

Memory Management

Index

1. [Overview](#)
2. [Protection](#)
3. [Address Binding](#)
4. [Memory Management Unit \(MMU\)](#)
5. [Fragmentation](#)
6. [Paging](#)

Overview

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Memory Management

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Today:

- Why do we need Memory Management?
- Protection (one process does not access the memory of another process)
- Logical Memory vs. Physical Memory
- MMU
- Variable Partition – First Fit, Best Fit, Worst Fit
- Fragmentation

Obiettivo: vogliamo avere tanti programmi in esecuzione in parallelo, e avere esecuzioni veloci.

Vogliamo avere un meccanismo di protezione: un processo non dovrebbe poter accedere alla memoria usata da un altro processo.

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run.
- Main memory and registers are only storage CPU can access directly.
- An instructions in execution and any data being used by them, must be in one of the direct access memory.
 - Register access is done in one CPU clock (or less)
 - Main memory can take many cycles, so the processor does not have the data to complete the instruction, causing a **stall**
 - **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Quando scriviamo un programma, e lo eseguiamo, chiediamo al processore di eseguire il codice del programma.

Il codice del programma viene caricato in memoria principale, e la cpu per eseguire un'applicazione ha accesso diretto solo a memoria principale o registri.

Il file eseguibile del programma però si trova in disco, quindi il primo passo è **spostare questi dati/codice dal disco alla memoria principale**, così che la CPU vi abbia accesso diretto.

Primo problema: abbiamo "poco" spazio in memoria principale, e dobbiamo trovare un meccanismo per alternare le cose che carichiamo dal disco, che potenzialmente sono invece tante.

Non abbiamo solo un programma: vorremmo poter eseguire tanti programmi in parallelo, quindi diversi programmi che dobbiamo caricare in memoria principale.

Ogni processo caricato in RAM poi deve avere accesso solo ai suoi dati e istruzioni, e non dovrebbe poter manipolare zone di memorie riservate ad altri processi.

Se ad esempio vogliamo eseguire un programma che banalmente faccia **$C=A+B$** , non solo dobbiamo spostare in RAM l'istruzione, ma anche i dati necessari per eseguire il calcolo.

Protection

Protection

- Need to ensure that a process can access only those addresses in its address space.
- Protecting the processes from each other which is fundamental for having multiple processes for concurrent executions.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process

Come attuiamo il meccanismo di protezione, per dire ad un processo qual è la sua zona di memoria principale in cui può operare e che non può andare altrove?

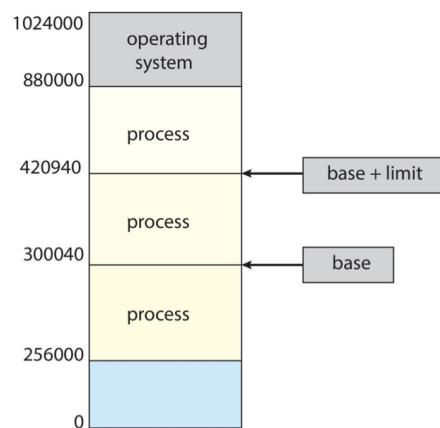
Abbiamo 2 "boundary registers":

base e **limit registers**, che definiscono l'indirizzo di base e limite in memoria principale che definiscono la zona di memoria riservata al processo.

Protection

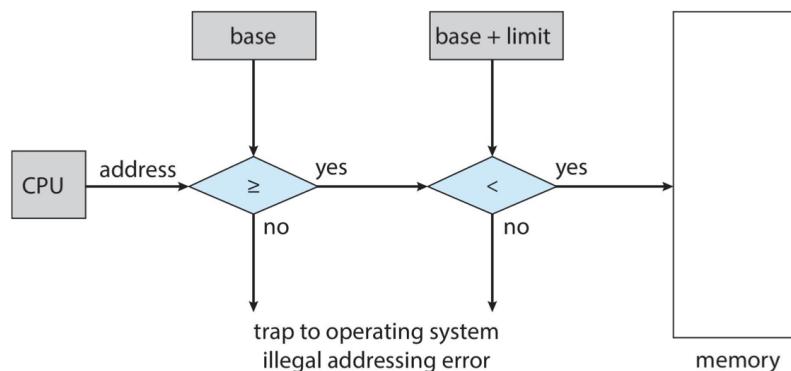
- To separate memory spaces, we need to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using a pair of **base** and **limit registers** to define the address space of a process.

- The **base register** specifies the smallest legal physical memory addresses.
- The **limit register** specifies the size of the range.



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- The base and limit registers can be loaded only by the operating systems using special privileged instructions.

La cpu deve controllare che l'indirizzo che sta cercando di usare sia compreso tra indirizzo **base** e **base + limit**, controllando appunto i registri **base** e **limit**.

Questo controllo, e il caricamento dei registri base e limit viene fatto tramite un modulo hardware apposito, ed è quindi **molto veloce**.

Address Binding

Address Binding

- Usually, a program is on disk as a binary executable file.
- To run, it should be brought into memory to be eligible for execution on the CPU.
- Most systems allow a user process to stay in any part of the physical memory, for example at 0000.
- Addresses in the source program are generally symbolic.
- A compiler typically **binds** these symbolic addresses to relocatable addresses.
- **Address binding in an operating system refers to the process of assigning a memory address to a program or a process at the time of execution.**

Quando creiamo un eseguibile, l'indirizzo di memoria definito per essere caricato in memoria nel **codice** è un indirizzo simbolico.

Ad esempio nel codice potremmo trovare che il programma deve essere caricato in 0000, ma ciò è un indirizzo simbolico: non è per niente detto che il programma verrà caricato lì.

Il compiler tipicamente **binda** gli indirizzi simbolici in indirizzi riallocabili.

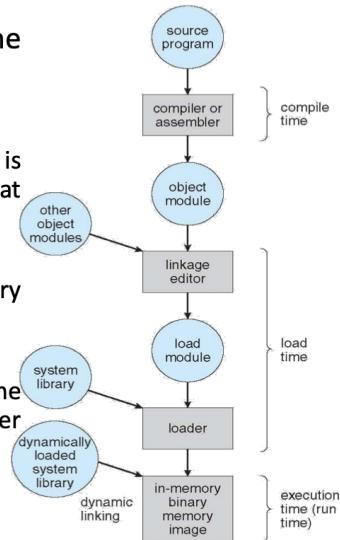
Multistep Processing of a User Program

- Compile time, load time, and run-time are important stages in the life cycle of a program:

- **Compile time:** This is the phase in which the source code of a program is converted into machine code. The machine code is an executable file that can be run on the target system.

- **Load time:** After the executable file is created, it is loaded into memory when it is executed.

- **Run time:** This is the phase in which the program is executed. During the run time, the program interacts with the operating system and other processes to perform its tasks.



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
 - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - **Load time:** if memory location is not known at compile time, then the compiler must generate **relocatable code** time. In this case, the final binding is delayed until load time.
 - **Execution time:** if the process can be moved during its execution from one memory segment to another, then binding delayed until run time.
 - Need hardware support for address maps (e.g., base and limit registers)

Il binding tra istruzioni/dati e memoria può avvenire in una di queste tre fasi.

- Compile time: specifichiamo direttamente qual è l'indirizzo di memoria effettivo che vogliamo usare
- Load time: se l'indirizzo di memoria non è noto a compile time, il compiler deve generare del codice **riallocabile** in qualsiasi zona di memoria (consentita)
- Execution time: se il processo viene spostato tra RAM e disco durante la sua esecuzione da una zona di memoria ad un altro, il binding viene fatto durante l'esecuzione del programma (*Swapping?*)

Logical vs. Physical Address Space

- An address generated by a CPU is commonly referred to as a **logical address**, also referred to as a **virtual address**.
- An address seen by the memory unit is referred to as a **physical address**.
- Binding addresses at the compile or load time generated identical logical and physical addresses.
- The execution time binding results in differing logical and physical addresses.
 - **Logical address space** is the set of all logical addresses generated by a program
 - **Physical address space** is the set of all physical addresses generated by a program

Logical address = indirizzo simbolico, non è l'indirizzo **definitivo** di memoria in cui troveremo i dati/istruzioni del nostro processo (è un **virtual address**).

Esso viene generato dalla CPU e convertito in indirizzo fisico tramite un meccanismo di mapping

virtual address → physical address.

Abbiamo dunque bisogno di *qualcosa in mezzo* tra CPU e RAM che faccia questo mapping.

Questa operazione può avvenire anche a **runtime**.

