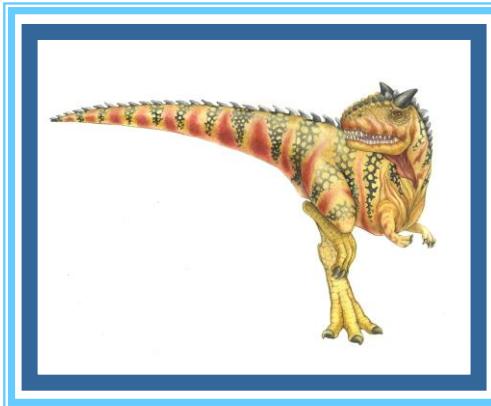


Scheduling della CPU





Obiettivi

- ❖ Descrivere vari algoritmi di scheduling della CPU
- ❖ Valutare gli algoritmi in base ai criteri di scheduling
- ❖ Spiegare le problematiche relative allo scheduling multiprocessore e multicore
- ❖ Descrivere vari algoritmi di scheduling real-time
- ❖ Utilizzare modellazione e simulazioni per valutare gli algoritmi di scheduling della CPU in contesti reali





Sommario

- ❖ Scheduling della CPU: concetti fondamentali
- ❖ Criteri di scheduling
- ❖ Algoritmi di scheduling
- ❖ Scheduling dei thread
- ❖ Scheduling multiprocessore
- ❖ Scheduling real-time
- ❖ Scheduling in Linux
- ❖ Valutazione degli algoritmi





Concetti fondamentali – 1

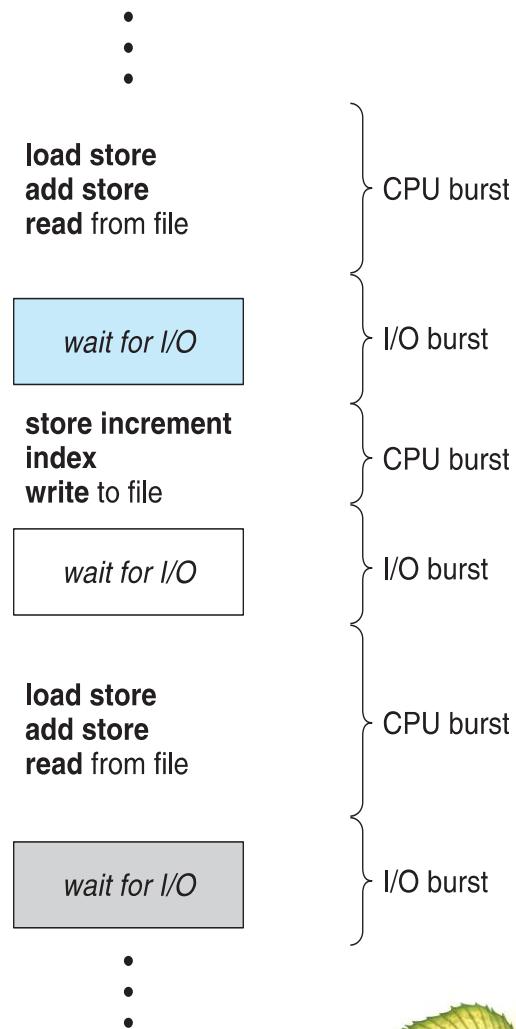
- ❖ Lo **scheduling** è una funzione fondamentale dei sistemi operativi
 - Si sottopongono a scheduling quasi tutte le risorse di un calcolatore
- ❖ Lo scheduling della CPU è alla base dei sistemi operativi multiprogrammati
 - Attraverso la commutazione del controllo della CPU tra i vari processi, il SO rende più “produttivo” il sistema di calcolo
 - Ottimizza l’utilizzo del/i processore/i

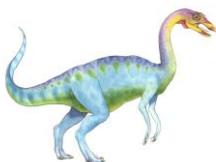




Concetti fondamentali – 2

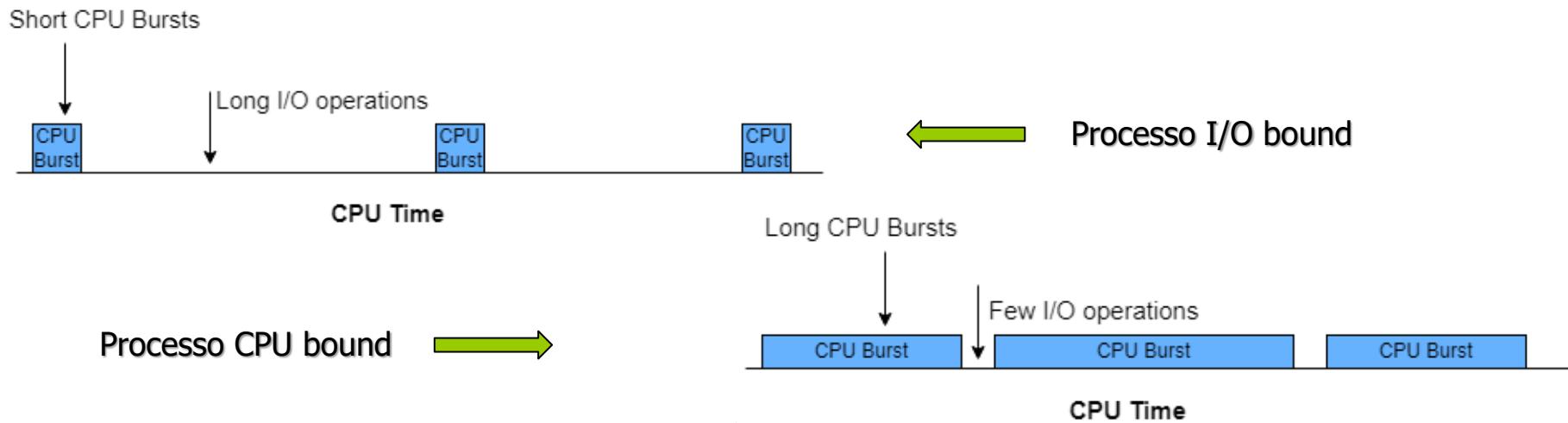
- ❖ Il massimo impiego della CPU è pertanto ottenuto con la multiprogrammazione
- ❖ **Ciclo di CPU–I/O burst** – L'elaborazione di un processo consiste di *cicli* di esecuzione nella CPU ed attese/utilizzo ai/dei dispositivi di I/O
- ❖ I **CPU–burst** si susseguono agli **I/O–burst** durante tutta la vita del processo





Concetti fondamentali – 3

- ❖ Un programma con prevalenza di I/O si definisce ***I/O-bound*** e produce generalmente molte sequenze di operazioni della CPU di breve durata
- ❖ Un programma con prevalenza di elaborazione si definisce ***CPU-bound*** e produce (poche) sequenze di esecuzione molto lunghe
- ❖ Avere contemporaneamente presenti in memoria processi di entrambi i tipi garantisce l'uso intensivo e "bilanciato" di tutte le risorse del sistema





Concetti fondamentali – 4

CPU scheduling

computers run 100s of programs at a time

```
$ ps aux | wc -l  
324
```

I have 324 processes "running"

but each CPU can only run 1 at a time

CPU 1 I'm running firefox!
CPU 2 I'm running gcc!
CPU 2 We're not on any CPU!
other programs

every program gets its turn on the CPU

CPU 1, run firefox!
Linux ... 1 ms later...
Linux ok, now run gcc!

what's running can change 1000x/second

cron I just run for 1ms every now and then, don't mind me!

this system is called the "scheduler"

Linux's scheduling algorithm is called the "Completely Fair Scheduler"

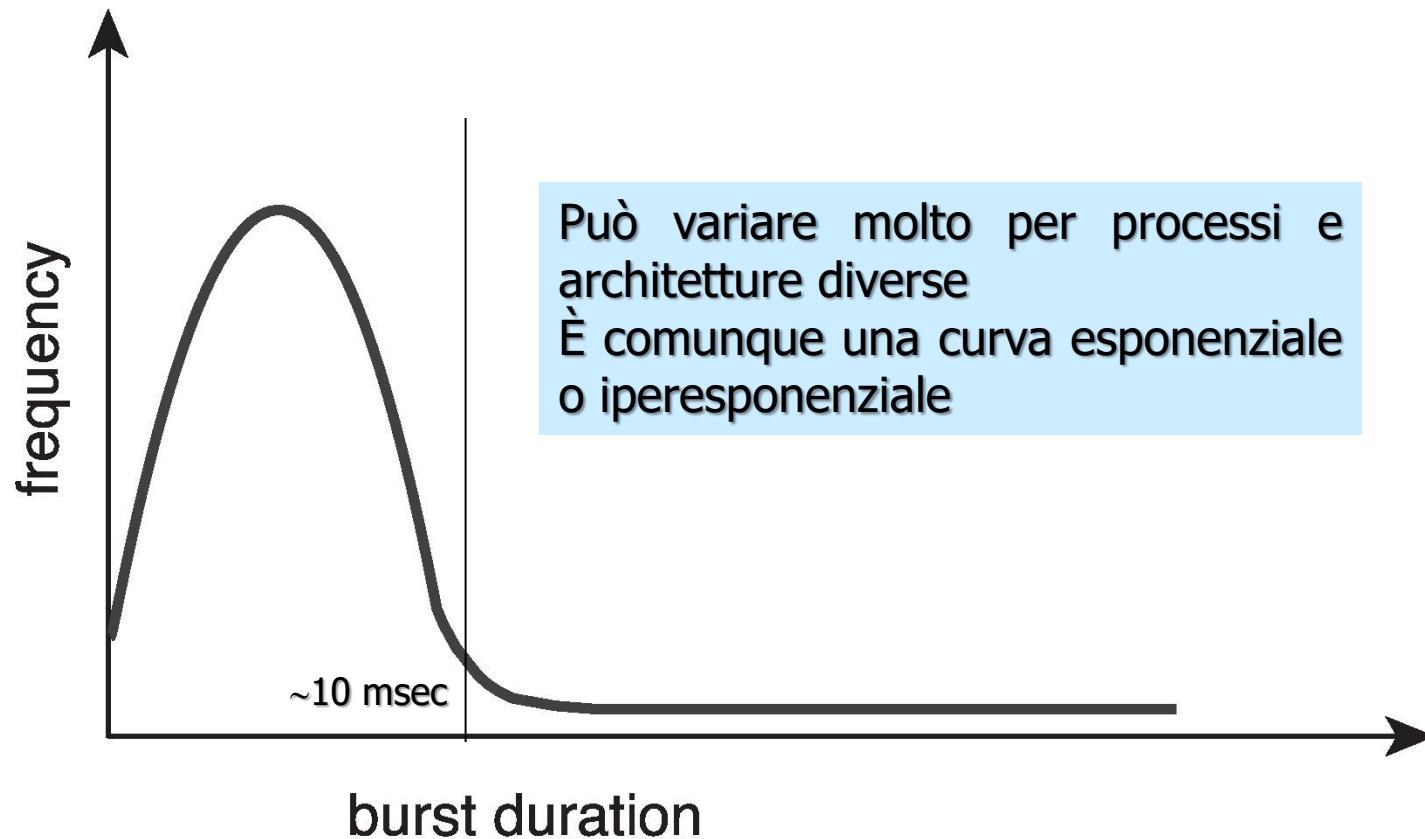
the scheduler wakes you up after a sleep

sleep (10)
program ... 10^{ish} seconds later...
time to run! Linux





Grafico della distribuzione dei CPU–burst

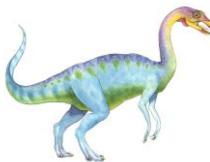




Scheduling dei processi o dei thread?

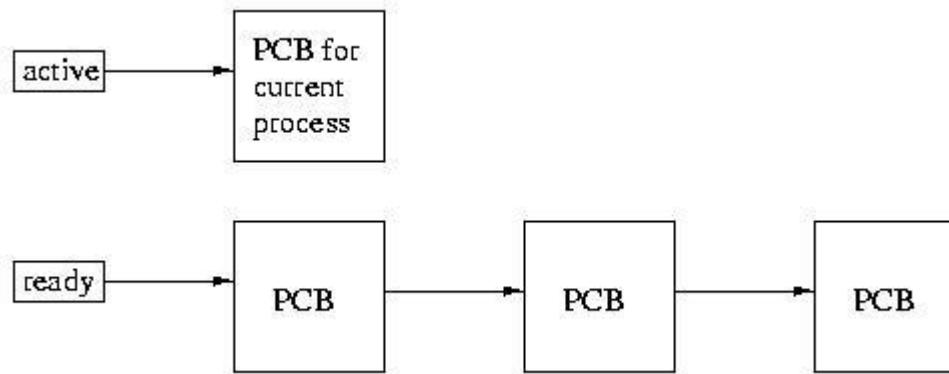
- ❖ Nei SO che li supportano, l'oggetto dell'attività di scheduling sono i thread
- ❖ Nella presente trattazione useremo:
 - il termine **scheduling dei processi** per analizzare i concetti generali del CPU scheduling
 - la locuzione **scheduling dei thread** nel trattare concetti strettamente inerenti al multithreading
- ❖ Allo stesso modo “eseguito su una CPU” coinciderà con “eseguito su un core” nel caso di architetture multicore

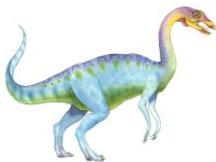




Lo scheduler della CPU – 1

- ❖ Lo scheduler della CPU gestisce la coda dei processi pronti, selezionando il prossimo processo cui verrà allocata la CPU
 - Gli elementi nella ready queue sono i PCB dei processi pronti
 - La ready queue può essere realizzata come una coda FIFO (*first-in-first-out*), una coda con priorità, una lista concatenata o un albero
 - La realizzazione (meccanismo) è spesso indipendente dalla politica implementata dallo scheduler





Lo scheduler della CPU – 2

- ❖ Lo scheduler della CPU deve prendere una decisione quando un processo:
 1. passa da stato *running* a stato *wait* (es.: richiesta di I/O o attesa terminazione di un processo figlio)
 2. passa da stato *running* a stato *ready* (interrupt)
 3. passa da stato *wait* a stato *ready* (es.: completamento di un I/O)
 4. termina
- ❖ Se lo scheduling viene effettuato solo nei casi 1 e 4, si dice che lo schema di scheduling è *nonpreemptive* (senza diritto di prelazione) o *cooperativo*
- ❖ Altrimenti si ha uno schema *preemptive*

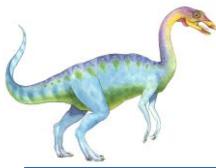




Lo scheduler della CPU – 3

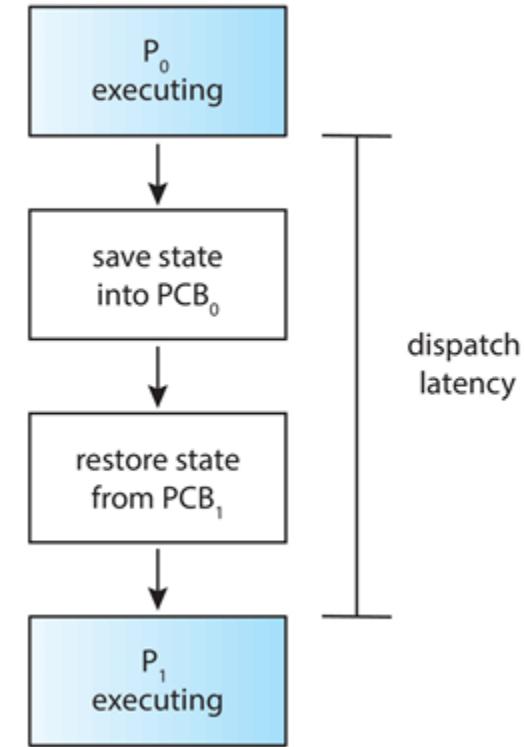
- ❖ In caso di scheduler preemptive, sono aspetti critici da tenere in considerazione...
 - L'accesso a dati condivisi (prelazione durante la modifica dei dati con risultati impredicibili)
 - La possibilità di prelazione di processi del kernel (che normalmente modificano dati vitali per il sistema)
 - L'interruzione di operazioni cruciali effettuate dal sistema operativo (system call, inibizione delle interruzioni)





Il dispatcher

- ❖ Il modulo *dispatcher* passa il controllo della CPU al processo selezionato dallo scheduler; il dispatcher effettua:
 - Context switch
 - Passaggio a modo utente
 - Salto alla posizione corretta del programma utente per riavviare l'esecuzione
- ❖ **Latenza di dispatch** — è il tempo impiegato dal dispatcher per sospendere un processo e avviare una nuova esecuzione





Context–switch

- ❖ Con quale frequenza avvengono i context–switch?
- ❖ In Linux, il comando **vmstat** fornisce il numero di cambi di contesto

```
$ vmstat 1 3
---cpu---
243
225
339
```

3 righe di output con numero medio di context-switch dall'avvio del sistema e nei due precedenti intervalli di tempo di 1 sec

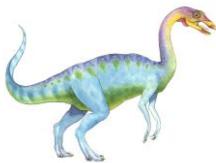
→ frequenza media (al secondo)
→ frequenze durante i due secondi precedenti

- ❖ È anche possibile usare il file system temporaneo **/proc**
- ❖ **Esempio**

```
$ cat /proc/2166/status
voluntary_ctxt_switches      150
nonvoluntary_ctxt_switches   8
```

...

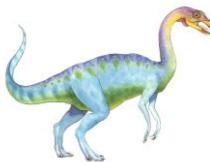




Criteri di scheduling

- ❖ **Utilizzo della CPU** — la CPU deve essere più attiva possibile (comando `top` in Linux, Mac OS e UNIX)
- ❖ **Throughput** (produttività) — numero dei processi che completano la loro esecuzione nell'unità di tempo
- ❖ **Tempo di turnaround** (tempo di completamento) — tempo impiegato per l'esecuzione di un determinato processo
 - Tempo di attesa nella ready queue e nelle code dei dispositivi, tempo effettivo di utilizzo della CPU e dei dispositivi di I/O
- ❖ **Tempo di attesa** — tempo passato dal processo in attesa nella ready queue
- ❖ **Tempo di risposta** — tempo che intercorre tra la sottomissione di un processo e la prima risposta prodotta
 - Nei sistemi time-sharing, il tempo di turnaround può essere influenzato dalla velocità del dispositivo di output





Criteri di ottimizzazione – 1

- ❖ Massimo utilizzo della CPU
- ❖ Massimo throughput
- ❖ Minimo tempo di turnaround
- ❖ Minimo tempo di attesa
- ❖ Minimo tempo di risposta





Criteri di ottimizzazione – 2

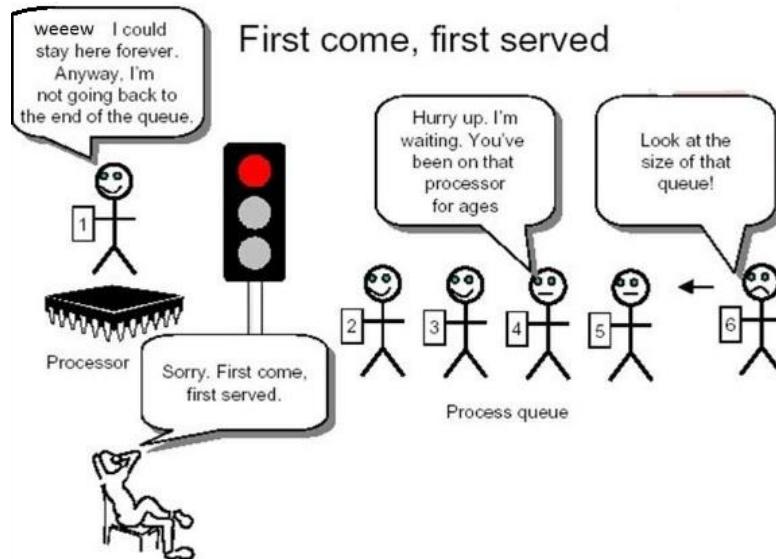
- ❖ Normalmente si ottimizzano i valori medi, ma talvolta è più opportuno ottimizzare i valori minimi/massimi
 - Per esempio, in sistemi interattivi (desktop), per garantire che tutti gli utenti ottengano un buon servizio
 - ⇒ ridurre il tempo massimo di risposta
- ❖ Per i sistemi time-sharing è invece più significativo ridurre la **varianza** rispetto al tempo medio di risposta: un sistema con tempo di risposta **prevedibile** è migliore di un sistema mediamente più rapido, ma molto variabile





Scheduling First–Come–First–Served (FCFS)

- ❖ La CPU viene assegnata al processo che la richiede per primo: la realizzazione del criterio **FCFS** si basa sull'implementazione della ready queue per mezzo di una coda FIFO
- ❖ Quando un processo entra nella ready queue, si collega il suo PCB all'ultimo elemento della coda; quando la CPU è libera, viene assegnata al processo che si trova alla testa della ready queue, rimuovendolo da essa





Scheduling FCFS (cont.)

