Scheduling della CPU

Pietro Braione

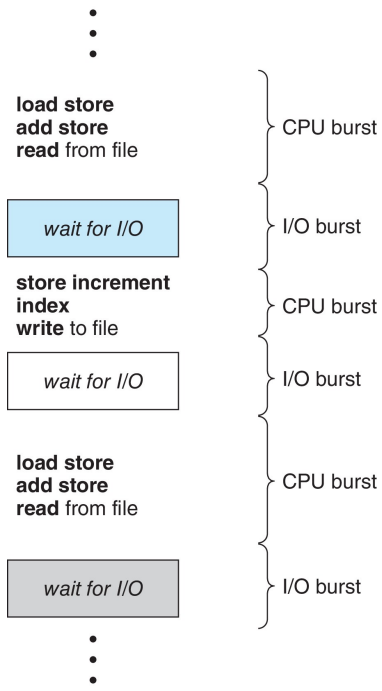
Reti e Sistemi Operativi – Anno accademico 2021-2022

Obiettivi

- Descrivere diversi algoritmi di scheduling della CPU
- Valutare gli algoritmi di scheduling sulla base dei criteri di scheduling
- Spiegare le problematiche relative allo scheduling in presenza di multithreading

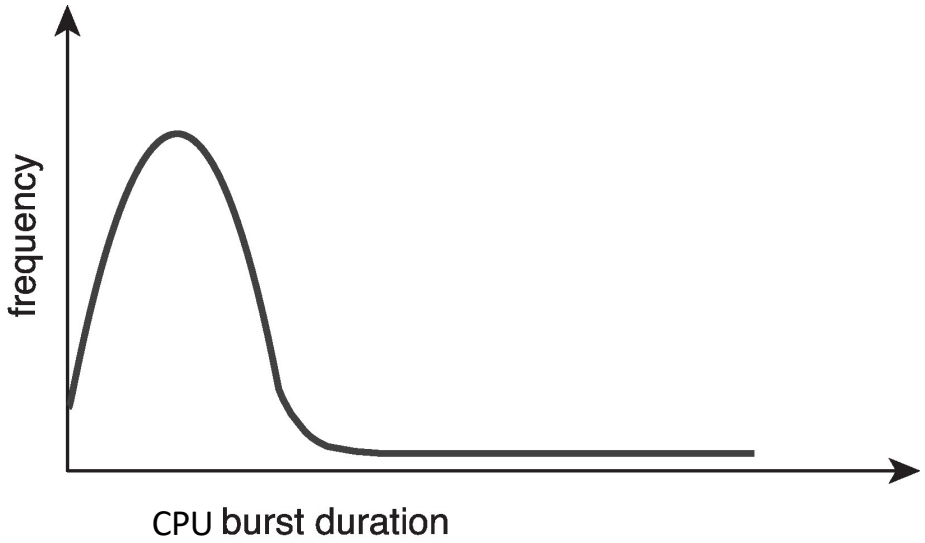
Concetti fondamentali

- L'obiettivo della multiprogrammazione è massimizzare l'utilizzo della CPU
- Gli algoritmi di scheduling sfruttano il fatto che di norma l'esecuzione di un processo è una sequenza di:
  - **Burst della CPU**: sequenza di operazioni di CPU
  - **Burst dell'I/O**: attesa completamento operazione di I/O



Distribuzione delle durate dei burst della CPU

- Programma con prevalenza di I/O (**I/O bound**):
  - Elevato numero di burst CPU brevi
  - Ridotto numero di burst CPU lunghi
  - Tipico dei programmi interattivi
- Programma con prevalenza di CPU (**CPU bound**):
  - Elevato numero di burst CPU lunghi
  - Ridotto numero di burst CPU brevi
  - Tipico dei programmi batch
- In entrambi i casi la curva della distribuzione ha la forma riportata accanto, ma:
  - I/O bound: il massimo sta più a sinistra
  - CPU bound: il massimo sta più a destra

