04 - MEMORIE CACHE, PAGING, MEMORIA VIRTUALE

MEMORIA CACHE

Le principali caratteristiche di una memoria sono:

- Dimensione
- Tempo di accesso
- Costo per bit

Chiaramente, esistono dei compromessi:

- Minor tempo di accesso --> maggior costo per bit
- Maggior capacità = minor costo per bit --> maggior tempo di accesso

Le memorie **RAM** garantiscono dimensioni relativamente grandi e basso costo per bit, ma sono *lente* in confronto alla velocità di elaborazione dati della CPU.
Questo fa si che molte computazioni siano rallentate dall'accesso in memoria perchè la CPU deve attendere i dati in arrivo.

NOTA: è improbabile che il gap venga colmato tecnologicamente.

GERARCHIA DI MEMORIA

Esistono memorie **più veloci** della RAM, ma sono *costose* e di *estensione ridotta*.
Non è possibile realizzare una memoria che sia grande, veloce ed economica, tuttavia è possibile sviluppare una *gerarchia di memoria* che soddisfa tutti i requisiti.

La memoria **CACHE** è una piccola memoria che viene posizionata fra la RAM e la CPU:

- E' molto *veloce* rispetto alla RAM.
- Ha *estensione limitata* ed è *costosa*.

La cache contiene una *copia di alcuni dati* presenti in memoria.
Per ogni richiesta di accesso a una word, la CPU controlla prima se essa si trova nella cache, in qual caso non sarà necessario accedere in memoria, *risparmiando tempo*.
Se la word è presente si ha **CACHE HIT** e la word viene consegnata in pochi cicli di clock, altrimenti (**CACHE MISS**) un blocco di word consecutive viene caricato in cache.

LOCALITA' DI RIFERIMENTO

La gerarchia di memoria funziona quando il processore usa *frequentemente* gli *stessi dati* (*Località* di **riferimento temporale**) oppure quando richiede *dati vicini* in memoria (es. elementi di un vettore) in un breve intervallo di tempo (*Località* di **riferimento spaziale**).

COSTO MEDIO DI ACCESSO IN MEMORIA

- T_1 : tempo per accedere ad un dato in cache.
- T_2 : tempo per accedere ad un dato in memoria.
- T_1 << T_2
- p : percentuale degli accessi a dati in cache (*hit rate*).

$$\bar{T} = T_1 p + T_2(1-p)$$

Ne segue che conviene *organizzare* la cache in modo da *massimizzare* l'hit rate.

ORGANIZZAZIONE CACHE

MEMORIA:

- N word : 2^n byte con indirizzo di n bit.
- Organizziamo le word in *blocchi* di K word.

CACHE:

- Contiene M=2^r *linee*.
- Ogni linea contiene *un blocco* di K word (2^w byte) e un *TAG* che identifica il contenuto.

L'*indirizzo di un blocco* che contiene una certa word si ottiene dagli *n-w* bit più significativi dell'indirizzo della word.

Come vengono *mappati* i blocchi di memoria sulle linee di cache?
Esistono 3 approcci:

- Cache a *mappatura diretta*.
- Cache *completamente associativa*.
- Cache *associativa a k-vie*.

