



UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI  
INGEGNERIA INFORMATICA,  
MODELLISTICA, ELETTRONICA  
E SISTEMISTICA

DIMES

Corso di Laurea Magistrale in Ingegneria dell'Automazione  
A.A. 2021/2022

# ALGORITMI DI SCHEDULING REAL-TIME

DOCENTE: PROF. PUPO

STUDENTE: IVONNE RIZZUTO

# SOMMARIO

## Sistemi Real-Time

- Sistemi Hard Real-Time e Soft Real-Time
- Proprietà dei sistemi Real-Time
- Classificazione dei processi

## Scheduling

- Caratteristiche dello Scheduling
- Tipologie di algoritmi di schedulazione Real-Time
- Esempi di algoritmi di scheduling Real-Time

## Progetto

- Dati dei processi
- Analisi tramite script MATLAB
- Confronto fra i due algoritmi di scheduling



# CLASSIFICAZIONE DEI SISTEMI REAL-TIME

I sistemi real-time possono essere classificati sulla base di quanto sia severo e stringente il vincolo temporale (*deadline*) posto per l'esecuzione di un determinato task. Allora, si distinguono in:

## Hard real-time

per cui devono necessariamente rispettare i vincoli di correttezza logica e temporale, in quanto la mancanza di una sola deadline potrebbe mettere a rischio il funzionamento dell'intero sistema e/o l'incolumità di esseri viventi.

## Soft real-time

in cui il mancato rispetto di una scadenza, benché negativo, non è fatale per il funzionamento dell'intero sistema, in quanto l'effetto può essere controllato e compensato.

# PROPRIETÀ DEI SISTEMI REAL-TIME

Un sistema Real-Time è caratterizzato dalle seguenti proprietà:

- **L'affidabilità**, o reliability, che consiste nella probabilità che un sistema si comporti correttamente per un certo lasso di tempo; dal punto di vista pratico, si riferisce alla misurazione di quanto spesso il sistema fallisce nel compimento dei suoi task;
- **La tolleranza ai guasti**, o fault-tolerance, che rappresenta l'abilità, di un sistema, di riconoscere e gestire i guasti, senza subire conseguenze significative; questo è reso possibile tramite la progettazione e l'attuazione di controlli di protezione, che diano robustezza al sistema, senza comprometterne le prestazioni.

Per misurare questo fattore si calcola il tempo medio che intercorre tra due fallimenti del sistema (Mean Time Between Failures).

# CLASSIFICAZIONE DEI PROCESSI

I processi, ossia gli eventi a cui deve far fronte un sistema real-time, sono classificati in:

- **periodici**, se costituiti da una sequenza di attività con cadenza regolare e descritti tramite i parametri:
  - $c$  "computation time", cioè il costo in termini di esecuzione;
  - $T$  "periodo di esecuzione", il lasso di tempo che intercorre tra due esecuzioni di un task;
  - $D$  "relative deadline", cioè il tempo massimo che impiega un processo per giungere a compimento, dal momento in cui è stato avviato (il suo valore è pari all'inverso del periodo);  
rispettando la relazione  $c \leq D \leq T$  ;
- **sporadici**, o "event-driven", se costituiti da una sequenza di attività che vengono avviate in maniera imprevedibile e descritti tramite i parametri  $c$ ,  $D$  e  $T$ , dove per  $T$  si intende la minima distanza temporale tra occorrenze successive dell'evento;
- **aperiodici**, se presentano delle possibili occorrenze dell'evento, in maniera ravvicinata ed imprevedibile;

# SCHEDULING

Un insieme di task è detto schedulabile se esiste almeno un algoritmo di scheduling che permette il rispetto di tutti i vincoli del problema, in termini di correttezza logica e temporale.

L'operazione di **scheduling** consiste, quindi, nel definire un ordine secondo cui diverse attività accedono ad una risorsa condivisa, ad esempio un processore.

In base al numero di unità di calcolo che deve gestire, un algoritmo di scheduling si definisce:

- **monoprocessore**, se fa dispatching dei vari task su una sola unità di calcolo;
- **multiprocessore**, se fa dispatching delle attività su più unità di controllo;

Il modulo del sistema operativo che passa il controllo della CPU ai processi scelti dallo *scheduler* è chiamato *dispatcher*.



# CARATTERISTICHE DELLO SCHEDULING

Un algoritmo di scheduling si può definire:

- **fattibile** (feasible), se è in grado di gestire i processi in modo tale che ognuno di questi soddisfi i propri vincoli temporali;
- **ottimo**, se è capace di determinare, se esiste, una schedulazione che sia fattibile;
- **on-line**, se le decisioni riguardanti l'ordine di esecuzione dei task dipendono dal loro ordine di attivazione;
- **off-line**, se la sequenza di dispatching dei task è nota a priori, poiché le decisioni riguardanti il loro ordine di esecuzione sono prese prima della loro stessa attivazione;
- di tipo **preemptive**, se è in grado di interrompere l'esecuzione, in una delle unità di calcolo, di un task a minor priorità, a favore di uno a priorità maggiore;
- di tipo **non-preemptive**, se non è in grado di interrompere l'esecuzione di un processo dopo che è stato inviato ad una delle unità di calcolo;



# TECNICHE DI SCHEDULING

In genere, uno scheduling real-time si occupa della gestione di processi periodici, il cui periodo di attivazione si mantiene costante ad ogni esecuzione e può essere basato su una tecnica di tipo:

## statico

se le regole di dispatching sono definite a partire da parametri che non variano durante l'esecuzione dell'algoritmo stesso e prima che il sistema inizi l'esecuzione dei processi.

## dinamico

se l'ordine di schedulazione dei processi viene stabilito a partire da parametri che possono variare durante l'esecuzione dell'algoritmo, quindi non è necessario che siano noti, a priori, i tempi di esecuzione e le deadline dei tasks.

# ALGORITMI DI SCHEDULAZIONE REAL-TIME

Le principali tipologie degli algoritmi di schedulazione real-time sono:

- **clock-driven**, se le decisioni relative alle attività da svolgere ed agli intervalli di tempo per cui devono rimanere in esecuzione vengono determinate in anticipo e messe in atto in istanti di tempo predefiniti;
- **weighted round-robin**, se l'intervallo di tempo per cui ogni attività occuperà il processore viene stabilito sulla base del peso che gli è stato assegnato, pur mandando in esecuzione i processi in maniera circolare (maggiore è il peso, più quanti di tempo avrà a disposizione il task per utilizzare la CPU);
- **priority-driven**, se cerca di evitare che il processore o qualsiasi altra risorsa rimangano inutilizzati, assegnando ai tasks dei valori numerici indicativi della priorità che ha, ciascun processo, nell'assegnamento dell'unità di calcolo;

