
Scheduling

In un computer multiprogrammato più processi competono per l'uso della CPU. La parte di sistema operativo che decide quale processo mandare in esecuzione è lo scheduler.

- Batch OS: scheduling semplice: run to completion, FIFO
- Timesharing OS: utenti batch e interattivi, background daemons
=> problema scheduling più complesso

Criteri/obiettivi

1. fairness: tutti i processi devono avere una parte della CPU
2. efficienza: 100% uso CPU
3. throughput: massimizzare n.job/ora
4. tempo di risposta: minimizzare tempo di attesa degli utenti interattivi
5. tempo di turnaround: minimizzare attesa output di job batch

1-5 sono in parte in conflitto; p.es. 4 vs. 5 (batch di notte/giorno) e, più in generale, i requisiti di classi di utenti diverse (Kleinrock)

Inoltre per sistemi real-time occorre rispettare le deadline (scadenze) e garantire predicibilità del sistema operativo

Per 1 e 4 è indispensabile il preemptive scheduling (clock-driven):

- + fairness, tempo di risposta
- context switch complicato (gestione interrupt) e con overhead
- (pseudo)parallelismo non richiesto dal programmatore => necessità di controllo/coordinazione

NB:

- preemptive s. inevitabile per general purpose time sharing
- non-preemptive s. (detto anche run-to-completion):
utile se OS/scheduler ha informazioni sui tempi di esecuzione dei processi (p.es. scheduler DB)

Round robin

- ogni processo è eseguito per un quanto, uguale per tutti
- al tic, scheduler controlla se quanto del processo running esaurito
- se lo è, lo mette nella coda (dei processi ready)

NB: running può anche:

- rinunciare a CPU deliberatamente (yield)
- rinunciare per attendere IO
- terminare

Struttura dati per coda ready:

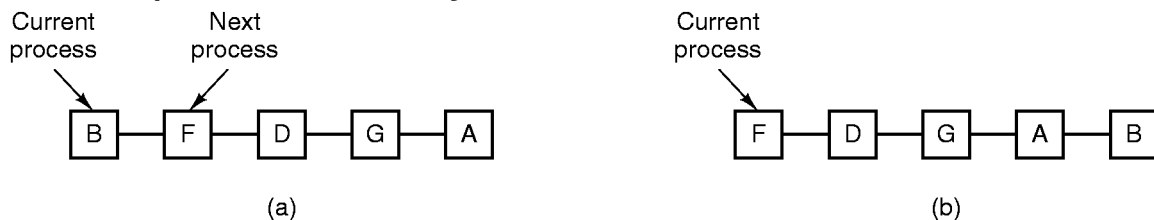


Figure 2-22. Round robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

Problema: scelta quanto (interattività vs. context switch overhead).
Si sceglie circa $100\text{ms} = 10^7$ istruzioni a 100MIPS .

Caso particolare (POSIX):

- POSIX prevede più code, di varia priorità
- per ognuna delle code vale FIFO (sorta di run to completion)

Priorità

- coda dei ready è una coda a priorità
- al tic, running viene sospeso solo se ha priorità < 1° ready

Determinazione priorità:

- statica (p.es. criteri “politici”)
- dinamica/1: a ogni tic, decrementa priorità running
- dinamica/2: abbassamento volontario (nice)
- dinamica/3: alta priorità a processi I/O bound: finché girano, non usano CPU (ma occupano memoria inutilmente)
p.es. priorità=1/f, dove f = frazione ultimo quanto usata (non IO)

Spesso si usa uno schema multiplo:

- classi/code di priorità diversa
- round robin all'interno di classe

Per fairness è opportuno permettere passaggi classe da bassa ad alta

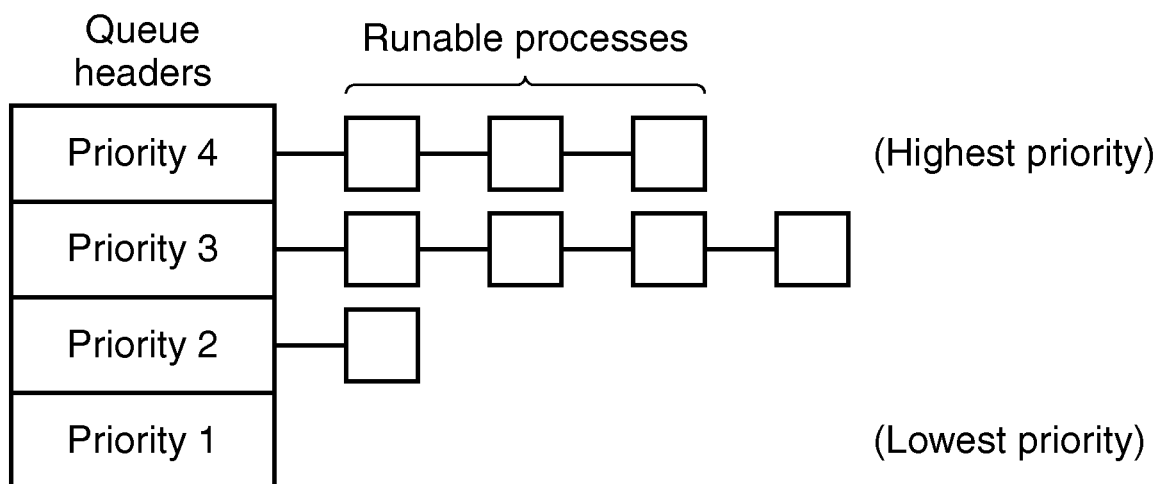


Figure 2-23. A scheduling algorithm with four priority classes.

Strategia tipica: alte priorità/piccoli quanti, basse p./grandi q.

=> max interattività, non penalizza CPU bound, min context switch

SJF (Shortest Job First)

Adatto per job batch (tempi di esecuzione noti/prevedibili).

Risultato: minimizza tempo di attesa/turnaround medio.



Figure 2-24. An example of shortest job first scheduling.

Applicabilità a ambiente interattivo:

- tempo di risposta singola interazione/comando viene considerato come il tempo di turnaround da minimizzare
- stima tempo di esecuzione comando come somma pesata:
prossima stima = $a(\text{ultima stima}) + (1-a)(\text{ultimo tempo misurato})$
 - P.es. $a=1/2$ (facile da implementare con shift) dà stime:
 - 1a stima: T_0
 - 2a stima: $T_0/2+T_1/2$
 - 3a stima: $T_0/4+T_1/4+T_2/2$
 - 4a stima: $T_0/8+T_1/8+T_2/4+T_3/2$
- aging progressivo del peso delle stime passate
- $a \rightarrow 0$ dà più peso ai valori misurati, $a \rightarrow 1$ alle stime

Problema: SJF ottimale solo se job disponibili tutti all'inizio.

Controesempio: A,B,C,D,E durano 2,4,1,1,1 e arrivano in 0,0,3,3,3.

- SJF on-the-fly dà A,B poi C,D,E: tempo di attesa medio = 4.6
- scheduling B,C,D,E,A ha tempo di attesa medio = 4.4
(aspettare B=4 (lungo) ha permesso che arrivassero i 3 job brevi C,D,E)

Scheduling Real-time

Nei sistemi real time è necessario tenere conto del tempo nel portare a termine un compito. Il risultato di un processo si considera corretto se viene fornito entro la propria deadline.

- ogni processo specifica una deadline (scadenza) entro cui terminare
- deadline espresse in genere come intervalli (es. terminare entro 10s)
- priorità = deadline più piccola => processo che ha più urgenza

Per sistemi hard-real time mancare una deadline significa provocare effetti catastrofici. Per sistemi soft-real time mancare una deadline è non desiderato ma tollerabile.