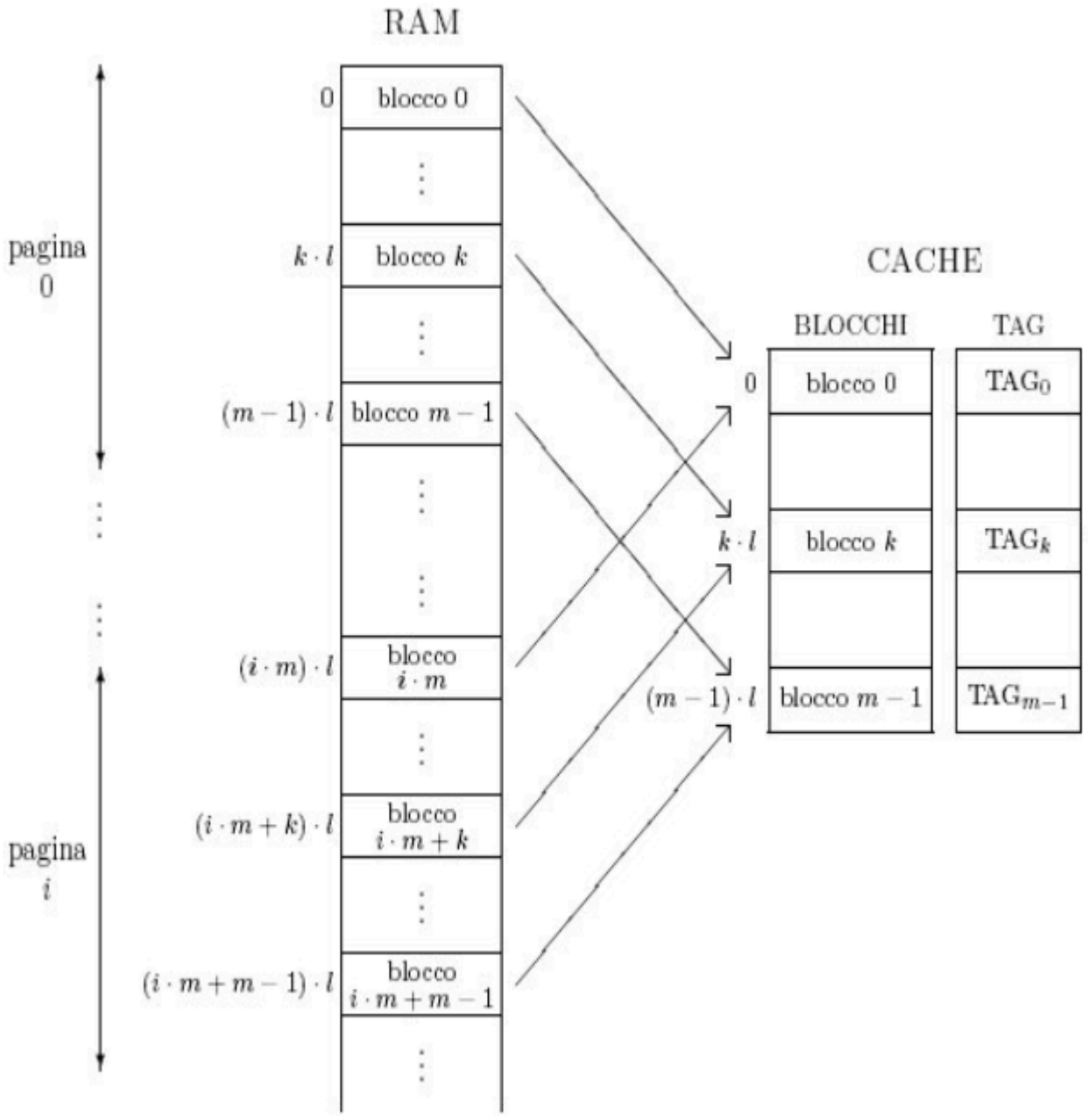
MAPPATURA DIRETTA

Il blocco in memoria con indice j viene mappato sulla linea i con

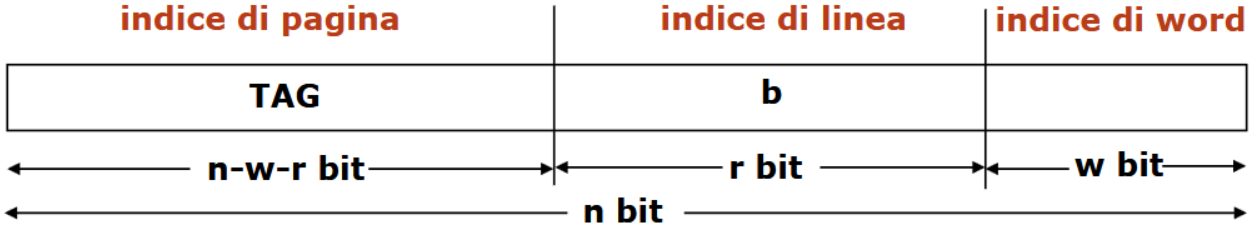
$$i = j \bmod m$$

Dove m è il numero di righe della cache.
E' come se la RAM fosse divisa in *pagine* e ciascuna pagina in *blocchi*:

- Le **pagine** hanno la stessa dimensione della cache.
- I **blocchi** hanno la stessa dimensione di quelli della cache.
- L'*indice di un blocco* all'interno della propria pagina è l'indice della riga della cache in cui sarà caricato.



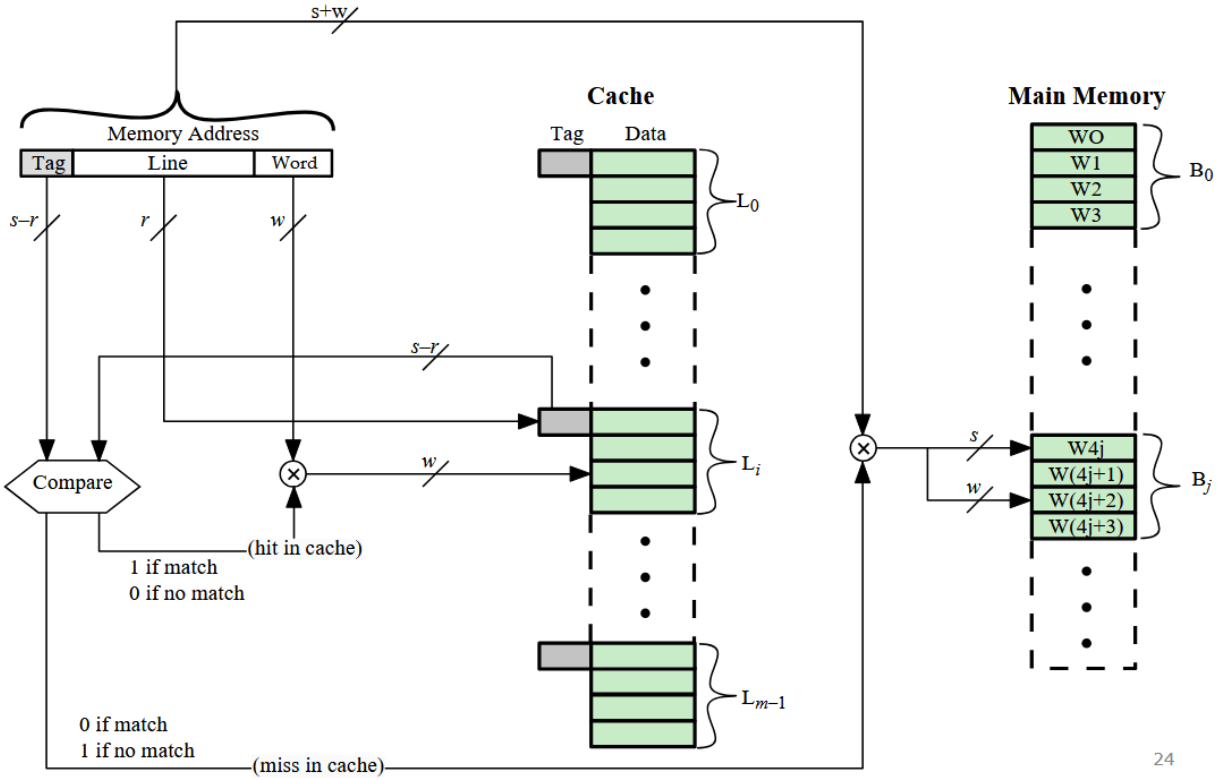
Gli indirizzi di memoria hanno pertanto il seguente formato:



Il TAG è costituito dagli *n-w-r* bit più significativi (notare che corrisponde quindi all'indice di pagina in RAM), e gli *r* bit centrali forniscono l'indice della *linea* (e quindi del blocco) nella *cache*.

Quando la CPU genera un indirizzo di memoria con indice di linea *b*, il suo TAG viene *confrontato* con il TAG associato alla *linea b* nella cache:

- Se sono **uguali**, il *blocco* presente nella cache è lo *stesso* in cui è contenuta la word richiesta.
- Altrimenti vuol dire che il blocco presente in cache appartiene a una *pagina diversa* (oppure la cache è vuota). In tal caso il blocco viene recuperato dalla RAM e inserito nella cache



PRO:

- Organizzazione *semplice*.