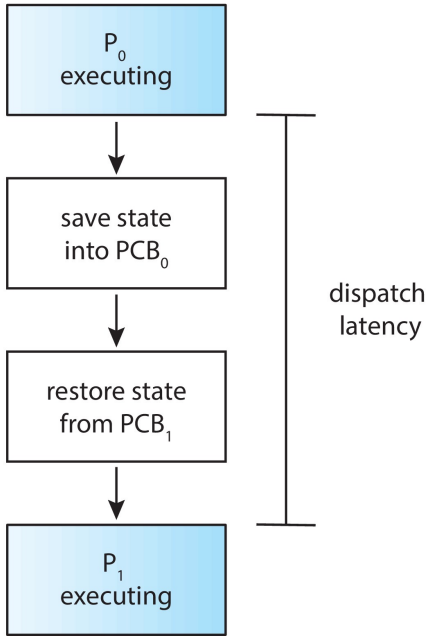


Scheduler della CPU

- Lo **scheduler della CPU**, o **scheduler a breve termine**, seleziona un processo tra quelli nella ready queue ed alloca un core libero ad esso
- Tali assegnamenti (decisioni di scheduling) possono essere effettuati in diversi momenti, corrispondenti a cambi di stato dei processi
    1. Quando un processo passa da stato running a stato waiting
    2. Quando un processo passa da stato running a stato ready
    3. Quando un processo passa da stato waiting a stato ready
    4. Quando un processo termina
- Se il riassegnamento viene fatto solo nelle situazioni 1 e 4 lo schema di scheduling è detto senza prelazione (**nonpreemptive**) o cooperativo, dal momento che un core è sempre liberato da un processo che volontariamente rinuncia ad esso
- Altrimenti è detto con prelazione (**preemptive**), dal momento che un core può essere anche liberato perché un core viene forzatamente sottratto dal kernel ad un processo che lo sta usando
- Lo schema di scheduling preemptive è più complicato da implementare:
    - Che succede se due processi condividono dati?
    - Che succede se un processo sta eseguendo in modalità kernel (system call o IRQ)?

Dispatcher

- Il **dispatcher** passa effettivamente il controllo della CPU al processo scelto dallo scheduler a breve termine:
  - Effettua il cambio di contesto
  - Passa in modalità utente
  - Salta nel punto corretto del programma del processo selezionato (ossia, dove era stato precedentemente interrotto)
- La **latenza di dispatch** è il tempo impiegato dal dispatcher per fermare un processo ed avviarne un altro
- Come già detto, lo scheduler implementa una *politica*, il dispatcher implementa un *meccanismo*



Criteri di scheduling

- Misure che servono per confrontare le caratteristiche dei diversi algoritmi
- (purtroppo non dipendono solo dall'algoritmo, ma anche dal carico)
- Principali criteri:
  - **Utilizzo della CPU**: % di tempo in cui la CPU è attiva nell'esecuzione dei processi utente (dovrebbe essere tra il 40% e il 90%, in funzione del carico)
  - **Throughput**: # di processi che completano l'esecuzione nell'unità di tempo (dipende dalla durata dei processi)
  - **Tempo di completamento**: tempo necessario per completare l'esecuzione di un certo processo (dipende da molti fattori: durata del processo, carico totale, durata dell'I/O…)
  - **Tempo di attesa**: tempo trascorso dal processo nella ready queue (meglio del tempo di completamento, meno dipendente da durata del processo e dell'I/O)
  - **Tempo di risposta**: negli ambienti interattivi, tempo trascorso tra l'arrivo di una richiesta al processo e la produzione della prima risposta, senza l'emissione di questa nell'output
- A noi interessano essenzialmente tempo di completamento e di attesa, e su quelli svolgeremo i nostri esercizi

Criteri di scheduling: ottimizzazione

- Massimo utilizzo della CPU
- Massimo throughput
- Minimo tempo di completamento (medio)
- Minimo tempo di attesa (medio)
- Minimo tempo di risposta (medio)

- Naturalmente nessun algoritmo di scheduling può ottimizzare tutti i criteri contemporaneamente…

Scheduling in ordine di arrivo

- **Scheduling in ordine di arrivo**, o **first-come-first-served** (FCFS): la CPU viene assegnata al primo processo che la richiede
- Vantaggio: Implementazione molto semplice (coda FIFO, nessuna prelazione)
- Svantaggio: Tempo di attesa medio può essere lungo («effetto convoglio»)

