# Scheduling della CPU (1)

- La gestione delle risorse impone al SO di prendere decisioni sulla loro assegnazione in base a criteri di *efficienza* e *funzionalità*.

- Le risorse più importanti, a questo riguardo, sono la CPU e la memoria principale.

# Scheduling della CPU (2)

- Scheduler (della CPU):
  - parte del S.O. che decide a quale dei *processi pronti* presenti nel sistema assegnare il controllo della CPU

- Algoritmo di scheduling (assegnazione della CPU):
  - realizza *un particolare criterio* di scelta tra i processi pronti (politica)

  In base a quali elementi ?

# Assegnazione della CPU (CPU scheduling)

- Un processo alterna attività di CPU e attesa per I/O:

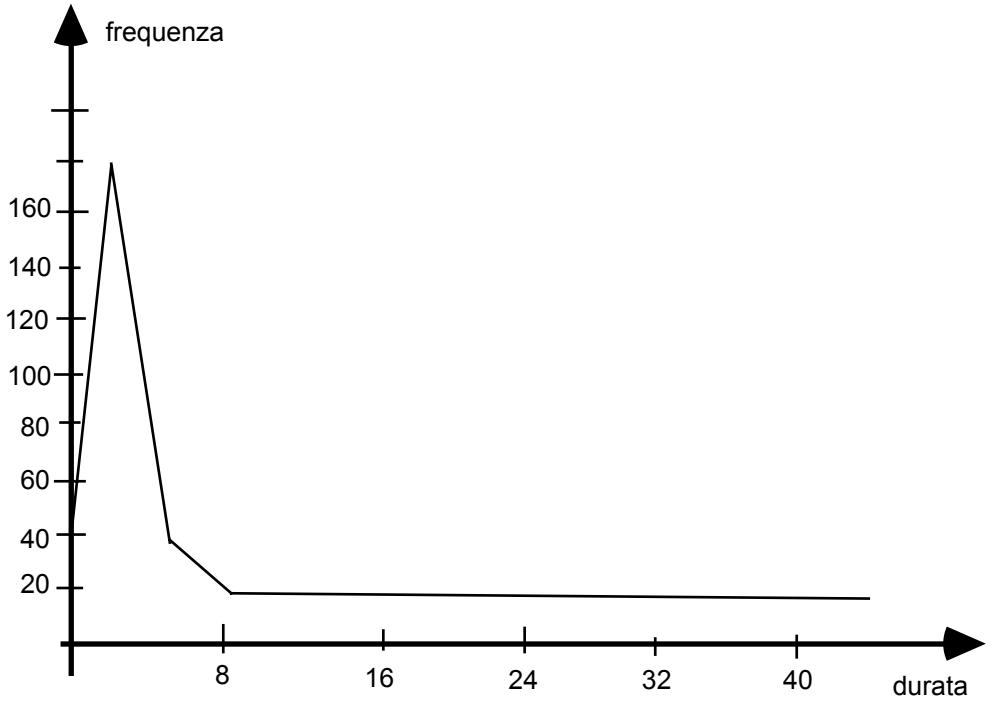
```
...
LOAD                }
ADD                 } CPU burst
STORE               } (seq. di operazioni di CPU contigue)
READ from file      }

WAIT for I/O        } I/O burst

STORE               }
INCREMENT index     } CPU burst
WRITE to file       }

WAIT for I/O        } I/O burst
...
```

