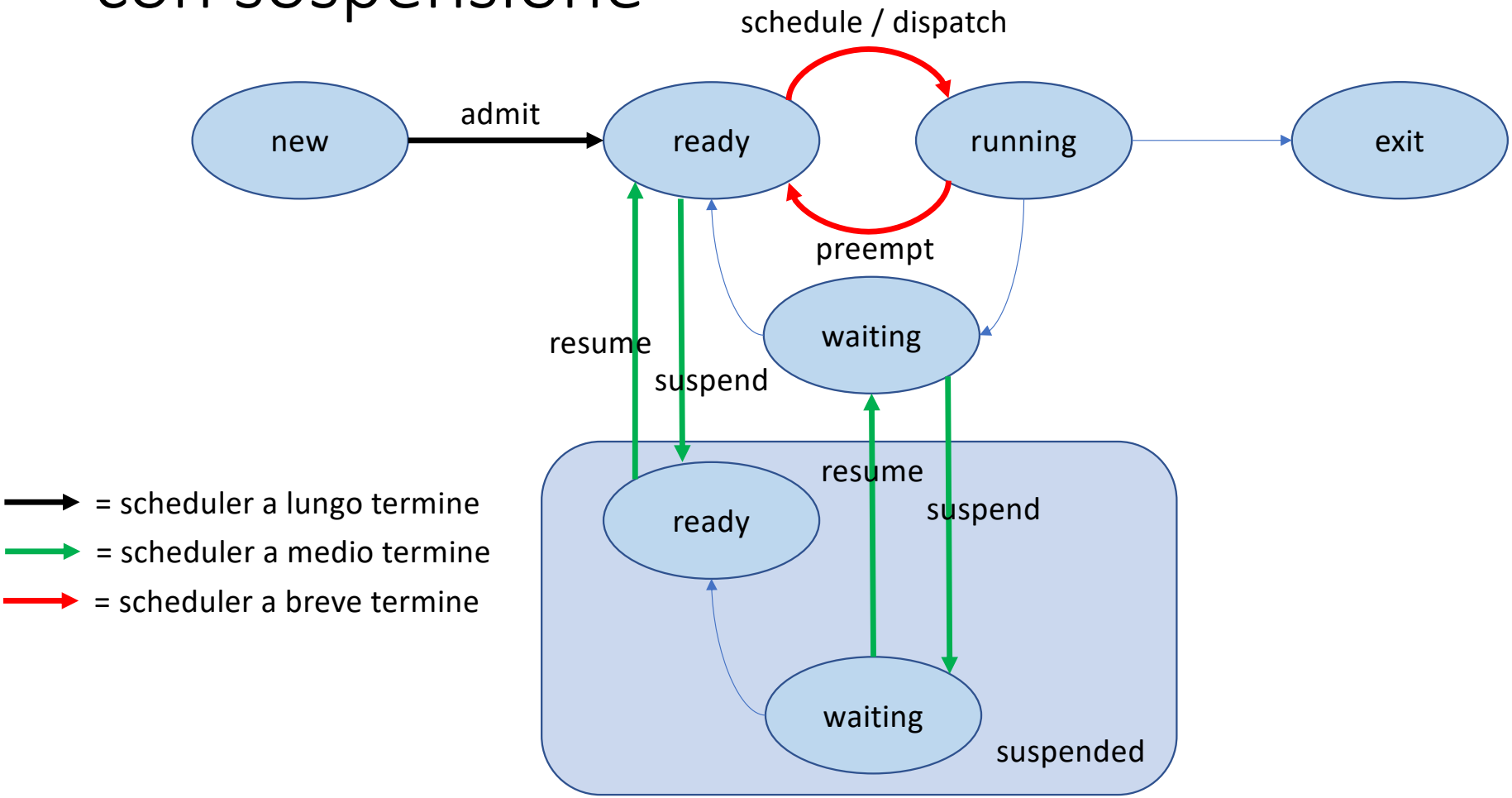
Algoritmi di allocazione dei frame

- Ogni processo ha bisogno di un numero minimo di frame, determinato dall'architettura, per avere prestazioni accettabili
- Ad esempio, nell'architettura IBM 370 servono 6 frame per poter eseguire l'istruzione macchina SS MOVE:
  - L'istruzione è lunga 6 byte, e quindi può esistere a cavallo di due frame
  - Ha due operandi con accesso a doppia indirezione alla memoria, quindi nel caso pessimo servono due frame per operando
- Il numero massimo di frame è limitato solo dalla quantità di memoria fisica
- Tra questi due estremi c'è un ampio ventaglio di scelta

