Memoria virtuale - hardware e strutture di controllo

Index

Concetti fondamentali

I riferimenti alla memoria sono degli indirizzi logici che sono tradotti in indirizzi fisici a tempo di esecuzione; questo perché un processo potrebbe essere spostato più volte della memoria principale alla secondaria e viceversa durante la sua esecuzione, ogni volta occupando zone di memoria diverse.

L’idea geniale

Ci si è accorti che non ci sta nessuna necessità che tutte le pagine o segmenti di un processo siano in memoria principale. Infatti per eseguire un processo ho la necessità che ci sia in memoria la pagina che contiene l’istruzione da eseguire e eventualmente i dati di cui l’istruzione ha bisogno e il resto può essere caricato in un momento successivo. Ciò fa passare dalla paginazione semplice alla paginazione con memoria virtuale.

Esecuzione di un processo

Il SO porta in memoria principale solo alcuni pezzi (pagine) del programma; la porzione del processo in RAM viene chiamato *resident set*.
Se il processo tanta di accedere ad un indirizzo che non si trova in memoria viene generato un interrupt di tipo *page fault* che risulta essere una richiesta I/O a tutti gli effetti, infatti il SO mette il processo in blocked
Quindi il pezzo di processo che contiene l’indirizzo logico viene portato in memoria principale (il SO effettua una richiesta di lettura su disco) e finché quest’operazione non viene completata, un altro processo va in esecuzione. Una volta completata, un interrupt farà si che il processo torni ready (non necessariamente in esecuzione)
Quando verrà eseguito, occorrerà eseguire nuovamente la stessa istruzione che aveva causato il fault

---

Conseguenze

Abbiamo due principali conseguenze a ciò:

- Svariati processi possono essere in memoria principale. Questo vuol dire che è molto probabile che ci sia sempre almeno un processo ready, aumentando così il grado di multiprogrammazione
- Un processo potrebbe richiedere più dell'intera memoria principale

---

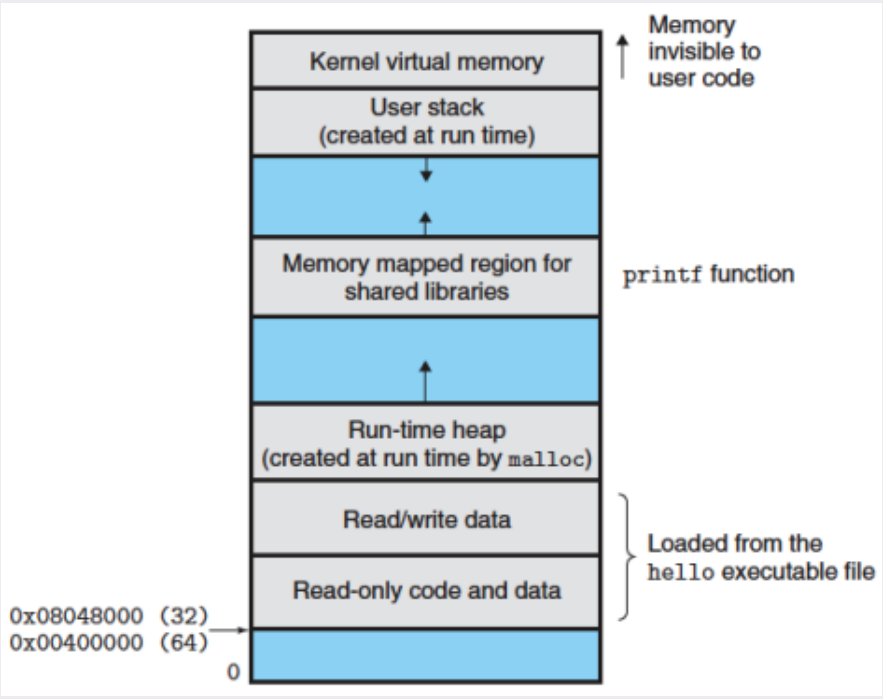
Memoria virtuale: terminologia

Memoria virtuale → schema di allocazione di memoria, in cui la memoria secondaria può essere usata come se fosse principale

- gli indirizzi usati nei programmi e quelli usati dal sistema sono diversi
- c'è una fase di traduzione automatica dai primi nei secondi
- la dimensione della memoria virtuale è limitata dallo schema di indirizzamento, oltre che ovviamente dalla dimensione della memoria secondaria
- la dimensione della memoria principale, invece, non influisce sulla dimensione della memoria virtuale

Memoria reale → quella principale (la RAM)
Indirizzo virtuale → l'indirizzo associato ad una locazione della memoria virtuale (fa sì che si possa accedere a tale locazione come se fosse parte della memoria principale)
Spazio degli indirizzi virtuali → la quantità di memoria virtuale assegnata ad un processo
Spazio degli indirizzi → la quantità di memoria assegnata ad un processo
Indirizzo reale → indirizzo di una locazione di memoria

> **Come un processo Linux vede la memoria**



Trashing

E’ ciò che succede quando il SO passa la maggior parte del tempo a swappare i processi piuttosto che ad eseguirli. Questo avviene quando un processo, la maggior parte delle richieste che fa, danno vita ad un *page fault*.
Per evitarlo, il SO cerca di indovinare quali pezzi di processo saranno usati con minore o maggiore probabilità sulla base della storia recente. Questo meccanismo sfrutta il **principio di località**

Principio di località

I riferimenti che fa un processo fa tendono ad essere vicini (sia che si tratti di dati che di istruzioni) quindi solo pochi pezzi di processo saranno necessari di volta in volta.
Si può dunque prevedere abbastanza bene quali pezzi di processo saranno necessari nel prossimo futuro