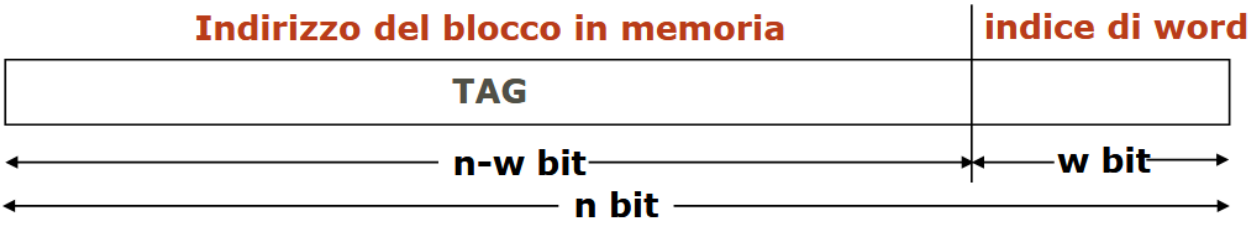
CONTRO:

- La *linea* da *sovrascrivere* è predeterminata: un blocco non può essere inserito in una linea tra quelle presumibilmente non più utilizzate.
- Hit rate *basso*.

COMPLETAMENTE ASSOCIATIVA

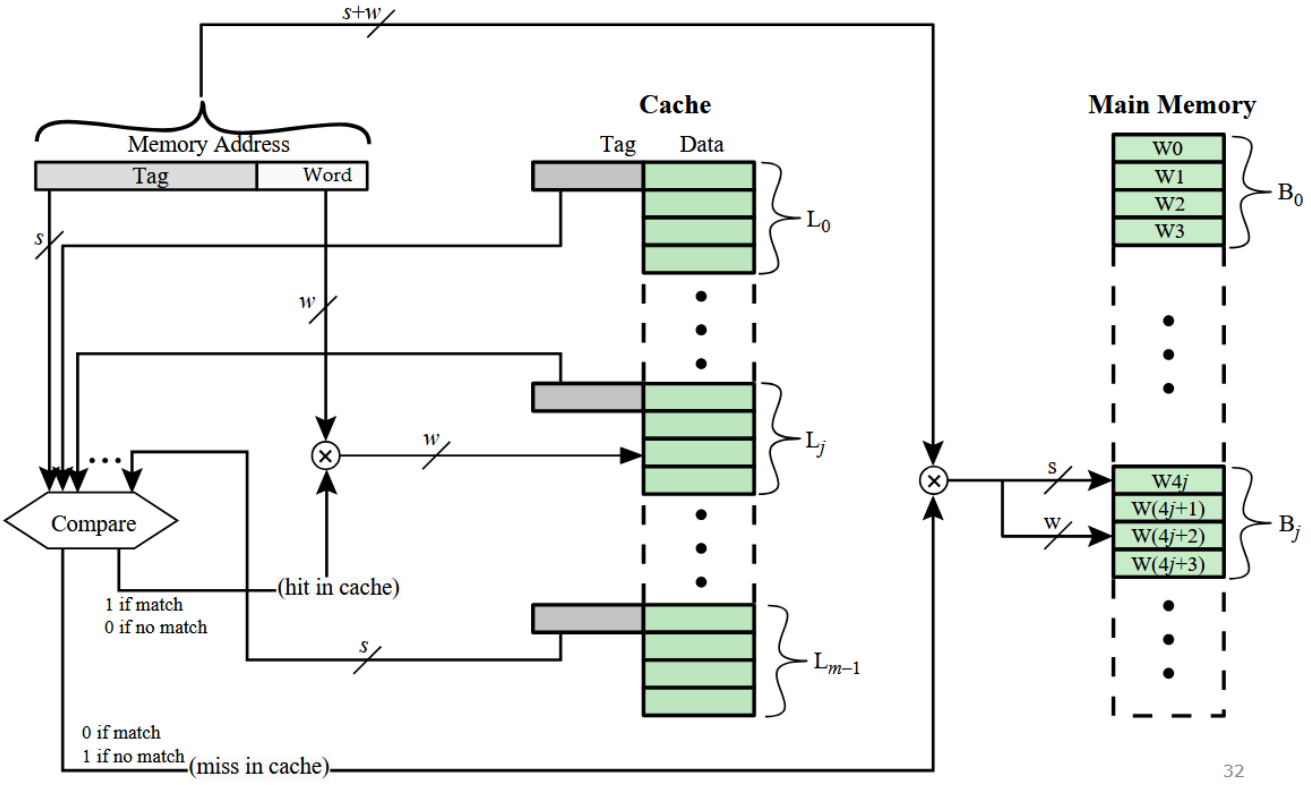
Un blocco viene inserito in una *linea vuota*. Se tutte le linee sono *occupate*, si sceglie quale linea svuotare con una **politica di rimpiazzo**.

Se n è il numero di bit di un indirizzo di memoria, gli indirizzi delle K word di ciascun blocco (2^w byte) hanno gli *n-w* bit più significativi uguali (**TAG**), mentre i *w* meno significativi forniscono l'indice della word all'interno del blocco.



Quando la CPU genera un indirizzo, gli *n-w* bit più significativi vengono confrontati con tutti i TAG delle m linee:

- Se il TAG viene *trovato* (HIT), l'accesso si risolve nella cache.
- Altrimenti (MISS), il blocco *non* è *presente* in cache: in tal caso viene *recuperato il blocco* dalla RAM e inserito in una *linea vuota* oppure si *libera una linea* per fare spazio.



La memoria associativa è di tipo **CAM** (*Content Addressable Memory*): è in grado di effettuare, in parallelo, il confronto tra un dato cercato e tutti i dati in essa contenuti. *Restituirà* quindi gli **indirizzi** dei dati uguali al dato cercato.

NOTA: il funzionamento è opposto alla RAM (che riceve indirizzo e restituisce dato), di conseguenza l'*hardware* che realizza una *CAM* è molto *complesso e costoso*.

PRO:

- **Hit rate** *alto* rispetto alla mappatura diretta (miglior sfruttamento dello spazio).

CONTRO:

- Organizzazione *costosa*, richiede CAM.

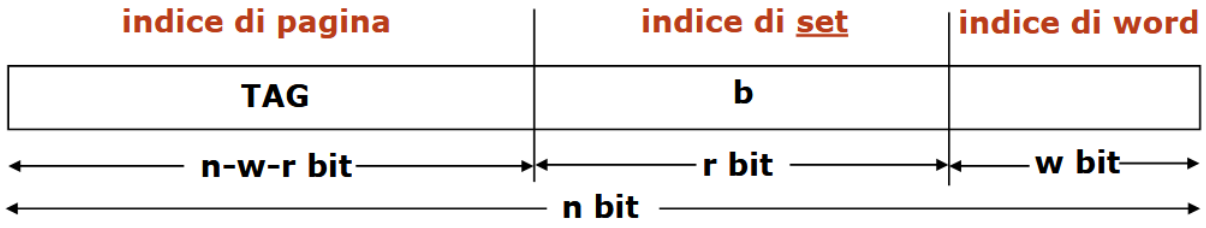
SET-ASSOCIATIVA A K VIE

Replicando *k volte* l'organizzazione della cache a *mappatura diretta* si ottiene un compromesso fra le soluzioni precedenti.

