SAPIENZA
UNIVERSITÀ DI ROMA

Corso di laurea in Ingegneria dell'Informazione
Indirizzo Informatica

Reti e sistemi operativi

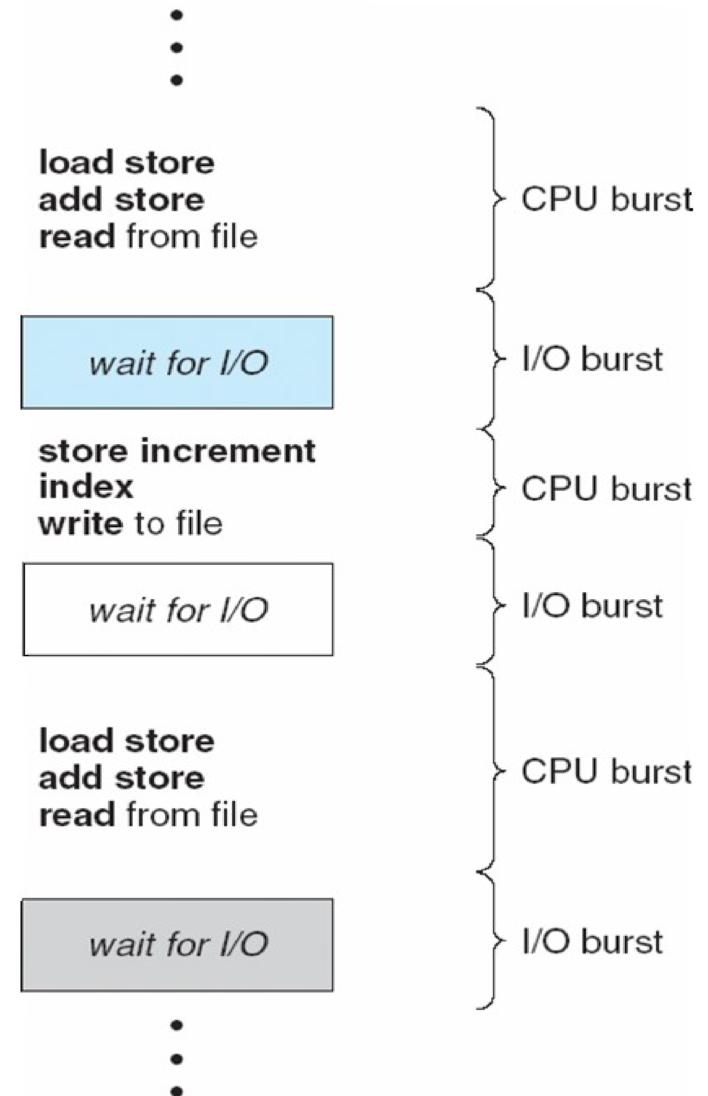
Scheduling di processi: metriche, politiche e algoritmi

Obiettivo dello scheduler

- L'obiettivo primario di uno scheduler short-term (o CPU-scheduler) è di ottenere il massimo utilizzo della CPU, ovvero evitare che la CPU resti un attesa di eventi asincroni (I/O, syscall, …) bloccando l'esecuzione di altri processi.
- Al di là di questo obiettivo, lo scheduler può essere specializzato per garantire determinate prestazioni dipendenti dal contesto, es. rapidità di reazione ad un'interazione.
- A seguito di un'interrupt e di un successivo context-switch, lo scheduler deve scegliere dalla **ready queue** di PCB (process control block) il prossimo processo da mettere in esecuzione.
  - La ready queue **non** è necessariamente una coda FIFO!

Ciclo CPU–I/O Burst (1/2)

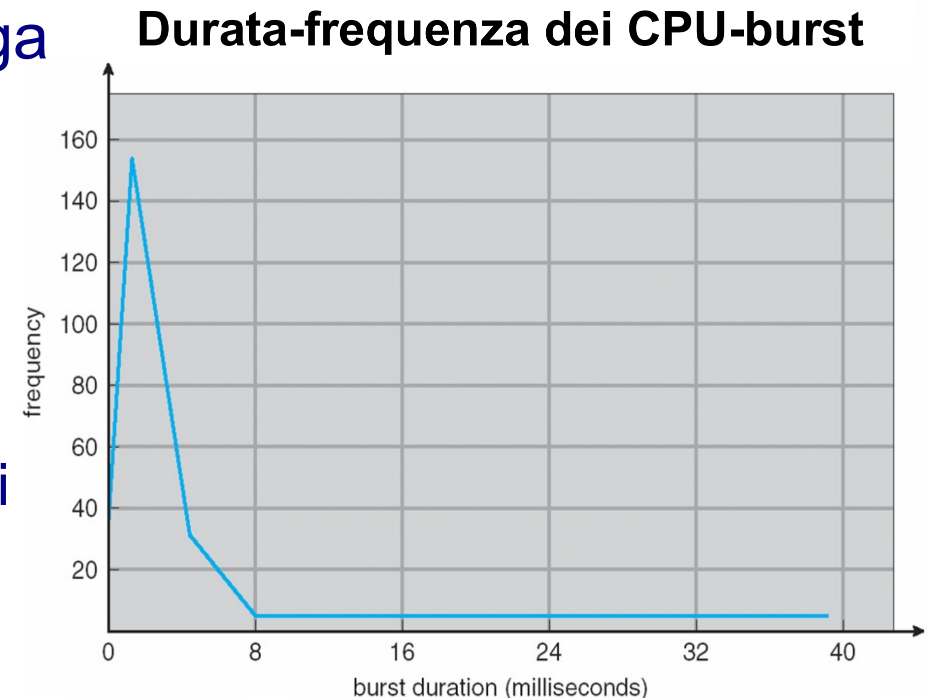
- L'esecuzione di un singolo processo si può definire attraverso un ciclo alternato di 2 stati:
  - Il processo è eseguito in CPU (**CPU-burst**)
  - Il processo attende un evento di I/O (**I/O burst**)
- L'esecuzione inizia sempre con un CPU-burst
- Attenzione: in questa definizione **non vi è corrispondenza diretta con lo scheduler**: i singoli CPU-burst possono essere "interrotti" da vari context-switch

