CPU

Sigla di central processing unit, la componente di un calcolatore (detta anche processore) che carica le istruzioni dei programmi in memoria, le interpreta e manipola i dati di conseguenza.

La CPU preleva le istruzioni dalla memoria, le decodifica e le esegue

Possiede dei registri di controllo che servono per controllarne il funzionamento.

Esempi registri controllo:

* Program counter (PC): Indirizzo prossima istruzione da prelevare.
* Stack pointer (SP): Puntatore al top della pila dei record di attivazione.
* Program status word (PSW): insieme di registri es

Privileged mode (PM): bit di modalità user/kernel (anche le istruzioni privilegiate sono eseguibili)

Le istruzioni privilegiate, ossia le istruzioni che modificano la PSW, sono eseguibili solo in stato kernel.

INTERRUPT

Gli interrupt costituiscono un meccanismo per notificare alla CPU un evento o una condizione avvenuti nel sistema che devono essere trattati dal sistema operativo.

L’obiettivo è:

* Interrompere il normale ciclo di esecuzione della CPU
* Richiedere l’intervento del sistema operativo

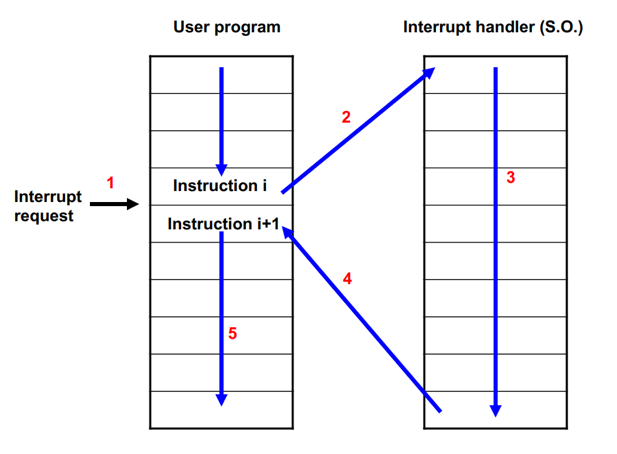
Hardware interrupt:

Si tratta di eventi asincroni rispetto al programma in esecuzione; infatti, il programma non ha modo di prevedere la loro esecuzione.

* I/O interrupt: usati dai dispositivi I/O per notificare eventi alla CPU che richiedono di essere trattati dal SO.
* Timer interrupt: vengono usati per misurare intervalli di tempo e innescare eventi periodici, usati dal clock per notificare il tick.

Fasi dell’interrupt hardware:

1. La CPU mentre sta eseguendo l’istruzione I di un programma P riceve un interrupt request che verrà presa in considerazione prima di prelevare l’istruzione I+1
2. Terminata l’esecuzione dell’istruzione I la CPU sospende P e salta alla procedura di gestione dell’interrupt
3. L’interrupt handler, che è parte del SO, gestisce l’interrupt
4. L’interrupt handler restituisce il controllo a P o P’ (in base all’algoritmo di scheduling)
5. P o P’ riprende l’esecuzione



Se arriva un altro interrupt mentre l’interrupt handler è in esecuzione, accade la stessa cosa di prima

Se arrivano più richieste di interrupt si ha una gerarchia di priorità, quindi vengono ignorate temporaneamente quelle con priorità minore

Alcune vengono ignorate grazie al ruolo svolto dall’interrupt mask (IM)

Per ogni classe di interrupt abbiamo un interrupt vector, memorizzato nella system area, che memorizza:

* L’indirizzo della prima istruzione dell’interrupt handler, da assegnare al PC per avviare l’interrupt handler.

A diagram of a computer

Description automatically generated

Per quanto riguarda il salvataggio dello stato dei registri del programma che subisce l’interrupt abbiamo la **SRIA**, ossia Saved Register Information Area. Essa salva lo stato di:

• Program counter  
• Stack pointer  
• Program Status word

Fare meglio

Program interrupt:

Oltre agli hardware interrupt, esistono due tipi di program interrupt: eccezioni e software interrupt.