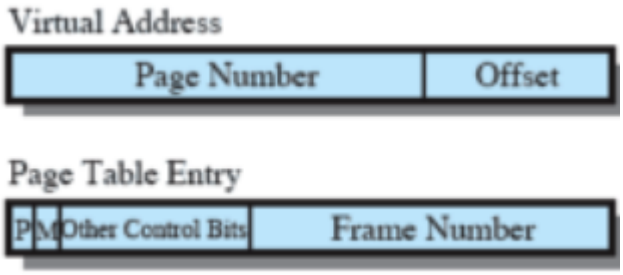
---

Supporto richiesto

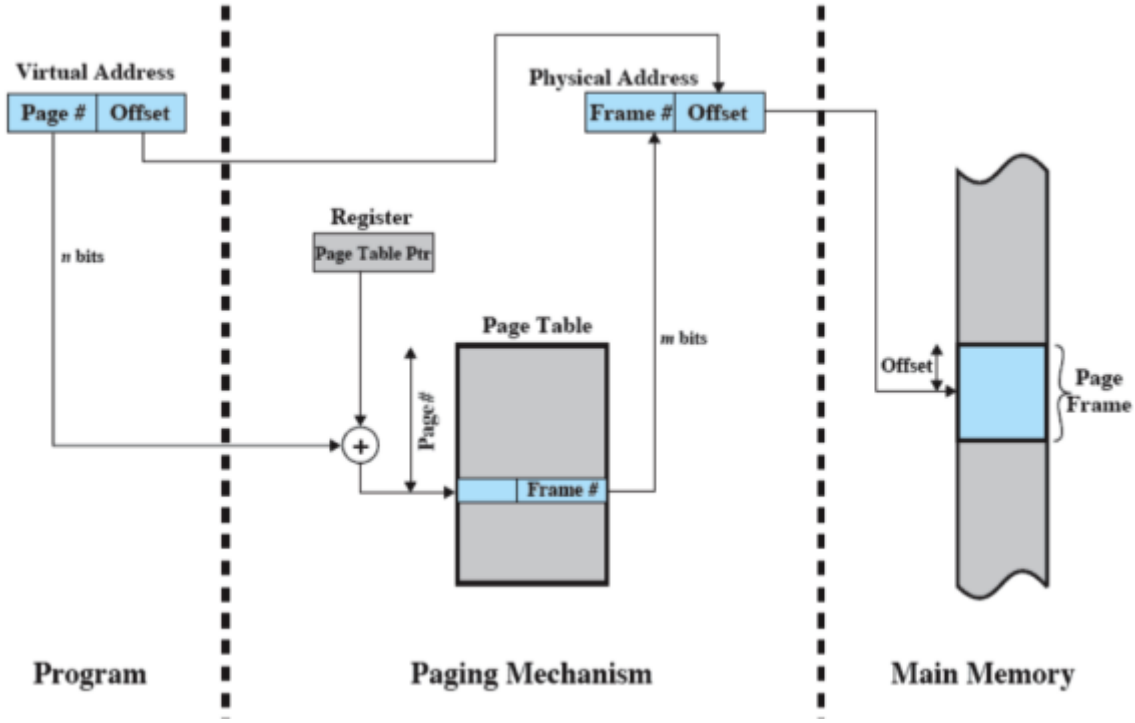
Anche qui sarebbe necessario un eccessivo overhead per poter fare tutto, risulta dunque utile del supporto hardware.

Paginazione

Ogni processo ha una sua tabella della pagine e il process control block di un processo punta a tale tabella
Le pagine contengono al loro interno, oltre che il numero di frame, anche dei bit di controllo; di questi, due sono particolarmente importanti: il **bit di presenza** (ci dice se un bit si trova in RAM oppure se è swappato) e **modified** (è zero se ci si è acceduto solo in lettura e diventa uno se è stato modificato)



Traduzione degli indirizzi



La traduzione dunque è fatta dall'hardware e la somma che si vede, non è una semplice somma infatti bisogna anche moltiplicare il numero di pagina per la dimensione della pagina oltre a sommare l'indirizzo base dell'inizio del page table

Affinché questo schema funzioni, non appena un processo viene caricato la prima volta oppure si ha un process switch, il sistema operativo deve:

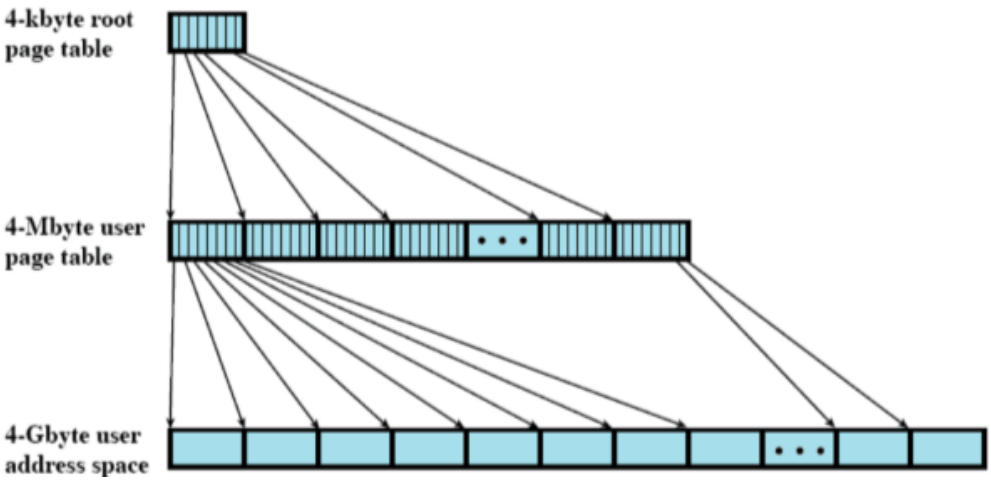
1. Caricare a partire da un certo indirizzo I la tabella delle pagine del processo (si trova all'interno del process control block)
2. Caricare il valore di I in un opportuno registro dipendente dall'hardware (`CR3` nei Pentium)

