

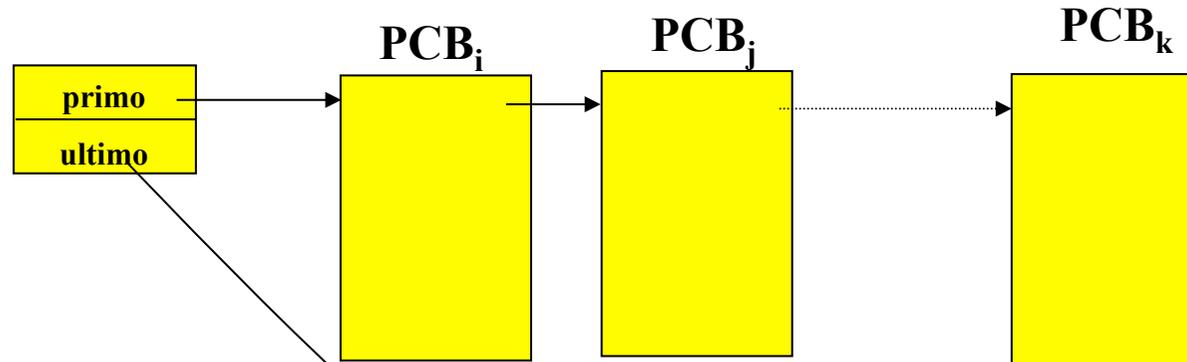
Scheduling della CPU

Scheduling della CPU

Obiettivo della multiprogrammazione:
massimizzazione dell'utilizzo CPU

- ➔ **Scheduling della CPU:**
commuta l'uso della CPU tra i vari processi
- ➔ **Scheduler della CPU (a breve termine):** è quella parte del SO che seleziona *dalla coda dei processi pronti* il prossimo processo al quale assegnare l'uso della CPU

Coda dei processi pronti (*ready queue*):



contiene i descrittori (process control block, PCB) dei processi pronti

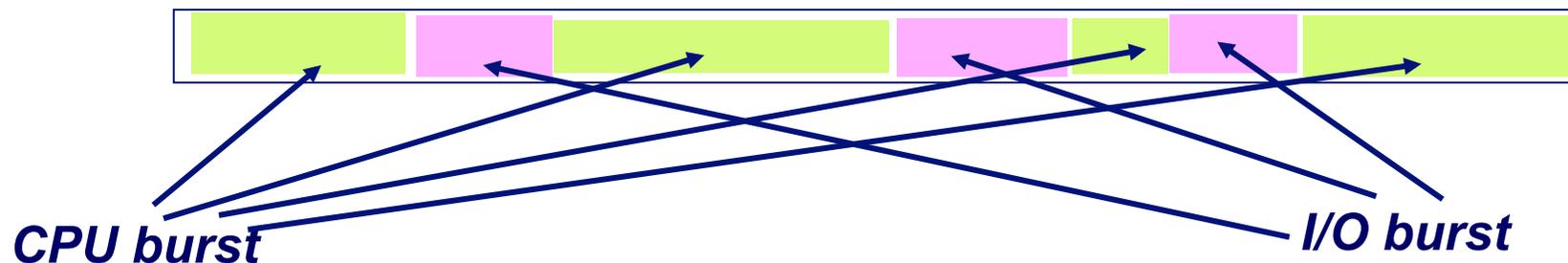
strategia di gestione della ready queue è realizzata mediante *politiche* (algoritmi) di scheduling

Terminologia: CPU burst & I/O burst

Ogni processo alterna

(burst = raffica)

- **CPU burst**: fasi in cui viene impiegata *soltanto la CPU senza interruzioni dovute a operazioni di I/O*
- **I/O burst**: fasi in cui il processo effettua *I/O da/verso una risorsa (dispositivo) del sistema*



- Quando un processo è in I/O burst, la CPU non viene utilizzata: in un *sistema multiprogrammato*, lo short-term *scheduler* assegna la CPU a un nuovo processo

