Scheduling della CPU

Obiettivi

- Comprendere i principali criteri per lo scheduling della CPU
- Descrivere vari algoritmi di scheduling della CPU
- Spiegare le problematiche relative allo scheduling multiprocessore e multicore
- Valutare gli algoritmi di scheduling della CPU tramite metodi differenti

Scheduling della CPU

- Concetti fondamentali
- Criteri di scheduling
- Algoritmi di scheduling
- Scheduling dei thread
- Scheduling per sistemi multiprocessore
- Valutazione degli algoritmi di scheduling

Scheduling dei processi/thread

- Nella maggior parte dei SO moderni sono i thread a livello kernel, non i processi, ad essere schedulati dal SO
- Tuttavia, i termini scheduling dei processi e scheduling dei thread sono spesso usati in modo interscambiabile
- Noi parleremo di scheduling dei processi quando ci riferiremo a concetti generali di scheduling e di scheduling dei thread quando vorremo fare specifico riferimento allo scheduling dei thread
- Useremo anche la terminologia generale di scheduling per l'esecuzione su una CPU, anche se in realtà l'esecuzione può avvenire su un core di una CPU

Scheduling della CPU

- Concetti fondamentali
- Criteri di scheduling
- Algoritmi di scheduling
- Scheduling dei thread
- Scheduling per sistemi multiprocessore
- Valutazione degli algoritmi di scheduling

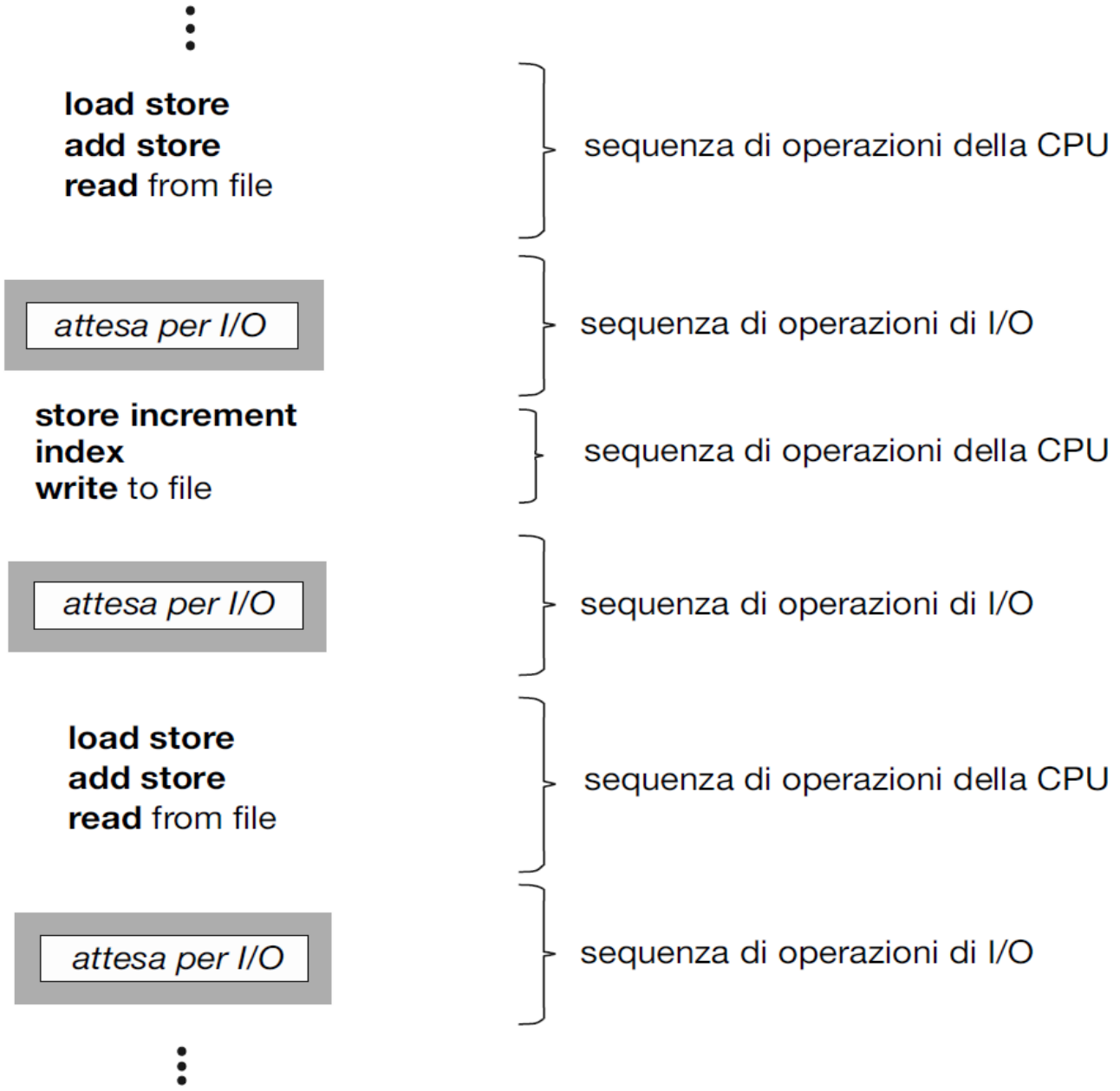
Concetti fondamentali

- Obiettivo della multiprogrammazione: utilizzare al meglio la CPU avendo sempre un processo in esecuzione
- Obiettivo del multitasking (o time-sharing): commutare la CPU tra i processi con una frequenza tale che gli utenti possano interagire con ciascun programma mentre questo è in esecuzione
- Per raggiungere questi obiettivi, il SO
  - mantiene contemporaneamente in memoria un insieme di processi
  - quando la CPU diventa disponibile lo scheduler a breve termine seleziona un processo tra quelli pronti per l'esecuzione

Alternanza ciclica

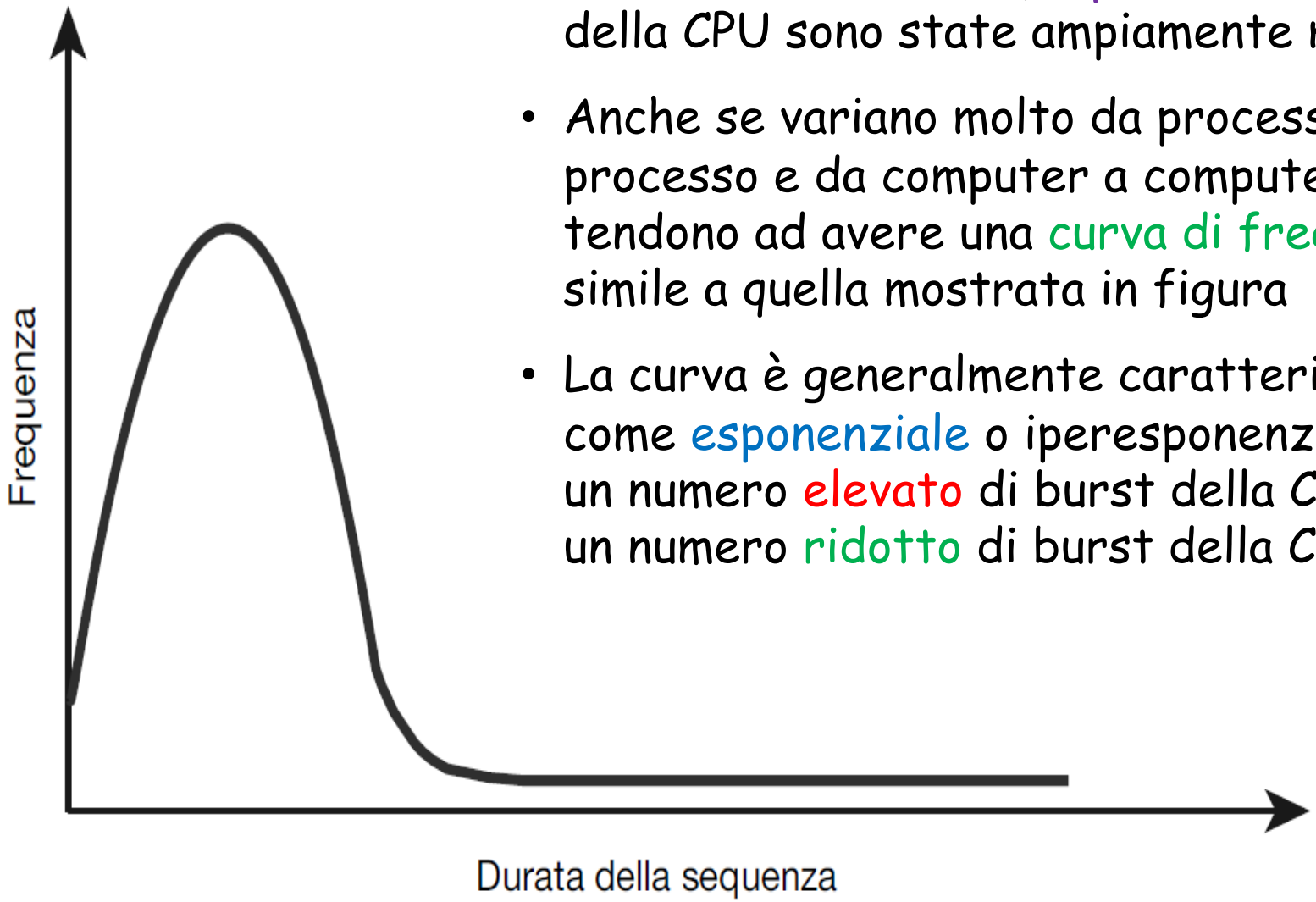
Ogni processo, durante l'esecuzione, alterna ciclicamente tra due fasi:

- Esecuzione di istruzioni (*CPU burst*)
- Attesa di eventi o operazioni esterne (*I/O burst*)



burst = sequenza

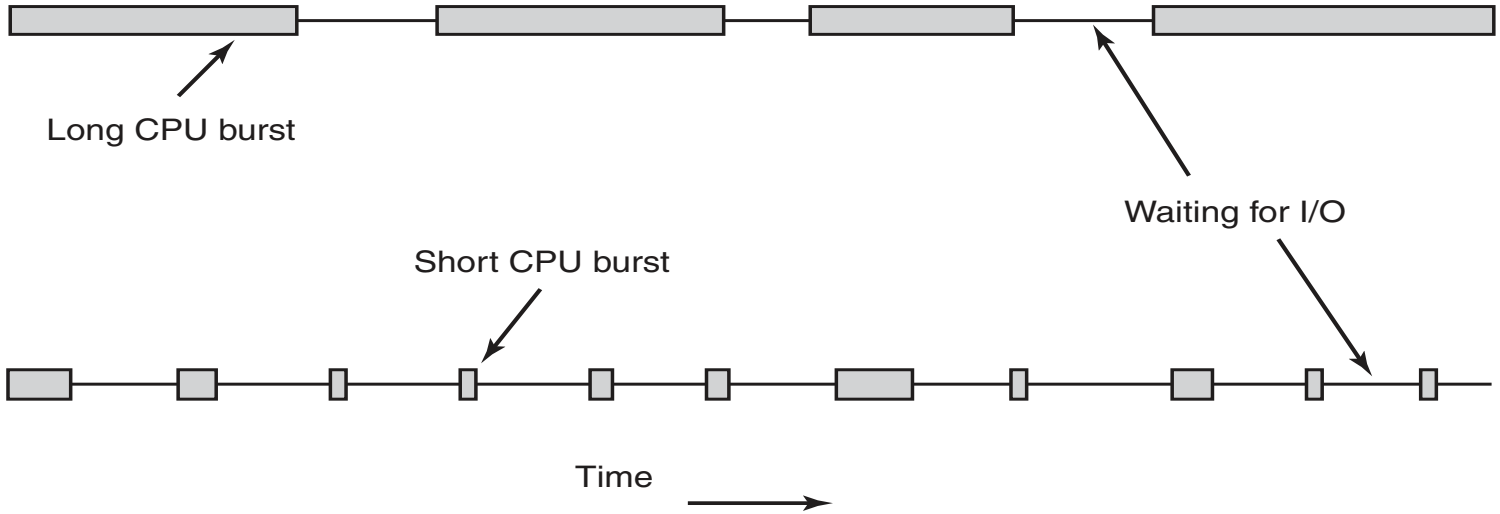
Diagramma dei tempi di CPU-burst



- Le durate dei burst (sequenza di istruzioni) della CPU sono state ampiamente misurate
- Anche se variano molto da processo a processo e da computer a computer, tendono ad avere una curva di frequenza simile a quella mostrata in figura
- La curva è generalmente caratterizzata come esponenziale o iperesponenziale, con un numero elevato di burst della CPU brevi e un numero ridotto di burst della CPU lunghi

Processi CPU-bound e I/O-bound

La distribuzione dei CPU burst è stata studiata statisticamente

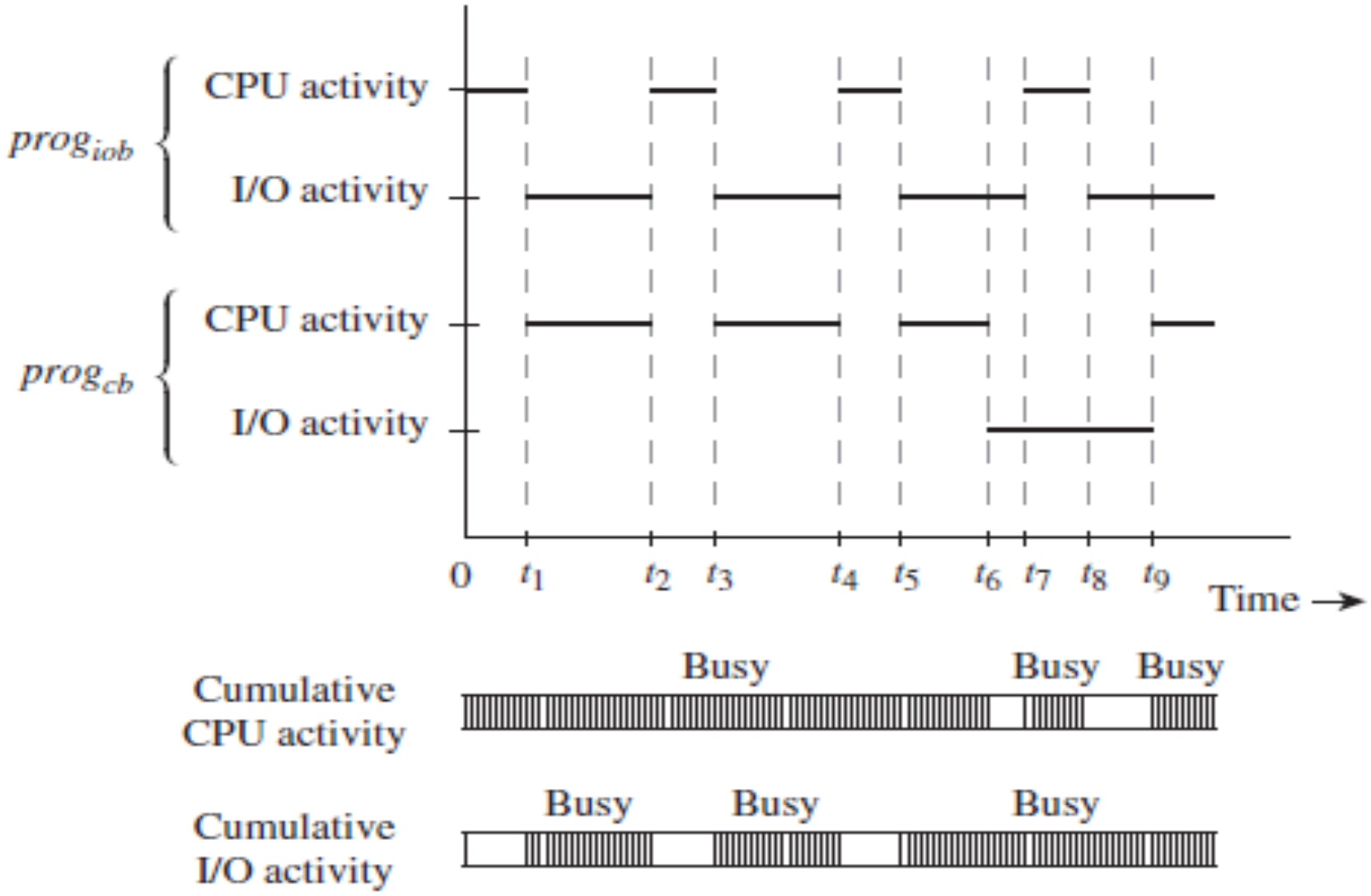


- Processo CPU-bound:
  - effettua una gran quantità di computazioni e pochi I/O, usa la CPU per lunghi intervalli
  - presenta generalmente *pochi burst* di CPU *molto lunghi*
- Processo I/O-bound:
  - effettua poche computazioni ed una grande quantità di I/O, usa la CPU per brevi intervalli
  - presenta generalmente *molti burst* di CPU di *breve durata*

