Gestione della memoria: la struttura

Pietro Braione

Reti e Sistemi Operativi – Anno accademico 2025-2026

Argomenti

- Allocazione contigua e protezione con registro base e limite
- MMU con registro di rilocazione
- Svantaggi dell’allocazione contigua
- Paginazione
- Swapping
- Grado di multiprogrammazione e utilizzazione dei processori
- Memoria virtuale

Allocazione contigua e protezione con registro base e limite

Ricapitoliamo un po’ di background…

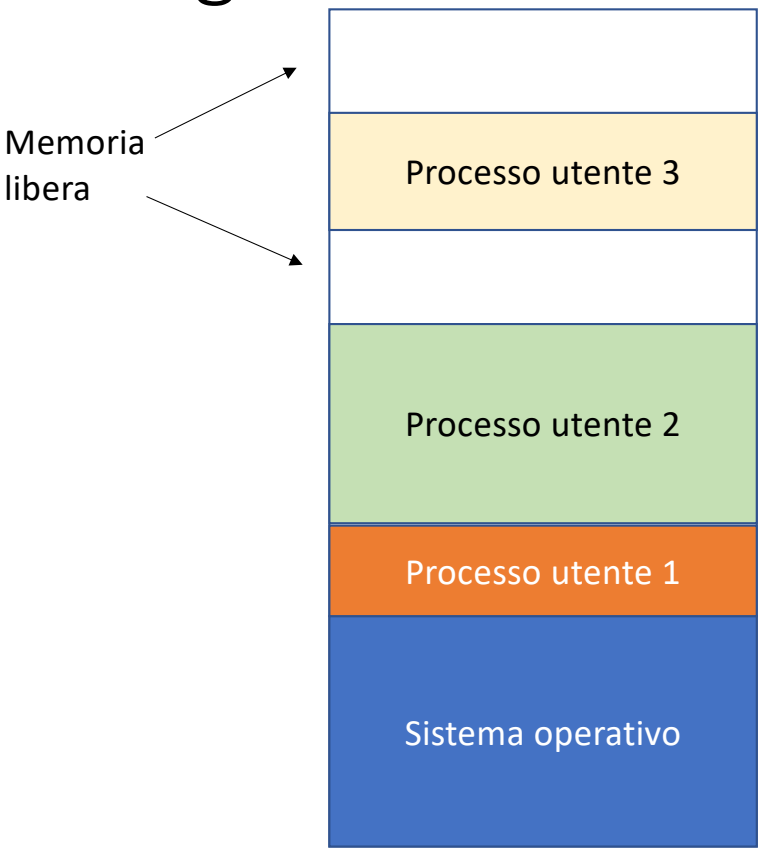
- Le aree di memoria che i processori possono usare direttamente per l’esecuzione dei programmi sono solo la memoria centrale ed i registri
- Pertanto un programma deve essere portato dal disco in memoria centrale perché possa essere eseguito, e il programma deve avere anche sufficiente memoria centrale per memorizzare i risultati della computazione
- La memoria centrale «vede» solo un flusso di indirizzi (richieste di lettura o scrittura) proveniente dal bus di sistema, e non è consapevole di chi ha generato tale flusso
- Le operazioni sui registri richiedono un ciclo di clock (o anche meno)
- Viceversa, le operazioni sulla memoria richiedono molti cicli di clock, anche diverse centinaia, causando uno stallo (stall) del processore, durante il quale un core multithread può eseguire un altro thread hardware
- Per aumentare l’efficienza degli accessi in memoria vengono utilizzati diversi livelli di memorie cache tra processore e memoria centrale

Il problema dell'allocazione della memoria

- Perché un processo possa andare in esecuzione la sua immagine (codice + dati statici + stack + heap) deve essere presente in memoria centrale
- In un sistema multiprogrammato, più immagini di più processi sono contemporaneamente nella memoria centrale
- Il sistema operativo deve, pertanto, allocare porzioni di memoria centrale ai diversi processi in funzione delle necessità di tali processi
- Questo pone diversi problemi:
  - Che **strategie di allocazione** possiamo adottare?
  - Come **proteggiamo** la memoria del sistema operativo dai processi, e quella di ogni processo da ogni altro processo?
  - Se un programma può essere caricato, in momenti diversi, in posizioni diverse della memoria, come possono le istruzioni del programma **referenziare** le locazioni di memoria usate dal programma? Ossia: come forniamo uno spazio di indirizzamento virtuale ai processi?

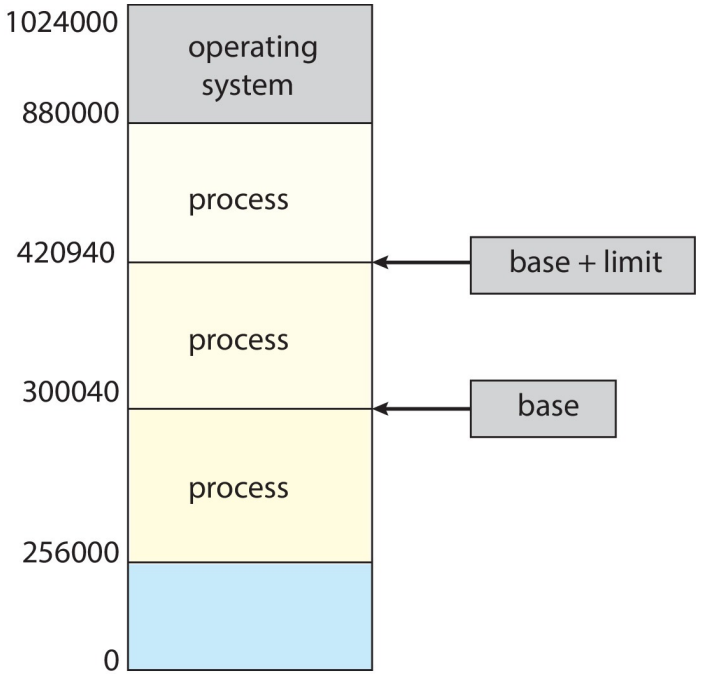
Strategia di allocazione contigua

- È la strategia più semplice di allocazione della memoria in un sistema multiprogrammato
- La memoria centrale è partizionata in due zone, una per il sistema operativo e una per i processi utente
- Ogni processo utente occupa un'area contigua di memoria nella partizione dei processi utente, e in quell'area viene caricata la sua immagine



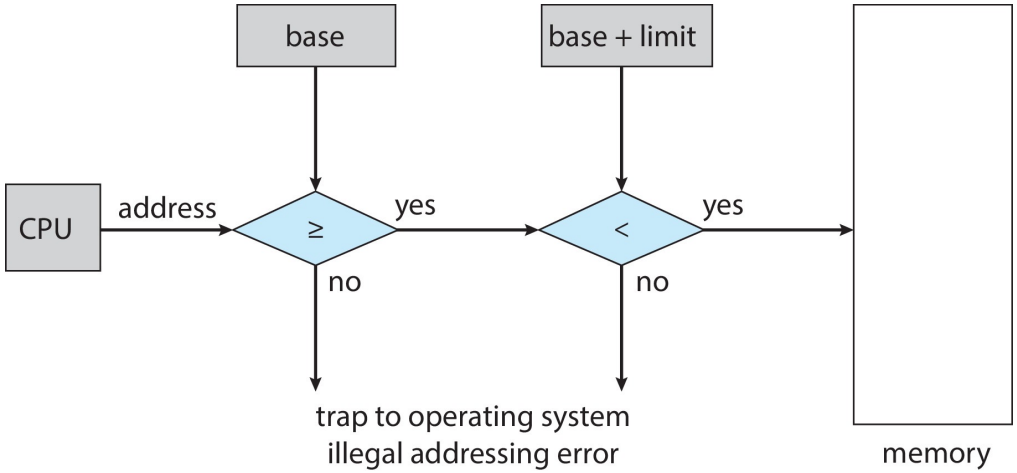
Protezione con registro base e limite (1)

- È il più semplice metodo di protezione utilizzabile con l'allocazione contigua
- Il processore possiede due registri, un registro **base** ed un registro **limite**
- Il registro base contiene il più piccolo indirizzo della memoria fisica che il processo corrente ha il permesso di accedere
- Il registro limite determina la dimensione dell'intervallo ammesso



Protezione con registro base e limite (2)

- I registri base e limite possono essere impostati solo in modalità di sistema
- In modalità utente il processore proibisce tutte le operazioni di lettura/scrittura fuori dall'intervallo individuato dai registri base e limite
- Nel caso in cui venga generato un indirizzo fuori dall'intervallo, l'indirizzo non viene messo sul bus e viene generata un'eccezione
- In modalità di sistema il processore può accedere a tutta la memoria



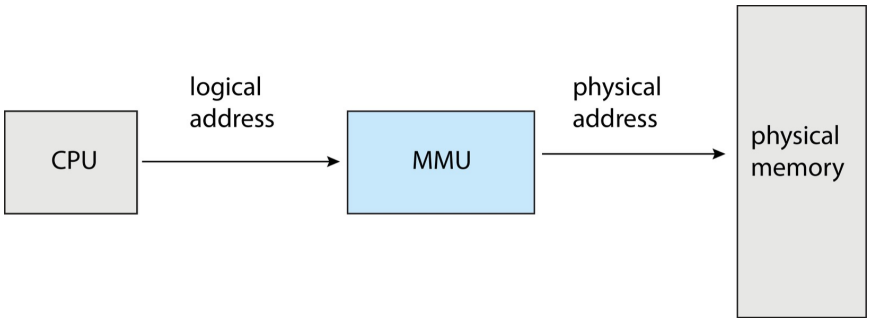
Svantaggi della soluzione con registro base e limite

- La soluzione con registro base e limite ha uno svantaggio: non fornisce uno spazio di indirizzamento virtuale ai processi
- L'unico modo per implementare uno spazio di indirizzamento virtuale con tale soluzione sarebbe il binding in fase di caricamento
- Ma questo è molto lento!
- Pertanto si preferiscono soluzioni basate sul binding in fase di esecuzione

