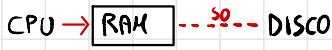
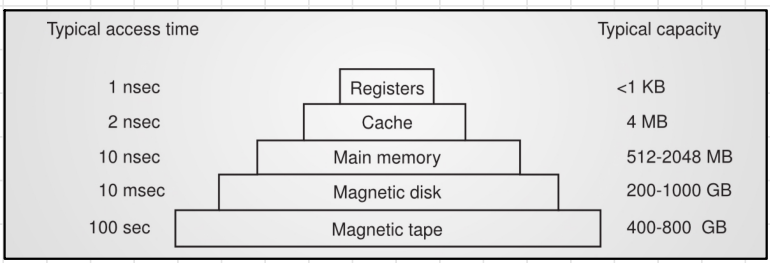
Classifichiamo le memorie in base alle loro caratteristiche, come la velocità di accesso o la posizione rispetto al processore. La CPU legge informazioni dalla memoria centrale (RAM) tramite accessi diretti e quest'ultima ha accesso alla memoria secondaria tramite il sistema operativo.



RAM (Random Access Memory)

La memoria RAM è un tipo di memoria volatile e ad accesso casuale. Ciò significa che i dati memorizzati nella RAM vengono persi quando il computer viene spento o riavviato e che la CPU può accedere direttamente ai dati memorizzati in qualsiasi punto della memoria, senza dover passare attraverso i dati precedentemente memorizzati

La condivisione di risorse relativa alla CPU è definita come multiplexing temporale, poiché i vari processi che utilizzano la CPU si alternano nel tempo, mentre nel caso della memoria si parla di multiplexing spaziale, poiché la stessa memoria viene divisa e condivisa contemporaneamente tra più processi.

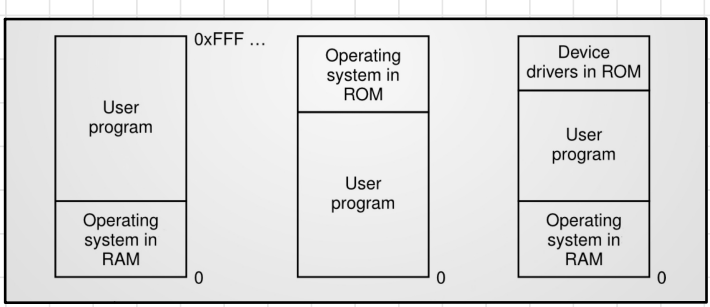


Modello senza astrazioni

I programmi accedono alla memoria attraverso i reali indirizzi fisici senza sfruttare nessuna astrazione, per tanto non viene introdotto alcun overhead dovuto a queste ultime. Il SO deve essere caricato in memoria centrale per poter essere eseguito.

Generalmente nei sistemi embedded, il SO è memorizzato in una componente dedicata chiamata ROM (Read-Only Memory), che è una memoria non volatile e fissa. Durante la fase di avvio del dispositivo, la ROM si occupa di caricare il SO in memoria centrale.

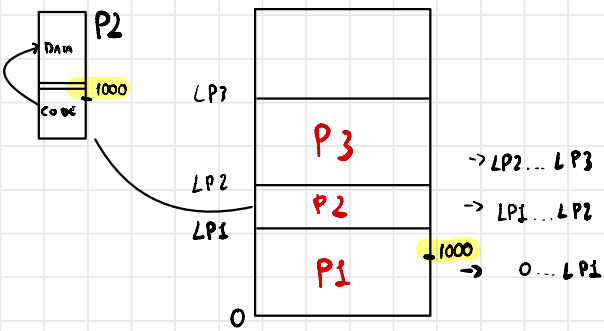
Nei personal computer, invece, il SO è caricato in memoria centrale dalla Basic Input/Output System (BIOS), che è un firmware preinstallato nella scheda madre. Durante la fase di avvio del computer, la BIOS si occupa di inizializzare il sistema e di caricare il SO in RAM. In questo modo, il SO può essere eseguito dalla CPU e gestire le risorse del sistema.



La programmazione multiprogrammata, che consente l'esecuzione simultanea di più processi sulla stessa macchina, richiede un'efficiente gestione della memoria al fine di permettere l'esecuzione dei vari processi (rilocazione e protezione).

