Gestione della memoria

Il sistema operativo svolge il duplice ruolo di

- **gestore delle risorse**: decide quali risorse assegnare ai diversi processi
- **controllore**: assicura che le risorse assegnate ad un altro processo non possano essere usate / alterate dagli altri processi

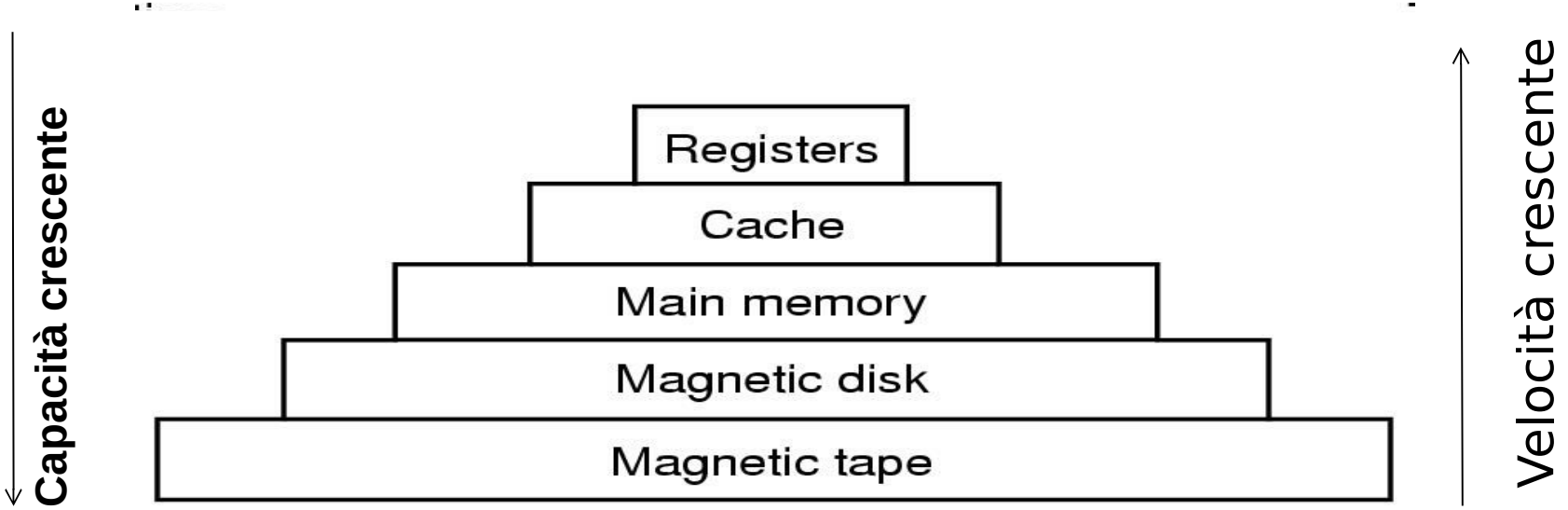
Tra le risorse da assegnare ai processi vi è la *memoria che dovrebbe essere:*

- *ampia*
- *veloce*

Ma non si può avere tutto, perciò si usa una gerarchia di memorie

- *cache: memoria di capacità ridotta, molto veloce e costosa*
- *RAM: memoria di capacità dell'ordine di alcuni Gb, velocità media e prezzo medio*
- *Dischi (o memorie di massa a stato solido): centinaia di Gb di memoria ad alta capacità, più lenta ed economica*

Gestione della memoria



L'idea della gerarchia della memoria consiste nel portare verso il vertice della piramide le informazioni più utilizzate dalla porzione di programma in esecuzione (che possono cambiare nel tempo) così che nella maggior parte dei casi il tempo di accesso coincide con quello delle memorie più rapide.

L'aggiornamento del contenuto della cache è gestito per lo più in hardware. Se *l'hit rate* è alto (che non è improbabile per via della *località* degli accessi: un processo accede ripetutamente ad alcuni indirizzi), si ha un tempo medio di accesso alla memoria intermedio tra quello della RAM e quello della cache

Il SO fa un lavoro simile fra RAM e memoria di massa

Indirizzi logici e fisici

Per poter allocare liberamente i processi in memoria (RAM) occorre distinguere tra *indirizzi logici* (che verrebbero utilizzati per accedere in RAM se non ci fosse la multiprogrammazione) e *indirizzi fisici* (quelli effettivamente utilizzati per accedere alla RAM)

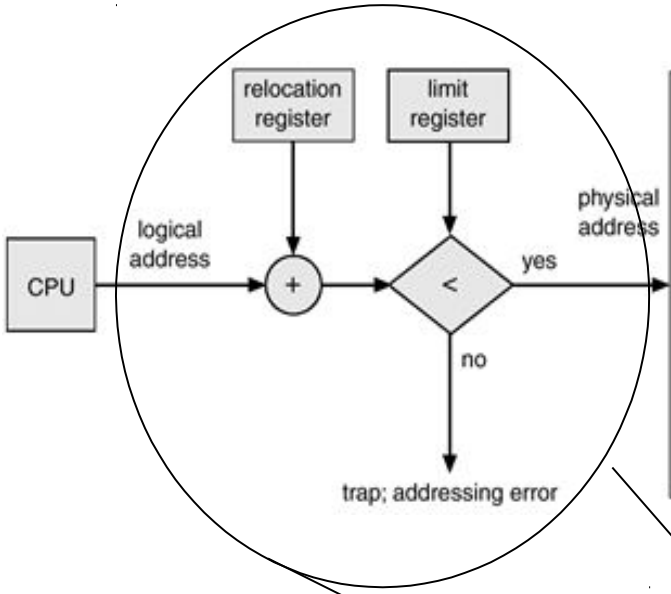
Il *binding* (associazione) tra indirizzi logici e indirizzi fisici può avvenire in momenti diversi: durante la compilazione, durante il caricamento oppure a run-time. Noi considereremo quest'ultimo approccio, che è il più flessibile (permette di spostare un processo da una parte all'altra della RAM anche durante l'esecuzione).

In questo caso quando nel file eseguibile c'è un indirizzo (operandi, destinazioni dei salti), è un indirizzo logico e una volta caricato il programma in RAM, gli indirizzi degli operandi, delle istruzioni (valore del PC, destinazioni dei salti), i valori delle variabili puntatore..., tutti gli indirizzi generati dal programma sono indirizzi logici

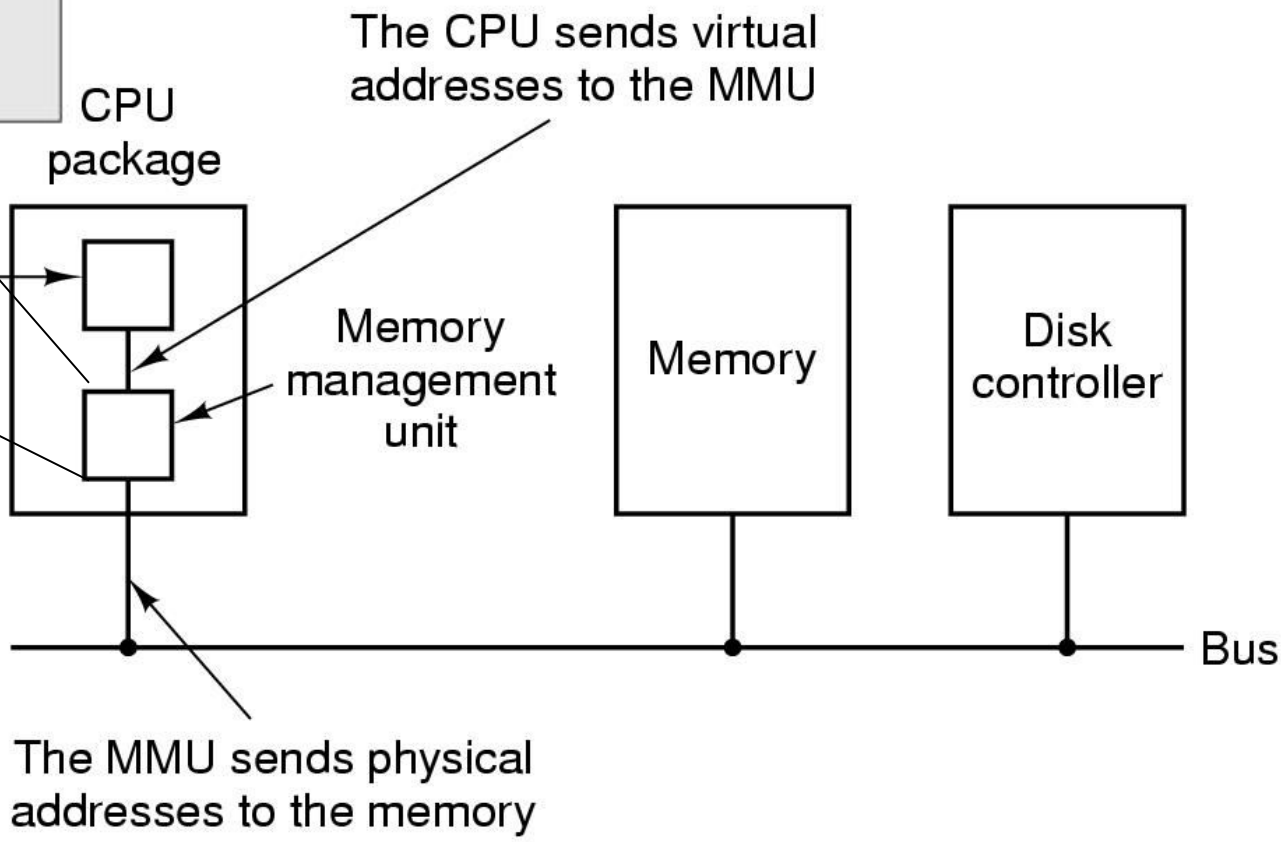
La MMU (Memory Management Unit) si occupa di operare la traduzione al momento dell'accesso in memoria.

