**Gestione della memoria**

**Considerazioni preliminari**

Nell’ottica degli utenti applicativi la memoria deve essere capiente, veloce e permanente (non volatile).

Solo l’intera gerarchia di memoria nel suo insieme possiede tutte queste caratteristiche.

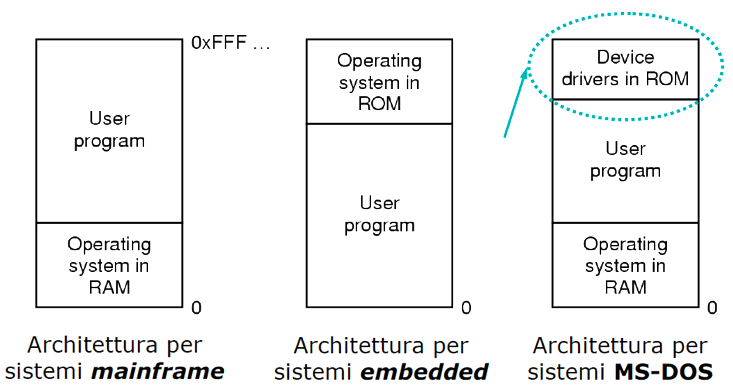
Il gestore della memoria è la componente di S/O incaricata di soddisfare le esigenze di memoria dei processi.

Esistono due classi fondamentali di sistemi di gestione della memoria:

1. Per processi allocati in modo fisso
2. Orientate a processi soggetti a migrazione da memoria principale a disco durante l’esecuzione

La memoria disponibile è in generale inferiore a quella necessaria per tutti iprocessi attivi simultaneamente

**Sistemi monoprogrammati**



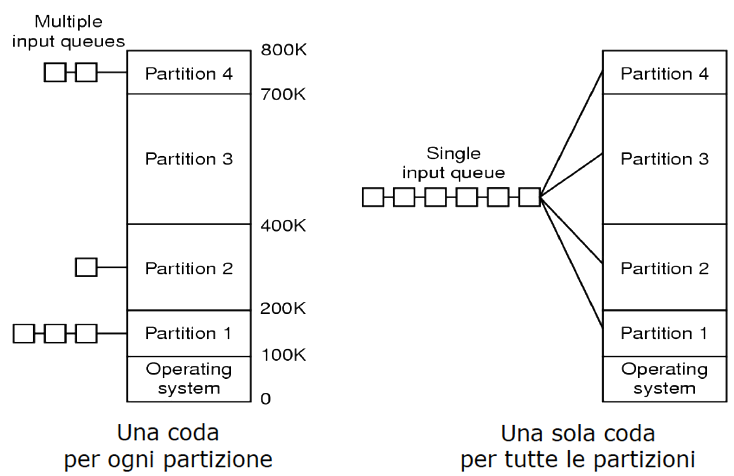
Esegue un solo processo alla volta.

La memoria disponibile è ripartita solo tra quel processo e il S/O.

L’unica scelta progettuale rilevante in questo caso è decidere dove allocare la memoria (dati e programmi) del S/O.

La parte di S/O ospitata in RAM è però solo quella che contiene l’ultimo comando invocato dall’utente.

**Sistemi multiprogrammati**



La forma più rudimentale di gestione della memoria per questi sistemi crea una partizione per ogni processo

* Staticamente all’avvio del sistema
* Le partizioni possono avere dimensione diversa

Il problema diventa assegnare dinamicamente processi a partizioni – Minimizzando la frammentazione interna

A ogni nuovo processo (o lavoro) viene assegnata la partizione di dimensione più appropriata

* Una coda di processi per partizione
* Scarsa efficacia nell’uso della memoria disponibile

Assegnazione opportunistica

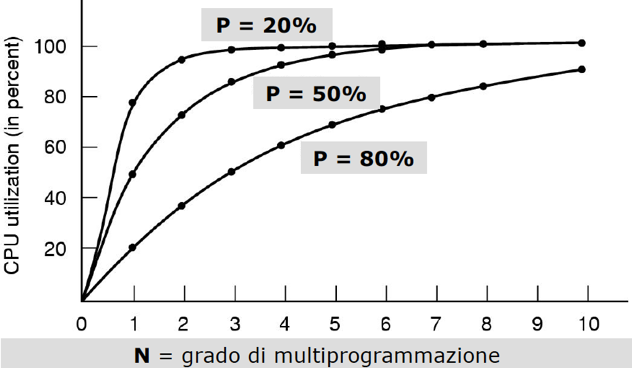
– Una sola coda per tutte le partizioni

• Quando si libera una partizione questa viene assegnata al processo a essa più adatto e più avanti nella coda

• Oppure assegnata al “miglior” processo scandendo l’intera coda

– I processi più “piccoli” sono discriminati quando invecemeriterebbero di essere privilegiati in quanto più interattivi

**Valutazione dei vantaggi della multiprogrammazione**

Valutazione probabilistica di quanti processi debbano eseguire in parallelo per massimizzare l’utilizzazione della CPU

– Sotto l’ipotesi che

• Ogni processo impegni il P% del suo tempo in attività di I/O

• N processi simultaneamente in memoria

– L’utilizzo stimato della CPU allora è 1 – PN

**Esempio Progettazione Memoria**

Si consideri un computer con 32 MB di memoria e 80% di attesa I/O media per ogni processo

– 16 MB riservati per il sistema operativo

– 4 MB riservati per ciascun processo

– In totale si hanno quindi 4 processi simultaneamente in memoria

– Con P = 0,8 si ha una utilizzazione della CPU di 1 -0,84 = 60%

Aggiungendo altri 16 MB

– Si possono avere 8 programmi simultaneamente in memoria

– Con P = 0,8 si ha una utilizzazione della CPU di 1 -0,88 = 83%

Aggiungendo altri 16 MB

– Si possono avere 12 programmi simultaneamente in memoria

– Con P = 0,8 si ha una utilizzazione della CPU di 1 -0,812 = 93%

**Rilocazione e protezione**

**Rilocazione** – Interpretazione degli indirizzi emessi da un processo in relazione alla sua collocazione corrente in memoria

• Occorre distinguere tra riferimenti assoluti permissibili al programma e riferimenti relativi da rilocare

**Protezione** – Assicurazione che ogni processo operi soltanto nello spazio di memoria a esso permissibile

• Soluzione storica adottata da IBM

– Memoria divisa in blocchi (2 kB) con codice di protezione per blocco (4 bit)

– La PSW di ogni processo indica il suo codice di protezione

– Il S/O blocca ogni tentativo di accedere a blocchi con codice di protezione diverso da quello della PSW corrente

• Soluzione combinata (rilocazione + protezione)

– Un processo può accedere memoria solo tra la base e il limite della partizione a esso assegnata

– Valore base aggiunto al valore di ogni indirizzo riferito (operazione costosa)

– Il risultato confrontato con il valore limite (operazione veloce)

**Swapping**

La tecnica più rudimentale per alternare processi in memoria principale senza garantire allocazione fissa.

