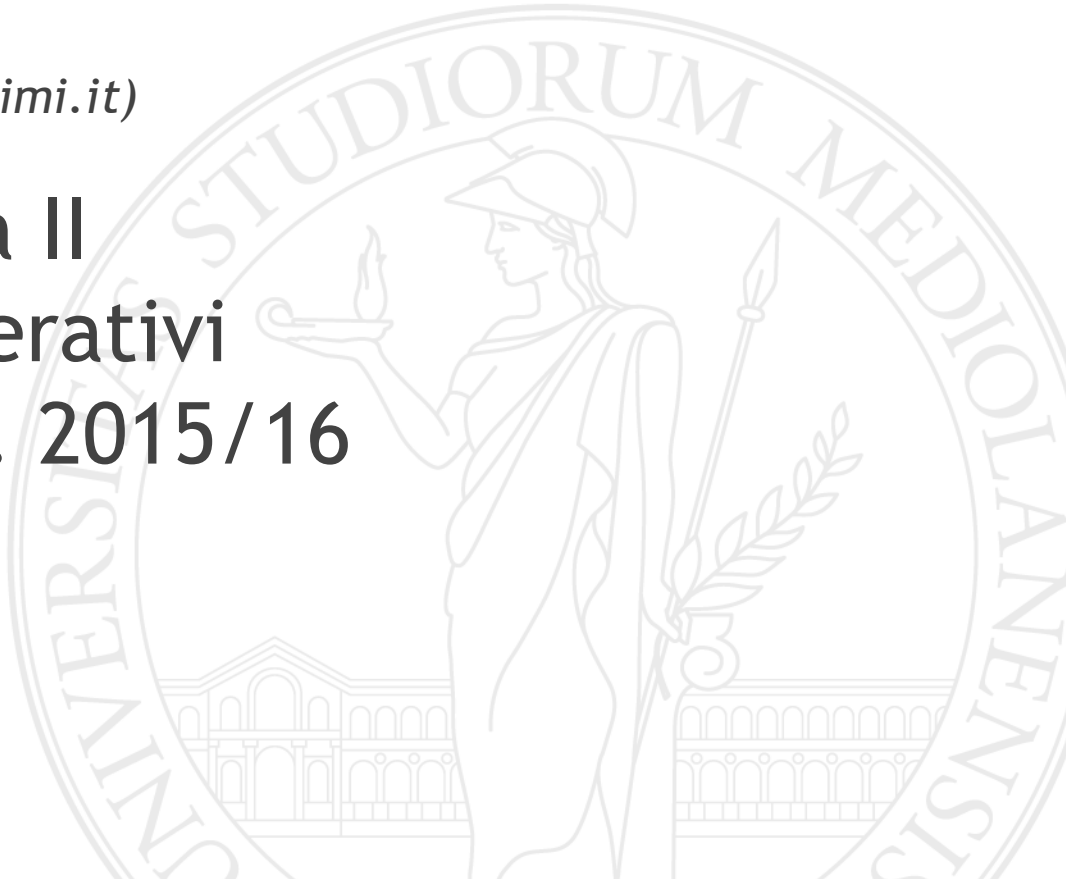
UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI INFORMATICA

Alberto Ceselli
(alberto.ceselli@unimi.it)

Informatica II
Sistemi Operativi
DIGIP - a.a. 2015/16



Sistemi Operativi

(modulo di Informatica II)

Schedulazione della CPU

Patrizia Scandurra

Università degli Studi di Bergamo

a.a. 2014-15

Sommario

- Concetti base
  - Come si realizza il multi-tasking
  - Come si realizza il time-sharing
  - Attivazione non pre-emptive/pre-emptive
  - Criteri di schedulazione
- Algoritmi di schedulazione
- Schedulazione per sistemi multiprocessore
- Schedulazione per sistemi in tempo reale
- Schedulazione in Windows e Linux

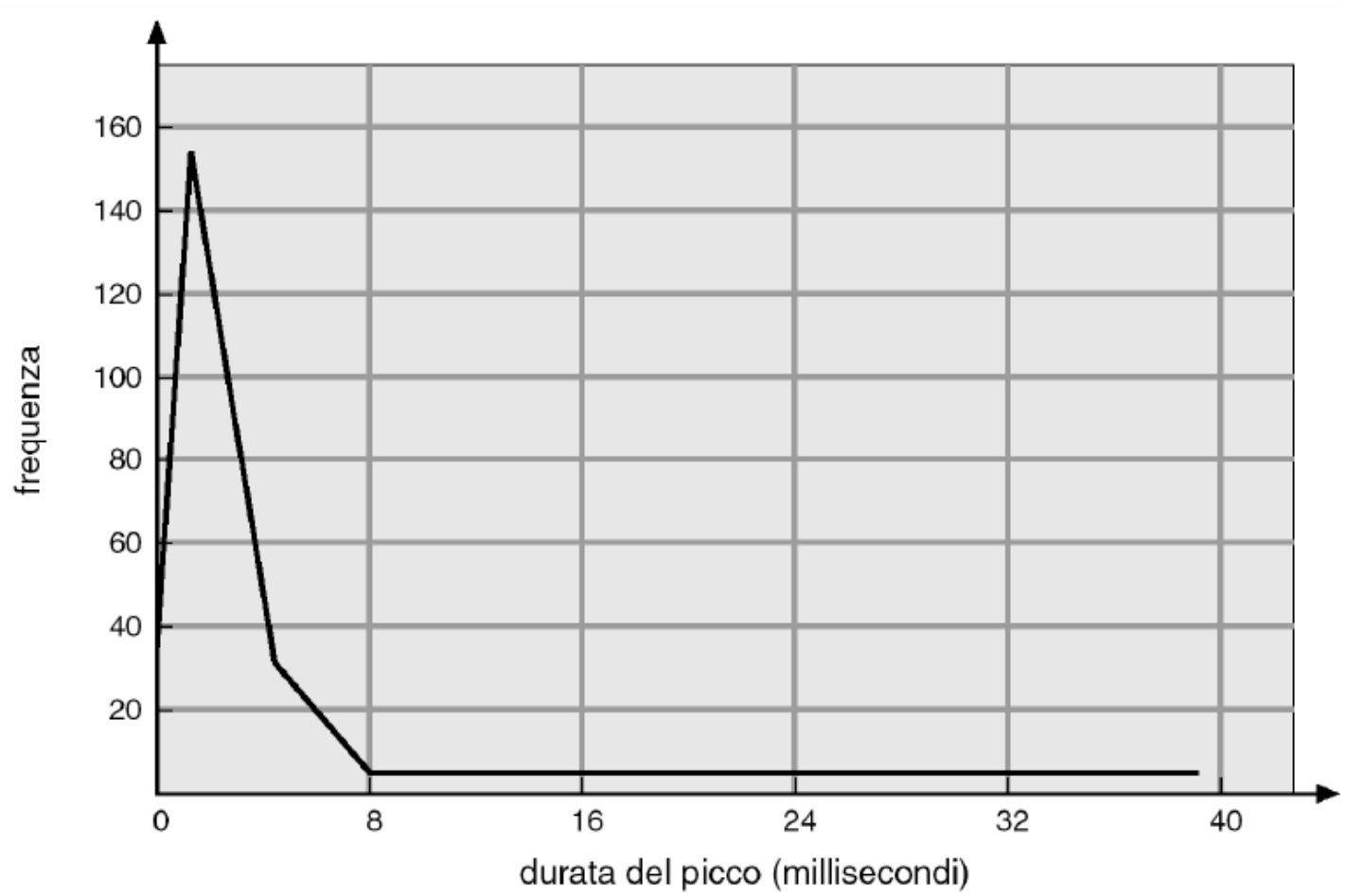
Schedulazione della CPU (1)