- Per ogni *indice di blocco* ci sono *k linee* disponibili.
- A ciascun **set** di *k linee* è associata una CAM.
- La RAM è ancora divisa in **pagine** (fatte da tanti blocchi quanti i set della cache) e ciascuna pagina in *blocchi* (il blocco b-esimo viene inserito nel set b-esimo, in una qualsiasi delle k linee).

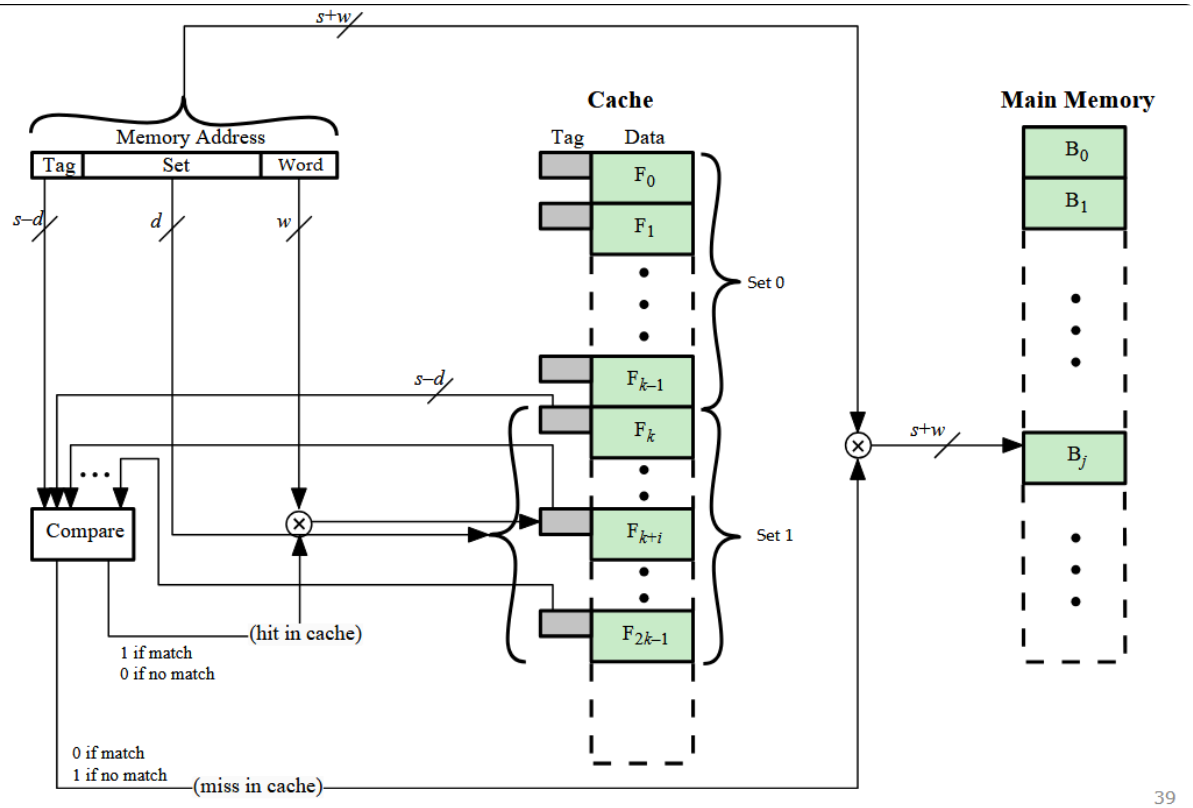
Gli indirizzi di memoria sono pertanto divisi in campi:

- I *w bit* meno significativi: indice della *word nel blocco*.
- Successivi *r bit*: indice del *blocco nella pagina* (quindi l'indice *di set*).
- *n-w-r bit* più significativi: **TAG**, cioè indice della *pagina nella RAM*.



Quando il processore genera un indirizzo di memoria, gli *r bit* che identificano l'indice del blocco individuano *quale set* della cache contiene il blocco (se presente).
Gli *n-w-r* bit più significativi (TAG) vengono confrontati in parallelo con il contenuto della CAM:

- Se trovato (HIT), viene restituita la word di *indice w* nel blocco.
- Altrimenti (MISS), il blocco di RAM viene caricato in una delle k linee del *set corrispondente*.



CONFRONTO CON LE ALTRE ORGANIZZAZIONI:

- VS *Mappatura Diretta*:
  Ha un *hit rate maggiore* perchè il blocco da rimpiazzare presenta maggior margine di scelta, ma è *più costosa* perchè richiede la CAM.
- VS *Completamente Associativa*:
  Ha un *hit rate inferiore* perchè la scelta del blocco da rimpiazzare non è fatta fra tutti quelli presenti in cache, ma solo fra quelli dello stesso set. Però è *meno costosa* perchè tante piccole CAM costano meno di una CAM unica e più estesa.

Dato che si tratta di un buon compromesso, è la *soluzione più diffusa*.

ALGORITMI DI RIMPIAZZO

Quando in seguito a un *cache miss* è necessario ricopiare un nuovo blocco nella cache, normalmente è necessario *sovrascriverne uno già presente*.

La politica con cui si sceglie il blocco da sovrascrivere dipende dall'**algoritmo di rimpiazzo**. Due esempi comuni sono:

- **FIFO**: si sovrascrive la linea contenente il blocco *presente da più tempo*.
- **LRU**: (*Least Recently Used*) si sovrascrive la linea che da più tempo non subisce accessi. Si possono usare uno o più *bit di controllo* per tenere traccia del tempo trascorso dall'ultimo accesso.

SOVRASCRITTURA DI UN BLOCCO

Se il blocco selezionato per essere sovrascritto ha *subito accessi in scrittura* mentre si trovava *nella cache*, un bit di controllo (**dirty bit**) lo segnala: la sua *copia in RAM* è **obsoleta** e prima di sovrascriverlo, il contenuto del blocco va ricopiato in RAM (*copy back*).

