



UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Sistemi Operativi

Scheduling

LEZIONE 16

prof. Antonino Staiano

Corso di Laurea in Informatica – Università di Napoli Parthenope

antonino.staiano@uniparthenope.it

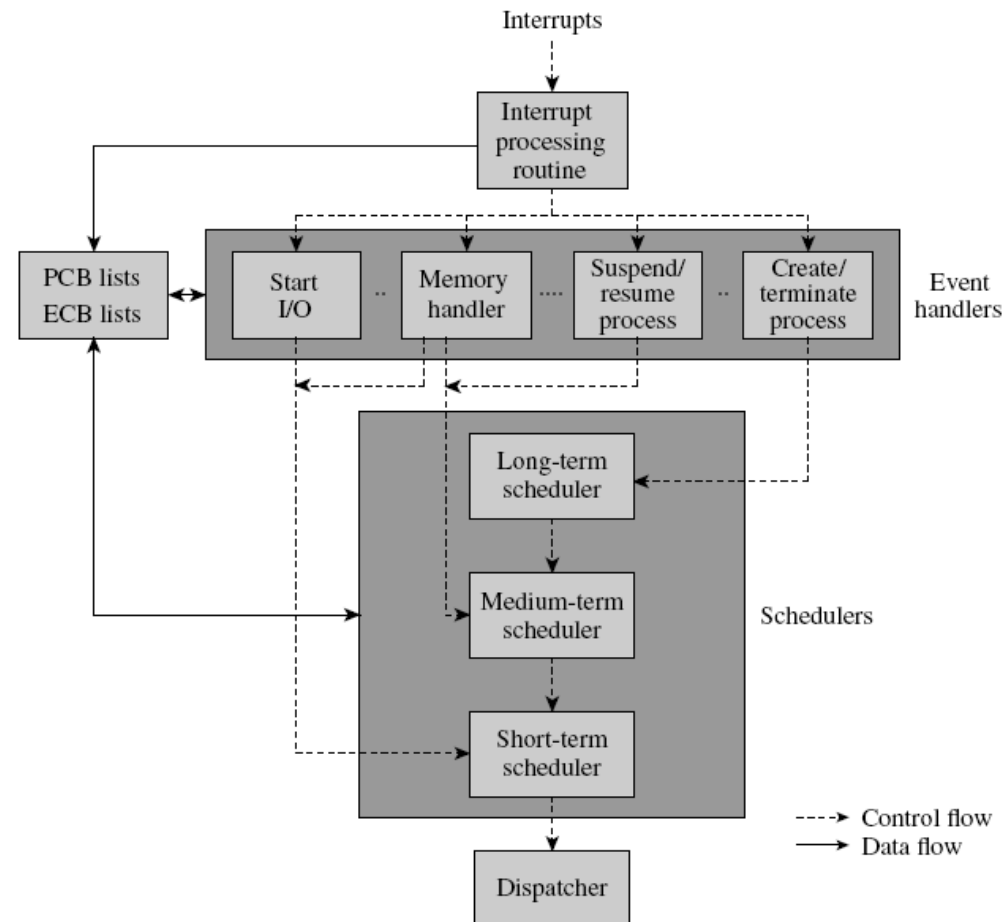
Scheduling in Pratica

- Per fornire una combinazione appropriata di prestazioni di sistema e servizio utente, il SO deve adattare il suo funzionamento alla natura e al numero delle richieste utente, e alla disponibilità delle risorse
 - Un singolo scheduler che usa una strategia di scheduling classica non può affrontare in modo efficace tutti questi problemi
- I SO moderni impiegano più scheduler
 - Fino a tre scheduler
- Alcuni scheduler possono usare una combinazione di diverse strategie di scheduling

