

# Sistemi Operativi

**C.d.L. in Informatica (laurea triennale)**

Anno Accademico 2019-2020

Canale A-L

Dipartimento di Matematica e Informatica – Catania

## **Gestione della Memoria**

Prof. Mario Di Raimondo

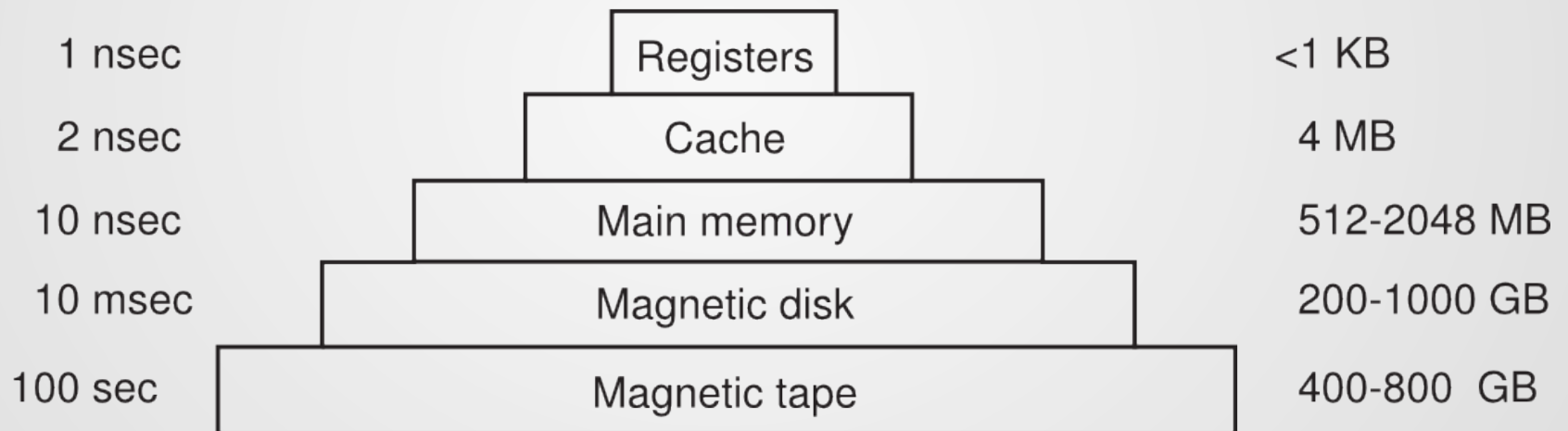
# Memoria centrale e processi

- **Gerarchia di memoria;**

- in particolare la memoria centrale (formata da **RAM**) rappresenta il livello più basso direttamente utilizzabile dalla CPU;

Typical access time

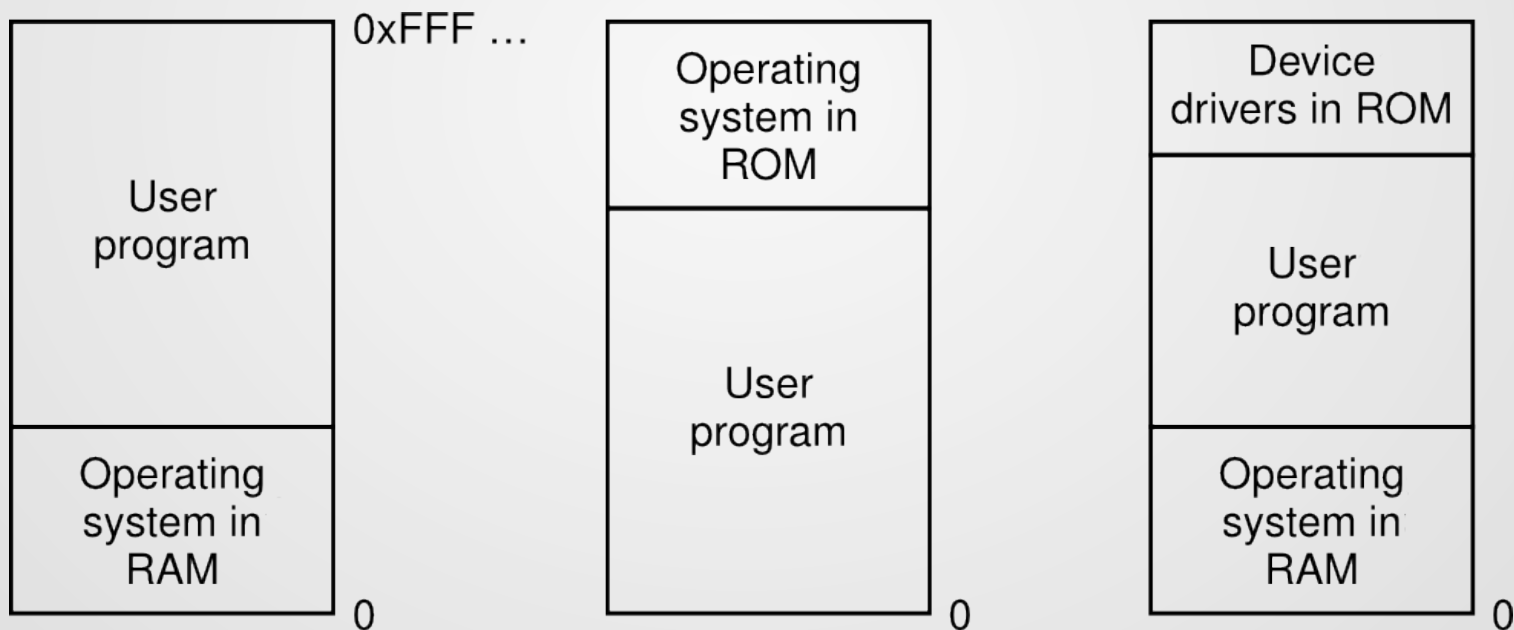
Typical capacity



- **astrazione della memoria** a favore dei processi;
  - vari livelli di astrazione (via via più complessi).

# Senza alcuna astrazione

- Modello usato sui primi mainframe (anni '60) e sui primi PC (primi anni '80);
- i programmi utilizzano direttamente gli **indirizzi fisici**;
- difficile eseguire due programmi contemporaneamente;
- vari **modelli**:

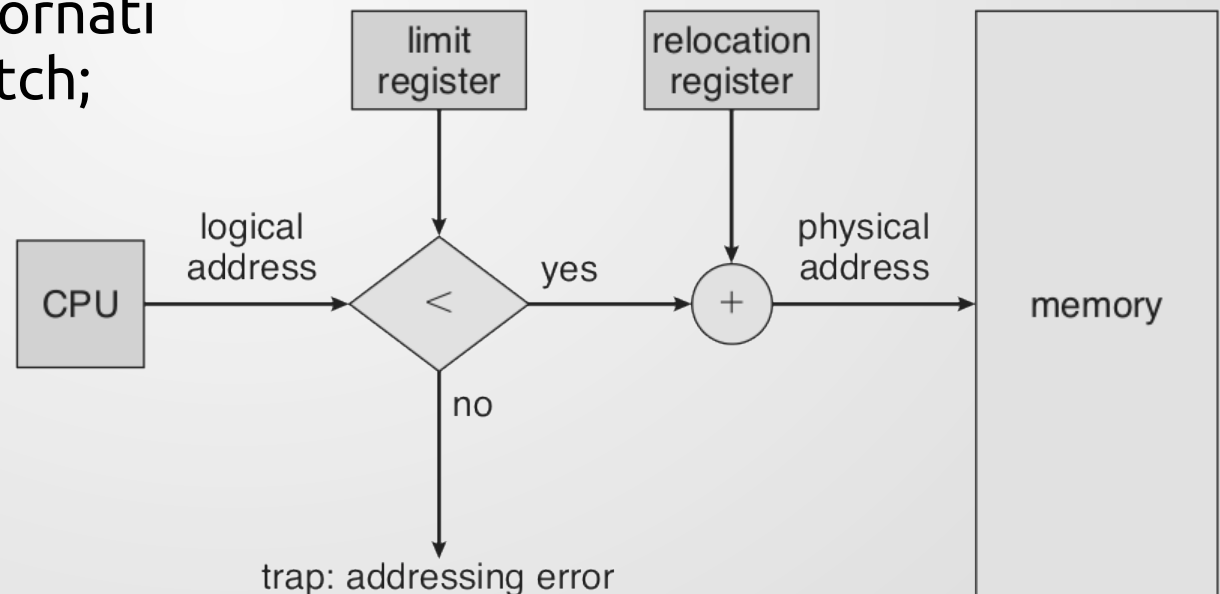


# Multiprogrammazione senza astrazione

- Manteniamo **più processi in memoria**;
- problemi:
  - **rilocazione**:
    - **rilocazione a compile-time**;
    - **rilocazione statica** in fase di caricamento:
      - rallentamento del loader;
  - **protezione della memoria**:
    - **lock & key**: blocchi di memoria con delle chiavi di protezione e PSW con la chiave del processo in esecuzione.

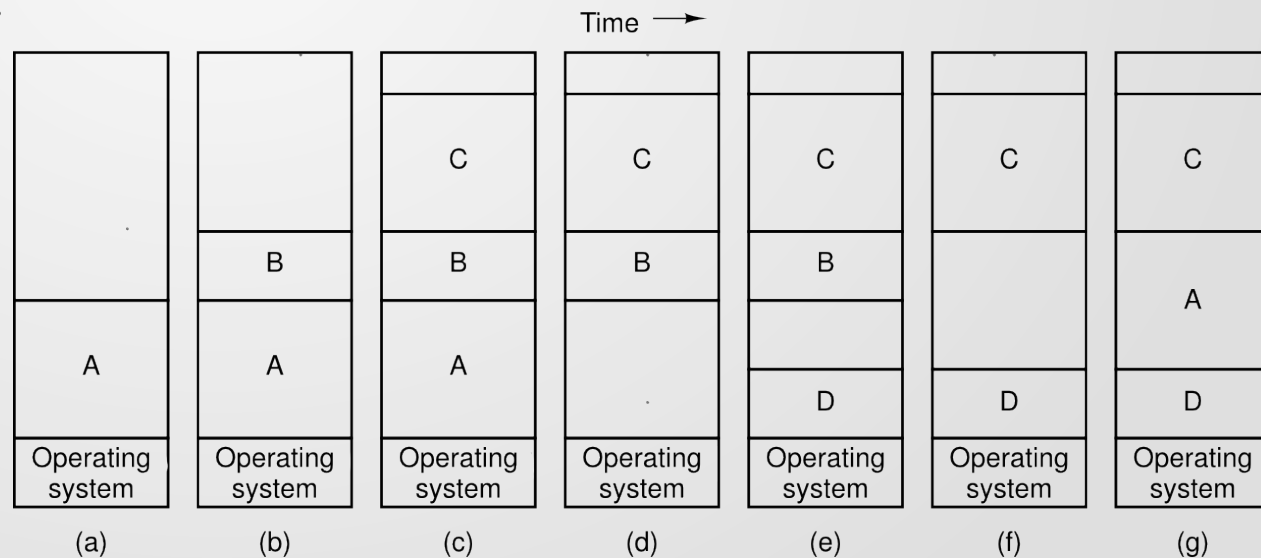
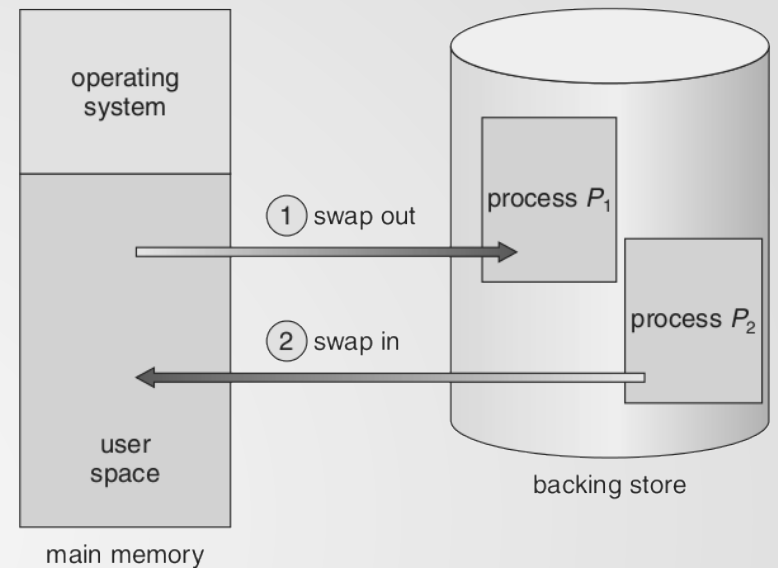
# Spazio degli indirizzi

- **Spazio degli indirizzi:** una astrazione per la memoria utilizzabile da un processo;
- **rilocazione dinamica con registro base e registro limite;**
  - la **CPU controlla** gli accessi alla memoria in base ai registri;
    - nei sistemi più evoluti questo è fatto dalla **Memory Management Unit (MMU)**;
  - i registri vanno aggiornati ad ogni context-switch;
  - usata su:
    - CDC 6600;
    - Intel 8088.



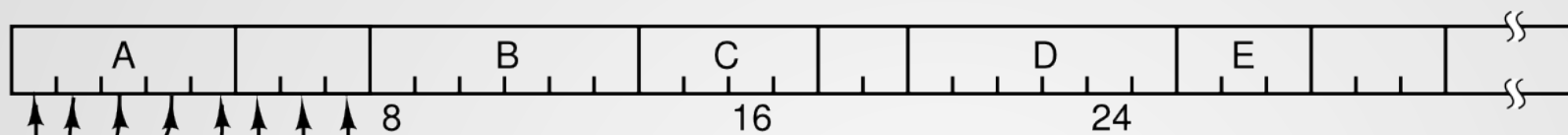
# Swapping

- Il **livello di multiprogrammazione** è seriamente limitato dalla dimensione della memoria centrale;
- prima soluzione: **swapping**;
  - **swapper** (scheduler di medio termine);
  - problemi con **I/O pendenti**;
- strategie di allocazione:
  - **dimensione fissa**;
  - **dimensione dinamica**;
- **frammentazione**:
  - **interna**;
  - **esterna**;
- **memory compaction**.