Terminologia: processi I/O bound & CPU bound

A seconda delle caratteristiche dei programmi eseguiti dai processi, è possibile classificare i processi in

- ***I/O bound***: prevalenza di attività di I/O
 - ➔ *Molti CPU burst di breve durata, intervallati da I/O burst di lunga durata*
- ***CPU bound***: prevalenza di attività di computazione
 - ➔ *CPU burst di lunga durata, intervallati da pochi I/O burst di breve durata*

Terminologia: *pre-emption*

Gli algoritmi di scheduling si possono classificare in due categorie:

- *senza prelazione (non pre-emptive)*: CPU rimane allocata al processo *running* finché esso non si sospende volontariamente o non termina
 - *con prelazione (pre-emptive)*: processo *running* può essere *prelaziato*, cioè SO può sottrargli CPU per assegnarla ad un nuovo processo
- I sistemi a divisione di tempo hanno sempre uno scheduling *pre-emptive*

Politiche & meccanismi

Lo scheduler decide a quale processo assegnare la CPU; a seguito della decisione, viene attuato il cambio di contesto (context-switch)

Dispatcher : è la parte di SO che realizza il cambio di contesto

Scheduler = POLITICHE
Dispatcher = MECCANISMI

Criteri di scheduling

Per analizzare e confrontare i diversi algoritmi di scheduling, vengono considerati alcuni *indicatori di performance*:

- **Utilizzo della CPU**: percentuale media di utilizzo CPU nell'unità di tempo
- **Throughput** (del sistema): numero di processi completati nell'unità di tempo
- **Tempo di Attesa** (di un processo): tempo totale trascorso nella ready queue
- **Turnaround** (di un processo): tempo tra la sottomissione del job e il suo completamento
- **Tempo di Risposta** (di un processo): intervallo di tempo tra la sottomissione e l'inizio della prima risposta (a differenza del turnaround, non dipende dalla velocità dei dispositivi di I/O)

Criteri di scheduling

In generale:

- devono essere **massimizzati**
 - Utilizzo della CPU
 - Throughput
- invece, devono essere **minimizzati**
 - Turnaround (sistemi *batch*)
 - Tempo di Attesa
 - Tempo di Risposta (sistemi *interattivi*)

Criteri di scheduling

Non è possibile ottimizzare tutti i criteri contemporaneamente

A seconda del tipo di SO, gli algoritmi di scheduling possono avere **diversi obiettivi**

nei sistemi *batch*:

➤ *massimizzare throughput e minimizzare turnaround*

nei sistemi *interattivi*:

➤ *minimizzare il tempo medio di risposta* dei processi

➤ *minimizzare il tempo di attesa*

Algoritmo di scheduling FCFS

First-Come-First-Served: la coda dei processi pronti viene gestita in modo FIFO

- i processi sono schedulati secondo *l'ordine di arrivo* nella coda
- algoritmo *non pre-emptive*

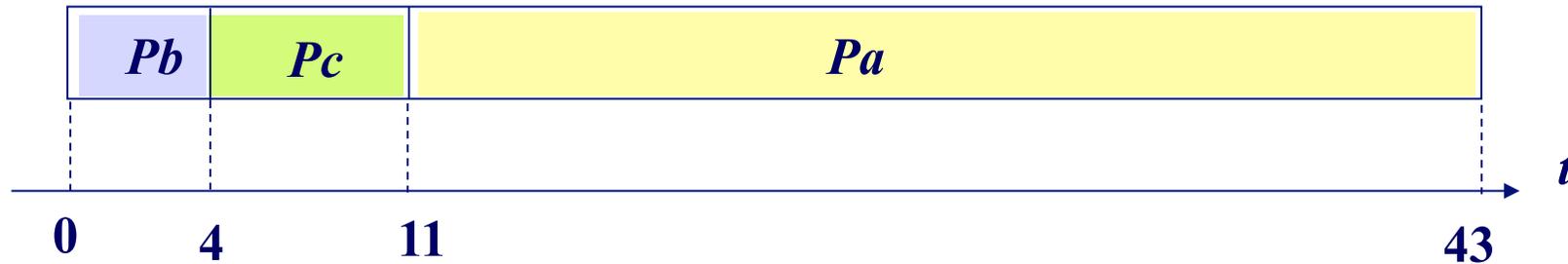
Esempio: tre processi [Pa, Pb, Pc] (diagramma di *Gantt*)