Tecniche di multiprogrammazione

- Il kernel mantiene in memoria un mix appropriato di processi I/O-bound e CPU-bound
- Il kernel effettua tipicamente la seguente gestione:
  - Assegna ad ogni processo una priorità, in modo che processi I/O-bound abbiano priorità più alta rispetto a processi CPU-bound
  - Alloca la CPU al processo con priorità maggiore tra quelli che la possono usare
  - Interrompe il processo in esecuzione se un processo a priorità maggiore è pronto ad usare la CPU
- Il kernel può aggiungere processi CPU-bound o I/O-bound per assicurare un uso efficiente delle risorse

Grafico nel tempo nel caso in cui un programma I/O-bound ha priorità più alta



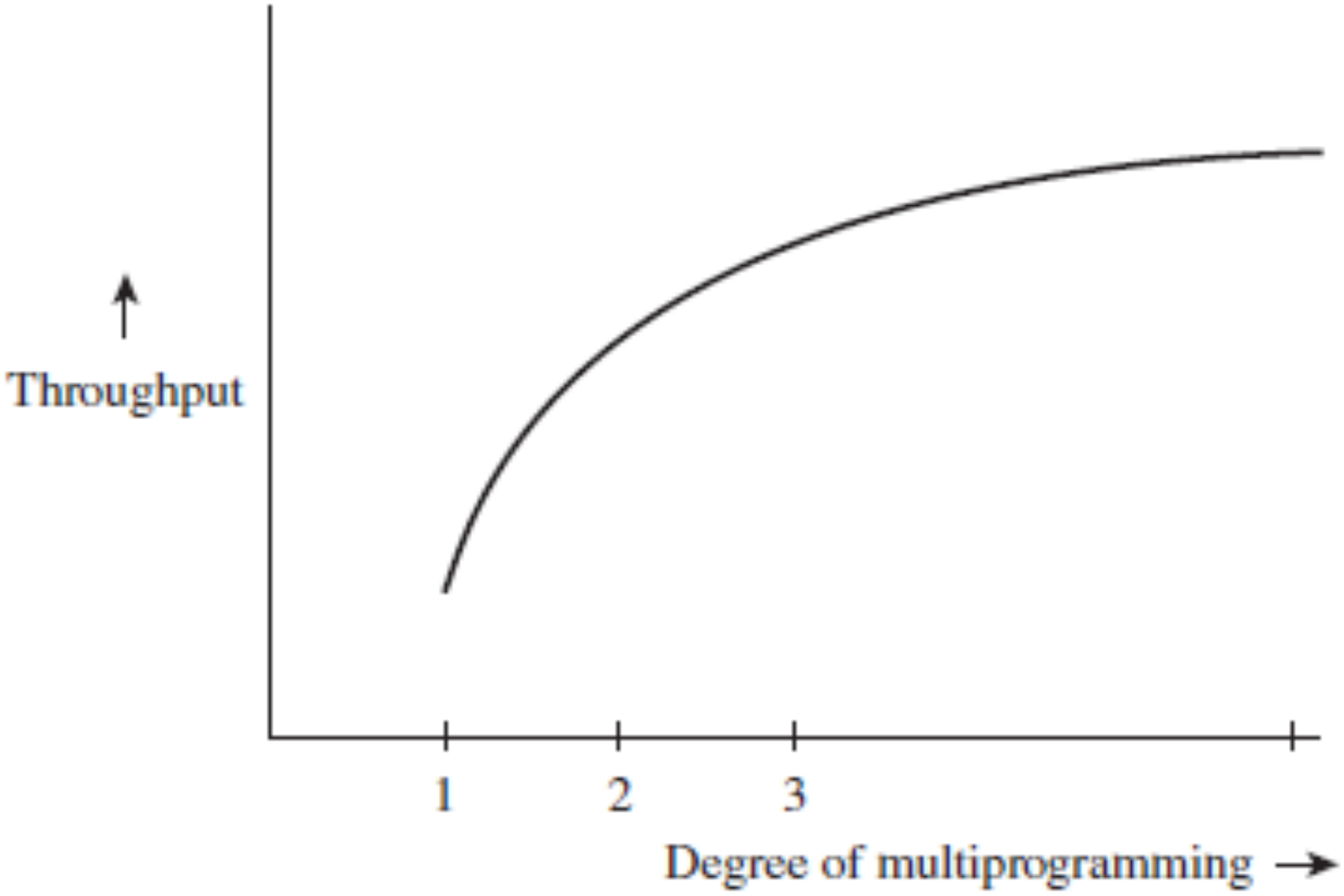
Effetti dell'aumento del grado di multiprogrammazione

- *Azione*: aggiungere un programma $prog_3$ CPU-bound
- *Effetto*:
  - $prog_3$ avrebbe la priorità più bassa, dunque la sua presenza non influirebbe sul progresso di $prog_{cb}$ e $prog_{iob}$
  - La presenza di $prog_3$ aumenterebbe l'utilizzo di tempo di CPU perché permetterebbe di utilizzare tempo di CPU non utilizzato, es. intervalli t_6-t_7 e t_8-t_9
- *Azione (alternativa)*: aggiungere un programma $prog_4$ I/O-bound
- *Effetto*:
  - $prog_4$ avrebbe una priorità compresa tra quella di $prog_{iob}$ e $prog_{cb}$
  - La presenza di $prog_4$ aumenterebbe l'utilizzo dell'I/O
  - Non danneggerebbe il progresso di $prog_{iob}$ poiché $prog_{iob}$ ha la priorità più alta, mentre ridurrebbe il progresso di $prog_{cb}$ solo marginalmente poiché $prog_4$ non utilizza una quantità significativa di tempo di CPU

Variazione del throughput al variare del grado di multiprogrammazione

- Frequenza di completamento (produttività o *throughput*): numero di processi completati da un sistema in una unità di tempo
- Quando si mantiene un appropriato mix di processi, ci si può aspettare che *un aumento del grado di multiprogrammazione risulti in un aumento del throughput*
- Quando il grado di multiprogrammazione è 1 il throughput è determinato dal tempo trascorso dall'unico programma nel sistema
- Quando più programmi sono attivi nel sistema, anche i programmi con priorità più bassa contribuiscono al throughput
  - Tuttavia il loro contributo è limitato dalle loro opportunità di utilizzo della CPU
- Il throughput non varia al crescere del grado di multiprogrammazione se i programmi a bassa priorità non hanno possibilità di essere eseguiti

Variazione del throughput al variare del grado di multiprogrammazione



Scheduler della CPU

- Lo scheduling a breve termine nei sistemi multiprogrammati è la funzionalità che determina quale tra i processi (o thread) che competono per l'uso della CPU otterrà la risorsa
- La politica (o algoritmo) con il quale viene effettuata la scelta è detta politica di scheduling
- La politica di scheduling determina la gestione a breve termine del processore così chiamata per distinguerla da:
  - Gestione a lungo termine che sceglie tra i programmi in memoria secondaria quali caricare in memoria principale regolando così il grado di multiprogrammazione del sistema
  - Gestione a medio termine (swapping) che sceglie i processi parzialmente eseguiti da trasferire temporaneamente in memoria secondaria (e viceversa) con l'obiettivo di ridurre il grado di multiprogrammazione o migliorare il bilanciamento delle tipologie di processi

Scheduler della CPU

- Lo scheduler a breve termine può entrare in azione quando un processo:
  1. passa dallo stato di esecuzione a quello di attesa: richiede I/O, aspetta un segnale, aspetta la terminazione di un figlio, …
  2. passa dallo stato di esecuzione a quello di pronto: si è verificato un interrupt, è scaduto il quanto di tempo, …
  3. passa dallo stato di attesa a quello di pronto: I/O completato, segnale arrivato, figlio terminato, …
  4. passa dallo stato di esecuzione a quello di terminazione: ha finito l'esecuzione
- Uno scheduler che interviene solo nei casi 1 e 4 è *senza prelazione* (*non-preemptive*): un processo rimane in possesso della CPU fino allla sua terminazione oppure fino al passaggio in stato di attesa
- Se lo scheduler interviene anche negli altri casi, allora si dice *con prelazione* (*preemptive*)

Scheduler con prelazione: considerazioni

- Hanno overhead maggiori, perché intervengono più spesso e quindi causano più invocazioni del dispatcher
- Possono provocare race condition quando i dati sono condivisi tra più processi
  - Si consideri il caso in cui due processi condividono dati e mentre un processo sta aggiornando i dati, viene prelazionato in modo da consentire l'esecuzione del secondo processo
  - Il secondo processo può a questo punto tentare di leggere i dati, che però sono stati lasciati in uno stato incoerente da primo processo
- Possono fornire un servizio migliore
  - Sono di fatto usati da tutti i moderni SO

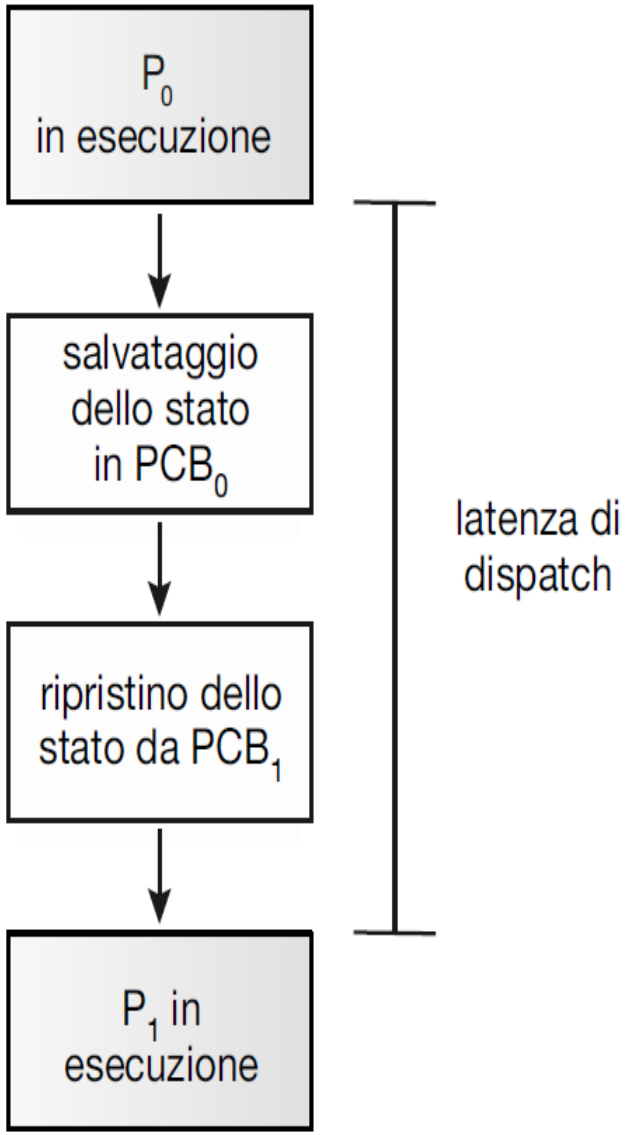
Dispatcher

- Il dispatcher è un modulo del SO che fornisce il controllo della CPU al processo scelto dallo scheduler a breve termine (il dispatcher implementa il "meccanismo")
- Tale funzione comporta:
  - il context switching
  - il passaggio (eventuale) della CPU al modo utente
  - il salto all'istruzione corretta da cui (ri)cominciare l'esecuzione del processo scelto dallo scheduler
- Il dispatcher dovrebbe essere il più veloce possibile, poiché viene richiamato ad ogni cambio di contesto

Latenza di dispatch

Intervallo di tempo necessario al dispatcher per arrestare un processo e avviare l'esecuzione di un altro

È importante minimizzare la latenza di dispatch



Scheduling della CPU

- Concetti fondamentali
- Criteri di scheduling
- Algoritmi di scheduling
- Scheduling dei thread
- Scheduling per sistemi multiprocessore
- Valutazione degli algoritmi di scheduling

Scelta dell'algoritmo

- Algoritmi di scheduling della CPU diversi hanno proprietà diverse e la scelta di un particolare algoritmo può favorire una classe di processi rispetto a un'altra
- Le caratteristiche utilizzate per il confronto possono fare una differenza sostanziale riguardo quale algoritmo viene giudicato migliore

Alcune caratteristiche

- Utilizzo della CPU: percentuale di utilizzo della CPU
- Frequenza di completamento (*throughput*): numero di processi completati dal sistema nell'unità di tempo
- Tempo di completamento *(turnaround time)*: tempo trascorso dall'ingresso di un processo nel sistema fino al completamento della sua esecuzione
- Tempo di attesa: somma degli intervalli di tempo che un processo passa in attesa nella coda dei pronti
- Tempo di risposta: tempo trascorso dall'invio al sistema di una richiesta da parte di un processo fino all'inizio della (prima) risposta (importante in sistemi time-sharing)

Misure dell'efficienza, delle prestazioni del sistema e del servizio per l'utente

Aspetto	Misura	Descrizione
Efficienza d'uso	CPU	Percentuale di utilizzo della CPU
	Memoria	Percentuale di utilizzo della memoria
Prestazioni del sistema	Frequenza di completamento	Quantità di lavoro svolto nell'unità di tempo
Servizio per l'utente	Tempo di completamento	Tempo necessario per completare l'esecuzione di un processo
	Tempo di risposta	Tempo necessario per rispondere ad una richiesta

Criteri di scheduling

Ci sono vari criteri in base ai quali un algoritmo di scheduling può operare

- Massimizzare l'utilizzo della CPU
  (di solito, l'utilizzo varia fra il 40% ed il 90%)
- Massimizzare la frequenza di completamento
- Minimizzare il tempo di completamento
- Minimizzare il tempo di attesa
- Minimizzare il tempo di risposta
- Minimizzare la varianza del tempo di risposta
- Ottimizzare il valore medio di una data caratteristica

Non esiste un algoritmo ottimo in base a tutti i criteri !