# STATIC CYCLIC SCHEDULING

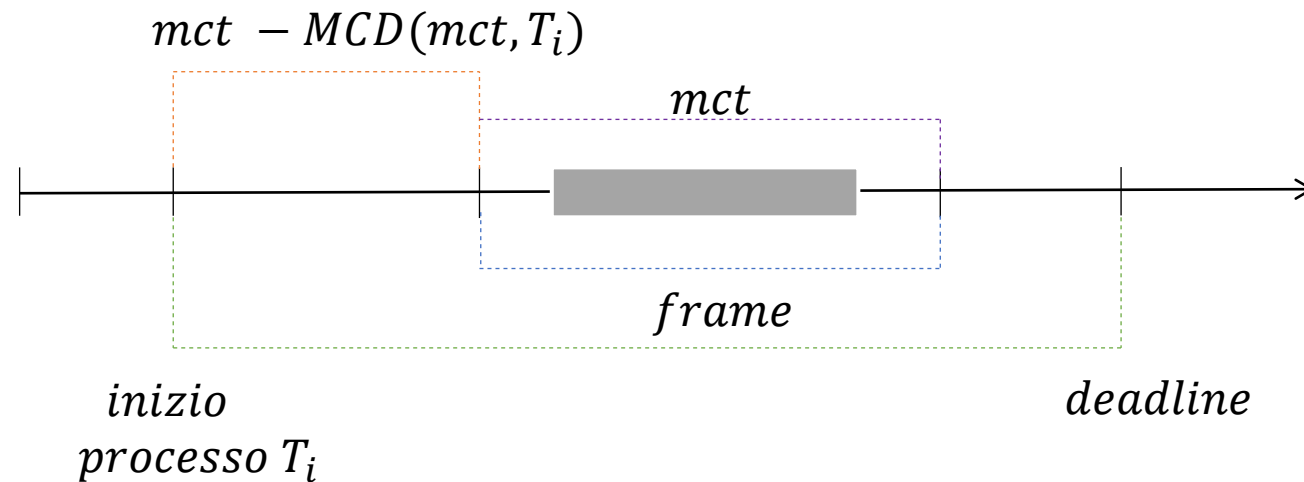
Questa tecnica è utilizzata per ottenere l'hard real-time in un sistema e si basa su una tabella di schedulazione, definita a priori e fornita come input ad un cyclic scheduler oppure ad un cyclic executive (cioè un'alternativa ad un sistema operativo in tempo reale, che consiste in una forma di multitasking cooperativo, in cui esiste un solo compito, che in genere è realizzato come loop infinito).

Ogni processo è visto come una successione di blocchi, detti azioni o *slice*, la cui durata è predeterminata e l'attività di schedulazione viene organizzata suddividendo l'asse temporale in intervalli (*frame*) della stessa lunghezza.

Si chiama **minor cyclic time** la lunghezza di ogni intervallo di tempo e **Major Cyclic Time** (o hyper period) il tempo dopo cui il processo di scheduling viene ripetuto per la sua interezza.

La temporizzazione dei tasks può essere verificata negli istanti di confine tra un frame ed il successivo, tramite il *clock tick interrupt*.

# STATIC CYCLIC SCHEDULING: CARATTERISTICHE



Bisogna scegliere:

- un valore di MCT divisibile per  $mct$ ;
- un valore di  $mct \leq D_i$  per ogni processo  $i$ -esimo, per assicurare che la sua esecuzione si concluda entro la sua deadline;
- rispettare la relazione  $mct + (mct - MCD(mct, T_i)) \leq D_i$ , poiché ogni processo, per essere eseguito, deve iniziare le sue azioni durante il periodo assegnato;

## STATIC CYCLIC SCHEDULING: CONCLUSIONI

Il vantaggio che si ottiene con questo algoritmo di schedulazione consiste nell'utilizzo di una tecnica semplice ed efficiente, che favorisce la *predicibilità* temporale del comportamento di un sistema real-time, facilitandone la realizzazione.

Tuttavia, trattandosi di uno scheduler a basso livello, risulta incompatibile con i metodi adottati nella programmazione attuale, a causa della sua vulnerabilità ai cambiamenti. Infatti, persino una piccola variazione della durata della slice di un processo potrebbe rendere questo tipo di scheduling non fattibile.

Inoltre, lo SCS non è dinamico e di conseguenza è poco adattabile, perché è difficile garantire che i dati che deve elaborare siano sempre certi e l'allocazione dei processi, ossia la schedulazione delle slice, costituisce un problema di complessità di tipo esponenziale.

# RATE MONOTONIC SCHEDULING

Questo scheduler consiste nell'assegnare, ad ogni processo, una priorità che è direttamente proporzionale alla propria frequenza, in modo tale che i processi con periodi più brevi abbiano una priorità più elevata.

Allora, la priorità viene stabilita secondo una funzione *monotona decrescente*, incentrata sul periodo di attivazione del processo.

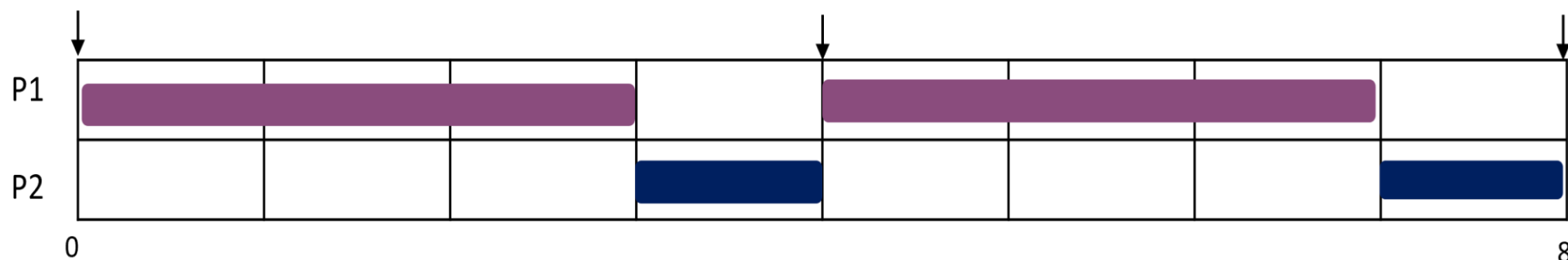
RMS è un algoritmo di tipo *statico*, perché le priorità vengono assegnate prima della sua esecuzione e rimangono invariate, e *pre-emptivo*, in quanto sull'unità di calcolo viene spacciato, tra quelli attivi, sempre il processo con priorità più alta e quindi con periodo di attivazione minore.

Per questo motivo è detto anche **Shortest Period First**.

Inoltre, è di tipo *on-line* perché la configurazione dei task in uscita dipende dal numero di attività date in ingresso allo scheduler.

## RATE MONOTONIC PRIORITY ORDERING: ESEMPIO

Dati i processi  $P_1 = (3,4,4)$  e  $P_2 = (2,8,8)$ , tra cui vale la seguente relazione di priorità  $\pi(P_1) > \pi(P_2)$ , si può delineare il seguente *diagramma di Gantt*:



Ricordando la definizione del generico processo  $P_i = (C, T, D)$ , si può notare come sia il processo  $P_1$  ad essere mandato per primo in esecuzione, perché RMPO assegna un valore della priorità che è inversamente proporzionale al periodo.

Infatti, considerati i valori  $\pi(P_1) = \frac{1}{T_1} = 0,25$  e  $\pi(P_2) = \frac{1}{T_2} = 0,13$ , ciò che risulta è che  $P_1$  sia il processo con la priorità più elevata.



# RATE MONOTONIC SCHEDULING: OTTIMALITÀ

Nel 1973, **Liu** e **Layland** hanno dimostrato l'ottimalità di RMPO, rispetto a tutti gli altri algoritmi di scheduling basati su un'assegnazione statica delle priorità, evidenziando come non fosse possibile determinare una soluzione ammissibile ad un dato problema di schedulazione, se questo fosse già risultato non risolvibile tramite RMPO.

La fattibilità di uno scheduler viene valutata tramite il calcolo del **fattore di utilizzazione**  $U$  del processore, cioè la percentuale di tempo in cui questo viene utilizzato per l'esecuzione dell'insieme dei task.

Questo è definito come:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

# TEST DI AMMISSIBILITÀ DI UNO SCHEDULING

La **condizione necessaria**, per cui viene garantita l'ammissibilità di qualsiasi scheduling, è la seguente:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

Tuttavia, dato una schedulazione di  $n$  processi, questa è sicuramente fattibile con RM se vale la seguente disuguaglianza, che definisce il valore del **Bound LL** (Liu-Layland):

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

Dove  $\lim_{n \rightarrow \infty} n(2^{1/n} - 1) \cong \ln(2) \cong 0,69$ , ossia il valore limite del fattore di occupazione dell'unità di calcolo, pari al **69,3%**.

Questa costituisce una **condizione sufficiente**, perciò l'insieme di task può essere schedulabile, anche se questa non viene soddisfatta, purché sia rispettata quella necessaria.