❖ Esempio 1

Processo	Tempo di burst (millisecondi)
P_1	24
P_2	3
P_3	3

- I processi arrivano al sistema nell'ordine: P_1, P_2, P_3
- Per descrivere come si realizza lo scheduling si usa un **diagramma di Gantt**
 - ▶ Un istogramma che illustra una data pianificazione, includendo i tempi di inizio e di fine di ogni processo





Scheduling FCFS (cont.)

Processo	Tempo di burst (millisecondi)
P ₁	24
P ₂	3
P ₃	3

- Il diagramma di Gantt per lo scheduling **FCFS** è:



- Tempi di attesa in msec: P₁→0, P₂→24, P₃→27
- Tempo medio di attesa T_a=(0+24 +27)/3=17msec





Scheduling FCFS (cont.)

- Supponiamo che l'ordine di arrivo dei processi sia

P_2, P_3, P_1

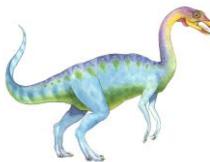
- Il diagramma di Gantt diventa:

Processo	Tempo di burst (millisecondi)
P_1	24
P_2	3
P_3	3



- Tempi di attesa in msec: $P_1 \rightarrow 6, P_2 \rightarrow 0, P_3 \rightarrow 3$
- Tempo medio di attesa $T_a = (6+0+3)/3 = 3$ msec
- Molto inferiore al precedente: in questo caso, non si verifica l'*effetto convoglio*, per cui processi di breve durata devono attendere che un processo lungo liberi la CPU
 - Situazione in cui vi è un solo processo CPU-bound e molti processi I/O-bound





Scheduling Shortest–Job–First (SJF)

- ❖ Si associa a ciascun processo la lunghezza del suo burst di CPU successivo; si opera lo scheduling in base alla brevità dei CPU–burst
- ❖ Due schemi:
 - **non-preemptive** — dopo che la CPU è stata allocata al processo, non gli può essere prelazionata fino al termine del CPU burst corrente
 - **preemptive** — se arriva un nuovo processo con burst di CPU minore del tempo rimasto per il processo corrente, il nuovo processo prelaziona la CPU: **SRTF**, *Shortest–Remaining–Time–First*
- ❖ **SJF** è *ottimo* — minimizza il tempo medio di attesa
 - Difficoltà nel reperire l'informazione richiesta



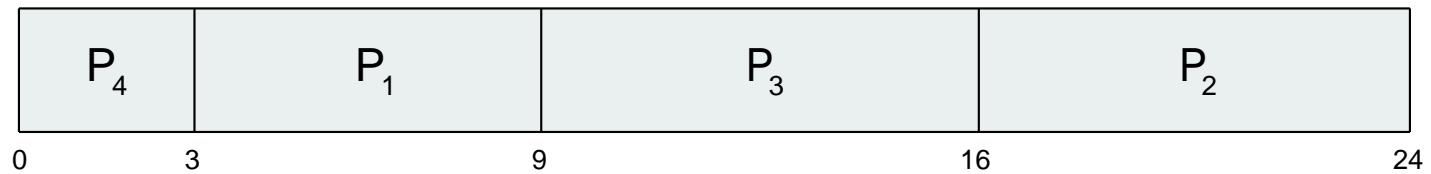


Scheduling SJF non-preemptive

❖ Esempio 2

Processo	Tempo di burst
P ₁	6
P ₂	8
P ₃	7
P ₄	3

- Diagramma di Gantt



- Tempo medio di attesa $T_a = (3+16+9+0)/4 = 7\text{msec}$
- Usando lo scheduling FCFS, $T_a = 10.25\text{msec}$



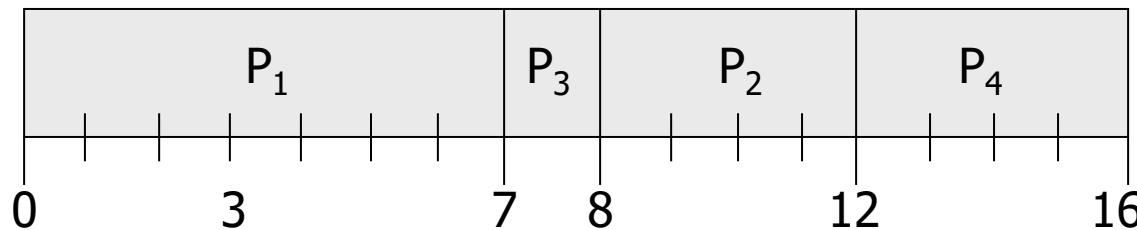


Scheduling SJF non-preemptive (cont.)

❖ Esempio 3

Processo	Tempo di arrivo	Tempo di burst
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

- Diagramma di Gantt



- Tempo medio di attesa T_a=(0+6+3+7)/4=4msec



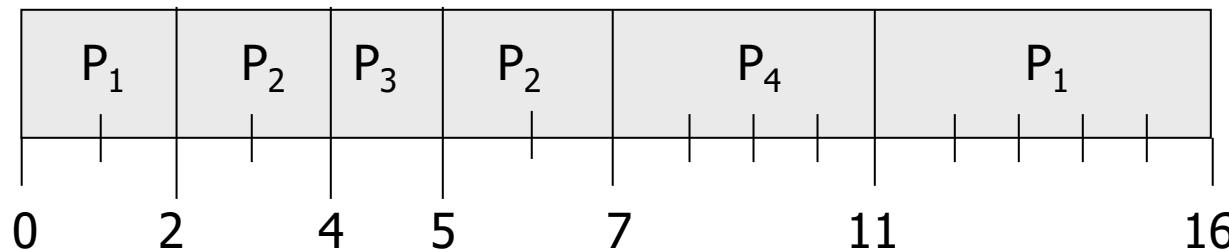


Scheduling SJF preemptive (SRTF)

❖ Esempio 4

Processo	Tempo di arrivo	Tempo di burst
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

- Diagramma di Gantt



- Tempo medio di attesa $T_a = (9+1+0+2)/4=3\text{msec}$



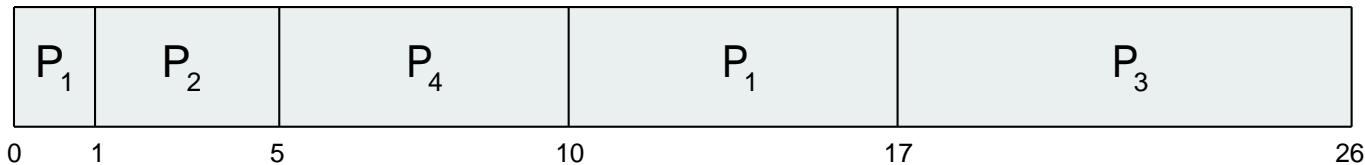


Scheduling SJF preemptive (SRTF)

❖ Esempio 5

Processo	Tempo di arrivo	Tempo di burst
P ₁	0	8
P ₂	1	4
P ₃	2	9
P ₄	3	5

- Diagramma di Gantt



- Tempo medio di attesa
- $T_a = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5\text{ msec}$
- Usando SJF non preemptive, $T_a = 7.75\text{ msec}$





Lunghezza del successivo CPU–burst

- ❖ Può essere stimata come una media esponenziale delle lunghezze dei burst di CPU precedenti (impiegando una combinazione convessa)

1. t_n = lunghezza dell' n -esimo CPU burst

2. τ_{n+1} = valore stimato del prossimo CPU burst

3. $0 \leq \alpha \leq 1$

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

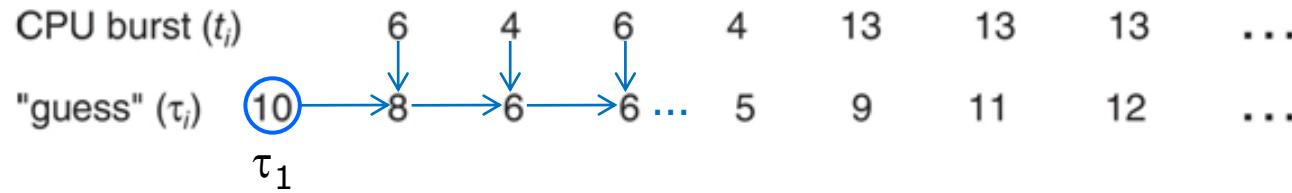
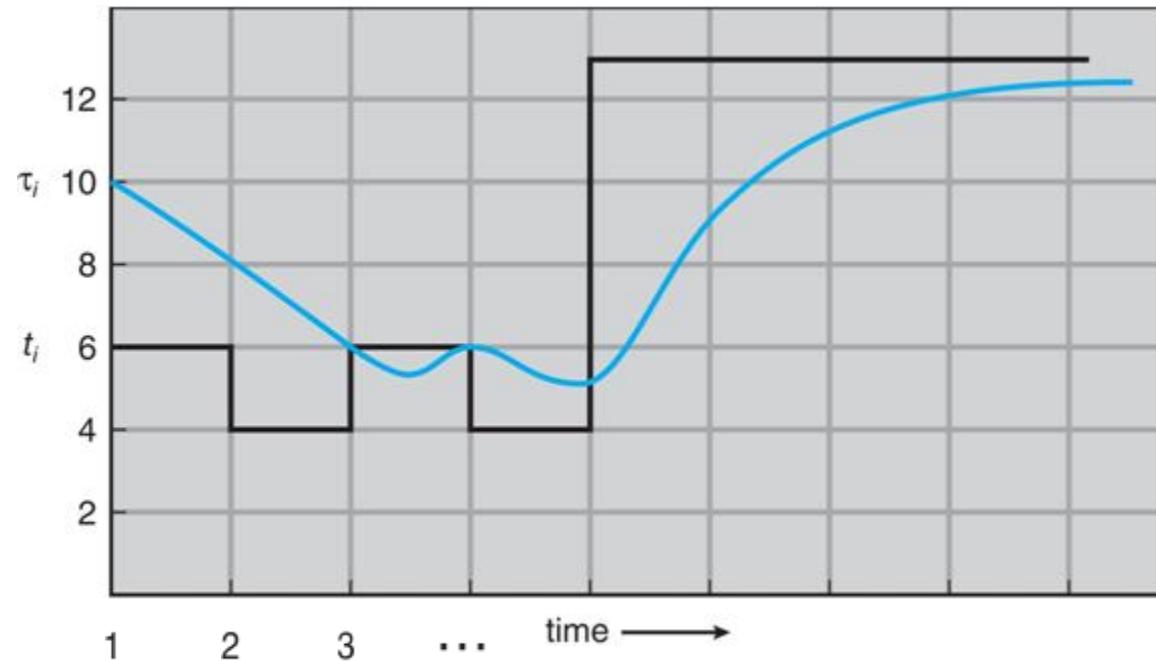
τ_1 = valore costante o stimato come media dei burst
del sistema

⇒ Solitamente, si usa $\alpha=1/2$





Predizione della lunghezza del successivo CPU–burst





Esempi di calcolo – 1

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

❖ $\alpha = 0$

- $\tau_{n+1} = \tau_n$
- La storia recente non viene presa in considerazione: le condizioni attuali sono transitorie

❖ $\alpha = 1$

- $\tau_{n+1} = t_n$
- Si considera solo l'ultimo CPU burst: la storia passata è irrilevante





Esempi di calcolo – 2

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

- ❖ Espandendo la formula si ottiene:

$$\tau_2 = \alpha t_1 + (1-\alpha)\tau_1$$

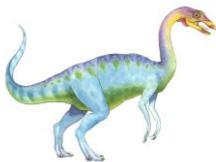
$$\begin{aligned}\tau_3 &= \alpha t_2 + (1-\alpha)\tau_2 = \alpha t_2 + (1-\alpha)[\alpha t_1 + (1-\alpha)\tau_1] \\ &= \alpha t_2 + (1-\alpha)\alpha t_1 + (1-\alpha)^2\tau_1\end{aligned}$$

...

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \dots + (1-\alpha)^j \alpha t_{n-j} + \dots + (1-\alpha)^n \tau_1$$

- ❖ Poiché α e $(1-\alpha)$ sono entrambi minori o uguali ad 1, ciascun termine ha minor peso del suo predecessore





Scheduling a priorità

- ❖ Un valore di priorità (intero) viene associato a ciascun processo
- ❖ La CPU viene allocata al processo con la priorità più alta (spesso, intero più basso=priorità più alta)
 - Preemptive
 - Non-preemptive
- ❖ SJF è uno scheduling a priorità in cui la priorità è calcolata in base al successivo tempo di burst
- ❖ Problema: *Starvation* ("inedia", blocco indefinito) – i processi a bassa priorità potrebbero non venir mai eseguiti
- ❖ Soluzione: *Aging* (invecchiamento) – aumento graduale della priorità dei processi che si trovano in attesa nel sistema da lungo tempo





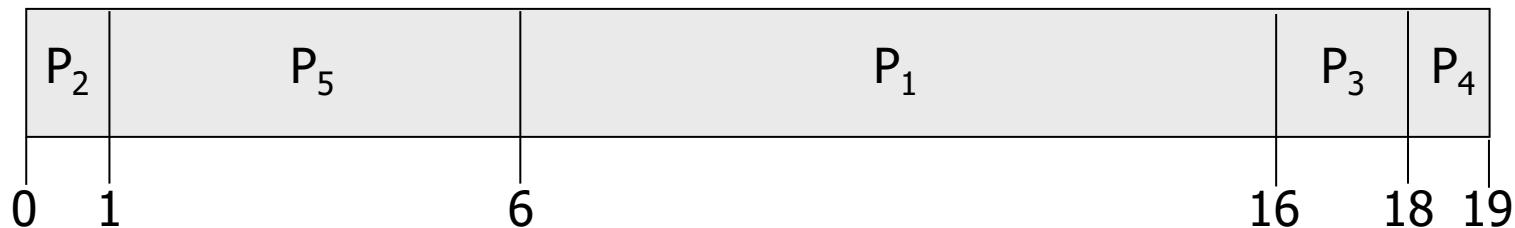
Scheduling a priorità (cont.)

❖ Esempio 6

Processo	Tempo di burst	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Priorità massima

- Il diagramma di Gantt è:



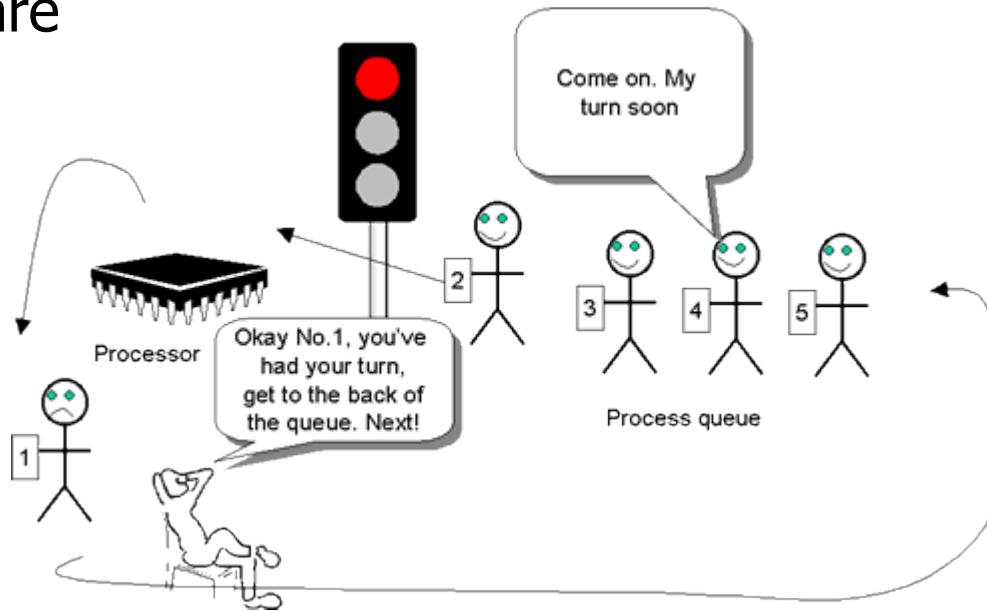
- Tempi di attesa: $P_1 \rightarrow 6$, $P_2 \rightarrow 0$, $P_3 \rightarrow 16$, $P_4 \rightarrow 18$, $P_5 \rightarrow 1$
- Tempo medio di attesa $T_a = (6+0+16+18+1)/5 = 8.2\text{msec}$





Scheduling Round Robin (RR)

- ❖ A ciascun processo viene allocata una piccola quantità di tempo di CPU, un *quanto di tempo* (*time slice*), generalmente 10–100 millisecondi
 - Dopo un quanto di tempo, il processo è forzato a rilasciare la CPU e viene accodato alla ready queue
 - La ready queue è trattata come una coda (FIFO) circolare

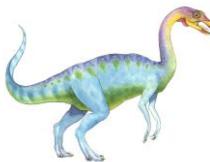




Scheduling RR (cont.)

- ❖ Se vi sono n processi nella ready queue ed il quanto di tempo è q , ciascun processo occupa $1/n$ del tempo di CPU in frazioni di, al più, q unità di tempo; nessun processo attende per più di $(n - 1) \times q$ unità di tempo
- ❖ Il timer interrompe la CPU allo scadere del quanto per programmare il prossimo processo
- ❖ Prestazioni:
 - q grande \Rightarrow FCFS
 - q piccolo \Rightarrow q deve essere grande rispetto al tempo di context switch (che, tuttavia, è normalmente $< 10\mu\text{sec}$), altrimenti l'overhead è troppo alto



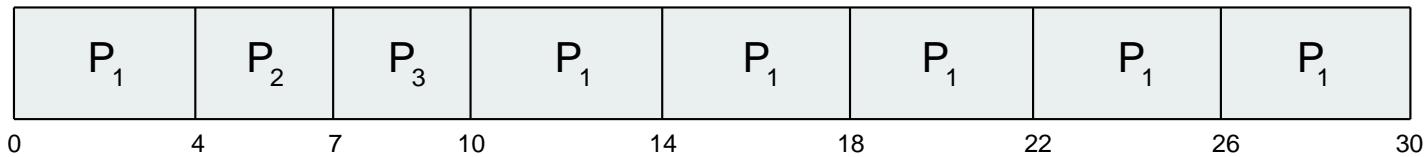


Scheduling RR (cont.)

