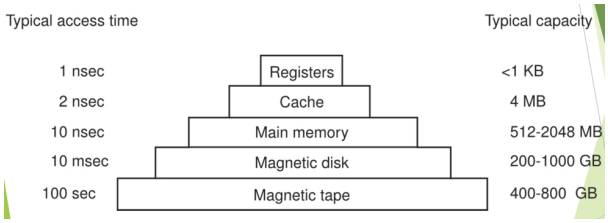
11-05-2023

Gestione della memoria

La gestione della memoria è un aspetto molto delicato visto che il calcolatore lavora con la CPU e la memoria.
La memoria si suddivide in varie parti secondo una determinata **gerarchia**.



> *Più è grande la memoria, più è lenta e viceversa*

Si sfrutta ogni singola memoria in modo gerarchico.
Quando si parlava di processi, in un sistema multiprocessore, tentano di far eseguire i successivi processi usando la stessa area di memoria.

Le **memorie più piccole** contengono i dati che risultano più **RECENTI** e quindi sono ***usati più spesso*** in modo da avere risposte più veloci e questa caratteristica deve essere *sfruttata al massimo*.
La **RAM** (che è ***FINITA***) contiene *tutto quello che deve essere mandato in esecuzione*.
Il ***gestore della memoria*** è quella parte del SO che gestisce tutto ciò che avviene all'interno della memoria considerando varie sfaccettature:

- *Allocazione* della memoria, tenere traccia di cosa (*blocchi di memoria*) è/non è occupato
- *Deallocazione* della memoria
- *Dove posizionare* gli elementi
- *Prelevare i dati* dal disco e caricarli (e *in che modo caricarli*)

Per la gestione di quanto sopra, serve un'***astrazione della memoria a favore dei processi***.

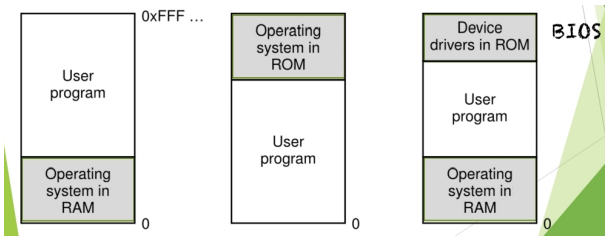
> *Quando si alloca uno spazio di memoria, raramente tutta la memoria destinata ad un programma viene occupta e molto spesso si occupa uno spazio di memoria più piccolo dello spazio di memoria effettivamente allocato per tale programma.*
> *Quindi internamente c'è uno **spazio di memoria sprecato**.*
> *"Allora un blocco di memoria come deve essere rappresentato?"*
> *Si fanno quindi determinate scelte per evitare tali problematiche e ridurre al minimo lo spreco*

> *Può capitare che **fra 2 blocchi di memoria** ci sia un **piccolo spazio di memoria non allocato** e non allocabile da nessun programma perchè **troppo piccolo**. Anche questo può rappresentare uno spreco di memoria.*

I primi PC non avevano un'astrazione della memoria e quindi i programmi operavano direttamente sugli indirizzi fisici, quindi **sulla memoria direttamente** (*se si hanno 2 programmi in memoria,possono usare indirizzi che vanno in conflitto fra loro*), che partivano da 0 fino alla dimensione massima. C'era il limite di ***non poter contenere più di un programma in memoria allo stesso tempo*** perchè in questo caso nasceva il problema del conflitto di indirizzo.

- I programmi potevano prendere inidirizzi che danneggiavano anche il SO

Vi erano diversi modelli (da sx a dx ->A B C):



- (A) Venivano allocati i **programmi e il SO all'inizio della RAM** (**indirizzo 0**, *figura a sx*)
- (B) Si memorizzava **tutto il SO all'interno della RAM** (sistemi *embedded*) e tutto il resto era libero ai programmi dell'utente (*figura centrale*)
- (C) I **driver più importanti nella ROM** mentre il **SO all'inizio della RAM** (MS-DOS, *figura a dx*)
  - Il **BIOS** è quella parte dei drivers presente nella ROM

Il modello A e C hanno un problema: il SO è nella RAM e c'è il rischio di conflitti con i programmi utente possono interferire con il SO (potrebbero anche cancellare file importanti) . Nel modello B c'è più protezione sotto questo punto di vista.
Per questo motivo ***si può eseguire solo un processo alla volta***.