Grado di multiprogrammazione

- Definiamo come **grado di multiprogrammazione** il numero massimo di processi che lo scheduler di breve termine può mandare in esecuzione (ready + running + waiting)
- Il grado di multiprogrammazione è un fattore legato all'utilizzazione del processore (se è basso anche l'utilizzazione è bassa)
- Il grado di multiprogrammazione però determina quanta memoria fisica usano i processi
  - In assenza di memoria virtuale, quando un processo può essere mandato in esecuzione solo se la sua immagine è interamente in memoria, determina il numero di immagini in memoria
  - In presenza di memoria virtuale determina il numero di «immagini parziali» (idealmente dovrebbero contenere ciascuna una località completa)
- Uno **scheduler a lungo termine** stabilisce il grado di multiprogrammazione sul lungo periodo decidendo se ammettere un nuovo processo nella ready queue
- Uno **scheduler a medio termine** può modulare il grado di multiprogrammazione sospendendo uno o più processi per ridurre il numero di processi nella ready queue
- Quando un processo è sospeso può essere soggetto a swapping

Diagramma di transizione di stato dei processi con sospensione



Allocazione uniforme

- Supponiamo di avere *m* frame di memoria fisica ed *n* processi
- L'allocazione uniforme assegna ad ogni processo *m*/*n* frame
- Gli eventuali *m*%*n* frame non assegnati possono essere usati come buffer di frame liberi
- Vantaggio: semplicità; svantaggio: processi diversi potrebbero avere esigenze diverse di memoria
- Notare che, all'aumentare del grado di multiprogrammazione *n*, diminuisce il numero di frame assegnato ad ogni processo

Allocazione proporzionale

- Supponiamo, ancora, di avere *m* frame ed *n* processi
- Si supponga che il processo i-esimo abbia bisogno di una quantità di memoria virtuale s_i
- Allora al processo i-esimo viene assegnato un numero di frame:

$$a_i = s_i / \sum_{i=1}^{n} s_i \cdot m$$

- All'aumentare del grado di multiprogrammazione, anche in questo caso diminuisce il numero di frame assegnato ad ogni processo

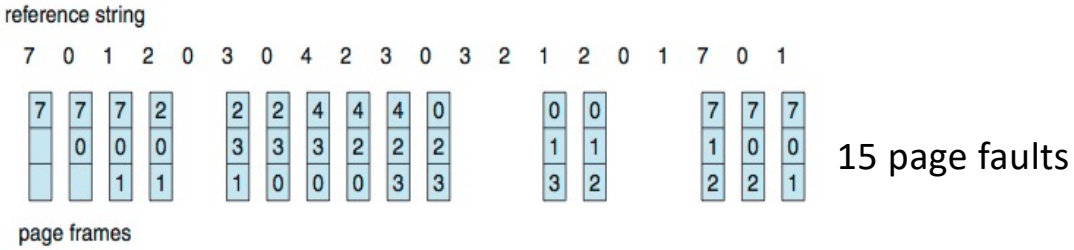
Algoritmi di sostituzione pagine

- Probabilmente ogni sistema operativo ha il suo
- Per confrontarli si utilizza come metrica il numero di page fault
- La valutazione si effettua data una stringa di accessi in memoria:
  - Di essi conta solo il numero di pagina
  - Più accessi in sequenza alla stessa pagina non generano page fault
  - I risultati dipendono fortemente dal numero di frame disponibili
- Considereremo come esempio una memoria con tre frame e la seguente stringa di accessi in memoria:

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

Algoritmo in ordine di arrivo

- L'algoritmo **in ordine di arrivo** (**FIFO**) seleziona come pagina vittima quella in memoria da più tempo
- Implementabile con una coda FIFO
- Esempio:

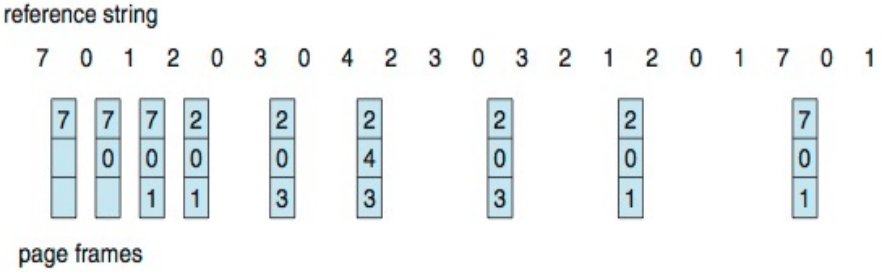


15 page faults

- Principale problema: **anomalia di Belady**, ossia può capitare che aggiungendo frame il numero di page fault aumenti!

