

Virtual Memory

Overview

La CPU genera il logical address (32b, 64b etc.. a seconda dell'architettura), dobbiamo mappare tale indirizzo con l'indirizzo fisico.

Questo è ciò che fa la MMU.

L'indirizzo fisico fa riferimento ad una zona della memoria, che contiene un'istruzione o un dato.

Ad esempio, se l'indirizzo è a 32bit, dunque l'indirizzo è grande 4 Byte.

Dentro la zona di memoria indirizzata dall'indirizzo fisico, troviamo qualcosa che può avere una certa dimensione, solitamente 1 Byte (dunque 1 Byte di dati o istruzione).

Tutto ciò che abbiamo studiato finora è per sfruttare la memoria che abbiamo e mettere più processi possibili in esecuzione.

Objectives

- Memory Management techniques have the goal to keep many processes in memory simultaneously.
 - Allowing multiprogramming
- However, they tend to require that an entire process is in memory before execution.
- Virtual Memory allows the execution of processes that are not completely in memory.
- Major advantages:
 - The program can be larger than physical memory.
 - Allowing processes to share files and libraries, and to implement shared memory.

Background

- The instructions being executed must be in physical memory.
 - It limits the size of the program to the size of physical memory.
- In many cases, the entire program is not needed.
 - Programs often have code to handle unusual error conditions which happens rarely. Therefore, this code is never executed.
 - Array, lists and tables often are allocated more memory than they actually required.
 - Certain options and features of a program may not be needed at the same time.

Finora il logical space di cui abbiamo parlato era limitato dalla dimensione della physical memory: cioè se ho 1GB di memoria fisica, non possiamo avere una logical memory più grande di 1GB.

Virtual Memory

- **Virtual memory** – separation of user logical memory from physical memory

- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes

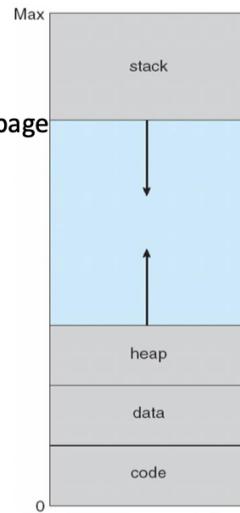
Concetto base: vorrei un modo che separi logical memory e physical memory. Se un processo ha bisogno di 100MB di spazio, a noi non serve 100MB di spazio in memoria fisica, ma abbiamo qualche pagina del processo che **non è mappata** in logical memory.

Quando un processo è in esecuzione, non tutte le sue pagine sono in memoria.

Se ragioniamo così, disaccoppiando logical memory e physical memory, possiamo immaginare di avere una logical memory molto più grande della physical memory.

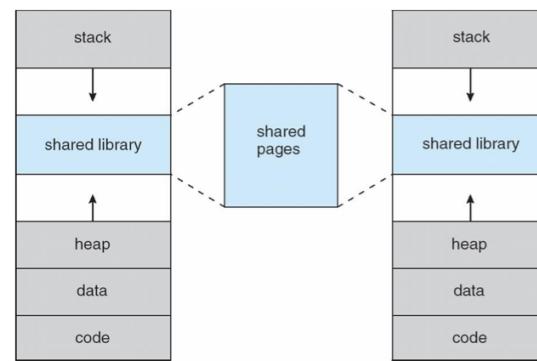
Virtual Address Space

- Usually design logical address space for the stack to start at Max logical address and grow “down” while the heap grows “up”
 - Maximizes address space use
 - Unused address space between the two is **hole**
 - No physical memory is needed until heap or stack grows to a given new page



Virtual Address Space

- Virtual Memory allows files and memory to be shared by two or more processes through page sharing:
 - System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space.



Virtual Address Space

- Virtual Memory allows files and memory to be shared by two or more processes through page sharing:
 - System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space.
 - Libraries are mapped read-only into the space of each process that is linked with that.
 - Processes can share memory. Virtual memory allows one process to create a region of memory that it can share with another process.
 - Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared.
 - System libraries shared via mapping into virtual address space

Implementation of Virtual Memory

- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation

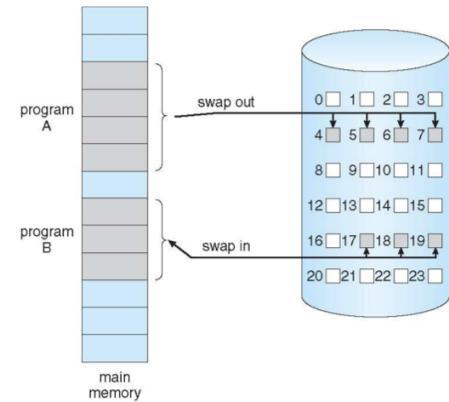
Dunque se un processo ha 10 pagine, all'inizio non servono tutte e 10 le pagine, magari ne servono solo 2, quindi carico quelle e mano a mano che il processo esegue, se serve porta le altre in memoria.

Noi studieremo la tecnica di **Demand paging**.

Demand paging

Demand Paging

- First option for executing a program is bringing all to the memory.
 - Initially, we may not need all the program in the memory.
- Loading pages only as they are demanded known as **Demand Paging**.
 - Pages that are never accessed are thus never loaded into physical memory.
 - A demand paging system is similar to a paging system with swapping where processes reside in secondary memory.

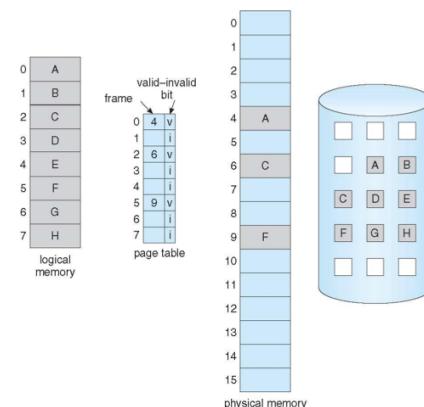


Carichiamo una pagina del processo solo quando è richiesta.

Con il Demand Paging limitiamo anche le operazioni di I/O, perché il caricare o scaricare pagine da/verso il disco viene fatto solo quando necessario, invece di caricare tutte le pagine del processo direttamente, e in caso scaricarle di nuovo tutte in caso di swapping.

Demand Paging – Basic Concepts

- During process execution, some pages will be in memory, and some will be in secondary storage.
- How to distinguish them? Asking support from hardware through **valid-invalid bit**.
 - When the bit is set to **valid**, the associated bit is both legal or in memory.
 - If the bit is set to **invalid**, the page is not in the logical address space of the process or is currently in secondary storage.



Come fa il processo a sapere se le pagine di cui ha bisogno sono in memoria?

Il processo deve controllare se una certa pagina è già in memoria, e quindi eseguirla, oppure se non è in memoria deve caricarla dal disco.

Per capirlo, esiste il **valid-invalid** bit, che troviamo nella page table.

Se un processo ha bisogno di una certa pagina, si cerca il mapping page-frame nella page table, se il bit di validità è valido, allora vuol dire che la pagina è in memoria e quindi può essere eseguita.

Se non è valido, vuol dire che la pagina non è in memoria e quindi va recuperata dal disco.

Ogni processo ha la sua page table.

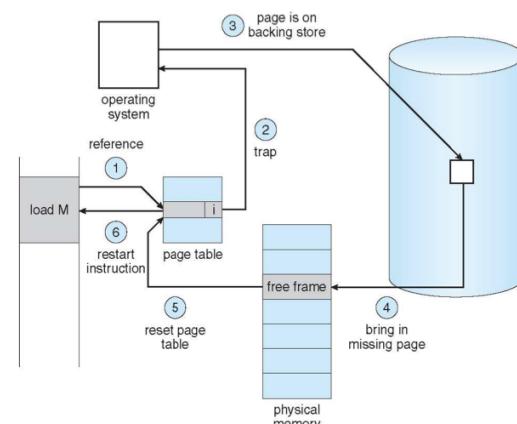
Immaginiamo che un processo abbia bisogno di 5 pagine.

Per indirizzare 5 pagine abbiamo bisogno di 3 bit. Con 3 bit però posso indirizzare 8 pagine, quindi solo 5 indirizzi logici vengono usati dal processo, gli altri 3 indirizzi logici sono nella page table ma

non fanno parte della logical memory del processo, e saranno quindi contrassegnati come **non validi**.

Demand Paging – Basic Concepts

- What happens if the process tries to access a page that was not brought into **memory**?
 - An access to a page marked invalid caused a **page fault**.
- How is the page fault handled?
 1. Reference
 2. Trap
 3. Page is on backing store
 4. Bring in missing page
 5. Reset page table
 6. Restart instruction



E se la pagina non si trova in memoria?

Si ha un

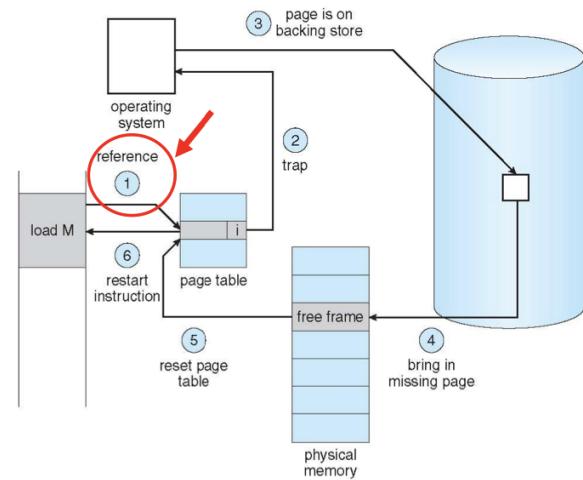
page fault, per cui la pagina non è in memoria, ma è sul disco e deve essere recuperata.

I passi che il SO segue quando si verifica un page fault sono 6:

Demand Paging – Basic Concepts

- How is the page fault handled?