Problemi dei vecchi sistemi:

- Per avere una **gestione più efficiente** i SO si sono evoluti per cercare di **eseguire più programmi contemporaneamente** (*o quasi*)
- Serviva fare uno swap fra memoria->disco e viceversa
- Caricano un programma alla volta in RAM

Mantenere i dati e **più programmi sulla RAM** velocizza le operazioni e quindi si dimiuiscono gli accessi all'hard disk ma ci sono dei problemi:

- ***PROTEZIONE DLELA MEMORIA***: *presenza di più processi nella RAM* e nascono conflitti visto che lavorano su indirizzi fisici (*un processo accede agli indirizzi fisici di un altro processo*)
  - **SOLUZIONE**: lock & key -> prende la memoria e la divide in blocchi. Ogni blocco contiene, in una parte, **i dati** e, dall'altra parte, **un'identificativo** (*un numero*) per quel blocco di memoria dei dati. Si ha anche la ***PSW*** di ogni processo in esecuzione (***CHIAVE DI PROTEZIONE*** di ogni processo).
  - *Quando il SO deve operare, vede se la **chiave memorizzata nel PSW** del processo corrisponde al blocco sul quale vuole operare e in tal caso fa accedere alla memoria ed esegue le operazioni*
  - in questo modo si evita che un processo interferisce su memorie di altri processi
  - Questa *non è la soluzione migliore* perchè, usando anche dello spazio di memoria per la PSW, **si perde memoria disponibile** e i **programmi lavorano direttamente su indirizzi fisici**
- ***RILOCAZIONE DELLA MEMORIA***:
  - ***STATICA***: si allocano/riallocano i programmi in memoria. Si riallocano, in fase di loading del processo, e si assegnano gli indirizzi. Ma in questo caso, per distinguere gli indirizzi fra i programmi, si aggiunge al caricamento del programma del processo un'**operazione in più** in fase di caricamento (controllare i valori identificativi numerici) e ciò comporta un ***RALLENTAMENTO DEL LOADER*** dei processi
  - ***A COMPILE-TIME***: si fa in fase di compilazione in modo da fare un **unico inserimento**. Questo approccio si usa maggiormente nei sistemi `embedded` .

Approccio con lo Spazio degli indirizzi

- Un'altra soluzione per mantenere più processi in memoria allo stesso tempo senza interferenze, si può generare lo ***SPAZIO DEGLI INDIRIZZI*** ed è un'*astrazione della memoria*
- Rappresenta l'insieme degli indirizzi che un processo può usare per indirizzare la memoria di suo interesse

> *Come si assegna lo sapzio degli indirizzi ad ogni processo?*

Tramite una ***rilocazione dinamica*** che fa uso di:

- **registro base**: impostato con l'indirizzo fisico a partire dal quale è memorizzato il programma (indirizzo di partenza)
- **registro limite**: contiene la lunghezza del programma
  La CPU controlla gli accessi alla memoria in base a questi registri.

> *Nei sistemi moderni non si usano questi registri ma si usa la **MMU** (Memory Managment Unit).*

Lo ***SVANTAGGIO*** di questo approccio è il seguente: ogni volta che si deve allocare, bisogna fare una somma del registro base fino al registro limite e quindi si fa anche un confronto periodicamente.
Anche ***se il confronto è veloce***, ***la somma non lo è affatto*** e ciò **appesantisce** la CPU.

L'`Intel 8088`, a differenza di prima (`CDC 6600`), usa **più registri base** ma comunque presenta dei problemi e quindi non *è una soluzione ottimale*.

Sovraccarico della memoria

Nei sistemi moderni il limite è che la **RAM** è comunque **limitata** e **i programmi** diventano sempre **più grandi**.

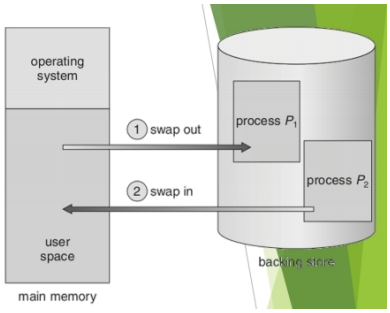
> *Si deve gestire una situazione dove la **RAM NECESSARIA per soddisfare tutti i programmi** è nettamente superiore rispetto alla **CAPACITA' della RAM** stessa*

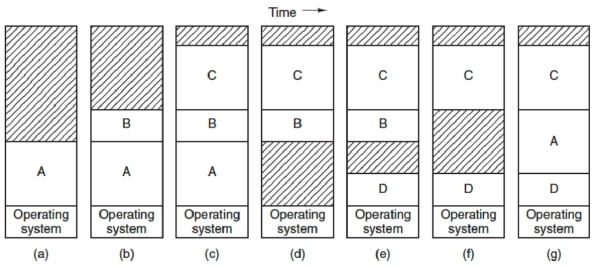
La gestione del sovraccarico si può gestire con 2 approcci: ***Swapping*** e ***Memoria virtuale***

Approccio con Swapping

- Consiste nel prendere ciascun **programma nella sua TOTALITA'** ed eseguirlo per un certo *periodo di tempo* e dopodiché si **swappa** (`pop()`) e si rimette nel disco.
- I programmi non attivi sono mantenuti all'interno del disco
- Lo scambio dei programmi dalla **RAM** al **disco** viene effettuato dallo **scheduler** chiamato ***SWAPPER***