La MMU

La MMU realizza la traduzione da indirizzo logico (anche detto indirizzo virtuale) a indirizzo fisico (quello da inviare alla RAM).



Esempio di lavoro della MMU con registro base (relocation register) e limite, per *allocazione contigua*



Rilocabilità e protezione dei sistemi multiprogrammati

Sempre nell'esempio, con:

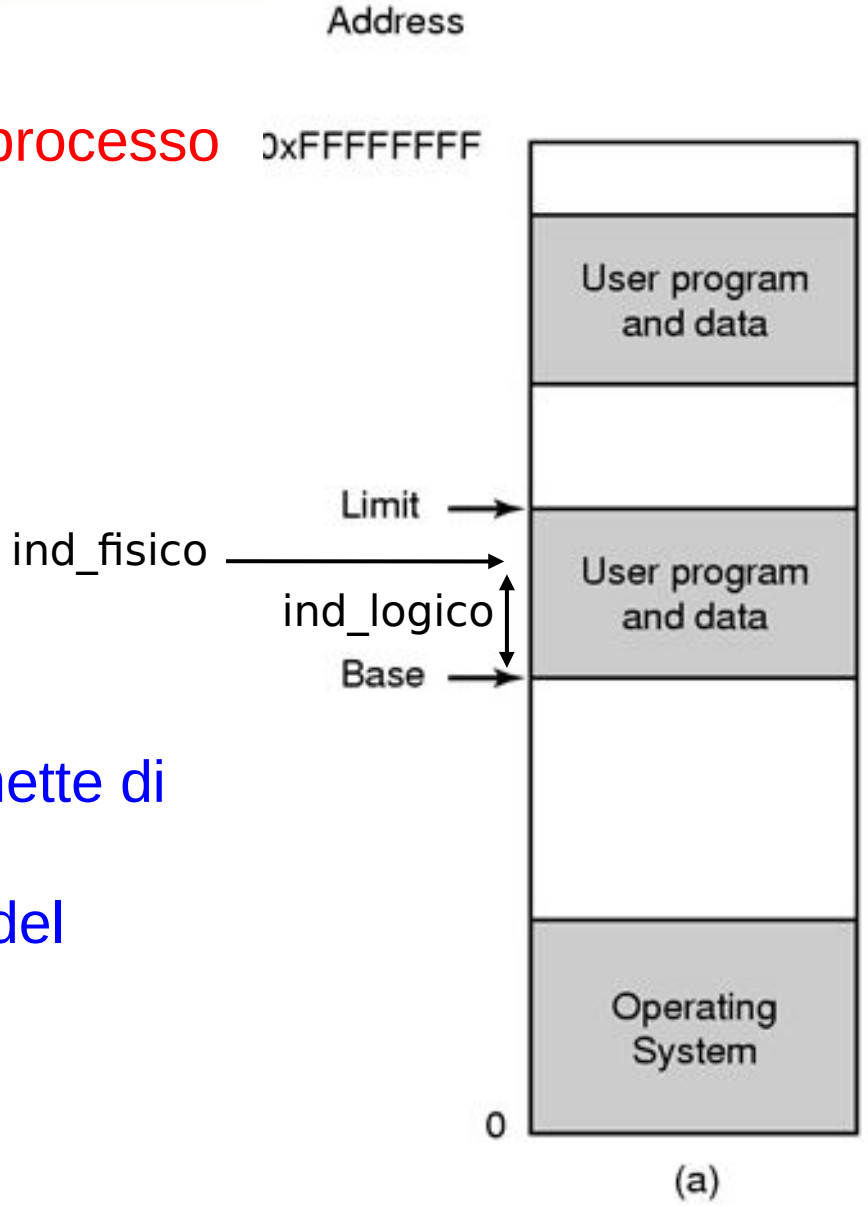
- allocazione contigua dell'immagine del processo
- un registro base e un registro limite

la MMU svolge in hardware la funzione:

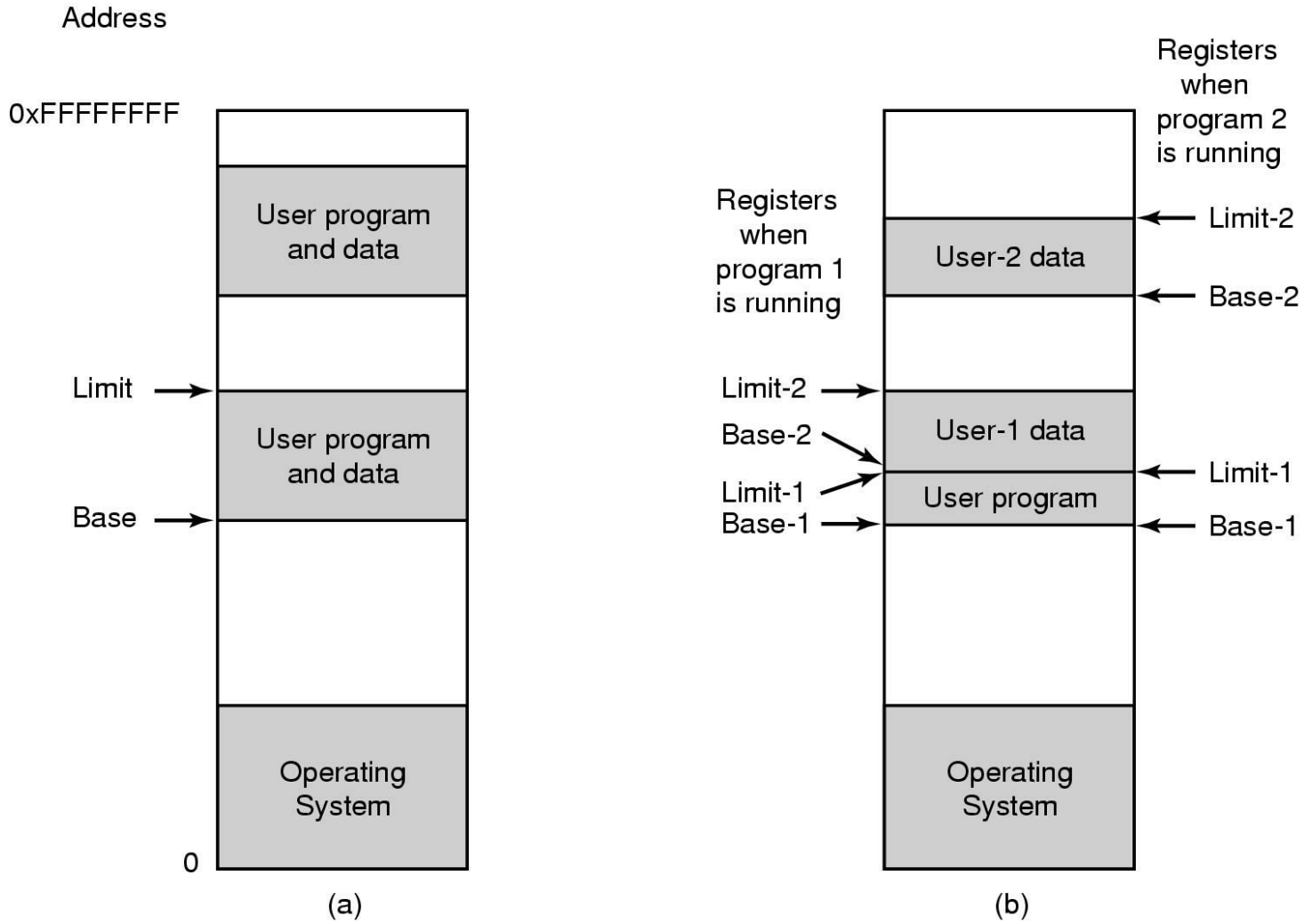
if (base + ind_logico >limite) *then*
trap("Segmentation" fault)

else ind_fisico = base + ind_logico

che oltre alla traduzione logico-fisico permette di confinare un processo allo spazio ad esso assegnato garantendo così la protezione del sistema operativo e degli altri processi