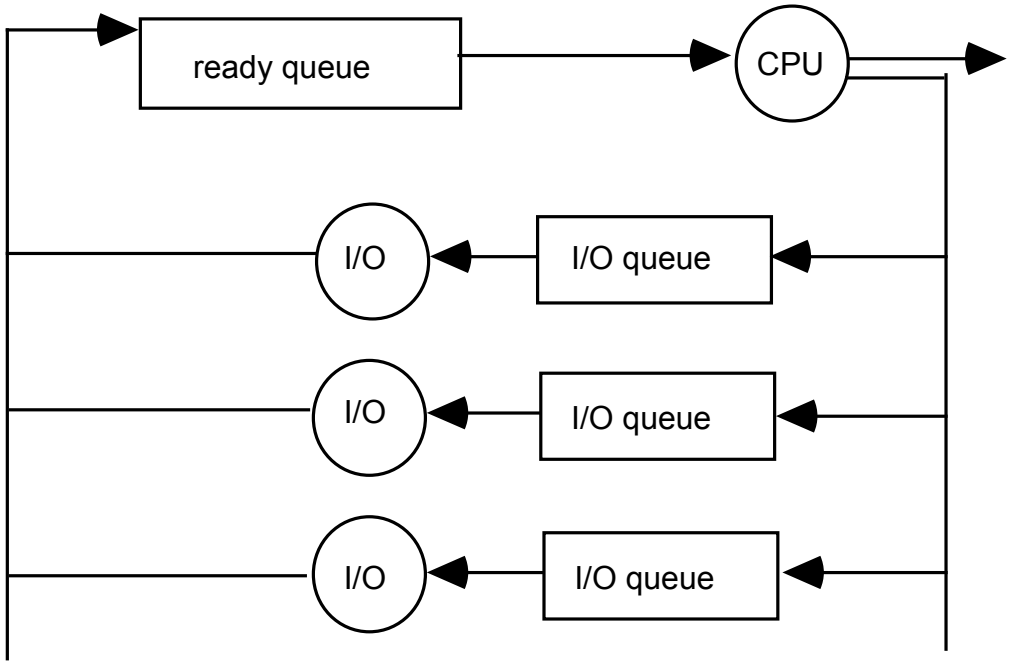
# CPU bursts (1)




Andamento esponenziale: un gran numero di burst molto brevi ed un piccolo numero di burst molto grandi

# CPU bursts (2)

Un programma I/O bound ha molti burst di CPU, brevi
Un programma CPU bound ha pochi burst di CPU, lunghi



# CPU schedulers (1)

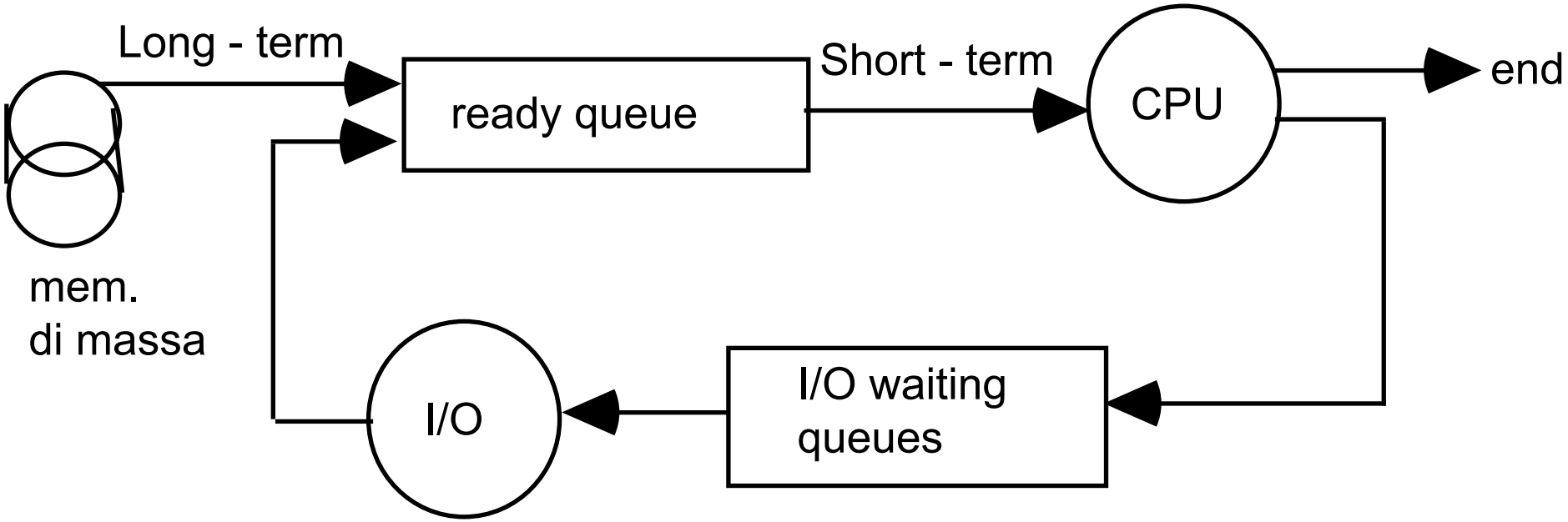
- Long-term scheduler (o *job scheduler*):
  - Determina *quali processi* dalla memoria di massa devono essere caricati in memoria principale pronti per l'esecuzione.
  - *Controlla il grado di multiprogrammazione* (numero di processi in memoria). Interviene, di regola, quando un processo abbandona il sistema.
  - Il *criterio di selezione* e' basato su un mix equilibrato di jobs I/O bound e CPU bound.

