

# La Memoria Virtuale

## Ottimizzazione della memoria centrale

### 1) Introduzione- Gerarchia della memoria

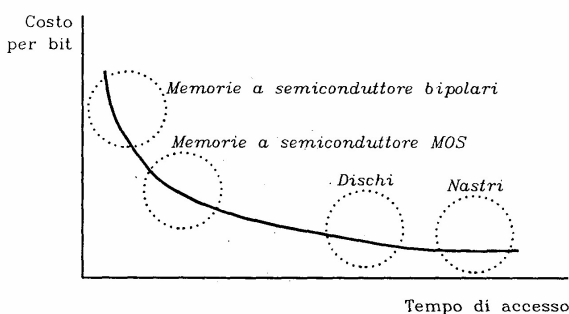
Da un punto di vista funzionale, ogni dispositivo di memorizzazione elettronica di informazioni presenta le stesse caratteristiche: costituisce infatti un serbatoio di sequenze di bit (*parole*) singolarmente accessibili mediante il loro *indirizzo*. Da un punto di vista comportamentale, esistono invece significative differenze fra memorie basate su principi fisici diversi o realizzate con diverse tecnologie e queste differenze sono tali da dividere in modo molto netto i campi applicativi delle diverse memorie.

Un parametro molto utile per qualificare i diversi tipi di dispositivi di memorizzazione di informazioni binarie è il costo per bit di informazione memorizzata. Tale parametro risulta ovviamente legato in modo molto stretto alla capacità di memoria dei vari dispositivi: se si ipotizza infatti di stanziare una certa cifra per la realizzazione di un qualsiasi dispositivo di memoria, sia esso di lavoro o di massa, il prodotto

$$\text{costo per bit} \times \text{numero di bit}$$

diventa una grandezza quasi costante, indipendente dal tipo di memoria.

Il motivo dell'uso di tale parametro è legato alla sua relazione con l'altro parametro fondamentale di qualificazione delle memorie: il tempo di accesso (cioè la rapidità di risposta alle richieste di trasferimento di informazioni).



Se si riporta infatti in un grafico l'andamento del costo per bit in funzione di tempo di accesso per diverse tipologie costruttive, il risultato ha un andamento iperbolico che mostra come il prodotto fra i due parametri sia praticamente una "costante universale", indipendente dalla tecnologia usata. Si può asserire che all'aumentare della capacità di memorizzazione di un supporto, diminuisce il costo per bit ma aumenta il tempo di accesso. Questa caratteristica è alla base delle considerazioni che portano al concetto di *gerarchia di memoria*.

### 2) Memoria virtuale

Premesso che il costo relativamente elevato dei dispositivi di memorizzazione ad alta velocità rende economicamente onerosa la realizzazione di calcolatori con grandi quantità di memoria di lavoro (dell'ordine a esempio delle centinaia o migliaia di Megabyte); al contempo avere, un'ampia disponibilità di memoria di lavoro si rivela particolarmente utile per almeno due motivi principali:

- uno spazio di indirizzamento ampio consente al programmatore di referenziare e utilizzare strutture dati complesse ed estese, come le matrici nei calcoli vettoriali o le tabelle nelle gestioni di archivi (basi di dati);
- i sistemi multiprogrammati (o addirittura multiutente) devono soddisfare la presenza contemporanea di vari programmi in memoria, le cui esigenze in termini di spazio — sia per il caricamento dei programmi stessi che per l'allocazione delle strutture dati — non sono note a priori.

Si potrebbe pensare di utilizzare dispositivi di memorizzazione a supporto magnetico mobile veloci (ad esempio dischi) per realizzare, a parità di costo, memorie di lavoro molto più estese (di almeno due ordini di grandezza); purtroppo i tempi di accesso necessari per ottenere da un disco l'informazione ricercata sono tali da impedirne l'uso come supporto di memoria di lavoro, soprattutto per la casualità con cui sono generati gli indirizzi da parte della CPU.