Scheduling della CPU

- Concetti fondamentali
- Criteri di scheduling
- Algoritmi di scheduling
- Scheduling dei thread
- Scheduling per sistemi multiprocessore
- Valutazione degli algoritmi di scheduling

Algoritmi di scheduling

- Si possono classificare in base a 3 aspetti:
  - senza/con prelazione
  - senza/con priorità
  - (se con priorità) statiche/dinamiche
- In pratica i SO utilizzano una combinazione dei 3 aspetti
  - Es. UNIX (time-sharing): prelazione e priorità dinamiche

Algoritmi di scheduling: prelazione

- Senza prelazione
  - Si basano sul rilascio spontaneo della CPU da parte del processo in esecuzione
  - Adatti per elaborazioni orientate all'uso intensivo della CPU
- Con prelazione
  - All'occorrenza, sono in grado di forzare l'interruzione del processo in esecuzione
  - Soddisfano esigenze di priorità o di equità di ripartizione delle risorse
  - Evitano il monopolio della CPU da parte di un processo CPU-bound

Algoritmi di scheduling: priorità

- Senza priorità
  - Considerano i processi equivalenti sul piano dell'urgenza di esecuzione
  - Si basano normalmente su strategie di ordinamento First Come First Served (FCFS)
- Con priorità
  - Dividono i processi in classi secondo l'importanza o la criticità rispetto al tempo di esecuzione
  - Necessari nei sistemi interattivi o con proprietà real-time

Algoritmi di scheduling: statiche/dinamiche

- Statiche
  - Un processo conserva nel tempo i suoi diritti di accesso alla CPU (priorità)
  - Penalizzano i processi a bassa priorità (ciò può comportare inedia o *starvation*)
- Dinamiche
  - I diritti dei processi sono modificati nel tempo sulla base del loro comportamento passato o estrapolando quello futuro
  - Bilanciano le esigenze tra processi CPU-bound e processi I/O-bound

Algoritmi di scheduling

- Scheduling First-Come, First-Served (FCFS)
- Scheduling Shortest-Job-First (SJF) e Shortest-Remaining-Time-First (SRTF)
- Scheduling circolare (Round-Robin, RR)
- Scheduling con priorità
- Scheduling a code multilivello
- Scheduling a code multilivello con retroazione

Assunzioni

- Descriveremo il funzionamento di questi algoritmi nel caso di un sistema con una singola CPU
  - Discuteremo successimente lo scheduling nei sistemi multicore
- Negli esempi che esamineremo, supporremo
  - che ciascun processo esegua solo una sequenza di operazioni della CPU, cioè solo un burst di CPU, senza alternare tra uso delle CPU e dei dispositivi di I/O, e
  - che la durata di tale burst sia espressa in millisecondi
- La caratteristica che adotteremo per confrontare gli algoritmi sarà il tempo di attesa medio

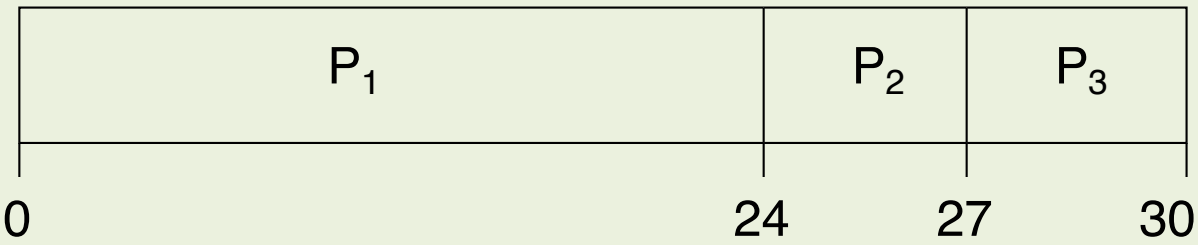
Scheduling First-Come First-Served (FCFS)

- I processi ottengono la CPU nello stesso ordine con cui diventano pronti
- Algoritmo senza prelazione e senza priorità

- Esempio:

Processo	Tempo di Burst
P_1	24
P_2	3
P_3	3

- Supponiamo che i processi arrivino al tempo 0 nell'ordine: P_1 , P_2 , P_3
  Il diagramma di Gantt per lo scheduling è:



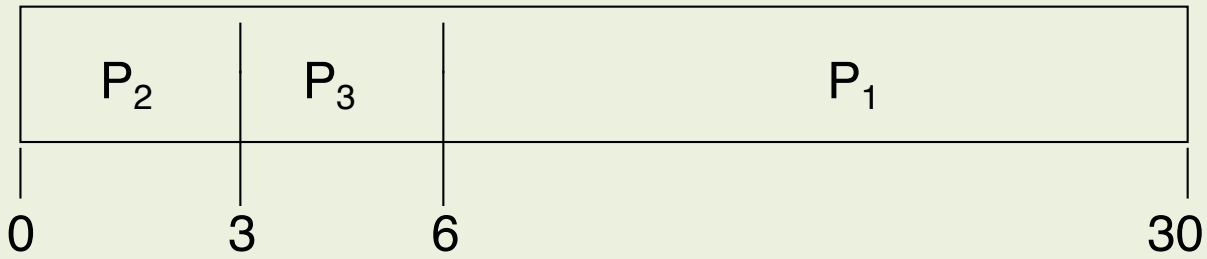
- Tempi di attesa: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Media dei tempi di attesa : (0 + 24 + 27)/3 = 17

Esempio di scheduling FCFS

Supponiamo ora che gli stessi processi arrivino al tempo 0 nell'ordine:

$$P_2 , P_3 , P_1$$

- IL diagramma di Gantt per lo scheduling è:



- Tempi di attesa: $P_1 = 6; P_2 = 0; P_3 = 3$
- Media dei tempi di attesa: $(6 + 0 + 3)/3 = 3$
- Molto meglio che nel caso precedente, pur avendo gli stessi processi!

- Effetto Convoglio: processi brevi possono dover attendere che processi lunghi rilascino la CPU, il che può causare una riduzione dell'utilizzo dei dispositivi di I/O (ma anche della CPU)
- Da ciò si evince che esistono algoritmi migliori di FCFS (anche per sistemi orientati all'uso intensivo della CPU)

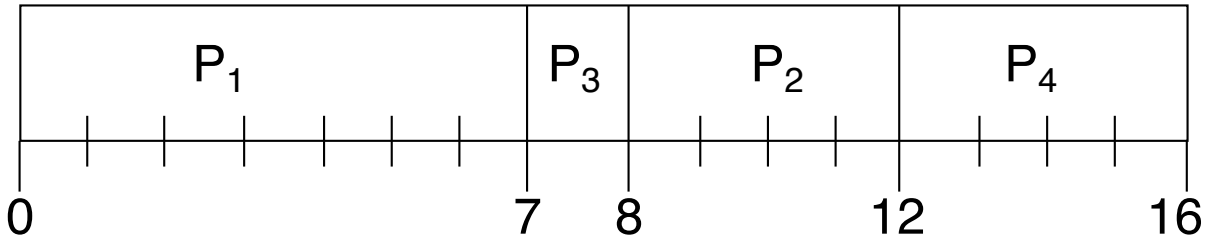
Scheduling Shortest-Job-First (SJF) & Shortest-Remaining-Time-First (SRTF)

- SJF assegna la CPU al processo con burst di CPU più breve
- La priorità dei processi è data dalla lunghezza (stimata) del loro prossimo burst di CPU
- Due varianti:
  - SJF (senza prelazione): un processo, una volta che ottiene la CPU, non può essere interrotto finché non completa il burst di CPU
  - SRTF (con prelazione): se diventa pronto un processo con lunghezza del burst di CPU minore del tempo restante di burst di CPU del processo in esecuzione, esso ha precedenza rispetto al processo in esecuzione ed ottiene la CPU
- SJF è l'algorimo che fornisce il minimo tempo di attesa medio per un dato insieme di processi (presenti in un certo momento nel sistema)
- SRTF può comportare *inedia (starvation)* di processi con CPU burst lunghi

Esempio di scheduling SJF

Processo	Tempo di arrivo	Tempo di Burst
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF senza prelazione



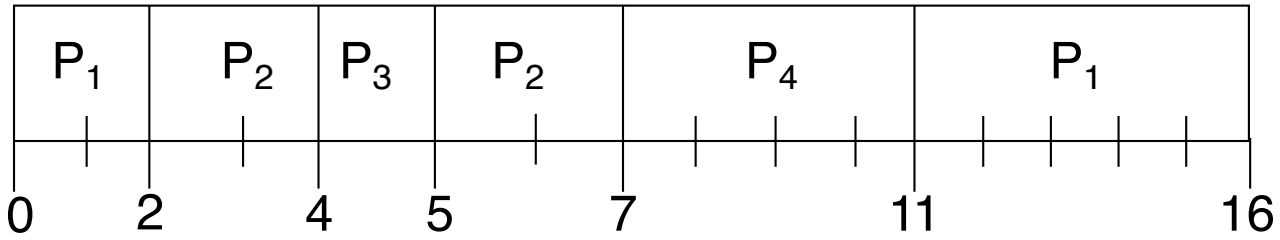
- Tempo di attesa medio = (0 + (8-2) + (7-4) + (12-5))/4 = (0 + 6 + 3 + 7)/4 = 4

tempo di attesa di un processo = tempo inizio esecuzione - tempo arrivo nel sistema

Esempio di scheduling SRTF

Processo	Tempo di arrivo	Tempo di Burst
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SRTF (SJF preemptive)



- Tempo di attesa medio = ((16-7)+(7-4-2)+(5-1-4)+(11-4-5))/4 = (9 + 1 + 0 +2)/4 = 3

tempo di attesa = (tempo fine esecuzione – tempo di burst) - tempo arrivo

Stima del burst di CPU successivo

- SJF/SRTF sono algoritmi ottimali, ma ideali
- Problema: non possiamo conoscere il futuro, quindi la lunghezza del successivo burst di CPU di un processo può essere solo stimata (per esempio, usando la lunghezza dei burst di CPU precedenti)
- Idea: il comportamento dei processi tende ad essere consistente nel tempo, quindi ci basiamo sul passato per predirne il comportamento futuro
- La lunghezza del successivo burst di CPU di un processo generalmente si stima calcolando la media esponenziale delle lunghezze misurate dei precedenti burst di CPU del processo

Media esponenziale

La formula utilizzata per il calcolo è:

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

- τ_{n+1} è il valore stimato per il successivo CPU burst
- τ_n è la stima precedente (contiene la storia passata)
- t_n è il valore misurato dell'n-esimo CPU burst (contiene le informazioni più recenti)
- α è un numero reale compreso tra 0 ed 1 (regola il peso delle informazioni recenti e della storia passata)

Media esponenziale

- Se sviluppiamo la formula in modo da esprimerla in termini di valori di CPU burst effettivamente misurati otteniamo:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\alpha\, t_{n-1} + \ldots + (1-\alpha)^j \alpha\, t_{n-j} + \ldots + (1-\alpha)^{n+1} t_0$$

$$(1-\alpha)\,\tau_n$$

- Poiché sia α che $(1-\alpha)$ sono minori o uguali a 1, nella somma a destra di = ciascun termine successivo ha un peso inferiore rispetto a quello precedente

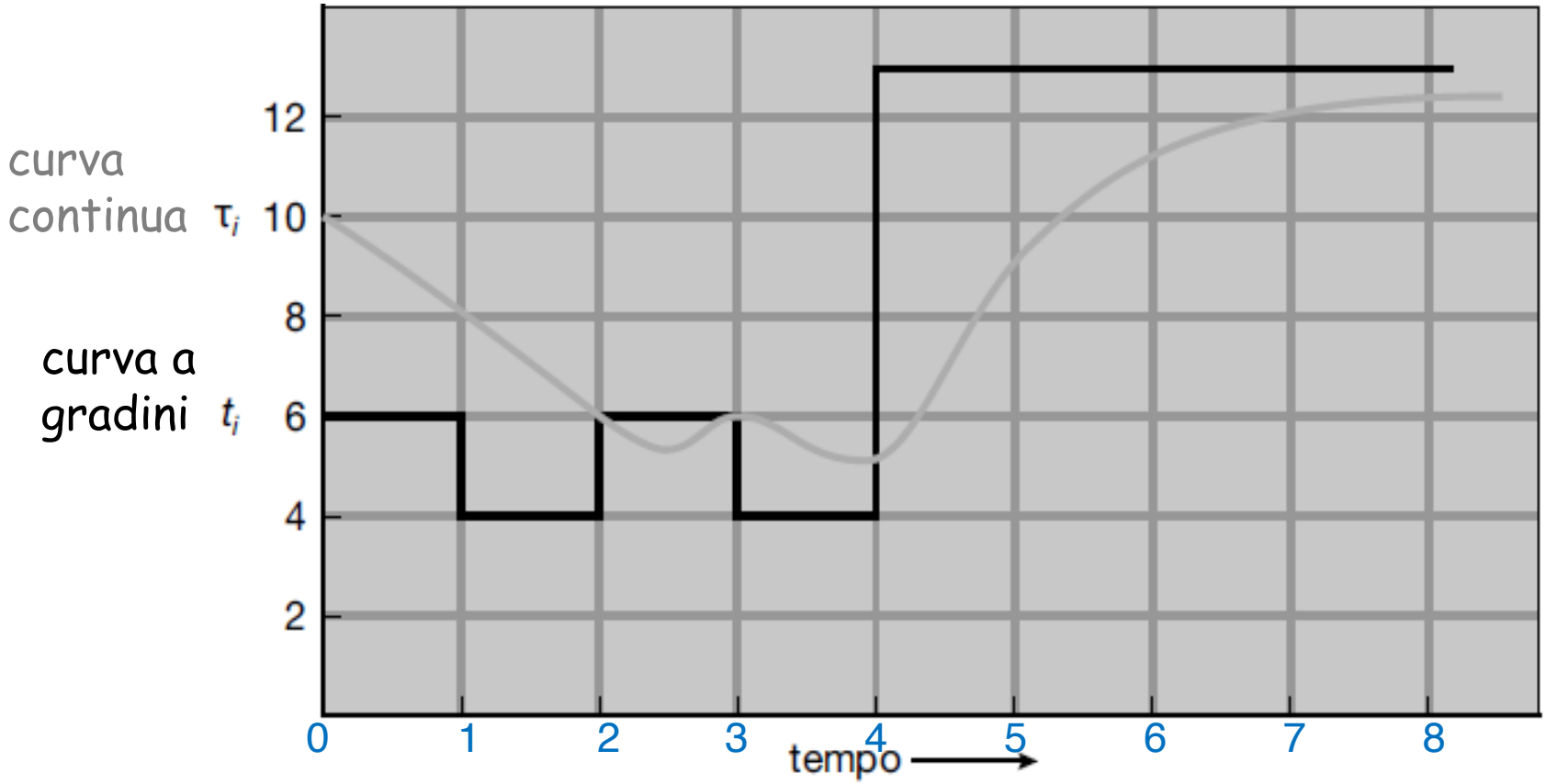
Esempi di media esponenziale

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

- τ_n contiene la storia passata
  t_n è il valore misurato dell'n-esimo CPU burst
  (contiene le informazioni più recenti)
  α è numero reale compreso tra 0 ed 1
  (regola il peso delle informazioni recenti e della storia passata)

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - le informazioni recenti (cioè l'ultimo burst effettivo t_n) non contano
  - si suppone cioè che le condizioni attuali siano transitorie
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - conta solo l'ultimo burst effettivo
  - si suppone cioè che la storia sia vecchia e irrilevante
- Il valore più comune per α è 0.5
  - in questo caso la formula diventa $\tau_{n+1} = 0.5 * (\tau_n + t_n)$

Stima della lunghezza del burst di CPU successivo (α = 0.5)



indice (i)	0	1	2	3	4	5	6	7	8	9
CPU burst (t_i)	6	4	6	4	13	13	13	13	13	...
Predizione (τi)	10	8	6	6	5	9	11	12	12,5	...