Earliest Deadline First (EDF) Scheduling

- la CPU viene data al processo che ha la deadline minore
- preemptive, toglie la CPU al processo corrente se un altro processo in ready ha deadline più vicina.
- la coda dei processi ready è ordinata in base alla deadline
- le priorità cambiano dinamicamente

Analisi per verificare se le deadline saranno soddisfatte:

Supponiamo che vi siano vari processi che vanno eseguiti periodicamente. Il periodo di ciascun processo è indicato con T_i ; ciascuno richiede al massimo un tempo di computazione C_i ed ha una deadline $D_i = T_i$

Scheduling Real-time

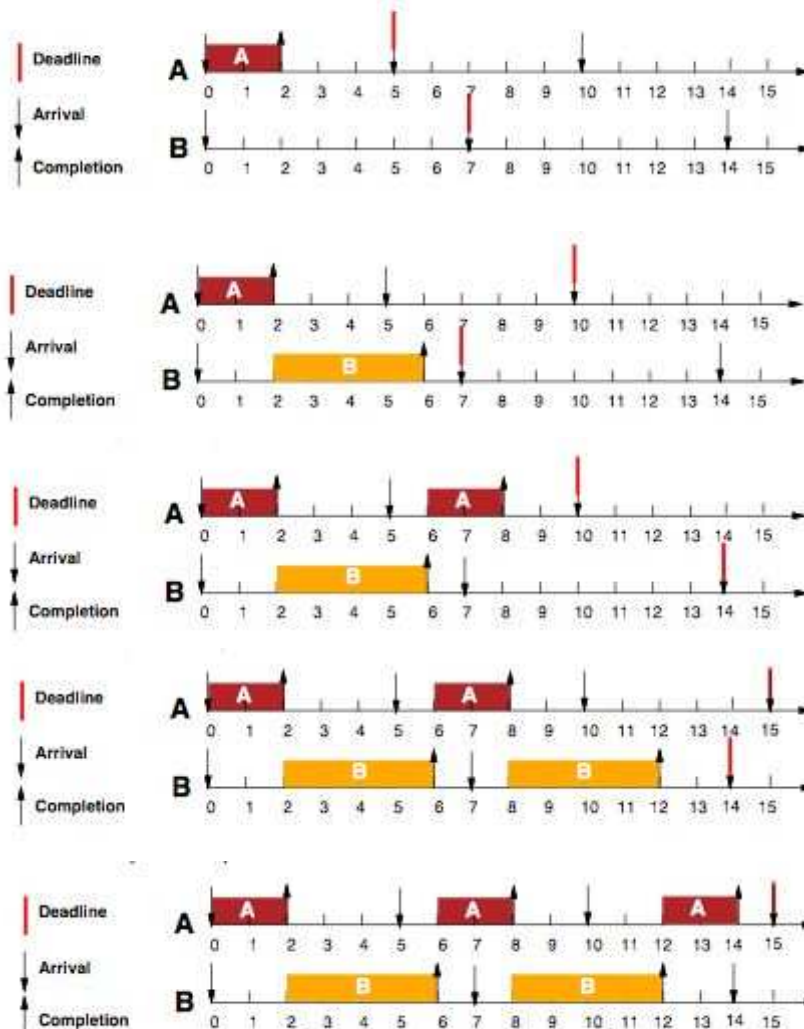
Se l'utilizzazione U del sistema non è maggiore di 1 allora tutte le deadline saranno soddisfatte.

$$U = \sum C_i / T_i$$

E' una condizione necessaria e sufficiente.

Task name	T	D	C
A	5	5	2
B	7	7	4

Utilization: $2/5 + 4/7 = 0.971$



Rate Monotonic Scheduling (RMS)

Ogni processo ha una priorità fissa che è assegnata pari alla frequenza del processo (il task con periodo più corto ha priorità più alta).

Il processo mandato in esecuzione è quello a priorità più alta.

Analisi per verificare se le deadline sono soddisfatte:

Per i processi periodici si ha $D_i = T_i$; le priorità sono uniche

Se $\sum C_i / T_i \leq n(2^{1/n} - 1)$ allora le deadline saranno soddisfatte.

Non è una condizione necessaria, ma sufficiente.