# CPU schedulers (2)

- Short-term scheduler:
  - *Seleziona* tra tutti i processi in memoria pronti per l'esecuzione quello cui assegnare la CPU.
  - Deve essere *efficiente* in quanto interviene *frequentemente*.

- Dispatcher:
  - *Esegue* le operazioni relative al cambiamento di contesto.

# CPU schedulers (3)



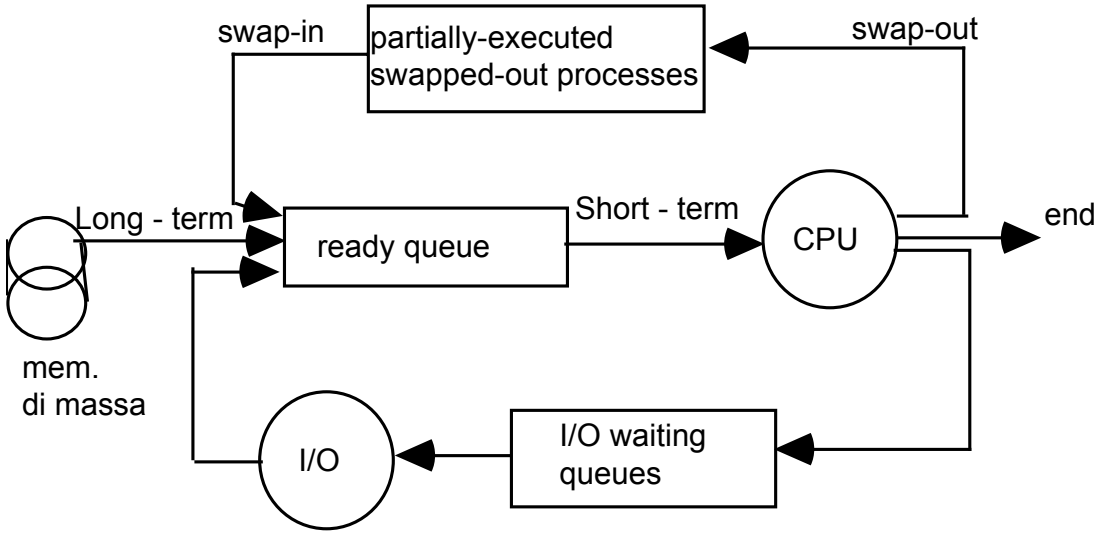
# CPU schedulers (4)

- Nei sistemi *time-sharing* non esiste long-term scheduling.

  I processi entrano immediatamente in memoria centrale. Il limite e' imposto o dal numero di terminali connessi o dal tempo di risposta che diviene troppo lungo (sconsigliando l'uso del sistema quando sovraccarico).

# CPU schedulers (5)

- Medium-term scheduler:




Puo' risultare vantaggioso rimuovere i processi dalla memoria e *ridurre il grado di multiprogrammazione*. I processi vengono successivamente reintrodotti. *(Swapping)*

# Scheduling e revoca della CPU (1)

La riassegnazione della CPU può avvenire a seguito di uno dei seguenti 4 *eventi:*

1. Un processo commuta *dallo stato di esecuzione a sospeso* (ad es., per richiesta di operazione di I/O, wait su semaforo, etc.).

