Lezione 8

Concetti fondamentali

L’obiettivo della multiprogrammazione è: avere sempre un processo in esecuzione in modo da massimizzare l’utilizzazione della CPU (solitamente un processo è in esecuzione finché non deve attendere un evento (di solito il completamento di una richiesta di I/O). In un sistema di calcolo semplice, la CPU resterebbe inattiva e tutto il tempo di attesa verrebbe sprecato.

Con la multiprogrammazione invece si tengono più processi in memoria contemporaneamente e quando un processo deve attendere un evento, il SO gli sottrae il controllo della CPU

L’obiettivo del time-sharing è:

commutare l’uso della CPU tra i vari processi così frequentemente, che gli utenti possano interagire con ciascun programma in esecuzione

Lo scheduling è una funzione fondamentale del SO; si sottopongono a scheduling tutte le risorse del calcolatore. Lo scheduling della CPU è alla base della progettazione dei SO

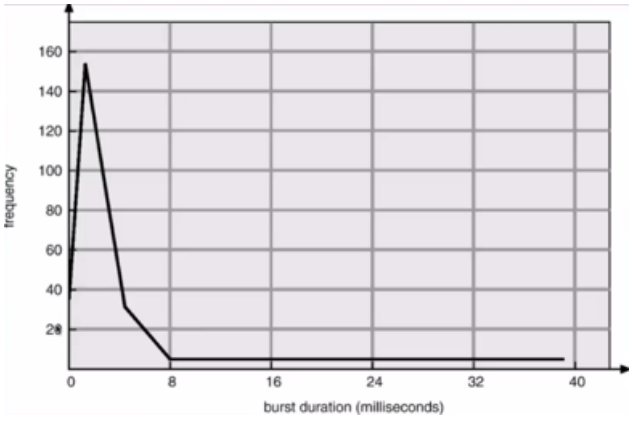
L’elaborazione di un processo è costituita da:

- un **ciclo** di esecuzione di CPU
- uno di attesa di I/O

I processi si alternano tra questi due stati

L’esecuzione di un processo inizia con una serie di operazioni di elaborazione della CPU (**CPU burst)** seguita da una serie di operazioni di I/O (**I/O burst**) e così a susseguirsi. L’ultima serie di operazioni della CPUI si conclude con una richiesta al sistema di terminare l’esecuzione

La durata delle sequenze di operazioni della CPU sono state sperimentalmente misurate e la loro curva di frequenza è in genere simile a questa:



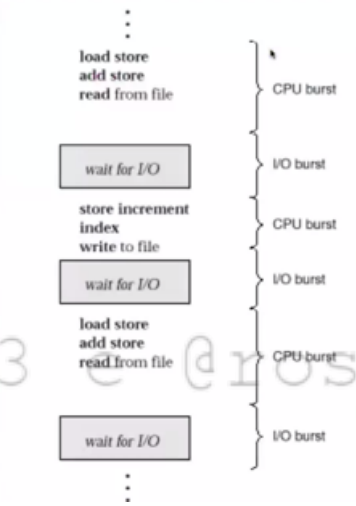
↑ Normalmente abbiamo un numero di istanti di uso della CPU che mediamente è compreso tra 0 e 8 ↑

La curva in genere è di tipo esponenziale con molte brevi sequenze di operazioni della CPU e poche sequenze dei operazioni della CPU molto lunghe

Un programma con prevalenza I/O (**I/O bound**) produce sequenze della CPU di breve durata
Un programma con prevalenza d'elaborazione (**CPU bound**) può produrre varie sequenza di operazioni della CPU molto lunghe

Queste caratteristiche sono spesso utili per la scelta di un appropriato algoritmo di schedulazione della CPU

Serie alternata di sequenze di operazioni della CPU e di sequenze di operazioni di I/O:



Scheduler della CPU

Ogni volta che la CPU passa nello stato di inattività, il SO sceglie uno dei processi dalla ready queue per l'esecuzione

Lo scheduler a breve termine (scheduler della CPU) sceglie il processo a cui assegnare la CPU tra quelli in memoria pronti per l'esecuzione

Generalmente gli elementi delle cose sono i PCB dei processi

Le decisioni riguardanti lo scheduling della CPU vengono prese nelle seguenti circostanze:

1. Quando il processo passa da uno stato running allo stato waiting:
    - In caso di richiesta di I/O, oppure di attesa per la terminazione di uno dei processi figli
2. Quando un processo passa da uno stato di running ad uno stato ready:
    - in caso di interrupt
3. Quando un processo passa da uno stato waiting ad uno stato ready:
    - in caso di completamento di un I/O

4. Quando un processo termina

Schema di scheduling:

- **senza diritto di prelazione *(non preemptive)* o cooperativo** → Quando lo scheduling interviene solo nelle condizioni 1 e 4. Uno scheduling è non preemptive se ogni processo a cui viene assegnata la CPU rimane in possesso della CPU fino a quando o termina la sua esecuzione oppure passa in uno stato di waiting
- **con diritto di prelazione *(preemptive)*** → Quando lo scheduling interviene solo nelle condizioni 2 e 3. Uno scheduling è preemptive quando non è non preemptive :P

Lo scheduling con prelazione, a differenza di quello senza prelazione, può portare a race condition quando i dati sono condivisi tra diversi processi

Dispatcher

Un altro elemento coinvolto nella funzione di scheduling della CPU è il **dispatcher**

Il dispatcher è il modulo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine

Questa funzione comprende:

- cambio di contesto
- il passaggio alla modalità utente
- il salto alla giusta posizione del programma utente per riavviarne l'esecuzione

Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è noto come **latenza di dispatch**

Criteri di scheduling

Diversi algoritmi per lo scheduling della CPU hanno proprietà differenti e possono favorire una particolare classe di processi

Per il confronto, esistono varie caratteristiche da tenere in considerazione:

- **Utilizzo della CPU:** La CPU deve essere più attiva possibile. L'utilizzo della CPU può variare dallo 0 al 100 % ma in media in un sistema reale dovrebbe variare dal 40%, per un sistema con poco carico, al 90%, per un sistema con utilizzo intenso
- **Produttività**: Una misura del lavoro svolto è il *throughput o produttività*

  Throughput (produttività) → è dato dal numero di processi completati nell'unità di tempo
- **Tempo di completamento (*turnaround time)***: Tempo necessario per eseguire il processo stesso.

  E' definito come l'intervallo di tempo che intercorre tra la sottomissione del processo ed il completamento dell'esecuzione

E’ la somma dei tempi passati in attesa dell’ingresso nella memoria nella coda dei processi pronti, durante l’esecuzione della CPU e nel compiere operazioni di I/O

- **Tempo di attesa**: L’algoritmo di scheduling non incide sul tempo necessario per l’esecuzione di un processo, solo su quanto tempo questo processo deve restare nella ready queue.

  E’ la somma degli intervalli d’attesa passati in questa cosa

- **Tempo di risposta:** E’ il tempo che intercorre tra la sottomissione di una richiesta e la prima risosta prodotta. E’ data dal tempo necessario per iniziare la risposta ma non dal suo tempo necessario per completare l’output.

Criteri di ottimizzazione

- Utilizzo massimo della CPU
- Produttività massima
- Minimo tempo di completamento
- Minimo tempo di attesa
- Minimo tempo di risposta

Algoritmi di scheduling

Scheduling First-Come, First-Served (FCFS)

L’algoritmo di **scheduling in ordine d’arrivo** è il più semplice tra gli algoritmi di scheduling della CPU. Con questo schema, la CPU si assegna al processo che la rich8iede per primo

Come è realizzato? La realizzazione si basa su una coda FIFO (First-In-First-Out): Qaudno un processo entra nella ready queue, si collega il suo PCB all’ultimo elemento della coda

Quando la CPU è libera, viene assegnata al processo che si trova in testa alla coda, rimuovendolo da essa

Un aspetto negativo di questo algoritmo è che il tenmpo medio solitamente è abbastanza lungo

Esempio:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se i processi arrivano nell’ordine P_1, P_2, P_3 e sono serviti in ordine FCFS, si ottiene il risultato riportato in questo **diagramma di Gantt**

Tempi di attesa:

$P_1 = 0\ millisecondi$
$P_2 = 24\ millisecondi$
$P_3 = 27\ millisecodi$

Quindi, tempo di attesa medio:

$\frac{(0+24+27)}{3} = 17\ millisecondi$

Se i processi arrivassero nell'ordine P_2, P_3, P_1 i risultati sarebbero i seguenti

Tempo di attesa:

$P_1 = 6\ millisecondi$
$P_2 = 0\ millisecondi$
$P_3 = 3\ millisecodi$

Tempo di attesa medio:

$\frac{(6+0+3)}{3} = 3\ millisecondi$

Si tratta di una notevole riduzione

In FCFS il tempo medio di attesa può variare notevolmente al variare della durata dei CPU burst dei processi e del loro ordine di arrivo

Esempio: Si suppongano di avere molti processi con prevalenza di I/O e un processo con prevalenza di elaborazione.