• Eccezioni o interrupt interni: sono errori a tempo di esecuzione

Eccezioni aritmetiche (divisione per zero) – Eccezioni di indirizzamento (pagge fault)

Queste situazioni sono causate dal programma.

• Software interrupt detti anche **trap**

Sono causati da un’istruzione apposita: TRAP

Sono usati dai programmi per chiedere esplicitamente l’intervento del S.O

l’istruzione TRAP ha un parametro che specifica il tipo di intervento richiesto. Il parametro può essere passato in 2 modi:

* Come operando della TRAP, se la TRAP prevede operandi
* Ponendolo sullo stack

DISPOSITIVI DI I/O E DMA

Dispositivi di I/O. Distinguiamo tra:

- Controller (adapter): componente elettronica che comunica con la CPU (e le altre unità) via bus

- Dispositivo vero e proprio: componente meccanica che viene gestita dal controller

L’interfaccia tra CPU e controller viene usata dai programmai del S.O che gestiscono i dispositivi driver.

Questa interfaccia prevede:

• registri di controllo, detti anche porte di I/O

• un buffer: serve per memorizzare dati durante le operazioni di I/O.

**Soluzioni per le comunicazioni tra la CPU e le porte di I/O:**

1. porte di I/O gestite con istruzioni macchina ad hoc
2. Memory Mapped I/O

Nella prima opzione, la CPU comunica con le porte I/O attraverso istruzioni specifiche, mentre nella seconda, le porte I/O sono mappate direttamente nella memoria del sistema, consentendo l’accesso tramite letture e scritture di memoria. Questi approcci offrono metodi efficienti per controllare e interagire con i dispositivi di input/output.

**Soluzioni per eseguire l ’ I/O per conto di un programma P:**

1. Programmed I/O (busy waiting)

Il programma P interagisce direttamente con i dispositivi di I/O. "busy waiting" per controllare lo stato del dispositivo in modo continuo fino a quando non è pronto a ricevere o inviare dati. Elevato uso di CPU

1. Interrupt Driven I/O (overhead)

Il dispositivo di I/O genera un interrupt per segnalare alla CPU che è pronto per trasferire dati. La CPU interrompe l'esecuzione del programma P per gestire l'evento di I/O. Introduce un overhead dovuto alla gestione degli interrupt

1. Direct Memory Access (DMA) capace di accedere direttamente alla memoria e di lavorare in parallelo con la CPU.

Un controller DMA può accedere direttamente alla memoria del sistema senza coinvolgere la CPU. La CPU inizializza il trasferimento DMA e può poi eseguire altre operazioni mentre il DMA trasferisce dati tra il dispositivo di I/O e la memoria.(in parallelo)

PROCESSI

**3.1 Il memory layout dei programmi**

L’esecuzione di un programma necessita di tre aree di memoria (regioni)

* Area di testo: contiene il codice eseguibile del programma.
* Area dati: contiene le variabili globali del programma.
* Area di stack: contiene i record di attivazione (frame) delle procedure già chiamate e non ancora terminate. Per le strutture dinamiche delle procedure si usa una parte dell’area dati detta **heap**, cui le variabili fanno riferimento

A diagram of a diagram

Description automatically generated

**3.2 il concetto di processo**

Un processo è un’esecuzione di un programma.

Possiamo avere **più processi relativi al medesimo** **programma**, in quanto possiamo avere più esecuzioni, anche in contemporanea, del programma.

* Il programma è un’entità passiva che non esegue nessuna azione di per sé.
* L’esecuzione del programma, chiamata processo, concretizza le azioni specificate nel programma.
* Il S.O. schedula processi, non programmi.

**3.3 parallelismo e concorrenza**

* Due eventi sono paralleli se occorrono nello stesso momento
* La concorrenza è l’illusione del parallelismo.