Trasferisce processi interi e assegna partizioni diverse nel tempo.

Il processo rimosso viene salvato su memoria secondaria – Ovviamente solo le parti modificate

Processi diversi richiedono partizioni di ampiezze diverse assegnate ad hoc-

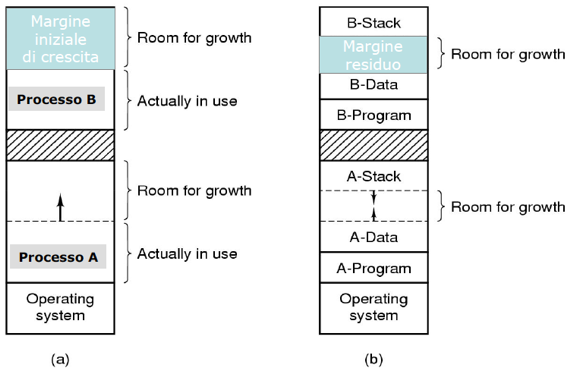
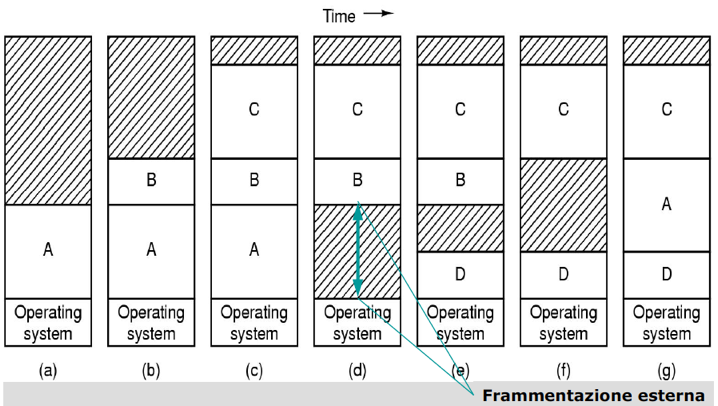
– Rischio di frammentazione esterna

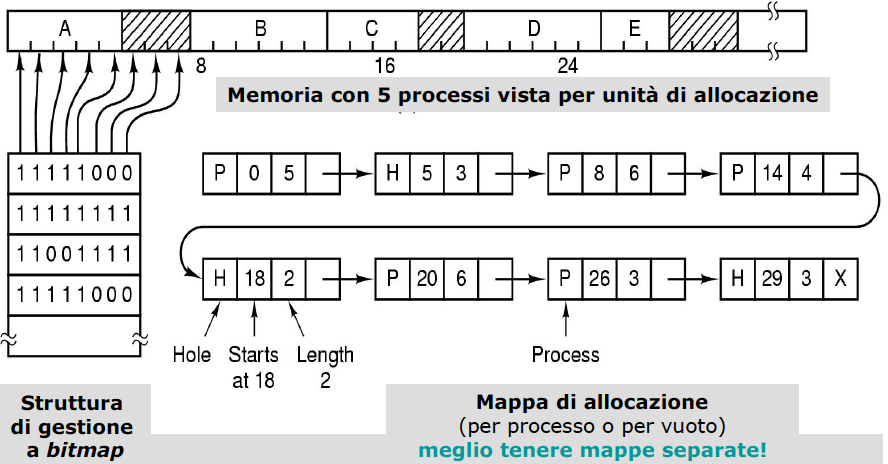
– Occorre ricompattare periodicamente la memoria principale. Pagando un costo temporale importante! - Spostando 4 B in 40 ns., servono 5.37 s. per una RAM ampia512 MB

Le dimensioni di memoria di un processo possono variare nel tempo!

– Difficile ampliare dinamicamente l’ampiezza della partizione assegnata

– Meglio assegnare con margine

**Strutture di gestione**

Con memoria principale allocata dinamicamente è essenziale tenere traccia del suo stato d’uso

Due strategie principali

– **Mappe di bit**

• Memoria vista come insieme di unità di allocazione (1 bit per unità) – Unità piccole → struttura di gestione grande » Esempio: Unità da 32 bit e RAM ampia 512 MB → struttura ampia 128 M bit = 16 MB → 3.1 % (= 1/32)

La strategia alternativa usa **liste collegate**.

Nella sua versione più semplice la memoria è vista a segmenti

• Segmento = processo | spazio libero tra processi

• Ogni elemento di lista rappresenta un segmento

– Ne specifica punto di inizio, ampiezza e successore

– Liste ordinate per indirizzo di base

Varie strategie di allocazione

• First fit : il primo segmento libero ampio abbastanza

• Next fit : come First fit ma cercando sempre avanti

• Best fit : il segmento libero più adatto

• Worst fit : sempre il segmento libero più ampio

• Quick fit : liste diverse di ricerca per ampiezze “tipiche”

**Memoria Virtuale**

Una singola partizione o anche l’intera RAM sono presto divenute insufficienti per ospitare un intero processo.

La prima soluzione fu di suddividere il processo in parti chiamate overlay

– Veniva caricata in RAM una parte alla volta

– Non appena “consumata” le veniva sovrapposta la parte successiva

– Suddivisione a cura del programmatore!

L’idea di memoria virtuale nasce nel ’61.

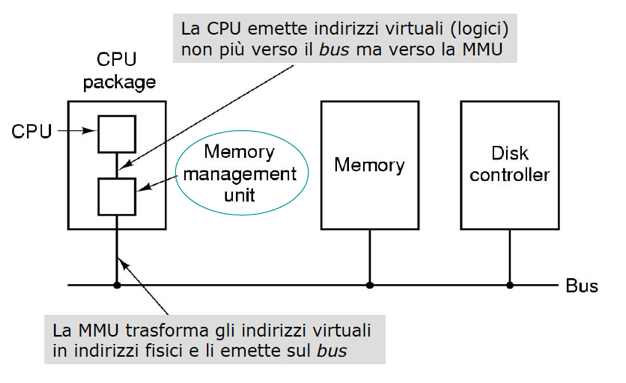
Il principio cardine è che un singolo processo può liberamente avere ampiezza maggiore della RAM disponibile

– Basta caricarne in RAM solo la parte strettamente necessaria lasciando il resto su disco

– Senza intervento del programmatore

Ogni processo ha un suo proprio spazio di memoria virtuale.