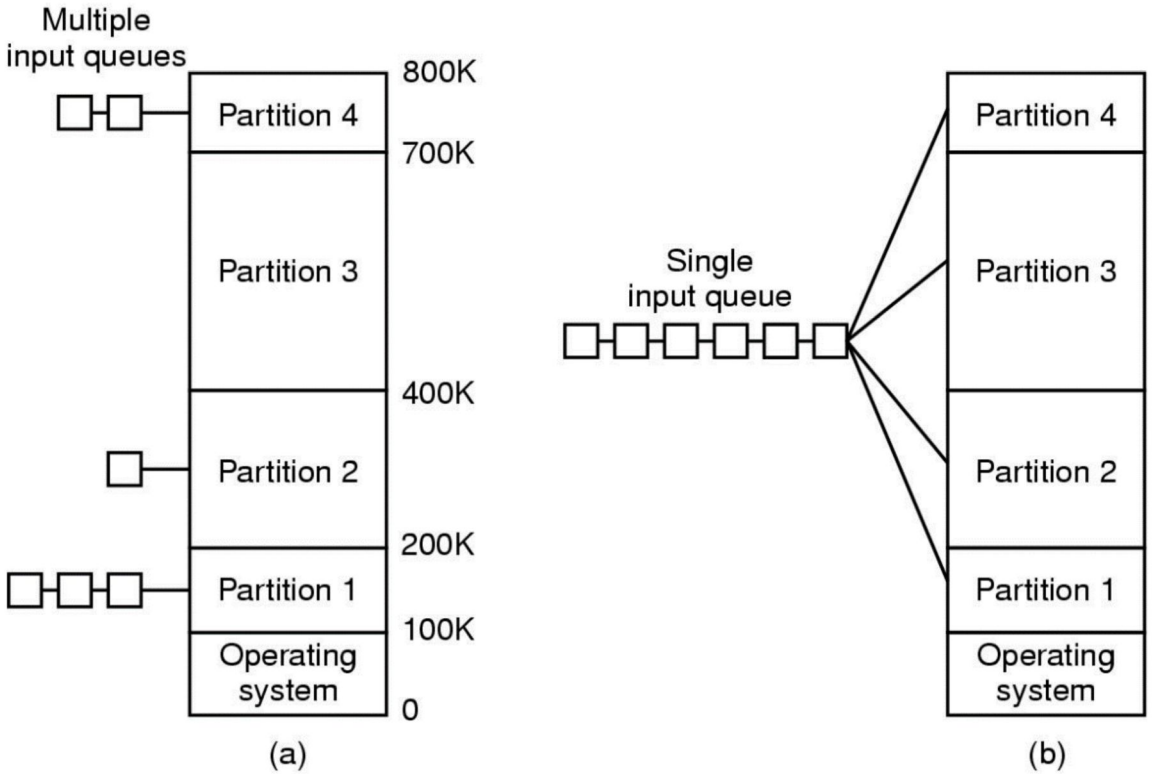
Rilocabilità e protezione tramite registri Limit e Base



I registri Limit e Base possono essere modificati solo in modo kernel (quindi solo dal S.O.). La traduzione indirizzo logico-indirizzo fisico e il controllo di non superamento dei limiti viene fatta dalla MMU. Il valore di base e limite di ogni processo sono memorizzati nel PCB.

Allocazione spazio in memoria: partizioni fisse

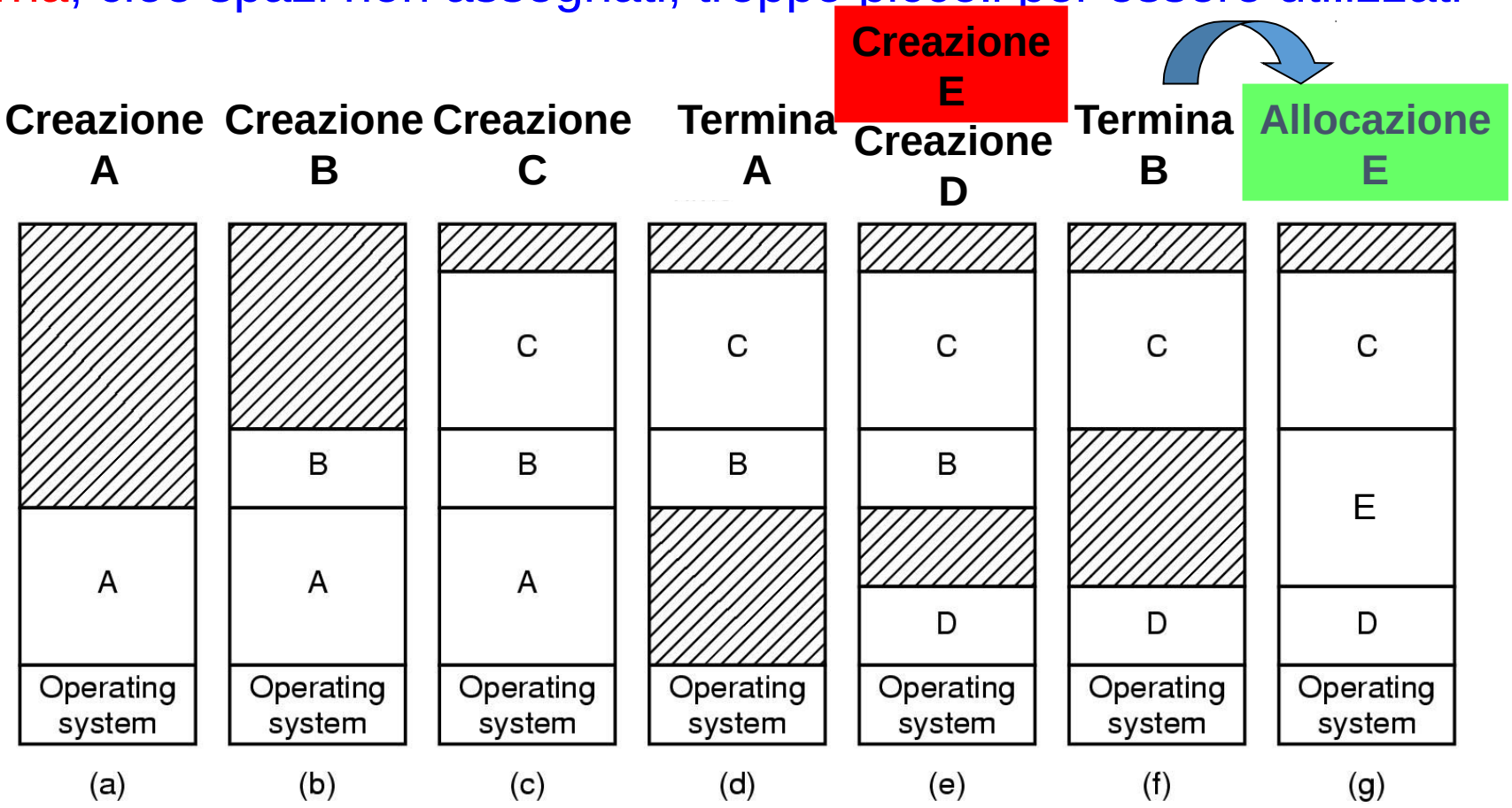
Gestione dello spazio in RAM nei sistemi multiprogrammati (batch): i job sottoposti al sistema sono caricati su disco; se ne tiene traccia in una coda, caricandoli in memoria quando si libera una partizione



La suddivisione in partizioni è configurata dal sistemista all'avvio del sistema. Le code possono essere separate per partizione o congiunte. Può esserci spreco di memoria, in particolare frammentazione interna: parte della memoria assegnata a un processo non è utilizzata. Occorre una politica di assegnazione delle partizioni ai job.

Allocazione spazio in memoria: partizioni variabili

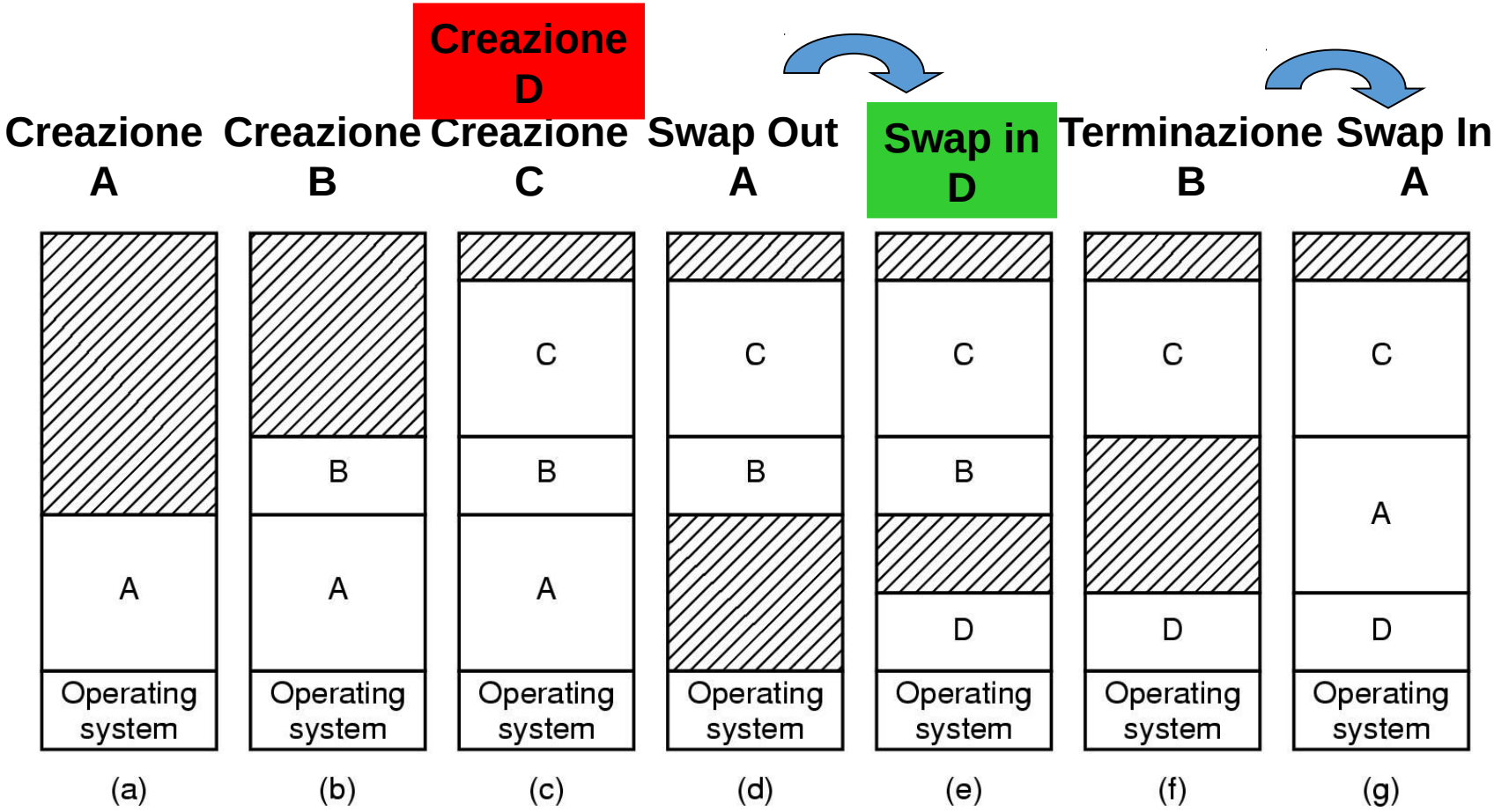
Una gestione un po' più flessibile: si cerca uno spazio libero contiguo abbastanza grande, e vi si carica il processo. Con la creazione o terminazione di processi si creano dei "buchi" non contigui di grandezza variabile. Può verificarsi una situazione in cui un nuovo processo deve essere creato, in totale ci sarebbe abbastanza memoria per caricarlo, ma questa è formata da molti frammenti non contigui: si ha frammentazione esterna, cioè spazi non assegnati, troppo piccoli per essere utilizzati