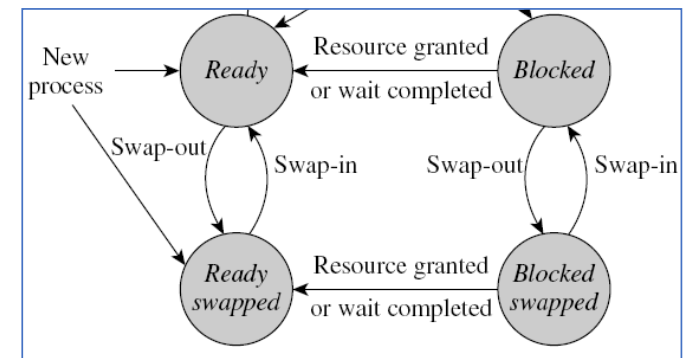
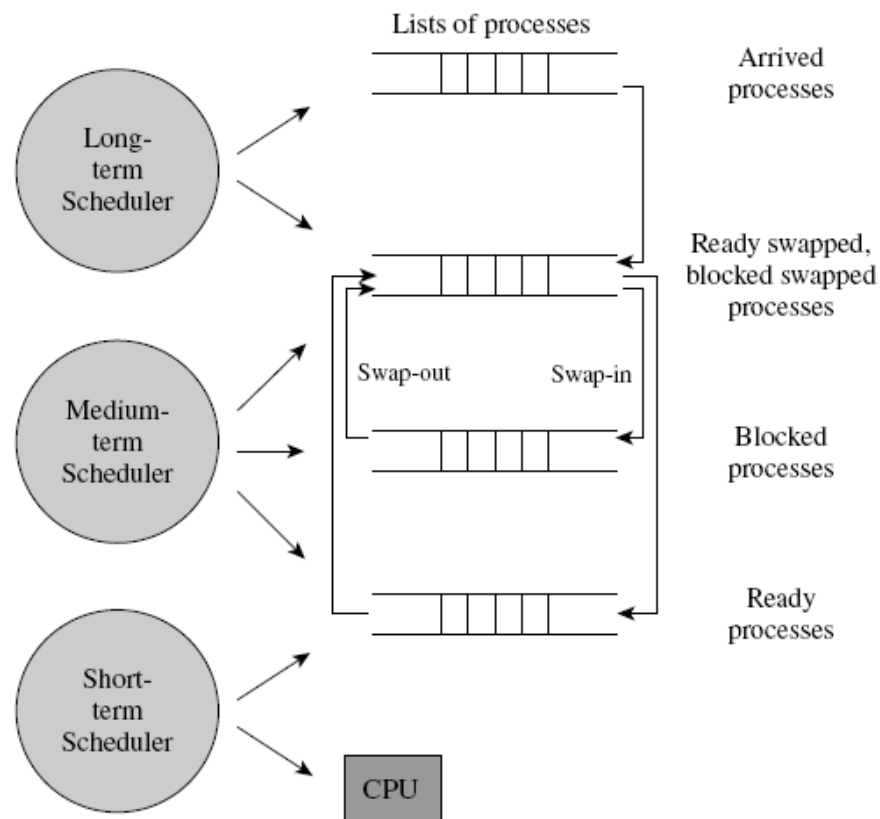
Scheduler a Lungo, Medio e Breve Termine

- Questi scheduler eseguono le funzioni seguenti:
 - **Lungo termine**: decide quando ammettere un processo appena arrivato allo scheduling, a seconda della:
 - Natura (se CPU-bound o I/O bound)
 - Disponibilità delle risorse
 - Strutture dati del kernel, spazio per lo swapping
 - **Medio termine**: decide quando fare swap-out di un processo dalla memoria e quando ricaricarlo in modo che ci sia un numero sufficiente di processi *ready* in memoria
 - **Breve termine**: decide quale prossimo processo ready servire con la CPU e per quanto tempo
 - Chiamato anche scheduler di processo, o scheduler

