01- Chapter9 Main Memory

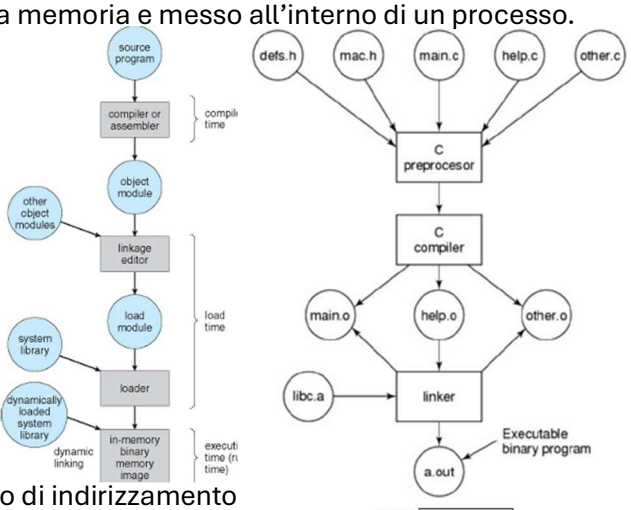
Per essere eseguito, un programma deve essere portato nella memoria e messo all'interno di un processo.

Come si **crea un eseguibile**:

- Si parte d vari .c e vari .h
- Compilatore + preprocessore
- Link: aggancia librerie

Eseguibile:

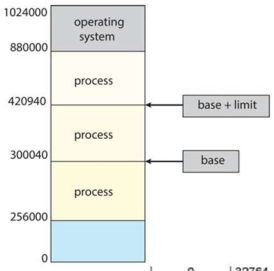
- Funzioni
- Variabili globali



Quando un processo viene creato, devo **proteggere gli altri**
→ fare in modo che un processo acceda solo al proprio spazio di indirizzamento

Ci basiamo sul concetto di **registri base e limit**, un processo deve avere:

- **Base**: registro da cui parte il mio spazio
- **Limit**: dimensione dell'intervallo in cui sta il processo

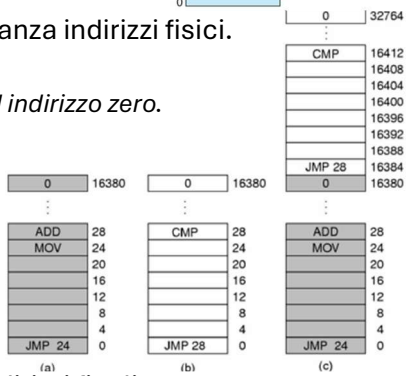


Queste due informazioni devono essere in due registri della cpu in modo tale da essere sempre accessibili.

INDIRIZZI LOGICI:
È molto importante la presenza di indirizzi logici in quanto non si hanno abbastanza indirizzi fisici.

- *Programma bianco e grigio vogliono entrambi partire da indirizzo zero*
  → *devo usare indirizzi logici facendo in modo che pensino di trovarsi ad indirizzo zero.*

  Esempio: *Se ho un jump 28, alla ram in realtà va mandato BaseRegister+28*



Address Binding *(assegnazione):*
Può comparire in 3 fasi:

- **Compile time**
  - Se si sa già dove il programma andrà in memoria, produci già gli indirizzi finali
    - *Es: se ho una macchina del caffè mono-processo*
- **Load time**
  - Codice **riallocabile** → quando ho compilato non sapevo ancora dove il mio programma sarebbe finito
  - Quando carico so però quanto devo sfasarlo per fabbricare indirizzo vero
- **Execution time** → alloco in fase di esecuzione (processo può essere spostato in altri segmenti di memoria)
  - Necessita un supporto HW per mappare indirizzi (registri base e limit)

Nota: Differenza tra caricamento ed esecuzione:

- **Load-caricamento**: quando carico il programma *(e non è detto che ci sia tutto tutto)*
- **Esecuzione**: quando effettivamente si esegue

con le librerie dinamiche, è possibile che il programma sia caricato a pezzi o che certe cose siano chiare già durante la load

INDIRIZZO LOGICO/VIRTUALE
Generato dalla cpu → indirizzo che vuole vedere il software.
Formano lo **spazio di indirizzamento logico**: intervallo tra indirizzo più piccolo e più grande che il sw può usare.