MMU con registro di rilocazione

Indirizzi logici e fisici

- Gli indirizzi generati dalla CPU quando esegue un programma sono detti **indirizzi logici**
- Gli indirizzi che arrivano alla memoria centrale sono detti **indirizzi fisici**
- Il binding in fase di esecuzione è l'unico metodo nel quale indirizzi logici e fisici differiscono
- La **memory management unit** (MMU) è il dispositivo hardware che traduce indirizzi logici in indirizzi fisici
- La MMU interviene solo in modalità utente: In modalità kernel gli indirizzi generati dalla CPU sono direttamente indirizzi fisici
- (Quindi, se il sistema operativo deve accedere alla memoria di un processo, deve tradurre «manualmente» gli indirizzi logici del processo in fisici)

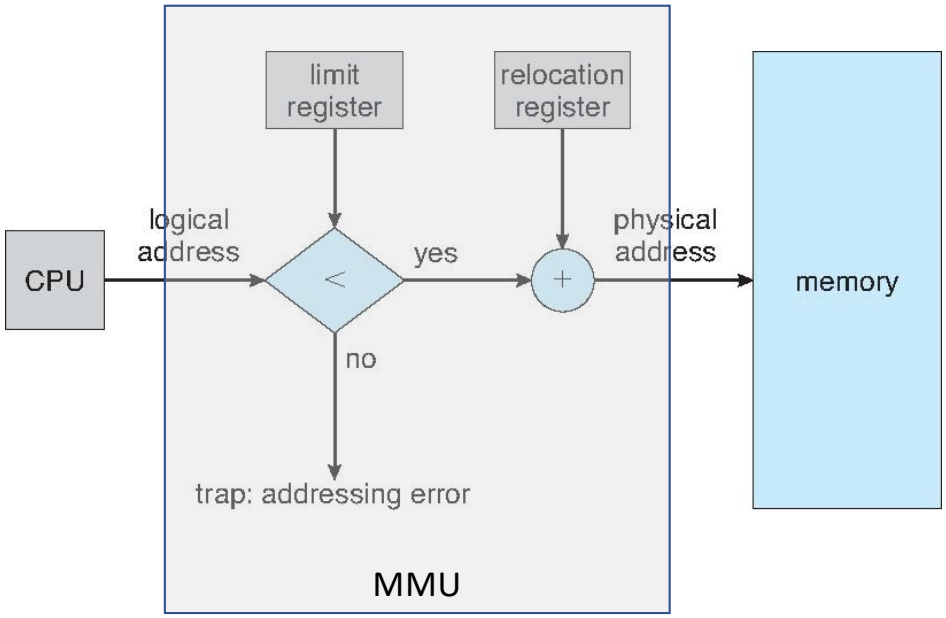


MMU con registro di rilocazione (1)

- La MMU più semplice utilizzabile con lo schema di allocazione contigua è la MMU con **registro di rilocazione**
- È una variazione dello schema con registri base e limite, dove il registro base è ora chiamato registro di rilocazione
- L'indirizzo fisico è ottenuto sommando all'indirizzo logico il valore del registro di rilocazione
- In tal modo i programmi hanno l'illusione di avere uno spazio di indirizzamento virtuale che va dall'indirizzo 0 a un indirizzo massimo pari al valore contenuto nel registro limite

MMU con registro di rilocazione (2)

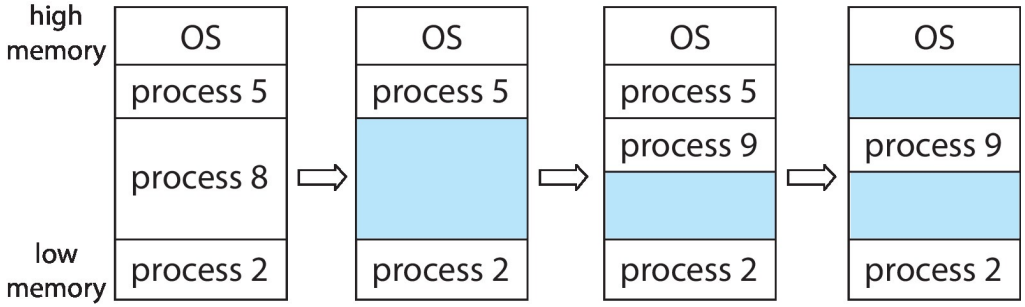
- Nel registro base viene caricato l'indirizzo più basso dell'area contigua di memoria assegnata al processo
- Nel registro limite viene caricata la dimensione di tale area di memoria
- Se l'indirizzo logico supera tale valore, viene generata un'interruzione (intercettata dall'OS, che di solito interrompe il processo)



Svantaggi dell’allocazione contigua

Allocazione contigua a partizioni variabili (1)

- Ogni processo ottiene una partizione di memoria distinta
- Schema a **partizioni variabili**: partizione di dimensione pari alla memoria necessaria al processo
- Un buco è una regione di memoria libera contigua, ed ogni processo riceve la sua partizione di memoria da un buco abbastanza grande da contenerla
- Quando un processo termina libera la sua partizione creando un nuovo buco, e buchi adiacenti sono uniti



Allocazione contigua a partizioni variabili (2)

- Il sistema operativo mantiene una lista dei buchi disponibili sparsi nella memoria centrale
- Alla creazione di un nuovo processo il sistema operativo sceglie un buco dal quale prendere la memoria necessaria ad esso secondo una strategia:
  - **First-fit**: sceglie il primo buco sufficientemente grande da contenere l'immagine del processo
  - **Best-fit**: sceglie il buco più piccolo
  - **Worst-fit**: sceglie il buco più grande
- First fit e best-fit sono sperimentalmente migliori in quanto a tempo ed efficienza

Frammentazione esterna

- **Frammentazione esterna**: la memoria libera è sufficiente per la creazione di un nuovo processo, ma è sparsa tra buchi non contigui troppo piccoli
- Con allocazione contigua a partizioni variabili vale la «**regola del 50%**»: lo spazio inutilizzabile per frammentazione esterna tende ad essere circa il 50% dello spazio utilizzato
- Possibile rimedio: **compattazione**
  - Spostare le partizioni in maniera da poter unire buchi separati
  - Richiede binding in fase di esecuzione (rilocazione dinamica)
  - È molto onerosa in termini computazionali
  - Inoltre se il processo effettua I/O non può essere rilocato; alternativamente, l'I/O va fatto solo in buffer interni al sistema operativo

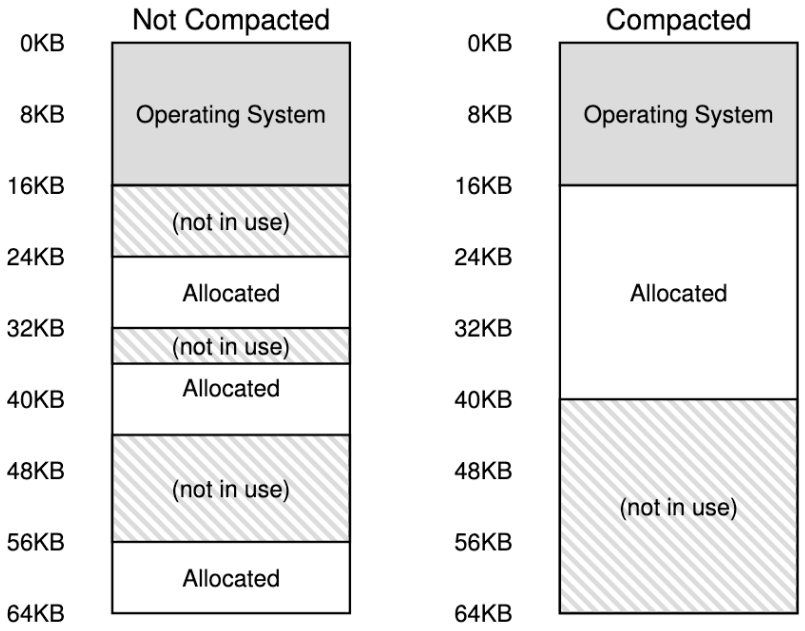


Figure 16.6: **Non-compacted and Compacted Memory**

Frammentazione interna

- E se usassimo partizioni a dimensioni fisse? Avremmo un problema di **frammentazione interna**: se la memoria allocata ad un processo è più grande della memoria necessaria, una partizione può contenere memoria inutilizzata
- La frammentazione è un problema generale che si verifica anche nelle memorie secondarie

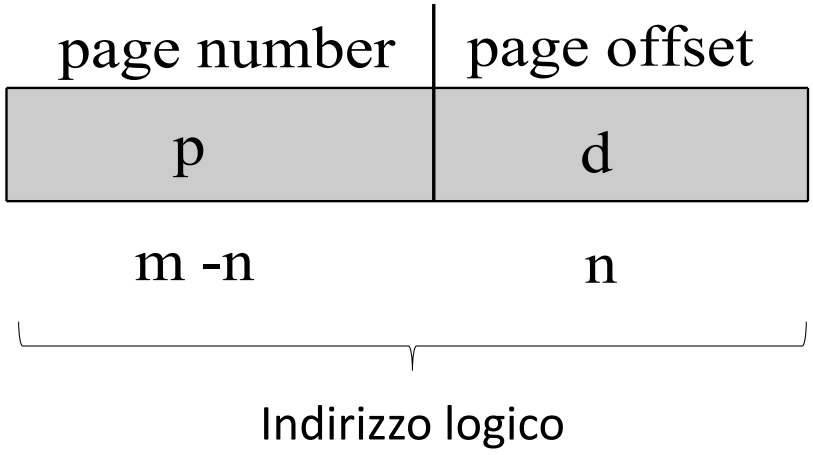
Paginazione

Paginazione

- Idea: permettere una allocazione di memoria ai processi *noncontigua*
- La memoria centrale viene divisa in **frames**, ossia blocchi di dimensione fissa (tra 512 bytes e 16 Mbytes)
- Similmente lo spazio di indirizzamento virtuale di ogni processo è diviso in **pagine**, ossia blocchi delle stesse dimensioni dei frames
- Una **tabella delle pagine** associa le pagine di un processo ai frames in memoria centrale, e permette alla MMU di tradurre gli indirizzi logici in fisici
- Vantaggi:
  - Non vi è più frammentazione esterna
  - Vi è piena indipendenza tra indirizzi logici e indirizzi fisici (ad esempio, si possono avere indirizzi logici a 64 bit anche se la memoria fisica è più piccola)
- Svantaggi: frammentazione interna