- Obiettivo: realizzare la turnazione dei processi sul processore in modo da
  - → massimizzarne lo sfruttamento della CPU
  - → creare l'illusione di evoluzione contemporanea dei processi in sistemi time-sharing

Ciclo di picco di CPU e di I/O

- L'esecuzione di un processo è una sequenza alternata di:
  - picchi (cicli) d'esecuzione di CPU e
  - di attesa di I/O

Grafico dei tempi di picco



Un processo I/O-bound ha molti picchi brevi

Un processo CPU-bound ha pochi picchi lunghi

Schedulazione della CPU (1)

- Lo **schedulatore a breve termine** *sceglie* fra i processi *in stato ready* ed assegna la CPU ad uno di essi
- La decisione può avvenire nei seguenti casi:
  1. Quando un processo passa **dallo stato running allo stato waiting** (per effetto di una richiesta di I/O o di join sulla terminazione di un sotto-processo)
  2. Quando un processo passa **dallo stato running allo stato ready** (tipicamente quando si verifica un interrupt)
  3. Quando un processo passa **dallo stato waiting allo stato di ready** (ad es. per il completamento di I/O)
  4. Quando un **processo termina**
- La schedulazione dei punti 1 e 4 è detta non-preemptive (senza sospensione dell'esecuzione)
- Tutte le altre schedulazioni sono dette preemptive (con sospensione dell'esecuzione)

Schedulazione della CPU (2)

- La schedulazione non-preempitive è tipica della realizzazione del **multi-tasking**
- La schedulazione preemptive è alla base del **time sharing**
  - Concetto di time-sharing
    - Quanto di tempo (time slice)

      intervallo di tempo massimo
      di uso consecutivo del processore
      consentito a ciascun processo
    - Pre-rilascio (pre-emption)
- La schedulazione preemptive richiede l'uso di meccanismi di sincronizzazione per l'accesso ai dati condivisi!

Caricamento del processo sulla CPU

- Il dispatcher è il modulo del SO che dà il controllo della CPU ad un processo selezionato dallo schedulatore a breve termine
- Questa funzione comprende:
  1. Passaggio in modalità kernel
  2. cambio di contesto (context switch)
     - Consiste nel salvataggio del contesto di esecuzione del processo corrente
       - Salvare tutti i registri del processore sullo stack
       - Salvare lo stack pointer nel Process Control Block (PCB), ecc..
  3. passaggio alla modalità utente
  4. salto alla corretta locazione nel programma utente per ricominciarne l'esecuzione -- riattivazione del contesto di esecuzione del nuovo processo
- ***Latenza del dispatcher*** – *tempo necessario al dispatcher* per fermare un processo e cominciarne un altro

Cambiamento del processo in esecuzione

Cambio di contesto

Context Switch

- **Sospensione** del processo in esecuzione

+

- **Attivazione** del processo da mettere in esecuzione

Criteri di schedulazione

1. Utilizzo della CPU % – mantenere la CPU il più impegnata possibile
2. Frequenza di completamento (*throughput)* – numero di processi completati per unità di tempo
   - 1 processo all'ora per processi di lunga durata, 10 processi al secondo per brevi transazioni
3. Tempo di completamento (*turnaround time)* – intervallo che va dal momento dell'immissione del processo nel sistema al momento del completamento
   - Tempo complessivo, comprende tempi di esecuzione e tempi spesi nelle code di attesa
4. Tempo di attesa – somma dei tempi spesi in attesa in coda ready
5. Tempo di risposta – tempo che intercorre dalla formulazione della richiesta fino alla produzione della prima risposta, escludendo il tempo di output (per gli ambienti di timesharing)

Criteri di ottimizzazione

- Massimizzare l'utilizzo della CPU
- Massimizzare il throughput
- Minimizzare il turnaround time
- Minimizzare il tempo di attesa
- Minimizzare il tempo di risposta

In genere si ottimizza:

- il **valore medio** e/o
- la **varianza**
  - per sistemi time-sharing, si preferisce minimizzare la varianza nel tempo di risposta (meglio un sistema *predicibile* che uno più veloce ma maggiormente variabile)

Algoritmi (politiche) di schedulazione

- Schedulazione First-Come, First-Served (FCFS)
- Schedulazione Shortest-Job-First (SJR)
- Schedulazione a priorità
- Schedulazione Round Robin (RR)
- Schedulazione con coda a più livelli

- NB: problema intrinsecamente **online** → non ho a disposizione informazione completa al momento della decisione

  - Il nostro termine di paragone sarà il **tempo di attesa medio**
  - Useremo il **diagramma di Gantt** per rappresentare l'utilizzo nel tempo della CPU da parte dei processi

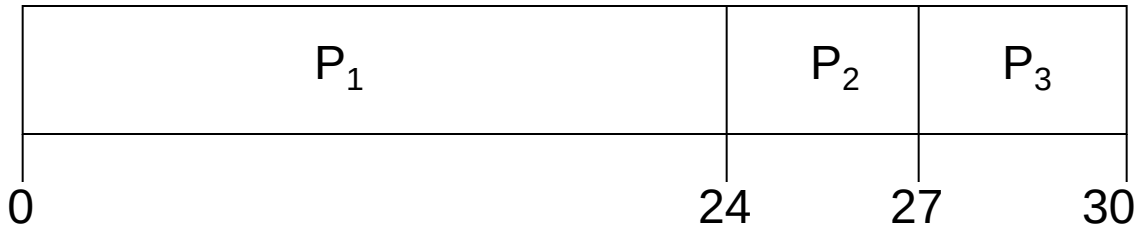
Schedulazione First-Come, First-Served (FCFS) (1)

- I processi vengono schedulati in ordine di arrivo
- Il primo ad entrare in coda è il primo ad essere servito
- L'algoritmo è di tipo *non-preemptive*
  - I processi lasciano la CPU solo di spontanea volontà (vanno in attesa o terminano)

Schedulazione First-Come, First-Served (FCFS) (2)

Process	Durata del picco
P_1	24
P_2	3
P_3	3

- Supponiamo che i processsi arrivano nell'ordine: P_1, P_2, P_3
- Si ha il seguente **diagramma di Gantt** per la schedulazione:



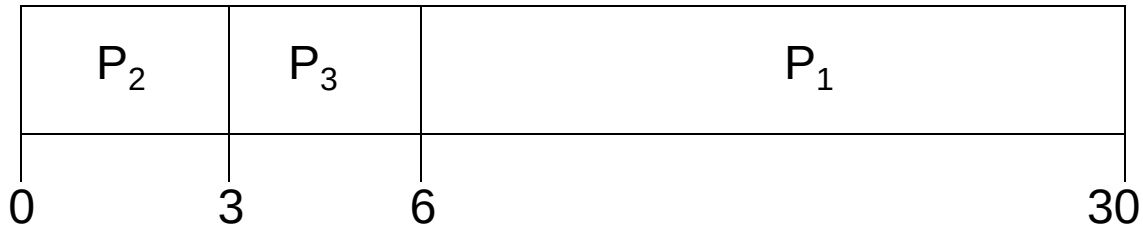
- Tempo di attesa per $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Tempo di attesa medio: $(0 + 24 + 27)/3 = 17$

Schedulazione FCFS (3)