Gestione degli eventi e Scheduling

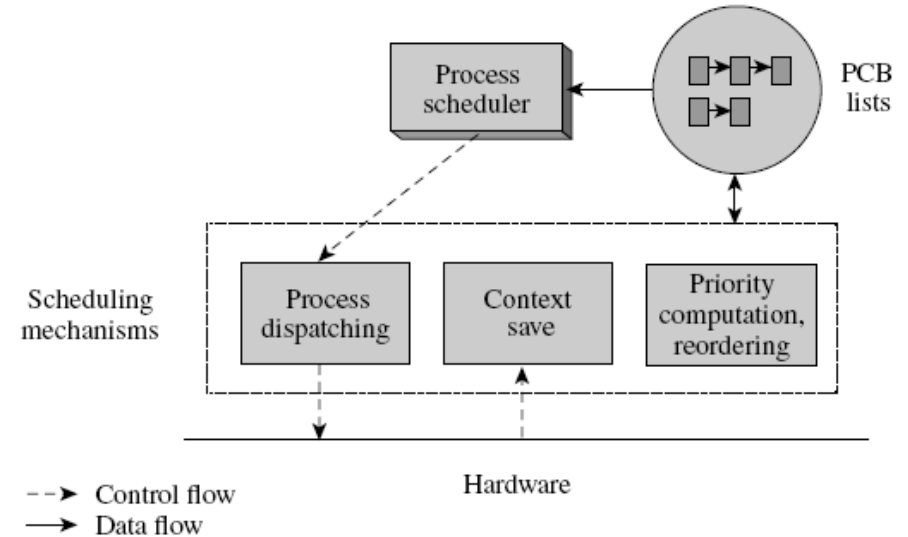


Esempio: Scheduling a Lungo, Medio e Breve Termine in un Sistema Time-Sharing



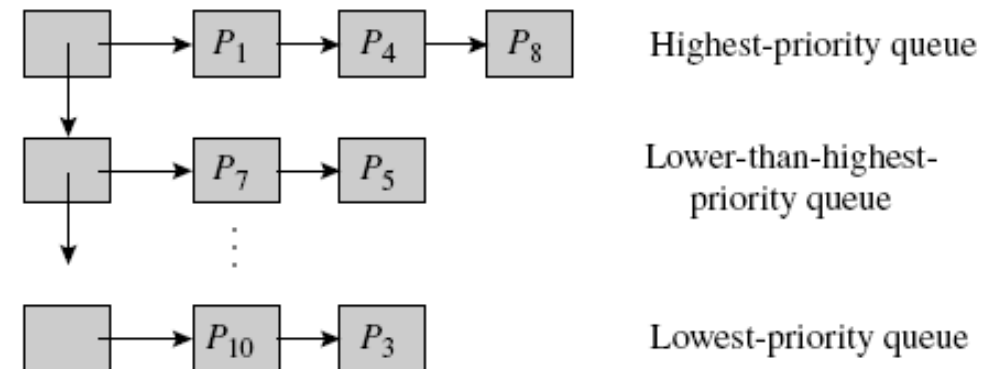
Strutture Dati e Meccanismi di Scheduling

- Lo scheduler usa diverse liste di PCB la cui organizzazione dipende dalla particolare tecnica di scheduling
- La routine di elaborazione dell'interrupt invoca il salvataggio del contesto
- Il dispatcher carica due campi del PCB, PSW e GPR nella CPU per ripristinare le operazioni del processo selezionato
- Il kernel esegue un *loop idle* se non ci sono processi ready



Scheduling basato su Priorità

- E' mantenuta una lista separata di processi ready per ogni valore di priorità
 - Organizzata come coda di PCB
- L'overhead dipende dal numero di priorità distinte, non dal numero dei processi ready
- Può portare a *starvation* dei processi a bassa priorità
 - L'aging può risolvere tale problema
- Può causare l'*inversione di priorità*
 - Gestito usando il protocollo di ereditarietà della priorità



Scheduling Round-Robin con Time Slicing

- Può essere implementato con una lista singola di PCB dei processi ready
 - La lista è organizzata come una coda
- Lo scheduler rimuove il primo PCB dalla coda e schedula il processo corrispondente
 - Se il *time slice* scade, il PCB è messo alla fine della coda
 - Se il processo avvia un'operazione di I/O, il suo PCB è aggiunto alla fine della coda quando la sua operazione di I/O è completata
- Il PCB di un processo ready si sposta verso la testa della coda fino a che il processo è schedulato

Scheduling Multilivello

- Combina lo scheduling con priorità e lo scheduling RR e gestisce diverse code di processi ready
- Ad ogni ready queue sono associati una priorità ed un time slice
 - All'interno viene eseguito uno scheduling RR con time slicing
 - La coda con alta priorità ha un piccolo time slice
 - Buoni tempi di risposta dei processi
 - La coda a bassa priorità ha un time slice più grande
 - Basso overhead per la commutazione di processo
- Un processo in testa ad una coda è schedulato solo se le code per tutti i livelli più elevati di priorità sono vuote
- Lo scheduling è con prelazione
- Le priorità sono **statiche**

Scheduling Multilivello Adattivo

- Chiamato anche scheduling con feedback multilivello
- Lo scheduler varia la priorità di un processo in modo che esso abbia un time slice consistente con il suo requisito di CPU
- Lo scheduler determina il livello di priorità «corretto» per un processo osservando il suo uso recente di CPU e I/O
 - Sposta i processi a tale livello
- Esempio: CTSS, un SO time-sharing per l'IBM 7094 negli anni '60
 - Struttura di priorità ad otto livelli

Multi-Level Feedback Queue (MLFQ)

- E' uno scheduler che impara dal passato per predire il futuro
- Obiettivi:
 - Ottimizzando il tempo di **turnaround**
 - Eseguendo prima i processi brevi
 - Minimizzare i tempi di risposta senza una conoscenza apriori della durata dei job

MLFQ: regole base

- MLFQ ha un numero di code distinte
 - Ad ogni coda è assegnato un livello di priorità
- Un processo pronto per l'esecuzione si trova su di una sola coda
 - Per l'esecuzione è scelto un processo sulla coda a priorità massima
 - Usa lo scheduling RR per i processi nella stessa coda

