

SISTEMI OPERATIVI

(MODULO DI INFORMATICA II)

Schedulazione della CPU

Prof. Luca Gherardi

Prof.ssa Patrizia Scandurra (anni precedenti)

Università degli Studi di Bergamo

a.a. 2012-13

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 WindowsXP e Linux

Schedulazione della CPU (1)

- Multiprogrammazione
 - Un processo è in esecuzione fino a quando non deve attendere un evento
 - Durante l'attesa la CPU è inattiva
 - La multiprogrammazione impiega questo tempo in modo produttivo eseguendo un altro processo
- Obiettivo dello scheduling: 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

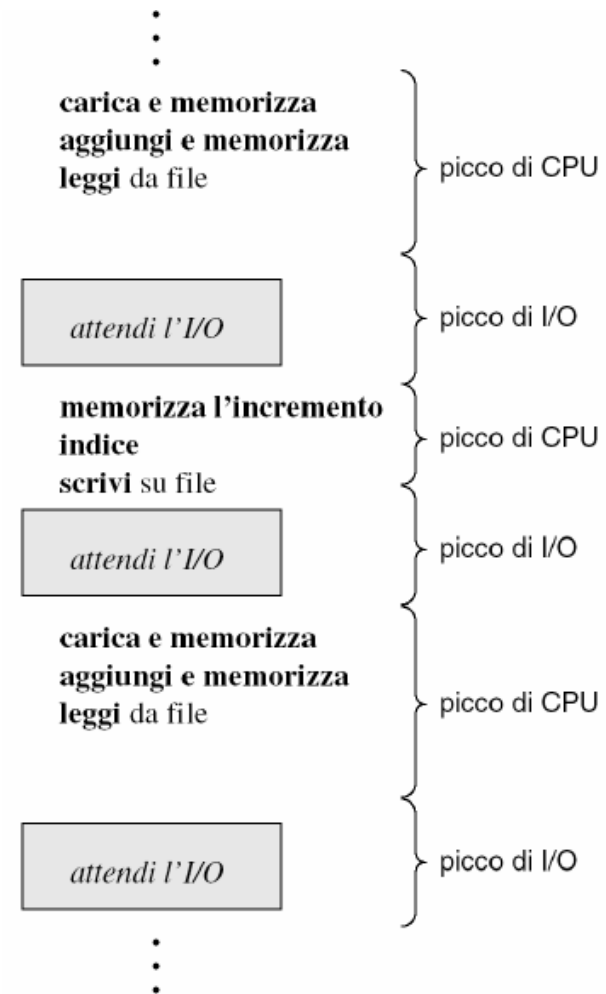
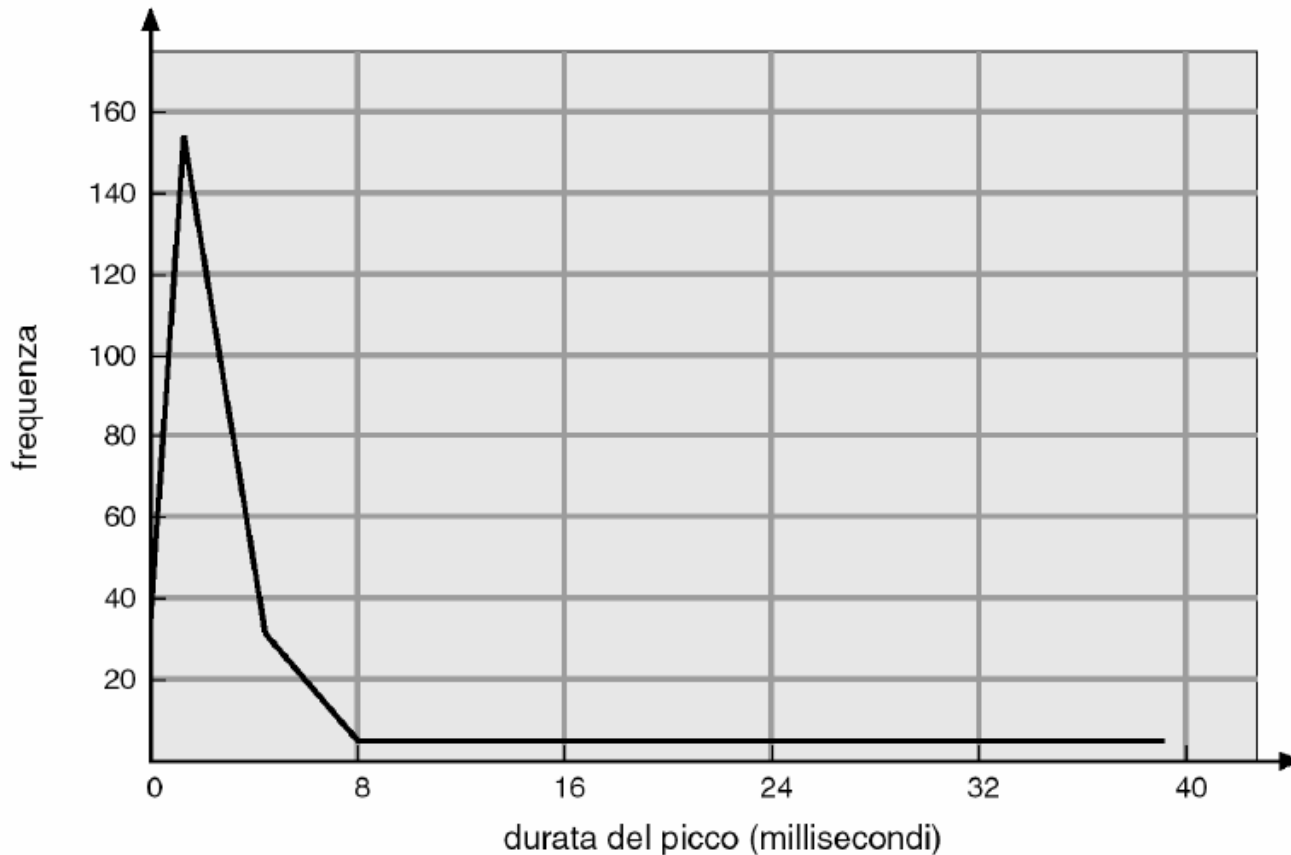


Grafico dei tempi di picco



- 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 pronti in memoria per l'esecuzione ed assegna la CPU ad uno di essi
- La decisione può avvenire nei seguenti casi:
 1. Quando un processo passa dallo stato di **esecuzione** allo stato di **attesa** (per effetto di una richiesta di I/O o di join sulla terminazione di un sotto-processo)
 2. Quando un processo passa dallo stato di **esecuzione** allo stato di **pronto** (tipicamente quando si verifica un interrupt)
 3. Quando un processo passa dallo stato di **attesa** allo stato di **pronto** (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-preemptive** 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): un processo può essere sospeso prima che termini il quanto di tempo
- La schedulazione **preemptive** richiede l'uso di meccanismi di **sincronizzazione** per l'accesso ai dati condivisi!