Due tecniche alternative di gestione: **paginazione** e **segmentazione**

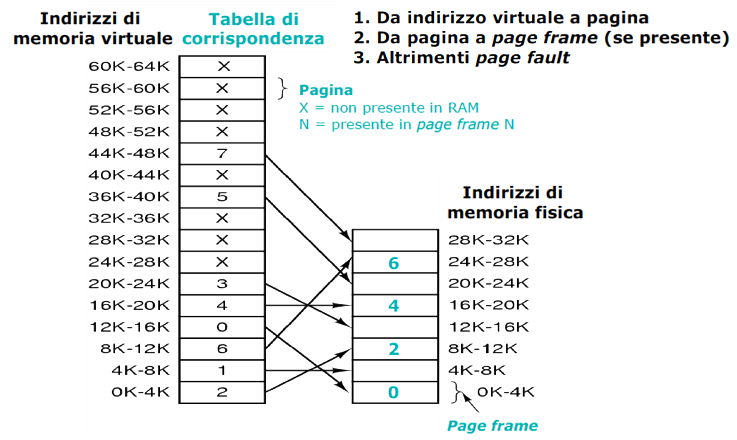


Gli indirizzi generati dal processo non denotano più direttamente una locazione in RAM

– Ma vengono interpretati da un’unità detta MMU che li mappa verso indirizzi fisici reali prima di essere emessi sul bus

– Il tipo di interpretazione a carico della MMU dipende dalla tecnica usata per la gestione della memoria virtuale

**Paginazione: premesse**

La memoria virtuale è suddivisa in unità a dimensione fissa dette pagine.

La RAM è suddivisa in unità “cornici” ampie come le pagine (page frame).

I trasferimento da e verso disco avvengono sempre in pagine.

Di ogni pagina occorre sapere se sia presente in RAM oppure no

– Bit di presenza

– Se una pagina è assente quando riferita si genera un evento page fault gestito dal S/O tramite trap

**Paginazione: strutture**

La traduzione da indirizzo virtuale a fisico avviene tramite una tabella delle pagine

– Indicizzata per numero di pagina: Indirizzo fisico = φ (indirizzo virtuale)

La tabella può essere molto grande

– Indirizzi virtuali da 32 bit e pagine da 4 KB → memoria virtuale da 4 GB = 1 M pagine!

Ciascun processo ha la sua (grande) tabella delle pagine – Poiché ha il suo spazio di indirizzamento virtuale

La traduzione deve essere molto veloce

– Ogni istruzione potrebbe fare riferimento più volte alla tabella delle pagine

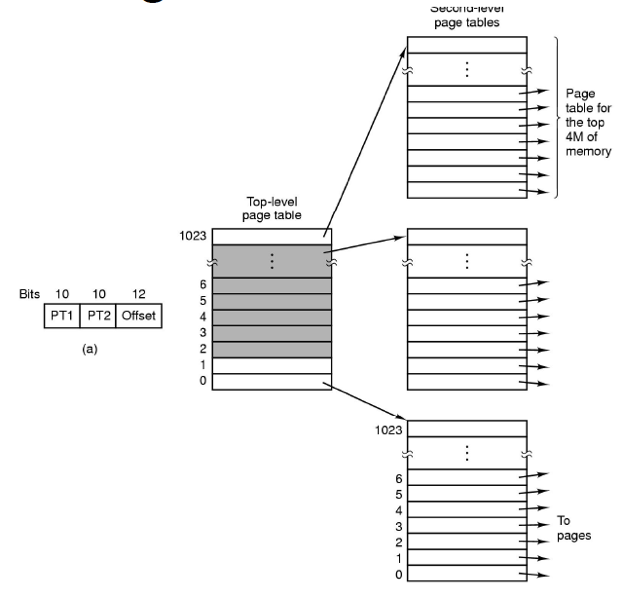
• Dunque se un’istruzione impiega ad es. 4 ns, allora il riferimento alla page table deve avvenire in circa 1 ns, viceversa sarà unbottleneck del sistema

– Ogni indirizzo emesso dal processo (istruzione o operando)

deve essere tradotto

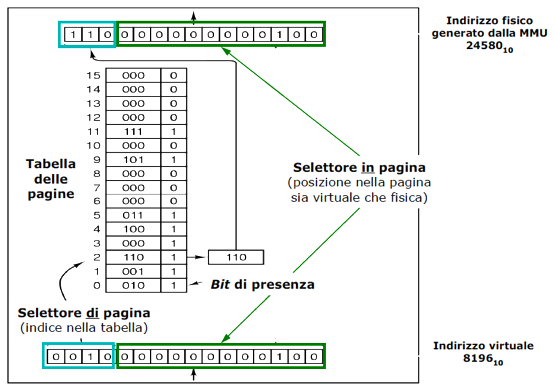
• Semplicemente (e concettualmente) potrebbe utilizzare un vettore di registri (uno per ogni pagina virtuale) caricato a ogni cambio di contesto (vedi figura nella slide seguente)

– Lineare e non si rischia di dover accedere a memoria per scoprire il riferimento, ma costoso cambiare tutti i registri ad ogni context switch

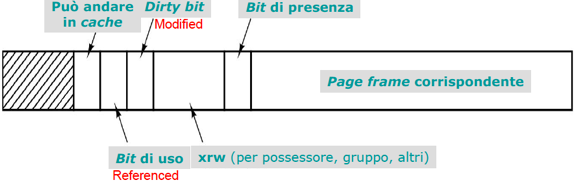
• Oppure come una struttura sempre residente in RAM

– Un singolo registro punta all’inizio della page table

– Difficile che sia usato come soluzione in modo puro



Una riga nella tabella delle pagine (ampiezza tipica 32 bit)



L’indirizzo di disco ove la pagina si trova quando non è in RAM non è nella tabella!