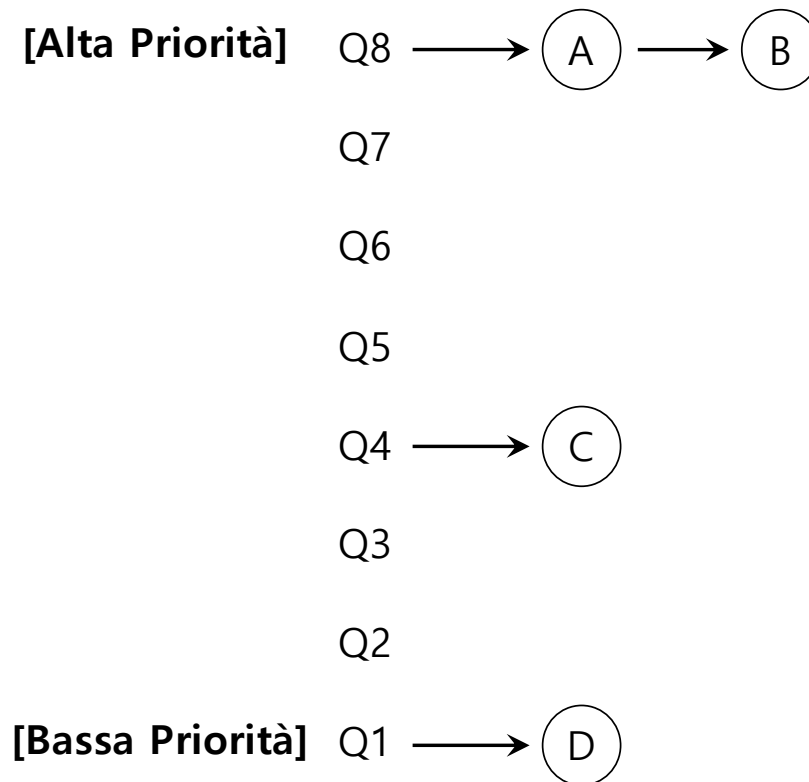
Regola 1: If $\text{Priorità}(A) > \text{Priorità}(B)$, eseguito A (e B no)

Regola 2: If $\text{Priorità}(A) = \text{Priorità}(B)$, A & B sono eseguiti con RR

MLFQ: regole base (Cont.)

- MLFQ modifica la priorità di un processo sulla base del suo **comportamento osservato**
- Esempio:
 - Un processo cede ripetutamente la CPU mentre è in attesa di I/O
 - Mantiene elevata la sua priorità
 - Un processo usa la CPU in modo intensivamente per lunghi periodi
 - Riduce la sua priorità

MLFQ esempio



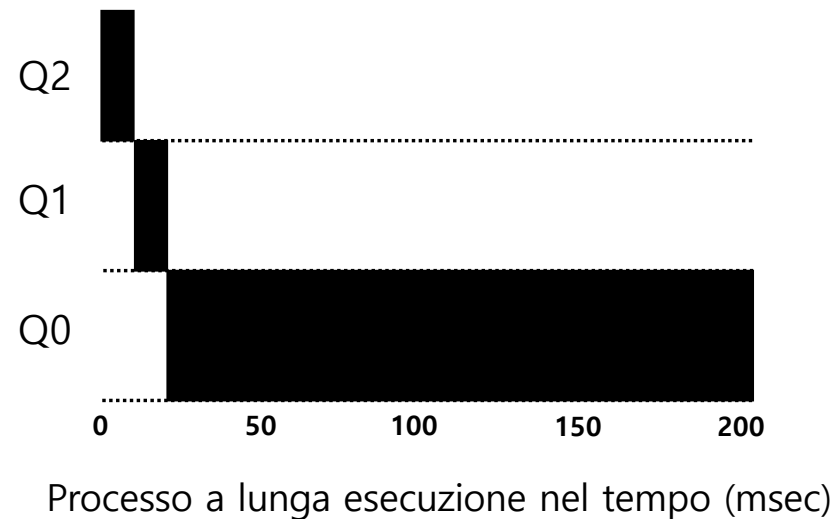
MLFQ: come modificare la priorità

- MLFQ algoritmo di modifica della priorità:
 - **Regola 3:** quando arriva un nuovo processo, è inserito nella coda con la priorità più alta
 - **Regola 4a:** se un processo usa un'intero time slice durante l'esecuzione, la sua priorità è abbassata
 - E' spostato nella coda di livello inferiore
 - **Regola 4b:** se un processo restituisce la CPU prima della fine del time slice, resta nella stessa coda (mantiene lo stesso livello di priorità)