Un più attento esame delle procedure di generazione degli indirizzi di memoria di lavoro durante l'esecuzione di un programma consente però di evidenziare come la loro distribuzione non sia di fatto completamente casuale, e mostri invece una elevata probabilità che, a partire dalla generazione di un certo indirizzo di memoria, ne venga generato uno simile (o addirittura uguale) entro breve tempo (cioè a distanza di pochi accessi a memoria). Questo comportamento, noto come località degli accessi a memoria, deriva da

tre motivi concomitanti:

- una località *temporale*, dovuta al fatto che ogni programma ha una elevata probabilità di riutilizzare a breve le informazioni appena acquisite (come le istruzioni appartenenti a un ciclo, le variabili temporanee, le locazioni di memoria utilizzate per la pila di indirizzi di chiamata a e ritorno da sottoprogramma);
- una località *sequenziale*, dovuta al fatto che l'esecuzione di una istruzione ha una elevata probabilità (circa 80%) di essere seguita dall'esecuzione dell'istruzione immediatamente successiva nel programma, come pure che le elaborazioni sugli elementi di un vettore sono generalmente organizzate in modo da scandire sequenzialmente il vettore stesso;
- una località *spaziale*, dovuta al fatto che ogni programma è generalmente strutturato in moduli (procedure e funzioni) di dimensioni ridotte, che hanno elevata probabilità di usare informazioni spazialmente vicine (come le variabili definite all'interno del singolo modulo) e di eseguire parti di codice spazialmente vicine (una istruzione di ramificazione del flusso di esecuzione molto raramente trasferisce il controllo a istruzioni lontane).

Questo comportamento fa sì che diventi fattibile una organizzazione *virtuale* della memoria di lavoro, nella quale lo spazio di memoria utilizzabile dal singolo programma (quindi a maggior ragione dall'insieme dei vari programmi di un sistema multiprogrammato) è largamente superiore alle dimensioni fisiche della memoria di lavoro (effettivamente presente nel calcolatore) nella quale vengono temporaneamente ricopiate dalla memoria di massa le informazioni correntemente utilizzate dall'unità centrale.

Il termine "virtuale" deriva dal fatto che in questa organizzazione di memoria lo spazio di indirizzamento dell'unità centrale, quindi il singolo indirizzo da essa generato, non ha un riscontro fisico nella memoria di lavoro, bensì fanno riferimento a una memoria di lavoro virtuale (corrispondente in pratica alla memoria di massa), che è solo in parte presente nella memoria di lavoro reale del sistema.

Questo richiede una prima politica di gestione della memoria virtuale consistente in un metodo di "conversione" dell'indirizzo virtuale, emesso dall'unità centrale, nell'indirizzo fisico della cella di memoria di lavoro nella quale è stato ricopiato il valore della locazione virtuale cercata. Tale compito è demandato ad una unità funzionale dedicata - l'*unità di gestione della memoria* (MMU: Memory Management Unit) - interposta fra l'unità centrale e il bus di sistema.

Inoltre, quando l'unità centrale richiede l'accesso a una locazione di memoria virtuale non presente in una cella di memoria reale, si rende necessario effettuare una ricopiatura della memoria virtuale desiderata da memoria di massa in memoria di lavoro, eventualmente preceduta da una ricopiatura in senso inverso per salvare su memoria di massa il contenuto della memoria di lavoro che si vuole liberare (l'operazione di trasferimento da memoria di lavoro a memoria di massa in un sistema a memoria virtuale prende il nome di *swapping*). Per ottimizzare le prestazioni, si sfrutta in queste situazioni il principio della località e si trasferisce da memoria di massa un intero blocco di locazioni adiacenti contando su una elevata probabilità di doverle usare a breve.

Questo richiede una seconda politica di gestione della memoria virtuale, relativa ai criteri usati per definire i blocchi di informazione da trasferire ad ogni accesso a memoria di massa, e ai criteri usati per decidere quali porzioni di memoria di lavoro assegnare ai vari blocchi.