❖ Esempio 7

Processo	Tempo di burst
P ₁	24
P ₂	3
P ₃	3

- Il diagramma di Gantt con $q = 4$ è:



- In genere si ha un tempo medio di attesa e di turnaround maggiore rispetto a SJF, tuttavia si ottiene un miglior tempo medio di risposta
- Tempi di attesa: P₁→6, P₂→4, P₃→7
- Tempo medio di attesa T_a=(6+4+7)/3=5. $\overline{6}$ msec



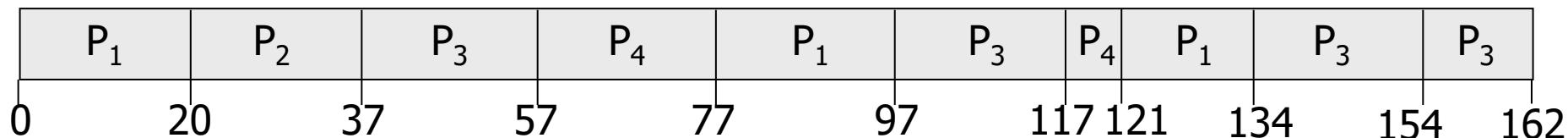


Scheduling RR (cont.)

❖ Esempio 8

Processo	Tempo di burst
P ₁	53
P ₂	17
P ₃	68
P ₄	24

- Il diagramma di Gantt con $q = 20$ è:

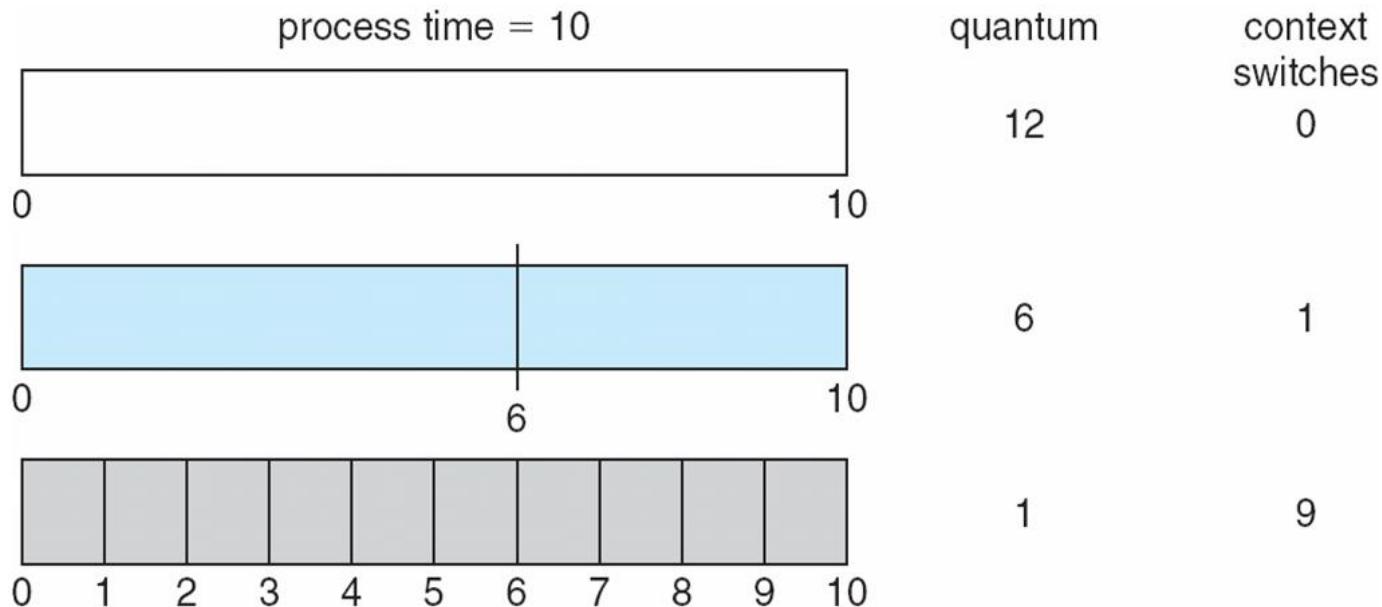


- Tempi di attesa: $P_1 \rightarrow 81$, $P_2 \rightarrow 20$, $P_3 \rightarrow 94$, $P_4 \rightarrow 97$
- Tempo medio di attesa $T_a = (81 + 20 + 94 + 97) / 4 = 73\text{msec}$





Quanto di tempo e tempo di context-switch

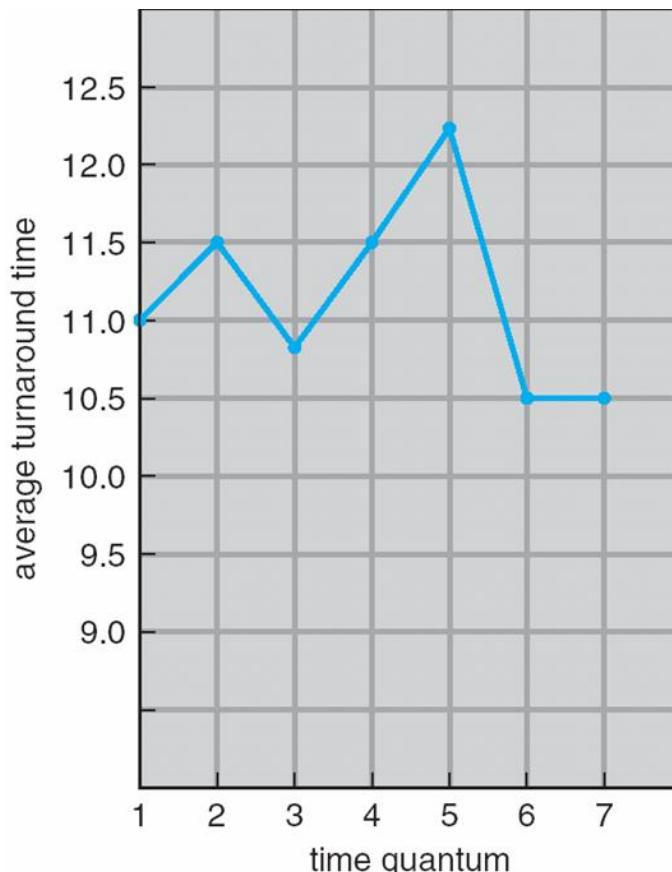


Un quanto di tempo minore incrementa il numero di context switch





Quanto di tempo e tempo di turnaround



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Empiricamente: il quanto di tempo deve essere più lungo dell'80% dei CPU burst

Variazione del tempo medio di turnaround in funzione della lunghezza del quanto di tempo



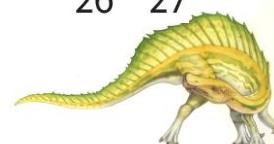
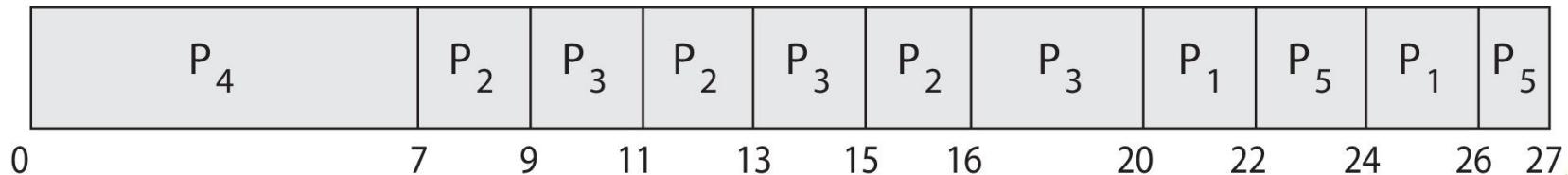


Scheduling a priorità con RR

❖ Esempio 9

Processo	Tempo di burst	Priorità
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

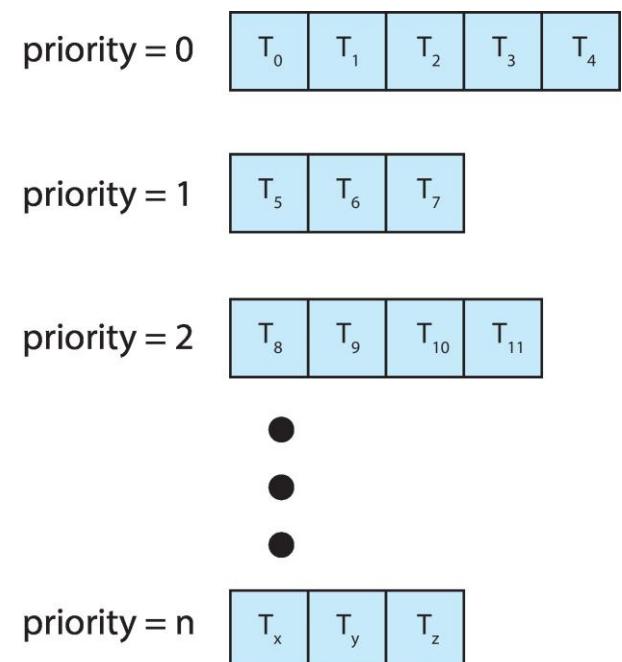
- Viene eseguito il processo a priorità maggiore; a parità di priorità, si usa RR
- Il diagramma di Gantt con $q = 2$ è:





Scheduling con code multiple – 1

- ❖ Nello scheduling a priorità tutti i processi possono essere collocati in una singola coda, da cui lo scheduler seleziona il prossimo processo da eseguire
 - La ricerca può costare $\mathcal{O}(n)$
- ❖ Nella pratica, risulta più conveniente organizzare i processi in code distinte
 - Lo scheduler assegna la CPU al processo “in testa” alla coda a priorità più alta
- ❖ L’approccio funziona bene anche per lo scheduling combinato con RR
 - Se ci sono più processi in una coda con una data priorità, questi verranno eseguiti con scheduling circolare





Scheduling con code multiple – 2

❖ Esempio 10

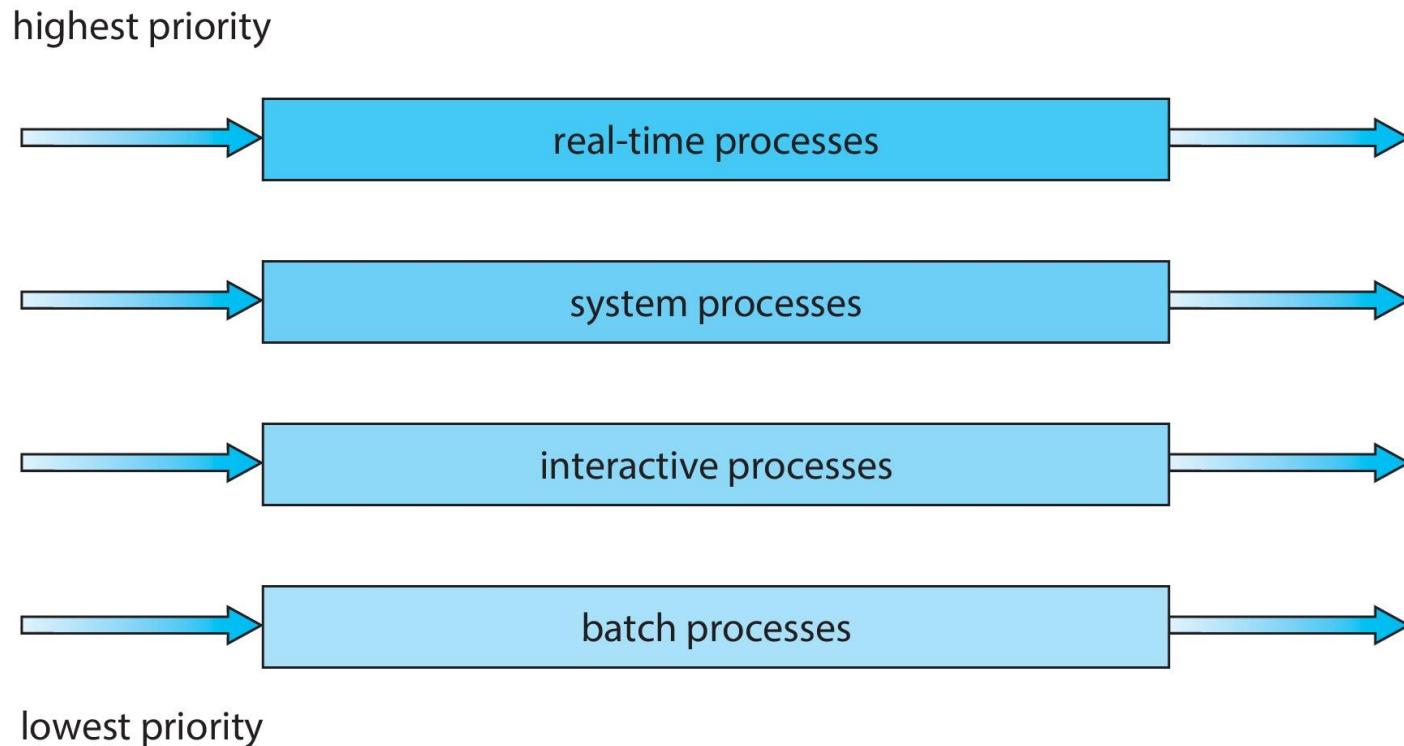
- La ready queue è suddivisa nelle due code
 - ▶ **foreground** (interattiva)
 - ▶ **background** (batch)
- Ogni coda ha il suo proprio algoritmo di scheduling
 - ▶ foreground: RR
 - ▶ background: FCFS
- È necessario effettuare lo scheduling tra le code
 - ▶ *Scheduling a priorità fissa*: si servono tutti i processi foreground poi quelli background
⇒ Rischio di starvation
 - ▶ *Time slice*: ciascuna coda occupa un certo tempo di CPU che suddivide fra i propri processi; ad esempio...
 - 80% per foreground con RR
 - 20% per background con FCFS

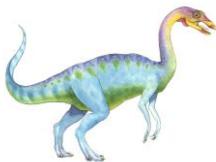




Scheduling con code multiple – 3

- ❖ I processi si assegnano in modo permanente ad una coda, generalmente secondo qualche caratteristica (invariante) del processo

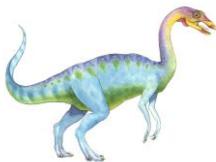




Code multiple con feedback – 1

- ❖ Se la priorità viene ricalcolata dinamicamente in base al comportamento dei processi...
- ❖ I processi possono spostarsi fra le varie code; si può implementare l'aging
- ❖ Lo scheduler a code multiple con feedback è definito dai seguenti parametri:
 - Numero di code
 - Algoritmo di scheduling per ciascuna coda
 - Metodo impiegato per determinare quando spostare un processo in una coda a priorità maggiore
 - Metodo impiegato per determinare quando spostare un processo in una coda a priorità minore
 - Metodo impiegato per determinare in quale coda deve essere posto un processo quando entra nel sistema





Code multiple con feedback – 2

❖ Esempio 11

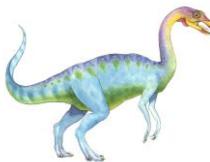
- Tre code:

- ▶ Q_0 – RR con quanto di tempo di 8 millisecondi
- ▶ Q_1 – RR con quanto di tempo di 16 millisecondi
- ▶ Q_2 – FCFS

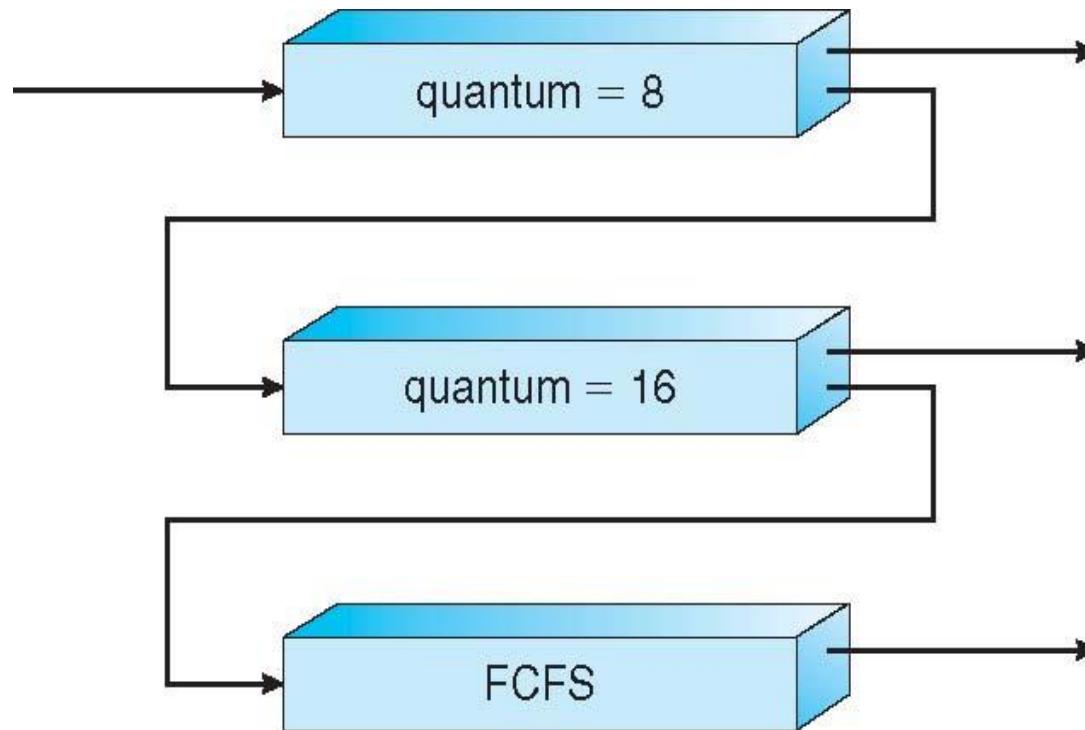
❖ Scheduling

- Un nuovo processo viene immesso nella coda Q_0 , che è servita con RR; quando prende possesso della CPU il processo riceve 8 millisecondi; se non termina, viene spostato nella coda Q_1
- Nella coda Q_1 il processo è ancora servito RR e riceve ulteriori 16 millisecondi; se ancora non ha terminato, viene spostato nella coda Q_2 , dove verrà servito con criterio FCFS all'interno dei cicli di CPU lasciati liberi dai processi delle code Q_0 e Q_1



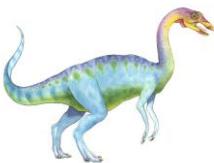


Code multiple con feedback – 3



- ❖ L'idea sottesa all'algoritmo di scheduling è quella di separare i processi in base alle loro caratteristiche d'uso della CPU
⇒ Massima priorità con CPU-burst brevi





Esempio 1

❖ Esercizio

Un insieme di task indipendenti, A, B, C e D, deve essere eseguito su di un unico processore. I task possono venire elaborati più di una volta. Appena un task ha terminato la propria esecuzione, è pronto per la successiva (che non necessariamente durerà lo stesso tempo). I tempi di arrivo e di esecuzione sono descritti nella seguente tabella:

Task	Tempo di arrivo	1° esecuzione	2° esecuzione	3° esecuzione
A	0	10	6	4
B	2	3	2	2
C	3	2	—	—
D	5	1	1	—

Si tracci il diagramma di Gantt per lo scheduling FCFS e si calcoli il tempo medio di attesa. Lo scheduling generato soffre dell'*effetto convoglio*? Motivare la risposta data.