- Supponiamo ora che i processi arrivano nell'ordine

$$P_2 , P_3 , P_1$$

- Il diagramma di Gantt per la schedulazione è:



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

- E' non-premptive
- C'è un effetto di ritardo a catena (*convoy effect*) mentre processi brevi (I/O-bound) attendono che quello grosso (CPU-bound) rilasci la CPU

Schedulazione FCFS (4)

- Essendo **non-preemptive** è problematico per sistemi time-sharing
  - Infatti i processi non possono usufruire della CPU a intervalli regolari
- C'è un effetto di **ritardo a catena (*convoy effect*)** mentre processi brevi (I/O-bound) attendono che quello grosso (CPU-bound) rilasci la CPU
  - Supponiamo di avere un processo CPU-bounded e molti processi I/O-bounded (come nel primo esempio)
  - I processi I/O-bounded che hanno un tempo di esecuzione breve aspettano molto a cause del lungo tempo di esecuzione del processo CPU-bounded

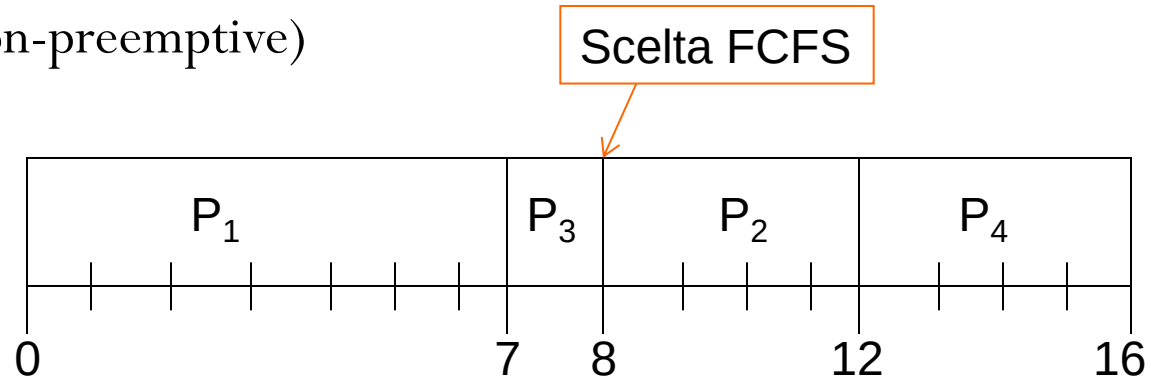
Shortest-Job-First (SJF) Scheduling

- Associa a ciascun processo "la lunghezza del successivo picco di CPU del processo medesimo", per **schedulare il processo con il minor tempo**
  - Nota bene: ci si basa sulla lunghezza del prossimo picco, non su quella totale
  - A parità di lunghezza del picco successivo si applica FCFS
- **La scelta di SJF è ottimale rispetto al criterio di minimizzazione dell'attesa media**
- Due schemi:
  - Non-preemptive – quando un processo arriva nella coda dei processi pronti mentre il processo precedente è ancora in esecuzione, l'algoritmo **permette al processo corrente di finire il suo uso della CPU**
  - Preemptive – **quando arriva un processo con un tempo minore del tempo che rimane al processo corrente**mente in esecuzione, **l'algoritmo ferma il processo corrente**
    - Questa schedulazione è anche detta *shortest-remaining-time-first*

Esempio di Non-Preemptive SJF

Processo	Tempo di arrivo	Durata del picco
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (non-preemptive)

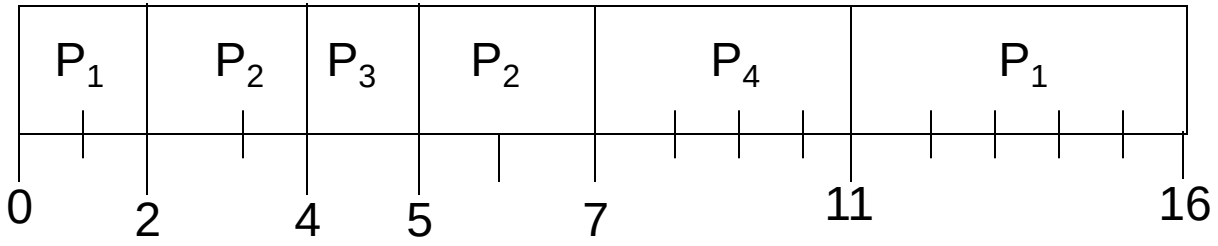


- Tempo di attesa medio = [0 + (8-2) + (7-4) + (12-5)]/4 = 4

Esempio di Preemptive SJF

Processo	Tempo di arrivo	Durata del picco
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (preemptive)



- Tempo di attesa medio = [(11-2) + 1 + 0 +(7-5)]/4 = [9 + 1 + 0 +2]/4 = 3

Schedulatore a breve e a lungo termine

- SJF è tipicamente usato per schedulatori a **lungo termine**
  - In un sistema a lotti si usa come tempo di elaborazione il tempo limite di elaborazione che gli utenti assegnano al processo
    - Più è breve, più il tempo medio d'attesa è basso
    - Troppo breve può però causare un errore di superamento del tempo limite e richiedere una nuova esecuzione
- Negli schedulatori a **breve termine** può essere usato cercando di predire il tempo di esecuzione.

Determinare la lunghezza del successivo picco di CPU

- **SJF ottimale, ma ideale:** presuppone di conoscere la lunghezza dei picchi di CPU (che tipicamente non è data)
- Possiamo però **stimarla** (ovvero farne una previsione) **usando ad esempio la media esponenziale** delle lunghezze dei picchi di CPU precedenti

1. t_n = lunghezza attuale dell' $n-esimo$ picco di CPU
2. τ_{n+1} = valore previsto per il prossimo picco di CPU, τ_0 cos tan *te*
3. α , $0 \leq \alpha \leq 1$ è il peso della nostra predizione , solitament e $\alpha = 1/2$
4. Definiamo :

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

Storia recente

Storia passata

Esempi di media esponenziale

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - La storia recente non ha nessun effetto