- Quasi sempre parte da 0

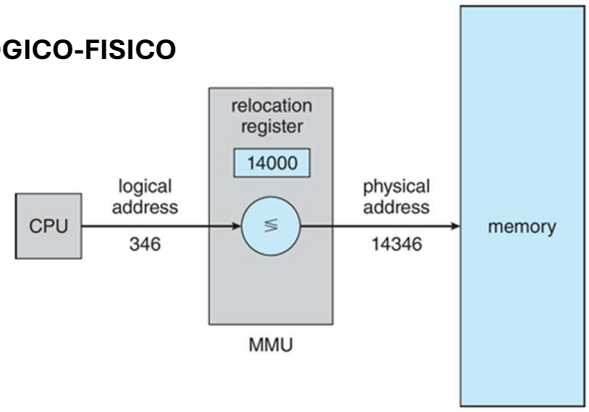
INDIRIZZO FISICO
Indirizzi fisici di RAM usabili dal programma quando è in esecuzione.

- Quello che va sull'address bus per la RAM

Formano lo **spazio di indirizzamento fisico:** insieme di indirizzi fisici che il sw può usare

MMU
Si trova all'interno della CPU e si occupa della **traduzione LOGICO-FISICO**
A runtime mappa indirizzi virtuali su indirizzi fisici.

- Basandosi sul relocation register (base register)
- Fa due operazioni:
  - Somma (+): traduce
  - Comparatore (≤): verifica che non sforo



Caricamento dinamico → dynamic loading
Non sempre un programma viene caricato tutto in un colpo solo, ci sono delle procedure che vengono caricate in memoria solo se sono effettivamente chiamate.

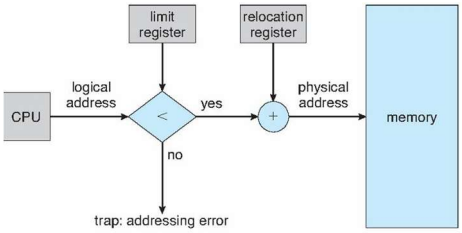
- Posso caricare in memoria meno codice
- Può succedere che certe parti di programma che non usiamo più vengano buttate via per fare posto ad altro

*Questo richiede che tutte le **istruzioni** in memoria debbano essere **riallocabili** → possibile piazzarle ovunque.*

DYNAMIC LINKING:
A differenza di quanto accadeva nello static linking, il loader non combina già tutto ma posticipa all'execution time.
Quindi, il linker lascia dei collegamenti irrisolti.
Si può avere dynamic linking senza dynamic loading ma ha poco senso perché ottimizza poco.

ALLOCAZIONE CONTINGUA
Spazio di indirizzamento fisico è un vettore senza spazi.

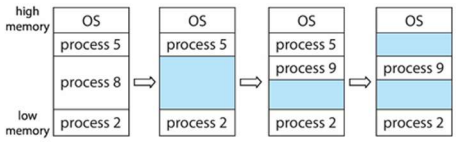
- processo in memoria in un intervallo di indirizzi contigui
- Spazio di indirizzamento fisico in questo caso ha un inizio e una fine ed è contiguo (senza buchi).



Memoria RAM: vede indirizzi fisici → physical address
CPU: vede indirizzi logici

È opportuno che ai programmi in esecuzione e quindi ai processi si allochino partizioni di dimensione opportuna:

- Un processo per essere schedulato deve essere in RAM
- Il **nr di partizioni** disponibili su RAM è **limitato** dalla piccola dimensione
- Nel momento in cui si usano partizioni variabili (di dimensioni diverse tra loro) nasce il problema della **frammentazione**
  - Buchi di dimensione variabile
    - Alternanza di partizioni allocate e partizioni libere



- Come scelgo dove allocare?
  - **First-fit**: la prima che trovo della dimensione che mi serve, è mia
  - **Best-fit**: prendo la più piccola tra quelle che hanno almeno la dimensioni che mi serve
  - **Worst-fit**: prendo la più grande possibile

FRAMMENTAZIONE:

- **Esterna**: ci sono spazi di memoria non allocata fuori dai processi → ci sono tanti pezzettini
- **Interna**: non uso tutto lo spazio allocato per il mio processo.

*Allocando spazio per un processo, potrebbe essere utile sovrastimare → quello che alloco in più rispetto a quello che serve è frammentazione interna*Nota: un'analisi a rilevato che su N blocchi allocati, si perdono 0.5N blocchi in frammentazione se si usa la first fit.