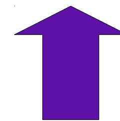


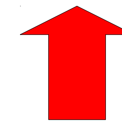
Ciclo CPU–I/O Burst (2/2)

- CPU-burst di breve durata sono solitamente (statisticamente) molto più frequenti dei CPU-burst di lunga durata.
- Programmi **I/O bound**:
  - Basso utilizzo della CPU, accesso intensivo ai file
  - Di solito composto da molti CPU-burst di breve durata
- Programmi **CPU bound**:
  - Alto utilizzo della CPU, pochi accessi ai file
  - Di solito composto da pochi CPU-burst di lunga durata

Durata-frequenza dei CPU-burst







I/O bound

CPU bound

Dispatcher

- Nelle operazioni di scheduling, il modulo che si prende carico di effettuare il context-switch (da user mode a kernel mode e viceversa) è chiamato **dispatcher**. Le operazioni base sono:
  - User mode → kernel mode: Salva il contesto
  - Kernel mode → user mode: ripristina il corretto contesto e setta l'user mode bit.
- Dispatch latency: overhead causato dalle operazioni di dispatching.

Preemptive scheduling (1/3)

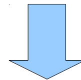
- L'intervento dello scheduler **può** essere richiesto quando un processo cambia di stato (o queue):
  1) Running state → waiting state (es. syscall per accesso I/O)
  2) Running state → ready state (es. interrupt "esterno", dovuto a I/O, timer, …)
  3) Waiting state → ready state (es. l'operazione di I/O attesa dal processo è completata)
  4) Running state → processo terminato
- Quando lo scheduler viene chiamato **solo** nelle circostanze 1) e 4) si dice **non-preemptive scheduling**, o scheduling cooperativo
- Se viene chiamato sempre, **preemptive scheduling**

Preemptive scheduling (2/3)

- In caso di non-preemptive scheduling, **è il processo in esecuzione stesso che determina il momento in cui un altro processo (quale lo decide lo scheduler) verrà messo in esecuzione**:
  - Quando esso richiede un'operazione di I/O
  - Ovviamente, quando termina
- **Caso limite**: in un S.O. con non-preemptive scheduling, se un processo in esecuzione non chiama syscall e non termina, nessun altro processo verrà mai eseguito.

Preemptive scheduling (3/3)

- Tutti i moderni S.O. implementano il preemptive scheduling, in questo caso:
  - E' necessario fornire degli **strumenti di sincronizzazione tra processi** (es. i semafori) che proteggano le comunicazioni tra di essi.
  - E' necessario che anche il S.O. stesso sia in grado di gestire in maniera sicura, coerente ed efficiente le **interruzioni che arrivano mentre sta servendo altre interruzioni**



- Le **interruzioni di solito vengono disabilitate solo per brevissimi periodi di tempo**, per evitare errati accessi concorrenti alle stesse risorse e permettere di completare operazioni critiche → disabilitare le interruzioni per troppo tempo porta ad un decadimento delle prestazioni in termini di real-time

Criteri di scheduling (1/2)

- **Utilizzo della CPU**: mantenere la CPU occupata il più possibile in operazioni “utili” (non “idle”)
- **Throughput** (~volume di produzione): numero di processi/job completati per unità di tempo
- **Tempo di turnaround**: tempo trascorso tra la messa in esecuzione e la fine del processo (o del CPU-burst)
- **Tempo di waiting**: tempo trascorso ad attendere nella ready queue.
- **Tempo di risposta**: tempo trascorso da quando un processo richiede una risorsa a quando la risposta alla richiesta **inizia** ad essere eseguita (es. significativo nei sistemi time-sharing e nei sistemi desktop)

Criteri di scheduling (2/2)

- Ovviamente nel progettare uno scheduler sarà necessario definire **quale/quali** tra questi criteri ottimizzare
- Di solito si tende a minimizzare/massimizzare la **media,** il **valore massimo** oppure la **varianza** di uno o più criteri, es.
  - Massimizzare l'utilizzo della CPU
  - Minimizzare l'average waiting time
  - Minimizzare il tempo massimo di risposta
  - Minimizzare la varianza del tempo di risposta (**predicibilità**)
- Per presentare gli algoritmi di scheduling verrà associato ad ogni processo il **prossimo CPU-burst**

