Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2018-2019

Pietro Frasca

Lezione 13

Martedì 20-11-2018

Shortest job first (SJF)

- Gli algoritmi SJF, possono essere sia non-preemptive **Shortest Next Process First (SNPF)** sia preemptive **Shortest Remaining Time First (SRTF)**.

Shortest Next Process First (SNPF)

- L'algoritmo SNPF prevede che sia eseguito sempre il processo con il tempo di esecuzione più breve tra quelli pronti.
- Supponiamo ad esempio che si trovino nello stato di pronto i seguenti processi, con la rispettiva durata di esecuzione in millisecondi:

  [p1, 10] → [p2, 2] → [p3, 6] → [p4, 4]

- Con SNPF i processi sono eseguiti nel seguente ordine:

  p2 → p4 → p3 → p1
- Trascurando il tempo necessario per il cambio di contesto, il processo p2 non attende nulla, perché va subito in esecuzione, p4 attende 2 millisecondi, perché va in esecuzione subito dopo p2, quindi p3 attende 6 millisecondi e p1 ne attende 12. Il tempo di attesa medio è pari a:

  $\text{Tempo}_{\text{attesa medio}} = (0+2+6+12)/4 = 5$ millisecondi

Shortest Remaining Time First (SRTF)

- L'algoritmo **SRTF** è la versione preemptive del precedente.
- Con SRTF, se un nuovo processo, entrante nella coda di pronto, ha una durata minore del tempo restante al processo in esecuzione per portare a terminare il proprio CPU-burst, allora lo scheduler provvede ad effettuare un cambio di contesto e assegna l'uso della CPU al nuovo processo.

- Si può dimostrare che teoricamente l'algoritmo SJF è ottimale, in quanto, selezionando dalla coda di pronto il processo più breve da eseguire, consente di ottenere sempre il valore più basso del tempo di attesa medio.
- Per poterlo applicare, nei sistemi batch, i programmi si avviano fornendo per ciascuno di essi un'informazione relativa alla durata dell'esecuzione. Tale valore, è ricavato dalle precedenti esecuzioni del job.
- Con SRTF lo scheduler, dato che non è possibile stabilire la durata del prossimo cpu-burst, implementa algoritmi di predizione che stimano il prossimo tempo di cpu-burst a partire da quelli precedentemente eseguiti.
- Una stima spesso usata si basa sulla media esponenziale:

  $s_{n+1} = \alpha \cdot T_n + (1 - \alpha)\, s_n$

  dove T_n è la durata dell'n-esimo CPU-burst, s_{n+1} la durata

prevista per la successiva sequenza e α [0..1] è il peso che deve essere assegnato al passato del processo.

- Espandendo la relazione si può notare come i valori dei singoli intervalli abbiano un peso tanto minore quanto più sono vecchi

:

$$s_{n+1} = \alpha \cdot T_n + (1 - \alpha)\, \alpha \cdot T_{n-1} + \ldots (1-\alpha)^i\, \alpha \cdot T_{n-i} + \ldots (1-\alpha)^{n+1}\, s_0$$

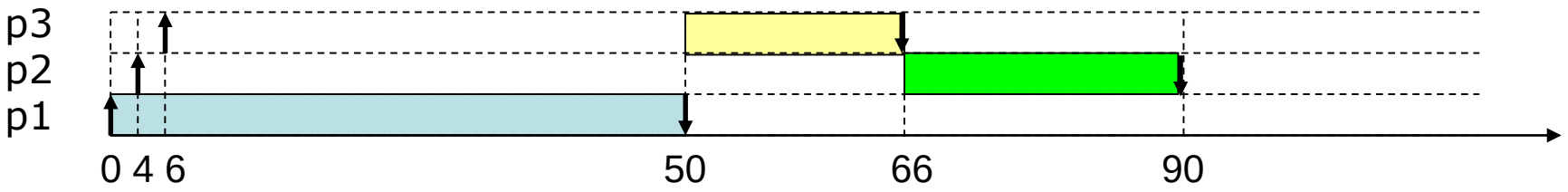
In genere per $\alpha = 0{,}5$ e $s_0 = 10$ si ha una buona approssimazione.