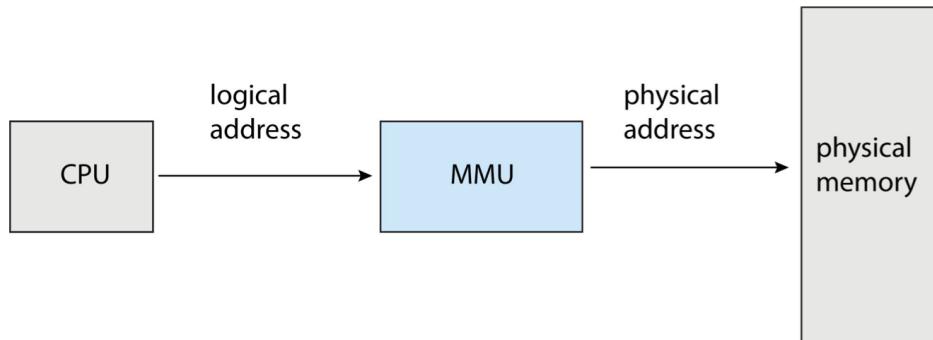
Logical vs. Physical Address Space

- An address generated by a CPU is commonly referred to as a **logical address**, also referred to as a **virtual address**.
- An address seen by the memory unit is referred to as a **physical address**.
- Binding addresses at the compile or load time generated identical logical and physical addresses.
- The execution time binding results in differing logical and physical addresses.
 - **Logical address space** is the set of all logical addresses generated by a program
 - **Physical address space** is the set of all physical addresses generated by a program
- The run-time mapping from the virtual to physical addresses is done by a hardware device called **Memory Management Unit (MMU)**

Il modulo **Hardware** che si occupa di questa operazione è la **MMU**.

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address

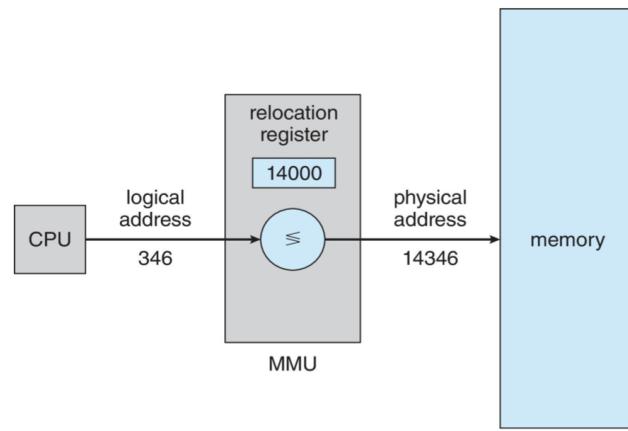


- Many methods possible, covered in the rest of this chapter

MMU

Memory-Management Unit

- Consider a simple scheme which is a generalization of the base-register scheme.
 - The base register now called the **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



Generalizzando, il base register viene chiamato **relocation register**.

Il suo valore viene sommato ad ogni indirizzo logico generato da un processo utente per definire l'
indirizzo fisico finale di memoria.

Dynamic Loading

- The entire program needs to be in memory to execute.
- Routine is not loaded until it is called.
- Loading of libraries or modules into memory at runtime – **Dynamic Loading**
- Enhances efficiency by loading only necessary components when needed.
- Reduces memory footprint and potentially speeds up program startup.

Quando eseguiamo un programma, l'intero codice del programma deve essere caricato in memoria per eseguire.

Però routines ed eventuali funzioni di librerie vengono caricate quando ve n'è l'esigenza a runtime.

Dynamic Linking

- **Static Linking** – system libraries and program code combine into the loaded into the binary program image.
- **Dynamic Linked Libraries (DLL)** are system libraries that are linked to user programs when the programs are run.
 - Collection of executable code and data
 - Shared by multiple programs simultaneously – **Shared Libraries**
- Enable sharing of resources and functionality
- Reduce memory usage and enhance modularity

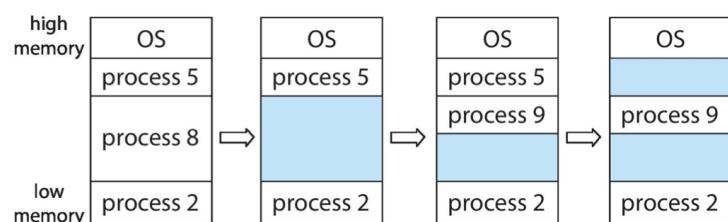
Se 2 processi usano una stessa libreria, non è necessario caricare la stessa libreria 2 volte, ma possiamo carregarla una volta sola e farla condividere dai processi.

Questo è possibile tramite le

Dynamic Linked Libraries, al contrario del meccanismo di **Static Linking** che prevede di combinare insieme codice del programma e delle librerie, rendendoli un tutt'uno e quindi caricando tutto insieme (quindi per più processi che usano la stessa libreria, questa verrebbe "caricata" più volte).

Contiguous Allocation

- Main memory must support both OS and various user processes.
- Main memory usually into two **partitions**:
 - Placing the OS in either low memory addresses or high memory addresses.
 - Residing the user processes in memory at the same time.



Come gestiamo lo spazio in RAM?

Un'idea sarebbe che, appena arriva un processo da eseguire, lo carichiamo nel primo posto disponibile in RAM.

Poi ne arriva un altro, facciamo lo stesso e così via..

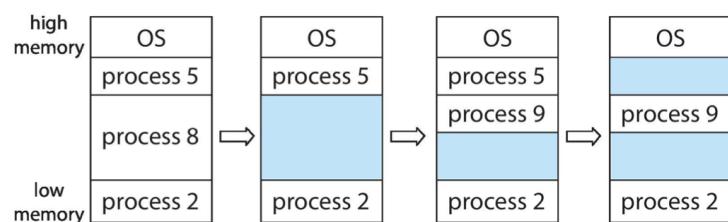
Quando un processo termina, si libera un "buco" in memoria, che eventualmente può essere occupato da un altro processo che deve essere eseguito.

Se non c'è spazio sufficiente per essere caricato, il processo viene messo in stallo e non può essere eseguito, anche se magari abbiamo 2 partizioni libere, separate da una zona di memoria occupata da un altro processo: se nessuna delle due partizioni è sufficiente per caricare il nuovo processo (ma magari la loro somma sì), il processo **non può essere caricato**.

Per questo esistono diverse tecniche per gestire lo spazio in memoria in modi più efficienti.

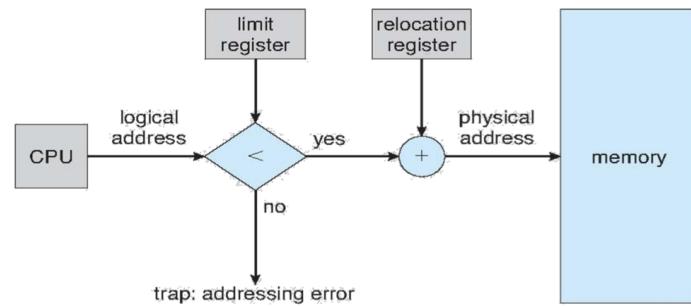
Contiguous Allocation

- How to allocate the available memory to the processes?
- In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.



Contiguous Allocation

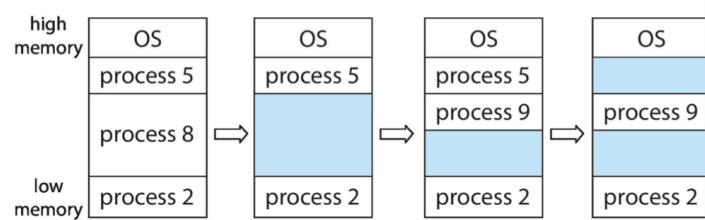
- **Relocation registers** used to protect user processes from each other, and from changing operating-system code and data
 - Base register contains the value of the smallest physical address
 - Limit register contains a range of logical addresses – each logical address must be less than the limit register
 - MMU maps logical addresses *dynamically*



Vediamo che quando il programma fa riferimento a un indirizzo di memoria, intanto vediamo se esso è < **limit register**, se sì vi sommiamo il **relocation register** per trovare l'indirizzo fisico finale mappato.

Variable Partition

- One of the simplest methods for allocating memory is to assign processes to variably sized partitions where each partition may contain exactly one process. In **Variable partition** scheme:
 - Initially, the memory is available for user processes, considered one large block.
 - Eventually, when the process terminates, it releases its memory, creating a **hole**.
 - When a new process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about allocated partitions and free partitions (hole)



Assumiamo che un grande processo venga caricato in memoria.

Quando termina lascerà un grande buco in memoria disponibile.

Se un processo piccolo viene caricato in tale buco, lascerà spazio in memoria inutilizzato.

Quindi in queste situazioni, in cui abbiamo buchi in memoria, bisogna scegliere una buona tecnica per decidere in quale "buco" caricare i nuovi processi che arrivano.