Overhead

Come abbiamo detto un problema da tenere sotto controllo è quello dell'overhead, infatti le pagine potrebbero contenere molti elementi. Quando un processo è in esecuzione, viene assicurato che almeno una parte della sua tabella delle pagine sia in memoria.
Facciamo qualche conto: supponiamo 8GB di spazio virtuale e 1kB per ogni pagina, il numero di entries che ci possono essere per ogni tabella delle pagine è $\frac{2^{33}}{2^{10}} = 2^{23}$ (ovvero per ogni processo)
Quanto occupa una entry? $1\ \text{byte}$ di controllo $\log_2(\text{size RAM in frames}) \rightarrow$ con massimo $4\ \text{GB}$ di RAM (architettura a 32-bits) fanno $4\ \text{bytes}$.
Max $32\ \text{bit} - 10\ \text{bit} = 22\ \text{bit}$ per i frame quindi $3\ \text{bytes}$ per il frame number, più il byte di controllo
Fanno $4 \cdot 2^{23} = 2^{23+2} = 32\text{MB}$ di overhead per ogni processo (con RAM di 1GB, bastano 20 processi per occupare più di metà RAM con sole strutture di overhead)

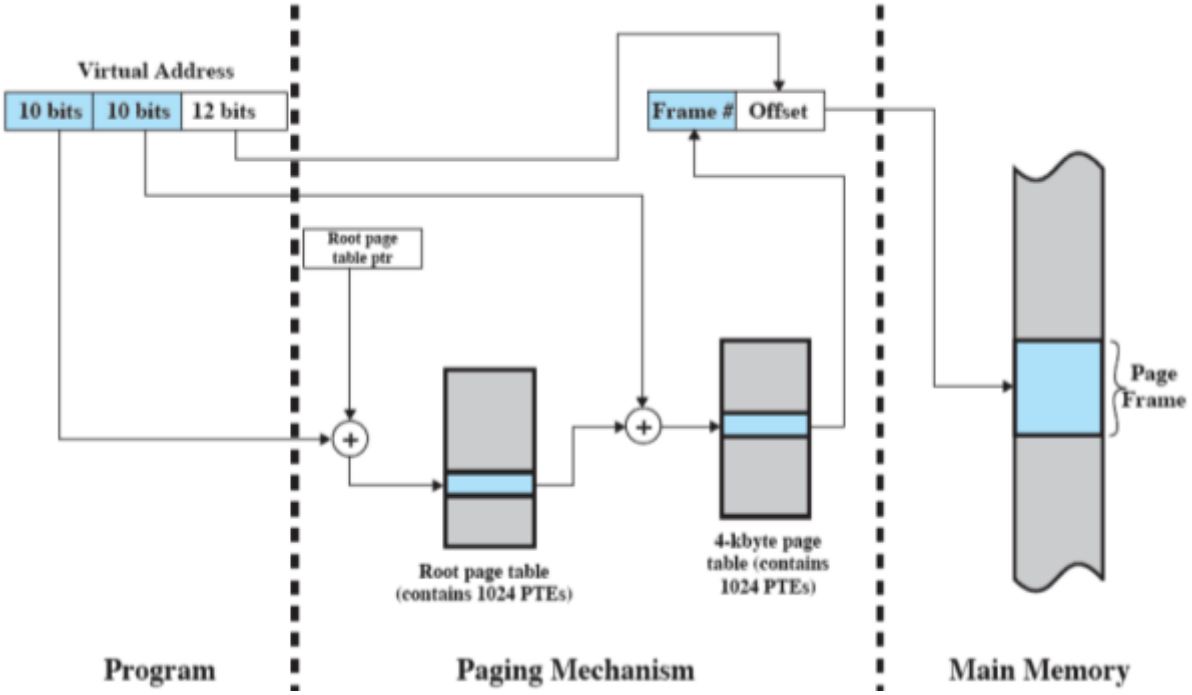
Tabella delle pagine a 2 livelli

Per risolvere il problema dell'**overhead** si è pensato di fare delle tabelle delle pagine a più livelli. Anche in questo caso il processore deve avere hardware dedicato per i 2 livelli di traduzione



Il primo livello punta al secondo livello che ha sua volta punta alla memoria principale

Traduzione



Perché conviene?

Supponiamo nuovamente che abbiamo 8GB di spazio virtuale, vuol dire che abbiamo 33 bits di indirizzo; facciamo ad es. 15 bit primo livello (*directory*), 8 bit di secondo livello, e i rimanenti 10 per l'offset.

