

Swapping

La multiprogrammazione è limitata dalla dimensione della RAM

Come soluzione adottiamo lo swapping:

Se non c'è spazio e voglio inserire un nuovo programma, un altro programma lo sposto su disco e inserisco quello nuovo per l'esecuzione.

A un certo punto quello su disco lo swapiamo con uno in RAM.

Permettiamo un più elevato parallelismo.

Il sistema diventa più sofisticato ed riduce bene il rischio di swapping (ed riduce un processo non interattivo).

Lo swapping è fatto dallo swapper che è uno scheduler a medio termine.

- basso termine: alta frequenza (scheduler processi)

- medio termine: media frequenza

Bisogna fare attenzione a non spostare programmi attivamente in attesa di I/O.

Anche meno in corso a scrittura / lettura della memoria (swap i bei programmi e avviene la scrittura nel programma sbagliato).

↑
Questi problemi sono ~~legati~~ I/O dipendenti lunghi.

Come allocare?

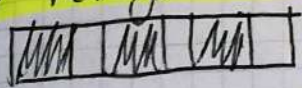
- bin fissa: semplice ma crea spreci di è limitante

- bin dinamica: bin variabile che può crescere.

Se ho due processi adjacenti e uno vuole più spazio allora o mettere P1 su disco e farlo aspettare o mettere P2 su disco o spostare un po' P2 (o i più). Può essere considerato come problema. Altre forme di spaco sono legate alla frammentazione.

La frammentazione può essere:

- interna: spazio legato a spazio (eccellente) fatto per una richiesta (allocazione 100 mezza ma 70 erano necessari)
- esterna: spazio fatto di spazi alle partizioni si vengono a creare buchi nella memoria tra processi



Gran Problema: ho 3 buchi da 10 mezza e devo mettere un processo da 30 mezza.

Lo spazio ci sarebbe ma non posso
Possibile soluzione: memory compaction

Compattazione: buchi in un unico buco. Ha un costo notevole

Gestione dell'allocazione

La memoria la amministriamo per blocchi.
Stiamo facendo una approssimazione per facilitare la gestione. Conseguenza: Spesso

Più è grosso il blocco più è alto statisticamente lo spreco. Lo spreco è al più un blocco di memoria (l'ultimo), al meno 0 e statisticamente $\frac{1}{2}$ di blocco

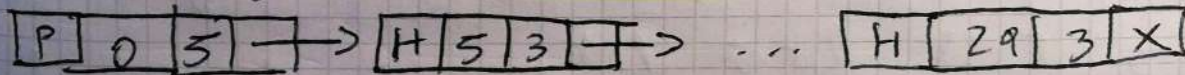
Non facciamo blocchi comunque troppo piccoli per chi lavora usare strutture dati per tenere traccia (se ci sono eccellenti blocchi la struttura è lenta)

1a Struttura: BIT MAP

tiene traccia dello stato di allocazione della memoria. La bit map è di bin line, ma tiene solo conto dell'informazione blocco allocato / NON allocato

2a Struttura: LISTE

Creiamo una lista che contiene i blocchi liberi o ~~liberi~~ buchi



P: processo
H: buco

↓
blocco iniziale
↓
estensione (5, 6, 7)

La lista è ordinata per indirizzo del 4o blocco ed è
bipartitamente linkata quindi la gestione è facile e
efficace.

Il doppio linking permette una più facile coesistenza
ovvero far avere blocchi liberi contigui (tre buchi
vicini diventano un solo buco)

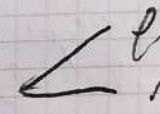
La coesistenza riduce la lista e permette di avere
buchi grandi invece di tanti piccoli buchi

La coesistenza non è indispensabile perché:

- Quando un processo deve morire e deve togliersi un pezzo
dalla lista ci serve facilmente con un puntatore nel
PCB
- il doppio linking la facilita.

Ricerca di una sequenza libera / buco

- first fit: ricerca del 1o blocco sufficientemente
grande.