Ordine di arrivo

Processo	Durata burst CPU	Tempo attesa
P_1	24	0
P_2	3	24
P_3	3	27

Quando iniziano ad eseguire P1, P2, P3

Tempo di attesa medio = (0 + 24 + 27) / 3 = 17

Tempo di completamento medio = (24 + 27 + 30) / 3 = 27

Quando terminano di eseguire P1, P2, P3

Scheduling in ordine di arrivo

- **Scheduling in ordine di arrivo**, o **first-come-first-served** (FCFS): la CPU viene assegnata al primo processo che la richiede
- Vantaggio: Implementazione molto semplice (coda FIFO, nessuna prelazione)
- Svantaggio: Tempo di attesa medio può essere lungo («effetto convoglio»)

Ordine di arrivo

Processo	Durata burst CPU	Tempo attesa
P_2	3	0
P_3	3	3
P_1	24	6

Tempo di attesa medio = (0 + 3 + 6) / 3 = 3 **(ridotto a circa 1/6)**
Tempo di completamento medio = (3 + 6 + 30) / 3 = 13 **(ridotto a circa 1/2)**

Scheduling per brevità

- **Scheduling per brevità**, o **shortest-job-first** (SJF): la CPU viene assegnata al processo che ha il successivo CPU burst più breve
  - Vantaggi: implementazione quasi identica a FCFS, ma minimizza il tempo di attesa medio (è **ottimale**)
  - Svantaggio: di solito non si sa in anticipo qual è il processo che avrà il CPU burst più breve (quanto durerà il prossimo CPU burst?)
- L'algoritmo **shortest-remaining-time-first** (SRTF) utilizza la prelazione per gestire il caso in cui i processi non arrivino tutti nello stesso istante: se nella ready queue arriva un processo con un burst più corto di quello running, quest'ultimo viene prelazionato dal nuovo processo

Scheduling per brevità: esempio

Ordine di arrivo

Processo	Durata burst CPU	Tempo attesa
P_1	6	0
P_2	8	6
P_3	7	14
P_4	3	21

Tempo di attesa medio FCFS = (0 + 6 + 14 + 21) / 4 = 10,25
Tempo di completamento medio FCFS = (6 + 14 + 21 + 24) / 4 = 16,25

Processo	Durata burst CPU	Tempo attesa
P_4	3	0
P_1	6	3
P_3	7	9
P_2	8	16

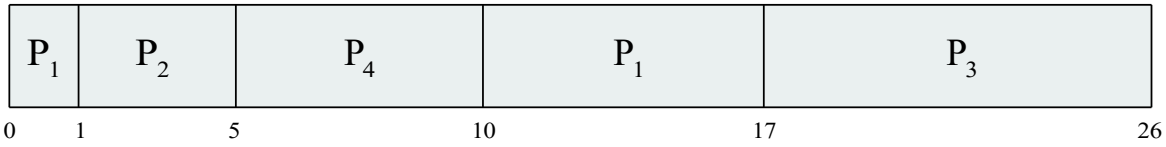
Tempo di attesa medio SJF = (0 + 3 + 9 + 16) / 4 = 7
Tempo di completamento medio SJF = (3 + 9 + 16 + 24) / 4 = 13

Shortest-remaining-time-first: esempio

- Con preemption e tempo di arrivo
- Il tempo di attesa di un processo è:
  istante terminazione processo - (tempo di arrivo + durata burst)
- Il tempo di completamento di un processo invece è:
  istante terminazione processo - tempo di arrivo

Processo	Tempo di arrivo	Durata burst CPU
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Determina l'ordine di arrivo



