

Università di Udine — Facoltà di Scienze MM.FF.NN.

Laurea in Informatica — A.A. 2017/18

Trasparenze del Corso di *Sistemi Operativi*

Marina Lenisa

Università di Udine

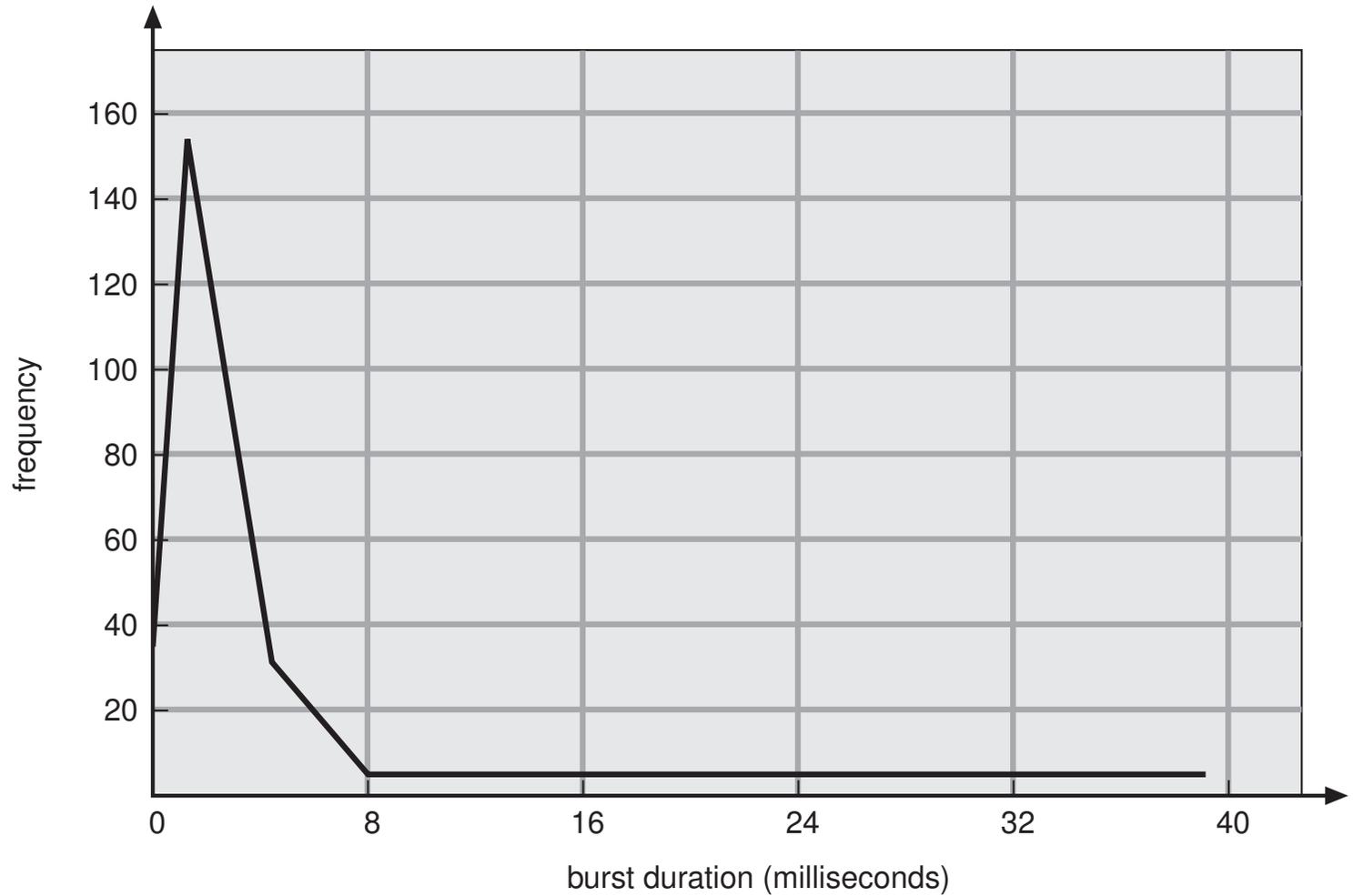
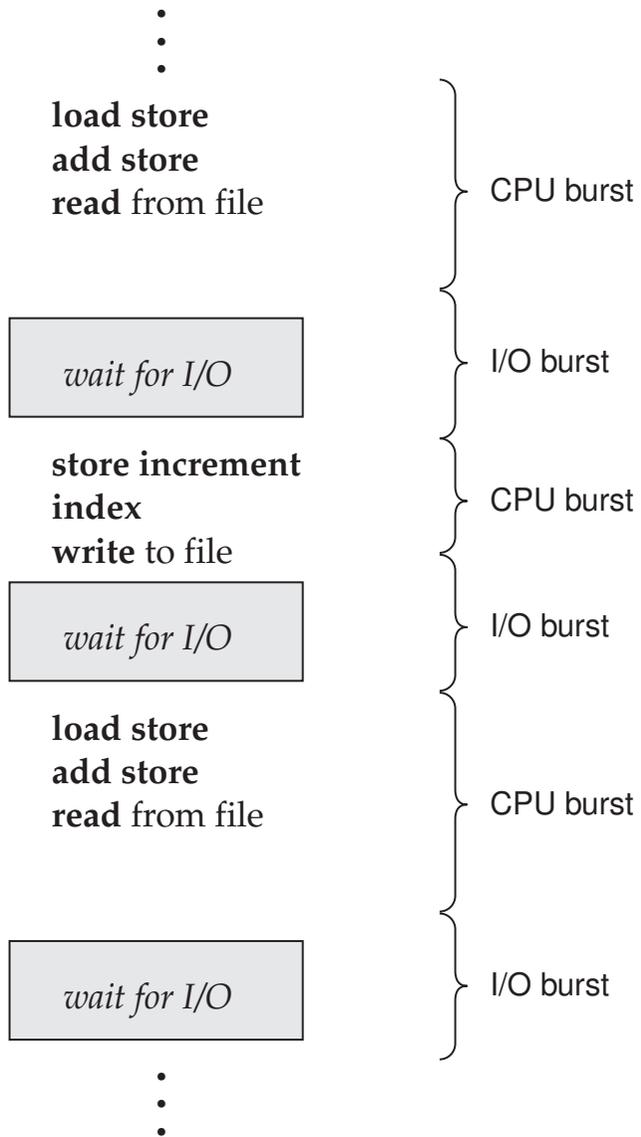
Copyright © 2000-04 Marino Miculan (miculan@dimi.uniud.it)

La copia letterale e la distribuzione di questa presentazione nella sua integrità sono permesse con qualsiasi mezzo, a condizione che questa nota sia riprodotta.

Scheduling della CPU

- Concetti base
 - Massimizzazione dell'uso della CPU attraverso multiprogrammazione
 - Ciclo Burst CPU–I/O: l'esecuzione del processo consiste in un ciclo di periodi di esecuzione di CPU e di attesa di I/O.
- Criteri di Scheduling
- Algoritmi di Scheduling in diversi contesti
- Esempi reali: Unix tradizionale, moderno, Linux, Windows.

I/O e CPU burst



Scheduler a breve termine

- Seleziona tra i processi in memoria e pronti per l'esecuzione, quello a cui allocare la CPU.
- La decisione dello scheduling può avere luogo quando un processo
 1. passa da running a waiting
 2. passa da running a ready
 3. passa da waiting o new a ready
 4. termina.
- Scheduling nei casi 1 e 4 è *nonpreemptive* (senza prelazione)
- Gli altri scheduling sono *preemptive*.
- L'uso della prelazione ha effetti sulla progettazione del kernel (accesso condiviso alle stesse strutture dati)

Dispatcher

- Il *dispatcher* è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler di breve termine. Questo comporta
 - switch di contesto
 - passaggio della CPU da modo supervisore a modo user
 - salto alla locazione del programma utente per riprendere il processo
- È essenziale che sia veloce
- La *latenza di dispatch* è il tempo necessario per fermare un processo e riprenderne un altro

Criteria di Valutazione dello Scheduling

- *Utilizzo della CPU*: mantenere la CPU più carica possibile.
- *Throughput*: # di processi completati nell'unità di tempo
- *Tempo di turnaround*: tempo totale impiegato per l'esecuzione di un processo
- *Tempo di attesa*: quanto tempo un processo ha atteso in ready
- *Tempo di risposta*: quanto tempo si impiega da quando una richiesta viene inviata a quando si ottiene la prima risposta (**non** l'output — è pensato per sistemi time-sharing).
- *Varianza del tempo di risposta*: quanto il tempo di risposta è variabile