2. Il processo in esecuzione *termina*.

# Scheduling e revoca della CPU (2)

3. Un processo commuta *dallo stato di esecuzione a pronto* (ad es., a seguito della elaborazione di un interrupt).

4. Un processo commuta *dallo stato sospeso a pronto* (ad es., per il completamento di un'operazione di I/O).

# Scheduling e revoca della CPU (3)

- Scheduling *non-preemptive:*
  - un processo in esecuzione prosegue fino al rilascio spontaneo della CPU;
  - la riassegnazione della CPU avviene solo a seguito di eventi di tipo 1 e 2.

- Scheduling *preemptive:*
  - il processo in esecuzione può perdere il controllo della CPU anche se in grado di proseguire;
  - la riassegnazione della CPU può avvenire anche a seguito di eventi di tipo 3 e/o 4.

# Algoritmi di scheduling (1)

- Criteri:
  - CPU utilization
  - Throughput
  - Turnaround time (tempo in mem. di massa, coda "pronti", esecuzione, I/O)
  - Waiting time (tempo speso nella coda dei processi pronti)
  - Response time
  - Fairness (assenza di privilegi)

# Algoritmi di scheduling (2)

- Scelto un criterio, si tenta di ottimizzare (minimizzare o massimizzare).

- Per sistemi interattivi (time sharing) é più importante minimizzare la *varianza* nel tempo di risposta piuttosto che il *tempo medio* di risposta.

- Misura di confronto scelta: *tempo medio di attesa* (waiting time).

# Algoritmi di scheduling (3)

- Un'analisi accurata dovrebbe comprendere molti processi, ciascuno costituito da una sequenza di diverse centinaia di:
  - CPU burst ed
  - I/O burst.

- Per semplicità nel seguito viene considerato *un solo burst di CPU* per ciascun processo.

# First-Come-First-Served (F.C.F.S.)

- La CPU viene assegnata al processo che l'ha richiesta per primo.

- La realizzazione di questa politica é ottenuta con code gestite FIFO.

- Le prestazioni di questo algoritmo sono in genere *basse* in termini di *tempo medio di attesa*.

# First-Come-First-Served (2)

- Es.:

| Processi | burst di CPU |
|---|---|
| 1 | 24 |
| 2 | 3 |
| 3 | 3 |

I processi arrivano nell'ordine 1, 2, 3 e sono serviti con politica FCFS

- Tempi di attesa:
  - per il processo 1 é 0
  - per il processo 2 é 24
  - per il processo 3 é 27

# First-Come-First-Served (3)

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

Se i processi arrivano nell'ordine 2, 3, 1 si ha:

Tempi di attesa: per il processo 1 é 6
per il processo 2 é 0
per il processo 3 é 3

- Tempo medio di attesa: (6 + 0 + 3) / 3 = 3

# Shortest-Job-First (S.J.F.)

- A ciascun processo é associata *la lunghezza del successivo burst di CPU*.
- Quando la CPU é libera, viene assegnata al processo con il burst di CPU più breve.

| Processi | Burst-time |
|---|---|
| 1 | 6 |
| 2 | 8 |
| 3 | 7 |
| 4 | 3 |

- Adottando S.J.F. il tempo medio di attesa é 7 (con F.C.F.S. si sarebbe ottenuto 10.25)

# Shortest-Job-First (2)

Si può dimostrare che S.J.F. fornisce la soluzione *ottima*.

Infatti, se si esegue un processo breve prima di uno lungo il tempo di attesa del processo breve diminuisce più di quanto aumenti il tempo di attesa di quello lungo.

La difficoltà sta nel conoscere la lunghezza della successiva richiesta di CPU. Si può *predire* tale lunghezza (facendo ad esempio l'ipotesi che sia simile a quella del burst precedente).