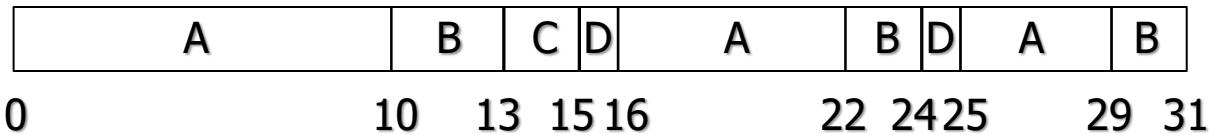


Esempio 1

❖ Soluzione

Task	Tempo di arrivo	1° esecuzione	2° esecuzione	3° esecuzione
A	0	10	6	4
B	2	3	2	2
C	3	2	—	—
D	5	1	1	—

Diagramma di Gantt

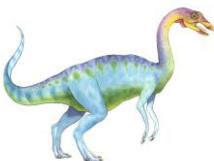


$$T_a(A)=9, T_a(B)=22, T_a(C)=10, T_a(D)=18$$

$$T_a = (9+22+10+18)/4 = 14.75 \text{ msec}$$

Si verifica l'effetto convoglio perché il processo A, che viene sempre eseguito per primo, ha burst significativamente più lunghi di tutti gli altri processi.





Esempio 2

❖ Esercizio

Si considerino i processi P_1 , P_2 e P_3 . Ciascun processo esegue un CPU-burst ed un I/O-burst, quindi nuovamente un CPU-burst ed un I/O-burst ed infine un ultimo CPU-burst. La lunghezza dei burst ed il tempo di arrivo dei processi (in millisecondi) è riportato in tabella:

Processo	Burst1	I/O_1	Burst2	I/O_2	Burst3	Arrivo
P_1	2	4	2	2	2	0
P_2	2	2	3	3	1	1
P_3	1	2	1	1	1	1

Si disegnino i diagrammi di Gantt che illustrano l'esecuzione dei tre processi utilizzando FCFS ed RR con quanto di tempo pari a 2. Se il termine di un servizio di I/O ed un timeout della CPU si verificano nello stesso istante, si assegna la precedenza al processo che ha appena terminato il proprio I/O. Si calcoli il tempo medio di attesa ed il tempo medio di turnaround nei due casi.



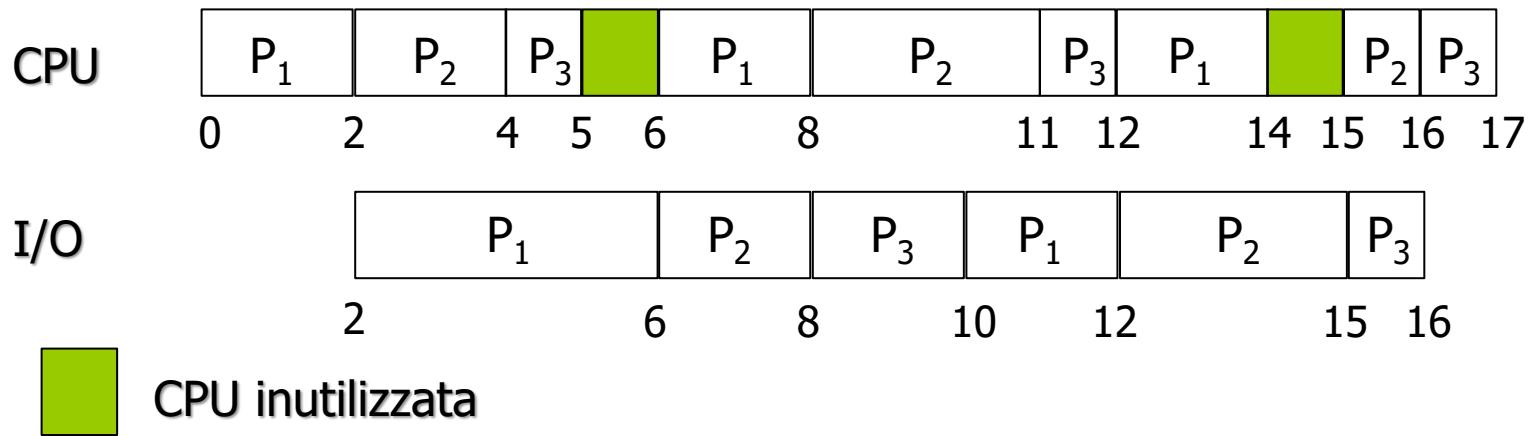


Esempio 2

❖ Soluzione FCFS

Diagrammi di Gantt

Processo	Burst1	I/O_1	Burst2	I/O_2	Burst3	Arrivo
P ₁	2	4	2	2	2	0
P ₂	2	2	3	3	1	1
P ₃	1	2	1	1	1	1

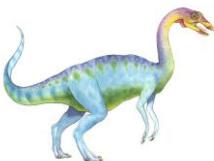


$$T_a = (0+1+4)/3 = 1.\overline{6} \text{ msec}$$

$$T_t = (14+15+16)/3 = 15 \text{ msec}$$

- ❖ Nota: Nel tempo di attesa deve essere considerato esclusivamente il tempo trascorso dai processi nella ready queue (senza considerare il tempo di attesa/servizio di I/O)



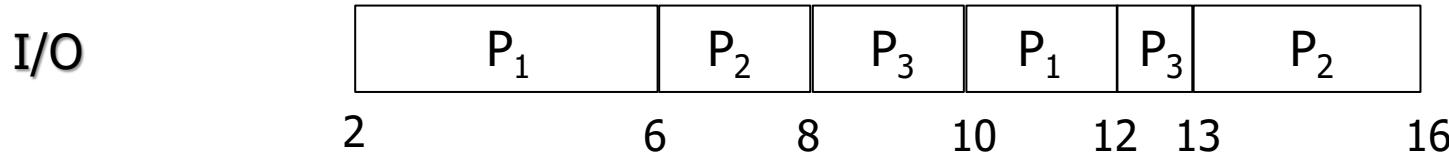
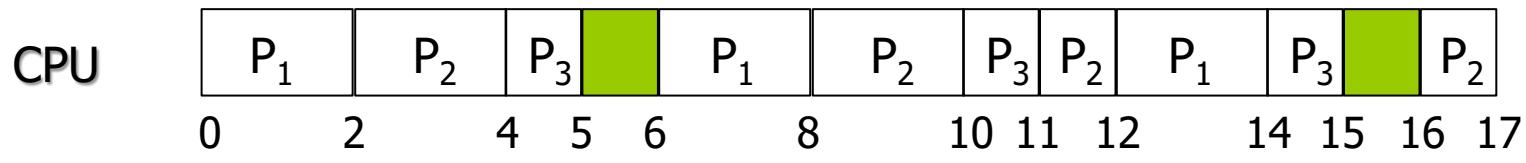


Esempio 2

❖ Soluzione RR, q=2

Diagrammi di Gantt

Processo	Burst1	I/O_1	Burst2	I/O_2	Burst3	Arrivo
P ₁	2	4	2	2	2	0
P ₂	2	2	3	3	1	1
P ₃	1	2	1	1	1	1



CPU inutilizzata

$$T_a = (0+2+4)/3 = 2\text{msec}$$

$$T_t = (14+16+14)/3 = 14.\overline{6}\text{msec}$$





Esempio 3

❖ Esercizio

Si considerino 5 processi (A,B,C,D,E) tali che:

Job	Tempo di arrivo	CPU burst	Priorità	Priorità massima
A	0	8	2	
B	1	6	3	
C	2	3	2	
D	3	4	2	
E	4	3	2	

Descrivere la sequenza di esecuzione dei job (tramite Gantt chart) e calcolare il tempo totale di completamento ottenuto con uno scheduling a code multiple a priorità e feedback, dove ogni coda è gestita con la strategia FCFS. Il feedback è definito come segue: la priorità diminuisce di 1 (fino al livello base 1) se si passano più di 6 unità di tempo consecutive in esecuzione nella CPU ed aumenta di 1 per ogni 6 unità di tempo passate in attesa in una qualsiasi coda.

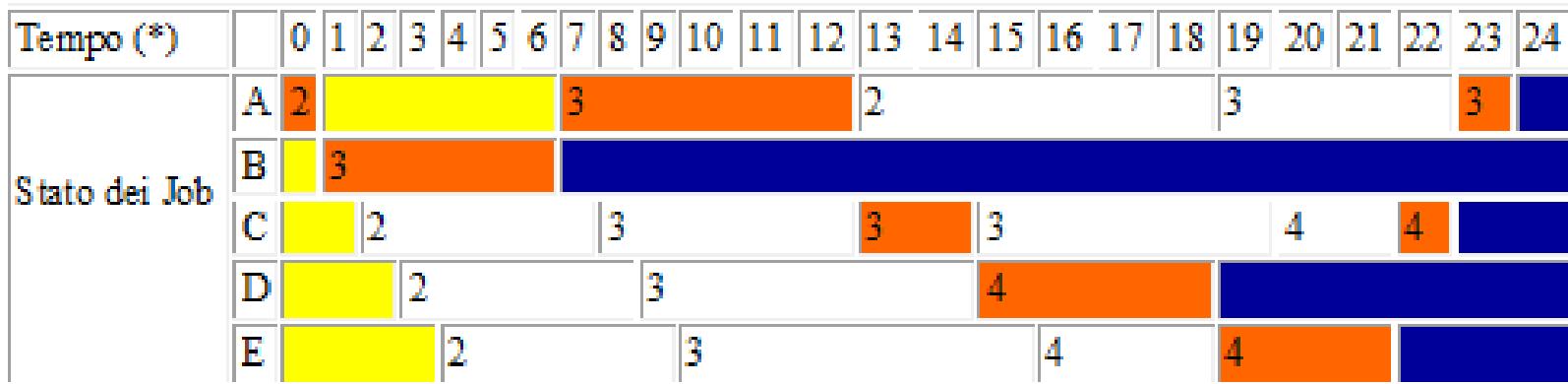




Esempio 3

❖ Soluzione

Job	Tempo di arrivo	CPU burst	Priorità
A	0	8	2
B	1	6	3
C	2	3	2
D	3	4	2
E	4	3	2

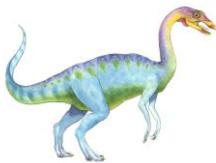


❖ Istante di completamento:

A→24, B→7, C→23, D→19, E→22

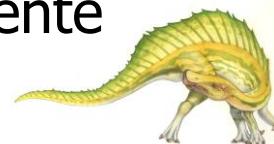
❖ Nota: La preemption avviene al tempo 13 – C sostituisce A – e al 15 – D sostituisce C. Al tempo 19, la coda di priorità 4 contiene solo E, mentre C entra in coda al tempo 20

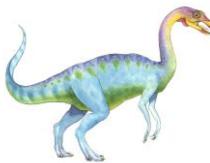




Scheduling dei thread – 1

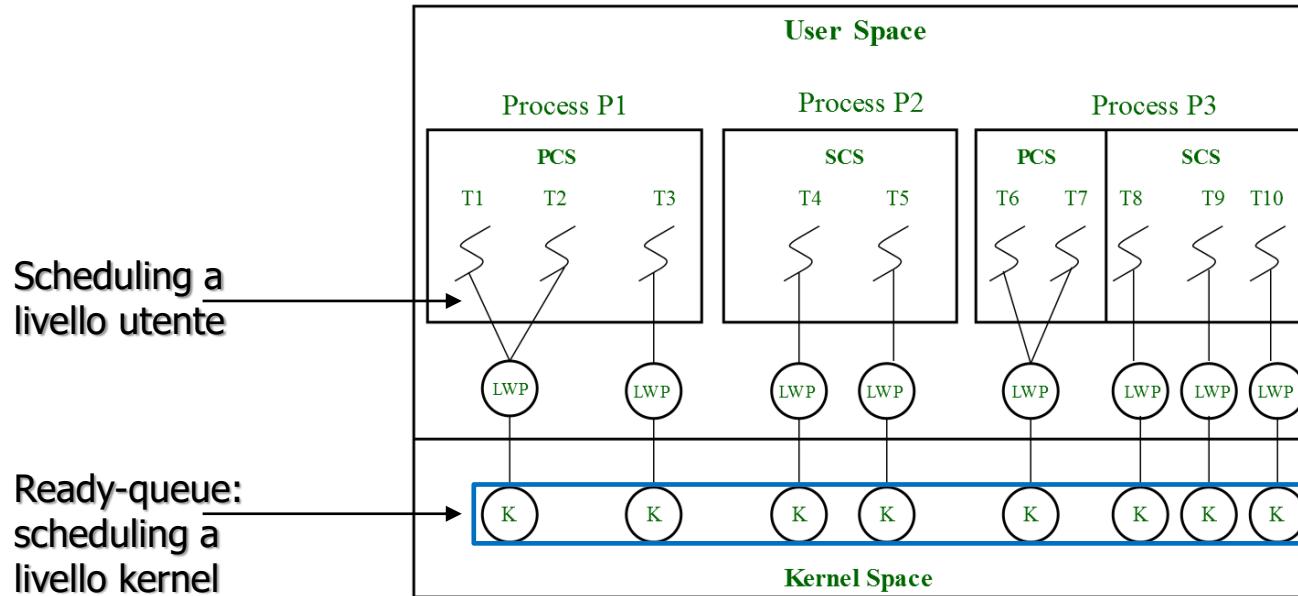
- ❖ Esiste una distinzione fondamentale fra lo scheduling dei thread a livello utente e a livello kernel ma, certamente, se i thread sono supportati dal sistema, lo scheduling avviene a livello di thread
- ❖ **Scheduling locale:** nei modelli multi-a-uno e multi-a-molti, è l'algoritmo con cui la libreria dei thread decide quale thread (a livello utente) allocare a un LWP disponibile (*ambito della competizione ristretto al processo, PCS*) – normalmente un algoritmo a priorità con prelazione definito dal programmatore
 - La libreria dei thread pianifica l'esecuzione su LWP
 - Tuttavia, il thread prescelto non è effettivamente in esecuzione su una CPU fisica finché il SO non pianifica l'esecuzione del thread a livello kernel corrispondente





Scheduling dei thread – 2

- ❖ **Scheduling globale:** l'algoritmo con cui il kernel decide quale thread a livello kernel dovrà venire eseguito successivamente (*ambito di competizione allargato al sistema, SCS*)
- ❖ I sistemi operativi caratterizzati dall'impiego del modello uno-a-uno (quali Windows, Linux e Mac OS) pianificano i thread unicamente con scheduling globale





API Pthreads per lo scheduling dei thread – 1

- ❖ La API Pthreads di POSIX consente di specificare **PCS** (Process–Contetion–Scope) o **SCS** (System–Contetion–Scope) nella fase di generazione dei thread
- ❖ Per specificare l'ambito della contesa, Pthreads usa i valori:
 - PTHREAD_SCOPE_PROCESS**: scheduling PCS
 - PTHREAD_SCOPE_SYSTEM**: scheduling SCS
- ❖ Nei sistemi con modello da molti–a–molti:
 - La politica **PTHREAD_SCOPE_PROCESS** pianifica i thread a livello utente sugli LWP disponibili
 - La politica **PTHREAD_SCOPE_SYSTEM** crea, in corrispondenza del thread a livello utente, un LWP ad esso vincolato, realizzando in effetti una corrispondenza secondo il modello da uno–a–uno



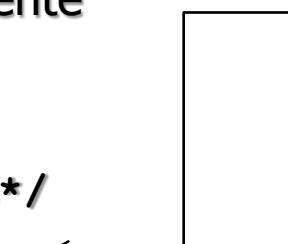


API Pthreads per lo scheduling dei thread – 2

Programma che determina l'ambito della contesa in vigore e lo imposta a `PTHREAD_SCOPE_SYSTEM`; si creano quindi 5 thread distinti in esecuzione con modello SCS

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* Ottiene gli attributi di default */
    pthread_attr_init(&attr);
    /* Per prima cosa, appura l'ambito della contesa*/
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Impossibile stabilire l'ambito della contesa \n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            printf(stderr, "Valore non ammesso.\n");
    }
}
```

Se restituisce un valore diverso da 0 la funzione non è stata eseguita correttamente





API Pthread per lo scheduling dei thread – 3