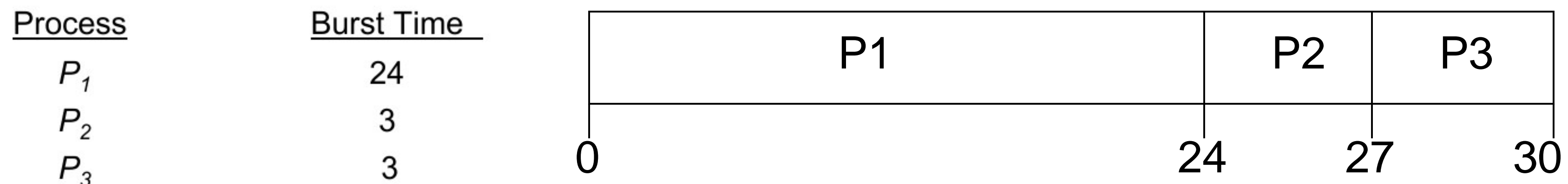
First-Come, First-Served Scheduling (1/2)

- I processi sono schedulati nell'ordine in cui arrivano nella ready-queue
- Essa è implementata come una FIFO queue: viene messo in esecuzione il processo in testa, i nuovi processi che entrano nella queue vengono messi in coda
  - Per questo lo scheduler FCFS viene a volte chiamato semplicemente FIFO
- E' uno scheduler è non-preemptive: un processo resta in esecuzione finché non richiede esplicitamente un'operazione di I/O oppure non termina → problematico con i processi I/O bound

First-Come, First-Served Scheduling (2/2)

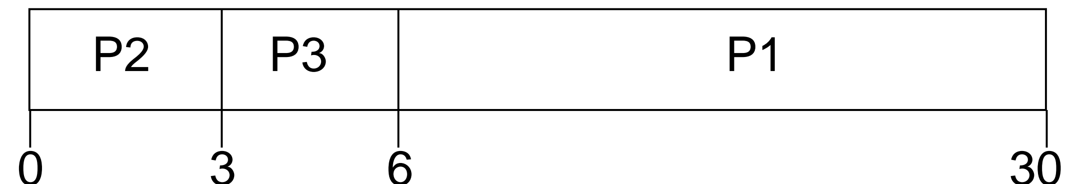
- Si supponga che tre processi arrivino nella ready queue nell'ordine P1, P2, P3:

Diagramma di Gantt



Process	Burst Time
P_1	24
P_2	3
P_3	3

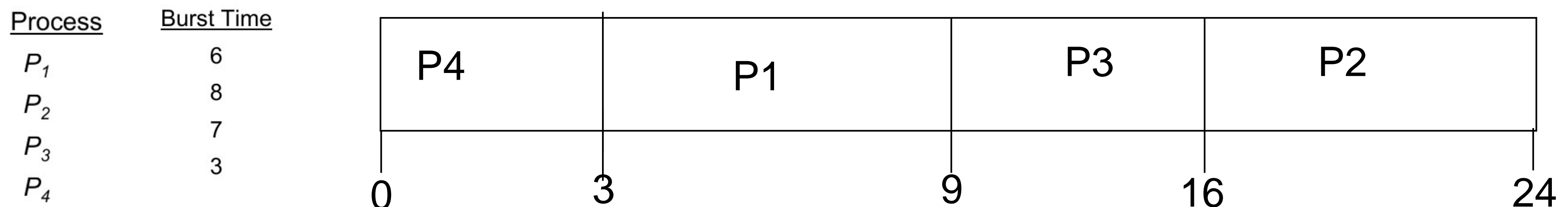
- Average waiting time: (0 + 24 + 27)/3 = 17
- Se l'ordine di arrivo fosse P2, P3, P1:



- Average waiting time: (6 + 0 + 3)/3 = 3
- L'algoritmo FCFS soffre dell'**effetto convoy**:
  - I processi rapidi attendono a lungo la conclusione di lunghi processi CPU bound

Shortest-Job-First Scheduling (1/5)

- Nello SJF scheduling, viene sempre messo in esecuzione il processo con il prossimo CPU-burst più breve tra tutti i processi presenti in quell'istante in ready queue, es.:



Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

- SJB è **ottimo per quel che riguarda l'average waiting time**:
  - Fornisce sempre l'average waiting time minimo per un dato set di processi

Shortest-Job-First Scheduling (2/5)

- **Problema SJF**: come possiamo conoscere a priori la lunghezza del prossimo CPU-burst?
  - L'utente fornisce una stima (sistemi batch)
  - Si calcola una stima basata sulla “storia” dei CPU-burst di un determinato processo
- *Media esponenziale*:

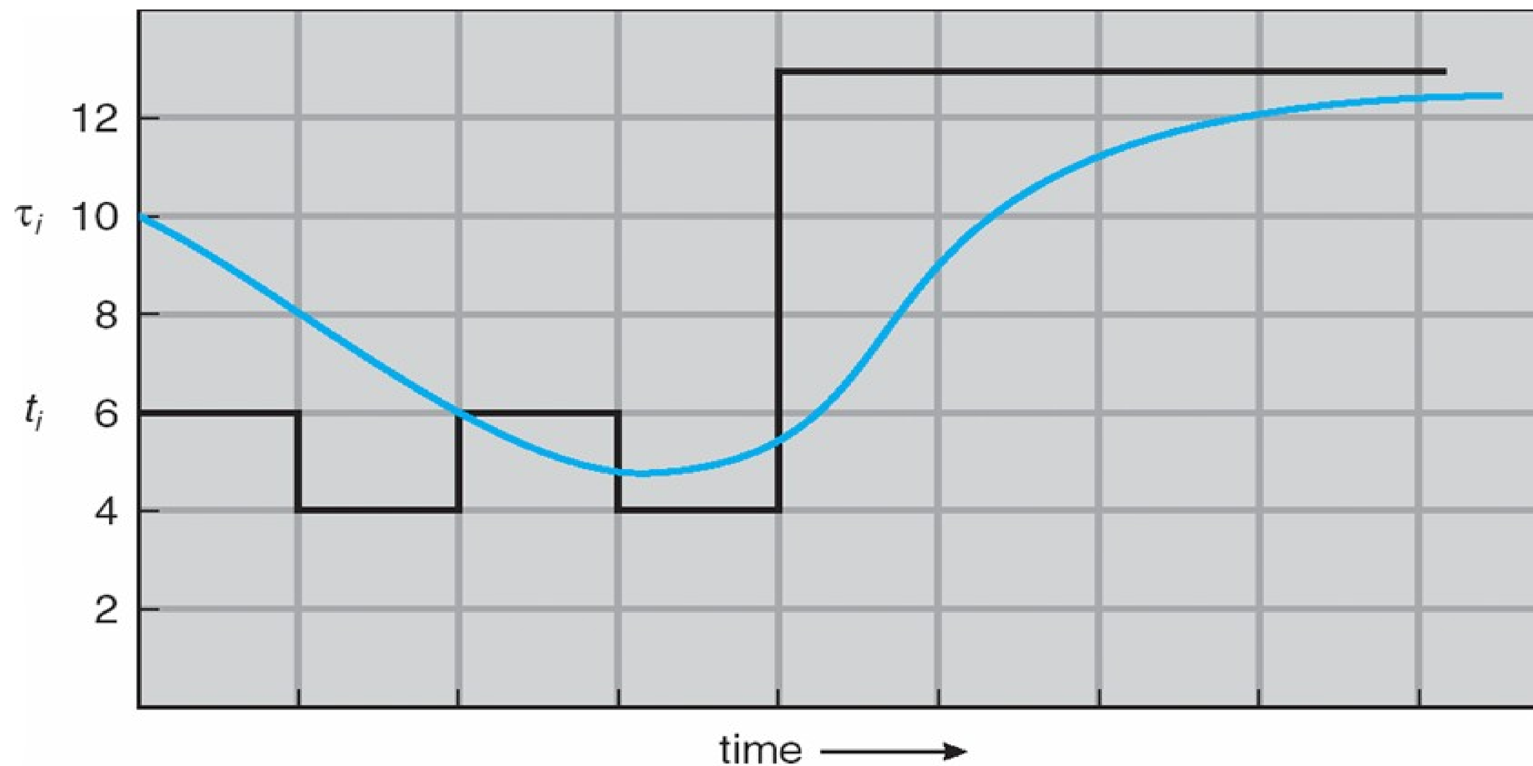
$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

con:

- t_n : durata misurata n-esimo CPU-burst
- τ_{n+1} : durata stimata (n+1)-esimo CPU-burst
- α : peso di solito, $0 \le \alpha \le 1$, ragionevole 0.5

Shortest-Job-First Scheduling (3/5)

- Esempio di predizione del prossimo CPU-burst con media esponenziale



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

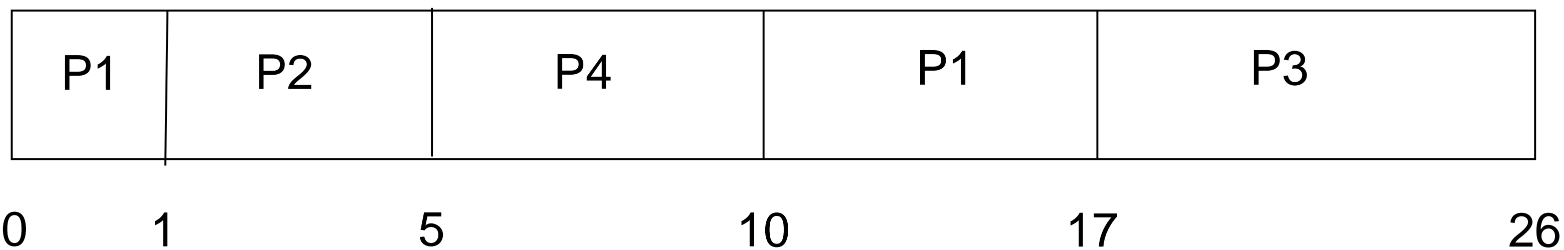
Shortest-Job-First Scheduling (4/5)

- Lo SJF scheduling può essere:
  - **Non-preemptive**: Si attende ogni volta la terminazione del processo attualmente in esecuzione, anche se nel frattempo (attraverso un interrupt) è arrivato in ready queue un processo con minore durata del prossimo CPU-burst
  - **Preemptive** (anche chiamato **Shortest-remaining-time-first**): viceversa, se è arrivato un processo con minore durata del prossimo CPU-burst rispetto a **quanto manca** al processo corrente in esecuzione per terminare il proprio CPU-burst, quest'ultimo viene bloccato e messo in esecuzione il nuovo arrivato.