NB: la lunghezza della lista dipende da  frammentazione
multiprogrammazione

- next fit: applica la ricerca come first fit
però si ricorda dove si è fermato alla precedente
ricerca e riparte da lì

- best fit: scelgo il blocco sufficientemente grande
per soddisfare la richiesta ma che sia il più
piccolo possibile.

È lento dato che dobbiamo scorrere (quasi sempre)
l'intera lista, che spedisce (almeno tanti piccoli
buchi).

- Worst fit: scegliere il blocco peggiore (il più
grande della lista)

Sperimentalmente si è visto che il migliore
algoritmo è il first fit

Ottimizzazione: Creare liste separate
• Ho due liste separate \leftarrow blocchi liberi ordinati per indirizzo
L'efficienza aumenta (rispetto per gli algoritmi di ricerca) ma la complessità diventa difficile

MEMORIA VIRTUALE

L'allocazione continua da parte l'interazione implementano il concetto di memoria virtuale e sfruttano l'allocazione NON continua della memoria fisica per evitare sprechi

Lo spazio di indirizzamento diventa virtuale, ovvero concettualmente i miei pezzi belli pagine sono contigue ma il sistema non devono stare per forza vicini nella memoria fisica. Ho addirittura la possibilità di avere pagine in RAM e pagine in Disco contemporaneamente

Se ho tante pagine piccole la struttura dati che le gestisce è troppo grande, e se ho poche pagine grande la struttura è piccola ma la suddivisione dello spazio in pagine viene meno e c'è frammentazione (int) la pagina è un taglio di bit finché prescelto dal SO e l'HW è progettato per gestire

Solus (e altri pochi SO) usano pagine di bit variabile

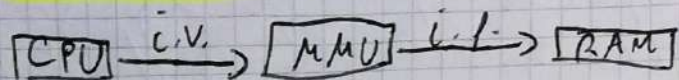
Anche la memoria fisica è suddivisa in pezzi detti frame, le cui dimensioni corrispondono alle dimensioni delle pagine.

Il fatto che pagine contigue nello spazio non lo sono per forza in memoria risolve il problema della frammentazione esterna.

E' implementato in modo implicito la protezione di memoria. Un p1 non ha modo di generare indirizzi di un p2

Paginazione

E' gestita dall' MMU. Si occupa di fare la traduzione indirizzo virtuale - indirizzo fisico



L' MMU dato un i.v. fa la traduzione verso un i.f. che sta o in RAM o in disco (page fault)

Page-fault: richiesta di una page NON in RAM
Nel caso di page-fault poi si esegue un procedimento per spostare la pagina in RAM (2 volte anche facendo swap con una pagina già presente)

- Nell' HW vi è un bit di presenza per sapere se una pagina è presente o meno
- Lo spazio di memoria virtuale è sempre maggiore dello spazio fisico di RAM.

Come avviene la traduzione $\text{i.v.} \rightarrow \text{i.f.}$?

- MMU ha come input un numero che è l'indirizzo di una word (nella virtuale)
- Dato questo i.v. ne indichiamo la pagina che lo contiene

$$\text{i.v. / size} = \text{\# P.V.}$$

offset: distanza word presa e 4a word della pagina

resto che rappresenta offset

Avendo il #P.V. e il bit per sapere se sta in RAM o no (page fault)

Se siamo nel 1o caso

- #frame • ~~size~~ con questa op ottengo l'indirizzo fisico della 4a word della frame
Il #frame supponiamo sia dato da una p

Sommiamo offset e questo risultato e shift ottenuto

Il procedimento reale si basa su questa strategia ma è ottimizzato sfruttando la tabella delle potenze di 2.

Se divido i.v. per $size = 2^x$ o shift a dx i.v. ~~x~~ volte è la stessa cosa ma le prestazioni sono migliori.

Quello che rimane a $1x$ è #P.V., quello che esce a dx è l'offset

i.v.

#P.V.	offset
-------	--------

Ora vorrei moltiplicare per $size$ ma basta lo shift a $1x$ di x volte.

Ora devo sommare l'offset ma se lo faccio

per OR (bit a bit) offset ottengo lo stesso risultato. (per il risultato all'og. precedente)