Algoritmo ottimale

- L'algoritmo **ottimale** (**OPT**) seleziona come vittima la pagina che non verrà usata per il periodo di tempo più lungo
- Esempio:

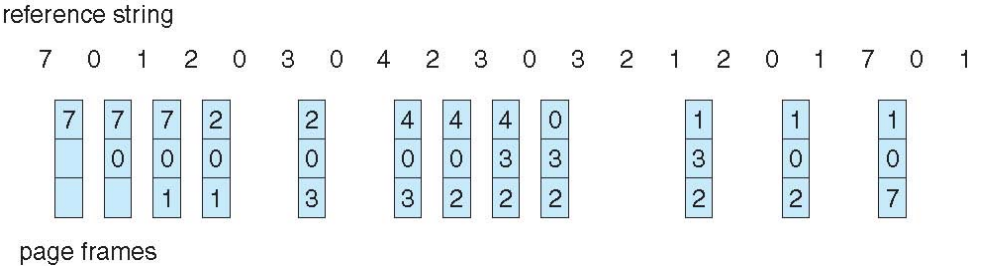


9 page fault

- Vantaggi:
  - Non soffre dell'anomalia di Belady
  - È teoricamente il migliore (minimizza il numero di page fault)
- Svantaggi:
  - Richiede la capacità di prevedere gli accessi futuri alla memoria
  - (per tale motivo si usa solo come confronto nei benchmark)

Algoritmo last recently used

- L'algoritmo **last recently used** (**LRU**) seleziona la pagina che non è stata usata per il periodo di tempo più lungo
- L'idea è che il passato può essere usato come approssimazione del futuro
- Esempio:



12 page faults

- Vantaggi:
  - Performance buona
  - Non soffre dell'anomalia di Belady
- Svantaggi:
  - Occorre tracciare qual è la pagina usata meno di recente (difficile!)

Implementazione algoritmo LRU

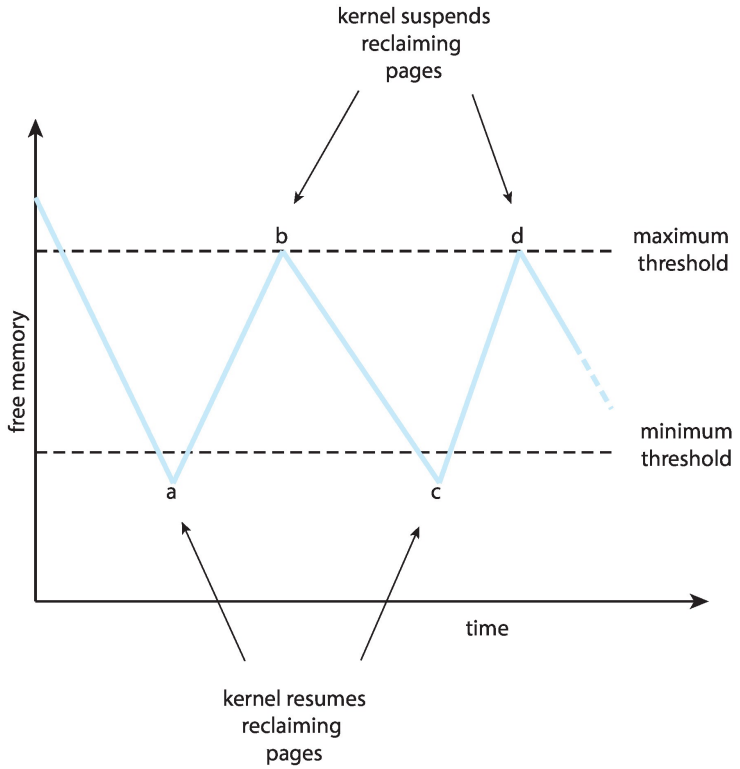
- L'algoritmo LRU richiede supporto hardware
- Due possibili implementazioni:
  - **Contatori**: un contatore viene incrementato ad ogni riferimento alla memoria, e l'entry della pagina corrispondente viene aggiornata con il valore del contatore
  - **Stack**: viene mantenuta una doubly linked list dei numeri delle pagine, e ad ogni riferimento alla memoria viene messa la pagina corrispondente in cima allo stack
- La soluzione con contatori richiede una ricerca nella tabella delle pagine per l'entry con il contatore più basso
- La soluzione con stack, di contro, richiede di aggiornare la doubly linked list ad ogni accesso in memoria

