# 5. Scheduling della CPU

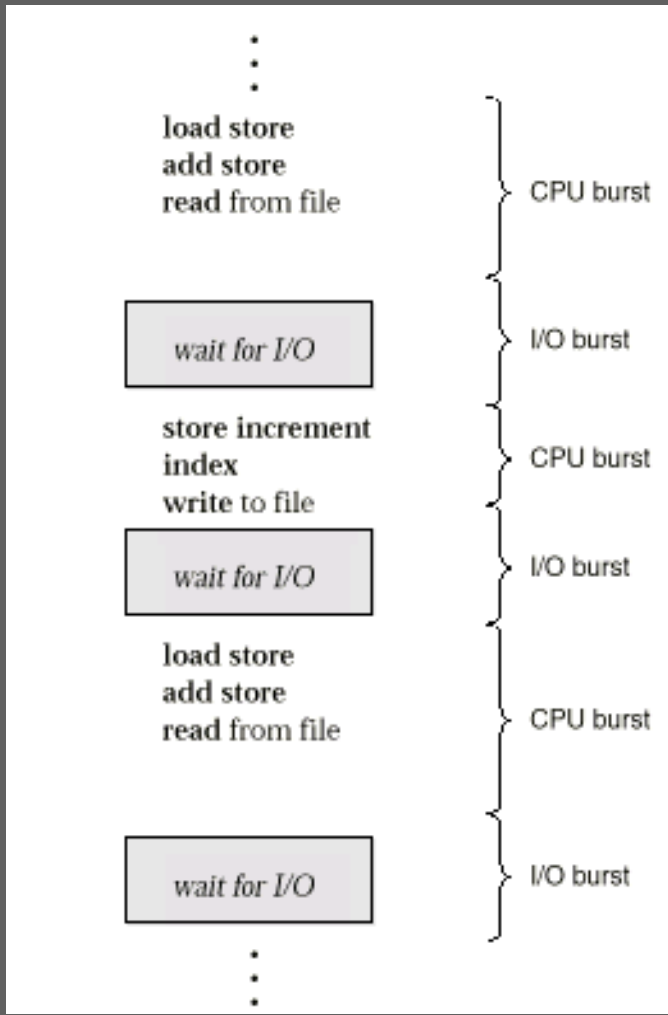
- Il multitasking e il time sharing cercano di massimizzare l'utilizzo della CPU.
- Per questo, il progettista del SO deve stabilire delle regole per decidere, quando un processo lascia la CPU, quale sarà il prossimo processo ad entrare in esecuzione.
- L'insieme di queste regole e la loro applicazione pratica prendono il nome di **Scheduling della CPU**

# 5.1.1 Fasi di elaborazione e di I/O

- Nella vita di un processo si alternano fasi di **uso di CPU (burst di CPU)** e fasi di attesa per il completamento di **operazioni di I/O (Burst di I/O)** (fig. 5.1). Distinguiamo fra:

- Processi **CPU-bound:** usano molto la CPU e poco i dispositivi di I/O (ad esempio un compilatore)

- Processi **I/O-bound:** usano poco la CPU e molto i dispositivi di I/O (ad esempio un editor o un browser)



# 5.1.2. Lo scheduler della CPU

- Consideriamo la situazione in cui un processo utente abbandona la CPU. Il SO si sveglia e deve decidere a quale, fra i processi in Coda di Ready (processi che diciamo essere **ready to run**), assegnare la CPU.

- Questa operazione è detta **Scheduling della CPU**, e viene fatta dal modulo del SO detto **scheduler**.

- Quando interviene lo scheduler per scegliere il successivo processo a cui assegnare la CPU? Possiamo considerare quattro situazioni, che ci porteranno a definire i concetti di *scheduling con* e *senza diritto di prelazione*.

# 5.1.3 Scheduling con e senza prelazione

1. Il processo che sta usando la CPU passa volontariamente dallo stato di running allo stato di  waiting (ad esempio per compiere una operazione di I/O).

2. Il processo che sta usando la CPU termina.

- In questi due casi, lo scheduler deve prendere un processo dalla coda di ready e mandarlo in esecuzione (in modo da non lasciare la CPU idle)

- **Un sistema operativo che intervenga nei casi 1 e 2 è sufficiente per implementare il multi-tasking.**

# 5.1.3 Scheduling con e senza prelazione

- Ma che succede se viene mandato in esecuzione un programma che contiene una istruzione del tipo:

```
for(; ;)
    printf("AHAHAHAHAH LA CPU E' MIIIIIAAA!!!");
```

- Il SO deve poter intervenire in modo da evitare che un processo si impossessi permanentemente della CPU, e quindi...

# 5.1.3 Scheduling con e senza prelazione

3. Il processo che sta usando la CPU **viene obbligato a passare** dallo stato di running allo stato di ready.

- Notate che il passaggio non avviene *volontariamente*: perché mai il processo dovrebbe lasciare la CPU a favore di un altro processo?

- proprio per evitare situazioni come quella della slide precedente, nei sistemi time sharing il SO non perde mai completamente il controllo del sistema.

# 5.1.3 Scheduling con e senza prelazione

- Abbiamo visto (sezione 1.4.3) che  il SO mantiene il controllo della CPU attraverso un timer hardware

- Quindi anche se è in esecuzione un processo che non ha nessuna intenzione di abbandonare la CPU, allo scadere del timer il controllo della CPU verrà restituito al SO, che sceglierà dalla RQ un processo da mandare in esecuzione.

- Più tardi, (anche solo *qualche decimo di secondo* più tardi...) il processo a cui la CPU era stata tolta verrà di nuovo scelto per usare la CPU, e potrà riprendere l'esecuzione del suo codice dal punto in cui era stato fermato.

# 5.1.3 Scheduling con e senza prelazione

4. Un processo Px entra in coda di ready arrivando da una coda di wait oppure perché è appena stato lanciato. Perché il SO interviene in questo caso? Per due ragioni:
   - **primo**: i processi non si spostano da soli da una coda all'altra: i loro PCB sono gestiti dal SO che (ad esempio) , si accorge del completamento dell'operazione di I/O per cui Px era in attesa e interviene per spostare (il PCB di) Px dalla coda di wait alla coda di ready
   - **secondo**: se Px è più "importante" del processo in esecuzione, il SO toglie quest'ultimo dalla CPU e manda in esecuzione Px

# 5.1.3 Scheduling con e senza prelazione

- Quando un sistema interviene solo nei casi 1 e 2 si parla di:
  **Scheduling senza (diritto di) prelazione**
  (in inglese: **Non-preemptive scheduling)**

- Quando il sistema interviene anche nei casi 3 e 4 si parla di
  **Scheduling con (diritto di) prelazione**
  (in inglese: **preemptive scheduling)**

# 5.1.3 Scheduling con e senza prelazione

- Chiaramente, **lo scheduling preemptive è più sicuro** per gli utenti, ma la sua implementazione richiede un sistema operativo e un'architettura hardware più sofisticati (ad esempio un timer dedicato)
- I moderni sistemi operativi "general purpose" usano tutti una qualche variante di scheduling preemptive.
- per applicazioni specifiche, può però essere sufficiente uno scheduling non-preemptive, il che permette di usare SO più semplici e leggeri.

# 5.1.3 Scheduling con e senza prelazione

- Lo scheduling preemptive può inoltre portare a situazioni che è necessario gestire con attenzione.
- Consideriamo il caso di un processo che deve compiere una operazione di I/O, e quindi chiama l'opportuna system call.
- Il controllo viene trasferito al SO che incomincia ad effettuare l'operazione per conto del processo utente. Nel frattempo scade il timer e il controllo viene trasferito ad un'altra porzione del codice del SO.