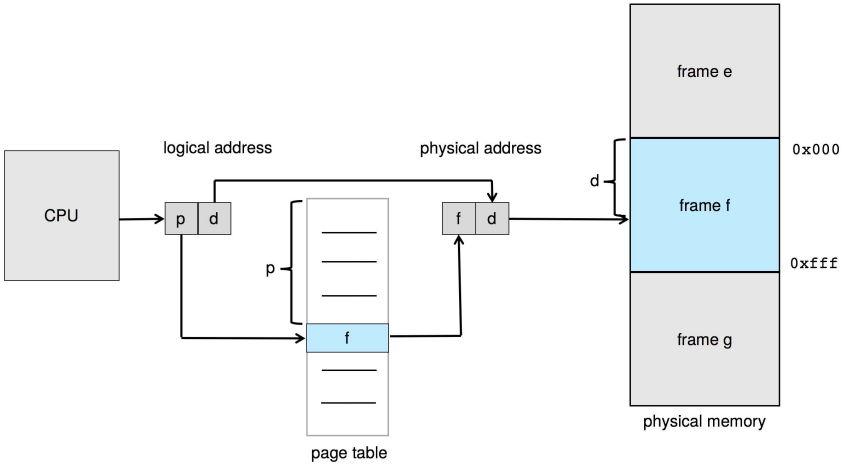
Paginazione: traduzione degli indirizzi logici (1)

- Un indirizzo logico è diviso in:
  - Numero di pagina (p): usato come indice nella tabella delle pagine
  - Offset di pagina (d): offset all'interno della pagina (identico all'offset nel frame)
- Se lo spazio di indirizzi logici ha dimensione 2^m…
- …e le pagine dimensione 2^n…
- …il numero totale di pagine è 2^{m-n}

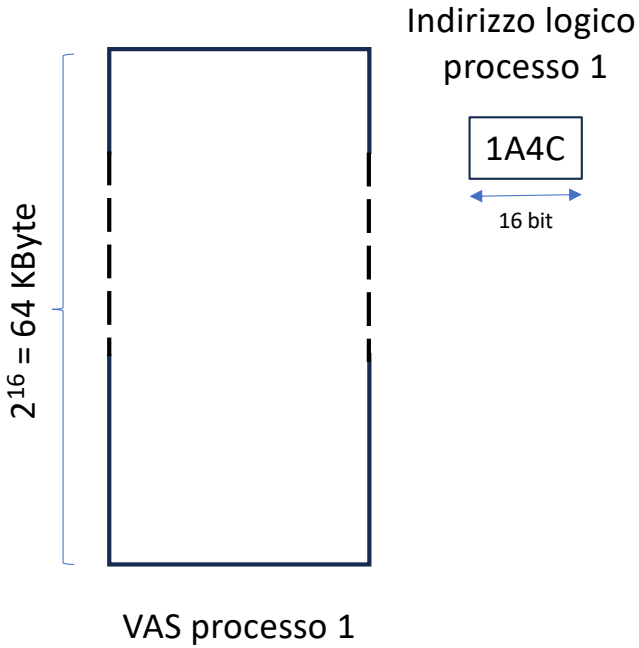


Paginazione: traduzione degli indirizzi logici (2)

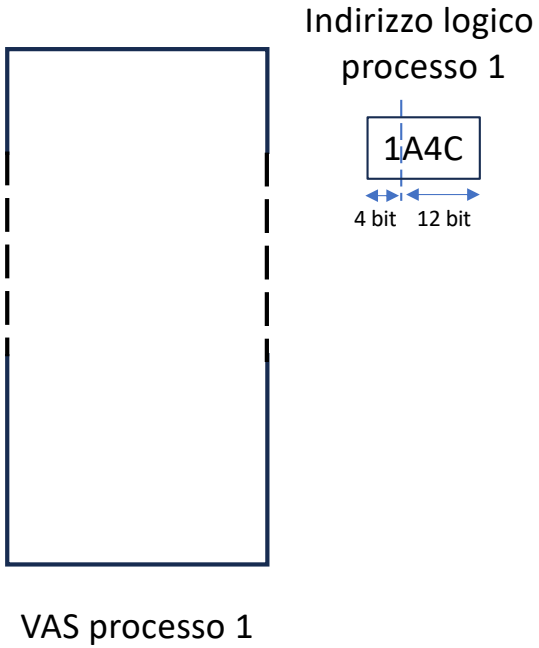
- La MMU traduce un indirizzo logico in fisico in questo modo:
  - Estrae il numero di pagina dall'indirizzo logico
  - Utilizza il numero di pagina per ricavare dalla tabella delle pagine il corrispondente numero di frame
  - Concatena il numero di frame all'offset di pagina e ottiene l'indirizzo fisico
- Vediamo un esempio