Assumiamo che l’hardware offra una sola CPU e un DMA controller: Assegnando la CPU a turno ai vari processi (interleaving) il S.O. realizza l’esecuzione concorrente di tali processi, in quanto in ogni istante al più un solo processo può usare la CPU.

La velocità di avanzamento di un processo non è uniforme nel tempo, perché non sempre dispone della CPU.

Se avessimo più di una CPU distingueremmo due casi:

* Multiprocessing: più processi possono eseguire su una macchina dotata di più CPU.
* Distributive Processing: più processi possono eseguire su più macchine distribuite ed indipendenti.

In questi casi il parallelismo può essere reale.

**3.4 stati di un processo**

Lo stato è un indicatore dell’attività che sta svolgendo il processo.

* **Running**: il processo è in esecuzione, la CPU sta eseguendo il programma associato al processo.

* **Ready**: il processo non è in esecuzione, ma se ottenesse la CPU potrebbe eseguire.
* **Waiting**: il processo non è in esecuzione e non sarebbe in grado di eseguire se ottenesse la CPU. Esso acquisirà la capacità di eseguire quando si verificherà un evento che il processo sta attendendo.

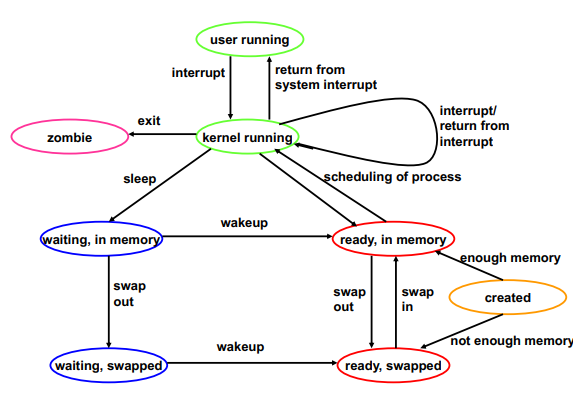
Esempio: processo che sta aspettando che un’operazione di I/O finisca.

A diagram of a process

Description automatically generated

* Caso **ready - running**: lo scheduler seleziona il processo per l’esecuzione tramite una politica di scheduling.
* Caso **running - ready**: Il processo perde il processore, perché un processo con priorità maggiore diventa ready. In un sistema con time-sharing può anche verificarsi la scadenza del timeslice.
* Caso **running - waiting**: Il processo viene messo in stato di waiting poiché si trova impossibilitato ad eseguire in attesa di un evento sbloccante.
* Caso **waiting - ready**: Si verifica l’evento sbloccante che il processo attendeva.

**3.4.1 Stati Unix system V**



Lo stato running è diviso in due parti:

* User running
* Kernel running

Lo stato ready e waiting sono stati divisi in due parti a loro volta:

* In memory
* Swap in: va verso RAM (porta processi ready in RAM), swap out: verso HD

Successivamente abbiamo lo stato zombie (quando un processo è terminato e viene solamente tenuto traccia della sua esistenza) e created.

**3.5 process control block (kernel area)**

Il S.O. tiene traccia di tutti i processi presenti nel sistema, avvalendosi di apposite strutture dati.

Il S.O. mantiene una process table, con una voce, Process Control Block (PCB), per ogni processo.