MLFQ approssima SJF

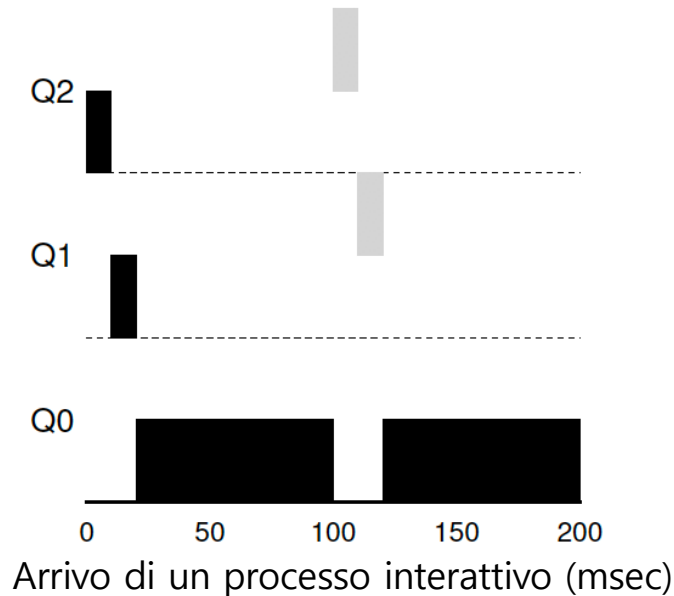
Esempio: un singolo a lunga esecuzione

- Uno scheduler a tre code con time slice di 10ms



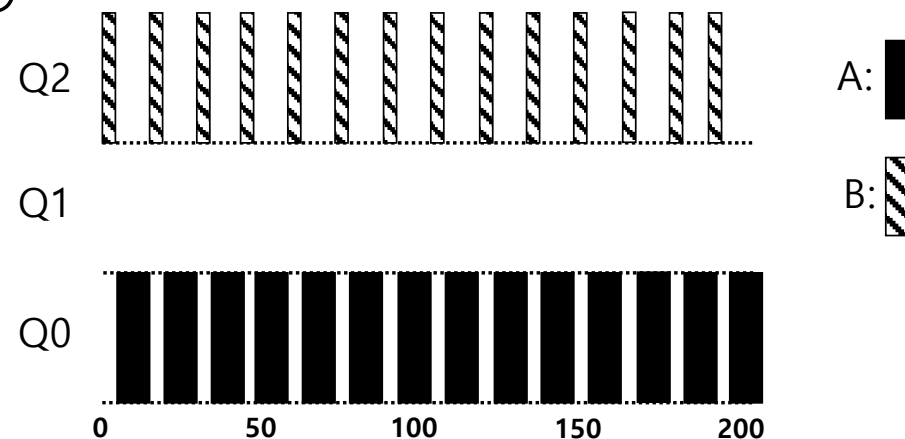
Esempio: arrivo di un processo breve

- Scenario:
 - **Processo A**: CPU-bound a lunga esecuzione
 - **Processo B**: processo interattivo a breve esecuzione (tempo servizio 20ms)
 - **A** eseguito durante un certo lasso di tempo, dopo arriva **B** al tempo $T=100$



Esempio: cosa accade se consideriamo l'I/O?

- Scenario:
 - **Processo A**: CPU-bound a lunga esecuzione
 - **Processo B**: processo interattivo che necessita della CPU per solo 1ms per poi eseguire un I/O



Un carico misto di processi I/O-bound e CPU-bound (msec)

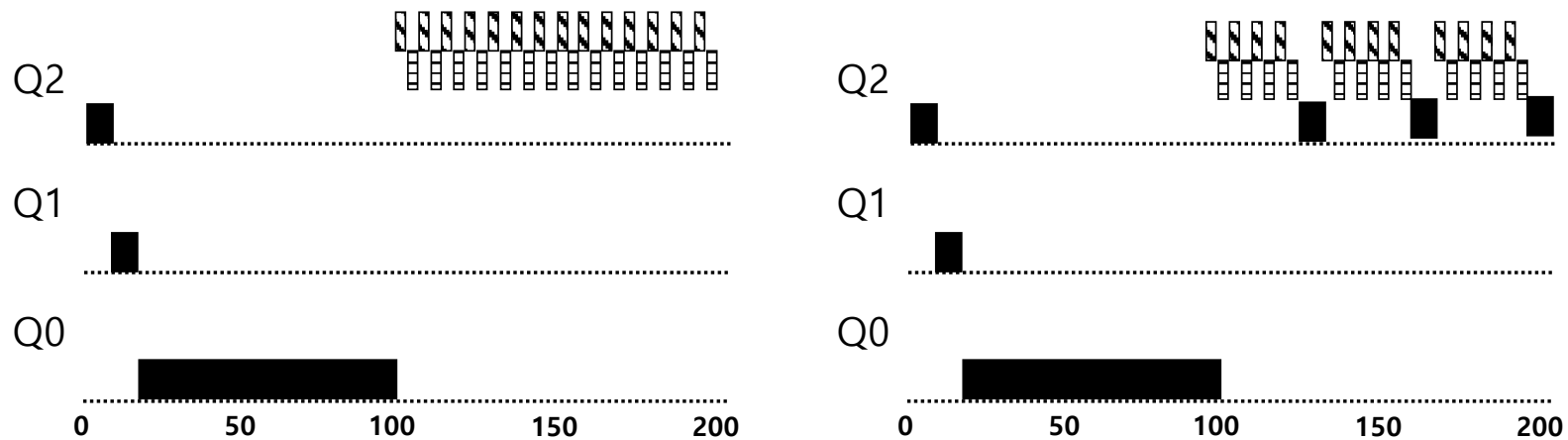
L'approccio MLFQ mantiene un processo interattivo alla priorità più alta

Problemi con l'MLFQ base

- **Starvation**
 - Se ci sono “troppi” processi interattivi nel sistema
 - Processi di lunga durata non riceveranno mai tempo di CPU
- **Manipolare lo scheduler**
 - Dopo l'esecuzione per il 99% del time slice, invoca un'operazione di I/O
 - Il processo guadagna una percentuale più alta del tempo di CPU
- Un programma può **cambiare il proprio comportamento** nel tempo
 - Processo CPU bound process → processo I/O bound