1. We check an internal table for this process to determine whether the reference was a valid or an invalid memory.



Andiamo nella page table, controlliamo il validity bit per capire se la pagina in memoria è valida oppure no.

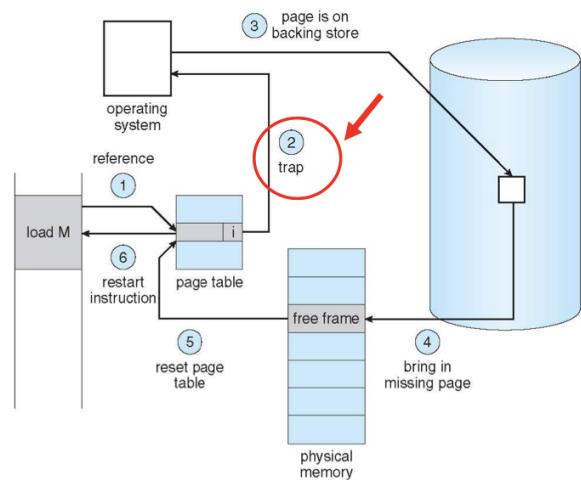
Se è valido, eseguiamo la pagina, altrimenti se non è valido e non fa parte della memoria logica del processo, esso sta compiendo un'azione illegale e quindi viene terminato.

Se non è valido, ma fa parte della memoria logica, allora l'operazione è valida ma la pagina deve essere recuperata dal disco.

Demand Paging – Basic Concepts

- How is the page fault handled?

1. Reference
2. If the reference was invalid, we terminate the process, If it was valid but we have not yet brought in that page, we now page it in.

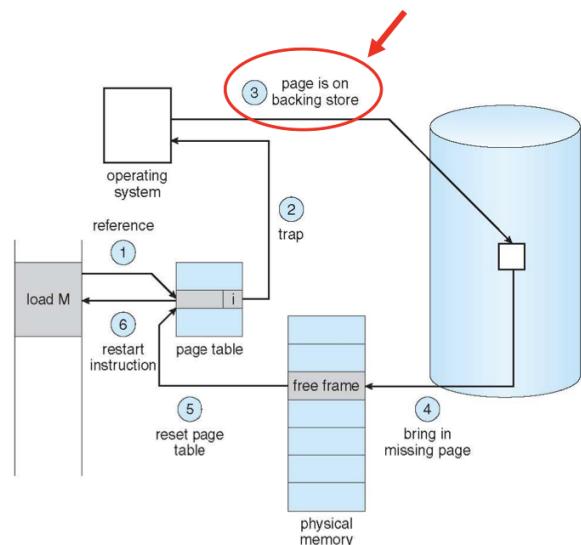


Dunque dobbiamo trovare un frame libero per allocare la pagina da caricare dal disco.

Demand Paging – Basic Concepts

- How is the page fault handled?

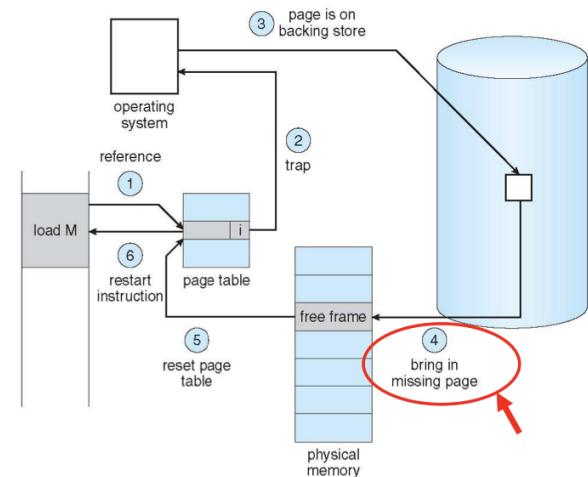
1. Reference
2. Trap
3. We find a free frame



Demand Paging – Basic Concepts

- How is the page fault handled?

1. Reference
2. Trap
3. Page is on backing store
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.

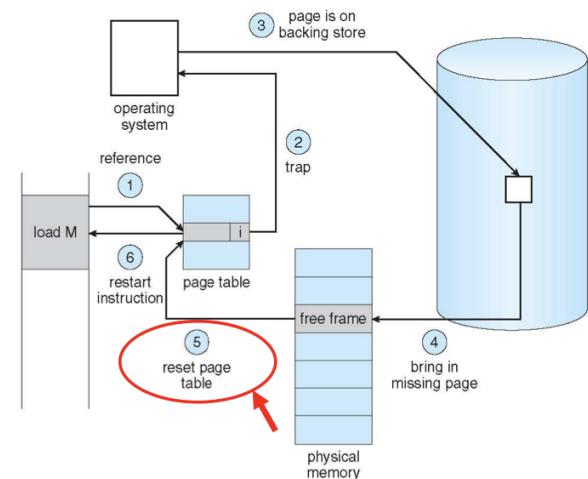


Una volta trovato il frame libero, dobbiamo fare una operazione di I/O per scrivere la pagina sul frame libero.

Demand Paging – Basic Concepts

- How is the page fault handled?

1. Reference
2. Trap
3. Page is on backing store
4. bring in missing page
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

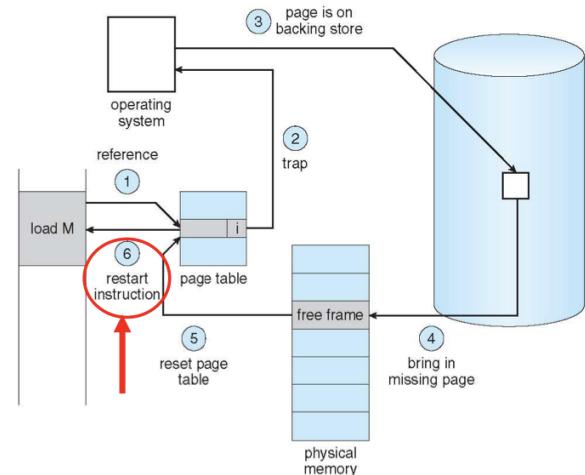


Una volta fatto ciò possiamo segnare l'entry nella Page Table mettendo il bit di validità a *valido*, e scriviamo il frame number a cui si trova la pagina richiesta.

Demand Paging – Basic Concepts

- How is the page fault handled?

1. Reference
2. Trap
3. Page is on backing store
4. bring in missing page
5. reset page table
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.



Il processo può ora riprendere da dove aveva lasciato e continuare la sua esecuzione.

Summary -1

- **Virtual memory** – separation of user logical memory from physical memory
- Virtual memory can be implemented via **Demand paging**
 - Loading pages only as they are demanded.
- During process execution, some pages will be in memory, and some will be in secondary storage, distinguishing them using **Valid-Invalid** bit.
- An access to a page marked invalid caused a **page fault**.

Aspects of Demand Paging

- Extreme case – start executing a process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access
 - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
 - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
 - This situation would result in unacceptable system performance.
 - However, it is not likely, because programs tend to have **locality of reference** which result in reasonable performance from demand paging.

Nel procedimento appena descritto ci sono tante scelte che devono essere prese.

Es:

quando inizia l'esecuzione di un processo, quante pagine carichiamo in memoria?

Zero, e mano a mano carichiamo quella necessaria?

Oppure un po'? Se sì, quante?

Pure demand paging

Pure demand paging: inizio un processo caricando **zero** pagine in memoria.

Alla prima istruzione di cui abbiamo bisogno, avviene un page fault e quindi mano a mano carichiamo le pagine.

In questo caso il processo è molto lento, perché tutte le istruzioni e dati di cui può avere bisogno non sono in memoria, e devono essere portate dal disco quando vengono richieste.

Se devo fare una somma (istruzione somma), oltre all'istruzione devo prelevare anche i dati che devono essere coinvolti nella somma.

Quindi se questi dati si trovano in una pagina diversa da quella dell'istruzione, si possono verificare 2 page faults: uno per la pagina che contiene l'istruzione e uno per la pagina dei dati.

Più avanti studieremo il concetto di **locality of reference**: un processo usa il 20% delle sue pagine **molto di più** dell'80% restante delle pagine.

Questo perchè spesso gran parte del programma è fatto da funzioni di controllo o

di handlers d'eccezioni che non vengono eseguite spesso, ma solo in casi eccezionali.

Quando dobbiamo liberare un frame per fare spazio per altri processi o altre pagine, tutte queste informazioni vengono prese in considerazione per fare una scelta accurata.

Free-Frame List

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- Operating system typically allocates free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.
- As free frames are requested, the size of the free-frame list shrinks.

Come possiamo sapere quali frame sono liberi?

Nel SO c'è una struttura dati (una lista) che viene usata come *pool* per memorizzare i frames liberi.

Quando liberiamo un frame, scarichiamo sul disco la pagina che stiamo deallocando, e poi **azzeriamo tutto il contenuto** del frame che stiamo liberando, per garantire la privacy tra processi (tecnica **zero-fill-on-demand**).

Performance of Demand Paging

- Computing the **Effective Access Time** for a demand-paged memory.
 - Assuming the memory access-time (ma) is 10 ns.
 - As long as we have no page faults, the effective access time is equal to the memory access time.
 - If a page fault occurs, we must first read the relevant page from secondary storage and then access the desired word.
 - Considering P as the probability of a page fault ($0 \leq p \leq 1$)

$$\text{Effective Access time} = (1-p) * \text{ma} + p * \text{page fault time}$$

Demand Paging: buono perchè non portiamo tutte le pagine di un processo in memoria, lasciando molto spazio libero per altri processi.

Però è una tecnica con performances basse (lentezza).

Assumiamo un access-time di 10ns, se non abbiamo PF allora ottimo e facciamo 1 solo accesso alla memoria (quando leggiamo il dato/istruzione, senza considerare l'accesso per la page table).