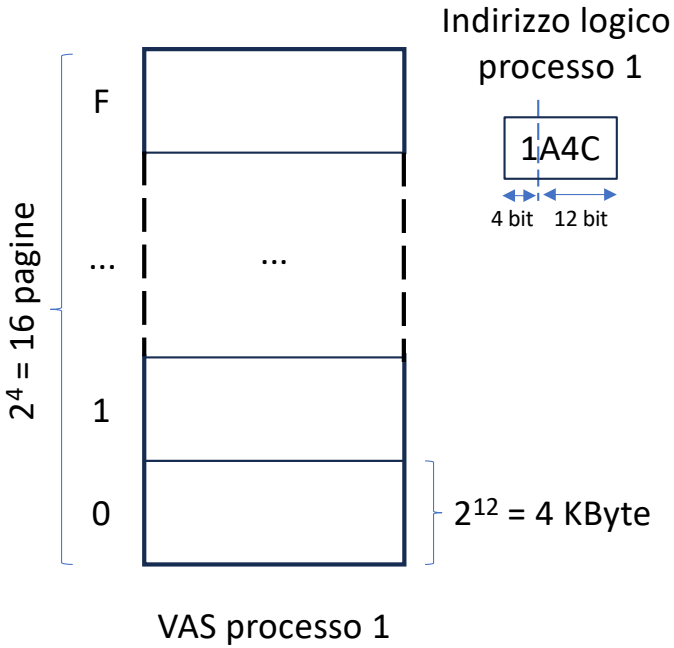
Paginazione: un esempio



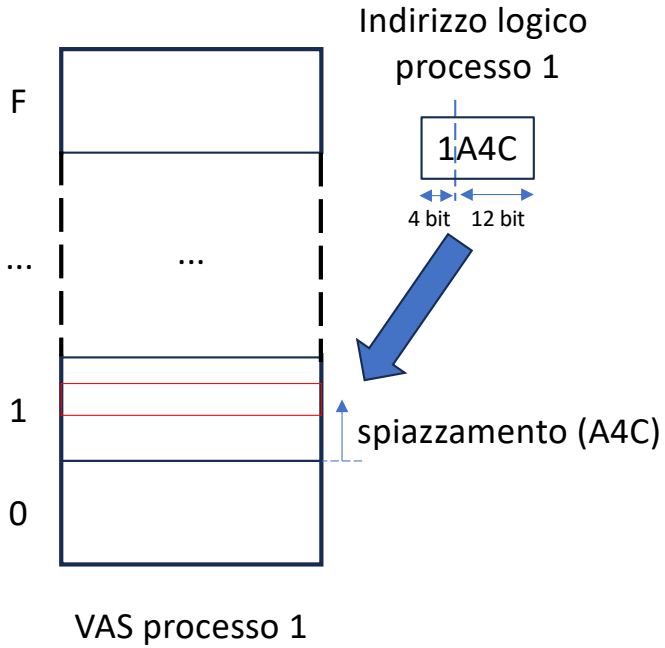
Paginazione: un esempio



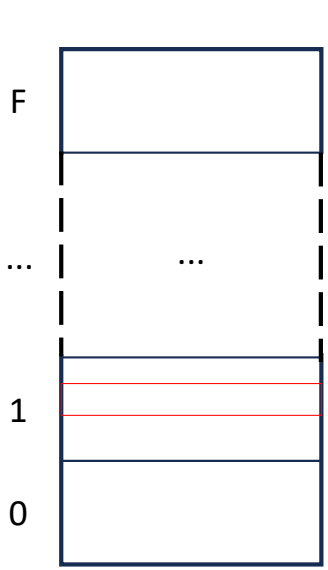
Paginazione: un esempio



Paginazione: un esempio

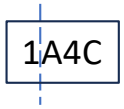


Paginazione: un esempio



VAS processo 1

Indirizzo logico
processo 1



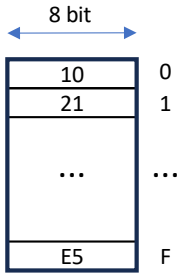
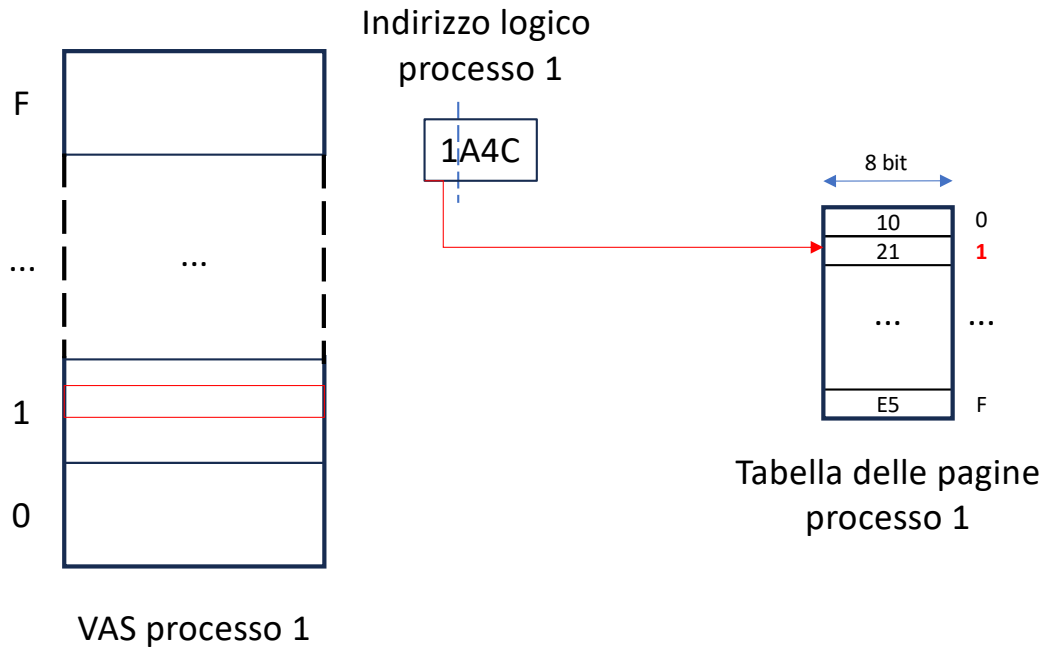
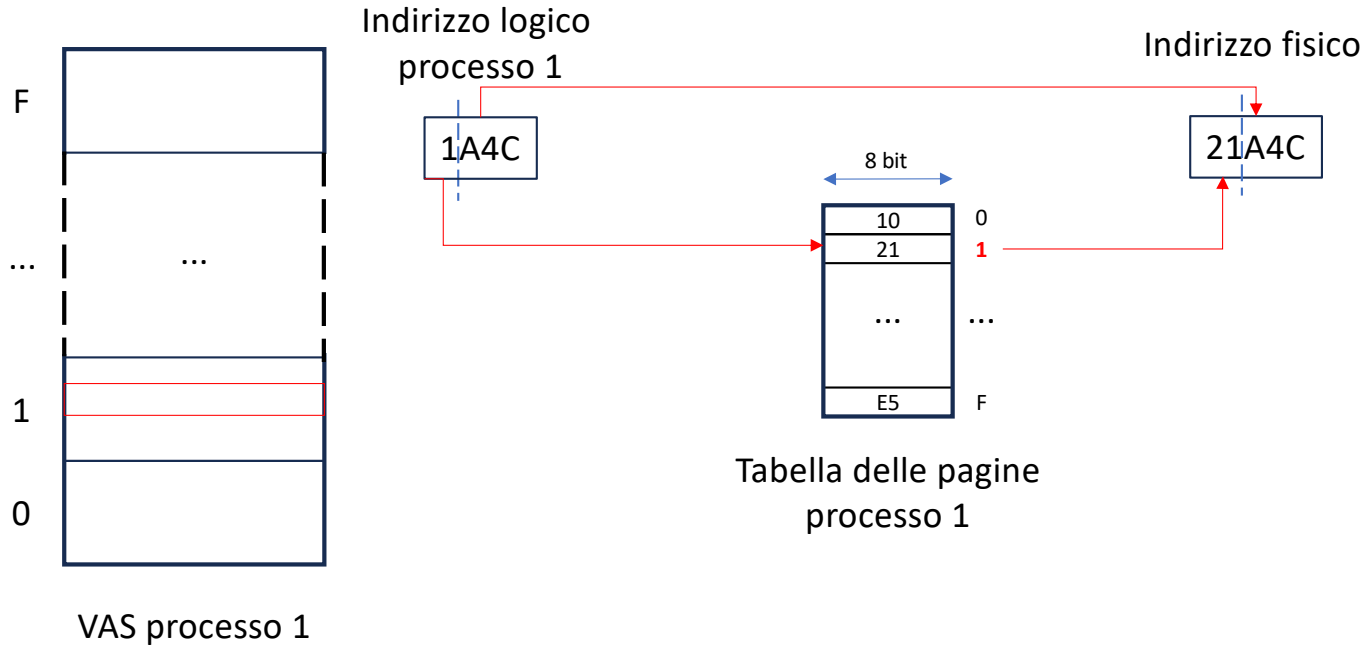


Tabella delle pagine
processo 1

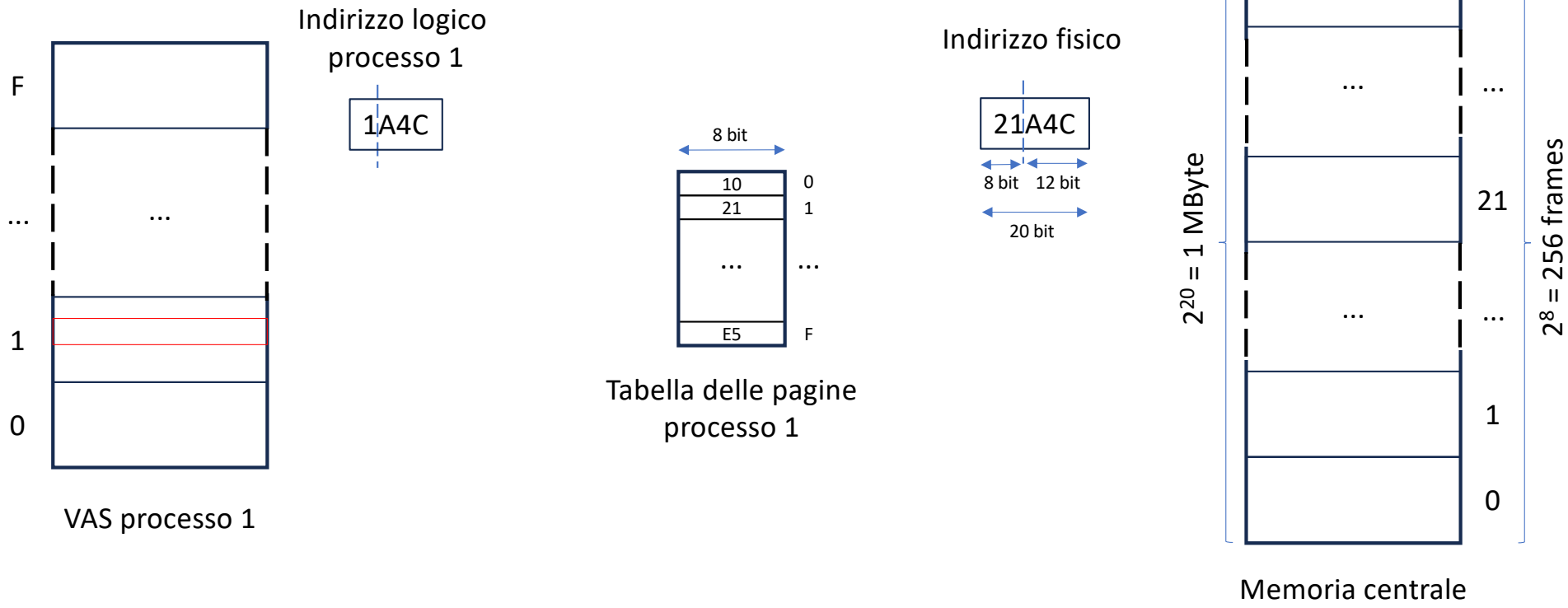
Paginazione: un esempio



Paginazione: un esempio

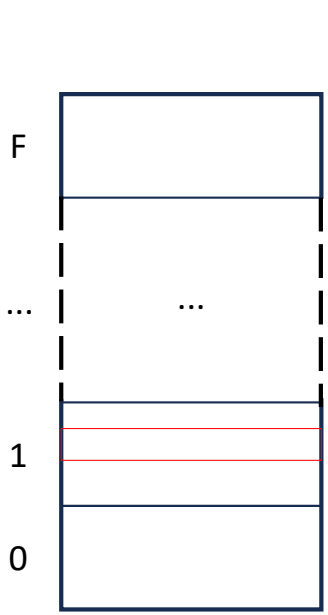


Paginazione: un esempio



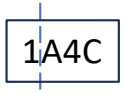
Paginazione: un esempio

Paginazione: un esempio



VAS processo 2

Indirizzo logico
processo 2
(lo stesso!)