```
/* Imposta l'algoritmo di scheduling a SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* Genera i thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* Il processo padre aspetta la terminazione di tutti i thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Ciascun thread inizia l'esecuzione da questa funzione */
void *runner(void *param)
{
    /* fai qualcosa... */
    pthread_exit(0);
}
```





Scheduling multiprocessore – 1

- ❖ Lo scheduling diviene più complesso quando nel sistema di calcolo sono presenti più CPU
- ❖ Le architetture multiprocessore possono essere:
 - **CPU multicore**
 - **Multicore multithread**
 - **Sistemi NUMA**
 - **Sistemi eterogenei**
- ❖ **Ipotesi:**
 - Nei primi tre casi, le unità di elaborazione sono, in relazione alle loro funzioni, identiche – **sistemi omogenei**
⇒ **Ripartizione del carico di lavoro**
 - Sistemi eterogenei composti da processori con diverse caratteristiche hardware

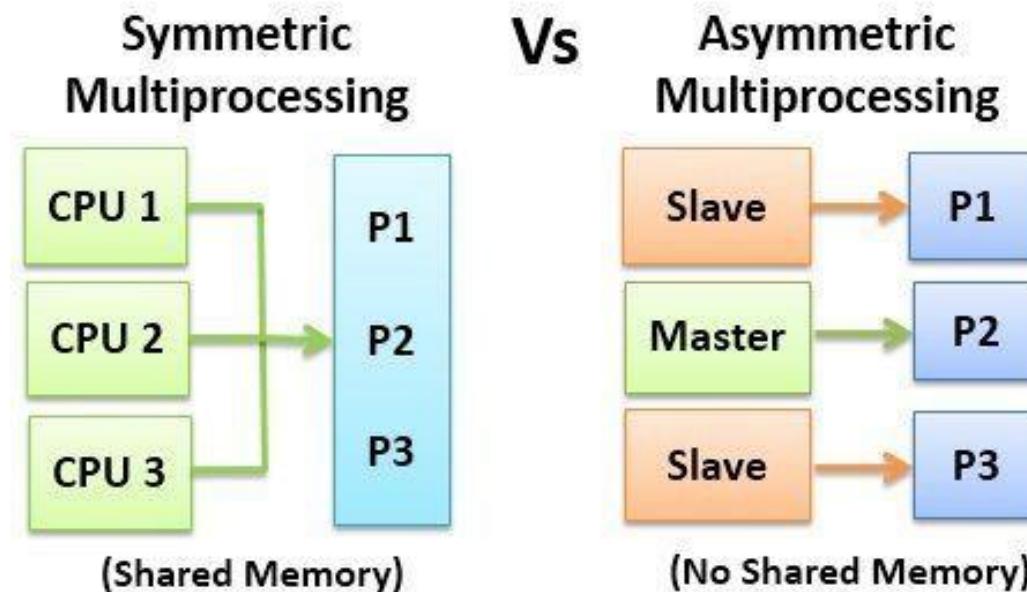


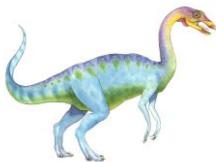


Scheduling multiprocessore – 2

❖ **Multielaborazione asimmetrica** – lo scheduling, l'elaborazione dell'I/O e, in genere, le attività di sistema sono affidate ad un solo processore, detto *master server*

- Si riduce la necessità di condividere dati, grazie all'accesso di un solo processore alle strutture dati del kernel
- Il master server costituisce un potenziale collo di bottiglia, capace di ridurre le prestazioni dell'intero sistema





Scheduling multiprocessore – 3

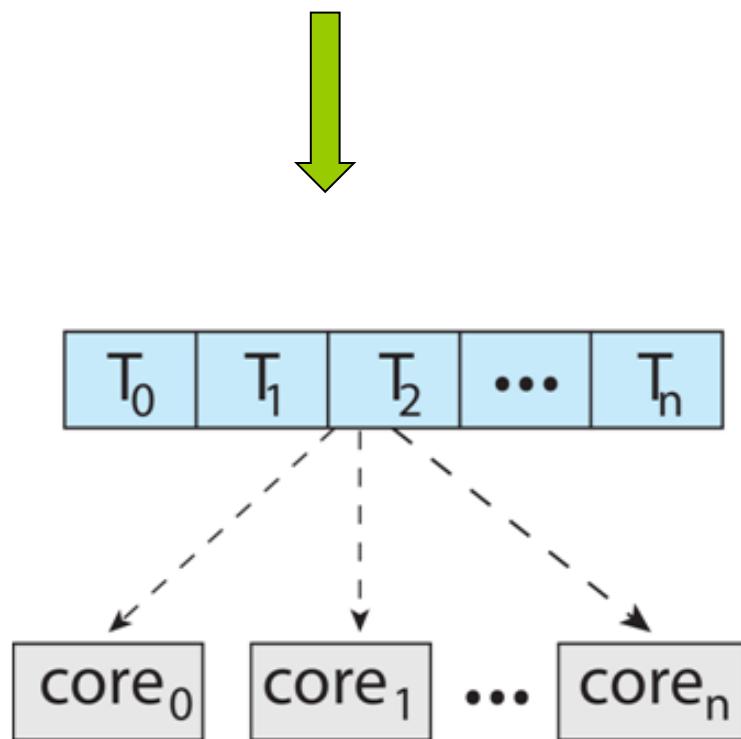
- ❖ **Multielaborazione simmetrica (SMP, *Symmetric Multi-Processing*)** – i thread pronti vanno a formare una coda comune oppure vi è un'apposita coda per ogni processore; ciascun processore ha un proprio scheduler che esamina la coda opportuna per prelevare il prossimo thread da eseguire
 - L'accesso concorrente di più processori ad una struttura dati comune rende delicata la programmazione degli scheduler che...
 - ▶ ...devono evitare di scegliere contemporaneamente lo stesso thread da eseguire
 - ▶ ...e devono evitare che qualche thread pronto “vada perso”
- ❖ Adottata da molti SO attuali, per esempio Windows, Mac OS, Linux, iOS e Android





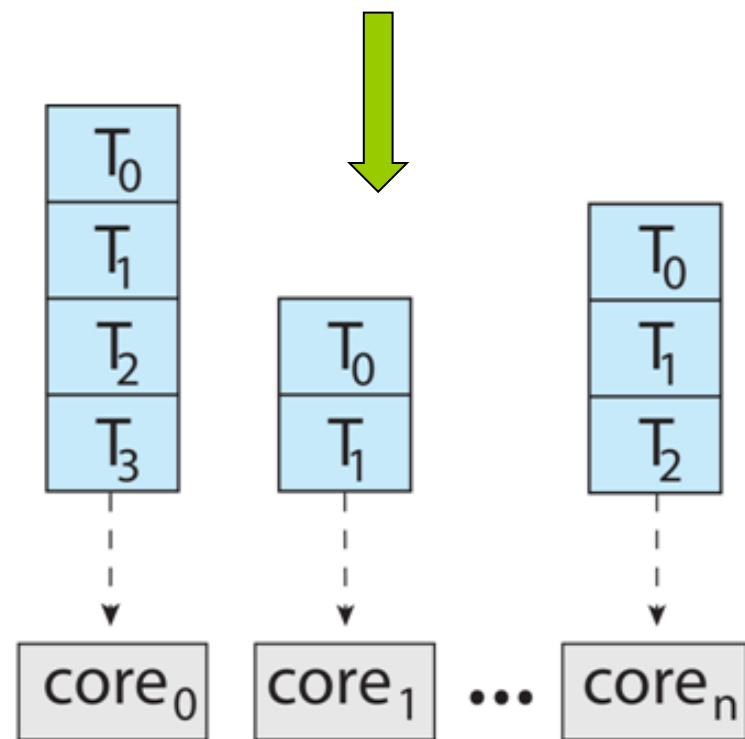
Scheduling multiprocessore – 4

Possibili race condition; l'accesso alla coda deve essere gestito tramite lock



common ready queue

Possibile sbilanciamento del carico di lavoro; uso più efficiente della cache



per-core run queues





Processori multicore – 1

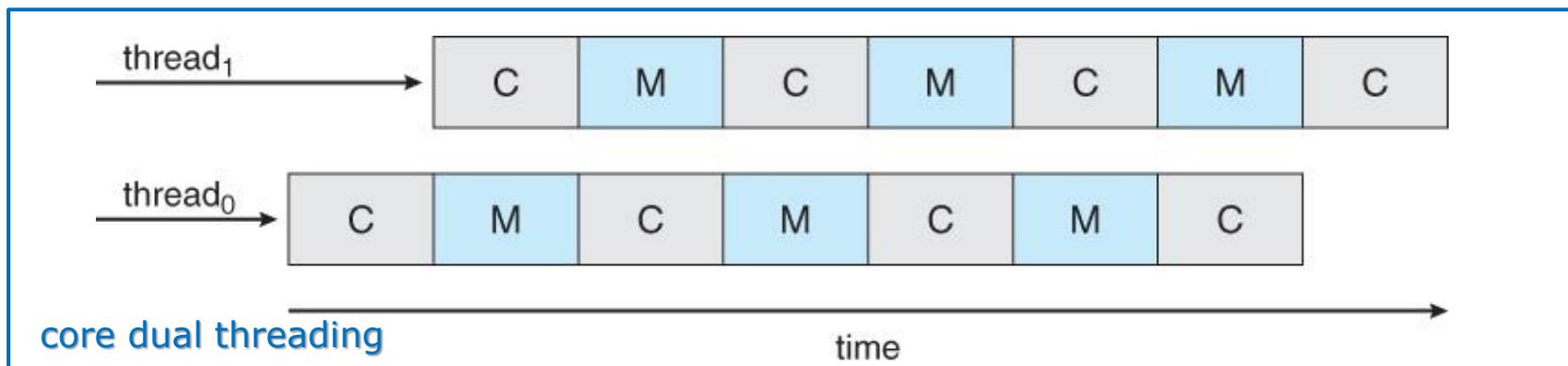
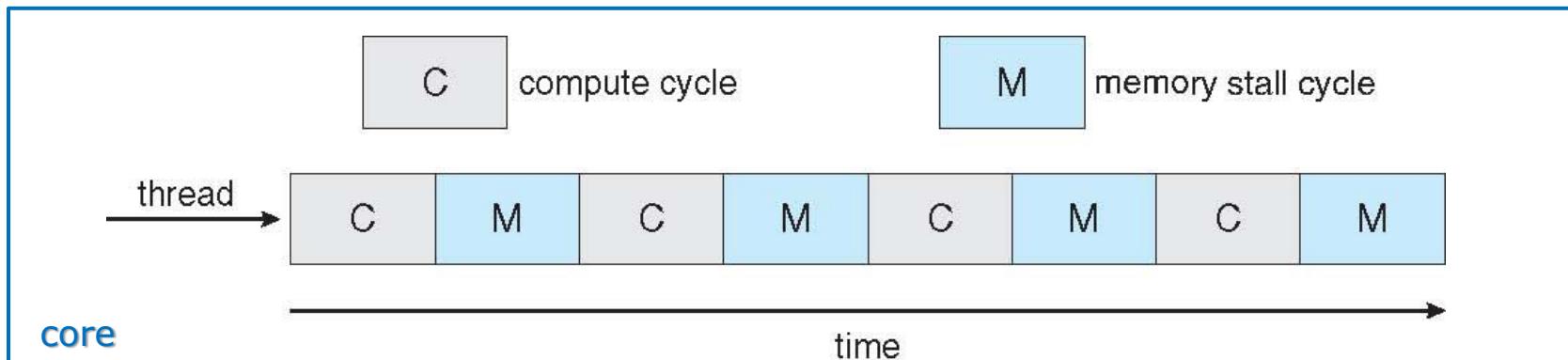
- ❖ Tradizionalmente i sistemi SMP hanno reso possibile la concorrenza fra thread con l'utilizzo di diversi processori fisici
- ❖ I sistemi SMP su processori multicore sono più veloci e consumano meno energia
- ❖ Quando un processore accede alla memoria (cache miss), una quantità significativa di tempo (fino al 50%) trascorre in attesa della disponibilità dei dati:
stallo della memoria
 - Progetti hardware recenti (*hyperthreading*, nella terminologia Intel) implementano unità di calcolo multithread, in cui due o più thread hardware sono assegnati ad un singolo core
- ❖ Dal punto di vista del SO, ogni thread hardware appare come un processore logico in grado di eseguire un thread software

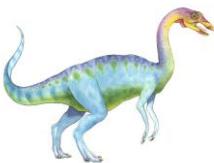




Processori multicore – 2

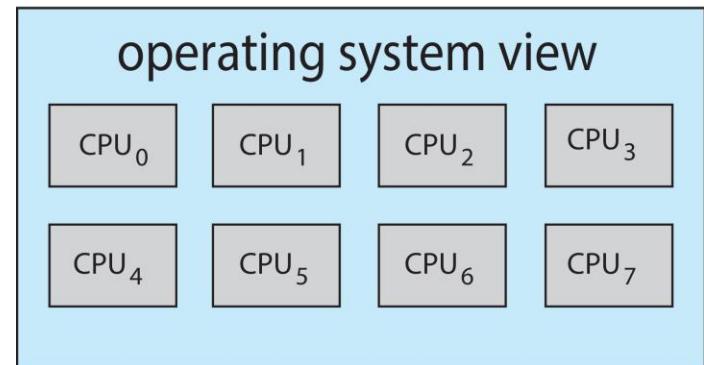
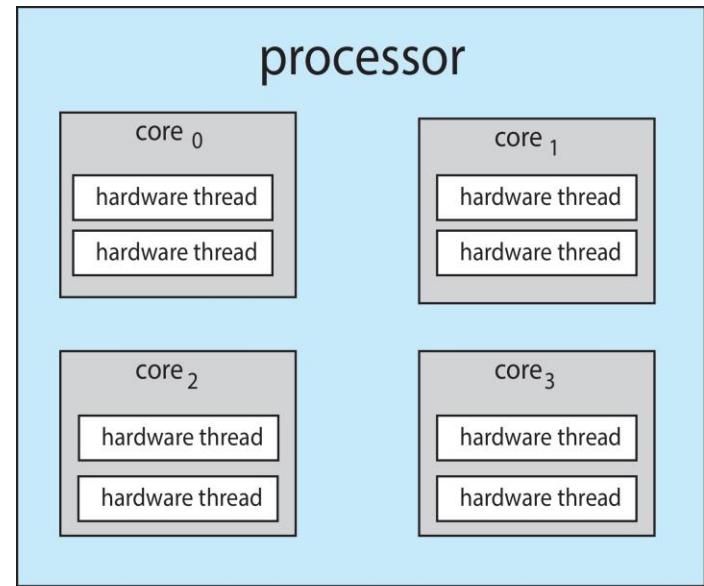
- ❖ **Esempio:** sistema dual core, dual threading, il SO vede quattro processori logici





Processori multicore – 3

- ❖ Nei **chip multithreading** (CMT), a ciascun core vengono assegnati un certo numero di thread hardware
- ❖ In un quad-core, con 2 thread hardware per core, si hanno 8 processori logici
- ❖ **Esempio:** Core i9–10900K di Intel, con 10 core, ha 20 thread quando l'hyperthreading è abilitato





Processori multicore – 4

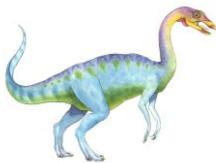
❖ Multithreading coarse–grained

- Il thread resta in esecuzione su un processore fino al verificarsi di un evento a lunga latenza (cache miss)
- Occorre ripulire la pipeline (tempo di latenza)

❖ Multithreading fine–grained

- Passa da un thread ad un altro al termine di un ciclo di istruzione o, in altre parole, il processore esegue una singola istruzione per ogni thread
- Necessaria una logica dedicata che garantisca una estrema rapidità nel context–switch





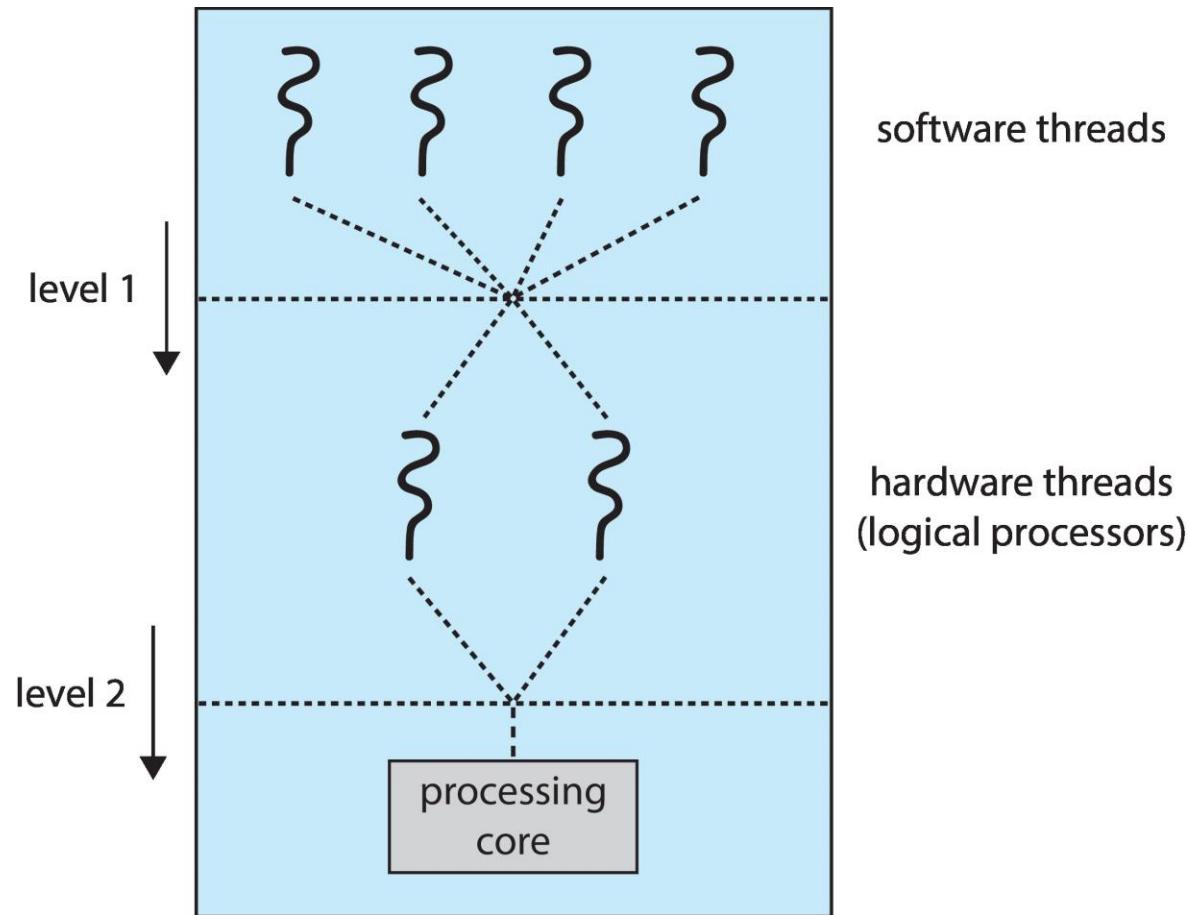
Processori multicore – 5

- ❖ Un processore multicore e multithread richiede due livelli di scheduling
 - Scheduling effettuato dal SO per stabilire quale thread software mandare in esecuzione su ciascun thread hardware → algoritmi standard
 - Scheduling relativo a ciascun core per decidere quale thread hardware mandare in esecuzione
 - ▶ Round Robin (UltraSPARC T3, 16 core a 8 thread)
 - ▶ Priorità o *urgency* (Intel Itanium, 2 core a 2 thread)
- ❖ Se il SO è consapevole del multithreading hardware può evitare di schedulare, qualora il sistema sia sufficientemente scarico, due thread software su due thread hardware relativi allo stesso core





Processori multicore – 6





Multielaborazione simmetrica – 1

- ❖ **Bilanciamento del carico** – Ripartire uniformemente il carico di lavoro sui diversi processori
 - Nei sistemi con ready queue comune è automatico
 - **Migrazione guidata** (*push migration*): un thread dedicato controlla periodicamente il carico dei processori per effettuare eventuali riequilibri
 - **Migrazione spontanea** (*pull migration*): un processore inattivo sottrae ad uno sovraccarico un thread in attesa
 - Linux le implementa entrambe: esegue il proprio algoritmo di bilanciamento ad intervalli regolari e ogniqualvolta si svuota la coda di attesa di un processore

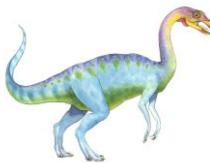




Multielaborazione simmetrica – 2

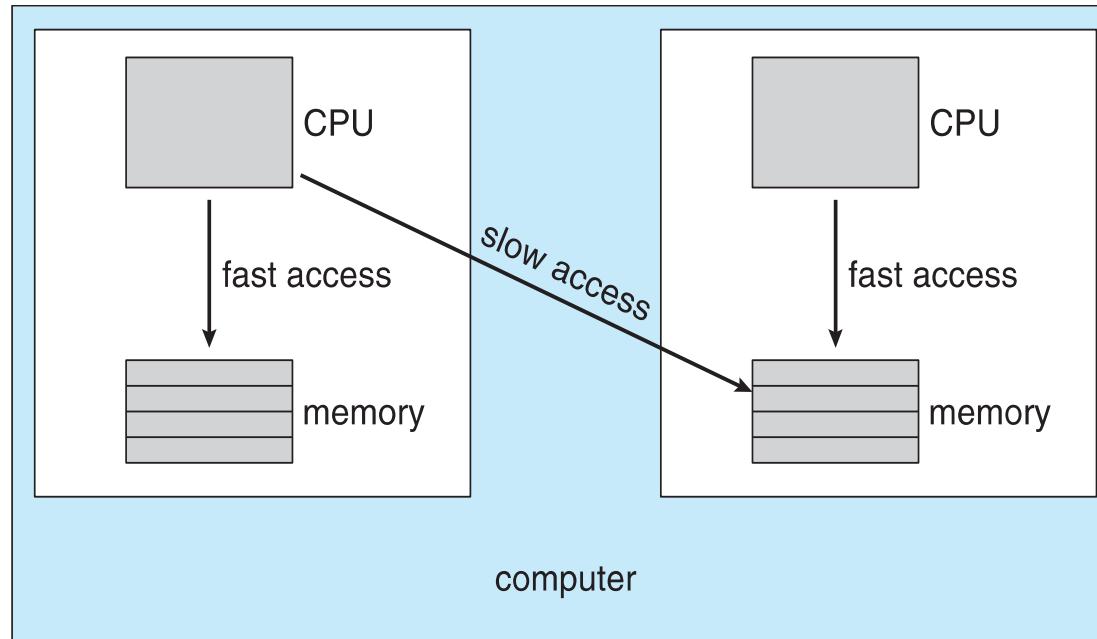
- ❖ **Predilezione per il processore** – Mantenere un thread in esecuzione sempre sullo stesso processore, per riutilizzare il contenuto della cache per burst successivi
 - **Predilezione debole** – soft affinity: il SO tenta di mantenere il thread su un singolo processore, ma la migrazione non è interdetta
 - **Predilezione forte** – hard affinity: il thread è vincolato all'esecuzione su uno o più processori
 - Linux implementa sia la **predilezione debole** che **forte**; per la **predilezione forte** dispone di chiamate di sistema (es., **`sched_setaffinity()`**) con cui specificare che un processo non può abbandonare un dato processore
 - Solaris supporta la **predilezione debole**: non garantisce che, per particolari condizioni di carico, i processi non subiscano spostamenti
- ❖ **Il bilanciamento del carico è antitetico rispetto alla filosofia di predilezione del processore**





NUMA e CPU scheduling

- ❖ L'architettura della memoria influenza la predilezione
 - In caso di **NUMA** (*Non Uniform Memory Access*), situazione standard in sistemi costituiti da diverse schede ognuna con una o più CPU e memoria, le CPU di una scheda accedono più velocemente alla memoria locale, rispetto alle memorie residenti sulle altre schede
 - ⇒ Ogni thread dovrebbe risiedere nella memoria locale al processore su cui viene eseguito





Multiprocessing eterogeneo – 1

❖ Sistemi (mobili) a multielaborazione eterogenea (HMP
– *Heterogeneous MultiProcessing*)

- Non seguono il paradigma del multiprocessing asimmetrico: tutte le attività possono essere eseguite su qualsiasi core
- Sebbene i sistemi mobili includano architetture multicore, sono spesso progettati utilizzando core che eseguono lo stesso set di istruzioni, ma differiscono sia per la velocità del clock che per la gestione energetica
- Possibilità di regolare il consumo energetico di un core fino a renderlo inattivo
- Per risparmiare energia, si assegna un task ad un determinato core in base alle sue specifiche esigenze





Multiprocessing eterogeneo – 2