Dynamic Storage-Allocation Problem

- When a process arrived and needs memory, the system searches for a hole that is large enough for this process.
- How to satisfy a request of size n from a list of free holes – **dynamic storage allocation problem**
 - **First-fit:** Allocate the *first* hole that is big enough
 - **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless the list is ordered by size
 - Produces the smallest leftover hole
 - **Worst-fit:** Allocate the *largest* hole; must also search entire list
 - Produces the largest leftover hole which might be more useful than the smaller leftover hole from best-fit
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Tre possibilità: **First-fit**, **Best-fit**, **Worst-fit**.

First-fit

Carichiamo nel primo buco abbastanza grande da accogliere il processo.

Quindi non c'è bisogno di "analizzare" l'intera memoria, ma appena troviamo un buco disponibile per il processo, lo usiamo.

Best-fit

Carichiamo il processo nel buco più piccolo disponibile per accogliere il processo. Richiede l'analisi dell'intera memoria, a meno che la lista di spazi disponibili non sia ordinata per dimensione.

Worst-fit

Carichiamo il processo nel buco più grande disponibile.

Richiede l'analisi dell'intera memoria, a meno che la lista di spazi disponibili non sia ordinata per dimensione.

Perchè è utile questa tecnica?

Perchè lo spazio che rimane libero dopo aver caricato il nuovo processo speriamo sia abbastanza grande da poter accogliere altri processi, quando invece nel

Best-fit magari lasciavamo pochissimo spazio libero, inutile per caricare un nuovo processo in quel poco spazio.

Esercizi

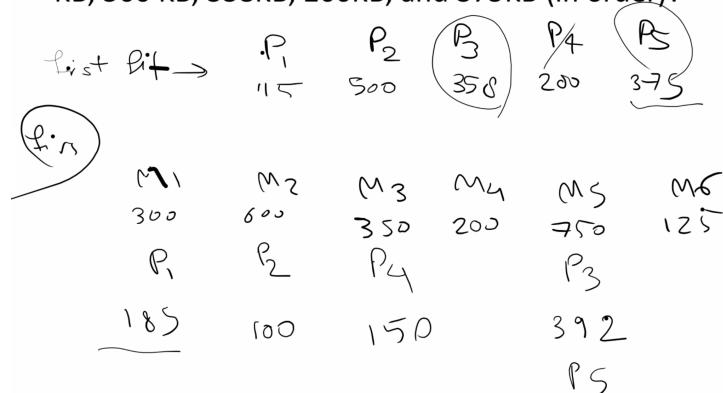
Exercise:

- Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358KB, 200KB, and 375KB (in order)?

FIRST-FIT

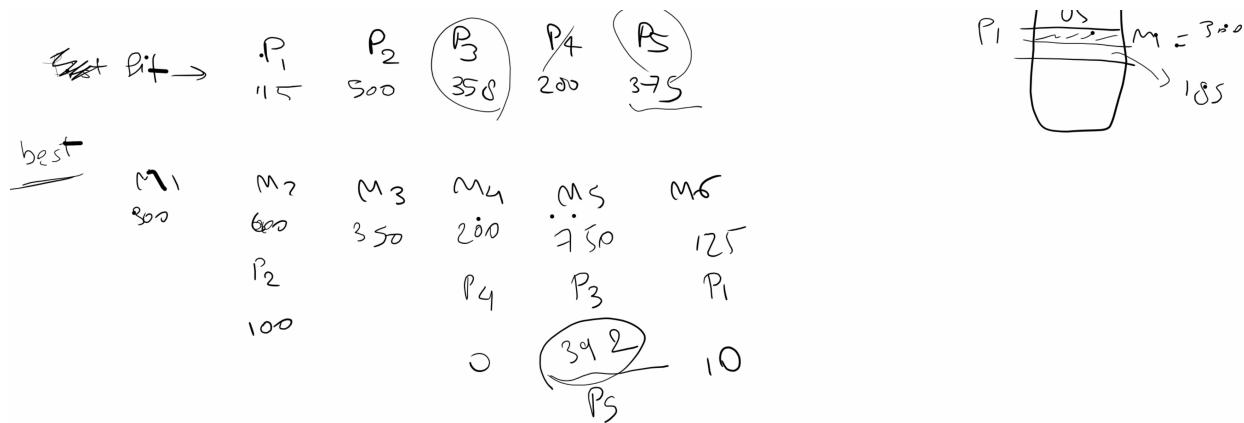
Exercise:

- Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358KB, 200KB, and 375KB (in order)?



Con questo approccio siamo stati veloci, non dovendo cercare l'intera lista di spazi disponibili. C'era tra l'altro abbastanza spazio per ciascun processo, quindi nessun processo è rimasto in attesa di avere uno spazio disponibile.

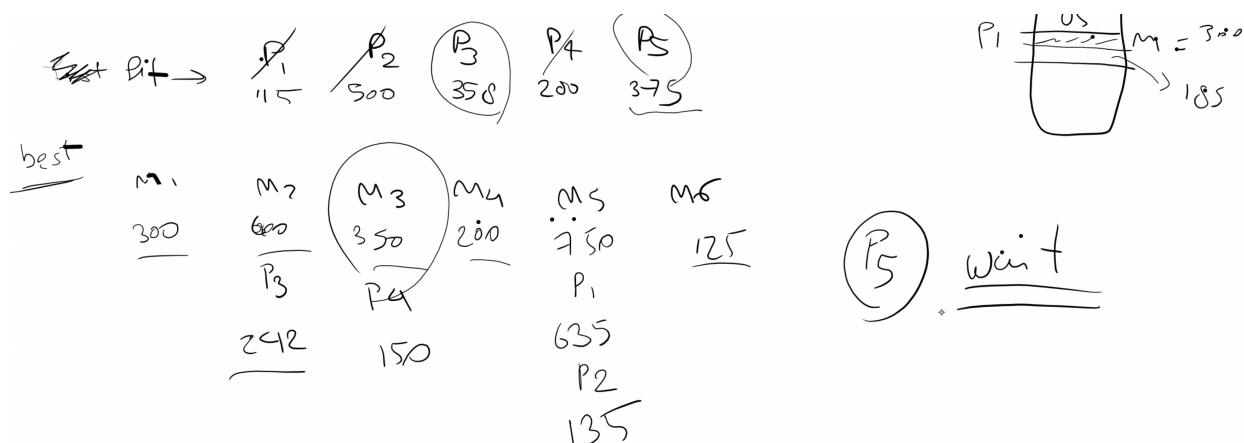
BEST-FIT



Abbiamo allocato tutti i processi che sono arrivati, senza lasciarne nessuno in attesa.

Più lento rispetto a First-fit.

WORST-FIT



Con questo approccio P5 deve rimanere in attesa perché non vi è uno spazio abbastanza grande da accoglierlo, anche se la somma dei singoli buchi sarebbe

sufficiente, ma avendo usato una tecnica di **allocazione di indirizzi di memoria contigui**, non possiamo farci nulla.

Summary

- **Why do we need Memory Management?**

- Program must be brought (from disk) into memory and placed within a process for it to be run

- **Protection (one process does not access the memory of another process).**

- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** to define the logical address space of a process

- **Logical Memory vs. Physical Memory**

- An address generated by CPU is commonly referred to as a **logical address**
- An address seen by the memory unit is referred to as **physical address**

Summary

- **MMU**

- Hardware device that at run time maps virtual to physical address

- **Contiguous memory allocation**

- each process is contained in a single section of memory that is contiguous to the section containing the next process.

- **Variable Partition – First Fit, Best Fit, Worst Fit**

- First-fit: Allocate the *first* hole that is big enough
- Best-fit: Allocate the *smallest* hole that is big enough
- Worst-fit: Allocate the *largest* hole; must also search entire list

To do:

- Fragmentation
- Paging
- Page Table
- Implementation of Page Table
- Translation look-aside buffer (TLB)
- Effective Access time
- Shared Memory

Lezione 2 — 5/03/2024

Fragmentation

Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- First fit analysis reveals that given N blocks allocated, $0.5 N$ blocks lost to fragmentation
 - 1/3 may be unusable -> **50-percent rule**

Quando i processi terminano liberano la memoria lasciando quindi dello spazio libero utilizzabile da altri processi.

Questa situazione causa

fragmentazione, che può essere di 2 tipi:

- **External Fragmentation**

Lo spazio totale per un nuovo processo esiste, ma lo spazio non è contiguo.

- **Internal Fragmentation**

Se decidiamo di partizionare la memoria in blocchi, potremmo ritrovarci nella situazione in cui un nuovo processo occupa una di tali partizioni, ma usa meno spazio dello spazio totale della partizione che occupa. Questo spazio inutilizzato **non** può essere usato da un altro processo.