$$T_{\text{attesa medio}} = (0 + 32 + 36) / 3 = 22,7$$

Algoritmo di scheduling FCFS

Esempio: se cambiassimo l'ordine di scheduling
[Pb, Pc, Pa]



$$T_{\text{attesa medio}} = (0 + 4 + 11) / 3 = 5$$

Problemi dell'algoritmo *FCFS*

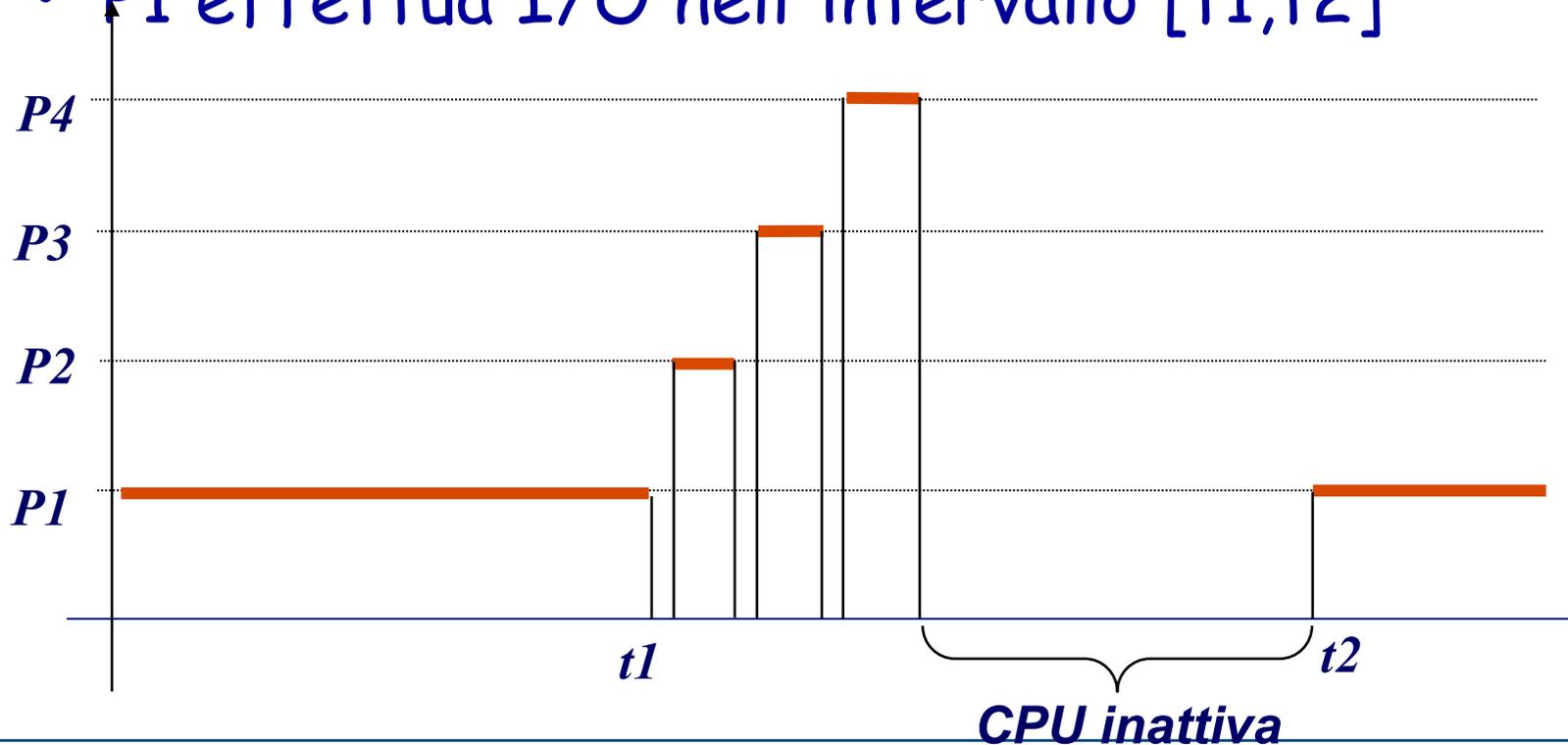
Non è possibile influire sull'ordine dei processi:

- nel caso di processi in attesa *dietro a processi con lunghi CPU burst (processi CPU bound)*, il tempo di attesa è alto
- **Possibilità di effetto *convoglio***
se molti processi I/O bound seguono un processo CPU bound: *scarso grado di utilizzo della CPU*

Algoritmo di scheduling FCFS: effetto convoglio

Esempio: [P1, P2, P3, P4]

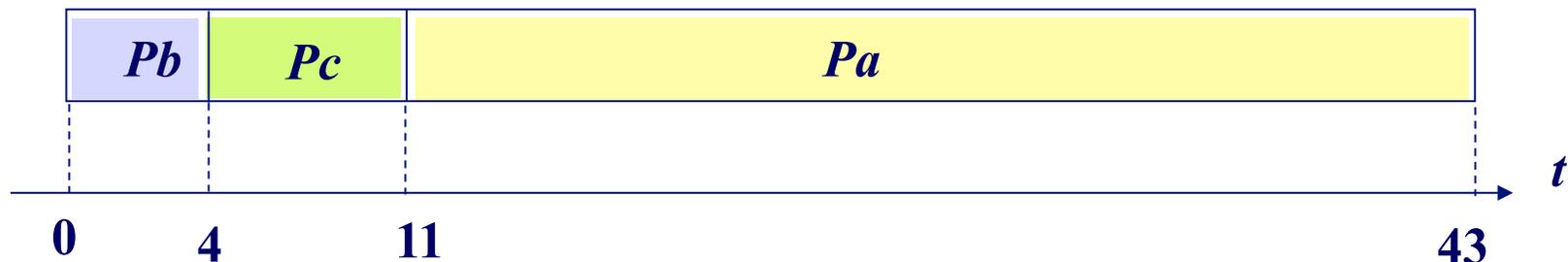
- P1 è CPU bound; P2, P3, P4 sono I/O bound
- P1 effettua I/O nell'intervallo [t1,t2]



Algoritmo di scheduling SJF (*Shortest Job First*)

Per risolvere i problemi dell'algoritmo FCFS:

- per ogni processo nella ready queue, viene stimata *la lunghezza del prossimo CPU-burst*
- viene schedulato il processo con il *CPU burst più corto* (*Shortest Job First*)



- si può dimostrare che questo algoritmo *ottimizza* il tempo di attesa

Algoritmo di scheduling SJF (*Shortest Job First*)

SJF può essere:

- *non pre-emptive*
- *pre-emptive*: (*Shortest Remaining Time First, SRTF*) se nella coda arriva un processo (Q) con CPU burst minore del CPU burst rimasto al processo running (P) ➔ *pre-emption*

Problema

- è difficile *stimare la lunghezza del prossimo CPU burst* di un processo (di solito, uso del passato per predire il futuro)

Stimare la lunghezza di CPU burst

Unica cosa ragionevole: stimare probabilisticamente la lunghezza in *dipendenza dai precedenti CPU burst di quel processo*

