SCHEDULING

Lo scheduling nel contesto dei sistemi operativi riguarda il come la **CPU** viene allocata ai vari processi. È il modo in cui si decide quale programma far andare per primo e per quanto tempo.

Ci sono **due parti** che riguardano la schedulazione dei processi:

- **Il context switch** salva i registri del processo attualmente in esecuzione e carica i registri del prossimo processo da eseguire. Questa operazione è scritta in **linguaggio macchina**, perché deve essere molto veloce.
- Lo **scheduler** è la parte del kernel che decide il prossimo processo da mandare in esecuzione, **seguendo una certa politica (policy) in C**

I sistemi operativi ci danno **l'illusione che i programmi vadano in parallelo**, anche se in realtà **il computer può fare una cosa alla volta**. *Come ci riesce?* **fa eseguire un programma per un piccolo intervallo di tempo**, poi lo mette in pausa e **fa partire un altro programma**, poi un altro ancora. Questo intervallo si chiama **time slice** o **quantum**, ed è talmente breve (millisecondi) che **noi non ce ne accorgiamo**: vediamo tutto come se stesse succedendo allo stesso momento. Tutto ciò è possibile grazie a **context switch.**

Come fa il sistema operativo a riprendere il controllo ed eventualmente uccidere il processo che si stava eseguendo e farne partire un altro?

Abbiamo diversi approcci:

- **Cooperativo:** il sistema operativo si "fida" dei processi.
  - i processi faranno (si spera) delle system call e chiamano il **kernel,** il quale può decidere cosa fare con il processo
  - se il processo non fa la system call il computer "muore", rimane per sempre nel processo e quindi **si blocca tutto**

- **Non cooperativo:** il sistema operativo riprende il controllo "a forza", tramite un **timer interrupt.** Metodo usato nei sistemi moderni. Logicamente funziona così:
  - C'è un timer hardware che **ogni tot millisecondi** manda un **segnale (interrupt)** alla CPU.
  - Quando questo interrupt arriva, **il processore interrompe quello che stava facendo** e **passa al sistema operativo**.
  - Così il kernel può decidere: "hai finito il tuo tempo, adesso tocca a qualcun altro".

Ogni processo ha due stack:

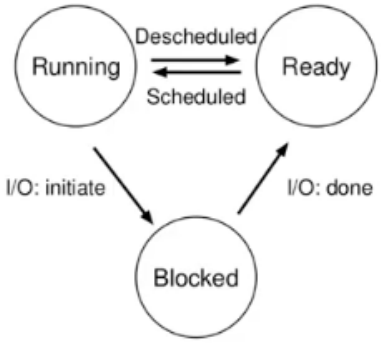
- **Stack utente:** lo usa quando esegue il codice normale (il programma)
- **Stack kernel:** lo usa quando entra nel kernel

Quando arriva un **interrupt**, la CPU passa automaticamente allo **stack del kernel**.
Questo è importante perché il **kernel ha sempre il suo stack personale**, che viene inizializzato all'avvio del sistema. In questo modo, quando il kernel deve gestire l'interrupt, **usa uno stack sicuro e valido**, e **non si affida allo stack del processo utente**, che potrebbe essere corrotto o malfunzionante. Infatti, lo **stack pointer** è un registro che punta alla cima dello stack, ma nulla vieta a un processo di usarlo male, modificandolo in modo errato o pericoloso.
Passando subito allo stack del kernel, **il sistema evita questo rischio** e garantisce che l'interrupt venga gestito in modo corretto.

Se vogliamo passare dal **processo A** al **processo B** quello che succede è:

- Si stava eseguendo il processo A
- Arriva l'interrupt
- Verranno salvati dei registri sul **Kernel stack** del processo A
- Si va ad eseguire **l'interrupt handler**
- Se il kernel decide di passare al processo B si salverà nel **PCB*** i valori attuali del registro della **CPU** e andrà a caricare quelli del proesso B
- Ritornerà dall'interrupt

*Il **PCB** è una struttura dati che contiene le informazioni dei processi.



Un processo nella sua vita può essere in tanti stati. I primi due stati sono lo stato **Running** e **Ready.**

Ready = Il processo è pronto ad essere eseguito ma in questo momento non si sta eseguendo.
Running = il processo è in esecuzione.

Esempio: abbiamo tre processi **A**, **B** e **C.** Mandiamo in esecuzione **A**, avremo **B** e **C** che sono **Ready** e A è **Running.** Dopo un certo tempo arriva interrupt e il kernel decide di **deschudelare** A e quindi ora **A** diventa **Ready** e **B running.** Questo processo continua così e abbiamo l'illusione che tutti siano eseguiti in parallelo.

Lo stato **Blocked/Waiting** significa che il processo non ha ricevuto un Input e sta aspettando.

le politiche di scheduling sono quegli algoritmi che decidono quale processo mandare in

esecuzione, abbiamo **delle metriche** per decidere che algoritmi usare. I processi sono spesso chiamati **job** e i processi che girano in un sistema sono detti **workload.**

Per misurare la "bontà" di un algoritmo di scheduling, rispetto a un altro, abbiamo bisongo di metriche (criteri). Inizialmente ci concentremo sulle **performance**, un altro aspetto è la **fairness.** Una metrica importante è **turn around** che si ottiene con la **differenza tra tempo di completamento e tempo di arrivo**. Questa metrica ci serve per capire quanto è efficace un algoritmo. Più è bassso meglio è.