Esempio:



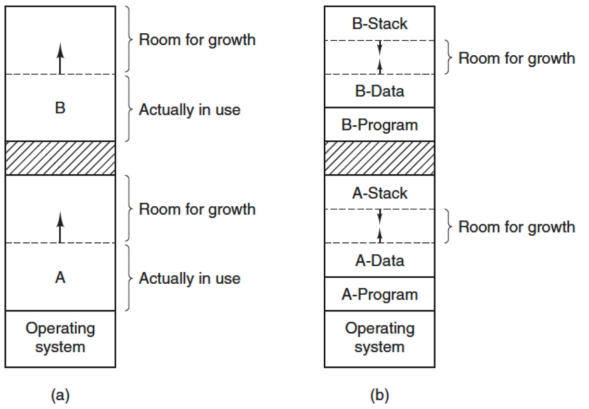


> *c) A viene swappato nel disco -> g) ritorna A e si trova in una nuova posizione rispetto all'inizio (* `a)` *) e quindi i nuovi indirizzi si devono rilocare*
> Se, dopo `g)` dovesse **tornare B**, allora qualche programma deve essere sacrificato in modo creare la memoria libera per B. Il programma da sacrificare, però, non deve avere **richieste I/O pendenti**

- Si nota che nei vari **passi intermedi** (`d)` , `e)` , `f)`) vi sono degli ***spazio vuoti fra blocchi di memoria*** e che ovviamente sono ***SPRECATI***.
- Gli spazi vuoti all'interno della memoria sono chiamati ***FRAMMENTAZIONE ESTERNA*** (avviene all'***esterno rispetto ai programmi allocati***)
- Gli spazi vuoti si devono ***compattare fra loro*** e si parla di ***MEMORY COMPACTATION*** ma questa è un'*operazione lenta* e *costosa* (per questo motivov viene *usata raramente*)
- La ***FRAMMENTAZIONE INTERNA***: alloco uno spazio di programma più grande rispetto a quanto ne richiede effettivamente e quindi ***all'interno dello stesso programma allocato***

Quanta memoria si deve allocare?

> *Se il blocco è da 64 e il programma da 50, i 14 sono sprecati (*frammentazione interna*). Se il programma richiede 68, allora devo allocare 2 blocchi da 64 e quindi viene occupato 1 blocco + 4 e il resto è sprecato*

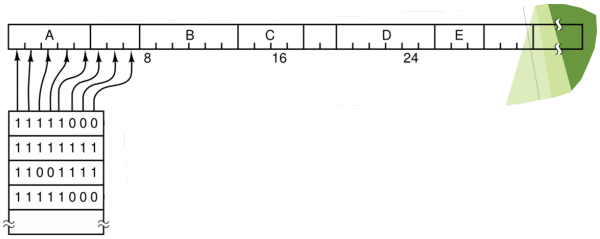


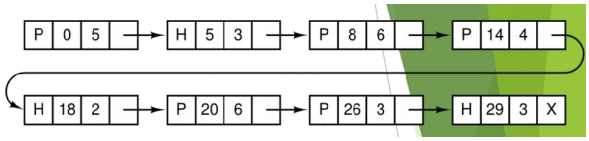
Vi sono diverse strategie e difficilmente si riesce a beccare il numero esatto di memoria necessaria:

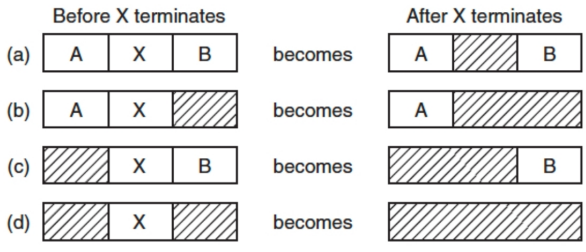
- Alloco blocchi di memoria a ***DIMENSIONE FISSA*** ma ciò rappresenta un problema con i programmi visto le loro *dimensioni* non sono fisse ma *variano spessissimo*.
- Alloco blocchi dimemoria a ***DIMENSIONE DINAMICA*** (`figura a`)dove si alloca una porzione di memoria più una quantità extra all'interno del quale i processi possono **crescere**.
  - Un'altra strategia migliore di questa (`figura b`), basata sulla dimensione dinamica, è quella di distinguere le parti rekative ai dati locali.
  - Si alloca un programma e, nella parte **superiore** extra, si aggiunge nella parte più bassa i dati del processo (che potrebbero crescere dal basso verso l'alto) e gli stack usati dal processo nella parte alte (che cresce dall'alto verso il basso).
  - Si ha quindi **dati** (basso->alto) e **stack** (alto->basso) e quindi le crescite si muovono in direzioni opposte