Scheduling circolare (o Round Robin, RR)

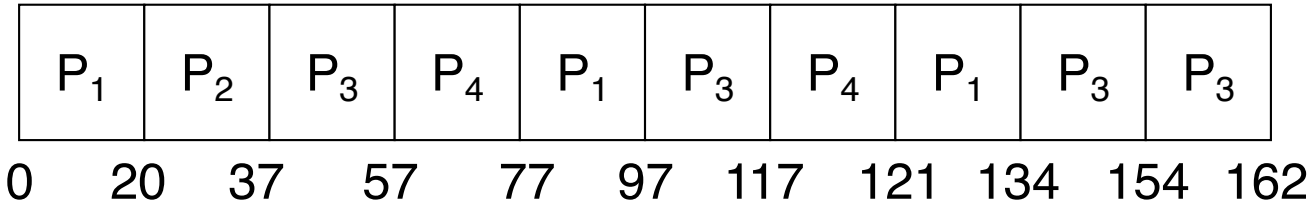
- A turno ogni processo ottiene la CPU per una piccola quantità di tempo (quanto di tempo o *time slice*), tipicamente tra 10 e 100 millisecondi

  Terminato questo tempo, il processo è interrotto e inserito nella coda dei pronti (in fondo)
- Simile a FCFS, ma con prelazione
- Progettato appositamente per sistemi *time-sharing*
- Se ci sono n processi nella coda dei pronti e il quanto di tempo è q, ogni processo ottiene $1/n$ del tempo di CPU in frazioni di al più q unità di tempo per volta
  - Nessun processo attende più di $(n-1)q$ unità di tempo (c'è quindi un limite al tempo di attesa della CPU)

Scheduling RR con quanto di tempo q = 20

Processo	Tempo di Burst
P_1	53
P_2	17
P_3	68
P_4	24

- IL diagramma di Gantt è:



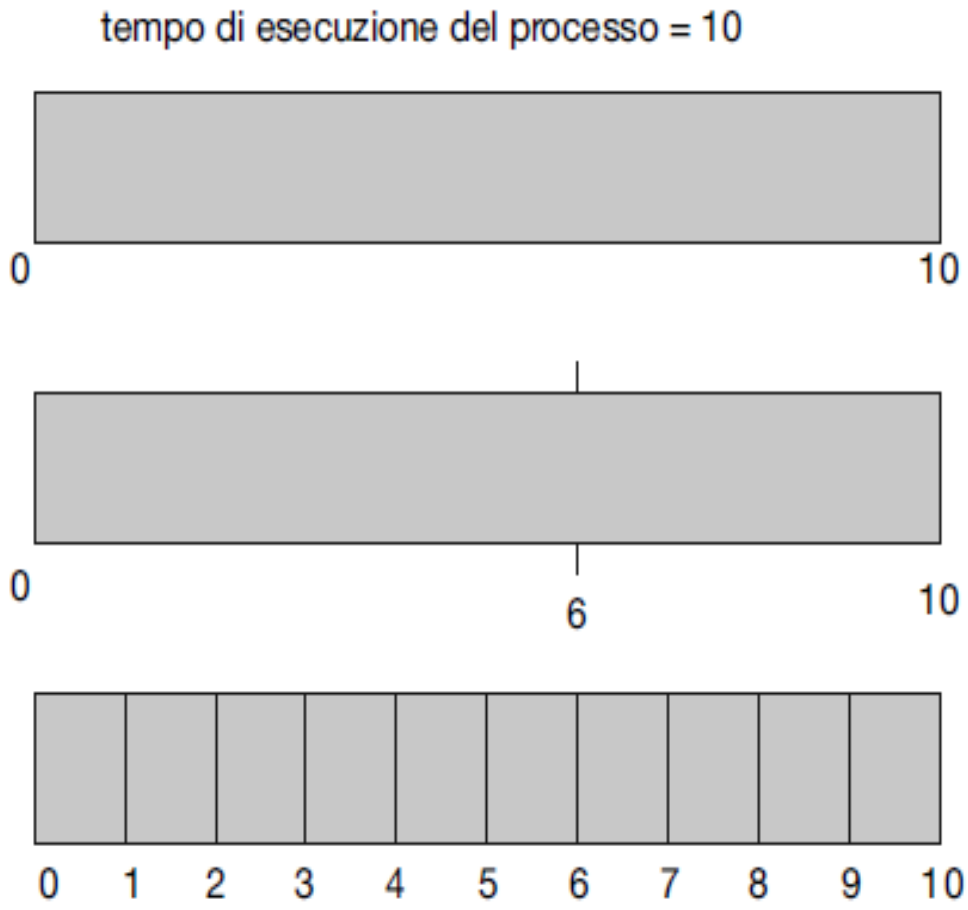
- Tipicamente, il tempo di completamento è più alto rispetto a SJF, ma è migliore il tempo di risposta

Scheduling circolare: prestazioni

- Dipendono dalla lunghezza del quanto di tempo
  - q grande $\Rightarrow$ si avvicina a FCFS
  - q piccolo $\Rightarrow$ produce un maggiore parallelismo virtuale
- q deve comunque essere grande rispetto al tempo di context switch, altrimenti il numero di context switch sarebbe eccessivo, così come il relativo overhead

Quanto di tempo e context switch

Supponiamo di dover eseguire un processo la cui lunghezza di CPU-burst è 10
Più è piccolo il quanto di tempo, maggiore è il numero di context switch

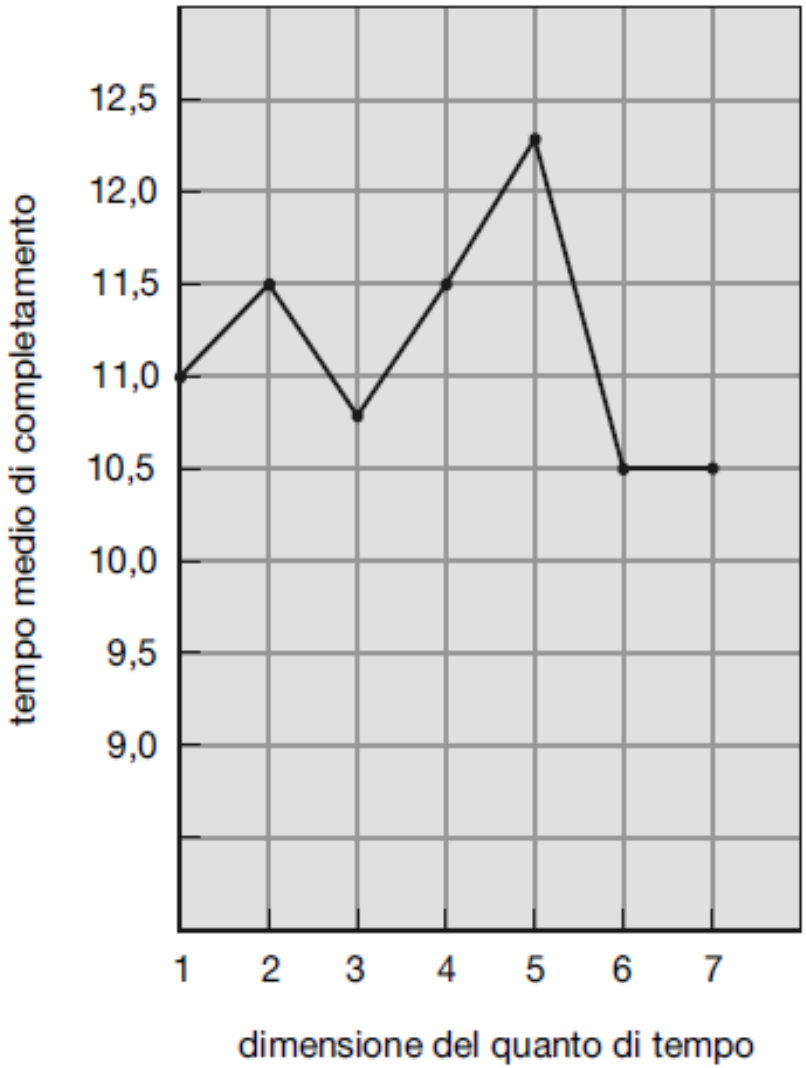


lunghezza del quanto di tempo	cambio di contesto
12	0
6	1
1	9

Scheduling circolare: tempo di completamento

- Dipende anch'esso dalla lunghezza del quanto di tempo
  - Il tempo di completamento medio di un insieme di processi non migliora necessariamente con l'aumento della lunghezza del quanto di tempo

Variazione del tempo di completamento medio in funzione del quanto di tempo



processo	tempo
P_1	6
P_2	3
P_3	1
P_4	7

q=1 t=44/4=11
P1 6+3+1+5=15
P2 3+3+1+2= 9
P3 1+1+1= 3
P4 6+3+1+7=17

q=5 t=49/4=12,25
P1 6+3+1+5=15
P2 5+3= 8
P3 5+3+1= 9
P4 6+3+1+7=17

q=6 t=42/4=10,5
P1 6=6
P2 6+3= 9
P3 6+3+1=10
P4 6+3+1+7=17

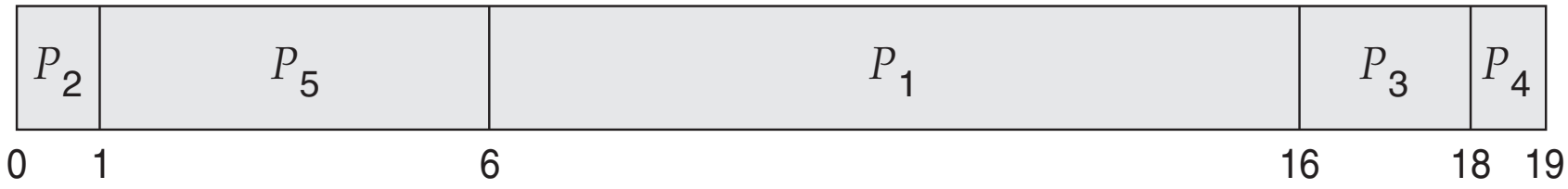
Scheduling a priorità

- Assegna la CPU al processo con priorità più alta
- Priorità: un numero (intero) associato ad ogni processo
  (tipicamente, ma non sempre, intero più piccolo significa priorità più alta)
- Le priorità possono essere definite
  - internamente: in base a parametri misurati dal sistema sul processo, es. tempo di CPU già impiegato, file aperti, memoria utilizzata, uso di I/O, …
  - esternamente: importanza del processo, dell'utente proprietario, dei soldi pagati, …
- Due possibili schemi: con prelazione e senza prelazione
- SJF è un esempio di scheduling a priorità e senza prelazione in cui la priorità è data dalla lunghezza del successivo tempo di burst di CPU

Scheduling a priorità

Processo	Tempo di Burst	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Il diagramma di Gantt è:



Scheduling a priorità: considerazioni

- Problema: inedia *(starvation)* – processi a bassa priorità non sono mai eseguiti
- Soluzione: invecchiamento (*aging*) – aumento graduale della priorità di un processo man mano che aumenta il suo tempo di attesa (priorità dinamiche)

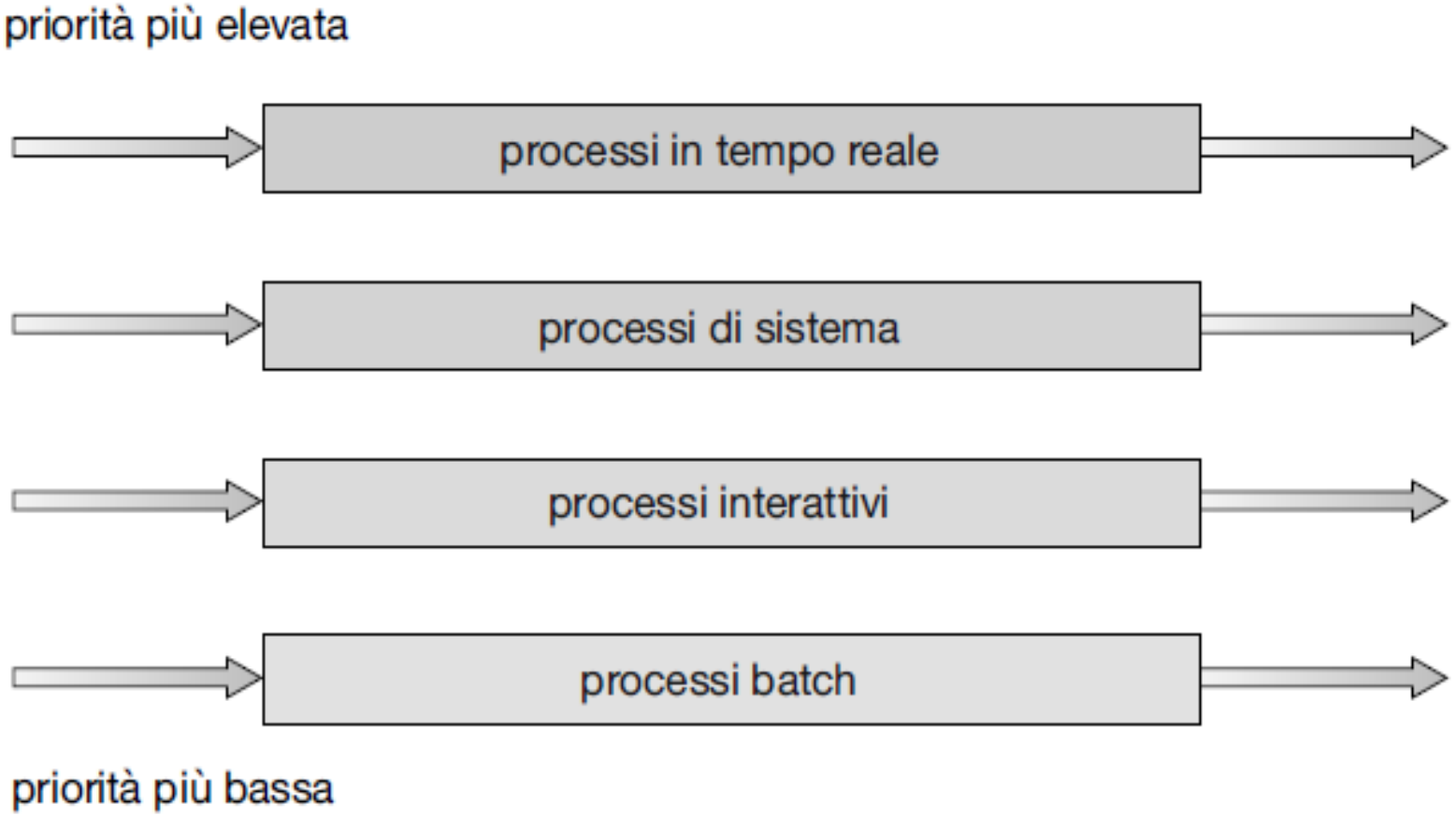
Scheduling a code multilivello

- Supponiamo di poter classificare i processi in gruppi sulla base della loro natura (es. I/O-bound e CPU-bound)
  - I requisiti sono diversi, quindi perché usare un unico algoritmo di scheduling?
- Soluzione: dividiamo la coda dei pronti in code separate; es.
  - *foreground* (processi I/O-bound)
  - *background* (processi CPU-bound)

  ognuna con il proprio algoritmo di scheduling; es.
  - foreground: RR
  - background: FCFS
- L'idea può essere ulteriormente generalizzata per avere sistemi con un numero maggiore di code

