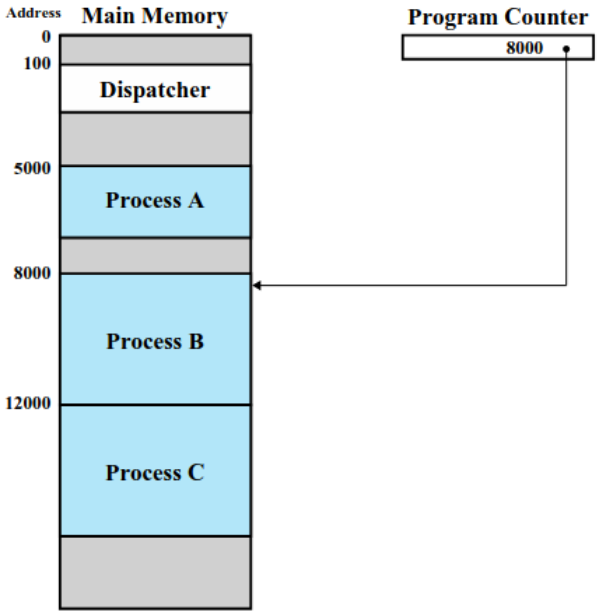
Lezione 3 (Processes)

Gestione dei processi:

- Inizialmente implementato totalmente *batch*
- Successivamente viene implementata la multiprogrammazione ed il time-sharing e da quel punto nacque il concetto di *processo*
- Un processo rappresenta un programma in esecuzione caratterizzato da sequenze di istruzioni, stato della CPU, variabili del programma e 'return addresses' correlati alle funzioni.

Per Von-Newmann ogni processo deve esistere nella memoria principale.



Il SO legge gli eseguibili e ne costruisce le strutture dati in memoria principale
Si usa un diagramma di stato per descrivere le fasi di esecuzione di un process:

- **New**: il processo viene creato e le strutture dati vengono inizializzate
- **Running**: nei sistemi "uniprocessore", solo un processo per volta può essere eseguito
- **Ready**: il processo è **pronto** ma la CPU è momentaneamente in uso
- **Blocked**: il processo è in attesa di un **evento**
- **Exit**: i risultati vengono rilasciati e le strutture dati vengono aggiornate

Il processo viene creato in Memoria Principale, se il processo viene effettivamente creato passa in stato di **Pronto**, ci sarà poi un ciclo di **Dispatch** o di **Timeout**(nel caso si dovesse eseguire un altro processo). Lo stato di **Bloccato** sta aspettando l'input di un certo evento e si sblocca una volta che l'evento avviene. Dopo l'evento lo stato torna in **pronto**.

