# Riassunti

## CPU

La **CPU** preleva le istruzioni dalla memoria, le decodifica e le esegue (Fetch, decode, execute, goto → Ciclo “tradizionale”)

Registri generali → memorizzano variabili e risultati temporanei

Registri di controllo → per controllarne il funzionamento

### Registri di controllo:

- Program Counter (PC) → Indirizzo istruzione da prelevare.
- Stack Pointer (SP) → Puntatore al top dello stack contenente i frame delle procedure già iniziate ma non ancora terminate.
- Program Status Word (PSW):
  - Privileged Mode (PM) → Bit di modalità user/kernel
  - Condition Code (CC) → Codici di condizione impostati da istruzioni di confronto / Codici di proprietà di operazioni artimetiche
  - Interrupt Mask (IM) → Bit usati per gestire interrupt
  - Interrupt Code (IC) → Bit usati per gestire interrupt
  - Memory Protection Information (MPI) → Informazioni sulla porzione di memoria accessibile

## Interrupt

Gli **interrupt** costituiscono un meccanismo per notificare alla CPU un evento o una condizione avvenuti nel sistema ce devono essere trattati dal SO ed interrompere il normale ciclo del programma.

### Hardware interrupt:

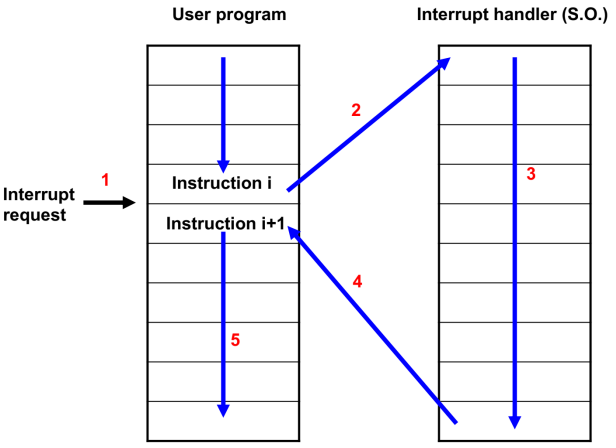
- I/O interrupt → Usati dai dispositivi I/O per notificare eventi alla CPU che richiedono di essere trattati dal SO.
- Timer interrupt → Usati dal clock per notificare il tick.

Si tratta di eventi asincroni rispetto al programma in esecuzione (il prog non ha modo di prevedere se/quando questi eventi si verificano)

### Fasi dell’interrupt HW:

1. 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 id prelevare l’istruzione i+1 (quella successiva)
2. Terminata l’esecuzione dell’istruzione i, la CPU sospende l’esecuzione di P (non preleva i+1) e salta alla procedura di gestione dell’interrupt (detta **interrupt handler**)
3. L’interrupt handler (che fa parte del SO) gestisce l’interrupt
4. L’interrupt handler restituisce il controllo a P (Oppure ad un altro programma interrotto in precedenza)

5. Il programma schedulato al punto 4 riprende la propria esecuzione dal punto in cui era stato interrotto (Se P, riprende l'esecuzione prelevando l'istruzione i+1), come nulla fosse accaduto




Come gestire gli interrupt a cascata (più interrupt di seguito)?

Vengono organizzati in classi di priorità, ecco la gerarchia:

- Machine error
- Clock interrupt
- Disk interrupt
- Fast device interrupt
- Slow device interrupt

# Program Interrupt

Tipi di program interrupt:

## ▼ Eccezioni (Interrupt interni)

**Eccezioni** (Interrupt interni) → Eventi causati da situazioni anomale interne al programma, che devono essere trattati dal SO. (Non sono asincroni rispetto al programma)

- eccezioni aritmetiche (divisione per zero, overflow)
- eccezioni di indirizzamento (page fault)
- violazione della protezione di memoria

## ▼ Software Interrupt (Trap)

**Software interrupt** (Trap) → Usati dai programmi per chiedere esplicitamente l'intervento del SO, chiamati da un'istruzione trap.

- Stessa gerarchia degli HW interrupt

**System call** → è una richiesta al SO effettuata da un programma, vengono realizzate con la trap.

System call di UNIX/Linux:

- create (creare file)

- open (creare un accesso ad un file/device)
- read (leggere n byte da un file/device)
- write (scrivere n byte da un file/device)
- close (invalidare un accesso ad un file/device)

Come invocare una system call:

1. Se programmiamo in assembly → TRAP
2. Se programmiamo in C → abbiamo una libreria di funzioni associate alle system call

# Dispositivi I/O e DMA

- **Controller** (Adapter) → componente elettronica che comunica con la CPU via bus (può gestire più dispositivi simili o uguali contemporaneamente)
- **Dispositivo** (vero e proprio) → componente meccanica che viene gestita dal controller.

