

Università degli studi di Modena e Reggio Emilia  
Dipartimento di Ingegneria Enzo Ferrari

---

# Real Time Embedded System

---

Anno Accademico 2023/24

# Indice

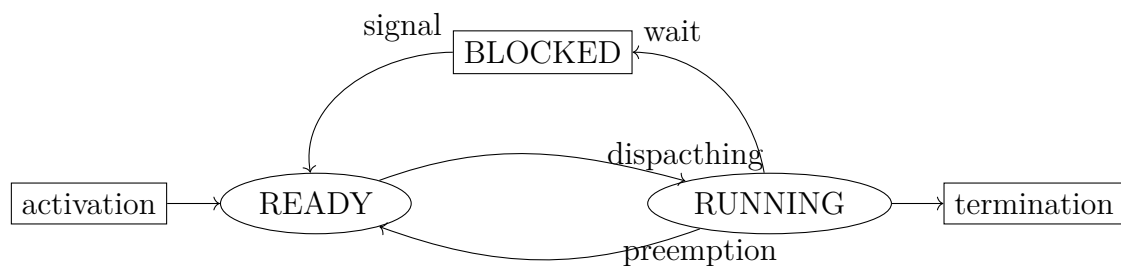
<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Non Real-time scheduling algorithms</b>	<b>6</b>
<b>3</b>	<b>Real-time scheduling algorithms</b>	<b>11</b>
3.1	Earlies Due Date . . . . .	11
3.2	Earliest Deadline First . . . . .	13
<b>4</b>	<b>Periodic Task Scheduling</b>	<b>14</b>
4.1	Timeline Scheduling . . . . .	15
4.2	Priority Scheduling . . . . .	17
4.2.1	Rate Monotonic . . . . .	17
4.3	Earliest Deafline First . . . . .	18
4.4	Task $D_i \leq T_i$ . . . . .	19
4.5	Response Time Analysis . . . . .	19
4.6	Processor Demand Criterion . . . . .	21

# Capitolo 1

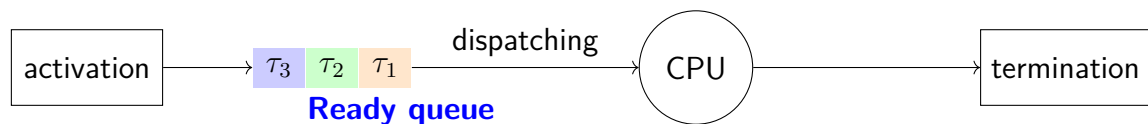
## Introduzione

**Task:** è un insieme di sequenze di istruzioni, che in assenza di altre attività, vengono continuamente eseguite dal processore finché non vengono completate.

Può essere un processo o un thread in base al sistema operativo.



**Ready Queue:** i task “pronti” (*ready*) sono contenuti all’interno di una coda di attesa, anche nota come *ready queue*. Le strategie con cui vengono scelti i task dalla coda per essere eseguiti sulla *CPU* sono gli **scheduling algorithms**.

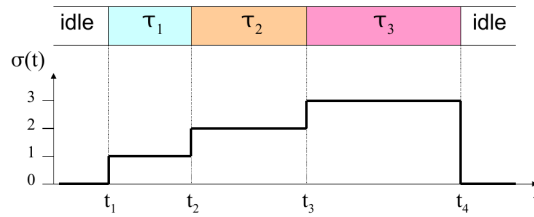


**Scheduling** può essere definito **preemptive** ovvero se il task in esecuzione in un certo istante di tempo  $t_i$  può essere temporaneamente sospeso per eseguire un task con importanza maggiore, mentre si dice **non-preemptive** se il task in esecuzione non può essere sospeso finché non viene completato.

**Schedule:** uno *schedule* è un particolare assegnamento di task ad un processore. Dato un **taskset**  $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$  uno *schedule* viene mappato to  $\sigma$ :

$$\mathbb{R}^+ \rightarrow \mathbb{N} \mid \forall t \in \mathbb{R}^+ \quad \sigma(t) = \begin{cases} k > 0 & \text{if } \tau_k \text{ is running} \\ 0 & \text{if the processor is idle} \end{cases}$$

Consideriamo il *task set*:  $\{\tau_1, \tau_2, \tau_3\}$



Nei punti  $t_1, t_2, t_3$  e  $t_4$  viene eseguito un **content switch**, ogni intervallo di tempo  $[t_i, t_{i+1})$  viene chiamato **time slice**.