> **Info**
>
> Spesso i processori (eg. Pentium) impongono che una tabella delle pagine di secondo livello entri in una pagina a sua volta, quello che occupa una pagina di secondo livello sta esattamente dentro una pagina di primo, infatti essa occupa $2^8 \cdot 2^2 = 2^{10}$ bytes

Per ogni processo l'overhead è $2^{23+2} = 32\text{MB}$, più l'occupazione del primo livello $2^{15+2} = 128\text{kB}$.
Seppur sia aumentato lo spazio, è più facile paginare la tabella delle pagine, così in RAM basta che ci sia il primo livello più una tabella del secondo così l'overhead scende a $2^{15+2} + 2^{8+2} = 128\text{kB}$.
Con RAM di 1GB, occorrono 1000 processi per occupare più di metà della RAM con sole strutture di overhead

---

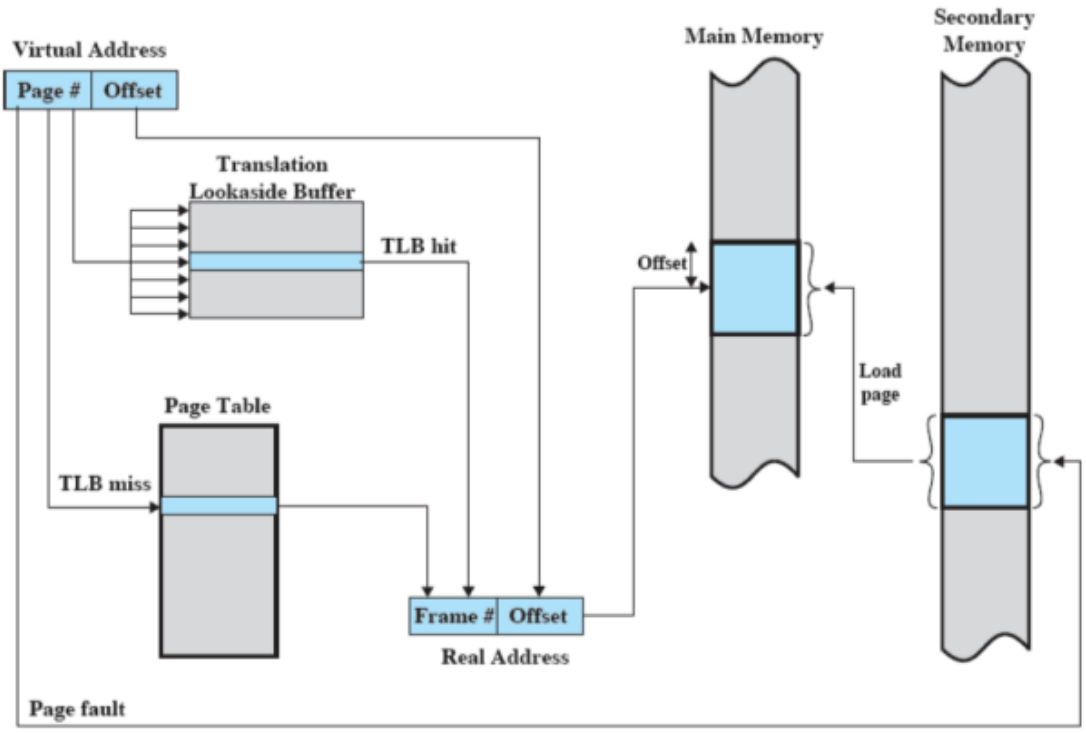
Translation Lookaside Buffer

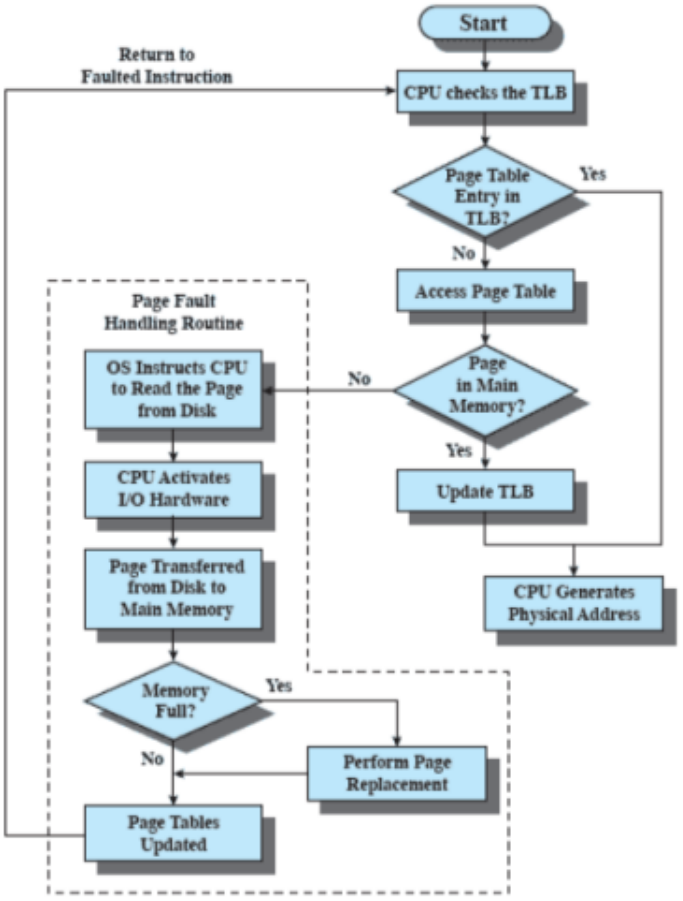
Translation Lookaside Buffer (TLB) letteralmente: memoria temporanea per la traduzione futura. Abbiamo visto che ogni volta che facciamo un riferimento alla memoria virtuale possono essere generati fino a due accessi alla memoria: uno per la tabella delle pagine e uno per prendere il dato

L'idea è quindi quella di usare una specie di cache per gli elementi delle tabelle delle pagine (contiene gli elementi che sono stati usati più di recente) ed è proprio il TLB

Come funziona?