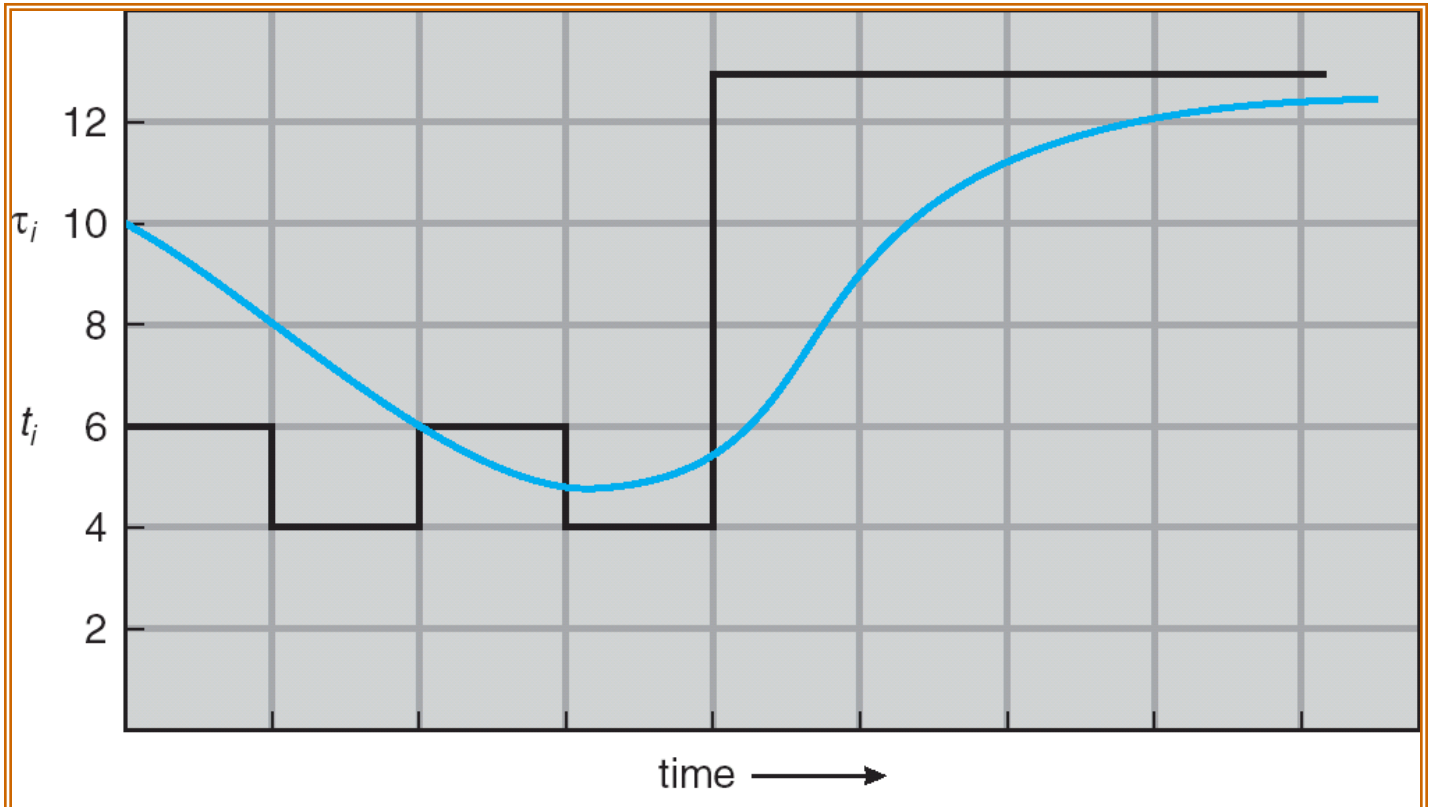
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Conta solo il picco più recente di CPU

- Se sviluppiamo la formula, otteniamo:

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

- Poichè sia α che $(1 - \alpha)$ sono $<= 1$, ciascun termine successivo ha un peso inferiore rispetto a quello precedente

Previsione della durata del prossimo picco di CPU



Picco di CPU	(t_i)		6	4	6	4	13	13	13	...
previsione	(τ_i)	10	8	6	6	5	9	11	12	...

Schedulazione a priorità (1)

- Si associa **una priorità numerica (un intero) a ciascun processo**
- La CPU è allocata al processo con priorità più alta
  (per convenzione: più piccolo è il numero ≡ più alta è la priorità)
  - **Preemptive**
  - **Non-preemptive**
  - I processi con priorità uguale vengono schedulati in ordine FCFS
- SJF è un algoritmo con priorità dove la priorità è l'inverso del prossimo picco (previsto) di CPU

Schedulazione a priorità (2)

- Le priorità possono essere definite in base a fattori
  - Interni: usano quantità misurabili per calcolare la priorità
    - limiti di tempo, richiesta di memoria, num. file aperti, ecc.
  - Esterni: usano criteri esterni al sistema
    - l'importanza del processo, il tipo e la quantità di fondi pagati per l'utilizzo del calcolatore, ecc..
- **Problema**: blocco indefinito (*starvation*) – processi a bassa priorità non vengono mai eseguiti
- **Soluzione**: Invecchiamento (*aging*) – *accresce gradualmente la priorità di un processo* che attende nel sistema per un lungo periodo

Schedulazione Round Robin (RR) (1)

- **FCFS + preemption** per alternare i processi (coda FIFO circolare)
- Ogni processo possiede un ***quanto di tempo* (*time slice*) q** di utilizzo della CPU
  - generalmente q=10-100 ms

  Se entro questo arco di tempo il processo non lascia la CPU, viene interrotto e rimesso nella coda dei processi pronti

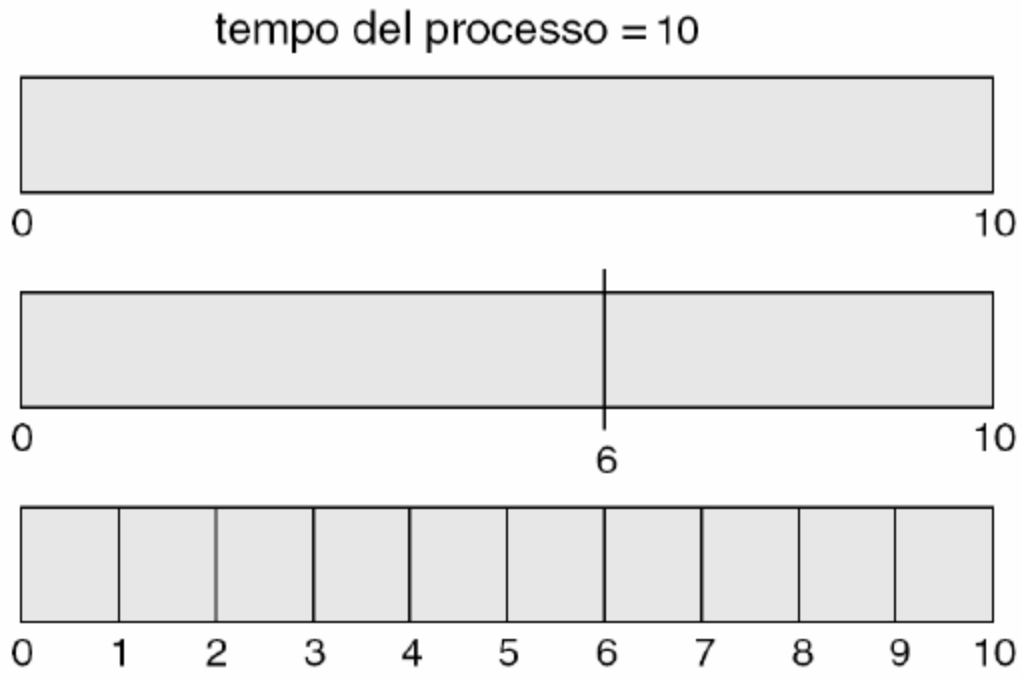
- Se ci sono n processi nella coda dei processi pronti e il quanto di tempo è q, allora
  - Ciascun processo ottiene 1/n del tempo di CPU in parti lunghe al più q unità di tempo
  - Ciascun processo non deve attendere più di (n – 1) x q unità di tempo
    - Ad es. con 5 processi e q=20 ms, ciascun processo avrà al max 20 ms ogni 100

Round Robin (RR) – scelta del quanto (2)

- **Prestazioni**:
  - q grande ⇒ FIFO (cioè al FCFS)
  - q piccolo ⇒ q produce un maggior effetto di "parallelismo virtuale" tra i processi,
    - È come se ogni processo abbia a disposizione una CPU n volte più lenta di quella fisica
    - però aumenta il numero di context-switch, e quindi l'overhead (sovraccarico) per gestirli

Quanto di tempo e cambio del contesto

Il modo in cui un quanto di tempo più piccolo aumenta i cambi di contesto



quanto di tempo	cambi di contesto
12	0
6	1
1	9

Round Robin (RR) – scelta del quanto (3)