Esempio

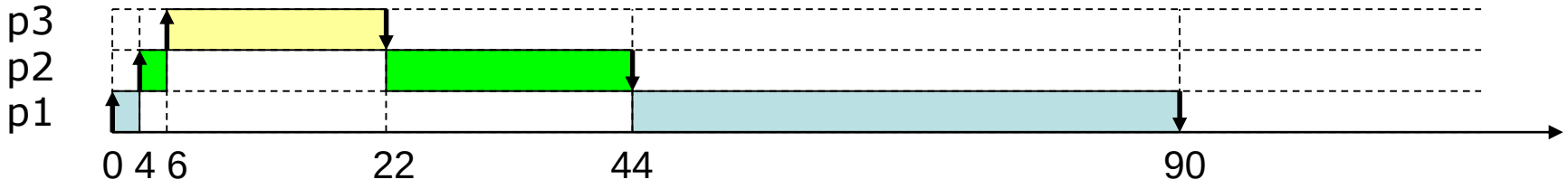
vediamo i diagrammi temporali relativi ai seguenti processi

- P1 [0,50]
- P2 [4,24]
- P3 [6,16]

per gli algoritmi SJF e SRTF



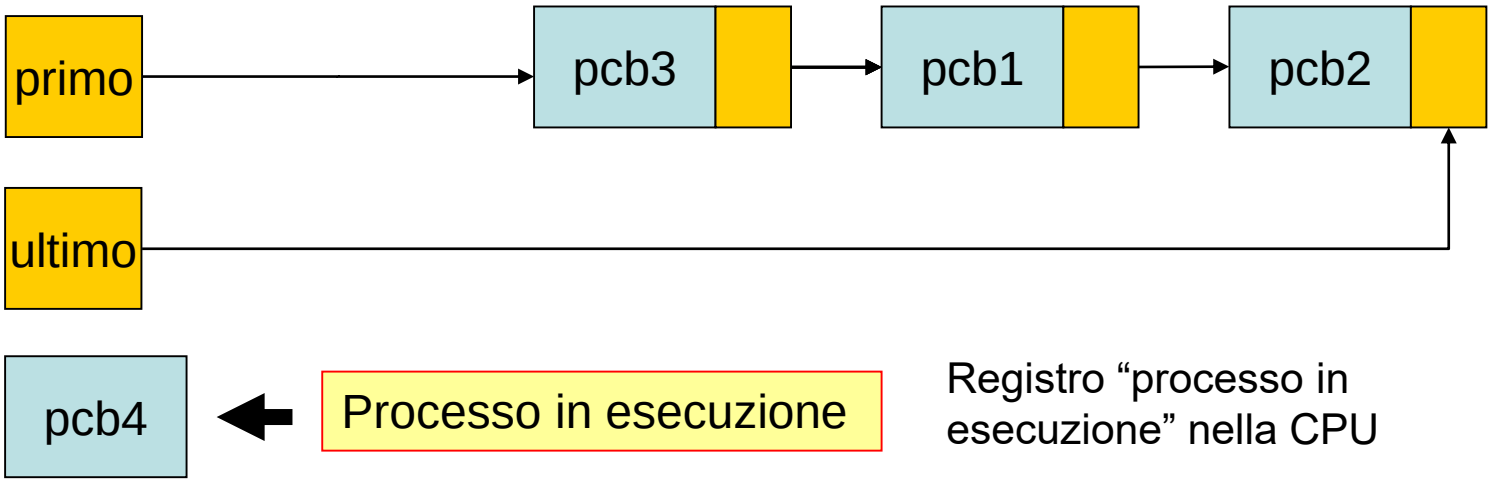
SJF



SRTF

Round Robin

- E’ stato realizzato **per i sistemi time-sharing**.
- Consente il prerilascio.
- La coda dei processi pronti è di tipo circolare. La CPU viene assegnata ad ogni processo per un **quanto di tempo**, tipicamente da 10 a 100 millisecondi.
- La coda è gestita in modalità **FIFO**: il processo a cui viene revocata la CPU è inserito in fondo alla coda e il successivo che avrà il controllo della CPU è il primo della coda.