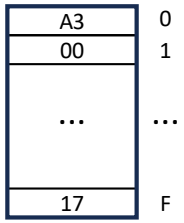
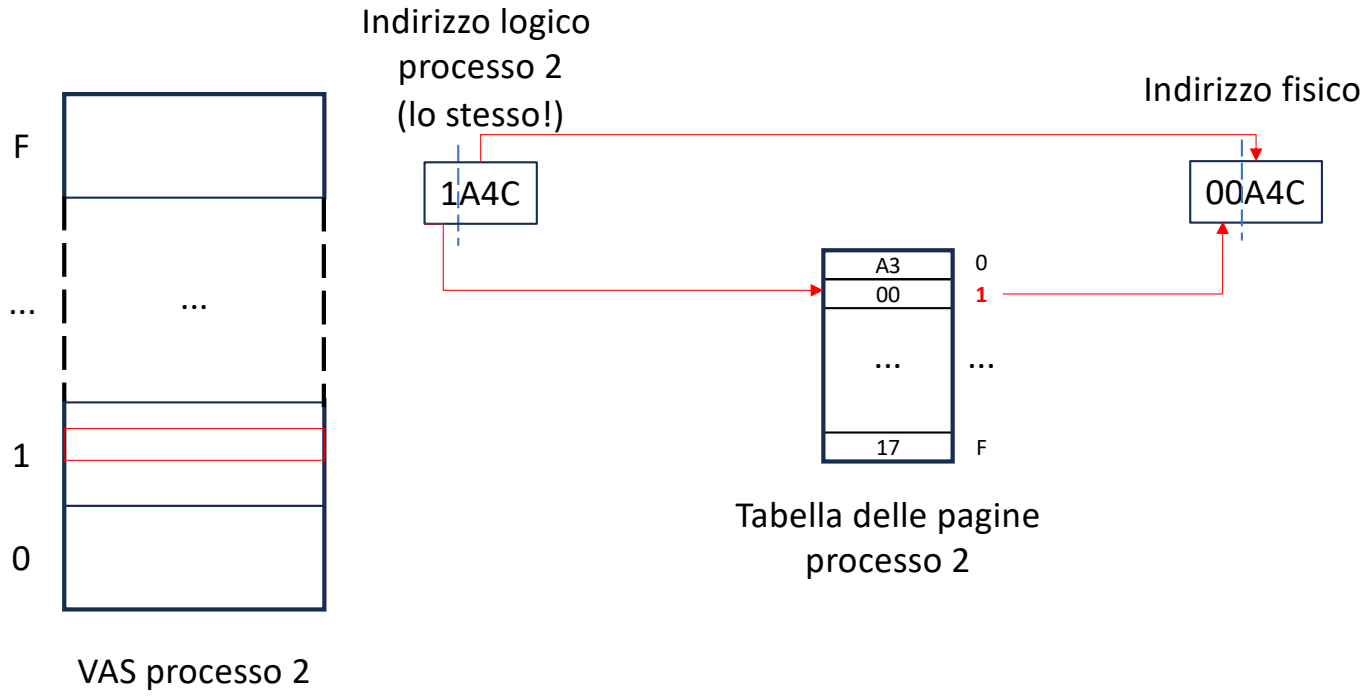
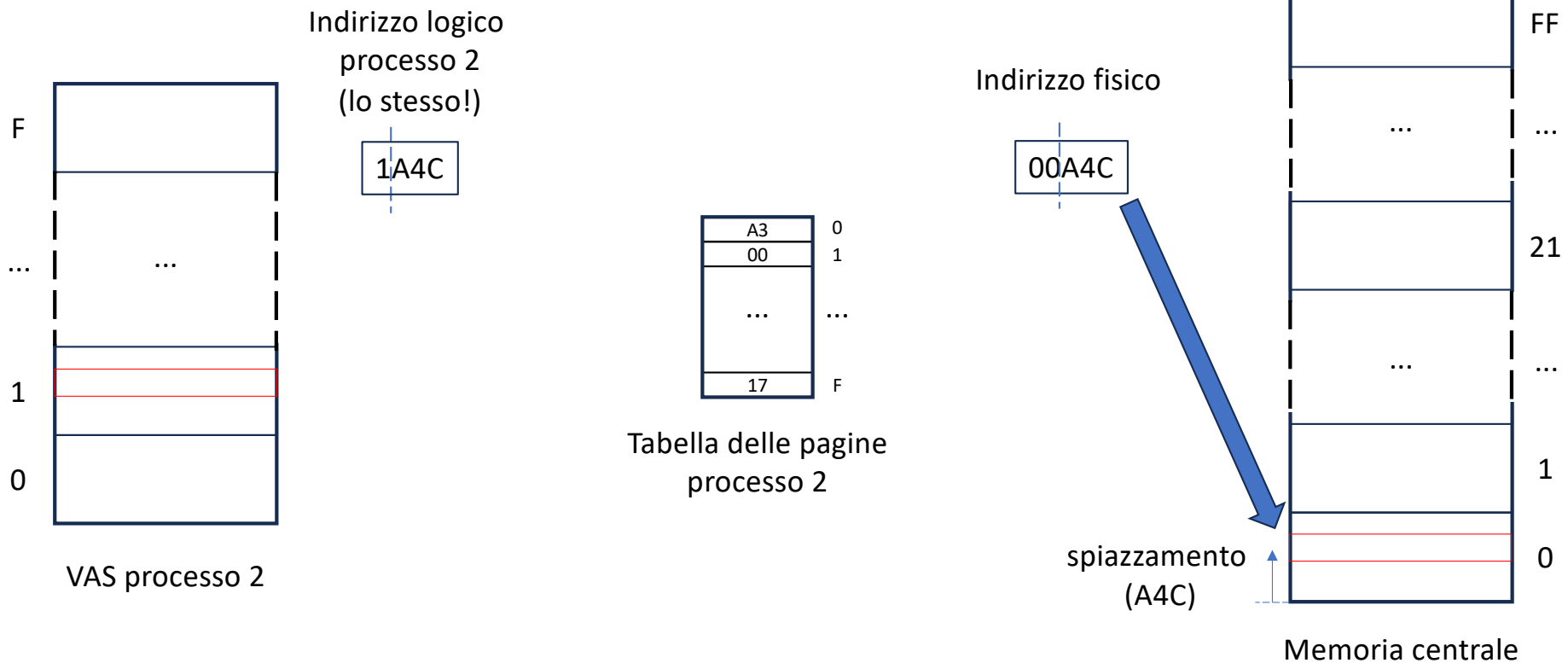


Tabella delle pagine
processo 2

Paginazione: un esempio



Paginazione: un esempio



Frammentazione interna e dimensione di pagina

- In un sistema con paginazione nel caso medio la frammentazione interna è di **mezzo frame per processo**
- Questo suggerisce di fare pagine di dimensioni ridotte per ridurre lo spreco di memoria
- Ma in tal caso aumenta il numero totale di pagine, e di conseguenza la dimensione della tabella delle pagine
- Per tale motivo nel tempo la tendenza è stata di avere pagine di dimensioni maggiori: ad esempio, Windows 10 supporta pagine di 4 KByte e pagine di 2 MByte

Supporto alla paginazione nel sistema operativo

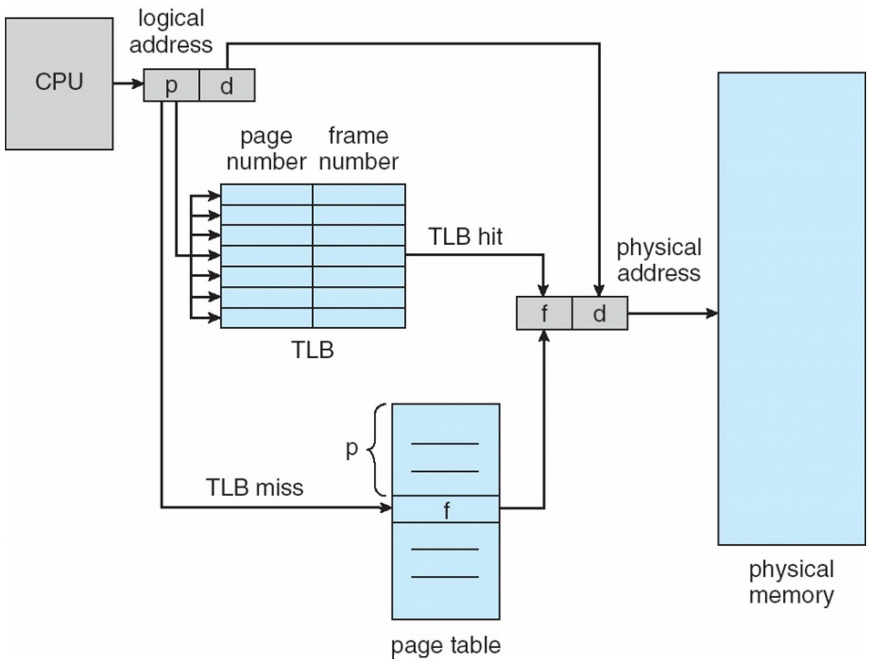
- Il sistema operativo deve mantenere la tabella delle pagine di ciascun processo
- Inoltre deve mantenere una **tabella dei frame**, che indica lo stato di ogni frame (libero o assegnato, e in tal caso a che processo/i)
- Quando la CPU è in modalità di sistema la MMU di regola non è attiva
  - Se il kernel deve accedere alla memoria di un processo, deve tradurre «manualmente» gli indirizzi logici del processo in indirizzi fisici
- Alcuni processori, al contrario, paginano anche lo spazio di indirizzi del kernel: in tal caso, il kernel ha una propria tabella delle pagine

Supporto hardware alla paginazione

- La tabella delle pagine corrente è mantenuta in memoria centrale (è di regola troppo grande per essere contenuta in un insieme di registri)
- La MMU utilizza due registri:
  - **Page table base register** (**PTBR**): indirizzo fisico dell'inizio della tabella delle pagine
  - **Page table length register** (**PTLR**): dimensione della tabella delle pagine
- In tale schema ogni accesso a dati/istruzioni richiede due accessi in memoria, uno alla tabella delle pagine, ed uno per recuperare il dato
- Essendo ciò troppo oneroso, i processori hanno una cache apposita per la tabella delle pagine corrente, il **translation lookaside buffer** (**TLB**)

Translation lookaside buffer

- Il TLB è di solito piccolo (tipicamente da 64 a 1024 entries)
- Principale problema: ad ogni cambio di contesto devo «azzerare» il TLB e riempirlo con le entry della tabella delle pagine del nuovo processo corrente, il che comporta una notevole riduzione di prestazioni
- Per tale motivo alcuni TLB memorizzano un **address space identifier** (**ASID**) nelle loro entry, che identificano univocamente uno spazio di indirizzi, così da poter mantenere le entry di più tabelle delle pagine al loro interno
- Gli ASID sono anche usati per implementare la protezione della memoria



Tempo effettivo di accesso alla memoria

- Ho un **TLB hit** se il numero di pagina di un indirizzo logico si trova nel TLB, altrimenti ho un **TLB miss**
- Il **tasso di successi** (**hit ratio**) è dato dalla percentuale di TLB hit sul totale degli accessi
- Supponendo che il tempo di accesso alla memoria sia *ma* e lo hit ratio sia *hr*, il **tempo medio di accesso alla memoria**, o **effective access time** (**EAT**) è:

$$EAT = ma\ hr + 2\ ma\ (1 - hr) = ma\ (2 - hr)$$

- (non teniamo conto delle maggiori o minori probabilità di cache hit o miss)
- Esempio: con *ma* = 100 nsec e *hr* = 97% otteniamo un *EAT* = 103 nsec

Politiche di sostituzione nel TLB

- Se capita un TLB miss ma il TLB è pieno, occorre sostituire un entry
- Diverse strategie:
  - Last recently used (LRU)
  - Round-robin
  - Casuale
- Alcuni processori generano un interrupt in maniera da permettere al sistema operativo di partecipare alla decisione di quale entry sostituire
- Alcuni processori permettono al sistema operativo di vincolare (wire down) delle TLB entry che non vogliono siano mai sostituite