Obiettivi generali di un algoritmo di scheduling

All systems

Fairness - giving each process a fair share of the CPU

Policy enforcement - seeing that stated policy is carried out

Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour

Turnaround time - minimize time between submission and termination

CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly

Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data

Predictability - avoid quality degradation in multimedia systems

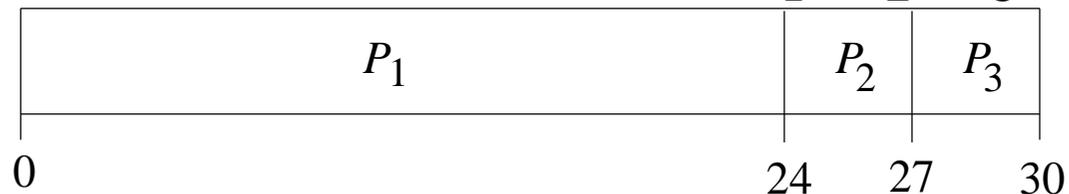
Nota: in generale, non esiste soluzione ottima sotto tutti gli aspetti

Scheduling First-Come, First-Served (FCFS)

- Senza prelazione — inadatto per time-sharing
- Equo: non c'è pericolo di starvation.
- Esempio:

<u>Processo</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

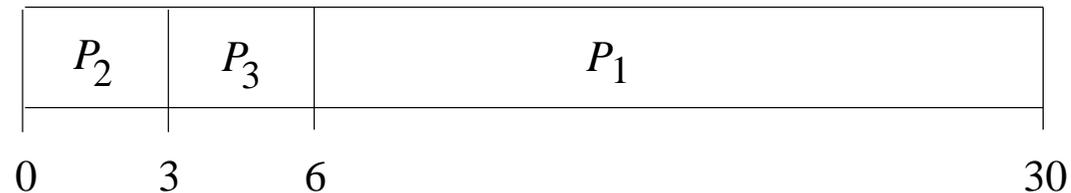
Diagramma di Gantt con l'ordine di arrivo P_1, P_2, P_3



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

Scheduling FCFS (Cont.)

- Supponiamo che i processi arrivino invece nell'ordine P_2 , P_3 , P_1 . Diagramma di Gantt:



- Tempi di attesa: $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Tempo di attesa medio: $(6 + 0 + 3)/3 = 3$
- molto meglio del caso precedente
- *Effetto convoglio*: i processi I/O-bound si accodano dietro un processo CPU-bound.

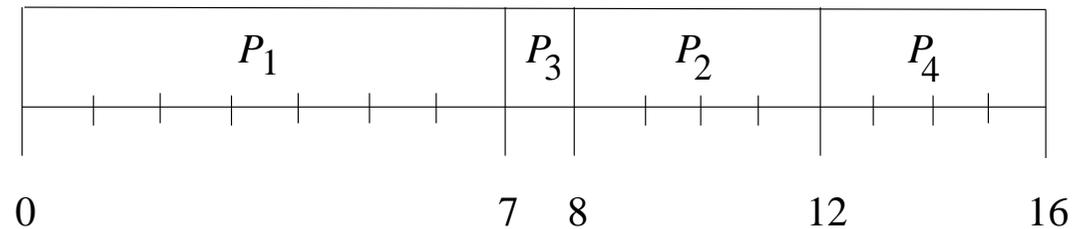
Scheduling Shortest-Job-First (SJF)

- Si associa ad ogni processo la lunghezza del suo prossimo burst di CPU. I processi vengono ordinati e schedulati per tempi crescenti.
- Due schemi possibili:
 - nonpreemptive: quando la CPU viene assegnata ad un processo, questo la mantiene finché non termina il suo burst.
 - preemptive: se nella ready queue arriva un nuovo processo il cui prossimo burst è minore del tempo rimanente per il processo attualmente in esecuzione, quest'ultimo viene prelazionato. (Scheduling *Shortest-Remaining-Time-First*, SRTF).
- SJF è ottimale: fornisce il minimo tempo di attesa per un dato insieme di processi.
- Si rischia la *starvation*

Esempio di SJF Non-Preemptive

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

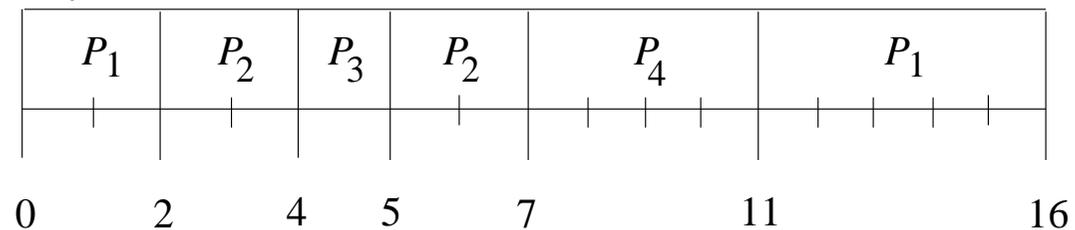


$$\text{Tempo di attesa medio} = (0 + 6 + 3 + 7)/4 = 4$$

Esempio di SJF Preemptive

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SRTF (preemptive)



$$\text{Tempo di attesa medio} = (9 + 1 + 0 + 2)/4 = 3$$

Come determinare la lunghezza del prossimo ciclo di burst?

- Si può solo dare una *stima*
- Nei sistemi batch, il tempo viene stimato dagli utenti
- Nei sistemi time sharing, possono essere usati i valori dei burst precedenti, con una media pesata esponenziale
 1. t_n = tempo dell' n -esimo burst di CPU
 2. τ_{n+1} = valore previsto per il prossimo burst di CPU
 3. α parametro, $0 \leq \alpha \leq 1$
 4. Calcolo:

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

Esempi di media esponenziale

- Espandendo la formula:

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

- Se $\alpha = 0$: $\tau_{n+1} = \tau_0$

– la storia recente non conta

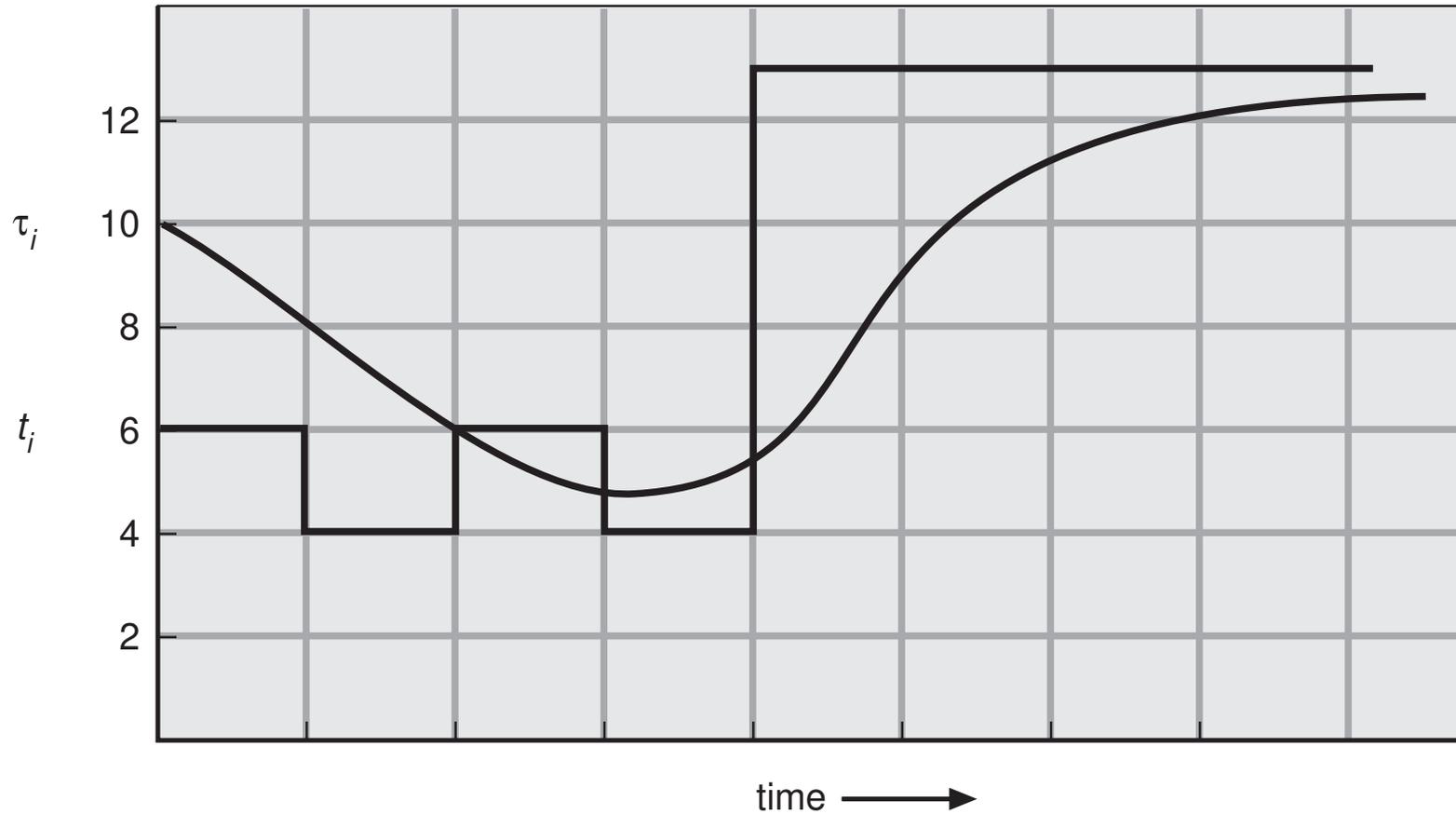
- Se $\alpha = 1$: $\tau_{n+1} = t_n$

– Solo l'ultimo burst conta

- Valore tipico per α : 0.5; in tal caso la formula diventa

$$\tau_{n+1} = \frac{t_n + \tau_n}{2}$$

Predizione con media esponenziale



CPU burst (t _i)	6	4	6	4	13	13	13	...
"guess" (τ _i)	10	8	6	6	9	11	12	...