Il processo con prevalenza di elaborazione occupa la CPU. durante questo periodo, tutti i processi con prevalenza I/O terminano le proprie operazioni e si spostano nella ready queue, in attesa della CPU

In questi casi si ha quello che si chiama
Effetto convoglio: tutti i processi attendono che un lungo processo liberi la CPU, il che causa una riduzione dell'utilizzo della CPU e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi.

L'algoritmo di scheduling FCFS è **non preemptive**

Questo algoritmo crea problemi nei sistemi time sharing, dove ogni utente deve disporre della CPU a intervalli regolari

Scheduling shortest-job-first

Un criterio diversi di scheduling della CPU eè l'algoritmo di **scheduling per brevità** (**SJF**)

Questo algoritmo associa a ogni processo la lunghezza del suo CPU burst successivo

quando la CPU è disponibile, viene assegnata al processo che ha il CPU burst successivo più breve

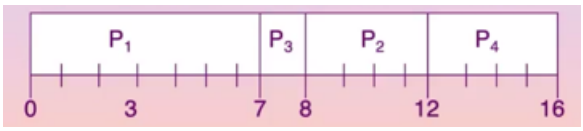
Se due processi hanno i CPU busrt successivi della stessa lunghezza, si applica l'algoritmo FCFS

Due schemi:

- **Non preemptive**: quando la CPU è allocata al processo non viene deallocata fino al completamento del burst di a CPU
- **Preemptive**: se arriva un nuovo processo con burst di CPU più corto del tempo di CPU rimanente al processo correntemente in esecuzione la CPU viene subito deallocata e allocata al nuovo processo
  - Detto anche Shortest Remaining Time First: SRTF

Esempio di SJF Non-Preemptive

Processo	Tempo di arrivo	Lunghezza del burst
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Il tempo di attesa è la somma degli intervalli di attesa passati nella coda dei processi pronti

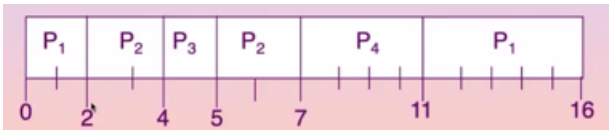
Tempo di attesa medio:

$\frac{(0+(8-2)+(7-4+(12-5))}{4} = 4\ millisecondi$

Il primo nella sottrazione è **quando il processo prende il controllo della CPU,** mentre il secondo è **quando arriva nella coda dei processi pronti**

Esempio di SJF Preemptive

Processo	Tempo di arrivo	Lunghezza del burst
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Tempo di attesa medio:

$\frac{((11-2)+(5-4)+0+(7-5))}{4} = 3\ millisecondi$

SJF è ottimale, nel senso che rende il tempo medio di attesa minimo per un dato insieme di processi.
La difficoltà di questo tipo di scheduling è di conoscere la durata della successiva richiesta di CPU.
Proprio per questa motivazione, l'SJF non può essere realizzato a livello dello scheduling della CPU a breve termine (poiché non esiste modo per conoscere la successiva sequenza di operazioni della CPU)

Se non è possibile conoscere la lunghezza della prossima sequenza di operazioni ella CPI, si può cercare di predire il suo valore: la lunghezza della sequenza successiva si stima calcolando la **media esponenziale** delle lunghezze misurate delle precedenti sequenze

Essa si definisce con la formula seguente

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

dove:

τ_{n+1} → il valore previsto per la successiva sequenze

t_n → lunghezza dell'n-esima sequenza di operazioni della CPU. Contiene le informazioni più recenti

τ_n → registra la storia passata

α → compreso tra 0 e 1: $0 \leq \alpha \leq 1$. Controlla il peso relativo sulla predizione della storia recente e di quella passata