# 5.1.3 Scheduling con e senza prelazione

- Quindi, una operazione potenzialmente delicata (altrimenti non verrebbe fatta dal SO) è stata interrotta a metà: forse alcune strutture dati sono in una situazione inconsistente, perché il codice della system call di I/O non ha finito di aggiornarle.
- Che succede se ora la CPU viene data ad un altro processo utente che tenta di usare lo stesso dispositivo di I/O che stava usando il processo utente precedente?
- Domanda: quale semplice soluzione si può adottare in tali casi?

# 5.1.3 Scheduling con e senza prelazione

- Mentre Unix è stato sviluppato quasi da subito come sistema di tipo preemptive, nei sistemi Microsoft la preemption è stata introdotta solo con Windows 95.

- Questo è giustificato dal fatto che i sistemi operativi di questa famiglia, a partire dall'MS-Dos, sono nati come sistemi mono utente, per cui era sufficiente un sistema operativo più semplice.

- E inoltre i primi sistemi per PC giravano su CPU semplici ed economiche, non dotate dei supporti hardware necessari ad implementare un SO preemptive.

# 5.1.4 Il Dispatcher

- Quando lo scheduler ha scelto il processo a cui assegnare la CPU interviene un altro modulo del SO, il **dispatcher**, che:
  - effettua l'operazione di context switch
  - effettua il passaggio del sistema in user mode
  - Posiziona il PC della CPU alla corretta locazione del programma da far ripartire

- si dice **Dispatch latency** il tempo impiegato per effettuare la commutazione da un processo ad un altro

# 5.2 Criteri di Scheduling

- Come abbiamo visto, lo scheduler della CPU interviene per assicurare il corretto funzionamento del sistema.
- Ma quando lo scheduler deve mandare in esecuzione un processo, quale criterio usa per scegliere tra tutti i processi presenti in coda di ready?
- Si possono prendere in considerazione diversi obiettivi:

# 5.2 Criteri di Scheduling

- Massimizzare l'**utilizzo** della CPU nell'unità di tempo, anche se ovviamente questo dipende dal carico…
- Massimizzare il **Throughput,** ossia la **produttività** del sistema: il numero di processi completati in media in una certa unità di tempo)
- minimizzare il **Tempo di risposta**: tempo che intercorre da quando si avvia un processo a quando questo incomincia effettivamente ad eseguire: è soprattutto importante per i sistemi interattivi (perché?)

# 5.2 Criteri di Scheduling

- minimizzare il **Turnaround time**: ossia il tempo (medio) di **completamento** di un processo, da quando entra in coda di ready per la prima volta a quando termina.

- minimizzare il **waiting time**, ossia il **tempo di attesa**: la somma del tempo passato dal processo in Ready Queue
  - cioé: il processo è pronto per eseguire il suo codice ma la CPU è occupata da un altro processo

- Supponendo che un processo non finisca mai in una coda di wait,
  **che relazione c'è tra waiting time e turnaround time?**

# 5.3 Algoritmi di Scheduling

- **First Come, First Served** (scheduling per ordine di arrivo)
- **Shortest Job First** (scheduling per brevità)
- **Priority scheduling** (scheduling per priorità)
- **Round Robin** (scheduling circolare)
- **Multilevel Queue** (scheduling a code multiple)
- **Multilevel Feedback Queue** (scheduling a code multiple con retroazione)

- N.B. Nel seguito considereremo processi con **un unico burst di CPU** e nessun burst di I/O e con una durata espressa in generiche "unità di tempo". Questo semplifica la comprensione degli algoritmi senza perdita di generalità.

# 5.3.1 First Come First Served (FCFS)

- E’ facile da implementare: gestisce la ready queue in modo FIFO:
  - il PCB di un processo che entra in RQ viene messo in fondo alla coda
  - quando la CPU si libera viene assegnata al processo il cui PCB è in testa alla coda FIFO

- FCFS è un algoritmo **non preemptive, per cui non va bene per i sistemi time-sharing**
- Inoltre, con FCFS Il **tempo di attesa** del completamento di un processo è spesso lungo.

# 5.3.1 FCFS: esempio

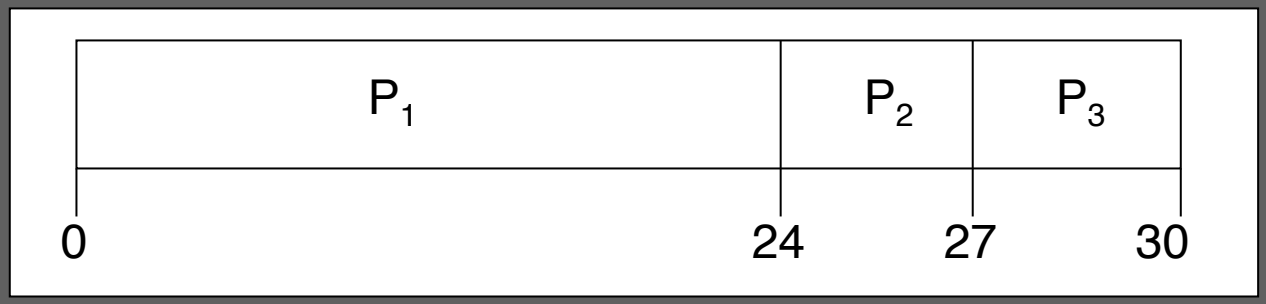
- Consideriamo tre processi che arrivano assieme al tempo t=0, e che entrano in CPU nell'ordine $P_1$, $P_2$, $P_3$.

- Come abbiamo già detto, i tre processi eseguono per un unico burst di CPU, e poi terminano.

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

# 5.3.1 FCFS: esempio

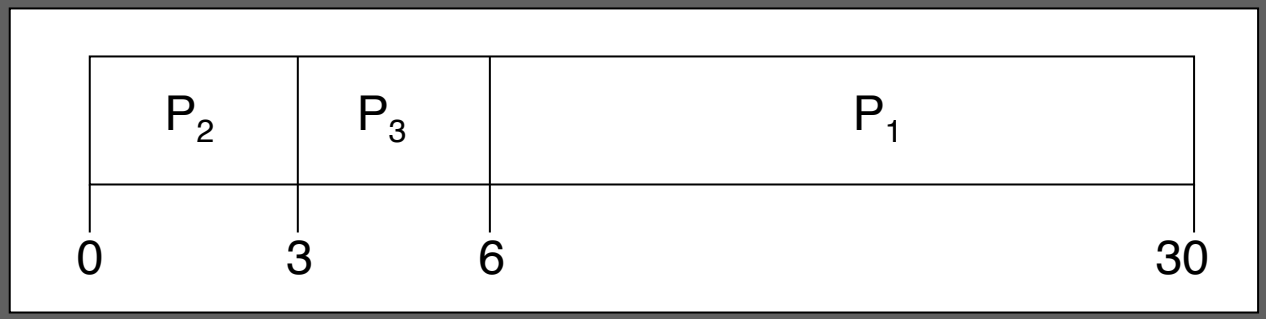
- Usiamo un **diagramma di Gantt** per rappresentare questa situazione:




- E facciamo alcuni calcoli:
- Tempi di attesa: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Tempo medio di attesa: $(0 + 24 + 27)/3 = \mathbf{17}$
- Si dice che si è verificato un **"effetto convoglio"**: i job più corti si sono dovuti accodare a quello lungo.

# 5.3.1 FCFS: esempio

- Se invece supponiamo che l'ordine di arrivo sia: $P_2$ , $P_3$ , $P_1$
  Il diagramma di Gantt per questa sequenza sarà:




- Tempi di attesa: $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Tempo medio di attesa: $(6 + 0 + 3)/3 = 3$
- **Molto meglio del caso precedente!**

# 5.3.1 FCFS: osservazioni

- Dunque, FCFS sembra “comportarsi male” nei confronti dei processi brevi
- Inoltre, FCFS va malissimo per i sistemi time-sharing perché non garantisce un tempo di risposta ragionevole
- E va ancora peggio per i sistemi real-time, perché non è preemptive
- Dall’esempio visto, sembra che le cose migliorino facendo passare prima i job più corti, indipendentemente dall’ordine in cui sono arrivati in ready queue, ma allora...

