Sistemi Operativi

Scheduling dei processi

Scheduling dei processi

- Se più processi sono eseguibili in un certo istante il sistema deve decidere quale eseguire per primo
- La parte del sistema operativo che prende questa decisione è lo

  Scheduler
- e l'algoritmo che utilizza è chiamato

  Algoritmo di scheduling

Scheduling dei processi

- Ogni processo è unico ed imprevedibile quindi per evitare tempi troppo lunghi di esecuzione di un processo molti sistemi operativi prevedono un

  Timer o Clock

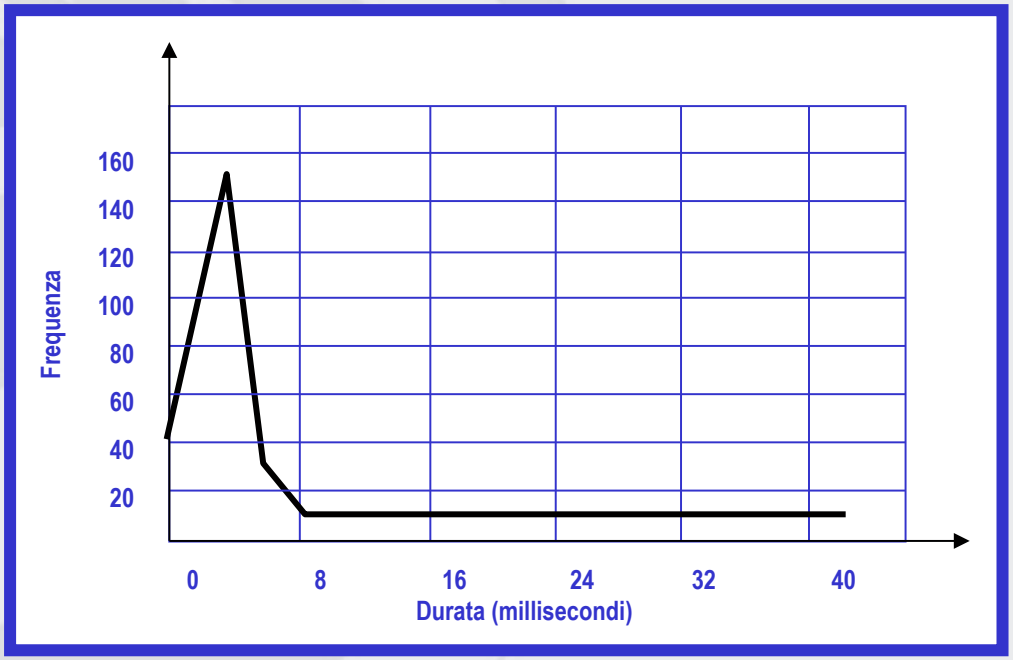
- Ad ogni interrupt del clock il sistema operativo decide se il processo può continuare oppure se deve essere sospeso per permettere l'esecuzione di un diverso processo presente in memoria
- La strategia che consente di sospendere temporaneamente processi che sono logicamente eseguibili viene detta

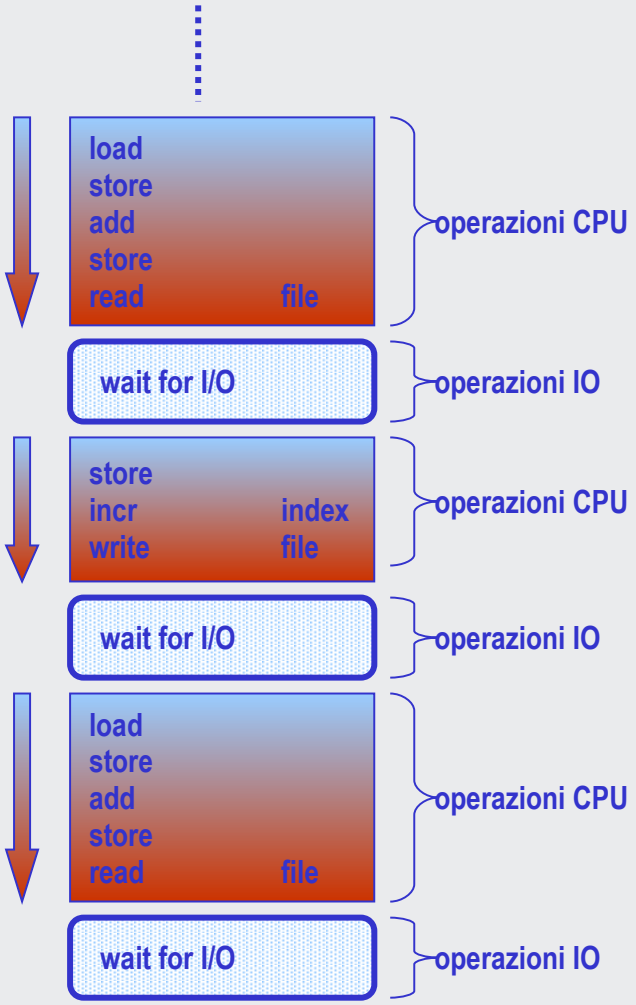
  Preemptive scheduling

  (scheduling con diritto di prelazione)

- In contrapposizione al metodo di esecuzione a completamento (tipico dei sistemi batch) o non preemptive scheduling (senza diritto di prelazione)

CPU e I/O Burst





Scheduler della CPU

- Seleziona fra i processi in memoria quelli pronti per l'esecuzione e assegna la CPU a uno di essi
- La decisione di scheduling della CPU può avvenire quando un processo:
  1. Passa da in esecuzione a in attesa
  2. Passa da in esecuzione a pronto
  3. Passa da in attesa a pronto
  4. Termina
- Lo Scheduling 1 o 4 è *non-preemptive*
- Lo Scheduling 2 o 3 è *preemptive*

Dispatcher

- Il Dispatcher è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler; l'operazione coinvolge:
  - Il cambio di contesto
  - Il passaggio al modo utente
  - Il salto alla giusta posizione del programma utente
- Il tempo necessario è chiamato latenza di dispatch (dovrebbe essere il più piccolo possibile)

Criteri di valutazione

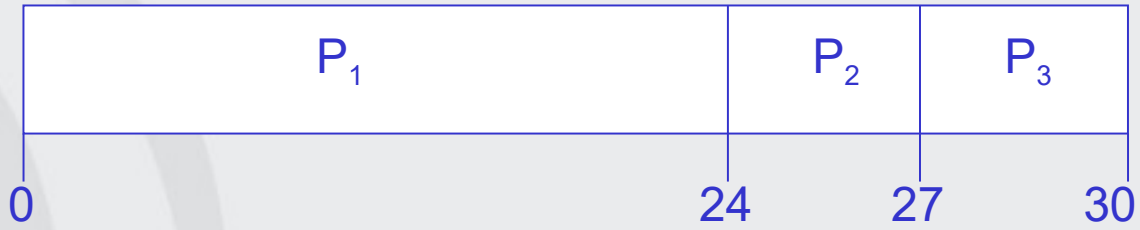
Un buon algoritmo di scheduling dovrebbe soddisfare i seguenti criteri:

- **Equità**: ogni processo deve avere a disposizione una corretta quantità di tempo di CPU
- **Efficienza**: la CPU dovrebbe essere utilizzata il 100% del tempo
- **Tempo di completamento**: deve essere il più piccolo possibile
- **Tempo di risposta**: deve essere minimizzato per gli utenti interattivi
- **Tempo di attesa**: il tempo trascorso nella coda di attesa deve essere minimizzato
- **Throughput**: occorre massimizzare il numero di job processati per unità di tempo

essendo il tempo di CPU finito alcuni di questi obiettivi non sono compatibili fra di loro

First Come First Served (FCFS) Scheduling

- I processi arrivano nell'ordine: P1 , P2 , P3
  - Il diagramma di Gantt corrispondente è:

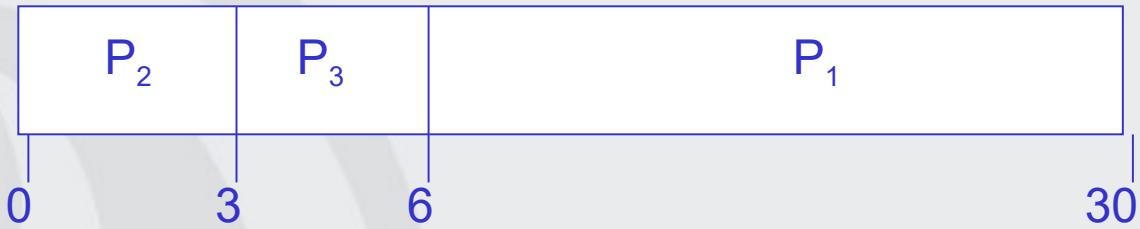


Processo	Burst Time
P_1	24
P_2	3
P_3	3

  - Tempi di attesa: P1 = 0; P2 = 24; P3 = 27
  - Tempo di attesa medio: (0 + 24 + 27)/3 = 17

FCFS Scheduling

- Se invece l'ordine è: P2 , P3 , P1
  - Si ha:



  - Tempi di attesa: P1 = 6; P2 = 0; P3 = 3
  - Tempo di attesa medio: (6 + 0 + 3)/3 = 3
- Il risultato è migliore
  - I processi brevi è meglio che vengano eseguiti per primi

Shortest job first

- Algoritmo particolarmente indicato per l'esecuzione batch dei job per i quali i tempi di esecuzione sono conosciuti a priori
- Lo scheduler dovrebbe utilizzare questo algoritmo quando nella coda di input risiedono job di uguale importanza
- SJF garantisce sempre il minimo tempo medio di risposta sarebbe utile estenderlo all'esecuzione dei processi interattivi
- Se per i processi interattivi si considera l'esecuzione di ogni comando come un singolo job, si ottiene il minimo tempo di risposta eseguendo il più breve per primo
  - **Problema:** determinare quale tra i processi eseguibili è il più breve
  - **Soluzione:** uso di stime basate sul comportamento passato ed esecuzione del processo con il minor tempo di esecuzione stimato

SJF

- Esistono due diversi schemi:
  - nonpreemptive – il processo in esecuzione non può essere interrotto
  - preemptive – se arriva un nuovo processo con un CPU burst atteso più breve del tempo rimanente per il processo corrente, il nuovo processo ottiene la CPU. Il criterio è noto anche come Shortest-Remaining-Time-First (SRTF)

Invecchiamento (aging)

- tecnica per la stima del valore successivo in una serie basata sul calcolo della media pesata del valore corrente misurato e la stima precedente