# Shortest-Job-First (3)

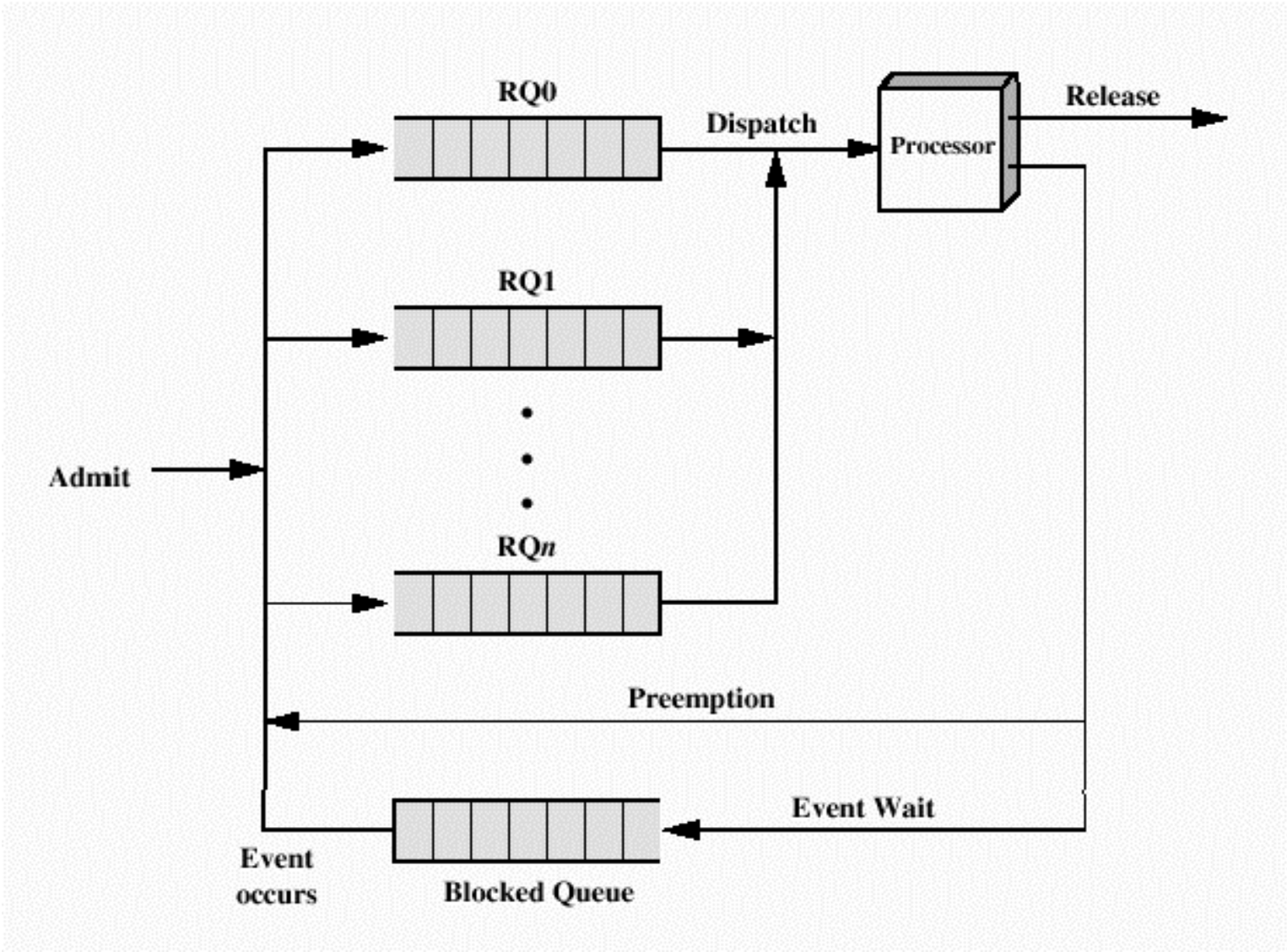
- E' possibile usare dei *Modelli analitici:* il burst successivo di CPU viene <u>stimato</u> come una media esponenziale delle lunghezze dei precedenti burst

$$T^*_{n+1} = a\, t_n + (1 - a)\, T^*_n \qquad 0 \le a \le 1$$

$t_n$ = lunghezza dell'n-esimo burst di CPU

$T^*_{n+1}$ = stima della lunghezza dell'n+1-esimo burst di CPU

# Priorità (1)


RQ0
Dispatch
Processor
Release
RQ1
Admit
RQn
Preemption
Event Wait
Event
occurs
Blocked Queue

# Priorità (2)

- Lo scheduler seleziona sempre il primo processo nella *coda al livello massimo di priorità* che contiene processi pronti.