Scheduling a priorità

- Un numero (intero) di priorità è associato ad ogni processo
- La CPU viene allocata al processo con la priorità più alta (intero più piccolo \equiv priorità più grande)
- Le priorità possono essere definite
 - internamente: in base a parametri misurati dal sistema sul processo (tempo di CPU impiegato, file aperti, memoria, interattività, uso di I/O...)
 - esternamente: importanza del processo, dell'utente proprietario, dei soldi pagati, ...
- Gli scheduling con priorità possono essere preemptive o nonpreemptive
- SJF è uno scheduling a priorità, dove la priorità è il prossimo burst di CPU previsto

Scheduling con priorità (cont.)

- Problema: *starvation* – i processi a bassa priorità possono venire bloccati da un flusso continuo di processi a priorità maggiore
 - vengono eseguiti quando la macchina è molto scarica
 - oppure possono non venire mai eseguiti
- Soluzione: invecchiamento (*aging*) – con il passare del tempo, i processi non eseguiti aumentano la loro priorità

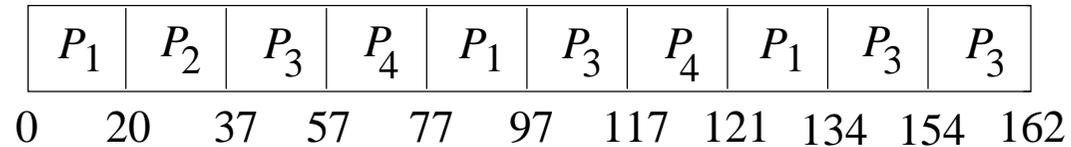
Round Robin (RR)

- Algoritmo con prelazione specifico dei sistemi time-sharing: simile a FCFS ma con prelazione quantizzata.
- Ogni processo riceve una piccola unità di tempo di CPU — il *quanto* — tipicamente 10-100 millisecondi. Dopo questo periodo, il processo viene prelatato e rimesso nella coda di ready.
- Se ci sono n processi in ready, e il quanto è q , allora ogni processo riceve $1/n$ del tempo di CPU in periodi di durata massima q . Nessun processo attende più di $(n - 1)q$

Esempio: RR con quanto = 20

<u>Processo</u>	<u>Burst Time</u>
P_1	53
P_2	17
P_3	68
P_4	24

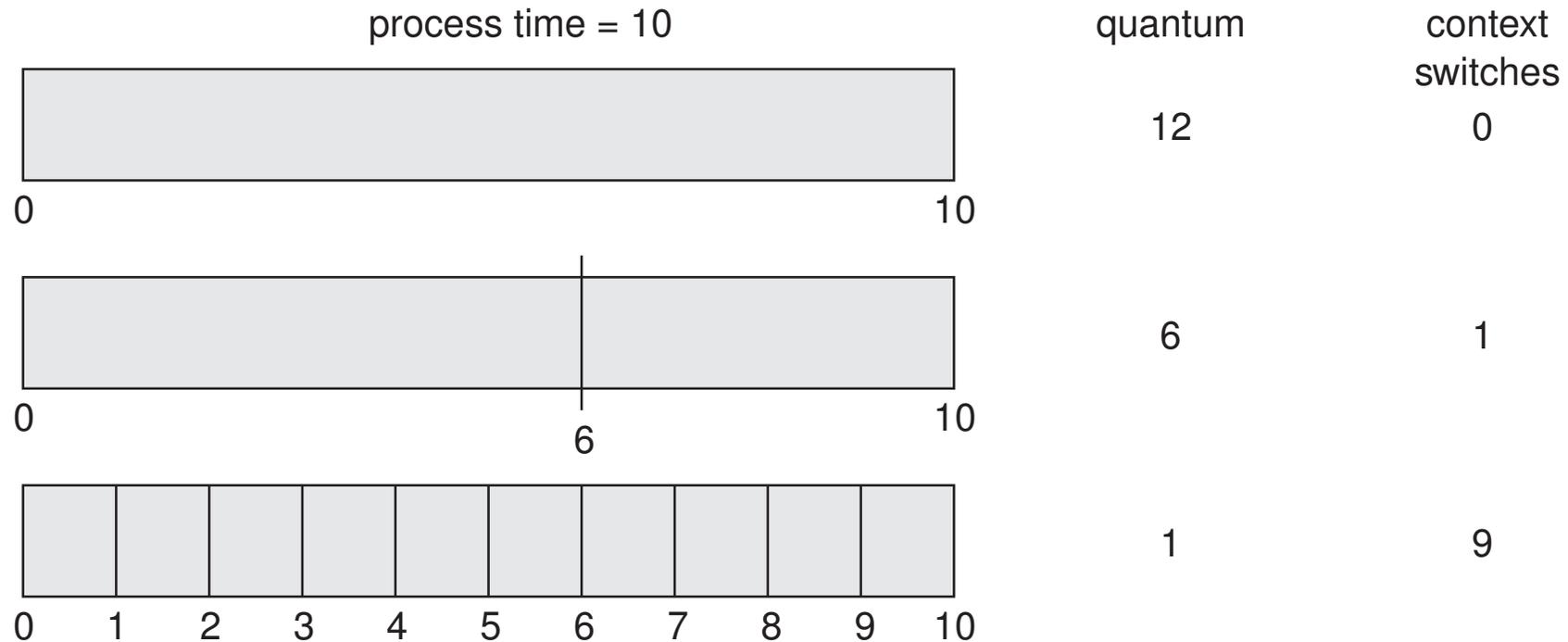
- Diagramma di Gantt



- Tipicamente, si ha un tempo di turnaround medio maggiore, ma minore tempo di risposta