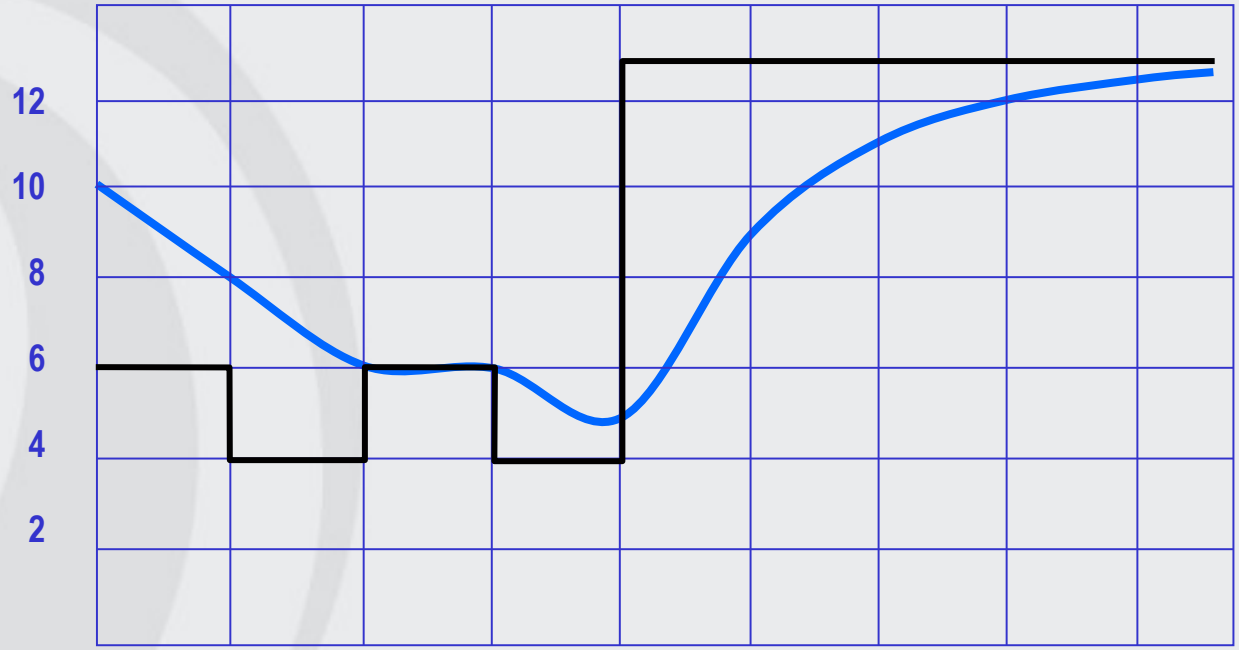
Ad esempio

$$T_{st} = \alpha T_{mis} + (1-\alpha) T_{st}$$

Con $\alpha = 0.5$

$$T_0,\ T_0/2 + T_1/2,\ T_0/4 + T_1/4 + T_2/2,\ T_0/8 + T_1/8 + T_2/4 + T_3/2,\ \ldots$$

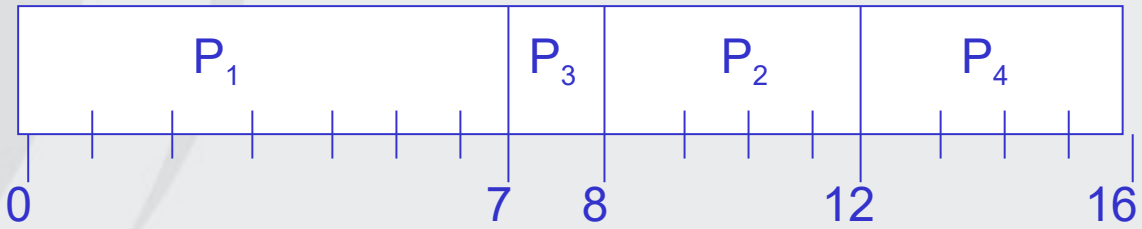
Predizione del successivo CPU burst



Tempo effettivo		6	4	6	4	13	13	13	13
Tempo predetto	10	8	6	6	5	9	11	12	

SJF Non-Preemptive

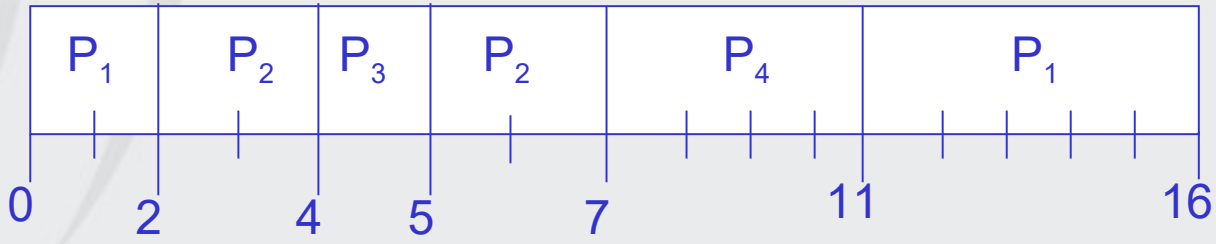
Processo	Tempo di arrivo	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



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

SJF Preemptive

Processo	Tempo di arrivo	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



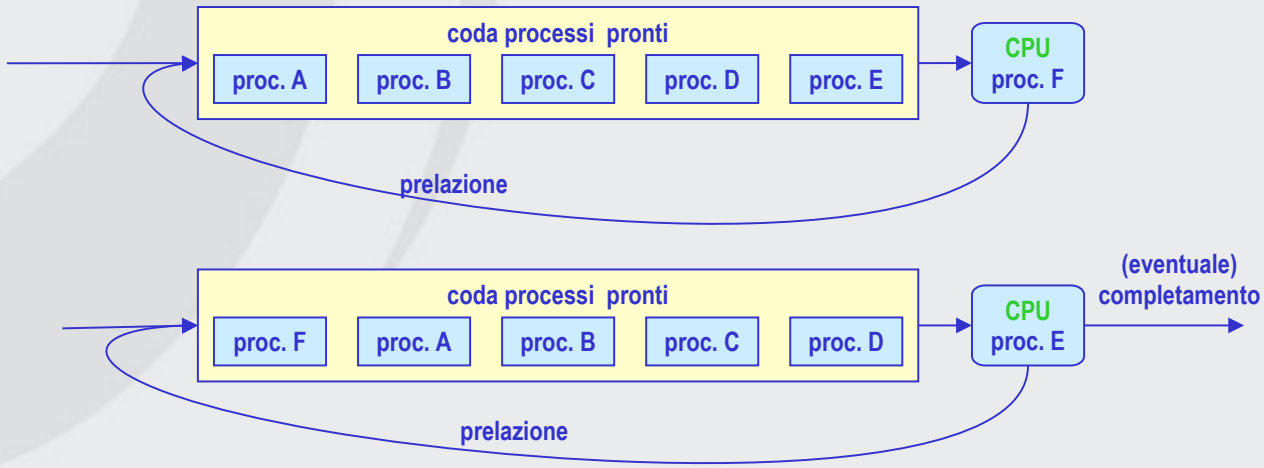
Tempo di attesa medio: (9 + 1 + 0 +2)/4 = 3

Scheduling round-robin

- Ad ogni processo viene assegnato un intervallo di tempo di esecuzione prefissato denominato

  Quanto (Time slice)

- se al termine di questo intervallo di tempo il processo non ha ancora terminato l'esecuzione, l'uso della CPU viene comunque affidato ad un diverso processo
- ogni processo ha uguale priorità di esecuzione



Context switch

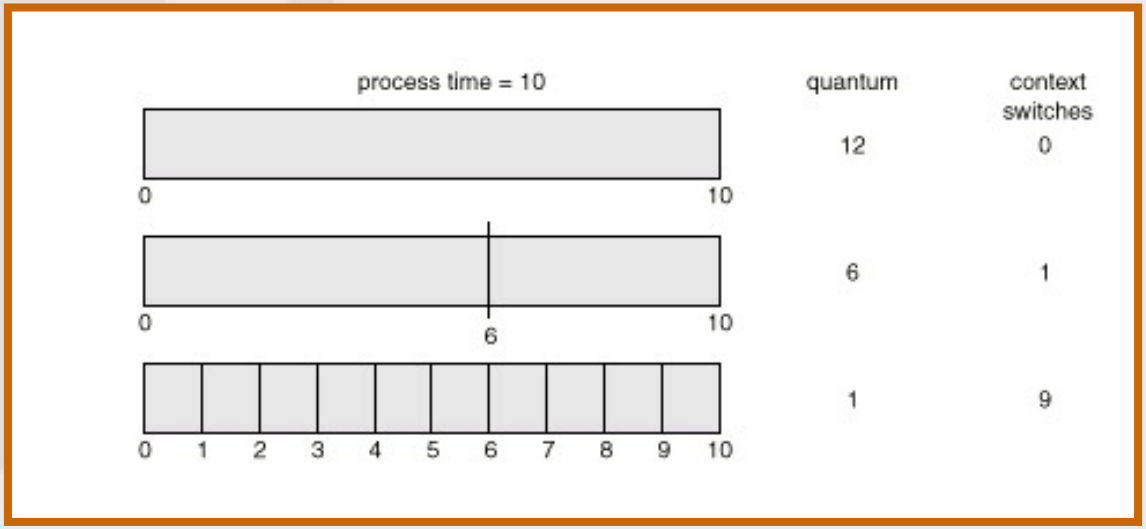
- **Il passaggio dell'esecuzione da un processo ad un altro richiede tempo per il salvataggio ed il caricamento dei registri e delle mappe di memoria, aggiornamento di tabelle e liste ... Tale operazione viene chiamata:**

  Context switch

- **la durata del quanto di tempo necessaria**
  - **non deve essere troppo breve per evitare**
    - molti context switch tra i processi
    - riduzione dell'efficienza della CPU
  - **non deve essere troppo lunga per evitare tempi di risposta lunghi a processi interattivi con tempo di esecuzione breve**