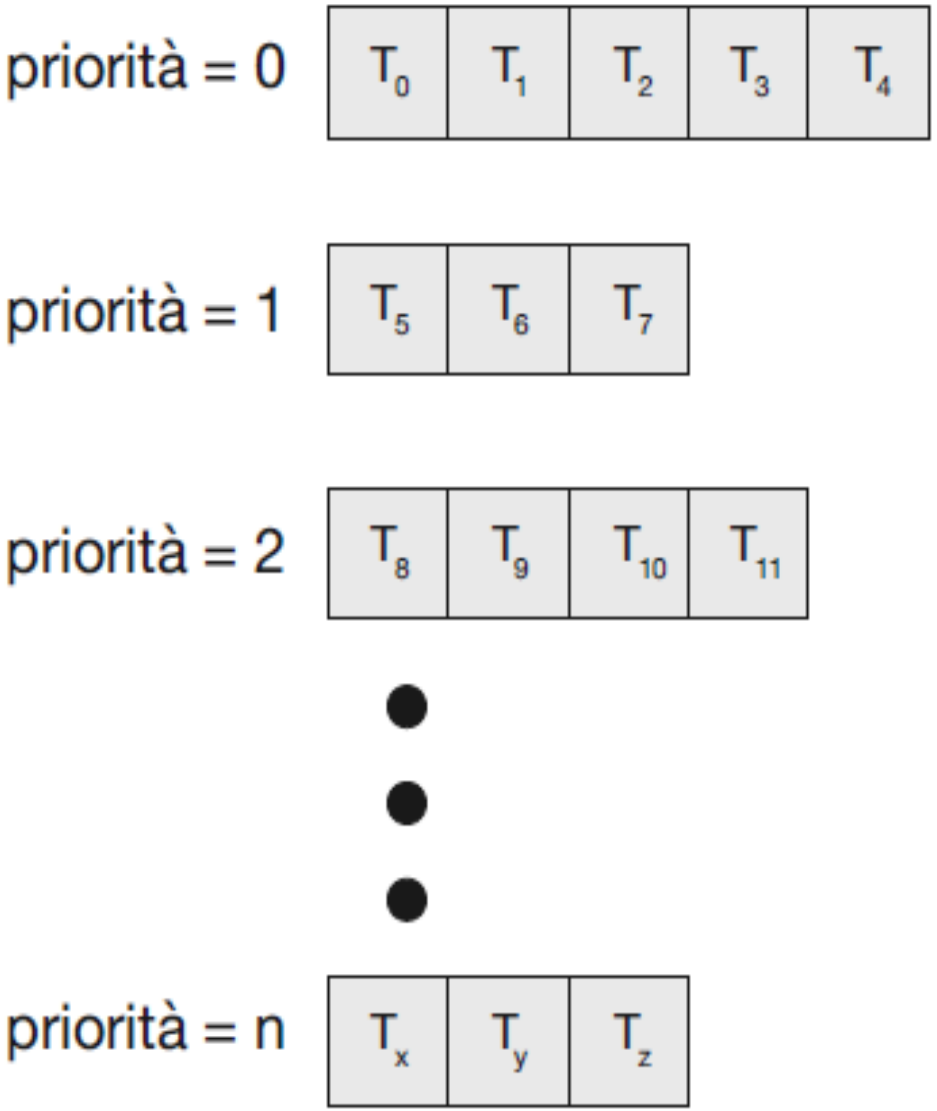
Scheduling a code multilivello

Code separate per ciascuna tipologia differente di processo, con relative priorità



Scheduling a code multilivello

Può essere utilizzato anche per suddividere i processi su più code separate per ciascuna priorità distinta (senza considerare il tipo) e quindi selezionare, di volta in volta, il processo nella coda con priorità più alta



Scheduling a code multilivello

Occorre gestire lo scheduling fra le code

- Scheduling a priorità fissa: per es., serve prima tutti i processi della coda di foreground quindi quelli della coda di background (possibilità di starvation)
- Partizione di tempo: ogni coda ottiene la CPU per un certo tempo che può suddividere fra i suoi processi; per es.,
  - 80% del tempo alla coda di foreground, gestita con politica RR
  - 20% del tempo alla coda di background, gestita con politica FCFS

Scheduling a code multilivello con retroazione (feedback)

- Diversamente dallo sheduling con code multilivello basate sulla tipologia dei processi, *un processo può essere spostato tra le varie code*; così si può implementare l'invecchiamento dei processi
  - Es., si può spostare in una coda a priorità più elevata un processo che attende troppo a lungo in una coda a priorità bassa
- Uno scheduler a code multilivello con retroazione è caratterizzato dai seguenti parametri:
  - numero di code
  - algoritmo di scheduling per ogni coda
  - metodo usato per determinare quando spostare un processo in una coda a priorità maggiore
  - metodo usato per determinare quando spostare un processo in una coda a priorità minore
  - metodo usato per determinare in quale coda deve essere posto un processo nel momento in cui richiede un servizio

Scheduling a code multilivello con retroazione (feedback)

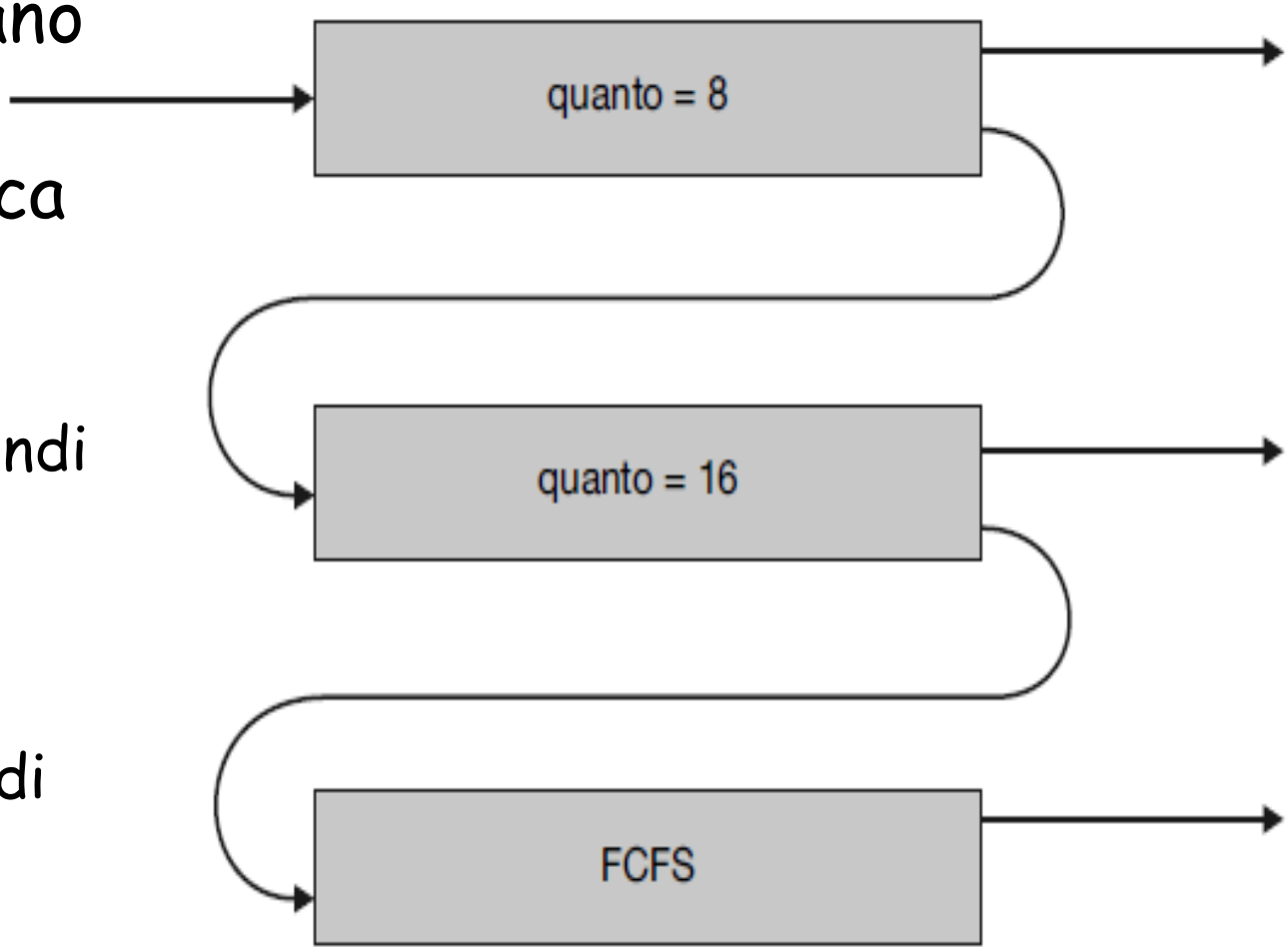
- È il criterio di scheduling della CPU più generale, ma corrisponde anche all'algoritmo più complesso, richiede infatti metodi particolari per la selezione dei valori dei diversi parametri
- Per esempio
  - Se un processo viene interrotto perché scade il suo quanto di tempo, può venire passato su una coda a priorità minore
  - Se un processo si sospende prima dello scadere del suo quanto di tempo, quando ridiventa pronto può essere passato ad una coda a priorità maggiore
- In questo modo si favoriscono i processi I/O-bound mentre i processi CPU-bound possono essere soggetti a *starvation*

  Per evitare il problema si può intervenire sulla durata del quanto di tempo: i processi CPU-bound hanno priorità più bassa, ma il loro quanto di tempo è più lungo

Scheduling a code multilivello con retroazione (feedback)

Supponiamo ci siano 3 code tutte gestite con politica RR:

- Q_0 con quanto di tempo 8 millisecondi
- Q_1 con quanto di tempo 16 millisecondi
- Q_2 senza quanto di tempo



Scheduling a code multilivello con retroazione (feedback)

- Scheduling interno alle code
  - Quando ottiene la CPU, un processo in Q_0 mantiene la CPU per al più 8 millisecondi

    Se non finisce in 8 millisecondi, il processo è interrotto e spostato in Q_1
  - Quando ottiene la CPU, un processo in Q_1 mantiene la CPU per al più 16 millisecondi

    Se non finisce in 16 millisecondi, il processo è interrotto e spostato in Q_2
  - Una volta che la CPU è assegnata ad un processo in Q_2, essa non può più essere prelazionata
- Scheduling tra le code (automatico)
  - Vengono prima eseguiti tutti i processi presenti nella coda Q_0; quando la coda Q_0 è vuota, si eseguono i processi nella coda Q_1
  - Analogamente, i processi nella coda Q_2 vengono eseguiti solo se le code Q_0 e Q_1 sono vuote
- Un processo nuovo è inserito nella coda Q_0; l'ingresso di un processo nella coda Q_0 prelaziona i processi della coda Q_1
  - Analogamente, l'ingresso di un processo nella coda Q_1 prelaziona i processi della coda Q_2
- Per evitare starvation, un processo che attende da troppo tempo in una coda a priorità bassa viene gradualmente spostato in una coda a priorità più elevata

Scheduling della CPU

- Concetti fondamentali
- Criteri di scheduling
- Algoritmi di scheduling
- Scheduling dei thread
- Scheduling per sistemi multiprocessore
- Valutazione degli algoritmi di scheduling

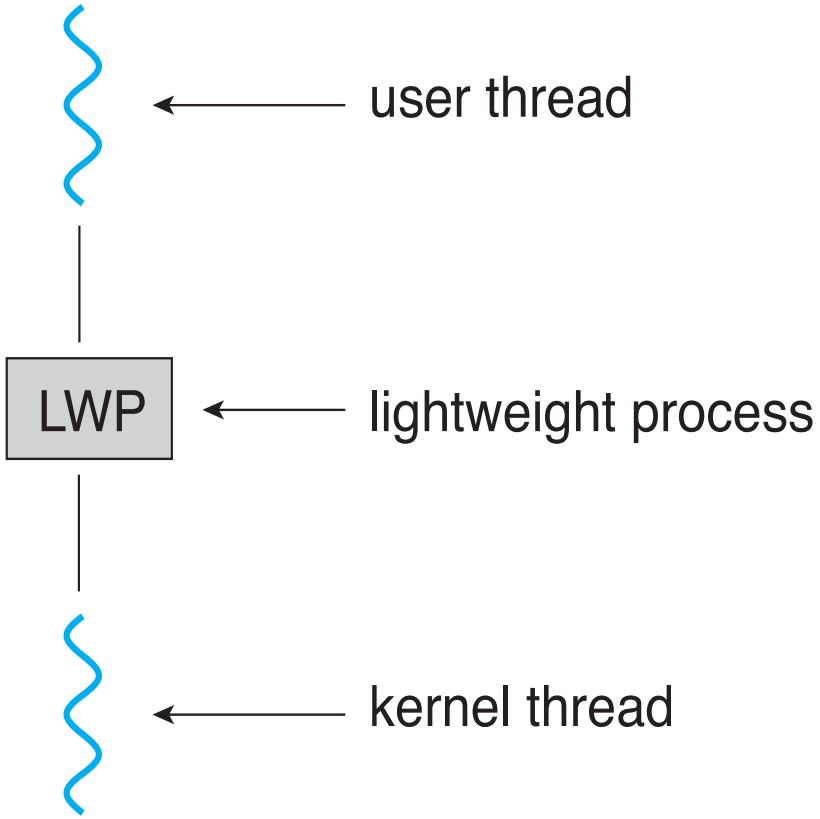
Scheduling dei thread

- Nella maggior parte dei SO moderni sono i thread a livello kernel, non i processi, ad essere soggetti a scheduling da parte del SO
- I thread a livello utente sono gestiti da una libreria di thread e il kernel non è consapevole della loro esistenza
  - Per essere eseguiti, devono quindi essere associati ad un thread a livello kernel
  - Questa associazione può essere fatta indirettamente, utilizzando una struttura dati intermedia chiamata *LightWeight Process* (LWP)

LightWeight Process (LWP)

Un LightWeight Process è una sorta di processore virtuale che permette la comunicazione tra il kernel e la libreria di thread a livello utente

- Il kernel fornisce a ogni applicazione uno o più LWP, ciascuno associato a un thread del kernel
- Il SO esegue lo scheduling dei thread del kernel sui processori fisici
- L'applicazione esegue lo scheduling dei thread utente sui LWP disponibili
- Se un thread del kernel si blocca, il LWP associato si blocca, così come il thread a livello utente associato
- Tramite una procedura nota come upcall, il kernel informa l'applicazione del verificarsi di determinati eventi
- Le upcall sono gestite dalla libreria di thread a livello utente mediante un apposito gestore eseguito su un LWP assegnato all'applicazione



Scheduling dei thread

Livello kernel

- Il kernel sceglie, con una delle politiche viste prima, il thread a livello kernel, e quindi il LWP, a cui assegnare la CPU
- Questo schema è noto come *ambito della contesa allargato al sistema* (*system-contention scope*, SCS) poiché la competizione avviene tra tutti i thread nel sistema
- Nei sistemi che implementano il modello uno-a-uno, come Windows e Linux, SCS è anche l'unica forma di scheduling dei thread