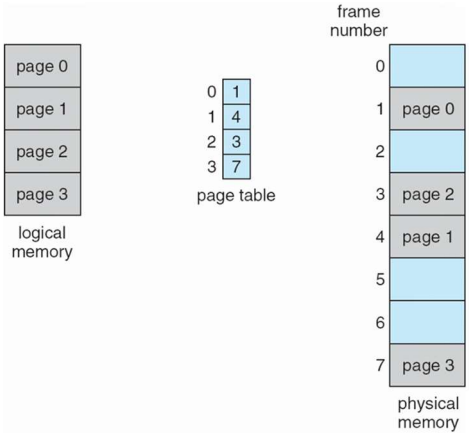
Come ridurre la frammentazione esterna:

- **Deframmentazione**: cercare di spostare tutti in contenuti della memoria, spostando tutte le cose allocate da una parte e le libere da un'altra.
  Questo si può fare solo se il tutto è dinamico → riallocazione deve essere dinamica e in fase di exe.
- **Problema dell'I/O** → se c'è un processo coinvolto in I/O, questo non posso spostarlo o altro.
  - Sol1: sposto solo se ready, mai quando sono in waiting per I/O
  - Sol2: basandomi sul fatto che il Kernel non viene mai spostato → i/o con OS buffer
    - fai in modo che la tua matrice non sia coinvolta direttamente in I/O.
      Se devi mandare la tua matrice fuori, copiala in un buffer kernel e da lì sarà mandata in I/O.
      Quindi la copia rimasta a te puoi spostarla perché si lavora in I/O con il kernel.
    - In questo modo riesco a non bloccare due dispositivi che devono comunicare

PAGING → paginazione: (no allocazione contigua perché non piace dim var)

L'address space fisico di un processo può essere **non contiguo**:

- Suddivido in **pagine di dimensione fissa** e alloco queste pagine
  - Evita frammentazione esterna



La memoria fisica è quindi divisa in **frame** (tra i 512bytes e 16 Mbytes) (potenze di 2)
La memoria logica è divisa in **pagine** di dimensioni pari ai frame.

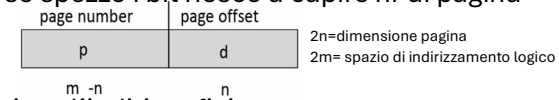
Da qualche parte, si tiene traccia di quali sono i frame liberi e quali quelli allocati.
Lo spazio logico di un processo continua ad essere contiguo ma spezzato in pagine → frammentazione interna nell'ultima pagina.

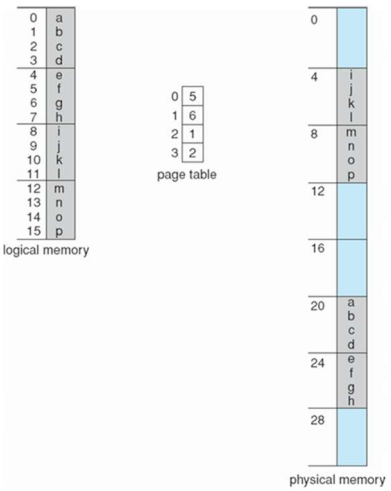
➔ Ogni pagina ha quindi bisogno del suo baseRegister e mi serve salvarlo per ogni pagina → salvo nella **page table**

NOTA: PageTable non è nella cpu; contiene l'indirizzo base della memoria fisica

MMU più complicata:

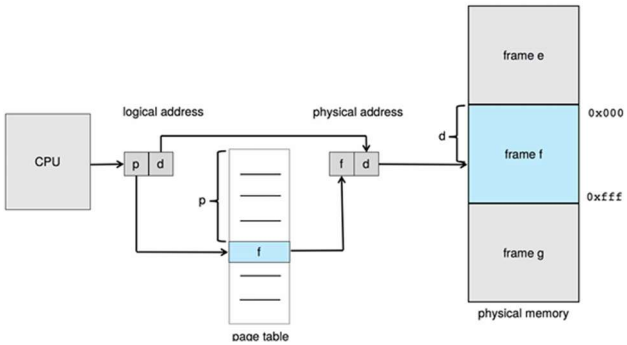
- Pagina e frame hanno dimensione che sono multipli id 2; se spezzo i bit riesco a capire nr di pagina velocemente:
  - **Numero di pagina**(p): indice nella page table
  - **Page offset** (d): combinato con il base address, definisce l'indirizzo fisico