Caricamento del processo sulla CPU

- Il **dispatcher** (caricatore sulla CPU) è il modulo del SO che dà il controllo della CPU ad un processo selezionato dallo schedulatore a breve termine
- Questa funzione comprende:
 1. cambio di contesto (**context switch**)
 2. passaggio alla modalità utente
 3. salto alla corretta locazione nel programma utente per ricominciare l'esecuzione
- **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

Sospensione del processo in esecuzione

- La **sospensione del processo di esecuzione** può avvenire attraverso una **chiamata**
 - Sincrona rispetto alla computazione, **in stato supervisore**
(in procedure di I/O, creazione processi)
 - Sincrona rispetto alla computazione, **in stato utente**
(in rilascio volontario)
 - Asincrona rispetto alla computazione
(allo scadere del time slice nel time sharing)
- **Salvataggio del contesto di esecuzione**
 - Salvare tutti i registri del processore sullo stack
 - Salvare lo stack pointer nel Process Control Block (PCB)

Riattivazione del processo

- **Ripristino del contesto di esecuzione**
 - Ripristinare il valore del registro che punta alla **base dello stack** prendendolo dal PCB del processo da riattivare
 - Ripristinare il valore dello **stack pointer** prendendolo dal PCB del processo da riattivare
 - Ripristinare **tutti i registri** del processore prendendoli dallo stack

Criteri di schedulazione (1)

1. Utilizzo della CPU: la CPU deve essere attiva il più possibile
 - In un sistema reale si va dal 40% (sistema poco carico) al 90% (utilizzo intenso)
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
 - Comprende tempi di esecuzione, tempi di attesa nelle varie code

Criteri di schedulazione (2)

4. Tempo di attesa: somma dei tempi spesi in attesa nella coda dei processi pronti
 - L'algoritmo di scheduling influisce solo sul tempo di attesa, non sul tempo di esecuzione
5. Tempo di risposta: tempo che intercorre dalla formulazione della richiesta fino alla produzione della prima risposta
 - Si conta il tempo necessario per iniziare la risposta, non per emetterla completamente

Criteri di ottimizzazione

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

In genere si ottimizza:

- il **valore medio** e/o
- Valore **minimo/massimo** 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)

Gli algoritmi 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

Il nostro termine di paragone sarà il **tempo di attesa medio**

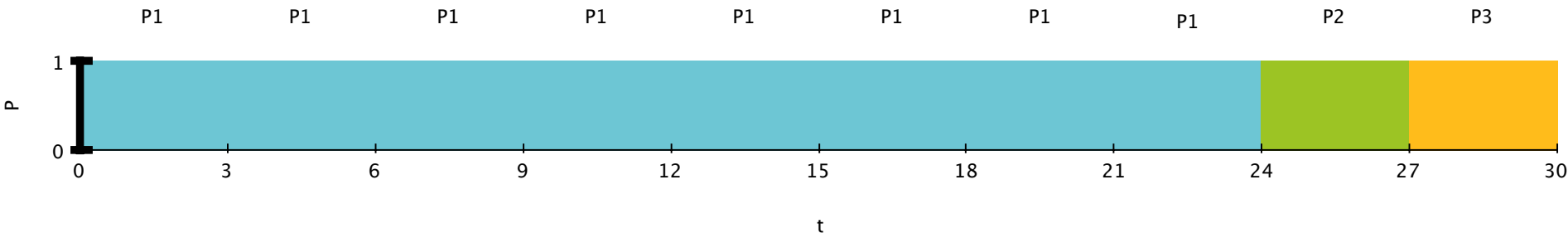
Schedulazione First-Come, First-Served (FCFS) (1)

- 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)

Processo	Durata del picco
P_1	24
P_2	3
P_3	3

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



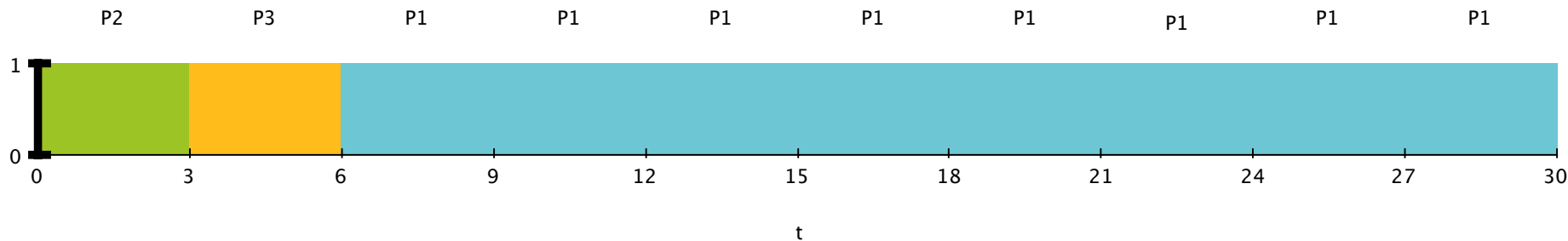
- Tempo di attesa per $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Tempo di attesa medio: $(0 + 24 + 27)/3 = 17$

Schedulazione First-Come, First-Served (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

Schedulazione First-Come, First-Served (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

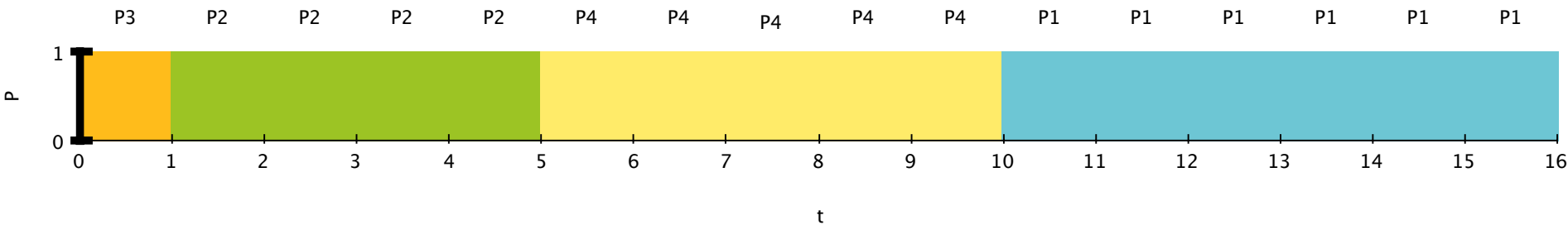
- Associa a ciascun processo “la lunghezza del successivo picco di CPU del processo medesimo”, per **schedulare il processo con il minor tempo**
 - Nota bene: ci si basa sulla lunghezza del prossimo picco, non su quella totale
 - A parità di lunghezza del picco successivo si applica FCFS
- **SJF è ottimale**: fornisce il minor tempo di attesa medio per un dato gruppo di processi

Esempio di SJF

Processo	Durata del picco
P ₁	6
P ₂	4
P ₃	1
P ₄	5

Tutti i processi
arrivano
contemporaneamente

- SJF (non-preemptive)



- Tempo di attesa medio = $(10 + 1 + 0 + 5)/4 = 4$

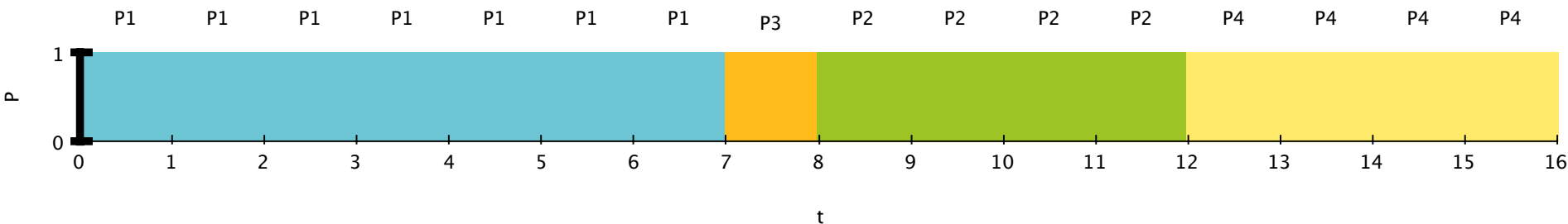
Shortest-Job-First (SJF): preemption

- Nel caso tutti i processi arrivino contemporaneamente e non ci sono problemi di preemption
- Quando invece i processi possono arrivare in tempi distinti (sistemi reali) esistono 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 un processo arriva nella coda dei processi pronti con un tempo di computazione minore del tempo che rimane al processo correntemente in esecuzione, **l'algoritmo ferma il processo corrente**
 - Questa schedulazione è anche detta *shortest-remaining-time-first*

Esempio di Non-Preemptive SJF

Processo	Durata del picco	Tempo di arrivo
P ₁	7	0
P ₂	4	2
P ₃	1	4
P ₄	4	5

- SJF (non-preemptive)

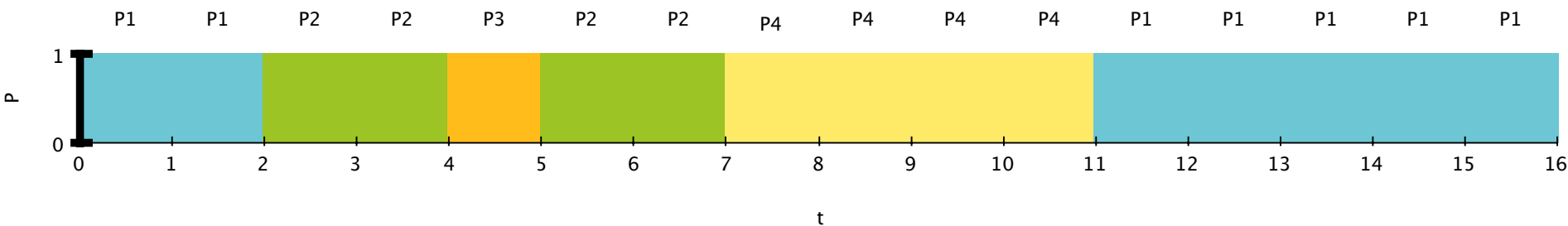


- Tempo di attesa medio = $[0 + (8-2) + (7-4) + (12-5)]/4 = 4$

Esempio di Preemptive SJF

Processo	Durata del picco	Tempo di arrivo
P ₁	7	0
P ₂	4	2
P ₃	1	4
P ₄	4	5

- SJF (preemptive)



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

Schedulatore a breve e 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 di **breve termine** può essere usato cercando di predire il tempo di esecuzione.