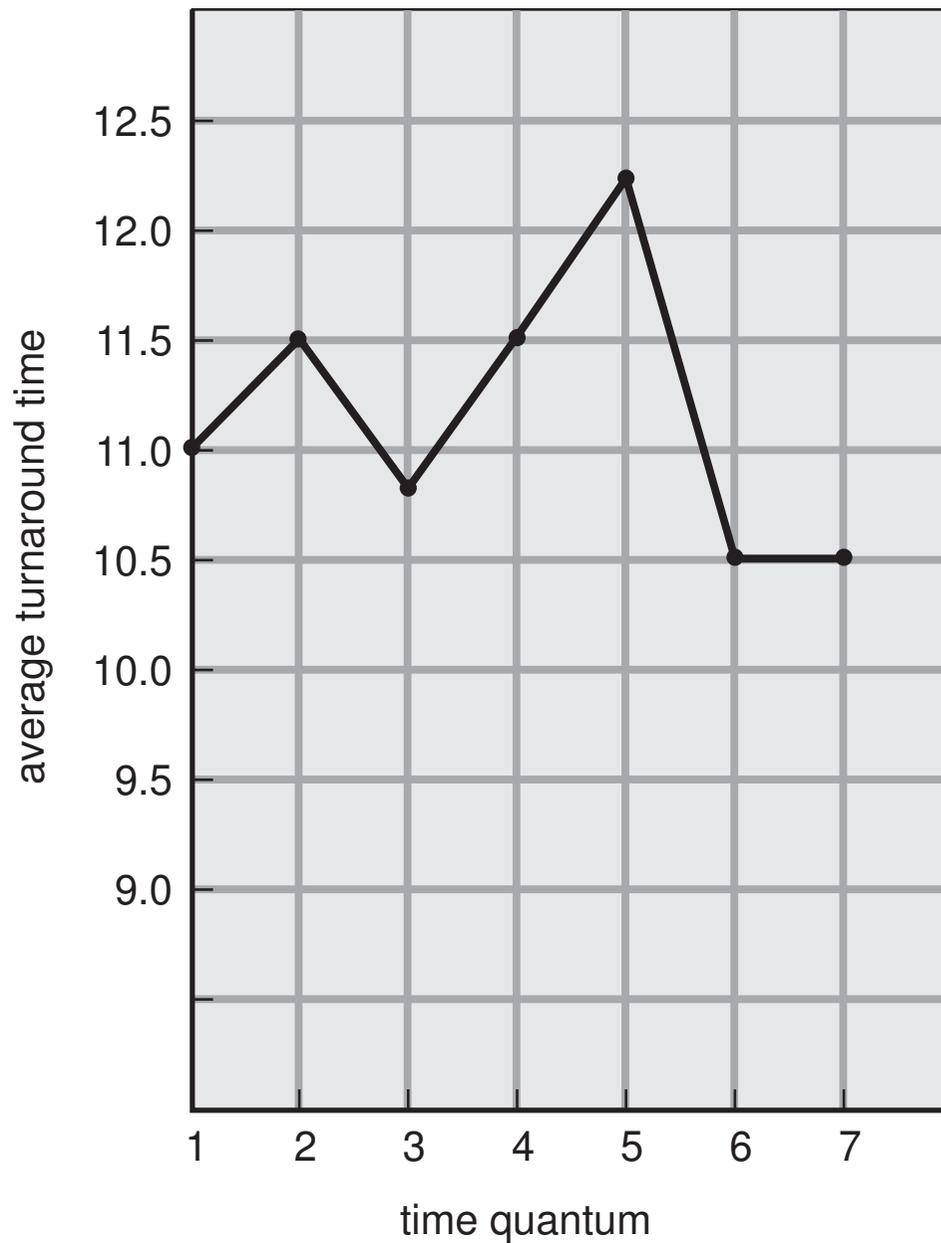
Prestazioni dello scheduling Round-Robin

- q grande \Rightarrow degenera nell'FCFS
- q piccolo $\Rightarrow q$ deve comunque essere grande rispetto al tempo di context switch, altrimenti l'overhead è elevato



- L'80% dei CPU burst dovrebbero essere inferiori a q

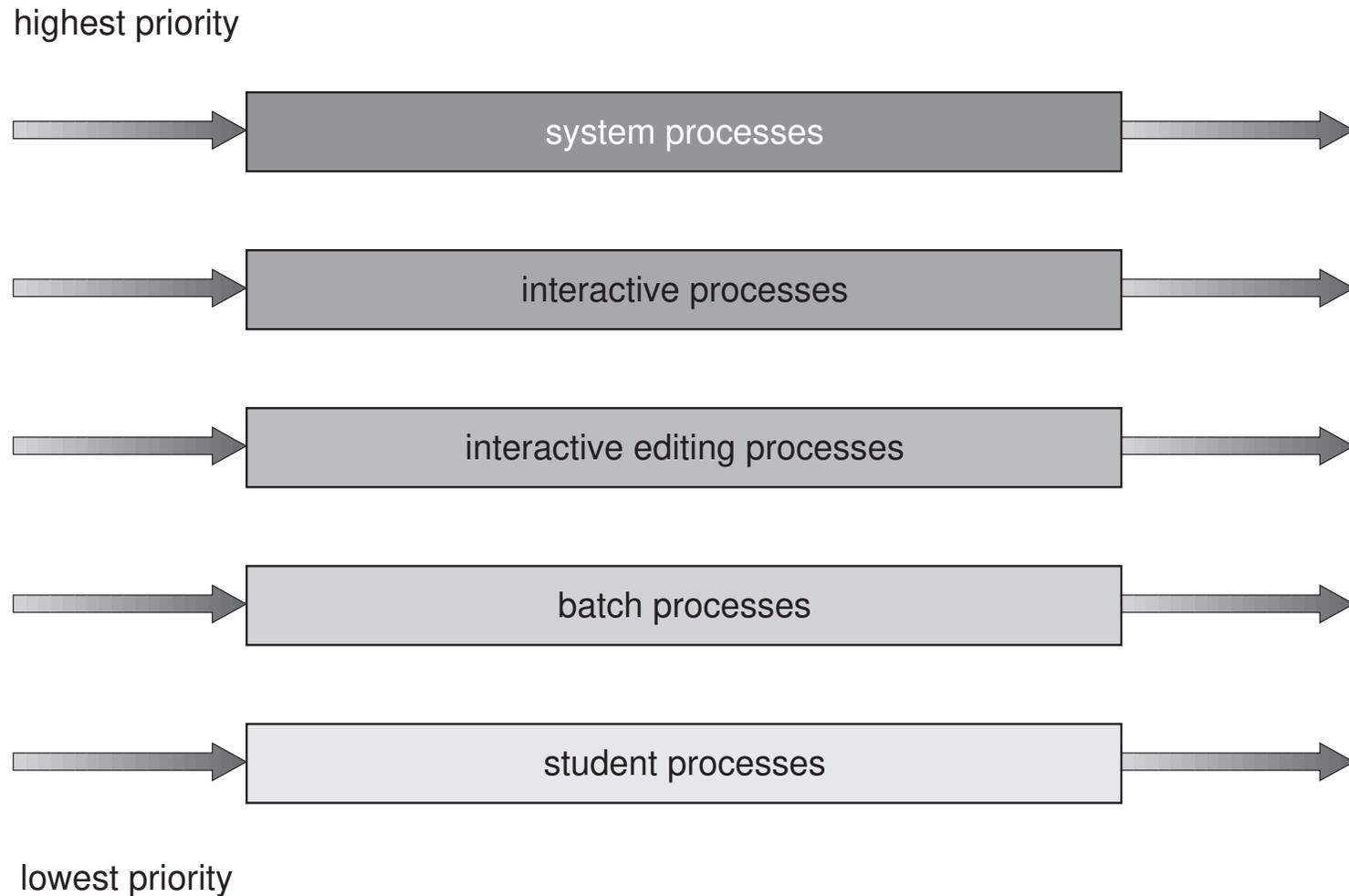
Prestazioni dello scheduling Round-Robin (Cont.)



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Scheduling con code multiple

- La coda di ready è partizionata in più code separate: ad esempio, processi “foreground” (interattivi), processi “background” (batch)



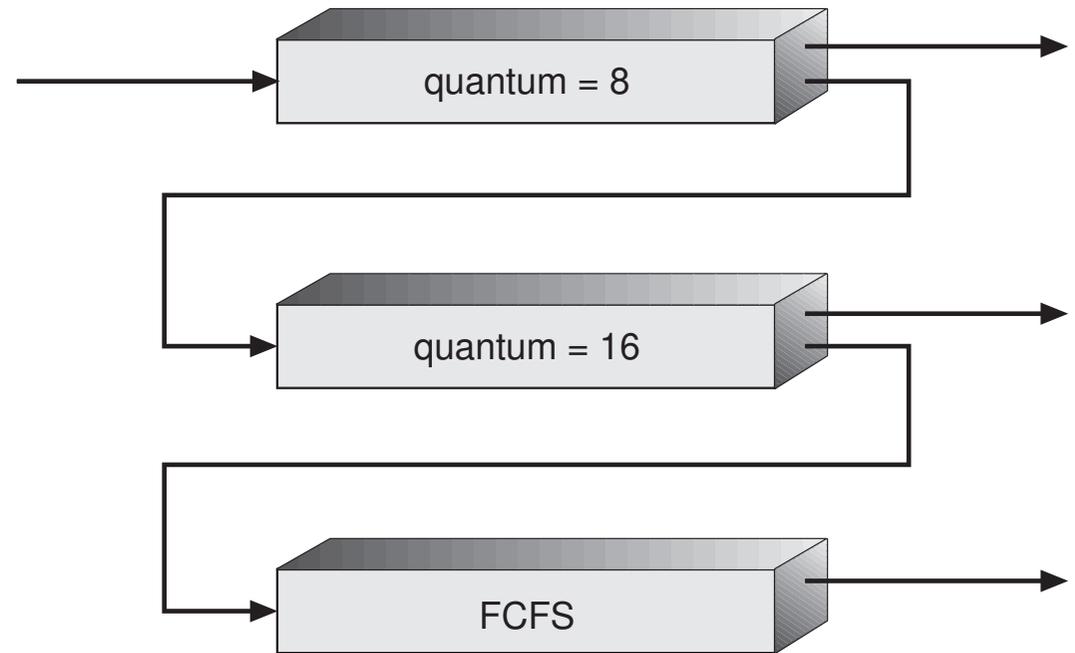
Scheduling con code multiple (Cont.)