- ❖ Nei processori ARM che la supportano, l'architettura HMP è detta **big.LITTLE** ed è costituita da grandi core con prestazioni elevate (**big**) combinati con piccoli core ad alta efficienza energetica (**LITTLE**)
 - Core big per applicazioni in foreground
 - Core LITTLE per applicazioni (lunghe) in background
 - Se il dispositivo mobile si trova in modalità di risparmio energetico, è possibile disabilitare i core big e forzare il sistema ad utilizzare solo i LITTLE
- ❖ Nell'iPhone 14 Pro, processore A16 Bionic con CPU con 6 core, con 2 core **performance** e 4 **efficiency** a ridotto consumo energetico

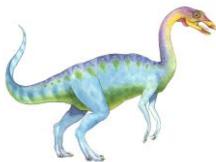




Scheduling real-time – 1

- ❖ **Sistemi hard real-time:** richiedono il completamento dei processi critici entro un intervallo di tempo predefinito e garantito
- ❖ **Sistemi soft real-time:** richiedono solo che ai processi critici sia assegnata una maggiore priorità rispetto ai processi di routine (nessuna garanzia sui tempi di completamento)
- ❖ I sistemi real-time, specialmente hard real-time, sono per loro natura guidati dagli eventi
 - Si attende che si verifichi un evento cui si deve reagire in tempo reale
 - **Latenza dell'evento:** tempo che intercorre tra l'occorrenza dell'evento e il momento in cui il sistema ne effettua la gestione

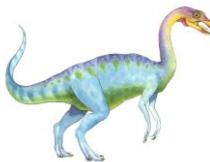




Scheduling real-time – 2

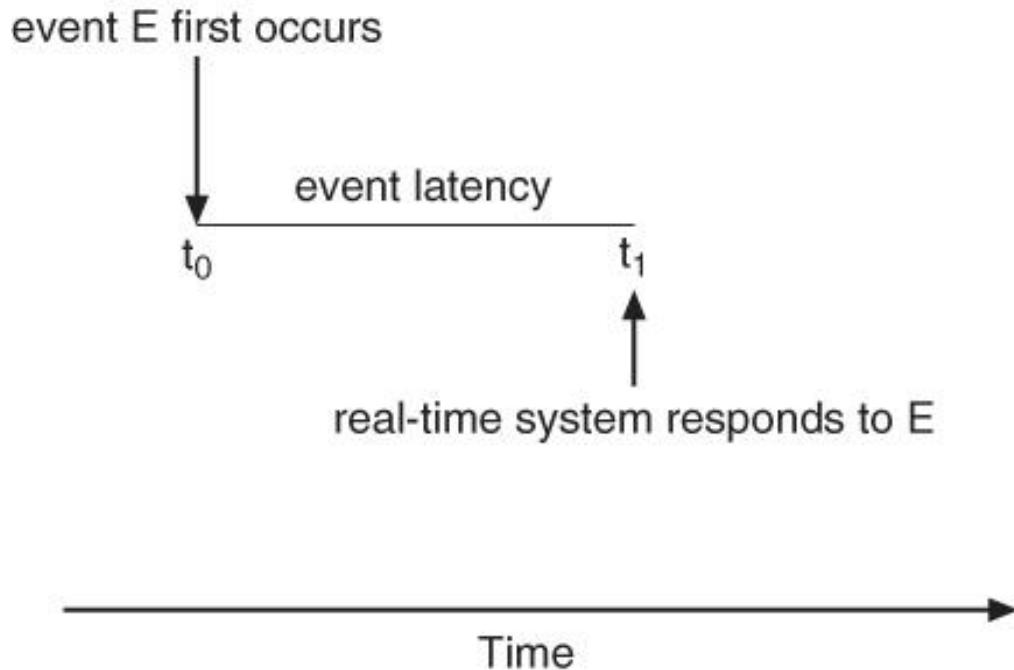
- ❖ Più in dettaglio: le applicazioni hard-real time richiedono un tempo di risposta agli eventi esterni che sia prevedibile
- ❖ Una tipica applicazione in tempo reale presume:
 - Un dispositivo che genera interruzioni
 - Una routine di servizio dell'interrupt che raccoglie dati dal dispositivo
 - Un codice a livello utente che elabora i dati raccolti
- ❖ La **reattività** (*responsiveness*) è una caratterizzazione della velocità con cui un SO ed il codice di gestione dell'evento sono in grado di reagire
- ❖ La latenza è quindi una misura della responsiveness





Scheduling real-time – 3

- ❖ Eventi diversi \Rightarrow diversi requisiti di latenza
 - ABS: 3–5 msec
 - Meccanismo di controllo radar su aerei di linea: <10 sec



Latenza relativa all'evento

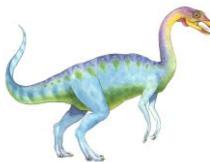




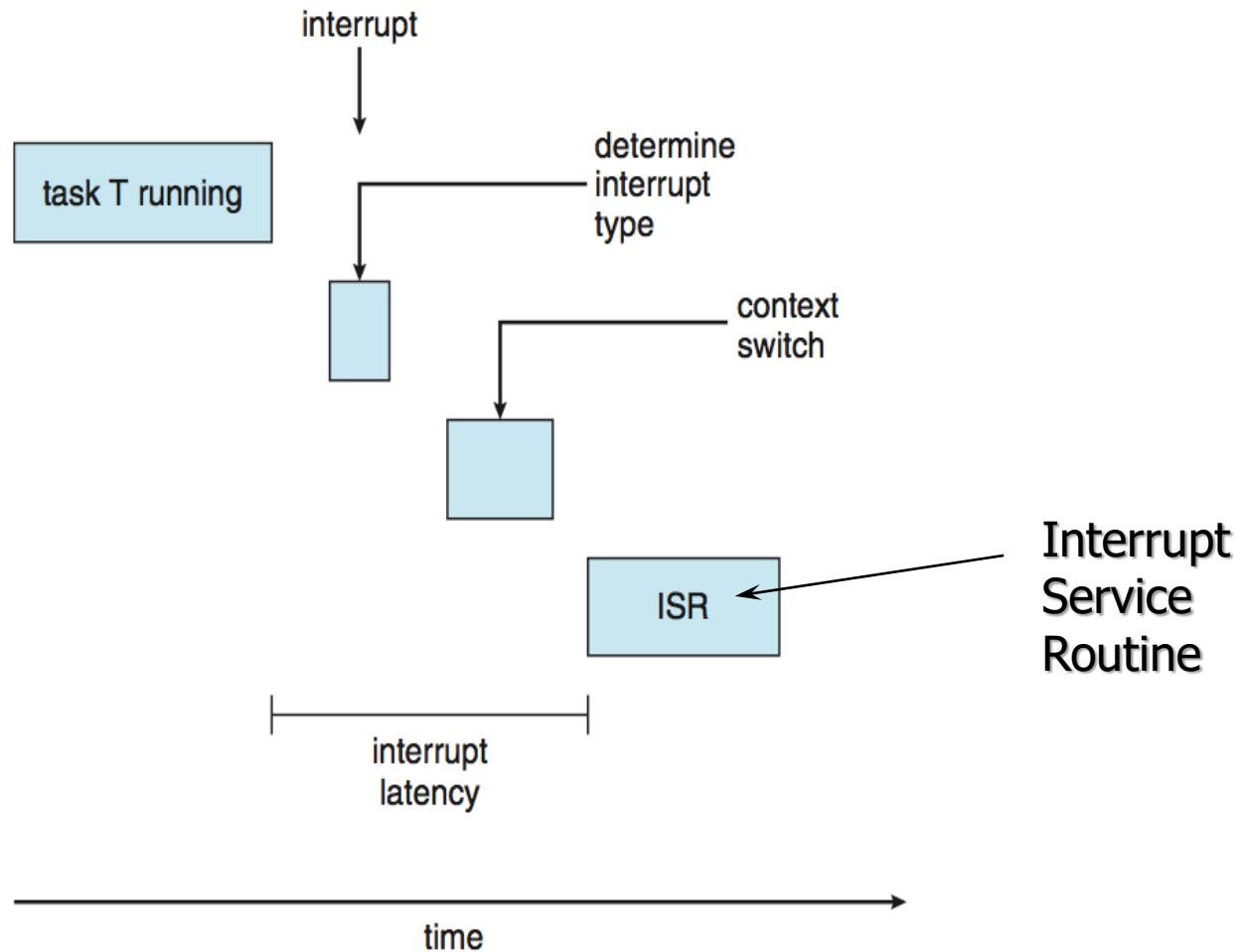
Scheduling real-time – 4

- ❖ Categorie di latenza che influiscono sul funzionamento dei sistemi real-time
 - **Latenza relativa alle interruzioni:** tempo compreso tra la notifica di un'interruzione alla CPU e l'avvio della routine che la gestisce
 - ▶ Fondamentale minimizzare l'intervallo di tempo in cui le interruzioni sono disattivate per l'aggiornamento dei dati del kernel
 - **Latenza di dispatch:** tempo necessario al dispatcher per interrompere un processo ed avviare il codice di gestione dell'evento





Scheduling real-time – 5



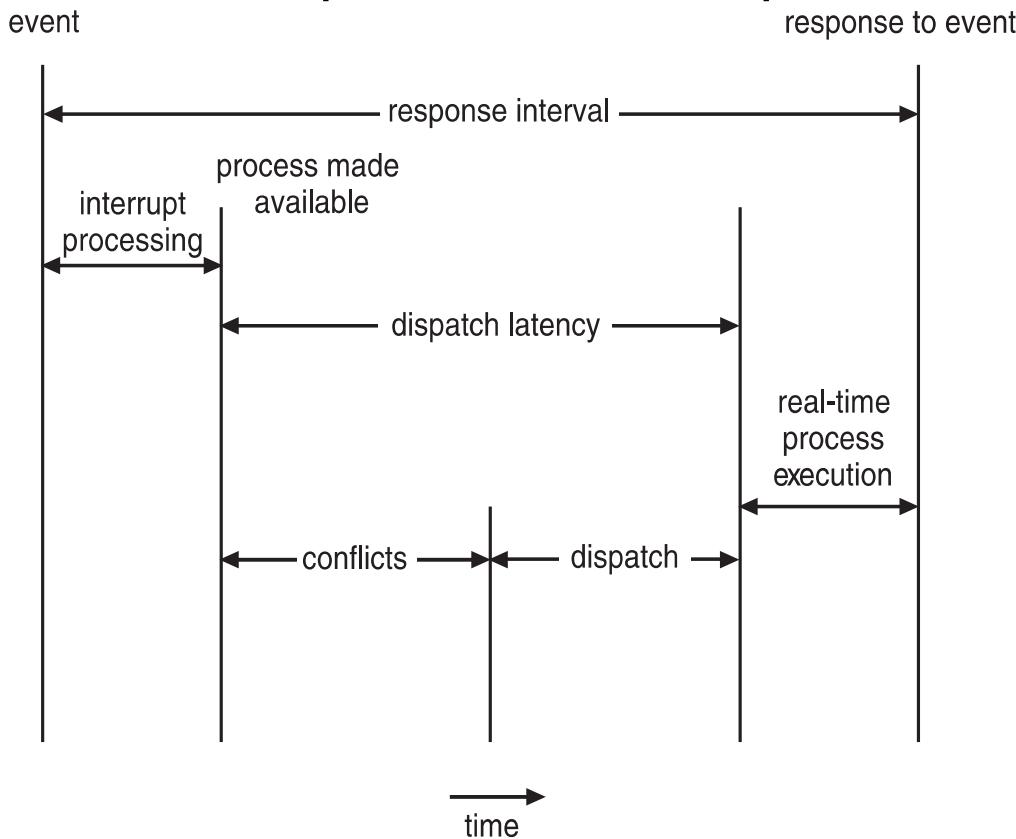
Latenza relativa alle interruzioni

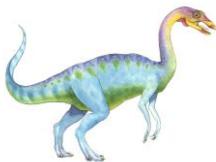




Scheduling real-time – 6

- ❖ La cosiddetta **fase di conflitto** nella latenza di dispatch consiste di due componenti
 - Prelazione di ogni processo in esecuzione nel kernel
 - Cessione, da parte dei processi a bassa priorità, delle risorse richieste dal processo ad alta priorità





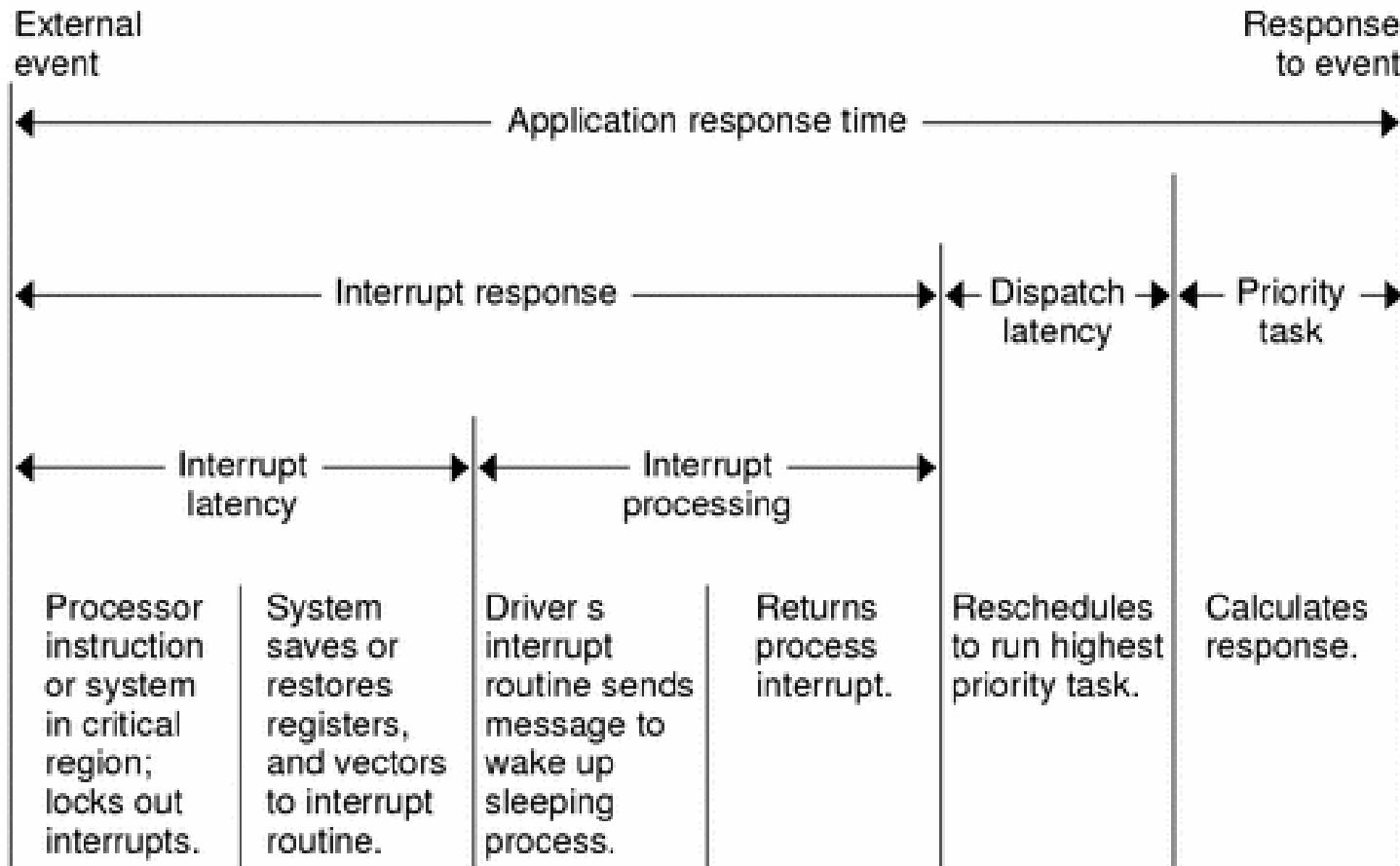
Scheduling real-time – 7

- ❖ La *event latency* complessiva è costituita quindi dal tempo di risposta all'interrupt, dalla latenza di dispatch e dall'esecuzione del task ad alta priorità
 - La latenza di interrupt è determinata dall'intervallo più lungo di tempo durante il quale il sistema esegue codice che impone che le interruzioni siano disabilitate
 - Durante l'elaborazione dell'interrupt, la routine di servizio cattura l'I/O e riattiva il processo ad alta priorità; il sistema rileva che è pronto un processo con priorità più elevata rispetto al processo interrotto e lo elabora
 - Il tempo per passare da un processo a priorità inferiore a un processo a priorità più alta è incluso nel tempo di latenza di dispatch





Scheduling real-time – 8





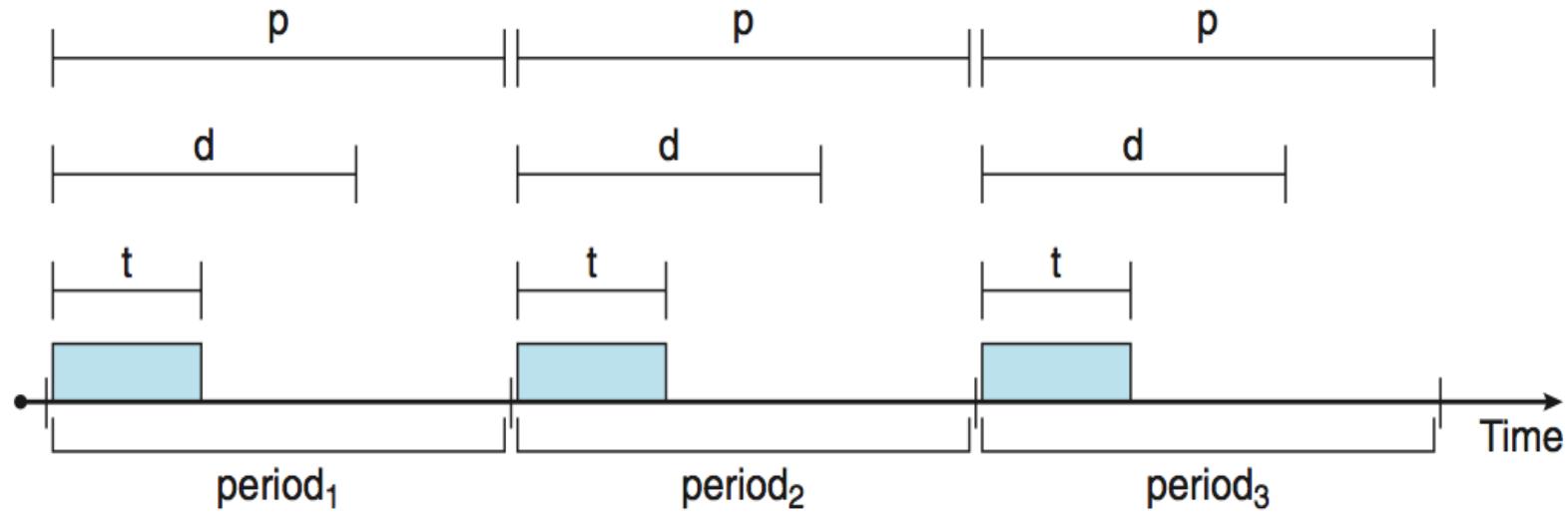
Scheduling real-time – 9

- ❖ Per realizzare lo scheduling real-time, lo scheduler deve implementare un algoritmo con prelazione basato su priorità
 - Si garantisce solo la funzionalità soft real-time
- ❖ Nel caso dell'hard real-time si deve anche garantire che le attività in tempo reale vengano servite rispettando le scadenze
 - Si considerino, per il momento, solo processi **periodici**, che richiedono la CPU a intervalli costanti di tempo (**periodi**); in particolare, ciascun processo...
 - ▶ ...ha un tempo di elaborazione fisso t , una scadenza d , e un periodo p
 - ▶ $0 \leq t \leq d \leq p$
 - ▶ La **frequenza** di un processo periodico è data da $1/p$





Scheduling real-time – 10



Processo periodico

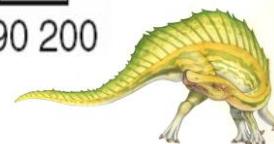
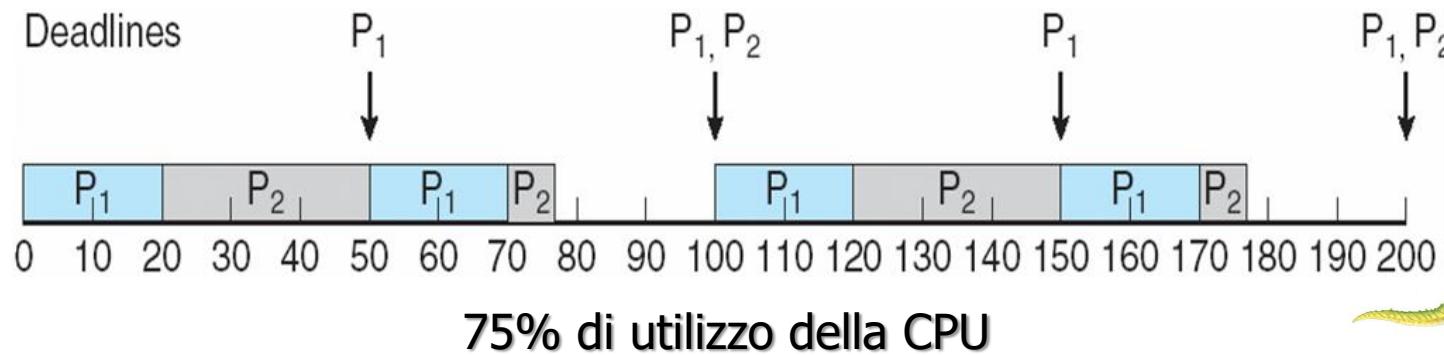




Scheduling hard real-time – 1

❖ Scheduling con priorità proporzionale alla frequenza (o *Rate Monotonic*)

- Si applica un modello statico di attribuzione delle priorità con prelazione
- La priorità è inversamente proporzionale al periodo
 - ▶ Si assegnano priorità più elevate ai processi che fanno uso più frequente della CPU
- **Esempio:** siano P_1 e P_2 due processi con periodi rispettivamente pari a 50 e 100; supponiamo che i tempi di elaborazione siano invece $t_1=20$ e $t_2=35$ e sia $d=p$
- P_1 ha priorità maggiore di P_2

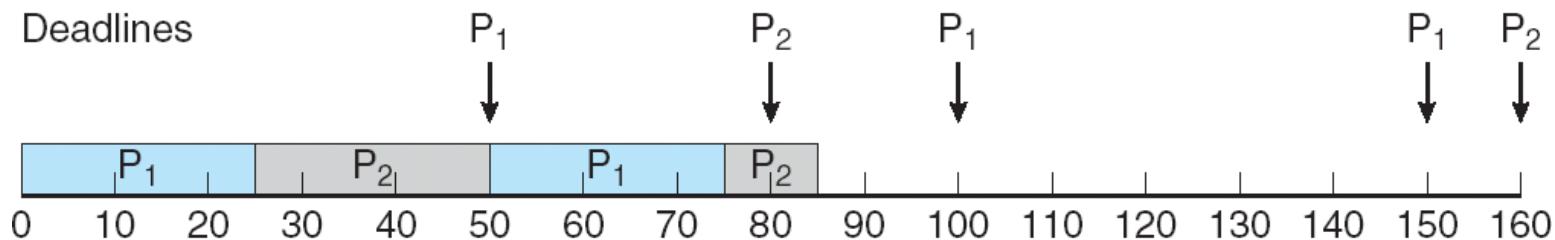




Scheduling hard real-time – 2

❖ Lo **scheduling con priorità proporzionale alla frequenza** è un algoritmo ottimale, nel senso che se non è in grado di pianificare l'esecuzione di una serie di processi rispettando i vincoli temporali, nessun altro algoritmo che assegna priorità statiche vi riuscirà

- **Esempio:** siano P_1 e P_2 due processi con periodi rispettivamente pari a 50 e 80; Supponiamo che i tempi di elaborazione siano invece $t_1=25$ e $t_2=35$
- P_1 ha ancora priorità maggiore di P_2 , ma...



- ...in questo caso la scadenza relativa a P_2 non può essere rispettata (anche se si ha il 94% di utilizzo della CPU)





Scheduling hard real-time – 3

- ❖ Nel caso di task periodici, se esistono m task e se il task i arriva con periodo P_i e richiede C_i secondi di tempo di CPU, il carico può essere gestito solo se:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

- ❖ Un sistema real-time che rispetta questo vincolo è detto **schedulabile**





Scheduling hard real-time – 4

- ❖ Tuttavia, è possibile dimostrare (Liu & Layland, 1973) che, per ogni sistema con processi periodici, il funzionamento dello scheduling con priorità proporzionale alla frequenza è garantito se:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m \cdot (2^{1/m} - 1)$$

- ❖ Per $m=2$, il carico della CPU non può superare l'83%
- ❖ Per $m \rightarrow \infty$, il termine di maggiorazione tende a $\ln(2)$, quindi l'utilizzo della CPU si assesta a ~69.3%
- ❖ Si noti che, essendo il test di schedutabilità solo sufficiente, l'insieme di task può essere schedulabile con RM anche se la condizione non è soddisfatta





Scheduling hard real-time – 5

❖ Scheduling EDF (Earliest-Deadline-First)

- Schedula i processi assegnando le priorità dinamicamente a seconda delle scadenze
- Quando un processo diventa eseguibile:
 - ▶ Deve annunciare la sua prossima scadenza allo scheduler
 - ▶ La priorità viene calcolata dinamicamente in base alla scadenza
 - ▶ Più vicina è la scadenza più alta è la priorità
 - ▶ La priorità di altri processi già nel sistema viene modificata per riflettere la scadenza del nuovo processo
- Si applica anche a processi non periodici e con tempo di elaborazione variabile
- Ottimale e (idealmente) porta ad un utilizzo della CPU pari al 100%

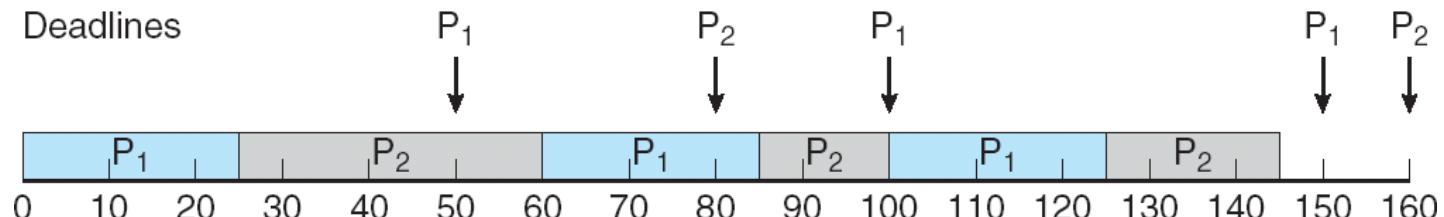




Scheduling hard real-time – 6

❖ Scheduling EDF

- **Esempio:** siano P_1 e P_2 due processi con periodi rispettivamente pari a 50 e 80; Supponiamo che i tempi di elaborazione siano invece $t_1=25$ e $t_2=35$





Scheduling hard real-time – 7

❖ Scheduling a quote proporzionali