Prevedere la lunghezza del successivo picco di CPU

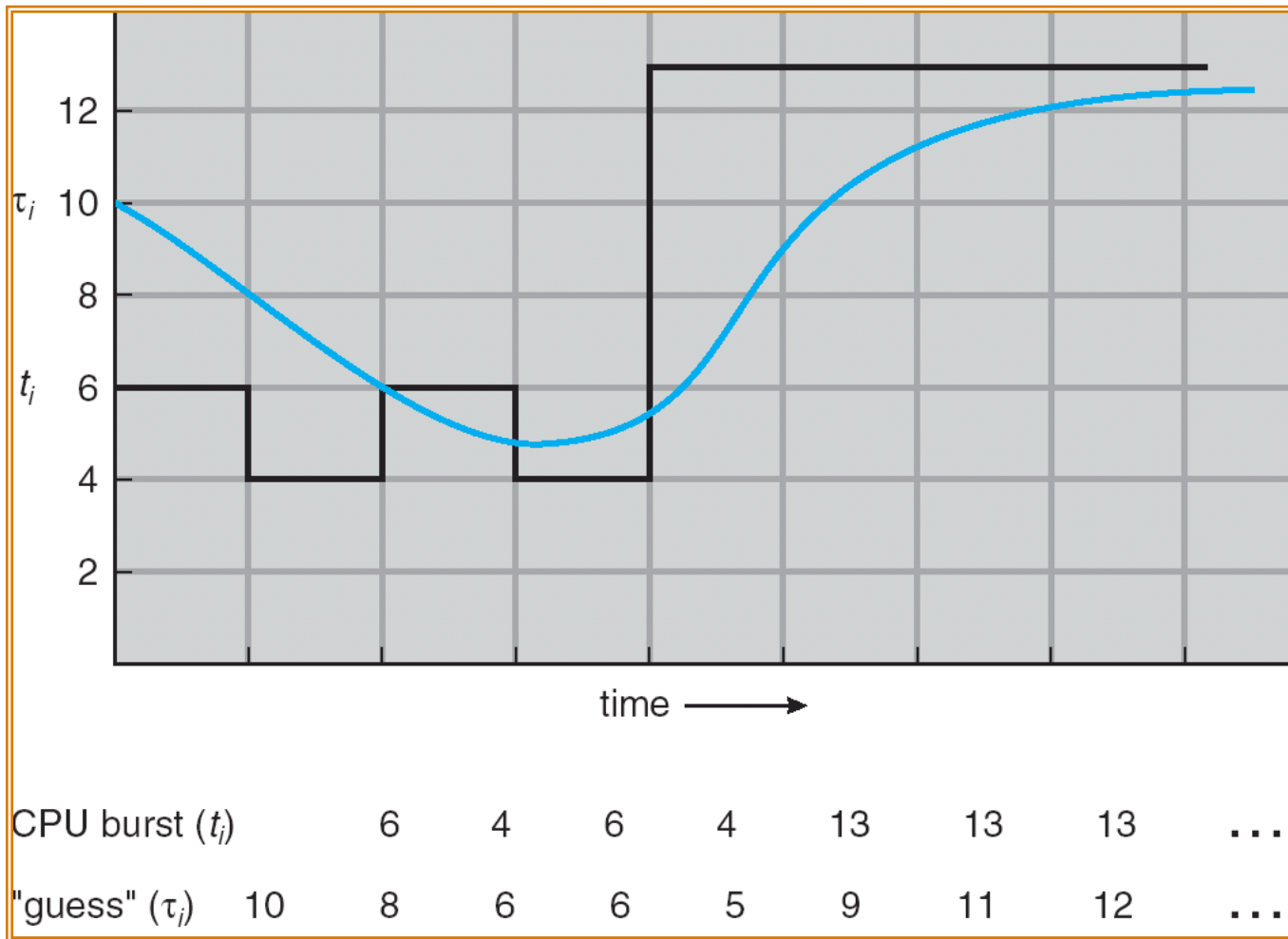
- **SJF ottimale, ma ideale**: non è possibile conoscere la lunghezza del successivo picco di CPU
- Possiamo però prevederla, ovvero **stimarla usando una media esponenziale** delle lunghezze dei picchi di CPU precedenti

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 costante
3. α , $0 \leq \alpha \leq 1$ è il peso della nostra predizione, solitamente $\alpha = 1/2$
4. Definiamo:

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

Storia recente \rightarrow t_n τ_n \leftarrow Storia passata

Previsione della durata del prossimo picco di CPU



Esempi di media esponenziale

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

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

- Se sviluppiamo la formula, otteniamo:

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

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

Schedulazione a priorità (1)

- Si associa **una priorità numerica (un intero) a ciascun processo**
- La CPU è allocata al processo con priorità più alta
 - **Preemptive**
 - **Non-preemptive**
 - I processi con priorità uguale vengono schedulati in ordine FCFS
- In alcuni sistemi la priorità più alta corrisponde al numero più basso
- In altri al numero più alto
- 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 (e.g. limiti di tempo)
 - Esterni: usano criteri esterni al sistema (e.g. importanza del processo)
- 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)

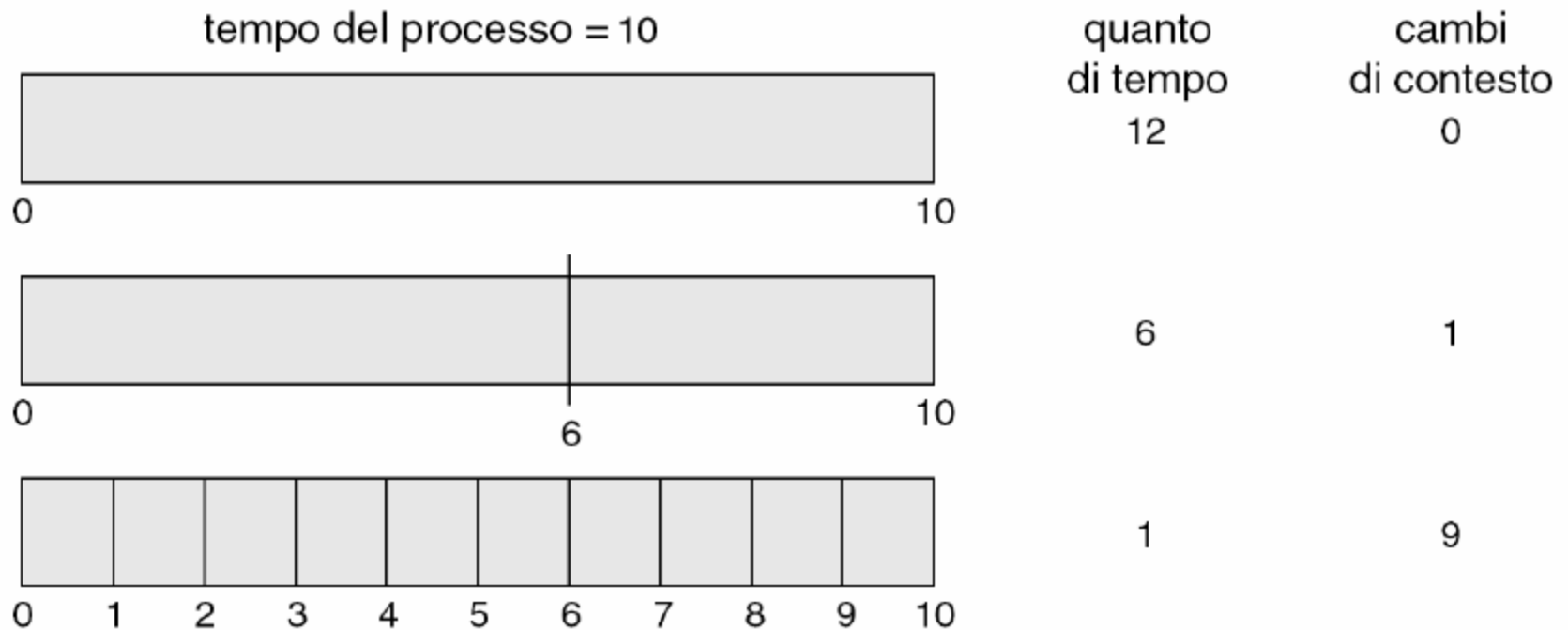
- **FCFS + preemption** per alternare i processi (coda FIFO circolare)
- Ogni processo possiede un **quanto di tempo (time slice) q** di utilizzo della CPU
 - generalmente $q=10-100$ ms
 - Se entro questo arco di tempo il processo non lascia la CPU, viene interrotto e rimesso nella coda dei processi 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 $(n - 1) \times q$ unità di tempo
 - Ad es. con 5 processi e $q=20$ ms, ciascun processo avrà al max 20 ms ogni 100
- Nuovi processi vengono aggiunti in alla fine della coda dei processi pronti

