

## Compiti del sistema operativo

Il **sistema operativo**, più in dettaglio, svolge principalmente due compiti:

- è il gestore delle **risorse hardware** (CPU, memoria, periferiche) che vengono usate da specifici programmi per eseguire il proprio compito;
- fornisce il supporto all'utente per impartire i comandi necessari al funzionamento del computer, cioè fa da **interfaccia** tra l'hardware e il software applicativo.

In particolare, sono gestite dal SO tutte le funzioni generali della macchina, come l'aspetto grafico delle visualizzazioni su monitor, la scrittura e la lettura dei dischi, la messa in esecuzione e la chiusura dei vari programmi, la ricezione e la trasmissione di dati attraverso tutti i dispositivi di **I/O**.

Il SO risiede sull'hard disk come tutti gli altri programmi e viene caricato nella memoria **RAM** all'accensione della macchina, o meglio, solo una sua parte viene caricata in memoria centrale: il **nucleo** (o **kernel**) che rimane sempre caricato in memoria.

## La schedulazione dei processi

Tra le operazioni che il sistema operativo deve effettuare rientra l'assegnazione della CPU ai processi che sono nella ready list, cioè nella lista dei processi pronti.

I meccanismi con i quali i processi vengono scelti prendono il nome di **politiche di gestione o di schedulazione** (scheduling) e il componente del SO che si occupa di questa gestione si chiama **job scheduler**.

In un SO con partizione del tempo un solo processo è in esecuzione e tutti gli altri sono posizionati in code, delle quali le due principali sono le già citate **coda dei processi pronti (RL)** e **coda di attesa di un evento (WL)**.

A seconda della modalità di gestione, lo scheduler sospende il processo che è in esecuzione mettendolo nella coda dei processi pronti e "risvegliando" un processo da tale coda per assegnargli la CPU: questo evento produce quello che si chiama **cambio di contesto (context-switch)**.

Il processo che viene sospeso dovrà successivamente essere ripristinato senza che rimanga traccia di quanto è successo: al processo deve "sembrare" di aver sempre posseduto la CPU, quindi il SO deve fare una "fotografia" di tutto quello che il processo utilizza e salvarlo, per poterlo poi ripristinare.

Particolare attenzione deve essere posta quindi allo stato del processo, in base alle seguenti osservazioni:

- naturalmente non è necessario salvare il codice del programma;
- lo **stack** e la memoria **heap** non devono essere salvati, in quanto sarà lo stesso sistema operativo a preoccuparsi di **non** modificarne i contenuti;
- sicuramente il contenuto dei registri verrà modificato dall'esecuzione del nuovo processo, quindi tutti questi devono essere salvati;
- lo **stack pointer** deve essere salvato insieme ai registri e al **program counter**.

Quando la CPU riattiva un processo che è stato sospeso per prima cosa analizza il suo **PCB** per individuare il suo **stack pointer**, quindi da lì recupererà i valori dei registri e del program counter da ripristinare per poter riprendere l'esecuzione proprio dall'istruzione che era stata sospesa.

La parte del **SO** che realizza il cambio di contesto si chiama **dispatcher**.



Le operazioni eseguite per il cambio di contesto hanno generalmente la durata di 1 millisecondo.

## Pre-emptive

Non tutti i processi possono essere sospesi dal SO in ogni istante della loro esecuzione: per loro natura alcuni processi devono terminare la loro esecuzione (oppure un insieme di istruzioni che devono essere eseguite senza interruzione, come un'operazione di I/O) e solo quando sono in particolari situazioni possono essere interrotti.

Questi processi vengono chiamati **non pre-emptive**, a differenza di quelli che possono essere interrotti che sono i processi **pre-emptive**.

I sistemi a divisione di tempo (time sharing) hanno uno scheduling pre-emptive.

## I criteri di scheduling

Prima di poter analizzare i criteri di **scheduling** è necessario introdurre la definizione di alcuni termini.

Gli obiettivi primari delle **politiche di scheduling** sono:

- massimizzare la percentuale di utilizzo della CPU; l'ideale sarebbe raggiungere una percentuale di utilizzo del 100% eseguendo lavoro utile, cioè "sprecare" il minor tempo per il cambio di contesto, minimizzando i tempi di **turnaround**;
- massimizzare il **throughput del sistema**, cioè il numero di processi completati nell'unità di tempo;
- ridurre al minimo i **tempi di risposta** del sistema quando un nuovo programma viene mandato in esecuzione;
- minimizzare i **tempi di attesa** tra un'esecuzione e l'altra e il **tempo totale di permanenza** di ciascun processo nel sistema;
- ottimizzare il **burst** di CPU e di I/O.