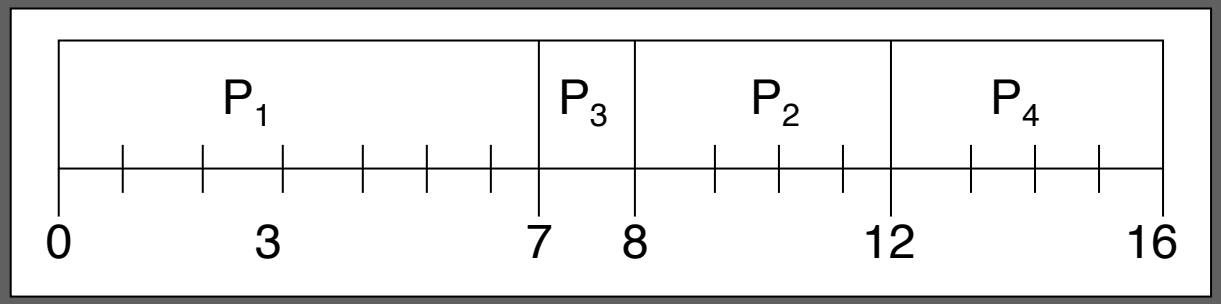
# 5.3.2 Shortest Job First (SJF)

- Si esamina la durata del **prossimo** burst di CPU di ciascun processo in RQ e si assegna la CPU al processo con il burst di durata minima
- Il suo nome esatto è “Shortest Next CPU Burst”.
- Può essere usato in modo pre-emptive e non pre-emptive
- nel caso preemptive, **se arriva in coda di ready un processo il cui burst time è inferiore a quanto rimane da eseguire al processo attualmente running,** quest’ultimo viene interrotto e la CPU passa al nuovo processo. Questo schema è noto come Shortest-Remaining-Time-First (***SRTF***).

# 5.3.2. Esempio di SJF non-preemptive

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 7 |
| $P_2$ | 2 | 4 |
| $P_3$ | 4 | 1 |
| $P_4$ | 5 | 4 |

turn
7
6
8
11
8

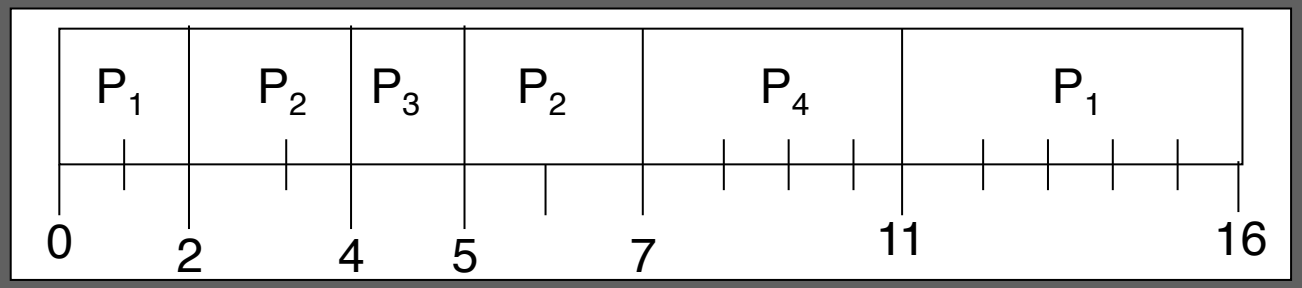



- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

# 5.3.2 Esempio di SJF preemptive

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 7 |
| $P_2$ | 2 | 4 |
| $P_3$ | 4 | 1 |
| $P_4$ | 5 | 4 |

turn
16
5
1
6
7




- Average waiting time = (9 + 1 + 0 +2)/4 = 3

# 5.3.2 SJF: osservazioni

- Si può dimostrare che SJF è **ottimale**: spostando un processo breve prima di uno di lunga durata (anche se quest'ultimo è arrivato prima) si migliora l'attesa del processo breve più di quanto si peggiori l'attesa del processo lungo

  ⇒ **il tempo medio di attesa diminuisce!**

- E di conseguenza diminuisce anche il turnaround
- **SJF ottimale:** nessun algoritmo di scheduling può produrre un tempo di attesa medio e un turnaround medio migliori
- già ma c'è un problema...

# 5.3.2 SJF: osservazioni

- Purtroppo, la durata del prossimo burst di CPU di un processo non è nota. **SJF non è implementabile!**
- SJF può al massimo essere approssimato usando medie pesate per stimare quanto durerà il prossimo burst di CPU di un processo in base alla durata dei precedenti suoi burst di CPU.
- E quindi eseguire lo scheduling sulla base dei valori di queste stime, fatte per tutti i processi in RQ in un certo momento.

# 5.3.2 SJF: osservazioni

- Dunque, SJF può essere sia preemptive che non preemptive
- **Nel caso preemptive**, se arriva un processo in RQ con next CPU burst più breve di **quanto rimane** al processo attualmente in CPU, si da subito la CPU al nuovo arrivato.
- **Nel caso non preemptive**, quando il processo running abbandona la CPU, questa viene assegnata al processo in RQ col burst di CPU più breve.
- Osserviamo che il tempo di attesa medio può migliorare usando SJF preemptive anziché SJF non preemptive

# FCFS e SJF: osservazioni

- In sostanza, **FCFS è il *peggiore*** degli algoritmi ragionevoli: funziona ma fornisce spesso tempi medi di attesa e di turnaround pessimi

- Al contrario, **SJF è il *migliore*** algoritmo possibile, ma non è implementabile, e possiamo solo usarlo per fare simulazioni con processi i cui burst di CPU sono noti a priori.

- FCFS e SJF rappresentano i due estremi di uno spettro di possibili algoritmi di scheduling. Un algoritmo di scheduling sarà tanto più buono quanto più le sue prestazioni si allontanano da FCFS e si avvicinano a SJF

# 5.3.4 Scheduling a Priorità

- Associamo una misura di **priorità** (di solito un numero intero) ad ogni processo
- La CPU è assegnata al processo con la priorità migliore tra quelli presenti in coda di ready.
- SJF è un tipo di scheduling a priorità. La durata del prossimo burst time è la priorità corrente di ogni processo
- anche FCFS è uno scheduling a priorità (che cosa si usa in questo caso come priorità di un processo?)

# 5.3.4 Scheduling a Priorità

- In generale, il calcolo della priorità dei processi può essere:
  - **interna al sistema**: calcolata dal SO sulla base del comportamento di ogni processo (ad esempio, in base alle risorse usate fino a quel momento un processo)
  - **esterna al sistema:** assegnata con criteri esterni al SO (ad esempio, una priorità che cambia in base a quale utente ha lanciato il processo)
- Lo cheduling a prioprità può essere implementato sia in modalità **preemptive** che **non preemptive**

# 5.3.4 Scheduling a Priorità: Starvation e aging

- Problema: che succede se un processo in RQ ha sempre una priorità peggiore di qualche altro processo in RQ?
- il processo non verrà mai scelto dallo scheduler.
  Si dice che va in **starvation** (muore di fame...)
- Per risolvere questo problema si usa un meccanismo di **aging**: il SO aumenta la priorità di un processo Px man mano che Px passa tempo in RQ: prima o poi Px avrà una priorità maggiore di qualsiasi altro processo e verrà scelto dallo scheduler.
- Domanda: FCFS, SJF preemptive e non preemptive, possono provocare starvation?