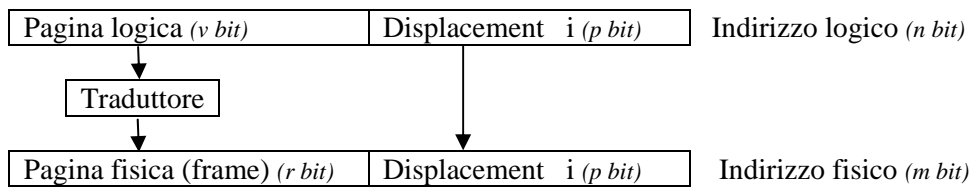
## 2.1) Memoria virtuale impaginata (paginazione)

Un primo metodo di partizione della memoria virtuale si basa sul concetto di *pagina*, cioè di blocco di parole consecutive - di dimensione prefissata - che costituisce l'unità minima di informazione trasferita dalla memoria di massa alla memoria di lavoro e viceversa durante le operazioni di *swapping* legate alla gestione della memoria virtuale. In questa visione, sia la memoria virtuale che la memoria di lavoro fisica vengono suddivise in pagine di uguali dimensioni, e la politica di gestione prevede il trasferimento di singole pagine dalla memoria di massa a quella di lavoro e viceversa.

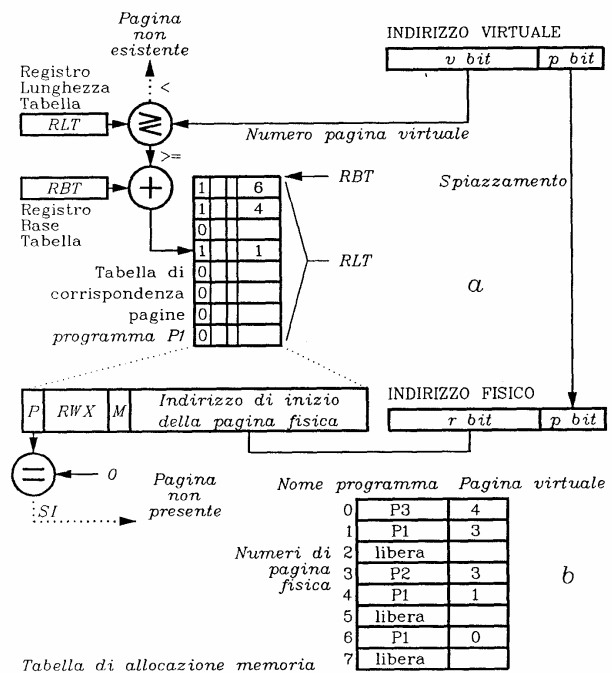
Nell'ipotesi che siano  $n$  i bit di indirizzamento virtuale, cioè i bit usati dell'unità centrale per referenziare una locazione di memoria, si può pensare che i  $p$  bit meno significativi di ogni indirizzo rappresentino la posizione (spiazzamento) della parola cercata rispetto all'inizio della pagina di  $2^p$  parole, mentre i  $v = n - p$  bit più significativi rappresentano il numero di pagina nello spazio di indirizzamento virtuale di  $2^v$  pagine.

L'operazione di conversione da indirizzo virtuale a indirizzo fisico - composto da  $m \leq n$  bit - consiste nella ricerca della posizione di memoria fisica nella quale la pagina virtuale referenziata è stata inserita, cioè nella sostituzione dei  $v$  bit di numero di pagina dell'indirizzo virtuale con gli  $r = m - p$  bit che rappresentano la

posizione della pagina fisica contenente la pagina virtuale cercata.



Questo si può fare costruendo, per ogni programma presente in memoria, una tabella del tipo di quella rappresentata in figura, e prevedendo all'interno dell'unità centrale un registro destinato a contenere l'indirizzo di inizio della tabella corrispondente al programma in esecuzione. Tale tabella ha un elemento (riga) per ogni pagina virtuale referenziabile dal programma; la posizione di ogni elemento nella tabella corrisponde alla posizione ( $v$ ) della pagina corrispondente nella memoria virtuale, e il contenuto di ogni elemento è la posizione ( $r$ ) occupata da tale pagina nella memoria fisica, più un bit ( $P$ ) che indica se la pagina è o meno presente in memoria di lavoro.



Ogni accesso a memoria da parte dell'unità centrale comporta la lettura dell'elemento della tabella la cui posizione è data dai  $v$  bit più significativi dell'indirizzo generato dall'unità centrale stessa e la verifica del bit di presenza  $P$ .