### THROUGHPUT

Il numero medio di job, programmi, processi o richieste completati dal sistema nell'unità di tempo.

### TEMPO DI COMPLETAMENTO (TURNAROUND TIME)

Il tempo dalla sottomissione di un job, programma o processo da parte di un utente nel momento in cui i risultati sono resi effettivamente disponibili all'utente stesso.

### TEMPO DI RISPOSTA (RESPONSE TIME)

Il tempo dalla sottomissione di una richiesta da parte dell'utente nel momento in cui il processo risponde, chiamato anche tempo di latenza.

### TEMPO DI ATTESA (WAIT TIME)

Si ottiene dalla somma degli intervalli temporali passati in attesa della risorsa.

### BURST DI CPU

Uso continuativo della CPU da parte di un processo.

### BURST DI I/O

Uso continuativo di un dispositivo di I/O da parte di un processo.

Possiamo inoltre elencare gli **obiettivi generali** che sono caratteristici di tutti i sistemi operativi:

- **equità**, ossia dare a ogni processo una porzione equa della CPU;
- **bilanciamento**, ossia tenere occupate tutte le parti del sistema;
- attuare politiche di **controllo**, ossia verificare che le politiche vengano messe in atto;
- uso della **CPU**, ossia tenere sempre occupata la CPU;



## Algoritmo di scheduling FCFS

Il primo algoritmo che analizziamo è l'**FCFS**, acronimo di **First-Come-First-Served**, cioè "il primo arrivato è il primo a essere servito".

In questo caso i processi vengono messi in coda secondo l'ordine d'arrivo, quindi **FIFO (First In First Out)**, e i processi pronti vengono schedati secondo il loro ordine d'arrivo, indipendentemente dal tipo e dalla durata prevista per la loro esecuzione; inoltre questo algoritmo è di tipo **non pre-emptive**, quindi i processi non possono essere sospesi e completano sempre la loro esecuzione.

I principali difetti di questo algoritmo consistono nel fatto che se un processo ha un lungo periodo di elaborazione senza interruzione (**CPU burst**), non potendolo sospendere, tutti gli altri devono aspettare la sua naturale terminazione, e questo potrebbe produrre elevati tempi di attesa soprattutto in presenza di una sequenza di processi con un elevato grado di operazioni di **I/O** che quindi provocano un basso utilizzo della **CPU** (**effetto convoglio**).

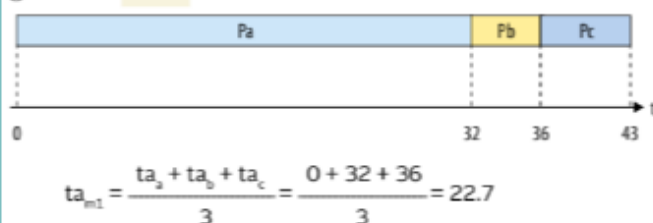
### ESEMPIO

In questo esempio vediamo come calcolare il **tempo di attesa medio** nel caso di tre processi e verificiamo numericamente come tale risultato sia casuale e quindi non possa essere preso come parametro di qualità, dato che dipende **solo dall'ordine di arrivo dei processi stessi**.

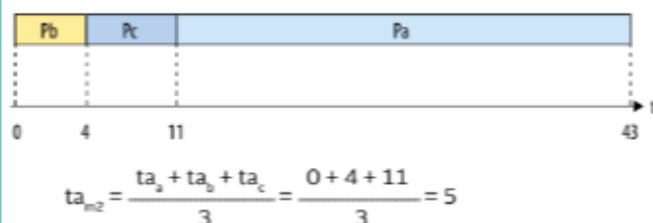
Supponiamo di avere tre processi Pa, Pb e Pc che rispettivamente richiedono:

- Pa = 32 unità di tempo/CPU
- Pb = 4 unità di tempo/CPU
- Pc = 7 unità di tempo/CPU

Nel primo caso i tre processi arrivano nell'ordine Pa, Pb e Pc. Calcoliamo il tempo medio di attesa  $ta_{m1}$  in questa situazione rappresentandolo su di un diagramma di **GANTT**:



Nel secondo caso l'ordine è diverso: Pb, Pc e Pa. Calcoliamo ora il nuovo tempo medio di attesa:




Evidentemente il parametro **tempo di attesa** non risulta essere significativo rispetto alla qualità dello scheduling in quanto il suo valore dipende dall'ordine dei processi.