- Ogni coda ha un suo algoritmo di scheduling; ad esempio, RR per i foreground, FCFS o SJF per i background
- Lo scheduling deve avvenire tra tutte le code: alternative
 - Scheduling a priorità fissa: eseguire i processi di una coda solo se le code di priorità superiore sono vuote.
⇒ possibilità di starvation.
 - Quanti di tempo per code: ogni coda riceve un certo ammontare di tempo di CPU per i suoi processi; ad es., 80% ai foreground in RR, 20% ai background in FCFS

Scheduling a code multiple con feedback

- I processi vengono spostati da una coda all'altra, dinamicamente. P.e.: per implementare l'aging: se un processo ha usato recentemente
 - molta CPU, viene spostato in una coda a minore priorità
 - poca CPU, viene spostato in una coda a maggiore priorità
- Uno scheduler a code multiple con feedback viene definito dai seguenti parametri:
 - numero di code
 - algoritmo di scheduling per ogni coda
 - come determinare quando promuovere un processo
 - come determinare quando degradare un processo
 - come determinare la coda in cui mettere un processo che entra nello stato di ready

Esempio di code multiple con feedback



Tre code:

- Q_0 – quanto di 8 msec
- Q_1 – quanto di 16 msec
- Q_2 – FCFS

Scheduling:

- Un nuovo job entra in Q_0 , dove viene servito FCFS con prelazione. Se non termina nei suoi 8 millisecondi, viene spostato in Q_1 .
- Nella coda Q_1 , ogni job è servito FCFS con prelazione, quando Q_0 è vuota. Se non termina in 16 millisecondi, viene spostato in Q_2 .
- Nella coda Q_2 , ogni job è servito FCFS senza prelazione, quando Q_0 e Q_1 sono vuote.

Schedulazione garantita

- Si promette all'utente un certo quality of service (che poi deve essere mantenuto)
- Esempio: se ci sono n utenti, ad ogni utente si promette $1/n$ della CPU.
- Implementazione:
 - per ogni processo T_p si tiene un contatore del tempo di CPU utilizzato da quando è stato lanciato.
 - il tempo di cui avrebbe diritto è $t_p = T/n$, dove T = tempo trascorso dall'inizio del processo.
 - priorità di $P = T_p/t_p$ — più è bassa, maggiore è la priorità

Schedulazione a lotteria

- Semplice implementazione di una schedulazione “garantita”
 - Esistono un certo numero di “biglietti” per ogni risorsa
 - Ogni utente (processo) acquisisce un sottoinsieme di tali biglietti
 - Viene estratto casualmente un biglietto, e la risorsa viene assegnata al vincitore
- Per la legge dei grandi numeri, alla lunga l’accesso alla risorsa è proporzionale al numero di biglietti
- I biglietti possono essere passati da un processo all’altro per cambiare la priorità (esempio: client/server)

Scheduling multi-processore (cenni)

- Lo scheduling diventa più complesso quando più CPU sono disponibili
- Sistemi *omogenei*: è indiff. su quale processore esegue il prossimo task
- Può comunque essere richiesto che un certo task venga eseguito su un preciso processore (*pinning*)
- Bilanciare il carico (*load sharing*) \Rightarrow tutti i processori selezionano i processi dalla stessa ready queue
- problema di accesso condiviso alle strutture del kernel
 - *Asymmetric multiprocessing (AMP)*: solo un processore per volta può accedere alle strutture dati del kernel — semplifica il problema, ma diminuisce le prestazioni (carico non bilanciato)
 - *Symmetric multiprocessing (SMP)*: condivisione delle strutture dati. Serve hardware particolare e controlli di sincronizzazione in kernel

Scheduling Real-Time

- *Hard real-time*: si richiede che un task critico venga completato entro un tempo ben preciso e garantito.
 - prenotazione delle risorse
 - determinazione di tutti i tempi di risposta: non si possono usare memorie virtuali, connessioni di rete, . . .
 - solitamente ristretti ad hardware dedicati
- *Soft real-time*: i processi critici sono prioritari rispetto agli altri
 - possono coesistere con i normali processi time-sharing
 - lo scheduler deve mantenere i processi real-time prioritari
 - la latenza di dispatch deve essere la più bassa possibile
 - adatto per piattaforme general-purpose, per trattamento di audio-video, interfacce real-time, . . .

Scheduling Real-Time (cont.)

- Eventi *aperiodici*: imprevedibili (es: segnalazione da un sensore)
- Eventi *periodici*: avvengono ad intervalli di tempo regolari o prevedibili (es.: (de)codifica audio/video).

Dati m eventi periodici, questi sono *schedulabili* se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

dove

- P_i = periodo dell'evento i
- C_i = tempo di CPU necessario per gestire l'evento i

Scheduling RMS (Rate Monotonic Scheduling)

- a priorità statiche, proporzionali alla frequenza.
- Lo schedulatore esegue sempre il processo pronto con priorità maggiore, eventualmente prelazionando quello in esecuzione
- Solo per processi periodici, a costo costante.

- Garantisce il funzionamento se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

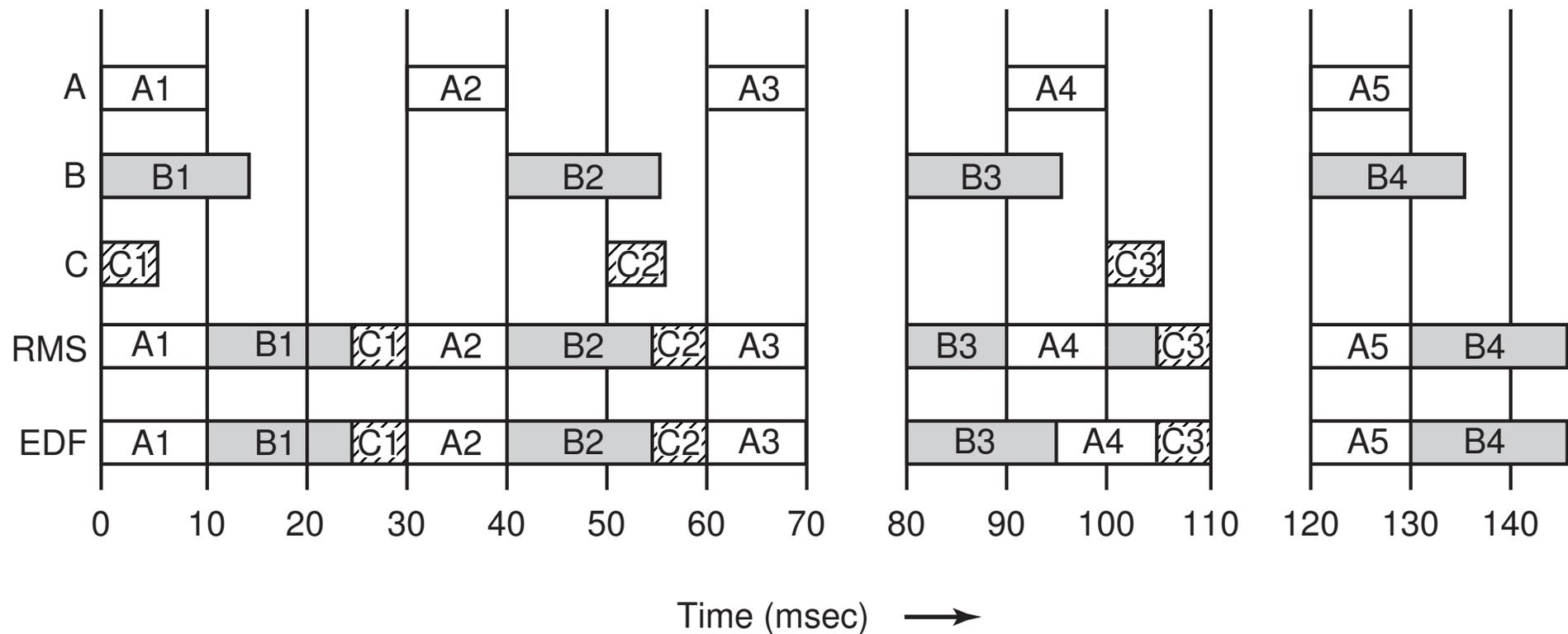
Con $m = 3$, il limite è 0,780.

Per $m \rightarrow \infty$, questo limite tende a $\log 2 = 0,693$.