Rilocazione a compile-time

Il range di indirizzi fisici su cui andrà a lavorare il processo deve essere già noto a compile-time, in quanto le istruzioni del processo prevedono la conoscenza degli indirizzi accedibili fisicamente per poterli gestire in modo consistente. Questa informazione che deve essere nota a priori in fase di compilazione è una forte limitazione quando si devono eseguire differenti processi senza doverli fare interlacciare tra loro. Uno dei probemi che ci si pone è la coerenza tra la porzione di memoria che ogni processo è tenuto ad utilizzare e gli indirizzi fisici che la memoria mette a dispozione.



In assenza di un'informazione data a priori sulla memoria si ha un disallineamento tra gli indirizzi utilizzati dai processi e quelli fisicamente presenti in memoria. Si necessita di una componente che gestisca il disallineamento andando a tradurre gli indirizzi "logici" del codice del processo in reali indirizzi fisici di memoria, aggiungendo un grado di sfasamento all'indirizzo utilizzato dal processo.

Questo però comporta molti limiti : ad esempio se si necessita di scambiare la posizione in memoria di due processi , allora si scambiano anche gli spazi di indirizzamento, di conseguenza il codice non sarà più compatibile poiché gli indirizzi relativi al primo processo non corrisponderanno più agli indirizzi fisici di memoria occupati dal secondo processo.
Pertanto, se si vuole scambiare gli spazi di indirizzamento di due processi, è necessario ricompilare i codici per intero per assegnare nuovi indirizzi di memoria relativi a questi spazi di indirizzamento scambiati.

Rilocazione statica

Quando viene istanziato un processo a partire da suo codice (loading time), ovvero in fase di caricamento, il loader (componente del SO) fa una scansione del codice del processo per vedere quali sono i riferimenti al codice e ai dati e ne gestisce l'associazione agli indirizzi fisici di memoria. Questa rilocazione si occupa di tradurre gli indirizzi usati dai processi sfruttando un offset rispetto agli indirizzi fisici, aggiungendo un grado di sfasamento agli indirizzi sfruttati dal codice del processo, al pari di quanto avviene con la rilocazione a compile-time. Questa traduzione di indirizzi permette di sfruttare in modo coerente gli indirizzi fisici della memoria ma ha un overhead importante che comporta il rallentamento del loader. Si nota che in questo tipo di rilocazione non si ha un intervento hardware, se non durante la fase di caricamento.

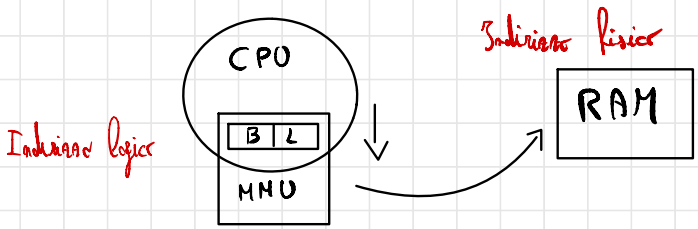
Protezione

Si necessita che nessun processo a causa di un bug riesca a scavalcare il proprio spazio di indirizzamento, riuscendo a danneggiare il codice di altri processi. In altre parole se si verifica un problema in un processo, questo deve rimanere circoscritto al processo stesso senza andare ad intaccare il codice di altri processi, sistema operativo compreso.
Un meccanismo hardware detto lock & key, che consiste nel suddividere la memoria in blocchi a cui vengono associati delle chiavi di protezione, ognuna delle quali corrisponderà a delle chiavi associate ai processi. Quando la CPU genera un indirizzo, lo farà in modo coerente facendo si che la chiave del codice del processo coincida con la chiave della porzione di memoria. Questo necessita di una tabella in grado di prevedere tutti i possibili indirizzi fisici e i relativi accoppiamenti.

Rilocazione dinamica (base-bound)

Una soluzione che riesce a risolvere entrambe le problematiche di rilocazione e protezione è quello della rilocazione dinamica. Una sottocomponente hardware della CPU , detta **MMU**(memory management unit), traduce gli indirizzi logici generati dalla CPU in indirizzi fisici di memoria, permettendo così ai programmi di accedere alla memoria RAM in modo trasparente rispetto alla sua effettiva organizzazione fisica.

Si hanno due registri noti come **indirizzo base** e **indirizzo limite** , appartenenti alla CPU e noti alla componente MMU che sono necessari alla traduzione. Con traduzione intendiamo la manipolazione di base e limite, in particolare essi vengono popolati in modo tale che siano sempre corrispondenti alle locazioni di memoria che delimitano lo spazio di lavoro del processo. Il codice generato negli indirizzi logici viene ricaricato a run-time (dinamico) in porzioni differenti della RAM in modo tale da non fare accesso agli indirizzi al di fuori del range base-limite.