Se:

- $\alpha = 0$: Allora $\tau_{n+1} = \tau_n$ e la storia recente non ha effetto
- $\alpha = 1$: Allora $\tau_{n+1} = t_n$ e ha significato solo la più recente sequenza di operazioni della CPU
- $\alpha = \frac{1}{2}$: Allora la storia recente e la storia passata hanno lo stesso peso. E' la condizione più comune

Per comprendere il comportamento della media esponenziale, si può sviluppare la formula sostituendo τ_n in modo da ottenere:

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

dove τ_0 si può definire come una costante o come una media complessiva del sistema

Giacché entrambi α e $(1-\alpha)$ sono minori o uguali di 1, ciascun termina successivo ha peso minore del suo predecessore

Scheduling con priorità

SJF è un caso particolare del più generale **algoritmo di scheduling con priorità**: si associa una priorità ad ogni processo e si assegna la CPU al processo con priorità più alta (i processi con priorità uguali si ordinano secondo uno schema FCFS)

SJF è uno scheduling a priorità dove la priorità è il valore predetto del prossimo burst di CPU

Un valore di priorità (intero) viene associato ad ogni processo

La priorità p è l'inverso della lunghezza della successiva sequenza di operazioni della CPU (cioè ha priorità più alta chi ha la lunghezza minore)

Per linux abbiamo che un valore di p più basso indica una priorità più alta

Le priorità possono essere definite sia:

- *internamente* → usano una o più quantità misurabili (ad esempio i limiti di tempo, i requisiti di memoria, il numero di file aperti e il rapporto tra la lunghezza media delle sequenze di operazioni I/O e la lunghezza media delle sequenze di operazioni della CPU)
- *esternamente* → si definiscono utilizzando criteri esterni al SO (come l'importanza di un processo, la quantità di soldi utilizzata per costruire il calcolatore etc)

La CPU viene allocata al processo con priorità più alta ed è sia preemptive che non preemptive

Un problema fondamentale di questo tipo di scheduling è quello del blocco indefinito o della **starvation**: questo algoritmo può lasciare processi a bassa priorità nell'attesa indefinita della CPU (processi a bassa priorità potrebbero non essere mai eseguiti)

Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'**invecchiamento** (**aging**)**:** con il passare del tempo si incrementa di priorità il processo

Scheduling circolare

L'**algoritmo di scheduling circolare** (**round-robin, RR**) è stato progettato appositamente per i sistemi time-sharing:

Ciascun processo riceve una piccola quantità fissata del tempo della CPU chiamata **quanto di tempo o porzione di tempo** (**time slice**) in genere 10-100 millisecondi

Lo scheduler della CPI scorre la ready queue, assegnando la CPU a ciascun processo per un intervallo di tempo della durata di un quanto di tempo. allo scadere di questo tempo il processo viene sospeso e aggiunto alla fine della coda dei processi pronti

Per implementare lo scheduling RR si gestisce la ready queue come una coda FIFO

A questo punto possiamo avere due situazioni:

- Il processo dura meno di un quanto di tempo: il processo rilascia volontariamente la CPU e lo scheduler passa al processo successivo
- Il processo dura più di un quanto di tempo: il timer scade, e invia un interrupt al SO, che esegue il cambio di contesto e mette il processo alla fine della ready queue. Lo scheduler seleziona poi il prossimo processo. L'algoritmo è quindi con diritto di prelazione

Se ci sono *n* processi nella coa dei processi pronti e il qanto di tempo è *q* allora ciascun processo riceve *1/n-esimo* del tempo di CPU in blocchi di al massimo *q* unità di tempo

Nessun processo rimane sospeso per più di *(n - 1) * q* unità di tempo

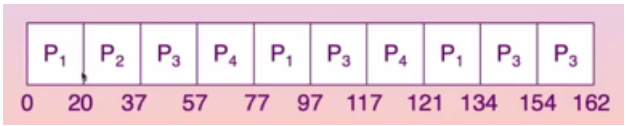
La valutazione delle prestazioni dipendono dalla grandezza del quanto di tempo:

- Valore di q **grande** → Lo scheduling RR si riduce ad essere lo scheduling FCFS
- Valore di q **piccolo** → q deve essere sufficientemente grande rispetto al tempo necessario per il cambio di contesto, altrimenti l'overhead è troppo alto