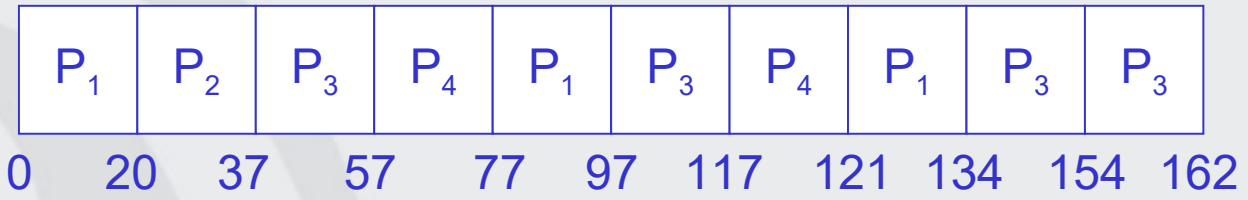
Context switch

- Supponiamo di utilizzare un time slice di 20 ms e che il cambiamento di contesto richieda 5 ms
  - Il tempo “sprecato” è un quarto del tempo utile
  - Con un time slice di 500 ms solo l' 1% viene “sprecato” ma se vi sono 10 utenti nel sistema si possono verificare attese fino a 4.5s



RR con Time Quantum = 20

- Il diagramma di Gantt (trascurando il CS) è:



  - Normalmente, si ottiene un tempo di turnaround (completamento) medio più alto che SJF, ma una migliore risposta

Processo	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

Scheduling a priorità

- Ad ogni processo viene assegnata una priorità
- La CPU viene assegnata al processo eseguibile con la priorità maggiore

  Per evitare che processi con una priorità alta vengano eseguiti indefinitamente, lo scheduler decrementa la priorità del processo in esecuzione ad ogni interrupt del clock

  Avviene un context switch quando la priorità del processo è minore di quella del processo successivo con priorità più alta
- Le priorità possono essere assegnate staticamente o dinamicamente

È utile raggruppare i processi in

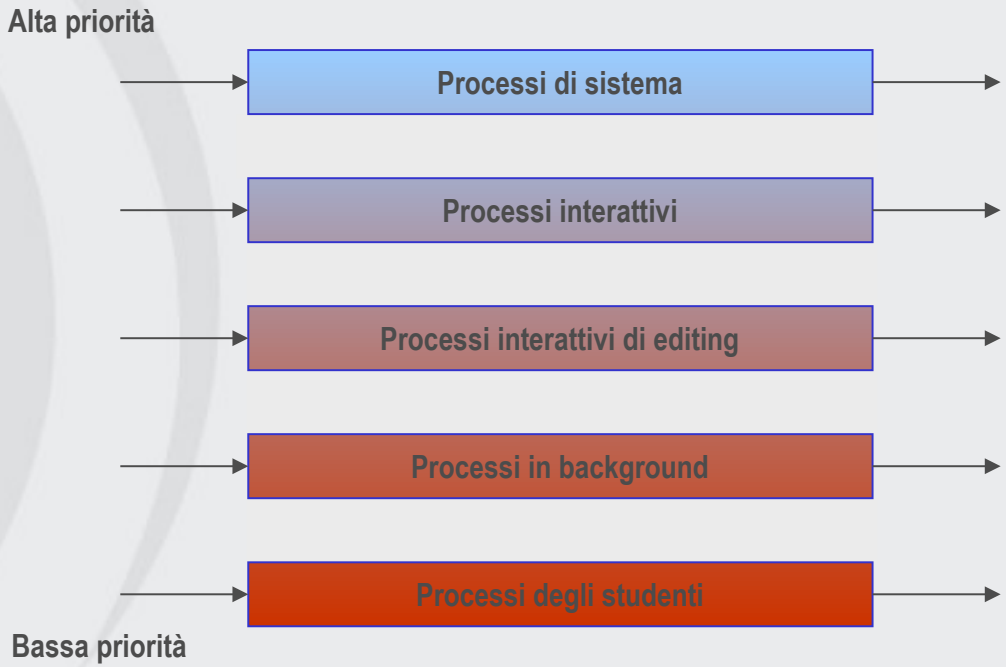
Classi di priorità

e utilizzare scheduling a priorità tra le classi e scheduling round robin all'interno di una classe

Code multilivello

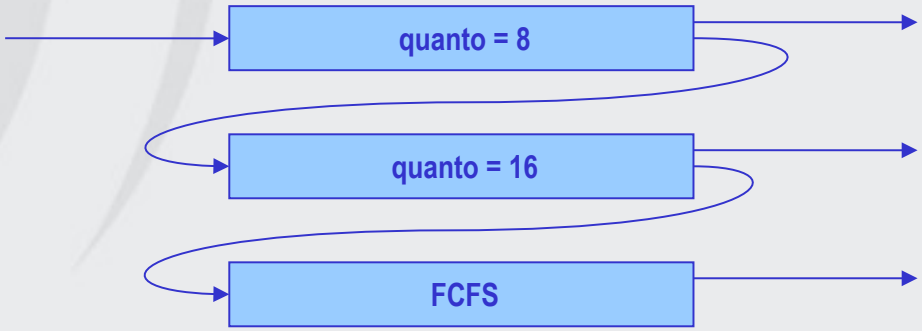
- La coda dei processi pronti può essere divisa in code separate:
  - Processi in foreground (interattivi)
  - Processi in background (batch)
- Ogni coda può utilizzare propri algoritmi
  - Foreground: round robin
  - Background: FCFS
- Occorre decidere anche uno scheduling fra le code
  - Priorità fissa
  - Time slice: ogni coda ha una sua percentuale (per esempio 80% processi interattivi, 20% processi batch)

Code multiple statiche



Code multiple dinamiche

- Un processo può spostarsi fra una classe e l'altra
  - I processi nella classe più alta vengono eseguiti per un quanto, quelli nella seconda per due quanti, quelli nella successiva per quattro quanti e così via
  - Quando un processo ha utilizzato i quanti ad esso assegnati, viene passato alla classe inferiore
  - Con questo metodo i processi lunghi scendono nelle code di priorità per dare la precedenza all'esecuzione dei processi interattivi brevi



Scheduling garantito

- Un approccio di scheduling completamente differente è quello di fare promesse reali all'utente e poi lasciare che si gestiscano
- Promessa: se ci sono n utenti connessi, ogni utente riceverà 1/n della potenza di CPU
- Per mantenere la promessa il sistema deve tenere traccia:
  - di quanto tempo di CPU un utente ha utilizzato per i suoi processi dopo la procedura di login e anche quanto tempo è passato dal login
  - del tempo di CPU che spetta a ogni utente (il tempo passato dal login diviso per n
- La priorità sarà calcolata in base al rapporto tra il tempo di CPU effettivamente utilizzato da un utente e il tempo che gli sarebbe spettato
  - L'algoritmo consiste nell'eseguire il processo con il rapporto minore finché il suo rapporto raggiunge quello del più vicino competitore

Scheduling garantito

- Questa idea può essere applicati ai sistemi Real time nei quali ci sono vincoli di tempo stretti da rispettare
- L'algoritmo consiste nell'eseguire il processo che rischia maggiormente di non rispettare le scadenze

Scheduling a estrazione

- Ad ogni processo si danno un certo numero di *biglietti* della lotteria (tempo di CPU)
- Sostituisce la priorità
- Reagisce velocemente ai cambiamenti
- I processi possono scambiarsi i *biglietti* (tempo di CPU)

Realtime scheduling

- **Hard realtime system:**
  - **Processi critici terminano entro un tempo stabilito**
    - Viene scelto il processo con scadenza più vicina
  - **Spesso i processi sono periodici**

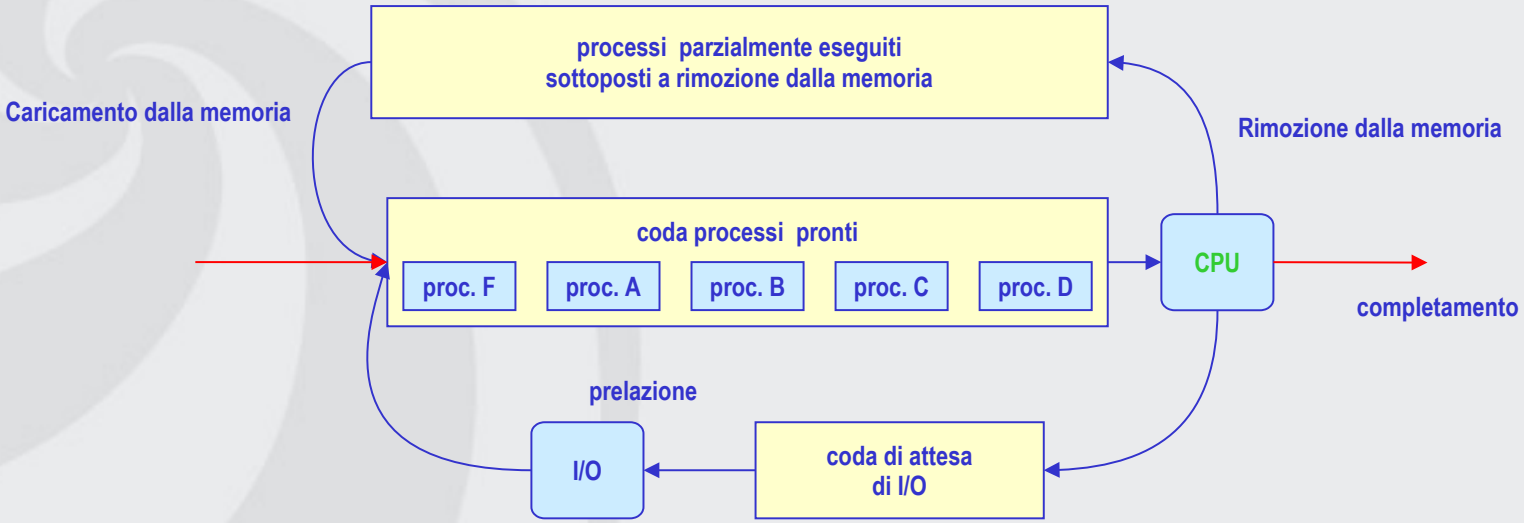
$$\sum_i \frac{T_{\max}(P_i)}{T_{per}(P_i)} < 1$$

- **Soft realtime:**
  - **I processi critici hanno una priorità maggiore degli altri**