## AMMISSIBILITÀ DI RMPO: ESEMPIO

Dati i processi  $P_1 = (2,8,8)$ ,  $P_2 = (3,16,16)$ ,  $P_3 = (5,12,12)$ , si calcola il fattore di utilizzazione:

$$U = \sum_{i=1}^3 \frac{C_i}{T_i} = \left( \frac{2}{8} + \frac{3}{16} + \frac{5}{12} \right) \approx 0,8542 < 1$$

Siccome il problema è ammissibile, si determinano le priorità statiche dei tre processi, che sono:

$$A_1 = 0,125, A_2 = 0,0625 \text{ e } A_3 = 0,08\bar{3}.$$

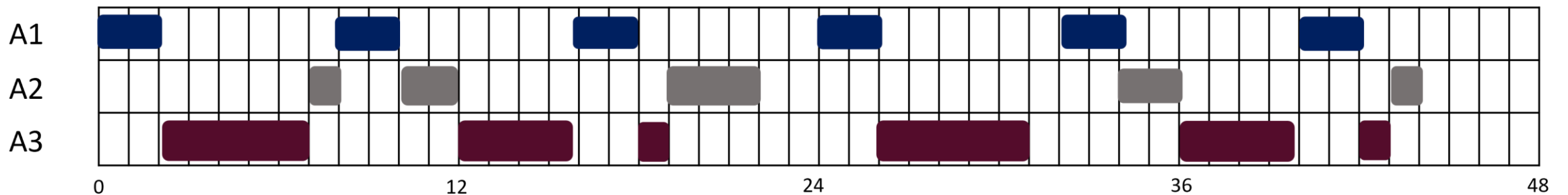
Invece, il numero minimo delle occorrenze  $\sigma$  da dare ad ogni task si calcola dividendo il valore  $m.c.m. (T_1, T_2, T_3) = 48$  per il periodo di attivazione di ciascuno.

Si è ottenuto:

$$\sigma(P_1) = 6, \sigma(P_2) = 3 \text{ e } \sigma(P_3) = 4$$

# AMMISSIBILITÀ DI RMPO: ESEMPIO

Segue il diagramma di Gantt fino a 48 unità di tempo, perché dopo questo valore lo scheduling si ripete uguale a sé stesso.



Si noti come, all'istante di tempo 8, l'algoritmo attui un'operazione di prelazione sul task  $P_2$ , a favore del processo  $P_1$ , a cui è stata assegnata una priorità più elevata.

Il processo  $P_2$ , inoltre, viene preemptato anche al termine del suo periodo di attivazione (nell'istante di tempo 36), anche se le sue operazioni non sono state ultimate, perciò le unità mancanti, in termini di costo, saranno recuperate quando verrà avviato nuovamente.

# FIXED PRIORITY SCHEDULING: ANALISI ESATTA

Ideato da **Joseph** e **Pandya**, consiste in un protocollo di schedulazione che estende il lavoro di Liu e Layland al caso di  $D_i \leq T_i$ .

Questo ha introdotto un metodo che consente di effettuare un'*analisi esatta* della schedulabilità di un insieme di processi (*task set*), superando il bound LL, che non è verificato.

Si procede calcolando il **tempo di risposta**, ovvero il *worst case time* richiesto dal task  $i$  – *esimo* per completare la propria esecuzione, definito come:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Considerando quante esecuzioni ricadono nel tempo dei task ad alta priorità  $hp(i)$  (*high-priority*).

Inoltre  $\lceil x \rceil$  rappresenta la funzione di ceiling (*soglia*) che restituisce il minimo valore intero  $\geq x$ , mentre  $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$  costituisce l'**interferenza** del processo  $P_j$  sul processo  $P_i$ .

## ANALISI ESATTA: EQUAZIONE DI RICORRENZA

La sommatoria, allora, si riferisce all'interferenza della totalità di tutti i processi e siccome non compare il termine  $T_i$  si può trasformare la relazione precedente in un'**equazione di ricorrenza**:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$$

Così si può calcolare, data un'approssimazione di  $R_i^n$ , la successiva approssimazione  $R_i^{n+1}$ .

Il termine iniziale sarà  $R_i^0 = 0$ , mentre il processo converge per  $R_i^{n+1} = R_i^n$ .

Il valore di  $R_i$  che si è ottenuto dovrà essere confrontato con la deadline  $D_i$  e per la schedulabilità deve valere che  $R_i \leq D_i$ .

In conclusione:

- se  $D_i = T_i$ , l'assegnamento delle priorità ai processi viene effettuato secondo Rate Monotonic;
- se  $D_i < T_i$ , allora l'assegnamento delle priorità segue la regola Deadline Monotonic;

# EARLIEST DEADLINE FIRST SCHEDULING

Questo algoritmo a **priorità dinamica** è uno dei più utilizzati nei sistemi real-time e seleziona come prossimo processo da mandare in esecuzione quello con la distanza minore dalla sua deadline. Si tratta di uno scheduling :

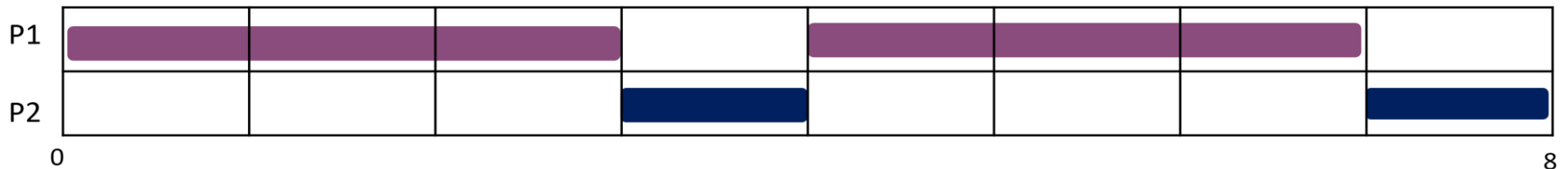
- *dinamico*, perché la priorità dei task attivi deve essere ricalcolata ad ogni attivazione di un qualsiasi processo;
- *on-line*, perché la configurazione dei task in uscita dipende dal numero dei task in ingresso e dai loro periodi di attivazione;
- *pre-emptivo*, perché assegna ad ogni task una priorità inversamente proporzionale alla sua deadline assoluta (la deadline relativa sommata all'istante di tempo dell'attivazione del task) ed a parità di questo valore, avrà priorità maggiore il processo con il minor numero di iterazioni; perciò quello con la deadline più imminente ha la massima priorità e pre-empta quello in esecuzione;
- *ottimo*, perché se un insieme di task periodici non è schedabile tramite EDF, allora non lo sarà con nessun altro algoritmo dinamico;



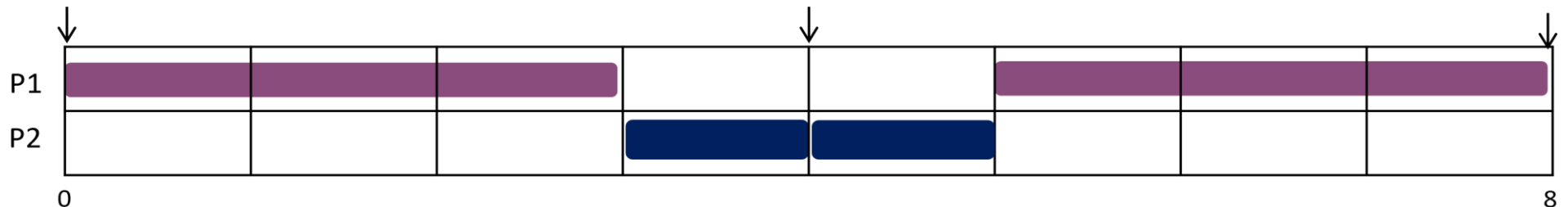
# EARLIEST DEADLINE FIRST SCHEDULING: ESEMPIO

Dati i processi  $P_1 = (3,4,4)$  e  $P_2 = (2,8,8)$ , si riportano due potenziali schedulazioni:

Scenario I



Scenario II



Per lo scheduler è da preferire il secondo perché evita un context switch. Nel primo scenario, infatti, viene effettuata un'azione di pre-emption di  $P_2$  in favore di  $P_1$ .