Allocazione spazio in memoria: partizioni variabili

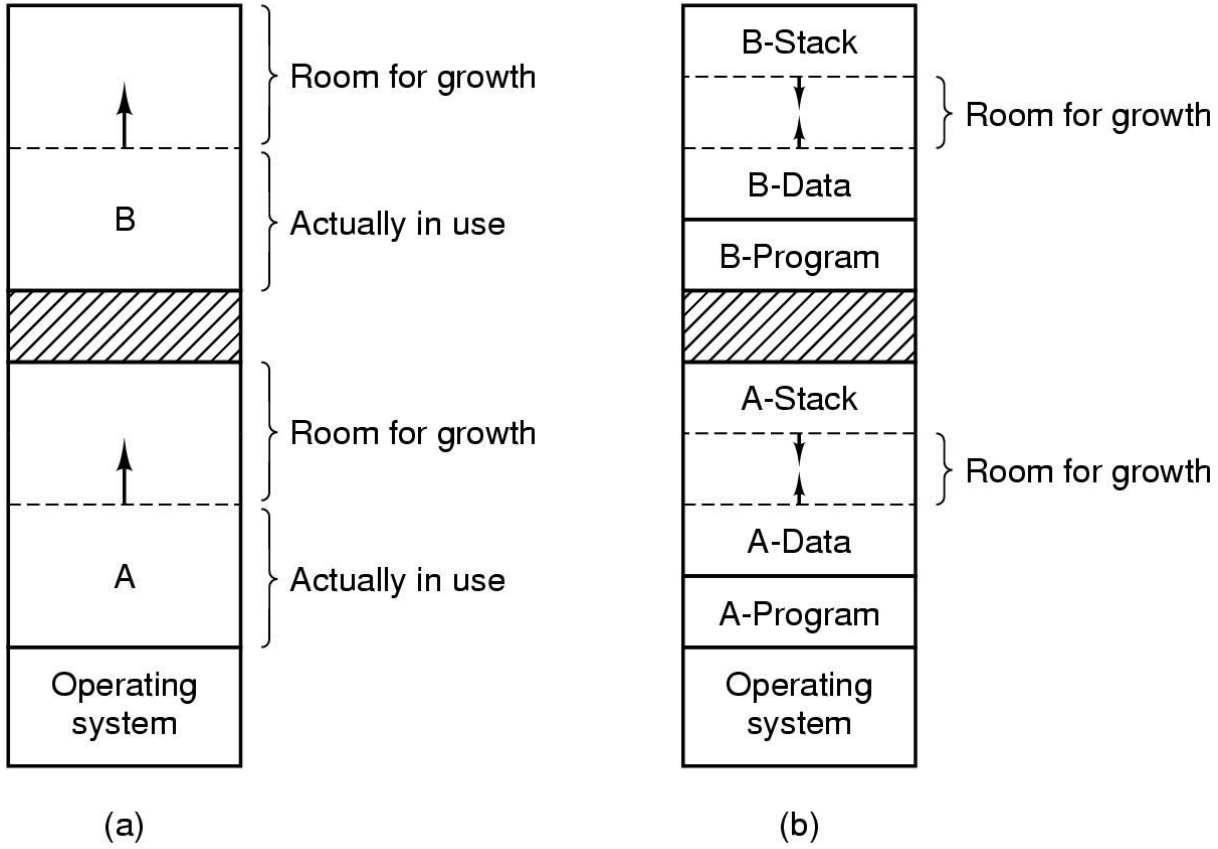
Come affrontare la frammentazione esterna (costoso):

- Effettuare periodicamente una compattazione della memoria
- Utilizzare il disco come “deposito temporaneo” di processi effettuando lo *swapping:* lo *swap out* di un processo per fare spazio ad uno nuovo lo *swap in* quando si libera nuovamente dello spazio (per la terminazione di un processo o perchè ho creato spazio facendo *swap out* di un altro)



Processi a dimensione variabile

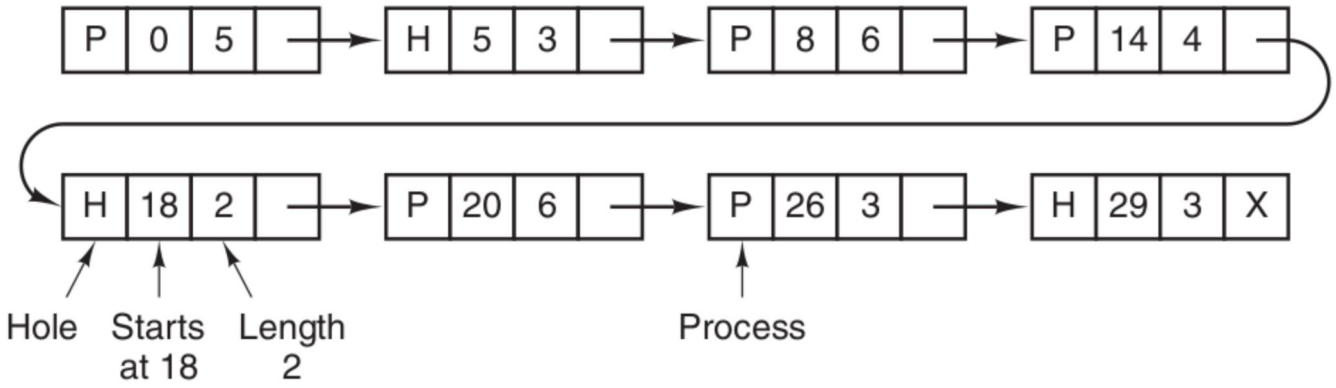
Se vi sono strutture dati che possono crescere (come la heap dove allocare dinamicamente strutture dati o lo stack) conviene allocare fin dall'inizio più spazio di quanto richiesto inizialmente. Se si eccede anche quello, o c'è un buco contiguo, oppure devo spostare il processo in uno spazio più grande, o ancora fare swap out. Nel peggiore dei casi si uccide il processo



Algoritmi di allocazione nello spazio con partizioni variabili

Se ci sono più spazi disponibili grandi abbastanza per caricare un nuovo processo, quale si sceglie?

- *First fit*: il primo grande abbastanza a partire dall'inizio della lista degli spazi liberi (bassi tempi di ricerca).
- *Next fit*: il primo grande abbastanza a partire dal punto della lista dove si era fermata l'ultima ricerca (idem).
- *Best fit*: lo spazio che meglio si adatta alle dimensioni del processo (tale che la differenza fra la dimensione del processo e lo spazio libero sia minima). Evita di occupare spazi grandi che potrebbero servire dopo per allocare processi più grandi. Crea rimanenze piccole pere ssere utilizzate.
- *Worst fit*: lo spazio che peggio si adatta alle dimensioni del processo (la differenza fra la dimensione del processo e lo spazio libero è massima). Serve per lasciare spazi abbastanza grandi da essere utilizzabili.



Algoritmi di allocazione nello spazio con partizioni variabili

In memoria sono presenti i seguenti spazi liberi:

100K, 500K, 200K, 300K, 600K (ordinati in base all'indirizzo di partenza)

Come verrebbero allocati i seguenti processi: P1 (212K), P2 (417K), P3 (112K), P4 (426K), utilizzando gli algoritmi First Fit, Best Fit e Worst Fit?