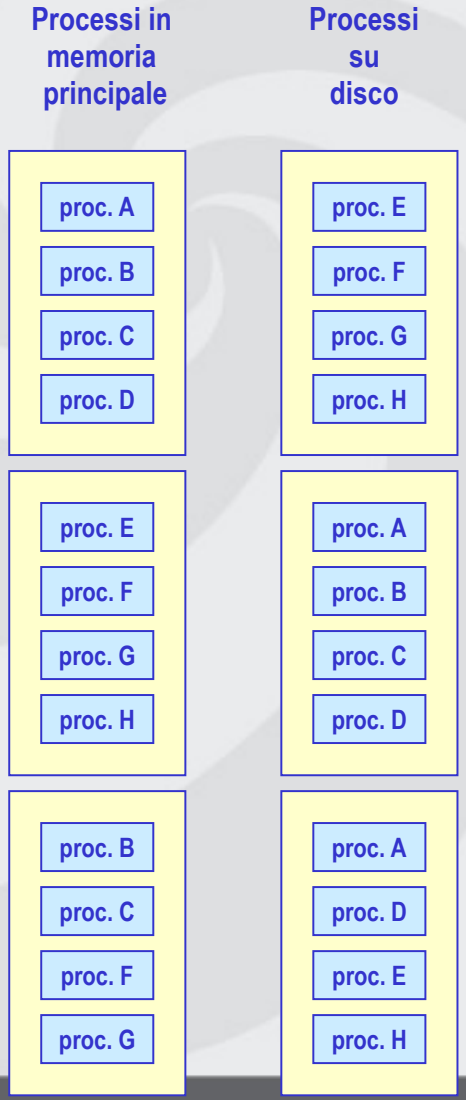
Scheduling a due livelli

- Se la memoria principale è insufficiente, alcuni processi eseguibili devono essere mantenuti su disco
- In questo caso lo scheduling dei processi comporta situazioni con tempi di switching molto diversi per i processi che risiedono su disco e i processi in memoria
- Un modo pratico per gestire questa situazione è l'utilizzo di uno scheduler a due-livelli con due tipi di scheduler:
  - scheduler di medio termine - si occupa degli spostamenti dei processi tra memoria e disco; rimuove i processi che sono stati in memoria per un tempo sufficiente e carica in memoria i processi che sono stati su disco a lungo
  - scheduler di breve termine - si occupa dell'esecuzione dei processi che sono effettivamente in memoria
- Soprattutto nei sistemi batch si parla di un terzo tipo di scheduler:
  - Scheduler di lungo termine – sceglie quali lavori (job) mandare in esecuzione

Scheduling a medio termine



Scheduling a medio termine



- I criteri di decisione dello scheduler di alto livello sono:
  - il tempo passato dall'ultimo spostamento da o in memoria
  - quanto tempo di CPU è stato assegnato al processo
  - la grandezza del processo (non è conveniente spostare processi piccoli)
  - priorità del processo
- l'algoritmo di scheduling può essere uno di quelli visti precedentemente (round robin, a priorità ecc.)

Scheduling di Windows 2000

- Usa un algoritmo basato su priorità e prelazione
  - esistono 32 livelli di priorità (una coda per ogni livello)
  - vi è una classe a priorità variabile (1-15)
  - e una classe “real time” (16-31)
  - la classe 0 è usata solo dal thread di gestione della memoria
- Il “dispatcher” sceglie il processo (thread) a priorità più alta pronto per l’esecuzione
  - se nessuno è pronto viene eseguito lo “idle thread” (ciclo idle del sistema – uno per ogni processore logico)

	real-time	high	above normal	normal	below normal	idle priority
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 2000

- Se un processo a priorità variabile esaurisce il suo tempo, gli viene tolta la CPU e la sua priorità abbassata
- Se rilascia spontaneamente la CPU la priorità viene innalzata (soprattutto se è in attesa di un evento da tastiera)
- Vengono distinti processi in primo piano dai processi in sottofondo
- Un processo può essere interrotto da un nuovo processo “Real time” a più alta priorità

Scheduling di Linux

- Linux usa tre diversi criteri di scheduling
  - FIFO in tempo reale
  - Round robin in tempo reale
  - Timesharing
- Processi della prima classe sono interrotti solo da processi della stessa classe a più alta priorità
- Processi della seconda classe sono eseguiti per un quanto di tempo e poi rimessi in coda

Scheduling di Linux - Kernel 2.4

- Processi normali sono eseguiti in base ai loro crediti
  - Ad ogni interruzione generata dal clock il processo in esecuzione perde un credito
  - Se i suoi crediti sono 0 viene sostituito da un nuovo processo pronto
  - Se nessun processo pronto dispone di crediti, i crediti di tutti i processi sono aggiornati secondo la formula

$$\text{crediti} = \text{crediti}/2 + \text{priorità}$$

Scheduling di Linux- Kernel 2.4

- Un processo cede la CPU quando un processo a più alta priorità prima bloccato diventa pronto
- Dopo una fork() ogni processo eredita metà dei crediti
- La priorità normale in genere è posta a 20 quanti (ticks) cioè circa 210 ms

Scheduling di Linux- Kernel 2.6

- 140 livelli di priorità
  - 0 – 99 per i processi real time
  - 100 – 139 per i processi normali
- StaticPriority = 100 + nice + 20
- Calcolo del quanto:

$$\text{quanto} = \begin{cases} (140 - SP) \times 20 & \text{se } SP < 120 \\ (140 - SP) \times 5 & \text{se } SP \geq 120 \end{cases}$$

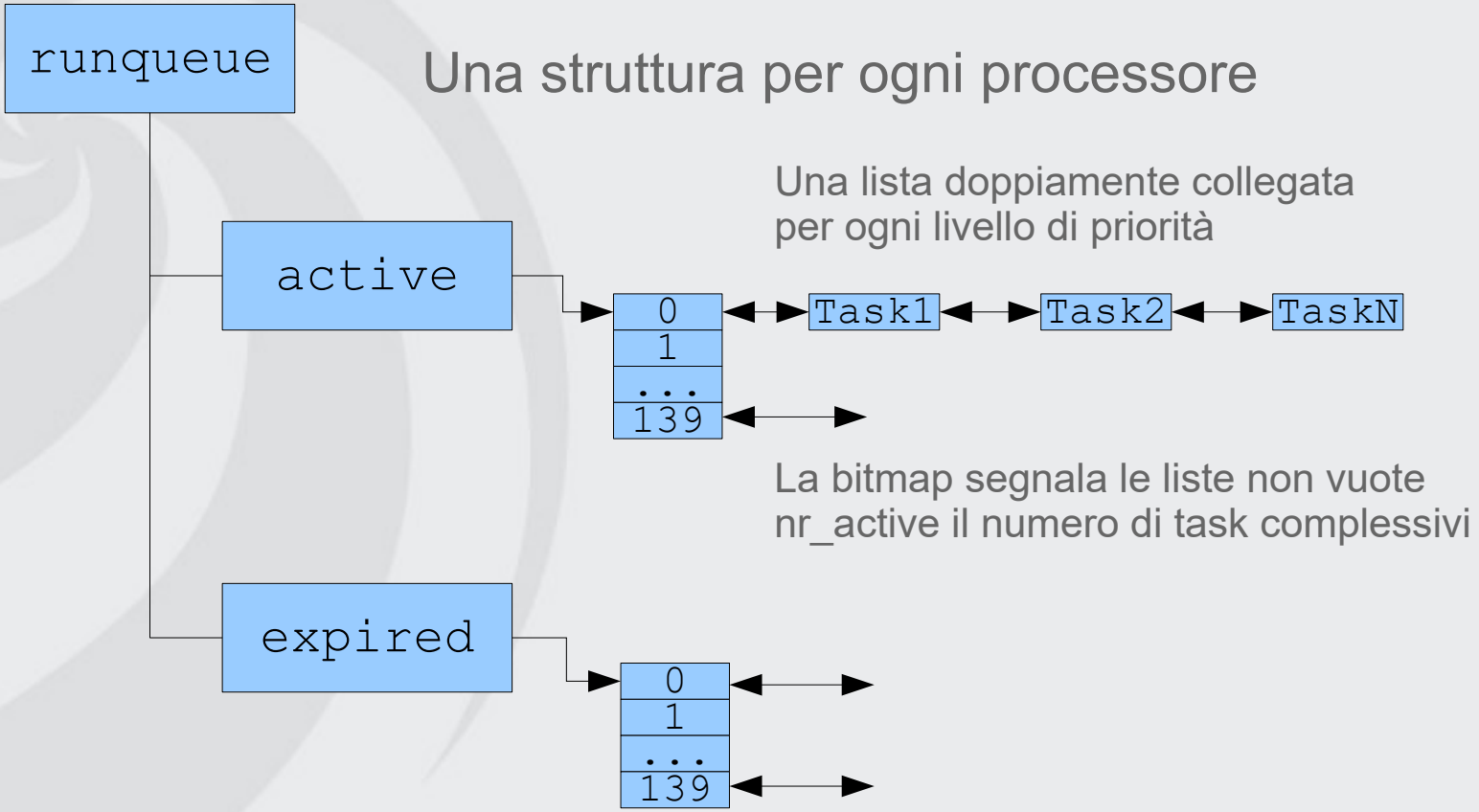
priorità statica		nice	quanto
Massima	100	-20	800 ms
Alta	110	-10	600 ms
Normale	120	0	100 ms
Bassa	130	10	50 ms
Minima	139	19	5 ms

Scheduling di Linux- Kernel 2.6

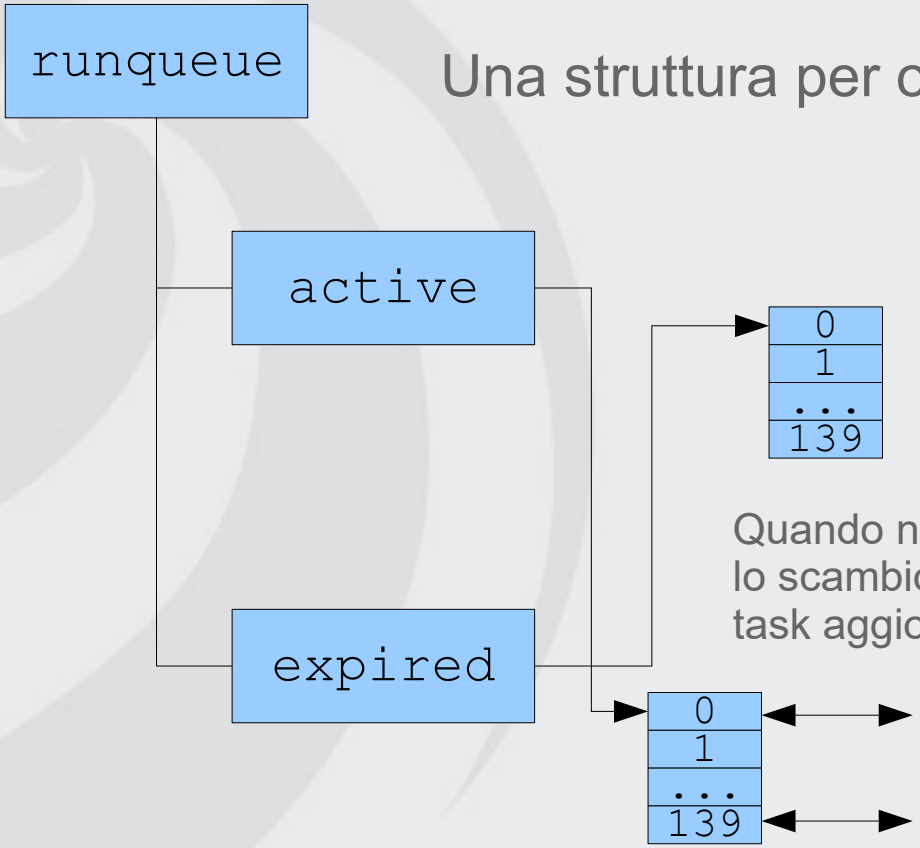
```
struct runqueue {
        struct prioarray *active;
        struct prioarray *expired;
        struct prioarray arrays[2];
};

struct prioarray {
        int nr_active; /* # Runnable */
        unsigned long bitmap[5];
        struct list_head queue[140]; // double linked list
};
```