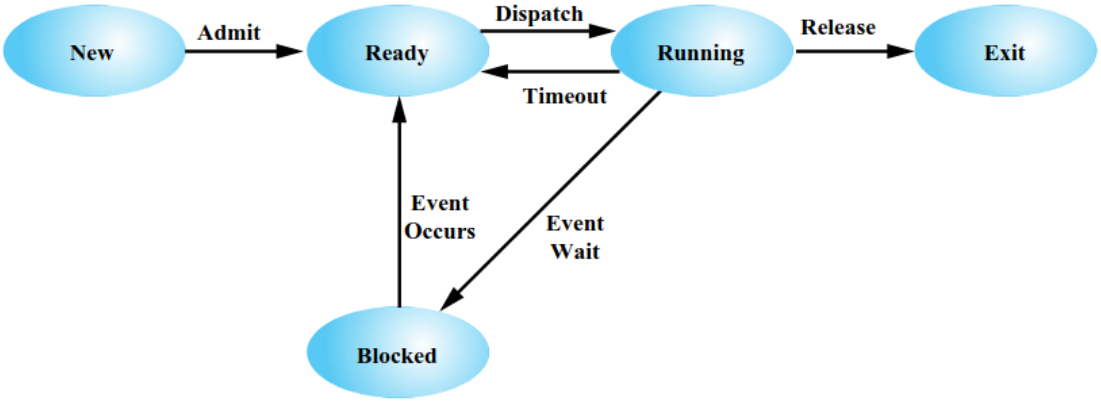
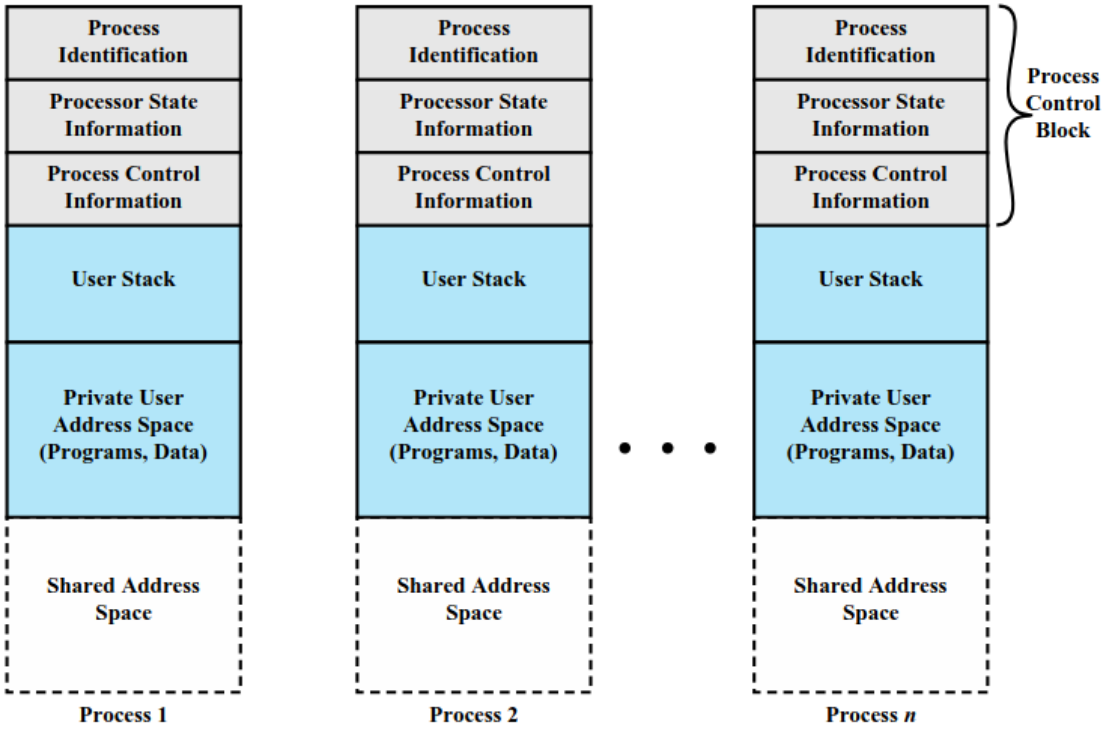


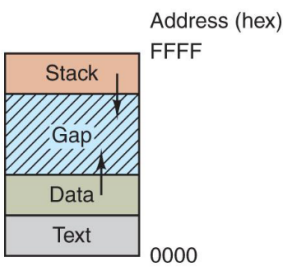
Immagine del Processo

Ogni sistema operativo ha il suo modo specifico di visualizzare le strutture dei processi. Per esempio in *linux* viene visualizzato nel formato **ELF** (Executable and Linkable Format), **BF** nel caso di *Windows*.

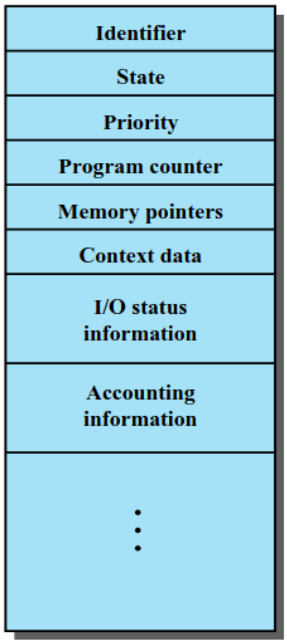


Three main segments:

- the **text** segment the sequenze of the binary instructions of the program
- the **data** segment contains all initialized data
- the **gap** segment contains the **heap** where unitialized or dynamically allocated variables are stored



- the **stack** contains the **return addresses** for function calls



La PCB contiene le informazioni necessarie dal SO per identificare i processi e controllarne l'esecuzione