Considerando l'effetto del context-switch…

- Si vuole che **q** sia **lungo rispetto al tempo per il cambio di contesto**, altrimenti l'overhead sarebbe troppo elevato

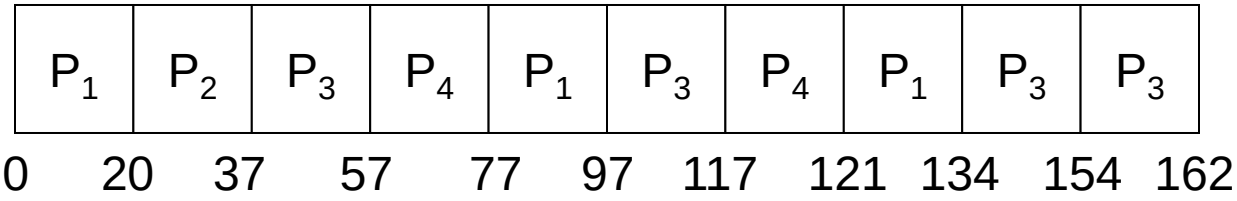
- Regola empirica: 80% dei picchi di CPU deve essere più breve del quanto di tempo

- In pratica, molti SO moderni hanno:
  - una durata del quanto q tra 10 e 100 ms (10^{-3} secondi)
  - tempo di context-switch di 10 μs (10^{-6} secondi)
    - 10 μs = 0,01 ms

Esempio di RR con quanto di tempo = 20

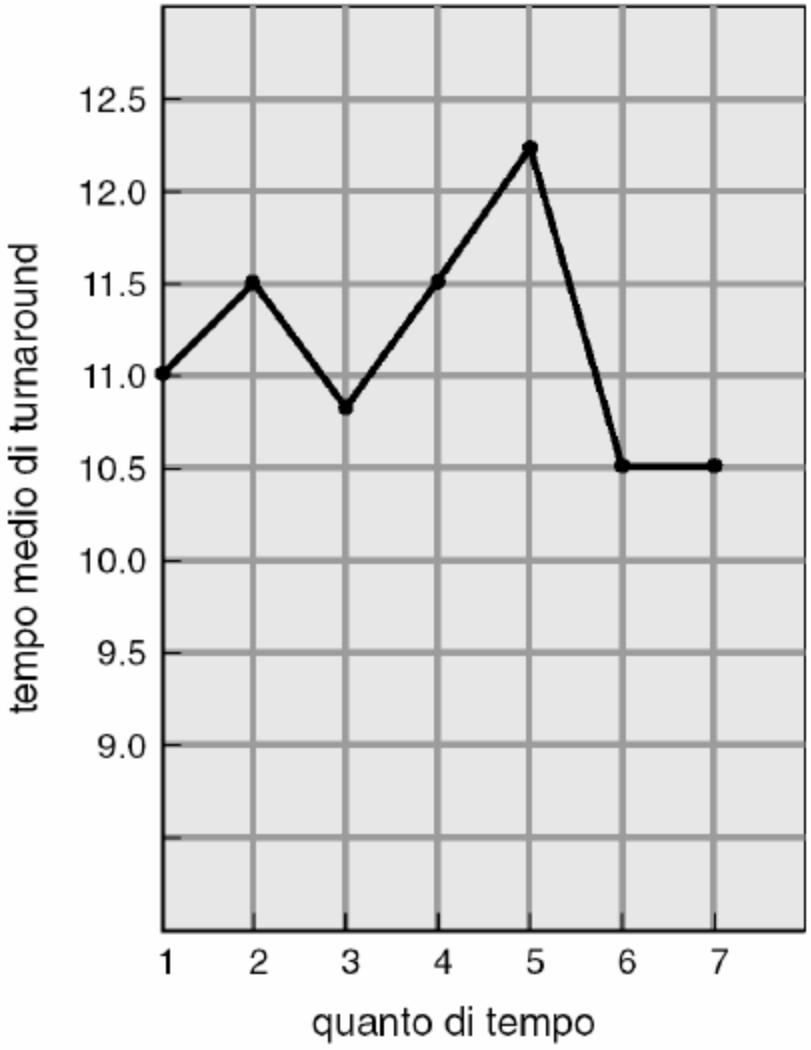
Process	Durata del picco
P_1	53
P_2	17
P_3	68
P_4	24

- Diagramma di Gantt:



- Di solito una **media di *turnaround*** (tempo di completamento di un processo) **più alta di SJF, ma migliore tempo di risposta**

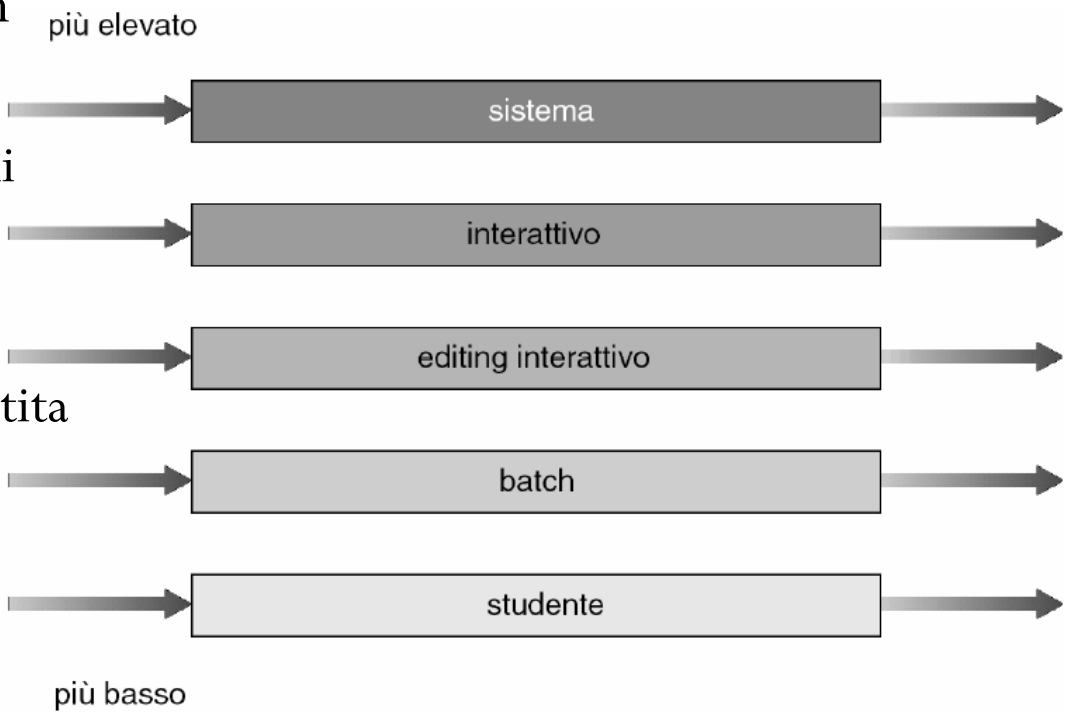
Variazione del Turnaround Time con il quanto di tempo



processo	tempo
P_1	6
P_2	3
P_3	1
P_4	7

Coda a più livelli (1)

- È adatto in situazioni in cui i processi possono essere divisi in gruppi
  - Per esempio in base hai tempi di risposta, alle necessità di scheduling e all'importanza
- La coda dei processi pronti è ripartita in **code separate**
  Tipicamente:
  - *foreground (interattivi)*
  - *background (batch – sullo sfondo)*
- Ciascuna coda ha il **suo algoritmo di schedulazione**:
  - foreground – RR
  - background – FCFS