Se abbiamo PF, prima dobbiamo leggere la pagina dal disco e caricarla su un frame libero (se c'è) in memoria.

Questa operazione richiede molto tempo, e tale tempo lo chiamiamo **page fault time**.

Quindi se

p è la probabilità di avere un PF, posso calcolare l'EAT con la formula mostrata in slide, dove $(1 - p)$ è la probabilità di **non** avere un PF.

Stages in Demand Paging – Worse Case

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference is legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
 1. Wait in a queue for this device until the read request is serviced
 2. Wait for the device seek and/or latency time
 3. Begin the transfer of the page to a free frame

Vediamo più nel dettaglio cosa succede quando si verifica un PF.

Scaturisce una trap che comunica al processore che il processo non può continuare ad eseguire.

Si salva lo stato del processo (registri e contesto).

Si determina se si è avuto effettivamente un PF o no.

Se sì, si controlla che l'indirizzo logico richiesto sia legale (all'interno del logical space) e se sì, si determina la posizione della pagina richiesta su disco.

Dunque si richiede una lettura dal disco su un frame libero in memoria principale.

Però può essere che il disco sia occupato già in un'altra operazione di I/O, quindi in caso bisogna attendere che il disco possa soddisfare la nostra richiesta.

Stages in Demand Paging – Worse Case

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Se il disco è occupato, possiamo dare CPU time ad altri processi e quando il disco è pronto a soddisfare la nostra richiesta, la CPU riceve un'interrupt.

Dunque si fa context switch del processo che stava eseguendo, aggiorniamo la page table del nostro processo per segnare che la pagina richiesta è ora in memoria, e si attende affinchè la CPU possa eseguire di nuovo il nostro processo (quando verrà rischedulato).

Quanto descritto è il

worst case di quello che può succedere in caso di page fault.

Performance of Demand Paging

- There are three major task components of the page-fault service time:
 - Service the interrupt – careful coding means just several hundred instructions needed
 - Read the page – lots of time
 - Restart the process – again just a small amount of time
 - Page Fault Rate $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)
 - $EAT = (1 - p) \times \text{memory access} + p (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

$$EAT = (1 - p) * 200 + p (8 \text{ milliseconds}) = (1 - p) * 200 + p * 8,000,000 = 200 + p * 7,999,800$$

- The effective access time is directly proportional to the **page-fault rate**.

Vogliamo portare meno pagine in memoria (+spazio libero in memoria) ma con il rischio di avere tanti PF, oppure avere più pagine in memoria (-spazio in memoria) e avere meno PF (+velocità).

Copy-on-Write

Copy-on-Write

- Process creation using the `fork()` system call may initially bypass the need for demand paging.
- Using a technique called **copy-on-write (COW)** that allows the parent and child processes to initially share the same pages.
 - If either process modifies a shared page, only then is the page copied.

Abbiamo detto che un processo può creare processi figli tramite **fork**.

Tutto ciò che succede

dopo la fork viene visto sia dal padre che dal figlio.

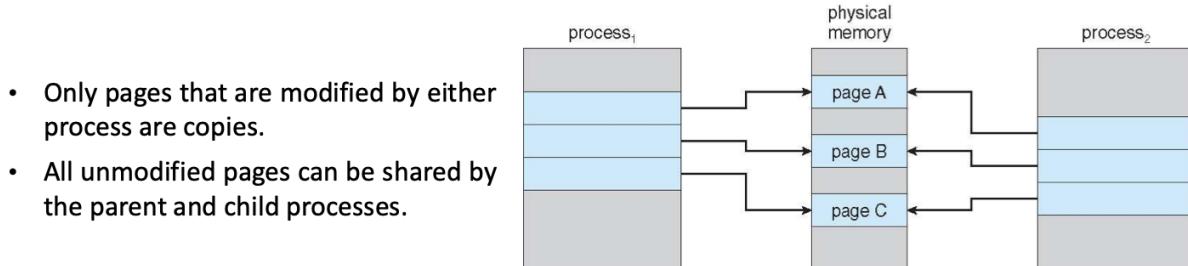
Il contesto del processo padre viene copiato e la copia viene allocata per l'uso del processo figlio.

Ma in realtà ciò che succede è che la porzione di memoria (eventualmente più pagine) dedicata al processo padre viene effettivamente copiata e riallocata in una nuova zona di memoria per il figlio solo se questo vuole modificare la zona di memoria.

Ma fino a quel momento il processo figlio ha accesso alla stessa pagina del padre, con cui la condivide.

Copy-on-Write

- Assume that the child process attempts to modify a page containing portion of the slack, with the pages set to be copy-on-write.
- The operating system will obtain a frame from the free-frame list and create a copy of this page, mapping it to the address space of the child process.
- The child will then modify its copied page and not the page belonging to the parent process.



Page replacement

Page Replacement

- If we increase the degree of multiprogramming, we are **over-allocating memory**.
- While a process is executing, a page fault occurs.
- The operating system determines where the desired page is residing on secondary storage.
- There are no free frames, all memory is in use.
- Operating system can do standard swapping and swap out a process.
 - Standard swapping leads to overhead of copying entire processes between memory and swap space.
- Most operating system now combine swapping pages with **Page Replacement**.

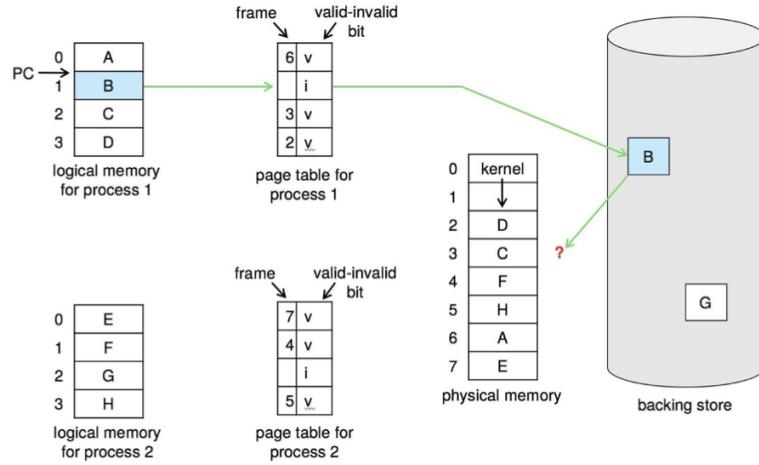
Se stiamo mettendo **troppi** processi in esecuzione in contemporanea, potremmo trovarci con la memoria piena, e tutti i processi incorreranno in PF.

Dunque la CPU verrà impegnata per la maggior parte del tempo a gestire PF per swappare pagine, invece di dedicare questo tempo a effettivamente eseguire i processi.

Con il demand paging possiamo però ora fare swapping "parziale" tra processi: cioè se dobbiamo fare spazio in memoria per un nuovo processo, magari non deallochiamo *tutte* le pagine di un processo, ma solo alcune, consentendo quindi al processo di continuare a eseguire e poi se necessario (dopo un PF) caricare altre pagine in memoria.

Page Replacement

- Most operating system now combine swapping pages with **Page Replacement**.



Tra le pagine caricate in memoria, bisogna scegliere quale "sacrificare" per fare spazio.

Vedremo poi come effettivamente fare questa scelta (*Random, Last Recently Used, Least Frequently Used* etc..).

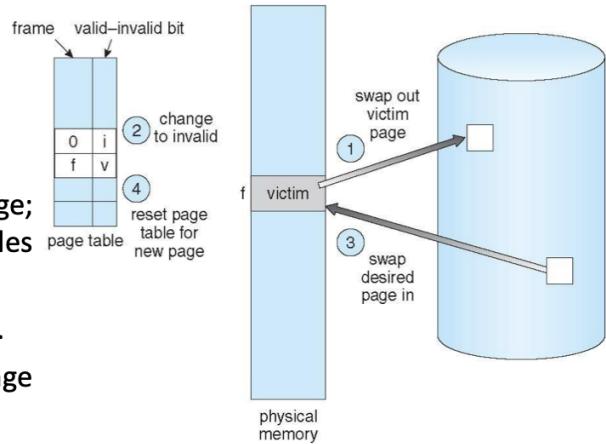
Basic Page Replacement

- Page replacement is: if no frame is free, we find one that is not currently being used and free it.
- Free frame by writing its content to swap space and changing the page table.
- Using the free frame to hold the page for which the process faulted.

Basic Page Replacement

- We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on secondary storage.
2. Find a free frame.
 - A. If the is a free frame, use it,
 - B. If not, select a **victim frame**.
3. Write the victim frame to secondary storage; change the page and frame tables accordingly.
4. Read the desired page into the freed frame.
5. Continue the process from where the page fault occurred.

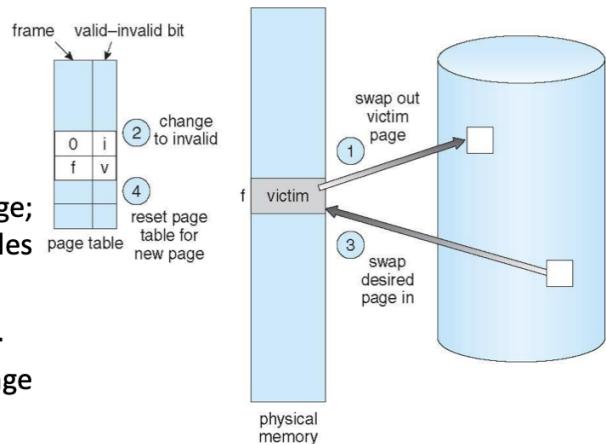


Summary -2

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
 - Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.
- Page replacement is: if no frame is free, we find one that is not currently being used and free it.
 - Free frame by writing its content to swap space and changing the page table.
 - Using the free frame to hold the page for which the process faulted.