Il context-switch viene gestito riprogrammando l'MMU, ovvero aggiornando la coppia di valori base-limite che contiene e le informazioni aggiornate sono presenti nel PCB del processo stesso.
Quando un processo tenta di accedere a zone di memoria al di fuori del proprio spazio di indirizzamento, il SO invoca una trap che tronca l'esecuzione del processo in quanto non può proseguire con un'esecuzione consistente.
Questo tipo di gestione funziona e garantisce coerenza e protezione solo a seguito di una separazione tra modalità kernel e user.

Swapping (scheduling a medio termine)

Il livello di mutiprogrammazione è seriamente limitato dalla dimensione della memoria centrale : quando si riempie lo spazio disponibile e vengono mandati in esecuzione altri processi, si necessità di scegliere quale processo viene sacrificato (scheduling a medio termine) per supportare l'esecuzione del processo entrante.

Lo swapping è un meccanismo che permette di bloccare (parcheggiare) un intero processo su disco, con lo scopo di liberare la memoria centrale magari a supporto di una richiesta dell'operatore. Il processo non verrà schedulato finchè non viene riportato in memoria centrale. L'operazione è invertibile in modo pulito e avviene riportando il PCB del processo in memoria centrale ripermettendone l'esecuzione.
Si nota che in questo caso si necessita di risistemare i registri base e limite in caso di rilocazione dinamica o di cambiare il delta (offset) nel caso di rilocazione statica. Il problema degli I/O pendenti si verifica quando un processo ha operazioni di input/output (I/O) in corso o in attesa al momento dello swapping. Poiché il processo viene bloccato e spostato su disco, le operazioni di I/O possono rimanere in uno stato indefinito, causando potenziali conflitti o malfunzionamenti nel sistema.

Partizione della memoria

Quando si decide come creare queste suddivisioni della memoria si deve rispettare il **vincolo di contiguità** il quale impone che un blocco di memoria sia unico e contiguo, ovvero che sia posizionato in una sola porzione di memoria fisica continua. Si potrebbe decidere di mantenere una **dimensione fissa** alla partizione di memoria assegnata ai processi ma questo potrebbe allocare partizioni eccessive a processi di piccola entità e viceversa. Una soluzione più flessibile è quella di assegnare una **dimensione dinamica** alle partizioni di memoria in modo da adattarle a quanto richiesto dal processo.

La frammentazione della memoria è uno spreco si verifica quando la memoria disponibile non è contigua e quindi non può essere utilizzata in modo efficiente per memorizzare nuovi dati o programmi.

- **Frammentazione interna :**

La frammentazione interna si verifica quando uno spazio di memoria allocato è maggiore di quello richiesto effettivamente dal processo che lo utilizza, lasciando uno spazio non utilizzato tra la fine dei dati memorizzati e il limite successivo della memoria allocata. Questo spazio non utilizzato viene chiamato frammentazione interna e può essere un problema soprattutto quando la memoria è limitata.

- **Frammentazione esterna :**

La frammentazione esterna si verifica quando ci sono molti piccoli spazi liberi nella memoria, ma nessuno di essi è abbastanza grande per soddisfare una richiesta di memoria di una dimensione specifica. Questo tipo di frammentazione può essere causato da processi che vengono caricati e scaricati dalla memoria, lasciando spazi vuoti.

Compattazione della memoria

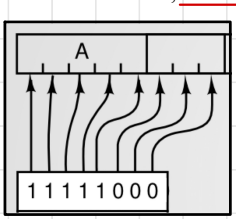
La compattazione della memoria è un'operazione che consiste nell'accorpare i buchi di memoria libera adiacenti, in modo da creare una porzione di memoria contigua e liberare spazio fruibile per l'allocazione di nuovi processi o per l'espansione di processi già presenti in memoria. Questa operazione viene effettuata quando la memoria centrale presenta frammentazione esterna, ovvero quando gli spazi liberi sono frammentati in blocchi non contigui, rendendo difficile allocare un blocco di memoria di dimensione sufficiente per ospitare un processo.