# CONFRONTO TRA RMPO E EDF

Dato un insieme qualsiasi di  $n$  task periodici:

## Rate Monotonic Priority Ordering

- garantisce la schedulabilità se è soddisfatta la condizione sufficiente, cioè  $\text{Bound LL} \leq 69,3\%$ ;
- calcola le priorità soltanto una volta;
- Può essere utilizzato solamente per task periodici, il cui periodo di esecuzione è fissato e noto a priori;

## Earliest Deadline First

- garantisce la schedulabilità se vale la condizione necessaria e sufficiente, ossia  $U \leq 1$ ;
- ricalcola le priorità dei task attivi ad ogni deadline;
- è utilizzabile sia per task periodici che aperiodici, perché la priorità non dipende dall'ipotesi di periodicità degli stessi;

L'ottimalità di EDF, rispetto a RMPO, si paga in termini di complessità computazionale.

# LEAST SLACK TIME SCHEDULING

Questo algoritmo è di tipo *dinamico* e basato su priorità, assegnata sulla base dello slack time, cioè l'ammontare del tempo rimanente per il completamento di un dato processo.

Il **tempo di slack** (o *laxity*) è definito come:

$$st = [(D - t) - C']$$

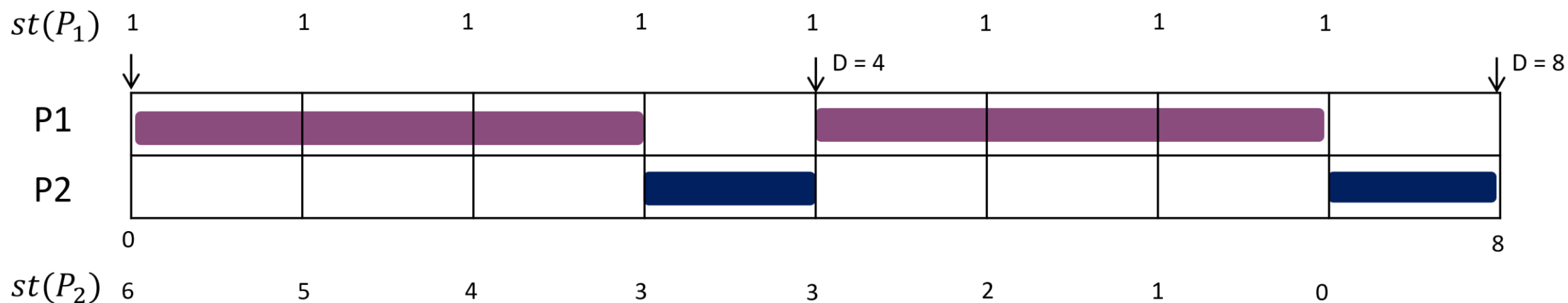
- $D$  è la deadline del task;
- $t$  è l'istante di tempo relativo, considerato all'inizio del ciclo corrente;
- $C'$  è la porzione del corpo del processo non ancora eseguita, in termini di tempo di esecuzione;

Quindi il processo con il minor tempo di slack avrà la priorità più alta, poiché i due parametri sono inversamente proporzionali.

Questo scheduler funziona in modo ottimale soltanto se è consentita la prelazione ed a parità di priorità, i processi vengono mandati in esecuzione secondo la logica *First Come First Served*.

## LEAST SLACK TIME SCHEDULING: ESEMPIO

Si considerino i processi  $P_1 = (3, 4, 4)$  e  $P_2 = (2, 8, 8)$  ed il seguente diagramma di Gantt:



Si può notare come, all'istante di tempo  $t=6$ ,  $P_1$  e  $P_2$  abbiano la stessa priorità, poiché per entrambi risulta  $C' = 1$ .

Tuttavia,  $P_1$  viene mandato in esecuzione per primo, perché LST utilizza la logica FCFS.

# INTERAZIONI TRA PROCESSI

Quando più processi si trovano ad interagire tra loro, in attesa che sia pianificata la loro esecuzione da parte dell'unità di calcolo, che rappresenta la risorsa condivisa, possono manifestarsi alcuni problemi relativi proprio alla loro schedulazione, come **l'inversione di priorità**.

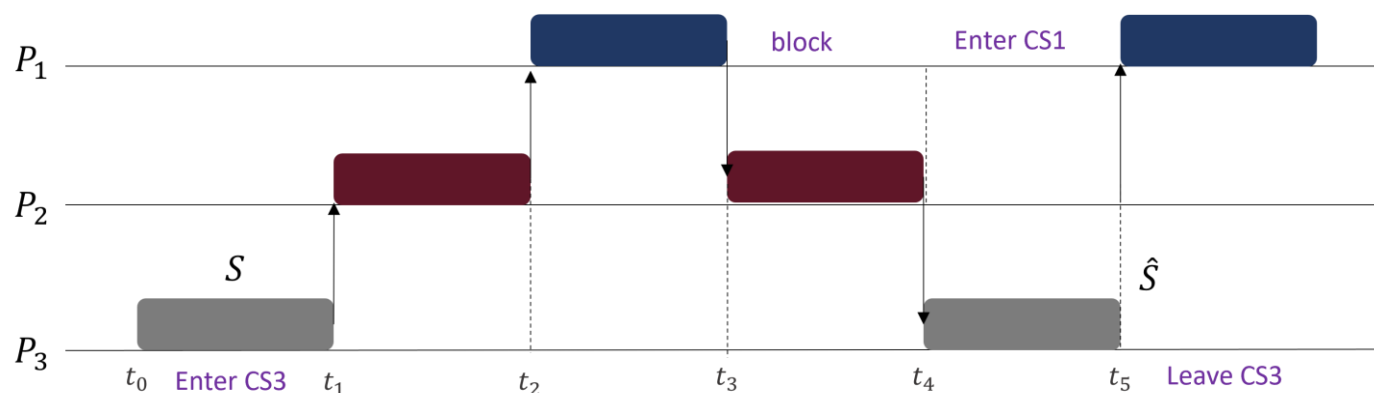
Questo si verifica in un ambiente multitasking, durante l'accesso, da parte di più attività, alla stessa sezione critica.

Allora può accadere che il processo a priorità più alta venga interrotto e bloccato da un altro a priorità inferiore, per un intervallo di tempo indefinito.

In un sistema rea-time il manifestarsi di una situazione di questo tipo rischia di compromettere la pianificazione della schedulazione di tutti i processi del sistema.

# PRIORITY INVERSION PROBLEM: ESEMPIO

Si considerino i processi  $P_1, P_2$  e  $P_3$ , a priorità decrescenti, ossia  $\pi_1 \geq \pi_2 \geq \pi_3$ , tenendo conto che  $P_1$  e  $P_3$  condividono la stessa risorsa, con accesso esclusivo, gestito tramite semaforo binario e che  $P_2$  non interagisce con nessuno dei due. La situazione sarà la seguente:



$P_1:: \text{begin} \dots \text{lock}(S); \text{CS1}; \text{unlock}(S); \dots \text{end}$

$P_2:: \text{begin} \dots \dots \dots; \dots \text{end}$

$P_3:: \text{begin} \dots \text{lock}(S); \text{CS3}; \text{unlock}(S); \dots \text{end}$

Anche se  $P_1$  ha la massima priorità, deve aspettare sia il processo  $P_3$ , che possiede il lucchetto nel momento in cui tenta di entrare in CS1, sia  $P_2$ , il processo con priorità intermedia, poiché ritarda l'uscita di  $P_3$  dalla sezione propria critica, prolungando ancora l'attesa di  $P_1$ .

# PRIORITY INHERITANCE PROTOCOL

Questo protocollo, proposto da *Sha, Rajkuman* e *Lehoczky* nel 1990, fornisce una soluzione al problema di *Priority Inversion*, poiché si occupa della gestione dell'accesso alle risorse condivise.