Voci presenti in ogni PCB:

* identificativo del processo (univoco)
* stato del processo (running, waiting, ready)
* stato della CPU (tiene traccia dei registri)
* parametri da usare per lo scheduling (area di testo, dati, stack)
* puntatore per accedere all’area di testo/dati/stack
* puntatore per accedere a dispositivi aperti
* un PCB pointer (per creare strutture dati dinamiche)

**3.6 implementazione dei processi: il context-switch**

Assegnare la CPU a un altro processo richiede di eseguire un salvataggio dello stato del processo attuale e un ripristino dello stato di un diverso processo. Questa operazione è conosciuta come “context switch”.

il contesto di un processo è costituito da:

* register context: il contenuto dei registri della CPU;
* user-level context: zona di RAM che il processo riesce a manipolare
* system-level context (inaccessibile dal programma user): PCB, PCB pointer

Il S.O. effettua tre operazioni fondamentali sui processi:

* **Context save**: quanto il processo running va in ready o waiting, il S.O. salva nel PCB tutti i dati necessari a far ripartire il processo in futuro.
* **Scheduling**: tenendo conto della politica di scheduling, il S.O. sceglie un processo tra quelli in ready per l’esecuzione.
* **Dispatching**: dopo che un processo viene schedulato, il S.O. deve ripristinarne il contesto sfruttando i dati salvati nel PCB al momento del context save.

Si parla di context switch quando il S.O. effettua il context save per un processo P, schedula un processo P’ diverso da P e ne effettua il dispatching

**3.7 creazione e terminazione di processi**

Per la creazione dei processi il S.O. mette a disposizione una system call: La **fork**()

**3.7.1 Fork**

La fork restituisce 0 al figlio ed il PID del figlio al padre.

* Dopo la fork, il padre ed il figlio sono due processi distinti ma quasi uguali: entrambi hanno il proprio PCB (quello del figlio è creato dalla fork), i PID sono diversi, la getpid da risultati diversi.
* i registri generali hanno il medesimo valore, eccetto il registro usato per il risultato della system call
* i registri PC, PSW, SP hanno i medesimi valori

**3.7.2 Exec**  
La system call exec serve per “cambiarsi il programma”.

* La execl è una delle funzioni di libreria involucro della exec.
* Il primo parametro è il path dell’eseguibile che si chiede di eseguire, il secondo è il nome dell’eseguibile e il terzo sono gli argomenti passati al programma eseguibile

Tramite la exec:

* Il processo ottiene nuove aree testo, dati e stack
* I registri PSW, PC, SP assumono valori nuovi
* I registri generali vengono “azzerati”
* I file vengono chiusi ed i dispositivi rilasciati

**La system call wait** è una richiesta al S.O. di mettersi in stato di WAITING: evento atteso è la terminazione di un processo figlio

Quando un processo termina, esegue la system call exit e va in stato ZOMBIE.

**3.8 Tipi di processo:**

* Processi utente: sono associati al terminale dell’user.
* Processi daemon: non sono associati a nessun utente ed eseguono funzioni vitali per il sistema (user)
* Processi kernel: sono daemon che eseguono in modalità kernel.

**3.8.5 terminazione di processi**

* system call exit
* interrupt handler delle eccezioni può forzare la terminazione del processo.
* Mediante un altro processo killer

**3.9 Sistem call in UNIX**

Oltre al PCB, UNIX assegna ad ogni processo una user area, dove il PCB ha un pointer.

La user area serve anche per memorizzare i parametri, il risultato e l’eventuale codice di errore delle system call.

THREADS

Un **task** è uno dei compiti che un’applicazione deve portare avanti in modo simultaneo. Con la programmazione concorrente si guadagna in semplicità ed efficienza.

Un **thread** è un’esecuzione di un programma che usa le risorse di un processo.

Un processo può avere più thread. Possibile implementazione:

* Condividono aree testo e dati e risorse logiche e fisiche.
* Ogni thread ha il proprio identificatore (Thread IDentifier (TID)), CPU state, stack e stato.

Per ogni thread è necessario avere un **Thread Control Block** che contiene: (Thread IDentifier, TID), CPU state, stack e stato (ready, running, waiting).

Il **thread switching** tra due thread dello stesso processo è più efficiente perché:

* non vanno aggiornati i dati della MMU relativi alle aree testo e dati;
* la cache della memoria e del disco richiedono meno aggiornamenti;

due processi possono scambiarsi informazioni mediante invio e ricezione di messaggi, implementabili con system call.