– La tabella delle pagine serve alla MMU (hardware)

– Il caricamento della pagina da disco viene effettuato dal S/O(software) all’occorrenza di un page fault

– L’informazione dell’uno non serve all’altro

La tabella delle pagine è così grande che non può risiedere su registri

– Dunque deve stare in RAM

– Riferirla per ogni indirizzo emesso (istruzioni e operandi) ha un

impatto devastante sulle prestazioni

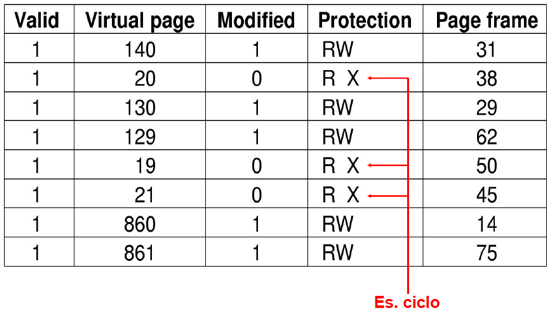
Serve una struttura supplementare (HW) più agile chene sia come una cache

– Piccola memoria associativa che consente scansione parallela (translation lookaside buffer, TLB)

• Solitamente interna alla MMU

• Ricerca su tutte le righe simultaneamente

– Basata sull’osservazione che un processo in genere usa più frequentemente poche pagine



Ogni indirizzo emesso verso la MMU viene prima trattato con la TLB

– Se la sua pagina è presente e l’accesso richiesto è permesso la traduzione avviene tramite TLB

• Senza accedere alla tabelle delle pagine

– Se non presente si ha l’equivalente di una cache miss e le informazioni richieste vengono caricate in TLB dalla tabella delle pagine

• Rimpiazzando una cella in TLB e riflettendone il valore nella tabella delle pagine – Ma solo se cambiato!

Oggi le TLB sono prevalentemente realizzate in software invece che in hardware nelle MMU

– Le prestazioni sono accettabili

– La MMU ne guadagna in semplicità e riduzione di spazio che viene dedicato ad altri usi ritenuti più vantaggiosi (cache)

Con le architetture a 64 bit però le tabelle delle pagine assumono dimensioni proibitive

– 64 bit → memoria virtuale da 16 EB (1 E = 1 G x 1 G)

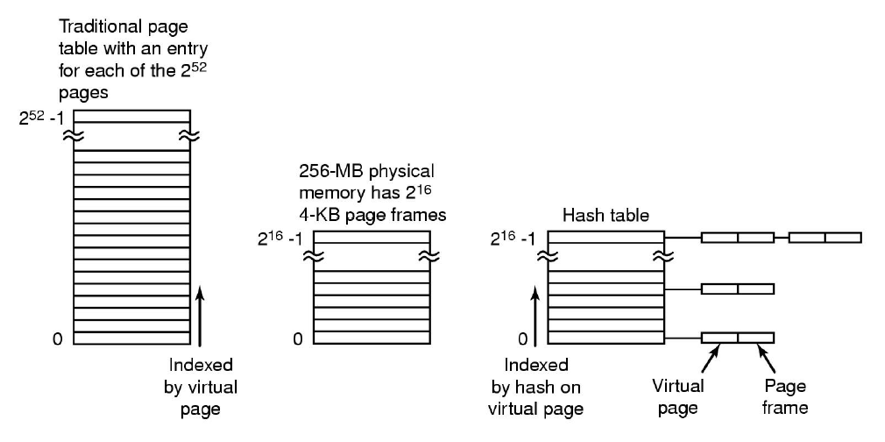
– Pagine da 4 KB V 4 P pagine (1 P = 1 M x 1 G)

– 32 bit per pagina in tabella → ampiezza 16 PB! - Serve un’altra soluzione

La soluzione adottata impiega una tabella invertita

* Non più una riga per pagina ma per page frame in RAM - Considerevole risparmio di spazio
* La traduzione da virtuale a fisico diventa però molto più complessa - Poiché la pagina potrebbe risiedere in qualunque page frame bisognerebbe scandire l’intera tabella per trovarla

– Per ogni indirizzo emesso dal processo!

– Grande dispendio di tempo

* Ricerca velocizzata dall’uso di TLB
* E anche realizzando la tabella invertita come una tabella hash indicizzata da f Hash (indirizzo virtuale)

• I dati relativi alle pagine i cui indirizzi virtuali indicizzino una stessa riga di tabella vengono collegati in lista

**Paginazione: rimpiazzo**

Quando si produce un page fault il S/O deve rimpiazzare una pagina

– Salvando su disco la pagina rimossa - Ma solo se modificata nell’uso

Inopportuno rimpiazzare pagine in uso frequente

– Altrimenti si paga prezzo doppio dovendole riportare troppo presto in RAM.

Problema del tutto analogo a quello della cache

– Anche di quelle emulate a software per la gestione di informazioni logiche

Rimpiazzo ottimale (optimal replacement)

– Rimpiazza la pagina in memoria che non sarà usata per maggior tempo

La scelta perfetta non è realizzabile

– Perché il S/O non ha modo di sapere quali pagine il processo accederà in futuro

• Un po' come scegliere il processo più breve

Le scelte realizzabili sono sempre e solo approssimazioni sotto-ottimali

– Sulla base di osservazioni empiriche sull’uso recente delle pagine attualmente in RAM

• NRU (Not Recently Used)

– Per ogni page frame vengono aggiornati

• Bit M (modified), inizializzato a 0 dal S/0

• Bit R (referenced), posto a 0 periodicamente dal S/O per stimare la frequenza d’uso

– Le pagine nei page frame sono classificate in

• Classe 0: non riferita, non modificata

• Classe 1: non riferita, modificata

• Classe 2: riferita, non modificata

• Classe 3: riferita, modificata

– NRU sceglie una pagina a caso nella classe non vuota a indice più basso

• FIFO

– Rimuove la pagina di ingresso più antico in RAM

• Basta una lista ordinata di page frame

– Ogni inserimento viene marcato in coda e la rimozione avviene dalla testa

• Second chance

– Corregge FIFO rimpiazzando solo le pagine con bit R = 0

• Altrimenti il page frame viene considerato come appena caricato, posto in fondo alla coda e R viene posto a 0

• Degenera in FIFO quando tutti i page frame siano stati recentemente riferiti

**Orologio**

– Come SC ma i page frame sono mantenuti in una lista circolare - L’indice di ricerca si muove come una lancetta

**LRU** (Least Recently Used)

– Approssima l’algoritmo ottimale

– Richiede lista aggiornata ad ogni riferimento a memoria

– Necessita di hardware dedicato

**NFU** (Not Frequently Used)

– Realizzabile a software

– Per ogni page frame aggiorna periodicamente un “contatore” C che cresce di più se R . 1

– PROBLEMA: non dimentica nulla!