Shortest-Job-First Scheduling (5/5)

- Esempio di preemptive SJF → è necessario definire anche i **tempi di arrivo in ready queue**.

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



- AWT = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec
- Caso non-preemptive AWT = **7.75 msec**

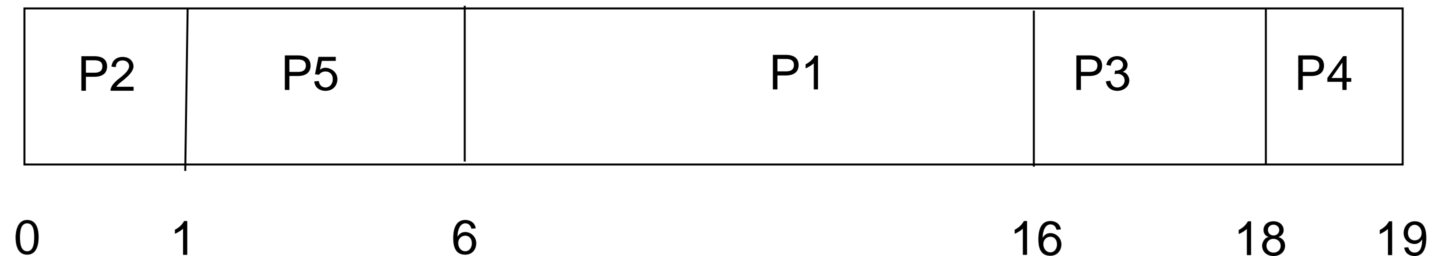
Priority Scheduling (1/3)

- Nello SJF scheduling, i processi con prossimo CPU-burst più breve hanno la priorità sui processi con CPU-burst dalla durata maggiore → esso è a tutti gli effetti un caso particolare di priority scheduling, ove la priorità è l'inverso della durata del prossimo CPU-burst.
- **Priorità** → valore associato dall'utente o dal sistema operativo utilizzando qualche metrica (es., numero di file aperti, utente che ha lanciato il processo, ...)
- Priorità maggiore → valore max o min, **dipende dalla convenzione**

Priority Scheduling (2/3)

- Esempio di PS → è necessario definire un valore di priorità per processo (nell'esempio minore è il valore, maggiore è la priorità)

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



  - AWT: 8.2 msec

Priority Scheduling (3/3)

- **Problema PS**: si prenda un processo con priorità bassa, se in ready queue c'è sempre un processo con priorità maggiore il primo non verrà **mai** eseguito → **starvation**.
- Possibile soluzione → **aging**: la priorità viene gradualmente (spesso temporaneamente) incrementata se il processo staziona per molto tempo nella ready queue senza mai venire eseguito.

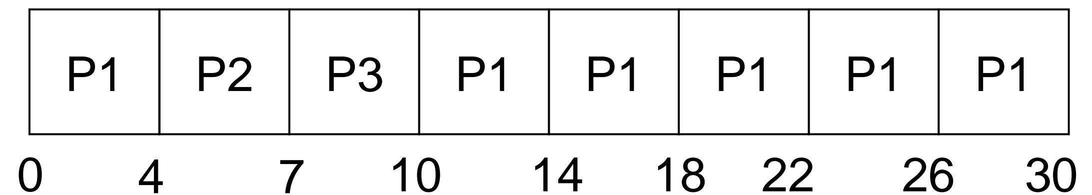
Round Robin scheduling (1/3)

- **Scheduler “democratico”**: ogni processo viene eseguito per un certo periodo di tempo ***q*** (**quanto**) fisso (es. 10-1000 msec) → scheduler intrinsecamente preemptive, particolarmente indicato per sistemi time-shared.
- Trascorso tale quanto, oppure il CPU-burst del processo corrente termina prima (es. viene richiesto un intervento di I/O), il processo viene tolto dall'esecuzione e sostituito con il prossimo processo in testa ad una FIFO queue (in questo senso, è simile allo FCFS scheduling).
- Il raggiungimento di un quanto di tempo dall'ultima schedulazione viene segnalato al sistema operativo da un **interrupt di un timer che raggiunge il valore 0** → il sistema operativo reimposterà tale timer al valore q.