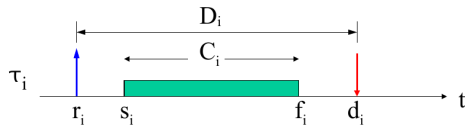


Figura 1.1: Real-time tasks

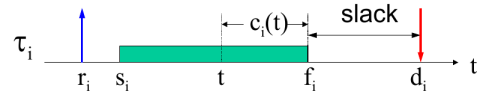


Figura 1.2: Real-time tasks

- $r_i$  è il **request time**.
- $s_i$  è lo **start time** ovvero il tempo in cui il task inizia l'esecuzione.
- $C_i$  è il tempo di esecuzione in caso peggiore (**WCET**).
- $d_i$  è la **deadline assoluta**, mentre  $D_i$  è la **deadline relativa**.
- $f_i$  è il **finishing time** ovvero il tempo effettivo in cui il task completa il suo lavoro
- **lateness**:  $L_i = f_i - d_i$ , è quindi la differenza tra il tempo di fine del task e la sua deadline assoluta, se  $\leq 0$  allora il task ha rispettato la sua deadline se no la deadline è stata missata [**tardiness**:  $\max(0, L_i)$ ]

- **Residual WCET**:  $c_i(t)$
- **laxity (o slack)**:  $d_i - t - c_i(t)$

**Tasks vs. Jobs**: un task è un infinita sequenza di istanze che vengono ripetute [*jobs*]. È possibile differenziare varie tipologie di *task* in base a quale deve essere la loro garanzia di rispetto delle loro *deadline*:

1. **Hard Task**: tutti i *jobs* devono rispettare le proprie deadline, mancare una deadline comporta serie conseguenze.
2. **Firm Task**: solo alcuni *jobs* possono missare la loro deadline.
3. **Soft Task**: i *jobs* possono missare la loro deadline, l'obiettivo è quello di massimizzare la **responsiveness**.

Un sistema operativo capace di gestire *hard task* viene chiamato **hard real-time system**. I *tasks* possono avere due modalità di **attivazione**:

1. **time driven**: anche noti come **tasks periodici**, i task vengono automaticamente attivati dal *kernel* ad intervalli regolari. Definiamo il task come:  $\tau_i(C_i, T_i, D_i)$  dove  $T_i$  è il periodo a cui quel task viene invocato.

$$\begin{cases} r_{i,k} = \Phi_i + (k-1) \cdot T_i & k = 1 \rightarrow r_{i1} = \Phi_i \\ d_{i,k} = r_{i,k} + D_i \end{cases}$$

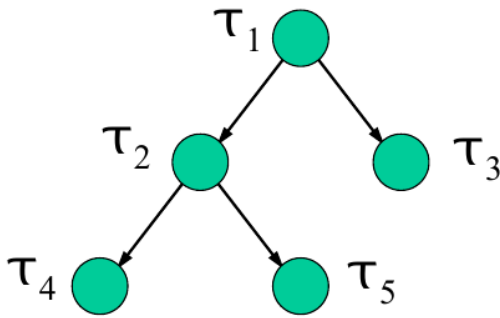
2. **event driven**: anche noti come **tasks aperiodici**, ovvero il task viene attivato all'arrivo di un evento o per un'invocazione esplicita della sua primitiva di invocazione. A loro volta possono dividersi in:

- **aperiodic**:  $r_{i,k+1} > r_{i,k}$
- **sporadic**:  $r_{i,k+1} \geq r_{i,k} + T_i$

Sui *tasks* possono essere imposti dei vincoli, che si differenziano in:

- **timing constraints**: ovvero dei vincoli sul tempo di esecuzione [*deadline, activation, completion e jitter*], possono essere **impliciti** o **espliciti**:

- **explicit constraints**: sono definite nelle specifiche del sistema di attivazione: apertura della valvola ogni 10s
- **implicit constraints**: non appaiono nelle specifiche direttamente, ma devono essere rispettate per seguire i vincoli di utilizzo del sistema: schivare ostacoli mentre si corre ad una velocità  $v$ .
- **precedence constraints**: alcuni task devono rispettare delle precedenze di esecuzione, normalmente specificate da un **Directed Acyclic Graph**:



predecessore

$$\tau_1 \prec \tau_4$$

predecessore immediato

$$\tau_1 \rightarrow \tau_2$$

- **resource constraints**: per preservare *data consistency* bisogna accedere alle risorse condivise in **mutua esclusione**, che però introduce un *delay*.