**4.1 Thread implementati nello spazio kernel e user:**

Kernel:

1. il S.O. gestisce i thread;
2. i TCB sono strutture dati gestite dal S.O.;
3. il S.O. offre system call per creare/terminare thread,
4. lo scheduler del S.O. deve schedulare thread anziché processi;
5. il programmatore maneggia i thread chiamando le funzioni di libreria wrapper di system call.

User:

1. il S.O. gestisce i thread, ma solo processi,
2. TCB sono strutture dati gestite dalla libreria di procedure;
3. esistono procedure per creare/terminare thread, per far cambiare loro il programma,
4. lo scheduler del S.O. schedula processi, una delle procedure della libreria è lo scheduler delle thread;
5. il programmatore maneggia i thread usando le procedure della libreria.

SINCRONIZZAZIONE DEI PROCESSI

Su un sistema uniprocessore, programmare un’applicazione con un insieme di processi e thread concorrenti

Tutti gli n processi eseguono lo stesso programma per prenotare un posto su un volo:

if (next\_seat <= max) {

booked = next\_seat;

next\_seat++;

}

else{

printf("sorry, flight sold out")

}

Dato che l’if verrà eseguito da più processi che condividono la variabile next\_seat possiamo indicare quattro tipi di interazione di processi:

1. **data sharing**: i processi condividono i dati della memoria condivisa e alcuni file (caso next\_seat).
2. **control** **synchronization**: un’azione ai di un processo Pi è abilitata solo dopo che un altro processo Pj ha svolto un’azione aj .  
   Ho i flussi di esecuzione dei due processi, questa azione richiede che l’altro processo abbia già fatto qualcos’altro. Questo perché sono dipendenti tra loro.
3. **message** **passing**: un processo Pi invia un messaggio ad un processo Pj che lo riceve.
4. **signals**: un processo Pi invia un segnale (un messaggio privo di contenuto) ad un altro processo Pj per segnalare una situazione particolare.  
   Ad esempio con la system call kill un processo invia un segnale di terminazione ad un altro processo.

Ri = dati letti dal processo

Wi = dati modificati dal processo

Due processi concorrenti Pi e Pj sono processi interagenti se vale almeno una delle seguenti due proprietà:

* Ri e Wj hanno intersezione non vuota,
* Rj e Wi hanno intersezione non vuota.

Due processi concorrenti Pi e Pj sono processi indipendenti se non sono interagenti.

Se due processi/thread concorrenti Pi e Pj sono indipendenti:

• competono per le risorse rallentandosi a vicenda,

• i loro comportamenti non dipendono dalla loro velocità relativa e **sono** riproducibili.

Se due processi/thread Pi e Pj sono interagenti:

• competono per le risorse rallentandosi a vicenda,

• i loro comportamenti dipendono dalla loro velocità relativa e **non sono** riproducibili.

**5.1 RACE** **CONDITIONS**

Le race condition hanno due conseguenze:

• il comportamento dei processi/thread non è corretto;

• i dati su cui ha luogo la race condition diventano inconsistenti, cioè assumono uno stato non previsto.

Prevenire la race condition:

mutua esclusione = quando un processo accede ad un dato condiviso d, nessun altro processo deve poter accedere concorrentemente a d.

**5.2 SEZIONI CRITICHE**

una sezione critica (critical section - CS) per un dato condiviso d è una porzione di codice che viene eseguita non concorrentemente con sé stessa o con altre sezioni critiche per d

Proprietà richieste alle implementazioni delle CS:

* **correttezza**: le CS per un dato d non possono essere eseguite concorrentemente;
* **progress**: se nessun processo sta eseguendo una CS per d, e alcuni processi manifestano la volontà di eseguire CS per d, allora uno di essi deve poter eseguire la propria CS per d;
* **bounded wait**: dopo che un processo Pi manifesta la volontà di accedere ad una CS per d, il numero di accessi alle CS per d da parte di un qualsiasi altro processo Pj che precedono l'accesso di Pi deve essere <= di un dato intero k

le ultime due prevengono la starvation(attesa infinita da parte dei processi)