**Aging** (not frequently used modificato)

– Realizzabile a software

– Per ogni page frame aggiorna periodicamente un “contatore” C che cresce di più se R . 1

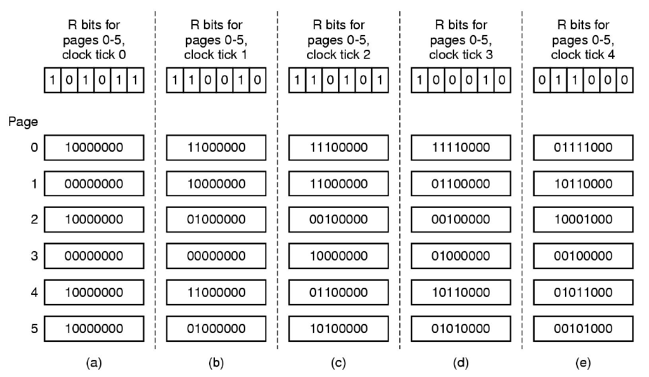
• Non incrementa C con R ma gli inserisce R a sinistra

– Approssima LRU con differenze importanti

• Valuta solo periodicamente (a grana grossa)

• Usando N bit per C perde memoria dopo N aggiornamenti

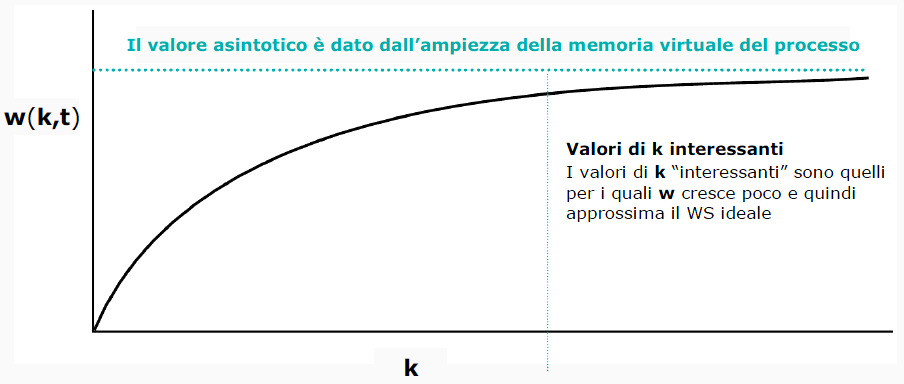
Aging (not frequently used modificato):



**Paginazione: working set**

Studi accurati mostrano come i processi emettano la maggior parte dei loro riferimento entro un ristretto spazio locale – Località dei riferimenti

Working set (WS) è l’insieme di pagine che un processo ha in uso a un dato istante

– Se la memoria non basta ad accogliere il WS si crea il fenomeno di thrashing

– Se il WS viene caricato prima dell’esecuzione si ha prepaging (evitando page fault)

– w(k, t) è l’insieme di pagine che soddisfano i k riferimenti emessi al tempo t • Funzione monotonica crescente

Se si conoscesse il WS dei processi le pagine da rimpiazzare sarebbero quelle che non vi fossero comprese.

Conoscere precisamente il WS dei processi a tempo d’esecuzione è però troppo costoso

– Quanto deve valere k?

– Più facile fissare t come (t, t + Δt)

• Considerando t come valore dell’effettivo tempo di

esecuzione di quel processo (tempo virtuale corrente) – Non del tempo trascorso!

• WS è fatto dalle pagine riferite dal processo nell’ultimo Δt

**Paginazione: rimpiazzo**

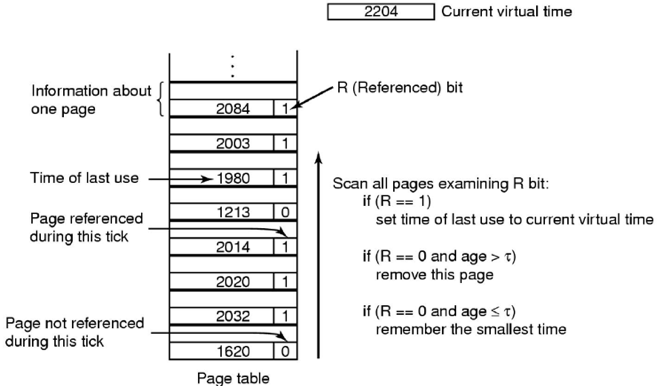
**WS approssimato**

– Simile all’Aging

• Ogni page frame in RAM ha un attributo temporale che indica

se a un dato istante appare come riferita (R =1)

– Tale attributo prende il valore t del tempo virtuale corrente all’arrivo di un page fault

– R e M sono posti a 1 dall’hardware

– R è posto a 0 (se non in uso) da un controllo periodico e al page fault

• Al page fault sono rimpiazzabili le pagine con R = 0 e valore

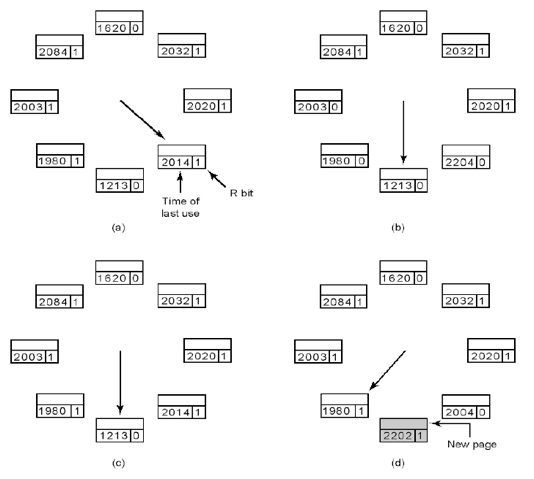
di attributo antecedente all’intervallo (t - Δt, t)

– Se non ci sono, si prende la più vecchia con R = 0

– Se all’istante t tutti i page frame avessero R = 1 verrebbe rimpiazzata una pagina scelta a caso, con M = 0

• Nel caso peggiore bisogna scandire l’intera RAM!