# Gestione dell'allocazione

- Due metodologie:
  - **bitmap;**
    - dimensione del blocco;



- **liste dei blocchi liberi e occupati;**

- ordinata per indirizzo; 

P	0	5	-
---	---	---	---

 → 

H	5	3	-
---	---	---	---

 → 

P	8	6	-
---	---	---	---

 → 

P	14	4	-
---	----	---	---
- coalescenza; 

H	18	2	-
---	----	---	---

 → 

P	20	6	-
---	----	---	---

 → 

P	26	3	-
---	----	---	---

 → 

H	29	3	X
---	----	---	---
- doppiamente collegata;
- algoritmi di ricerca in allocazione:
  - first fit, next fit, best fit, worst fit;
- liste separate;
  - lista blocchi liberi ordinata per dimensione.

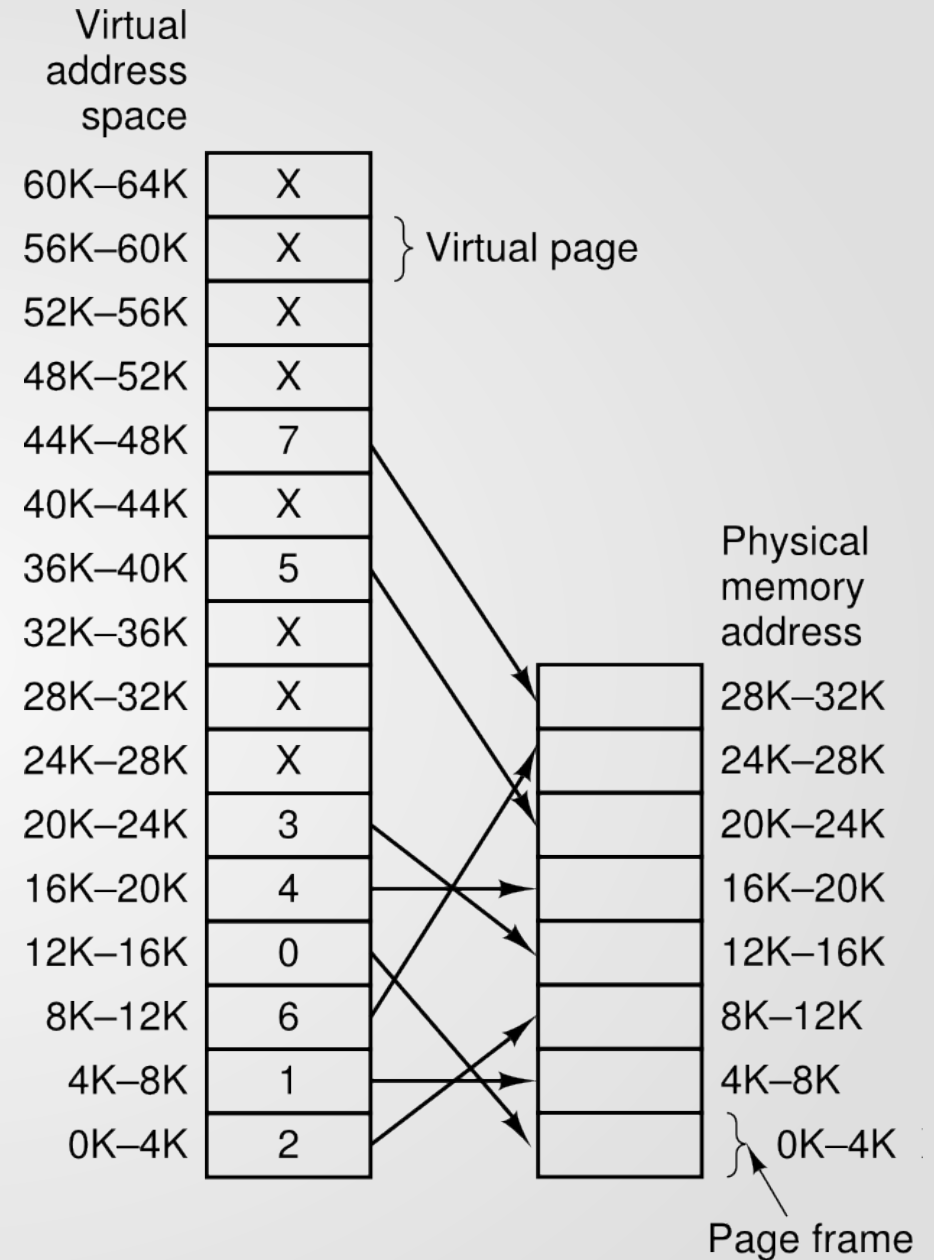
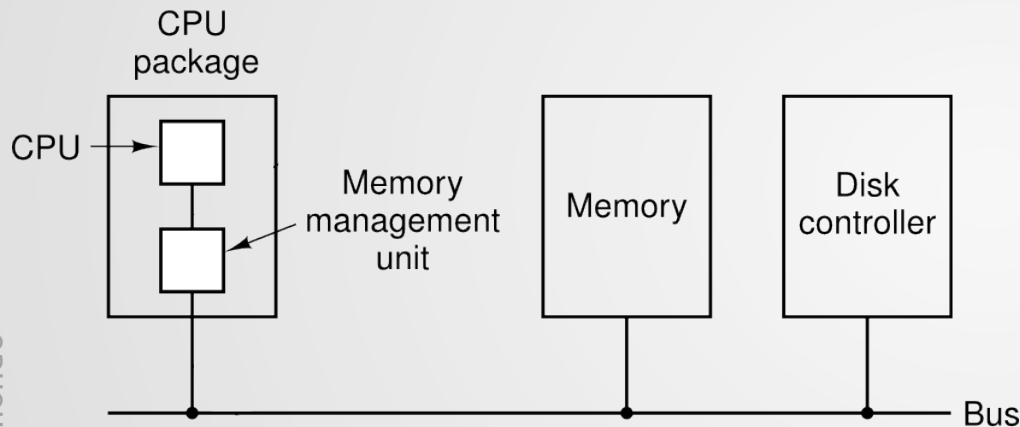
# Memoria virtuale

- Si passa all'**allocazione non contigua** della memoria fisica;
  - spazio di indirizzamento virtuale diviso in **pagine**;
  - spazio della memoria fisica diviso in **frame**;
  - **dimensione** delle pagine/frame:
    - frammentazione interna;
    - in alcuni casi è variabile (Solaris);
  - alcune pagine possono anche non risiedere in memoria;
    - all'occorrenza vengono caricate dalla **memoria secondaria**;
    - si adatta perfettamente ad un **sistema multiprogrammato**;
  - **protezione implicita** tra processi.



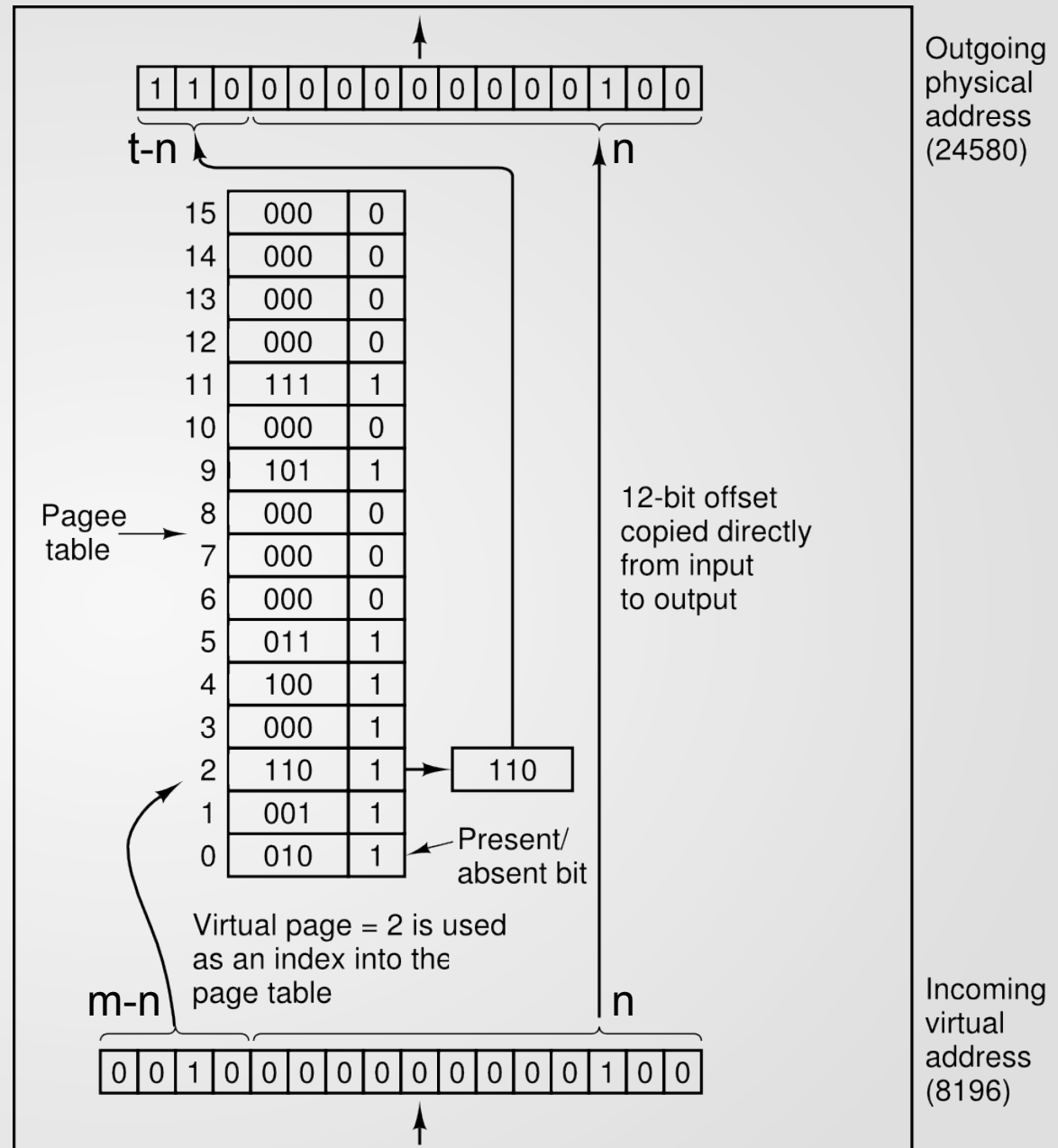
# Paginazione

- Gestita dalla **MMU**;
  - **bit di presenza**;
  - errore di pagina (**page fault**).

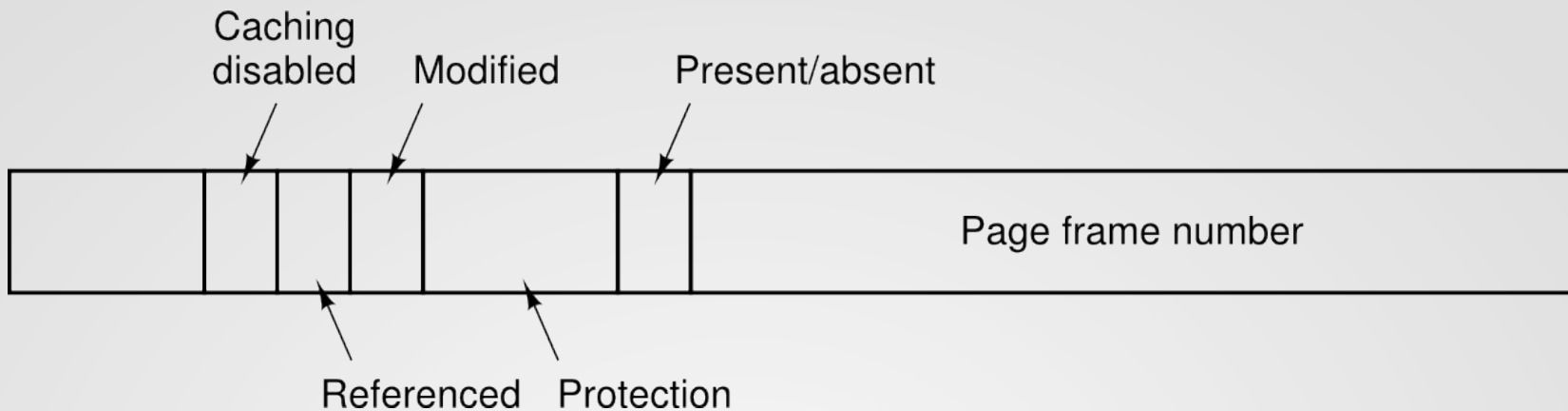


# Uso di una tabella delle pagine

- spazio indirizzi virtuali:  $2^m$
- dim. pagina:  $2^n$ 
  - numero di pagina:  $(m-n)$  bit più significativi dell'indirizzo virtuale
  - $2^{m-n}$  pagine
  - offset:  $n$  bit meno significativi
- spazio indirizzi fisici:  $2^t$ 
  - $2^{t-n}$  frame
  - $m > t$



# Dettaglio su una voce della tabella delle pagine



- **Numero del frame;**
- **bit presente/assente;**
- **protezione:** lettura/scrittura ed eventualmente esecuzione;
- **bit modificato (dirty bit):** indica se la pagina è stata modificata;
- **bit referenziato:** impostato quando si fa un riferimento qualunque alla pagina;
- bit per disabilitare la **cache;**
- nella pratica esiste anche un **bit di validità** (o di allocazione).