- Opera distribuendo un certo numero di quote, T , fra tutte le applicazioni nel sistema
- Un'applicazione può ricevere N quote di tempo, assicurandosi così l'uso di una frazione N/T del tempo totale di CPU
- **Esempio:** $T=100$ quote a disposizione da ripartire fra i processi A , B e C ; se A dispone di 50 quote, B di 15 e C di 20, la CPU è occupata all'85%
- Lo scheduler deve lavorare in sinergia con un meccanismo di controllo dell'ammissione, per garantire che ogni applicazione possa effettivamente ricevere le quote di tempo che le sono state destinate (altrimenti non viene momentaneamente ammessa al sistema)





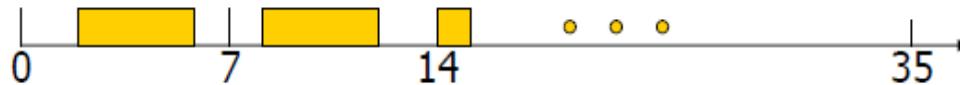
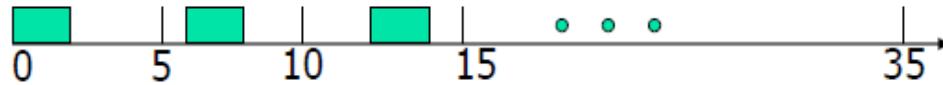
Esempio 4

❖ Esercizio

Si considerino i due task periodici real-time J_1 , con CPU-burst pari a 2msec e periodo di 5msec, e J_2 con tempo di esecuzione di 4msec e periodo di 7msec. Si valuti se i due task sono effettivamente schedulabili tramite EDF e, in caso affermativo, si mostri il relativo diagramma di Gantt.

❖ Soluzione

- Task set: $\{(2,5),(4,7)\}$
- $U = 2/5 + 4/7 = 34/35 \sim 0.97$ (schedulable!)





Esempio 5

❖ Esercizio

Si consideri il seguente insieme di processi con relativi tempi di arrivo, tempi di esecuzione e di scadenza:

Processo	Tempo di arrivo	Esecuzione	Scadenza
J_1	0	3	16
J_2	2	1	7
J_3	0	6	8
J_4	8	2	11
J_5	13	3	18

Si tracci il diagramma di Gantt in base alla politica di scheduling EDF e si dica se tutte le scadenze risultano rispettate (lo scheduling è feasible, cioè ammissibile).



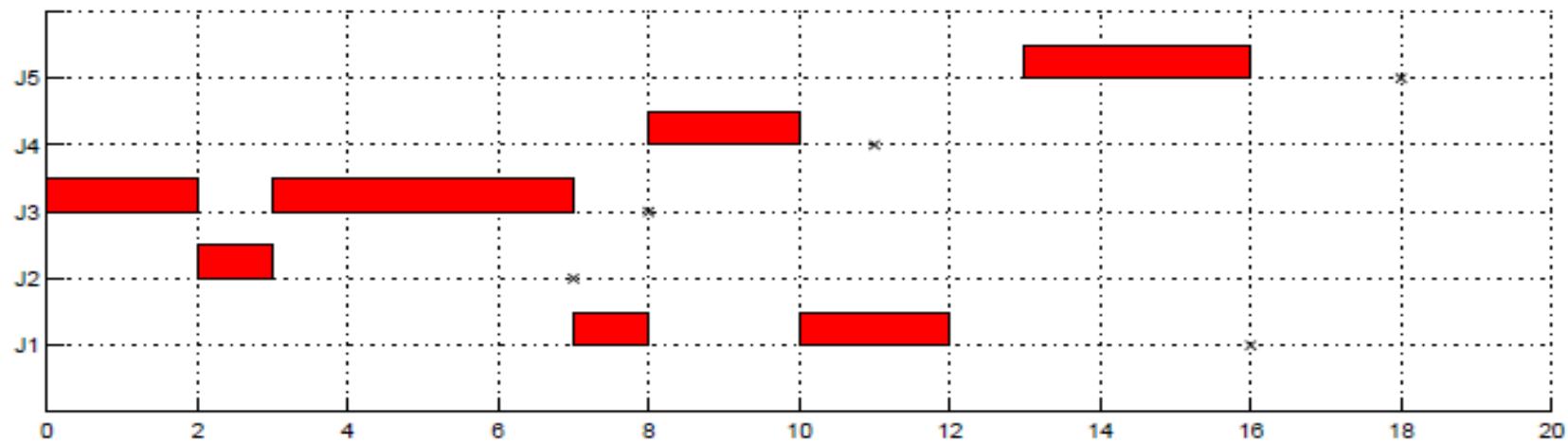


Esempio 5

❖ Soluzione

Processo	Tempo di arrivo	Esecuzione	Scadenza
J_1	0	3	16
J_2	2	1	7
J_3	0	6	8
J_4	8	2	11
J_5	13	3	18

Il diagramma di Gantt è:



Lo scheduling è dunque ammissibile.





Virtualizzazione e scheduling

- ❖ Il SO ospitante crea e gestisce le macchine virtuali, ciascuna dotata di un proprio SO ospite e proprie applicazioni; ogni SO ospite può essere messo a punto per usi specifici
 - Ogni algoritmo di scheduling che fa assunzioni sulla quantità di lavoro effettuabile in tempo prefissato verrà influenzato negativamente dalla virtualizzazione, perché i singoli SO virtualizzati sfruttano a loro insaputa solo una parte dei cicli di CPU disponibili
 - ▶ Tempi di risposta dilatati
 - ▶ L'orologio di sistema delle macchine virtuali può essere rallentato
 - La virtualizzazione può vanificare i benefici di un buon algoritmo di scheduling implementato dal SO ospite sulla macchina virtuale

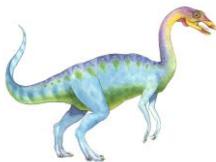




Scheduling in LINUX – 1

- ❖ Prima della versione 2.5 del kernel, era una variante dell'algoritmo tradizionale di UNIX (code multiple con feedback gestite perlopiù con RR)
 - Prelazione sulle code con priorità attribuite internamente, dipendenti dall'utilizzo della CPU da parte dei processi (più è grande, minore è la priorità)
 - Periodicamente il kernel calcola quanta CPU un processo ha consumato dall'ultimo controllo: questa quantità è "inversamente proporzionale" alla priorità del processo fino al prossimo controllo (l'aging si realizza "naturalmente")
 - Poco scalabile e non adeguato ai sistemi SMP





Scheduling in LINUX – 2

- ❖ Con la **versione 2.5** del kernel, lo scheduler è stato rivisitato per ottenere un algoritmo noto come $\mathcal{O}(1)$, indipendente dal numero di task presenti nel sistema
 - Algoritmo a priorità preemptive
 - Due intervalli di priorità, relativi a task time-sharing, detti *nice*, e (soft) real-time
 - Priorità dei task real-time con valori compresi fra 0 e 99, fra 100 e 139 per i task nice (valori numericamente inferiori indicano priorità maggiori)
 - Task a priorità maggiore ottengono time-slice più lunghi
 - I task vengono eseguiti (rimangono attivi) fino ad esaurimento del time-slice
 - Esaurito il quanto di tempo, il task viene considerato “scaduto” e non verrà più eseguito finché tutti gli altri task del sistema siano scaduti
- ❖ Adatto ai sistemi SMP (supporta bilanciamento del carico e predilezione)
- ❖ Tempi di risposta elevati per task time-sharing





Scheduling in LINUX – 3

❖ Dalla versione 2.6.23: *Completely Fair Scheduler* (CFS)

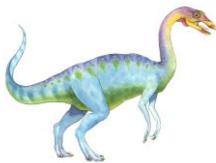
▪ Classi di scheduling

- Ciascuna dotata di una specifico range di priorità
- Lo scheduler seleziona il processo a più alta priorità, appartenente alla classe di scheduling a priorità più elevata
- Invece di assegnare un quanto di tempo, lo scheduler CFS assegna ad ogni task una percentuale del tempo di CPU
- Il kernel Linux standard implementa due classi di scheduling, dette **default** e **real-time** (possono però essere aggiunte altre classi)

▪ Per la classe default, la percentuale del tempo di CPU viene calcolata sulla base dei valori **nice** associati a ciascun task (da -20 a +19, valori minori per priorità maggiori)

- CFS non utilizza valori discreti per i quanti di tempo, ma definisce una **latenza obiettivo**, cioè un intervallo di tempo entro il quale ogni task eseguibile dovrebbe ottenere la CPU almeno una volta
- La latenza obiettivo cresce con il numero di task





Scheduling in LINUX – 4

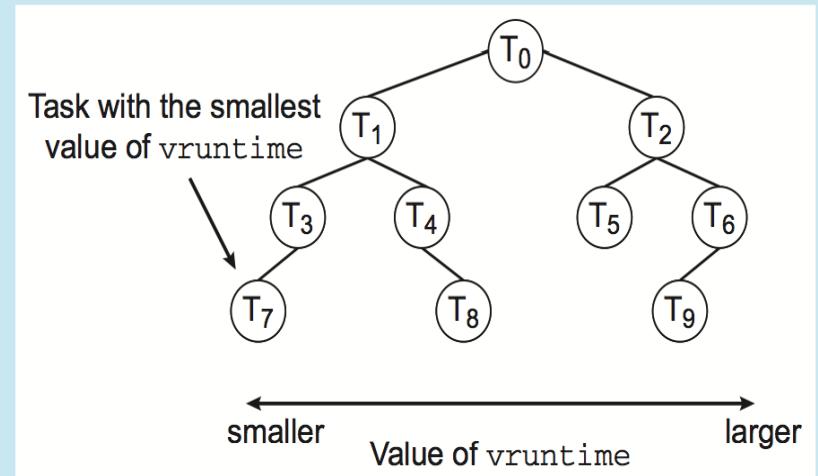
- ❖ Lo scheduler CFS non assegna direttamente le priorità, ma registra per quanto tempo è stato eseguito ogni task, mantenendo il **tempo di esecuzione virtuale** di ogni task nella variabile **vruntime**
 - Il tempo di esecuzione virtuale è associato ad un **fattore di decadimento** che dipende dalla priorità del task (task a bassa priorità hanno fattori di decadimento più alti e **vruntime** maggiore rispetto al tempo effettivo di esecuzione)
 - Per i task con priorità normale (nice=0), il tempo di esecuzione virtuale coincide con il tempo effettivo di esecuzione
- ❖ Per decidere il prossimo task da eseguire, lo scheduler seleziona il task con il valore **vruntime** più piccolo





Scheduling in LINUX – 5

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.



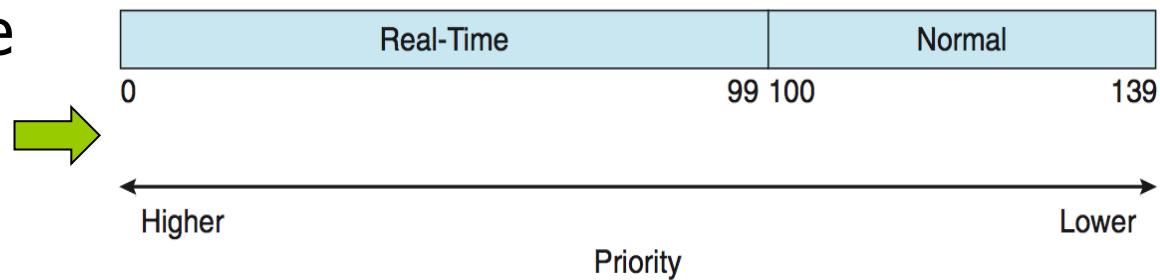
Prestazioni di CFS

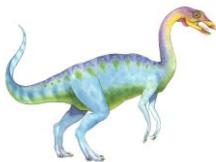


Scheduling in LINUX – 6

- ❖ **Real-time** (priorità 0–99)
 - Soft real-time: implementa lo standard POSIX.1b
 - Priorità (esterne) statiche
- ❖ **Task default**
 - Nice=−20 si mappa sul valore di priorità globale 100, mentre +19 viene mappato su 139
- ❖ Le priorità relative dei processi in tempo reale sono assicurate, ma il nucleo non fornisce garanzie sui tempi di attesa nella coda dei processi pronti
- ❖ Se un segnale d'interruzione dovesse rendere eseguibile un processo real-time mentre il nucleo è impegnato nell'esecuzione di una chiamata di sistema, il processo attende

“Real-time plus normal processes map into a global priority scheme”





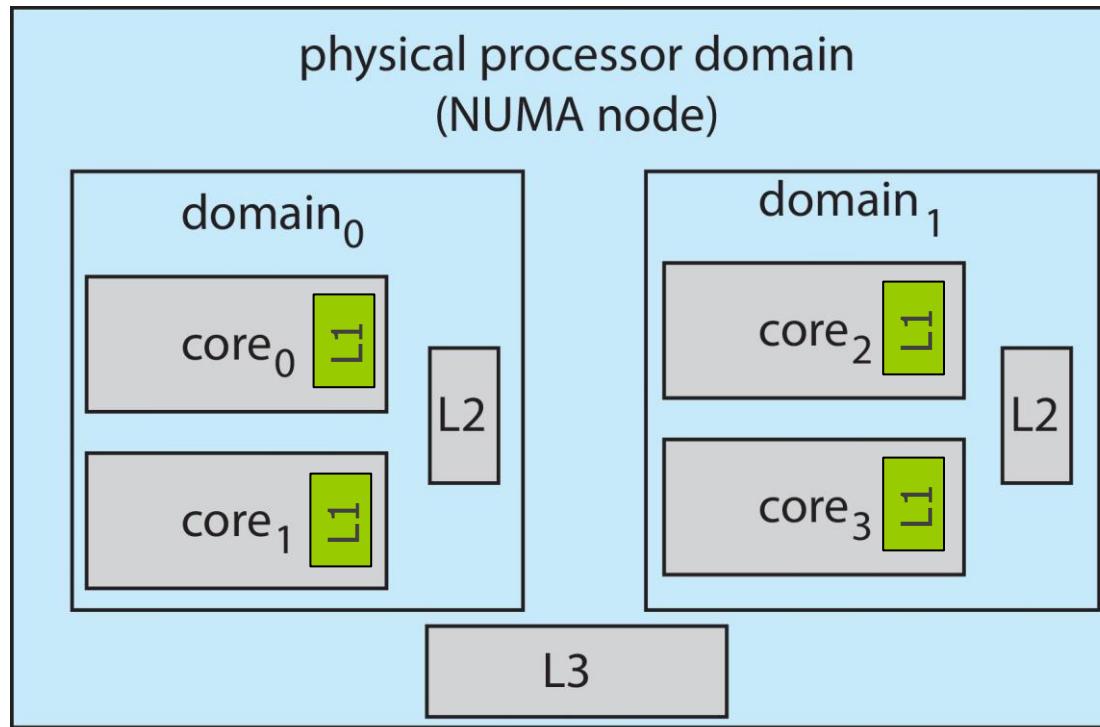
Scheduling in LINUX – 7

- ❖ Lo scheduler CFS supporta il bilanciamento del carico, ma è anche compatibile con NUMA (*NUMA-aware*) e riduce al minimo la migrazione dei thread
 - Il carico di ogni thread è dato dalla combinazione della sua priorità e del suo tasso medio di utilizzo della CPU
 - ▶ **Esempio:** Carico basso per task ad alta priorità, ma I/O-bound, o con bassa priorità e CPU-bound
 - ▶ Il carico delle code, che è la somma dei carichi di tutti i task che le compongono, deve essere approssimativamente uguale
 - Per ridurre la migrazione:
 - ▶ Linux identifica un sistema gerarchico di domini di scheduling, cioè un insieme di core che possono essere bilanciati con il minor impatto possibile sull'accesso alla memoria
 - ▶ I domini sono organizzati in base a ciò che condividono (ad esempio, memorie cache di diversi livelli)
 - ▶ Limitare la migrazione tra domini





Scheduling in LINUX – 8



Bilanciamento del carico NUMA-aware nello scheduler CFS di Linux





Valutazione degli algoritmi – 1

- ❖ Come selezionare un algoritmo di scheduling della CPU per un particolare sistema operativo?
 - ⇒ Definizione dei criteri da usare per la scelta dell'algoritmo
 - **Esempio 1:** massimizzare l'utilizzo della CPU, con tempo massimo di risposta inferiore a 300 millisecondi