- Se il bit  $P$  di tale elemento è attivo, la pagina cercata è presente in memoria e l'indirizzo fisico della cella richiesta è ricavabile semplicemente facendo precedere i  $p$  bit di spiazzamento dagli  $r$  bit di posizione della pagina (è il caso della seconda pagina virtuale del programma schematizzato in figura  $a$ , caricata nella quinta pagina di memoria fisica).
- Se il bit  $P$  non è attivo, la pagina cercata non è presente e deve quindi essere caricata da memoria di massa prima di essere usata. In questo caso, l'esecuzione passa al sistema operativo, che per prima cosa deve identificare una pagina di memoria di lavoro libera nella quale caricare la pagina richiesta. A questo scopo, deve esistere una tabella di allocazione della memoria fisica — simile a quella rappresentata in figura  $b$  — che riporta l'indicazione dello stato delle pagine di memoria fisica (libere o occupate); qualora tutte le pagine siano occupate, la politica di scelta generalmente usata è quella che prevede di eliminare la pagina utilizzata meno di recente (politica LRU: Least Recently Used).

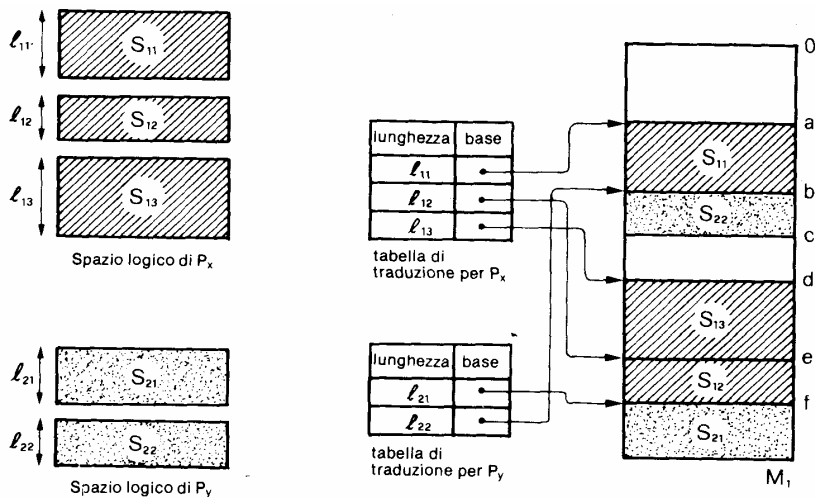
Con questo tipo di gestione, ogni accesso a memoria richiede in realtà almeno due accessi: uno alla tabella di conversione e l'altro alla locazione effettiva. Per ottimizzare il tempo di esecuzione, la tabella del programma in esecuzione viene generalmente ricopiata in registri ad alta velocità interni all'unità di gestione della memoria. In pratica, la tabella di conversione associata a ogni processo riporta in genere informazioni aggiuntive per ogni pagina virtuale, come i bit  $RWX$  (Read, Write, eXecut.e) usati per indicare i *diritti di accesso* alla pagina (se cioè è possibile leggerne, modificarne o eseguirne il contenuto, inteso come sequenza di istruzioni macchina), e il bit di modifica  $M$ , disattivato all'atto del caricamento della pagina da memoria di massa e attivato la prima volta che una parola di tale pagina viene alterata da un'operazione di scrittura da parte dell'unità centrale. Grazie a questo bit, è possibile - al momento in cui si rende necessario eliminare la pagina per sostituirla con una nuova - riconoscere se tale pagina è stata modificata o meno, e solo in caso affermativo salvarla ricopiandola su memoria di massa.

I problemi di questo tipo di gestione sono legati al **page fault** (mancanza di pagina) che provoca un momentaneo blocco (dell'ordine dei millisecondi) dell'elaborazione, per effettuare il **page swapping** della pagina richiesta. Da una parte si potrebbe pensare di aumentare la dimensione della pagina (per diminuire i

page fault) ma questo comporterebbe un eccessivo spreco di memoria dovuto al fatto che ogni programma utilizza solo una parte (il 50% in media) dell'ultima pagina. D'altra parte la diminuzione della dimensione delle pagine comporta al contrario un numero elevato di pagine ed una eccessiva dimensione delle tabelle associate ai vari programmi (caso limite : le dimensioni di ciascuna tabella possono diventare abbastanza grandi da richiedere una gestione a pagine delle tabelle stesse, con evidenti inefficienze e pesantezze di gestione). Altro problema legato al page fault è il **trashing** un caso limite in cui i page swapping si verificano con una frequenza eccessiva bloccando praticamente tutti i processi in esecuzione. Tale situazione si può contenere adottando : algoritmo L.R.U di sostituzione, bit modifica M e limitazione sul numero massimo di processi in memoria.