Tempo di attesa medio = ((17-8) + (5-5) + (26-11) +(10-8)) / 4 = 6,5

Tempo di completamento medio = ((17-0) + (5-1) + (26-2) +(10-3)) / 4 = 13

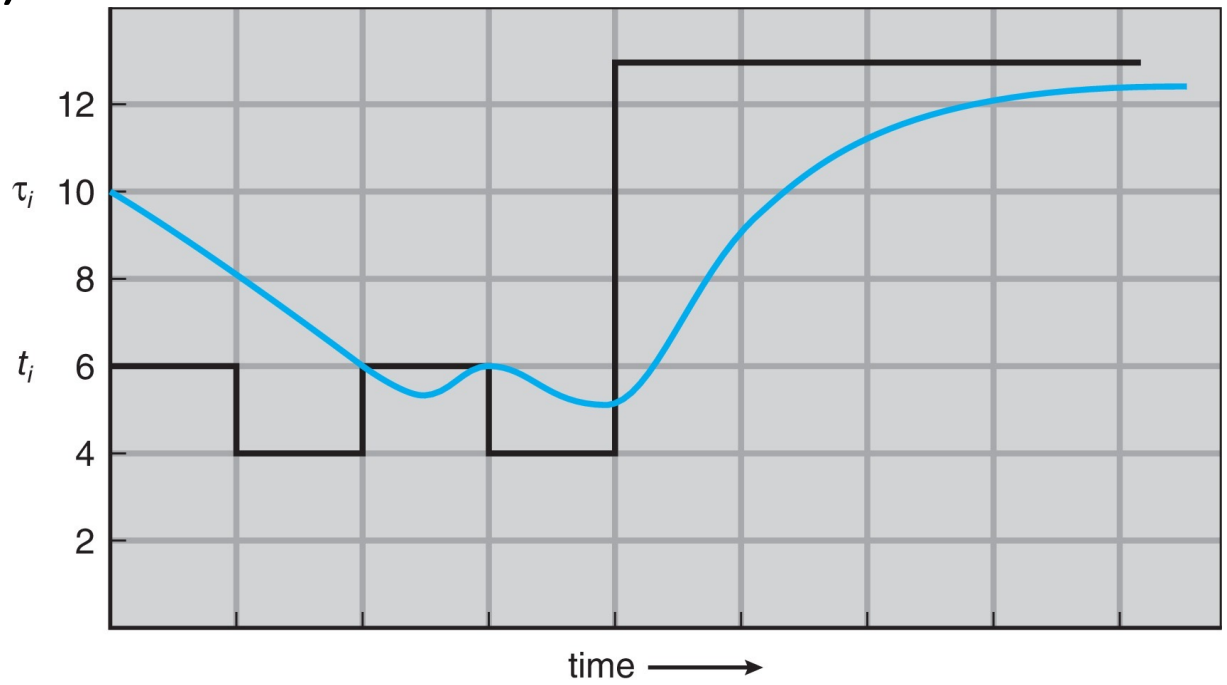
Stima lunghezza prossimo CPU burst

- Idea: Registrare la lunghezza dei CPU burst precedenti ed applicare una **media esponenziale**:
  - t_n = durata effettiva dell' n-esimo burst
  - Sia α un valore compreso tra 0 e 1
  - Sia $\tau_{k+1} = \alpha\, t_k + (1 - \alpha)\, \tau_k$
  - La media esponenziale è data da τ_{n+1}
- Espandendo la formula otteniamo:
  - $\tau_{n+1} = \alpha\, t_n + \alpha\, (1 - \alpha)\, t_{n-1} + \ldots + \alpha\, (1 - \alpha)^k\, t_{n-k} + \ldots + (1 - \alpha)^{n+1}\, \tau_0$
  - Sia α che $1 - \alpha$ sono compresi tra 0 e 1, quindi ogni termine pesa meno del precedente

La media esponenziale: parametro α

- Il parametro α «bilancia» il peso della storia recente vs. storia passata (di solito si usa α = 0,5)
- α = 0:
  - $\tau_{n+1} = \tau_n$
  - La storia recente non conta
- α = 1:
  - $\tau_{n+1} = t_n$
  - Solo la durata dell'ultimo CPU burst conta

Stima lunghezza prossimo CPU burst: esempio ($\alpha = 0{,}5$)