L'approccio consiste nel modificare la priorità dei processi che causano il blocco, in modo che la priorità di  $P_i$ , in attesa all'entrata di una sezione critica, sia trasmessa al processo  $P_j$ , che si trova al suo interno. Allora la priorità di  $P_j$  verrà modificata, assegnandogli il valore massimo tra le priorità di tutti i processi che ha bloccato, facendo sì che termini le sue operazioni e rilasci la risorsa condivisa.

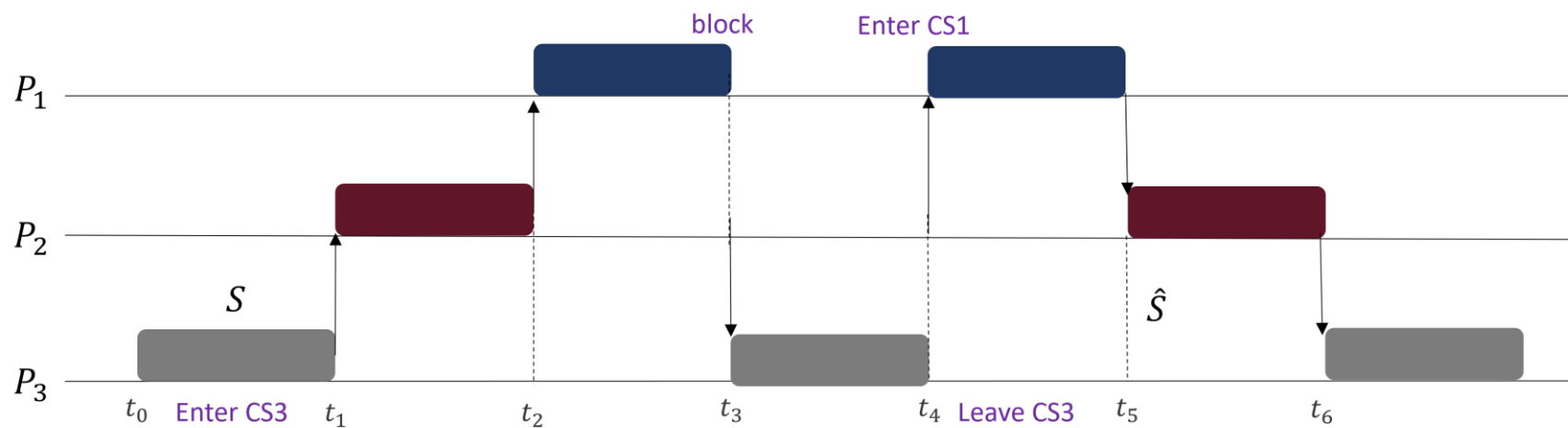
Tuttavia, pur evitando l'inversione di priorità, non è possibile prevenire le condizioni di stallo e quindi di *deadlock* tra i processi; inoltre, in uno scenario multi-level blocking (sezioni critiche innestate) il tempo di blocco di un processo, seppur limitato, potrebbe essere elevato.

Per **PI** vale la proprietà per cui se  $n$  è il numero dei processi a priorità più bassa di  $\pi$  ed  $m$  è il numero dei semafori che possono bloccare  $\pi$ , allora  $\pi$  può essere bloccato, al massimo, per la durata di  $\min(n, m)$  sezioni critiche.



# PRIORITY INHERITANCE PROTOCOL: ESEMPIO

Si considerino i processi  $P_1, P_2$  e  $P_3$ , con le seguenti priorità  $\pi_1 \geq \pi_2 \geq \pi_3$ , che condividono la stessa risorsa, il cui accesso è gestito dal semaforo  $S$ , la situazione sarà la seguente:



Per PI, siccome  $P_3$  detiene  $S$ , allora erediterà come priorità il valore massimo tra la sua e quella di tutti i processi in attesa sulla risorsa, ovvero  $P_1$  (il processo bloccato). Non appena  $P_3$  avrà terminato le sue operazioni ed abbandonato la sezione critica, riacquisirà la sua effettiva priorità, lasciando a  $P_1$  la possibilità di accedere alla risorsa.

# PRIORITY CEILING PROTOCOL

Questo protocollo, proposto da *Sha, Rajkuman e Lehoczky* nel 1990, consente di risolvere sia il problema di priority inversion, riducendo i tempi di blocco e assicurando la mutua esclusione, sia di evitare le situazioni di deadlock e di multi-level blocking.

Questo si basa sull'assegnazione di un tetto di priorità o **priority ceiling** a ciascun semaforo  $S$ , indicato come **PC(S)**, di valore pari alla più alta priorità dei processi che possono acquisire il permesso di accedere alla sua sezione critica.

In questo modo il processo  $P_i$  che tenta di ottenere il lock  $S$  verrà sospeso, a meno che il suo valore di priorità non risulti strettamente maggiore di  $PC(S)$ , per tutti i semafori lockati da processi diversi da  $P_i$ . Allora, nel caso in cui il processo  $P_i$  venga sospeso, il task  $P_j$  che detiene l'accesso alla risorsa, poiché ha il valore più alto di PC, erediterà la priorità di  $P_i$ , cioè del processo che ha bloccato, come avviene per il protocollo di *Priority Inheritance*.

Questo approccio consente di prevenire lo stallo e garantisce che un processo possa essere bloccato, al massimo, per la durata di una sola sezione critica.

Tuttavia, rispetto al protocollo PI, è di difficile implementazione.

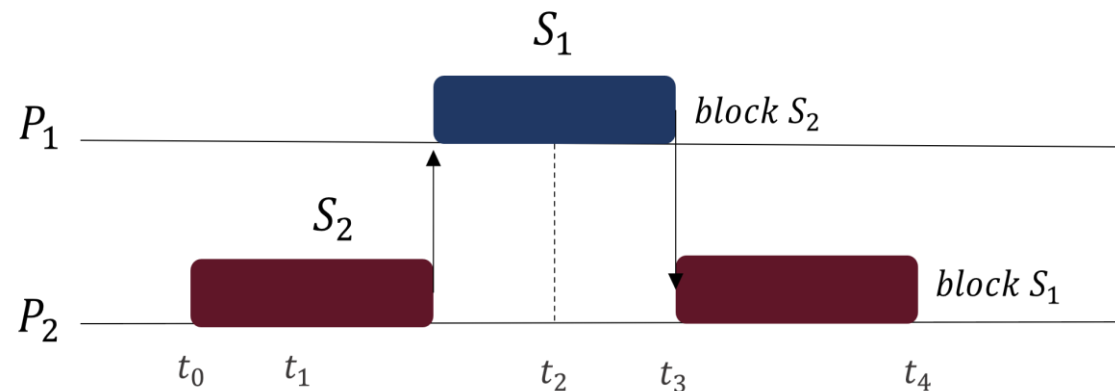
# PIP VS PCP: SITUAZIONE DI DEADLOCK

## Priority Inheritance Protocol

$P_1:: \text{lock}(S_1); \dots \text{lock}(S_2); \dots \text{unlock}(S_2); \dots \text{unlock}(S_1);$

$P_2:: \text{lock}(S_2); \dots \text{lock}(S_1); \dots \text{unlock}(S_1); \dots \text{unlock}(S_2);$

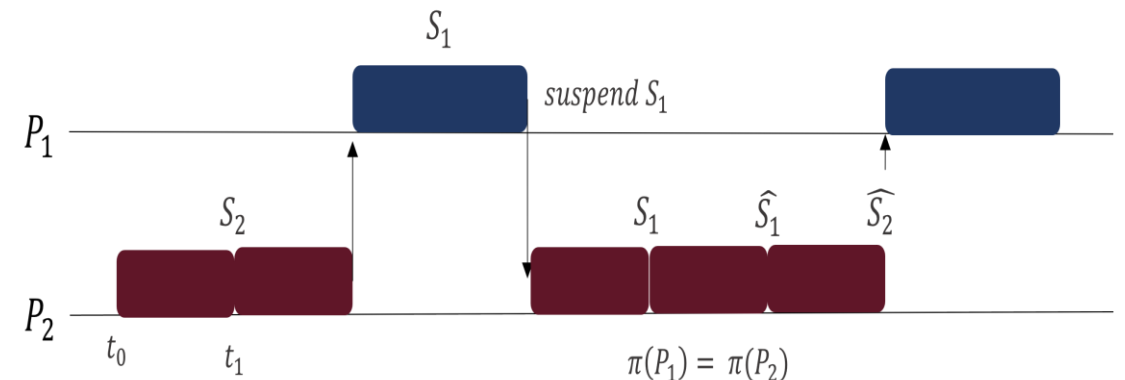
In questa situazione viene rispettato il vincolo piramidale dell'utilizzo dei lock, ossia il rilascio in un ordine inverso a quello di acquisizione, però se  $\pi(P_1) \geq \pi(P_2)$ , si verifica il deadlock (entrambi bloccati).