# 5.3.3 Scheduling Round Robin (RR) (scheduling circolare)

- Ogni processo ha a disposizione una certa quantità di tempo di CPU, chiamata **quanto di tempo** (valori ragionevoli vanno da 10 e 100 millisecondi). Per ora assumiamo un unico quanto di tempo prefissato assegnato a tutti i processi.

- Se entro questo arco di tempo il processo non lascia volontariamente la CPU, viene interrotto e rimesso in RQ.

- La RQ è vista come una coda circolare, e si verifica una sorta di "girotondo" di processi

# 5.3.3 Round Robin

- L'implementazione dello scheduling round robin è concettualmente molto semplice:
  - lo scheduler sceglie il primo processo in RQ (ad esempio secondo un criterio FCFS)
  - lancia un timer inizializzato al quanto di tempo
  - passa la CPU al processo scelto

- Se il processo ha un CPU burst minore del quanto di tempo, il processo rilascierà la CPU **volontariamente** prima dello scadere del tempo assegnatogli

# 5.3.3 Round Robin

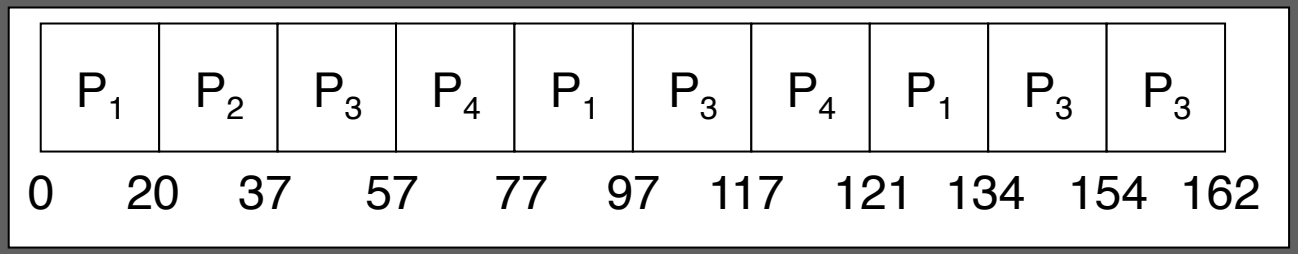
- Se invece il CPU burst del processo è maggiore del quanto di tempo, allora
  - il timer scade e invia un interrupt
  - il SO riprende il controllo della CPU
  - toglie la CPU al processo running e lo mette in fondo alla RQ
  - prende il primo processo in RQ e ripete tutto

# 5.3.3 Round Robin: osservazioni

- Se ci sono ***n* processi in coda ready** e il quanto di tempo è *q*, allora ogni processo riceve 1/*n* del tempo della CPU e nessun processo aspetta per più di (*n*-1)*q unità di tempo*.
- Il RR è l'algoritmo di scheduling naturale per implementare il time sharing, ed è quindi particolarmente adatto per i sistemi interattivi: nel caso peggiore un utente non aspetta mai più di *(n-1)q unità di tempo* prima che il suo processo venga servito.
- Come vederemo negli esempi di casi reali, il SO adotta poi ulteriori misure per migliorare il tempo di risposta dei processi interattivi

# 5.3.3 RR con quanto di tempo = 20

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



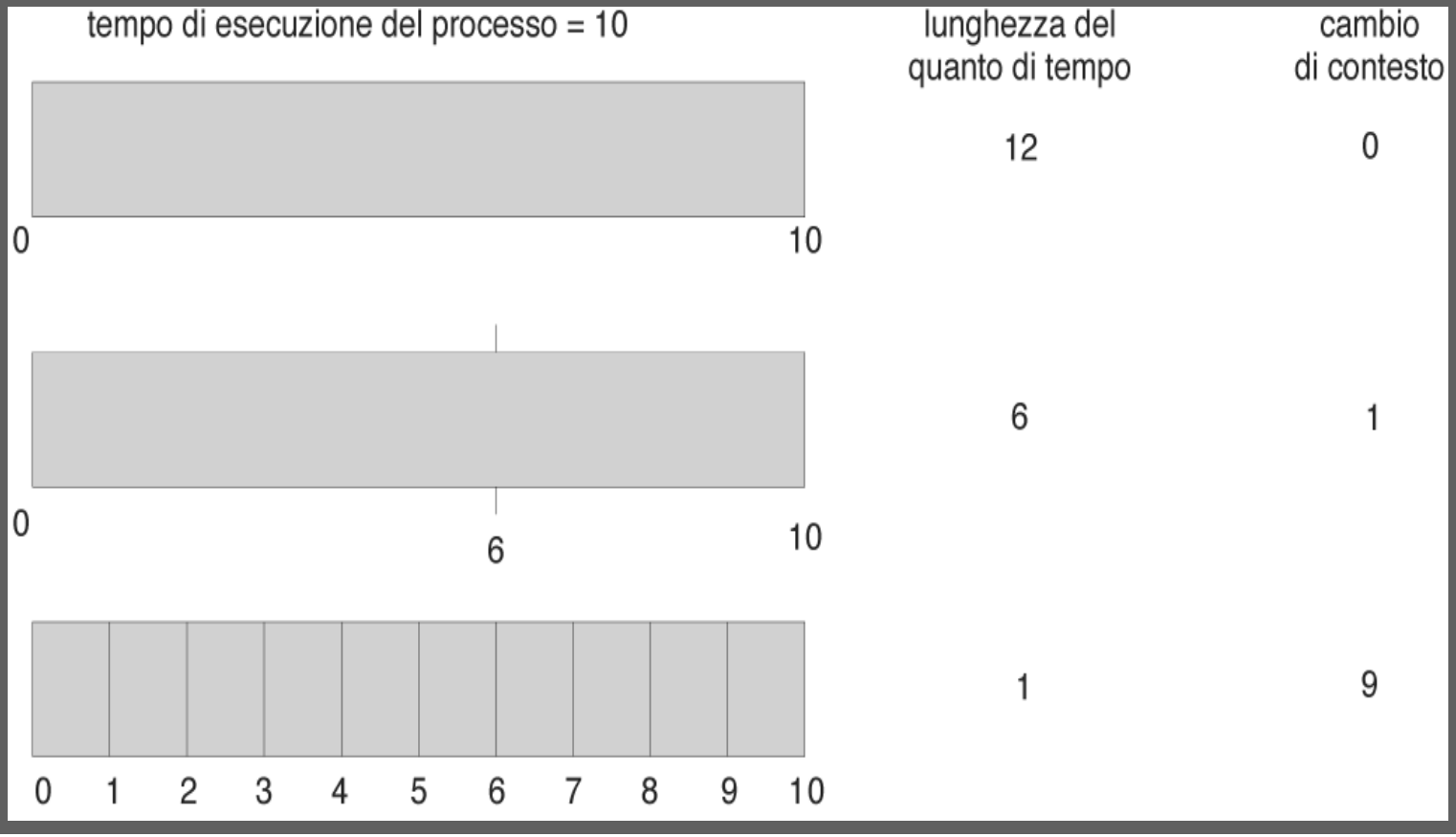

Tipicamente, si ha un *turnaround* medio maggiore di SJF, ma un migliore *tempo di risposta*.