- In presenza di *preemption*, in ogni istante é in esecuzione un processo a priorità massima.

# Priorità (3)

Valutata *internamente* o *esternamente*.

- Nel primo caso la priorità é individuata sulla base di qualche quantità misurabile. Ad es., limiti di tempo, richieste di memoria, numero di file aperti, rapporto tra i burst medi di I/O e di CPU, etc.

- Nel secondo caso la priorità é imposta da condizioni esterne.

# Priorità e Starvation

- Processi a bassa priorità possono rimanere indefinitamente ritardati se lo scheduler seleziona sempre i processi ad alta priorità.
  - Questo problema si dice *starvation*.

- Una soluzione é quella di *aumentare gradualmente* la priorità dei job che attendono.

# Priorità statica e dinamica (1)

- Priorità *statica:* attribuita ai processi all'atto della loro creazione in base alle loro caratteristiche o a politiche riferite al tipo di utente.

- In generale vengono favoriti i processi di sistema e di I/O; inoltre:
    - processi foreground (interattivi) => alta priorità
    - processi background (batch) => bassa priorità

- Possibilità di starvation.

# Priorità statica e dinamica (2)

- Priorità *dinamica:* modificata durante l'esecuzione del processo.

  - per penalizzare i processi che impegnano troppo la CPU
  - per evitare starvation
  - per favorire i processi I/O-bound

L'obiettivo finale è quindi di mantenere (indirettamente) un buon job-mix.

# Round Robin (R.R.)

- Time sharing systems. Quanto di tempo assegnato a tutti i processi (10 - 100 msec).

- La coda dei processi pronti é *circolare* e la CPU é assegnata a ciascuno dei processi per un quanto di tempo.

- Un processo, interrotto per l'esaurimento del suo quanto, viene inserito come *ultimo* nella coda dei processi pronti (preemption).

# Round Robin (2)

La politica di scheduling round-robin é caratterizzata da:

- *elevata fairness* e
- *assenza di starvation*.

Tuttavia il context-switch *imposto* all'esaurimento del quanto di tempo determina un *incremento dell'overhead*.

# Algoritmi di scheduling (1)

- Gli algoritmi FCFS, SJF, e a priorità visti in precedenza sono di tipo *non-preemptive*, cioé quando la CPU é stata assegnata ad un processo questi ne mantiene il controllo fino
  - al completamento, o
  - alla richiesta di una operazione di I/O, o
  - all'esecuzione di una sincronizzazione sospensiva.

# Algoritmi di scheduling (2)

- Gli algoritmi SJF e a priorità possono essere anche di tipo *preemptive*. Tale possibilità nasce quando, durante l'esecuzione di un processo, un nuovo processo entra nella coda dei processi pronti.
  Il nuovo processo può:
  - richiedere un tempo di CPU inferiore (per SJF) o
  - avere una priorità maggiore di quello in esecuzione (a priorità).

- L'algoritmo SJF di tipo preemptive viene chiamato anche *shortest remaining time first.*

# Code a più livelli (1)

- Suddivisione tra job *foreground* (interattivi) e *background* (batch):
  Algoritmi diversi in quanto sono diverse le esigenze.

- Più in generale, possono esistere *classi diverse di job* che vengono assegnati *staticamente* ad una coda.

- Ogni coda ha un proprio algoritmo di scheduling. Ad esempio, per i job foreground l'algoritmo R.R., per quelli background l'algoritmo FCFS.