 - **Esempio 2:** massimizzare la produttività, con tempo di completamento proporzionale (in media) al tempo di esecuzione effettivo





Valutazione degli algoritmi – 2

- ❖ **Valutazione analitica** – in base all'algoritmo ed al carico di lavoro del sistema, fornisce una formula per valutarne le prestazioni
- ❖ **Modelli deterministic**i – valutazione analitica che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro
 - **Esempio:** si considerino cinque processi che arrivano tutti al tempo 0

<u>Process</u>	<u>Burst Time</u>
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

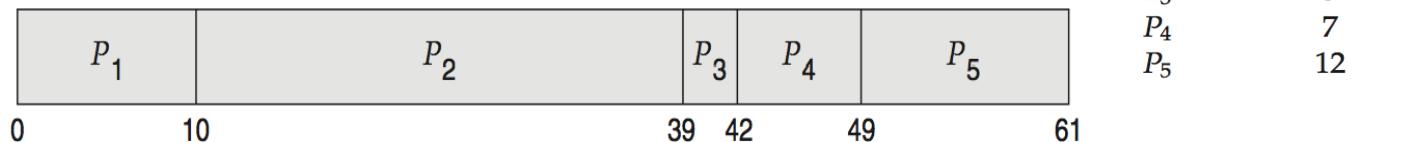




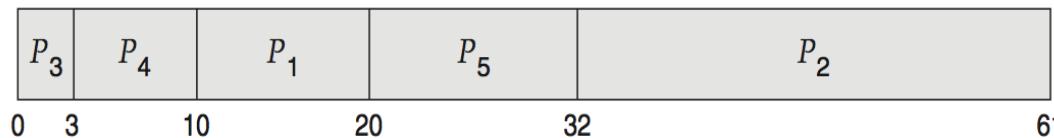
Valutazione degli algoritmi – 3

- ❖ Per ciascun algoritmo si calcola il tempo medio di attesa
- ❖ Metodo semplice e veloce che richiede però una conoscenza delle dinamiche del sistema normalmente non disponibile

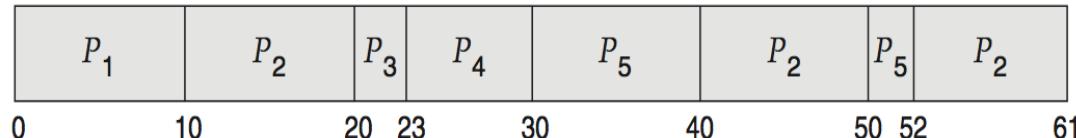
- **FCFS** — $T_a = (0+10+39+42+49)/5 = 28\text{ msec}$



- **SJF (non preemptive)** — $T_a = (10+32+0+3+20)/5 = 13\text{ msec}$



- **RR** con time-slice=10 — $T_a = (0+32+20+23+40)/5 = 23\text{ msec}$





Valutazione degli algoritmi – 4

- ❖ **Reti di code** – il sistema di calcolo viene descritto come una rete di server, ciascuno con una coda d'attesa
 - La CPU è un server con la propria coda dei processi pronti, ed il sistema di I/O ha le sue code dei dispositivi
 - Se sono noti l'andamento degli arrivi e dei servizi (sotto forma di distribuzioni di probabilità), si può calcolare utilizzo di CPU e dispositivi, lunghezza media delle code, tempo medio d'attesa, throughput medio, etc.
 - ▶ Normalmente, si rilevano distribuzioni esponenziali, descritte attraverso valori medi
 - Problema: difficoltà nell'utilizzo di distribuzioni complicate/metodi matematici appositi \Rightarrow modelli non realistici





Valutazione degli algoritmi – 5

- ❖ Sia n la lunghezza media di una coda e siano W il tempo medio di attesa nella coda e λ l'andamento medio di arrivo dei nuovi processi
- ❖ Se il sistema è stabile, il numero di processi che lasciano la coda deve essere uguale al numero di processi che vi arrivano

$$n = \lambda \times W$$

è la **formula di Little**, valida per qualsiasi algoritmo di scheduling e distribuzione degli arrivi

- ❖ È utilizzabile per il calcolo di una delle tre variabili, quando sono note le altre due: per esempio, se ogni secondo arrivano 7 processi e la coda ne contiene mediamente 14, il tempo medio d'attesa per ogni processo è pari a 2 secondi





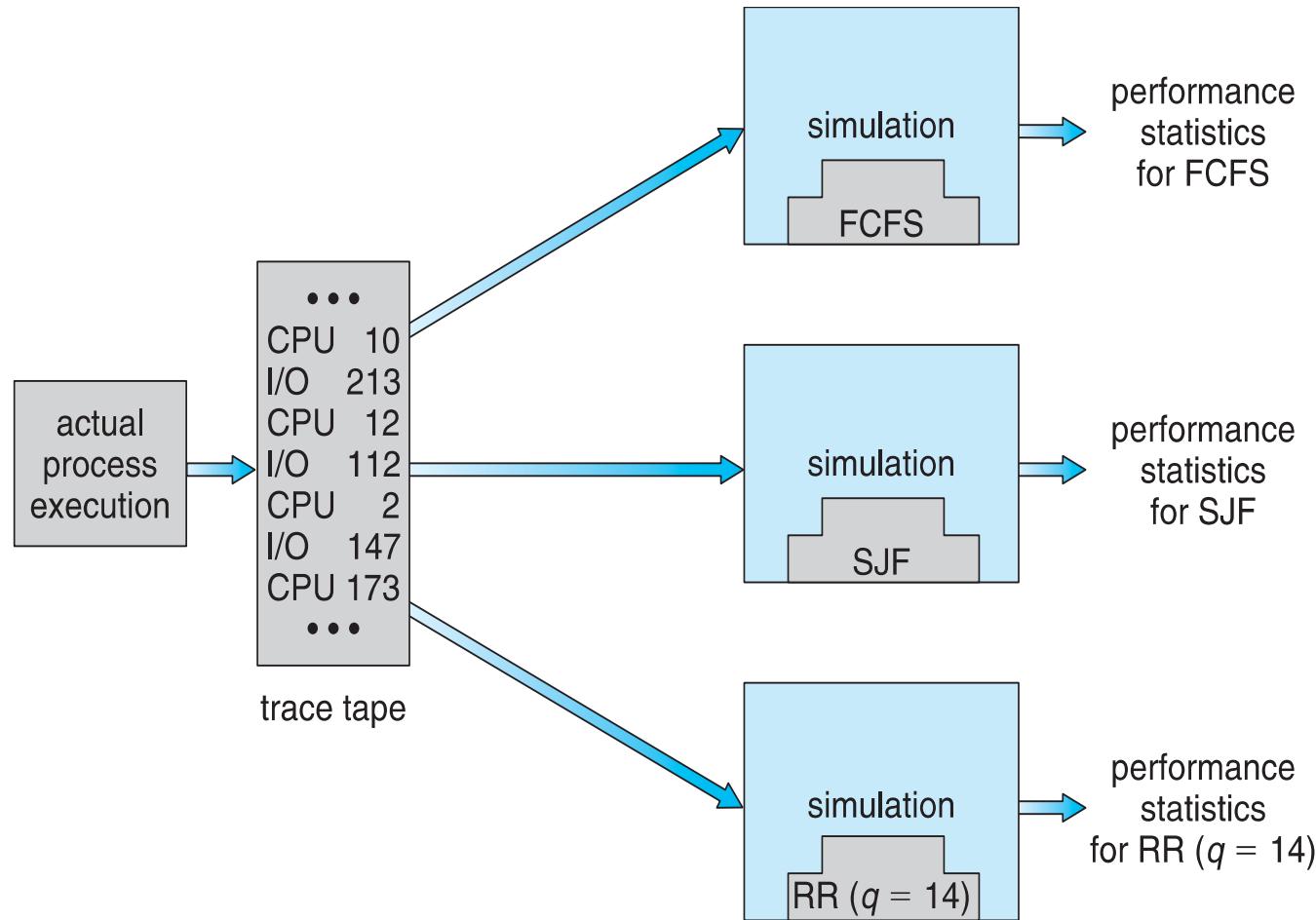
Valutazione degli algoritmi – 6

- ❖ **Simulazione** – implica la programmazione di un modello del sistema di calcolo; le strutture dati rappresentano gli elementi principali del sistema
 - Il simulatore dispone di una variabile che rappresenta il clock e modifica lo stato del sistema in modo da descrivere le attività dei dispositivi, dei processi e dello scheduler
 - Durante l'esecuzione della simulazione, si raccolgono e si stampano statistiche che descrivono le prestazioni degli algoritmi
 - I dati per la simulazione possono essere generati artificialmente, modellati da distribuzioni matematiche o empiriche o raccolti da un sistema reale mediante un *trace tape*





Valutazione degli algoritmi – 7



Valutazione di algoritmi di CPU scheduling tramite simulazione





Valutazione degli algoritmi – 8

- ❖ **Implementazione** – implica la codifica effettiva degli algoritmi di scheduling da valutare, ed il loro inserimento nel sistema operativo, per osservarne il comportamento nelle condizioni reali di funzionamento del sistema
 - Difficoltà nel fare accettare agli utenti un sistema in continua evoluzione
 - Alti costi, alti rischi
 - ▶ Adeguamento del comportamento degli utenti alle caratteristiche dello scheduler
 - Gli scheduler più flessibili possono essere modificati *per-site* o *per-system*

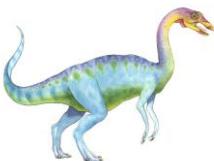




Considerazioni finali

- ❖ Gli algoritmi di scheduling più adattabili sono quelli che possono essere tarati dagli amministratori di sistema, che li adattano al particolare ambiente di calcolo, con la propria specifica gamma di applicazioni
- ❖ Molti degli attuali sistemi operativi UNIX-like forniscono all'amministratore la possibilità di calibrare i parametri di scheduling in vista di particolari configurazioni del sistema (es. Solaris)





Esercizi

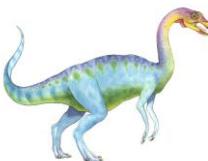
❖ Esercizio 1

Cinque lavori, indicati con le lettere da A a E, arrivano al calcolatore approssimativamente allo stesso istante. I processi hanno un tempo di esecuzione stimato di 8, 10, 2, 4 e 8 secondi, rispettivamente, mentre le loro priorità (determinate esternamente) sono 2, 4, 5, 1, 3 (con 5 priorità max). Per ognuno dei seguenti algoritmi di scheduling

- Round robin (2 sec)
- Scheduling a priorità (non preemptive)
- FCFS
- SJF (non preemptive)

si calcoli il tempo medio di turnaround. Si ignori l'overhead dovuto al cambio di contesto.





Esercizi (cont.)

❖ Esercizio 2

Si consideri il seguente insieme di processi:

Processo	Tempo di arrivo	Tempo di burst (msec)
P_0	0	7
P_1	2	4
P_2	3	4
P_3	5	2
P_4	7	3
P_5	10	2

I processi arrivano nell'ordine indicato. Si calcoli il tempo medio di turnaround con scheduling SJF preemptive.





Esercizi (cont.)

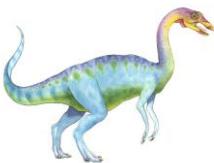
❖ Esercizio 3

Si consideri il seguente insieme di processi:

Processo	Tempo di arrivo	Tempo di burst (msec)
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Si calcoli il tempo medio di attesa ed il tempo medio di turnaround, nel caso di scheduling FCFS, RR con quanto di tempo 1 e 4 e SJF non preemptive.





Esercizi (cont.)

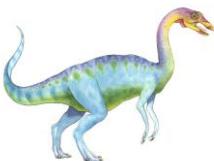
❖ Esercizio 4

Si consideri un sistema con scheduling a priorità con tre code, A, B, C, di priorità decrescente e con prelazione tra code. Le code A e B sono servite con modalità RR con quanto di 15 e 20 ms, rispettivamente; la coda C è FCFS. Se un processo nella coda A o B consuma il suo quanto di tempo, viene spostato in fondo alla coda B o C, rispettivamente. Si supponga che i processi P_1, P_2, P_3, P_4 arrivino rispettivamente nelle code A, C, B, A, con CPU burst e tempi indicati nella tabella seguente:

Processo	Tempo di arrivo	Tempo di burst (msec)
P_1	0	20
P_2	10	25
P_3	16	20
P_4	25	20

Si tracci il diagramma di Gantt relativo all'esecuzione dei quattro processi e si calcoli il tempo medio di turnaround.





Esercizi (cont.)

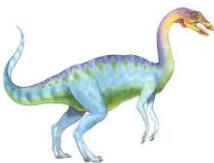
❖ Esercizio 5

Si considerino i seguenti processi, attivi in un sistema multiprogrammato:

Processo	Tempo di arrivo	Tempo di burst (msec)	Priorità
P_1	0	8	2
P_2	1	6	3
P_3	2	3	2
P_4	3	4	2
P_5	4	3	2

Descrivere la sequenza di esecuzione dei job, tramite diagramma di Gantt e calcolare l'istante in cui ogni job completa il suo task, ed il relativo tempo di turnaround, ottenuto mediante scheduling a code multiple a priorità e feedback, dove ogni coda è gestita con la strategia FCFS. Il feedback è definito come segue: la priorità diminuisce di 1 (fino al livello base 1) se si passano più di 6 unità di tempo (consecutive) in esecuzione ed aumenta di 1 per ogni 6 unità di tempo passate in attesa in una qualsiasi coda.





Esercizi (cont.)

❖ Esercizio 6

Si considerino N ($N \geq 2$) processi in contesa su una singola CPU gestita con scheduling Round Robin. Supponiamo che ciascun context switch abbia una durata di S msec e che il quanto di tempo sia pari a Q . Per semplicità, si assuma inoltre che i processi non siano mai bloccati a causa di un qualche evento e semplicemente passino dall'essere in esecuzione all'essere in attesa nella ready queue.

Si calcoli il massimo quanto di tempo Q tale che, per ciascun processo, non trascorrano più di T msec tra l'inizio di due burst successivi.

Si calcoli il massimo quanto di tempo Q tale che ciascun processo non debba attendere più di T msec nella ready queue.

I valori di Q dovranno essere calcolati in funzione di N , S e T .



Fine del Capitolo 5