Summary -2

- We modify the page-fault service routine to include page replacement:
 1. Find the location of the desired page on secondary storage.
 2. Find a free frame.
 - A. If the is a free frame, use it,
 - B. If not, select a **victim frame**.
 3. Write the victim frame to secondary storage; change the page and frame tables accordingly.
 4. Read the desired page into the freed frame.
 5. Continue the process from where the page fault occurred.



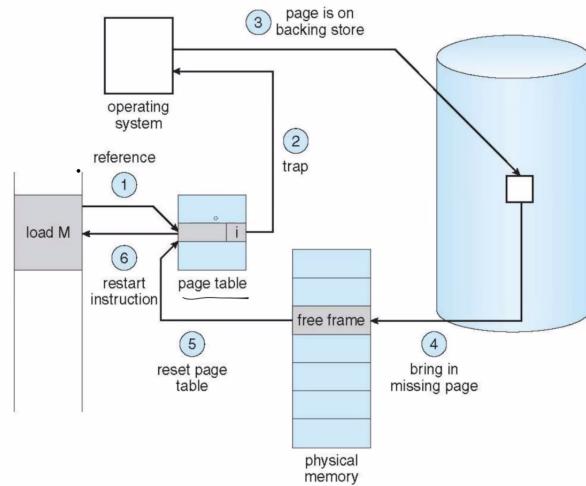
— LEZIONE 8/04/24

Summary -1

- **Virtual memory** – separation of user logical memory from physical memory
- Virtual memory can be implemented via **Demand paging**
 - Loading pages only as they are demanded.
- During process execution, some pages will be in memory, and some will be in secondary storage, distinguishing them using **Valid-Invalid** bit.
- An access to a page marked invalid caused a **page fault**.

Summary -1

- How is the page fault handled?
 - Reference
 - Trap
 - Page is on backing store
 - Bring in missing page
 - Reset page table
 - Restart instruction

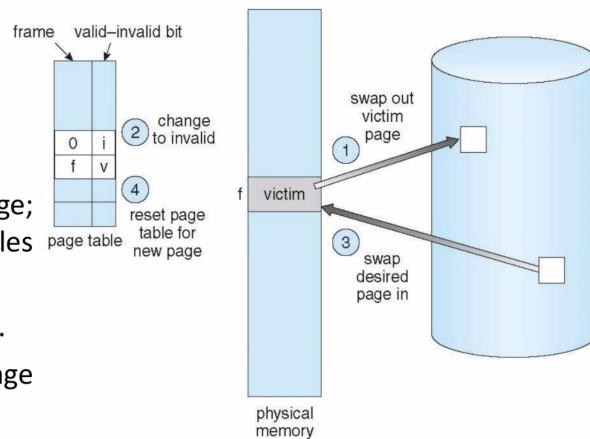


Summary -2

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
 - Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.
- Page replacement is: if no frame is free, we find one that is not currently being used and free it.
 - Free frame by writing its content to swap space and changing the page table.
 - Using the free frame to hold the page for which the process faulted.

Summary -2

- We modify the page-fault service routine to include page replacement:
1. Find the location of the desired page on secondary storage.
 2. Find a free frame,
 - A. If the is a free frame, use it,
 - B. If not, select a **victim frame**.
 3. Write the victim frame to secondary storage; change the page and frame tables accordingly.
 4. Read the desired page into the freed frame.
 5. Continue the process from where the page fault occurred.



Dirty bit

Page Replacement

- If no frames are free, two-page transfers are required.
 - This doubles the page-fault service time.
- Use **modify (dirty) bit** to reduce overhead of page transfers.
 - The modify bit for a page is set by the hardware whenever any bytes in the page is written into, indicating that the page has been modified.
 - If the bit is set, the page is modified since it was read, we must write the page to storage.
 - If the bit is not set, we do not need to write the page to the storage, it is already there.

Quando avviene un PF, sono necessari due trasferimenti di pages: uno che scarica la victim page su HDD, e uno che carica un'altra page in RAM.

Usiamo il **dirty bit** per ottimizzare il processo di page replacement.

Se una pagina viene solo letta, e non modificata, possiamo tenere il dirty bit a 0, anche se l'abbiamo letta.

La page, che può essere selezionata come

victim page per essere sostituita, non ha dunque bisogno di essere trasferita su HDD, perchè esiste già una copia **non modificata** della page in HDD.

Page and Frame Replacement Algorithms

- Two major problems to be solved for implementing demand paging:
 - **Frame-allocation algorithm**
 - If we have multiple process in memory, we must decide how many frames to allocate to each process.
 - **Page-replacement algorithm**
 - When page replacement is required, we must select the frames that are to be replaced.
 - Want lowest page-fault rate

Due problemi da risolvere nel contesto del Demand Paging:

- *Quanti frames dobbiamo avere disponibili per ogni processo, all'inizio della loro esecuzione?*
Ne riserviamo solo uno, come nel
Pure Demand Paging?
Oppure esistono altre strategie?
- *Se dobbiamo sostituire una page in RAM, quale scegliamo?*
Che algoritmo usiamo per scegliere la
victim page?

Page Replacement Algorithms

Page and Frame Replacement Algorithms

- How to select a particular replacement algorithm?
- Evaluate algorithm by running it on a particular string of memory references (**reference string**) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
 - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is
 - **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

Per rispondere alla seconda domanda, dovremmo stabilire un criterio che ci permetta di valutare le performance di un algoritmo piuttosto che di un altro.

Come valutiamo quindi le performances degli algoritmi usati per scegliere la victim page?

Calcoliamo il numero di PF che ogni algoritmo fa.

Vorremmo un numero di PF il più basso possibile → sistema più “veloce” (nel senso che dedica più tempo ad eseguire effettivamente i nostri processi, piuttosto che overhead per gestione della memoria).

Page Replacement Algorithms

- There are many different page-replacement algorithms:
 - FIFO Page Replacement
 - Optimal Page Replacement
 - LRU Page Replacement
 - LRU-approximation Page Replacement
 - Counting-based Page Replacement
 - Page-Buffering Algorithm
 - Applications and Page Replacement

FIFO

First-In-First-Out (FIFO) Algorithm

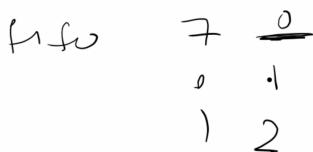
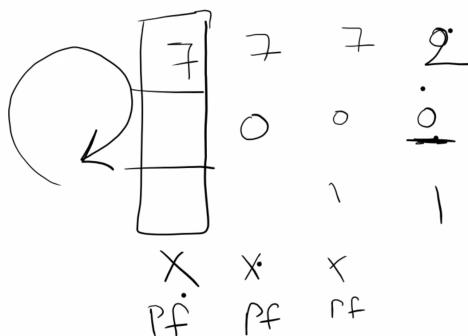
- The FIFO replacement algorithm associates with each page the time when the page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- It is not necessary to record the time when a page is brought in.
 - Creating a FIFO queue to hold all pages in memory.
 - Replacing the page at the head of the queue
 - Inserting the page that is brought into memory at the tail of the queue.

Prima page caricata in memoria → è anche la prima selezionata per essere sostituita.

First-In-First-Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

R₁



First-In-First-Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
page frames

Below the table, the page frames are shown as a sequence of boxes:

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

FIFO Illustrating Belady's Anomaly

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5 ↙
 - The number of faults for four frames (10) is greater than the number of faults for three frames (9).

Potremmo pensare intuitivamente che se abbiamo più frames a disposizione, il numero di PF dovrebbe diminuire.

In realtà con FIFO non è esattamente così, perché FIFO tiene conto solo del tempo di arrivo delle pages in memoria.

FIFO Illustrating Belady's Anomaly

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - The number of faults for four frames (10) is greater than the number of faults for three frames (9).

1 1 1
2 2
X X 3
X

↑

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

f₁ f₂ f₃

9 PF 3 frame

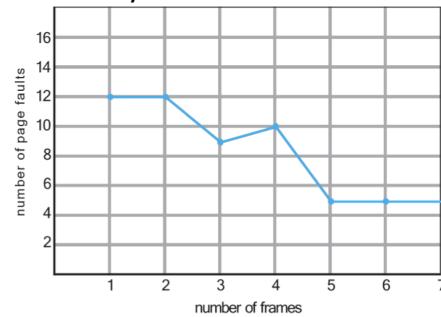
↑ ↑ ↑

1	2	3	4	1	2	3	4	5	
2	3	4	5	1	2	3	4		
3	4	5	1	2	3	4			
4	5	1	2	3	4	5			
X	X	X	X	X	X	X	X	X	inf

10 PF → inf

FIFO Illustrating Belady's Anomaly

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - The number of faults for four frames (10) is greater than the number of faults for three frames (9).
 - This effect is called **Belady's anomaly**.
 - For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increase.

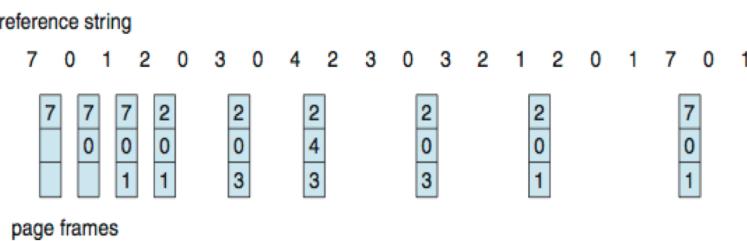


FIFO non tiene conto se la pagina caricata da più tempo in RAM (quindi la victim page di FIFO) è stata usata più volte, o se è stata modificata, o se è stata usata recentemente etc..