# Tabella dei frame

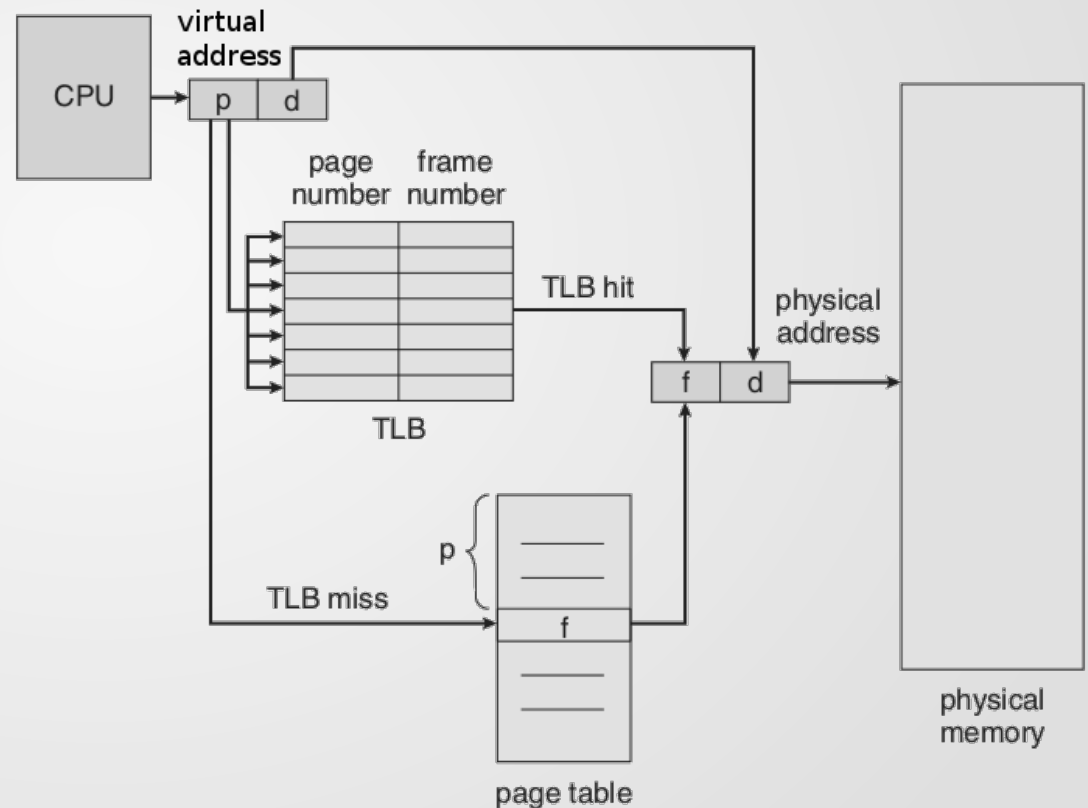
- Il S.O. tiene traccia dello stato di occupazione di ogni frame fisico attraverso la **tabella dei frame**;
  - stato: **occupato / libero**;
  - se occupato: **da quale processo?**
- viene consultata:
  - ogni volta che viene creato un **nuovo processo** per creare la relativa tabella delle pagine di quel processo;
  - ogni volta che un processo chiede di allocare **nuove pagine**.

# Progettazione di una tabella delle pagine

- Sono due gli aspetti principali da curare:
  - **velocità** nella consultazione;
  - **dimensione**.
- Affrontiamo per ora il fattore **velocità**:
  - due possibili idee di implementazione:
    - avere un numero sufficiente di registri su cui caricare l'intera tabella;
    - tabella interamente residente in memoria con registro **PTBR (Page-Table Base Register)**;
      - servono **due accessi** alla memoria per prelevare un dato dalla memoria;
      - context switch molto veloce.

# Uso di memoria associativa

- **Translation Lookaside Buffer (TLB)** dentro la MMU, dette anche **Memorie (o registri) associative**;
- **osservazione** di partenza: in genere un programma usa un gran numero di riferimenti ad un piccolo numero di pagine;
- un numero ridotto di registri (tra 64 e 1024), con le seguenti **voci**:
  - numero di pagina virtuale;
  - bit per validità della voce della TLB;
  - codice di protezione;
  - dirty bit;
  - numero di frame.
- **ricerca parallelizzata** in hardware;
- possibilità di **voci vincolate**;
- **TLB miss vs. TLB hit**;
- uso di **address-space identifiers (ASID)** vs. **flush** della TLB.



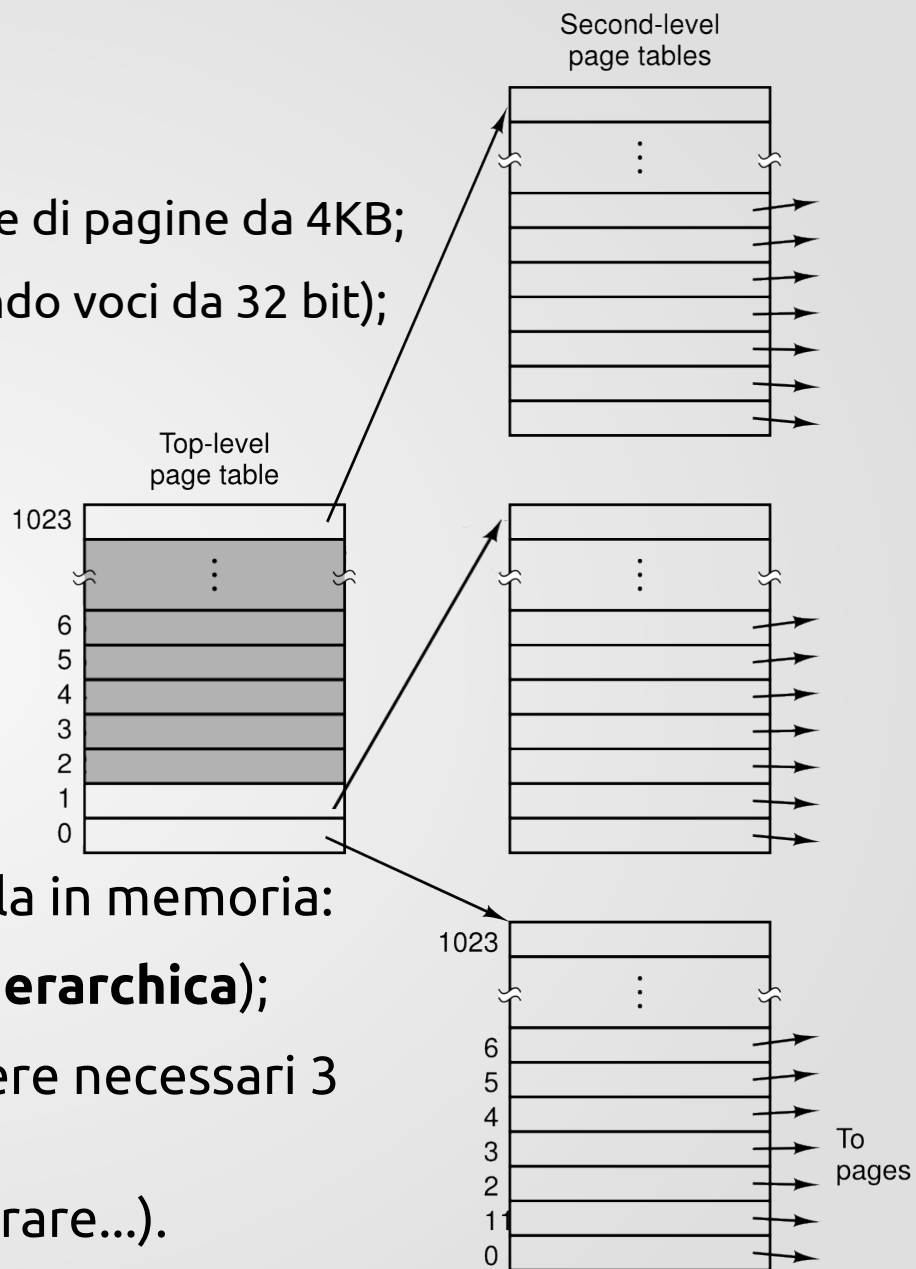
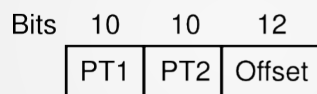
# Effective Access Time (EAT)

- Facciamo un **esempio**:
  - tempo di accesso alla memoria = 100 nsec;
  - tempo di accesso alla TLB = 20 nsec;
- **tempo effettivo di accesso** sarà in questo caso:
  - 120 nsec per TLB hit;
  - 220 nsec per TLB miss;
- ipotizziamo un TLB ratio (percentuale di successi) dell'80%;
  - tempo (medio) effettivo di accesso:  $0.8 \times 120 + 0.2 \times 220 = 140$  nsec
- in generale:
  - **tempo di accesso alla memoria**:  $\alpha$
  - **tempo di accesso alla TLB**:  $\beta$
  - **TLB ratio**:  $\varepsilon$ 
    - **EAT** =  $\varepsilon (\alpha + \beta) + (1 - \varepsilon) (2\alpha + \beta)$

# Tabella delle pagine multilivello

- Resta il **problema delle dimensioni**:

- indirizzi virt. a 32 bit (Pentium): 1 milione di pagine da 4KB;
  - tabella grande circa 4MB (supponendo voci da 32 bit);
  - necessita di memoria contigua;
- indirizzi virt. a 64 bit:  $2^{52}$  pagine da 4KB;



- soluzione: non mantenere l'intera tabella in memoria:

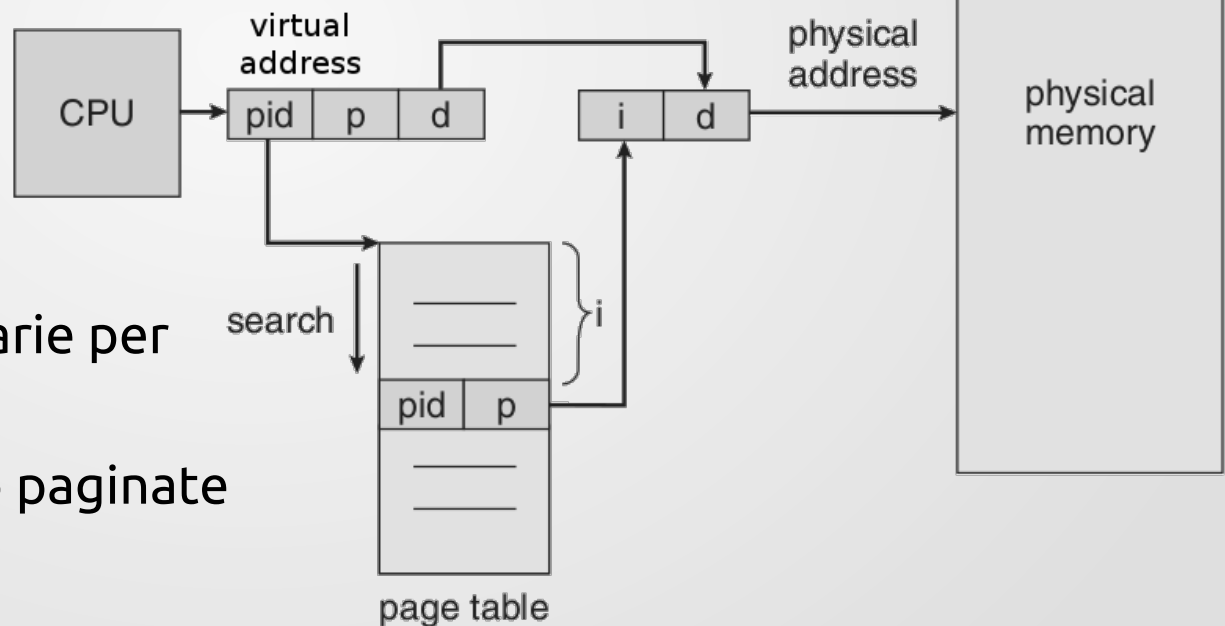
- **tabelle multilivello (paginazione gerarchica);**
  - nel caso di 2 livelli, possono essere necessari 3 accessi alla memoria;
- anche su **più livelli** (ma senza esagerare...).



# Tabella delle pagine invertita

- Una voce per ogni frame fisico;
  - ogni voce riporta: **(id processo, pagina virtuale)**;
- tabella alquanto piccola a paragone delle t.p.;
  - questa implicherebbe una ricerca molto lenta:
    - **tabella hash** indicizzata sull'indirizzo virtuale;
    - accoppiata con una **TLB**.

- servono ancora le **tabelle per processo**:
  - informazioni necessarie per gestire i page fault
  - possono essere però paginate a loro volta.