Ad una pagina logica corrisponde un frame fisico, la cosa che mi serve memorizzare è a quale numero di pagina corrisponde un determinato numero di pagina fisico.
Numero pagina -numero frame → viene salvato nella page table:

- *Vettore che ha come:*
  - *Indice la pagina*
  - *Valore il frame*

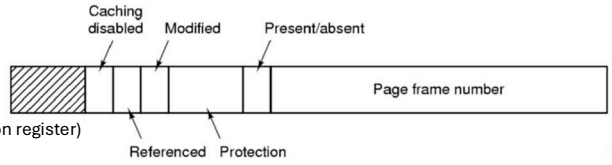


ESEMPIO CON CALCOLI:

- Page size=2048bytes, process size 72766bytes
  - 35 pagine +1086 byte
  - Frammentazione interna da 2048-1086=962 bytes
- **Peggior caso frammentazione= 1frame-1byte**
- **Frammentazione media= metà dimensione frame**

Nella **page table**, oltre al page frame number, posso aggiungere **altre informazioni**:

- Caching disabled
- Valid: pagina c'è o non c'è nell'address space del processo?
- Modified
- Protection: lettura/Scrittura/Esecuzione.
  - Execute → posso fare solo fetch (lettura che porta il dato nell'instruction register)
  - Diritti miniori di lettura (non può fare copia ad esempio)
- Present/absent

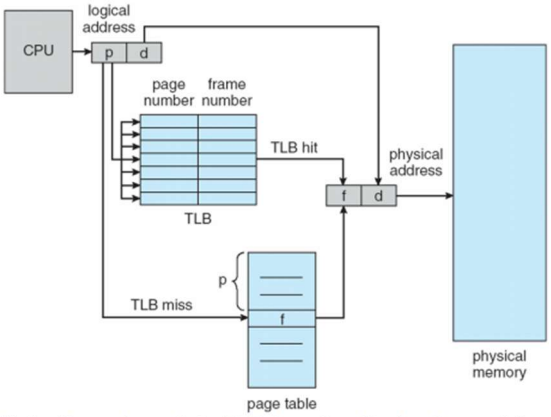


Se la **Page table è in memoria principale**:

- Ho due che registri che indicano dove finisce e quanto è grossa la page table:
  - **PTBR**: Page-table base register → punta alla page table
  - **PTLR**: indica dimensioni page table
- Page table deve essere **contigua**

Nota negativa: con questo schema, ogni accesso mi costa 2 perché devo accedere prima alla page table e poi al valore

- **TLBs → cache della page table**, si trova nella cpu
  - Non può usare il nr di page come indice
  - Memoria associativa
  - Ad ogni riga della TLB c'è un comparatore HW di uguaglianza
    - Max 1 può trovare perché i contenuti sono mutualmente esclusivi
  - Per decidere cosa mappare nella TLB, si usa la teoria della probabilità
  - Una riga della TLB contiene:
    - VirtualPage
    - PageFrame
    - Valid: riga usata/non usata
      - Per gestire eliminazioni, gestione
    - Modified
    - Protection: *RW/RX/X*
  - Cosa fare quando cambio contesto?
    - Se **tlb** è **piccola**: pulisco totalmente tlb (quando rientro ci saranno dei miss in tlb)
    - Se **tlb** è **grossa**, può essere usata da più processi contemporaneamente:
      - A questo punto devo considerare che posso avere p1 su pagina 100 con BR(0) e p2 su pagina 100 ma con BR(1000) → devo aggiungere il ASID → nr di processo
        - ASID-P-F-valid-modified-protection.



NOTA: la dimensione della TLB è molto più piccola perchè deve stare nella cpu;
la page table è molto più grandeee

***NOTA**: Invece che usare p come indice e f come contenuto, trovi p ed f insieme (+asid) con un algoritmo **associativo O(1)** e non più 2 come se accedessi direttamente alla page table.*

***NOTA: c'è sempre all'esame**: come valutare il risultato che ti dà una TLB in termini di prestazioni:*

- *Qual è la probabilità/frequenza di accessi positivi su un nr tot di accenti:*
  - *DATI: 80% di hit nella tlb*
  - *Tempo di accesso alla memoria 10ns*
    - *Tlb 10ns*
    - *Altrimenti 20ns* (tlb+memory)

Effective Access Time (EAT)

EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

implying 20% slowdown in access time

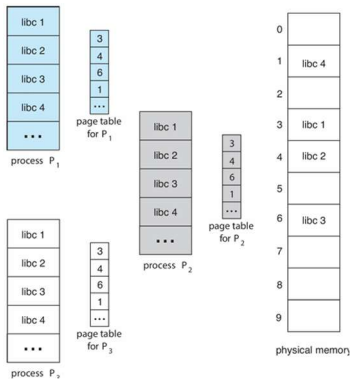
Consider amore realistic hit ratio of 99%,

EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

implying only 1% slowdown in access time.