- Comunque sia la dimensione è limitata e lo spazio si può *esaurire*

*Ma come si **tiene traccia dei blocchi liberi o dei blocchi occpati** e come si identificano?*
Esistono 2 modi con cui il SO tiene traccia dei blocchi liberi/occupati:

- ***BITMAP*** (*matrice di bit*): una volta stabilito che la memoria è divisa in blocchi, per identificare i blocchi liberi/occupati, a ogni blocco si associa un bit che identifica `0` se la locazione è ***LIBERA*** e `1` se la locazione è ***OCCUPATA***:
  

  - *Questo singolo bit quanto deve essere grande?*
  - Se è da 64 posso identificare blocchi più piccoli di 64
  - La **dimensione del bit** incide su vari aspetti: la dimensione della bitmap -> ***Più è piccola la dimensione di ogni bit***, ***più è grande la bitmap***.
    - ***Più è grande la bitmap***, **minore** è la probabilità di generare **frammentazione interna**.
    - ***Più è piccola la bitmap***, più è grande la dimensione del singolo bit e quindi **maggiore** è la probabilità di generare **frammentazione interna**
  - Nell'***ultimo blocco***, se la dimensione non corrisponde, c'è ***per forza frammentazione interna*** e quindi l'obiettivo è ***MINIMIZZARLA*** al fine di non sprecare memoria
  - Lo **svantaggio** è la ***RICERCA DELLO SPAZIO DA ALLOCARE*** in caso di allocazione:
    - Se devo allocare un blocco di k unità, devo cercare nella bitmap una sequenza di k **bit consecutivi a** `0` e questa è un'***operazione lenta*** (in particolar modo quando la memoria è a regime)
  - L'alternativa è usare una lista concatenata
- ***LISTA CONCATENATA***: ogni blocco dice se il blocco è **occupato** o **no**. Ogni blocco contiene ***4 informazioni***:
  

  - Se è il blocco occupato/libero (`P` Processo presente, `H` Hole -> Spazio vuoto)
  - Da dove parte il processo/segmento
  - Quanto è lungo il processo
  - Puntatore al nodo successivo
- Se la ***lista è ordinata per indirizzo***, la ricerca è semplice ma in particolar modo quando si libera una locazione di memoria perchè, in questo caso, si distinugono varie casistiche:
  

    - La gestione è molto più semplice nelle 4 casistiche sopra indicate: si analizza il precedente/successivo del programma X e spessissimo si usa una ***LISTA DOPPIAMENTE CONCATENATA*** e si mantiene sempre ***ordinata per indirizzo***

Vi sono 4 **ALGORITMI DI RICERCA** per allocare blocchi di memoria:

- ***FIRST FIT***: parto dalla testa, appena trovo **il primo blocco libero per soddisfare la richiesta**, lo prendo e lo occupo (molto veloce e molto usato). Il problema: se prendo il primo nodo utile è probabile che ***creo frammentazione interna***
- ***NEXT FIT***: la stessa del precedente ma questa volta parte da dove si era fermato al passo precedente (*quindi non parte più dalla testa*). Le prestazioni sono leggermente inferiori al *FIRST FIT* (*anche questo fa frammentazione interna*)
- ***BEST FIT***: cerca di trovare la **migliore locazione vuota** la cui dimensione è quanto più **vicina a quella richiesta**. Si ha spazio vuoto interno ma minore rispetto al *FIRST FIT*. E' un algoritmo meno efficiente rispetto al *FIRST FIT* perchè deve ***scansionare tutta la lista***
- ***WORST FIT***: Cerca e alloca **lo spazio di memoria *più grande possibile***. Crea grande frammentazione interna. Non è usato proprio per questo motivo perchè crea grande frammentazione interna

Si può ***ancora ottimizzare*** questo approccio separando le liste: una lista doppiamente concatenata per **blocchi vuoti** e un'altra per **blocchi occupati**.
In questo caso, se da un lato ottimizzo la gestione degli algoritmi in caso di allocazione, dall'altro lato sto peggiorando il caso di deallocazione: **le due liste devono INTERAGIRE FRA LORO** in caso di deallocazione e quindi si perde più tempo ma si guadagna nei tempi di ricerca.

Si possono separare le liste e ***ordinarle per dimensione*** piuttosto che per indirizzo. In questo caso il ***BEST FIT diventa l'algoritmo migliore*** perchè trovo subito la locazione utile in testa.

> *Se lo ordino per indirizzo ***FIRST FIT*** è il più veloce (frammentazione alta). Il BEST FIT scorre tutta la lista ma la frammentazione è piccola
> *Se ordino la lista per dimensione BEST FIT è il miglior algoritmo da usare e FIRST FIT fa, di conseguenza, la stessa cosa.*