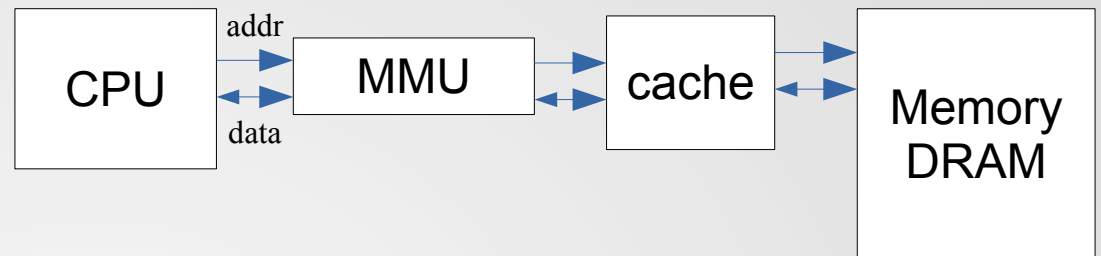


# Cache della memoria vs. Memoria virtuale

- La **cache della memoria** può essere:

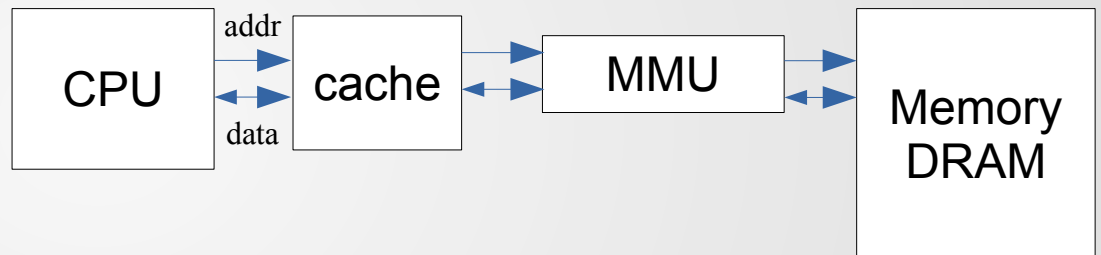
- **basata sugli indirizzi fisici:**

- ➔ non serve invalidarla sul context-switch;
- ➔ caching poco efficace;



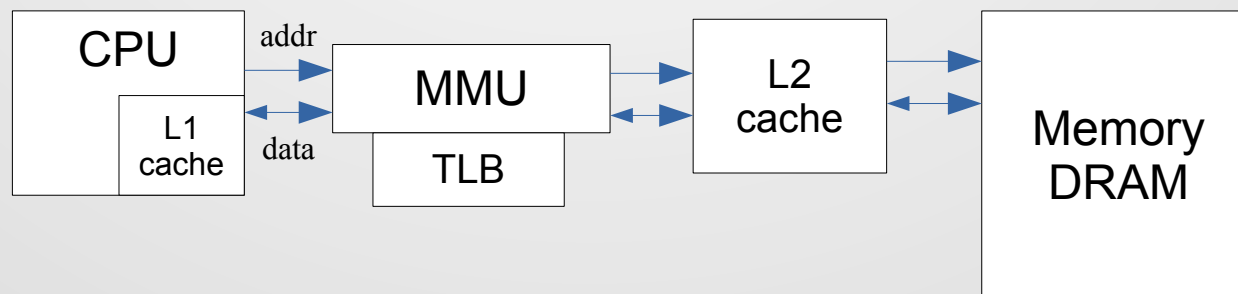
- **basata sugli indirizzi virtuali:**

- ➔ servono gli ASID per non invalidarla ogni volta;
  - scala male su L2;
- ➔ maggiore efficacia.



- Cosa si usa **in pratica?**

- cache L1 basata su indirizzi virtuali;
- cache L2 e successive basate su indirizzi fisici.



# Algoritmi di sostituzione delle pagine

- In caso di **page fault** ed in assenza di frame liberi è necessario scegliere una **pagina vittima**;
  - **come sceglierla?**
    - problema simile alla **gestione delle cache**;
    - **obiettivo**: minimizzare il numero di page fault in futuro;
- **soluzione ottimale** (algoritmo **OPT**):
  - scegliamo la pagina che verrà referenziata in un futuro più lontano;
  - ottimale ma **difficilmente realizzabile**;
  - rappresenta comunque un **termine di paragone**.

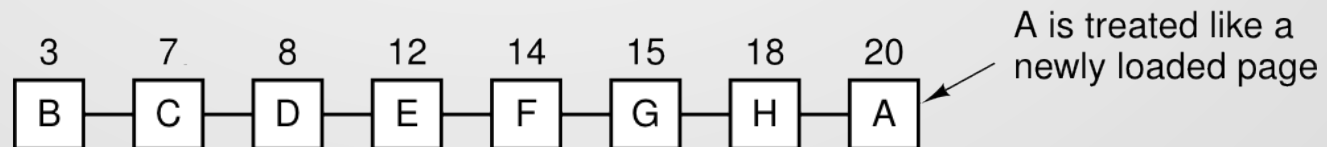
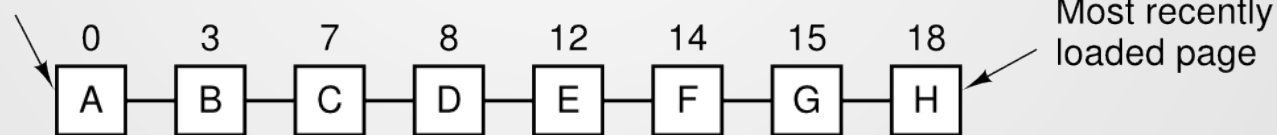
# Algoritmo NRU

- Raccogliamo un po' di **statistiche** sull'uso della pagine caricate:
  - bit di **referenziamento** (R) e di **modifica** (M);
    - aggiornati tipicamente in **hardware**;
    - **azzerati** dal S.O.;
      - bit di referenziamento azzerato **periodicamente**;
- algoritmo **Not Recently Used** (NRU):
  - distinguiamo **4 classi di pagine**:
    - **classe 0**: non referenziato, non modificato;
    - **classe 1**: non referenziato, modificato;
    - **classe 2**: referenziato, non modificato;
    - **classe 3**: referenziato, modificato;
  - viene scelta una pagina dalla classe non vuota di numero più basso.

# Algoritmo FIFO e della seconda chance

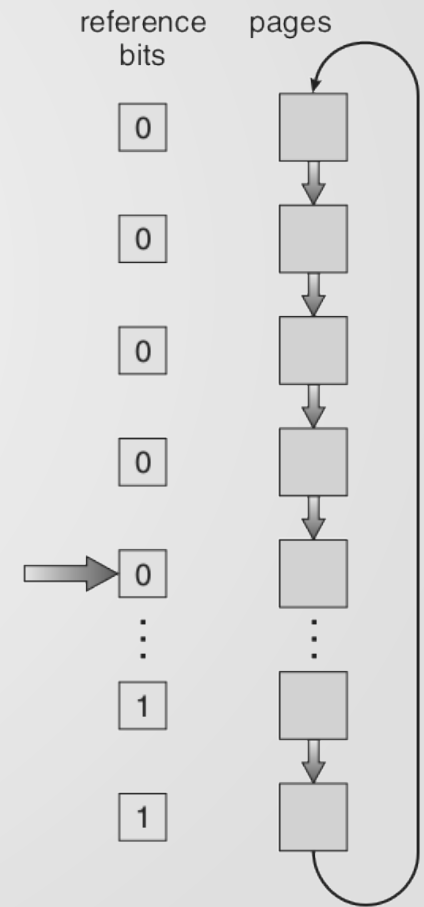
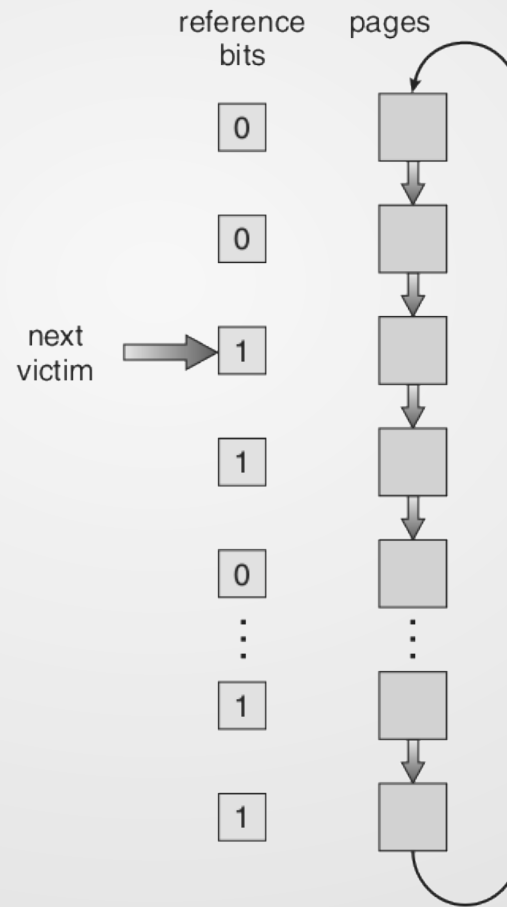
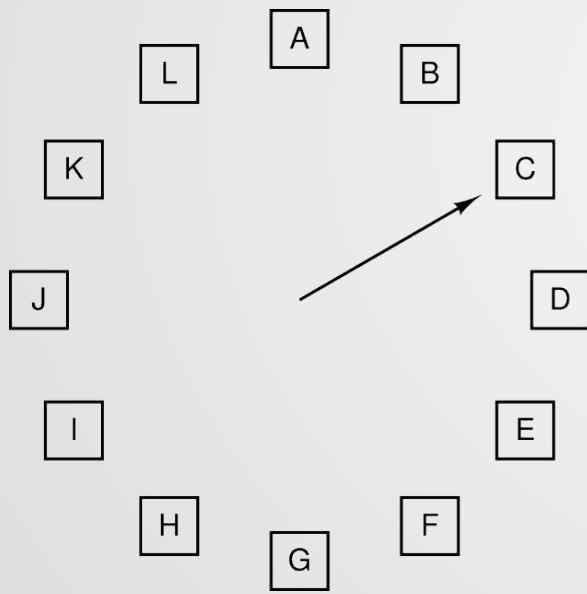
- Algoritmo **First-In First-Out (FIFO)**:
  - viene rimossa la **pagina più vecchia**;
  - scelta non sempre felice: può rimuovere pagine, sì vecchie, ma magari molto usate;
- algoritmo della **Seconda Chance**:
  - si tiene conto dell'attuale stato del **bit R**;
  - viene rimossa la pagina più vecchia se non usata di recente.

Page loaded first



# Algoritmo Clock

- L'idea dell'algoritmo della seconda chance è buona ma si può implementare in modo più efficiente:
  - algoritmo dell'orologio (**clock**).



# Algoritmo LRU

- Probabilmente le pagine più usate di recente lo saranno anche in futuro;
  - **idea:** rimuovere le pagine meno usate di recente;
- algoritmo **Least Recently Used** (LRU):
  - buona idea ma non semplice e dispendiosa da implementare:
    - con **supporto hardware:**
      - **contatore** nella CPU e campi relativi nella tabella delle pagine;
      - tramite una **matrice** di bit:

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

→frame 0

	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

→frame 1

	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	1	1	0	1
3	0	0	0	0

→frame 2

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

→frame 3

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	1
3	1	1	0	0

→frame 2

0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

→frame 1

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

→frame 0

0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0

→frame 3

0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0

→frame 2

0	1	0	0
0	0	0	0
1	1	0	0
1	1	1	0

→frame 3

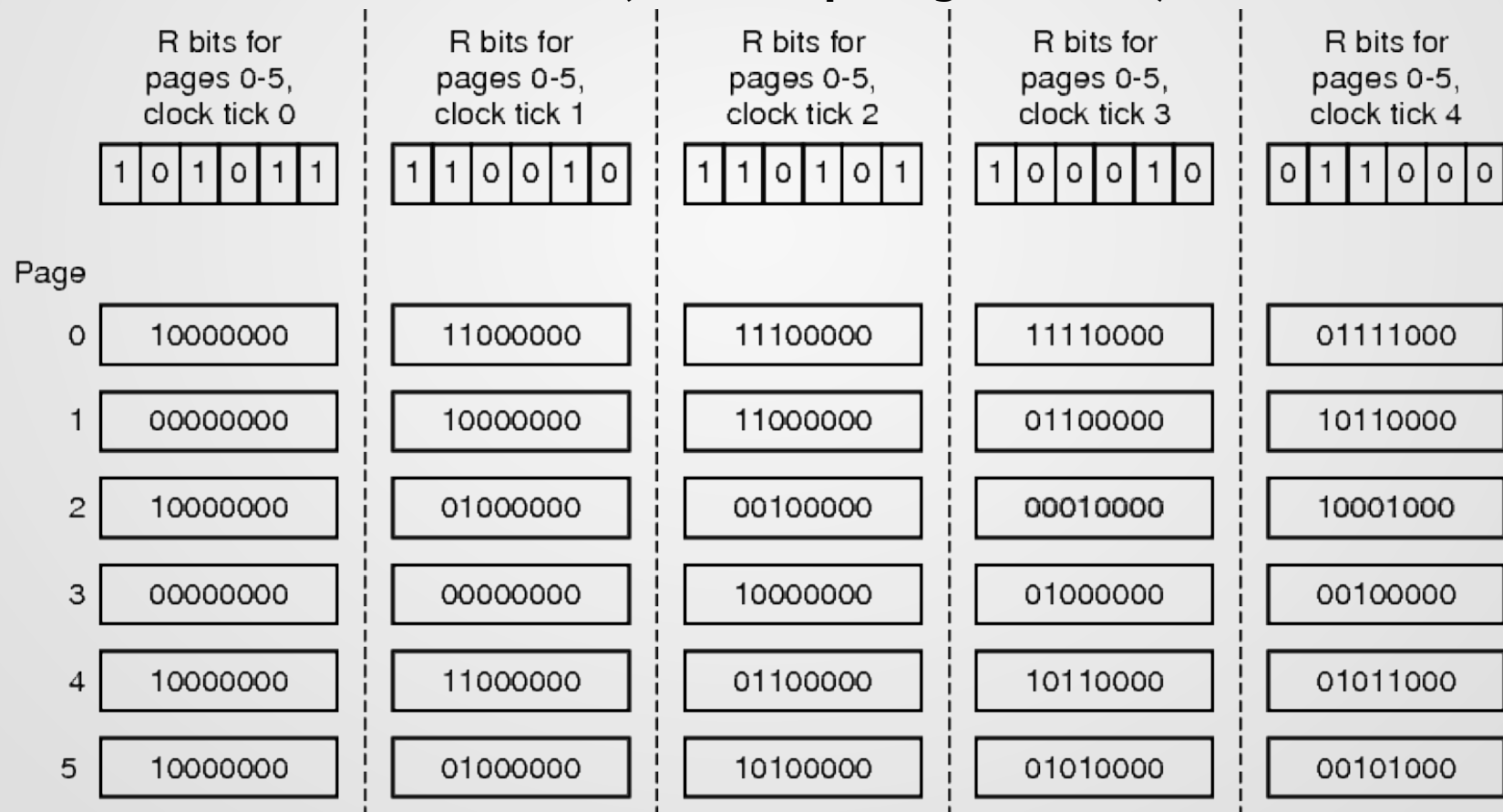
# Algoritmo NFU

- LRU, anche se implementato in hardware, è dispendioso:
  - pensiamo ad una sua approssimazione;
- algoritmo **Not Frequently Used** (NFU):
  - si tratta di una approssimazione (via software) dell'algoritmo LRU;
  - un **contatore** in ogni voce della tabella delle pagine;
  - periodicamente il valore del **bit R**, prima di essere azzerato, viene sommato a tale contatore;
  - viene rimossa la pagina con il **contatore più basso**;
- **problema:** può erroneamente privilegiare pagine che sono state molto utilizzate in passato ma che invece sono scarsamente usate di recente: queste lo saranno, probabilmente, anche nel prossimo futuro.