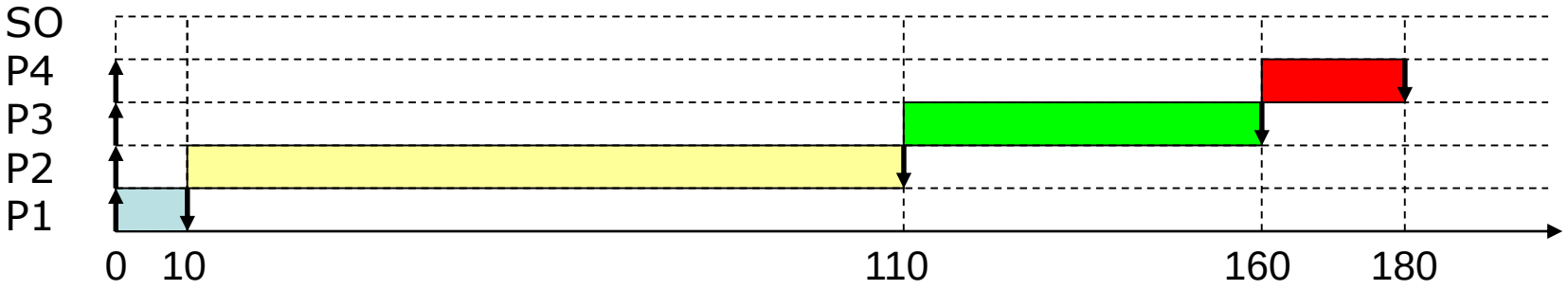


- E’ usato nei i sistemi time-sharing in quanto assicura tempi di attesa medi brevi che dipendono principalmente dal valore del quanto di tempo e dal numero medio di processi pronti.
- Il tempo medio di risposta tende a migliorare diminuendo il valore del quanto di tempo, ma se questo assume valori troppo piccoli diventa significativo l’overhead prodotto dalle operazione del cambio di contesto.
- E’ necessario che sia:

Tempo_cambio_di_contesto << durata_quanto_di_tempo

Esempio

- Vediamo come **RR** privilegi i processi interattivi rispetto ai CPU-bound. Supponiamo che all'istante T_0 siano presenti i seguenti quattro processi nella coda di pronto. Calcoliamo il tempo medio di attesa e di risposta per quattro processi P1, P2, P3 e P4 aventi rispettivamente i tempi di arrivo e durata di CPU-burst (in millisecondi) pari a: P1 [0,10], P2 [0,100], P3 [0,50], P4 [0,20]. Con **FCFS** si ha:



A1 = 0
A2 = 10
A3 = 110
A4 = 160

tempo medio di attesa

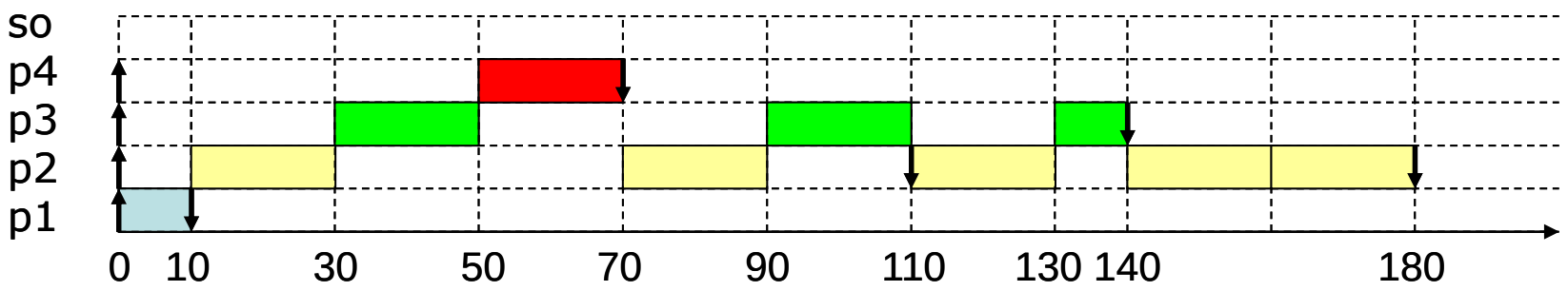
Am = (0 + 10 + 110 + 160)/4 = 70

R1 = 10
R2 = 110
R3 = 160
R4 = 180

tempo medio di risposta

Rm = (10 + 110 + 160 + 180)/4 = 115

- Con **RR** si ha:



A1 = 0
A2 = 10 + 40 + 20 + 10 = 80
A3 = 30 + 40 + 20 = 90
A4 = 50

tempo medio di attesa

Am = (0 + 80 + 90 + 50)/4 = 55

R1 = 10
R2 = 180
R3 = 140
R4 = 70

tempo medio di risposta

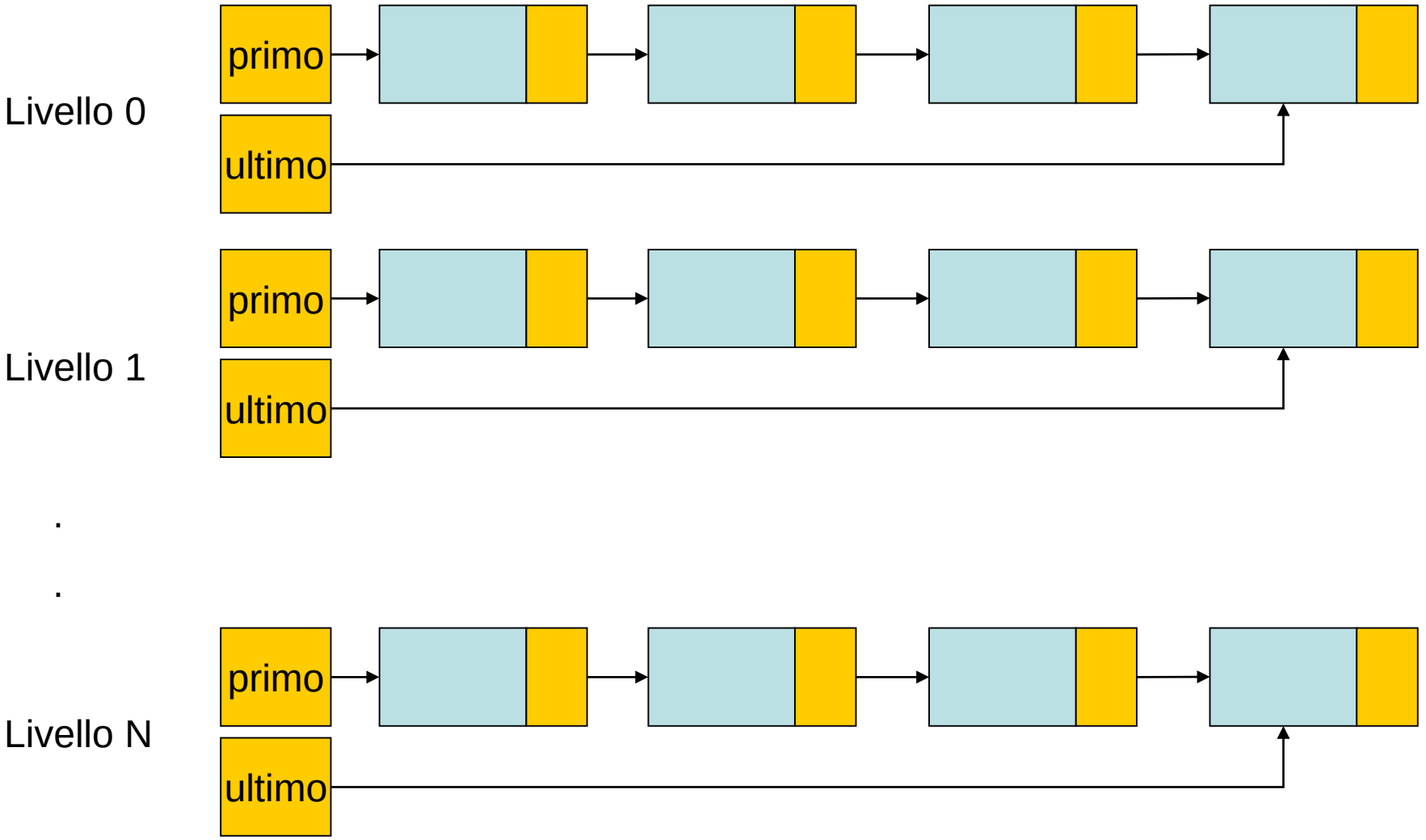
Rm = (10 + 180 + 140 + 70)/4 = 100

Da questo esempio si vede che RR consente di avere tempi di risposta tanto minori quanto minore è il periodo di tempo di esecuzione richiesto a prescindere dall'ordine in cui i processi sono entrati nella coda di pronto. Nell'esempio si è posto un quanto di tempo **Δ = 20 ms**.

Algoritmi di scheduling basati sulle priorità

- Assegnano la CPU al processo pronto con priorità più alta. I processi con le stesse priorità si ordinano con politica **FCFS**.
- Le priorità possono essere statiche o dinamiche.
  - **Priorità statiche** se non si modificano durante l'esecuzione dei processi;
  - **Priorità dinamiche** se si modificano durante l'esecuzione dei processi, in base a qualche criterio come ad esempio in base al tempo di utilizzo di cpu.
- La coda di pronto può essere suddivisa in varie code, ciascuna relativa ad un valore di priorità. In questo caso l'algoritmo di scheduling seleziona il processo pronto dalla coda a più alta priorità, applicando all'interno di ogni coda un tipo di algoritmo, ad esempio Round Robin.
- Se non si utilizzano priorità dinamiche si corre il rischio che processi con bassa priorità attendano tempi enormi al limite infiniti (**starvation**).