# 5.3.3 Round Robin: osservazioni

- Le prestazioni di RR dipendono molto dal valore del quanto di tempo **q** scelto:
- q tendente a infinito rende RR uguale a FCFS
- q tendente a zero produce un maggior effetto di "parallelismo virtuale" tra i processi
- però aumenta il numero di context switch, e quindi l'overhead

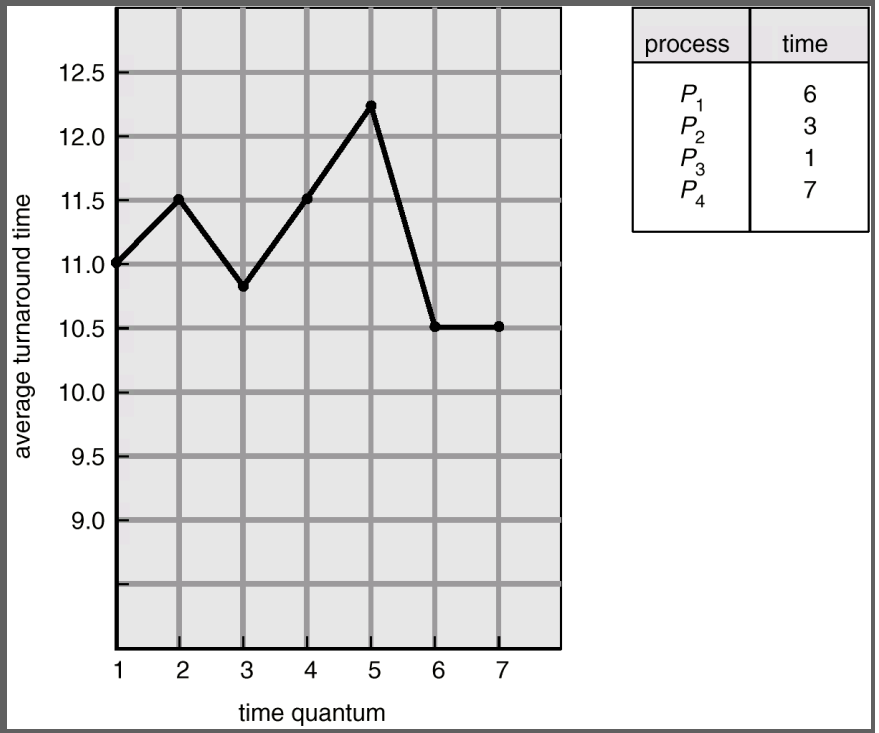
# 5.3.3 overhead del context switch nel Round Robin

- Quanti di tempo brevi producono un maggior numero di context switch, e quindi un aumento dell'overhead (fig. 5.5)



# 5.3.3 Tempo di Turnaround e quanto di tempo nel RR

- Il tempo di **turnaround** dipende dalla durata del quanto, ma non c'è una relazione precisa fra di loro (fig. 5.6).




| process | time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

- Una regola empirica dice che l'80% dei CPU burst dovrebbe essere minore di **q**

# 5.3.5 Scheduling a Code Multiple (Multi-level Queue Scheduling)

- si può realizzare quando i processi sono facilmente divisibili in classi differenti; ad esempio:
  - ***foreground*** (interattivi, ad esempio un editor)
  - ***background*** (non interagiscono con l'utente)
  - ***batch*** (la loro esecuzione può essere differita)
- allora si può partizionare la RQ in più code
  - inserire i processi in una delle code, sulla base delle proprietà del processo
  - gestire ogni coda con lo scheduling appropriato per le proprietà dei processi di quella coda

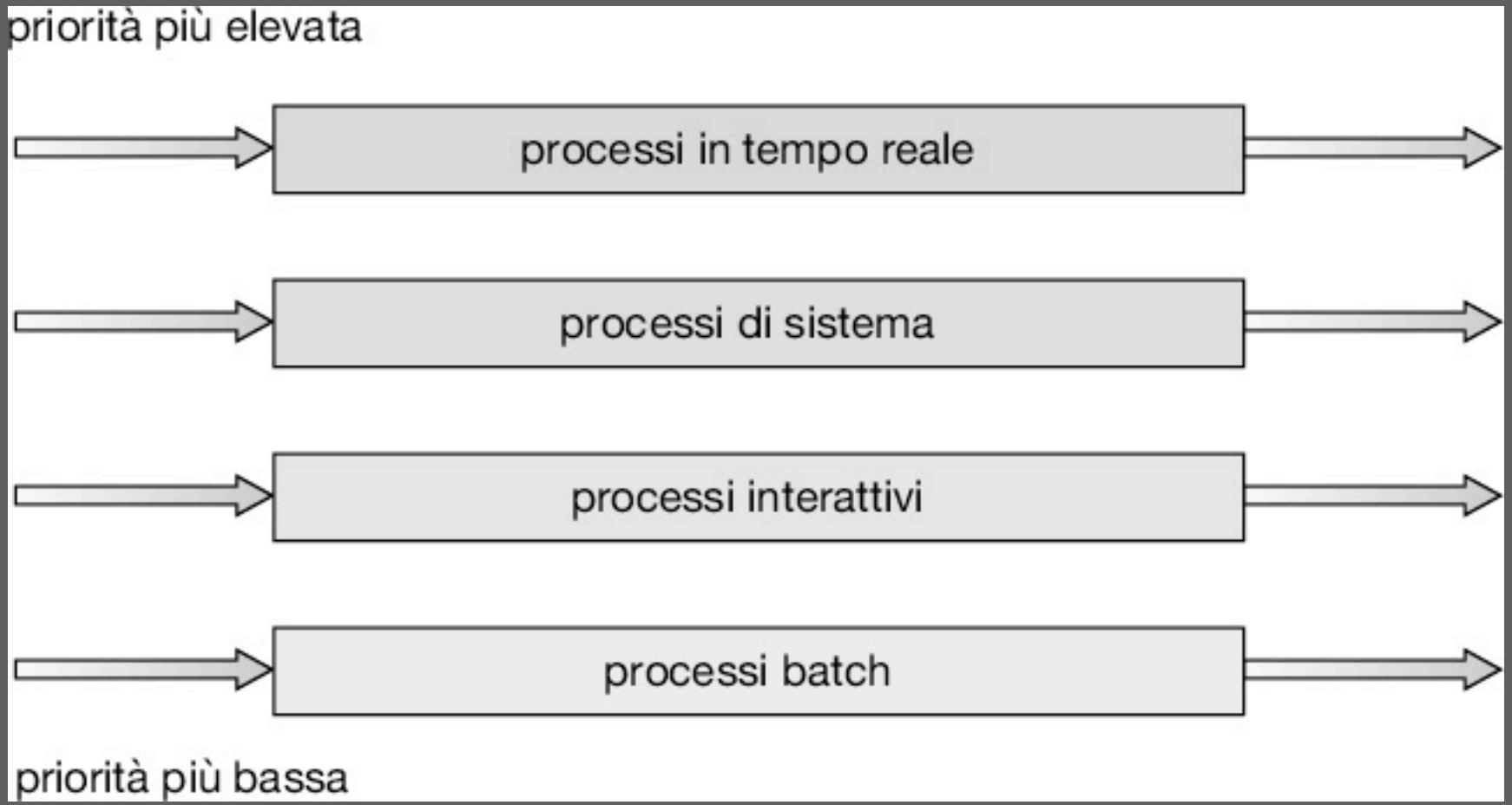
# 5.3.5 Scheduling a Code Multiple

- Ogni coda ha quindi la sua politica di scheduling, ad esempio:
  - ***foreground: RR***
  - ***background e batch: FCFS***

- ma come si sceglie fra le code?
  - **Scheduling a priorità fissa**: servire prima tutti i processi nella coda foreground e poi quelli in background e batch. *Possibilità di starvation.*
  - **Time slice**: ogni coda ha una certa quantità di tempo di CPU, ad esempio: 80% alla coda foreground e 20% alla coda background e batch

# 5.3.5 Scheduling a Code Multiple

- Un esempio di partizionamento dei processi in più code (fig. 5.8)