Coda a più livelli (2)

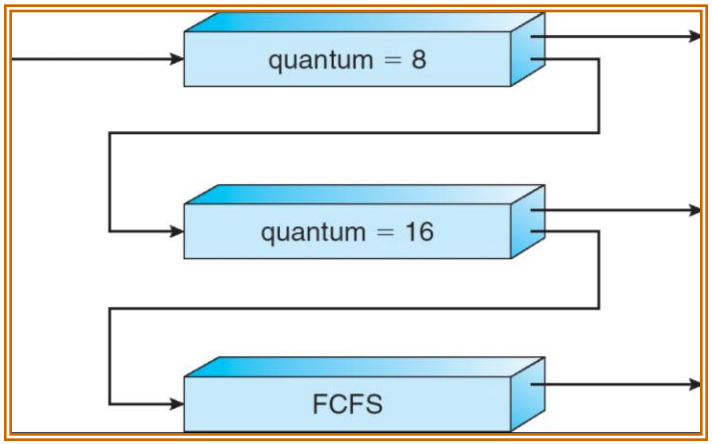
- Ci deve essere una schedulazione **anche tra le code!**
- Due possibili alternative:
  - **Schedulazione preemptive a priorità fissa**
    - La coda dei processi in foreground può avere priorità assoluta su quella dei processi in background
    - Possibilità di starvation
  - **Partizione del tempo tra le code** (time slice) – ciascuna coda ha una certa quantità di tempo di CPU, che può schedulare fra i processi in essa contenuti

    Ad esempio:
    - il **foreground ha l'80%** del tempo di CPU per la schedulazione RR
    - il **background riceve il 20%** della CPU da dare ai suoi processi secondo l'algoritmo FCFS

Coda a più livelli con retroazione (*feedback*)

- **Un processo può muoversi tra le varie code**; questa forma di invecchiamento previene la starvation
- Uno schedulatore con coda a più livelli con feedback è definito dai seguenti **parametri**:
  - numero di code
  - algoritmo di schedulazione per ciascuna coda
  - metodo utilizzato per determinare *quando promuovere* un processo facendolo risalire verso una coda a priorità più alta (ad es. se ha atteso troppo)
  - metodo utilizzato per determinare *quando degradare* un processo in una coda a più bassa priorità (ad es. se è troppo CPU-bound)
  - metodo utilizzato per determinare in quale coda entrerà un processo quando avrà bisogno di un servizio

Esempio di una coda a più livelli con retroazione



- Tre code:
  - Q_0 – RR con quanto di tempo 8 ms
  - Q_1 – RR con quanto di tempo 16 ms
  - Q_2 – FCFS

- Schedulazione:
  - Un nuovo processo "pronto" entra inizialmente nella coda Q_0
  - I processi lunghi affondano automaticamente nella coda Q_2 e sono serviti in ordine FCFS utilizzando i cicli di CPU lasciati dalle code Q_0 e Q_1 (quando sono vuote)
    - In Q_0, quando ottiene la CPU, il processo riceve 8 ms. Se non termina in 8 ms, il processo viene spostato nella coda Q_1
    - In Q_1 il processo riceve 16 ms aggiuntivi. Se ancora non ha completato, viene spostato nella coda Q_2

Schedulazione dei Thread

- Nei SO che li supportano, **sono i thread a livello kernel** – non i processi – **ad essere schedulati**!
- Il kernel non è a conoscenza dei thread a livello utente
  - Schedulazione locale – Come la libreria dei thread decide quali thread inserire in un thread a livello kernel
    - *Process Contention Scope (PCS):* Competizione della CPU tra i thread di uno stesso processo
  - Schedulazione globale – Come il kernel decide quale sarà il prossimo thread da eseguire
    - *System Contention Scope (SCS):* Competizione della CPU tra tutti i thread del sistema
    - Windows XP, Solaris, Linux, etc. che implementano la mappatura uno-a-uno, utilizzano solamente la SCS

Schedulazione *multiprocessing*

- Con più CPU, la schedulazione diviene più complessa
- Nell'ipotesi di avere **processori omogenei**:
  - Suddivisione del carico (*load sharing*)
    - Una coda di processi per ciascun processore o un'unica coda condivisa ma che richiede sincronizzazione/mutua esclusione
  - Due approcci di schedulazione
    - **Multiprocessamento asimmetrico** – solo un processore (il *master*)
      - prende le decisioni relative allo scheduling, gli altri processori fanno solo elaborazione
      - accede alle strutture dati del sistema, limitando così la condivisone dei dati
    - **Multiprocessamento simmetrico** (**SMP**) – ciascun processore schedula se stesso selezionando un processo dalla coda comune dei processi pronti
      - Vanno progettati con cura per garantire sincronizzazione per l'accesso alla coda
      - Alcuni sistemi SMP applicano il principio di *"predilezione del processore"*

Schedulazione Real-Time

- Sistemi **hard real-time (in tempo reale stretto)**
  - **devono** completare un'operazione critica entro una quantità di tempo garantita

- Computazione **soft real-time (in tempo reale lasco)**
  - **richiede** che i processi critici ricevano priorità su quelli meno importanti

Schedulazione Hard Real-Time (1)

- Sistemi **hard real-time (in tempo reale stretto) –** devono completare un'operazione critica entro una quantità di tempo garantita
  - **Lo schedulatore o rifiuta o accetta** (garantendo che il processo sia completato in tempo sotto *resource reservation*) – *tecnica di controllo dell'ammissione*
  - SW specifico e HW dedicato (no storage secondario, perché causano variazioni nel tempo)
  - Tipicamente, i processi sono *periodici*:

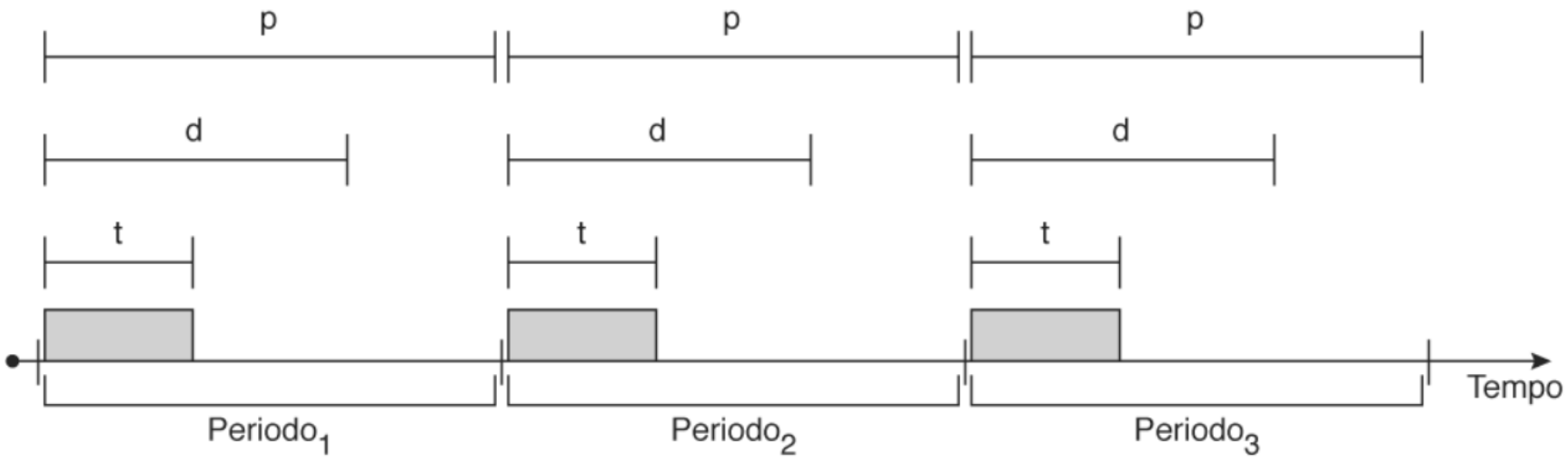
$$0 <= t <= d <= p$$

Tempo di elaborazione fisso

scadenza

periodo

Schedulazione Hard Real-Time (2)