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Scheduling circolare

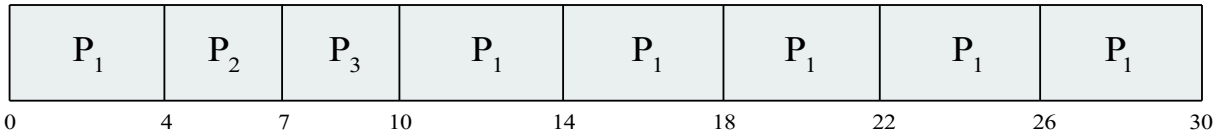
- Nello **scheduling circolare**, o **round-robin** (RR) ogni processo ottiene una piccola quantità fissata di tempo di CPU (**quanto di tempo**), di solito 10-100 millisecondi, per il quale può essere in esecuzione
- Trascorso tale tempo il processo in esecuzione viene interrotto e messo in fondo alla ready queue, che è gestita in maniera FIFO
- In tal modo la ready queue funziona essenzialmente come un buffer circolare, e i processi vengono scanditi dal primo all'ultimo, per poi ripartire dal primo nello stesso ordine
- Timer che genera un interrupt periodico con periodo q per effettuare la prelazione del processo corrente (passaggio del processo da stato running a ready)

Scheduling circolare: esempio

- Ricordiamo che il tempo di attesa di un processo è:
  istante terminazione processo - (tempo di arrivo + durata burst)
- E il tempo di completamento di un processo è:
  istante terminazione processo - tempo di arrivo
- In questo esempio il tempo di arrivo è 0 per tutti i processi

Ordine di arrivo

Processo	Durata burst CPU
P_1	24
P_2	3
P_3	3



q = 4

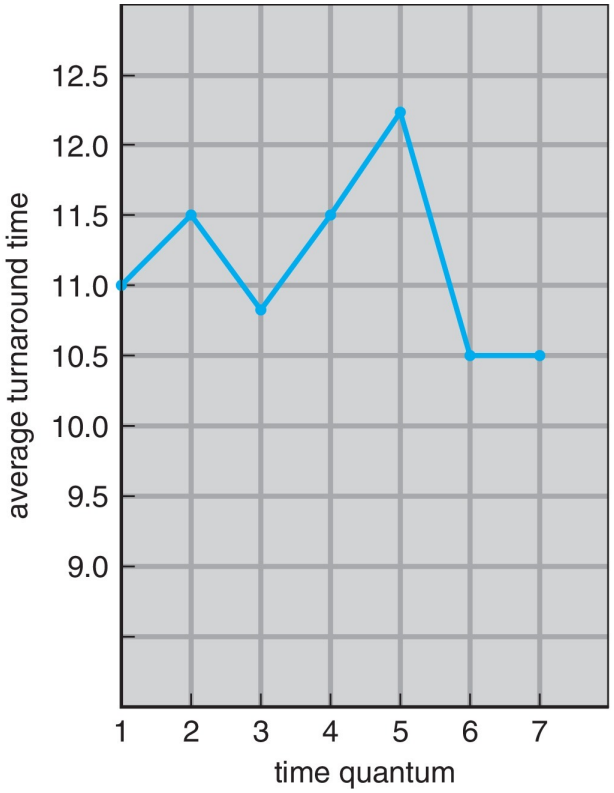
Tempo di attesa medio = ((30-24) + (7-3) + (10-3)) / 3 = 5,67

Tempo di completamento medio = (30 + 7 + 10) / 3 = 15,67

Scheduling circolare: caratteristiche

- Se ci sono n processi nella ready queue e il quanto temporale è q:
  - Ogni processo ottiene 1/n del tempo totale di CPU, in maniera perfettamente equa
  - Nessun processo attende più di q (n - 1) unità di tempo nella ready queue prima di ridiventare running per un altro quanto di tempo
- Comportamento in funzione di q:
  - q elevato: RR tende al FCFS
  - q basso: deve comunque essere molto più lungo della latenza di dispatch, altrimenti questa si «mangia» un tempo comparabile al tempo di esecuzione dei processi utente e l'utilizzo della CPU diventa inaccettabilmente basso
- Performance:
  - Rispetto a SJF tipicamente RR ha un tempo di completamento medio più alto
  - Ma un tempo di risposta medio più basso (va bene per i processi interattivi)
  - Il tempo di completamento medio non necessariamente migliora con l'aumento di q