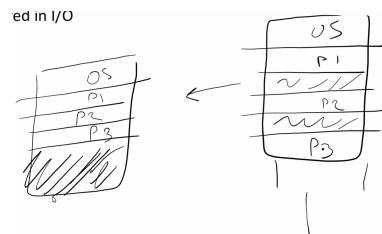
Fragmentation

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
 - I/O problem
 - Latch job in memory while it is involved in I/O
 - Do I/O only into OS buffers

Vediamo alcune soluzioni:

- **Compaction**

La CPU decide di non eseguire ciò che l'utente chiede di eseguire in quel momento, ma si prende qualche ciclo di clock per "compattare" lo spazio disponibile in memoria, dunque per spostare i processi che occupano la memoria "tutti insieme" vicini, lasciando un unico grande spazio disponibile in memoria.



Due problemi: i processi vengono spostati, ma i processi sanno qual è il loro spazio in memoria tramite base e limit registers, che quindi devono essere **aggiornati** quando i processi vengono spostati.

Altro problema: overhead, infatti la CPU è impegnata in queste operazioni piuttosto che eseguire i processi che deve eseguire.

Paging

Paging

- Another possible solution to the external fragmentation problem is to permit the logical address space of processes to be **non-contiguous**.
 - This is possible by adopting a strategy called **paging**.
 - Avoids external fragmentation
 - Needs for compaction
 - Since paging offers numerous advantages, it is used in most operating systems, from those for large servers through those for mobile devices.
 - Paging is implemented through cooperation between the operating system and the computer hardware.
- **Paging**

Finora abbiamo usato una tecnica di indirizzamento **contiguo**, quindi se ad esempio un processo ha bisogno di 4KB di spazio, abbiamo bisogno di 4KB di spazio contiguo in memoria.

Vediamo ora una tecnica di indirizzamento
non contiguo.

Dividiamo la memoria fisica in diversi blocchi di dimensione fissa, chiamati **frames**.

D'altra parte dividiamo anche la **logical memory** in diversi blocchi di dimensione fissa, che chiamiamo **pages**.

Invece di mappare tutta la memoria di cui ha bisogno un processo, possiamo mappare le pages di un processo ai frames in memoria fisica.

Quindi se ad esempio abbiamo un processo che ha bisogno in totale di 4KB, ma dividiamo il processo in 4 pages, possiamo allocare queste 4 pages in 4 frames diversi, anche non contigui.

Se prima però avevamo base e limit register, ora abbiamo bisogno di elementi più complessi per far capire al processo quali sono le zone di memorie a lui riservate.

La MMU diventa quindi più complessa, avendo da un lato

pages e dall'altro lato **frames**.

La dimensione di una

page e di un **frame** è uguale.

Paging

- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes, depending on the computer architecture
- Divide logical memory into blocks of same size called **pages**
- When a process is to be executed, its pages are loaded into any available memory frames.
- Keep track of all free frames
 - It has great functionality, for example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.

Address Translation Scheme

- Address generated by CPU is divided into:
 - **Page number (*p*)** – used as an index into a **page table** which contains base address of each page in physical memory
 - **Page offset (*d*)** – combined with base address to define the physical memory address that is sent to the memory unit

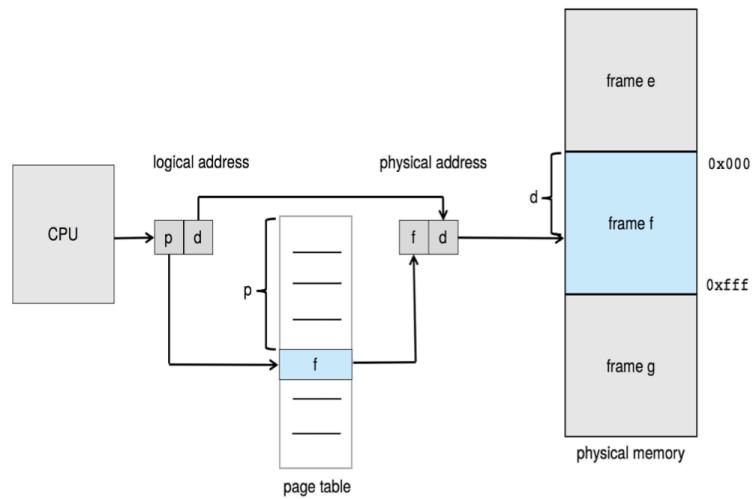
page number	page offset
<i>p</i>	<i>d</i>
$m - n$	n

- For given logical address space 2^m and page size 2^n
- This information is used by Page Table

In questa situazione, quando facciamo riferimento a un'istruzione o un dato, dobbiamo ora capire in che pagina si trova, e all'interno di tale pagina quanto dobbiamo spostarci per trovare ciò che ci interessa (offset).

Paging Hardware

- The **page table** is a data structure used by the MMU to translate virtual memory addresses into physical memory addresses.
- The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being references.
- The based address of a frame is combined with the page offset to define the physical memory address.



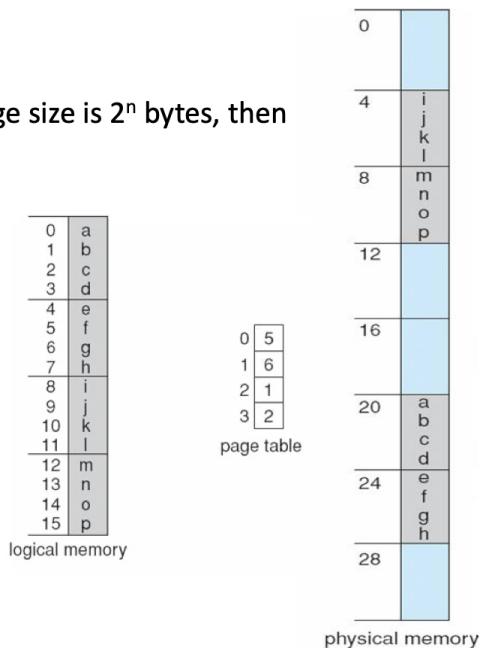
La page table memorizza il mapping tra page e frame in cui è effettivamente memorizzata la porzione di istruzioni/dati della page a cui vogliamo accedere.

Vediamo che il corrispettivo dell'offset della page, l'offset del frame in memoria fisica, è **uguale**, dato che abbiamo detto che la dimensione di page = dimensione di frame.

Dunque l'offset per trovare l'istruzione/dato in memoria fisica è uguale.

Paging Example

- If the size of the logical address space is 2^m and the page size is 2^n bytes, then the 2^{m-n} is designate the page number.
- Logical address: $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

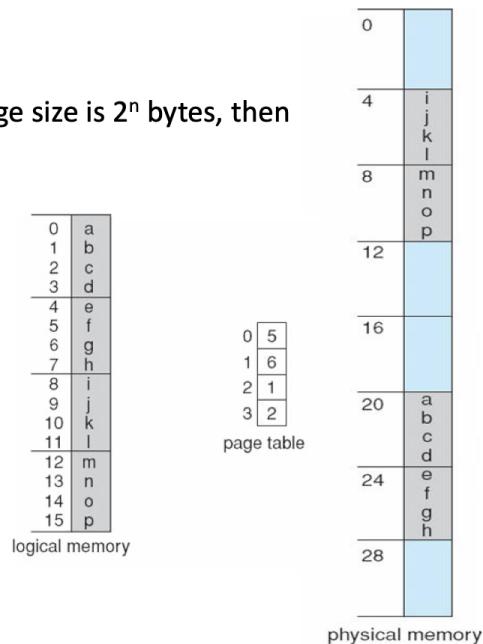


Paging Example

- If the size of the logical address space is 2^m and the page size is 2^n bytes, then the 2^{m-n} is designate the page number.
- Logical address: $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).

Logical address 0 is page 0, offset 0. from page table, page 0 is in frame 5. Logical address 0 maps to physical address:

$$20 = (5 \text{ (frame number)} * 4 \text{ (frame size)}) + 0 \text{ (offset)}$$

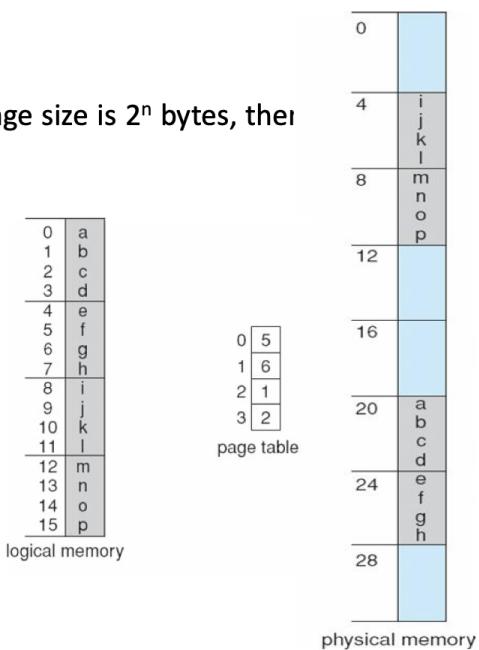


Paging Example

- If the size of the logical address space is 2^m and the page size is 2^n bytes, then the 2^{m-n} is designate the page number.
- Logical address: $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).