Una tecnica alternativa consiste nell'effettuare tutte le operazioni di scrittura nella cache *anche in RAM* (**write-through**).
In ogni caso, la scrittura in RAM avviene tramite un *write buffer* ad accesso veloce in modo che la CPU non debba attendere il trasferimento del dato.

CACHE MULTIPLE E MULTILIVELLO

Spesso i processori comprendono *due cache*: una per le *istruzioni* e una per i *dati*.

GERARCHIA MULTILIVELLO

Possono essere presenti *più livelli di cache*:

- Una cache di **1° LIVELLO** integrata nel chip del processore e ad accesso rapidissimo.
- Una cache di **2° LIVELLO**, integrata o meno, ad accesso rapido.
- Eventualmente anche una cache di **3° LIVELLO**.

I livelli *più vicini* conterranno i dati necessari nell'immediato futuro, mentre i dati che serviranno più avanti si troveranno nei livelli *più lontani*, che sono più lenti ma *più capienti*.

GESTIONE DELLA MEMORIA

CPU e memoria sono risorse che possono essere *condivise* da *più programmi*. Si parla quindi di sistema **multi-programming** o **multi-tasking**.
In un tale sistema, un processo può trovarsi in uno di 3 *stati*:

- **Ready**: lista delle istruzioni *pronta* per l'esecuzione.
- **Running**: in esecuzione.
- **Blocked**: lista delle istruzioni *non pronta* per l'esecuzione (in attesa).

In ogni istante, il SO può scegliere quale/i, tra i processi ready, passare allo stato running.

Per consentire la gestione multi-tasking dei processi, un SO utilizza meccanismi forniti dall'*hardware* che consentono di:

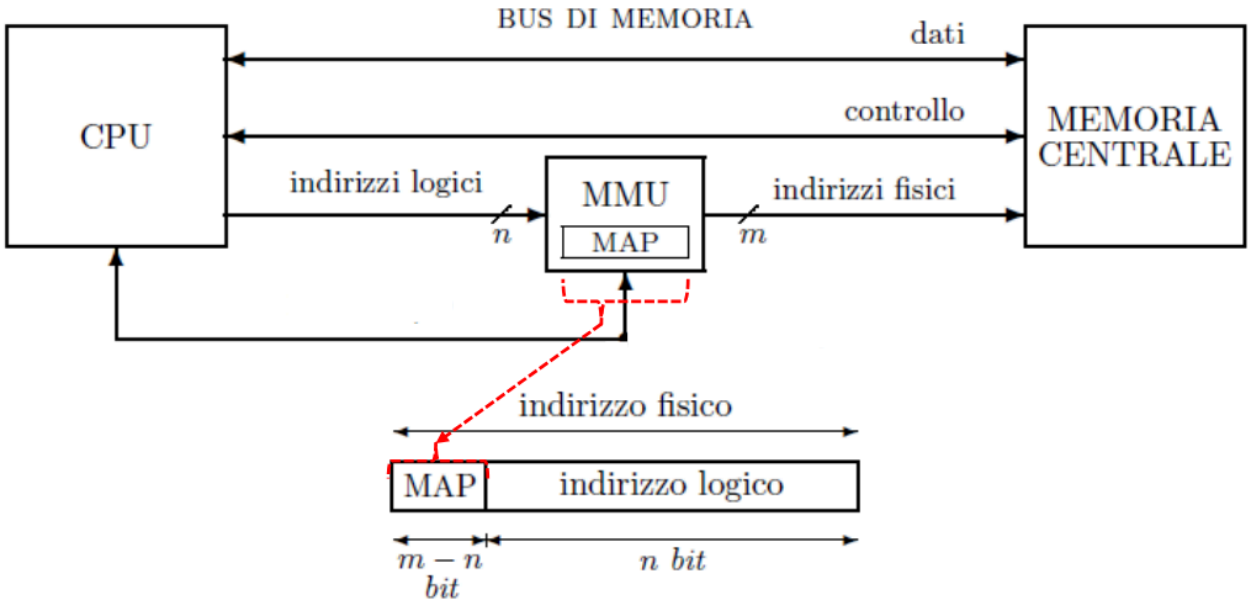
- *Suddividere la memoria* per consentire in essa la presenza di più processi (maggior probabilità che vi siano processi ready).
- Effettuare la suddivisione della memoria in *modo efficiente*.

PAGINAZIONE

La *memoria fisica* viene divisa in blocchi detti **pagine fisiche** di dimensioni uguali.
Lo spazio di memoria indirizzato dai *processi* è diviso in blocchi delle medesime dimensioni, detti **pagine logiche**.

La **paginazione** consente di *mappare* le pagine *logiche in quelle fisiche*.
Tale meccanismo è realizzato, a livello hardware, dall'**MMU** (*Mapping and Management Unit*).

I primi **MMU** erano utilizzati per consentire a CPU che lavoravano con indirizzi piccoli (es *16 bit*) di accedere a memorie fisiche più estese (di dimensione maggiore di 2^{16} bit).



Il modulo MMU contiene un insieme di *registri di mappa* (**PAGE TABLE**) utilizzati per la conversione fra indirizzi *logici* e indirizzi *fisici*.

- **INDIRIZZO LOGICO:** (*n* bit) i *q* più significativi forniscono l'indice della *pagina logica* (**IPL**) a cui l'indirizzo appartiene. I rimanenti *n-q* contengono l'*offset* dell'indirizzo all'interno della pagina logica.
- **INDIRIZZO FISICO**: (*m≠n* bit) gli *l* più significativi forniscono l'indice della *pagina fisica* (**IPF** o **frame #**) che andrà a sostituire l'IPL. I rimanenti *m-l* (*=n-q*) contengono l'offset all'interno della pagina, che *rimane il medesimo* specificato nell'indirizzo logico.

Quindi l'indirizzo fisico si ottiene *sostituendo* l'**IPL** con l'**IPF**.
In particolare, l'**IPL** individua l'*i-esimo* registro mappa, in cui è contenuto l'**IPF** *associato*.

NOTA:

- Se $q<l$ (indirizzo logico più corto di quello fisico) la memoria fisica è più estesa rispetto al range indirizzabile dalla CPU (situazione comune in passato).
- Se $q>l$ (indirizzo logico più lungo di quello fisico) la memoria logica è più estesa di quella fisica (situazione attuale: architetture a 64 bit permettono di indirizzare 2^{64} locazioni = 2^{34} G).