Quando il numero di processi attivi scende a 0, i due array vengono scambiati, vi è una coda diversa per ogni processore

Runqueue


runqueue
Una struttura per ogni processore
Una lista doppiamente collegata
per ogni livello di priorità
active
0
1
...
139
Task1
Task2
TaskN
La bitmap segnala le liste non vuote
nr_active il numero di task complessivi
expired
0
1
...
139

Runqueue



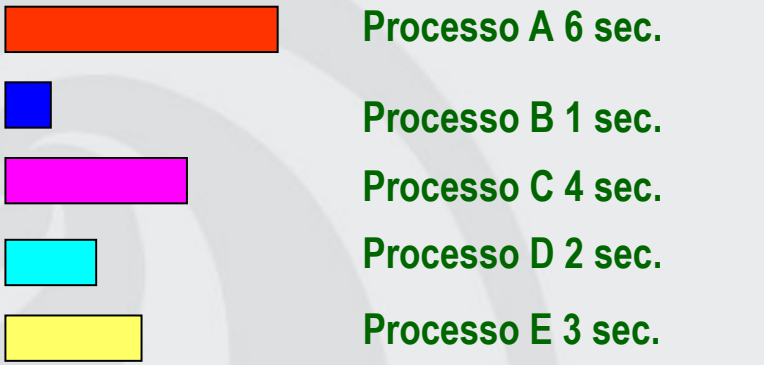
Una struttura per ogni processore

Quando nr_active diventa 0 viene fatto lo scambio delle liste con le priorità dei task aggiornate

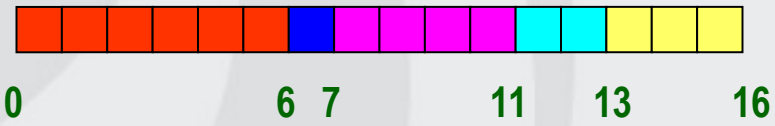
Esercizi

Scheduling dei processi

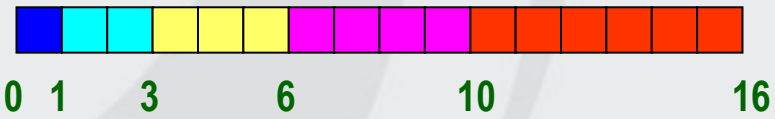
Tempo medio di completamento



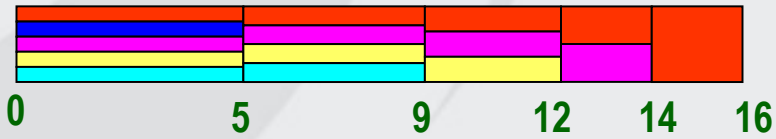
Si supponga che il CS sia trascurabile



FIFO → (6+7+11+13+16)/5=53/5=10.6

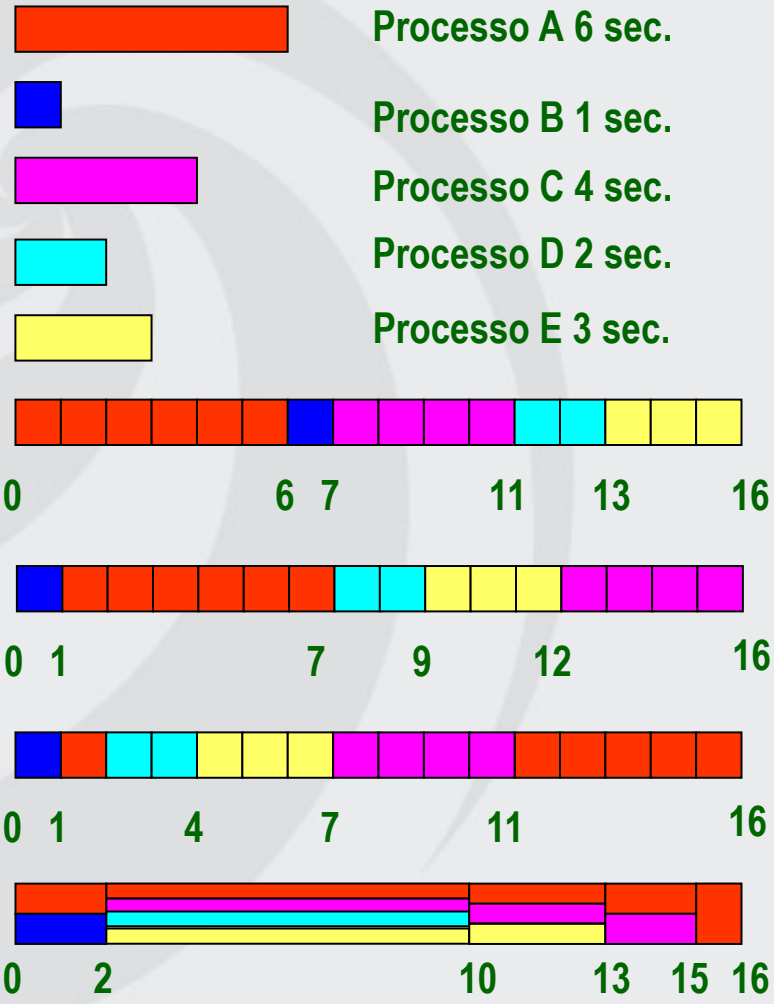


SJF → (1+3+6+10+16)/5=36/5=7.2



TS → (5+9+12+14+16)/5=56/5=11.2

Tempo medio di completamento



I processi C D E entrano dopo 2 sec.

FIFO → (6+7+9+11+14)/5=47/5=9.4

SJF → (1+7+7+10+14)/5=39/5=7.8

SJF → (1+2+5+9+16)/5=33/5=6.6
con possibilità di prelazione

TS → (2+8+11+13+16)/5=50/5=10.0

- In un sistema vi sono 3 processi:
  - Il primo entra all'istante 0, richiede quindi 25 ms di cpu, 50 di I/O, 30 di cpu, 60 di I/O, infine 50 di cpu (105 cpu, 110 I/O)
  - Il secondo entra all'istante 10, richiede quindi 20 ms di cpu, 50 di I/O, 10 di cpu, 40 di I/O, infine 10 di cpu (40 cpu, 90 I/O)
  - Il terzo entra all'istante 20 e richiede 240 ms di cpu
  - Il CS è di 5 ms, il quanto di tempo di 40 ms
  - Tempo totale di CPU 385 ms

Scheduling

0-25	Processo A	Pronto al tempo 75	BC
30-50	Processo B	Pronto al tempo 100	C
55-95	Processo C	Va in coda (200)	AC
100-130	Processo A	Pronto al tempo 190	CB
135-175	Processo C	Va in coda (160)	BC
180-190	Processo B	Pronto al tempo 230	CA
195-235	Processo C	Va in coda (120)	ABC
240-280	Processo A	Va in coda (10)	BCA
285-295	Processo B	Ha finito	CA
300-340	Processo C	Va in coda (80)	AC
345-355	Processo A	Ha finito	C
360-400	Processo C	(40)	
400-440	Processo C	Ha finito	

Coda al momento dello scheduling

Tempo di esecuzione rimanente

È l'unico processo, non vi è Context Switch

Per conferma 385 + 11*5 (CS) = 440

Considerare il caso in cui A e B usino lo stesso dispositivo di I/O

Priorità a chi termina I/O

0-25	Processo A	Pronto al tempo 75	BC
30-50	Processo B	Pronto al tempo 100	C
55-75	Processo C	Va in coda (220)	C
80-100	Processo A	Va in coda (10)	CA
105-115	Processo B	Pronto al tempo 155	CA
120-155	Processo C	Va in coda (185)	AC
160-170	Processo B	Ha finito	AC
175-185	Processo A	Pronto al tempo (245)	C
190-245	Processo C	Torna in coda (130)	C
250-290	Processo A	Va in coda (10)	CA
295-335	Processo C	Va in coda (90)	AC
340-350	Processo A	Ha finito	C
355-445	Processo C	Ha finito	

Per conferma 385 + 12*5 (CS) = 445

Considerare il caso in cui A e B usino lo stesso dispositivo di I/O

Unico dispositivo di I/O

0-25	Processo A	Pronto al tempo 75	BC
30-50	Processo B	Pronto al tempo 125	C
55-75	Processo C	Va in coda (220)	C
80-110	Processo A	Pronto al tempo 185	C
115-125	Processo C	Va in coda (210)	C
130-140	Processo B	Pronto al tempo 225	C
145-185	Processo C	Va in coda (170) C	
185-225	Processo A	Va in coda (10)	CA
230-240	Processo B	Ha finito	CA
245-285	Processo C	Va in coda (130)	AC
295-305	Processo A	Ha finito	
310-440	Processo C	Ha finito	