Logical address 5 is in page 1, offset 1. from page table, page 1 is in frame 6. Logical address 5 maps to physical address:

$$25 = (6 \text{ (frame number)} * 4 \text{ (frame size)} + 1 \text{ (offset)}$$



Paging -- Calculating internal fragmentation

- With paging, we have no external fragmentation, but we may have internal fragmentation. For example, considering a case in which:
 - Page size = 2,048 bytes
 - Process size = 72,766 bytes
 - The process will need 35 pages + 1,086 bytes
 - It will be allocated 36 frames with an internal fragmentation of 962 (2,048-1,086)
 - Worst case fragmentation = 1 frame – 1 byte, almost an entire frame
- If the process size is independent of the page size, we expect internal fragmentation to average one-half page per process.
- So, small page size is suggested, but it will result in overhead in page table entry. This overhead is increasing as the page size is decreasing.

Abbiamo risolto l'External Fragmentation, ma possiamo avere ancora **Internal Fragmentation**, in quanto vediamo dalla slide che il processo in esempio viene diviso in 35 pagine da 2048B, ma rimane ancora 1086B da "mappare".

Dunque questa quantità viene allocata in un frame, che però non verrà utilizzato completamente (1086/2048).

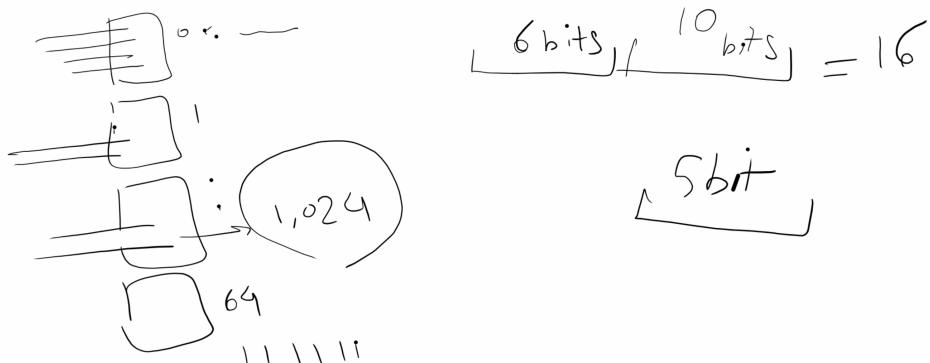
Una soluzione sarebbe ridurre la dimensione di pages/frames.

Il problema è che ogni processo ha la sua page table, e se riduciamo la dimensione di pages la dimensione della page table aumenterà (abbiamo più pages → più frames → più entries nella tabella per fare il mapping tra pages e frames).

Esercizio

Exercise:

- Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.
 - How many bits are there in the logical address?
 - How many bits are there in the physical address?



Logical address: 64 pagine da 1024 di size → 6 bit per indirizzare le pagine e 10 bit per l'offset.

Physical address: 32 frames, sempre da 1024 di size → 5 bit per indirizzare i frames e 10 bit per l'offset.

6bit+10bit = 16 bit (logical address)

5bit+10bit = 15 bit (physical address)

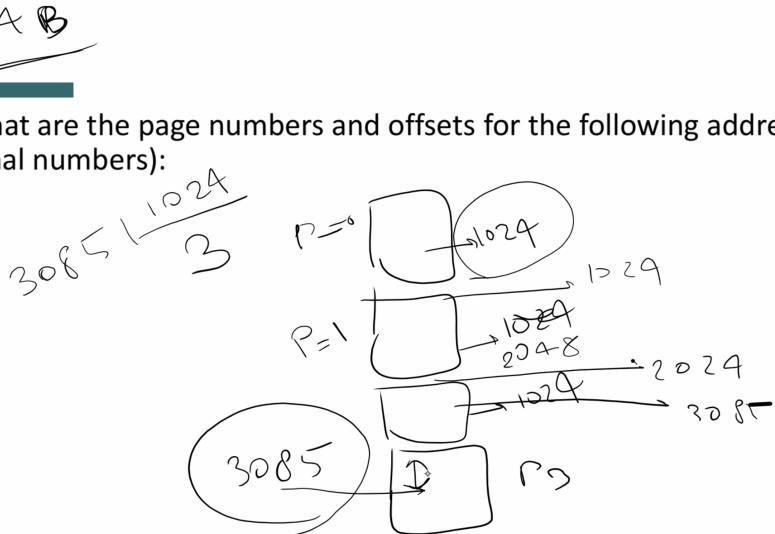
Exercise:

- Assuming a 1.KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

- 3085
- 42095
- 215201
- 650000
- 2000001

- Answer:

- Page = 3, offset = 13,
- Page = 41, offset = 111,
- Page = 210, offset = 161
- Page = 634, offset = 784
- Page = 1953, offset = 129



Divido 3085 per 1024 per vedere fino a dove arriviamo.

$$3085/1024 = 3 \text{ con resto di } 13 \quad (1024*3 = 3072).$$

Quindi troviamo in page 3 con offset 13.

E così via per gli altri indirizzi logici chiesti..

Summary:

- Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
 - External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous.
 - Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

- Paging

- Permit the logical address space of processes to be **non-contiguous**.
- Divide physical memory into fixed-sized blocks called **frames**.
- Divide logical memory into blocks of same size called **pages**.

Frammentazione: non ci piace perchè abbiamo spazio non contiguo inutilizzabile.
Allora siamo passati ad una tecnica di spazio non contiguo, dividendo la memoria

in blocchi di dimensioni fissa (pages per mem virtuale, frames per mem fisica). Ora bisogna mappare le pages ai frames in memoria fisica. Quindi la MMU ora implementa una page table, che si occupa di memorizzare il mapping tra page e frame.

Page table

Implementation of Page Table

- Page table is kept in main memory
 - [Page-table base register \(PTBR\)](#) points to the page table (Memory address of page table)
 - [Page-table length register \(PTLR\)](#) indicates the size of the page table
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data/instruction
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called [translation look-aside buffers \(TLBs\)](#).

Vengono usati due registri per dire al processo dove trovare la sua page table in memoria.

Il PTBR memorizza il puntatore alla page table del processo in memoria fisica, il PTLR indica la dimensione della page table.

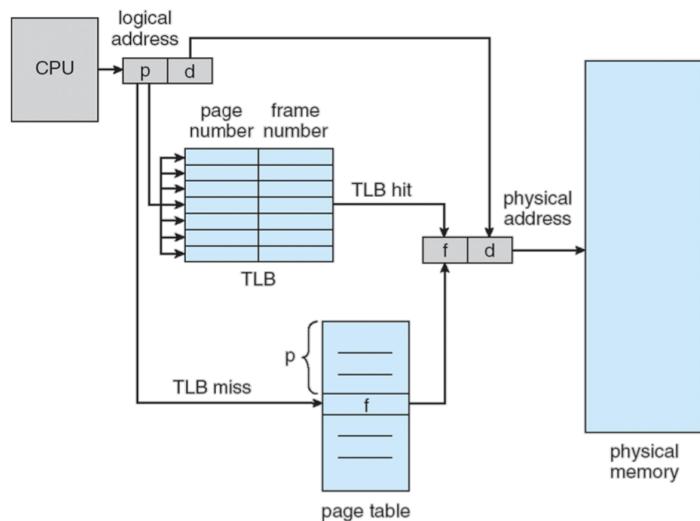
Andiamo avanti: in questa situazione quindi ogni accesso a istruzione/dato richiede 2 accessi a memoria, uno per leggere la page table e uno per leggere l'istruzione/dato.

Quindi siamo più lenti..

Troviamo un'altra soluzione: usare un'altra tabella, implementata tramite hardware, chiamata **TLB**, che è una memoria molto veloce e risolve il problema della lentezza, perchè è talmente veloce che l'accesso alla page table non viene considerato un accesso in memoria se usiamo una TLB. La TLB memorizza, come la page table, le corrispondenze page number-frame number.

Paging Hardware With TLB

- The TLB contains only a few of the page-table entry.
- When a logical address is generated by the CPU, the MMU first checks if its page number is present in TLB.
- If the page number is found, its frame number is immediately available and used to access memory.
- These steps adds negligible performance penalty.
- If the page number is not in the TLB, **TLB miss**, address translation is done as before.



La TLB è una cache molto veloce, quindi molto costosa e quindi molto piccola. Non può ovviamente contenere il mapping per tutti i processi in esecuzione, quindi quando dobbiamo tradurre un indirizzo, per prima cosa controlliamo la TLB, se troviamo la corrispondenza allora bene, altrimenti abbiamo un **TLB miss** e quindi dobbiamo fare un accesso alla memoria per andare a leggere la page table del processo in RAM.