OSSERVAZIONI

- Immaginiamo che un processo utilizzi k pagine logiche, a cui sono associate altrettante pagine fisiche. Quando la sua esecuzione verrà sospesa e al suo posto entrerà in esecuzione un processo diverso, la *page table* sarà *aggiornata* con gli **IPF** del nuovo processo (situazione $q<l$), oppure esso accederà ad una *porzione diversa* della tabella che viene mappata nelle stesse pagine fisiche (situazione $q>l$).
  Per cui, a ciascun *processo* è *associato il set di* **IPF** a cui il processo accede.
- Nel caso in cui il numero di pagine *logiche* sia *elevato*, **NON** è praticabile l'uso di un MMU con una page table delle dimensioni necessarie a contenere così tanti registri mappa.
  In tal caso, la *page table è collocata in memoria* in un'**area protetta** e l'MMU contiene solo un puntatore alla page table associata al processo attivo.
  Così, tuttavia ogni *accesso in memoria* ne richiede in realtà *due*: uno all'area della page table e uno alla pagina fisica cercata (per questo si rimedia con l'implementazione di una cache dedicata alla page table).

CARATTERISTICHE PAGINAZIONE

- **Protezione**:
  associando dei *bit di controllo* è possibile realizzare meccanismi di controllo degli *accessi*.
- **Rilocamento**:
  una pagina logica può essere rilocata in una *diversa pagina fisica* per gestire meglio lo spazio, in base al numero di processi caricati in memoria.
- **Frammentazione dello spazio libero**:
  se lo spazio libero non è sufficiente per l'inserzione di un nuovo processo in locazioni *contigue*, è possibile rilocare le pagine (vedi sopra).
- **Comunicazione fra processi**:
  processi diversi possono accedere ad *aree di memoria condivise* se le tabelle a essi associate contengono **IPF** *comuni*.

SEGMENTAZIONE

I programmi sono costituiti da *moduli* scritti e compilati separatamente: il programmatore vede la memoria come costituita da *spazi di indirizzi separati* (non come uno spazio lineare unico). Tali moduli hanno tuttavia *lunghezze variabili* e un'organizzazione in pagine comporterebbe *sprechi di spazio* ogni qual volta un modulo dovesse occupare solo una porzione di una pagina (non è possibile allocare frazioni di pagine, ma solo un numero intero).

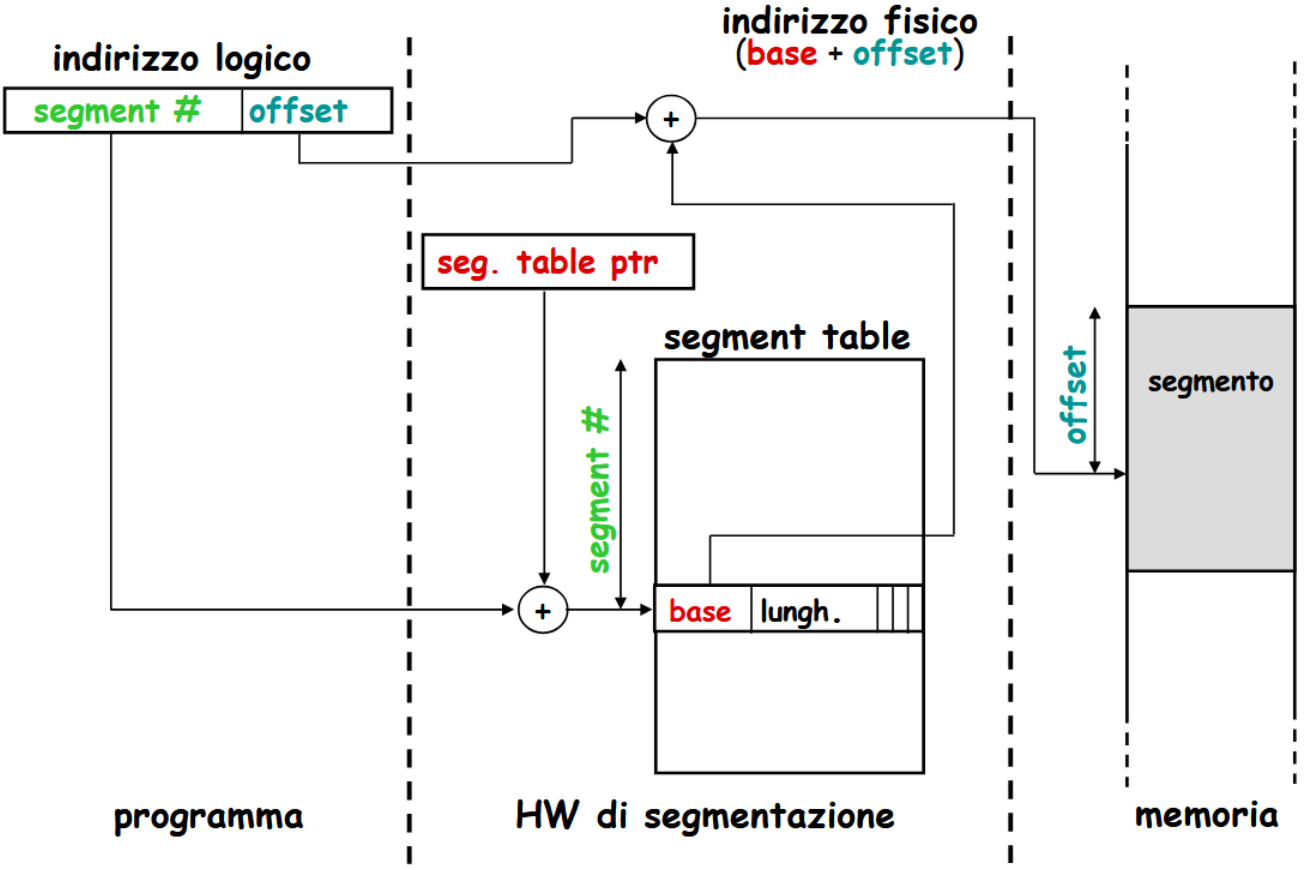
La tecnica appropriata per la gestione a moduli degli indirizzi è costituita dalla **SEGMENTAZIONE**.

A ciascun processo è associato un *insieme di segmenti* (uno per ciascun modulo) e una **segment table** contenente per ciascun segmento:

- **Indirizzo iniziale** del segmento.
- **Dimensione** del segmento.
- **Bit di controllo** (presente, modificato...).
- **Bit di protezione** (read only, permessi...).