Optimal Algorithm

Optimal Algorithm

- Replace page that will not be used for longest period of time
 - Guaranteeing the lowest possible page-fault rate for a fixed number of frames.
 - Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 >> 9 page fault
 - Difficult to implement since it requires future knowledge of the reference string.
 - It is used mainly for comparison study, measuring how well a new algorithm performs.



Sostituiamo la pagina che sappiamo *non verrà usata per un tempo maggiore rispetto alle altre*.

Optimal Algorithm ↩

- Replace page that will not be used for longest period of time
 - Guaranteeing the lowest possible page-fault rate for a fixed number of frames.
 - Reference string: ~~7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 >> 9 page fault~~
 - Difficult to implement since it requires future knowledge of the reference string.
 - It is used mainly for comparison study, measuring how well a new algorithm performs.

f_1	7	7	7	2	2	2	2	2	2	2	2	2	2	7	7	2	7	7	7	
f_2	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0	
f_3	1	1	1	3	3	3	3	3	3	3	1	1	1	1	1	1	1	1	1	
	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	

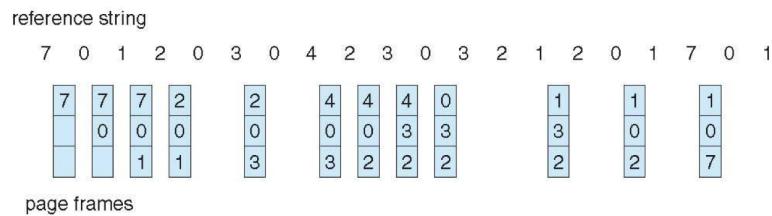
Il problema di questo algoritmo è che noi **non possiamo sapere** quali sono le pagine che verranno usate in futuro.

In linea teorica è l'algoritmo più efficiente, ma non possiamo implementarlo. Quindi studiamo altri algoritmi che possano avvicinarsi più o meno alle sue prestazioni.

Least Recently Used

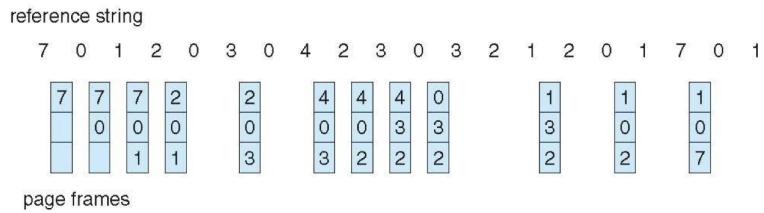
Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
 - Replace page that has not been used in the most amount of time
 - Associate time of last use with each page
 - The main difference between the FIFO and OPT algorithms is that FIFO uses the time when a page was brought into memory, OPT uses the time when a page is to be used.



Least Recently Used (LRU) Algorithm

- The LRU produces 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?



Come possiamo implementarlo?

Dobbiamo tenere conto di quando una pagina è stata chiamata.

Due tecniche:

LRU Algorithm Implementation

- Two implementation is feasible:
 - Counters
 - Associating with each page-table entry, a time-of-use field and ass to the CPU a logical clock or counter.
 - Stack
 - Keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top.

Counter

LRU Algorithm Implementation - Counter

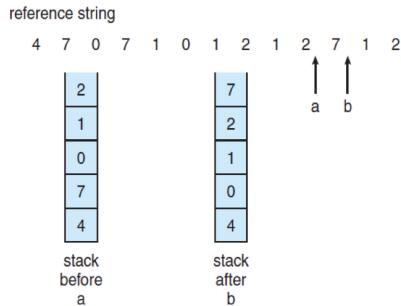
- Counter implementation
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
 - In this way, we always have the time to the last reference to each page.
 - When a page needs to be changed, look at the counters to find smallest value
- This scheme requires the search through the page table to find the LRU page and write to memory for each memory access.

Svantaggi: dobbiamo leggere tutta la page table per trovare la pagina meno recentemente usata, e inoltre ad ogni accesso in memoria dobbiamo anche scrivere (per aggiornare il contatore).

Stack

LRU Algorithm - Stack

- Stack implementation
 - Keep a stack of page numbers
 - Whenever a page is referenced, it is removed from the stack and put on the top.
 - In this way, the most recently used page is always at the top of the stack and the least recently on the bottom.



A fondo dello stack troviamo la pagina usata meno recentemente, nella cima invece la pagina usata più recentemente.

Dunque la pagina usata più recentemente è sempre sul top dello stack.

LRU Algorithm - Stack

- Stack implementation
 - Since entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer.
 - Removing a page and putting it on the top of the stack requires changing six pointers at worst.
 - Each update is a little more expensive, but there is no search for a replacement.
 - The tail pointer points to the bottom of the stack, which is the LRU page.
- LRU and OPT are cases of [stack algorithms](#) that do not have Belady's Anomaly.

LRU Approximation Algorithms

- LRU needs special hardware and still slow. Many systems provide some help in form of [Reference bit](#)
 - Reference bits are associated with each entry in the page table.
 - Initially, all bits are cleared (to 0) by the operating system.
 - When page is referenced, the bit set to 1 by the hardware.
 - After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use.
 - This is referred to as approximate LRU algorithm.

Reference bit: viene posto a 1 se la pagina viene *referenziata* (cioè viene letta o scritta, indipendentemente).

Con questo bit possiamo sapere quali pagine sono state usate, ma non possiamo sapere **quante volte, o in quale ordine**.

Dopo tot tempo controlliamo quali pagine sono state usate tramite il **Reference bit**, e buttiamo fuori una pagina con **Reference bit a 0**.

Second-chance

Second-chance Algorithm

- **Second-chance algorithm** is a FIFO replacement algorithm
- When a page has been selected, however, we inspect its reference bit.
 - If the value is 0, we proceed to replace this page.
 - If the value is 1:
 - We give the page a second chance and move on to select the next FIFO page.
 - When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
 - replace next page, subject to same rules

FIFO ma con l'aggiunta del **Reference bit**.

Quando stiamo per sostituire la pagina secondo FIFO, controlliamo il suo Reference Bit.

Se è 0, vuol dire che da quando è stata caricata in memoria **non è stata usata**, quindi possiamo sostituirla.

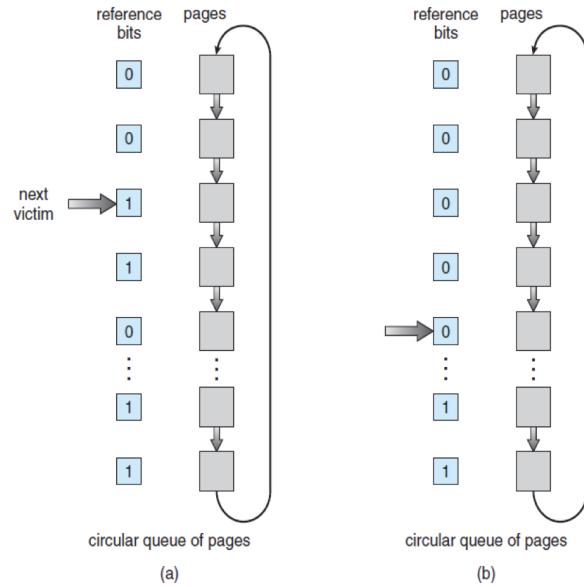
Se è 1, le diamo una

second chance e ci spostiamo sulla prossima pagina della coda FIFO, ripetendo il procedimento.

Quando una pagina ottiene una

second chance, il suo Reference bit viene resettato a 0 e il suo *arrival time* viene resettato al tempo attuale.

Second-chance Algorithm



Notiamo che stiamo usando un **circular buffer**, per cui quando carichiamo una nuova pagina in memoria, il puntatore all'inizio del buffer viene **spostato** in modo che la nuova pagina risulti essere l'**ultima** pagina del buffer.

Enhanced Second-Chance

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit
- With these two bits, we have the following four possible classes:
 - **(0, 0)** neither recently used nor modified – best page to replace
 - **(0, 1)** not recently used but modified – not quite as good, must write out before replacement
 - **(1, 0)** recently used but clean – probably will be used again soon
 - **(1, 1)** recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement is called for, use the clock scheme but use the four classes to replace the page in the lowest non-empty class
 - Might need to search circular queue several times

Algoritmo più completo che include le informazioni di **Reference bit** e **Dirty bit**.

Counting-based Algorithms (LFU, MFU)

Counting-based Algorithms

- Keep a counter of the number of references that have been made to each page
 - Not common
- **Least Frequently Used (LFU) Algorithm:** replaces page with smallest count
 - The problem is when a page is heavily used during the initial phase of a process but then is never used again.
- **Most Frequently Used (MFU) Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

Page-Buffering Algorithms

- Systems commonly keep a pool of free frames, always
 - When a page fault occurs, a victim frame is chosen as before.
 - The desired page is read into a free frame from the pool before the victim is written out.
 - The process restart as soon as possible, without waiting for the victim page to be written out.
 - The victim is written out, its frame is added to the free-frame pool.

Esercizi

Esercizio 1

Exercise 1:

→ 3 fl.