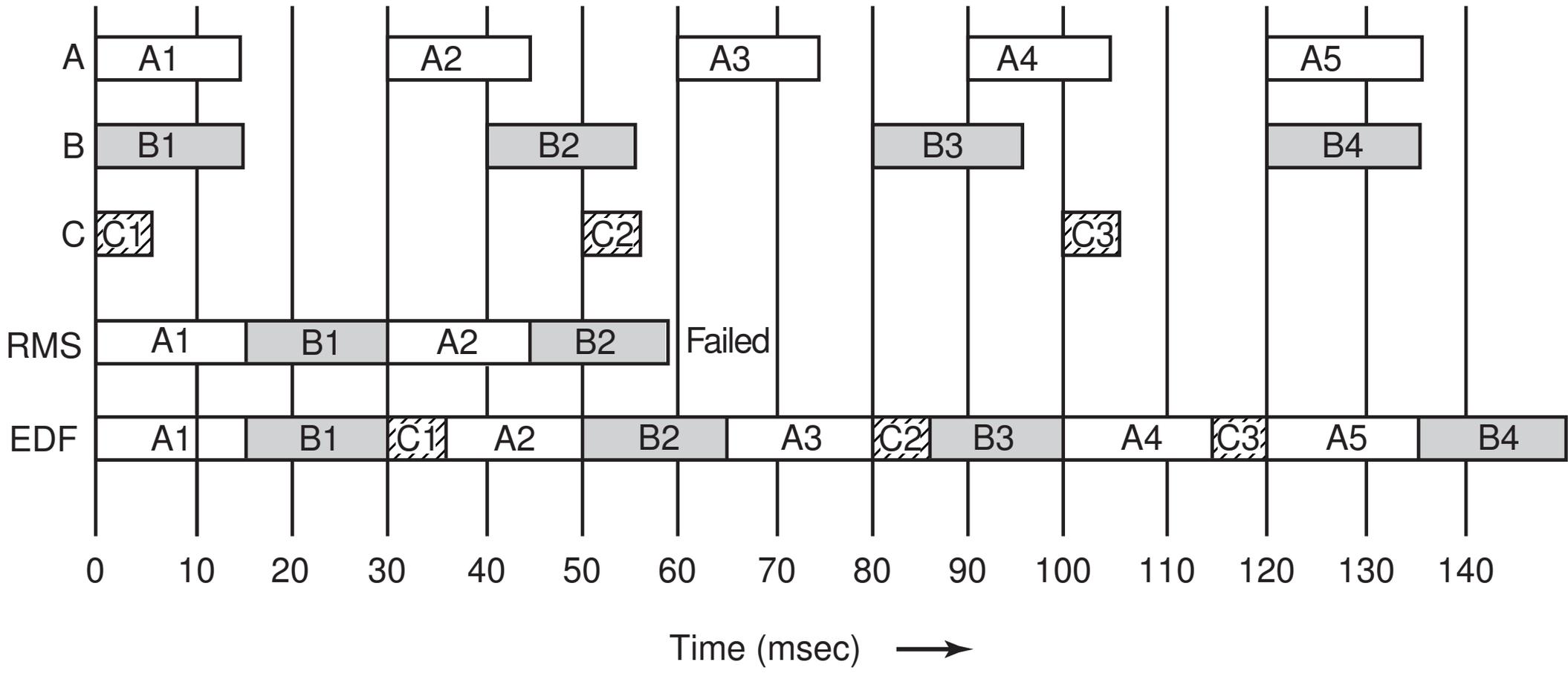
- Semplice da implementare

Scheduling EDF (Earliest Deadline First)

- a priorità dinamiche, in base a chi scade prima.
- Adatto anche per processi non periodici.
- Permette di raggiungere anche il 100% di utilizzo della CPU.
- Più complesso (e costoso) di RMS



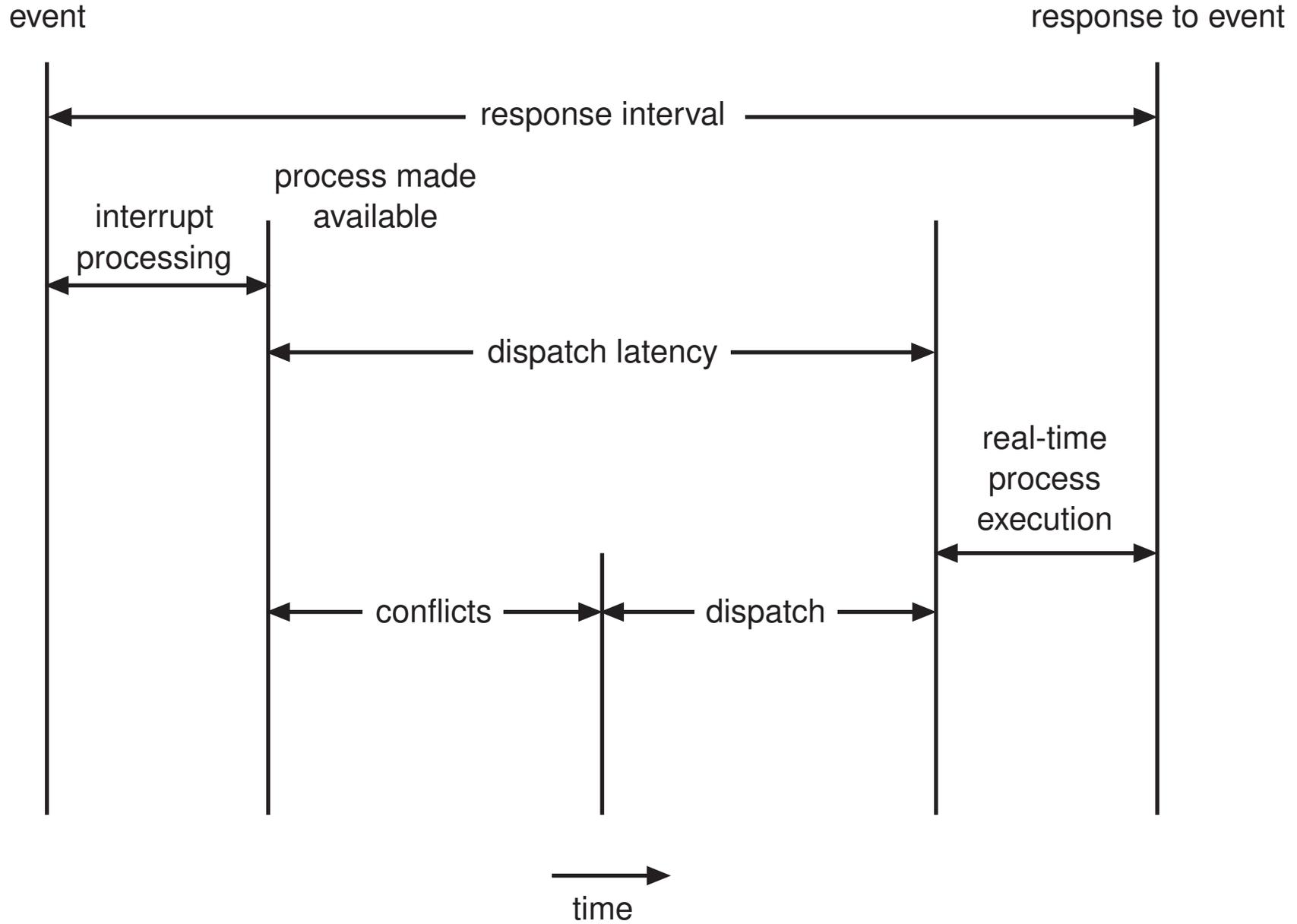
Esempio di fallimento di RMS



Minimizzare il tempo di latenza

- Un kernel *non prelazionabile* è inadatto per sistemi real-time: un processo non può essere prelazionato durante una system call
 - *Punti di prelazionabilità (preemption points)*: in punti “sicuri” delle system call di durata lunga, si salta allo scheduler per verificare se ci sono processi a priorità maggiore
 - *Kernel prelazionabile*: tutte le strutture dati del kernel vengono protette con metodologie di sincronizzazione (semafori). In tal caso un processo può essere sempre interrotto.
- *Inversione delle priorità*: un processo ad alta priorità deve accedere a risorse attualmente allocate da un processo a priorità inferiore.
 - *protocollo di ereditarietà delle priorità*: il processo meno prioritario eredita la priorità superiore finché non rilascia le risorse.

Latenza di dispatch (cont.)