Scheduling basato sulle priorità

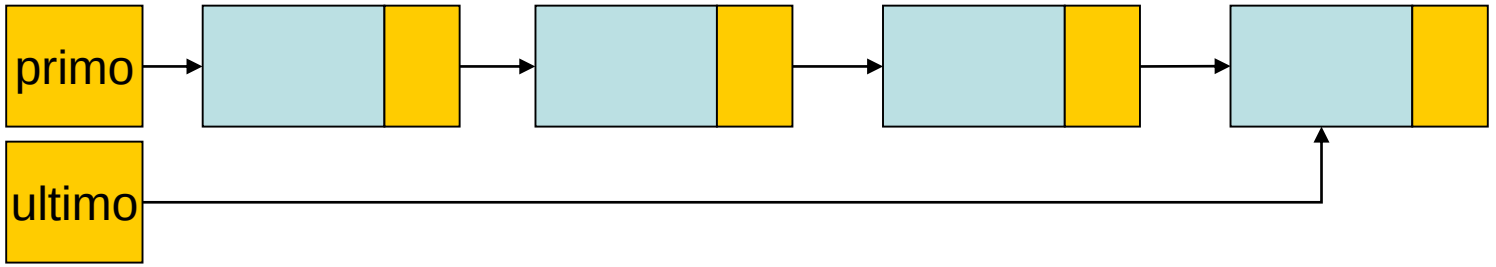


Algoritmi di scheduling a code multiple

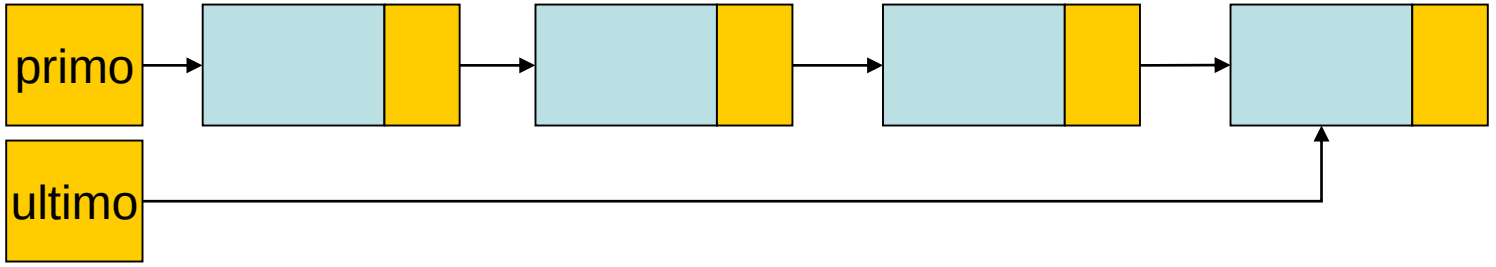
- Nei SO complessi sono in esecuzione processi con caratteristiche diverse.
- Sono presenti processi CPU-bound e IO-bound, oppure processi interattivi e processi tipo “batch” detti rispettivamente processi **foreground** e **background**.
- Pertanto i processi pronti sono inseriti in varie code, ciascuna con diversa priorità. Lo scheduler seleziona per primo un processo dalla coda a più alta priorità.
- Nel caso più semplice sono presenti due code una (livello 0) per i processi interattivi che viene gestita con RR e l’altra (livello 1) a priorità inferiore per i processi background che viene gestita con FCFS.
- Casi più complessi prevedono molte code a diverse priorità in cui i processi possono passare, durante la loro esecuzione da una coda all’altra (multilevel feedback queue)