Per conferma 385 + 11*5 (CS) = 440

Chiamata di sistema fork

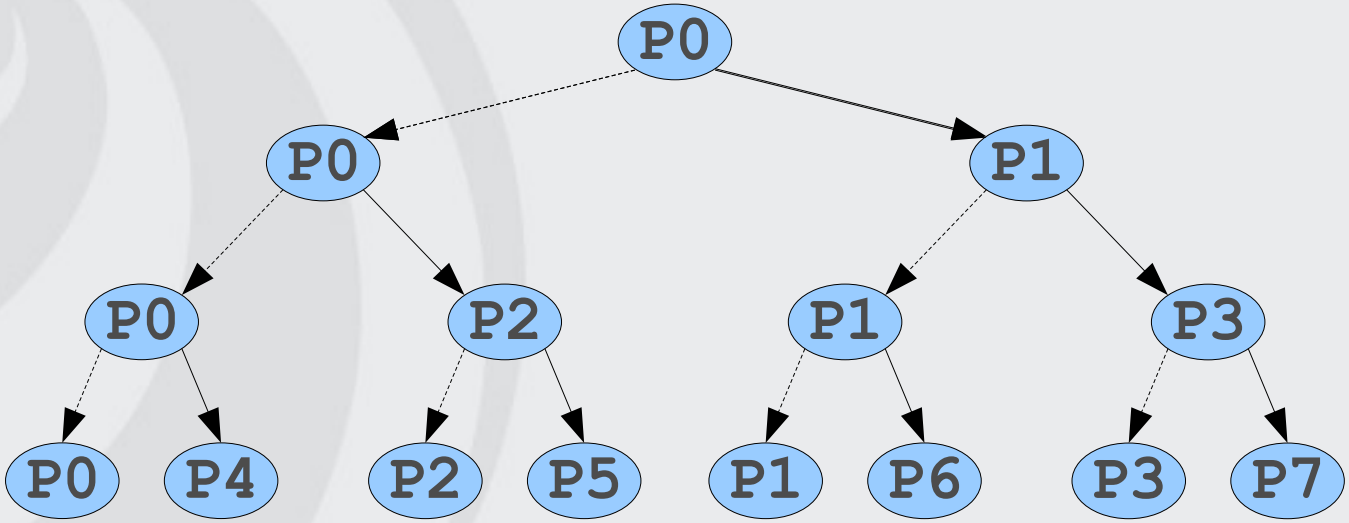
Quanti processi creano rispettivamente i due programmi (si conti anche il processo padre):

```
#define N 3

int main()
{
    int i, p;
    for(i=0; i<N; i++) {
       if(p=fork()) break;
    }
}
```

```
#define N 3

int main()
{
    int i, p;
    for(i=0; i<N; i++) {
       p=fork();
    }
}
```

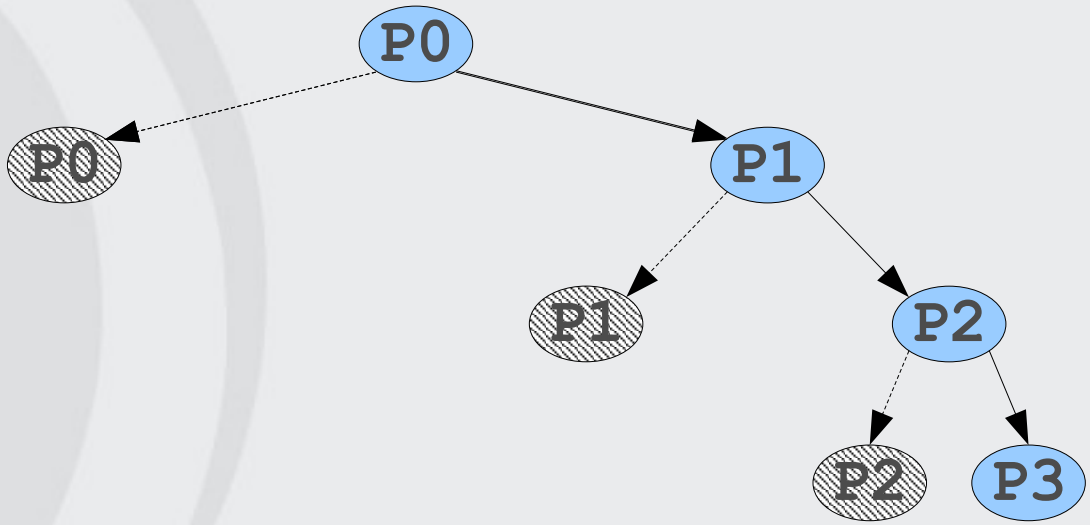
Chiamata di sistema fork

Seconda versione: 2^N



Chiamata di sistema fork

Prima versione: N + 1



Scheduling

Gli schedulatori round-robin mantengono normalmente una lista di tutti i processi pronti nella quale ciascun processo è presente una sola volta.
Cosa potrebbe succedere se questo vincolo non viene implementato e per quale motivo si potrebbe utilizzare un simile criterio?

Scheduling a lotteria

- **In un sistema vi sono 3 processi:**
  - **Il primo è un processo server, ogni servizio richiede 100 ms di CPU. Non possiede biglietti**
  - **Il secondo richiede prima 100 ms di cpu, poi si blocca in attesa di un servizio da parte del primo, infine usa altri 200 ms di cpu. Possiede 10 biglietti.**
  - **Il terzo richiede 300 ms di cpu. Possiede 10 biglietti.**
  - **Il 10% del tempo viene usato dal SO (sono compresi anche i CS)**
- **Mostrare uno schema temporale del sistema**

Scheduling a lotteria

Trascurando per semplicità il SO

Per i primi 200 ms i processi B e C ottengono mediamente il 50% del tempo (100 ms ognuno)

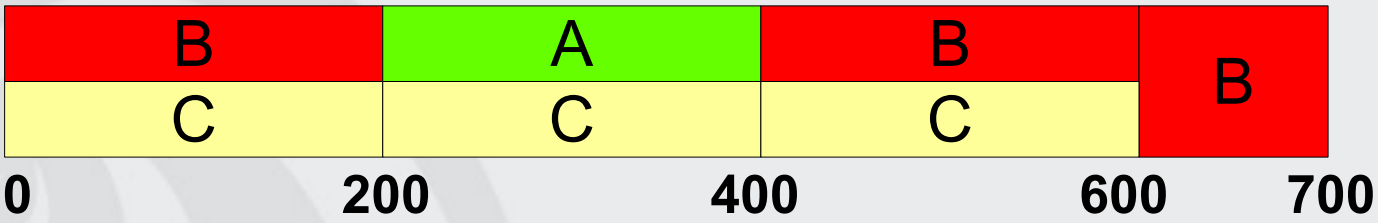
A questo punto B cede i biglietti ad A, per i successivi 200 ms il tempo è diviso fra A e C (100 ms ognuno)

A termina (tempo 400) e restituisce l'informazione richiesta (e i biglietti) a B che riprende l'esecuzione

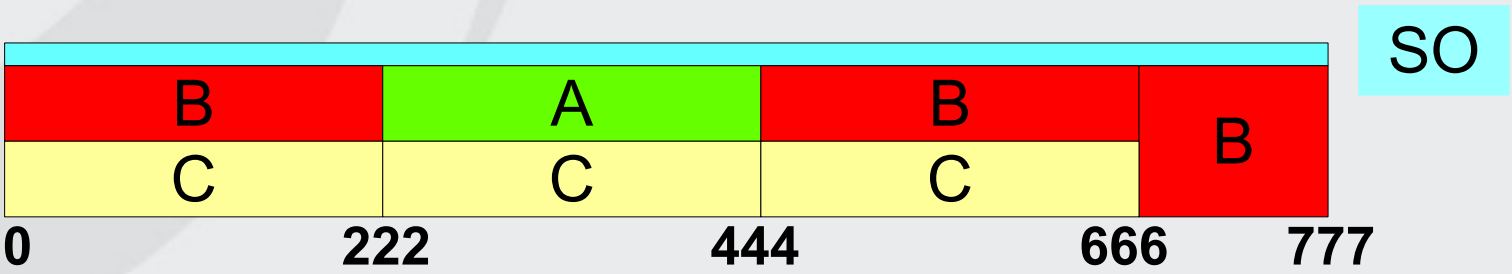
Per i successivi 200 ms i processi B e C ottengono mediamente il 50% del tempo (100 ms ognuno). Quindi C termina (al tempo 600).

B ha ancora bisogno di 100 ms di CPU che a questo punto è a sua completa disposizione (quindi termina al tempo 700)

Scheduling a lotteria



Considerando anche il SO tutti i tempi vanno moltiplicati per 1/0.9 (1.111111)



Scheduling a lotteria

- In un sistema vi sono 3 processi che richiedono 5 s:
  - Il primo possiede 30 biglietti
  - Il secondo e il terzo 10 biglietti.
  - Il 10% del tempo viene usato dal SO (sono compresi anche i CS)
- Mostrare uno schema temporale del sistema

Scheduling a lotteria

Trascurando per semplicità il SO

Per i primi 5*(5/3) s il processo A usa 3/5 (cioè 5s) del tempo e B e C 1/5 (5/3s) ognuno

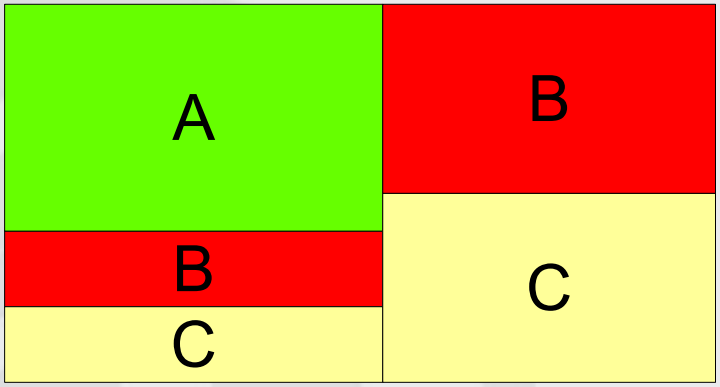
A questo punto A ha finito e B e C si spartiscono in modo eguale l'uso della CPU (hanno entrambi 10 biglietti) fino al termine (devono ancore utilizzare la CPU per 5-5/3=10/3s)

Dopo 20/3s terminano entrambi

Complessivamente sono trascorsi 25/3+20/3=15s (come dovuto 3*5)

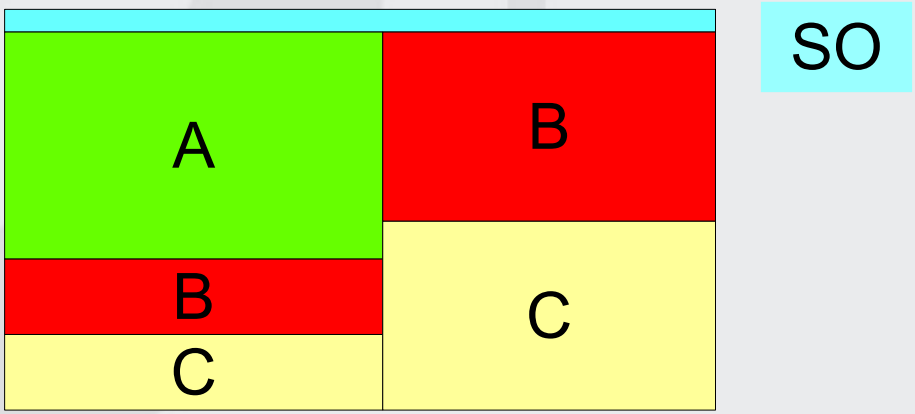
I tempi sono da compensare tenendo conto della frazione di uso da parte del SO (vedi esercizio precedente *1.111111)