Creazione dei processi

Un processo viene creato da un'altro processo in esecuzione, il nuovo processo viene chiamato **processo figlio** ed il processo chiamante viene chiamato **processo genitore**. Ogni processo viene identificato da un numero (detto PID)
Il processo genitore può condividere le risorse con il processo figlio mediante locazioni di memoria, files aperti, canali di comunicazione,...
Dopo che il figlio viene creato, il processo genitore continua la sua esecuzione finché il SO non cambia al figlio, il genitore dovrà successivamente *aspettare* fino alla completa esecuzione del processo figlio

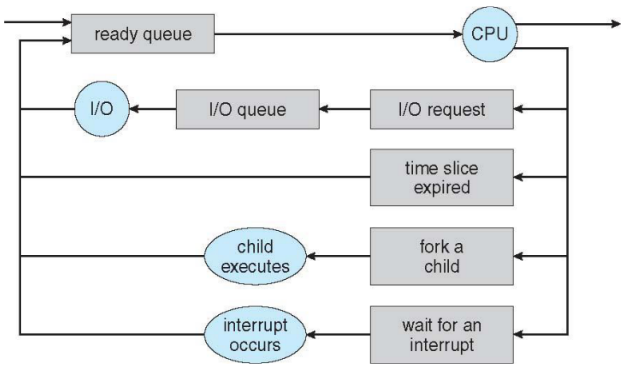
Tutti i programmi terminano con una chiamata ad *exit*()

Lezione 5

Process Scheduling

- **Multiprogramming**: Massimizza l'utilizzo della CPU
- **Time-sharing**: La CPU viene condivisa attraverso diversi processi ed utenti e l'obiettivo è minimizzare il tempo di risposta per ogni processo.
- **Scheduler**: è una delle componenti principali del SO. Ogni volta che un processore è in stato di *idle*, lo scheduler seleziona uno dei processi in ready queue

- **Dispatcher**: Seleziona la task successiva, nel modo più veloce ed efficiente possibile, dalla ready queue



InterProcess Communication (IPC)

Un gruppo di system calls che implementa meccanismi di comunicazione e sincronizzazione dei processi.
Due tecniche principali:

- uso di *shared memory locations*, dove il kernel viene chiamato per creare l'area di memoria, dopodiché i processi possono comunicare tra loro senza dover chiamare il kernel ogni volta
- message passing (il kernel viene chiamato per la "consegna")

Producer

Il buffer viene calcolato in maniera "circolare". Quando gli indicatori *in* e *out* sono sovrapposti, significa che il buffer è vuoto, contrariamente se il buffer è pieno, (?)hanno una determinata distanza(?)
La sicnronizzazione consiste nel fermare il produttore se il buffer è pieno e fermare il consumatore e il buffer è vuoto.

Threads

Implementati in risposta del bisogno di avere una maggiore capacità di calcolo. Permettono di eseguire multiple tasks per lo stesso fine, ergo esegue le stesse attività in maniera *concorrente*. Prima dei threads questo si otteneva per mezzo dell'utilizzo di processi figli. Di conseguenza un thread viene utilizato per ricreare il concetto di processo figlio senza effettivamente doverne creare uno, permettendo di scrivere un unico programma con diversi punti di esecuzione che gestiscono il SO, agendo nella stessa area di memoria.

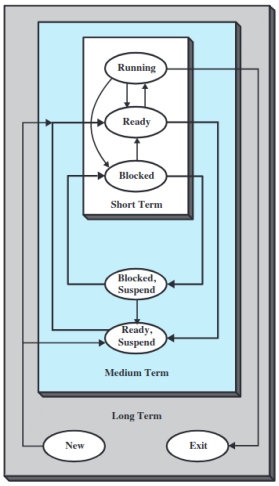
Sono implementati mediante:

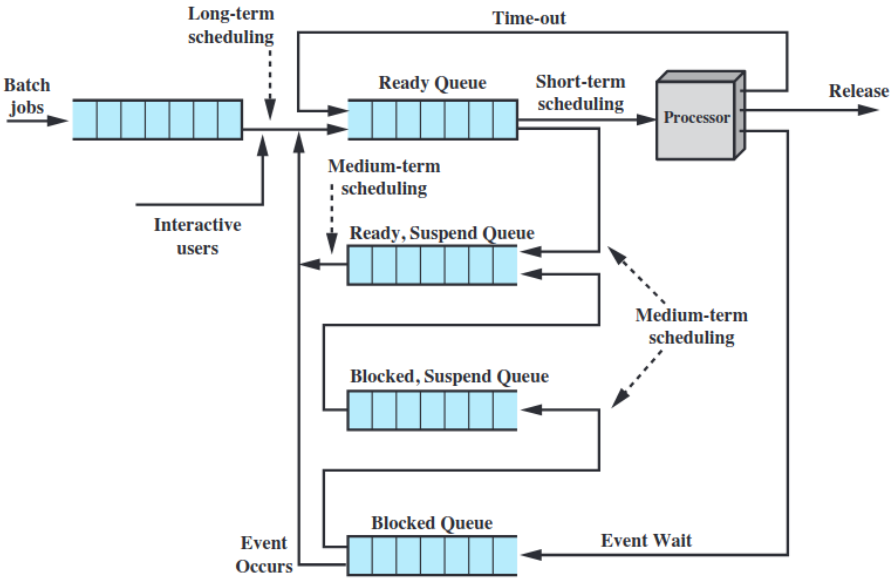
- Librerie

- Supporto Hardware
- Architetture Multi-Core

Lezione 6 (Processor Scheduling)

I sistemi operativi permettono di mandare sul processore altri programmi mentre i programmi già presenti sono in stato di attesa. Questo permette che l'esecuzione di più processi avvenga nel modo più fluido ed efficiente possibile.





Short term Scheduling

- La multiprogrammazione permette di massimizzare l'utilizzo della CPU
- **CPU-I/O Burst Cycle**: L'esecuzione di un processo consiste in un ciclo di esecuzione in CPU e di attesa di I/O
- Il CPU burst viene seguito da un *I/O Burst*

La distribuzione della CPU Burst è di principale interesse

- **CPU Bound**
- **I/O Bound**

Lo Short Time Scheduling si occupa delle operazioni di Ready e di Run.
I processi di Run sono interrotti sono il caso di un'operazione di Exit, di Block o quando interviene il SO. Il movimento di un programma da Run a Ready viene appunto preso in carico dal SO.
L'obiettivo è di allocare tempo al processore in modo da ottimizzare alcuni aspetti del comportamento del sistema. Ci sono due criteri per valutare lo Scheduling:

- **User-Oriented Criteria**: in relazione al comportamento del sistema come percepito dall'utente o dal processo (es. tempo di risposta)
- **System-Oriented Criteria**: Focus su l'effettivo ed efficiente utilizzo del processore, generalmente di minore importanza sui sistemi a singolo utente.

Le prestazioni si possono misurare in maniera quantitativa o qualitativa.

- **Max CPU utilization**: Utilizzo della CPU
- **Max throughput**: Numero di processi che completano la loro esecuzione ogni unità di tempo
- **Min Turnaround Time**: Quantità di tempo per eseguire un certo processo
- **Min Waiting Time**: Quantità di tempo che un processo ha passato attendendo in ready queue
- **Min Response Time**: Quantità di tempo richiessto da quando una richiesta viene inviata fino a quanto la prima risposta viene prodotta

Vengono utilizzati degli **algoritmi** per poter misurare queste quantità:

- **First-Come First-Served (FCFS)**
  - Detto anche First-In First-Out o Scrict Queuing Scheme, è la tecnica di scheduling più semplice (data l'implementazione facile e la sua velocità)

  :

// ADD FCFS Example Slide (Esercizio) //

- **Shortest Process Next (SPN)**
  - Il processo con il tempo previsto più corto viene selezionato come successivo
- **Shortest Remaining Time (SRT)**
  - Il processo in run può essere svuotato dal nuovo processo che si sta unendo alla ready-queue se la sua CPU-Burst è più piccola della CPU-Burst del processo in run.