- Considering the following page reference string: 7,2,3,1,2,5,3,4,6,7,7,1,0,5,4,6,2,3,0,1
- Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms:

- LRU Replacement
- FIFO Replacement
- Optimal Replacement

7	0	1	3	3	3	7	7	7	5	5	5	2	2	2	1
2	2	2	2	4	4	4	9	1	1	1	4	4	4	3	3
3	3	5	5	5	6	6	6	0	0	0	6	6	6	0	0
3	X	X	X	X	.	X	X	X	X	X	X	X	X	X	X
															18X

→ LRU Replacement >> 18

- FIFO Replacement >> 17
- Optimal Replacement >> 13

LRU

Exercise 1:

- Considering the following page reference string: 7,2,3,1,2,5,3,4,6,7,7,1,0,5,4,6,2,3,0,1
- Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms:

- LRU Replacement
- FIFO Replacement
- Optimal Replacement

7	7	1	1	1	1	1	1	1	1	1	1	1	1	1)
2	2	5	5	5	5	5	5	5	4	6	2	3			
3	3	3	4	6	7	0	0	0	0	0	0	0			
3	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
															13

Optimal Replacement

Esercizio 2

Exercise:

- Consider the following page reference string: 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6.
- How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames?
- Remember that all frames are initially empty, so your first unique pages will cost one fault each: 1. LRU replacement 2. FIFO replacement 3. Optimal replacement

Frame #	LRU	FIFO	Optimal
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7

(DA FARE)

Esercizio 3

Exercise:

Consider the two-dimensional array A: int A[][] = new int [100][100]; Where A[0][0] is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (location 0 to 199). Thus, every instruction fetch will be from page 0.

- For three-page frames, how many page fault are generated by the following array_initialization loop? Use LRU replacement and assume that **page frame 1 contains the process** and the other two are initially empty.

a. for (int j = 0; j < 100; j++)
 for (int i = 0; i < 100; i ++){
 A[i][j] = 0;
 } **answer = 5,000**

b. for (int i = 0; i < 100; i++)
 for (int j = 0; j < 100; j ++){
 A[i][j] = 0;
 } **answer = 50**

(ESERCIZIO SIMILE ALL'ESAME)

Exercise:

`int A[][] = new int [100][100]` - pages of size 200 - For three-page frames, how many page fault are generated by the following array_initialization loop? Use LRU replacement - page frame 1 contains the process

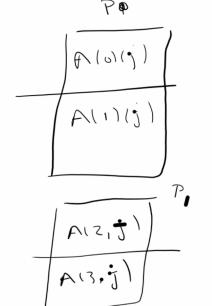
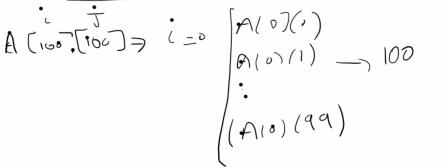
a. for (int j = 0; j < 100; j++)
 for (int i = 0; i < 100; i ++){
 A[i][j] = 0;
 } **answer = 5,000**

b. for (int i = 0; i < 100; i++)
 for (int j = 0; j < 100; j ++){
 A[i][j] = 0;
 } **answer = 50**

Exercise 3:

int A[100][100] - pages of size 200 - For three-page frames, **LRU replacement - page frame 1 contains process**

a. for (int j = 0; j < 100; j++)
 for (int i = 0; i < 100; i ++){
 A[i][j] = 0;
 }
answer = 5,000



b. for (int i = 0; i < 100; i++)
 for (int j = 0; j < 100; j ++){
 A[i][j] = 0;
 }
answer = 50

In ogni pagina, che è grande 200, possiamo ospitare due *iterazioni* dell'array, quindi ad esempio $[0][0..99]$ e $[1][0..99]$.

In totale abbiamo dunque bisogno di 50 pagine (da 0 a 49) per memorizzare l'array.

Sappiamo poi che il primo page frame viene usato per contenere il processo. Abbiamo quindi 2 frames disponibili per caricare pagine di array.

Nel ciclo a), si ha che abbiamo un page fault ogni 2 elementi di indice i (ciclo interno), ma poi c'è il ciclo esterno (di j) che ripete i page faults.

Quindi in totale abbiamo: 50PF per il ciclo interno

$\times 100$ (iterazioni del ciclo esterno) = 5000 PF totali.

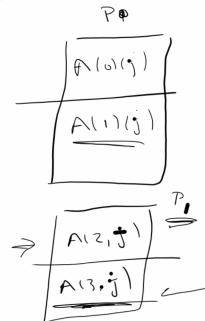
Exercise 3:

int A[100][100] - pages of size 200 - For three-page frames, LRU replacement - page frame 1 contains process

a. for (int j = 0; j < 100; j++)
 for (int i = 0; i < 100; i ++){
 A[i][j] = 0;
 }
 answer = 5,000

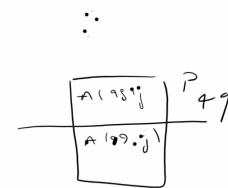
$$A[100][100] \Rightarrow i=0 \begin{cases} A(0,0) \\ A(0,1) \rightarrow 100 \\ \vdots \\ A(0,99) \end{cases}$$

$$\cdot 100 \times 50 =$$



b. for (int i = 0; i < 100; i++)
 for (int j = 0; j < 100; j ++){
 A[i][j] = 0;
 }
 answer = 50

$$50 \times 100 \Rightarrow i=1 \begin{cases} A(1,0) \\ A(1,1) \rightarrow 100 \\ \vdots \\ A(1,99) \end{cases}$$



Vediamo il ciclo b).

Abbiamo solo un PF iniziale quando referenziamo A[0][1], e portiamo in f0 p0, che contiene le prime 2 iterate dell'array.

Per com'è scritto il ciclo, non ci sono altri PF finchè non raggiungiamo A[2][0].

Quindi in totale abbiamo 50PF!

Esercizio 4

Exercise (Exam 1/7/2016):

- Si consideri la seguente sequenza di riferimenti in memoria nel caso di un programma di 1000 parole:
261, 409, 985, 311, 584, 746, 632, 323, 470, 915, 858.
- Si determini la stringa dei riferimenti a pagine, supponendo che la loro dimensione sia di **200 parole**. Si utilizzi un algoritmo di sostituzione pagine di tipo **second-chance** (con limite di **3 frame disponibili**).
- Si supponga che i riferimenti inizino a programma già avviato, con i 3 frame già allocati alle pagine n. **4, 3, e 1**, e la coda FIFO contenente, nell'ordine, $4_0, 3_1, 1_0$ (Il pedice rappresenta il reference bit).
- Determinare **quali e quanti page fault** (accessi a pagine non presenti nel resident set) si verificheranno (è richiesta la visualizzazione del resident set dopo ogni riferimento). Si supponga che il bit di riferimento di una pagina venga inizializzato a 0 in corrispondenza a un page-fault.

	1	2	4	1	2	3	3	1	2	4	4
4_0											
3_1											
1_0											

Prima cosa da fare: capire in che pagine si trovano le parole referenziate.

$1000 / 200 = 5 \rightarrow 5$ pagine in totale per memorizzare 1000 parole.

261 → Seconda pagina: partendo da p0, sta su **p1**

409 → Terza pagina:

p2

E così via..

Ricorda che partiamo da 0, quindi p0 = da 0 a 199.

p1 = da 200 a 399.

p2 = da 400 a 599 etc..

Per capire direttamente ad esempio dove si trova 985, facciamo $985 / 200 = 4, \dots$

Cioè 4 virgola qualcosa → 985 si trova nella

quinta pagina → **p4**

Passiamo ora all'esercizio vero e proprio: arriva p1, ma vediamo che è già in memoria, quindi non c'è PF ma dobbiamo cambiare il Reference bit: da 0 a 1.

Il puntatore di FIFO in questo caso non cambia, rimane su p4.

	1	2	4	1	2	3	3	1	2	4	4
4 ₀	1₀	2 ₀									
3 ₁	3 ₁										
1 ₀	1 ₀										

Arriva p2, e dobbiamo sostituire una pagina.

FIFO punta a p4, che ha Reference bit a 0, quindi possiamo sostituire.

Il puntatore FIFO si sposta su p3.

	1	2	4	1	2	3	3	1	2	4	4
4 ₀	1₀	2 ₀									
3 ₁	3 ₁	3₁	3 ₁								
1 ₀	1 ₀	1 ₀									

Arriva p4, e dovremmo sostituire p3. Ma p3 ha Reference bit a 1 → Quindi second chance: manteniamo p3, cambiamo il Reference bit da 1 a 0 e spostiamo avanti il puntatore (dunque su p1).

Lo stesso succede con p1, quindi sostituiamo p2.

Il puntatore ritorna su p3.

	1	2	4	1	2	3	3	1	2	4	4
4 ₀	1₀	2 ₀	4₀	4 ₀	6 ₀						
3 ₁	3 ₁	3₁	3 ₀	3₀	2 ₀						
1 ₀	1 ₀	1 ₀	1 ₀	1 ₁	1 ₂						

E così via..



	1	2	4	1	2	3	3	1	2	4	4
4 ₀	4 ₀	2 ₀ .	4 ₀	4 ₀	6 ₀	3 ₀	3 ₁	3 ₁	3 ₁	3 ₀	
3 ₁	3 ₁	3 ₁	3 ₀	3 ₀	2 ₁	2 ₀					
1 ₀	1 ₁	1 ₁	1 ₀	1 ₁	1 ₀	1 ₀	1 ₁	1 ₁	1 ₁	1 ₀	
	X	X		X	X						

Nel caso in cui abbiamo tutte e tre le pagine con reference bit a 1, si azzerano tutti perché la pagina viene selezionata per la sostituzione, ma con Reference bit a 1 si dà second chance e quindi si mette a 0 e si sposta il puntatore.

Alla fine si avranno tutti a 0 e il puntatore torna su 2, che verrà sostituito.

Alla fine si ha:



	1	2	4	1	2	3	3	1	2	4	4 ✓
4 ₀	4 ₀	2 ₀ .	4 ₀	4 ₀	6 ₀	3 ₀	3 ₁	3 ₁	3 ₁	3 ₀	3 ₀
3 ₁	3 ₁	3 ₁	3 ₀	3 ₀	2 ₁	2 ₀	4 ₁				
1 ₀	1 ₁	1 ₁	1 ₀	1 ₁	1 ₀	1 ₀	1 ₁	1 ₁	1 ₁	1 ₀	1 ₀
	X	X		X	X						

— LEZIONE 15/04/24

Summary

Summary

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Page replacement is: if no frame is free, we find one that is not currently being used and free it.
 - Free frame by writing its content to swap space and changing the page table.
 - Using the free frame to hold the page for which the process faulted.