First Fit: P1 in 500K (rimane 288K), P2 in 600K (rimane 183K)
P3 in 288K (rimane 176K), P4 deve attendere

Best Fit: P1 in 300K (rimane 88K), P2 in 500K (rimane 83K)
P3 in 200K (rimane 88K), P4 in 600K (rimane 174K)

Worst Fit: P1 in 600K (rimane 388K), P2 in 500K (rimane 83K)
P3 in 388K (rimane 276K), P4 deve attendere

Quindi la Best è la best? No, in generale non c'è una politica "buona"

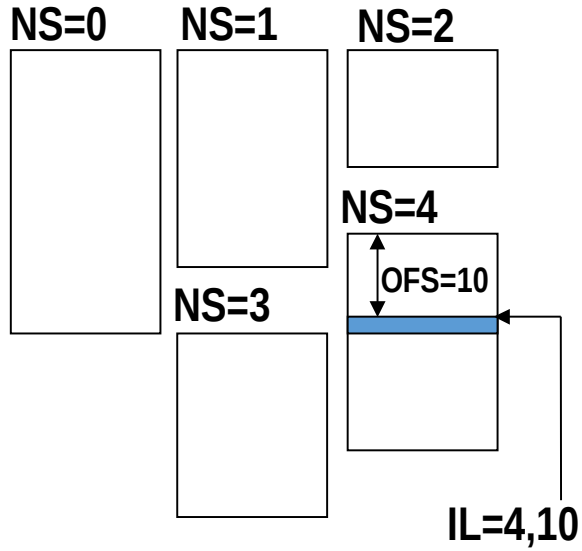
Allocazione non contigua

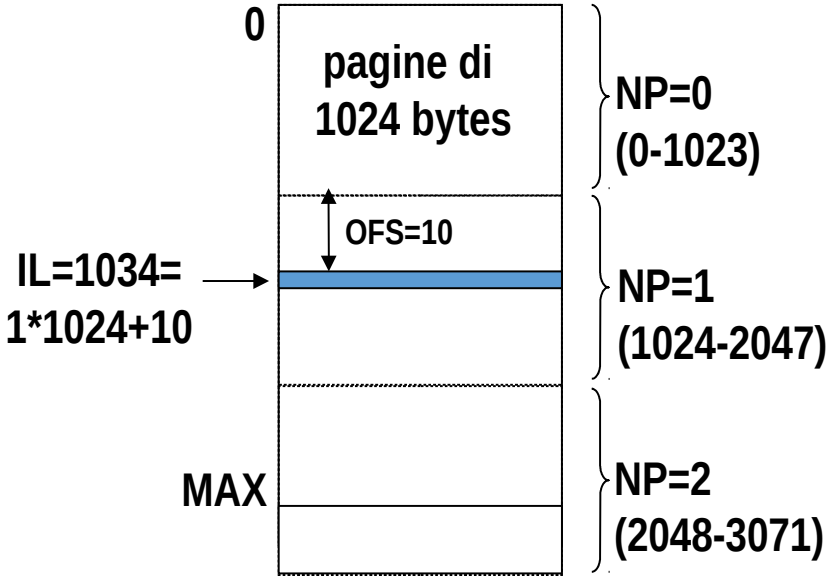
Schemi alternativi di allocazione dei processi in memoria prevedono di poter suddividere l'immagine di un processo in porzioni:

- di dimensione qualsiasi (segmenti) oppure
- di dimensione predefinita (pagine)

Segmentazione: l'immagine del processo è costituita da N spazi lineari di indirizzi (i segmenti). L'indirizzo logico è formato da una coppia (NUM.SEGMENTO,OFFSET)

Paginazione: l'immagine del processo è costituita da N pagine di dimensione prefissata. L'indirizzo è in un unico spazio di indirizzi, ma può essere scomposto in NUMERO DI PAGINA, OFFSET





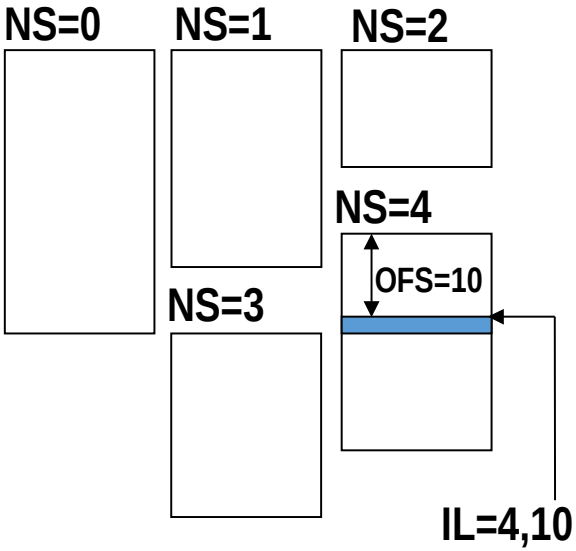
Allocazione non contigua

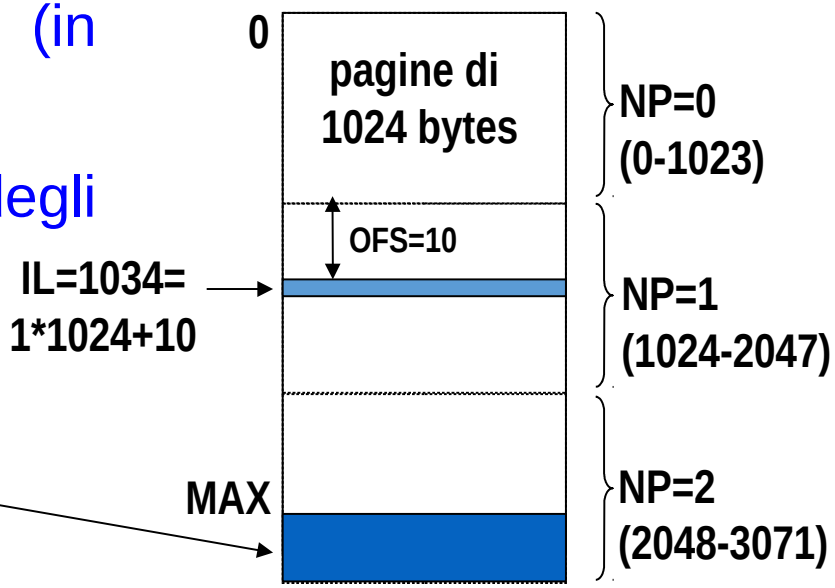
Segmenti e Pagine vengono allocati in memoria in posizioni non necessariamente contigue, migliorando l'utilizzazione della memoria:

- Con la segmentazione c'è frammentazione esterna, ma in misura minore rispetto all'allocazione contigua: è più facile trovare posto per un segmento che per un intero processo
- Con la paginazione c'è frammentazione interna, ma limitata a mezza pagina (in media) per processo

Ma non gratis… si complica la traduzione degli indirizzi

Nel seguito tratteremo solo la paginazione





La memoria virtuale (creare l'illusione di una RAM più grande)

Una volta che l'immagine del processo è suddivisa in parti, si può fare anche altro:

utilizzare il disco per contenere l'intera immagine del processo (e di tutti i processi in vita nel sistema) e caricare in RAM solo una parte dell'immagine del processo, quella effettivamente utile alla sua esecuzione nella fase attuale. In RAM si può mantenere una porzione più grande del processo di quanta se ne tiene in cache. Lo spostamento da disco a RAM e viceversa avviene a blocchi (pagine o segmenti).

Questo permette di eseguire uno o più processi la cui dimensione, o la somma delle cui dimensioni, è maggiore di quella della RAM

Perché ha senso la memoria virtuale

Alcune parti dell'immagine di un processo possono, in un'alta percentuale delle esecuzioni del programma, non essere mai necessarie:

- routine che vengono chiamate raramente: funzionalità di un programma utilizzate poco di frequente dall'utente, routine di gestione di errori
- strutture dati (es. array) allocate di grandi dimensioni ma, in molte esecuzioni, utilizzate soltanto per una piccola parte

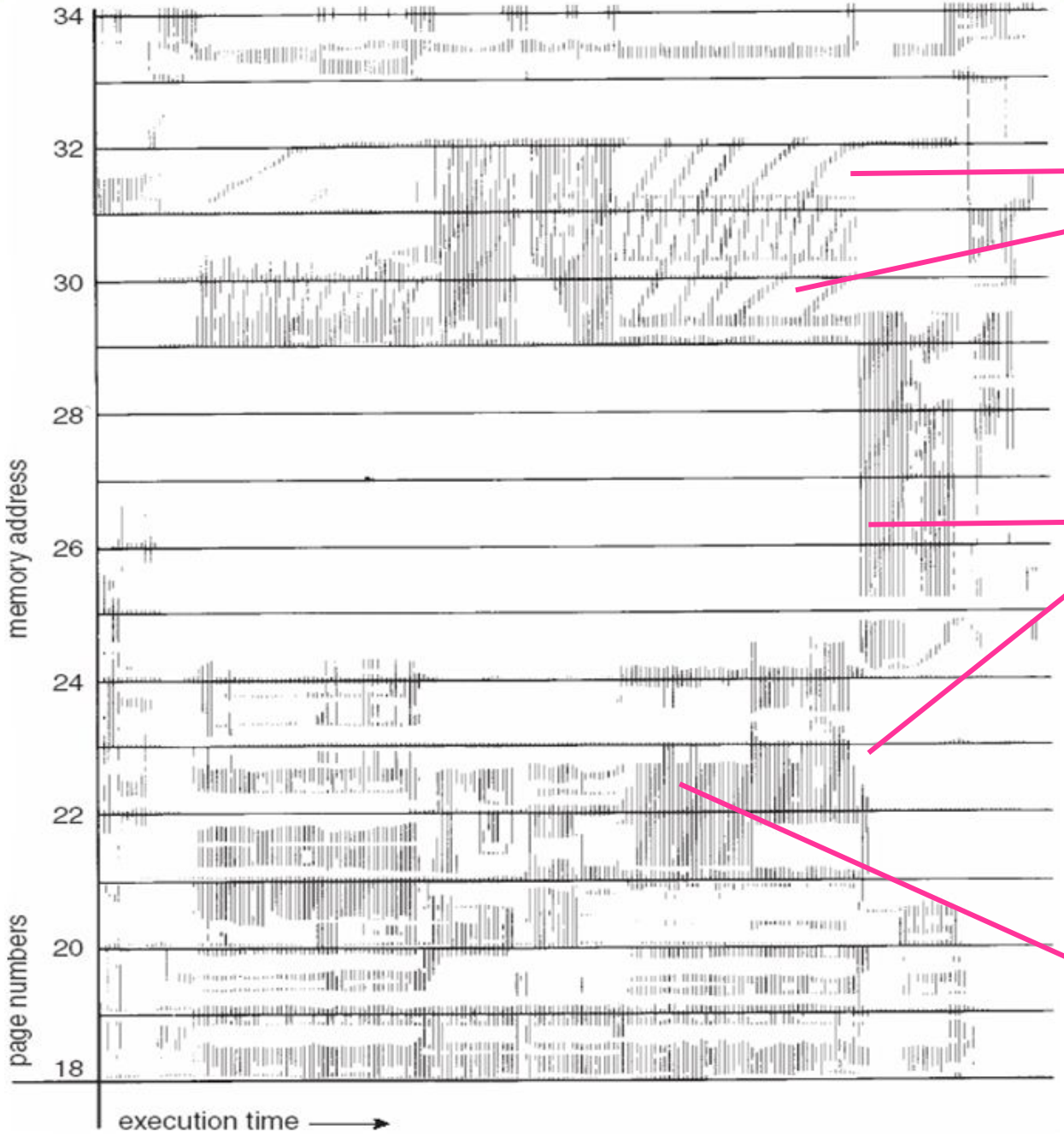
La memoria virtuale *su richiesta* (che cioè carica una pagina o un segmento quando serve) è sufficiente a caricare solo le pagine utilizzate