Le interfacce tra controller e dispositivo sono standard.

L'interfaccia tra CPU e controller viene usata dai programmi del SO che gestiscono i dispositivi (**driver**). Questa interfaccia prevede:

- registri di controllo (detti anche porte I/O)
- un buffer

Abbiamo 2 possibili soluzioni per le comunicazioni tra la CPU e le porte di I/O:

1. Porte di I/O gestite con istruzioni macchina ad hoc (Le istruzioni devono essere previlegiate)
2. Memory Mapped I/O → ad ogni porta di I/O è assegnato un indirizzo di memoria e non servono istruzioni ad hoc

Abbiamo 3 possibili soluzioni per eseguire l'I/O per conto di un programma P. Supponiamo di voler trasferire n byte verso il (buffer del controller del) dispositivo da un buffer di memoria b:

1. Programmed I/O → Si busy waiting (la CPU rimane inutilizzata mentre aspetta che il dispositivo abbia finito di trattare il singolo byte. Non si sfruttano gli interrupt.)
2. Interrupt Driven I/O → No busy waiting, si overhead (abbiamo n interruzioni, una per byte)
3. **Direct Memory Access (DMA)** → Capace di accedere direttamente alla memoria e di lavorare in parallelo con la CPU.

# Processi

## Memory layout dei programmi

L'esecuzione di un programma necessita di 3 aree di memoria (in Unix si chiamano tradizionalmente **regioni**):

1. **Area di testo** → contiene il testo del programma in linguaggio macchina (non modificabile dal programma)
2. **Area dati** → contiene le variabili globali del programma
3. **Area di stack** → contiene i record di attivazione (frame) delle procedure già chiamate ma non ancora terminate
4. (Per strutture dinamiche) **Area heap** → una parte dell'area dati

Queste regioni si riferiscono al programma in esecuzione, quindi sono associate al processo, non al programma. (Vedere concetto di processo per altro)

## Il concetto di processo

**Processo** → è un'istanza di un programma in esecuzione. / 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.
- Le entità che vengono schedulate dal SO sono pertanto i processi, non i programmi.

Area di testo/dati/di stack, risorse logiche (file) e fisiche (dispositivi) assegnate al processo e lo stato della CPU (registri) sono associate al processo.

## Parallelismo e concorrenza

Due eventi sono **paralleli** se occorrono nello stesso momento.

La **concorrenza** è l'illusione del parallelismo.

(Assumiamo che l'HW offra una sola CPU e DMA controller) Assegnando la CPU a turno ai vari processi (detto **interleaving**), il SO realizza **l'esecuzione concorrente**. Il parallelismo è solo simulato, 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 (in alcuni momenti dispone della CPU ed in altri no), dipende dal numero di processi presenti e dal loro comportamento (pertanto non è riproducibile).

Nel caso di più CPU abbiamo 2 casi:

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

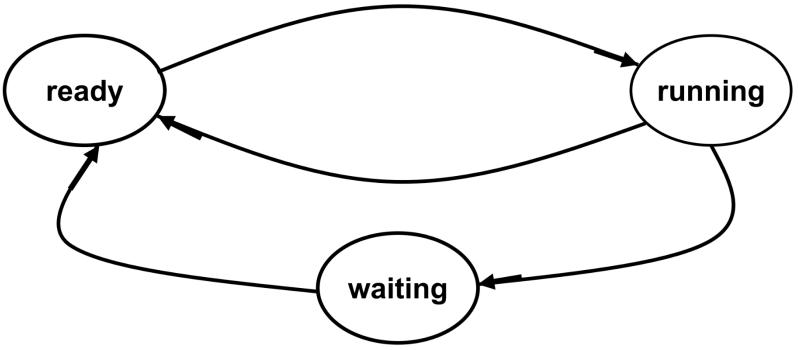
## Stati di un processo

In ogni istante, ogni processo gestito del SO si trova in un certo stato. Lo stato è un indicatore dell'attività che sta svolgendo il processo.

Abbiamo 3 stati:

- **Running** → il processo è in esecuzione (CPU sta eseguendo il programma associato al processo)
- **Ready** → il processo non è in esecuzione, ma sarebbe in grado se ottenesse la CPU (numero di processi è maggiore del numero di CPU).