**WS approssimato con orologio (molto usato in pratica)**

– Page frame organizzati in lista circolare

• Come per l’orologio semplice

• Ma con le informazioni del WS approssimato

– Una “lancetta” indica il page frame corrente

• Al page fault se R . 1 la lancetta avanza e R . 0

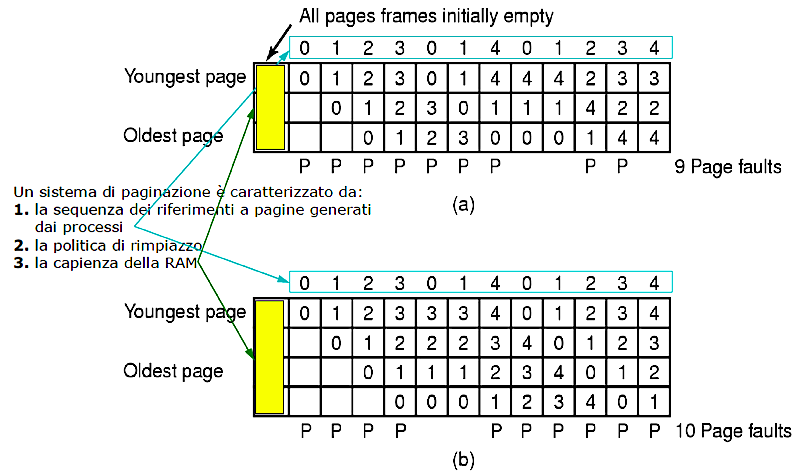
• Se R . 0 si valuta l’attributo temporale

– Se fuori da w(k,t) e con M . 0 allora rimpiazzo

– Altrimenti il page frame va in una coda di trasferimento su disco e la lancetta avanza » Alla ricerca di un page frame rimpiazzabile direttamente » Quando N pagine in coda si trasferisce su disco

• Se nessun page frame è rimpiazzabile allora si sceglie una

pagina con M . 0 altrimenti quella cui punta la lancetta

**Paginazione: l’anomalia di Belady**

Nel 1969 Lazlo Belady mostrò che la frequenza di page fault non sempre decresce al crescere dall’ampiezza della RAM

– Un semplice contro-esempio usando FIFO come strategia di rimpiazzo

• Sequenza di riferimenti: 0 1 2 3 0 1 40 1 2 34

• RAM con 3 page frame : 9 page fault

• RAM con 4 page frame : 10 page fault

LRU è immune dall’anomalia di Belady – Ma la sua forma “pura” è irrealizzabile

Una classe di algoritmi particolarmente interessante è quella che soddisfa la proprietà:

– M(m, r) M (m+1, r)

– Dove m rappresenta il numero di page frame, mentre r sono i riferimenti

– “assumendo gli stessi riferimenti, le pagine caricate con m page frame sono un sottoinsieme di quelle caricate con m+1 page frame”

Detti stack algorithms – Sono immuni dall’anomalia di Belady

• LRU, Optimal Replacement

**Paginazione: criteri di progetto**

Nel rimpiazzare una pagina occorre scegliere consapevolmente tra

– Politiche locali

• Rimpiazzo nel WS del processo che ha causato il page fault

• In tal caso ogni processo conserva una quota fissa di RAM

– Politiche globali

• La scelta avviene tra page frame senza distinzione di processo

• L’allocazione di RAM a disposizione di ogni processo varia dinamicamente nel tempo

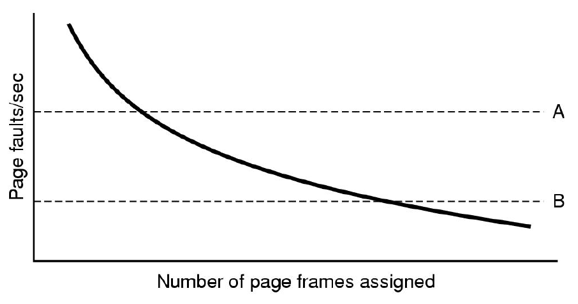
Le **politiche globali** sono più efficaci

– Specialmente se l’ampiezza del WS può variare durante

l’esecuzione, però bisogna decidere quanti page frame assegnare a ogni singolo processo

Le **politiche locali** hanno prestazioni inferiori

– Se il WS di un processo cresce l’allocazione fissa causa

rimpiazzi indesiderati - Thrashing

– Anche con RAM disponibile non usata da altri processi

– Se il WS si riduce si ha invece spreco di memoria

Non tutte le politiche si adattano all’uso in entrambe le varianti

È evidente che al crescere del WS diminuisce il rischio di page fault

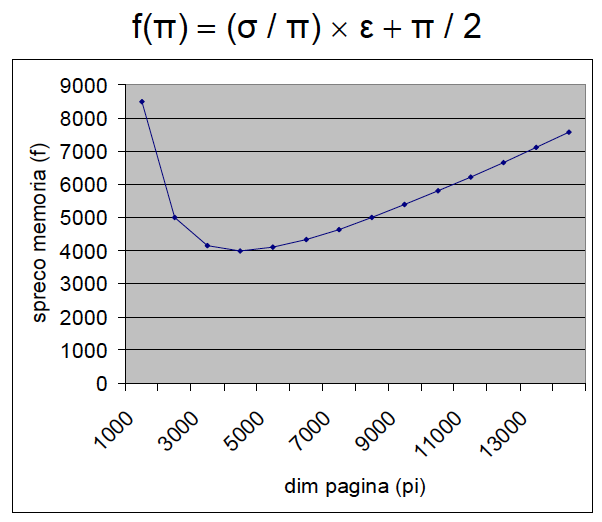
Controllo del carico

– Anche con le migliori politiche può accadere che a volte il sistema subisca thrashing

• Se i WS di tutti i processi eccedono la capacità di

memoria – PFF (Page Fault Frequency) indica che alcuni hanno bisogno di più memoria ma nessuno ha bisogno di meno memoria

• SWAP! – Rimuoviamo in successione alcuni processi finché il thrashing si ferma

Quale dimensione di pagina?