**2.2) Memoria virtuale segmentata**

In questo secondo metodo di gestione della memoria virtuale, questa viene suddivisa non in blocchi di uguale dimensione (le pagine) aventi un corrispettivo nella suddivisione della memoria fisica, bensì in unità logicamente separate, i *segmenti*, derivanti ad esempio dalla compilazione di programmi modulari, nei quali le traduzioni in codice macchina dei diversi moduli e le strutture dati di tali moduli diventano altrettanti segmenti. Un segmento è dunque un'unità di memoria di livello concettuale molto più elevato della pagina, in quanto ha un nome (definito dal programmatore), un tipo (dipendente dalla natura delle informazioni — ad

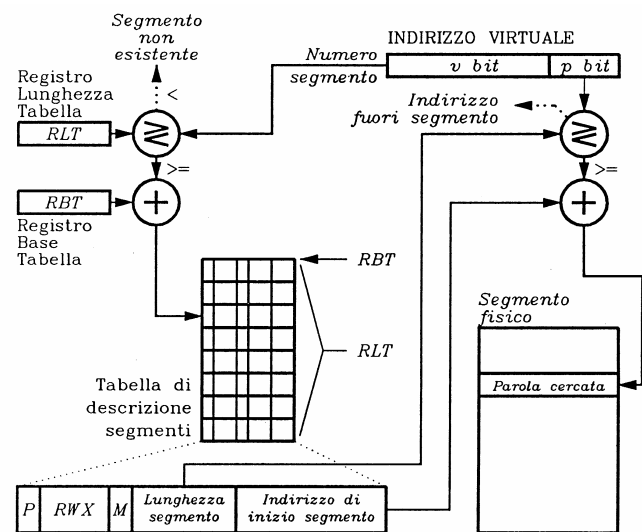


esempio codice o dati — in essa contenute) e una dimensione, e può anche avere un'appartenenza (nel caso di segmenti privati di un programma) o essere pubblico (nel caso di segmenti condivisi da più programmi). Consente quindi una gestione molto più sofisticata della memoria, in particolare per quanto riguarda la verifica dei diritti di accesso e di utilizzo delle informazioni comuni a più programmi (è ad esempio possibile definire un segmento come leggibile da un certo numero di programmi e modificabile solo da parte di alcuni).

La conversione di indirizzo nel caso di memoria segmentata richiede una struttura simile a quella rappresentata in figura: ad ogni programma viene associata una tabella che, per ogni segmento del programma, riporta l'indirizzo *i* di inizio di tale segmento in memoria di lavoro, la lunghezza *l* (espressa in numero di parole) del segmento, e il bit di presenza *P* (oltre a informazioni aggiuntive come i diritti di accesso *RWX* o il bit di modifica *M*, descritti a proposito della paginazione).

Ogni indirizzo virtuale è costituito da *p* bit meno significativi, che indicano la posizione della parola cercata all'interno del segmento, e da *v* bit che indicano il numero del segmento. La conversione comporta la lettura dell'elemento di posto *v* nella tabella di conversione e la verifica del bit di presenza *P*.

- Se il bit *P* è attivo (segmento presente) si confronta per prima cosa la lunghezza *l* con la posizione *p* della parola cercata: se  $l < p$ , l'unità centrale tenta di accedere a una cella esterna al segmento referenziato, e si genera un errore di superamento dei limiti gestibili a livello di sistema operativo. Nel caso invece l'accesso sia lecito, l'indirizzo fisico



Gestione della memoria centrale mediante segmentazione.

viene ricavato sommando la posizione  $p$  all'indirizzo di memoria fisica, letto dalla tabella, a partire dal quale il segmento interessato è stato caricato.

- Se invece  $P$  non è attivo (segmento assente), si rende necessario il relativo caricamento da memoria di massa. In questo caso, la gestione della memoria diventa però molto più complessa: la differente lunghezza dei segmenti comporta infatti una suddivisione molto più articolata della memoria fisica, non più partizionata in pagine di uguale lunghezza ma assegnata in base alle richieste dei programmi in esecuzione.