Aumento della priorità

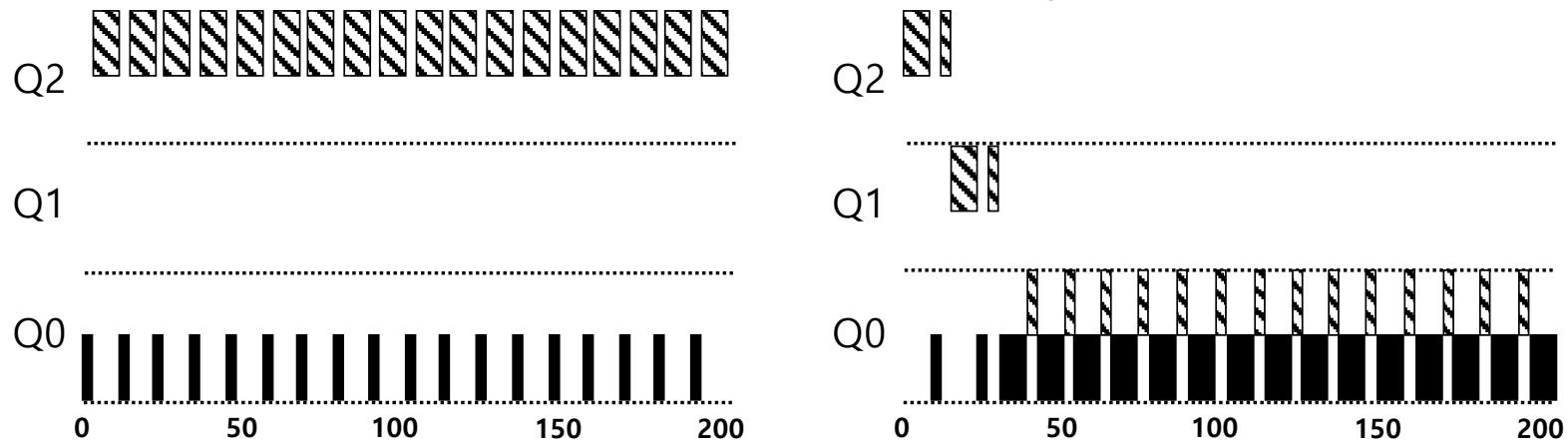
- **Regola 5:** dopo un certo periodo S , sposta tutti i processi nel sistema nella coda in cima a tutte
- Esempio:
 - Processo lungo (A) con due processi brevi interattivi (B, C)



Senza (sinistra) e con (destra) aumento della priorità A:  B:  C: 

Miglior gestione

- Come prevenire la manipolazione dello scheduler?
- Soluzione:
 - **Regola 4** (sostituisce Regole 4a e 4b): una volta che un processo **usa il suo allotment** ad un dato livello (indipendentemente da quante volte ha ceduto la CPU), la sua **priorità è abbassata** (spostata in una coda in basso)

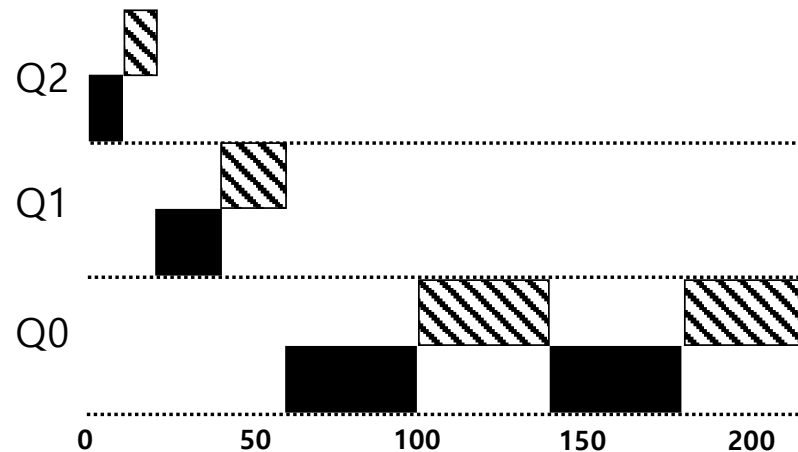


Senza (sx) e Con (dx) tolleranza alla manipolazione

Regolazione MLFQ e altri problemi

Minor priorità, time slice più lungo

- Code a Maggiore priorità → time slice più brevi
 - Ad esempio, 10 o meno millisecondi
- Coda a bassa priorità → time slice più lunghi
 - Ad esempio, 100 millisecondi



Esempio: 10ms per la coda più alta, 20ms per quella centrale e 40ms per la più bassa