// ADD SRT Slide //

Un *priority number* (integer) viene associato con ogni processo, viene calcolato dal SO e impostato dall'utente. In Linux più un processo diventa anziano, più il suo valore di priorità viene incrementato (il commando "nice" in linux permette di impostare la priorità di un processo). Di conseguenza diventa conveniente implementare uno scheduling basato sulla priorità:

- **Priority Scheduling**

- **Round Robin (RR)**: Scheduling circolare, ogni processo ottiene una piccola unità di tempo di CPU (time quantum q), generalmente 10-100 millisecondi. Dopo che questa unità di tempo scade, il processo viene svuotato e viene messo alla fine della ready-queue. Il successivo processo in queue viene eseguito.
  Se ci sono n processi nella ready-queue e *quantum time* q, nessun processo di conseguenza dovrà aspettare più di $q(n-1)$ unità di tempo.

// ADD RR example Slide //

Multilevel Queue Scheduling

Il SO sceglie l'algoritmo di scheduling più appropriato in base a ciascuna queue.

Multiple-Processor Scheduling

Il primo processore libero viene occupato

Lab 3 Linux

Ogni processo in Linux è generato da un processo **padre**, ogni processo, di conseguenza avrà un certo PPID (parent process identifier).
Si può vedere ogni processo attivo nella macchina usando il comando "ps -el"
Il SO ha bisogno di un syscall per creare un nuovo processo, si usa fork() per questo. fork() ha dei valori di ritorno:

- 0, al processo figlio
- PID del Figlio, al processo padre
- -1, nel caso in cui la syscall fork fallisca

Il fork() crea un'esatta copia del processo, ritornando un valore di 0 al figlio, il PID al processo padre.

Lezione 7

// FCFS Algorithm image exercise //

// SPN Image ex. //

Shortest Process Next (SPN): significa che la durata di ogni processo viene *stimata*. L'ordine dei processi varia in base alla loro durata (in quanto il focus dell'algoritmo è di ridurre l'attesa dei processi più corti).

// SRT Ex. //

Shortest Remaining Time (SRT): è un algoritmo pre-emptive. Quando c'é una coda di processi lo scheduler guarda se il processo nella coda richiede meno tempo di quello in running e se ha un valore di priorità maggiore(nel caso in cui l'algoritmo lo prendesse in considerazione, nel caso dell'esempio, la priorità viene ignorata).

// RR ex //

Round Robin (RR) [q = 4]: ogni processo viene eseguito ogni q unità di tempo (q = 4 in questo caso) e viene poi rimesso in coda.

Real-Time Operating Systems
Devono essere

- Deterministici
- Responsivi
- Utilizzabili dall'utente
- Affidabili
- Essere a prova di errori

Hanno algoritmi di scheduling come:

- Round-Robin (...)
- A (...)
- Priority-Driven Pre...
- (...)

Lezione 8

Classi di RTOS

- Static table-driven approaches
- Static priority-driven preemptive approaches
- Dynamic planning-based approaches
- Dynamicc best effort approaches (Più frequenti negli RTOS commerciali)

Informazioni usate per lo Scheduling

- Ready time
- Starting Deadline
- Completion Deadline
- (...)
- Resource requirements
- Priority
- Subtask Scheduler

Unbounded Priority Inversion (Inversione di Priorità)

(...)

Concorrenza

I processi possono overlappare ed intrecciarsi, e la loro velocità relativa di esecuzione non può essere predetta nei sistemi uniprocessore.
Si ha un rapporto Producer/Consumer, dove le risorse vengono prodotte per poi essere consumate da qualcun'altro.

```
// PRODUCER
while (true) {
        \* produce an item in next_product */
        while (count == DIM_BUFFER)
                /* do nothing */
        buffer[in] = next_product;
        in = (in + 1) % DIM_BUFFER;
        count++;
}

// CONSUMER
while (true) {
        while (count == 0)
                /* do nothing */;
        next_consumed = buffer[out];
        out = (out + 1) % DIM_BUFFER;
        count--;
        /* consume an item in next_consumed */
}
```

Se i processi di Producer/Consumer si overlappano, risulterà un errore.

Critical Section

Ogni protocollo che indirizza la sezione critica deve soddisfare il progresso, la mutua esclusione e i limiti d'attesa.
Si può arginare il problema dei process che si sovrascrivono attraverso degli algoritmi appositi.
Di seguito, è descritto **l'Algoritmo di Peterson** per un Processo P_i:

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = false;
                /* remainder section */
} while (true);
```

Due processi P_i e P_j, dove $i = 1 - j$
Variabili condivise:
int turn;
boolean flag[2];

Lezione 9

Semafori

Ogni semaforo funziona come waiting queue per più processi

```
wait(semaphore *S) {
  S -> value++;
  if ( S -> value <= 0 ) {
        // Code...
  }
}
```

-BACI - Ben Ari Concurrent Interpreter

L'implementazione dei semafori ha una serie di problemi standard:

- **Deadlock**: (o stallo) condizione dove nessun processo può avanzare.