Round Robin (RR) – scelta del quanto (1)

- 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

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

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



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

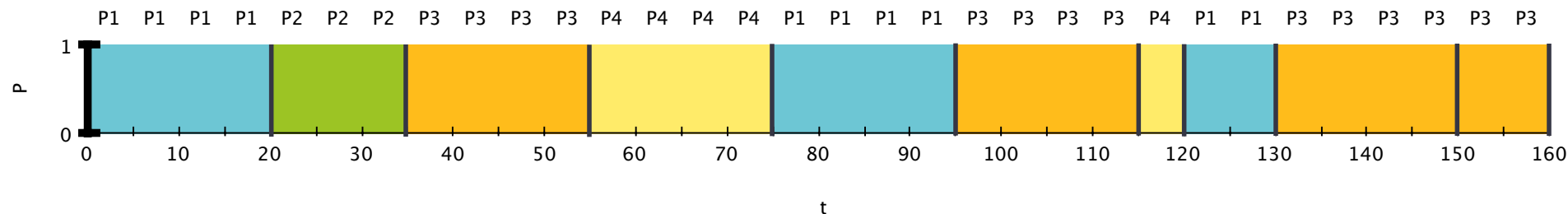
Considerando l'effetto del context-switch

- Si vuole che **q** sia **lungo rispetto al tempo per il cambio di contesto**, altrimenti l'overhead sarebbe troppo elevato
- Regola empirica: l'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 μ s (10^{-6} secondi)

Esempio di RR con quanto di tempo = 20

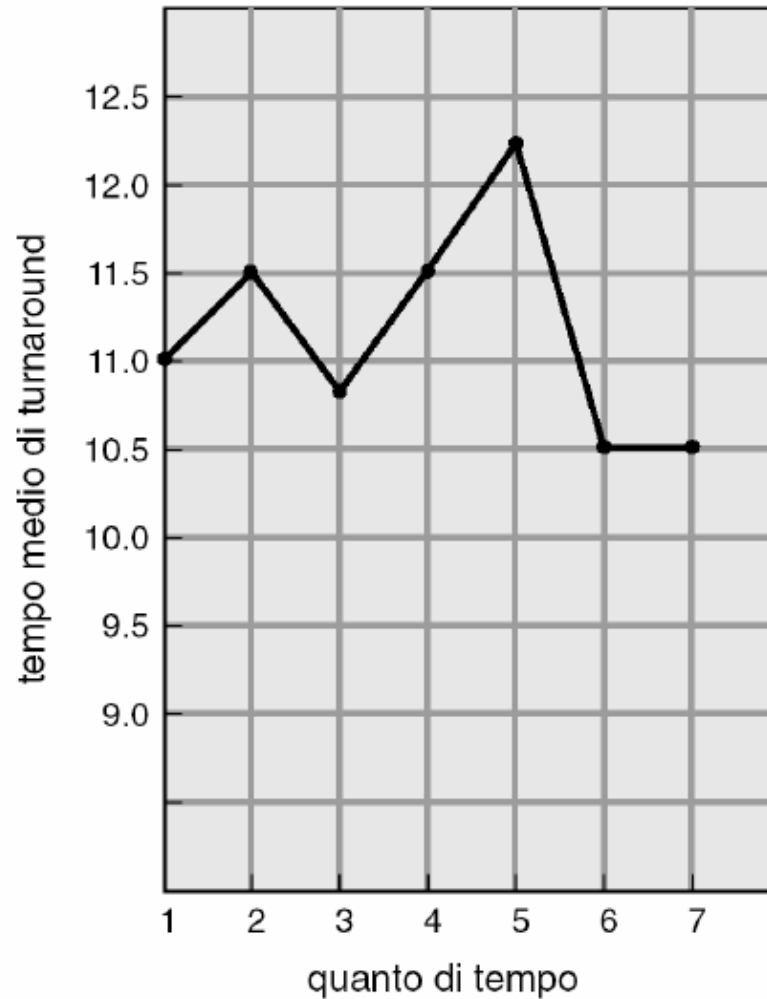
Processo	Durata del picco
P ₁	50
P ₂	15
P ₃	70
P ₄	25

- Diagramma di Gantt:



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

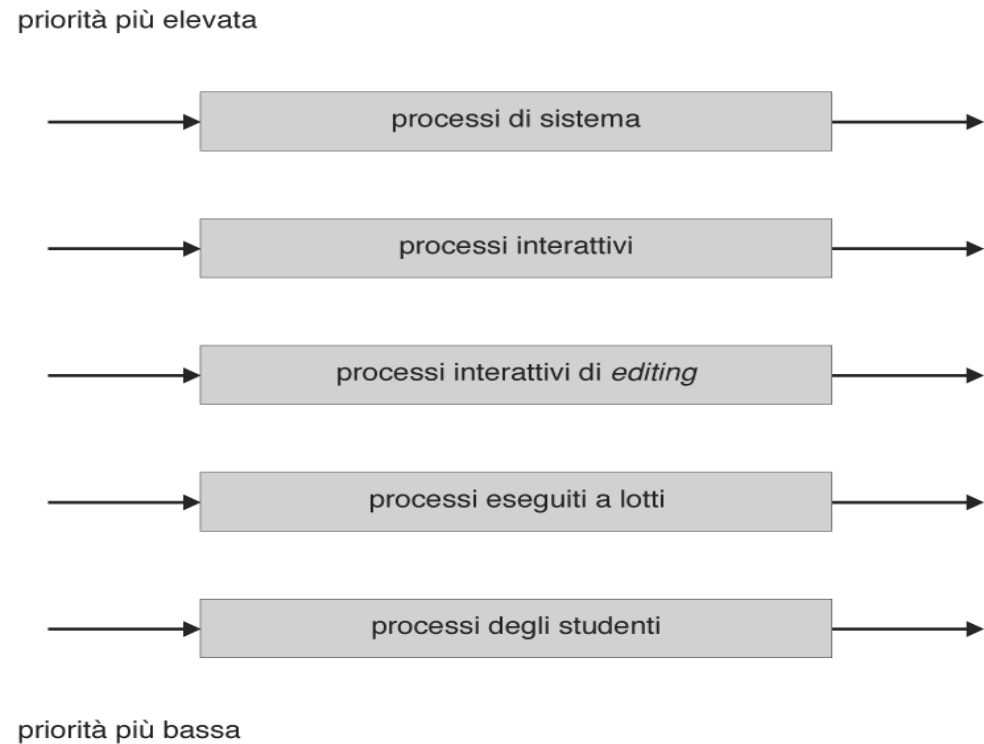
Variazione del Turnaround Time con il quanto di tempo