* Pagine ampie, Maggiore rischio di frammentazione interna – In media ogni processo lascia inutilizzata metà del suo ultimo page frame
* Pagine piccole - Maggiore ampiezza della tabella delle pagine

Il valore ottimo può essere definito matematicamente

– σ B dimensione media di un processo

– π B dimensione media di una pagina

– ε B per riga in tabella delle pagine

– Spreco per processo come f(π)=(σ / π) x ε + π / 2

• Parte di tabella delle pagine + frammentazione interna

• Derivata prima è - σ ε / π2 + 1/2

• Ponendo uguale a zero si ha che il minimo di f(π) si ha per π = √ (2 σ ε)

Per σ= 1 MB e ε= 8 B si ha π = 4 KB

Per RAM di ampiezza crescente può convenire un valore di π maggiore – Ma di certo non linearmente

In generale la memoria virtuale non è distinta per dati e istruzioni

– Nella prima metà del ’70 vi sono stati elaboratori importanti (PDP-11) che fornivano invece spazi di indirizzamento distinti- Programmed Data Processor (2 KB cache, 2 MB RAM)

– Aiuta a gestire pagine condivise tra più processi

**Paginazione: realizzazione**

Il S/O compie azioni chiave

– A ogni creazione di processo

• Per determinare l’ampiezza della sua allocazione

• Per creare la tabella delle pagine corrispondente

– A ogni cambio di contesto

• Per caricare la MMU e “pulire” la TLB

– A ogni page fault

• Per analizzare il problema e operare il rimpiazzo

– A ogni terminazione di processo

• Per rilasciarne i page frame

• Per rimuoverne la tabella delle pagine

Per trattare un page fault bisogna capire quale riferimento è fallito – Per poter completare correttamente l’istruzione interrotta

Il Program Counter dice a quale indirizzo il problema si è verificato – Ma non sa distinguere tra istruzione e operando

Capirlo è compito del S/O

– Orrendamente complicato dai molti effetti laterali causati dagli “acceleratori” hardware

– Il S/O deve annullare lo stato erroneo e ripetere da capo l’istruzione fallita

Page fault: l’hw fa trap al kernel e salva il PC sullo stack.

Un programma assembler salva i dati nei registri e poi chiama il sistema operativo.

Il S.O. scopre il page fault e cerca di capire di quale pagina (visionando i registri o recuperando il PC e simulando l’istruzione).

Ottenuto l’indirizzo virtuale causa del page fault, il S.O. verifica che si tratti di indirizzo valido (altrimenti kill del processo) e cerca page frame vuoto o con pagina rimpiazzabile.

Se la pagina da rimpiazzare è dirty, si imposta il suo spostamento su disco (il processo corrente viene sospeso nel frattempo) e il frame viene bloccato.

Quando il page frame è libero, vi copia la pagina richiesta (il processo viene di nuovo sospeso nel frattempo).

All’arrivo dell’interrupt del disco, la page table è aggiornata e il frame è indicato come normale.

Il PC viene reimpostato per puntare all’istruzione causa del page fault.

Il processo causa del page fault è pronto per esecuzione e il S.O. ritorna al programma assembler che lo aveva chiamato.

Il programma assembler ricarica i registri e altre info; poi torna in user space per continuare l’esecuzione.

Un’area del disco può essere riservata per ospitare le pagine temporaneamente rimpiazzate

– Area di swap

Ogni processo ne riceve in dote una frazione

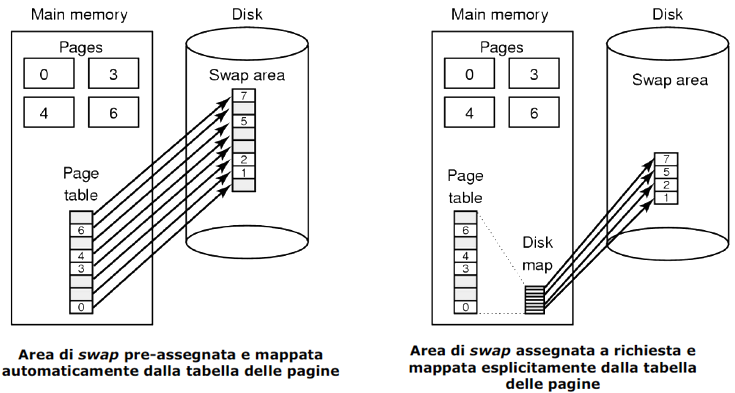
– Che rilascia alla sua terminazione

– I puntatori (base, ampiezza) a questa zona devono essere mantenuti nella tabella delle pagine del processo - Ogni indirizzo virtuale mappa nell’area di swap direttamente rispetto alla sua base

• Idealmente

– L’intera immagine del processo potrebbe andare subito nell’area di swap alla creazione del processo

– Altrimenti potrebbe andare tutta in RAM e spostarsi nell’area di swap quando necessario

Però sappiamo che i processi non hanno dimensione costante

– Allora è meglio che l’area di swap sia frazionata per codice e dati

Se l’area di swap non fosse riservata allora occorrerebbe ricordare in RAM l’indirizzo su disco di ogni pagina rimpiazzata

– Informazione associata alla tabella delle pagine

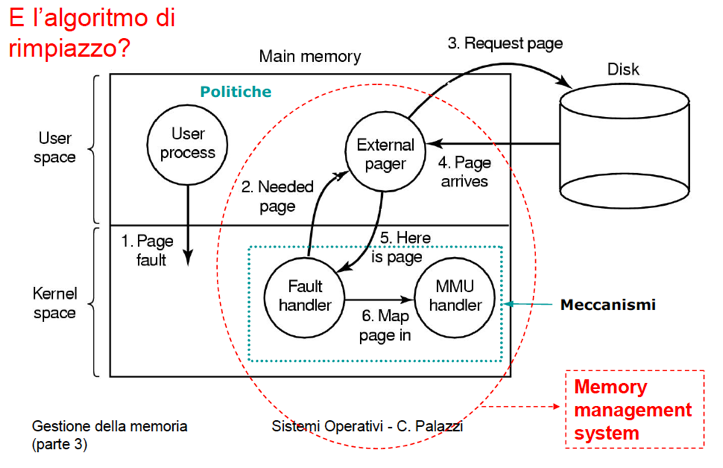
**LINUX**

– Partizione dedicata allo SWAP, con file system apposito

• Consumo una delle possibili partizioni del disco

– Dimensione impostabile dall’utente in fase di installazione

• Almeno la stessa dimensione della RAM se si vuole gestire l’ibernazione (copia di tutto il contenuto della RAM nell’area di SWAP e ricaricamento in fase di riattivazione)

Windows (2000, XP, …)

– Uso di file di swap

– hiberfil.sys (usato per copiare la RAM in caso di ibernazione del sistema)

– pagefile.sys (usato quando la memoria RAM non è sufficiente)