Non ci piace che un processo porti in memoria tutta la sua memoria anche quando non è necessario.

Quindi abbiamo pensato di dividere la memoria di un processo in pagine, e portiamo in RAM solo quelle necessarie.

Se la page che serve al processo ad un certo punto non è presente in RAM, si ha un PF che verrà gestito per portare la pagina richiesta in RAM.

Se non c'è spazio, bisogna sostituire una pagina già presente in memoria.

Summary

- Page Replacement Algorithms
 - FIFO Page Replacement
 - Optimal Page Replacement
 - LRU Page Replacement
 - LRU-approximation Page Replacement
 - Counting-based Page Replacement
 - Page-Buffering Algorithm

Abbiamo quindi visto i meccanismi di **page replacement**.

Summary

- Two major problems to be solved for implementing demand paging:
 - **Page-replacement algorithm**
 - When page replacement is required, we must select the frames that are to be replaced.
 - Want the lowest page-fault rate
 - **Frame-allocation algorithm**
 - If we have multiple processes in memory, we must decide how many frames to allocate to each process.

Dobbiamo gestire 2 cose:

- **Page-replacement**
- **Frame-allocation**

Se abbiamo più di un processo, come dividiamo i frames di RAM disponibili tra questi processi?

Allocation of Frames

Allocation of Frames

- How do we allocate the fixed amount of free memory among various processes?
 - If we have 93 free frames and two processes, how many frames does each process get?
 - Operating system may take 35 frames, 93 frames for user space.
 - Pure demand paging, all 93 frame be put on free-frame list.
 - The first 93 pages faults would all get free frames from the free-frame list.
 - For a new page demand, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with 94th.
 - When the process terminated, the 93 frames would be placed on free-frame list.

Prima idea di Frame Allocation: ***pure demand paging***.

Immaginiamo di avere P1 e P2 che voglio eseguire, e che abbiamo 93 frames liberi.

Nel *pure demand paging* non stiamo tenendo conto di quanti frames stiamo allocando per ogni processo: ogni processo che fa PF ottiene "indiscriminatamente" un nuovo frame.

Allocation of Frames

- How do we allocate the fixed amount of free memory among various processes?
 - If we have 93 free frames and two processes, how many frames does each process get?
- Each process needs **minimum** number of frames
 - When a page fault occurs before an execution of the instruction is complete, the instruction must be restarted.
 - We need enough frames to hold all the different pages that any single instruction can reference.
- **Minimum** number of frames per process is defined by the architecture.
- **Maximum** is the total frames in the system.

Vediamo altre soluzioni — vorremmo dedicare un numero minimo di frames ad **ogni processo**.

Se abbiamo sempre 93 frames liberi e P1 e P2 da voler eseguire, quanti frames dovremmo allocare per P1 e P2?

Allocation of Frames

- Two major allocation schemes
 - Equal Allocation
 - Proportional Allocation

Due strategie:

- **Equal Allocation**

- **Proportional Allocation**

Equal Allocation

Equal Allocation

- **Equal allocation** – to split m frames among n processes is to give everyone an equal share, m/n frames (ignoring frames needed from OS for now).
 - For example, if there are 93 frames (after allocating frames for the OS) and 5 processes, give each process 18 frames
 - Keep the 3 left as free frame buffer pool.



Dividiamo il numero di frames liberi totale per il numero di processi.

Ad esempio, avendo 93 frames e 5 processi, ogni processo otterebbe 18 frames.

Svantaggi: può succedere che un processo non abbia bisogno di tutti i 18 frames, magari ha bisogno di meno frames.

Inoltre in questo modo non teniamo conto della priorità dei processi.

Proportional Allocation

Proportional Allocation

- **Proportional allocation** – allocation of available memory to each process according to its size.

- Considering a system with a 1KB frame size.
- If a small student process of 10KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames.
 - The student process does not need more than 10 frames.

$$P_1 = 10 \text{ KB} \rightarrow 10 \text{ P}$$

$$P_2 = 127 \text{ KB} \rightarrow 127 \text{ P}$$

62 frames

$$P_1 = \frac{10}{137} \times 62$$

$$P_2 = \frac{127}{137} \times 62$$

$$P_1 = 10 \text{ KB}$$

$$P_2 = 127 \text{ KB}$$

$$\Delta x = \frac{10 \times 127}{137} = 31$$

62 frames

Allocare la memoria disponibile ad ogni processo tenendo conto della dimensione del processo.

Se P1 è più grande di P2, P1 dovrebbe avere più frames rispetto a P2.

Diamo dei "pesi" quindi, dati dal rapporto tra il numero di pagine di cui ha bisogno il processo, fratto la somma di pagine necessarie per tutti i processi.

Quindi i frames assegnati ai due processi sono rispettivamente:

$$P_1 = \frac{10}{137} * 62 = 4,55 \Rightarrow 4 \quad P_2 = \frac{127}{137} * 62 = 57,47 \Rightarrow 57$$

Proportional Allocation

- **Proportional allocation** – allocation of available memory to each process according to its size.

s_i = size of virtual memory for process p_i

$$S = \sum s_i$$

m = total number of available frames

Allocating a_i frames to the process p_i , where a_i is approximately $\gg a_i = S_i / S * m$

Splitting 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames.

$$\begin{array}{l} \overbrace{\quad\quad\quad}^{P_1=10} \quad \overbrace{\quad\quad\quad}^{P_2=127} \\ P_1 = 4 \qquad \qquad P_2 = 57 \end{array}$$

Proportional Allocation

- **Proportional allocation** – allocation of available memory to each process according to its size.
 - If the multiprogramming level increases, each process will lose some frames to provide the memory needed for the new process and vice versa.
 - Either in equal or proportional allocation, a high-priority process is treated the same as a low-priority process.
 - However, you may want to give the high-priority process more memory to speed its execution.
 - One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

In tutte e due le strategie, quando arriva un processo nuovo, i processi in esecuzione devono liberare un po' dei loro frames, per dare un po' di memoria al nuovo processo.

Quanti ne devono liberare dipende se stiamo usando Equal Allocation o Proportional Allocation.

Possiamo essere più raffinati nell'allocazione della memoria per i processi: ad esempio tenendo conto della priorità dei processi.

Global and Local replacement

Global vs. Local Allocation

- With multiple processes competing for frames, we can classify the page-replacement algorithm into two broad categories:
 - Global replacement**
 - Local replacement**

Quando allochiamo un minimo numero di frames ad ogni processo, ovviamente non è detto che quei frames bastino per il processo.

Quindi se la memoria è piena e il processo richiede un'altra pagina, dovremo liberare un frame dallo stesso processo, o da un altro.

Esistono 2 categorie di strategie di replacement.

Global vs. Local Allocation

- With multiple processes competing for frames, we can classify the page-replacement algorithm into two broad categories:
 - Global replacement** – process selects a replacement frame from the set of all frames even if the frame is currently allocated to some other process
 - Set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes.
 - The process execution time can vary greatly.

Quando un processo ha bisogno di potare in RAM una pagina, l'operazione di sostituzione la può fare su **tutti i frames** del sistema, anche se il frame è occupato da un altro processo.

In questo caso il set di frames allocati non dipende solo dal processo stesso, ma anche dagli altri processi in esecuzione: se anche gli altri fanno PF, le sostituzioni

possono impattare sul processo che stiamo valutando (perchè anch'esso può vedersi sottratti dei frames).

Global vs. Local Allocation

- With multiple processes competing for frames, we can classify the page-replacement algorithm into two broad categories:
 - Local replacement – each process selects from only its own set of allocated frames.
 - The set of pages in memory for a process is affected by the paging behavior of only that process.
 - More consistent per-process performance.

L'altra strategia si chiama Local Allocation — Quando il processo ha bisogno di un frame libero va a scegliere solo tra i frames dedicati al processo stesso.

In questo caso è più facile fare predizioni sul comportamento del processo.

Reclaiming Pages

Reclaiming Pages

- A strategy to implement a global page-replacement policy.
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero, before, we begin selecting pages for replacement.
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Tutto ciò che abbiamo detto va bene ma creerà una coda per i processi.

Un altro algoritmo è

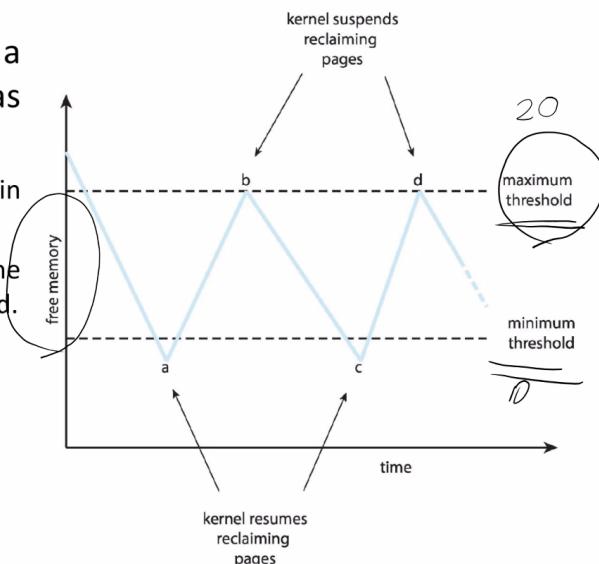
Reclaiming Pages, che fa parte della categoria *global page replacement*.

Invece di chiedere frames al momento del bisogno, creiamo un pool di frames liberi e definiamo 2 thresholds:

- Massimo
- Minimo

Reclaiming Pages Example

- When it drops below this threshold, a kernel routine is triggered known as **reapers**.
 - It begins reclaiming pages from all processes in the system.
 - When the amount of free memory reaches the max threshold, the reaper routing is suspended.



