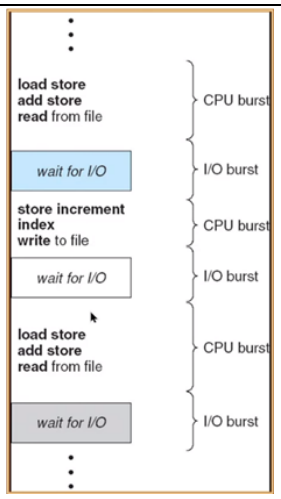
## 6. SCHEDULING DELLA CPU

Verranno illustrati quelli che sono gli algoritmi utilizzati dallo **schedulatore a breve termine**.
In realtà, durante l'esecuzione di un processo, ci sono una serie di cicli d'esecuzione della CPU (per un certo tempo uso la CPU) e attese di I/O.
Il ciclo di vita di un processo può essere visto come un'alternanza di CPU burst e I/O burst, come si vede nell'immagine al lato.
Se si considerano i CPU-burst di piccola entità sono molto più frequenti.
Lo scheduler della CPU può avvenire quando:

1. Si passa dallo stato di **esecuzione** allo stato di **attesa** per una operazione I/O o per una wait. In questo caso la schedulazione è **non preemptive** (senza diritto di prelazione).
2. Si passa dallo stato di **esecuzione** allo stato di **pronto** per l'arrivo di un segnale di interruzione. In questo caso la schedulazione è **preemptive** (con diritto di prelazione).
3. Si passa dallo stato di **attesa** allo stato di **pronto** perché è terminata un'operazione di I/O In questo caso la schedulazione è **preemptive** (con diritto di prelazione).
4. **Termina**. In questo caso la schedulazione è **non preemptive** (senza diritto di prelazione).




Quando parliamo di schedulazione **non preemptive** intendiamo che non si ha il diritto di interrompere chi è in esecuzione. Al contrario si dice **preemptive**.
Per capire qual è l'algoritmo migliore che seleziona uno dei processi pronti ad entrare nella CPU, si adottano questi criteri:

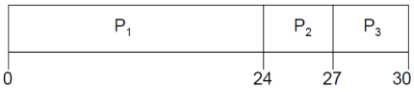
- **Utilizzo della CPU**: mantenere la CPU il più possibile impegnata. Si vuole massimizzare tale criterio.
- **Frequenza di completamento** (***througput***): numero di processi completati per unità di tempo. Si vuole massimizzare tale criterio.
- **Tempo di completamento** (***turnaround time***): intervallo che va dal momento dell'immissione del processo nel sistema al momento del completamento. Si vuole minimizzare tale criterio.
- **Tempo di attesa** (***waiting time***): somma dei tempi spesi in attesa nella coda dei processi pronti. Si vuole minimizzare tale criterio.
- **Tempo di risposta**: tempo che intercorre dalla formulazione della prima richiesta fino alla produzione della prima risposta, non l'output (per gli ambienti time-sharing). Si vuole minimizzare tale criterio.

### 6.0.1 FIRST-COME, FIRST-SERVER (FCFS)

Il primo processo che arriva è il primo che viene servito. In generale si hanno vari processi in coda, per ogni processo c'è il proprio CPU Burst, ossia per quanto tempo richiede la CPU.
Il processo $P_1$ ha bisogno di 24ms nella CPU, $P_2$ e $P_3$ per 3ms.
Siccome l'algoritmo è di tipo **FCFS**, entrano i processi in base all'ordine di ingresso.

| Process | CPU Burst |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

Si parte dall'istante 0 in cui si comincia. Naturalmente entra il processo $P_1$ e rimane nella CPU per 24ms, e di conseguenza esce a 24. Subito dopo entra $P_2$ per un tempo che è pari a 3 (esce a 27). Rimane $P_3$ che entra e rimane per 3ms (esce a 30). Tutto ciò si rappresenta nel seguente diagramma:




Se voglio valutare il tempo di attesa medio, effettuo i seguenti calcoli:
Chiamiamo **WT** (Waiting Time) il tempo di attesa di ogni Processo:

$WT - P_1 = 0$; $WT - P_2 = 24$; $WT - P_3 = 27$

Chiamiamo **Turnaround** il tempo totale in cui processo è stato all'interno del sistema:

$T - P_1 = 24$; $T - P_2 = 27$; $T - P_3 = 30$

Il **Tempo medio di attesa** è il seguente:

Tempo medio WT= (0+24+27)/3 = 17

Tempo medio Turnaround= (24+27+30)/3= 27

Supponiamo di considerare l'ordine di arrivo in maniera differente rispetto a prima. In questo caso i processi arrivano nel seguente ordine: $P_2$, $P_3$, $P_1$
In questo caso il diagramma è il seguente:


Valutiamo il WT:

$WT - P_1 = 6$; $WT - P_2 = 0$; $WT - P_3 = 3$

Tempo medio WT= (6+0+3)/3= 3 → questo valore è ben diverso da quello di prima.