**5.3 Implementare le CS:**

tentativi errati:

1. disabilitare gli interrupt: corretta solo su sistemi uniprocessore e non garantisce la proprietà progresso.
2. uso di variabili lock condivise: non è corretta, perché la race condition può aver luogo sulla variabile lock, più processi potrebbero trovare lock = 0(porta aperta).
3. uso di variabili turno condivise: Questa soluzione è corretta ma non soddisfa la proprietà del progresso

un'operazione indivisibile su un dato condiviso d è un'operazione che è con certezza eseguita in modo non concorrente rispetto ad altre operazioni su d.

tentativi corretti:

1. Istruzione **Test and Set**, controlla se una locazione in memoria ha valore 0, imposta questa locazione a 1. In sostanza accede ad una locazione sia in scrittura che in lettura.
2. Istruzione **swap**: istruzione indivisibile che scambia il contenuto di due locazioni di memoria.

**5.4 APPROCCIO ALGORITMICO ALLE SEZIONI CRITICHE**

L'approccio algoritmico prevede di implementare le CS senza usare istruzioni hardware ad hoc, system call o costrutti del linguaggio ad hoc

Approccio algoritmico alle CS idee funzionanti:

- Uso delle variabili turno, algoritmo di Dekker. Soluzione vincente, ma valida solo per 2 processi.

- Uso delle variabili turno, algoritmo di Peterson. Soluzione vincente, ma valida solo per 2 processi.

**Semafori**

A screenshot of a computer

Description automatically generated

**6.1 uso dei semafori nelle situazioni critiche**

Supponiamo che **sem** si un semaforo inizializzato a 1

A computer code with black and green text

Description automatically generated

1. Quando un processo fa la wait, il semaforo andrà a 0.  
   Il primo processo che fa la wait mette il semaforo a 0, ed esegue la sezione critica. Una volta finita la sezione critica viene eseguita la signal e il tutto ricomincia.
2. Una situazione che può capitare `e quella di perdere il processore quando si `e nella sezione critica. Quando il processo che ha ottenuto il processore esegue la wait, trova il semaforo a 0: dunque rimane in waiting. Quindi abbiamo la certezza che nessun processo può entrare la sezione critica, la correttezza `e soddisfatta.

**GESTIONE DELLA MEMORIA**

Obiettivi del S.O.:

1. Allocare più processi in memoria, per avere parallelismo .

2. Proteggere la memoria dei processi, impedendo ai processi di accedere alle aree degli altri processi.

## Traduzione, linking, loading

• Source program

• Object module: programma in linguaggio macchina con riferimenti a funzioni di librerie.

• Binary program: programma eseguibile.

• Binary program in memoria: programma eseguibile in memoria.

rilocazione statica da parte del loader: il loader modifica gli indirizzi del programma prima che il programma venga eseguito.

il linking dinamico, il linker statico sostituisce un riferimento con la logica per effettuare il calcolo dell'indirizzo durante l'esecuzione del programma.

**7.2 Modello di allocazione della memoria**

Durante l'esecuzione del programma, vengono allocati due tipi di dati:

* le variabili il cui scope è associato a blocchi, funzioni e procedure.
* i dati creati dinamicamente con i costrutti appositi.

**heap allocator/deallocator** sono le routine che si occupano di allocare/deallocare memoria nello heap.

Es. free area di 50, voglio allocare un’area di memoria di 10 e una di 20, dopo le allocazioni avrò 20 di free e le due aree allocate. Se faccio dealloca si libera quello che specifico

Modello di allocazione della memoria usato dal S.O.:

* il codice ed i dati statici sono allocati in un'area di dimensione statica,
* l'area di heap e lo stack condividono un'area di dimensione statica, ma le due aree hanno dimensione variabile e "crescono" in direzione opposta.