Ad esempio possiamo decidere di avere **sempre** un numero minimo e massimo di frames nel nostro sistema.

Quando arriviamo sotto il minimo, viene chiamata una routine (

reaper) che chiede a tutti i processi di liberare frames.

Questo processo continua finchè non superiamo il numero massimo.

Non-Uniform Memory Access

Non-Uniform Memory Access

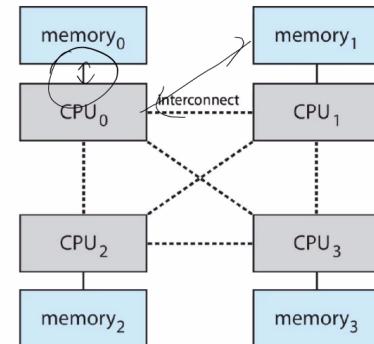
- So far all memory accessed equally.
- On **Non-Uniform Memory Access** (NUMA), systems with multiple CPU, that is not the case.
 - A given CPU can access some sections of main memory faster than it can access others.

Finora, quando abbiamo parlato di processo e di memoria, abbiamo considerato l'accesso alla memoria come un valore costante, es 10ns.

Ma in realtà, se abbiamo sistemi con più di una CPU, non abbiamo lo stesso valore per tutti i processi.

Non-Uniform Memory Access

- On **Non-Uniform Memory Access** (NUMA), systems with multiple CPU.
- A CPU can access its local memory faster than local memory of another CPU.
- NUMA systems are slower than systems in which all accesses to main memory are treated equally.



Ad esempio, se abbiamo 4 CPU, ogni cpu ha una sua local memory vicina.

Il tempo di accesso alla memoria che è fisicamente vicina alla CPU è più basso rispetto al tempo di accesso in memorie più lontane.

Ad esempio CPU0→Mem0 è più breve di CPU0→Mem1.

Trashing

Trashing

- If a process does not have “enough” pages, the page-fault rate is very high
 - Page fault to get page
 - Replace an active page
 - But quickly need replaced page back >> quickly faults again.
- A process is **trashing** if it is spending more time paging than executing.
 - This leads to:
 - Low CPU utilization
 - Operating system thinking that it needs to increase the degree of multiprogramming
 - Another process added to the system

Ultimo concetto importante di Virtual Memory: **Trashing**.

Immaginiamo di allocare un minimo numero di frames ad un processo, immaginiamo che non sia abbastanza per il processo.

Se P1, ad esempio, ha 5 pagine allocate, ma ha bisogno in totale di 150 pagine, P1 avrà molti PF.

Quando fa PF dovrà scaricare su disco una pagina attiva, di cui avrà di nuovo bisogno a breve, e quindi causerà anch'essa un PF a breve! → PF rate molto alto!

Il processo spende più tempo a spostare le pagine avanti e indietro dal disco, rispetto al tempo che impiega per effettivamente eseguire.

Questo significa che il processo è in

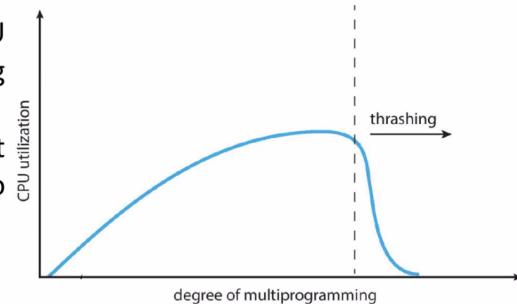
trashing.

Abbiamo allocato un numero basso di frames al processo, e fa PF così spesso che alla fine il tempo che impiega a gestirli è di più del tempo che impiega per eseguire.

Quando un processo è in trashing, la CPU utilization è molto bassa, quindi la CPU pensa che non ha molto carico di lavoro, e quindi aggiunge un altro processo alla coda di scheduling, peggiorando soltanto la situazione!

Trashing

- Operating system monitors CPU utilization.
 - Low CPU utilization, increasing the level of multiprogramming by a new process.
 - It starts faulting and taking frames from other processes while other processes need those frames.
 - These faulting must use the paging device to swap pages in and out.
- The CPU scheduler sees a decrease in CPU utilization, increasing the level of multiprogramming even more.
- At this point, thrashing is happening, and we must decrease the degree of multiprogramming to increase CPU utilization.



Vediamo dal grafico che avremo un crescendo di CPU utilization, poi i processi faranno molti PF e saranno in coda di I/O, e la coda di scheduling della CPU sarà vuota → la CPU carica altri processi perchè pensa di non avere carico di lavoro.

Working-Set Model

Demand Paging and Thrashing

- Limiting the effect of thrashing by using **Local Replacement Algorithm**.
 - It requires that each process select from only its own set of allocated frames.
 - If one process starts thrashing, it cannot steal frames from another process and cause the other to thrash as well.
- It will keep the process in the queue for a long time, increasing the access time even for a process that is not thrashing.
- To prevent thrashing, providing a process with as many frames as it needs
 - **Locality model**

Possiamo risolvere il thrashing tramite una tecnica di **Local Replacement Algorithm**.

Ogni processo va a prendere dai suoi frames, quando deve liberare un frame in seguito a PF.

Possiamo usare il Locality model: dedichiamo un numero minimo adeguato di frames ad ogni processo, per evitare che vadano in thrashing.

Demand Paging and Thrashing

- **Locality model:** The locality is a set of pages that are actively used together.
 - A running program is generally composed of several different localities.
- If we allocate enough frames to a process to accommodate its current locality, it will fault for the pages in its locality until all these pages are in memory.
- Then, it will not page fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality, the process will trash.

Di solito il numero di pagine in stato di **attivo** vengono usate insieme.

Ad esempio una funzione, su una pagina, che ha bisogno di chiamare altre funzioni presenti su altre pagine → le pagine sono attive. Le pagine vengono usate insieme.

Quando quel pezzo di codice sarà finito, avremo alcuni PF per portare in RAM il prossimo locality model del processo.

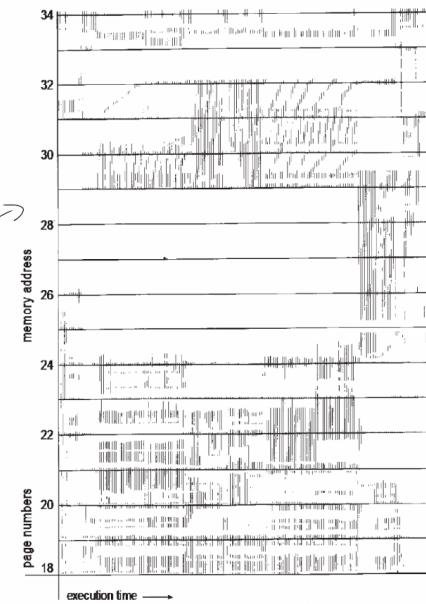
Quando tutto il

working set sarà caricato, non avremo PF di nuovo per tot tempo (finchè questo nuovo pezzo di codice non viene eseguito tutto).

Basta capire qual è il locality model di ogni processo per capire il numero minimo di frames di cui il processo ha bisogno.

Locality in a Memory Reference Pattern

- The concept of locality and how a process's locality changes over time.



Working-Set Model

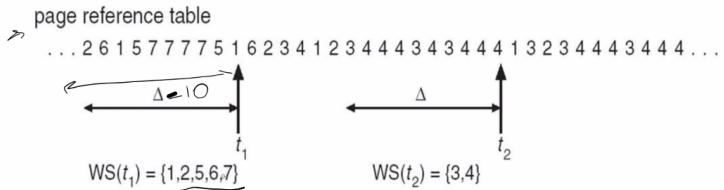
- The **working set model** is based on an assumption of locality.
- It is using a parameter Δ to define the **working-set window**.
- The set of pages in the most recent Δ page references is **working set**.
- The working set is the approximation of the program's locality.

Basta che definiamo un Working-Set Model basato sulla località spaziale e temporale.

Usiamo un parametro delta per definire la finestra temporale del working-set.

Working-Set Model

- The **working set model** is based on an assumption of locality.
- The accuracy of the working set depends on the selection of Δ .
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- The most important property of the working set is its size.
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality



Delta è il tempo che intercorre tra i momenti in cui calcoliamo il working set nuovo che serve al processo.

Come scegliamo questo delta? È variabile.

Working-Set Model

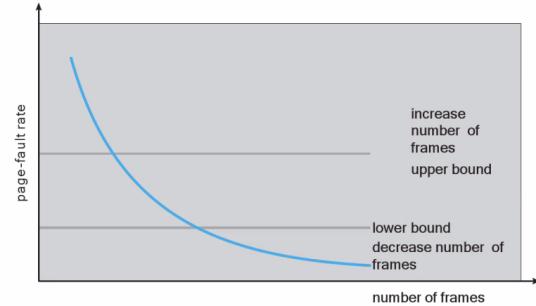
- Once Δ is chosen, the OS allocates to the working sets, enough frames to provide it with its working-set size.
- If there are enough extra frames, another process can be initiated.
- If the sum of working-set sizes exceed the total number of available frames, the operating system selects a process to suspend and swap out the pages, reallocating the frames to other processes.
- This prevents thrashing while keeping the degree of multiprogramming as high as possible.
- Challenge: How to keep track of working sets?**

Piuttosto che *guardare nel futuro* (impossibile) guardiamo nel passato.

Page-Fault Frequency

- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - When PFF is too high, the process needs more frames.
 - When PFF is too low, the process may have too many frames.

Establishing upper and lower bounds on the desired page-fault rate.



Per ogni processo monitoriamo il PF rate — se è troppo alto vuol dire che il processo ha bisogno di più frames, se è troppo basso, il processo potrebbe avere troppi frames allocati.