Parlando di strategie per gestire la memoria centrale emergono delle strutture dati di supporto alle soluzioni per la visualizzazione dell'organizzazione della memoria :

Bitmap

La Bitmap è una rappresentazione grafica della memoria, dove ogni bit rappresenta un'unità minima allocabile. In questo modo, per allocare spazio per un processo, è sufficiente ricordare quanti blocchi sono necessari per la partizione del processo e memorizzare l'unità di allocazione di inizio e il numero di unità allocate per il processo. La Bitmap consente di rappresentare in modo efficiente lo stato di allocazione, in quanto l'accesso a un bit richiede solo un'operazione di lettura o scrittura a basso costo.

Tuttavia, la dimensione del blocco può diventare problematica. Se l'unità di allocazione è grande, il costo associato alla bitmap diventa elevato e inoltre si verifica una frammentazione interna, dovuta al fatto che si deve arrotondare la locazione dedicata al processo al minimo allocabile (es. per allocare 2.5 unità si necessita di usarne 3 e avere 0.5 unità di frammentazione interna). La memoria dedicata all'allocazione della bitmap è già allocata staticamente e ha un costo fisso che si sottrae alla CPU. Inoltre, ogni volta che un processo viene caricato o scaricato dalla memoria, la Bitmap deve essere aggiornata per riflettere lo stato attuale della memoria.



Liste dinamiche

Una soluzione che gestisce dinamicamente l'organizzazione dei processi in memoria centrale è una lista il cui generico elemento è o un buco o un blocco dedicato ad un processo. I buchi sono blocchi di memoria che non sono ancora stati assegnati a nessun processo e che quindi possono essere allocati . I blocchi dei processi sono invece dei blocchi di memoria che sono già stati allocati a processi attivi. Normalmente se si viene a liberare la partizione del processo si sostituisce con un buco.

La **coalescenza dei blocchi liberi** è un'algoritmo che va a compattare buchi liberi contigui, in modo da ottenere buchi grandi unici. Attraverso la conversione di un blocco da tipo P (processo) a tipo H (buco) si riesce a mantenere una struttura molto compatta. Le liste sono ordinate per **indirizzo crescente** in quanto l'ordine degli indirizzi crescenti facilita l'implementazione dell'algoritmo di coalescenza dei blocchi liberi, perchè consente di trovare facilmente i blocchi di memoria adiacenti da unire per formare un unico blocco più grande.

Una buona implementazione di questa struttura dati prevede la lista doppiamente concatenata, in quanto questa soluzione velocizza le operazioni sui blocchi.

Algoritmi di ricerca in allocazione :

- **First fit** : si scorre la lista a partire dalla testa (indirizzi più piccoli) alla ricerca e si ferma nel primo nodo che ha dimensioni sufficienti per contenere il blocco del processo. Sostanzialmente di aumenta la dimensione della lista sostituendo un buco con un buco più piccolo.

- **Next fit** : si riparte a scorrere dalla locazione successiva a quella in cui si era interrotto per ricercare un buco con lo stesso criterio del First fit.

- **Best fit** : scorre l'intera lista alla ricerca del blocco migliore, ovvero il blocco che tra tutti quelli in grado di ospitare il blocco del processo ha dimensione minore, ovvero che mi consenta di allocare il processo con il minor spreco possibile. Questo algoritmo è uno dei peggiori in termini di frammentazione in quanto lascia una frammentazione non sufficiente agli altri processi.

- **Worst fit** : cerca il buco più grande possibile che consenta di allocare il processo mantenendo una frammentazione rimanente il più grande possibile per l'allocazione di ulteriori processi .

First e Next fit hanno una complessità computazionale ridotta rispetto a Best e Worst fit che necessitano di scorrere l'intera lista alla ricerca del blocco che si adatta meglio e quindi hanno una complessità lineare.

Liste separate

L'implementazione delle liste separate per blocchi liberi e blocchi occupati è una soluzione migliore per la gestione della memoria centrale. Questa soluzione permette di evitare la ricerca inutilmente all'interno dei blocchi già occupati, ottimizzando così il processo di allocazione della memoria.

Inoltre, **l'ordinamento delle due liste per dimensione dei blocchi crescente** consente di scegliere il blocco di memoria migliore da allocare o rilasciare in base alla dimensione richiesta. Ciò consente di evitare l'uso inefficiente di blocchi di memoria.

Un'altro vantaggio derivante dall'utilizzo delle liste separate, che questo approccio migliora l'efficienza dell'algoritmo di coalescenza dei blocchi liberi, in quanto permette di effettuare operazioni di fusione solo sui blocchi adiacenti, riducendo così il tempo di esecuzione dell'algoritmo.