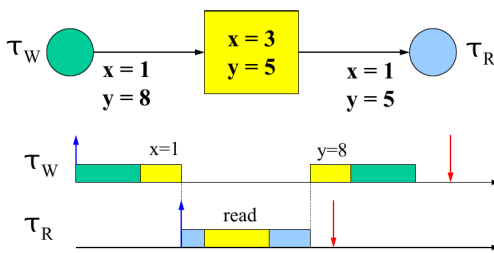


Figura 1.3: *no mutual exclusion*

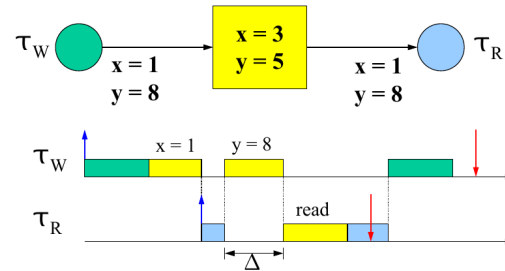


Figura 1.4: *mutual exclusion*

Mentre si analizza un *tasks set* e si cerca che il tempo di esecuzione sia vincolato da vincoli imposti in fase di progettazione, ad esempio  $t_r \leq 10$ , anche se si aumenta il numero di processori, si diminuisce il tempo di esecuzione dei task o si rilassano i vincoli

di precedenza, se non si uno *scheduler* appropriato si rischia in ogni caso di missare i vincoli imposti. L'approccio più *safe* è quello di utilizzare meccanismi predicibili del kernel e analizzare il sistema per predirne il comportamento. La concorrenza deve essere progettata utilizzando:

- appropriati algoritmi di *scheduler*.
- appropriati protocolli di **sincronizzazione**.
- efficienti meccanismi di **comunicazione**.
- predicibilità negli *interrupt handling*.

# Capitolo 2

## Non Real-time scheduling algorithms

Lo *scheduling* è l'attività che permette di selezionare quale processo o *thread* bisogna eseguire come successivo. In generale nei sistemi operativi, possiamo distinguere tre tipologie di *scheduling*:

- ***long term scheduling***: prima di creare il processo, viene deciso se attivarlo o meno. Viene implementato tramite un **test di ammissione**, se il processo passa questo controllo allora viene inserito nella *ready queue*, se no viene interrotto finché non gli viene permesso di essere *schedulato* [ se il *load* del processore è troppo alto il nuovo task rischia di essere solo di “intralcio” ].
- ***medium term scheduling***: permette di decidere se un processo deve essere *preemptato* o meno.
- ***short term scheduling***: decide quale processo deve essere eseguito come successivo. Possiamo distinguere:
  - ***selection function***: decide quale processo viene selezionato dalla *ready queue*, seguendo alcune regole.
  - ***decision mode***: quando la decisione è stata presa si può comportare in maniera *preemptive* oppure *non-preemptive*.



**Scheduling Criteria:** come si possono valutare le performance di uno *scheduler*:

- **user-oriented:** si va ad analizzare il *response-time* del processo.
- **system-oriented:** si va ad analizzare il *throughput*, ovvero quanto lavoro il sistema può eseguire in un certo intervallo di tempo.

Per quanto le performance siano importanti in certe circostate ci possono interessare la **predicibilità** (*real-time system*) o la **fairness**.

Tra i processi possiamo differenziare anche il tipo di risorsa che viene utilizzata: **CPU-Bound** e **I/O Bound**, nel primo caso il processo è orientato a lavorare sul processore, mentre nel secondo caso i processi possono essere in attesa di un *I/O device*. La stragrande maggioranza dei processi è un mix dei due.

Uno *schedule*  $\sigma$  si dice **fattibile** (*feasible*) se tutti i *tasks* sono capaci di completare entro un insieme di vincoli.

Un *tasks set*  $\mathcal{T}$  si dice **schedulable** se esiste uno *schedule* fattibile per esso.

**The General Scheduling Problem:** dato un *tasks set*  $\mathcal{T}$  di  $n$  *tasks*, un set  $\mathcal{P}$  di  $m$  processori e un set  $\mathcal{R}$  di  $r$  risorse, trovare un assegnamento di  $\mathcal{P}$  e  $\mathcal{R}$  per  $\mathcal{T}$  che produce uno *schedule* fattibile.