## Priority Ceiling Protocol

Al tentativo di ottenere  $S_1$ ,  $P_1$  viene sospeso. Riprende  $P_2$ , che ottiene  $S_1$ , rilasciando poi  $S_1$  e  $S_2$ , per poi essere pre-emptato da  $P_1$ , evitando il deadlock.

$$PC(S_1) = PC(S_2) = \max(\pi(P_1), \pi(P_2) = \pi(P_1))$$



# IMMEDIATE INHERITANCE PROTOCOL

È un protocollo che viene utilizzato con lo schema piramidale di assunzione dei lucchetti.

Si può dire che sia simile al Priority Ceiling, in quanto è possibile bloccare un processo al più una volta soltanto, a causa di un processo a più bassa priorità.

Per ottenere un lock, un generico processo  $P_i$  dovrà impostare immediatamente la sua priorità pari ad un valore, così ottenuto:

$$\max(\pi(P_i), PC(S))$$

Inoltre, non avrà bisogno di bloccare il lock  $S$  esplicitamente, perché  $P_i$  non potrà essere pre-emptato, in nessun caso, da processi meno prioritari.

# JITTER

Il fenomeno *Release Jitter* descrive il tempo che impiega un processo a riprendere il controllo del lucchetto, non appena viene risvegliato.

Queste fluttuazioni del tempo di rilascio, dovute al ritardo dell'unità di calcolo, potrebbero, ad esempio, causare il mancato raggiungimento di qualche deadline, perciò è necessario tenerne conto per il calcolo del *tempo di risposta*.

Dunque, si definiscono  $J^{max}$  e  $J^{min}$  come la differenza, massima e minima, tra il tempo di rilascio ed il tempo di invocazione di un processo, che scatta all'inizio del periodo. Si considera la seguente relazione:

$$J = J^{max} - J^{min}$$

Dove  $J$  è il valore di **Release Jitter**, per cui un generico processo, anche se contrassegnato come schedulabile, potrebbe non esserlo più, poiché per  $n$  rilasci si verificano  $n - 1$  fenomeni Jitter.

# STIMA DEL TEMPO DI ESECUZIONE

L'analisi di schedulabilità di un task set, a prescindere dalla strategia adottata, necessita della conoscenza di alcune nozioni, come la durata dei segmenti di codice. Allora, per predirne i tempi di esecuzione si deve stimare il **Worst Case Execution Time**, ossia il caso peggiore del tempo di esecuzione di un generico pezzo di codice, in modo da ottenere un lower bound ed un upper bound del tempo che verrà impiegato per eseguirlo.

Alcuni dei metodi utilizzati per il calcolo del WCET sono:

- La misurazione del tempo del sistema prima e dopo l'esecuzione del codice, oppure durante, se si utilizza un oscilloscopio;
- Il conteggio dei cicli di clock necessari per eseguire ogni istruzione del codice a basso livello;
- Approcci di tipo statico, che non prevedono l'esecuzione del codice;
- Un metodo sistematico, basato sull'analisi dei blocchi base, cioè delle sequenze di istruzioni elementari (blocchi condizionali o cicli) a basso livello;



# PROGETTO

CONFRONTO FRA DUE ALGORITMI DI SCHEDULING



## DATI DEI PROCESSI

I processi per cui si vuole verificare l'esistenza di una schedulazione sono caratterizzati dai dati sottostanti:

$$P_1 = (2,9,9) ; P_2 = (3,6,6); P_3 = (4,24,24);$$

Secondo la seguente relazione di priorità:

$$\pi(P_2) > \pi(P_1) > \pi(P_3)$$

Poiché  $\pi(P_1) = 0.111$ ,  $\pi(P_2) = 0.167$  e  $\pi(P_3) = 0.042$ .

Per cui il processo  $i$ -esimo è definito come  $P_i = (C_i, T_i, D_i)$ , i cui parametri sono, rispettivamente, costo, periodo e deadline. La sua priorità è definita come  $\pi(P_i) = \frac{1}{T_i}$ .

La schedulabilità di questi processi è stata verificata tramite uno script implementato in Matlab, denominato ***scheduling\_test.m***.

# TEST DI SCHEDULABILITÀ

Lo script ***scheduling\_test.m***, che avvia l'analisi di schedulazione dei processi dati, utilizza le seguenti funzioni:

- *process\_data\_extraction*, che riceve in ingresso la matrice contenente i dati dei processi e restituisce costi, periodi e deadline di questi ultimi;
- *processor\_utilization\_factor*, utilizzato per il calcolo del fattore di utilizzazione della CPU;
- *LL\_conditions\_evaluation*, che permette di verificare le condizioni di Liu & Layland;
- *process\_priority\_calculation*, che calcola il valore della priorità di ogni processo;
- *deadline\_check*, che si occupa di verificare il soddisfacimento della relazione che implica che il tempo di risposta di un dato processo sia minore o uguale della sua deadline;
- *response\_time\_exact\_analysis*, che fornisce il tempo di risposta di ciascun processo tramite lo svolgimento dell'analisi esatta di Joseph & Pandya, qualora la condizione sufficiente di Liu & Layland non fosse verificata;

## RISULTATI

Il fattore di utilizzo della CPU per i processi dati è risultato pari all'89%, essendo  $U = 0.89$ .

Da ciò si è potuto dedurre che la condizione necessaria di Liu & Layland ( $U \leq 1$ ) sia soddisfatta, per cui esiste una schedulazione ammissibile per il task set in esame.

Ad esempio, si potrebbe utilizzare l'algoritmo di scheduling real-time Earliest Deadline First.

Al contrario, la condizione sufficiente di Liu & Layland ( $U \leq n(2^{\frac{1}{n}} - 1)$ ), non è verificata. Infatti, il valore limite di Liu e Layland (*bound\_LL*) è pari a 0.78, perciò non è possibile determinare se la schedulazione dei processi in questione sia fattibile con l'algoritmo Rate Monotonic.

Questa incertezza può essere risolta svolgendo l'analisi esatta del tempo di risposta di Joseph & Pandya, dalla quale è risultato che i processi dati possono essere schedulati anche tramite Rate Monotonic.