```
P_0                     P_1
wait(S);                wait(Q);
wait(Q);                wait(S);
...                             ...
signal(S);              signal(Q);
signal(Q);              signal(S);
```

- **Sincronizzazione**:
  - *Bounded buffer Produttore/Consumatore*: sia il produttore che il consumatore scrivono nello stesso Buffer, creando problemi di memoria

Una soluzione può essere riservare il buffer al produttore finché non è pieno. Se non ci sono celle disponibili, il consumatore deve aspettare.

- *Readers/Writers problem*: un'area di memoria viene *condivisa* tra più processi, alcuni leggono soltanto (readers), altri invece scrivono (writers).
  Le condizioni che vanno soddisfatte sono dunque:
  * Tutti i readers possono leggere simultaneamente un file
  * Solo un writer alla volta può scrivere nel file
  * Se tutti i writer scrivono nel file, nessun reader può leggerlo
- *Dining Philosophers problem*: Cinque filosofi a cena, spendono il loro tempo alternando l'atto di mangiare e di pensare. Condividono tra loro una ciotola di spaghetti e delle forchette. Ogni filosofo alla fine si ritrova con una forchetta a sinistra ed una sulla destra.
  Il fulcro del problema è che *non esiste* una soluzione ottimale.

  Per evitare un deadlock, si possono attuare delle restrizioni:
  * Solo uno dei cinque filosofi può richiedere le forchette
  * Le forchette devono essere prese in paio e non una alla volta
  * Ogni filosofo che occupa una posizione *dispari* prende prima la forchetta a sinistra e dopo quella a destra. Ogni filosofo che occupa un posto *pari* invece prende prima la destra e poi la sinistra

Lezione 10

Monitor

E' un costrutto di sincronizzazione che impedisce ai threads di accedere simultaneamente ad uno stesso oggetto condiviso.
(...)

append(), producer
take(), consumer (...)

// Structure of a monitor with condition variables image //

Mutua esclusione, i processi in coda sono bloccati.

Concurrency: Deadlock and Starvation

Per capire il Deadlock, si può considerare ogni processo come un'entità che può richiedere o mantenere delle risorse. Dato un set di processi, il set è in deadlock se il meccanismo smette di muoversi.

Le condizioni per un deadlock sono

- Mutua Esclusione
- Hold-and-Wait
- No Pre-emption
- Circular Wait

Sono tutte condizioni sufficienti affinché si verifichi un deadlock.
Non esiste una soluzione generale a questo problema, tuttavia esistono delle strategie che si possono applicare a tutti i tipi di deadlock (stallo). Ci sono tre approcci comuni:

- Prevenzione dello stallo: disabilita una delle tree condizioni necessarie per lo stallo
- Evitare lo stallo: previene l'accesso ad una delle risorse
- Rilevazione dello stallo: (...)

Altri due approcci per la prevenzione di stalli:

- Resource Allocation Denial: conosciuto anche come **Banker's Algorithm**, cerca di predirre lo stato del processo per portarlo ad uno stato di sicurezza
- Deadlock Detention: algoritmi che rilevano lo stallo

Il recupero dallo stallo si può ottenere fermando tutti i processi in stallo, in base alla priorità, l'importanza, etc... del processo. Recovery (...).

Lezione 11. Pipes

Lezione 12

I programmi devono essere portati dal disco nella memoria e messi in un processo per poter essere eseguiti. La gestione della memoria ci serve per azione di rilocazione, protezione (fare in modo che i programmi non interferiscano l'un l'altro), apetti di condivisione dei dati. Un processo già in esecuzione non ha bisogno di essere rilocato, a meno che più programmi vengano avviati, in tal senso allora la memoria necessita di rilocare i processi. Ci sono diverse tecniche attuate per la rilocazione.
La protezione dei processi si attua avendo nel processore una sorta di 'tabella' nel quale viene memorizzato l'inizio e la fine (in termini di indirizzi di memoria) dei vari processi in run.