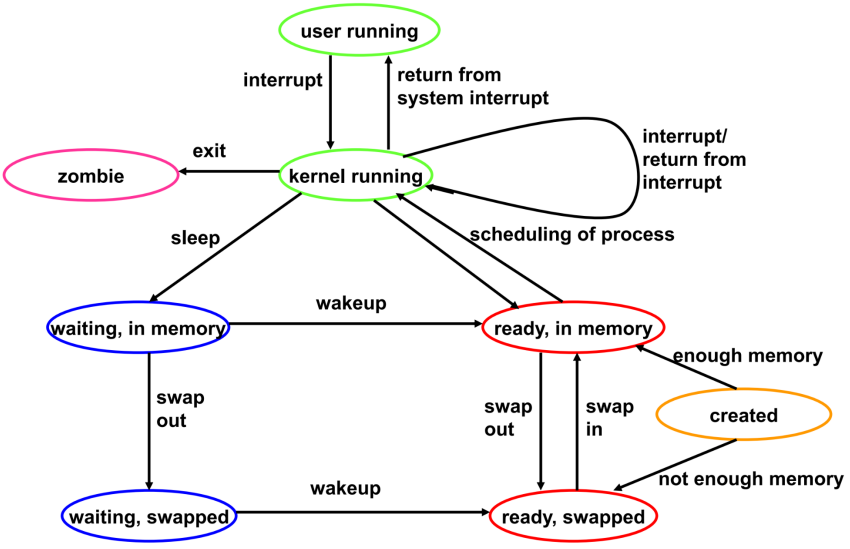
- **Waiting** → il processo non è in esecuzione, non sarebbe in grado di eseguire se ottenesse la CPU. (acquisirà la capacità di eseguire se/quando si verifica un evento che sta attendendo, come il compimento di un'operazione di I/O).




Abbiamo 4 possibili transizioni:

- Ready → Running: Lo scheduler seleziona il processo per l'esecuzione (Se ci sono più processi in stato di ready, sceglie sulla base della politica di scheduling).
- Running → Ready: Il processo subisce una preemption perché un processo con priorità maggiore diventa ready (In un sistema di timesharing può anche verificarsi la scadenza del time slice).
- Running → Waiting: Il processo si trova impossibilitato ad eseguire, si blocca in attesa di un evento sbloccante (Es: completamento I/O).
- Waiting → Ready: Si verifica l'evento sbloccante atteso dal processo.

Schema degli stati di UNIX System V:




## Implementazione dei processi → Process Control Block (PCB)

Il SO deve tener traccia di tutti i processi presenti nel sistema, avvalendosi di apposite strutture dati.

Il SO mantiene una **process table**, con una voce, **Process Control Block (PCB)**, per ogni processo (Le voci nel PCB variano in base al SO) (i PCB sono nella kernel area).

Voci presenti in ogni PCB:

- Process Identifier (PID): identificativo del processo
- stato del processo
- stato della CPU
- priorità (ed altri parametri per lo scheduling)
- puntatore ad una zona di memoria che contiene info per accedere all'area di testo
- puntatore ad una zona di memoria che contiene info per accedere all'area dati
- puntatore ad una zona di memoria che contiene info per accedere all'area di stack
- puntatore ad una zona di memoria che contiene info per accedere ai file aperti (file descriptor)
- puntatore ad una zona di memoria che contiene info per accedere ai dispositivi aperti
- un PCB pointer (che serve per creare:
  - la lista dei processi ready
  - per ogni evento che può essere atteso dai processi, una lista dei processi waiting)

**Page fault** → è un'eccezione causata da un accesso ad una pagina di memoria virtuale che non è nella memoria fisica.

**Swap out** → Il SO effettua ad intervalli regolari degli swapping out delle pagine meno usate per "liberare memoria".

## Implementazione dei processi → Context Switch

Il contesto di un processo (**process context**, detto anche ambiente) è costituito da:

- Register context → contiene dei registri della CPU
- Uer-level context → aree testo, dati e stack
- System-level context → PCB e informazioni per accedere a memoria, file e dispositivi cui punta il PCB
- Kernel stack → contiene i frame degli interrupt handler e delle procedure chiamate dagli interrupt handler.

Il SO effettua 3 **operazioni fondamentali sui processi**:

- Context save → quando il processo running va in ready o waiting, il SO salva nel PCB tutti i dati necessari a far ripartire il processo in futuro. (via HW)
- Scheduling → tenendo conto della politica di scheduling, il SO sceglie un processo tra quelli in ready per l'esecuzione
- Dispatching → dopo che un processo viene schedulato, il SO deve ripristinarne il contesto sfruttando i dati salvati nel PCB al momento del context save.