Paginazione: protezione della memoria

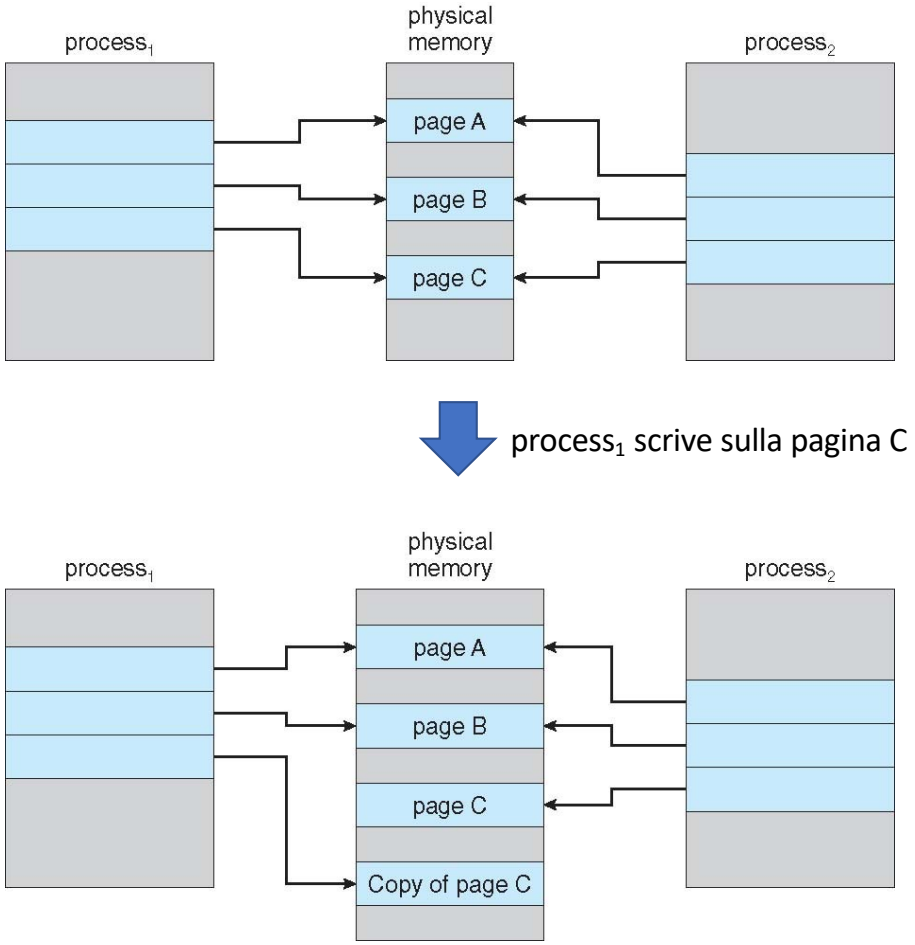
- Il registro PTLR permette di «tagliare» la tabella delle pagine in maniera da ridurre gli indirizzi logici disponibili al processo
- Vantaggi registro PTLR: si può ridurre la dimensione della tabella delle pagine
- Alternativa: mettere in ogni entry della tabella delle pagine un **bit di validità**, che indica se l'entry è valida o no (ossia, se la pagina è effettivamente mappata su un frame)
- Vantaggi bit validità: maggiore flessibilità in quanto posso creare intervalli di indirizzi logici inaccessibili in qualsiasi punto dello spazio di indirizzi logici
- Altri bit contenuti nella tabella delle pagine permettono di indicare se il frame associato è:
  - Read-only o read-write (**bit di protezione**)
  - Eseguibile o no (**bit di esecuzione**)

Pagine condivise

- Con la paginazione è facile condividere memoria fisica tra più processi: è sufficiente che i frame siano nelle tabelle delle pagine dei processi che li condividono
- Applicazioni:
  - Condividere il codice (in maniera read-only!) tra più processi, ad esempio il codice delle librerie di sistema, risparmiando così spazio in memoria
  - Realizzare comunicazione interprocesso tramite memoria condivisa

Copy-on-write

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



Strutture delle tabelle delle pagine per grandi spazi di indirizzamento virtuali

- Con spazi di indirizzamento virtuali grandi (32 o 64 bit) le tabelle delle pagine possono diventare a loro volta molto grandi:
  - Abbiamo detto che, se le pagine sono grandi 2^n e lo spazio di indirizzamento virtuale 2^m, le pagine sono in tutto 2^{m-n}
  - Supponendo che ogni entry occupa e bytes di dimensione, in totale la tabella delle pagine ha dimensione $e \cdot 2^{m-n}$
  - Esempio: in un processore a 32 bit con pagine di 4 KB (12 bit) ed entries di 4 bytes, la dimensione della tabella delle pagine è 4 MB
  - In un processore a 64 bit, a parità degli altri parametri, la tabella delle pagine (di un solo processo) dovrebbe avere dimensione di 16 PB!
- Per tale motivo la struttura della tabella delle pagine vista fino ad ora (tabella delle pagine **lineare**) diventa inapplicabile
- Esistono varie soluzioni per strutturare le tabelle delle pagine in maniera che siano sufficientemente piccole/efficienti, tra le quali:
  - Tabelle delle pagine gerarchiche
  - Tabelle delle pagine di tipo hash
  - Tabelle delle pagine invertite

Tabelle delle pagine gerarchiche

- Servono per evitare di allocare una tabella delle pagine in una regione di memoria contigua
- Idea: paginare la tabella delle pagine
- Tabella a due livelli:
  - Ogni numero di pagina è diviso a sua volta in un numero di pagina e in un offset
  - Questi sono utilizzati per recuperare da una tabella esterna delle pagine l'indirizzo della pagina della tabella delle pagine
  - Questa, infine, è utilizzata per costruire l'indirizzo fisico
- Schemi a più di due livelli sono possibili

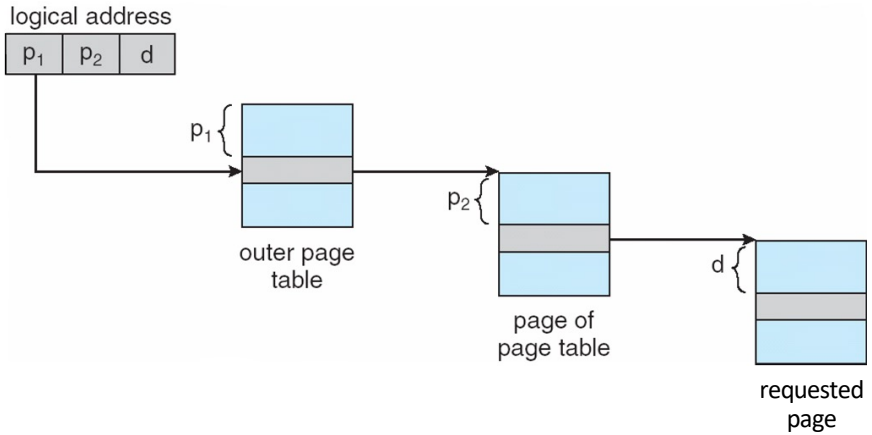


Tabelle delle pagine gerarchiche: vantaggi e svantaggi

- Un vantaggio è che la tabella delle pagine gerarchica permette di supportare contemporaneamente pagine di dimensioni diverse
  - Basta marcare un'entry nella tabella delle pagine esterne perché sia considerata un'entry di ultimo livello
  - Soluzione usata ad esempio nelle architetture IA-32 (a destra) e ARMv8
  - Permette di ridurre il numero di livelli (e quindi accessi in memoria dopo un TLB miss) e la dimensione della tabella
- Lo svantaggio principale è che aumenta il numero di accessi in memoria per recuperare un dato/istruzione nel caso pessimo
- Per tale motivo con i processori a 64 bit si tende a non usarle

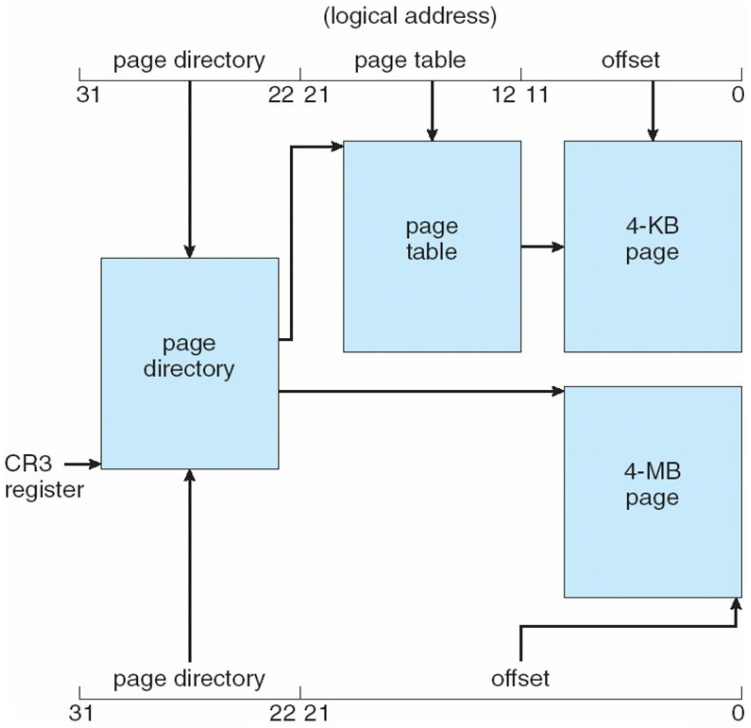


Tabelle delle pagine di tipo hash

- Idea: la tabella delle pagine è organizzata come una tabella hash
- Si applica una funzione hash (semplice!) al numero di pagina
- Ogni entry della tabella ha una lista concatenata di elementi per gestire le eventuali collisioni