Dato un indirizzo virtuale, il processore cerca la pagina all'interno del TLB.
Se la pagina è presente (*TLB hit*), si prende il frame number e si ricava l'indirizzo reale
Se la pagina non è presente (*TLB miss*), si prende la "normale" tabella delle pagine del processo
Se la pagina risulta in memoria principale a posto, altrimenti si gestisce il page fault.
Quindi il TLB viene aggiornato includendo la pagina appena acceduta (usando un qualche algoritmo di rimpiazzamento se il TLB è già pieno: solitamente LRU)





Memoria virtuale e process switch

Seppur il TLB sia totalmente hardware ci sono casi in cui è necessario un intervento del sistema operativo. In particolar modo il TLB deve essere resettato è infatti relativo ad un singolo processo, ma risulta essere la soluzione peggiore dal punto di vista prestazionale.

Per fare un po' meglio, alcuni processori permettono:

- di etichettare con il PID ciascuna entry del TLB (es. Pentium), non serve fare il reset ma basta fare un confronto tra il PID attuale e quello presente nella entry del TLB
- di invalidare solo alcune parti del TLB (alla fine inefficiente)

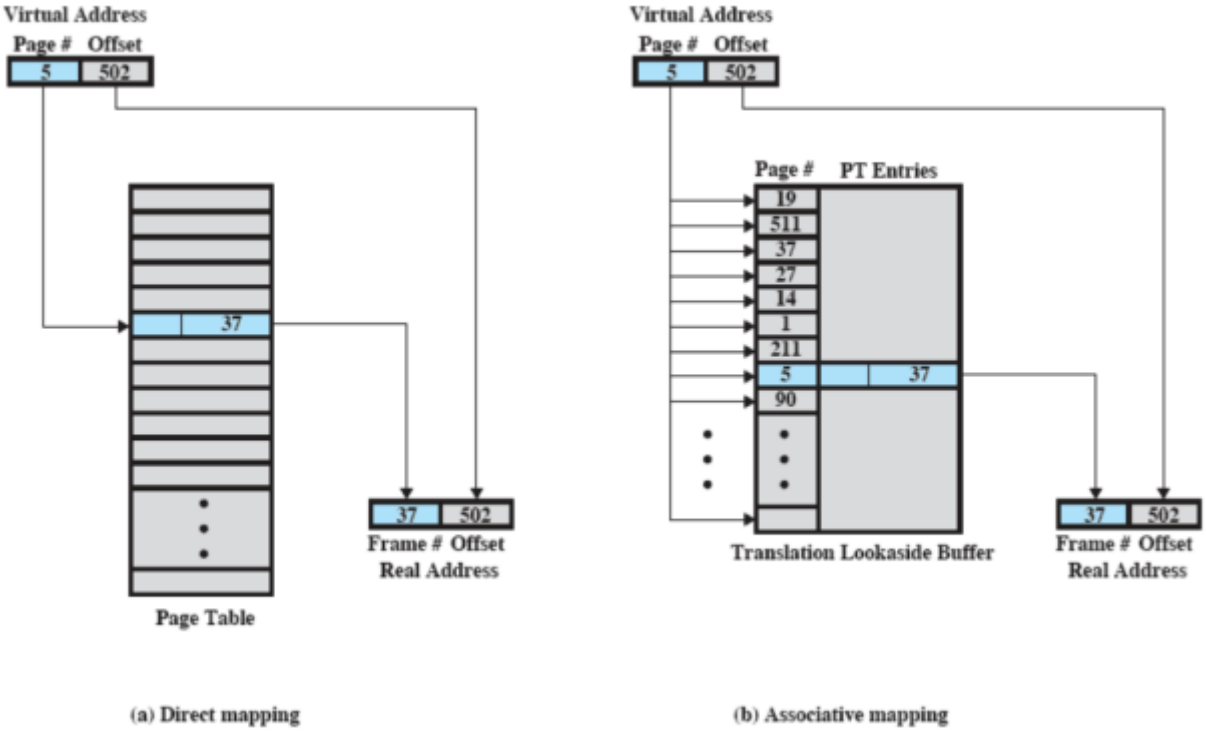
E' comunque necessario, anche senza TLB, dire al processore dove è la nuova tabella delle pagine (nel caso sia a 2 livelli, basta la page directory e gli indirizzi sono caricati in opportuni registri)