– Se il file viene frammentato, le prestazioni calano

Per separare le politiche dai meccanismi

– Conviene svolgere nel nucleo del S/O solo le azioni più delicate

• Gestione della MMU – Specifica dell’architettura hardware

• Trattamento immediato del page fault – Largamente indipendente dall’hardware

– Demandando il resto della gestione a un processo esterno al nucleo

• Scelta delle pagine e loro trasferimento – Trattamento differito del page fault

**Segmentazione: premesse**

Spazi di indirizzamento completamente indipendenti gli uni dagli altri

– Per dimensione e posizione in RAM

• Entrambe possono variare dinamicamente

Entità logica nota al programmatore e destinata a contenere informazioni coese

– Codice di una procedura

– Dati di inizializzazione di un processo

– Stack di processo

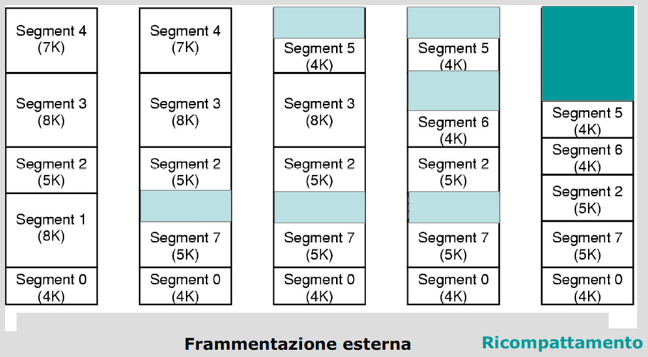
Si presta a schemi di protezione specifica

– Perché il tipo del suo contenuto può essere stabilito a priori

• Ciò che non si può fare con la paginazione

Causa frammentazione esterna

**Segmentazione: realizzazione**



Vista la grande ampiezza potenziale i segmenti sono spesso paginati

Nel caso del Pentium di Intel

– Fino a 16 K segmenti indipendenti

• Di ampiezza massima 4 GB (32 bit)

– Una LDT per processo

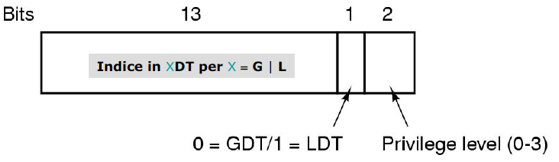
• Local Descriptor Table – Descrive i segmenti del processo

– Una singola GDT per l’intero sistema

• Global Descriptor Table – Descrive i segmenti del S/O

Per accedere a un segmento, un programma Pentium prima carica

selettore di quel segmento in uno dei sei registri di segmento



6 registri di segmento

– Di cui 1 denota il segmento corrente

LDT e GDT contengono 213 = 8 K descrittori di segmento

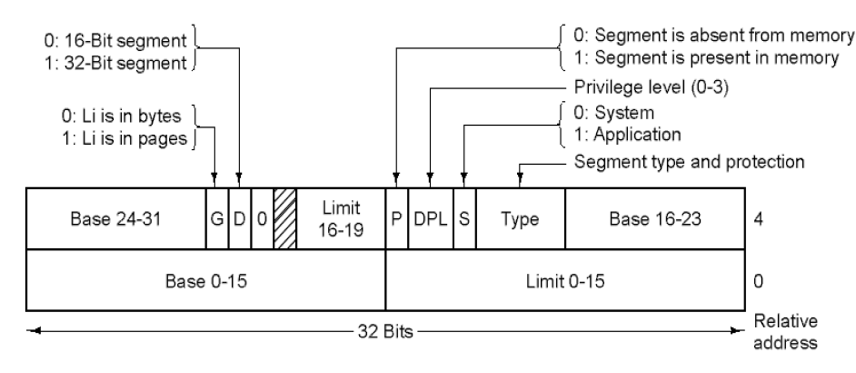
– I descrittori di segmento sono espressi su 8 B

• La base del segmento in RAM è espressa su 32 bit

• Il limite su 20 bit per verificare la legalità dell’offset fornito dal processo

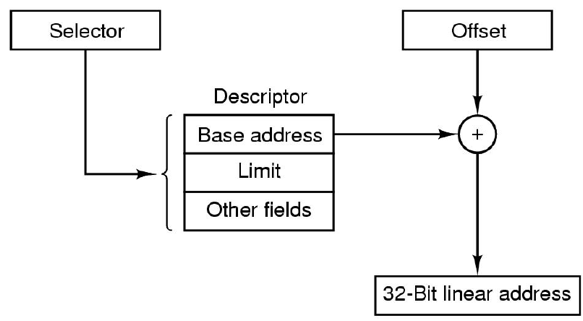
– Consente ampiezza massima a 1 MB (per granularità a B)

– Oppure 1 M pagine da 4 KB ovvero 4 GB (per granularità a pagine)



Descrittore di segmento di Pentium relativo al codice (lievi differenze

con quello relativo ai dati)



L’indirizzo lineare ottenuto da (base di segmento + offset) può essere interpretato come

– Indirizzo fisico se il segmento considerato non è paginato

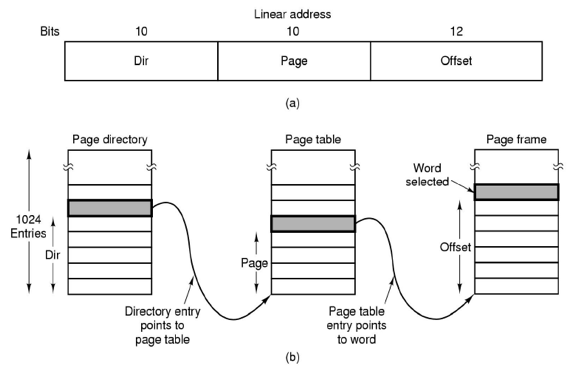
– Indirizzo logico altrimenti

• Nel qual caso il segmento viene visto come una memoria virtuale paginata e l’indirizzo come virtuale in essa

– 10 bit : indice in catalogo di tabelle delle pagine » 210 righe da 32 bit ciascuna (base di tabella denotata)

– 10 bit : indice in tabella delle pagine selezionata » 210 righe da 32 bit ciascuna (base di page frame)

– 12 bit : posizione nella pagina selezionata » Offset in pagina da 4 KB



L’indirizzo lineare mappato sullo spazio virtuale

**Segmentazione: protezione**