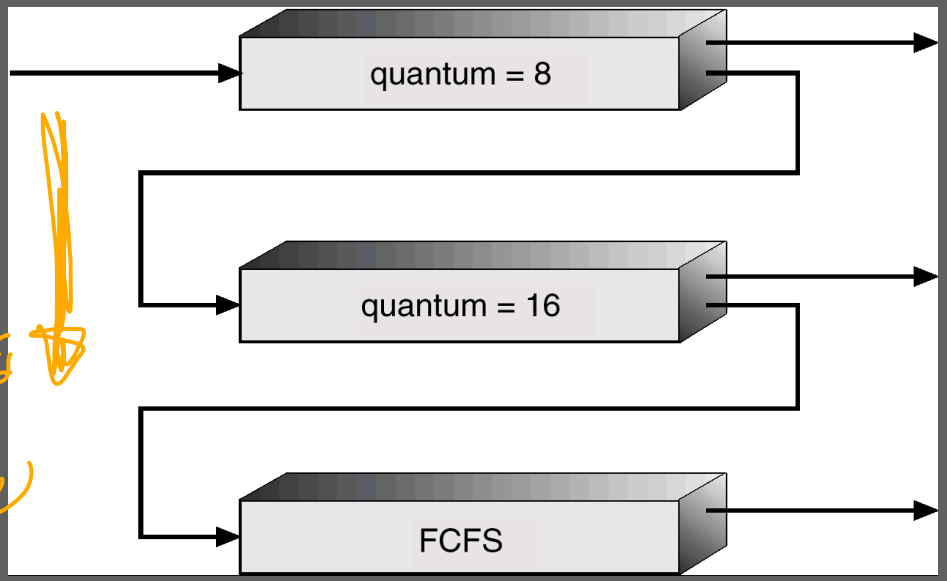


# 5.3.6 Scheduling a Code Multilivello con retroazione (MFQS)

- Il tipo più generale di algoritmo di scheduling è lo **scheduling a code multilivello con retroazione**. Tutti i sistemi operativi moderni adottano una qualche variante di questo algoritmo
- l'assegnamento di un processo a una coda non è più fisso. I processi possono venire spostati dal SO da una coda ad un'altra per:
  - adattarsi alla lunghezza del CPU burst del processo
  - gestire ogni coda con lo scheduling più adatto per i processi che mostrano un certo tipo di comportamento

# 5.3.6 MFQS: esempio

- In quest'esempio (fig. 5.9) le prime due code sono gestite con RR, la terza con FCFS.
- Quando un processo nasce, viene inserito nella prima coda (q=8) ma se non finisce il **CPU burst** nel quanto assegnatogli, viene "retrocesso" alla coda successiva.
- E' definita una priorità tra le code (quale?) e queste sono gestite con preemption.
- Domanda: possiamo pensare anche ad una forma di "promozione" dei processi?



# 5.3.6 Scheduling a Code Multilivello con retroazione (MFQS)

- La politica MFQS è definita dai seguenti parametri:
  - numero delle code
  - algoritmo di scheduling di ogni coda
  - quando declassare un processo
  - quando promuovere un processo
  - in che coda inserire un processo quando arriva (dall'esterno oppure da un I/O burst)

- MFQS è il tipo di scheduling più generale, e il più complesso da configurare adeguatamente in modo da ottenere le prestazioni migliori.

# 5.5 Scheduling per sistemi multi-core

- Sono ormai comuni le architetture con CPU a 2, 4, 8 core. (cfr. le slides sulle CPU multi-core del capitolo 1).
- Sono, in sostanza, dei piccoli sistemi multiprocessore in cui sullo stesso chip sono presenti due o più core che vedono la stessa memoria principale e condividono un livello di cache.
- La presenza di più "unità di esecuzione" dei processi, permette naturalmente di aumentare le prestazioni della macchina, posto che il SO sia in grado di sfruttare a pieno ciascun core.

# 5.5 Scheduling per sistemi multi-core

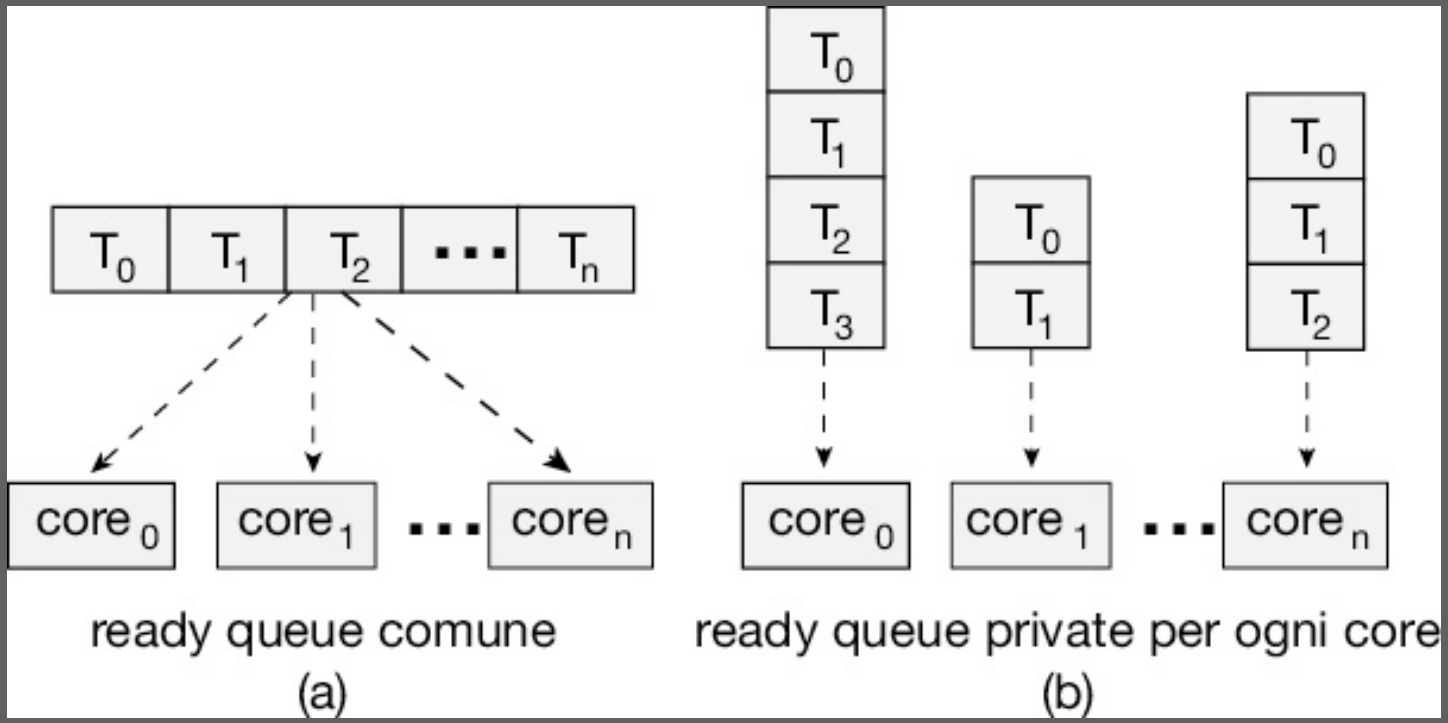
- Tutti i moderni SO prevedono in particolare la cosiddetta **multielaborazione simmetrica** (o **SMP**, da **Symmetric Multi-Processing**), in cui (tralasciando un po' di dettagli) uno scheduler gira su ciascun core.

- I processi "ready to run" possono essere inseriti in un'unica coda, vista da ciascun scheduler, oppure vi può essere una coda "ready to run" separata per ciascun core.

- Quando lo scheduler di un core si attiva, sceglie uno dei processi "ready to run" e lo manda in esecuzione sul proprio core.