Scheduling Real-Time

- Lo scheduling real-time deve gestire due vincoli di scheduling speciali per cercare di soddisfare le deadline delle applicazioni
 1. I processi nelle applicazioni real-time sono processi interagenti
 - La deadline di un'applicazione dovrebbe essere tradotta in deadline appropriate per i processi
 2. I processi possono essere periodici
 - Le differenti istanze di un processo possono arrivare a intervalli fissi e tutte devono soddisfare le rispettive deadline

Precedenze dei Processi e Schedulazioni ammissibili

- Sono considerate le dipendenze tra i processi (ad esempio, $P_i \rightarrow P_j$, P_i precede P_j) mentre si determinano le deadline e lo scheduling

Un grafo delle precedenze dei processi (PPG) è un grafo orientato $G \equiv (N, E)$ tale che $P_i \in N$ rappresenti un processo, ed un arco $(P_i, P_j) \in E$ implica $P_i \rightarrow P_j$. Quindi, un cammino P_i, \dots, P_k in PPG implica $P_i \rightarrow^* P_k$. Un processo P_k è un discendente di P_i se $P_i \rightarrow^* P_k$.

- E' garantito che i requisiti di risposta siano soddisfatti (sistemi real-time hard) o soddisfatti probabilisticamente (sistemi real-time soft), a seconda del tipo di sistema RT
 - Una **schedulazione ammissibile** è una sequenza di decisioni di scheduling che permette ai processi di un'applicazione di operare in accordo con le rispettive precedenze e di soddisfare i requisiti dell'applicazione
- Data l'applicazione, lo scheduling RT è orientato sull'implementazione di una schedulazione ammissibile, se ne esiste una

Approcci per lo scheduling Real-Time

Approach	Description
Static scheduling	A schedule is prepared <i>before</i> operation of the real-time application begins. Process interactions, periodicities, resource constraints, and deadlines are considered in preparing the schedule.
Priority-based scheduling	The real-time application is analyzed to assign appropriate priorities to processes in it. Conventional priority-based scheduling is used during operation of the application.
Dynamic scheduling	Scheduling is performed when a request to create a process is made. Process creation succeeds only if response requirement of the process can be satisfied in a guaranteed manner.

- Altra politica di scheduling dinamico: scheduling ottimista
 - Ammette tutti i processi; può perdersi qualche deadline

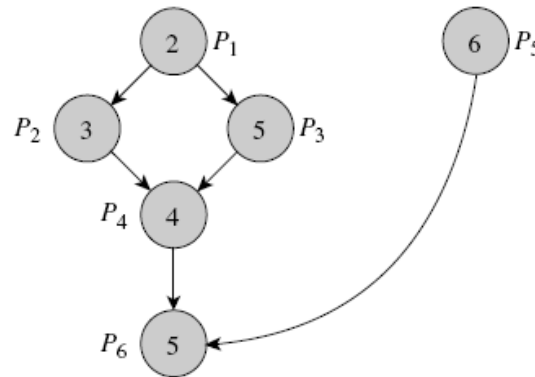
Scheduling con Deadline

- Possono essere specificati due tipi di deadline
 - **Deadline di inizio**: l'ultimo istante di tempo entro cui le operazioni del processo devono iniziare
 - **Deadline di completamento**: istante in cui le operazioni del processo devono terminare
- La stima della deadline è fatta considerando le precedenze dei processi e lavorando all'indietro dal requisito di risposta dell'applicazione

$$D_i = D_{application} - \sum_{k \in descendant(i)} x_k$$

dove $D_{application}$ è la scadenza dell'applicazione e x_k è il tempo di servizio del processo P_k

Esempio: determinare le deadline dei processi



- Il **totale dei tempi di servizio** dei processi è **25** secondi
- Se l'applicazione deve produrre una **risposta in 25 secondi**, le deadline dei processi sarebbero:

Process	P_1	P_2	P_3	P_4	P_5	P_6
Deadline	8	16	16	20	20	25

Earliest Deadline First (EDF)

- Determinare la deadline è più complesso in pratica
 - Deve considerare molti altri vincoli
 - Ad esempio, l'overlap delle operazioni di I/O con le elaborazioni della CPU
- Lo scheduling **Earliest Deadline First (EDF)** seleziona sempre il processo con la deadline più imminente
 - Sia **seq** la sequenza in cui i processi sono elaborati
 - Se **pos(P_i)** è la posizione di P_i nella sequenza delle decisioni di scheduling, lo sfioramento della deadline di P_i, ovvero D_i, non avviene se

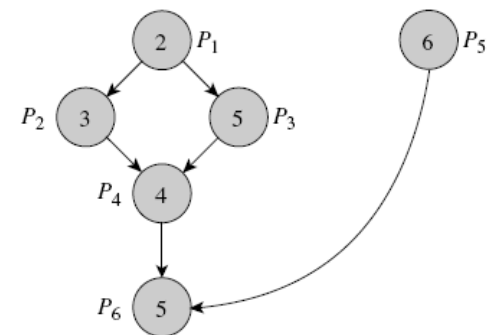
$$\sum_{k: pos(P_k) \leq pos(P_i)} x_k \leq D_i$$

- La condizione vale quando esiste una schedulazione ammissibile
- Vantaggi: semplicità e natura senza prelazione
- Buona politica per lo scheduling statico

Scheduling con Deadline (cont.)