Round Robin scheduling (2/3)

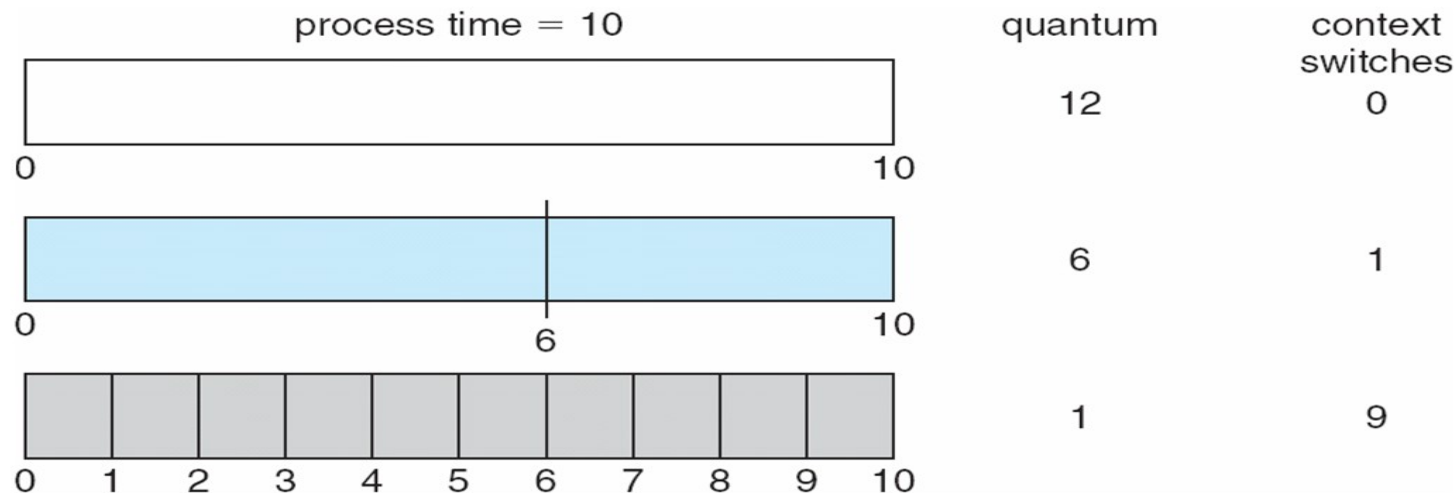
- Se la ready queue contiene n processi, nessun processo attende più di (n-1)*q unità di tempo prima di essere messo in esecuzione.
- Al crescere del quanto q, lo scheduler RR approssima lo scheduler FCFS
- Esempio di RR (q = 4 msec):

Process	Burst Time
P_1	24
P_2	3
P_3	3



Round Robin scheduling (3/3)

- Performance: **peggiore turnaround medio** (tempo medio di completamento dei CPU-burst) rispetto a SJF, ma **minor tempo di risposta** (miglior responsività)
- Attenzione: la durata q del quanto deve essere molto maggiore rispetto al tempo (overhead) di contex switch