# 5.5 Scheduling per sistemi multi-core

- Fig. 5.11. Notate: nel caso di RQ comune occorre evitare che lo stesso processo venga scelto per errore da due scheduler distinti e mandato in esecuzione due volte.



# 5.5 Scheduling per sistemi multi-core

- Un aspetto importante dei sistemi multi-core è il **bilanciamento del carico**.
- Non ha infatti senso avere un sistema con 2 o 4 o 8 core se poi i vari processi da eseguire non sono distribuiti più o meno omogeneamente tra i vari core.
- Nel caso di un'unica RQ, il bilanciamento del carico sarebbe sostanzialmente automatico: quando un core è inattivo, il suo scheduler prenderà un processo dalla coda comune e lo manderà in esecuzione su quel core.

# 5.5 Scheduling per sistemi multi-core

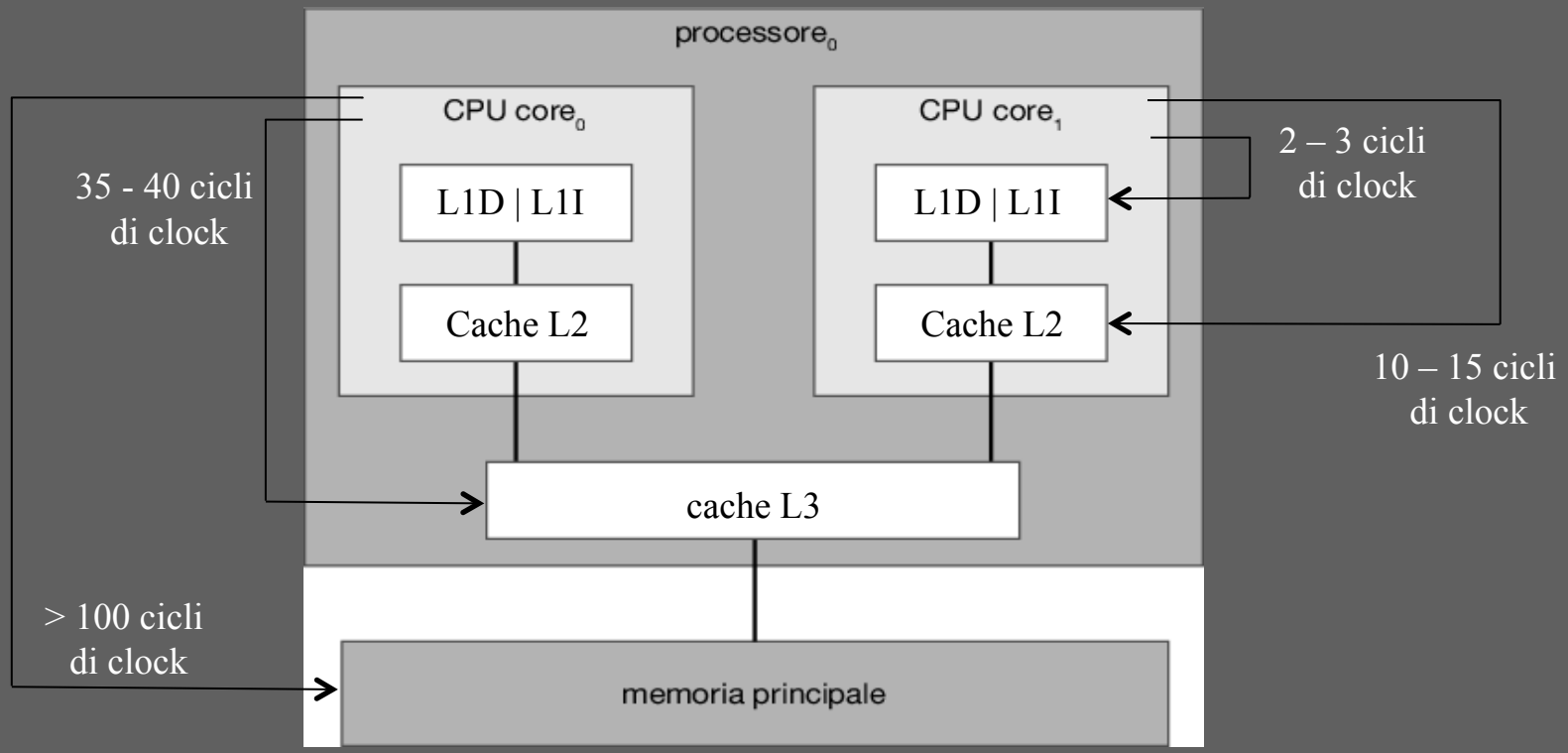
- I SO moderni preferiscono però avere una coda separata per ciascun core: ciò evita i problemi della coda unica, ma richiede un meccanismo di bilanciamento del carico: prendere un processo in attesa sulla RQ di un core sovraccarico e spostarlo nella RQ di un core scarico.
- Ad esempio, Linux SMP attiva il proprio meccanismo di bilanciamento del carico ogni 200 millisecondi, e ogni qualvolta si svuota la coda di un core.
- Tuttavia, la presenza di vari livelli di cache fa si che possa non convenire spostare un processo da un core ad un altro (una ragione in più per usare una RQ per ciascun core).

# 5.5 Scheduling per sistemi multi-core

- Infatti, i dati e le istruzioni di un processo si trovano inizialmente in memoria primaria, e man mano che vengono indirizzati vengono copiati nei vari livelli di cache, dove sono più velocemente accessibili per i successivi indirizzamenti.
- Ma se un processo si sposta su un altro core, inizialmente non troverà più nelle cache private quei dati e istruzioni, e dovrà sprecare più tempo per recuperarli, se va bene, dalla cache L3, che condivide con gli altri core.
- Per evitare rallentamenti dovuti a questo problema, specifiche system call permettono di vincolare un processo ad un certo core, indipendentemente dal suo carico

# 5.5 Scheduling per sistemi multi-core

- Qui vediamo il costo in cicli di clock necessari per accedere ad un dato/istruzione in un certo livello di cache. I valori si riferiscono ad un processore Intel core-i7, ma valgono per la maggior parte dei processori moderni.



# 5.7 Esempi di sistemi operativi: 5.7.3 Lo scheduling in Solaris

- *(Nota: nel testo, in questa sezione si parla anche di scheduling dei "thread". Va bene sostituire il termine con quello di "processo". Qui vedremo versioni semplificate di questi algoritmi)*
- Solaris usa uno scheduling a priorità con code multiple a retroazione, in cui i processi sono suddivisi in 4 classi:

1. real time (priorità maggiore)
2. Sistema
3. interattiva
4. time sharing (priorità minore)

- un processo usa la CPU fino a che non termina, va in wait, esaurisce il suo quanto di tempo, è soggetto a prelazione.

# 5.7.3 Lo scheduling in Solaris

- Normalmente, un processo utente nasce nella classe *time sharing*, se non diversamente specificato.
- Le classi *time sharing* e *interattiva* hanno gli stessi criteri di scheduling, con 60 diversi livelli di priorità.
- Per l'assegnazione del quanto di tempo e il ricalcolo della priorità di un processo, il sistema usa una tabella con 60 righe, così fatta (mostriamo solo alcune righe):

| Priorità corrente | Quanto di tempo (millisecondi) | Nuova priorità (quanto esaurito) | Nuova priorità (quanto non esaurito) |
|---|---|---|---|
| 0 | 200 | 0 | 50 |
| 20 | 120 | 10 | 52 |
| 30 | 80 | 25 | 53 |
| 59 | 20 | 49 | 59 |

# 5.7.3 Lo scheduling in Solaris

- La **Priorità corrente** del processo, determina il **Quanto di tempo** che gli verrà assegnato. Notate che priorità e tempo assegnato sono inversamente proporzionali.
- **Quanto esaurito**: la nuova priorità assegnata al processo se questo ha esaurito tutto il quanto di tempo assegnatogli senza sospendersi. Notate che la nuova priorità è più bassa, per cui in futuro riceverà un quanto di tempo più grande.
- **Quanto non esaurito**: la nuova priorità di un processo che si è fermato prima di consumare tutto il suo quanto di tempo. Notate come la nuova priorità sia più alta, per cui in futuro riceverà un  quanto di tempo più breve.