È stato dimostrato nel 1975 da Garey e Johnson che il *general scheduling problem* rientra nella categoria **NP hard**. È però possibile, rilassando i vincoli e specificando certe condizioni, ricordarci ad un algoritmo *polynomial time*.

Per il ora consideriamo:

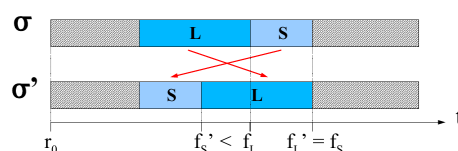
- processore singolo
- *fully preemptive tasks*
- attivazione simultanea
- nessun vincolo di precedenza
- nessun vincolo sulle risorse

Algorithm taxonomy	
<i>preemptive</i>	<i>non-preemptive</i>
<b><i>off line:</i></b> tutte le decisioni sullo scheduling vengono prese prima dell'attivazione dei task, normalmente lo <i>schedule</i> viene salvato in una tabella ( <b><i>table-driven scheduling</i></b> )	<b><i>on line:</i></b> le decisioni di scheduling vengono prese <i>runtime</i> sul set dei task attivi
<b><i>static:</i></b> le decisioni di scheduling vengono prese basandosi su parametri fissati, staticamente assegnati al task prima dell'attivazione	<b><i>dynamic:</i></b> le decisioni di scheduling vengono prese su parametri che possono variare nel tempo
<b><i>best effort:</i></b> trova sempre uno <i>schedule</i> fattibile, se esiste	<b><i>optimal:</i></b> fa del suo meglio per trovare uno <i>schedule</i> fattibile, se esiste, ma non lo garantisce.

Le *policies* classiche di **scheduling**, che però non sono adatte per sistemi *real-time*, sono:

1. **First Come First Served (FCFS)**: assegna l'utilizzo della CPU al task basandosi sull'ordine di arrivo, non è *preemptive*, è dinamico, online e *best effort*.  
→ molto **impredicibile**: il *response time* è fortemente dipendente dall'ordine di arrivo dei task.
2. **Shorter Job First (SJF)**: seleziona il task che ha il minor *computational time*, può essere sia *preemptive* che *non-preemptive*, è statico (il parametro  $C_i$  è fissato da configurazione), può essere usato sia *online* che *off-line* e permette di minimizzare la *response time media*.

**Dimostrazione dell'ottimalità di SJF**: consideriamo uno scheduler  $\sigma \neq \text{SJF}$  e un'altro scheduler  $\sigma'$  che è uguale a SJF fino all'istante  $f_s$



Presi due task  $L$  e  $S$  che hanno *request time*  $r_i$   $i \in \{L, S\}$  e *finish time*  $f_i$   $i \in \{L, S\}$ . Lo *schedule*  $\sigma$  schedula il task  $L$  prima (non conforme con SJF), mentre  $\sigma'$  schedula il task  $S$  come primo (conforme a SJF). Possiamo dire che  $f'_L = f_S$  in quanto la somma del tempo dei due task non cambia, ma cambia solo l'ordine di schedulazione. È intuitivo che il *finish time* del primo task è però sbilanciato verso lo scheduler  $\sigma'$  infatti avremo  $f'_S < f_L$ .

Avremo perciò  $f'_S + f'_L \leq f_S + f_L$

$$\rightarrow \bar{R}(\sigma') = \frac{1}{n} \cdot \sum_{i=1}^n (f'_i - r_i) \leq \frac{1}{n} \cdot \sum_{i=1}^n (f_i - r_i) = \bar{R}(\sigma)$$

Lo scheduler  $\sigma'$  è equivalente a SJF solo fino all'istante  $f'_L = f_S$ , bisogna andare quindi ad iterare su ogni scheduler  $\sigma \in \{\sigma', \sigma'', \dots, \sigma^*\}$ , andando a riproporre l'analisi appena condotta avremo che:  $\bar{R}(\sigma) \geq \bar{R}(\sigma') \geq \bar{R}(\sigma'') \geq \dots \geq \bar{R}(\sigma^*)$

$\rightarrow \sigma^* = \sigma_{sjf}$  e quindi avremo che  $\bar{R}(\sigma_{sjf})$  è la **minima response time media** ottenibile da ogni **algoritmo**.