Il concetto di condivisione invece è complementare a quello di protezione, ovvero può capitare che alcuni programmi abbiano bisogno di una certa risorsa (es. librerie DLL), di conseguenza allora una volta messa in memoria la risorsa verrà condivisa a tutti i processi che ne hanno bisogno.
I programmi vengono scritti in moduli, ed anche la memoria principale tende ad avere la stessa organizzazione logica, ovvero è divisa in **segmenti**, ciscuno corrispondente ad i moduli dei programmi.
Generalmente il programmatore non può gestire la memoria, poiché non sa quanto spazio dovrà occupare un determinato programma, inoltre la memoria necessaria per un programma ed i suoi dati potrebbe non essere sufficiente.(?) L'overlaying è una tecnica che permette di segmentare un programma e riassemblarlo in base alle necessità (?)
Il calcolo degli indirizzi può avvenire in 3 stages(?):

- Compile time: se l'area di memoria si conosce a priori, il codice assoluto può essere generato
- (...)
- (...)

Il SO gestisce degli indirizzo logici che sono confinati da un indirizzo fisico separato, gli indirizzi logici sono generati dalla CPU (indirizzi virtuali), gli indirizzi fisici sono invece visti dall'unità di memoria.
Gli indirizzi logici e fisici sono identici in compile-time e load-time (?)
Gli indirizzi logici(virtuali) e fisici differiscono in execution-time address binding scheme (??)

Swapping

Tecnica (usata molto negli anni '90, implementata tutt'oggi in maniera simile negli smartphones), e' simile allo stato di bloccato, ed un programma viene messo in 'stasi' e poi rimesso in esecuzione quando riportato in primo piano (similmente allo swap delle applicazioni in esecuzione degli smartphone). Non tiene in programma l'.exe, ma tiene in memoria lo stato del processo all'ultimo istante.

Memory Partitioning

Allocazione Contigua: ovvero all'avvio il SO viene caricato sempre nella stessa locazione nella memoria principale, dopodiché tutti i processi successivi vengono caricati in successione, il problema è che se termina un processo, potrebbe lasciare buchi in memoria. Questa implementazione è perfetta dal punto di vista dell'esecuzione dei programmi. Il problema è appunto, la differenza di dimensioni dei programmi, che lasciano buchi di memoria alla fine dell'esecuzione. Quindi il buco di memoria lasciato da un programma potrebbe non essere sufficientemente grande per lasciare spazio ad un'altro programma. Qui nasce il problema della **Dynamic Storage-Allocation**. Come soddisfare una richiesta di dimensione n da una lista di buchi liberi?

- First-fit: Alloca il primo buco che è grande abbastanza
- Best-fit: Alloca il buco più piccolo, che però è grande abbastanza (richiede però di stilare una lista in base alla grandezza dei buchi)
- Worst-fit: Alloca il buco più grande possibile (deve comunque scorrere l'intera lista)

Segmentation

Il concetto di segmentazione rende il SO più complicato, però al contempo più efficiente. La segmentazione è la divisione di ogni programma in segmenti, dove un segmento è un'unità logica come:

- programma principale

- procedura
- funzione
- metodo
- oggetti
- variabili locali, variabili globali
- blocchi comuni
- stack
- tavola dei simboli (symbol table)
- arrays

Perciò il SO non vede il programma come un monolite, ma vede le varie segmentazioni. In memoria perciò i segmenti non saranno allocati necessariamente in maniera contigua, ma saranno a loro volta allocati in parti diverse. Tuttavia il SO dovrà accessoriarsi di una tabella dei segmenti per ogni processo, dove ogni tavola ha una **base** ed un **limite**.

// Example of Logical-to-Physical Address Translation //

Paging

Un altro meccanismo accoppiato a quello della segmentazione è quello della paginazione. La tecnica della paginazione permette di gestire la memoria principale logicamente suddivisa in pagine, il problema è che la dimensione delle pagine è statica, e la suddivisione in pagine potrebbe tagliare a metà anche delle istruzioni.
Partizioniamo la memoria in chunks di uguale dimensione (di grandezza relativamente piccola), e dividiamo anche il processo in chunks di stessa dimensione.