Sostituzione globale e locale

- **Sostituzione globale**: il frame vittima è selezionato tra tutti i frame di tutti i processi
  - Svantaggio: le pagine in memoria di un processo dipendono dal comportamento di tutti i processi ⇒ il tempo di esecuzione di un processo è molto dipendente dal carico
  - Vantaggio: utilizzo efficiente della memoria
- **Sostituzione locale**: il frame vittima è selezionato tra i frame del processo che ha generato il page fault
  - Vantaggio: ogni processo ha la garanzia di avere sempre lo stesso numero di pagine ⇒ il tempo di esecuzione di un processo è meno dipendente dal carico
  - Svantaggio: possibile sottoutilizzo della memoria
- La sostituzione globale è il metodo più usato

Politica di recupero di pagine

- È una politica di sostituzione globale
- L'obiettivo è mantenere il numero di frame liberi all'interno di un certo intervallo
- Quando tale numero scende sotto una certa soglia minima viene avviata una routine del kernel (**reaper**) che recupera pagine da tutti i processi
- Quando il numero raggiunge una soglia massima la routine reaper viene sospesa



Thrashing

- Quando uno o più processi hanno un numero troppo basso di frame in memoria diventa elevata la probabilità di un page fault
- Ma se non ci sono frame liberi e tutte le pagine sono attive, verrà sostituita una pagina che servirà di nuovo subito dopo
- Quindi è elevata la probabilità di un successivo page fault e di un'altra sostituzione di una pagina che sarà subito necessaria, e così via
- Quando l'attività di sostituzione di pagina occupa più tempo dell'esecuzione dei programmi si è in una situazione detta **thrashing**

Modello di località e thrashing

- Abbiamo detto che, secondo il modello di località, un processo in ogni istante utilizza un sottoinsieme delle sue pagine di memoria (località)
- Inoltre abbiamo detto che i page fault sono pochi quando tutte le località di tutti i processi in memoria sono completamente in memoria
- Perché avviene il thrashing? Perché ad un certo punto:

$$\sum_i locality_i > memory$$

- Quando questo si verifica, ci sono dei processi in memoria che non hanno la rispettiva località completamente in memoria

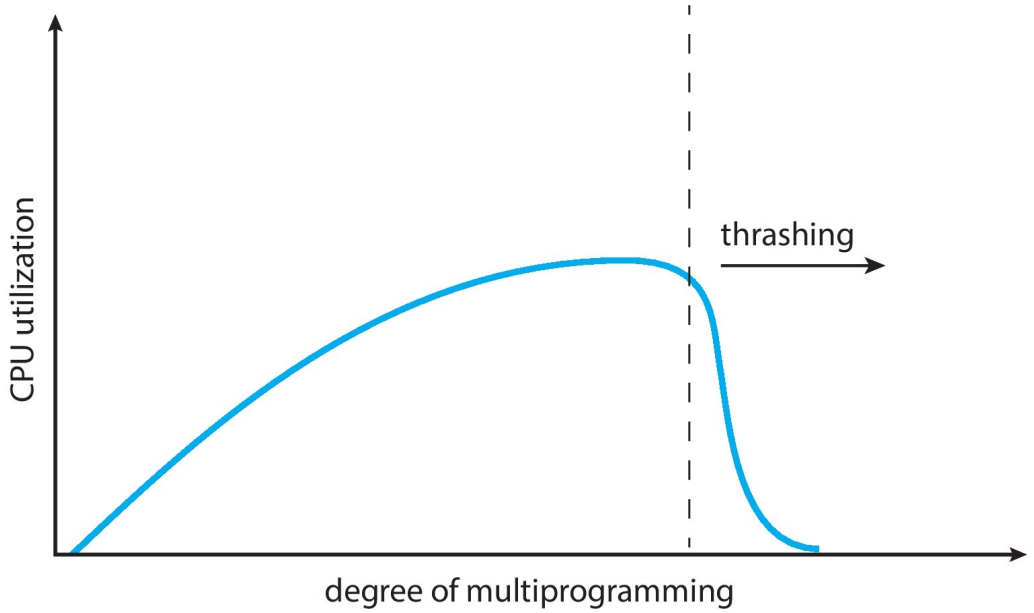
Anatomia del thrashing (1)

- Supponiamo che in un certo momento l'utilizzo della CPU sia basso: lo scheduler a medio termine può allora cercare di aumentare l'utilizzo della CPU aumentando il grado di multiprogrammazione, ossia il numero di processi ready
- In questo modo però diminuiscono i frame a disposizione per ciascun processo, ed alcuni processi potrebbero avere memoria fisica insufficiente a contenere la propria località, iniziando a generare page fault
- Se c'è scarsità di frame liberi, e la politica di sostituzione è globale, questi processi sottraggono frame ad altri processi
- Anche questi potrebbero finire per non avere sufficiente memoria fisica per contenere la propria località, e quindi iniziare a generare a loro volta page fault, finendo per sottraendo frame ai primi processi