Time	Process completed	Deadline overrun	Processes in system	Process scheduled
0	—	0	$P_1 : 8, P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_1
2	P_1	0	$P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_2
5	P_2	0	$P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_3
10	P_3	0	$P_4 : 20, P_5 : 20, P_6 : 25$	P_4
14	P_4	0	$P_5 : 20, P_6 : 25$	P_5
20	P_5	0	$P_6 : 25$	P_6
25	P_2 P_6	0	—	—

- P4: 20 indica che P4 ha la deadline a 20
- P2, P3 e P4, P5 hanno le stesse deadline
 - Sono possibili altre tre schedulazioni
 - Nessuna incorre nello sfioramento delle deadline

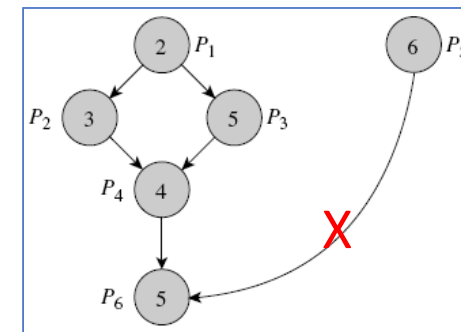


Esempio: problemi con lo scheduling EDF

- Se rimuoviamo l'arco (P5, P6) dal PPG
 - Due applicazioni indipendenti: P1-P4 e P6, e P5
 - Se tutti i processi devono essere completati in 19 secondi
 - Non esiste una soluzione ammissibile
 - Deadline dei processi

Process	P_1	P_2	P_3	P_4	P_5	P_6
Deadline	2	10	10	14	19	19

- Lo scheduling EDF può schedulare i processi come segue:
 - P1, P2, P3, P4, **P5, P6** oppure P1, P2, P3, P4, P6, **P5**
 - Quindi **il numero dei processi che non rispettano le rispettive deadline non è predicibile**



Ammissibilità schedulazione per Processi Periodici

- Frazione del tempo di CPU usato da $P_i = x_i / T_i$
- Nell'esempio, le frazioni di tempo di CPU usato somma a 0.93

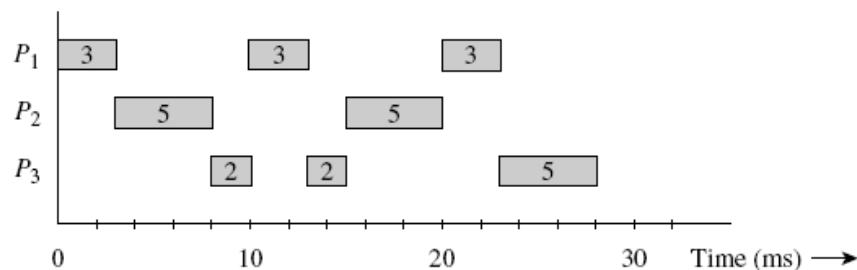
Process	P_1	P_2	P_3
Time period (ms)	10	15	30
Service time (ms)	3	5	9

- Se l'overhead è trascurabile, è ammissibile servire i tre processi
- In generale, l'insieme dei processi periodici P_1, \dots, P_n che non eseguono I/O può essere servito da un sistema hard RT che ha un overhead trascurabile se:

$$\sum_{l=1 \dots n} \frac{x_l}{T_l} \leq 1 \quad (7.4)$$

Scheduling Rate Monotonic (SRM)

- Determina i tassi a cui il processo deve ripetersi
 - Tasso di $P_i = 1 / T_i$ in secondi
- Assegna lo stesso rate come priorità del processo
 - Un processo con un periodo più piccolo ha maggiore priorità
- Impiega uno scheduling basato su priorità
 - Nell'esempio, P3 termina a 28ms



Process	P_1	P_2	P_3
Time period (ms)	10	15	30
Service time (ms)	3	5	9

Scheduling Rate Monotonic (SRM)

- SRM non garantisce uno scheduling ammissibile in tutte le situazioni
 - Per esempio, se P_3 avesse un periodo di 27 secondi
- Se l'applicazione ha un elevato numero di processi, potrebbe non essere in grado di raggiungere più del 69% di utilizzo di CPU se deve rispettare le deadline dei processi

$$\sum_{i=1}^m \frac{x_i}{T_i} \leq m \left(2^{\frac{1}{m}} - 1 \right) \rightarrow 0.69 \text{ per } m \rightarrow \infty$$

- L'algoritmo di *scheduling guidato da deadline* assegna dinamicamente le priorità ai processi sulla base delle loro deadline correnti
 - Può raggiungere il 100% di utilizzo di CPU
 - Le prestazioni possono essere più basse a causa dell'overhead dell'assegnamento dinamico delle priorità