Si parla di **context switch** quando il SO effettua il context save per un processo P, schedula un processo P' diverso da P e ne effettua il dispatching.

## Creazione e terminazione di processi

Il SO mette a disposizione un'apposita system call per la creazione di processi (fork in UNIX/Linux, CreateProcess in APIWin32).

In UNIX/Linux, tra processo che esegue la system call (processo padre) ed il processo creato (processo figlio), si creare una relazione gerarchica.

La system call **exec** serve per "cambiarsi il programma". Quando si esegue exec:

- il processo ottiene nuove aree di testo, dati e stack
- registri PSW, PC, SP assumono valori nuovi
- registri generali azzerati
- file vengono chiusi e dispositivi rilasciati

La system call **wait** è una richiesta al SO di mettersi in stato di WAITING. L'evento atteso è la terminazione di un processo figlio. (Risultato system call = PID del figlio).

La system call **exit** viene eseguita quando un processo termina, va in stato ZOMBIE.

Tipi di processi in UNIX:

- User process → associati ad un utente che lavora ad un terminale
- Deamon process → non associati ad un utente, non terminano, eseguono funzioni vitali per il sistema, modalità user
- Kernel process → sono deamon che eseguono in modalità kernel (accedono a procedure kernel senza system call, sono "potentissimi" ad esempio possono controllare i propri parametri di scheduling)

Al momento del boot è necessario che venga eseguito del codice di inizializzazione che "costruisce" almeno un processo, dal quale possono discendere tutti gli altri.

Circostanze che causano la terminazione di un processo:

- System call per terminarsi o terminare altri processi
- L'interrupt handler delle eccezioni può forzare la terminazione del processo
- System call dove i processi chiedono la terminazione di altri processi al SO (kill in UNIX, TerminateProcess in APIWin32)

Quando un processo termina:

- rilascia tutte le risorse
- la distribuzione del PCB non è sempre immediata
- se ha figli:
  - in UNIX non vengono uccisi ma messi come figli del processo init
  - in Windows non vengono uccisi

## System call in UNIX

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

Le voci della PCB sono inserite nella u area solo quando il processo è running, mentre nel PCB servono sempre.

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

# Thread

Un'applicazione può avere più compiti (detti task) da portare avanti contemporaneamente.

Se il linguaggio offre i costrutti di programmazione concorrente, cioè i costrutti per:

- definire flussi di esecuzione sequenziale indipendenti per i vari task
- eseguire i flussi contemporaneamente

si può guadagnare in termini di:

- semplicità di programmazione
- efficienza di esecuzione

Ad esempio, il C offre le funzioni di libreria che consentono di realizzare i flussi mediante processi.

Programmando l'applicazione con più processi rende l'attività di programmazione più semplice, ma si rischia di non guadagnare in efficienza.

> Infatti, l'ambiente a processi è stato pensato per processi associati ad applicazioni diverse (multiprogrammazione) e non per realizzare applicazioni con processi concorrenti.
>
> Soluzione? Avere gruppi di "processi light" che condividano memoria e risorse e che differiscano solo per il CPU state, cioè i thread.

**Thread** → è un'esecuzione di un programma che usa le risorse di un processo.

Un processo può avere più thread.

Per ogni thread è necessario avere un Thread Control Block che contiene:

- Thread Identifier (TID)
- CPU state
- stack
- stato

Il thread switching tra due thread dello stesso processo introduce meno overhead rispetto al classico context switch, a vantaggio dell'efficienza.

È possibile implementare i thread nello spazio kernel e spazio user.

…

# Synchronization

## Interazioni tra processi

…

## Race condition

Definizione formale:
Sia
$d$ un dato condiviso dai processi $P_1$ e $P_2$

Siano $o_1$, $o_2$ operazioni su $d$ eseguite da $P_1$, $P_2$, rispettivamente.

Siano $f_1$, $f_2$ funzioni tali che, se $d$ ha valore $v_d$:

- $f_1(v_d)$ è il valore assunto da $d$ eseguendo $o_1$;
- $f_2(v_d)$ è il valore assunto da $d$ eseguendo $o_2$;

Le operazioni $o_1$ e $o_2$ danno luogo ad una **race condition** (corsa critica) sul dato condiviso $d$ se, eseguendo $o_1$ e $o_2$ con $d$ avente un valore iniziale $v_d$, accade che $d$ assume un valore $v_d^l$ diverso da $f_1(f_2(v_d))$ e da $f_2(f_1(v_d))$.

…

# Memory

…