# Code a più livelli (2)

- In taluni casi può essere consentito ad un job di cambiare, eventualmente temporaneamente, coda:
  - un job che usa troppo tempo di CPU può passare ad un livello inferiore
  - un job che attende da troppo tempo può passare ad un livello superiore.

# Code a più livelli (3)

Occorre definire:

- il numero di code
- l'algoritmo di scheduling per ogni coda
- un criterio per decidere quando spostare un job ad una coda di priorità più elevata
- un criterio per decidere quando spostare un job ad una coda di priorità più bassa
- un metodo per determinare in quale coda un job deve entrare quando inizia il servizio.

# Valutazione degli algoritmi (1)

*Scelta dei criteri*

Ad esempio:

- massimizzare l'utilizzazione della CPU con il vincolo che il tempo di risposta massimo sia 1 secondo.

- massimizzare il throughput in modo che il tempo di risposta sia (in media) proporzionale al tempo di esecuzione totale.

# Valutazione degli algoritmi (2)

Valutazione analitica

Esistono diversi metodi per valutare analiticamente le prestazioni degli algoritmi di scheduling:

1. Modelli deterministici
2. Modelli basati sulla teoria delle code
3. Modelli basati su reti di Petri temporizzate
4. …

# Valutazione degli algoritmi (3)

a) *Modelli deterministici:*

Fissato un carico di lavoro vengono definite le prestazioni di ciascun algoritmo.

- Esempio: supponiamo che al tempo 0 arrivino 5 job nel seguente ordine:

| job | burst time |
|---|---|
| 1 | 10 |
| 2 | 29 |
| 3 | 3 |
| 4 | 7 |
| 5 | 12 |

# Valutazione degli algoritmi (4)

Si considerino i tre algoritmi:

- FCFS
- SJF
- RR con quanto = 10

I tempi medi di attesa sono:

| | |
|---|---|
| FCFS | T = (0 + 10 + 39 + 42 + 49) / 5 = 28 |
| SJF | T = (10 + 32 + 0 + 3 + 20) / 5 = 13 |
| RR (q = 10) | T = (0 + 32 + 20 + 23 + 40) / 5 = 23 |

# Valutazione degli algoritmi (5)

b) *Modelli basati sulla teoria delle code:*

- Il sistema di calcolo é descritto come una *rete di servitori*. Ciascun servitore ha una coda di job in attesa. Conoscendo le *frequenze di arrivo* e i *tempi di servizio* (in termini di distribuzione di probabilità) si può calcolare la lunghezza media delle code, i tempi medi di attesa, etc.

- Moltissime limitazioni. Occorre conoscere le distribuzioni di probabilità dei burst di CPU e di I/O e dei tempi di arrivo nel sistema dei job. E' difficile o impossibile esprimere sincronizzazioni. La soluzione é approssimata e spesso la sua validità é incerta.

# Valutazione degli algoritmi (6)

c) *Modelli basati su reti di Petri temporizzate*.

- Il sistema viene descritto in termini di *eventi* e *condizioni*. Le attività sono tipicamente rappresentate tramite transizioni temporizzate.

- Occorre caratterizzare le durate delle attività dei job in maniera analoga a quanto avviene con le reti di code. E' possibile esprimere sincronizzazioni ma la risolubilità analitica é condizionata da limitazioni forti sulle distribuzioni ammesse. Inoltre la modellazione di alcune politiche di scheduling, possibile in linea di principio, dà luogo a reti complicate.

# Valutazione degli algoritmi (7)

Simulazione

Si costruisce un *modello del sistema* utilizzando le reti di code o le reti di Petri come strumento di rappresentazione. Non si risolve analiticamente il modello ma lo si simula tramite un programma di calcolo.

Le distribuzioni sono definite matematicamente o empiri-camente, desumendole da misure effettive sul sistema.

La simulazione affidabile di un modello é sempre molto più costosa della sua risoluzione analitica.

# Lo scheduling in UNIX (1)

- L'algoritmo di scheduling favorisce i *job di tipo interattivo* (foreground).