Tempo di completamento scheduling circolare



process	time
P_1	6
P_2	3
P_3	1
P_4	7

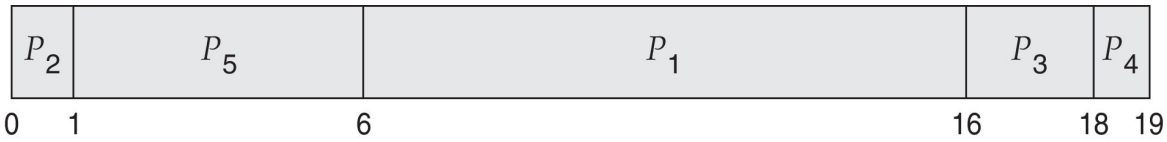
Il tempo di completamento medio migliora se la maggioranza (~80%) dei CPU burst è più breve di q

Scheduling con priorità

- Nello scheduling con priorità ad ogni processo è associato un numero intero che indica la sua priorità
- Viene eseguito il processo con priorità più alta, gli altri aspettano (in UNIX-like numero più basso = priorità più alta, in Windows il contrario)
- Può essere preemptive o no
- Possono essere permessi più processi con pari priorità o no; in caso positivo occorre stabilire un secondo algoritmo di scheduling per arbitrare tra i processi a pari priorità (di solito si utilizza RR)
- SJF è scheduling con priorità, dove la priorità è l'inverso della durata (stimata) del prossimo CPU burst
- Problema della **attesa indefinita** (**starvation**): un processo a priorità troppo bassa potrebbe non venir mai schedulato
- Soluzione: **invecchiamento** (**aging**), ossia aumento automatico di priorità di un processo al crescere del tempo di permanenza nella ready queue

Scheduling con priorità: esempio

Processo	Durata burst CPU	Priorità (UNIX)
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

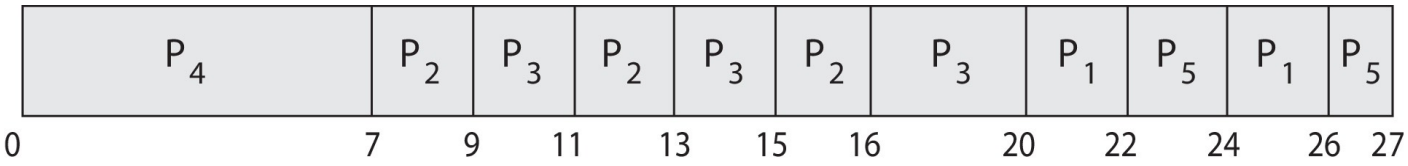


Tempo di attesa medio = (0 + 1 + 6 + 16 + 18) / 5 = 8,2
Tempo di completamento medio = (1 + 6 + 16 + 18 + 19) / 5 = 12

Scheduling con priorità + RR: esempio

Processo	Durata burst CPU	Priorità (UNIX)
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Ordine di arrivo



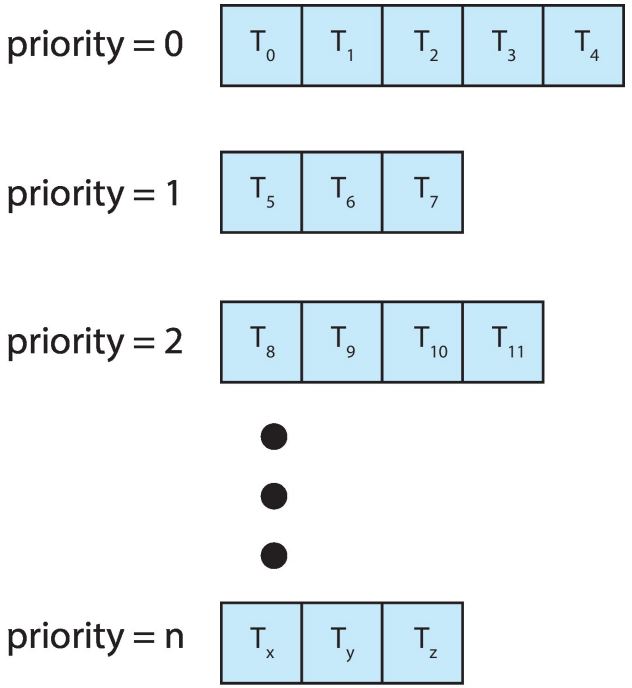
q = 2 per processi con stessa priorità

Tempo di attesa medio = ((26 - 4) + (16 - 5) + (20 - 8) + (7 - 7) + (27 - 3)) / 5 = 13,8