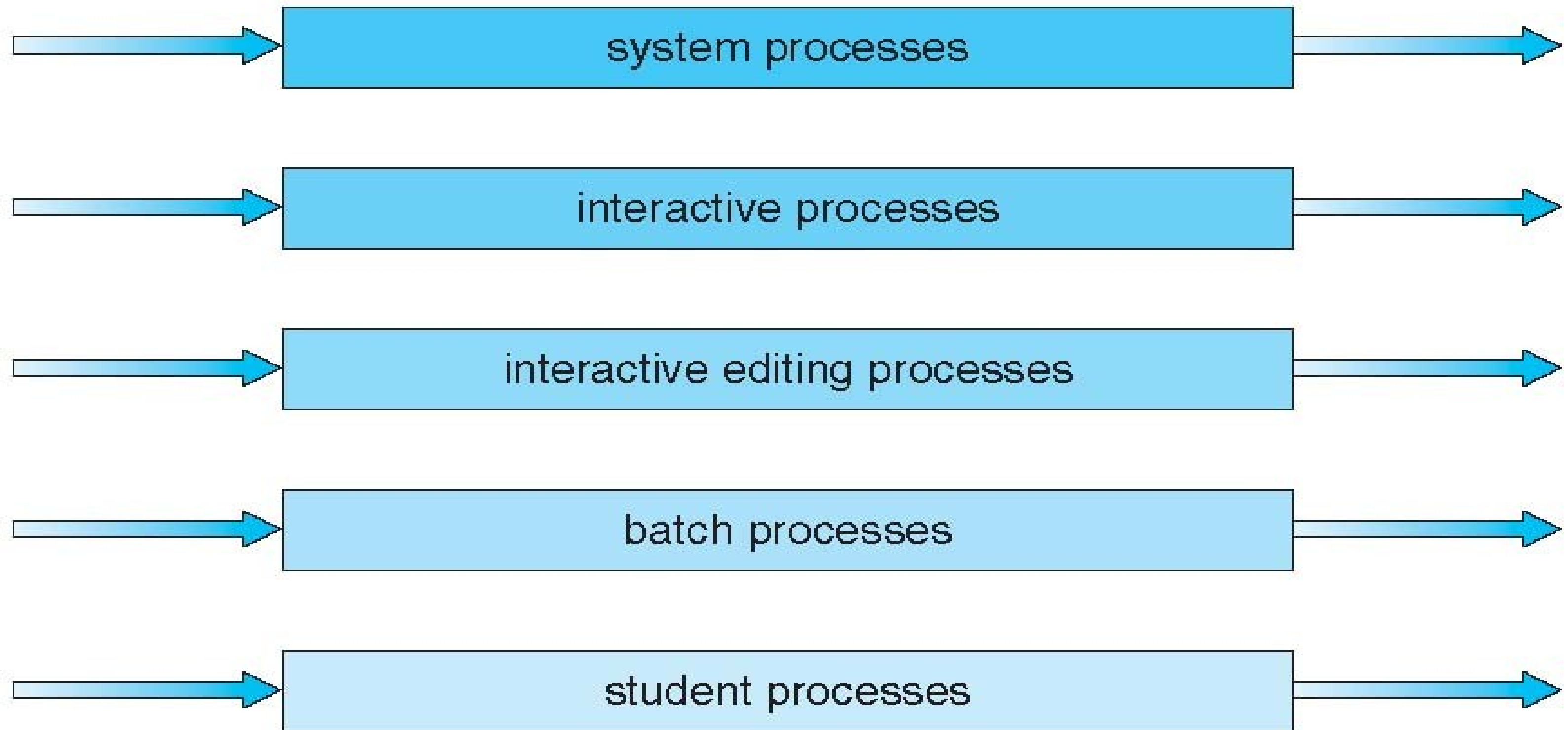


- Regola generale: l'80% dei CPU-bursts dovrebbe avere durata minore di q

Multilevel Queue scheduling (1/3)

- Idea di base: **fornire priorità distinte non ai singoli processi, ma a classi (sottoinsiemi) di processi**, organizzati in queue distinte ognuna con priorità ben definita e **non mutabile**, es.:
  - Processi batch (in *background*),ad esempio processi di calcolo scientifico, servizi del sistema non prioritari → queue a bassa priorità
  - Processi interattivi (in *foreground*), ad esempio processi che ricevono comandi dall'utente → queue ad alta priorità
- Oltre alla priorità, ad ogni queue è assegnato un algoritmo di scheduling, es.:
  - Alta priorità → RR
  - Bassa priorità → FCFS

Multilevel Queue scheduling (2/3)



Multilevel Queue scheduling (3/3)

- **Problema MLQ**: processi appartenenti a queue a bassa priorità devono attendere che tutte le queue a priorità maggiore siano vuote → possibilità di **starvation**.
- Possibile soluzione: ad ogni queue è assegnato un **time slice** , ovvero una certa percentuale di tempo riservato in CPU, "proporzionale" alla priorità della queue, es.:
  - 80% per i processi batch con scheduler RR
  - 20% per i processi interattivi con scheduler FCFS

Multilevel Feedback Queue (1/3)

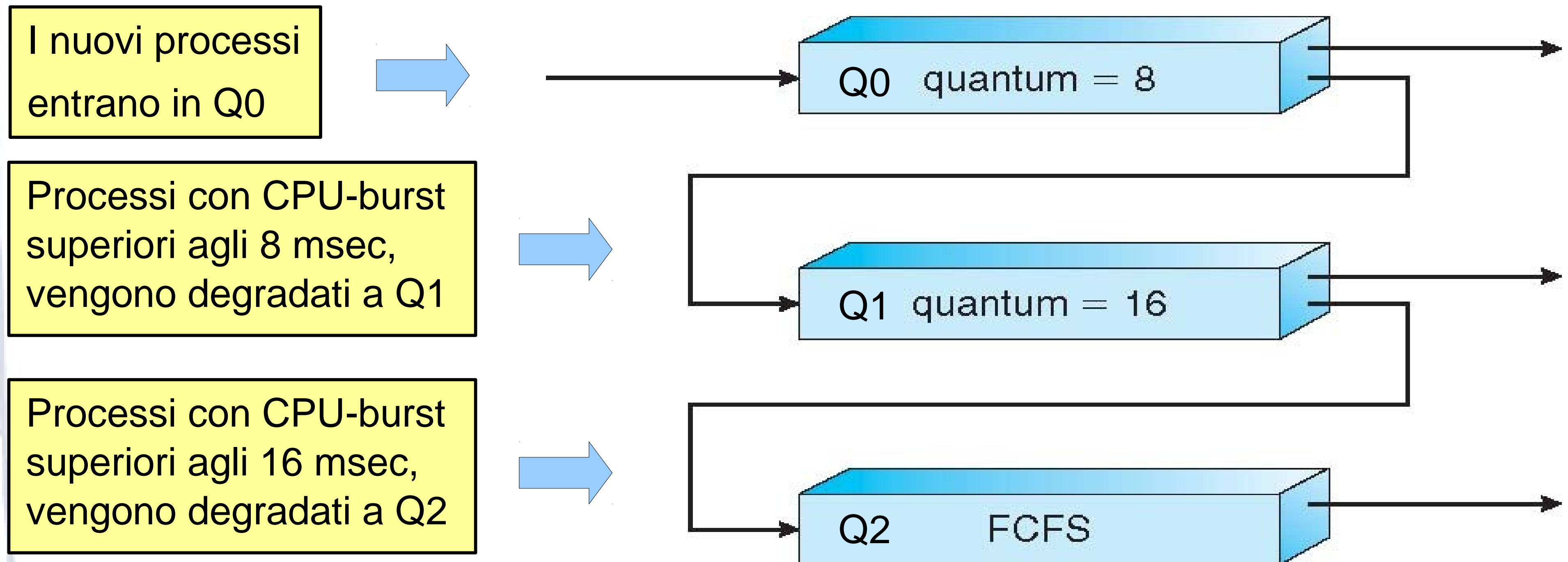
- Un'altra soluzione al problema dello starvation per le MLQ è quella di permette ai processi di “muoversi” tra le multilevel queue → **MLFQ**.
- L'idea ad esempio è di applicare la strategia **aging**: processi appartenenti a queue a bassa priorità vengono “promossi” (spesso solo temporaneamente) a queue a priorità maggiore se non vengono eseguiti da molto tempo.
- Viceversa, processi con CPU-burst molto lunghi potrebbero essere “degradati” a queue con minore priorità