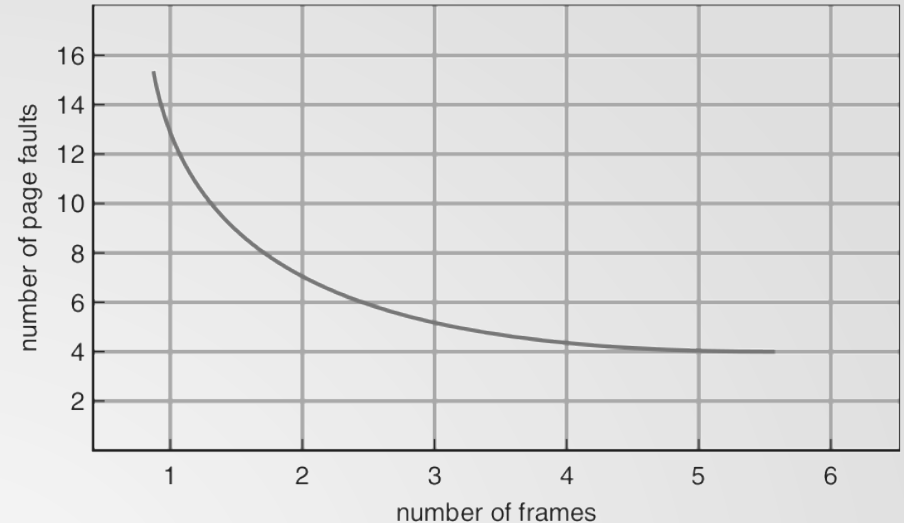
# Algoritmo di Aging

- Algoritmo di **Aging**:
  - ad ogni scadenza del clock:
    - **shift a destra** del contatore associato ad ogni pagina;
    - accostamento a sinistra (**come bit più significativo**) del bit R.



# Confronto delle prestazioni

- Scegliamo una **metrica**:
  - numero di fault di pagina;
- a parità di condizioni, fissiamo:
  - ♦ **numero di frame**: 3 frame
  - ♦ sequenza degli indirizzi virtuali a cui accedere;
    - in modo equivalente: **sequenza compatta delle pagine**;
    - **7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1**
- algoritmo **OPT**:



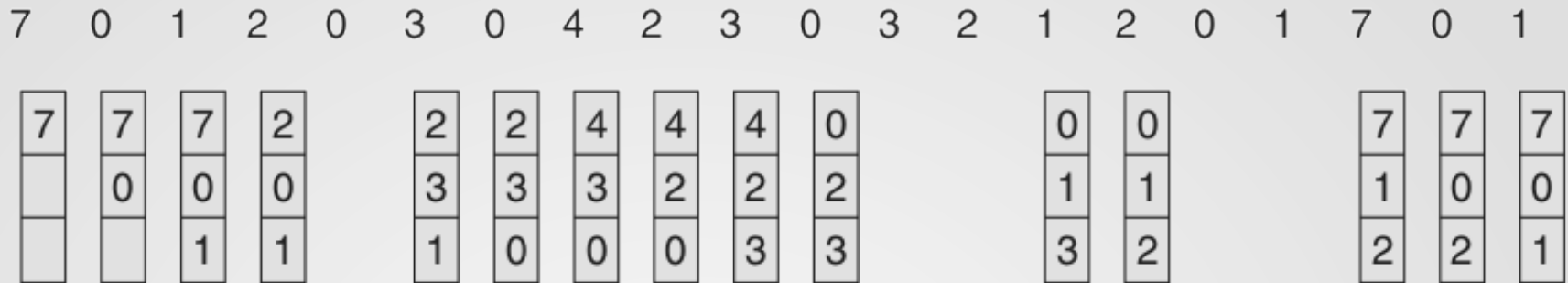
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



→ 9 fault di pagina

# Confronto delle prestazioni

- algoritmo **FIFO**:

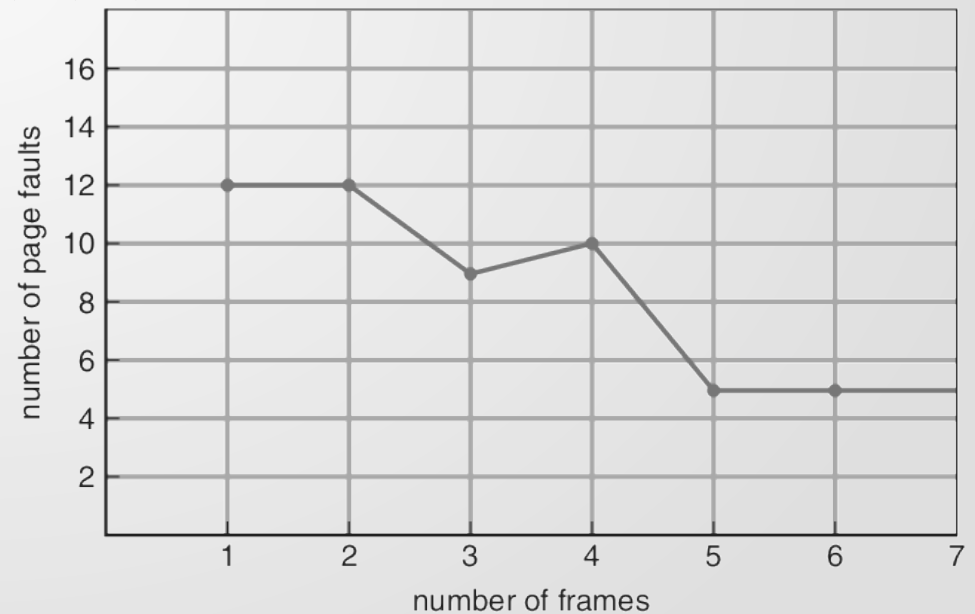


→ 15 fault di pagina

- ma c'è qualcosa di strano...

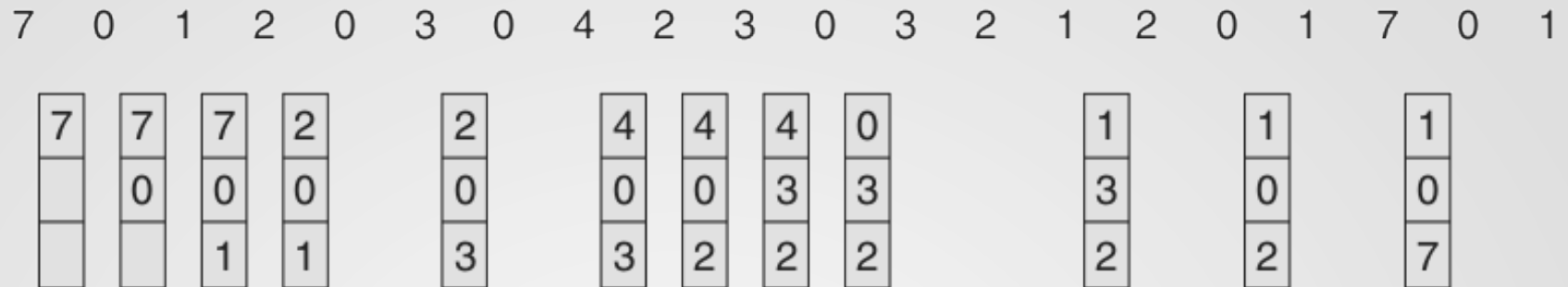
- sequenza: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **anomalia di Belady:**  
aumentando i frame disponibili, aumentano i fault di pagina (!?!)



# Confronto delle prestazioni

- algoritmo **LRU**:



→ 12 fault di pagina

- non soffre dell'anomalia precedente, infatti vale la
  - **proprietà di inclusione**: l'insieme di pagine caricate avendo  $n$  frame è incluso in quello che si avrebbe avendo  $n+1$  frame.
    - ♦  $B_t(n) \subseteq B_t(n+1) \quad \forall t, n$
- altri algoritmi visti:
  - **NFU, aging**: godono della proprietà di inclusione (appross. LRU);
  - **seconda chance, clock**: soffrono dell'anomalia (riducono a FIFO);
  - **NRU**: soffre dell'anomalia (riduce a FIFO).

# Riepilogo sugli algoritmi

- **OPT**: non implementabile, ma utile come termine di paragone;
- **NRU\***: approssimazione rozza dell'LRU;
- **FIFO\***: può portare all'eliminazione di pagine importanti;
- **Seconda chance\***: un netto miglioramento rispetto a FIFO;
- **Clock\***: come S.C. ma più efficiente;
- **LRU**: eccellente idea (vicina a quella ottima) ma difficilmente realizzabile se non in hardware;
- **NFU**: approssimazione software abbastanza rozza dell'LRU;
- **Aging**: buona approssimazione di LRU con implementazione software efficiente.

\* soffre dell'anomalia di Belady

# Allocazione dei frame

- **Paginazione su richiesta** (pure demand paging);
- quanti frame **assegnare** ad ogni processo?
  - **minimo:**
    - ♦ strutturale (set istruzioni, livelli di indirizzamento indiretto);
  - **massimo:** memoria libera;
- strategie:
  - allocazione **equa**;
  - allocazione **proporzionale:**
    - ♦ al processo  $i$  di dimensione  $s_i$ :
      - assegniamo ( $a_i = s_i / S \times m$ ) frame;
      - dove,  $S = \sum s_i$ ;
    - ♦ adeguamenti al livello di multiprogrammazione;
  - allocazione per **priorità**.

# Allocazione dei frame

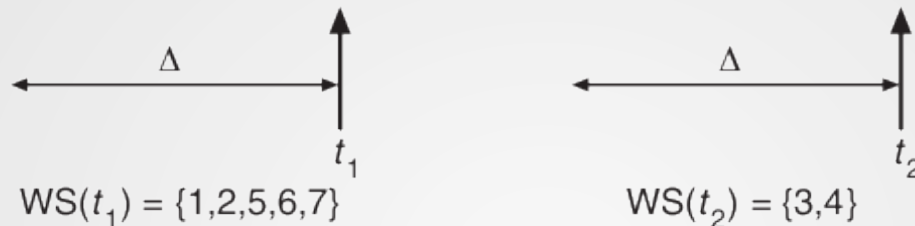
- Quali pagine considerare per la rimozione?
  - solo quelle dello stesso processo: **allocazione locale**;
  - tutte le pagine (anche di altri processi): **allocazione globale**;
- Cosa succede se ci sono **pochi frame** assegnati ad un processo?
  - sotto il **minimo strutturale**: viene sospeso e si fa swapping su disco;
  - un po' sopra:
    - può andare in **trashing**;
  - quando il trashing riguarda tutti i processi si parla di sistema in **sovraccarico** (eccessivo livello di multiprogrammazione);
- Bisognerebbe assegnare un numero di frame commisurato alle "**necessità** del processo":
  - **modello di località**;
  - concetto di **località**;
  - strategia di base.

# Working set

- Per ogni processo manteniamo un **working set**:
  - pagine usate negli ultimi  $\Delta$  accessi alla memoria;
  - scelta del **parametro**  $\Delta$  per adattarlo alla **località corrente**;

page reference table

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

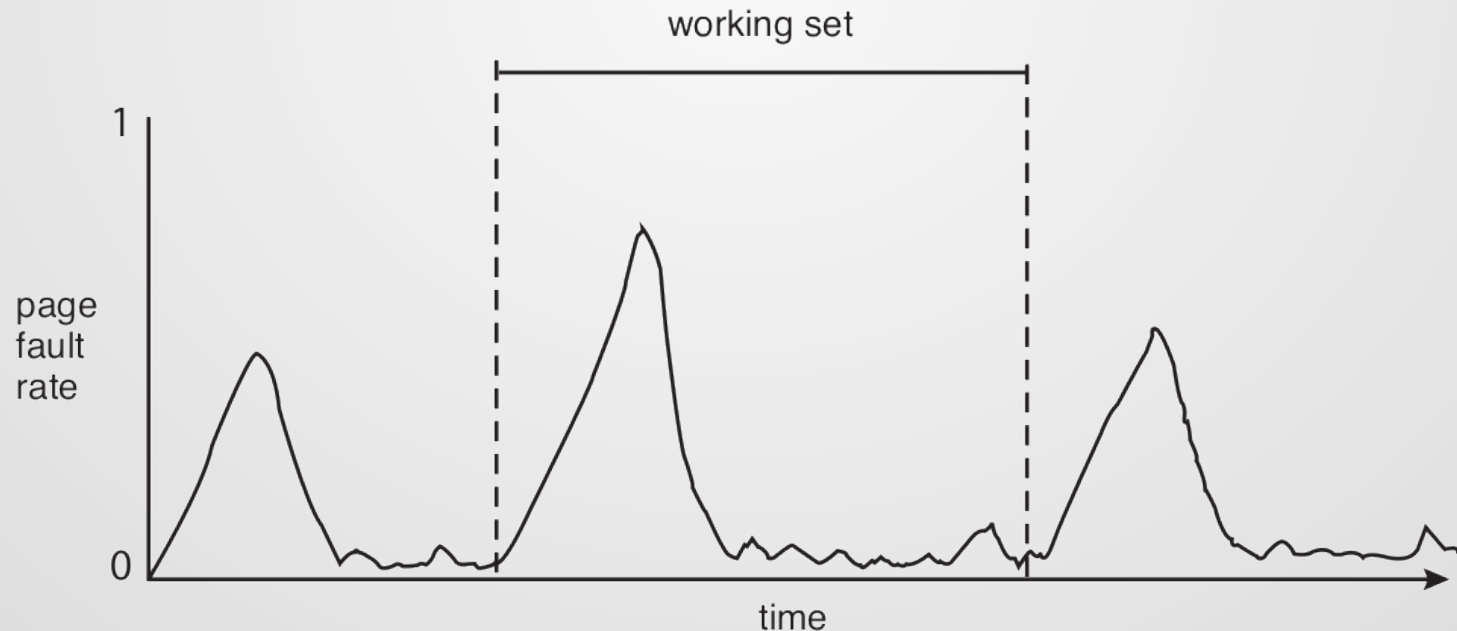
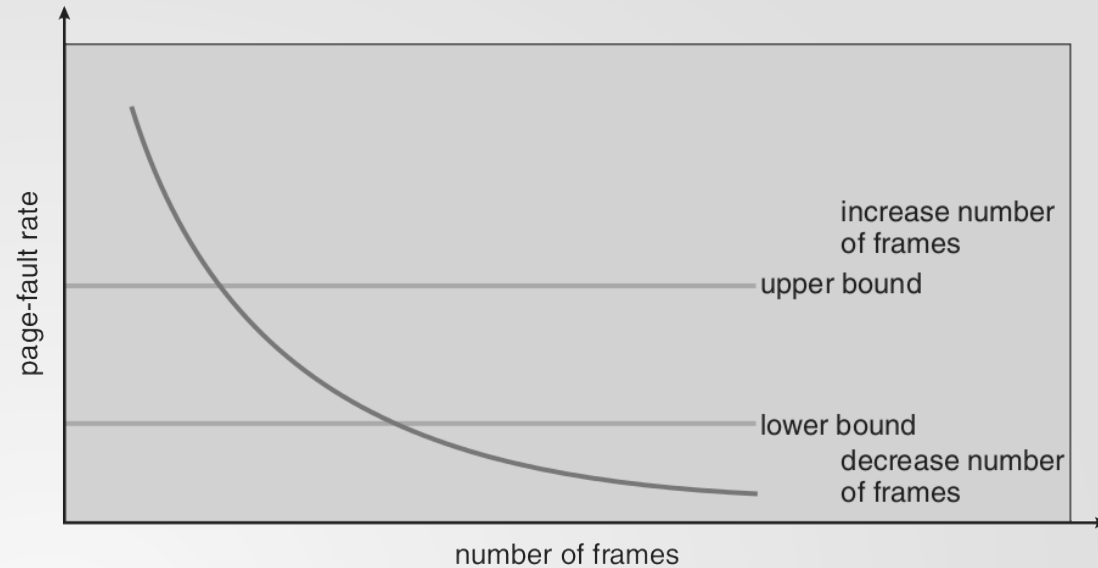