Mapping associativo

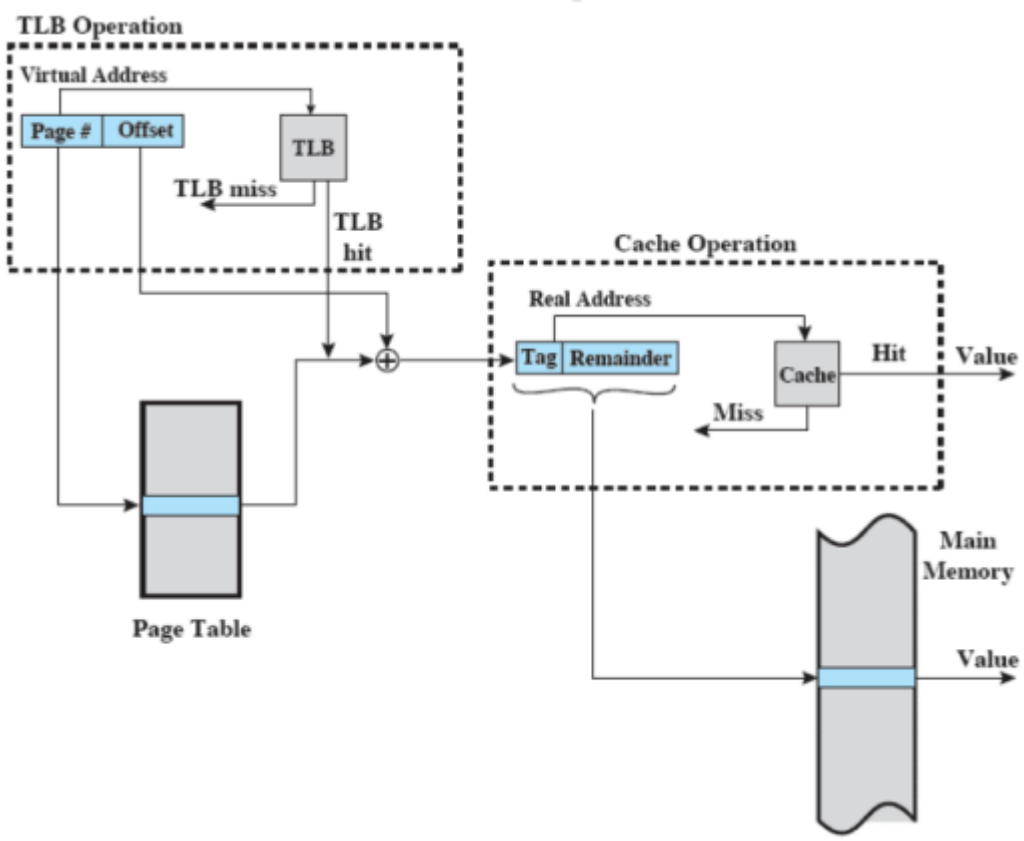
La tabella delle pagine contiene tutte le pagine di un processo, mentre per quanto riguarda il TLB, essendo una cache, non può contenere un'intera tabella delle pagine e non si può usare un indice per accedervi quindi teoricamente bisognerebbe scorrerla tutta

Si può risolvere questo problema sfruttando il parallelo e controllando contemporaneamente tutte le entry del TLB. Questo supporto hardware che mi permette di fare questa ricerca veloce è chiamato **mapping associativo**

Ci sta però un altro problema, bisogna fare in modo che nel TLB contenga solo pagine in RAM, altrimenti si incorrerebbe in un page fault dopo un TLB git, ma sarebbe impossibile accorgersene (il bit di presenza potrebbe infatti essere obsoleto). Quindi quando viene messo il bit di presenza a zero deve essere opportunamente modificato il TLB attraverso un reset parziale



TLB e cache



Dimensione delle pagine

Ma quanto dovrebbe essere la giusta dimensione di una pagina

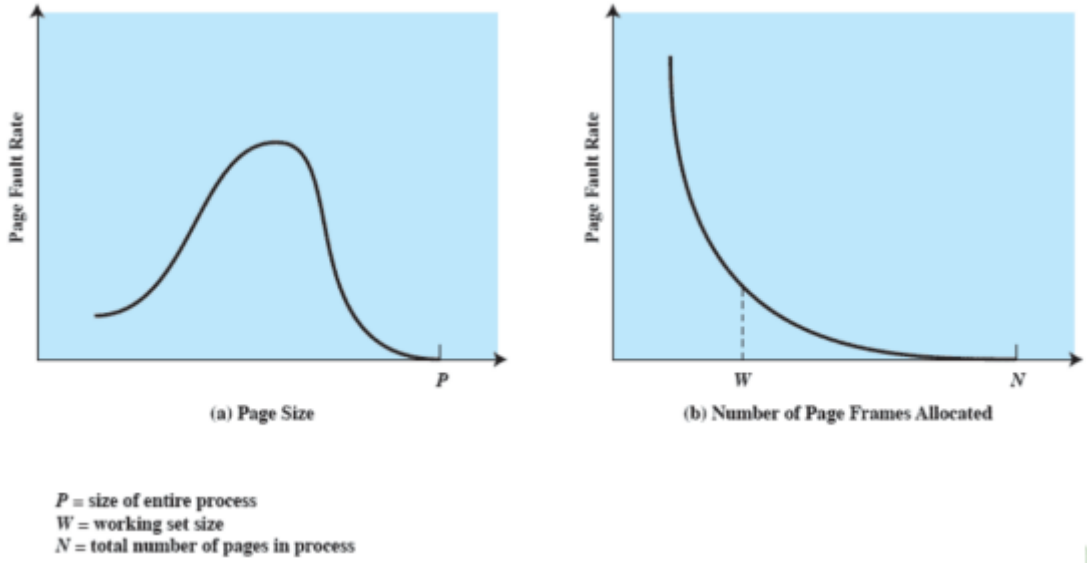
Perché grande

Più piccola è una pagina, minore è la frammentazione all'interno delle pagine ma è anche maggiore il numero di pagine per processo. Il che significa che è più grande la tabella delle pagine per ogni processo e quindi la maggior parte delle tabelle finisce in memoria secondaria.
La memoria secondaria è ottimizzata per trasferire grossi blocchi di dati, quindi avere le pagine ragionevolmente grandi non sarebbe male

Perché piccola

Più piccola è una pagina, maggiore il numero di pagine che si trovano in memoria principale. Lo stesso processo può riuscire ad avere un resident set più grande facendo diminuire i page fault e aumentanto la multiprogrammazione

Per risolvere questo problema sono stati fatti diversi esperimenti per capire quale fosse il giusto compromesso