Scheduling a lotteria



0 **25/3** **25/3+20/3=15**

Considerando anche il SO i tempi vanno moltiplicati per 10/9



0 **250/27** **150/9**

Scheduling a code multiple

- In un sistema vi sono 3 processi:
  - Il primo entra all'istante 0, richiede quindi 25 ms di cpu, 50 di I/O, 30 di cpu, 60 di I/O, infine 50 di cpu
  - Il secondo entra all'istante 10, richiede quindi 20 ms di cpu, 50 di I/O, 10 di cpu, 40 di I/O, infine 10 di cpu
  - Il terzo entra all'istante 20 e richiede 240 ms di cpu
- La schedulazione viene gestita tramite due code a priorità diversa
  - I processi entrano nella prima coda (alta priorità) e scendono nella seconda (bassa) se esauriscono il loro quanto di tempo
  - Entrambe le code sono gestite con schedulazione round-robin, la prima con quanto di tempo di 40 ms, la seconda di 80
  - Il context-switch è di 5 ms
- Mostrare uno schema temporale del sistema

Scheduling a code multiple

0-25	Processo A	Torna in coda 1 al tempo 75
30-50	Processo B	Torna in coda 1 al tempo 100
55-95	Processo C	Va in coda 2 (200)
100-130	Processo A	Torna in coda 1 al tempo 190
135-145	Processo B	Torna in coda 1 al tempo 185
150-230	Processo C	Torna in coda 2 (120)
235-245	Processo B	Ha finito
250-290	Processo A	Va in coda 2 (10)
295-375	Processo C	Torna in coda 2 (40)
380-390	Processo A	Ha finito
395-435	Processo C	Ha finito

Per conferma

105 (A) + 40 (B) + 240 (C) + 10*5 (CS) = 435

Se ci fosse diritto di prelazione fra le code a 185 C sarebbe sospeso e sostituito da B

Scheduling a code multiple

0-25	Processo A	Torna in coda 1 al tempo 75
30-50	Processo B	Torna in coda 1 al tempo 100
55-95	Processo C	Va in coda 2 (200)
100-130	Processo A	Torna in coda 1 al tempo 190
135-145	Processo B	Torna in coda 1 al tempo 185
150-185	Processo C	Torna in coda 2 (165)
190-200	Processo B	Ha finito
205-245	Processo A	Va in coda 2 (10)
250-330	Processo C	Torna in coda 2 (85)
335-345	Processo A	Ha finito
350-435	Processo C	Ha finito

Per conferma

105 (A) + 40 (B) + 240 (C) + 10*5 (CS) = 435

Considerare il caso in cui A e B usino lo stesso dispositivo di I/O

Scheduling a code multiple

0-25	Processo A	Torna in coda 1 al tempo 75
30-50	Processo B	Torna in coda 1 al tempo 125
55-95	Processo C	Va in coda 2 (200)
100-130	Processo A	Torna in coda 1 al tempo 190
135-145	Processo B	Torna in coda 1 al tempo 230
150-190	Processo C	Torna in coda 2 (170)
195-235	Processo A	Va in coda 2 (10)
240-250	Processo B	Ha finito
255-335	Processo C	Torna in coda 2 (80)
340-350	Processo A	Ha finito
355-435	Processo C	Ha finito

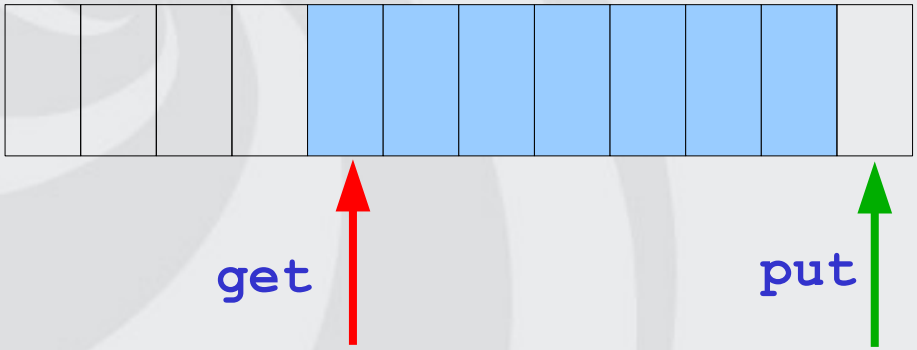
Per conferma

105 (A) + 40 (B) + 240 (C) + 10*5 (CS) = 435

Code fifo

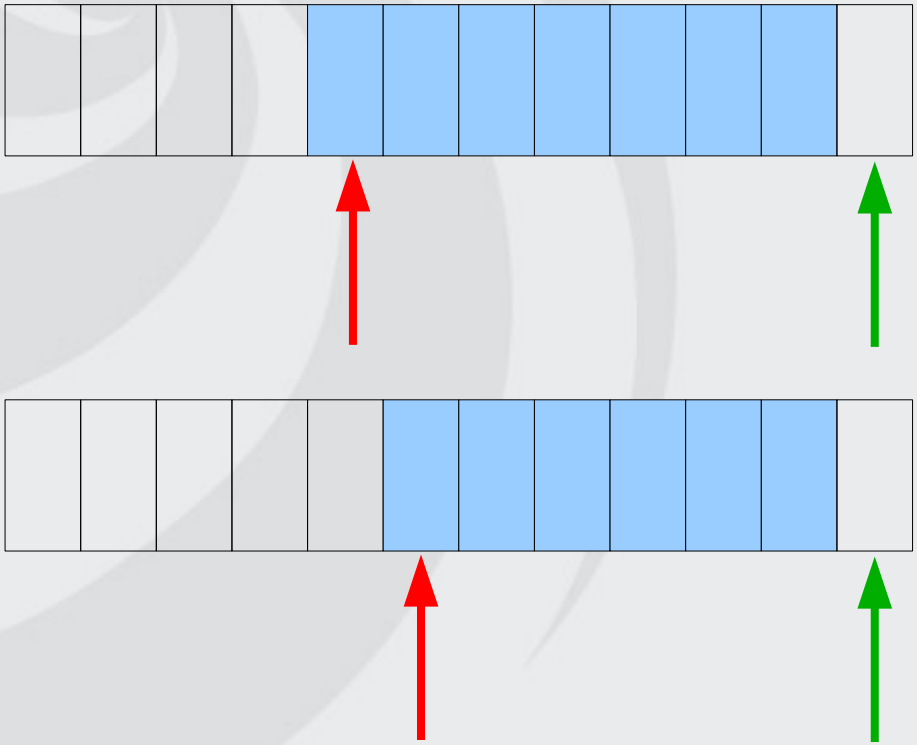
Una struttura dati per le code fifo
Bounded buffer

Bounded buffer - proprietà



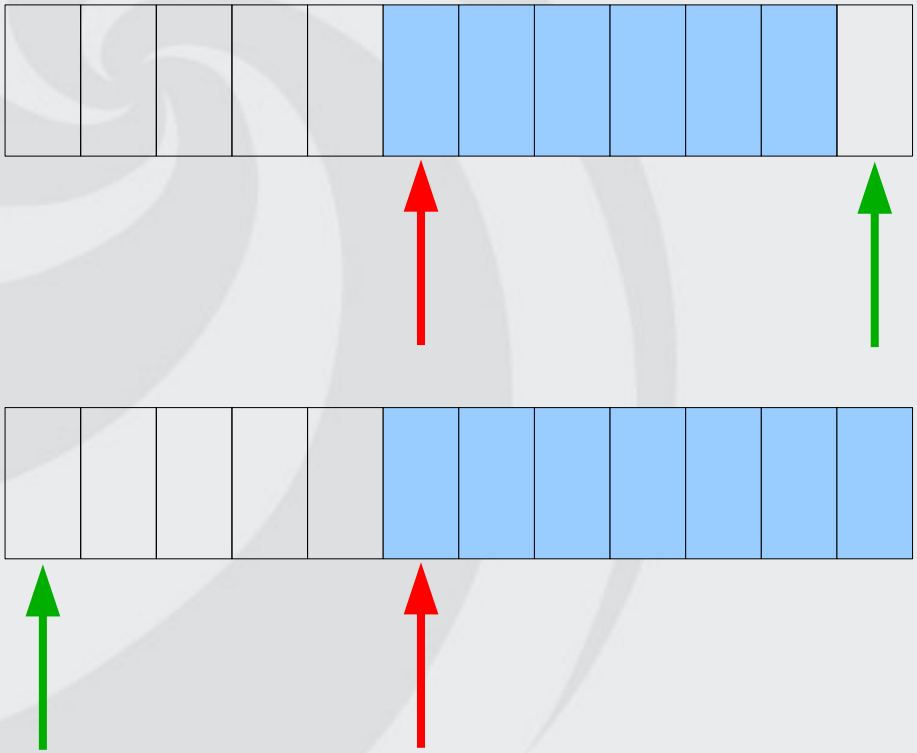
- Il buffer ha dimensione fissa
- Si utilizzano 2 indici (puntatori)
- L'inserimento (put) avviene nell'elemento puntato dall'indice di ***Inserimento***
- L'estrazione (get) avviene nell'elemento puntato dall'indice di ***Estrazione***
- L'aggiornamento avviene in un tempo costante (non dipende dal numero di dati memorizzati)

Estrazione

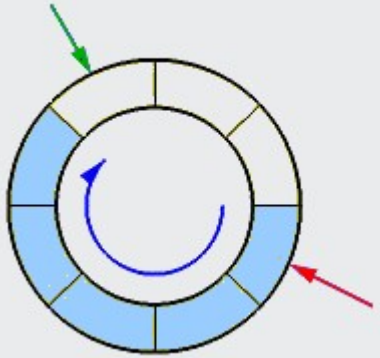


- Il puntatore avanza di una posizione

Inserimento



- Il puntatore avanza di una posizione
- Il buffer è considerato come un nastro chiuso