A diagram of a computer

Description automatically generated

**7.3 Gerarchia di memoria**

i sistemi di computazione offrono una gerarchia di unità di memoria con velocità diverse: le unità più veloci e costose sono quelle di minor capacità.

A diagram of a computer hardware system

Description automatically generated

* La CPU accede alla gerarchia di memoria tramite la MMU (Memory Management Unit), che traduce gli indirizzi logici in indirizzi fisici.
* Il byte che si trova all'indirizzo fisico della main memory generato dalla MMU viene in realtà cercato nella cache. Se non viene trovato, allora è necessario trasferirlo dalla main memory alla cache.
* Se il sistema offre la memoria virtuale, una parte delle aree di memoria del processo non si trovano nella main memory ma sul disco. Gli indirizzi fisici generati dalla MMU sono virtuali.

**7.4 riuso della memoria**

L'allocazione della memoria è un problema che possiamo vedere a due livelli:

- il S.O. deve allocare memoria per i processi;

- il RTS deve allocare memoria per i dati PCD.

In entrambi i casi è necessario riutilizzare la memoria liberata. Per farlo, è possibile tener traccia della memoria libera in una free list.

Se avessimo necessità di allocare un’area di k byte e avessimo più zpne di quella dimensione potremmo utilizzare 3 tecniche per scegliere l’area:

• first fit: si cerca la prima area libera di dimensione >=k. Se l'area ha dimensione d, si allocano k byte e l'area rimanente di d-k byte rimane nella free list; (c’è frammentazione)

A diagram of a building with a few windows

Description automatically generated

• best fit: si cerca l'area libera più piccola tra quelle di dimensione >=k. Se l'area ha dimensione d, si allocano k byte e l'area rimanente di d-k byte rimane nella free list; (è poco efficiente e dopo un po' c’è frammentazione)

A white rectangular object with black text

Description automatically generated

A diagram of a building

Description automatically generated

• next fit: si adotta la first fit partendo, però, dal punto in cui era stata effettuata l'ultima allocazione.(compromesso)

A white rectangular object with black text

Description automatically generated

A diagram of a diagram

Description automatically generated

**Frammentazione**= aree di memoria non utilizzabili in un sistema di computazione

Tecniche per prevenire la frammentazione:

* **boundary tag** = all'inizio ed alla fine di ogni area c'è un tag, le aree libere sono collegate da una catena di pointer quando un'area occupata viene liberata, se è possibile la si unisce ad un'area libera confinante
* **memory compaction** = le aree libere sono collegate da una catena di pointer a determinati intervalli di tempo tutte le aree libere vengono unite in un'unica area libera.

**7.5 Allocazione della memoria**

* **allocazione contigua** della memoria quando ogni processo è allocato in una singola area di memoria contigua.

Allocare in memoria un processo che richiede k byte è possibile solo se è disponibile un'area di memoria dimensione>=k.

Può essere integrata con lo swapping.

* **allocazione non contigua** della memoria quando ogni processo può essere allocato in più aree di memoria non adiacenti.

Ogni porzione del processo che viene allocata in un'area contigua è detta **componente**

Allocare in memoria un processo che richiede k byte **non** necessita di una **singola** area di memoria di dimensione >=k

Il problema della frammentazione è risolto, oppure limitato

2 tecniche per l’allocazione non contigua:

**Paginazione**: Questa tecnica consiste nel dividere la memoria fisica in piccoli blocchi di dimensioni uniformi chiamati pagine, e la memoria logica in blocchi corrispondenti chiamati frame. Quando un programma cerca di accedere a una porzione di memoria, il sistema operativo mappa gli indirizzi di memoria logica alle pagine di memoria fisica. La paginazione aiuta a ridurre il problema della frammentazione esterna, poiché permette al sistema di allocare spazio per i processi in modo più flessibile.