Questo ci dice che il tempo di attesa dipende dall'ordine in cui i processi vengono serviti. Il fatto di aver messo $P_1$ alla fine ci consente di accorciare i tempi di attesa di quelli che lo precedono, dato che $P_1$ ha un CPU-burst molto grande. FCFS è una scelta poco oculata.

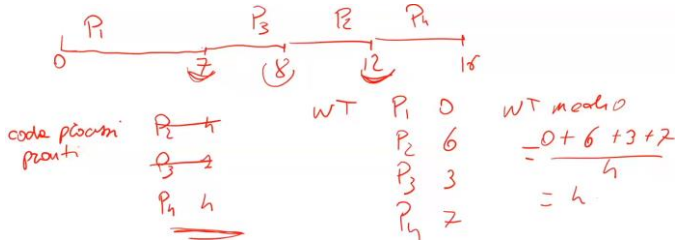
### 6.0.2 SHORTEST-JOB-FIRST (SJF)

Viene schedulato il processo con il prossimo CPU burst più breve. Spostando un processo breve prima di un processo lungo, il tempo di attesa del processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza il tempo d'attesa medio diminuisce.
Immaginiamo il seguente scenario, in cui oltre ad avere il classico Burst Time, abbiamo anche i tempi di arrivo dei vari processi →

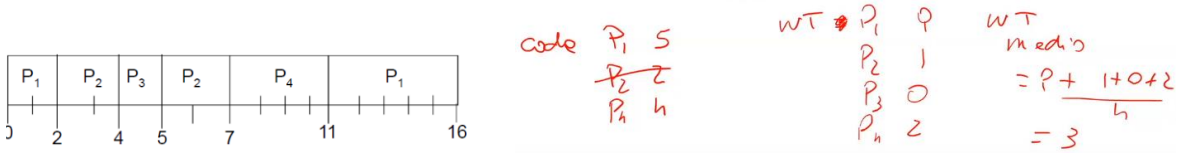
| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

In questo caso il criterio è di scegliere, quando possibile, il processo con Burst Time più piccolo.

Al tempo 0 naturalmente non c'è scelta, entra il processo $P_1$ in quanto è l'unico disponibile. Facciamo attenzione a questa cosa: stiamo considerando un caso **non-preemptive,** di conseguenza il processo $P_1$ ottiene la CPU e rimane per tutto il tempo che gli serve, dunque fino a 7.
Siamo al tempo 7, di conseguenza tutti i processi sono arrivati (dunque in una coda ho i processi $P_2 – P_3 – P_4$), in questo caso, dato l'algoritmo SJF, scelgo quello che ha Burst Time minore, dunque faccio entrare il processo $P_3$. Una volta processato $P_3$, nella coda mi rimangono $P_2$ e $P_4$, i quali hanno entrambi Burst Time pari a 4. In questo caso scelgo in maniera FIFO, il primo che è entrato in coda.
Di conseguenza servirò prima $P_2$ e poi $P_4$. Il diagramma di Gantt è il seguente:




Quando invece esaminiamo il caso **preemptive**, un processo che entra in CPU, può essere interrotto. Di conseguenza, esaminando l'esempio precedente, il processo $P_1$ che vuole stare in CPU per 7 secondi, non posso fare questa cosa immediatamente in quanto mi devo sempre porre il problema di cosa succede quando un nuovo processo arriva.
In questo caso, il processo $P_1$ è il solo a tempo 0 e dunque parte. All'istante 2 è arrivato $P_2$ che momentaneamente si mette in coda. Devo confrontare il tempo residuo di $P_1$ e quello di $P_2$ e devo scegliere quello che ce l'ha minore: considerando che $P_1$ ha tempo residuo 5 e $P_2$ ha Burst Time 4, scelgo $P_2$ che dunque all'istante 2 entra nella CPU. $P_1$ si mette in coda con tempo rimanente 5. Adesso devo ripetere la stessa identica idea nel momento in cui all'istante 4 arriva il processo $P_3$…. E arrivo alla conclusione che entra $P_3$ all'istante 4 e $P_2$ va in coda con tempo residuo 2.