Multilevel Feedback Queue (2/3)

- Una strategia di MLFQ permette grande flessibilità nel progettare lo scheduler. E' necessario definire:
  - Numero di queues
  - Algoritmo di scheduling per ognuna di esse
  - Metodo di promozione dei processi a queue a priorità maggiore, e viceversa
  - A quale queue deve appartenere un processo quando viene lanciato
  - Quali processi **non** devono essere mai spostati dalla queue

Multilevel Feedback Queue (3/3)

- Esempio di MLFQ che utilizza 3 queue:
  - Q0 - scheduler RR con q = 8 msec
  - Q1 - scheduler RR con q = 16 msec
  - Q2 - scheduler FCFS



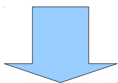
Caso di studio: MLQ in Linux (1/2)

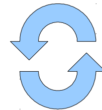
IMPORTANTE:
Per abilitare la schedulazione real-time, aggiungere la seguente riga al file /etc/security/limits.conf:

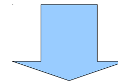
<nome_utente>   -   rtprio   99

(per editare tale file, è necessario avere credenziali da amministratore, ad esempio avviare l'editor da terminale preceduto dal comando sudo, es. comando **sudo gedit** (verrà chiesta la password)

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sched.h>

typedef struct
{
  int id;
  unsigned long sleep_ms;
}ThreadParams;
```



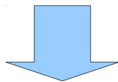


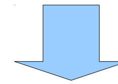


```
void *workAndSleep(void * par)
{
  ThreadParams *th_param = (ThreadParams*)par;

  usleep(500000);
  while (1)
  {
    printf("THREAD %d : Running \n", th_param->id);
    /* Il thread entra in un lungo CPU-burst, saturando la CPU */
    int i=0;
    for(i=0; i<100000000; i++)
    {
      /* Ogni tanto notifica a video dov'è arrivato
       * (chiamando una syscall) */
      if (!(i%20000000))
        printf("THREAD %d : i = %d\n",th_param->id,i);
    }
    printf("THREAD %d : Sleep\n\n",th_param->id);
    /* Si ferma per qualche istante (chiamando una syscall) */
    usleep( th_param->sleep_ms * 1000 );
  }
  pthread_exit(NULL);
}
```

Caso di studio: MLQ in Linux (2/2)

```
int main(int argc,char **argv)
{
  pthread_t thread1, thread2, thread3, thread4;
  ThreadParams tp1 = {1,3000}, tp2 = {2,3000},
               tp3 = {3,1000}, tp4 = {4,1000};

  pthread_attr_t attr;
  struct sched_param param;
  memset(&param,0,sizeof(param));

  pthread_attr_init(&attr);

  /* Assicuro che il thread possa essere sechedulato
   * con gli scheduler realtime FIFO e RR */
  pthread_attr_setscope(&attr,
          PTHREAD_SCOPE_SYSTEM);
  pthread_attr_setinheritsched(&attr,
          PTHREAD_EXPLICIT_SCHED);

  /* Impongo che thread siano eseguiti
   * su di una sola CPU */
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(0, &cpuset);
  pthread_attr_setaffinity_np(&attr,
          sizeof(cpu_set_t), &cpuset);
```





```
  /* Setto priorità 99 e tipo di scheduler FIFO
   * (utilizzate per i thread 1 e 2) */
  param.sched_priority = 99;
  pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
  pthread_attr_setschedparam(&attr, &param);
  /* Lancio i thread 1 e 2 */
  pthread_create(&thread1, &attr, workAndSleep, &tp1);
  pthread_create(&thread2, &attr, workAndSleep, &tp2);

  /* Setto priorità 98 e il tipo di scheduler RR
   * (utilizzate per i thread 3 e 4) */
  param.sched_priority = 98;
  pthread_attr_setschedpolicy(&attr, SCHED_RR);
  pthread_attr_setschedparam(&attr, &param);
  /* Lancio i thread 3 e 4 */
  pthread_create(&thread3, &attr, workAndSleep, &tp3);
  pthread_create(&thread4, &attr, workAndSleep, &tp4);

  /* Attendo la terminazione di tutti i thread
   * (in realtà i thread non terminano)*/
  pthread_join(thread1,NULL);
  pthread_join(thread2,NULL);
  pthread_join(thread3,NULL);
  pthread_join(thread4,NULL);

  pthread_exit(NULL);
}
```