Livello utente

- Nei sistemi che implementano i modelli molti-a-uno e molti-a-molti, la libreria di thread schedula i thread a livello utente per l'esecuzione su un LWP disponibile
- Questo schema è noto come *ambito della contesa ristretto al processo* (*process-contention scope*, PCS) poiché la competizione avviene tra thread appartenenti allo stesso processo
- Generalmente lo scheduling PCS è a priorità e le priorità dei thread sono decise dal programmatore
- Inoltre, lo schema è solitamente con prelazione del thread in esecuzione, a vantaggio di thread a priorità più alta (in caso di uguale priorità, non c'è garanzia di suddivisione di tempo)

Scheduling della CPU

- Concetti fondamentali
- Criteri di scheduling
- Algoritmi di scheduling
- Scheduling dei thread
- Scheduling per sistemi multiprocessore
- Valutazione degli algoritmi di scheduling

Scheduling per sistemi multiprocessore

- Se sono disponibili più processori (unità di elaborazione) allora è possible bilanciare il carico dell'esecuzione (*load balancing*) ma il problema dello scheduling è più complesso
- Tradizionalmente il termine multiprocessore faceva riferimento a sistemi equipaggiati con più processori fisici, ciascuno contenente una CPU single-core
- Attualmente il termine si applica alle seguenti architetture
  - Processori multicore
  - Core multithread
  - Sistemi NUMA (Accesso Non Uniforme alla Memoria)
  - Sistemi multiprocessore eterogenei
- Nei primi tre casi, i processori sono identici in termini di funzionalità
  - Qualsiasi processore può eseguire qualsiasi processo in coda ready
- Nell'ultimo caso, i processori hanno funzionalità diverse tra loro

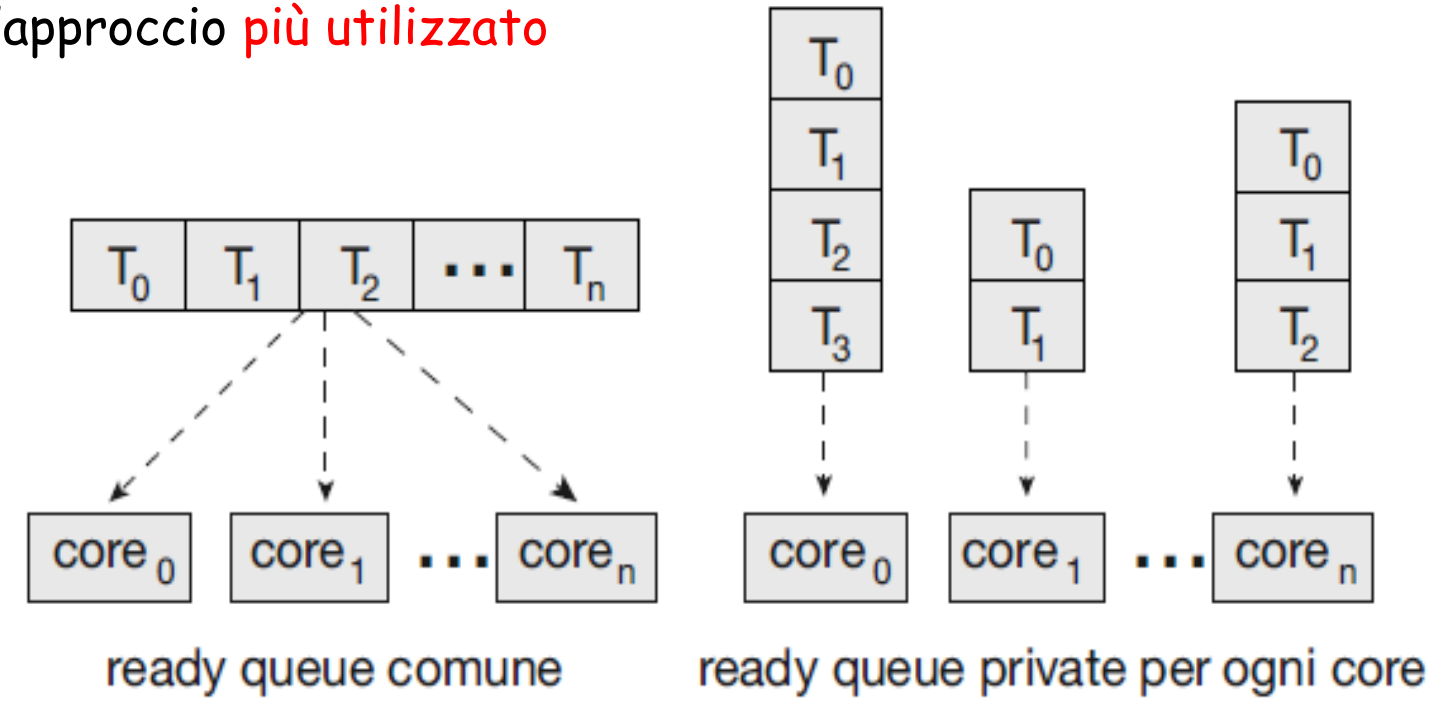
Scheduling per sistemi multiprocessore: approcci possibili

- *Multieleborazione asimmetrica*
  - Un processore assume il ruolo di unità di elaborazione centrale (*master server*): prende le decisioni di scheduling, effettua l'elaborazione delle operazioni di I/O e le altre attività di sistema
  - Gli altri processori eseguono solo programmi utente
  - Semplifica il problema dello scheduling perché solo il master server accede alle strutture dati del sistema, ma diminuisce le prestazioni perché il master server può diventare un collo di bottiglia
- *Multieleborazione simmetrica*
  - Ogni processore ha uno scheduler che esamina la coda ready e sceglie il thread da eseguire
  - È l'approccio standard per supportare i sistemi multiprocessore, utilizzato da tutti i SO moderni

Multieleborazione simmetrica (SMP)

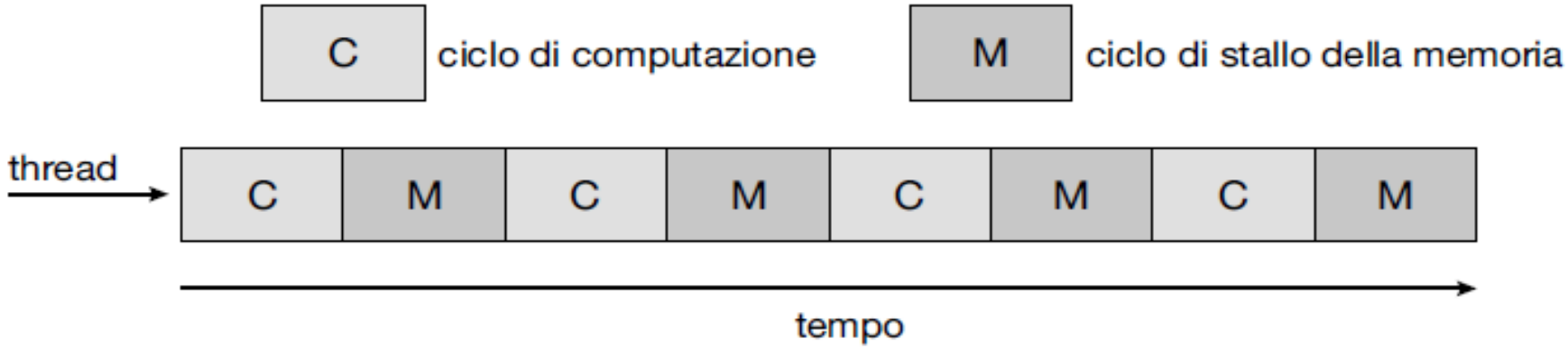
Strategie per organizzare i thread pronti per l'esecuzione:

1. Tutti i thread selezionabili sono in una coda ready comune
   - Richiede meccanismi di sincronizzazione nel kernel per regolare l'accesso alla coda ready condivisa ed evitare race condition, e quindi può creare problemi di performance
2. Ogni processore ha una propria coda ready privata di thread pronti
   - Può necessitare di algoritmi di bilanciamento del carico
   - È l'approccio più utilizzato



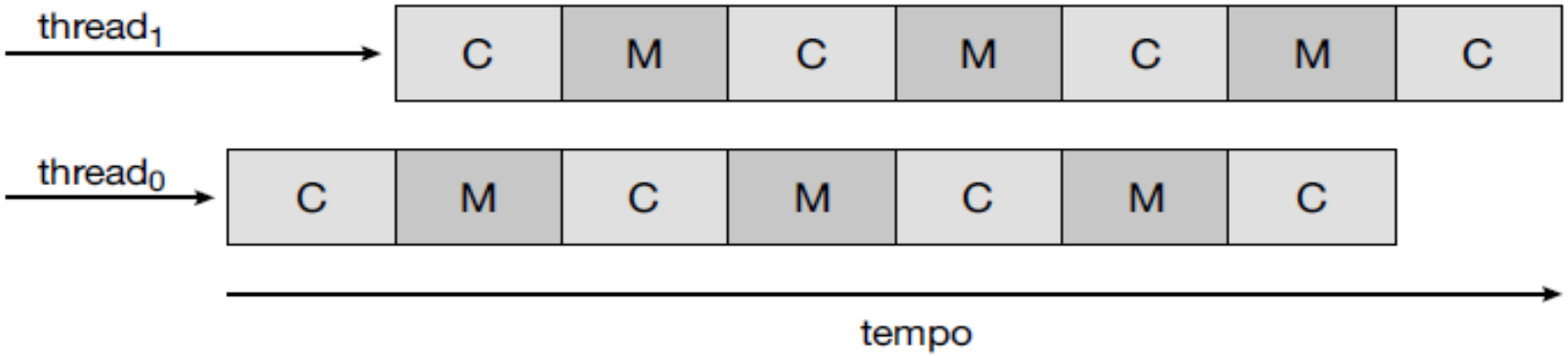
Processori multicore

- Più core di elaborazione sono inseriti in un unico chip fisico
  - Ogni core mantiene il proprio stato architetturale (PC, IR, altri registri) e quindi appare al SO come un processore separato
  - Sono più veloci e consumano meno energia dei sistemi in cui ciascun processore è costituito da un chip separato
- Stallo della memoria: quando un core accede alla memoria, poiché lavora ad una velocità molto superiore rispetto alla memoria, trascorre una quantità significativa di tempo in attesa che i dati diventino disponibili
  - Il core può trascorrere anche il 50% del tempo in stallo



Chip MultiThreading (CMT)

- Soluzione: HW recente implementa core multithread, in cui due (o più) thread HW sono assegnati ad ogni core
  - Così se un thread HW si blocca, il core può eseguirne un altro

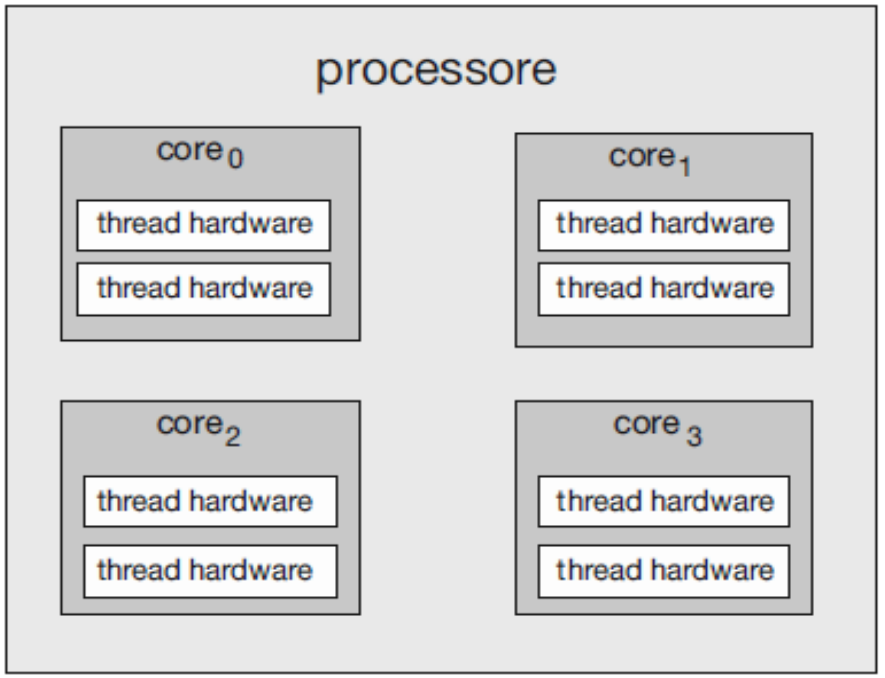


Core a due thread che si avvicendano nel tempo

Da un punto di vista del SO, ogni thread HW mantiene il proprio stato architetturale (PC, IR, altri registri), così appare come una CPU logica in grado di eseguire thread SW (questa tecnica è nota come Chip MultiThreading)

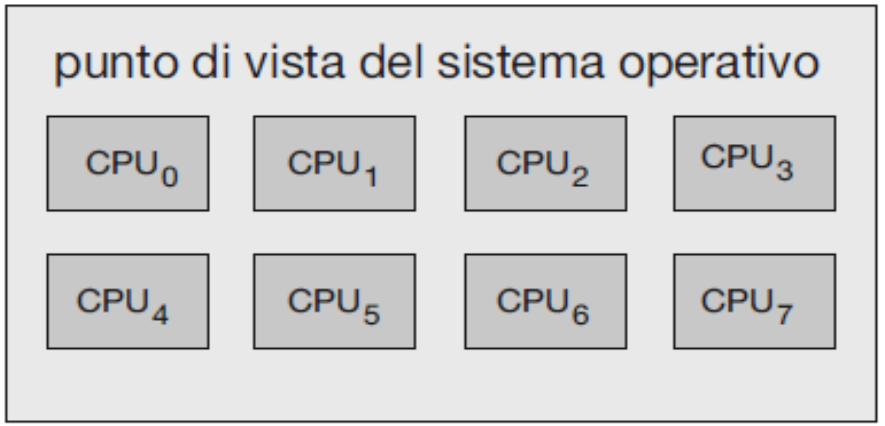
Chip MultiThreading (CMT)

Esempio: un processore contiene quattro *core di elaborazione*, ognuno dei quali contiene due *thread HW*: dal punto di vista del SO sono presenti otto CPU logiche



Intel ha introdotto il termine hyperthreading (2003) per indicare l'assegnazione di più thread HW ad un singolo core