Tale algoritmo può essere:

- **Nonpreemptive**: quando un processo ha ottenuto la CPU, non può essere prelazionato fino al completamento del suo cpu-burst.
- **Preemptive**: quando un nuovo processo è pronto, ed il suo CPU-burst è minore del tempo di cui necessita ancora il processo in esecuzione, c'è la prelazione. Questa schedulazione è detta *shortest-remaining-timefirst*.

**STIMA DEL PROSSIMO CPU BURST**

Quando abbiamo visto gli algoritmo di scheduling, abbiamo detto che un processo arriva con la sua richiesta di CPU burst. Consideriamo un processo che arriva e si mette in coda con il suo PCB che contiene tutte le sue informazioni. Nel PCB di un processo ci sta scritto a che punto sta il program counter, tabella dei file aperti, contenuto dei registri, etc…. Ma il processo non conosce il futuro, nel senso che dire "a me serve un CPU burst di 7", significa che sta prevedendo che quando entrerà in CPU gli serviranno 7ms.
Questo concetto di prevedere per quanti millisecondi verrà dedicata la CPU ad un processo, è un numero stimato. Per fare questa stima, ci si basa sulla seguente relazione di ricorrenza:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n \text{ dove:}$$

- $T_n$= valore reale dell'ennesimo CPU burst.
- $\tau_{n+1}$= valore previsto per il prossimo CPU burst.
- $\alpha$= prarametro con valore $0 \leq \alpha \leq 1$
- $\tau_1$= previsione primo CPU burst (valore di default)

Sviluppiamo la relazione di ricorrenza scritta in precedenza:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \quad \text{ma } \tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$$
$$= \alpha t_n + (1-\alpha)\left[\alpha t_{n-1} + (1-\alpha)\tau_{n-1}\right] =$$
$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \tau_{n-1} \quad \text{ma } \tau_{n-1} = \alpha t_{n-2} + (1-\alpha)\tau_{n-2}$$
$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2\left[\alpha t_{n-2} + (1-\alpha)\tau_{n-2}\right]$$
$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + (1-\alpha)^3 \tau_{n-2}$$

Notiamo che i coefficienti diventano sempre più piccoli, dunque io sommo tutta una serie di cose che vanno a decrescere. Nella stima che si fa per il prossimo CPU burst, si dà più peso ai CPU burst più recenti, rispetto a quelli più vecchi.

## 6.0.3 SCHEDULING A PRIORITÀ

In questo tipo di scheduling si assegna a ciascun processo, indipendentemente dal suo CPU burst, una priorità. Un processo all'interno del suo PCB mantiene il valore della sua priorità.
Ci può essere anche in quest'algoritmo il caso **preemptive** (durante l'esecuzione entra un processo con priorità maggiore) e **nonpreemptive**.
Anche lo **SJF** è un algoritmo a priorità, dove in quel caso la priorità è l'inverso della lunghezza del prossimo CPU burst (previsto).
Per questo tipo di algoritmo potrebbero esserci dei problemi: supponiamo di avere un processo in coda che ha una priorità bassa, se in un certo lasso di tempo continuano ad arrivare processi che hanno una priorità maggiore rispetto a quello con bassa priorità, quest'ultimo processo non entrerebbe

mai in CPU (questo fenomeno prende il nome di **starvtation**). Una soluzione per un caso del genere è quello di effettuare un'operazione di **invecchiamento** (aging) secondo cui si accresce gradualmente la priorità dei processi nel sistema in modo che ad un certo punto avrà una priorità tale da entrare nella CPU. Naturalmente tutte queste informazioni sono contenute nel PCB di un processo.

La priorità da assegnare può essere:

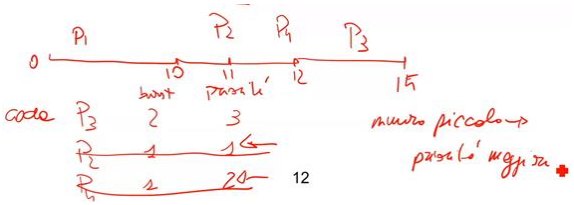
- **INTERNA** se dipende da fattori del processo governati dal SO (numero di file aperti, lunghezza media delle operazioni di I/O, etc.)
- **ESTERNA** (importanza del processo).