Translation Look-Aside Buffer

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
 - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
 - Replacement policies must be considered
 - Some entries can be **wired down** for permanent fast access

Poichè la TLB può contenere entries relative a processi diversi, e ovviamente questi potrebbero usare gli stessi page numbers, c'è bisogno di capire se l'entry che eventualmente troviamo in TLB corrisponde effettivamente al nostro processo.

Per questo alcune TLB implementano gli **ASIDs**, identificatori dello spazio d'indirizzamento dei processi, associati ad ogni entry della TLB.

In alternativa dovremmo flushare la TLB ad ogni context switch.

Effective Access Time

- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**.
 - An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that it takes 10 ns to access memory.
 - If we find the desired page in TLB then a mapped-memory access takes 10 ns
 - If we fail to find the page number in the TLB, then we must first access memory for the page table and frame number (10 ns) and then access to desired byte in memory (10 ns), for a total of 20 ns.
- **Effective Access Time (EAT)**
$$EAT = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$
 - Suffering from a 20% slowdown in average memory access

Effective Access Time

- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**.
 - An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that it takes 10 ns to access memory.
- **Effective Access Time (EAT)**
$$EAT = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$
 - Suffering from a 20% slowdown in average memory access
- Consider a more realistic hit ratio of 99%,
$$EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$$
 - Implies only 1% slowdown in access time.

Summary:

- Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
 - **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
 - **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

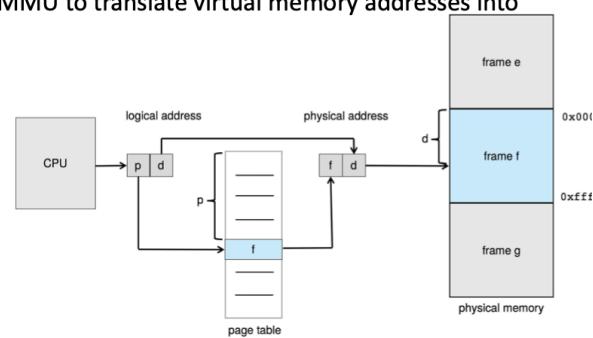
- Paging

- Permit the logical address space of processes to be **non-contiguous**.
- Divide physical memory into fixed-sized blocks called **frames**.
- Divide logical memory into blocks of same size called **pages**.

Summary:

- Page Table

- Address generated by CPU is divided into:
 - **Page number (*p*)** – used as an index into a **page table** which contains base address of each page in physical memory
 - **Page offset (*d*)** – combined with base address to define the physical memory address that is sent to the memory unit.
- The **page table** is a data structure used by the MMU to translate virtual memory addresses into physical memory addresses.



Summary:

- Implementation of Page Table
 - Page-table base register ([PTBR](#)) points to the page table (Memory address of page table)
 - Page-table length register ([PTLR](#)) indicates the size of the page table.
- Translation look-aside buffer (TLB)
 - The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called [translation look-aside buffers \(TLBs\)](#).
- Effective Access time
 - The percentage of times that the page number of interest is found in the TLB is called the [hit ratio](#).

LEZIONE 3 — 8/03/2024

Memory Protection

- Memory protection in a paged environment is accomplished by **protection bits** associated with each frame.
- These bits are kept in the page table which can define a page to be read-write or read-only.
- Every reference to memory goes through the page table to find the correct frame number.
- At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.

Su page table è memorizzato il mapping page-frame, e possiamo anche avere informazioni aggiuntive in ogni entry.

Ad esempio

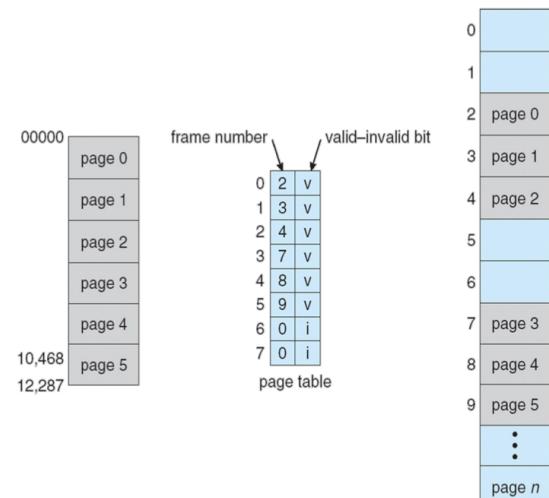
protection bit, con cui identifichiamo se può essere usata per essere letta, scritta o eseguita.

Es. Se protection bit = R, allora la page è read-only

Come funziona il protection bit?

Lo vedremo più in là con la trattazione su virtual memory.

- One additional bit, **Valid-invalid** bit, is attached to each entry in the page table
- When the bit is set to **valid**, the associated page is in the process's logical address and is legal
- When the bit is set to **invalid**, the page is not in the process's logical address space, illegal addresses are trapped.
- The operating system sets this bit for each page to allow or disallow access to the page.



Valid bit: quando un processo cerca di avere accesso ad una pagina, con questo bit controlliamo se è valida oppure no per il processo stesso. In particolare viene usato quando dobbiamo *swappare* pagine di uno stesso processo tra RAM e disco, magari bisogna fare spazio in memoria quindi una pagina nella page table diventa non più valida perché è stata sostituita da un'altra pagina (di un altro processo).

Ricorda che ogni processo ha la propria page table.

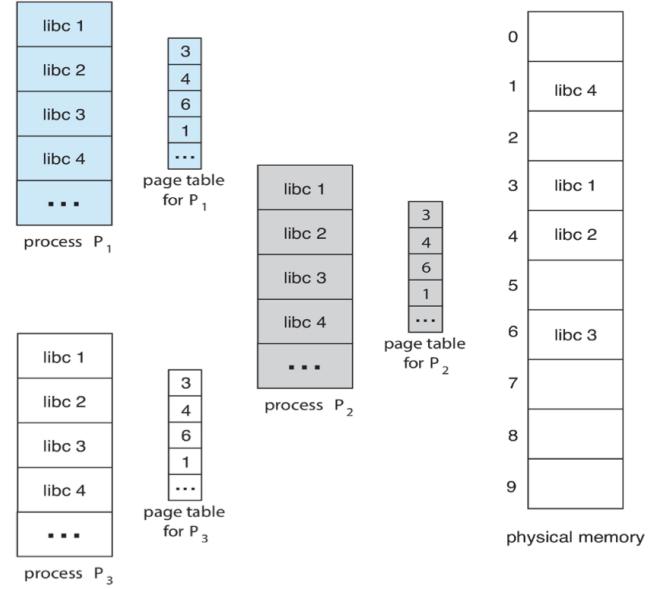
Shared Pages

- An advantage of paging is the possibility of sharing common code which is really important in multiple processes environment.
 - If the code is **reentrant code**, it can be shared among processes.
 - Reentrant code is non-self modifying code, it never changes during execution., so more processes can share the same code at the same time.
- **Private code and data**
 - Each process keeps a separate copy of the code and data
 - The pages for the private code and data can appear anywhere in the logical address space

Esempio quando includiamo <stdlib.h>.

Reentrant code: nessun processo può modificarlo, ma solo leggere o eseguire tale codice.

Shared Pages Example



Più processi possono condividere e usare contemporaneamente tale codice *reentrant*, che appunto non può mai essere modificato durante l'esecuzione.

L'indirizzo del frame in cui viene caricato il reentrant code viene mappato su ciascuna page table dei processi che condividono la libreria.

Structure of the Page Table

- Most modern computer systems support a large logical address space. Therefore, the page table itself becomes excessively large.
- Therefore, we would not want to allocate the page table continuously in main memory.
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes → each process 4 MB of physical address space for the page table alone
 - Don't want to allocate that contiguously in main memory
- One simple solution is to divide the page table into smaller units
 - **Hierarchical Paging**
 - **Hashed Page Tables**
 - **Inverted Page Tables**

La page table vista finora, che appunto è memorizzata anch'essa in RAM, ha ancora bisogno di memoria contigua per essere caricata.

Sennò avremmo bisogno di un meccanismo esattamente come la page table che ci dica dove si trovano i frames in cui troviamo la page table.. insomma!

Le page tables dei processi possono prendere "tanto" spazio, come si vede dalla slide.