- Possibilità molto usata: *exponential averaging*

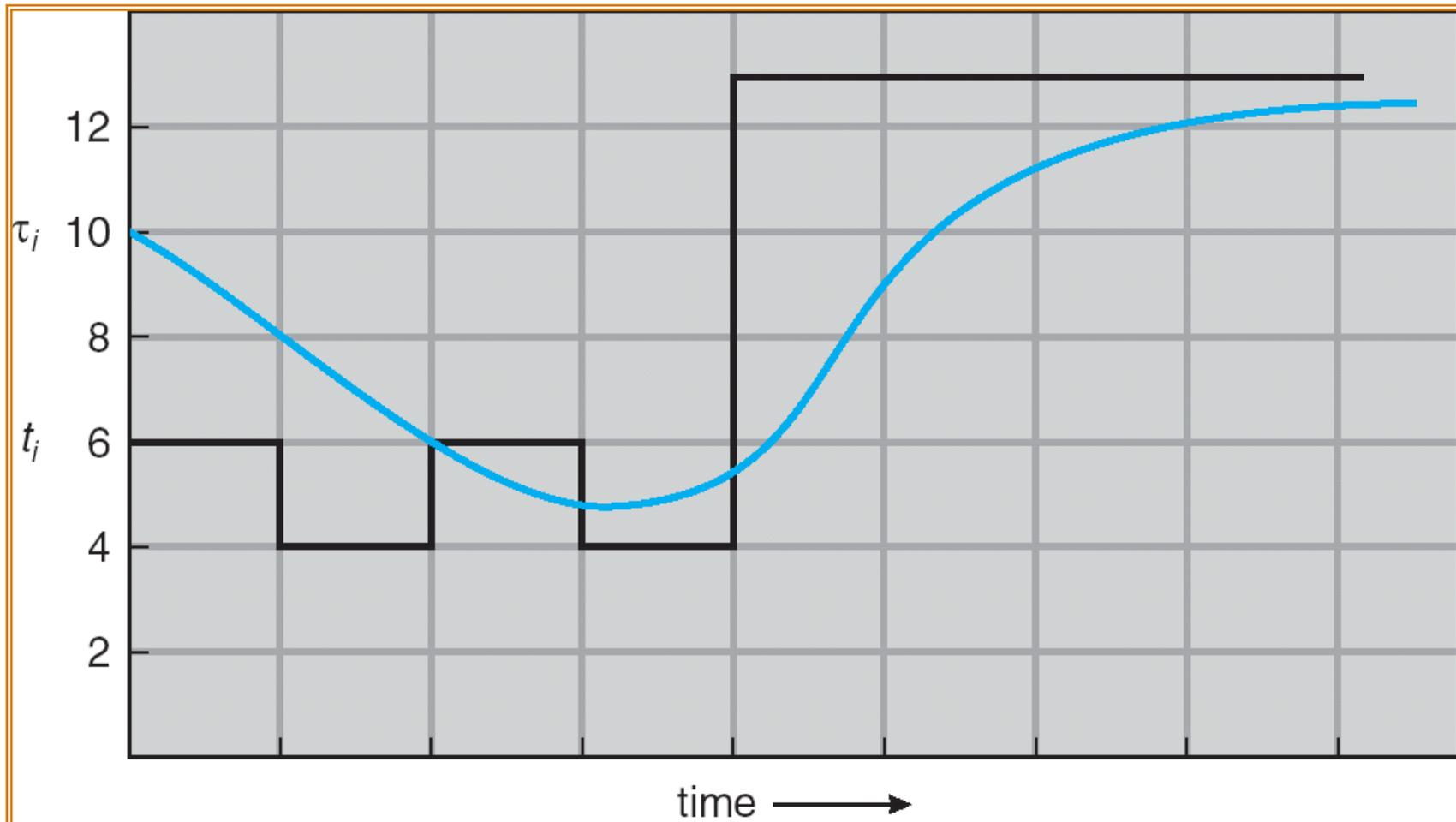
t_n = actual length of n^{th} CPU burst

τ_{n+1} = predicted value for the next CPU burst

$\alpha, 0 \leq \alpha \leq 1$

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

Stimare la lunghezza di CPU burst ($\alpha=1/2$)

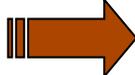


CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

SJF con exponential averaging

Sviluppando l'espressione: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

 ogni termine successivo ha meno peso del termine precedente

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - ovvero la storia recente degli attuali valori non conta
- $\alpha = 1$
 - $\tau_{n+1} = \alpha t_n$
 - ovvero conta solo l'ultimo valore reale