- conoscendo il working set attuale di ogni processo:
  - **richiesta globale di frame** ( $D = \sum WSS_i$ ) vs. memoria disponibile;
  - prevenzione del **trashing**
- come si calcola il working set in **pratica**? Si può approssimare usando:
  - **interrupt** periodici;
  - bit di **referenziamento** R;
  - un log che conserva la "storia di R" in base al parametro  $\Delta$ .



# Page Fault Frequency

- Esiste un modello più diretto per approcciare il trashing:
  - monitoraggio della **Page Fault Frequency (PFF)** dei processi;
- caratterizzazione dei sistemi in **sovraccarico**;
- in realtà i due modelli sono **in relazione**:



# Politica di pulitura

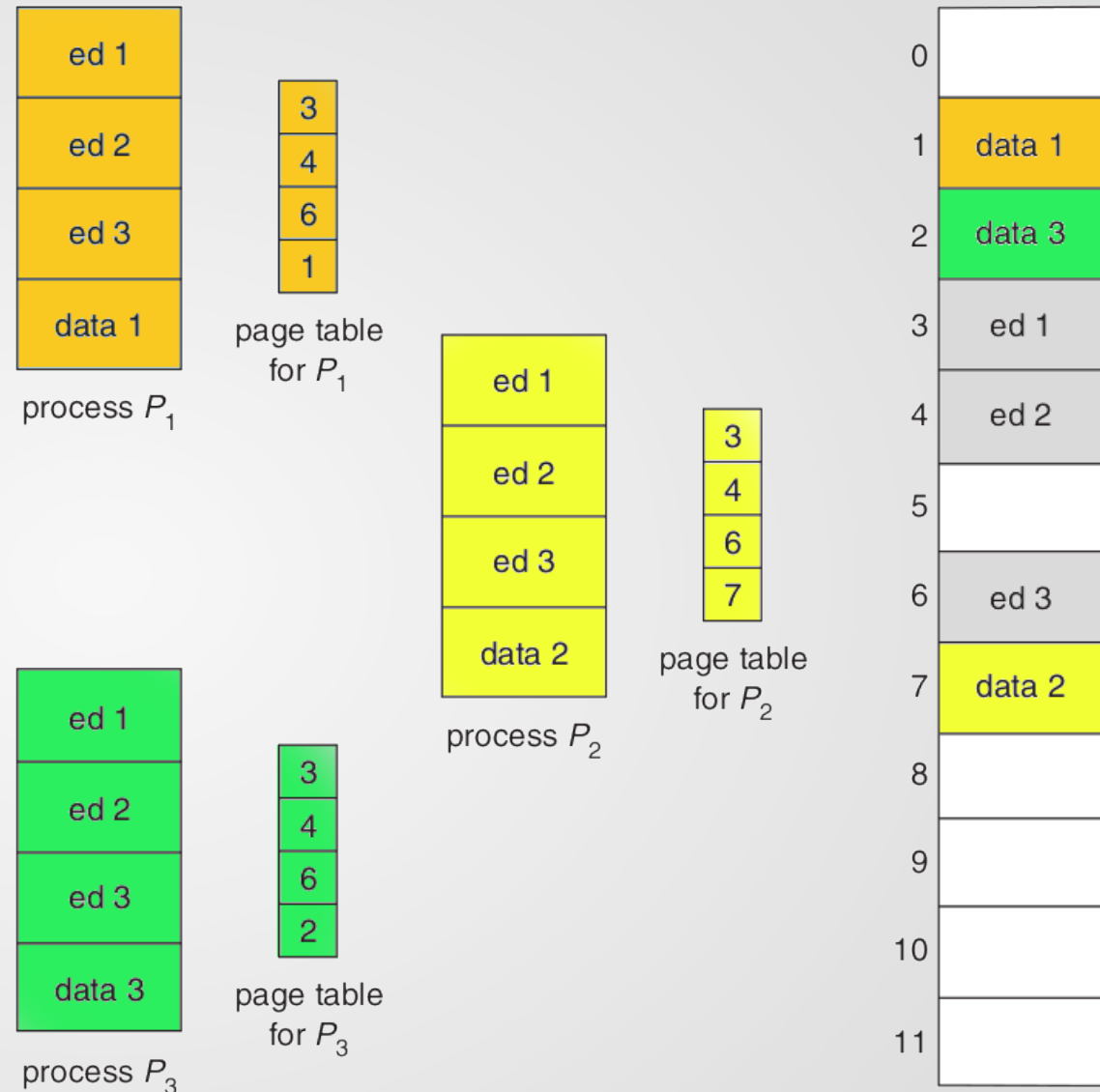
- Il meccanismo di gestione dei page fault è efficiente soprattutto se ci sono **frame liberi** sempre disponibili;
- ciò velocizza la gestione dei page-fault;
- **paging daemon**: processo di servizio che controlla lo stato di occupazione globale dei frame del sistema;
  - seleziona, libera ed, eventualmente, pulisce pagine;
  - mantiene un **pool di frame liberi**;
  - possibilità di **ripescaggio** dal pool in caso di richiesta;
  - usato ad esempio su Linux e Windows.

# Dimensione della pagina

- La scelta della dimensione della pagina di base è importante:
  - vantaggi di una **pagina grande**:
    - ♦ tabella della pagine più piccola;
    - ♦ migliore efficienza nel trasferimento I/O;
    - ♦ tende a minimizzare il numero di page fault (minore overhead);
  - vantaggi di una **pagina piccola**;
    - ♦ minore frammentazione interna;
    - ♦ migliore risoluzione nel definire il working set in memoria (meno memoria sprecata);
- relazione con la **dimensione del blocco su disco**.

# Pagine condivise

- I processi possono anche condividere in vari modi alcune pagine:
  - **solo lettura:**
    - ♦ codice eseguibile condiviso (**codice rientrante**);
  - **lettura/scrittura:**
    - ♦ IPC tramite memoria condivisa;
- implementazione su **tabella delle pagine ordinaria o multilivello:**
  - semplice ed efficiente.



# Pagine condivise

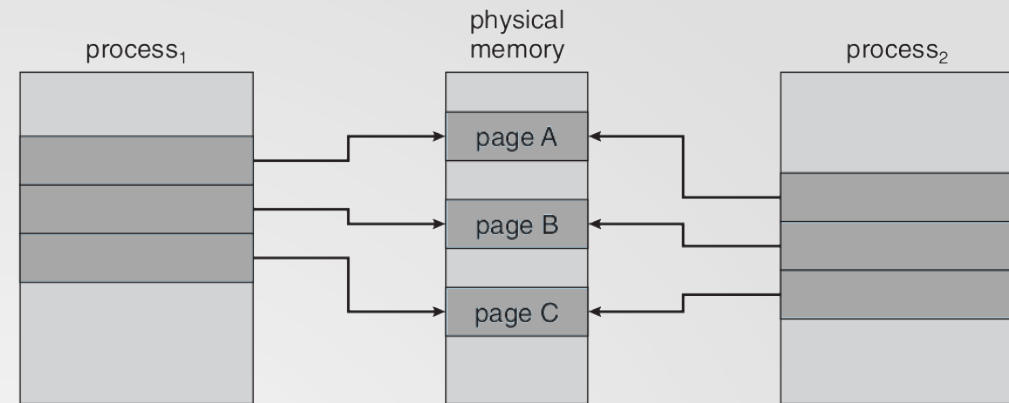
- Difficoltà:
  - **gestione della cache:**
    - ♦ **problemi di sincronizzazione** con cache basate su indirizzi virtuali (anche se usano gli ASID);
    - ♦ soluzioni:
      - ➔ disabilitare la cache sulle pagine condivise;
      - ➔ usare **cache con ricerca basata su indirizzi virtuali e tag fisici:**
        - la cache ricerca in parallelo con la TLB sulla base dell'indirizzo virtuale;
        - per capire se si tratta di un duplicato dobbiamo aspettare che la TLB dia in output l'indirizzo fisico;
  - **tabella delle pagine invertita:**
    - ♦ *singolo core*
      - ➔ alterazione tabella su context switch o su page fault;
    - ♦ *multi core*
      - ➔ difficilmente gestibile se non con teoriche tabelle delle pagine invertite con corrispondenze multi-a-uno.

# Copy-on-write e Zero-fill-on-demand

- Possibili ottimizzazioni:

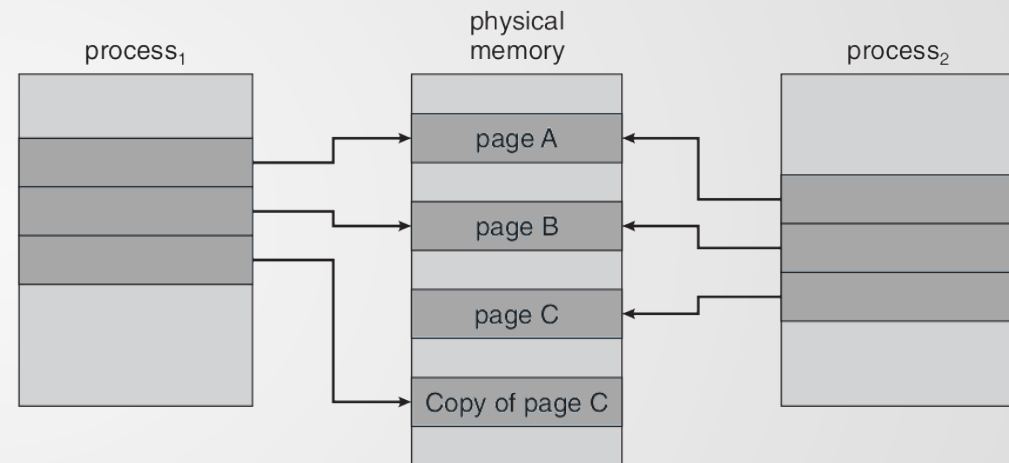
- **copy-on-write:**

- ♦ condivide finché possibile tutti i tipi di pagine (codice e dati);



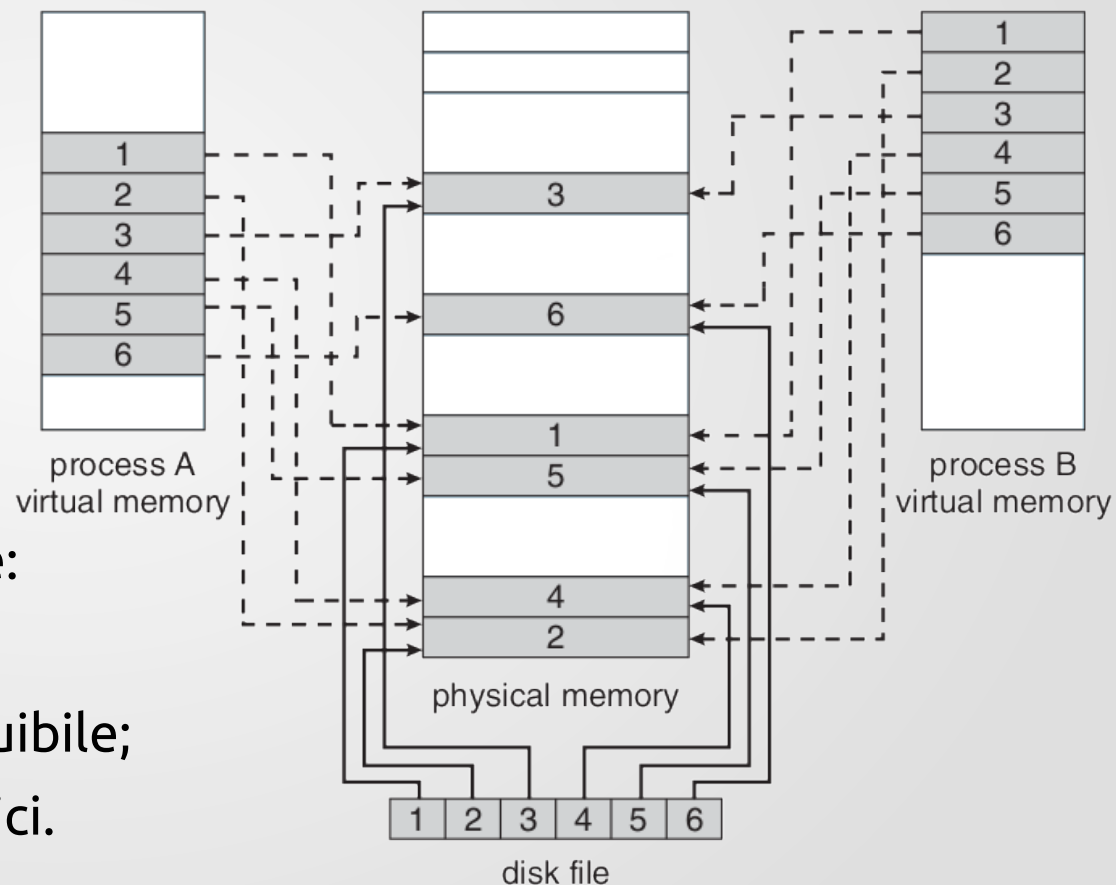
- **zero-fill-on-demand:**

- ♦ principio di base: le nuove pagine sono vuote e allocate su richiesta;
  - ➔ azzeramento efficiente gestito dal kernel;
  - ➔ sicurezza;
- ♦ pool di pagine vuote;
- ♦ copy-on-write su una *read-only static zero page*.