PAGINE CONDIVISE:

- **Codice condiviso**: copia di codice read-only condivisa tra processi:
  - Se 3 processi stanno usando lo stesso editor e quindi le stesse librerie
    - Se gli eseguibili fossero statici troveremmo il tutto replicato
    - Se gli eseguibili sono ad esempio delle librerie non personalizzate e che non sono rientranti (non dipendono da variabili globali, non ricordano chiamate precedenti) posso avere dei frame condivisi
      - Il primo che li usa scatena la load, gli altri solo linking
        - Si usa meno la RAM
- **Codice privato:**
  - Ogni processo ha bisogno di una copia separata di codice e/o dati



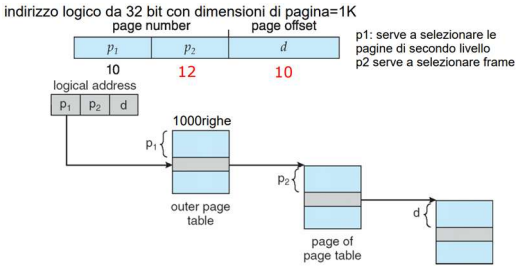
Struttura della page table: (gestita via sw) (si trova nel kernel per avere allocazione contigua)

Considerando uno spazio di indirizzamento logico a 32-bit e una dimensione di pagina= 4kB (2^{12})

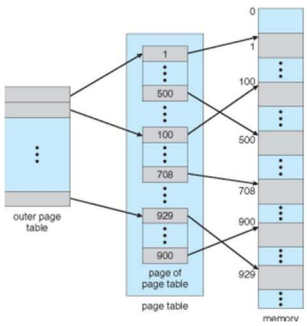
➔ La Page table avrà 1 milione di entries $2^{32}/2^{12}$

- Ogni **entry** avrà: nr di frame, qualche bit aggiuntivo, dato che vogliamo multipli pari del byte, considero 4 bytes:
  - Ogni processo ha 4MB di spazio di indirizzamento fisico per la page table
  - Nota: ci potrebbero essere contesti in cui 4MB sono troppi da allocare in maniera contigua su memoria principale
    - uso strategie alternativa
      - **PAGING GERARCHICO:**

        divido la page table in 10 tabelle da 400kB ciascuna, inserisco una **outer page table** che mi indica dove si trova ogni frame

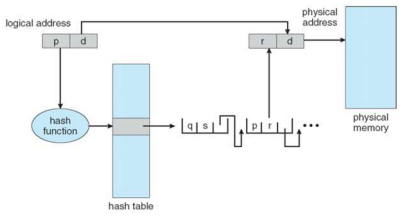


        - non risparmio in memoria ma ottengo meno requisiti di memoria contigua
        - aumenta il costo di accesso ad O(3) se miss tlb
        - se in questa tabella delle pagine, ci fosse una con una serie di invalid, posso evitare totalmente di allocare quelle pagine (da 400kB ciascuna) → dunque in realtà potrei risparmiare memoria
        - *si possono avere schemi a + livelli (vedi slide)*

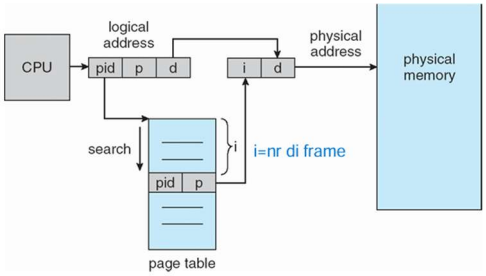


      - **HASH**:

        fare in modo che una molteplicità di dati, riescano ad essere ben organizzati



        - Collisioni: 2+ vogliono entrare nella stessa riga della tabella
          - P viene rimappato ad un certo indirizzo della hash table e quindi mi tocca cercarlo diversamente
          - Bisogna fare in modo che non avvengono
      - **INVERTED PAGE TABLE**: traccia le pagine fisiche (non c'è più una page table per ogni processo)
        - Unica tabella che ha dentro le informazioni sui frame di tutti i processi
          - 1 entry per ogni pagina reale di memoria
            - Entry: indirizzo virtuale della pagina salvata nella memoria + informazioni sul processo che possiede la pagina (PID/ASIC + nr pagina p)
          - Misurata sulla RAM
        - Davanti all'inverted page table, metti una tabella di hash per la ricerca in modo tale da renderla O(1) e non O(n)



HASH BASE: 1 per processo → chiave di ricerva P

HASH per inverted page table: 1 in generale → chiave di ricerca PID+p

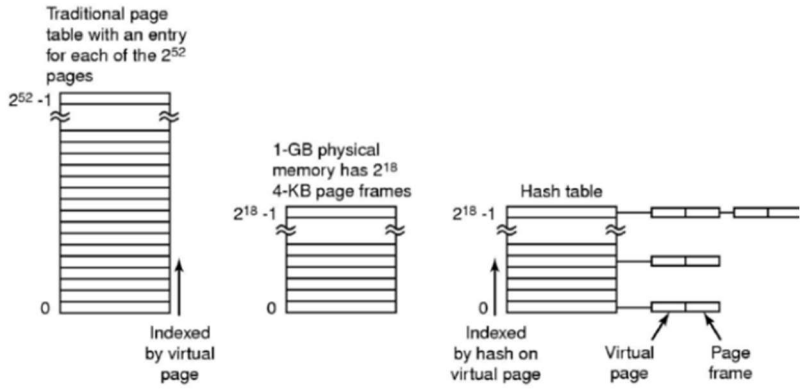


Figure 3-14. Comparison of a traditional page table with an inverted page table.

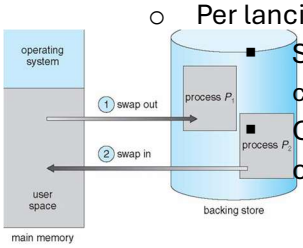
Sottile questione sw: liste concatenate sono dei blocchetti da gestire dinamicamente

- Se questi blocchetti corrispondono ognuno ad 1 frame → si può pensare ad avere una hash table che nelle liste mette direttamente un page frame
    - → inverted page table può essere un vettore con elementi pid-p-altro
    - Alcuni elementi/tutti gli elementi usati sono concatenati in una lista
- Modo diverso per fare allocazione-deallocazione e linking dei nodi in lista
    - Invece di cercare pid-p linearmente. Lo ottieni usando una funzione di hash che ti manda in un vettore in cui hai un puntatore alle liste di queste cose.

***NOTA**: non è detto che p ed f siano dimensionati in un determinato modo.*

ORACLE SPARC SOLARIS:
64bit di memoria e hashing complicato:

- **2 tabelle di hash**
  - Kernel
  - Processi utenti
    Mappano pagine virtuali ad indirizzi fisici
- **Tlb**: contiene delle entry chiamati tte e contiene le pagine dove si sono fatti gli accessi più recenti
  - Tlb miss: CPU copia nella tlb un nuovo entry preso dalla page table
- Ad un certo punto la RAM può essere piena ma io devo far comunque partire un processo
  - Frame finiti
  - Per lanciare un altro processo → **SWAPPING**



Si genera spazio disponibile in RAM buttando via un altro processo (di cui si suppone non ce ne sia bisogno per un po'), salvandolo momentaneamente su disco → **backing store**
Quando il processo torna dentro, posso salvarlo anche ad un indirizzo diverso da quello che lo avevo messo inizialmente (se il sistema di binding implementato me lo permette)

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds) molto lento
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Nota: problema dell'I/O→ se c'è I/O pending devo gestire

- O lascio stare e provo a fare swapping con altri
- Oppure gestisco I/O con kernel

Nota: sui mobile system, si sposta il problema a livello della app e non più a livello S.O.
→ dico all'app di salvarsi i dati che servono per ripartire (meno dati di quelli che salverebbe il S.O. perché l'app conosce i dati e quali sono importanti) e chiuditi.

SWAPPING con paginazione: problema del salva e ripristina viene spostato a livelli di sotto-insiemi → pagine e non interi processi.