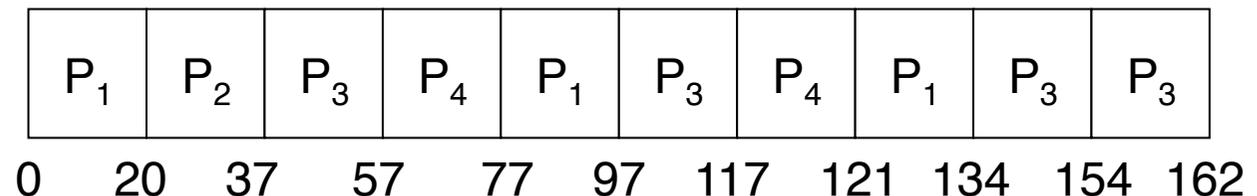
Algoritmo di scheduling *Round Robin*

È tipicamente usato in sistemi *time sharing*:

- Ready queue gestita come una *coda FIFO circolare* (FCFS)
- ad ogni processo viene allocata la CPU per un *intervallo di tempo costante Δt* (*time slice* o, *quanto di tempo*)
 - il processo usa la CPU per Δt (oppure si blocca prima)
 - allo scadere del quanto di tempo: **prelazione** della CPU e re-inserimento in coda

Es: $\Delta t = 20\text{ms}$

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	17
P_3	68
P_4	24



- RR può essere visto come un'estensione di FCFS con **pre-emption periodica**

Round Robin

- Obiettivo principale è la *minimizzazione del tempo di risposta* (adeguato per sistemi interattivi)
- Tutti i processi sono trattati allo stesso modo (*assenza di starvation*)

Problemi:

- *dimensionamento del quanto di tempo*
 - Δt *piccolo* (ma non troppo: $\Delta t \gg T_{\text{context switch}}$)
tempi di risposta ridotti, ma alta frequenza di context switch
 - Δt *grande*
overhead di context switch ridotto, ma tempi di risposta più alti
- *trattamento equo dei processi*
 - possibilità di degrado delle prestazioni del SO

Scheduling con priorità

Ad ogni processo viene assegnata una priorità:

- lo scheduler seleziona il processo pronto con *priorità massima*
- processi con *uguale priorità* vengono trattati in modo *FCFS*

Priorità possono essere definite

- **internamente**: SO attribuisce ad ogni processo una priorità in base a politiche interne
 - **esternamente**: criteri esterni al SO (es: `nice` in UNIX)
- Le priorità possono *essere costanti o variare dinamicamente*

Scheduling con priorità

Problema: *starvation* dei processi

Starvation: si verifica quando uno o più processi di priorità bassa vengono *lasciati indefinitamente nella coda dei processi pronti*, perchè vi è sempre almeno un processo pronto di priorità più alta

Soluzione: modifica dinamica della priorità dei processi

ad esempio

- la *priorità decresce* al crescere del *tempo di CPU* già utilizzato (feedback negativo o aging)
- la *priorità cresce dinamicamente* con il *tempo di attesa* del processo (feedback positivo o promotion)