Anatomia del thrashing (2)

- In tal modo si arriva ad una situazione in cui i processi ready si sottraggono in continuazione frame a vicenda, generando molti page fault in cascata
- I processi che generano page fault entrano in stato di attesa per il dispositivo di paginazione, diminuisce il numero di processi ready, e quindi l'utilizzo della CPU
- Lo scheduler a medio termine potrebbe cercare allora di aumentare l'utilizzo della CPU aumentando il grado di multiprogrammazione
- Ma questo riduce ulteriormente i frame disponibili per processo, generando un feedback positivo che fa precipitare rapidamente la situazione fino al punto in cui l'utilizzo della CPU crolla a zero

Thrashing e grado di multiprogrammazione

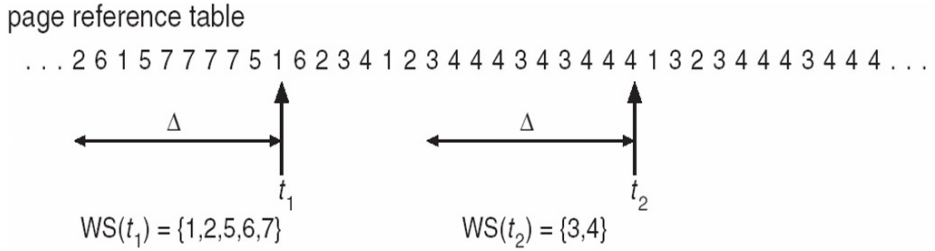


Rimediare al thrashing

- Come si può rimediare al thrashing:
  - Individuare quando tale situazione si verifica, e quindi
  - Ridurre il grado di multiprogrammazione
- Il secondo punto nel dettaglio:
  - Far scegliere allo scheduler a medio termine uno o più processi vittima
  - Far passare i processi vittima in stato sospeso
  - Fare swapping dei processi sospesi così da liberare i loro frame per gli altri processi
  - Lo swapping può essere standard (swap out completo) oppure con paginazione (graduale page out fino alla risoluzione del thrashing)
- Vediamo due tecniche per individuare il thrashing:
  - Modello del working set (globale)
  - Frequenza dei page fault (locale)

Modello del working set (1)

- Tecnica per calcolare la località di un processo in maniera approssimata
- Si definisce a priori un parametro Δ, la **finestra del working set**, come un numero fisso di riferimenti alla memoria
- Il working set in un certo istante è l'insieme dei riferimenti alla memoria che vengono effettuati nella finestra:
  - Se Δ è troppo piccola, il working set non include l'intera località
  - Se è troppo grande, sovrappone più località
  - Per Δ → ∞ il working set include tutte le pagine usate dal processo



Modello del working set (2)

- Se calcoliamo per ogni processo *i* la dimensione del working set WSS_i otteniamo che la domanda totale di frame *D* è:

$$D = \sum_i WSS_i$$

- Se $D > m$ vi è thrashing (dove *m* è il numero totale di frame liberi)

Calcolare il working set

- Si può calcolare il working set utilizzando timer + bit di riferimento
  - Ogni entry nella tabella delle pagine ha un **bit di riferimento**, inizialmente a 0
  - Quando una pagina viene riferita da un'istruzione, l'hardware pone il suo bit di riferimento a 1
  - Richiede pertanto hardware speciale a supporto
- Si memorizzano gli ultimi k bit di riferimento in memoria (con scorrimento)
  - Se la finestra è Δ imposto un timer ogni Δ/k
  - Alla scadenza faccio scorrere la parola di k bit, copio il bit di riferimento in ultima posizione e azzero il bit di riferimento
- Se in un certo istante almeno uno dei k bit è impostato ad 1, la pagina fa parte del working set in quell'istante
- Il calcolo è più preciso (e più oneroso) più alto è il valore di k

Frequenza dei page fault

- Un metodo alternativo al modello del working set (e più diretto) è calcolare la frequenza dei page fault dei processi
- Si stabilisce un range accettabile nella frequenza dei page fault (**page fault frequency**, PFF) e si usa una politica di sostituzione locale:
  - Se PFF < soglia minima, il processo perde frame
  - Se PFF > soglia massima, il processo acquista frame
- Se non ci sono abbastanza frame liberi per abbassare la PFF di un processo, occorre sospendere un altro processo per liberare frame