# ANALISI ESATTA DI JOSEPH E PANDYA

Equazione di ricorrenza del processo  $P_3$ :

Il suo high-priority set  $hp(P_3)$  è costituito dai processi  $P_1$  e  $P_2$ , entrambi più prioritari.

$$R_3^0 = 0; R_3^1 = C_3 = 4;$$

$$R_3^2 = C_3 + \left\lceil \frac{R_3^1}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_3^1}{T_2} \right\rceil \cdot C_2 = 4 + \left\lceil \frac{4}{6} \right\rceil \cdot 3 + \left\lceil \frac{4}{9} \right\rceil \cdot 2 = 9;$$

$$R_3^3 = C_3 + \left\lceil \frac{R_3^2}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_3^2}{T_2} \right\rceil \cdot C_2 = 4 + \left\lceil \frac{9}{6} \right\rceil \cdot 3 + \left\lceil \frac{9}{9} \right\rceil \cdot 2 = 12;$$

$$R_3^4 = C_3 + \left\lceil \frac{R_3^3}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_3^3}{T_2} \right\rceil \cdot C_2 = 4 + \left\lceil \frac{12}{6} \right\rceil \cdot 3 + \left\lceil \frac{12}{9} \right\rceil \cdot 2$$

Siccome  $R_3^5 = R_3^6$   
l'analisi si interrompe.

$$R_3^5 = C_3 + \left\lceil \frac{R_3^4}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_3^4}{T_2} \right\rceil \cdot C_2 = 4 + \left\lceil \frac{14}{6} \right\rceil \cdot 3 + \left\lceil \frac{14}{9} \right\rceil \cdot 2$$

La relazione  $R_3 \leq D_3$  è  
rispettata ( $D_3 = 24$ ).

$$R_3^6 = C_3 + \left\lceil \frac{R_3^5}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_3^5}{T_2} \right\rceil \cdot C_2 = 4 + \left\lceil \frac{17}{6} \right\rceil \cdot 3 + \left\lceil \frac{17}{9} \right\rceil \cdot 2 = 17;$$

# ANALISI ESATTA DI JOSEPH E PANDYA

Equazione di ricorrenza del processo  $P_1$ :

Il suo high-priority set  $hp(P_1)$  è costituito dal processo  $P_2$ .

$$R_1^0 = 0; R_1^1 = C_1 = 3;$$

$$R_1^2 = C_1 + \left\lceil \frac{R_1^1}{T_2} \right\rceil \cdot C_2 = 3 + \left\lceil \frac{3}{6} \right\rceil \cdot 2 = 3 + [0.5] \cdot 2 = 3 + 1 \cdot 2 = 5;$$

$$R_1^3 = C_1 + \left\lceil \frac{R_1^2}{T_2} \right\rceil \cdot C_2 = 3 + \left\lceil \frac{5}{6} \right\rceil \cdot 2 = 3 + [0.833] \cdot 2 = 3 + 1 \cdot 2 = 5;$$

$$R_1^2 = R_1^3$$

La relazione  $R_1 \leq D_1$   
è rispettata ( $D_1 = 6$ ).

Equazione di ricorrenza del processo  $P_2$ :

Il suo high-priority set  $hp(P_2)$  è vuoto perché il processo a maggiore priorità.

$$R_2^0 = 0;$$




$$R_2^1 = C_2 = 2;$$

$$R_2^2 = C_2 = 2;$$

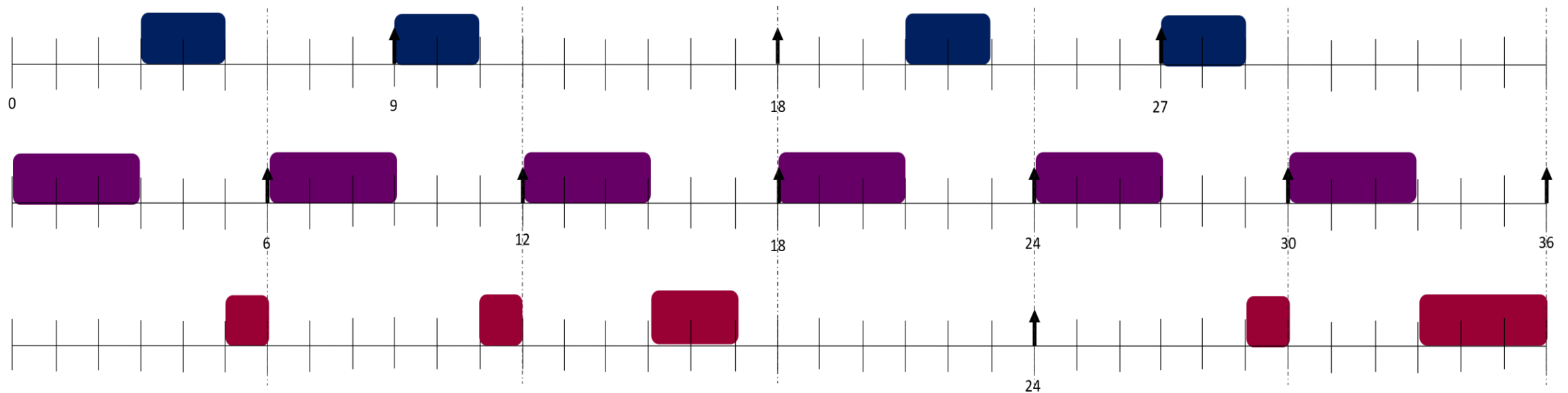
$$R_2^1 = R_2^2$$

La relazione  $R_2 \leq D_2$   
è rispettata ( $D_2 = 9$ ).