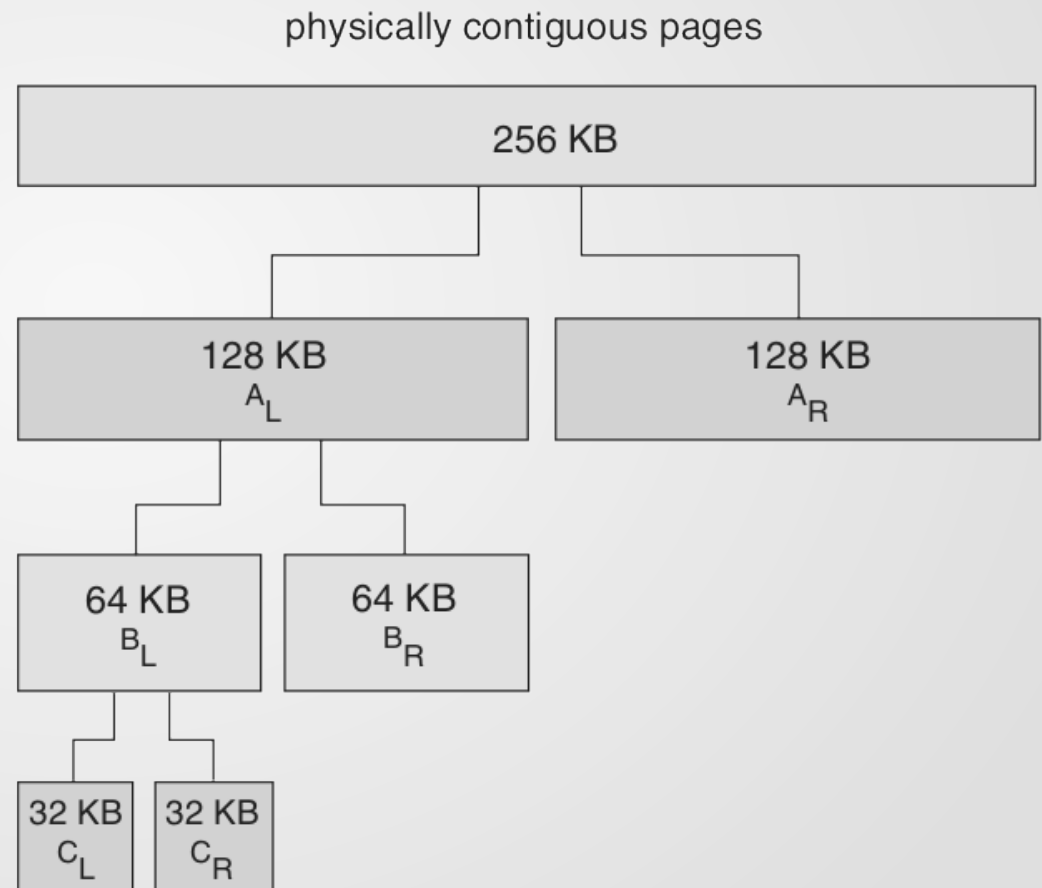
# Librerie condivise e file mappati

- Grandi librerie condivise sono comunemente usate;
  - **linking statico:** inclusione del codice in fase di linking;
  - **linking dinamico:** collegamento e caricamento a run-time di librerie condivise:
    - risparmio di spazio su disco e in RAM;
    - sviluppo indipendente e facilità di aggiornamento;
- **File mappati:**
  - modello alternativo di I/O su file;
  - possibilità di condivisione;
  - gestiscono automaticamente:
    - librerie condivise;
    - caricamento codice eseguibile;
    - caricamento dei dati statici.



# Allocazione della memoria per il kernel

- Memoria dei processi utente: paginata ma con frammentazione interna;
- Memoria interna al kernel:
  - miriamo a **frammentazione interna** minima o nulla;
  - **impossibilità di paginare** l'allocazione in alcuni casi.
- Prima soluzione: **buddy system**.





# Allocazione della memoria per il kernel

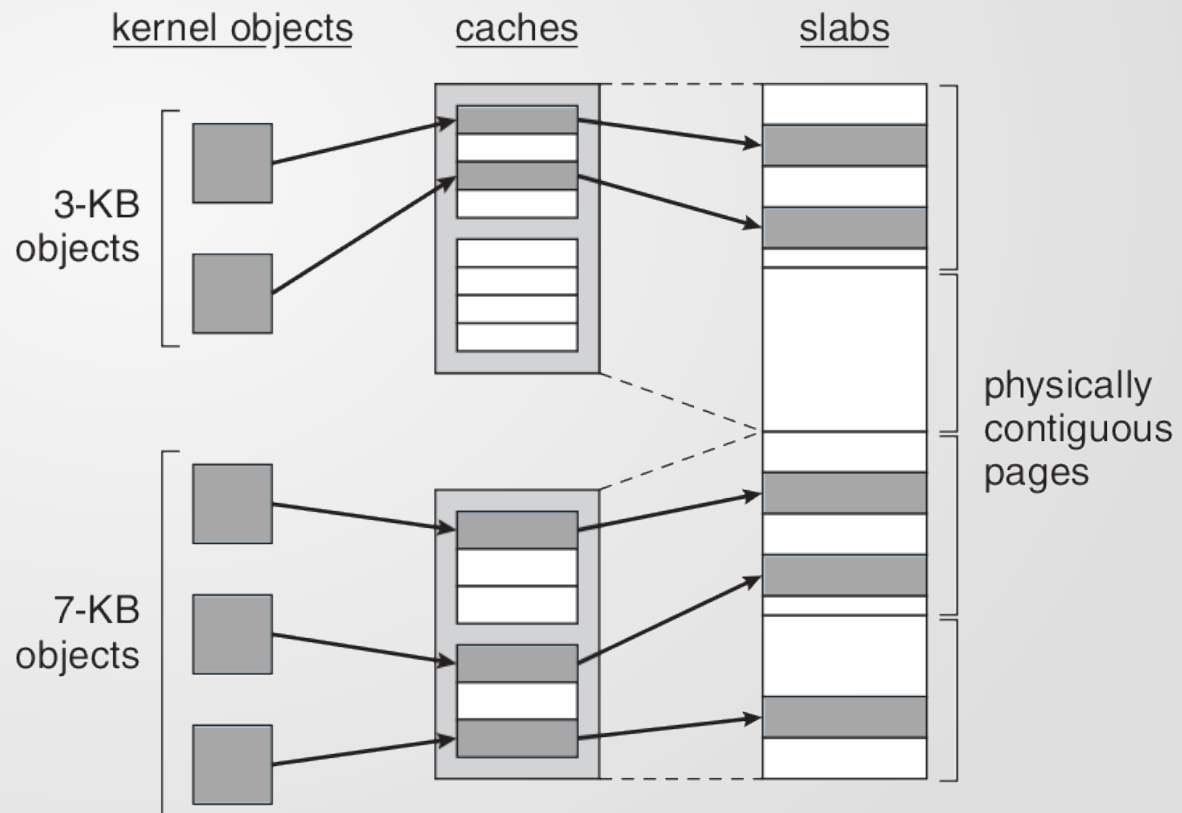
- Seconda soluzione: **slab allocation**;
  - **slab**: sequenza di pagine contigue;
  - **cache**: uno o più slab;
  - una cache per tipo di **struttura dati interna omogenea**;

- gestione:

- **stato** di uno slab:
  - ♦ pieno, vuoto, parziale;
- dinamica;

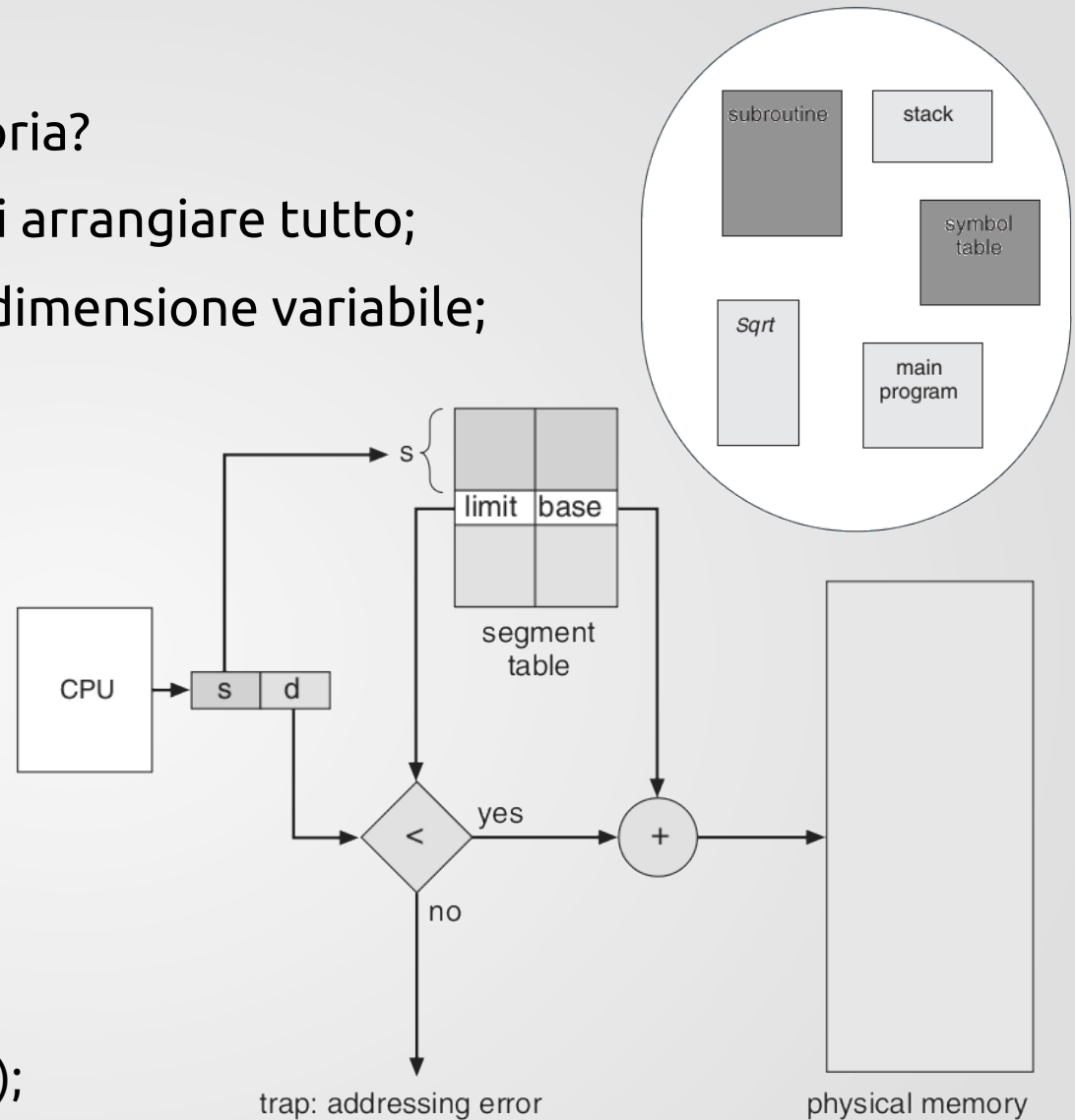
- **vantaggi**:

- niente spreco;
- efficienza.



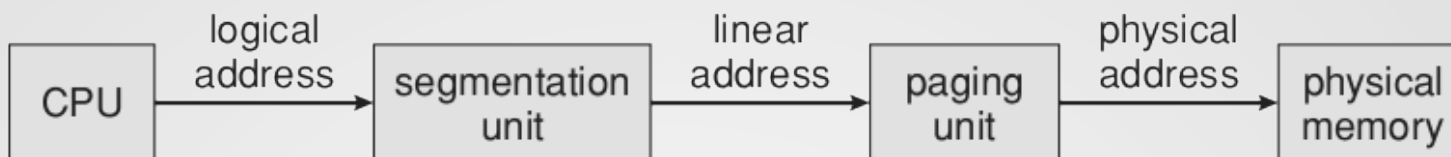
# Segmentazione

- **Visione dell'utente** della memoria?
  - un array lineare di byte in cui arrangiare tutto;
  - una collezione di oggetti di dimensione variabile;
- Alternativa: **segmentazione**
  - asseconda la **visione logica** dell'utente (programmatore);
  - un segmento per ogni oggetto: procedura, stack, libreria;
- implementazione:
  - **indirizzi bidimensionali:**  
(# segmento, offset);
  - **tabella dei segmenti:**  
# segmento  $\rightarrow$  (base, limite);
- problema: presenta **frammentazione esterna**;
- soluzione: approccio misto (**segmentazione & paginazione**).