Dimensione delle pagine in alcuni sistemi

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit word
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbyes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Nelle moderne architetture hardware si possono supportare diverse dimensioni delle pagine (anche fino ad $1\mathrm{GB}$) e il sistema operativo ne sceglie una: Linux sugli x86 va con $4\mathrm{kB}$
Le dimensioni più grandi sono usate in sistemi operativi di architetture grandi: cluster, grandi server, ma anche per i sistemi operativi stessi (kernel mode)

Segmentazione

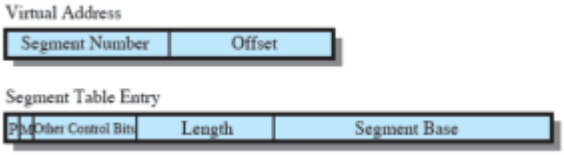
Permette al programmatore di vedere la memoria come un insieme di spazi (segmenti) di indirizzi la cui dimensione può essere dinamica. Questo viene usato per semplificare la gestione delle strutture dati che crescono.
Permette inoltre di:

- modificare e ricompilare i programmi in modo indipendente
- condividere dati
- proteggere dati

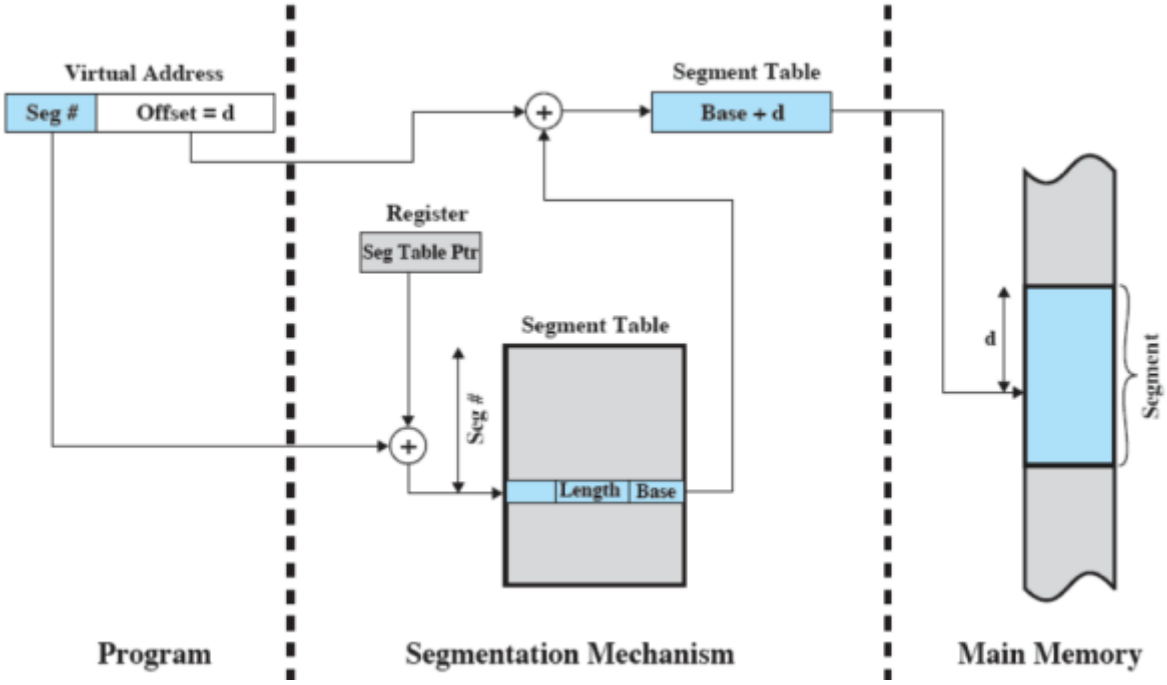
Organizzazione

Anche qui ogni processo ha una sua tabella dei segmenti e il process control block di un processo punta a tale tabella
Ogni entry di questa tabella contiene:

- indirizzo di partenza (in memoria principale) del segmento
- la lunghezza del segmento
- un bit per indicare se il segmento è in memoria principale o no
- un altro bit per indicare se il segmento è stato modificato in seguito all'ultima volta che è stato caricato in memoria principale