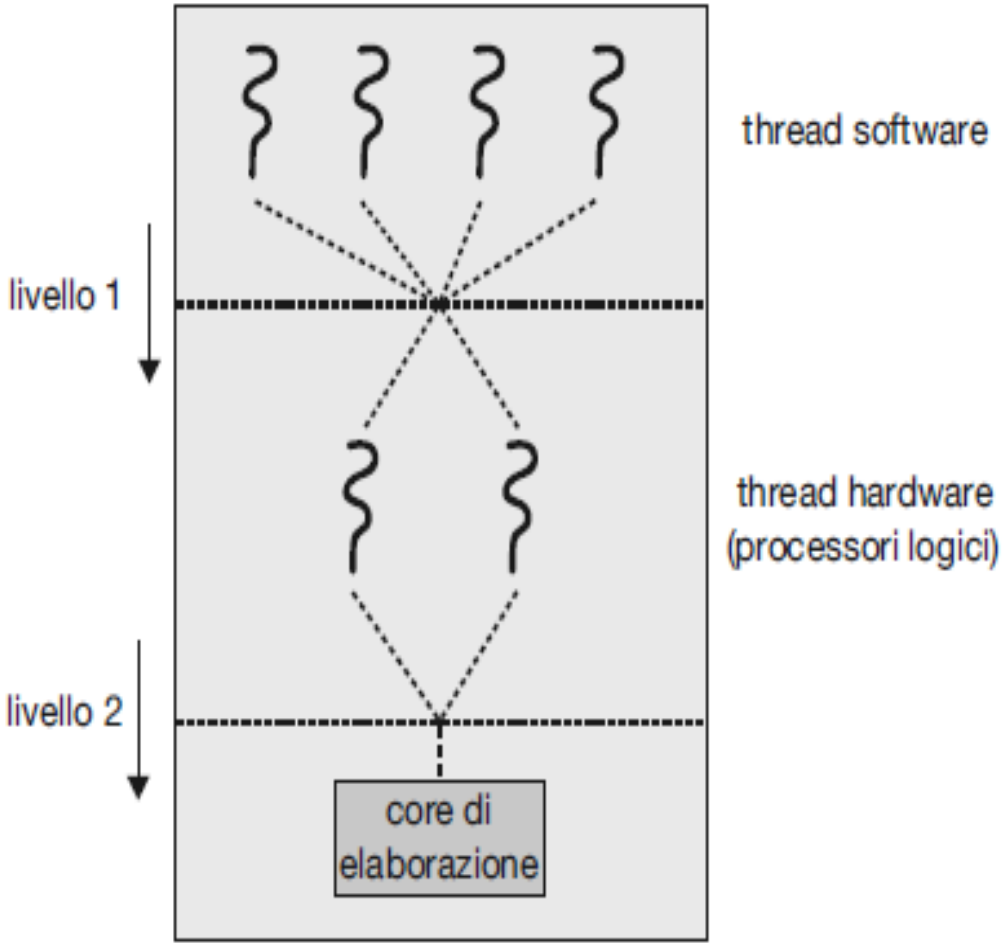
- i7 ha 2 thread per core



Core multithread

Le risorse del core fisico (es. cache e pipeline) sono condivise tra i suoi thread HW, quindi il core può eseguire un solo thread HW alla volta

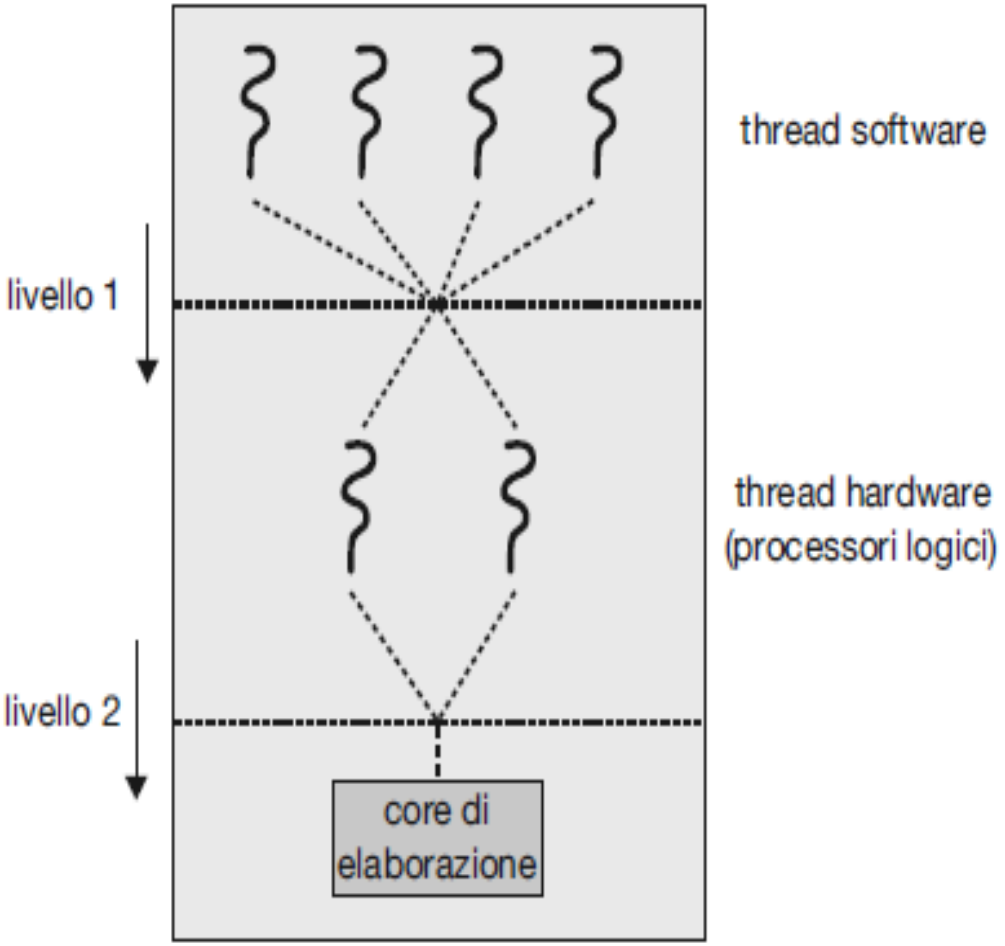
Un core multithread richiede due diversi livelli di scheduling



Core multithread

Le risorse del core fisico (es. cache e pipeline) sono condivise tra i suoi thread HW, quindi il core può eseguire un solo thread HW alla volta

- Lo scheduling a livello 1 (thread SW) è responsabilità del SO, che può usare una qualsiasi delle strategie che abbiamo visto
- Lo scheduling a livello 2 (thread HW) è gestito dal core, spesso con una strategia round-robin o a priorità



Processori multicore e multithread

In un processore multicore con core multithread i due livelli di scheduling non sono necessariamente scorrelati

- Supponiamo che un processore abbia due core di elaborazione e ogni core abbia due thread HW
- Se due thread SW sono in esecuzione su questo sistema, possono essere eseguiti sullo stesso core o su core separati
- Se sono entrambi assegnati allo stesso core, devono condividere le risorse del processore e quindi è probabile che procedano più lentamente rispetto al caso in cui siano assegnati a core separati
- Se il SO è a conoscenza del livello di condivisione delle risorse tra i core del processore, può schedulare thread SW su thread HW che non condividono risorse, così da migliorare le prestazioni

Bilanciamento del carico

Per sfruttare appieno i vantaggi di avere più di un processore è importante mantenere il carico di lavoro bilanciato tra tutti i processori

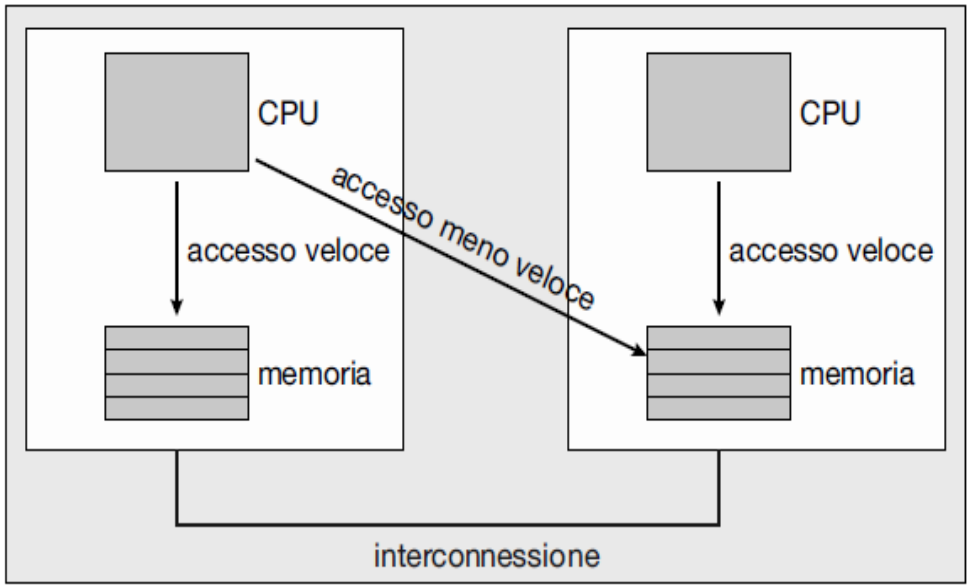
- Nei sistemi con una coda ready comune, il bilanciamento del carico non è necessario, poiché quando un processore diventa inattivo, estrae immediatamente un thread eseguibile dalla coda pronta condivisa
- Nei sistemi in cui ogni core ha la propria coda ready privata, esistono due approcci generali al bilanciamento del carico:
  - migrazione push: un processo apposito controlla periodicamente il carico su ciascun processore e, se rileva uno squilibrio, sposta i thread dal processore sovraccarico a uno inattivo o con meno carico
  - migrazione pull: un processore inattivo estrae esso stesso un processo in attesa da un processore sovraccarico
- Le due strategie non sono mutuamente esclusive ma tipicamente convivono sullo stesso sistema, come ad esempio è il caso dello scheduler CFS di Linux e dello scheduler ULE dei sistemi FreeBSD

Predilezione per il processore (processor affinity)

- Un thread ha una predilezione per il processore su cui è correntemente in esecuzione, per via del costo necessario per invalidare la cache sul processore e ripopolarla su un altro nel caso in cui il SO spostasse il thread
  - Se la coda ready è privata, la predilezione per il processore è realizzata automaticamente
  - La predilezione per il processore è in conflitto con l'obiettivo di bilanciamento del carico
- Predilezione debole (*soft affinity*): il SO fa il possibile per rischedulare il thread sullo stesso processore, ma non c'è garanzia e il load balancer potrebbe spostarlo
- Predilezione forte (*hard affinity*): il SO mette a disposizione alcune system call per specificare un sottoinsieme di processori utilizzabili per eseguire il thread
- Può essere influenzata dall'architettura della memoria principale del sistema

Predilezione per il processore & NUMA

- Consideriamo un'architettura con *accesso non uniforme alla memoria* (NUMA) in cui sono presenti due chip fisici di elaborazione, ciascuno con una propria CPU e memoria locale
- Sebbene tutte le CPU in un sistema NUMA condividano uno spazio di indirizzi fisico tramite interconnessione, l'accesso di una CPU alla sua memoria locale è più veloce che a quella condivisa



- Se lo scheduler e gli algoritmi di gestione della memoria del SO hanno conoscenza dell'architettura NUMA e lavorano di concerto, allora ad ogni thread potrebbero assegnare memoria vicina alla CPU su cui il thread è schedulato

Sistemi multiprocessore eterogenei (heterogeneous multiprocessing, HMP)

- In alcuni sistemi, quali certi dispositivi mobili, i core possono differire tra loro per velocità del clock e gestione dell'energia
  - Ciò non rientra in asymmetric multiprocessing perché, in principio, i thread utente e di sistema possono essere eseguiti su qualsiasi core
  - L'obiettivo piuttosto è di gestire meglio il consumo energetico
- Nei processori ARM che supportano HMP, l'architettura è nota come big.LITTLE: big core ad alte prestazioni (da usare per periodi brevi) sono combinati con LITTLE core a basso consumo energetico
  - Task da eseguire per lunghi periodi (esempio in background) possono essere allocati su core LITTLE così da preservare la batteria
  - Task interattivi che richiedono maggiore potenza di calcolo, ma per meno tempo, possono essere allocati su core big
  - Se il dispositivo è in modalità risparmio energetico, i core big possono essere disabilitati

Sistemi multiprocessore eterogenei (heterogeneous multiprocessing, HMP)

- In alcuni sistemi, quali certi dispositivi mobili, i core possono differire tra loro per velocità del clock e gestione dell'energia
  - Ciò non rientra in asymmetric multiprocessing perché, in principio, i thread utente e di sistema possono essere eseguiti su qualsiasi core
  - L'obiettivo piuttosto è di gestire meglio il consumo energetico
- Nei processori ARM che supportano HMP, l'architettura è nota come big.LITTLE: big core ad alte prestazioni (da usare per periodi brevi) sono combinati con LITTLE core a basso consumo energetico
- Similmente, i processori Intel si basano su una architettura ibrida con un mix di core ad alte prestazioni (P-core) per i compiti più gravosi e di core a basso consumo energetico (E-core) per le operazioni meno impegnative
- Windows 10 supporta lo scheduling HMP permettendo a ciascun thread di selezionare la politica di scheduling che meglio soddisfa le proprie necessità di gestione energetica

Scheduling della CPU

- Concetti fondamentali
- Criteri di scheduling
- Algoritmi di scheduling
- Scheduling dei thread
- Scheduling per sistemi multiprocessore
- Valutazione degli algoritmi di scheduling

Valutazione degli algoritmi di scheduling

Come scegliere un algoritmo di scheduling per un sistema specifico?

1. Fissare i criteri in base ai quali valutare e scegliere l'algoritmo
   - Massimizzare l'utilizzo della CPU
   - Massimizzare la frequenza di completamento
   - Minimizzare il tempo di completamento
   - Minimizzare il tempo di attesa
   - Minimizzare il tempo di risposta
   - …
2. Utilizzare un metodo per la valutazione
   - Modelli deterministici
   - Modelli a reti di code
   - Simulazione
   - Implementazione

Criteri di valutazione e di scelta

- Dei criteri abbiamo già ampiamenti parlato, sono a volte contrastanti per cui bisogna stabilire la relativa importanza in relazione all'ambiente di elaborazione
- Ad esempio, potremmo essere interessati a:
  - Massimizzare l'utilizzo della CPU rispettando il vincolo che il tempo di risposta massimo dev'essere di 300 millisecondi
  - Massimizzare la frequenza di completamento in modo tale che il tempo di completamento sia (in media) linearmente proporzionale al tempo di esecuzione totale

Modelli deterministici

- È una forma di valutazione analitica: considerato un carico di lavoro predeterminato, definisce le prestazioni di un algoritmo per quel carico
- **Vantaggi**: semplice e rapido
  - Restituisce dei valori numerici che rappresentano le prestazioni di ogni singolo algoritmo e che sono quindi semplici da confrontare
  - Utile ai fini didattici (è il metodo che useremo noi per risolvere gli esercizi relativi allo scheduling della CPU)
- **Svantaggi**: per essere effettivamente usato in pratica, sono necessarie conoscenze non sempre disponibili
  - Non sempre è possibile conoscere a priori il carico di lavoro di un sistema
  - In generale, tale carico non è sempre lo stesso nel tempo

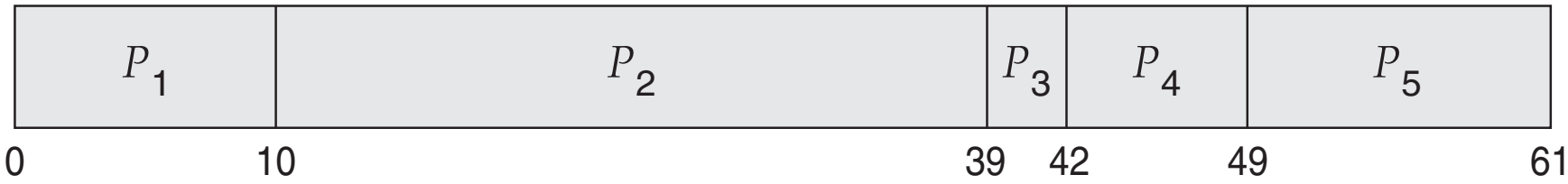
Modelli deterministici (I esempio)

Si supponga di avere il seguente carico di lavoro:

- cinque processi entrano nel sistema al tempo 0
- durata in msec. dei burst di CPU: P_1 (10), P_2 (29), P_3 (3), P_4 (7), P_5 (12)

Considerando come criterio di valutazione il *tempo di attesa medio,* confrontare le prestazioni degli algoritmi FCFS, SJF, RR (con quanto pari a 10 msec.)