Scheduling basato sulle code multiple

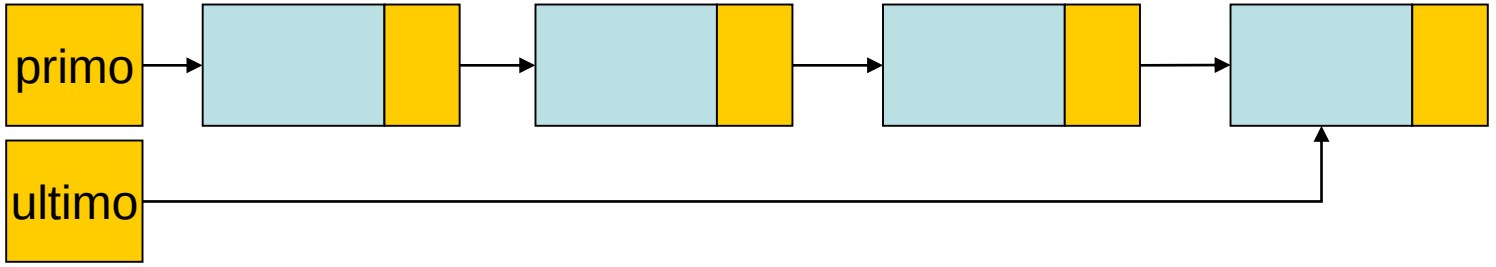
Livello 0 coda RR con quanto = 20 ms



Livello 1 coda RR con quanto = 50 ms



Livello 2 coda FCFS per processi background



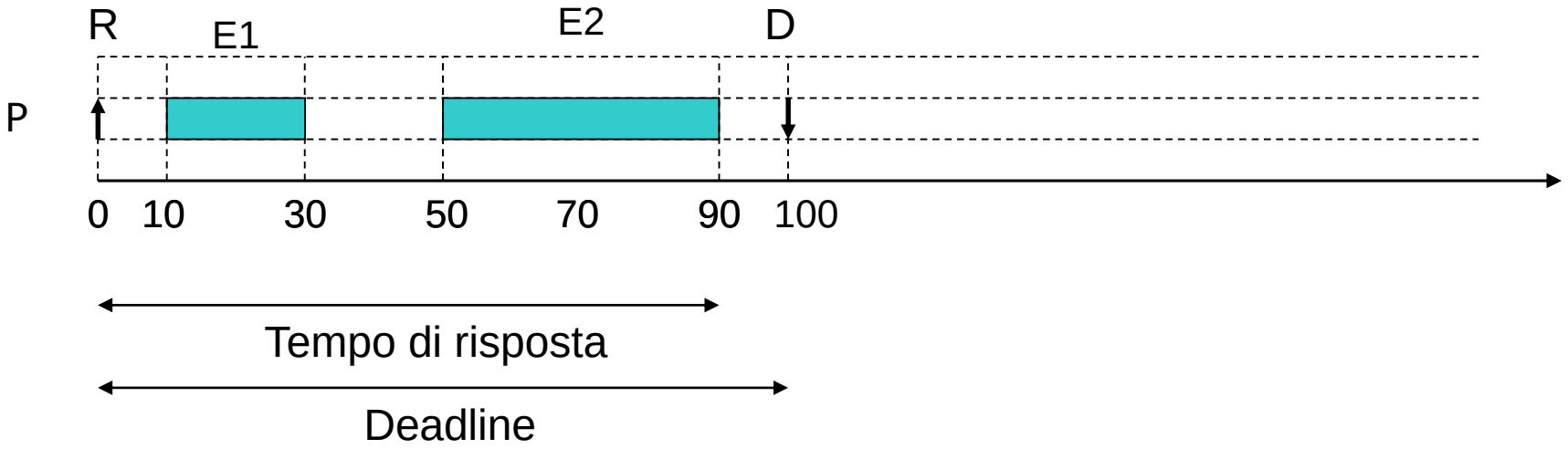
Algoritmi di scheduling real-time

- Generalmente i SO real-time utilizzano algoritmi di scheduling basati sulle priorità. Gli algoritmi possono essere statici o dinamici.
- Gli algoritmi statici assegnano le priorità ai processi in base alla conoscenza di alcuni parametri temporali dei processi noti all'inizio. Al contrario gli algoritmi dinamici cambiano la priorità dei processi durante la loro esecuzione.
- I processi real-time fondamentalmente sono caratterizzati dai seguenti parametri:
  - **istante di richiesta**: l'istante in cui il processo entra nella coda di pronto.
  - **deadline**: istante entro il quale il processo deve essere terminato.
  - **tempo di esecuzione**: tempo di CPU necessario al processo per svolgere il suo lavoro.

Esempio

Per il processo in figura si ha:

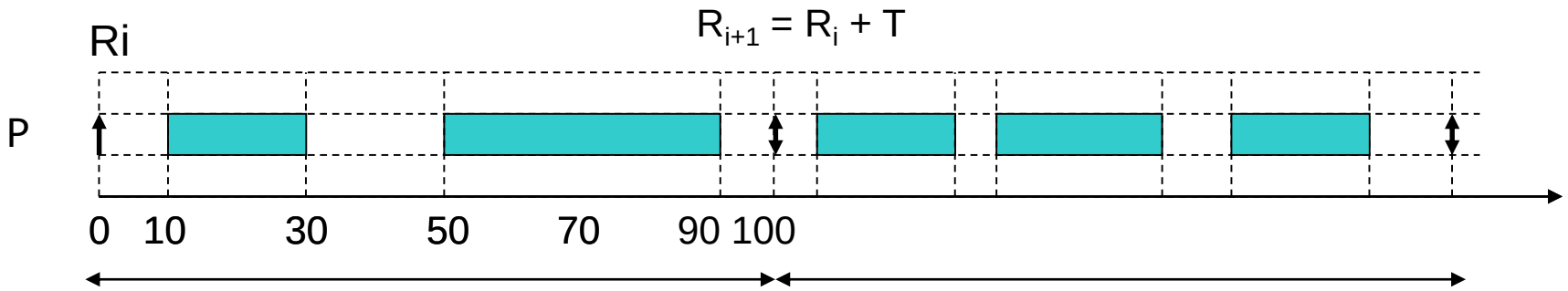
- Istante di richiesta **R** = 0;
- tempo di esecuzione **E** = E1 + E2 = 20 + 40 = 60
- deadline **D** = 100
- Tempo di risposta = 90



- I processi real-time possono essere **periodici** o **aperiodici**. I processi periodici vengono attivati ciclicamente a periodo costante che dipende dalla grandezza fisica che il processo deve controllare.
- I processi non periodici vengono avviati in situazioni imprevedibili.
- Consideriamo un algoritmo di scheduling per processi periodici.