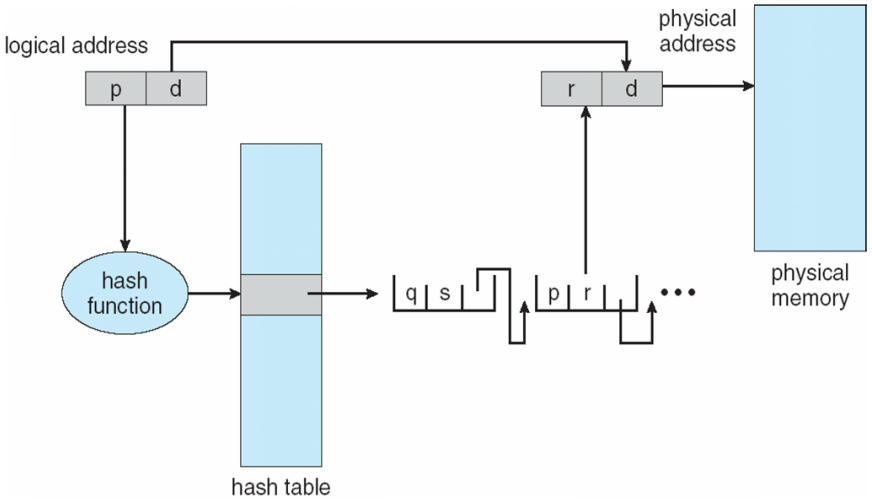
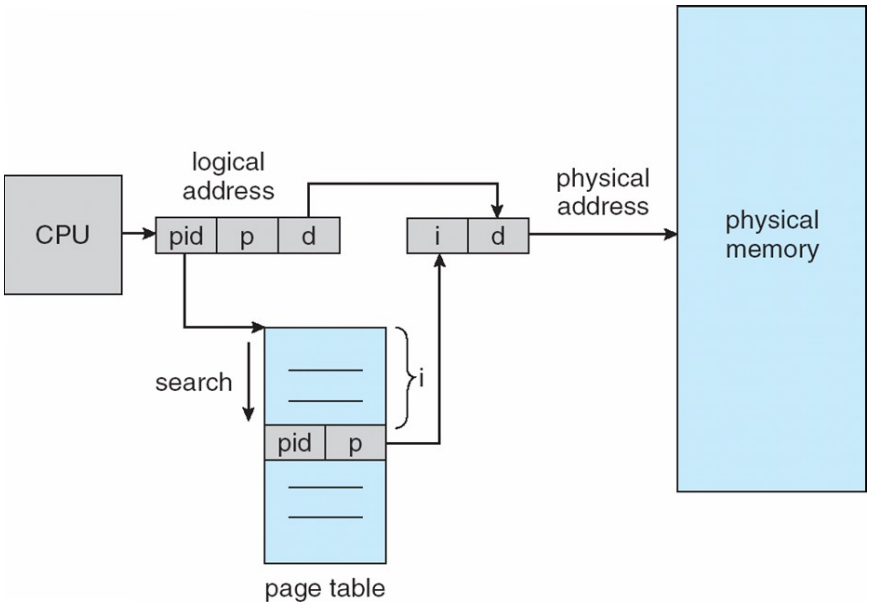


Tabelle delle pagine invertite

- Idea: un entry per ogni frame, anziché per ogni pagina (vengono tracciati i frame anziché le pagine)
- Ogni entry riporta il numero di pagina del corrispondente frame, più informazioni addizionali tra cui L'ASID (necessario!)
- Vantaggi:
  - Una sola tabella per tutti i processi
  - Se lo spazio di indirizzi fisici è più piccolo dello spazio di indirizzi virtuali, c'è un ulteriore risparmio
- Svantaggi:
  - Maggiore tempo di accesso (occorre fare una ricerca per trovare l'entry); mitigabile con tabella hash e TLB
  - Non è possibile condividere pagine tra processi diversi
- Utilizzata nelle CPU POWER e UltraSPARC 64



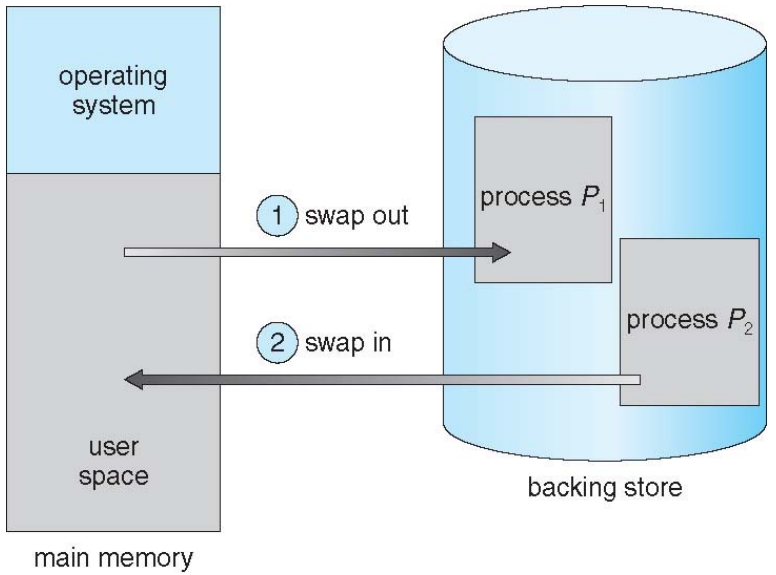
Swapping

Swapping

- Se la memoria è poca rispetto al numero dei processi, pochi processi possono essere ammessi all'esecuzione
- Lo swapping è una tecnica che permette di eseguire più processi di quanti la memoria fisica ne possa contenere
- Idea: spostare temporaneamente l'immagine di un processo pronto o in attesa dalla memoria centrale in una memoria secondaria (**backing store**); diciamo che tale processo viene **sospeso**
- L'obiettivo è permettere ad un altro processo di andare in memoria, e quindi in prospettiva di andare in esecuzione
- Possibili varianti:
  - Swapping standard (quello che di solito si intende con «swapping»)
  - Swapping con paginazione (quello che di solito si intende con «paginazione»)

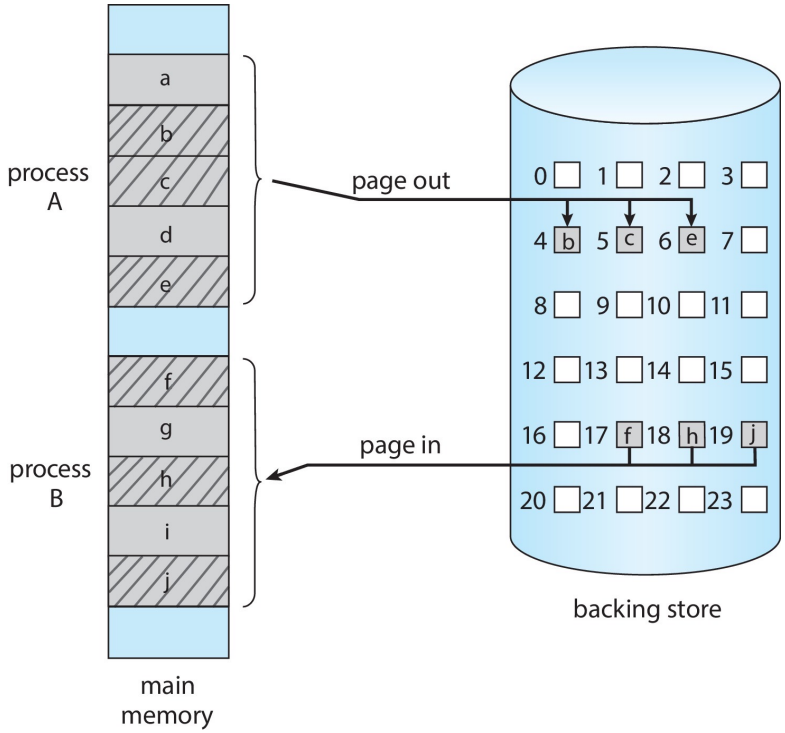
Swapping standard

- Un intera immagine viene spostata dalla memoria centrale alla backing store (**swap out**)
- Occorre spostare anche tutte le strutture dati del sistema operativo relative al processo e ai threads
- Svantaggio: spostare un intero processo è molto oneroso (solo Solaris usa ancora lo swapping standard in circostanze estreme)



Swapping con paginazione

- Sposta solo un sottoinsieme delle pagine di un processo, fino a quando non vi è sufficiente memoria per caricare l'immagine del nuovo processo
- Molto meno oneroso dello swapping standard
- L'operazione di scaricamento di una pagina dalla memoria centrale è detta **page out**, l'operazione inversa è detta **page in**