### GANTT

Il **Gantt** (o diagramma a barre) è uno strumento puramente grafico in cui la durata delle attività è indicata con linee e i momenti critici del progetto (**milestone**) sono indicati con simboli (per esempio dei triangoli). Il **Gantt** fornisce una visione rapida del piano temporale di un processo e/o di un progetto.

### AREA DIGITALE

 **ESEMPIO DI SCHEDULAZIONE DI UN PROGETTO CON GANTT**

## Algoritmo di scheduling SJF

Per migliorare l'algoritmo **FCFS** in modo da ottenere sempre il minor tempo di attesa, basta scegliere tra la lista dei processi pronti quello che **occuperà per meno tempo** la CPU e che quindi verrà mandato in esecuzione per primo (**SJF, Shortest Job First**).

È però necessario che il sistema operativo sia in grado di effettuare una **stima dei tempi di utilizzo della CPU (burst di CPU)** di tutti i processi che sono pronti nella RL, e questa operazione, oltre a essere onerosa, non sempre dà risultati attendibili.

Inoltre potrebbe verificarsi la situazione in cui, mentre un processo è in esecuzione, se ne aggiunge uno in coda con un tempo stimato minore; in questo caso possiamo avere due possibilità:

- nel caso di situazione **non pre-emptive** non si fa nulla;
- nel caso di situazione **pre-emptive** è necessario stimare per quanto tempo ancora il processo deve rimanere in esecuzione e confrontarlo con il tempo previsto per il nuovo processo: se quest'ultimo è minore, si deve effettuare la sospensione e il cambio del contesto assegnando la CPU al nuovo processo.

In questa seconda situazione l'algoritmo cambia anche nome e diviene **SRTF (Shortest Remaining Time First)**, cioè si misura non il tempo del job intero, ma della parte rimanente che deve essere ancora eseguita.

### ESEMPIO

Calcoliamo il tempo medio per la situazione presentata di seguito, dapprima utilizzando l'algoritmo **SJF** e quindi migliorandolo con il **SRTF**.

Job	Durata	Arrivo
P1	9	0
P2	4	1
P3	10	2
P4	5	3

La sequenza di attivazione con l'algoritmo **SJF** è la seguente (dopo P1 viene mandato in esecuzione in processo pronto che richiede meno tempo):

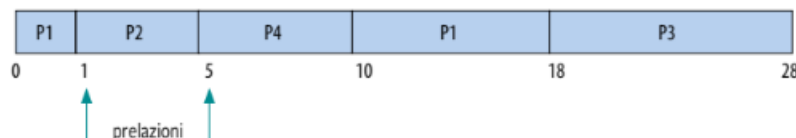


Il tempo di attesa medio è calcolato sottraendo a ogni job il "tempo di arrivo in coda":

$$Tw = (P1, P2, P4, P3) = [0 + (9 - 1) + (13 - 3) + (18 - 2)] / 4 = 8.5$$

In questo caso si è ipotizzata una situazione di **SJF non pre-emptive**: vediamo ora come si comporterebbe un **SRTF pre-emptive**: se arriva un nuovo processo con burst di esecuzione più corto questo effettua una "prelazione" rispetto al processo in esecuzione.

Il job P1 va in esecuzione ma all'istante di tempo 1 giunge il processo P2 che ha un burst minore del burst rimanente di P1 (4 contro 9 - 1): P1 in esecuzione viene sospeso e viene mandato in esecuzione P2; analoga situazione avviene all'arrivo del job P4 che ha un burst di durata 5, inferiore sia a quello di P3 sia al residuo di P1. Il diagramma di Gantt completo è il seguente:



Il tempo di attesa medio è:

$$Tw = (P1, P2, P4, P1, P3) = [0 + (1 - 1) + (5 - 3) + 10 + (18 - 2)] / 4 = 7$$

## Scheduling con priorità

Nella procedura di **scheduling con priorità** a ogni processo viene associato un numero intero che corrisponde a un livello di priorità con il quale il processo deve essere poi mandato in esecuzione: lo scheduler seleziona tra tutti i processi in coda quello a priorità più alta, che avrà la precedenza di esecuzione su tutti.



Alla sua terminazione verrà scelto il processo che ha la massima priorità tra quelli rimasti, e via di seguito; nel caso di due processi con medesima priorità si serve per primo il primo arrivato in coda, come nella **FCFS**. La priorità viene assegnata ai processi dal sistema operativo stesso, in base a criteri legati al tipo di processo e all'utente che lo ha mandato in esecuzione.

Il valore del **livello di priorità** viene dato tramite l'assegnazione di numeri interi  $p = x$  con diversi significati di priorità a seconda del sistema operativo: per esempio, sistemi come **Windows** e **Linux** associano a  $p = 0$  la priorità **più bassa** mentre in altri sistemi si attribuisce a  $p = 0$  la priorità **più alta**.