Schedulazione Hard Real-Time (3)

Due algoritmi:

- *Algoritmo di schedulazione a frequenza monotona* (priorità statica)
- *Algoritmo di schedulazione a scadenza più urgente* (priorità dinamica)

Algoritmo di schedulazione a frequenza monotona

- **Algoritmo di schedulazione a frequenza monotona** schedula processi periodici usando priorità statiche con preemption
  - Quando un processo entra nel sistema, gli viene assegnata una priorità inversamente proporzionale al suo periodo
    - ☺ si assegna una priorità maggiore ai processi che richiedono la CPU più frequentemente!
  - Quando un processo diventa pronto, se il processo in esecuzione è a priorità più bassa viene sospeso (preemption)
  - Controllo dell'ammissione: si considera

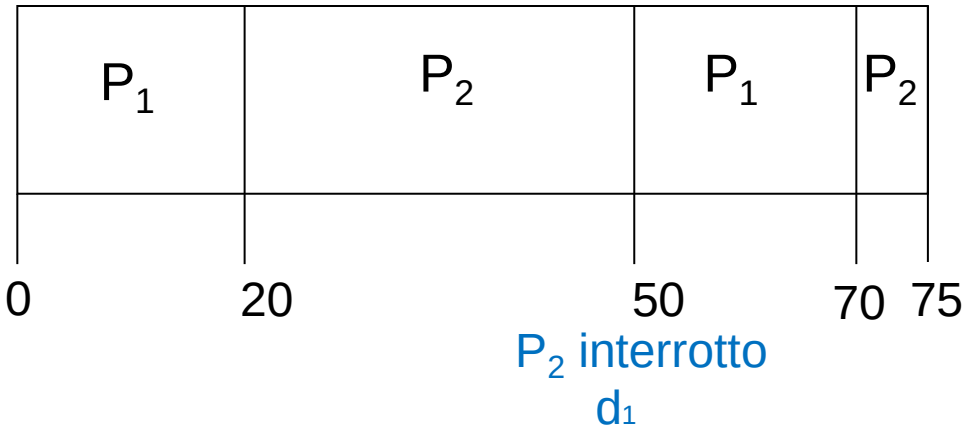
    % uso della CPU per un processo = t/p
  - Algoritmo **esatto** per priorità statica
    - se questo algoritmo non è in grado di trovare una sequenza di scheduling, allora nessun algoritmo a priorità statica può riuscirci
    - l'utilizzo della CPU potrebbe non essere massimo

Esempio di schedulazione Hard Real-Time a frequenza monotona

Processo	t	p=d	% uso CPU
P_1	20	50	0.40
P_2	35	100	0.35

Totale: 75% < 100% schedulazione ammessa!

P_1 ha priorità maggiore di P_2 perché ha periodo più breve

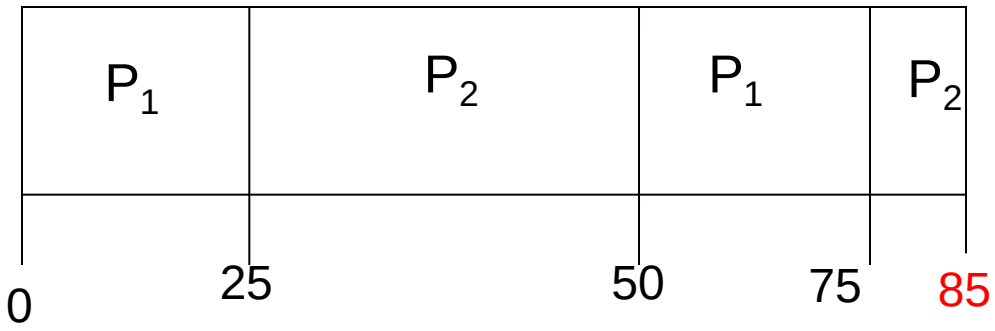


Hard Real-Time a frequenza monotona- un **esempio non schedulabile**

Processo	t	p=d	% uso CPU
P_1	25	50	0.50
P_2	35	80	0.44

Totale: 94% < 100% schedulazione ammessa!

P_1 ha priorità maggiore di P_2 perché ha periodo più breve



P_2 interrotto
(anche se mancano 10 ms)

Non va bene:
P_2 si completa a 85 > 80

Algoritmo di schedulazione a scadenza più urgente

- Schedula i processi assegnando le priorità dinamicamente a seconda delle scadenze
  - Quando un processo diventa eseguibile, annuncia la sua scadenza
    - prima è la sua scadenza più alta è la sua priorità
    - la priorità di altri processi già nel sistema viene modificata per riflettere la scadenza del nuovo processo
- Valido anche per processi non periodici e con tempo di elaborazione variabile
- Schedulazione ottima rispetto al criterio di utilizzo della CPU (idealmente 100%)

Esempio di schedulazione Hard Real-Time a scadenza più urgente

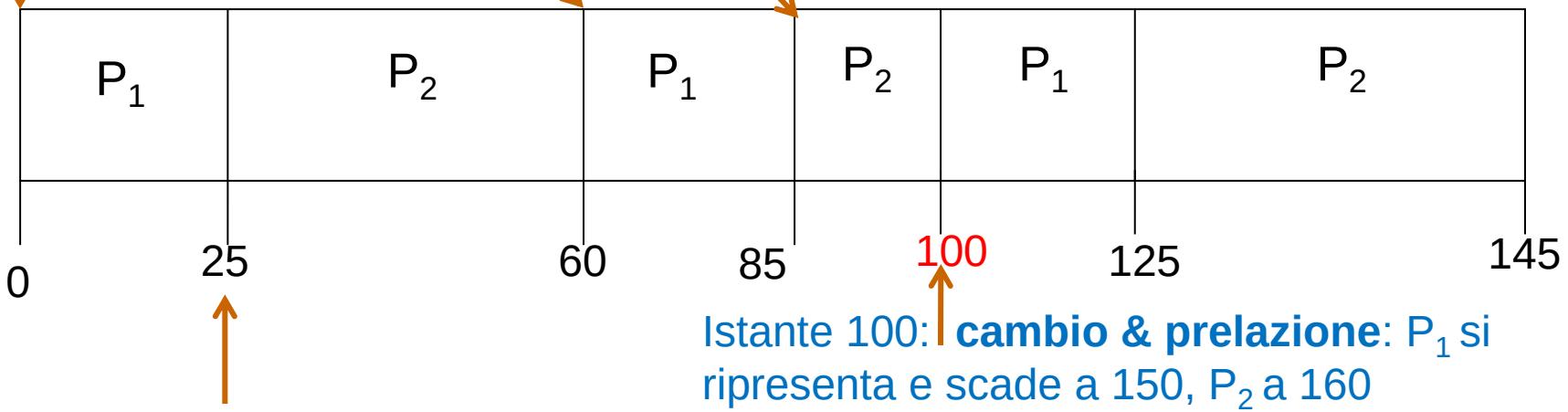
Processo	t	p=d
P_1	25	50
P_2	35	80

Stesso esempio non schedulabile con l'algoritmo a frequenza monotona!