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 ai tempi di risposta, alle necessità di scheduling e all'importanza
- La coda dei processi pronti è ripartita in **code separate**, ad esempio:
 - foreground (interattivi)
 - background (batch – sullo sfondo)
- Ciascuna coda ha il **suo** algoritmo di schedulazione:
 - foreground – RR
 - background – FCFS



Coda a più livelli (2)

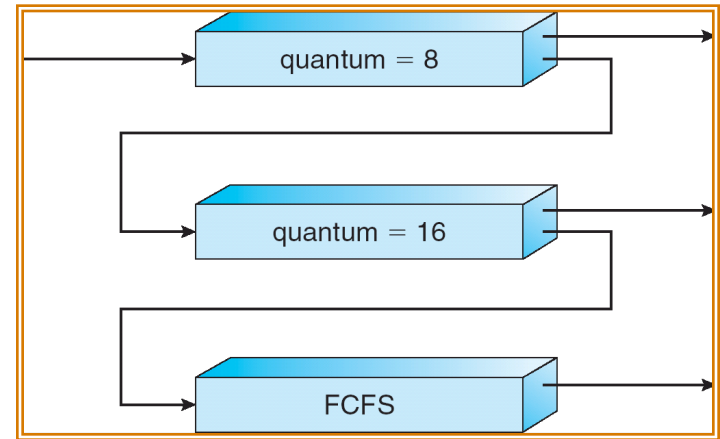
- Ci deve essere una schedulazione **anche tra le code**
 - Tipicamente una schedulazione preemptive a priorità fissa
 - La coda dei processi in foreground può avere priorità assoluta su quella dei processi in background
 - Significa che un processo di background può essere eseguito solo se nessun processo di foreground è in coda
 - E che un processo di foreground può interrompere un processo di background
 - Possibilità di starvation
- **In alternativa**, partizione del tempo tra le code: 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 schedatore 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 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 (passati 24ms), 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
- Distinguiamo tra:
 - **Schedulazione locale**: come la libreria dei thread decide quali thread eseguire su un LWP libero
 - *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
 - I SO che implementano la mappatura uno-a-uno, utilizzano solamente la SCS

Schedulazione *multiprocessing*

- Con più CPU, la schedulazione diviene più complessa
- Si ipotizza di avere **processori omogenei**:
 - Suddivisione del carico (load sharing)
 - Due approcci di schedulazione
 - **Multiprocessamento asimmetrico**: solo un processore (il master) prende le decisioni relative allo scheduling. Gli altri processori fanno solo elaborazione.
 - **Multiprocessamento simmetrico (SMP)**: ciascun processore schedula se stesso selezionando un processo dalla coda comune dei processi pronti o da una coda specifica per se stesso
 - Vanno progettati con cura per garantire sincronizzazione nell'accesso alla coda
 - Alcuni sistemi SMP applicano il principio di “predilezione del processore”

Predilezione del processore

- Tipicamente un processo che viene eseguito su un processore salve dei dati nella cache del processore stesso
- Se il processo viene successivamente mosso su un altro processore i dati nella cache del vecchio processore devono essere invalidati
 - Questo è molto costoso
- Per questo motivo molti sistemi SMP cercano di evitare che un processo cambi processore (predilezione del processore)
 - Predilezione debole (non garantito)
 - Predilezione forte (garantito)

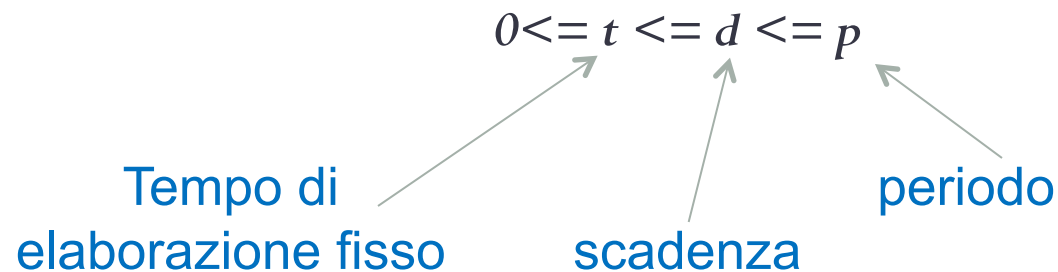
Schedulazione Real-Time

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

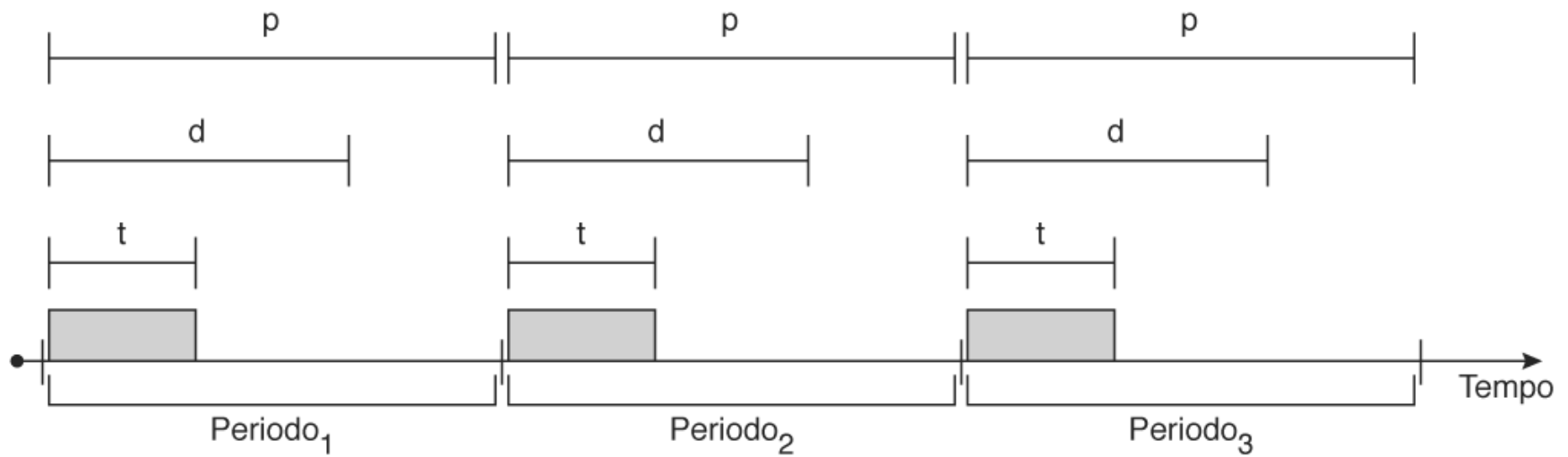
- Computazione **soft real-time (in tempo reale lasco)**
 - **richiede** che i processi critici ricevano priorità su quelli meno importanti

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*:



Schedulazione Hard Real-Time (2)