Similmente al caso della paginazione, gli indirizzi *logici* sono costituiti da

- Un *indice di segmento* **IS** che individua l'elemento della segment table da cui estrarre l'*indirizzo fisico iniziale* **IIS**, ma anche la *dimensione* del segmento.
- Un *offset* **OS**, che viene confrontato con la dimensione **DS**:
  - Se **OS>DS** l'indirizzo è *non valido*.
  - Altrimenti è *valido*, e allora l'indirizzo *fisico* è dato da **IIS + OS**.



SOLUZIONE MISTA

Segmentazione e paginazione non sono mutualmente esclusive: è possibile implementare una soluzione mista, per esempio *dividendo i segmenti in piccole pagine*.

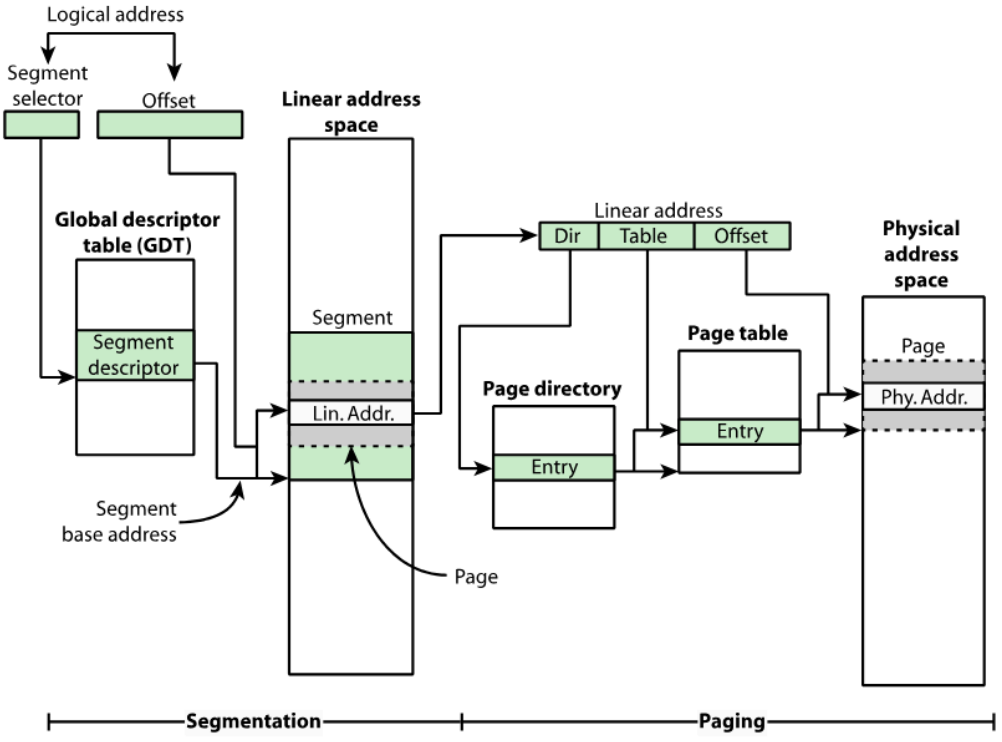


Figure 8.21 Intel x86 Memory Address Translation Mechanisms

NOTA: i meccanismi di paginazione e segmentazione sono forniti dall'*hardware*, e sono quindi intrinsecamente dipendenti dall'architettura del processore.

MEMORIA VIRTUALE

Paginazione e segmentazione permettono al SO di effettuare *swap in* e *swap out* dei programmi presenti in memoria oppure di *rilocarli*.

In particolare, *non* è *necessario* che tutti i blocchi di un processo siano *presenti contemporaneamente* in memoria durante l'esecuzione: la parte presente è detta **resident set**.

Quando un processo **P** richiede un indirizzo al di *fuori del resident set* (**memory fault**):

- P viene *interrotto* e posto nello stato *waiting*.
- Il SO manda in esecuzione un *altro processo* **Q** e avvia la lettura da disco del blocco richiesto da **P** (*swap in*).
- Quando la lettura da disco è completata, **Q** viene *interrotto* e **P** viene riportato allo stato *ready*.

Questo meccanismo è il sistema della **MEMORIA VIRTUALE** e permette:

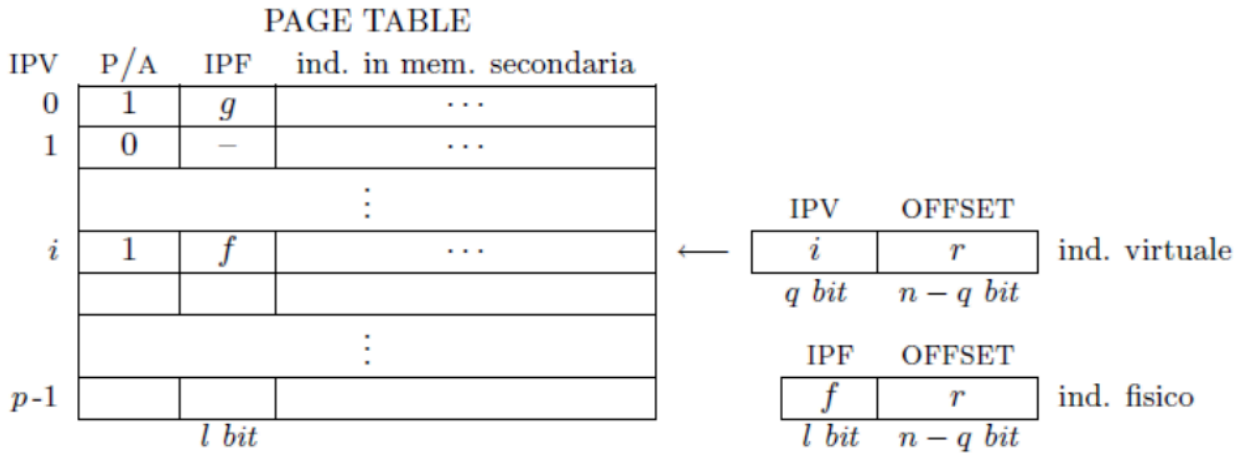
- Un *maggior numero* di *processi* presenti *in memoria* (perchè basta che sia presente il resident set), che garantisce *maggior efficienza* nell'uso della *CPU*.
- I processi possono *indirizzare più memoria* di quella *fisicamente disponibile*.

La **memoria virtuale** indica quindi lo spazio di indirizzi di memoria (anche maggiore di quello fisico) disponibile a un processo.
La *gestione del resident set* segue gli stessi *principi di località* secondo cui opera la CACHE e anche le politiche di rimpiazzo sono analoghe.