Vediamo come può funzionare un algoritmo di scheduling a priorità, caso **nonpreemptive**:

Teniamo a mente questa informazione: **più il numero è piccolo più la priorità è maggiore**.

| Process | Arrival Time | Burst Time | Priorità |
|---|---|---|---|
| $P_1$ | 0 | 10 | 3 |
| $P_2$ | 5 | 1 | 1 |
| $P_3$ | 3 | 2 | 3 |
| $P_4$ | 10 | 1 | 2 |

In questo caso all'istante 0 abbiamo solo il processo $P_1$ che di conseguenza entra in CPU e rimane al suo interno per tutto il tempo richiesto dal suo Burst Time (in quanto stiamo analizzando il caso non preemptive). All'istante 10 nella coda dei processi in attesa sono entrati tutti i restanti processi ($P_2 – P_3 – P_4$). La scelta, tenendo conto che più il numero è piccolo più la priorità è maggiore, ricade nel processo $P_2$. Una volta servito questo processo ci rimangono da scegliere $P_3$ e $P_4$. Procediamo in tal senso con $P_4$ e successivamente con $P_3$.



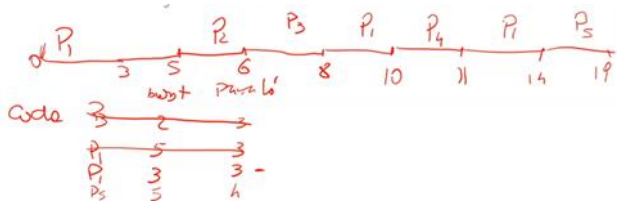

$WT – P_1 = 0;$
$WT – P_2 = 5;$
$WT – P_3 = 9;$
$WT – P_4 = 1;$

Il **Tempo medio di attesa** è il seguente: $WT = (0+5+9+1)/4 = 3,75$

Vediamo come può funzionare un algoritmo di scheduling a priorità, caso **preemptive:**

| Process | Arrival Time | Burst Time | Priorità |
|---|---|---|---|
| $P_1$ | 0 | 10 | 3 |
| $P_2$ | 5 | 1 | 1 |
| $P_3$ | 3 | 2 | 3 |
| $P_4$ | 10 | 1 | 2 |
| $P_5$ | 11 | 5 | 4 |

In questo caso il processo $P_1$ entra nella CPU all'istante 0. Arrivato all'istante 3 devo fermarmi e considerare che è arrivato il processo $P_3$ con priorità 3. Siccome $P_1$ e $P_3$ hanno la stessa priorità, faccio continuare il processo $P_1$, mentre $P_3$ va in coda. Arrivato all'istante 5 mi fermo e considero il processo $P_2$. Poiché quest'ultimo ha priorità 1 allora il processo $P_1$ esce dalla CPU e va in coda con tempo rimanente pari a 5 ed entra nella CPU il processo $P_2$. Ripeto ragionamenti analoghi fino alla fine… **Attenzione**: quando ho due processi in coda con la stessa priorità, faccio entrare nella CPU quello che è entrato per primo nella coda dei processi in attesa.




$WT – P_1 = 4;$
$WT – P_2 = 0;$
$WT – P_3 = 3;$
$WT – P_4 = 0;$
$WT – P_5 = 3$

Il **Tempo medio di attesa** è il seguente: $WT = (4+0+3+0+3)/5 = 2$

### 6.0.4 ROUND ROBIN (RR)

Nel RR ad ogni processo viene assegnato un quanto di tempo. Cioè si stabilisce che per ogni processo che entra nella CPU viene assegnato un quanto di tempo **q**. Qualunque processo sia, qualunque priorità, rimarrà nella CPU per un tempo fissato **q**. Abbiamo detto che un processo rimane nella CPU per un certo tempo **q**. Finito questo tempo, se il CPU burst non è terminato, il processo va in coda ed entra il successivo, secondo un ordine FIFO.

Adesso dobbiamo analizzare quanto deve essere grande questo valore **q**. Se **q** è molto grande ai fini pratici è come se stessimo analizzando un algoritmo FIFO. Se **q** è molto piccolo, si creano problemi in quanto ogni processo rimane nella CPU per un lasso di tempo molto piccolo. Per consumare tutto il suo CPU Burst, dovrebbe entrare e uscire un numero elevato di volte, generando troppi cambi di contesto (tempo perso inutilmente). Bisogna trovare dunque una buona misura di **q**.

Vediamo un esempio di applicazione di RR considerando **q=20**:

| Process | Burst Time |
|---|---|
| $P_1$ | 53 |
| $P_2$ | 17 |
| $P_3$ | 68 |
| $P_4$ | 24 |

Al tempo 0 arriva il processo $P_1$. Rimane in CPU per 20ms, dopodiché poiché avrà un tempo rimanente pari a 33, entra in coda e lascia spazio al processo $P_2$. Poiché $P_2$ ha un Burst Time pari a 17 e q=20, riesce a consumare tutto il suo Burst Time e di conseguenza non verrà più considerato. Ripeto lo stesso ragionamento per i successivi step…