#### ESEMPIO

Visualizziamo la sequenza dei processi in base alla seguente tabella di priorità.

Processo	Durata	Priorità
P1	8	0
P2	1	2
P3	3	3
P4	5	4
P5	3	1

La sequenza di attivazione è la seguente:



Il tempo di attesa medio è:

$$Tw = (P1, P5, P2, P3, P4) = (0 + 8 + 11 + 12 + 15) / 5 = 9.2$$

Gli **algoritmi con priorità** possono essere di tipo sia **non pre-emptive** sia **pre-emptive**: in questo secondo caso, se si sta servendo un processo con priorità più bassa di uno nuovo appena giunto in coda, si cede la CPU a quello con priorità maggiore sospendendo il processo in esecuzione in quel momento.

Se continuano ad arrivare processi con alta priorità può avvenire il fenomeno della **starvation** dei processi, cioè i processi con priorità maggiore vengono sempre serviti a scapito di quelli con priorità bassa, che possono rimanere in coda anche per tempi indefiniti.

#### AREA DIGITALE



HIGHEST RESPONSE RATIO  
NEXT SCHEDULING

Una possibile soluzione è quella di introdurre le priorità variabili, cioè di modificare dinamicamente la priorità di un processo in base al tempo di attesa: se è da tanto tempo in coda, cioè è "invecchiato" (aging), gli viene alzato il livello di priorità mentre viene diminuito quello del processo in esecuzione man mano che aumenta il suo utilizzo della CPU.

La soluzione ottimale la si ottiene cambiando radicalmente politica, cioè adottando il **Round Robin**.

### Algoritmo di scheduling Round Robin

L'algoritmo classico utilizzato nei sistemi a partizione di tempo è il **Round Robin (RR)** dove tutti i processi pronti vengono inseriti in una coda circolare di tipo **FIFO (First In First Out)**, cioè inseriti in ordine di arrivo, tutti senza priorità, e a ogni processo viene assegnato un intervallo di tempo di esecuzione prefissato denominato **quanto di tempo** (o **time slice**) di durata che varia tra 10 e 100 millisecondi.

Se al termine di questo intervallo di tempo il processo non ha ancora terminato l'esecuzione, l'uso della CPU viene comunque affidato a un altro processo, prelevandolo sequenzialmente dalla coda e sospendendo il processo che era in esecuzione.

Possiamo osservare che l'algoritmo **RR** può essere visto come un'estensione di **FCFS** con **pre-emption** periodica a ogni scadenza del **quanto di tempo**.

Con questo algoritmo tutti i processi sono trattati allo stesso modo, in una sorta di "correttezza" (fairness), e possiamo essere certi che non ci sono possibilità di **starvation** perché tutti a turno hanno diritto a utilizzare la CPU.

#### ESEMPIO

Consideriamo la sequenza di attivazione riportata di seguito, supponendo che il quanto di tempo sia uguale a 4:

Processo	Durata
P1	20
P2	3
P3	7

Il tempo di attesa medio è:



$$Tw = (P1, P2, P3) = 11 / 3 = 3.6$$

Il dispositivo che rende possibile la sospensione di un processo ancora in esecuzione allo scadere del **time slice**, è il **Real-time clock (RTC)**. Esso non è altro che un chip impiantato sulla scheda madre contenente un cristallo di quarzo che viene fatto oscillare in modo estremamente stabile con segnali elettrici: tali oscillazioni scandiscono il tempo generando periodicamente delle interruzioni da inviare al sistema operativo.

È necessario però prestare molta attenzione al dimensionamento del time slice, in quanto le prestazioni del sistema sono direttamente legate alla sua durata:

- quando è **piccolo** abbiamo tempi di risposta ridotti ma è necessario effettuare frequentemente il cambio di contesto tra i processi, con notevole spreco di tempo e quindi di risorse (**overhead**);
- quando è **grande** i tempi di risposta possono essere elevati e l'algoritmo degenera in quello di **FCFS**.

Potrebbero poi sorgere problemi di decadimento delle prestazioni in quanto non sono presenti differenziazioni tra processi di sistema e processi utente, quindi anche i processi di sistema devono attendere il loro turno nel **Round Robin**.

La **soluzione ottimale perciò non esiste**: i moderni sistemi operativi combinano tra loro gli algoritmi qui presentati cercando in primo luogo di eliminare i problemi tipici di ogni algoritmo per ottenere una soluzione che possa essere mediamente buona per tutte le situazioni.

#### AREA DIGITALE

 PERIODO DELL'RTC