Istante 0: P_1 ha scadenza più urgente, e quindi inizialmente ha priorità più alta di P_2

Istante 60: nuovo cambio, P_2 termina, P_1 scade a 100

Istante 85: nuovo cambio, P_1 termina, P_2 scade a 160



Istante 100: **cambio & prelazione**: P_1 si ripresenta e scade a 150, P_2 a 160

Istante 25: Primo cambio, P_1 termina e si ripresenterà a 50, P_2 scade a 80

Schedulazione Soft Real-Time

- Computazione **soft real-time (in tempo reale lasco) – richiede** che i processi critici ricevano priorità su quelli meno importanti
  - Meno restrittiva
  - per supportare multimedia, grafica interattiva ad alta velocità, ecc.
- **Proprietà**:
  - La priorità dei processi real-time non deve diminuire con il tempo (mentre quelli non real-time si)
  - Mantenere bassa la *latenza di dispatch*
    - Più piccola è la latenza, più in fretta i processi real-time possono iniziare l'esecuzione

Schedulazione Soft Real-Time

- Come si fa a mantenere bassa la latenza di dispatch?
- E' sufficiente permettere che il kernel sia interrotto
- Due possibili modi:
  - **Preemption point** (punti di sospensione): inseriti in punti *sicuri* delle chiamate di sistema lunghe
    - Il punto di prelazione verifica se un processo a più alta priorità deve essere eseguito, ed in tal caso avviene un cambio di contesto;
    - al termine di questo processo, riprende l'esecuzione della chiamata di sistema
  - **Kernel interrompibile** per intero (usato in *Solaris*)
    - Prevede meccanismi di sincronizzazione per proteggere le strutture dati del kernel in fase di aggiornamento da parte di processi ad alta priorità

Schedulazione in Windows (1)

- **Schedulazione a code multiple con feedback**
- Scheduling basato su **priorità dinamica** e **preemption**
- Un processo viene eseguito fino a che:
  - Non è sottoposto a prelazione da parte di un processo con priorità più alta
  - Non finisce il quanto di tempo assegnato
  - Esegue una chiamata di sistema bloccante
  - Termina
- I processi sono divisi in classi, ognuna con intervalli di priorità
  - Classe variabili (da 1 a 15)
  - Classe real-time (da 16 a 31)
- Lo schedulatore ha **una coda per ogni classe di priorità** e percorre le code dalla più alta alla più bassa fino a che trova un processo pronto

Schedulazione in Windows (2)

- Priorità di classe (32 livelli) in Windows XP: a classe variabile e soft-real time
- e priorità relativa

Priorità a classe variabile (da 1 a 15)

Priorità di classe

Priorità relativa	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Classe real-time (da 16 a 31)

Schedulazione in Windows (3)

- **Algoritmi di scheduling**: ogni coda è gestita attraverso l'algoritmo di Round Robin, tutte tranne l'ultima che è servita tramite FCFS.
- **Feedback dello schedulatore:** un processo a priorità variabile può cambiare coda quando si verificano due eventi:
  - Termina il suo quanto di tempo
    - La priorità del processo viene ridotta (limita l'uso della CPU ai thread CPU-bound)
  - Viene sospesa a causa di un'operazione di attesa
    - La priorità del processo viene alzata
    - La quantità dell'incremento dipende dall'evento che si attende
    - Questo permette di favorire ad esempio le interfacce grafiche
- Lo scheduler inoltre distingue fra **processi in foreground e background**
  - I processi in foreground hanno un quanto maggiore (tipicamente di un fattore 3)

Schedulazione in Linux (1)

Prima della versione 2.5 di Linux:

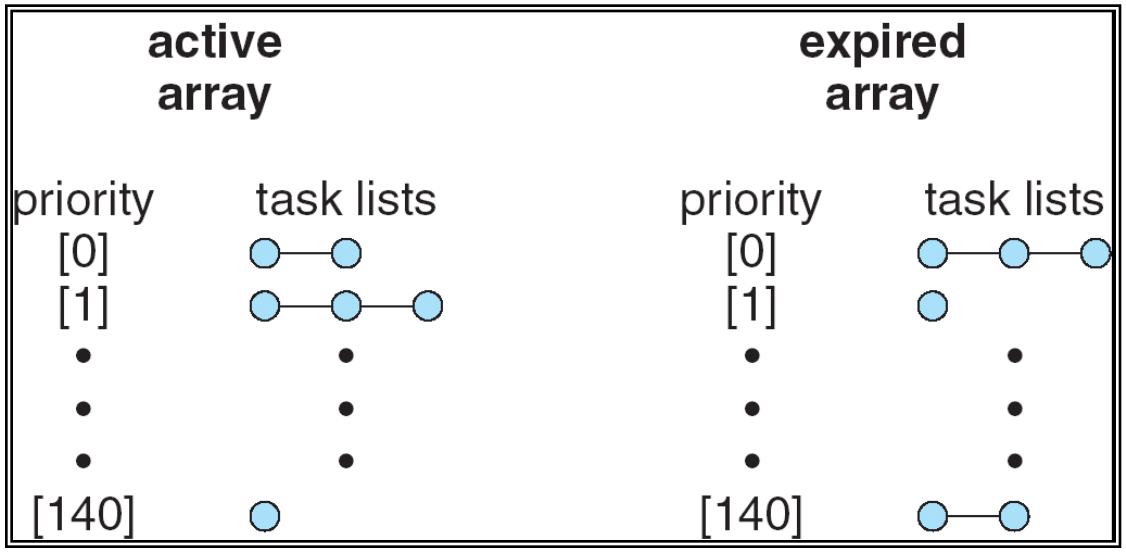
- **Time-sharing (a quanto variabile, maggiore per priorità alte)**:
  - Priorità basata su **tick** (**crediti**) – il processo con più crediti sarà il prossimo ad essere schedulato
  - Ogni volta che avviene un interrupt del timer il processo in esecuzione in quel momento perde un credito
  - Quando il credito è uguale a 0 viene scelto un altro processo
  - Quando tutti i processi hanno credito 0, c'è una **redistribuzione dei crediti**
    - basata su fattori che includono la storia e la priorità del processo
    - Per maggiori dettagli vedi pagina manuale Linux relativa alla chiamata di sistema *sched_setscheduler:*
      - *sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)*

      "sets both the scheduling policy and the associated parameters for the process identified by pid"

Schedulazione in Linux (2)

Da 2.5 in poi:

- **Supporto SMP**: ogni processore ha una coda dei processi, predilezione del processore, schedulazione basata su scala di priorità -- esecuzione veloce O(1) -- e non sui crediti
- Ogni processore ha una **coda di due livelli: dei task attivi e una dei task scaduti**
  - I primi non hanno ancora completato il loro quanto di tempo
  - Al completamento vengono spostati nella seconda coda
  - In entrambe le code i task sono ordinati per priorità
- Quando la coda dei task attivi è vuota le due code vengono invertite
  - La coda dei task scaduti diventa quella dei task attivi e vice-versa



Schedulazione in Linux (3)

- Due classi di priorità
  - Real-time: da 0 a 99
  - Nice: da 100 a 140
- I processi con priorità più alta ricevono un quanto di tempo più lungo