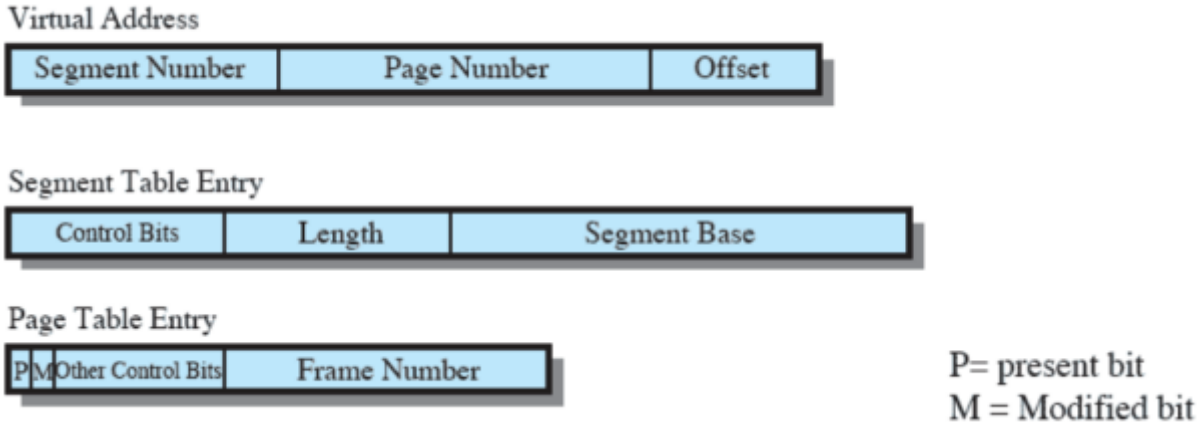
Traduzione degli indirizzi



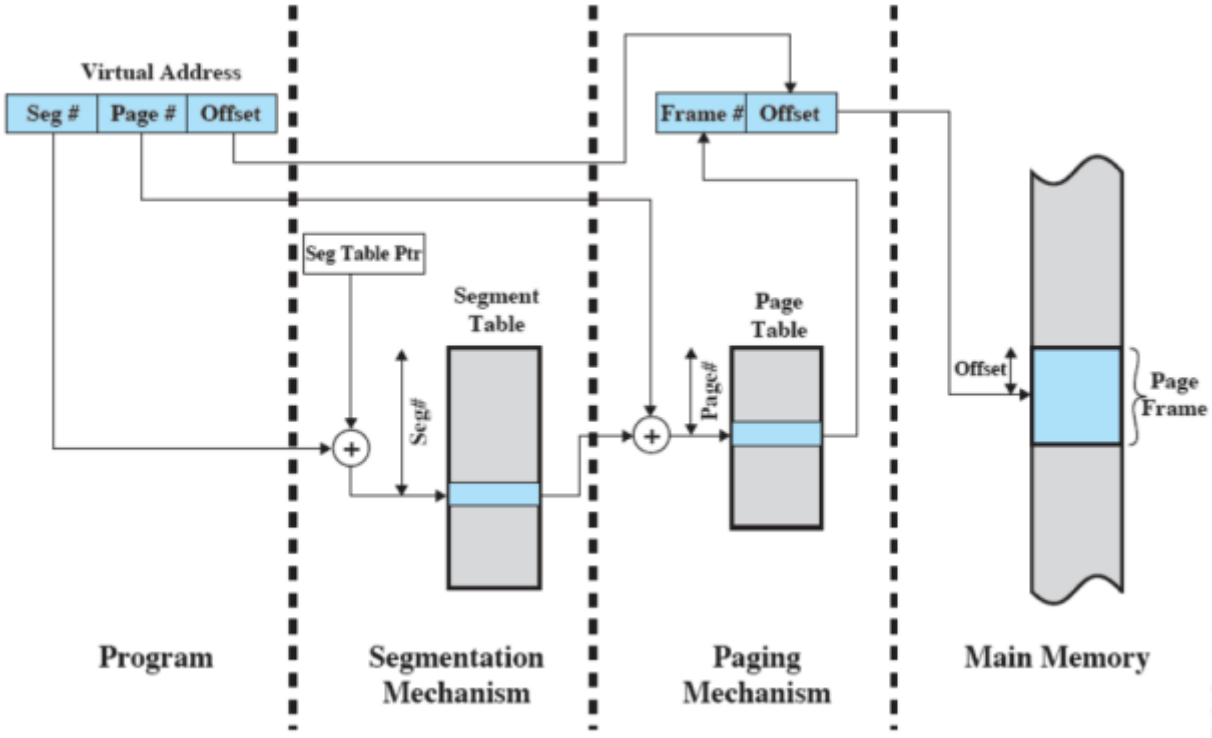
Paginazione e segmentazione

La paginazione è trasparente al programmatore; il programmatore infatti non ne è a conoscenza

La segmentazione è invece visibile al programmatore (se il programma è in assembler, altrimenti ci pensa il compilatore ad usare i segmenti)
Quindi ogni segmento viene diviso in più pagine

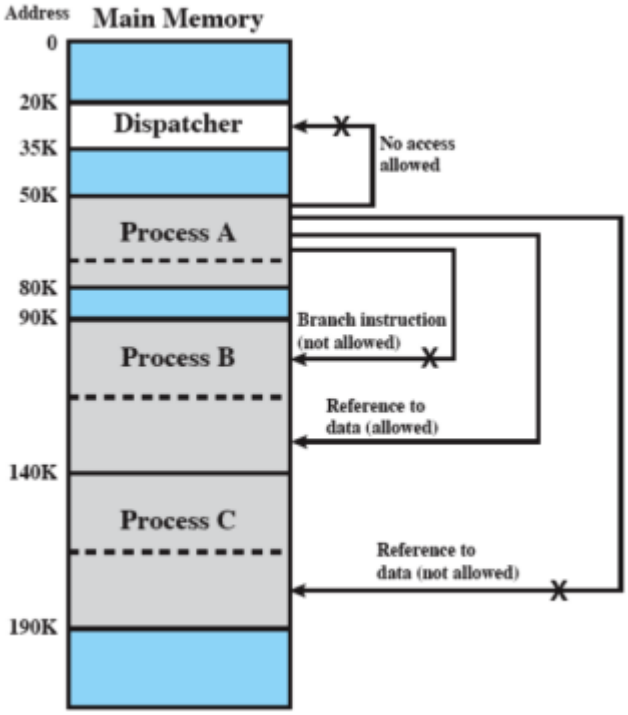


Traduzione degli indirizzi



Protezione e condivisione

Con la segmentazione, implementare protezione e condivisione. Dato che ogni segmento ha una base e una lunghezza è facile controllare che i riferimenti siano contenuti nel giusto intervallo. Per la condivisione, basta dire che uno stesso argomento serve a più processi


Address
Main Memory
0
20K
35K
50K
80K
90K
140K
190K
Dispatcher
Process A
Process B
Process C
No access allowed
Branch instruction (not allowed)
Reference to data (allowed)
Reference to data (not allowed)