1. Multiprogramming: più job vengono caricati in memoria, che viene quindi partizionata. Quando un job inizia un’operazione di I/O, la CPU non rimane inattiva (**idle**), ma viene assegnata ad un altro job. È necessaria una politica di **scheduling** per scegliere il job da eseguire tra quelli eseguibili (caricati in memoria che non stanno eseguendo I/O. la gestione della memoria fa si che ogni job deve accedere ai propri dati e nessun job deve riuscire ad accedere ai dati degli altri. La CPU ha i registri **L**ower **B**ound **R**egister (**LBR**) ed **U**pper **B**ound **R**egister (**UBR**); gli indirizzi non compresi tra LBR e UBR non sono validi causano un’interruzione. La multiprogrammazione permette ad ogni job di poter usare i dispositivi ma nessun job deve poter interferire sull’uso dei dispositivi da parte degli altri job. Due modalità principali:
2. **Partitionig**: ad ogni job vengono assegnati i dispositivi staticamente;
3. **Pooling**: ad ogni job vengono assegnati i dispositivi dinamicamente.

L’hardware deve consentire alla CPU ed ai dispositivi di I/O di lavorare in parallelo. Ciò si realizza sfruttando il **DMA** (Direct Memory Access). Quando un dispositivo ha terminato di eseguire un’operazione per conto di un job, il job deve essere inserito nell’elenco dei job schedulabili: ciò si realizza grazie alla tecnica dell’**interrupt**. Il rapporto tra il numero di programmi eseguiti ed il tempo si chiama **throughput**. Con la multiprogrammazione vengono introdotti due concetti:

1. **Program priority**: ad ogni programma è assegnata una priorità, che viene considerata in fase di scheduling;
2. **Preemption**: in alcuni casi la CPU viene tolta forzatamente ad un programma in esecuzione.

L’introduzione degli hard disk consente lo **spooling**: **S**imulaneous **P**heripheral **O**peration **O**n **L**ine. Sottrarre la CPU ad un job ed assegnarla ad un altro (**context switch**) è un’operazione costosa in termini di tempo. Durante il context switch la CPU non lavora per nessun job, quindi sotto un certo punto di vista non effettua lavoro utile. Si parla di **overhead**.

1. La CPU preleva le istruzioni dalla memoria, le decodifica e le esegue. La CPU si avvale di **registri generali**, che costituiscono il primo livello di gerarchia di memoria e che memorizzano variabili e risultati temporanei e di **registri di controllo** che servono per controllarne il funzionamento. Uno dei registri di controllo è il **Program Status Word** (**PSW**) che contiene il **Memory Protection Information** (**MPI**) che contiene informazioni sulla porzione di memoria accessibile.
2. Gli **interrupt** costituiscono un meccanismo per notificare alla CPU un evento o una condizione avvenuti nel sistema. L’obbiettivo è duplice:
3. Interrompere il normale ciclo di esecuzione della CPU;
4. Richiedere l’intervento del S.O.

Gli interrupt possono essere inviati o dal clock o dal controller di un dispositivo (**hardware interrupt**). Es. la CPU, mentre sta eseguendo l’istruzione **i** del programma **P**, riceve un interrupt request su una linea del bus, che verrà presa in considerazione prima di prelevare l’istruzione **i+1**; terminata l’esecuzione dell’istruzione **i**, la CPU sospende l’esecuzione di **P**, cioè non preleva l’istruzione **i+1**, e salta alla procedura di gestione dell’interrupt, detta **interrupt handler**; l’interrupt handler gestisce l’interrupt e restituisce il controllo a **P**; **P** riprende l’esecuzione prelevando l’istruzione **i+1**, come se nulla fosse accaduto. Gli interrupt vengono organizzati in **classi di priorità**, gli interrupt della medesima classe e delle classi con minor priorità vengono ignorati temporaneamente.

1. Una **system call** è una richiesta al S.O. effettuata da un programma
2. Un **processo** è un’esecuzione di un programma. Possiamo avere più processi relativi al medesimo programma. **Le entità che vengono schedulate dal S.O. sono i processi, non i programmi**. Due eventi sono paralleli se occorrono nello stesso momento. La concorrenza è l’illusione del parallelismo.
3. Un’applicazione può avere più compiti, detti comunemente **task**, da portare avanti contemporaneamente. Un **thread** è un’esecuzione di un programma che usa le risorse di un processo.
4. Due processi concorrenti **Pi** e **Pj** sono **processi interagenti** se vale almeno una delle seguenti due proprietà:
5. **Ri** e **Wj** hanno intersezione non vuota;
6. **Rj** e **Wi** hanno intersezione non vuota;

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

1. Una **sezione critica** per un dato condiviso **d** è una porzione di codice che viene certamente eseguita non concorrentemente con se stessa o con altre sezioni critiche di **d**. 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**.
2. L’istruzione **Test and Set** (**TS**) esegue due azioni:
3. Controlla se una locazione di memoria ha valore 0, ponendo il risultato nel bit **CC** (**C**ondition **C**ode) della **PSW**;
4. Imposta la locazione di memoria con sequenza di 1.

L’istruzione TS a volte si chiama **TSL** – **T**est and **S**et **L**ock.

1. **L’istruzione swap**: istruzione indivisibile che scambia il contenuto di due locazioni di memoria.
2. Un **semaforo** è una variabile intera condivisa che può assumere solo valori non negativi e su cio sono possibili sole 3 operazioni:
3. Inizializzazione (con un valore >= 0);
4. Operazione indivisibile **wait**:
5. Se il semaforo ha valore >0, viene decrementato;
6. Se il semaforo ha valore = , il processo che esegue la wait viene “bloccato sul semaforo”
7. Operazione indivisibile **signal**:
8. Se c’è almeno un processo bloccato sul semaforo, uno dei processi bloccati sul semaforo viene “sbloccato”;
9. Se nessun processo è bloccato sul semaforo, il semaforo viene incrementato.
10. Un tipo **monitor** consente di:
11. Dichiarare variabili, alcuni delle quali chiamate **variabili di condizione**;
12. Inizializzare delle variabili;
13. Creare procedure che usano le variabili
14. Si parla di allocazione contigua della memoria quando ogni processo è allocato in una singola area di memoria contigua.
15. **Principio di località**: un indirizzo logico generato eseguendo un’istruzione ha probabilità elevata di essere vicino agli indirizzi logici generati eseguendo le istruzioni più recenti.
16. Esistono due tipi di allocazione:
17. **Allocazione fissa**: il numero di frame allocati ad un processo è scelto staticamente e non varia mai;
18. **Allocazione variabile**: il numero di frame allocati ad un processo può variare durante l’esecuzione.

Abbiamo due tipi di replacement:

1. **Replacement locale**: per allocare un frame ad una pagina del processo **P** può essere liberato solo un frame allocato a **P’**.
2. **Replacement globale**: per allocare un frame ad una pagina del processo **P** può essere liberato un frame allocato a qualsiasi processo.
3. Dato un parametro **d**, il **working set** di un processo **P** è l’insieme delle pagine visitate nelle ultime **d** istruzioni di **P**.