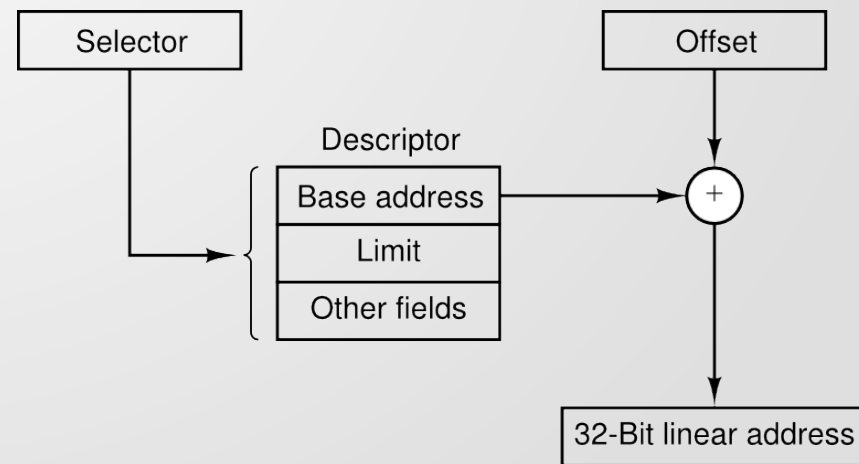


# Segmentazione su Pentium Intel

- Supporta: **segmentazione e/o paginazione;**

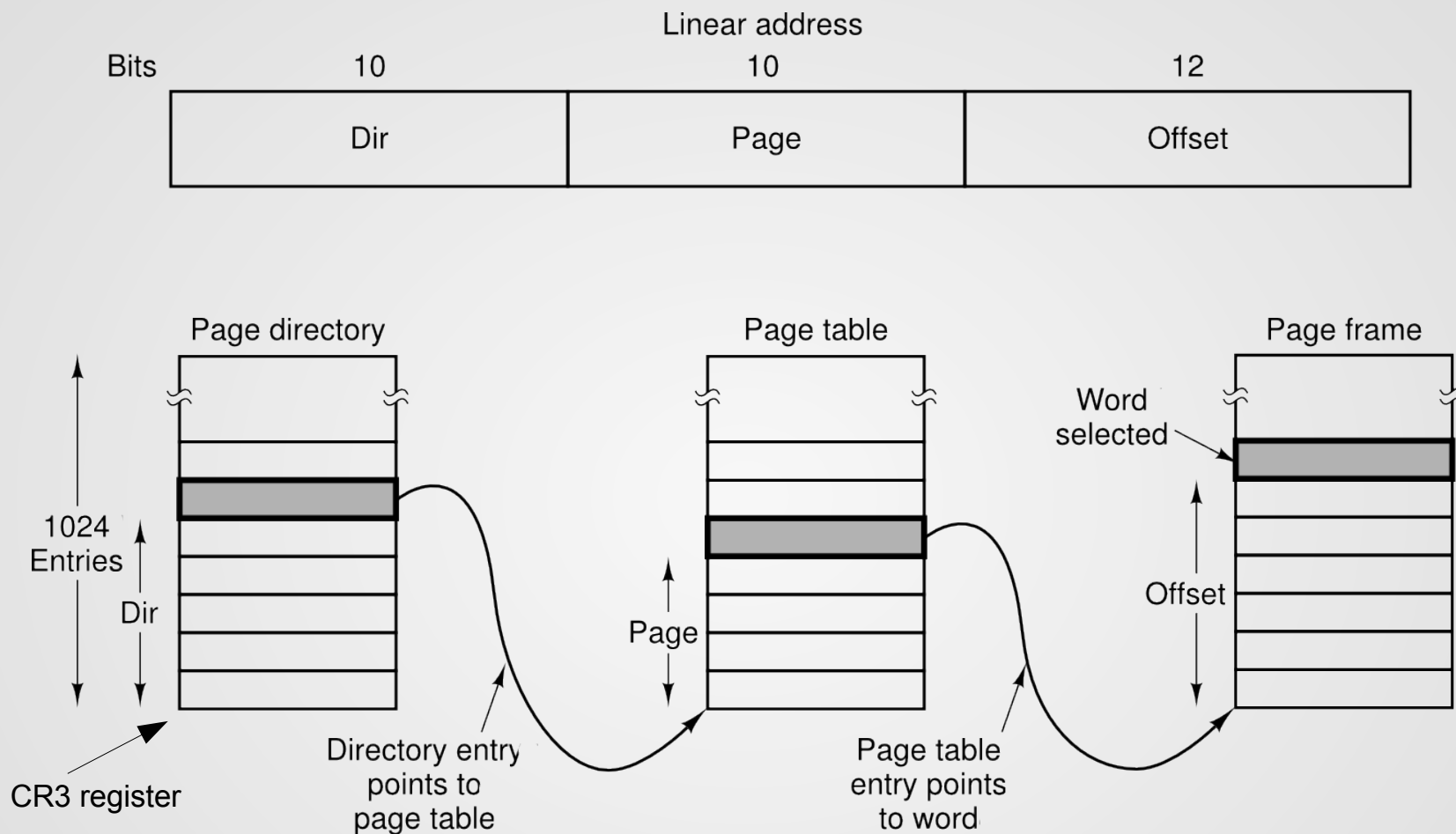


- circa 16000 **segmenti** con indirizzamenti a 4GB (32 bit);
- 2 tipi di **tabelle** per i segmenti (con circa 8000 voci ciascuna):
  - **local descriptor table** (LDT): segmenti dei programmi;
  - **global descriptor table** (GDT): segmenti del sistema operativo;
- 6 **registri segmento**: CS, DS, ...;
- 4 livelli di **protezione**.



# Segmentazione su Pentium Intel

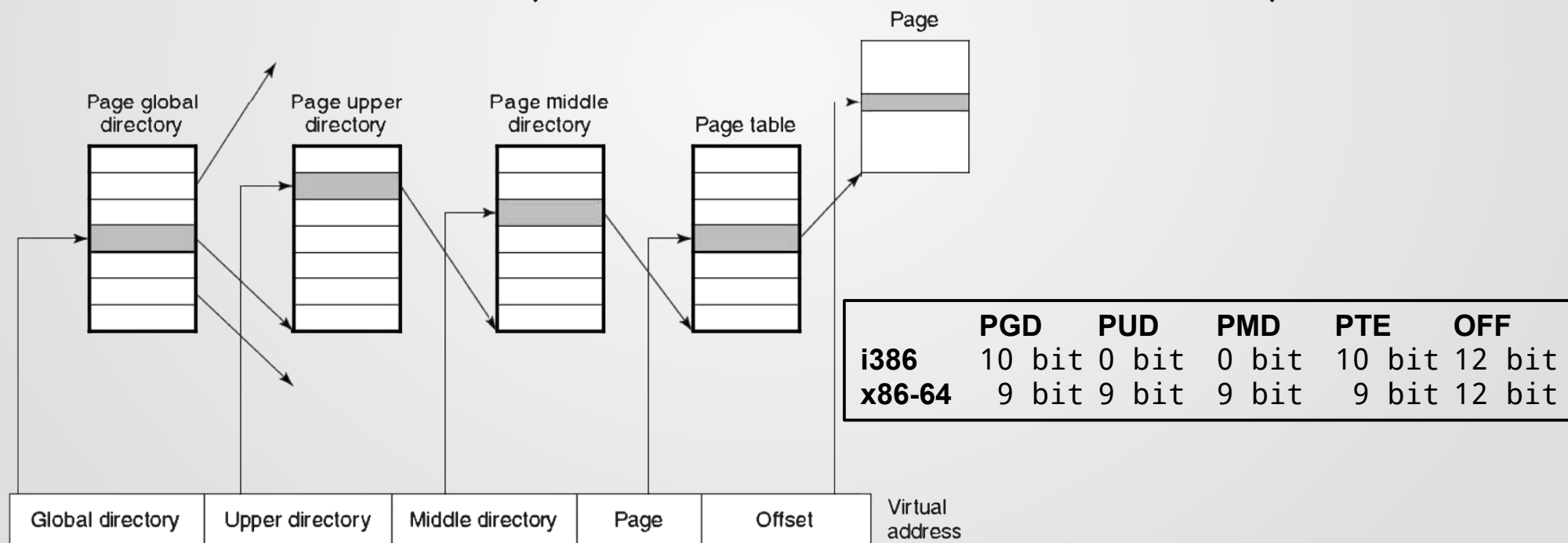
- **pagine** da 4 KB a 4 MB;



- possibilità di fare **swapping** delle tabelle della pagine (solo il 2° livello);
- impiega memoria associativa (**TLB**);
- architettura **x86-64**: supporto a tabelle con **4 livelli** e segmentazione limitata.

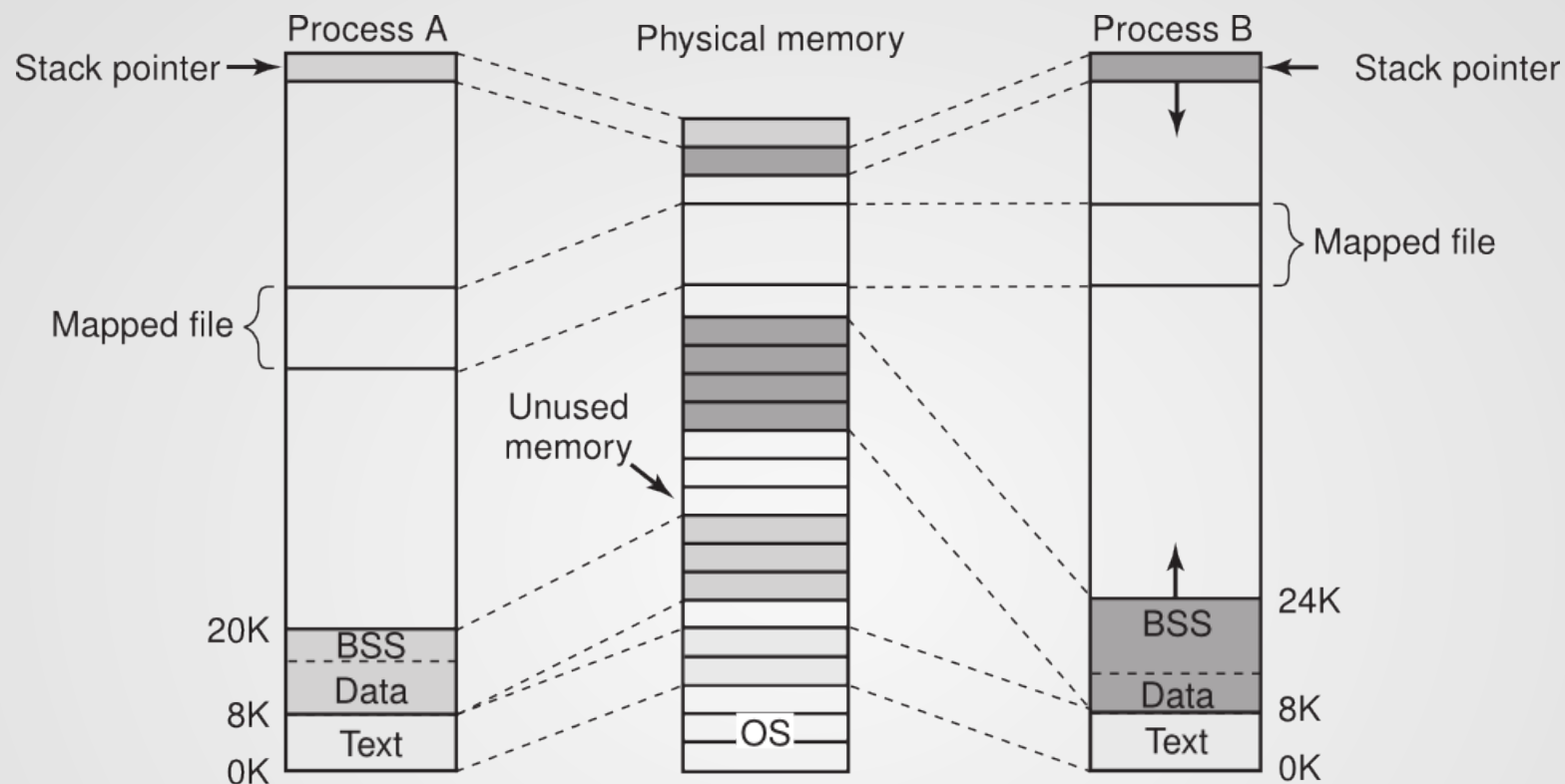
# Gestione della memoria su Linux

- **1 sistema operativo vs. tante architetture:**
  - uso limitato della segmentazione (quasi sparito su x86-64);
  - solo **2 livelli di protezione** (kernel mode, user mode);
- tabella delle pagine fino a **4 livelli** (dalla versione 2.6.11) e recente supporto a 5;
  - i386: VM da 4 GB (3 GB in user-mode e 1 GB in kernel-mode);
  - x86-64: indirizzi virtuali effettivi da 48 bit (hack!)  
→ VM da 256 TB (128 TB user-mode e 128 TB in kernel-mode).



# Gestione della memoria su Linux

- Struttura tipica dello **spazio di indirizzamento** di un processo;



- **copy-on-write** e **pagina statica zero**;
- **file mappati** (sezione codice e librerie condivise);
- **allocazione dinamica** dell'heap con chiamata di sistema `brk()`;
- algoritmo sostituzione delle pagine: ibrido tra l'algoritmo dell'orologio e NRU;
- **slab allocation** per la memoria del kernel (dalla versione 2.20).