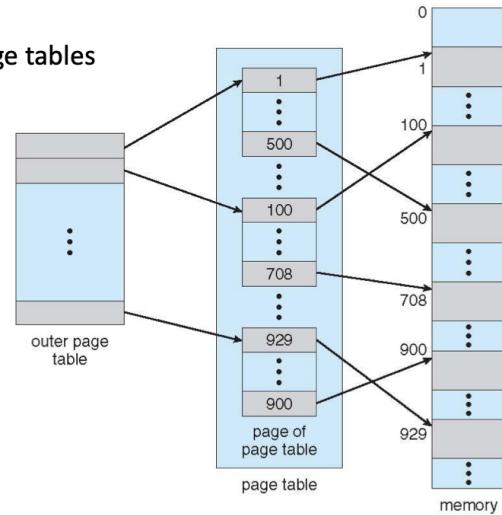
Vediamo dunque alcuni modi più efficienti (dal punto di vista dello spazio occupato in memoria) di implementare una page table:

- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**

Hierarchical Paging

Hierarchical Page Tables

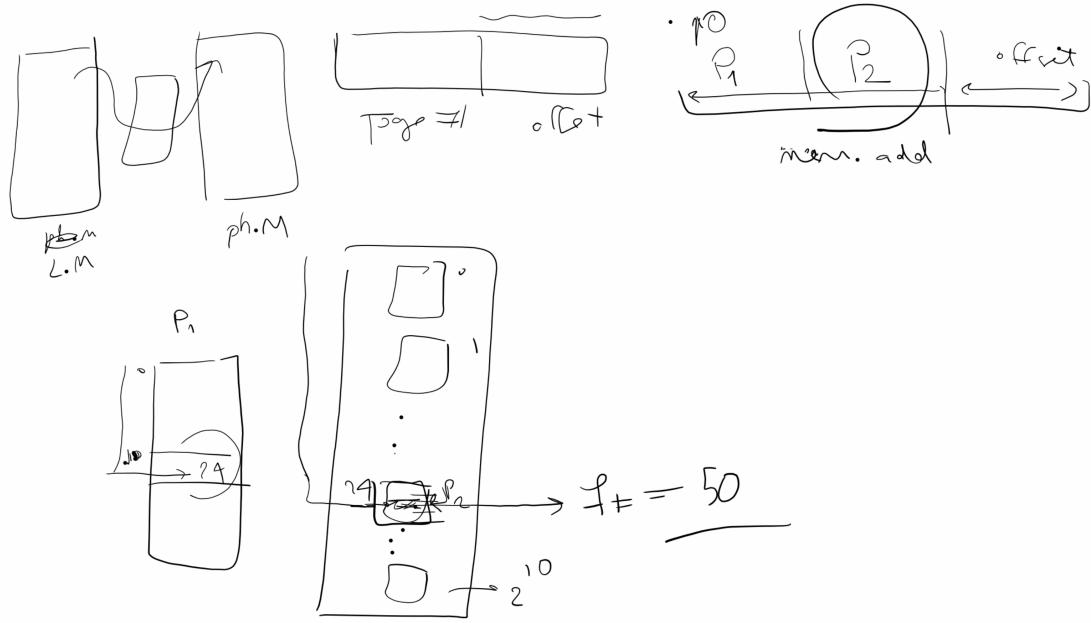
- One solution is to use a two-level paging algorithm, in which the page table itself is also paged.
 - Break up the logical address space into multiple page tables
 - A simple technique is a two-level page table
 - We then page the page table



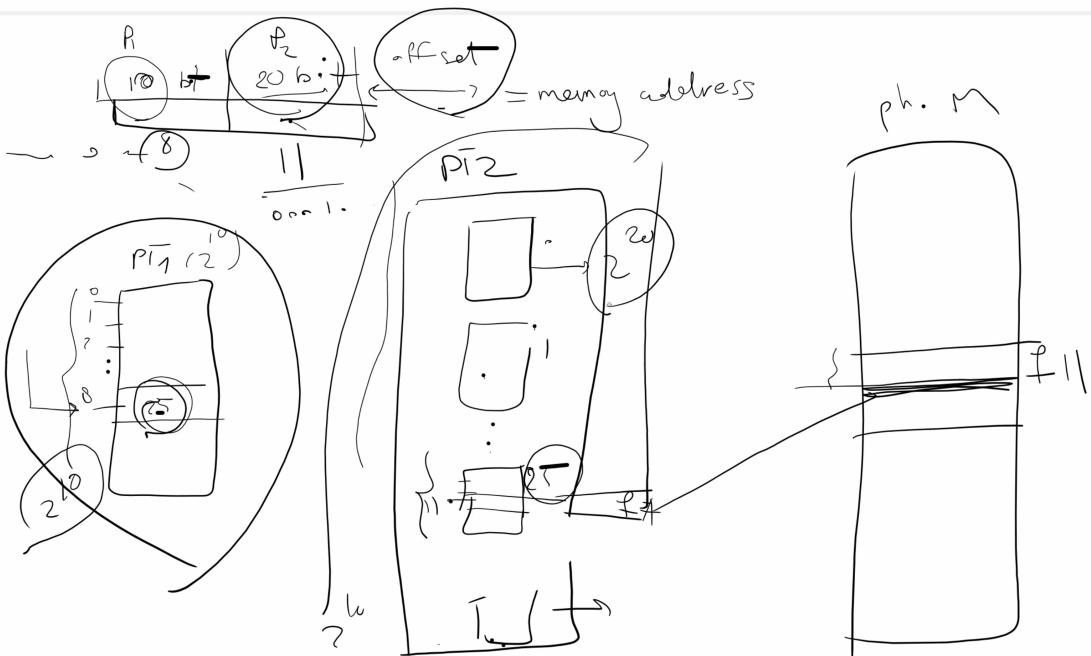
Non abbiamo una singola page table che fa riferimento a memoria fisica, ma abbiamo diversi livelli di page table.

In pratica andiamo a dividere la page table stessa in pagine, e quindi avremo un'ulteriore page table "esterna" che mappa pagine (della page table "interna") con i frames in cui si trovano.

Se usiamo un meccanismo del genere, l'operazione di MMU che facevamo prima per trovare i dati della pagina non è più funzionante: a questo punto dobbiamo "attraversare" un ulteriore livello di paging.



L'offset rimane sempre come prima perchè è sempre di quanto ci dobbiamo spostare nella pagina (di livello 2) in cui andiamo a trovare effettivamente i dati. La parte restante la dividiamo in 2: pagina di livello 1, e pagina di livello 2. La prima parte dell'indirizzo logico rappresenta, ad esempio con 10 bit, il primo livello di gerarchia: cioè tramite questi bit andiamo a trovare nella page table, con 2^{10} entries, il **page number** della page table di livello 2, che contiene a sua volta un mapping page-frame.



Il vantaggio è che, anche se entrambe hanno bisogno di spazio contiguo per essere allocate, ciascuna ne necessita "di meno", perché se dovessimo allocare una singola page table da 32 bit avremmo bisogno di molto spazio (come dicevamo nella slide di prima).

Quindi dividiamo in 2 cosicchè ciascuna necessiti di meno spazio contiguo.

Con 2 livelli di page tables miglioriamo l'occupazione in memoria della page table però, se abbiamo TLB miss abbiamo peggiorato le performances perchè dobbiamo fare 3 accessi alla memoria invece di 1.

Se un processo non ha bisogno di tanto spazio per la sua page table, in uno scenario del genere possiamo pensare che le sue pagine siano mappate direttamente nella page table di livello 2, senza passare dalla page table di livello 1. Come possiamo indicare una cosa del genere alla CPU?

Potremmo riservare ad esempio il primo "indirizzo" di page table di livello 1 (tipo 0x00000000), e a quel page number iniziale della page table di livello 1 troviamo le page tables dei processi che hanno bisogno di poco spazio.

Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a page number consisting of 20 bits
 - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
 - a 10-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:

page number	page offset
p_1	p_2
10	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page in the inner page table
- Known as [forward-mapped page table](#)

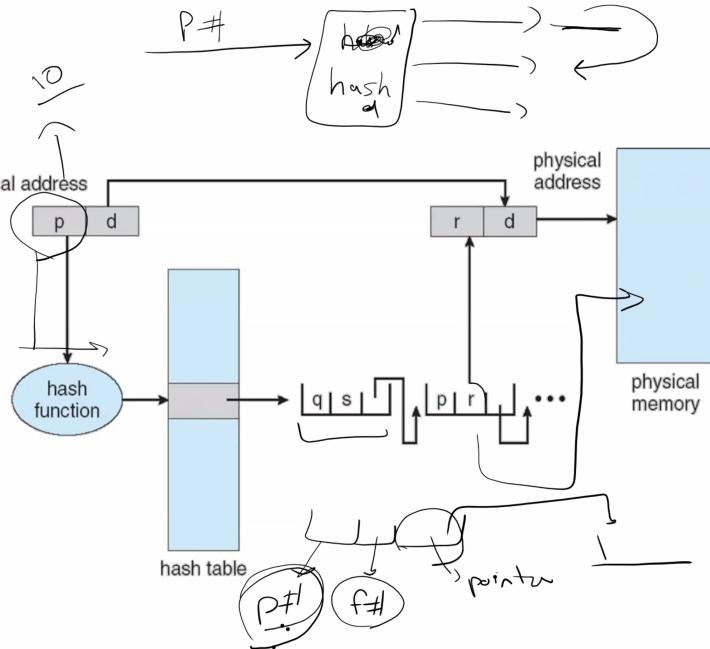
Hashed Page Tables

Hashed Page Tables

- One approach for handling address spaces larger than 32 bits is to use a [hashed page table](#), with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location.
- Each element consists of three fields:
 - 1) The virtual page number
 - 2) the value of the mapped page frame
 - 3) a pointer to the next element in the linked list

Hashed Page Tables

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame is used to form the physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



Il page number è l'input della funzione di hash, tramite cui vorremmo mappare la page al frame.