# 5.7.3 Lo scheduling in Solaris

- Di fatto, è il comportamento del processo utente a stabilire se questo sta nella classe *time sharing* (priorità da 0 a 49) o nella classe *interattiva* (priorità da 50 a 59).
- I processi nelle classi *real time* e *sistema* hanno invece priorità fissa, superiore alla priorità dei processi *time sharing* e *interattivi*.
- Lo scheduler calcola la **priorità globale** di un processo in base alla sua priorità e alla classe a cui appartiene, e assegna la CPU al processo con priorità globale più alta (si usa RR in caso di più processi di uguale priorità).
- L'algoritmo è preemptive: il processo attivo può essere interrotto da un processo con priorità globale più alta.

# 5.7.2 Lo scheduling in Windows

- Lo scheduling in Windows è basato su priorità con retroazione e prelazione.
- Lo scheduler usa uno schema a 32 livelli di priorità: i processi real-time hanno una priorità da 16 a 31, gli altri processi hanno una priorità da 1 a 15 (0 è un valore riservato).
- Nel seguito ci limitiamo ai soli processi non real time: quando un processo nasce gli viene assegnata la priorità 1.
- In generale, lo scheduler sceglie il processo a priorità più alta e gli assegna la CPU. Se ci sono più processi con la stessa priorità, si usa RR.

# 5.7.2 Lo scheduling in Windows

- Se il processo che ha la CPU va in wait prima di esaurire il suo quanto di tempo, la sua priorità viene alzata (fino al massimo a 15)
- L'entità dell'incremento dipende dal tipo di evento che il processo attende: se è un dato dalla tastiera (quindi siamo in presenza di un processo interattivo) si avrà un grosso incremento della priorità.
- Se il processo attende un dato dal disco, l'incremento è minore.
- Al contrario, se il processo che ha la CPU esaurisce il proprio quanto di tempo, la sua priorità viene abbassata (ma mai sotto la soglia 1).

# 5.7.2 Lo scheduling in Windows

- Questa strategia favorisce ovviamente i processi che interagiscono con il mouse e la tastiera, per i quali è importante un tempo di risposta molto basso.
- Per la stessa ragione, Windows distingue tra i processi in background e il processo in foreground.
- Quando un processo passa in foreground il quanto di tempo assegnatogli viene moltiplicato per 3, cosicché il processo può continuare l'esecuzione per un tempo tre volte più lungo, prima di abbandonare la CPU.

# 5.7.1 Lo scheduling in Linux

- Linux ha adottato diversi algoritmi di scheduling, ereditando i primi dal suo fratello maggiore Unix, ma a partire dal 2007( versione 2.6.23) l'algoritmo predefinito è diventato il **Completely Fair Scheduler (CFS)** che è sostanzialmente diverso dall'algoritmo usato nelle precedenti versioni.

- Il CFS cerca di distribuire equamente il tempo di CPU tra tutti i processi ready to run, seguendo l'assunzione che se ci sono N processi ready to run, allora ad ogni processo dovrebbe spettare esattamente 1/N-esimo del tempo di CPU

# 5.7.1 Lo scheduling in Linux

- Ad ogni context switch, il CFS ricalcola per quanto tempo si dovrebbe dare la CPU ad un processo P in modo che tutti i processi abbiano avuto, fino a quel momento, la stessa quantità di tempo di CPU. Siano:

- **P.expected_run_time**: il tempo di CPU spettante a P

- **P.vruntime:** il tempo di CPU già consumato da P

- **P.due_cputime:** il tempo di CPU che ancora spetta a P

- Dunque:
  **P.vruntime = P.expected_run_time – P.due_cputime**

# 5.7.1 Lo scheduling in Linux

- Il criterio di scheduling è poi molto semplice: **la CPU viene data al processo con il più basso valore di P.vruntime**, ossia al processo che fin'ora ha usato la CPU per meno tempo.

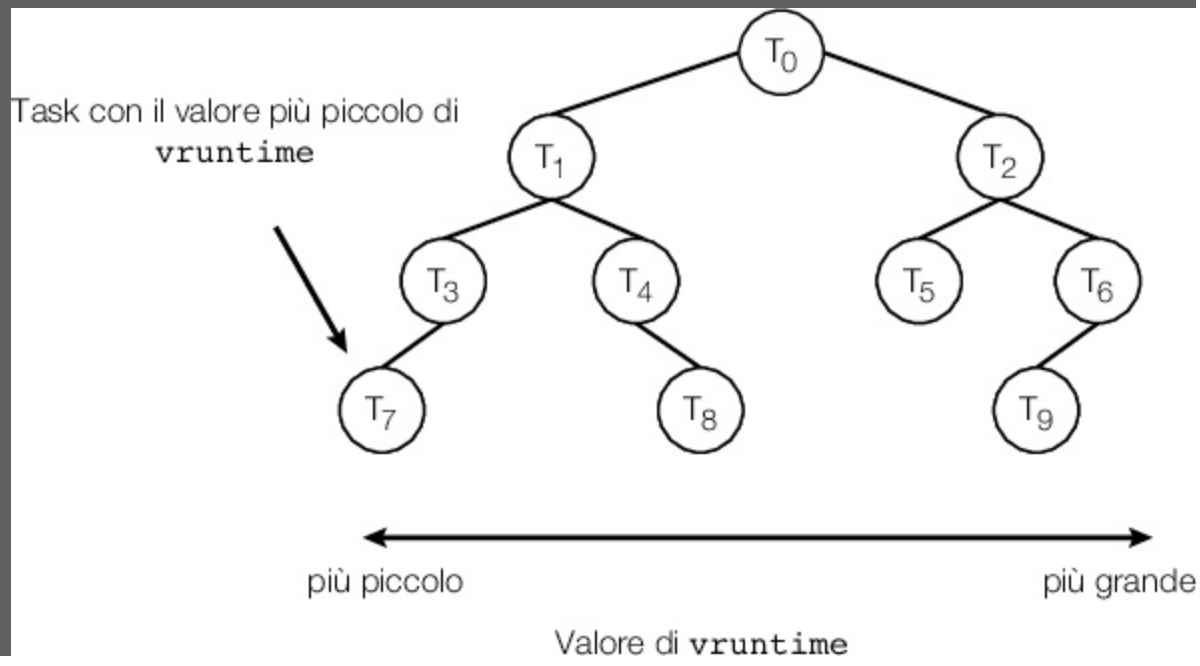
- Nell'implementazione del CFS i processi "ready to run" non vengono più organizzati in code di scheduling ma diventano i nodi di un albero di ricerca bilanciato detto **red-black tree (o R-B tree)**

- Un red-black tree permette operazioni molto efficienti: se è formato da n nodi, allora cercare, inserire o cancellare uno dei nodi ha un costo computazionale **O(log n)**

# 5.7.1 Lo scheduling in Linux

- Negli alberi R-B il nodo più a sinistra è sempre quello col valore chiave più basso, e nel CFS i processi sono inseriti nel R-B tree usando come chiave **P.vruntime**

- Dunque, il processo associato al nodo più a sinistra ha il valore **P.vruntime** più basso, cioè è il processo che ha usato la CPU per meno tempo, e al context switch sarà scelto per entrare in esecuzione.



# Per chi vuole approfondire

- Sezione 5.8: valutazione degli algoritmi (di scheduling)