**Segmentazione**: Questa tecnica divide la memoria logica in segmenti logici che rappresentano unità logiche di dati o codice del programma. I segmenti possono avere dimensioni variabili e rappresentare parti distinte di un programma come il codice, i dati globali, lo stack, ecc. La segmentazione è utile per organizzare in modo logico le varie parti di un programma, facilitando la gestione della memoria per processi più complessi. Tuttavia, può portare al problema della frammentazione interna, in cui spazi non utilizzati all'interno dei segmenti possono rendere inefficiente l'utilizzo della memoria.

**Segmentazione + paginazione** combina gli aspetti della paginazione e della segmentazione per ottimizzare l'utilizzo della memoria e semplificare la gestione dei processi. La memoria virtuale di un processo è organizzata sia in segmenti che in pagine. All'interno di ogni segmento, la memoria è divisa in pagine di dimensioni uniformi.

Quando un processo richiede l'accesso a una porzione di memoria, il sistema operativo utilizza una tabella di segmentazione per mappare gli indirizzi logici dei segmenti agli indirizzi fisici delle pagine. Questa tabella contiene le informazioni necessarie per individuare le pagine fisiche corrispondenti a ciascun segmento logico.

**7.7 memoria virtuale**

Si parla di m.v. quando il S.O. crea l'illusione che il sistema abbia più memoria di quella effettiva.

L'obiettivo è di rendere i processi indipendenti dalla capacità di memoria del sistema.

L'implementazione si basa sull'uso del disco.

**Demand paging:**

caso 1:

-la memoria è divisa in page frame ed i processi sono costituiti da pagine

-l'intero spazio logico dei processi è memorizzato su un paging device,

-l'area del paging device allocata per un processo viene detta swap space del processo

-quando un processo inizia l'esecuzione, viene allocato solo un page frame per la pagina che contiene la prima istruzione

Caso 2:

-quando il processo tenta di accedere ad una pagina cui non è assegnato nessun page frame, la pagina viene copiata dallo swap space in un page frame libero;

-se servono page frame, il S.O. può liberare i page frame associati alle pagine dei processi, in tal caso le immagini di tali pagine sullo swap space vanno aggiornate.

Concetti demand paging:

• page fault: eccezione che viene sollevata dalla MMU quando un processo tenta di accedere ad una pagina che non è associata a nessun page frame;

• page-in: in seguito al page fault, il S.O. copia la pagina dallo swap space in un page frame libero.

• page-out: quando il S.O. libera un page frame, se la pagina associata è stata modificata dopo l'ultimo page-in, deve essere copiata sullo swap space;

• page replacement

Traduzione con uso del TLB:

-prevede due cicli di memoria per ogni accesso alla memoria

-Per risparmiare uno dei due cicli, la MMU può usare un Translation Look-aside Buffer (TLB),

la MMU genera l'indirizzo fisico a partire dall'indirizzo logico nel modo seguente:

1. usando la entry del TLB dedicata alla pagina, se esiste;

2. sfruttando la entry relativa alla pagina sulla page table,

3. generando un page fault se i casi 1) e 2) non sono applicabili

la MMU accede alla page table del processo running; in memoria sono presenti le page table di tutti i processi; per implementare facilmente questo aspetto l'architettura può offrire il Page Table Address Register (PTAR), contenente l'indirizzo della page table del processo running;

Protezione della memoria:

• se un processo tenta di accedere ad un indirizzo al di fuori del proprio spazio logico, viene generata un memory protection exception. il controllo può essere fatto facilmente se l'architettura offre un Page Table Size Register (PTSR) che memorizza il numero dell'ultima pagina del processo running.

• il campo prot della page table codifica le protezioni in lettura/ scrittura delle pagine se l'architettura fornisce un adeguato supporto, se il tipo di accesso non è consentito, la MMU genera una memory protection exception.