Perché ha senso la memoria virtuale

Un motivo più interessante è il fenomeno della località dei riferimenti alla memoria: per periodi significativamente lunghi, *tipicamente* un processo:

- accede ripetutamente ad indirizzi in un sottoinsieme delle sue pagine (*working set*);
- in particolare:
  - accede ripetutamente agli stessi indirizzi (località nel tempo): variabili usate ripetutamente, parti di codice usate ripetutamente
  - o accede a indirizzi vicini (e quindi probabilmente nella stessa pagina) a quelli usati poco prima (località nello spazio): codice sequenziale, “spazzolare” array; per i loop si eseguono più volte istruzioni fra loro vicine
- *non accede* a quelli di altre pagine, che quindi possono non stare in memoria

La località



località nello spazio

transizione

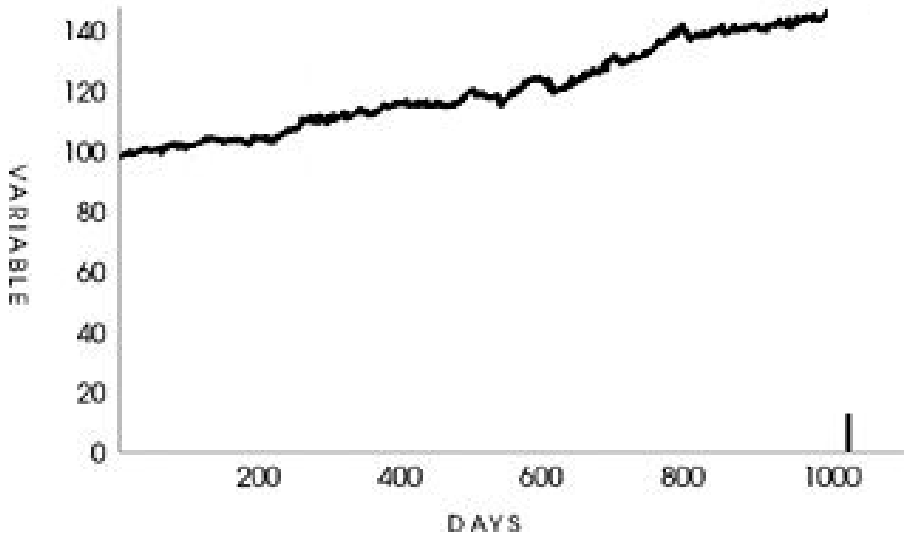
località nel tempo

La località

Per predire quali pagine servono ci baseremo sul passato, supponendo di non essere nei (pochi) momenti di transizione

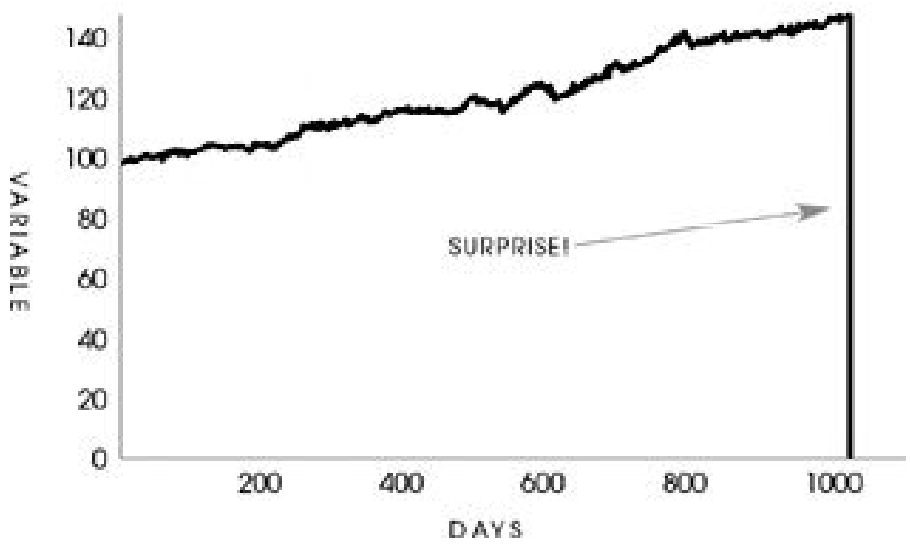
Ne *Il Cigno Nero* (sugli eventi inaspettati, come la scoperta di cigni neri) l'autore Nassim Nicholas Taleb (un consulente finanziario che studia seriamente i modelli probabilistici) presenta la storia di un tacchino, nutrito dagli umani…

supponiamo di misurare il suo benessere o fiducia negli esseri umani:



La località

Ma poi arriva il “giorno del ringraziamento”, quando negli Stati Uniti…



Per fortuna nel rimpiazzamento delle pagine non siamo come quel tacchino: se i “momenti di transizione” della località sono davvero pochi, a quelli sopravviveremo, al costo di rimpiazzare un po’ di pagine

La memoria virtuale

La motivazione per cui è stata inventata la memoria virtuale è proprio la necessità di eseguire programmi che richiedono più spazio della RAM disponibile. In sistemi "primitivi" questo si realizzava permettendo ai programmatori di spezzare codice e dati di un programma in "sezioni" chiamate overlay. Il sistema operativo forniva la possibilità di rimpiazzare un *overlay* con un altro durante l'esecuzione. Lo svantaggio di questo approccio è che la gestione degli *overlay* è a carico del programmatore

La tecnica di paginazione su richiesta (*demand paging,* caricare una pagina assente se si cerca di accedervi) svolge in automatico, in modo completamente trasparente per il programmatore, la stessa funzione.

In un sistema multiprogrammato tali tecniche permettono anche di mantenere in memoria un insieme di processi che complessivamente occuperebbero più spazio della memoria RAM disponibile.

La memoria virtuale paginata

- L'immagine in memoria di ciascun processo viene suddivisa in blocchi di uguale dimensione chiamati pagine (logiche)
- La memoria fisica viene suddivisa in *frame* (o pagine fisiche) della stessa dimensione delle pagine logiche
- Per ogni processo viene mantenuta una tabella delle pagine, che indica per ciascuna pagina dello spazio di indirizzi del processo se si trova o meno in memoria RAM, e in quale frame. Occorre inoltre tenere traccia della posizione delle pagine logiche su disco (l'immagine completa del processo deve comunque essere memorizzata su disco).
- Nella paginazione senza memoria virtuale *tutte le pagine* devono essere caricate in RAM; con memoria virtuale, solo alcune