Esempio di RR con quanto di tempo = 20:

Processo	Lunghezza dei burst
P_1	53
P_2	17
P_3	68
P_4	24

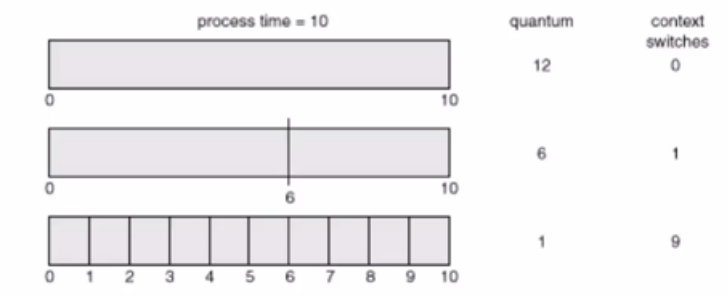
Il diagramma di Gantt per lo scheduling è:



P1-P2-P3-P4 - ricomincia - *P1-P3-P4* - P2 è terminato - *P1-P3* - P4 è terminato - *P3* - P1 è terminato

Generalmente ha tempo di completamento medio più alto di SJF ma tempo di risposta migliore

Un quanto di tempo minore incrementa il numero di cambi di contesto:



Scheduling a code multilivello

Una distinzione comune che viene fatta tra i processi è:

- Processi in **foreground** (*primo piano*) o **interattivi**
- Processi in **background** (*sottofondo*) o **batch** (a lotti)

Questi due tipi di processi hanno differenti requisiti sui tempo di risposta e possono avere diverse necessità di scheduling

Un **algoritmo di scheduling a code multilivello** suddivide la ready queue in code distinte

I processi si assegnano ad una determinata coda permanentemente (i processi non possono essere spostati tra le varie code) a seconda di una determinata caratteristica e ogni coda ha il proprio scheduling

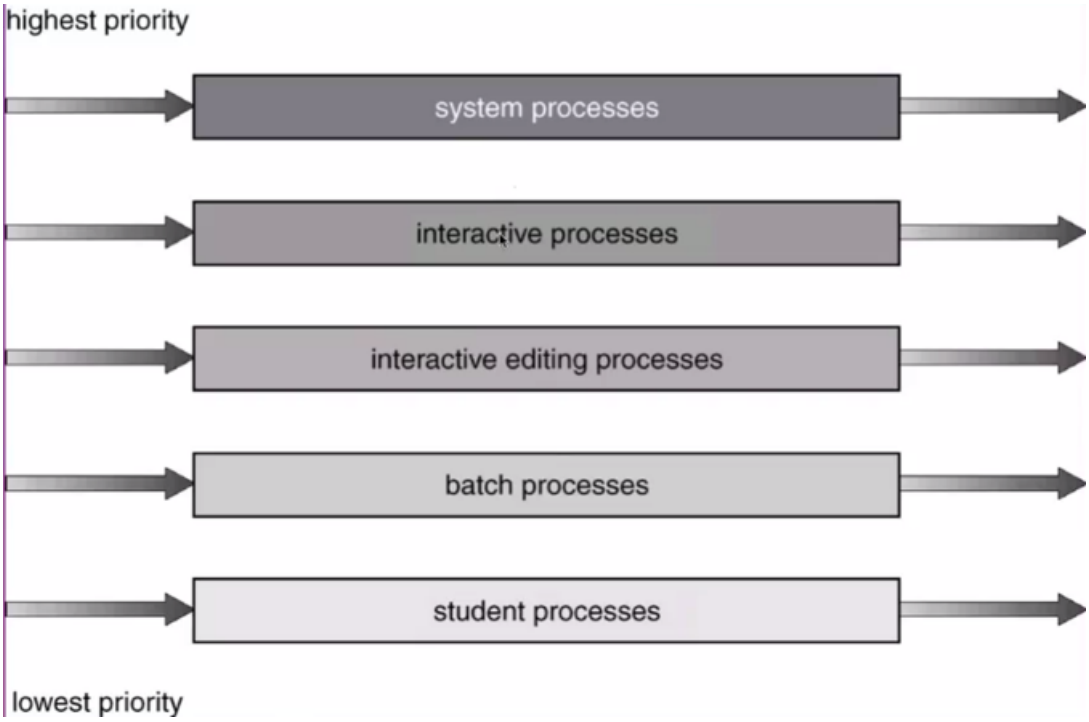
Si necessita, quindi, di un ulteriore scheduling tra le code.