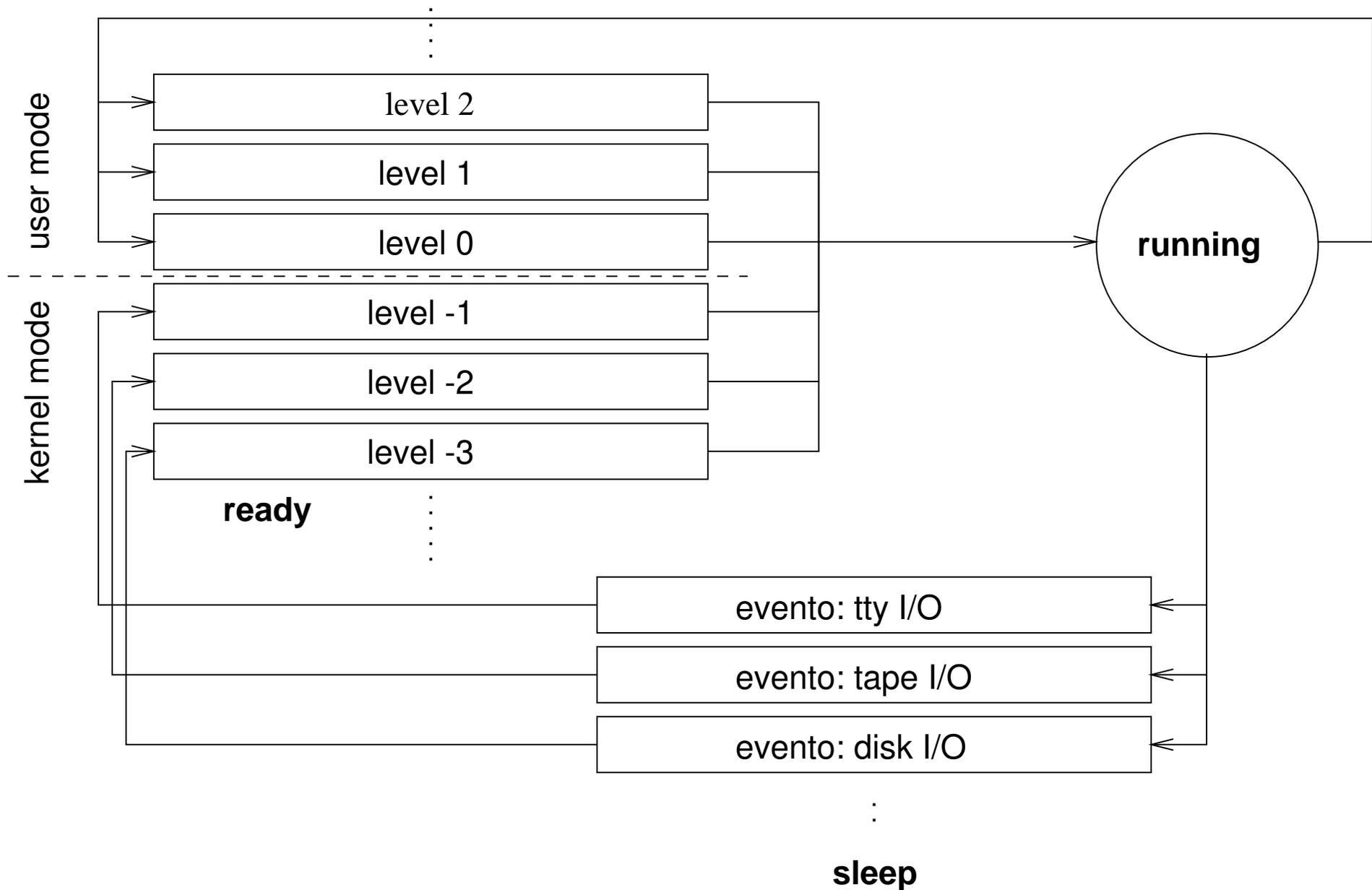


Scheduling di breve termine in Unix tradizionale

(fino a 4.3BSD e SVR3)

- a code multiple, round-robin
- ogni processo ha una priorità di scheduling; numeri più grandi indicano priorità minore
- Feedback negativo sul tempo di CPU impiegato
- Invecchiamento dei processi per prevenire la starvation
- Quando un processo rilascia la CPU, va in *sleep* in attesa di un *event*
- Quando l'evento occorre, il kernel esegue un *wakeup* con l'indirizzo dell'evento e *tutti* i processi che erano in *sleep* sull'evento vengono messi nella coda di ready
- I processi che erano in attesa di un evento in modo kernel rientrano con priorità *negativa* e non soggetta a invecchiamento

Scheduling in Unix tradizionale (Cont.)



Scheduling in Unix tradizionale (Cont.)

- 1 quanto = 5 o 6 tick = 100 msec
- alla fine di un quanto, il processo viene prelazionato
- quando il processo j rilascia la CPU
 - viene incrementato il suo contatore CPU_j di uso CPU
 - viene messo in fondo alla stessa coda di priorità
 - riparte lo scheduler su tutte le code
- 1 volta al secondo, vengono ricalcolate tutte le priorità dei processi in user mode (dove $nice_j$ è un parametro fornito dall'utente):
$$CPU_j = CPU_j / 2 \quad (\text{fading esponenziale})$$
$$P_j = CPU_j + nice_j$$

I processi in kernel mode non cambiano priorità.

Scheduling in Unix tradizionale (Cont.)

In questo esempio, 1 secondo = 4 quanti = 20 tick

Tempo	Processo A		Processo B		Processo C	
	Pr _A	CPU _A	Pr _B	CPU _B	Pr _C	CPU _C
0	0	0	0	0	0	0
	0	5	0	0	0	0
	0	5	0	5	0	0
	0	5	0	5	0	5
1	0	10	0	5	0	5
	5	5	2	2	2	2
	5	5	2	7	2	2
	5	5	2	7	2	7
2	5	5	2	12	2	7
	5	5	2	12	2	12
	2	2	6	6	6	6
	2	7	6	6	6	6
3	2	12	6	6	6	6
	2	17	6	6	6	6
	2	22	6	6	6	6
	11	11	3	3	3	3
...	11	11	3	8	3	3
	11	11	3	8	3	8
	11	11	3	13	3	8

Scheduling in Unix tradizionale (Cont.)

Considerazioni

- Adatto per time sharing generale
- Privilegiati i processi I/O bound - tra cui i processi interattivi
- Garantisce assenza di starvation per CPU-bound e batch
- Quanto di tempo indipendente dalla priorità dei processi
- Non adatto per real time
- Non modulare, estendibile

Inoltre il kernel 4.3BSD e SVR3 non era prelaizionabile e poco adatto ad architetture parallele.

Scheduling in Unix moderno (4.4BSD, SVR4 e successivi)

Applicazione del principio di separazione tra il meccanismo e le politiche

- Meccanismo generale
 - 160 livelli di priorità (numero maggiore \equiv priorità maggiore)
 - ogni livello è gestito separatamente, event. con politiche differenti
- *classi di scheduling*: per ognuna si può definire una politica diversa
 - intervallo delle priorità che definisce la classe
 - algoritmo per il calcolo delle priorità
 - assegnazione dei quanti di tempo ai vari livelli
 - migrazione dei processi da un livello ad un altro
- Limitazione dei tempi di latenza per il supporto real-time
 - inserimento di punti di prelaionabilità del kernel con check del flag `kprunrun`, settato dalle routine di gestione eventi
 - kernel più oderni completamente prelaionabili.

Scheduling in Unix moderno (4.4BSD, SVR4 e successivi)

Assegnazione di default: 3 classi

Real time: possono prelazionare il kernel.

Hanno priorità e quanto di tempo fisso.

Kernel: prioritari su processi time shared.

Hanno priorità e quanto di tempo fisso.

Ogni coda è gestita FCFS.

Time shared: per i processi “normali”.

Ogni coda è gestita round-robin, con

quanto minore per priorità maggiore.

Priorità variabile secondo una tabella

fissa: se un processo termina il suo

quanto, scende di priorità.

Priority Class	Global Value	Scheduling Sequence
Real-time	159 • • • • 100	first ↓
Kernel	99 • • 60	↓
Time-shared	59 • • • • 0	↓ last

Considerazioni sullo scheduling SVR4

- Flessibile: configurabile per situazioni particolari
- Modulare: si possono aggiungere altre classi con politiche proprie (p.e., batch)
- Le politiche di default sono adatte ad un sistema time-sharing generale
- manca(va) uno scheduling real-time FIFO (aggiunto in Solaris, Linux, ...)