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, può interrompere il processo in esecuzione se esso è a priorità più bassa
 - Controllo dell'ammissione: si considera

$$\% \text{ uso della CPU per un processo} = t/p$$

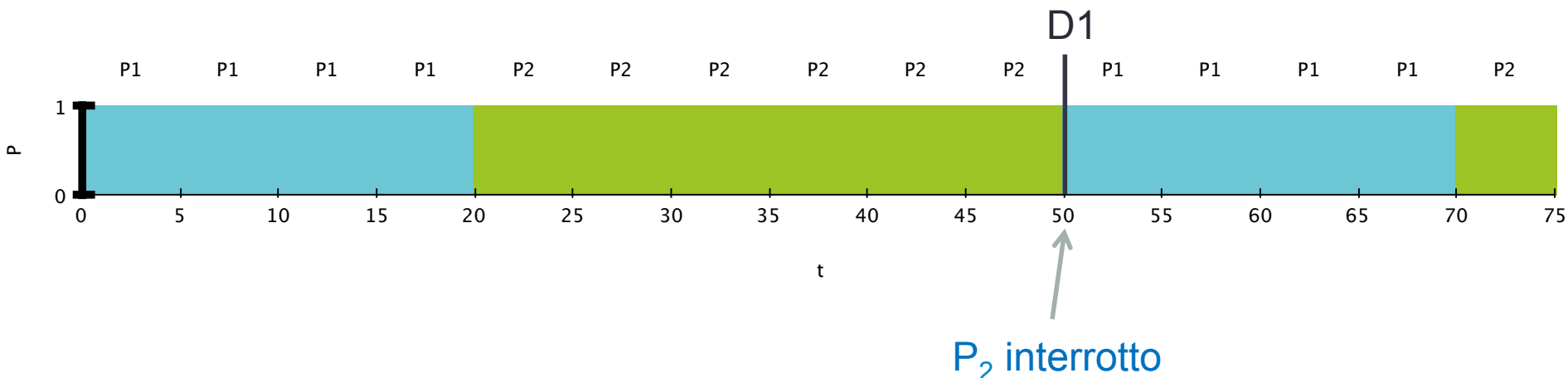
- Algoritmo ottimale, ma l'utilizzo della CPU non è massimizzato
 - Ottimale nel senso che se questo algoritmo non è in grado di trovare una sequenza di scheduling, allora nessun algoritmo a priorità statica può riuscirci

Esempio di schedulazione Hard Real-Time a frequenza monotona (1)

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P ₁	20	50	0.40
P ₂	35	100	0.35

Totale: 75% < 100%: schedulazione teoricamente ammissibile!

P₁ ha priorità maggiore di P₂ perché ha periodo più breve

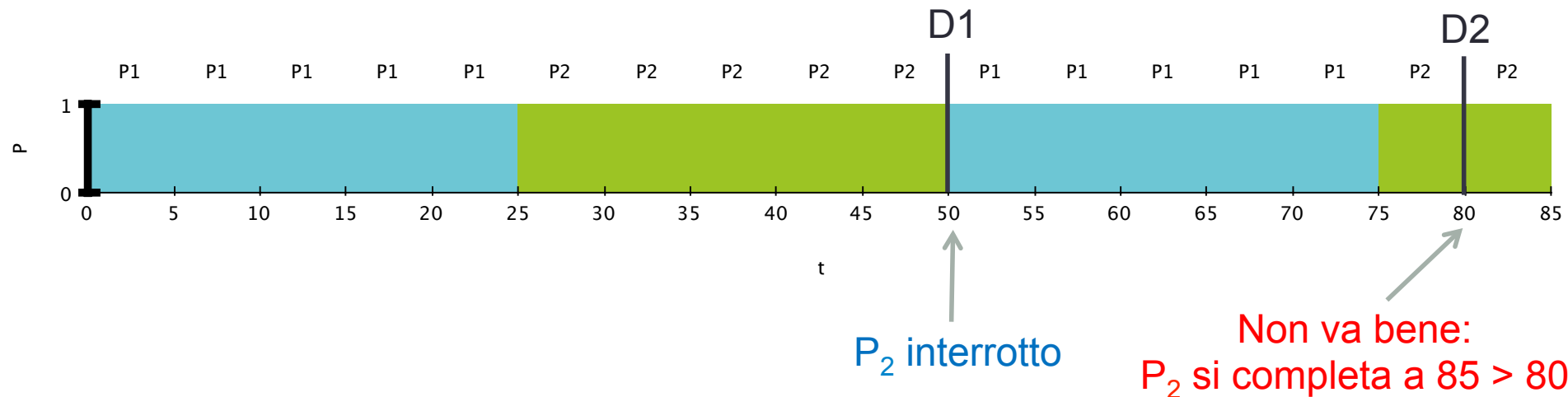


Esempio di schedulazione Hard Real-Time a frequenza monotona (2)

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P ₁	25	50	0.50
P ₂	35	80	0.44

Totale: 94% < 100%: schedulazione teoricamente ammissibile!

P₁ ha priorità maggiore di P₂ perché ha periodo più breve



Algoritmo di schedulazione a frequenza monotona

Ammissibilità

- L'algoritmo permette di utilizzare la CPU in modo limitato
- In particolare il caso peggiore è:

$$2(2^{\frac{1}{n}} - 1)$$

- Per $n=2$ l'utilizzo della CPU è limitato al 83% circa
 - Questo è il motivo per cui nel primo caso è possibile schedulare i processi
 - Utilizzo CPU = 75% < 83%
 - Mentre nel secondo no
 - Utilizzo CPU = 94% > 83%

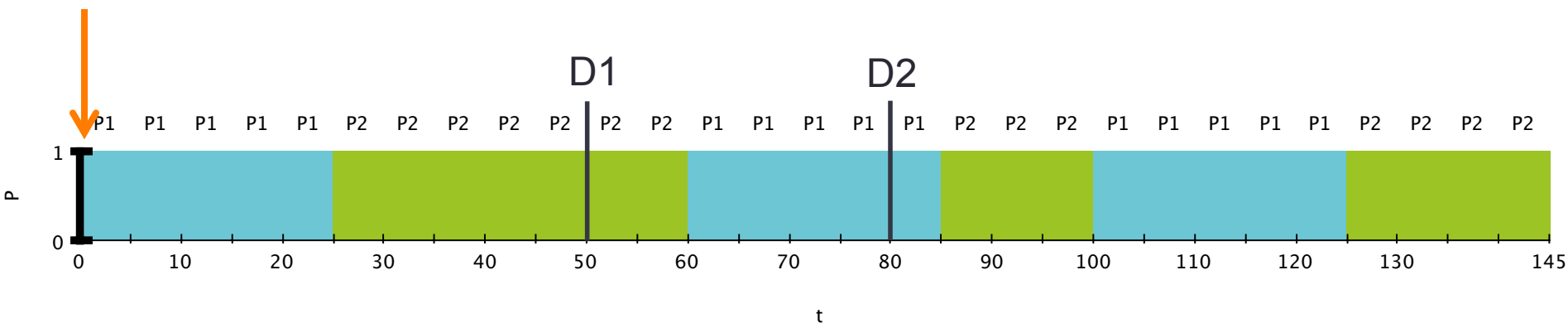
Algoritmo di schedulazione a scadenza più urgente

- Schedula i processi assegnando le priorità dinamicamente a seconda delle scadenze
 - Quando un processo diventa eseguibile
 - Deve annunciare la sua prossima scadenza allo schedulatore
 - La priorità viene **dinamicamente** calcolata in base alla **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
- Ottimale e (idealmente) porta l'utilizzo della CPU al 100%

Esempio di schedulazione Hard Real-Time a scadenza più urgente

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P_1	25	50	0.50
P_2	35	80	0.44

- Istante 0
 - P_1 pronto, scadenza = 50, priorità = 2
 - P_2 pronto, scadenza = 80, priorità = 1
 - P_1 va in esecuzione

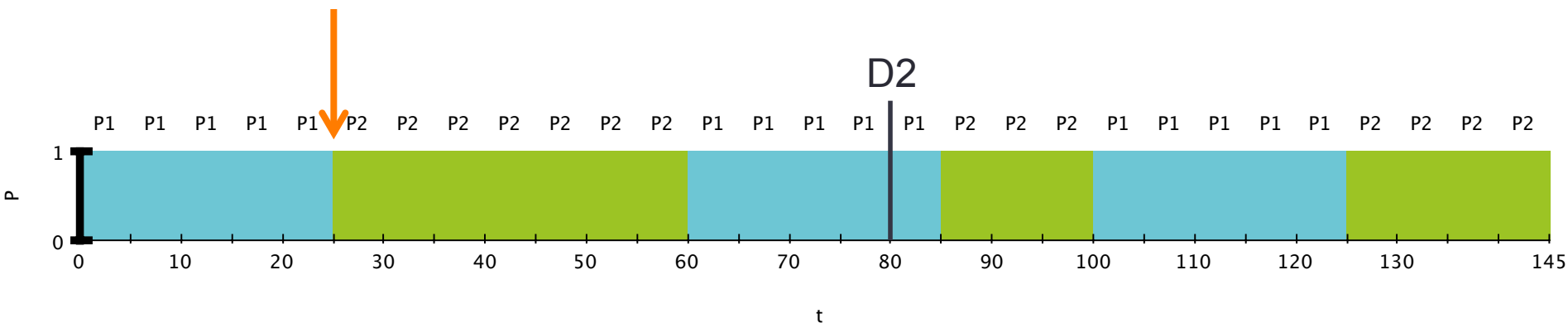


Esempio di schedulazione Hard Real-Time a scadenza più urgente

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P ₁	25	50	0.50
P ₂	35	80	0.44