Esso può essere:

- **Scheduling a priorità fissa:** Per esempio eseguire tutti i processi nella coda foreground per poi eseguire quelli in background, o l’esempio sottostante.

Esempio: Il seguente algoritmo di scheduling a code multilivello

1. Processi di sistema
2. Processi interattivi
3. Processi interattivi di editing
4. Processi batch
5. Processi degli studenti



Ogni coda con priorità maggiore ha priorità assoluta rispetto le altre, nel senso che un processo della coda batch non potrà essere eseguito finché non si sono svuotate le code precedenti

- **Quanti di tempo**: ciascuna coda ottiene un quanto di tempo della CPU con cui può schedulare i processi in coda (questo tempo può essere diviso ulteriormente tra i processi)

Esempio: 80% alla coda dei processi in foreground in RR e 20% all'altra in FCFS

Scheduling a code multilivello con retroazione

A differenza del precedente, lo **scheduling a code multilivello con retroazione** permette di spostare i processi tra le varie code.

Il funzionamento principale si basa di dividere i processi in base al loro CPU burst: se un processo utilizza troppo tempo di elaborazione della CPU viene spostato in una coda con priorità più bassa.
Rimangono in una coda con priorità più alta i processi con prevalenza di I/O e i processi interattivi

I processi vengono anche spostati per *invecchiamento*

Consideriamo un esempio con 3 code:

Lo scheduler esegue tutti i processi della coda 0 → una volta svuotata la coda 0, si eseguono quelli della coda 1 → una volta svuotata la coda 0 e la coda 1, si eseguono quelli della coda 2

Processi in coda 0 → prelazione su processi coda 1
Processi in coda 1 → prelazione su processi coda 2

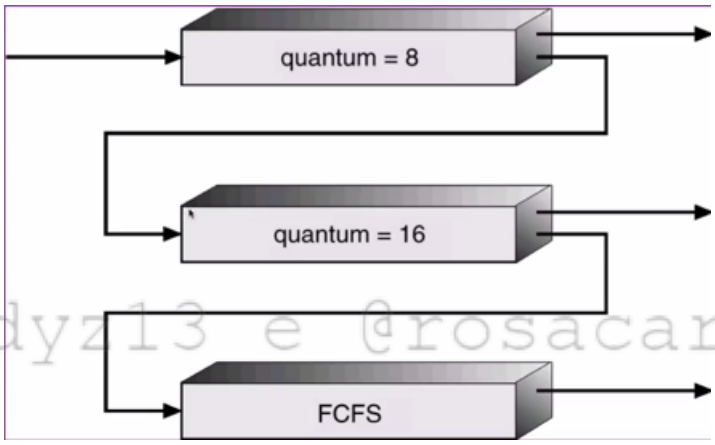
Andiamo più nello specifico in questo esempio:

Le tre code:

- Q_0 - quanto di tempo 8 millisecondi
- Q_1 - quanto di tempo 16 millisecondi
- Q_2 - FCFS

Scheduling:

1. Un nuovo processo entra nella coda Q_0
2. Quando gli viene assegnata la CPU, riceve 8 millisecondi
3. Se non finisce in 8 millisecondi, viene spostato nella coda Q_1
4. Nella coda Q_1 riceve altri 16 millisecondi
5. Se non termina entro questo quanto di tempo, viene deallocato e spostato nella coda Q_2 schedulata con FCFS



Principali caratteristiche di uno scheduler a code multilivello con retroazione:

- numero di code
- algoritmo di scheduling per ciascuna coda
- metodo usato per determinare quando spostare un processo in una coda con priorità maggiore
- metodo usato per determinare quando spostare un processo in una coda con priorità minore
- metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio

Esempio: scheduling di Linux

Lo scheduler di Linux ricorre ad un algoritmo di scheduling con prelazione basato sulle priorità

Ci sono due gamme di priorità separate:

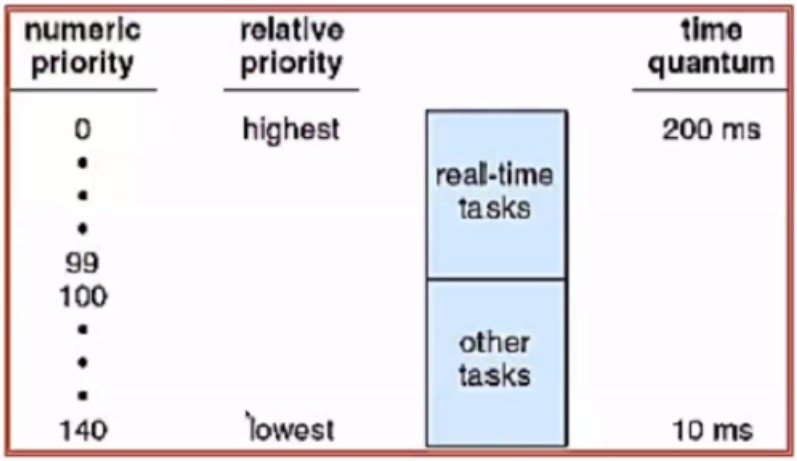
- un intervallo real-time che va da 0 a 99
- un intervallo nice compreso tra 100 e 140

I valori di questi due range sono mappati in un valore di priorità globale: numeri bassi implicano priorità alta

Il kernel mantiene una lista di tutti i task in una **runqueue**

La runqueue contiene due array di priorità:

- attivvo e scaduto (**active array** ed **expired array**)
  - active array : contiene tutti i task che hanno ancora tempo da sfruttare
  - expired array: contiene task scaduti



Valutazione degli algoritmi

Per la valutazione degli algoritmi si utilizzano vari metodi:

- **Modelli deterministici**:
  Fra i metodi di valutazione più importanti ci sono quelli della **valutazione analitica:** essa fornisce una formula o un numero che valita le prestazioni dell'algoritmo per quel carico di lavoro

  Tra i principali tipi di valutazione analitica troviamo la **modellazione deterministica:** considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun alforitmo di schedulazione per quel carico di lavoro

  Il problema è che necessitano in genere di conoscenze troppo dettagliate ed impossibili da ottenere

- **Reti di code:**
  Visto che la quantità di processi cambia di giorno in giorno, non è possibile applicare il modello deterministico. Si possono però prevedere le distribuzioni delle sequenze di operazioni della CPU e delle sequenze di operazioni di I/O. Si può inoltre analizzare la distribuzione degli istanti d'arrivo dei processi nei sistemi

  Da queste possono essere calcolati:

  - l'utilizzo
  - la lunghezza media delle code
  - il tempo medio di attesa

- etc

Il sistema può essere descritto come una *rete di server,* ciascuno con la propria coda di attesa (La CPU è un server con la propria ready queue, così come il sistema di I/O con le code dei dispositivi etc)

Questo tipo di studio si chiama **analisi delle reti di code**

Essa però presenta dei limiti: spesso le distribuzioni di arrivo dei processi e dei servizi (CPU e I/O) sono approssimazioni di un sistema reale

- **Simulazioni:**
  Tramite questa si può avere una valutazione più precisa degli algoritmi di scheduling
  Implicano la programmazione di un modello del sistema (come per esempio un generatore di numeri che simula la quantità di risorse necessarie ad ogni processo)

  Un simulatore dispone di una variabile che rappresenta un clock: con l'aumento del valore di questa variabile, il simulatore modifica lo stato del sistema in modo da descrivere le arrività dei dispositivi, dei processi e dello scheduler

  Durante l'esecuzione delle simulazioni, si raccolgono e si stampano statistiche che indicano le prestazioni degli algoritmi

  Una simulazione può non essere precisa: per rimediare si può sottoporre il sistema reale a un monitoraggio registrando la sequenza degli eventi effettivi, ottenendo così una **tace tape**, che si usa poi per condurre la simulazione

- **Realizzazione:**

  L'unico modo assolutamente sicuro di valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel SO ed osservarne il comportamento nelle reali condizioni di funzionamento.
  Il problema principale di quest'approccio, però, è il suo costo

  Inoltre, un'altra difficoltà è anche il cambio di ambiente in cui l'algoritmo è inserito