# SCHEDULAZIONE CON RATE MONOTONIC

-   $P_1 = (2, 9, 9)$
-   $P_2 = (3, 6, 6)$
-   $P_3 = (4, 24, 24)$

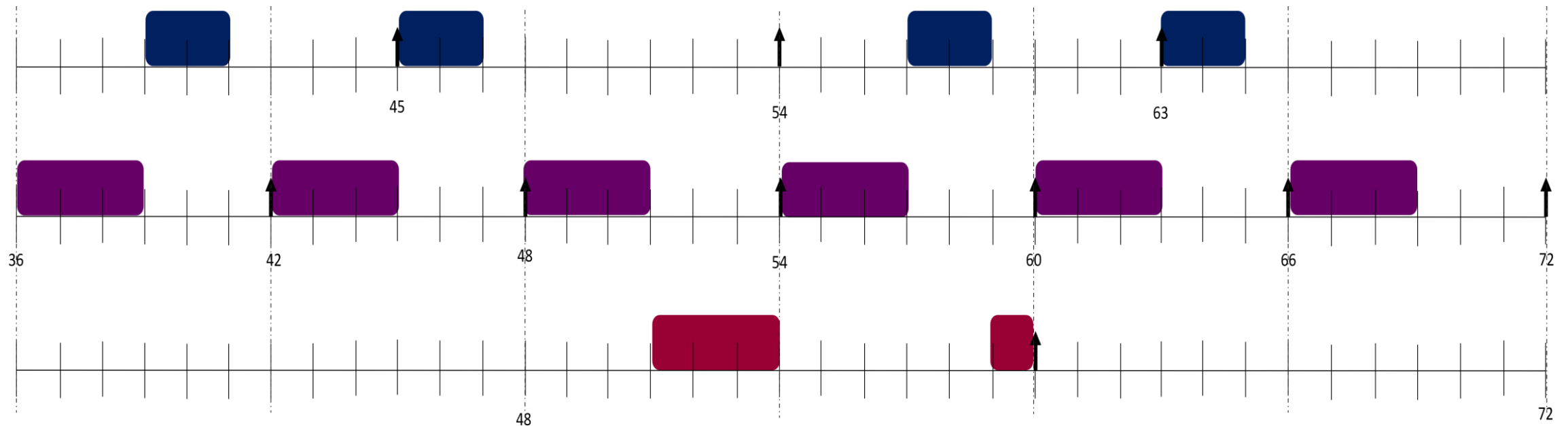
$$\pi(P_2) > \pi(P_1) > \pi(P_3)$$






# SCHEDULAZIONE CON RATE MONOTONIC

-   $P_1 = (2, 9, 9)$
-   $P_2 = (3, 6, 6)$
-   $P_3 = (4, 24, 24)$

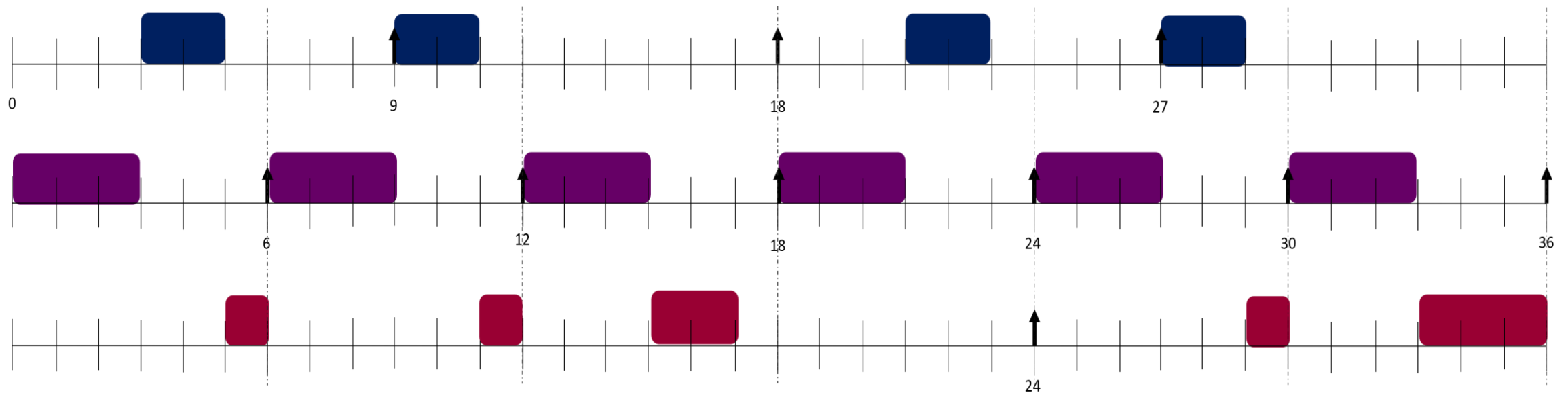
$$\pi(P_2) > \pi(P_1) > \pi(P_3)$$



# SCHEDULAZIONE CON EARLIEST DEADLINE FIRST




-   $P_1 = (2, 9, 9)$
-   $P_2 = (3, 6, 6)$
-   $P_3 = (4, 24, 24)$

$$\pi(P_2) > \pi(P_1) > \pi(P_3)$$

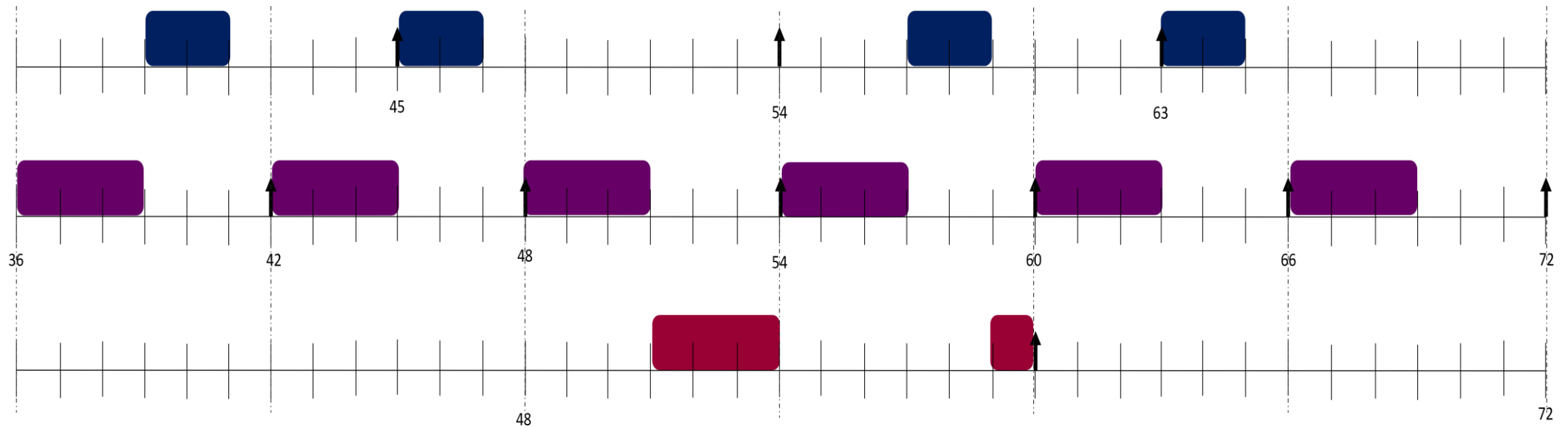




# SCHEDULAZIONE CON EARLIEST DEADLINE FIRST

-   $P_1 = (2, 9, 9)$
-   $P_2 = (3, 6, 6)$
-   $P_3 = (4, 24, 24)$

$$\pi(P_2) > \pi(P_1) > \pi(P_3)$$



## RISULTATI

La schedulazione realizzata tramite ciascuno degli algoritmi scelti è stata riportata soltanto fino all'istante di tempo  $t = 72$ , poiché dopo questo valore si ripete uguale a sé stessa.

Questo fattore è stato calcolato come il *m.c.m.* tra i periodi di attivazione dei tre processi ed è stato impiegato per verificare anche che venisse rispettato il numero minimo di occorrenze che deve avere ogni processo.

I valori ottenuti sono i seguenti:

$$\sigma(P_1) = \frac{72}{T_1} = 8; \quad \sigma(P_2) = \frac{72}{T_2} = 12; \quad \sigma(P_3) = \frac{72}{T_3} = 3;$$

Il loro soddisfacimento può essere riscontrato nei diagrammi di Gantt della schedulazione con Rate Monotonic e con Earliest Deadline First.

## CONFRONTO TRA I DUE ALGORITMI: PRIORITÀ

Entrambi gli algoritmi di schedulazione utilizzati nella simulazione sono di tipo pre-emptivo. Per quanto riguarda le differenze, invece, una prima distinzione da evidenziare riguarda il meccanismo di assegnamento delle priorità dei processi.

Ad esempio, Rate Monotonic assegna ad ogni processo una priorità che viene calcolata come l'inverso del suo periodo di attivazione, una volta soltanto, prima dell'esecuzione dello scheduler.

Dunque, se questo algoritmo si basa su un assegnamento statico delle priorità, Earliest Deadline First, al contrario, prevede che questo valore sia ricalcolato, per tutti i task attivi, ad ogni deadline, adottando un approccio dinamico.

Il valore della priorità, inoltre, sarà inversamente proporzionale a quello della deadline, in modo da mandare in esecuzione il processo con la scadenza imminente.

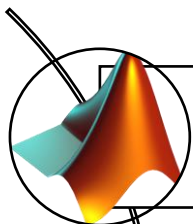
## CONFRONTO TRA I DUE ALGORITMI: SCHEDULABILITÀ

Un'altra differenza, come si è potuto notare dai risultati ottenuti durante il test di schedulazione del task set considerato, consiste nella diversità di comportamento che gli algoritmi dimostrano in condizioni di sovraccarico del processore, ovvero quando il numero dei processi che lo scheduler si trova a dover gestire aumenta.

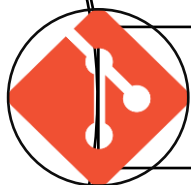
Infatti, se Rate Monotonic, consente di determinare una schedulazione ammissibile soltanto se il valore di occupazione della CPU non supera il 69%, con Earliest Deadline First non sussiste questa limitazione, poiché la schedulabilità viene sempre garantita.

Tuttavia, questo vantaggio viene pagato in termini di complessità computazionale, perché è necessario ricalcolare, ogni volta, le priorità di tutti i processi del task set, anche se questo permette di utilizzarlo sia con task periodici che aperiodici, al contrario di Rate Monotonic, che può essere impiegato esclusivamente per task periodici, poiché il periodo di esecuzione deve essere costante e noto a priori.

# STRUMENTI UTILIZZATI



**Matlab**, un ambiente per il calcolo numerico e l'analisi statistica scritto in C, comprensivo dell'omonimo linguaggio di programmazione.



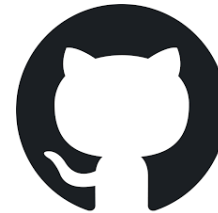
**Git**, il sistema di controllo di versione distribuito.



**Microsoft Office**, una suite di applicazioni desktop, server e servizi di ufficio.



**Email:** [ivonne.rizzuto@gmail.com](mailto:ivonne.rizzuto@gmail.com)



**Github:** [ivochan](#)

# CONTATTI

IVONNE RIZZUTO



GRAZIE PER L'ATTENZIONE!