- Si tratta di un algoritmo *round-robin con priorità* variabile.

- Ad ogni processo é associata una priorità di scheduling. La priorità é rappresentata in senso decrescente: più é basso il valore, più é elevata la priorità.

- Processi che svolgono attività di I/O su disco hanno priorità negativa e non possono essere interrotti.

# Lo scheduling in UNIX (2)

- La priorità *varia dinamicamente*: al crescere del tempo di CPU utilizzato da un processo diminuisce la sua priorità. Analogamente, al crescere del tempo di attesa di un processo aumenta anche la sua priorità (per evitare starvation).

- BSD 4.2 ha un quanto di tempo di 0.1 s  e ricalcola la priorità *ogni secondo*.

# Lo scheduling in UNIX (BSD 4.3)

Le priorità variano tra 0 (massima) e 127 (minima):

- da 0 a 49 per i processi che eseguono in modo kernel
- da 50 a 127 per i processi in modo utente.

Ogni 40 ms si ricalcala la priorità dei processi in modo utente secondo la seguente relazione:

p_usrpri = PUSER + (p_cpu / 4) + 2 * p_nice

# Lo scheduling in UNIX (BSD 4.3)

$$p\_usrpri = PUSER + (p\_cpu / 4) + 2 * p\_nice$$

dove:

- p_usrpri  viene alla fine saturato a 127
- PUSER = 50 (base priority for user mode execution)
- p_nice  varia tra -20 e 20, default 0
- p_cpu  incrementato ad ogni tick in cui il processo viene trovato in esecuzione;
- un correttivo, applicato ogni 1 s, fa decadere il 90% di p_cpu in circa 5 s
- un correttivo diminuisce p_cpu per i processi a lungo sospesi.

# Lo scheduling in UNIX (3)

- Viene usato il meccanismo di *time-out:*
  ogni quanto di tempo (0.1 … 1 s) l'interruzione di clock mette in funzione una procedura che:
  - cambia il contesto,
  - ricalcola le priorità e
  - predispone il clock per essere nuovamente chiamata.

# Lo scheduling in UNIX (4)

Un processo sospende la propria esecuzione tramite la primitiva del kernel *sleep* che ha come parametro l'indirizzo di una struttura dati del kernel relativa ad un *event* che il processo attende prima di risvegliarsi.

Quando si verifica *event*, il nucleo provvede a risvegliare *tutti* i processi in attesa di *event*. I processi vengono inseriti nella coda dei processi pronti per essere scelti dal meccanismo di scheduling.

# Lo scheduling in UNIX (5)

Possono nascere "condizioni di corsa" relative al meccanismo degli eventi:

- se un processo decide di sospendersi in attesa di un evento e l'evento si verifica prima che il processo completi la primitiva *sleep*, il processo rimane in attesa indefinita (deadlock). (Non c'é memoria associata agli eventi).

Una soluzione al problema consiste nell'impedire all'evento di verificarsi durante l'esecuzione della primitiva (innalzando la priorità hardware della CPU in modo che non si possano verificare interruzioni).

# Sistemi real-time

- In un *sistema real-time* la correttezza della elaborazione dipende sia dalla sua correttezza logica sia *dall'istante in cui il risultato viene generato*. Il mancato rispetto dei vincoli temporali equivale ad un guasto del sistema (*system failure*).

- Garantire la correttezza del comportamento temporale richiede che il sistema sia *altamente predicibile*. Le esigenze di tempo reale sono spesso in conflitto con gli usuali requisiti di efficienza nell'uso delle risorse.

- Sistemi *soft-real-time*: i vincoli sui tempi di risposta sono espressi come distribuzioni statistiche e scostamenti massimi ammessi.
- Sistemi *hard-real-time*: i vincoli devono essere rispettati in modo rigoroso.

- Nei *sistemi real-time organizzati a processi* tipicamente sono presenti sia *processi critici*, che devono soddisfare i vincoli temporali, che *processi non critici*, eseguiti con algoritmi di scheduling convenzionali (es. FCFS) negli intervalli di tempo residui.