- Istante 25

- P1 ha completato la sua elaborazione, prossima attivazione = 50, priorità = 1
- P2 pronto, scadenza = 80, priorità = 2
- P2 va in esecuzione

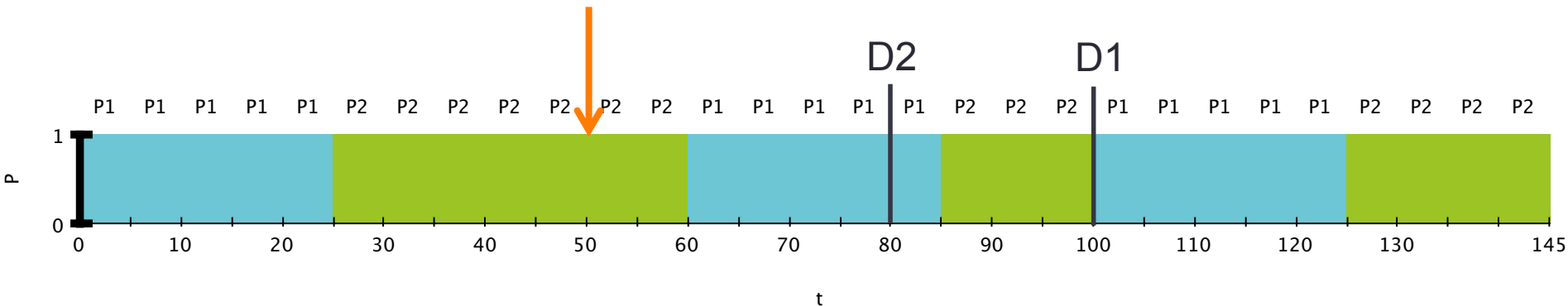


Esempio di schedulazione Hard Real-Time a scadenza più urgente

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P ₁	25	50	0.50
P ₂	35	80	0.44

- Istante 50

- P1 pronto, scadenza = 100, priorità = 1
- P2 pronto, scadenza = 80, priorità = 2
- P2 rimane in esecuzione, rispetto allo scheduler precedente P2 non viene interrotto

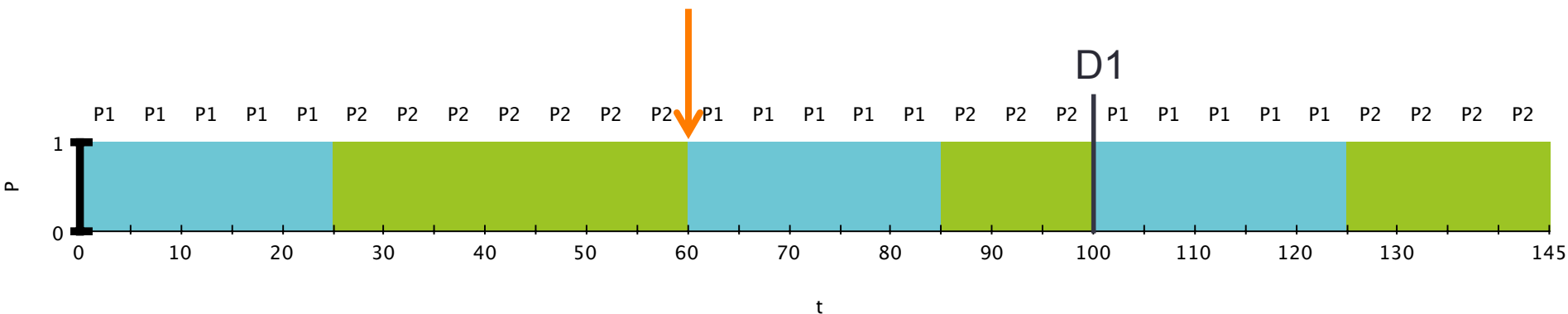


Esempio di schedulazione Hard Real-Time a scadenza più urgente

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P_1	25	50	0.50
P_2	35	80	0.44

- Istante 60

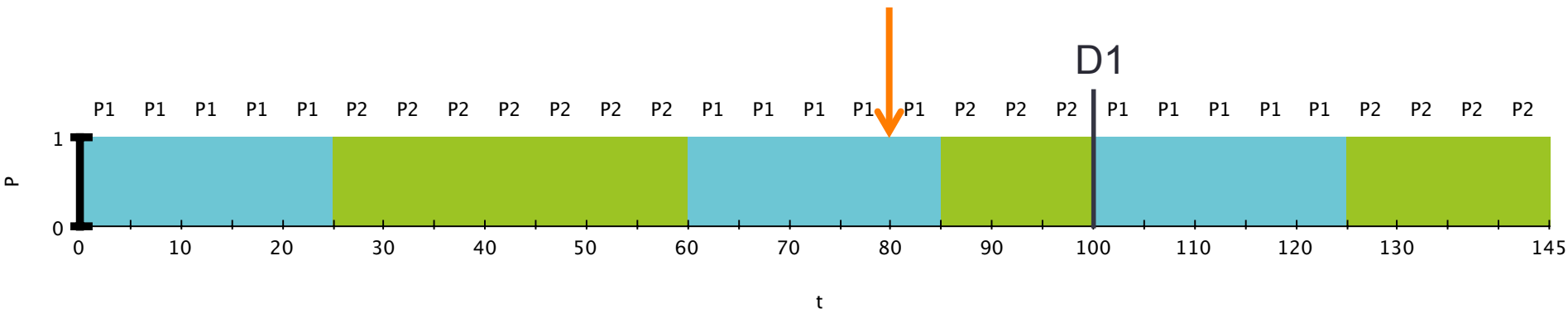
- P_1 pronto, scadenza = 100, priorità = 2
- P_2 ha completato la sua elaborazione, prossima attivazione = 80, priorità = 1
- P_1 va in esecuzione



Esempio di schedulazione Hard Real-Time a scadenza più urgente

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P_1	25	50	0.50
P_2	35	80	0.44

- Istante 80
 - P_1 in esecuzione, scadenza = 100, priorità = 2
 - P_2 pronto, scadenza = 160, priorità = 1
 - P_1 rimane in esecuzione

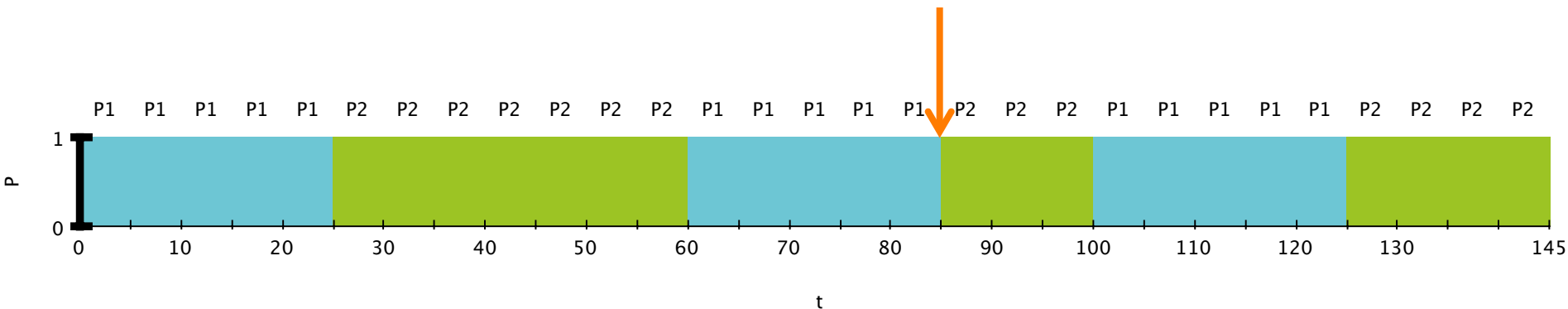


Esempio di schedulazione Hard Real-Time a scadenza più urgente

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P ₁	25	50	0.50
P ₂	35	80	0.44