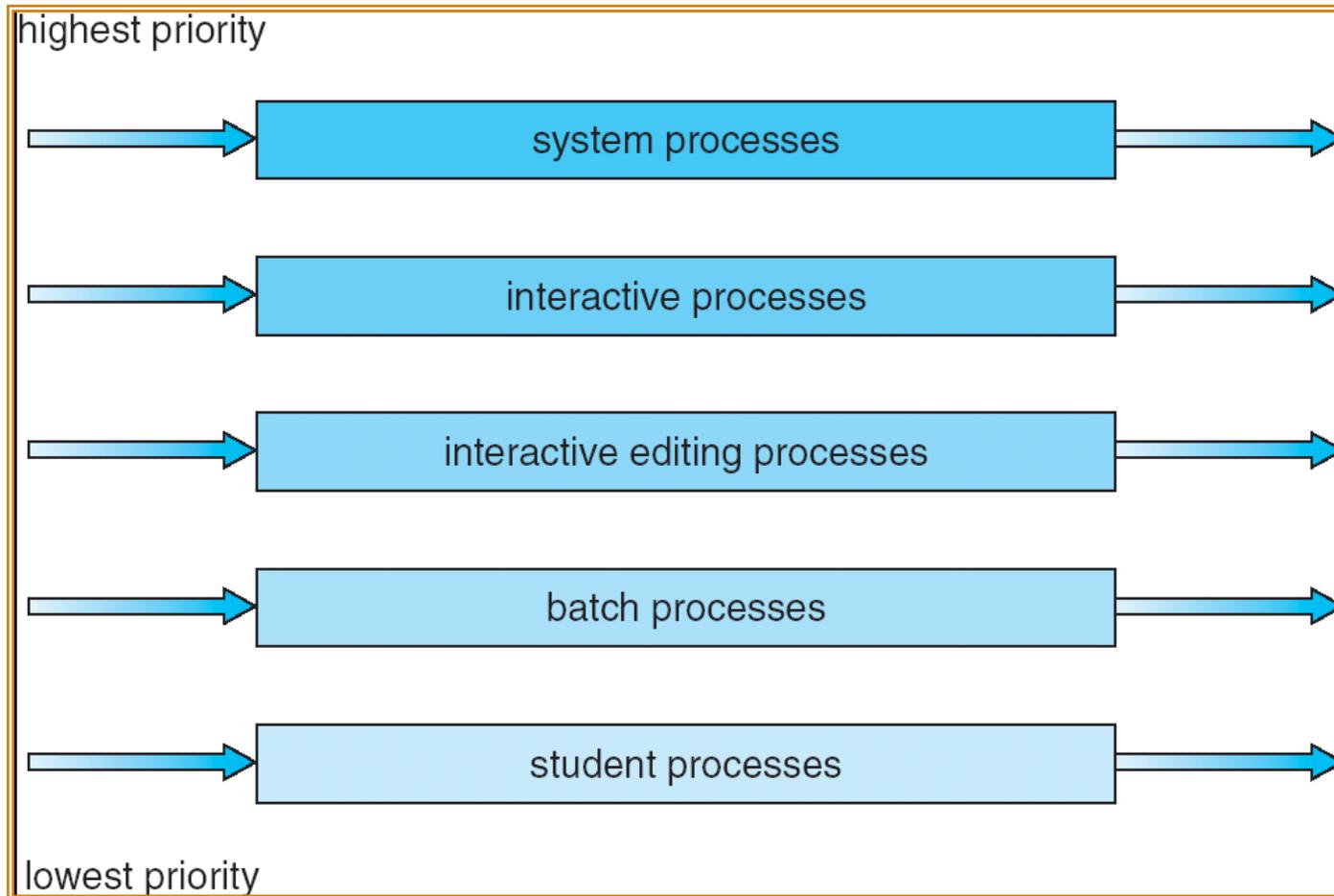
Approcci misti

Nei SO reali, *spesso si combinano diversi algoritmi di scheduling*

Esempio: *Multiple Level Feedback Queues*

- *più code*, ognuna associata a un tipo di job diverso (batch, interactive, CPU-bound, ...)
- ogni coda ha una *diversa priorità*: scheduling delle code con priorità
- ogni coda viene gestita con un algoritmo di scheduling distinto, eventualmente diverso (es. *FCFS o Round Robin*)
- i processi possono muoversi da una coda all'altra, in base alla loro storia:
 - passaggio da priorità bassa ad alta: processi in attesa da *molto* tempo (feedback positivo)
 - passaggio da priorità alta a bassa: processi che hanno già utilizzato *molto* tempo di CPU (feedback negativo)