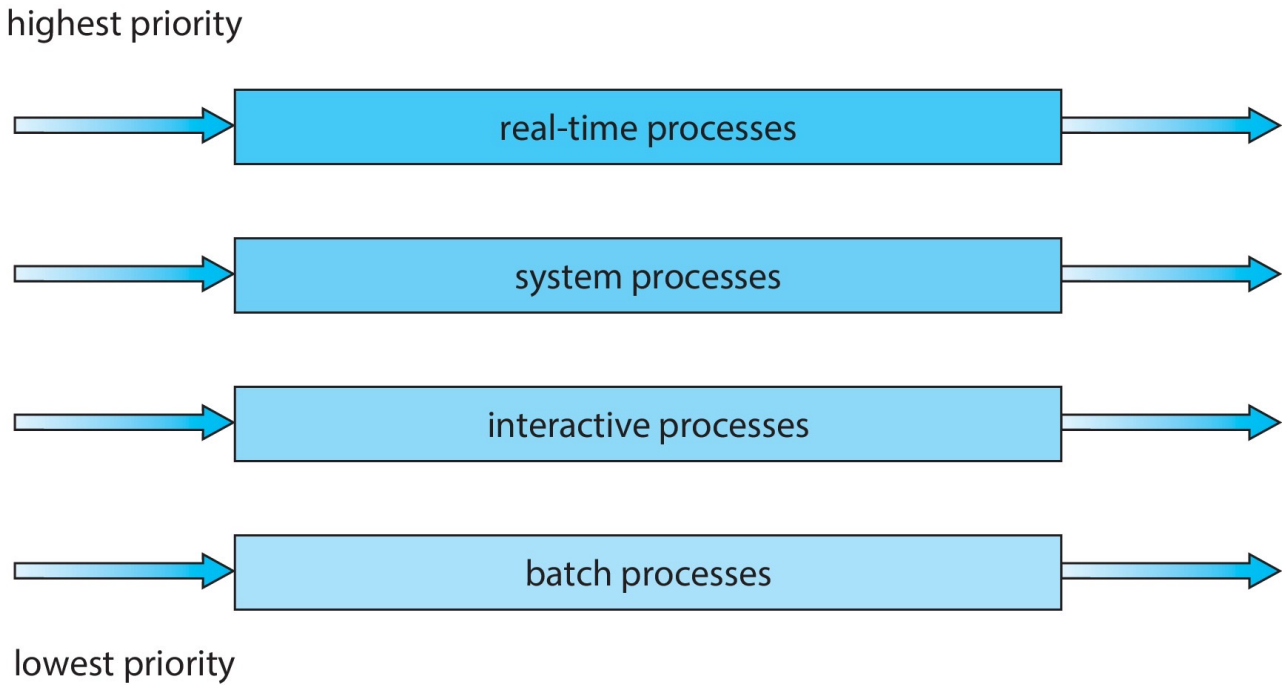
Tempo di completamento medio = (26 + 16 + 20 + 7 + 27) / 5 = 19,2

Code multilivello

- Lo scheduling con priorità usa code separate per ogni priorità
- Viene schedulato il processo nella coda non vuota con priorità maggiore
- La priorità può essere basata sul tipo di processo:
  - Priorità più alta ai processi interattivi o cyber-fisici che devono reagire rapidamente all'I/O (tipicamente I/O bound)
  - Priorità più bassa ai processi che effettuano lunghe computazioni, o processi batch (tipicamente CPU bound)