Il problema principale della gestione della memoria a segmenti è legato proprio alle conseguenze della diversa dimensione dei vari segmenti: ogni caricamento di un nuovo segmento, infatti, va ad occupare lo spazio lasciato libero da un segmento di dimensioni generalmente maggiori (molto raramente uguali, ovviamente mai minori) di quelle del segmento che si sta caricando, con il risultato di lasciare inutilizzata la parte di memoria corrispondente alla differenza dimensionale fra segmento vecchio e segmento nuovo (strategie di allocazione **first fit**, **best fit** e **worst fit**). Dopo un certo numero di operazioni di swapping, la memoria appare *frammentata*, cioè piena di "buchi" - residui dei vari caricamenti - singolarmente piccoli ma abbastanza numerosi da ridurre in modo inaccettabile lo spazio di memoria di lavoro utilizzabile, e da richiedere pertanto una compattazione della stessa (operazione estremamente lenta in quanto implica la ricopiatura dell'intero contenuto della memoria fisica).

### 2.3) Memoria virtuale segmentata e impaginata

Questo metodo di gestione prevede una convivenza dei due metodi appena discussi per sfruttare le caratteristiche positive di entrambi minimizzandone gli aspetti negativi. In questo caso, la divisione della memoria virtuale è ancora realizzata a segmenti di dimensione variabile, che sono però a loro volta suddivisi in pagine di lunghezza fissa. La conversione di indirizzi richiede l'uso di due livelli di tabelle, il primo destinato a identificare la posizione del segmento, il secondo a individuare la pagina all'interno del segmento (con un aumento considerevole di complessità dell'unità di gestione della memoria per effettuare la conversione in tempi tali da non penalizzare eccessivamente le operazioni di accesso). Il risultato è comunque una gestione particolarmente efficiente dello spazio di memoria fisico, che elimina i problemi di frammentazione e di relativa compattazione pur consentendo la gestione dei diritti di accesso a informazioni condivise da più programmi, tipica di una visione segmentata della memoria virtuale.

### 3) Esercizi

1. Considerare le seguenti richieste di riferimenti alla memoria di un processo in un sistema con memoria virtuale:

6 2 4 6 8 3 1 4 5 11 8 7 6 10 9 7 8 11 2

Illustrare il comportamento dell'algoritmo L.R.U di sostituzione delle pagine per una memoria fisica di 5 blocchi. Calcolare inoltre il numero di page fault che si verificano.

2. Un sistema di gestione basato su swapping utilizza una lista per rappresentare lo stato di allocazione della memoria.

Ad esempio:

$(P,0,12) \rightarrow (H,12,5) \rightarrow (P,17,6) \rightarrow (H,23,15) \rightarrow (P,38,5) \rightarrow (H,43,12) \rightarrow (P,55,10) \rightarrow (H,65,8)$

dove  $P$  indica memoria occupata da un processo e  $H$  denota memoria libera, il primo numero indica il blocco di partenza ed il secondo numero il numero di blocchi allocati contigui.

Supponiamo che arrivino in tempi successivi le richieste di allocazione di memoria per tre processi, il primo di dimensione 7, il secondo di dimensione 10 ed il terzo di dimensione 1.

Dire (fornendo la lista di allocazione) come si comporta il gestore della memoria nel caso di politica

- Best fit
- First fit

3. Un processo, composto da 8 pagine logiche, viene eseguito su un sistema di gestione basato su swapping, con algoritmo di sostituzione LRU e che gli mette a disposizione al massimo 6 frame di memoria per essere eseguito. Immaginando di aver già ricevuto le richieste per le pagine logiche 1,2,7,5, 6 e che la situazione della memoria allocata sia la seguente:

Si dia una rappresentazione della memoria logica e fisica nel caso in cui le successive chiamate siano le seguenti: 3,4,5,6,8,1. Dire anche quanti saranno page fault ed i page swapping.

Pag.logica	Frame	v/i
1 (A)	5	v
2 (B)	6	v
3 (C)		i
4 (D)		i
5 (E)	2	v
6 (F)	1	v
7 (G)	3	v
8 (H)		i