- Istante 85

- P1 ha completato la sua elaborazione, prossima attivazione = 100, priorità = 1
- P2 pronto, scadenza = 160, priorità = 2
- P2 va in esecuzione

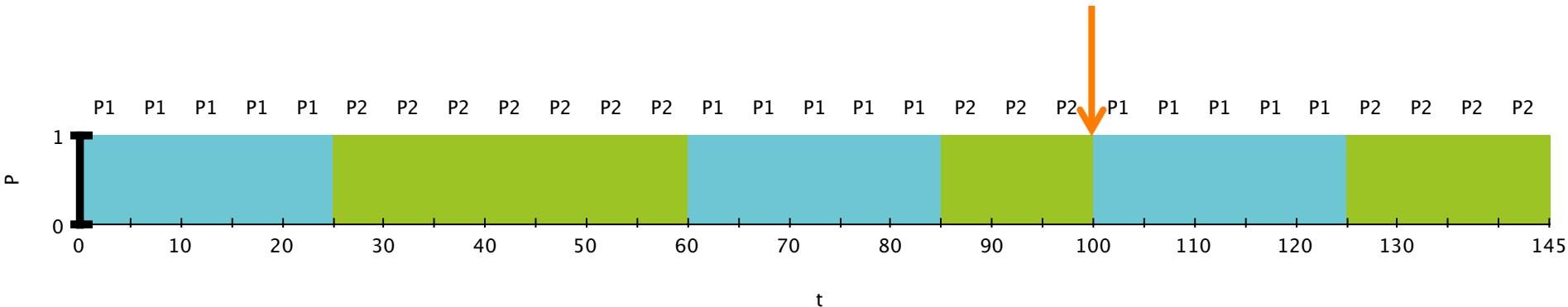


Esempio di schedulazione Hard Real-Time a scadenza più urgente

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P ₁	25	50	0.50
P ₂	35	80	0.44

- Istante 100

- P1 pronto, scadenza = 150, priorità = 2
- P2 pronto, scadenza = 160, priorità = 1
- P1 interrompe P2 e va in esecuzione, a P2 restano 20 ms per completare la sua elaborazione

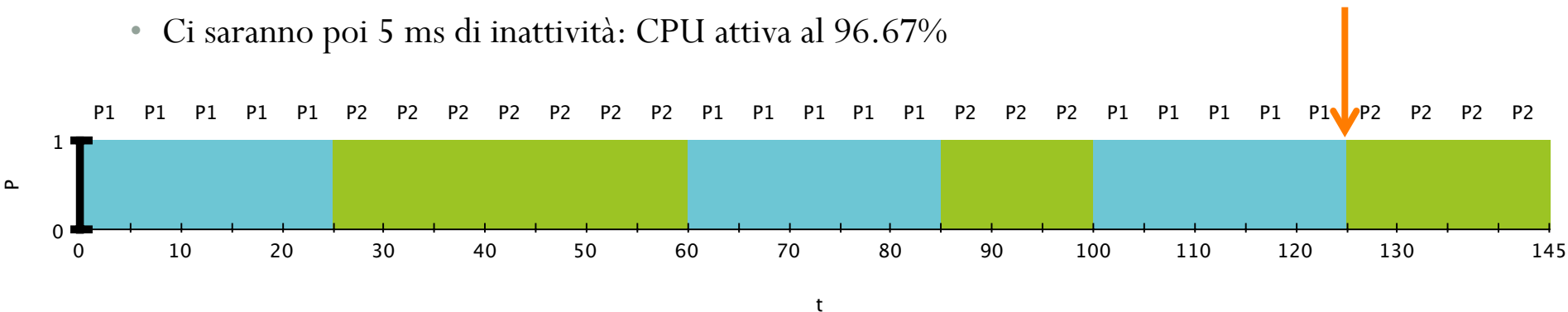


Esempio di schedulazione Hard Real-Time a scadenza più urgente

Processo	Durata	Periodo = Deadline	% uso CPU t/p
P ₁	25	50	0.50
P ₂	35	80	0.44

- Istante 125

- P1 ha completato la sua elaborazione, prossima attivazione = 150, priorità = 1
- P2 in attesa, scadenza = 160, priorità = 2
- P2 riprende l'esecuzione sospesa all'istante 100 e continua fino all'istante 145
- Ci saranno poi 5 ms di inattività: CPU attiva al 96.67%



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* (periodo che intercorre tra il tempo impiegato dallo scheduler per bloccare un processo e avviarne un altro)
 - Più è bassa più velocemente i processi real-time possono iniziare l'esecuzione

Schedulazione Soft Real-Time

- Come si fa a mantenere bassa la latenza di dispatch?
- È sufficiente permettere che il kernel sia interrotto
- Due possibili modi:
 - **Preemption point** (punti di sospensione): inseriti in punti **non inconsistenti** delle chiamate di sistema lunghe
 - Il punto di prelazione verifica se un processo a più alta priorità deve essere eseguito. In tal caso esegue un cambio di contesto; al termine di questo processo riprende l'esecuzione della chiamata di sistema

Schedulazione Soft Real-Time

- **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
 - **problema inversione priorità**
 - Tre processi L, M, H con priorità $P_L < P_M < P_H$
 - Supponiamo che L stia accedendo una risorsa R, la quale viene richiesta da H
 - H viene messo in attesa
 - Supponiamo che M diventi eseguibile e avendo priorità più alta di L lo sospenda
 - In questo modo M ha indirettamente allungato l'attesa di H sulla risorsa R, nonostante esso abbia priorità più bassa
 - Si risolve con il **protocollo di ereditarietà della priorità**: i processi che accedono alla risorsa ereditano momentaneamente la priorità più alta fra i processi in attesa

Schedulazione in Windows (1)

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

Schedulazione in Windows (2)

- Ogni coda è gestita attraverso l'algoritmo di Round Robin, tutte tranne l'ultima che è servita tramite FCFS.
- Un thread a priorità variabile può cambiare coda quando si verificano due eventi (feedback dello schedulatore):
 - Termina il suo quanto di tempo
 - La priorità del thread viene ridotta (limita l'uso della CPU ai thread CPU-bound)
 - Viene sospesa a causa di un'operazione di attesa
 - La priorità del thread 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 di tempo maggiore (tipicamente di un fattore 3)

Schedulazione in Windows (3)

- La schedulazione di un thread dipende dalla classe di priorità ma anche da una priorità relativa associata ad ogni thread
- Soft-real time

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)

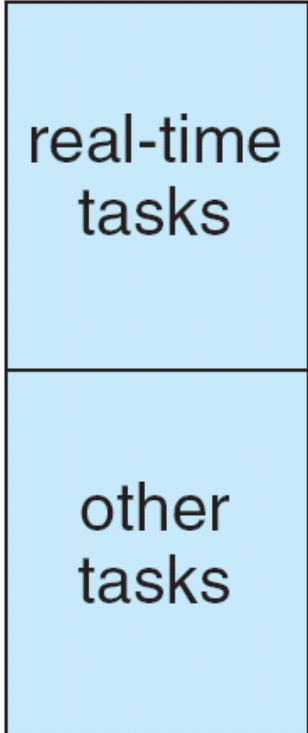
Schedulazione in Linux (1)

- Prima della versione 2.5
- 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

Schedulazione in Linux (2)

- Dopo la versione 2.5
- Supporto per SMP (ogni processore ha una coda dei processi)
- Schedulazione su scala di priorità
- Ogni processore ha una coda 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 esauriti diventa quella dei task attivi e vice-versa
- 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

Schedulazione in Linux (3)

<u>numeric priority</u>	<u>relative priority</u>		<u>time quantum</u>
0	highest		200 ms
• • • 99			
100			
• • •			
140	lowest		10 ms

Schedulazione in Linux (4)

- I task real-time hanno priorità statiche
- I task nice hanno priorità dinamiche
 - Valore di nice ± 5 punti
 - Il valore sommato dipende dal tempo trascorso in attesa dai task prima di eseguire l'I/O
 - Tipicamente i task più interattivi prendono un bonus maggiore (più negativo, in quanto la priorità maggiore corrisponde al numero più basso)