Esempio di codice java

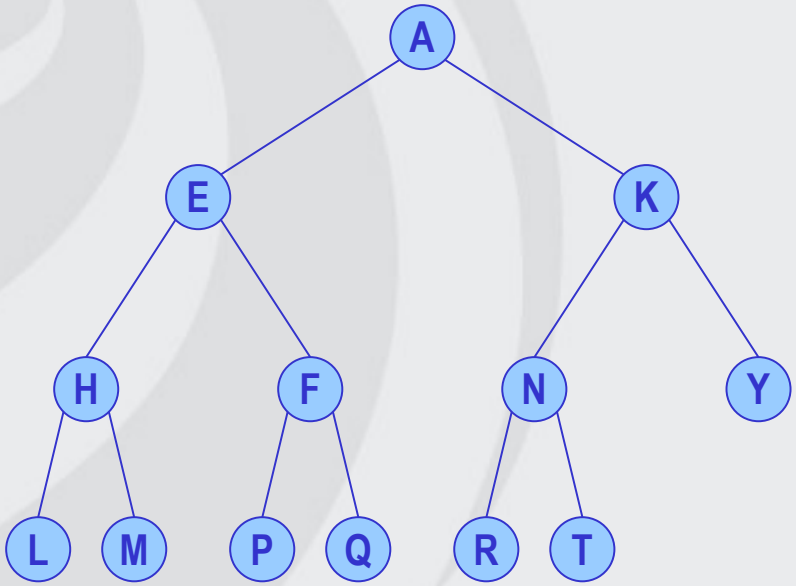
```
interface Buffer<T> {
    public void put(T object);
    public T get();

// metodi non necessari, ma utili
    public boolean isFull();
    public boolean isEmpty();
}

interface BoundedBuffer<T> extends Buffer<T> {}

interface PriorityQueue<T> extends Buffer<T> {}

interface Queue<T> extends Buffer<T> {}
```

Code con priorità

Una struttura dati per le code con priorità

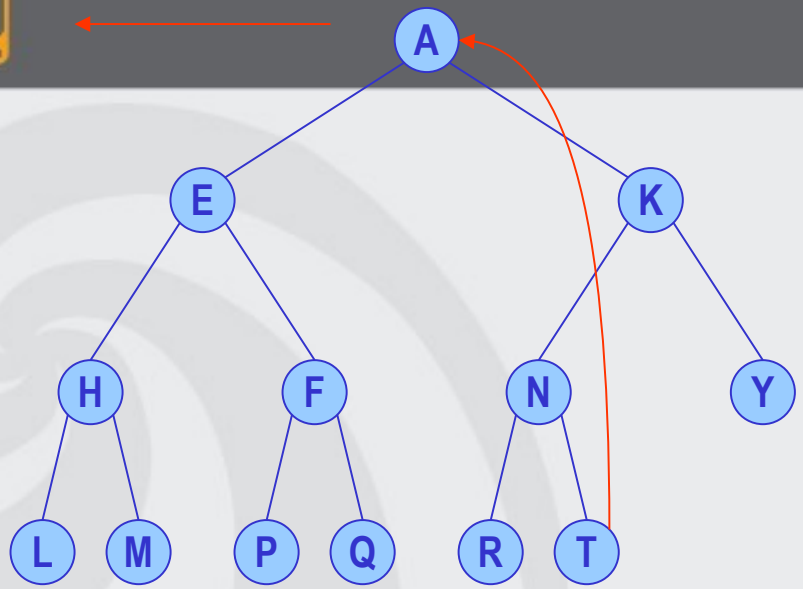
Heap tree

Heap tree

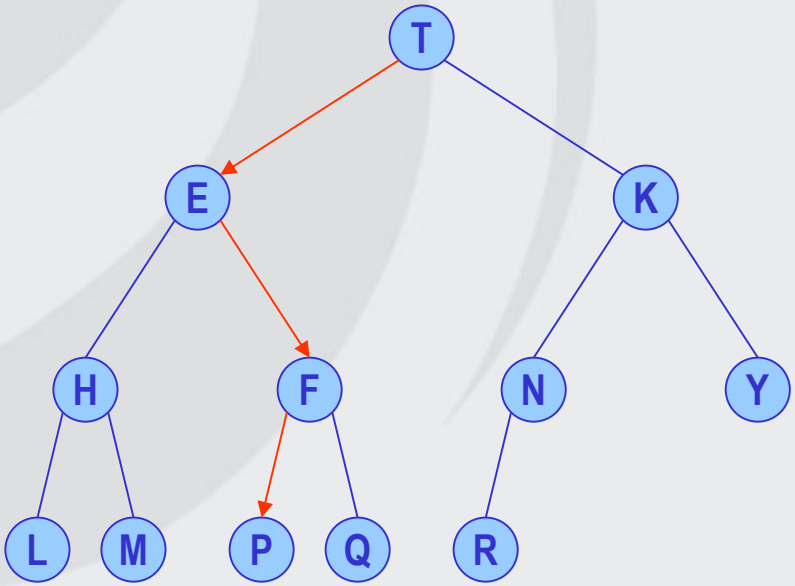


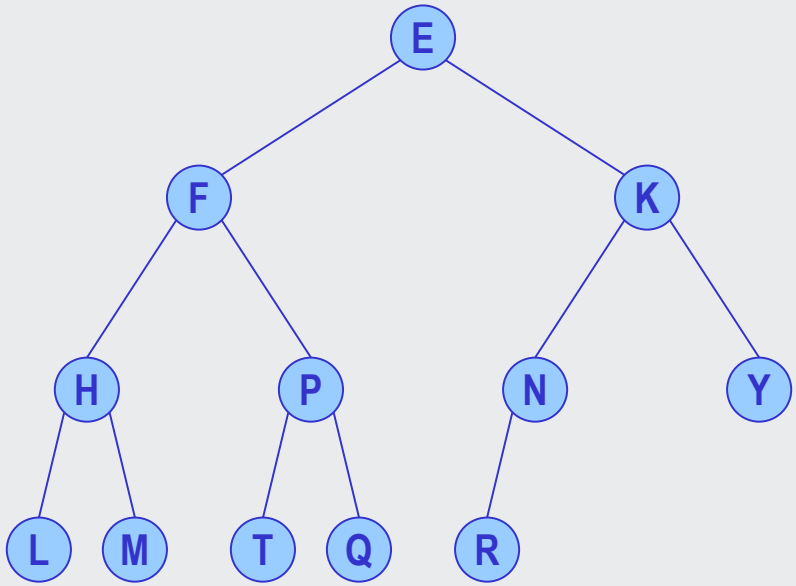
- Proprietà
  - Ogni nodo è maggiore (minore) di tutti gli elementi del sottoalbero di cui è radice
  - L'estrazione avviene alla radice
  - L'inserimento sul primo nodo libero a partire da sinistra dell'ultimo livello
  - L'aggiornamento avviene in un tempo proporzionale al logaritmo della dimensione

Estrazione

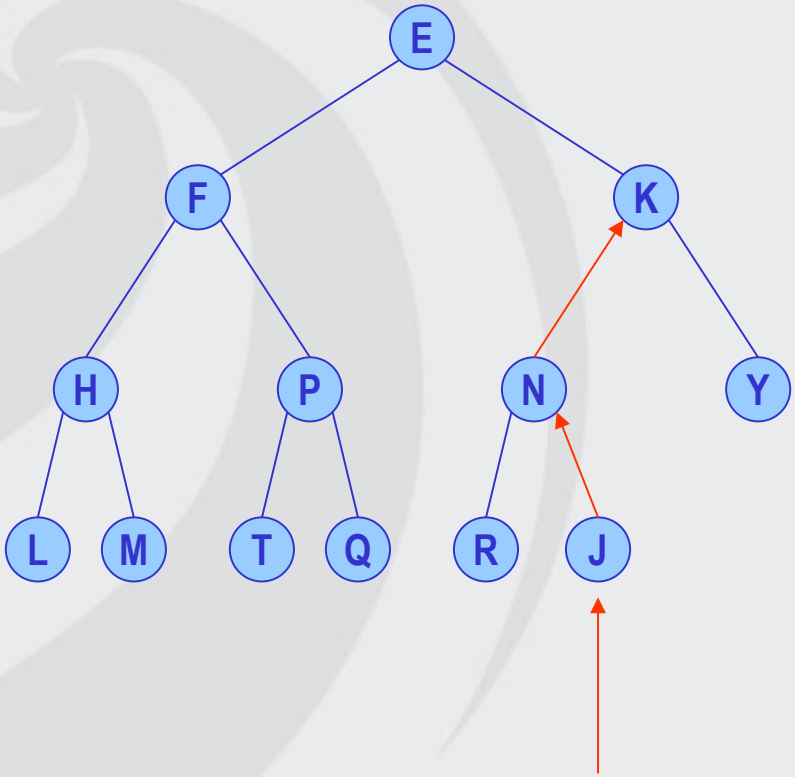


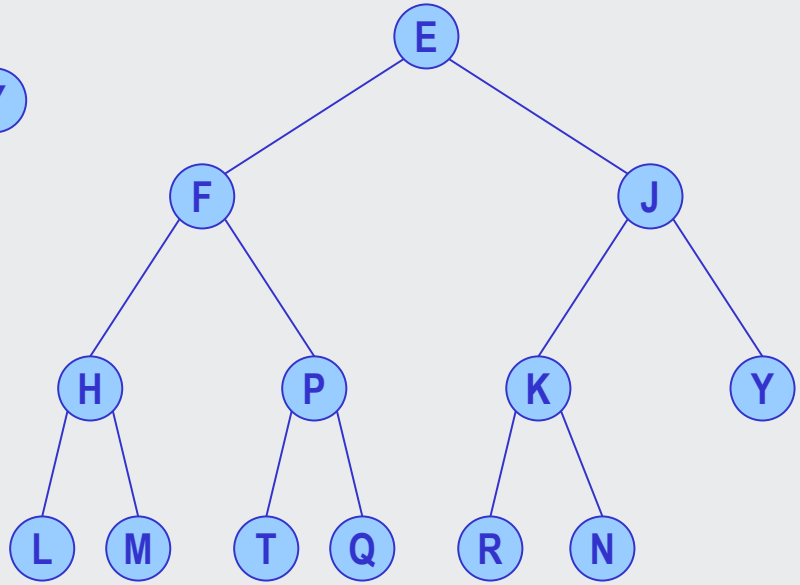
Il nodo più a destra dell'ultimo livello sostituisce la radice, quindi si aggiorna l'albero





Inserimento





Implementazione

- Può essere implementato tramite un vettore
  - La radice è l'elemento di indice 1
  - I figli di un nodo di indice i sono memorizzati nelle posizioni (2*i) e (2*i+1)
- Un albero di dimensione N occupa gli elementi del vettore da 1 a N
  - L'estrazione coinvolge l'indice 1
  - L'inserimento l'indice N+1
  - Non rimangono mai *buchi* nel vettore
- L'albero è riempito su tutti i livelli tranne eventualmente l'ultimo