PAGE TABLE PER MEMORIA VIRTUALE

Per realizzare un sistema di memoria virtuale, si utilizzano delle *page table* simili a quelle viste sopra, ma con dei campi aggiuntivi:

- Bit di **presenza**: 1 se la pagina è *presente in memoria fisica*, 0 se è in memoria secondaria.
- **Indirizzo in memoria secondaria**, usato per reperire la pagina da disco nel caso in cui non sia fisicamente presente in memoria.



Gestione degli indirizzi virtuali.

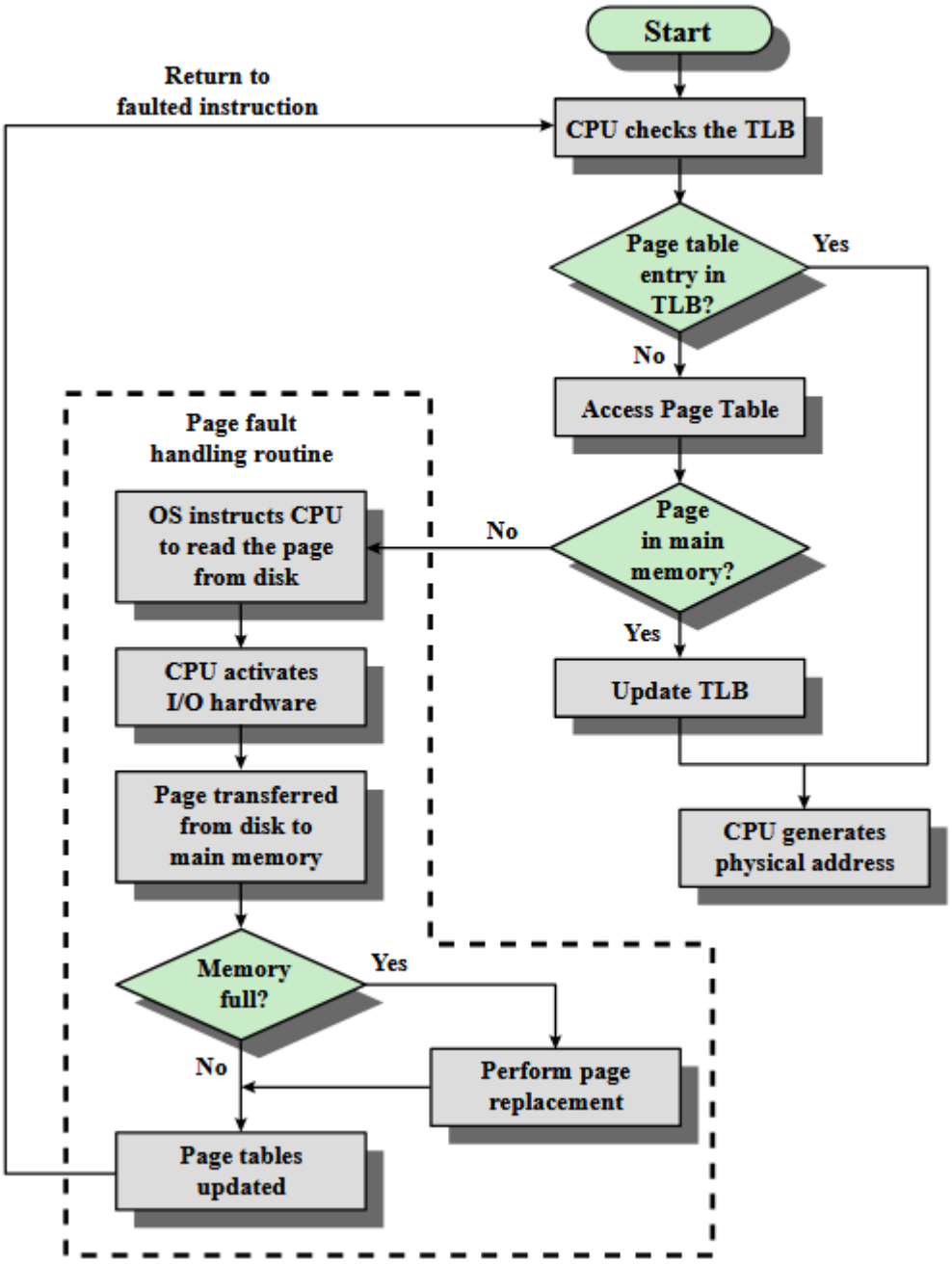
A *ogni processo* è associata una *page table specifica*. Chiaramente, le page table sono potenzialmente molto estese, per cui non possono essere memorizzate nei registri, ma devono essere *collocate in memoria*.

Questo introduce però un *problema*: ogni accesso in memoria ne richiede in realtà due (uno per la page table e uno per il dato effettivo). Vedi > OSSERVAZIONI.

Per *limitare* questo effetto, è possibile collocare una sua *porzione* (gli indirizzi delle pagine fisiche più utilizzate, sempre secondo i principi di località) in una memoria *cache* chiamata **Translation Lookaside Buffer (TLB)**.

Il TLB contiene gli IPV (organizzati in una memoria *CAM*) e gli IPF *associati*.
Dato un *indirizzo virtuale*, il suo IPV viene cercato nel TLB:

- Se trovato (*HIT*) il estra l'IPF associato, si costruisce l'*indirizzo fisico* e si accede in memoria.
- Se non trovato (*MISS*), si accede prima alla *page table*: nel caso sfortunato in cui la pagina non sia fisicamente in memoria, si ha *page fault* e la pagina viene caricata in memoria dal disco, la page table viene aggiornata e il suo IPV viene inserito nel TLB (principio di località).



SEGMENT TABLE PER MEMORIA VIRTUALE

Funziona in maniera del tutto *analoga* alla *paginazione con memoria virtuale*, con l'unica differenza che la **segment table** non contiene IPV e IPF associati, ma **IS** con relativi **IIS** e **DS** associati.

SEGMENTAZIONE E PAGINAZIONE

Anche in un sistema di memoria virtuale è possibile combinare paginazione e segmentazione, con i seguenti vantaggi:

- La **paginazione** è *trasparente* al programmatore: facilita la gestione della memoria senza che esso se ne debba occupare.
- La **segmentazione** è *visibile* al programmatore: consente la *modularità*, la *protezione* e la *condivisione* dei segmenti.