Turn-around time

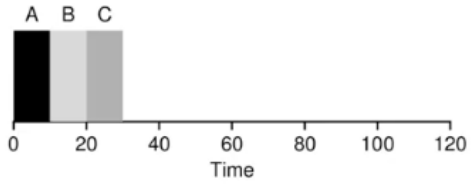
$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$$

con le assunzioni iniziali, $T_{\text{arrival}} = 0$ quindi:

$$T_{\text{turnaround}} = T_{\text{completion}}$$

Algoritmo FIFO (first in, first out)

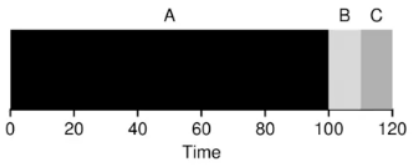
Se **A**, **B** e **C** arrivano nello stesso tempo, l'algoritmo deve scegliere un ordine. È facile da implementare ma può essere poco equo se un processo lungo arriva prima di tutti bloccando gli altri **(effetto convoglio)**



T_turnaround = (10 + 20 + 30) / 3 = 20

Effetto convoglio

Se **A** dura 100, **B** e **C** durano 10



Il tempo medio questa volta è di: (100 + 110 + 120) / 3 = 110.

L'effetto convoglio è quando gli altri due job sono corti e il primo è lungo.

Algoritmo SJF (Shortest Job First)

L'idea è quella di eseguire **i job più corti**. è ottimale in termini di **tempo di attesa medio** perché eseguendo prima i job più corti, riduce il tempo totale di attesa dei processi più lunghi.

STCF (shortest time to completion first)

mando in esecuzione quello che terminerà per primo. Non possiamo usarlo per davvero in quanto in un sistema operativo non sappiamo quanto sia lungo un processo.

Un'altra metrica importante è il **tempo di risposta** (response time), il tempo di risposta si calcola con la **differenza tra il tempo di qualcosa che inizia l'esecuzione e il tempo di arrivo**.

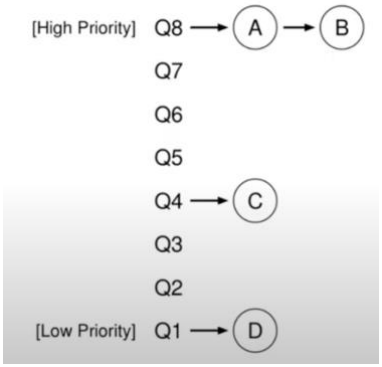
Round Robin
Se volessimo migliorare il **tempo di risposta**, potremmo usare l'algoritmo **Round Robin**. L'idea è quella di mandare in esecuzione **tutti i job pronti** per **una piccola fetta di tempo**, poi passare al prossimo, e così via a rotazione. In questo modo, **ogni processo inizia a girare quasi subito**, anche se per poco tempo. L'aspetto negativo è che, alternando continuamente tutti i job, **l'esecuzione si diluisce**: i processi impiegano più tempo a terminare, e quindi **il tempo di turnaround aumenta**.

Algoritmici realistici

Multi-level Feedback Queue

È un algoritmo che **cerca di bilanciare turnaround e tempo di risposta,** dando priorità ai processi interattivi (fanno tanto I/O) rispetto a quelli "pesanti" (usano tanta CPU).

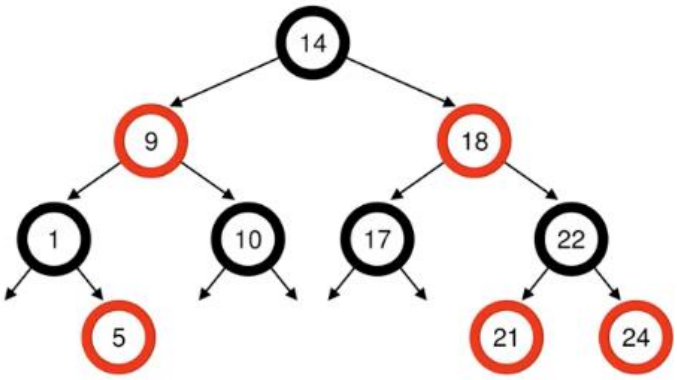
L'idea è di avere **più code a priorità diverse**. Per i processi sulla stessa coda vanno in **Round-Robin**. La priorità di un processo verrà variata in base a come il processo si comporta. Quindi l'idea di questo algoritmo è di usare la storia di un processo per predirne il futuro.



Descriviamo l'algoritmo dando una serie di regole. Nella figura abbiamo 8 code, **Q1** è bassa priorità mentre **Q8** alta priorità. Le regole ci dicono che se la priorità di **A** è maggiore della priorità di **B** allora gira **A** e non gira **B**. Se A e B sono uguali (stessa priorità) li mando a **Round Robin**. Nella figura abbiamo appunto A e B che vanno in RR mentre C e D vanno in **starvation.**
Quando arriva un job nuovo questo viene messo in priorità massima in modo che viene eseguito subito e il **response time** di conseguenza è basso. Se un job usa tutta la sua fetta di tempo allora gli abbasso la priorità. Poi, ogni s secondi spostiamo tutti i job alla priorità più alta.

Linux usa un algoritmo completamente diverso e si chiama **CFS**.
L'idea del **CFS** è che se ho tre processi che vogliono usare una **CPU** ne do un terzo a ciascuno. In generale l'idea è quella di mandare in esecuzione il processo che ha usato la **CPU** per meno tempo. Il **CFS** misura il tempo utilizzato sulla **CPU** tramite un valore chiamato **virtual runtime.** Quando c'è da scegliere un processo da mandare in esecuzione si usa quello col vruntime più piccolo. **CFS** usa diversi parametri, tra cui **sched_latency.**

Per trovare efficientemente il minimo/aggiornare il vruntime dei processi pronti, essi vengono tenuti in un albero binario bilanciato di tipo rosso/nero:



Costo: log n