Classi di Scheduling: Solaris

```
lenisa@maxi:lenisa$ priocntl -l
CONFIGURED CLASSES
=====
SYS (System Class)
TS (Time Sharing)
    Configured TS User Priority Range: -60 through 60
RT (Real Time)
    Maximum Configured RT Priority: 59
IA (Interactive)
    Configured IA User Priority Range: -60 through 60
lenisa@maxi:lenisa$
```

In aggiunta due ulteriori classi introdotte in Solaris 9:

- FSS (fair share), a ripartizione equa: un insieme di processi, *progetto*, condivide una *quota* di CPU
- FP, a priorità fissa: stesso livello di priorità di TS, ma priorità fissa

Classi di Scheduling in Solaris: Real-Time

```
lenisa@maxi:lenisa$ dispadmin -c RT -g
# Real Time Dispatcher Configuration
RES=1000
```

```
# TIME QUANTUM          PRIORITY
# (rt_quantum)         LEVEL
    1000                #          0
    ...
    1000                #          9
    800                 #         10
    ...
    800                 #         19
    600                 #         20
    ...
    600                 #         29
    400                 #         30
    ...
    400                 #         39
    200                 #         40
    ...
    200                 #         49
    100                 #         50
    ...
    100                 #         59
```

```
lenisa@maxi:lenisa$
```

Classi di Scheduling in Solaris: Time-Sharing

```
lenisa@maxi:lenisa$ dispadmin -c TS -g
# Time Sharing Dispatcher Configuration
RES=1000
```

```
# ts_quantum ts_tqexp ts_slpret ts_maxwait ts_lwait PRI LEVEL
  200         0        50          0         50    #    0
  200         0        50          0         50    #    1
  200         0        50          0         50    #    2
  ...
  200         0        50          0         50    #    9
  160         0        51          0         51    #   10
  160         1        51          0         51    #   11
  160         2        51          0         51    #   12
  ...
  160         8        51          0         51    #   18
  160         9        51          0         51    #   19
  120        10        52          0         52    #   20
  ...
  120        19        52          0         52    #   29
   80        20        53          0         53    #   30
   80        21        53          0         53    #   31
  ...
   80        29        54          0         54    #   39
   40        30        55          0         55    #   40
   40        31        55          0         55    #   41
  ...
   40        48        58           0         59    #   58
   20        49        59       32000        59    #   59
```

```
lenisa@maxi:lenisa$
```

Classi di Scheduling in Solaris: Time-Sharing

RES: *resolution* della colonna `ts_quantum` (1000=millesimi di secondo)

ts_quantum: quanto di tempo

ts_tqexp: *time quantum expiration level*: livello a cui portare un processo che ha terminato il suo quanto

ts_slpret: *sleep priority return level*: livello a cui portare il processo dopo un wakeup

ts_maxwait, ts_lwait: se un processo aspetta in coda ready per più di `ts_maxwait` secondi, viene portato a `ts_lwait` (ogni secondo)

L'utente `root` può modificare run-time le tabelle di scheduling con il comando `dispadm`. **ESTREMA CAUTELA!!**

Scheduling di Linux

- gira in tempo costante, $O(1)$
- adatto per SMP (Symmetric MultiProcessing)
- scheduling con prelazione basato su priorità
- 2 tipi di priorità: *real time* (0–99), *nice* (100–140); valori più bassi corrispondono a priorità più alte
- a priorità più basse corrispondono quanti più lunghi (a differenza da Unix e Windows)

Linux: relazione tra priorità e quanto

valore numerico della priorità	priorità relativa		quanto di tempo
0	massima		200ms
⋮		task real-time	⋮
99			
100			
⋮		altri task	⋮
140	minima		10ms

Scheduling in Linux (Cont.)

- ogni processore mantiene una *coda di esecuzione* (runqueue) con due liste di task: *attivi* e *scaduti*
- un task è attivo se non ha terminato il suo quanto, altrimenti è scaduto
- le due liste sono ordinate secondo la priorità dei task
- lo scheduler sceglie il task attivo di maggior priorità
- quando l'array attivo si svuota, i due array si scambiano di ruolo

Scheduling in Linux (Cont.)

- task real-time hanno priorità statica
- gli altri task hanno priorità dinamica basata su nice +5 o -5, a seconda del grado di interattività
- tempi di attesa più lunghi sui dispositivi di I/O danno un bonus di -5
- la priorità dinamica di un task è ricalcolata quando passa da attivo a scaduto

Scheduling di Windows

Un thread esegue lo scheduler quando

- esegue una chiamata bloccante
- comunica con un oggetto (per vedere se si sono liberati thread a priorità maggiore)
- alla scadenza del quanto di thread

Inoltre si esegue lo scheduler in modo asincrono:

- Al completamento di un I/O
- allo scadere di un timer (per chiamate bloccanti con timeout)

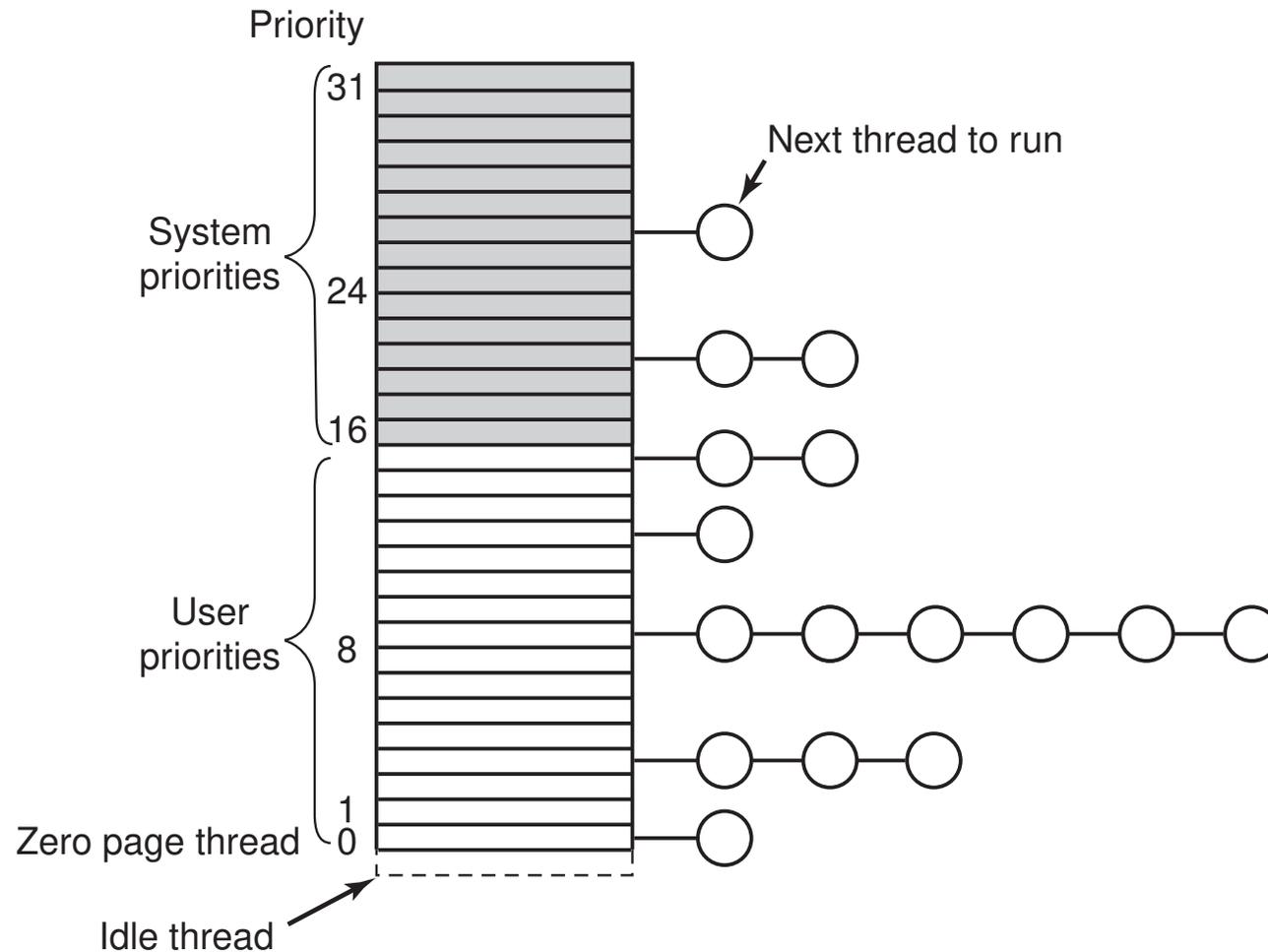
Scheduling di Windows

- I processi possono settare la classe priorità di *processo* (SetPriorityClass)
- I singoli thread possono settare la priorità di *thread* (SetThreadPriority)
- Queste determinano la *priorità di base* dei thread come segue:

		Win32 process class priorities						
Win32 thread priorities		Realtime	High	Above Normal	Normal	Below Normal	Idle	
		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

Scheduling di Windows

- I thread (NON i processi) vengono raccolti in code ordinate per priorità, ognuna gestita round robin. Quattro classi: *system* (“real time”, ma non è vero), *utente*, *zero*, *idle*.



Scheduling di Windows (cont.)

- Lo scheduler sceglie sempre dalla coda a priorità maggiore
- La priorità di un thread utente può essere temporaneamente maggiore di quella base (*spinte*)
 - per thread che attendevano dati di I/O (spinte fino a +8)
 - per dare maggiore reattività a processi interattivi (+2)
 - per risolvere inversioni di priorità