Code multilivello: esempio priorità basata sul tipo di processo

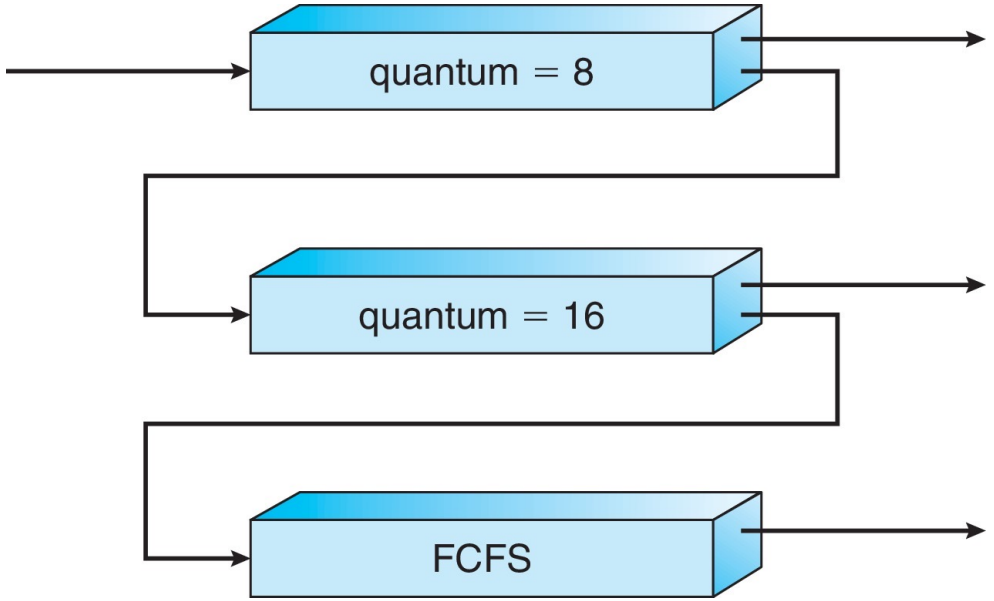


Code multilivello con retroazione

- Un processo può essere spostato tra le diverse code di priorità facendo in tal modo variare dinamicamente la sua priorità
  - L'invecchiamento è di solito implementato in questo modo
  - Ma anche la identificazione di un processo come I/O bound o CPU bound può essere effettuata dinamicamente, e determinare il cambio di priorità del processo, e quindi il suo spostamento di coda
- Uno scheduler di questo tipo è detto con code multilivello con retroazione

Code multilivello con retroazione: esempio

- Code:
  - Q_0: RR con q_0 = 8 msec
  - Q_1: RR con q_1 = 16 msec
  - Q_2: FCFS (equivalente a RR con $q_2 = \infty$)
- Q_0 ha priorità alta, Q_1 intermedia, Q_2 bassa
- Scheduling:
  - Alla creazione un processo entra in Q_0
  - Se quando un processo in Q_0 (Q_1) diventa running non termina il burst entro q_0 (q_1), avviene preemption e viene messo all'inizio di Q_1 (Q_2)
  - Per evitare starvation c'è un processo di invecchiamento che sposta i processi in direzione opposta, da Q_0 a Q_1 e da Q_1 a Q_2
- Effetto ricercato: mantenere i processi I/O bound (con burst della CPU corti) nelle code ad alta priorità e quelli CPU bound nelle code a bassa priorità



Scheduling dei thread

- Se il kernel supporta il multithreading, le entità schedulate dallo scheduler del kernel non sono più i processi, ma i thread kernel
- Per i thread utente la questione è più complicata, dal momento che, a seconda del modello di threading, possono intervenire due scheduler: lo scheduler nella libreria dei thread a livello utente, e lo scheduler del kernel
- A tale scopo si parla di **ambito della contesa di un thread utente** per indicare come il thread utente contende un LWP (ossia, un thread kernel) con altri thread utente dello stesso processo:
  - Nei modelli molti-a-uno e molti-a-molti un thread utente condivide i LWP del processo con altri thread utente dello stesso processo: si dice che il thread utente ha un **ambito della contesa ristretto al processo** (**process-contention scope**, PCS)
  - Nel modello uno-a-uno (e per i thread opportunamente configurati nel modello a due livelli), invece, un thread utente ha un suo LWP non condiviso con altri thread utente: si dice in tal caso che il thread utente ha un **ambito della contesa allargato al sistema** (**system-contention scope**, SCS)
- Notare la differenza:
  - Nel caso di PCS, la contesa/condivisione dei LWP di un processo tra i thread utente del processo è regolata dallo scheduler integrato nella libreria dei thread a livello utente, mentre lo scheduler del kernel si occupa della contesa/condivisione dei processori tra tutti i thread kernel del sistema
  - Nel caso di SCS, conta solo la contesa/condivisione dei processori tra i thread kernel, il LWP del thread utente non è conteso con altri thread e la libreria dei thread a livello utente non deve effettuare scheduling per quel thread utente