Multi Level Feedback Queue



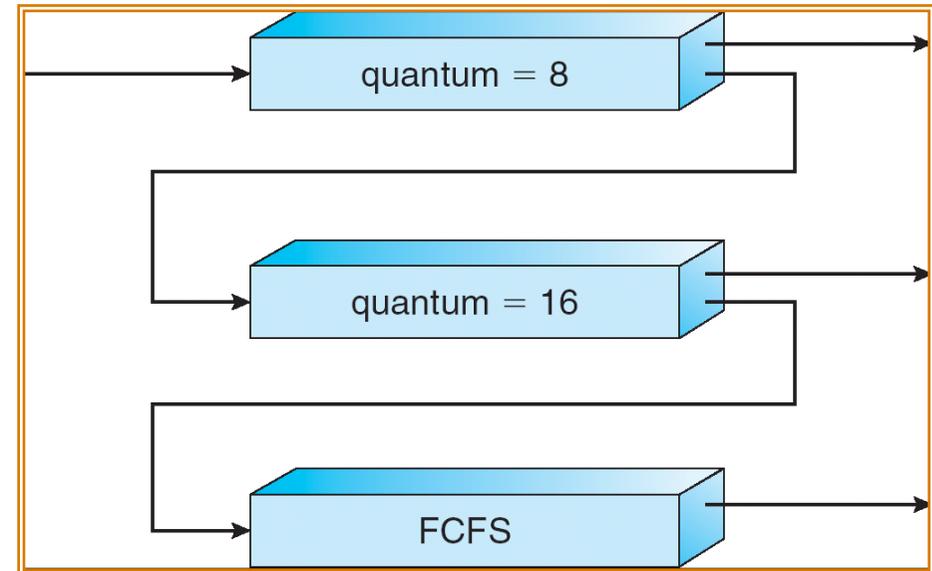
Esempio di Multi Level Feedback Queue

3 code

- ❑ Q_0 - RR con time quantum=8ms
- ❑ Q_1 - RR con time quantum=16ms
- ❑ Q_2 - FCFS

Scheduling

- ❑ Un **processo nuovo entra in Q_0** ; quando acquisisce la CPU ha 8ms per utilizzarla; se non termina nel quanto di tempo viene **spostato in Q_1**
- ❑ In Q_1 il processo è servito ancora RR e riceve 16ms di CPU; se non termina nel quanto di tempo, viene **spostato in Q_2**



Priorità elevata a processi con breve uso CPU

Scheduling in UNIX (BSD 4.3)

Obiettivo: *privilegiare i processi interattivi*

Scheduling MLFQ:

- più livelli di priorità (circa 160): la priorità è rappresentata da un intero; più grande è il suo valore, più bassa è la priorità
- Viene definito un valore di riferimento **pzero**:
 - ✓ $\text{Priorità} \geq \text{pzero}$: processi di utente ordinari
 - ✓ $\text{Priorità} < \text{pzero}$: processi di sistema (ad es. esecuzione di system call), non possono essere interrotti da segnali (`kill`)
- Ad ogni livello è associata una coda, gestita **Round Robin** (quanto di tempo: 100 ms)

Scheduling in UNIX

- **Aggiornamento dinamico delle priorità:** ad ogni secondo viene ricalcolata la priorità di ogni processo
- La priorità di un processo **decresce al crescere del tempo di CPU già utilizzato**
 - feedback negativo
 - di solito, processi interattivi usano poco la CPU: in questo modo vengono favoriti
- L'utente può influire sulla priorità: comando **nice** (ovviamente soltanto per decrescere la priorità)

Scheduling dei thread Java

Java Virtual Machine (JVM) usa *scheduling con prelazione e basato su priorità*

- FCFS fra thread con stessa priorità

JVM mette in stato di running un thread quando:

1. thread che sta usando la CPU esce dallo stato Runnable
2. un thread a priorità più alta entra nello stato Runnable

* NB: JVM non specifica se i thread hanno *quanto di tempo* oppure no

Time-Slicing

Siccome JVM *non garantisce time-slicing*, andrebbe usato il metodo `yield()`, per trasferire il controllo ad altro thread di uguale priorità:

```
while (true) {  
    // perform CPU-intensive task  
    . . .  
    thread.yield();  
}
```

Si possono assegnare valori di priorità tramite il metodo `setPriority()`