In questa tecnica, ogni entry della hash table è una linked list di mapping page-frame in cui ogni page number genera lo stesso hash.

È come se stessimo memorizzando in una singola entry più di un mapping.

Dunque il page number dell'indirizzo logico è l'input della funzione hash, che è implementata tramite un certo hardware, e in questo caso genera 3 dati in output: page number, frame number, e un puntatore all'elemento successivo.

Si scorre quindi la lista (grazie ai puntatori) fino a trovare il match con il page number che stiamo esaminando, per trovare quindi il corrispondente frame number.

Inverted Page Table

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
 - TLB can accelerate access
- But how to implement shared memory?
 - One mapping of a virtual address to the shared physical address

Finora abbiamo visto che ogni processo ha una sua page table.

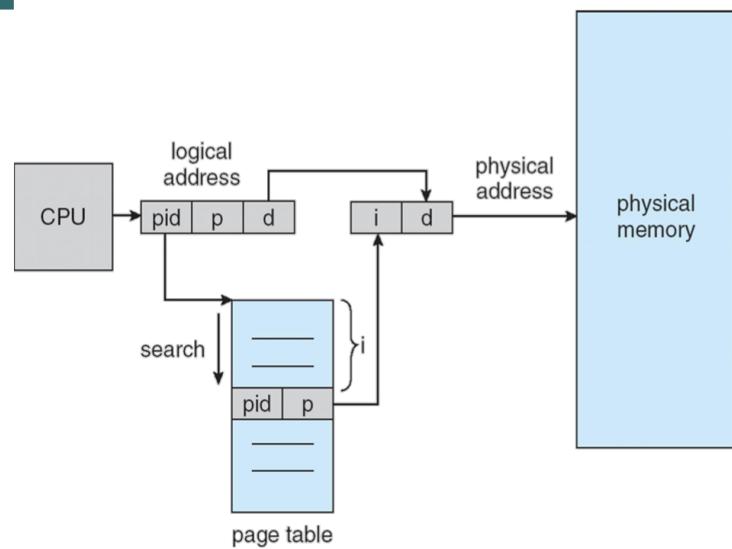
Inverted Page Table funziona al contrario: non iniziamo dalla pagina, ma dal frame! Quindi memorizziamo il mapping frame-page, e vediamo ciascun frame quale pagina ha allocato.

Quindi a sto giro la “page table” è in

condivisione tra tutti i processi (perchè la memoria fisica è una, e noi mappiamo i frame fisici con le pagine che allocano, che possono essere di qualsiasi processo).

Il vantaggio è che abbiamo una sola page table, quindi non una per processo → meno spazio occupato.

Inverted Page Table Architecture



Ora però, essendo unica la page table e accessibile da tutti i processi, dobbiamo aggiungere informazioni per capire a quale processo appartiene una certa pagina allocata in un frame.

Possiamo usare il PID, che aggiungiamo alle informazioni da memorizzare nella page table.

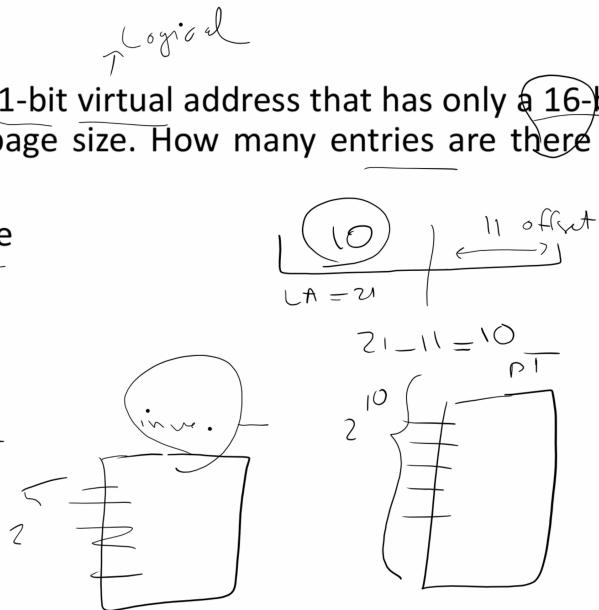
Il problema è che dobbiamo fare una ricerca nella page table, e quindi possibilmente abbiamo tanti accessi a memoria → performance peggiori.

Esercizio

Exercise

- Consider a computer system with a 21-bit virtual address that has only a 16-bit physical address. It also has a 2KB page size. How many entries are there in each of the following?
- A conventional, single-level page table
- An inverted page table

$$\begin{aligned}
 VA &\rightarrow 21 \\
 PA &\rightarrow 16 \\
 \text{page size} = 2\text{KB} &\Rightarrow 2 \times 2^10 \rightarrow 11 \text{ bits} \\
 \underline{5} & \quad \underline{11} \quad \underline{2} \\
 \text{ph. } PA &= 16
 \end{aligned}$$

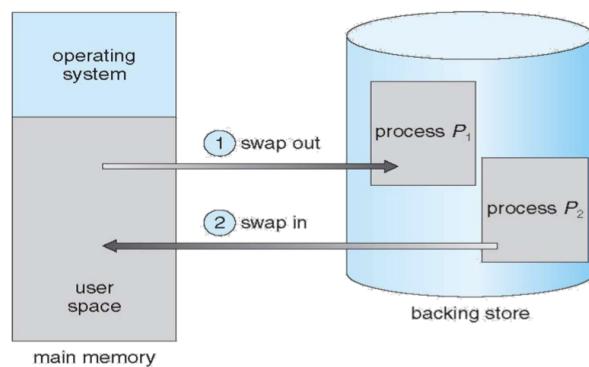


Swapping

Swapping

- Process instructions and data for operating must be in memory to be executed.
- However, a process or a portion of a process can be **swapped** temporarily out of memory to a backing store and then brought back into memory for continued execution.

- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system.



Può capitare che durante l'esecuzione di un processo, alcune pagine allocate per quel processo debbano essere sostituite, in memoria, da pagine appartenenti ad altri processi (perchè c'è bisogno di spazio).

Quindi tali pagine che vengono sostituite vengono riportate su disco finché poi non sono nuovamente necessarie e vengono riportate in RAM.

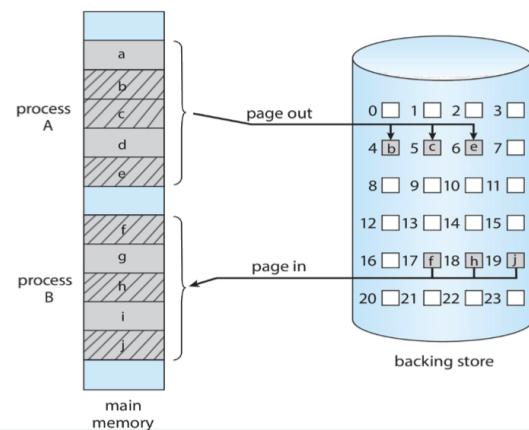
Standard Swapping

- Standard swapping involves moving entire processes between main memory and a backing store.
- The backing store must be large enough to accommodate whatever parts of the process needs to be stored and retrieved.
- When a process is swapped to the backing store, the data structures associated with the process must be written to the backing store.
- The advantage is that it allows physical memory to be oversubscribed, so the system can accommodate more processes than there is actual physical memory.

Swapping with Paging

- A problem with standard swapping is that the amount of time required to move entire process between memory and the backing store is expensive.
- Therefore, instead of the entire process, pages of a process can be swapped.

- A **page out** operation moves a page from a memory to the backing store.
- The reverse process is known as **page in**



Swapping on mobile systems

- Typically, mobile systems do not support swapping.
 - Mobile devices generally use flash memory rather than more spacious hard disks with limited number of writes that it can tolerate before it becomes reliable.
- Instead of swapping, when free memory falls below threshold:
 - Apple iOS asks application to voluntarily relinquish allocated memory.
 - The application that fails to free up sufficient memory may be terminated by the operating system.
 - Android may terminate a process if insufficient free memory is available. However, before terminating, android writes its application state to flash memory so it can be quickly restarted.