  In tal caso deve essere:

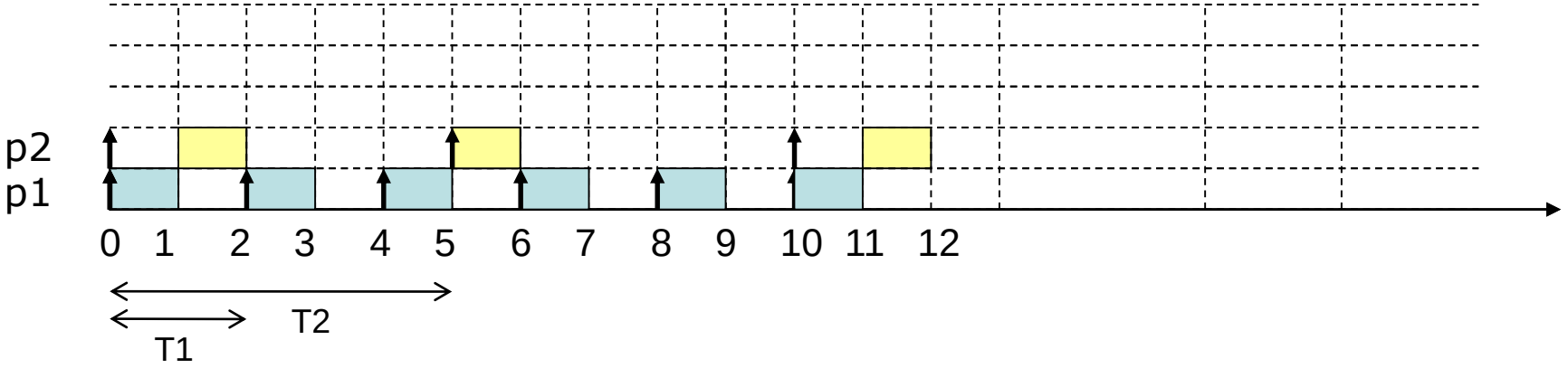
$$R_{i+1} = R_i + T$$
$$T >= D$$



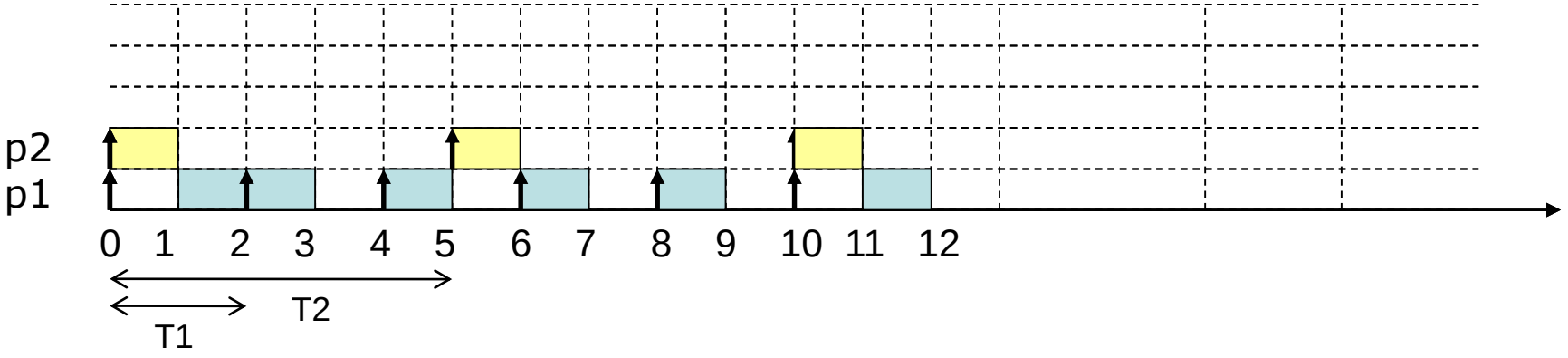
Periodo T

- Dato che in ciascun periodo il tempo di esecuzione di un processo potrebbe cambiare, indichiamo con $\mathbf{E_{max}}$ il tempo massimo di esecuzione di un processo in ciascun periodo **T**.
- $\mathbf{E_{max}}$ e **T** sono tempi noti a priori, imposti dall'applicazione real-time.
- Nei SO RT generalmente si utilizzano algoritmi di scheduling con priorità.
- Il problema è come assegnare la priorità ai processi.
- L'algoritmo ***Rate Monotonic (RM)*** è un algoritmo ottimo, nell'ambito della classe degli algoritmi basati sulle priorità statiche. E' un algoritmo preemptive. Esso assegna la priorità ai processi in base alla durata del loro periodo. In particolare priorità maggiore ai processi che hanno periodo minore.
- Per mostrare la validità di tale criterio, consideriamo un esempio in cui due processi P1 e P2 hanno i seguenti parametri temporali:

P1: T1 = 2; E1 = 1
P2: T2 = 5; E2 = 1



Priorità P1 > priorità P2

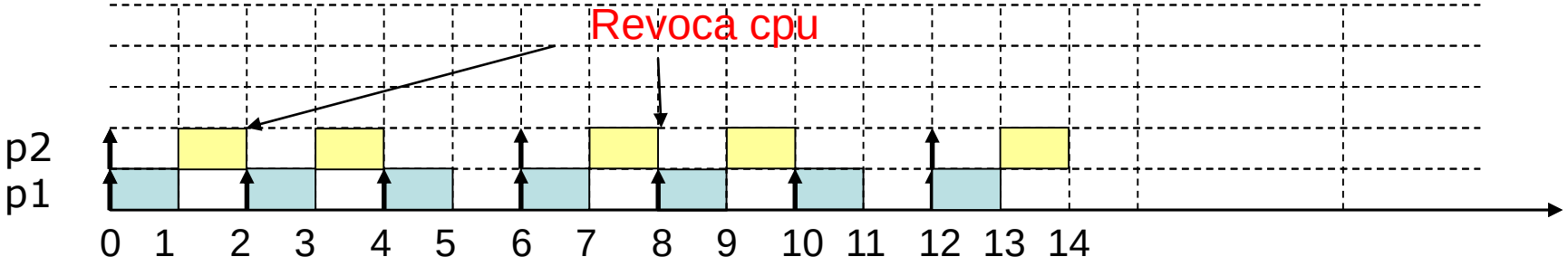


Priorità P2 > priorità P1

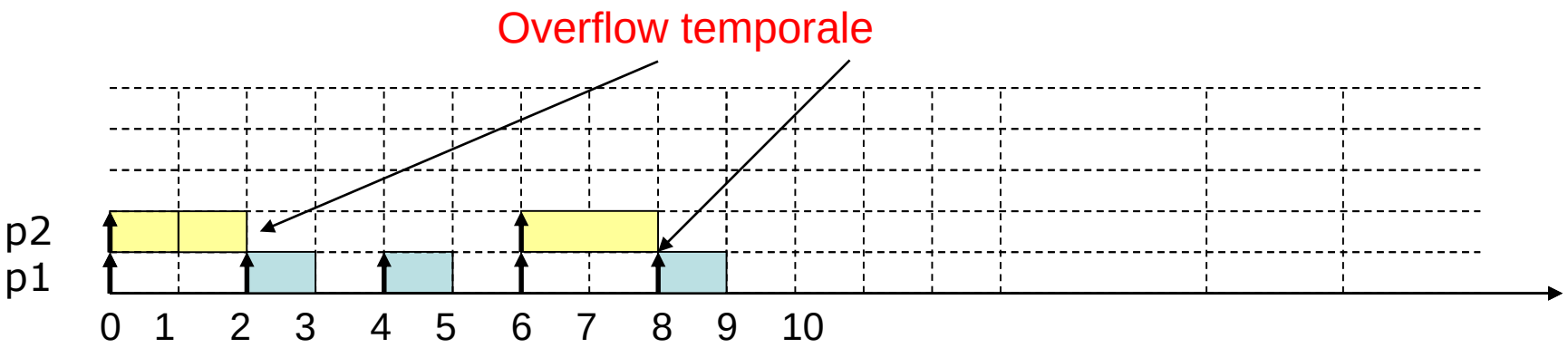
Supponiamo ora che sia E2 = 2 T2=6, mentre sia ancora

E1 =1 e T1 = 2.

Si ha che nel primo caso i due processi sono ancora schedulabili mentre nel secondo caso no.



Priorità P1 > priorità P2



Priorità P2 > priorità P1

- Tuttavia, anche adottando un criterio ottimo come RM è necessario (ma non sufficiente) che sia verificata la seguente relazione:

  $U = \sum (Ei/Ti) <= 1$

  Dove **U** è detto **coefficiente di utilizzazione** della CPU.
- E' stato dimostrato, utilizzando ***Rate Monotonic***, che affinché un insieme di **N processi** sia schedulabile è sufficiente che il coefficiente di utilizzazione sia:

  $U <= N(2^{1/N} – 1)$

  Ad esempio per N=4 si ha U=0.76

  per N=50 U=0.698

  per N=100 U=0.695

  per N->infinito U -> ln 2 = 0,693147180559453094172