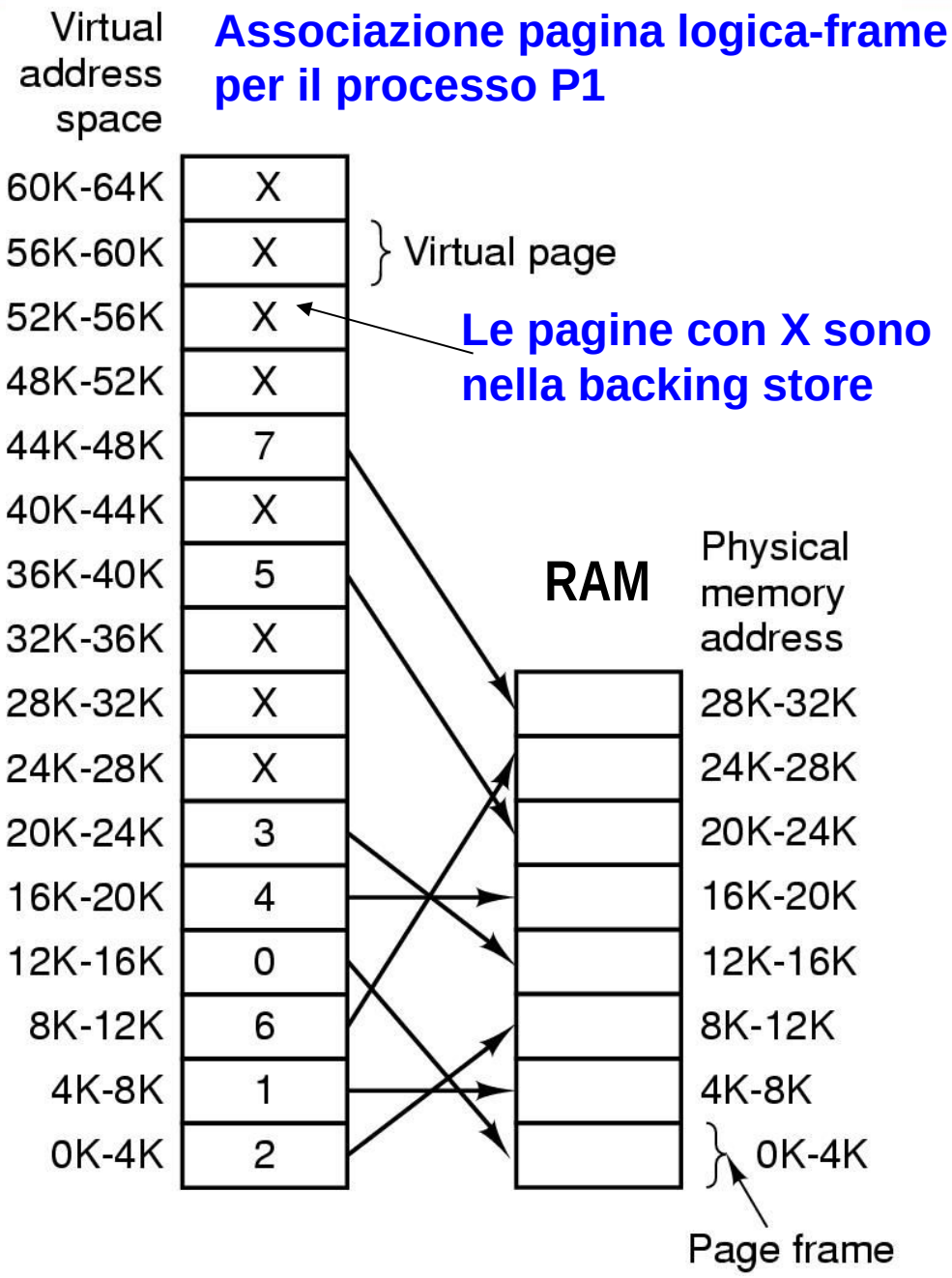
La memoria virtuale paginata

In questo esempio lo spazio di indirizzi del processo P1 comprende 16 pagine logiche da 4kbyte ciascuna (per un totale di 4K • 16 = 2^16 byte).

La RAM ha solo 8 frame (da 4kbyte ciascuno): la figura mostra una situazione in cui sono caricate in RAM le pagine logiche 0, 1, 2, 3, 4, 5, 9, e 11 del processo, rispettivamente nei frame 2, 1, 6, 0, 4, 3, 5 e 7.

Le pagine logiche possono essere caricate in frame non consecutivi, e in ordine sparso

Di tutto ciò teniamo traccia in una **tabella delle pagine**

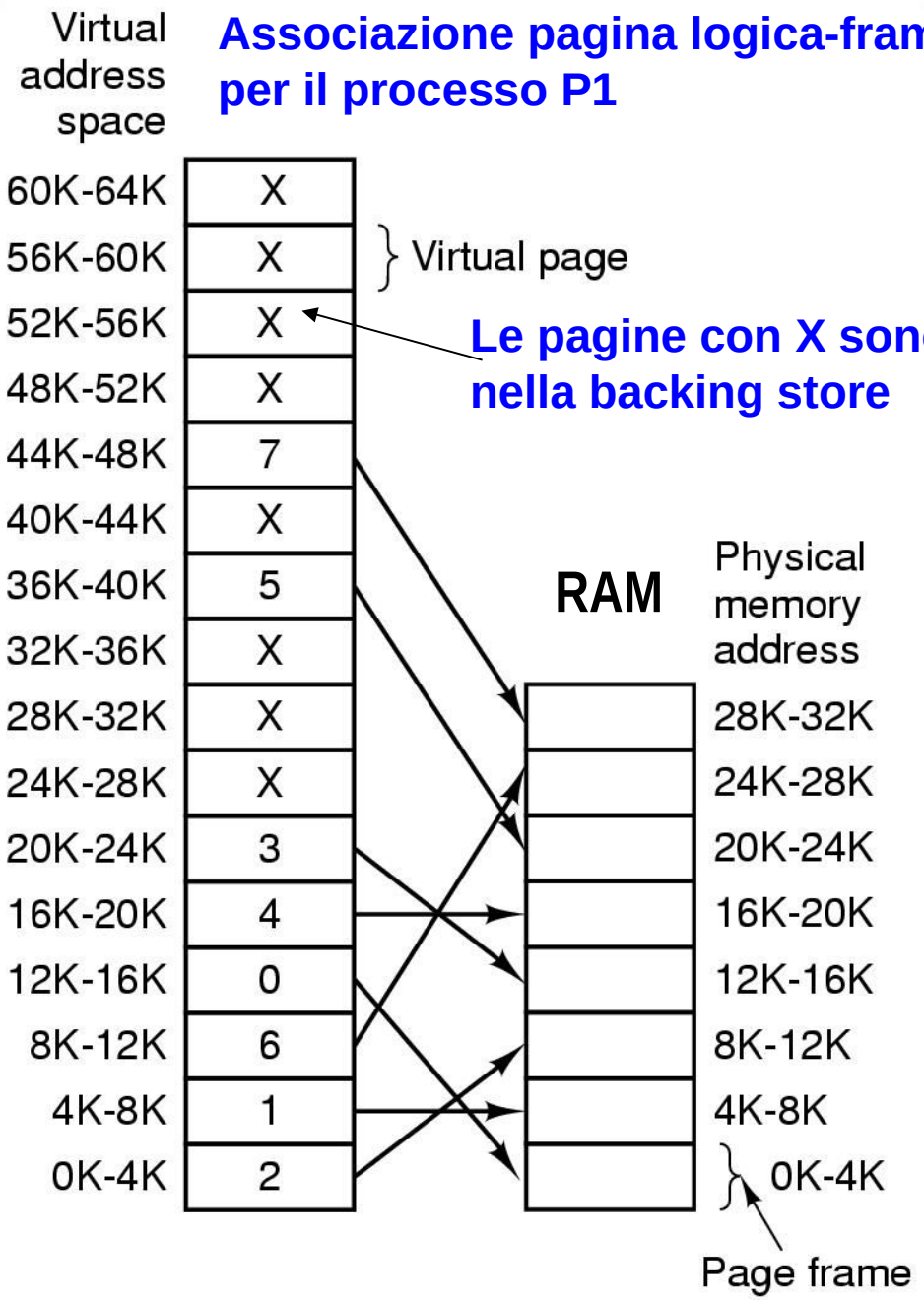


La tabella delle pagine

TABELLA DELLE PAGINE di P1

Indirizzi logici	Pagina logica	Present	Frame
0-4095	0	1	2
4096-8191	1	1	1
8192-12287	2	1	6
12288-16383	3	1	0
16384-20479	4	1	4
20480-24575	5	1	3
24576-28671	6	0	
28672-32767	7	0	
32768-40959	8	0	
36864-40959	9	1	5
40960-45055	10	0	
45056-49151	11	1	7
49152-53247	12	0	
53248-57343	13	0	
57344-61439	14	0	
61440-65535	15	0	

Associazione pagina logica-frame per il processo P1



Traduzione da indirizzo logico a indirizzo fisico

Quale operazione deve eseguire la MMU per tradurre gli indirizzi logici (IL) in indirizzi fisici (IF)?

In generale l'operazione da fare sarebbe:

1) calcolo numero di pagina logica (NP) e offset (O):

   NP = IL / DimPag; O = IL % DimPag

   quoziente e resto della divisione intera, perché:

   IL è DimPag * NP + O, con O < DimPag

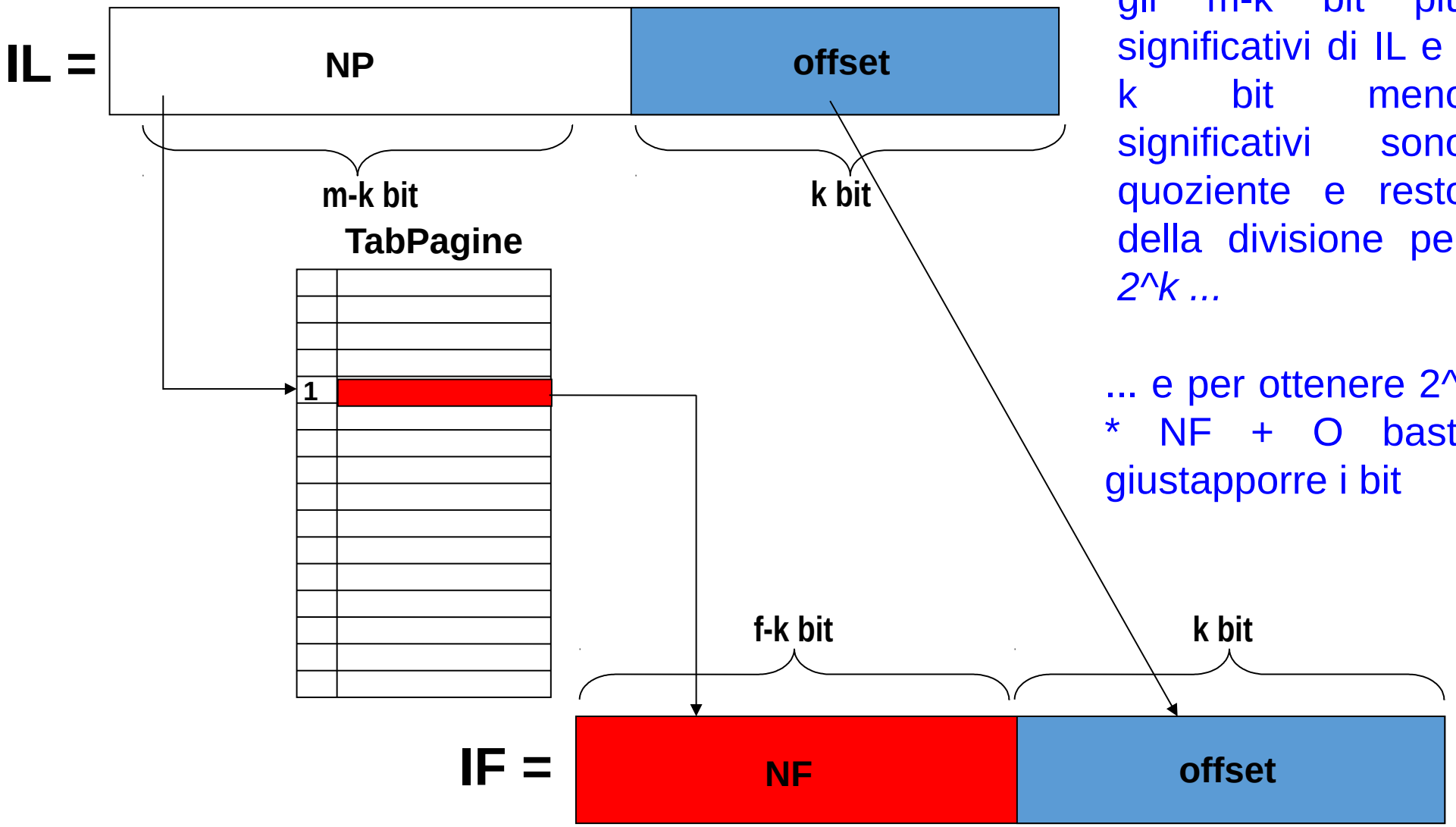
2) if (TabPag[NP].Present == 0) then trap(page fault)

   else NF = TabPag[NP].Frame

3) indirizzo fisico IF = indirizzo inizio frame NF + O = DimPag * NF + O

Traduzione degli indirizzi

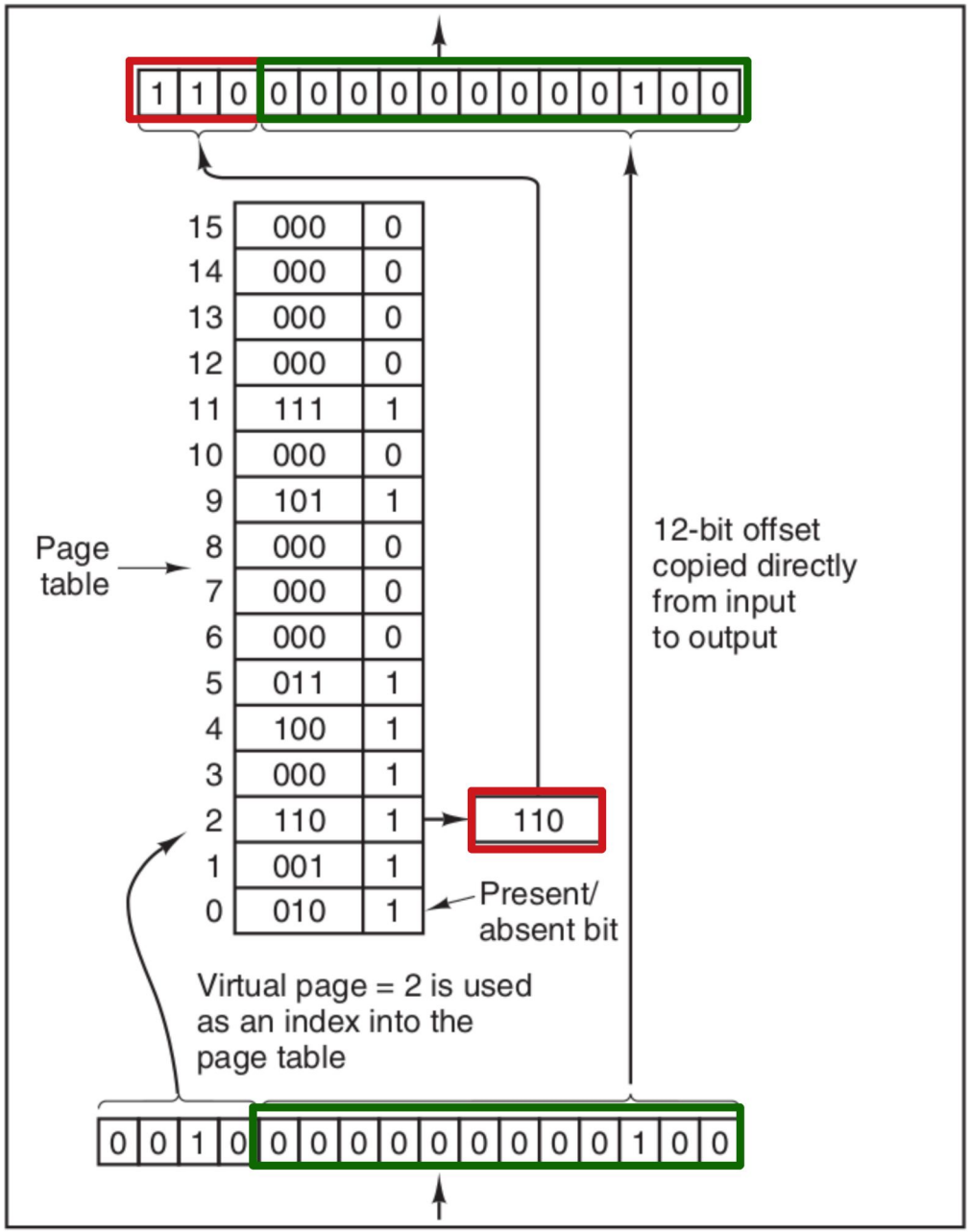
Ma non c'è bisogno di usare hw in grado di fare divisioni e moltiplicazioni se *DimPag = 2^k*; se supponiamo che il numero di bit dell'IL sia *m* (spazio di indirizzamento virtuale *0 ÷ 2^m*) e il numero di bit nell'indirizzo fisico sia *f* (dimensione della RAM = *2^f*):



gli m-k bit più significativi di IL e i k bit meno significativi sono quoziente e resto della divisione per *2^k ...*

... e per ottenere 2^k * NF + O basta giustapporre i bit

Esempio di traduzione



8196 / 4096 = 2, con resto 4,

in binario è:

001000000000100 /

1000000000000 =

0010

con resto 100

In questo caso la pagina 0010 (cioè 2) è presente e si trova nel frame 110 (cioè 6), allora:

IndFis =

110 *

1000000000000 =

110000000000000 +

100 =

110000000000100

Trattamento dei page fault

Se solo un sottoinsieme delle pagine è presente in memoria, si può verificare un accesso ad una pagina non caricata: in questo caso la MMU genera una trap di tipo *page fault,* la cui gestione è:

1) si cerca un frame libero in RAM: se non c'è, si sceglie (con un *algoritmo di rimpiazzamento*) una pagina "vittima"
2) il frame selezionato viene etichettato *busy* in modo che non venga scelto di nuovo come vittima
3) se necessario (pagina modificata rispetto al disco), si salva la "vittima" su disco
4) si carica la pagina che ha causato il page fault da disco a RAM (nel frame libero); mentre la pagina viene (salvata e) caricata, la CPU può essere data ad un altro processo;
5) quando l'interruzione da disco dice che la pagina è caricata, si aggiorna la tabella delle pagine (Present = 1, Frame = il frame in cui si è caricata la pagina) e si rimette il processo pronto; quando girerà eseguirà di nuovo l'istruzione che ha causato il *page fault*

Trasferimento delle pagine da disco a RAM (e viceversa)

Quando viene creato il processo, viene allocato lo spazio per la sua immagine sulla backing store (disco), poi le pagine vengono caricate in RAM secondo necessità.

Le operazioni di *swap-in* (caricamento) delle pagine avvengono (1) su richiesta (*demand paging*) durante l'esecuzione del processo, cioè in seguito ad un *page fault*, ed eventualmente (2) durante una fase di *pre-paging* attivata dal sistema operativo prima di rendere *running* un processo (occorre avere una previsione di quale sia il *working set* del processo in quel momento della sua esecuzione).

Le operazioni di *swap-out* (copia da RAM a disco) delle pagine avvengono quando si deve liberare spazio per nuove pagine (rimpiazzamento delle pagine). Conviene che l'hardware fornisca traccia in un bit (*dirty bit*), per ogni pagina, di quali pagine sono state effettivamente modificate (utilizzate in scrittura) dall'ultimo caricamento, e al momento del rimpiazzamento copiare su disco la pagina rimpiazzata solo se il suo *dirty bit* vale 1.

La scelta della pagina da rimpiazzare viene fatta da un *algoritmo di rimpiazzamento*