Grado di multiprogrammazione e utilizzazione dei processori

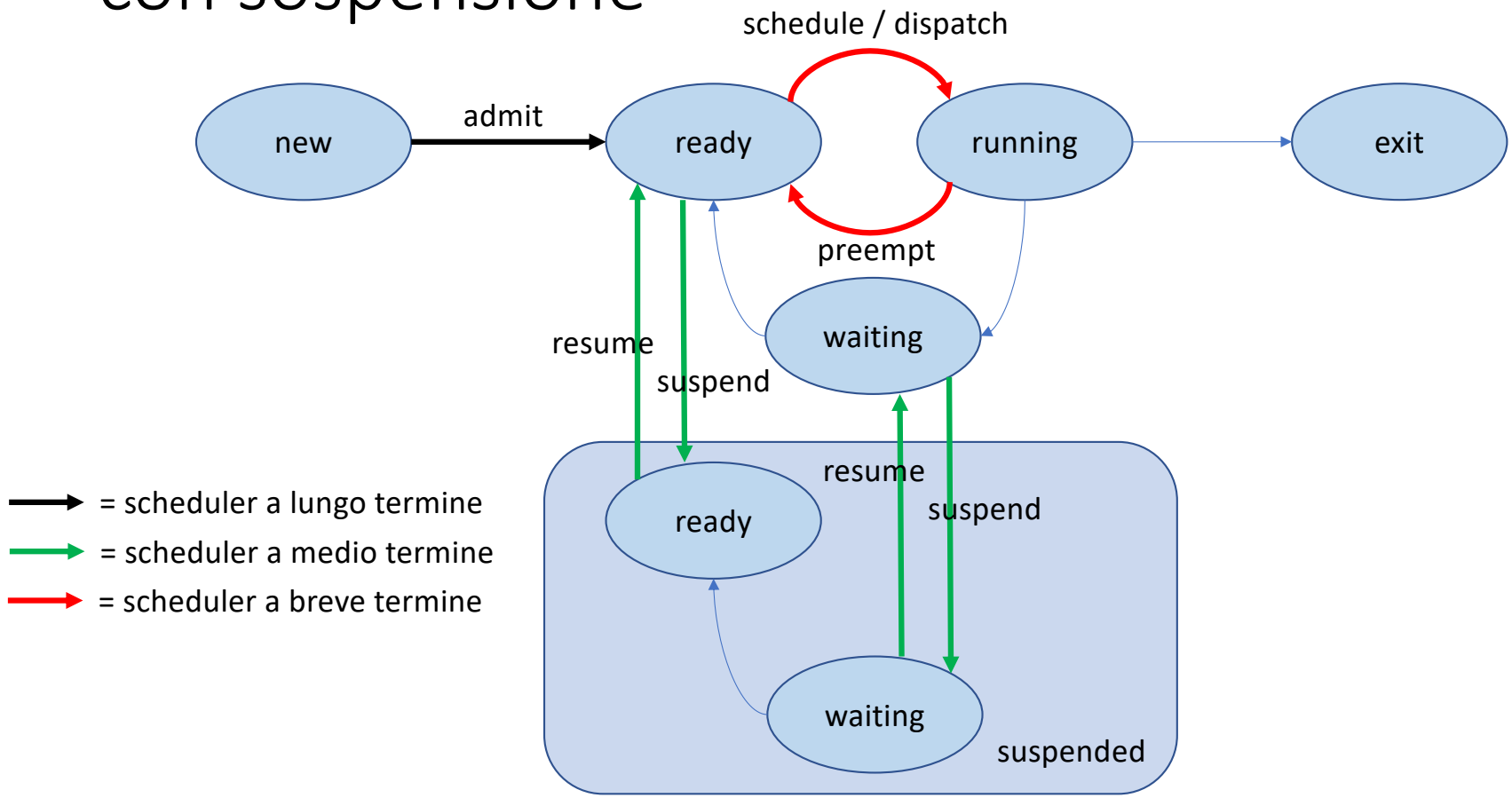
Grado di multiprogrammazione

- Definiamo come **grado di multiprogrammazione** il numero massimo di processi che lo scheduler della CPU può mandare in esecuzione (ready + running + waiting)
- Il grado di multiprogrammazione determina l'occupazione totale di memoria fisica da parte dei processi:
  - In assenza di memoria virtuale, quando un processo può essere mandato in esecuzione solo se la sua immagine è interamente in memoria, il grado di multiprogrammazione è uguale al numero di immagini in memoria
  - In generale, chiamiamo **processi in memoria** l'insieme dei processi ready + running + waiting e **processi fuori memoria** tutti gli altri i processi: il grado di multiprogrammazione è il numero di processi in memoria
- Il grado di multiprogrammazione è correlato anche con l'utilizzazione delle CPU:
  - Se è basso anche l'utilizzazione tende ad essere bassa, perché quando i processi vanno in attesa non ci sono abbastanza processi ready per tenere occupati i processori
  - Al crescere del grado di multiprogrammazione aumenta il numero di processi che, in media, sono ready, e quindi anche l'utilizzazione dei processori

Controllo del grado di multiprogrammazione

- Lo **scheduler a lungo termine** (o admission scheduler) decide se ammettere un nuovo processo dopo la sua creazione nella ready queue: in questo modo influisce sul grado di multiprogrammazione sul lungo periodo
- Lo **scheduler a medio termine** sospende un processo per farne swapping e portarlo fuori memoria, e viceversa fa riprendere dalla sospensione un processo riportandolo in memoria: in questo modo influisce sul grado di multiprogrammazione sul medio periodo
- Lo **scheduler di breve termine** (scheduler della CPU) non influisce sul grado di multiprogrammazione

Diagramma di transizione di stato dei processi con sospensione



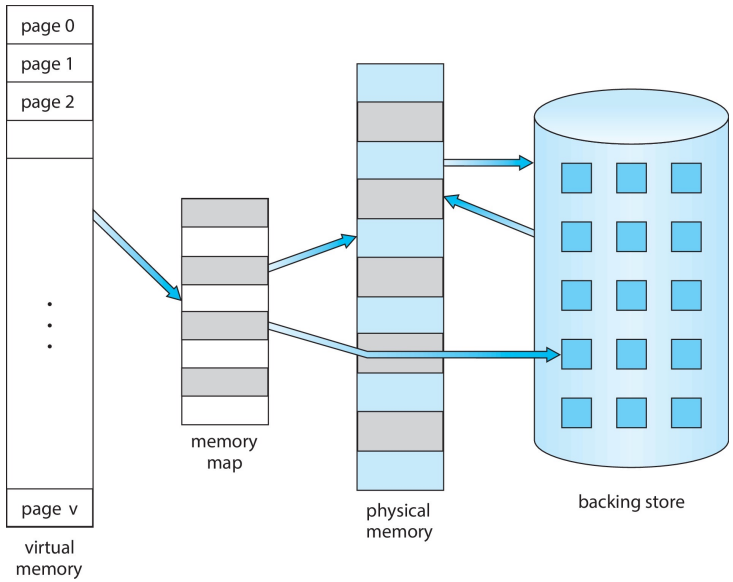
Memoria virtuale

Motivazioni

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

Memoria virtuale

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



Paginazione su richiesta

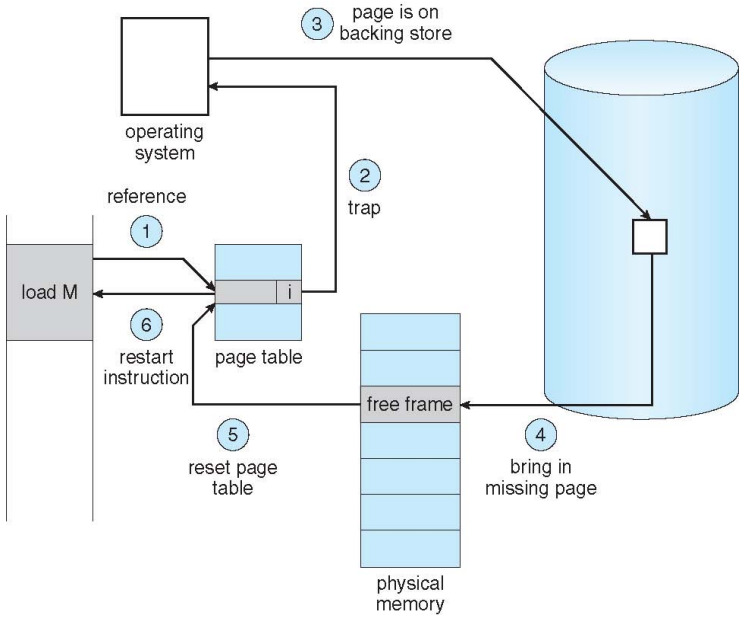
- La **paginazione su richiesta** (demand paging) è basata sull'hardware di paginazione
- Idea: non portare l'intero processo in memoria alla creazione, ma solo le pagine che vengono volta per volta usate
  - Una pagina viene caricata in memoria quando viene utilizzata durante l'esecuzione del programma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

Supporto HW alla paginazione su richiesta

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

Gestione di un page fault

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



Commenti

- Politiche di caricamento pagina:
  - Paginazione su richiesta **pura:** le pagine vengono caricate solo quando servono (ad esempio, all'avvio di un processo non viene caricata alcuna pagina)
  - **Prefetching**: vengono caricate anche altre pagine in un «intorno» della pagina richiesta
- La riesecuzione dell'istruzione interrotta da un page fault può essere complessa quando l'istruzione occupa/modifica più parole di memoria su pagine diverse (cosa particolarmente vera per i processori CISC)
  - Ad esempio, l'IBM 370 aveva un'istruzione macchina SS MOVE lunga 6 byte, che quindi poteva esistere a cavallo di due frame
  - Aveva due operandi a doppia indirezione alla memoria, quindi nel caso pessimo effettuava 4 accessi in memoria per ottenere gli operandi
  - In totale l'esecuzione di tale istruzione, nel caso pessimo, poteva generare 6 page fault
  - Altre CPU hanno istruzioni la cui esecuzione può generare ancora più page faults!

Prestazioni della paginazione su richiesta

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

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

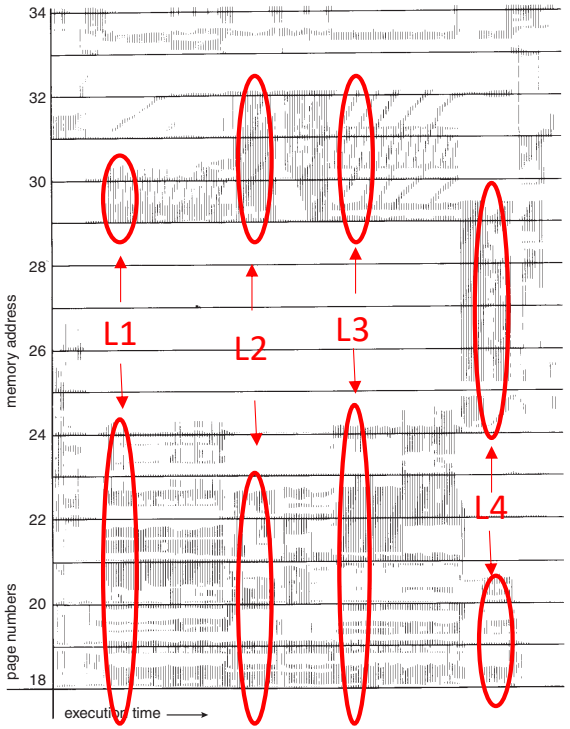
- Notare che *pf* è composto da servizio eccezione + caricamento pagina da backing store + ripristino processo, ma è dominato dal tempo di caricamento pagina da backing store
- Ad esempio, per *pf* = 5 msec e *ma* = 100 nsec, se vogliamo un rallentamento massimo del 10% occorre che i page fault siano meno di uno ogni 500.000 accessi a memoria

Un po’ di numeri...

- Tempo medio di accesso:
  - Cache liv. 1: ~1 nsec
  - Cache liv. 2: ~10 nsec
  - Cache liv. 3: ~20 nsec
  - DRAM: ~100 nsec
- Tipici TLB hit rate: 95-99%
- Tempo gestione page fault:
  - Minor page fault (pagina in memoria): ~10 µsec
  - Major page fault (SSD): ~500 µsec
  - Major page fault (HDD): ~10 msec

Il modello di località

- La paginazione su richiesta è efficiente se i page fault sono pochi rispetto alle istruzioni eseguite, cosa di regola garantita dal fenomeno della **località dei riferimenti**
- Secondo il **modello di località** in un certo momento un processo opera su una certa **località**, ossia su un certo sottoinsieme di pagine
- I processi nel tempo «migrano» da una località all'altra
- Località diverse possono sovrapporsi
- Se ogni processo riesce a tenere in memoria, in ogni istante, la sua località, i page fault sono pochi, e sono concentrati nei momenti in cui il processo migra di località
- Ma se il grado di multiprogrammazione è troppo alto, e non si riesce a tenere in memoria le località di tutti i processi, il sistema può cadere nel **thrashing** (continui page fault)



Gestione dei frame liberi

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