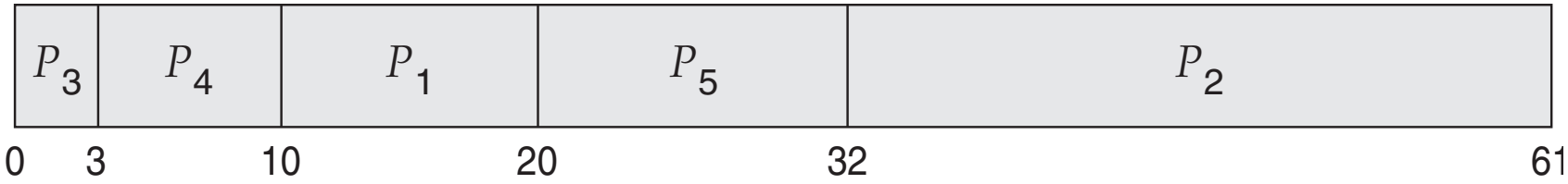
Algoritmo FCFS



Tempo di attesa medio: (0+10+39+42+49)/5 = 28

Modelli deterministici (I esempio)

Algoritmo SJF



Tempo di attesa medio: (10+32+0+3+20)/5 = 13

Si supponga di avere il seguente carico di lavoro:

- cinque processi entrano nel sistema al tempo 0
- durata in msec. dei burst di CPU: P_1 (10), P_2 (29), P_3 (3), P_4 (7), P_5 (12)

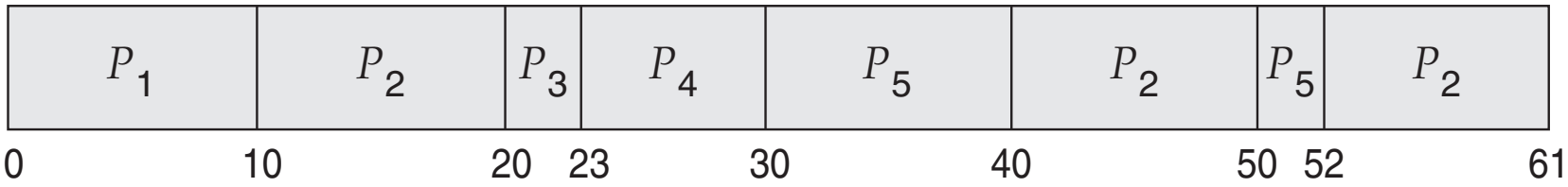
Considerando come criterio di valutazione il *tempo di attesa medio.*

Modelli deterministici (I esempio)

Si supponga di avere il seguente carico di lavoro:
- cinque processi entrano nel sistema al tempo 0
- durata in msec. dei burst di CPU: P_1 (10), P_2 (29), P_3 (3), P_4 (7), P_5 (12)

Considerando come criterio di valutazione il *tempo di attesa medio.*

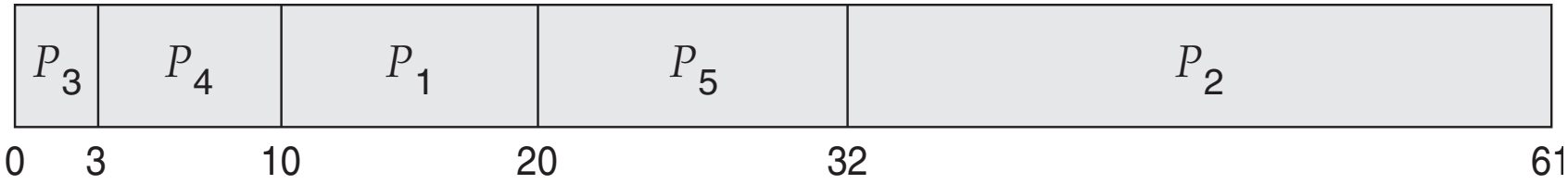
Algoritmo RR (con quanto pari a 10 msec.)



Tempo di attesa medio: (0+32+20+23+40)/5 = 23

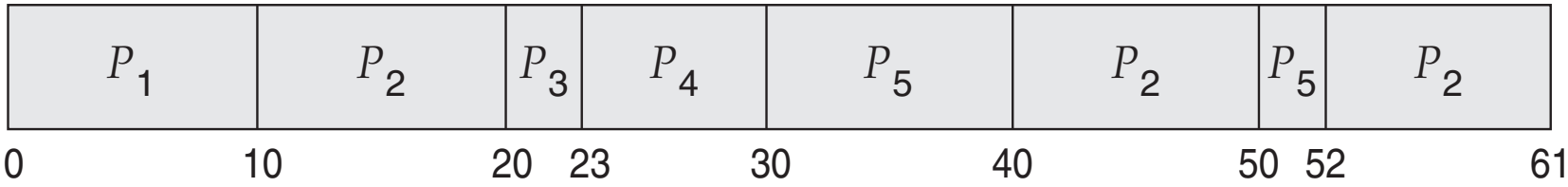
Modelli deterministici (I esempio)

Algoritmo SJF



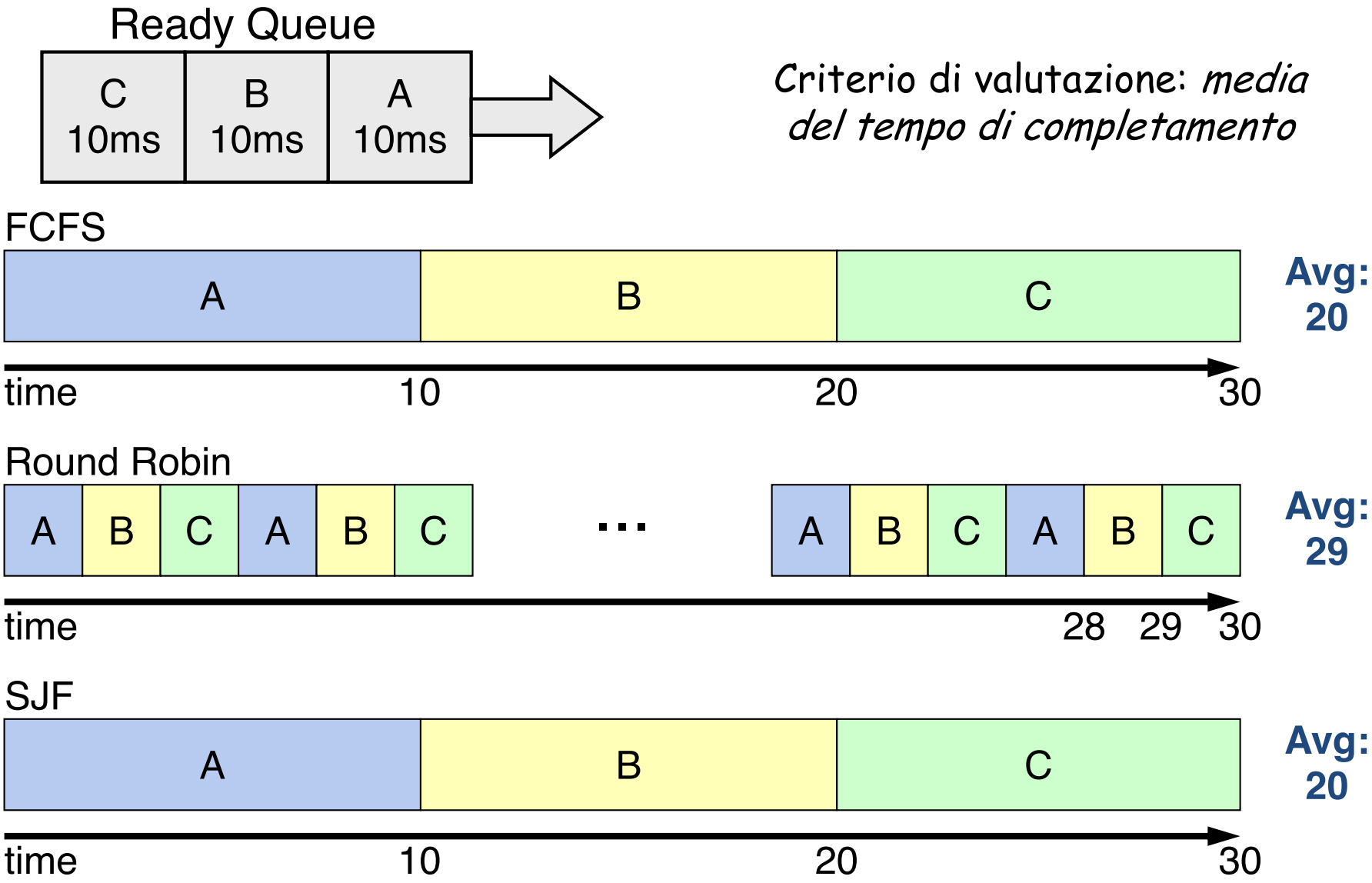
Tempo di attesa medio: (10+32+0+3+20)/5 = 13

Algoritmo RR (con quanto pari a 10 msec.)

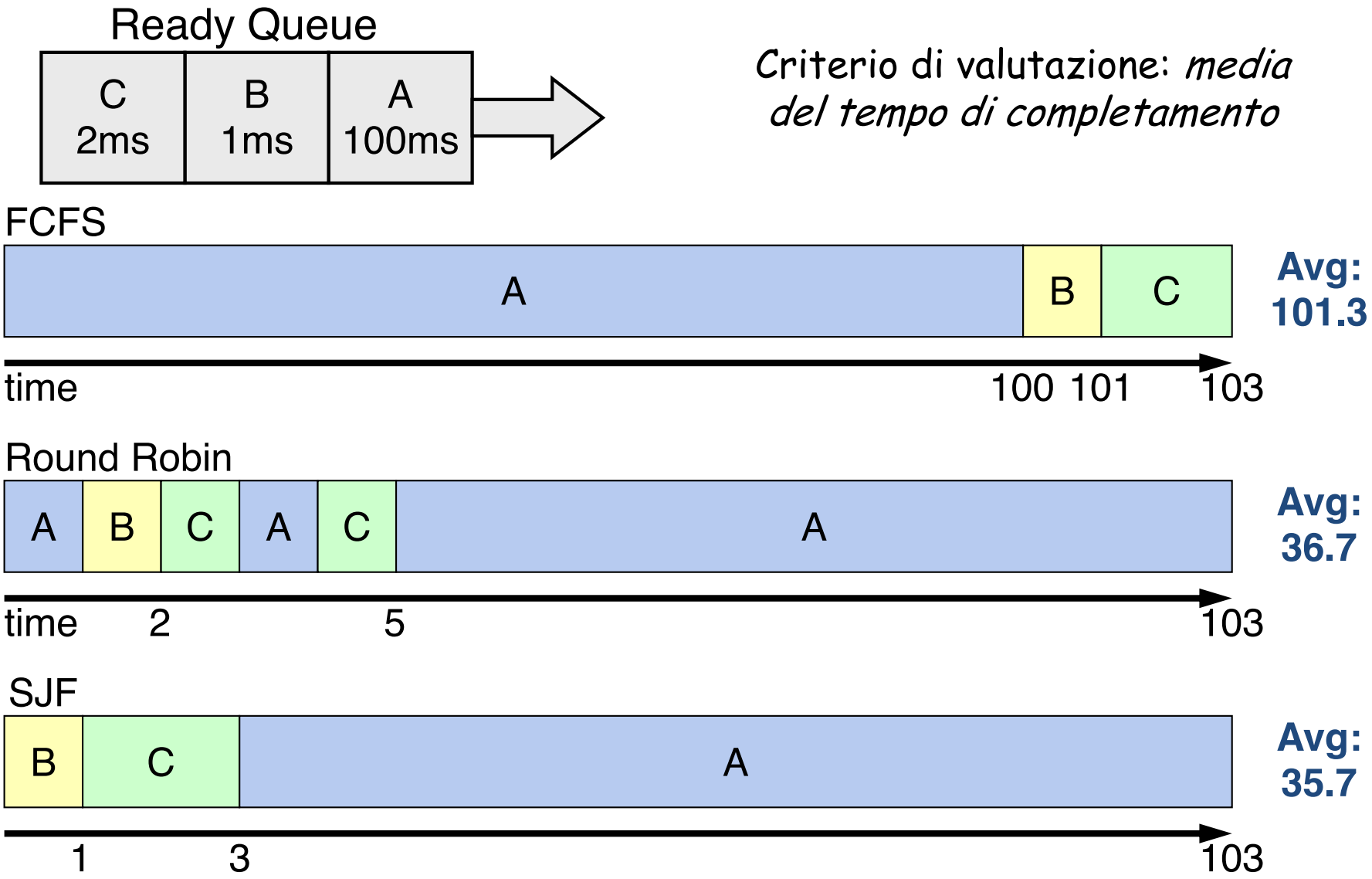


Tempo di attesa medio: (0+32+20+23+40)/5 = 23

Modelli deterministici (II esempio)



Modelli deterministici (III esempio)



Modelli con reti di code

- Anche se il carico di lavoro di un sistema non è sempre lo stesso, si possono determinare tramite misurazione ed approssimazione o tramite stima le distribuzioni di probabilità (tipicamente esponenziali)
  - delle durate dei CPU burst e degli I/O burst dei processi
  - degli istanti di arrivo dei processi
- Il sistema di calcolo è quindi modellato come una rete di server ciascuno con associata una coda di processi in attesa:
  - alla CPU è associata la coda dei pronti
  - ai dispositivi sono associate le code corrispondenti
- Le distribuzioni precedenti permettono allora di calcolare l'utilizzo dei server, la lunghezza media delle code, il tempo medio di attesa, …
- Svantaggi:
  - le classi di algoritmi e di distribuzioni trattabili sono piuttosto limitate poiché
  - l'analisi è complessa e spesso richiede semplificazioni ed assunzioni inaccurate che inficiano la precisione dei risultati

Simulazioni

- Si tratta di programmare un modello del sistema in cui:
  - le strutture dati rappresentano le componenti principali del sistema, il loro contenuto rappresenta lo stato della componente
  - una variabile rappresenta il clock, cosicché all'aumentare del valore di questa variabile, il simulatore modifica lo stato del sistema per riflettere le attività dei dispositivi e dei processi
  - mentre la simulazione viene eseguita, vengono raccolte e mostrate alcune statistiche che indicano le prestazioni dell'algoritmo di scheduling
- I dati necessari per la simulazione possono essere ottenuti
  - facendo uso di generatori di numeri casuali programmati per generare: processi, durate delle sequenze di operazioni, ingressi, terminazioni, …
  - tenendo conto di distribuzioni di probabilità definite matematicamente o calcolate empiricamente monitorando il comportamento effettivo di un sistema reale (*trace tape*)
- La valutazione può dare risultati molto precisi, ma è computazionalmente onerosa, richiede molta memoria per i trace tape, e lo sviluppo del simulatore è complesso

Implementazione

- Anche una simulazione può essere limitata per quel che riguarda la precisione
- Il modo più certo per valutare un algoritmo di scheduling è di programmarlo, inserirlo nel SO e osservarne il comportamento durante il normale uso del sistema
- Svantaggi:
  - costo, sia per la codifica dell'algoritmo che per le modifiche da apportare al SO
  - disagi per gli utenti a fronte di modifiche al SO