**SJF non è un algoritmo fattibile per il *Real-Time*.**

3. **Priority Scheduling:** ad ogni task viene assegnata una **priorità**, il task con la priorità maggiore viene eseguito come primo, mentre per i task con pari priorità siene eseguito uno scheduler o FCFS o RR. Può essere utilizzato per fini *real-time* se le priorità sono assegnate seguendo specifiche regole. Lo *scheduler POSIX* è uno scheduler con 99 priorità. Può essere sia statico che dinamico.

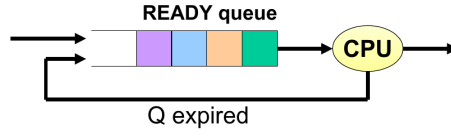
- può avere problemi di **starvation** infatti dei task a bassa priorità possono accumulare ritardo per via della *preemption* dovuto a task con più alta priorità.
- Una possibile **soluzione** è quella dell'**aging** ovvero che la priorità viene incrementata con il passare del tempo:

$$p_i \propto \frac{1}{C_i} \simeq \text{SJF}$$

$$p_i \propto \frac{1}{r_i} \simeq \text{FCFS}$$

4. **Round Robin:** la *ready queue* viene servita con un **FCFS**, ma il sistema conosce il concetto di **time quantum (Q)**, ogni task  $\tau_i$  non può eseguire più un **Q** unità

di tempo. Quando  $Q$  scade,  $\tau_i$  viene riaccodato nella *ready queue*.



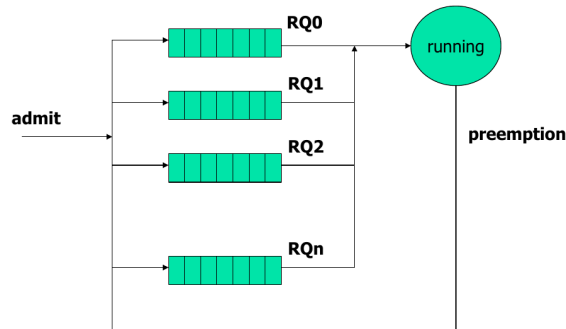
Introduce il concetto di **time sharing**: ovvero che ogni task esegue in solitaria su un “processore virtuale”  $n$  volte più lento rispetto a quello reale.

$$R_i \simeq (nQ) \frac{C_i}{Q} = nC_i$$

Se  $Q > \max(C_i)$  allora  $RR \equiv FCFS$ , e se consideriamo che ogni volta che viene *preemptato* un task bisogna eseguire un *context switch* definito da un tempo  $\delta$  allora avremo che:

$$R_i \simeq n \cdot (Q + \delta) \frac{C_i}{Q} = nC_i \cdot \left(\frac{Q+\delta}{Q}\right)$$

5. **Multiple-feedback Queues**: questo *scheduler* consiste in:  $N$  code, ognuna delle quali viene ordinata tramite FIFO a unità di tempo *quantum* fisse. Lo *scheduler* sceglie il primo processo dalla coda con più alta priorità e imposta un timer a  $Q$ . Consideriamo  $RQ_k$  come la coda priorità maggiore che ha un task pronto per essere eseguito. Se il processo viene completato entro o si blocca prima che il scada bisogna selezionare il processo successivo dalla coda con più alta priorità e impostare il timer, se no sposta il processo nella coda  $RQ_{k+1}$ . Quindi in maniera periodica, se un processo non viene completato allora viene “spostato” nella priorità più alta (questo viene fatto per evitare *starvation*).



# Capitolo 3

## Real-time scheduling algorithms

I task possono essere schedulati i task in base alla *deadline*, che può essere quella **relativa** o **assoluta**.

### 3.1 Earliest Due Date

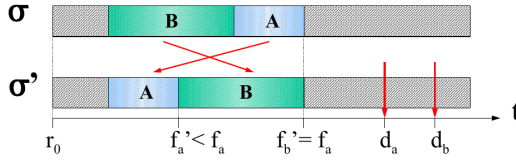
Esegue come primo task quello con la *deadline* **relativa** più imminente.

- tutti i task arrivano simultaneamente.
- priorità fissati
- i task sono *preemptabili*.
- permette di minimizzare la massima **lateness**  $L_i \rightarrow$  nessun task manca la sua *deadline*.

$$\begin{cases} L_i = f_i - d_i & f_i \text{ è il finish time e } d_i \text{ è la deadline} \\ L_{max} = \max_i(L_i) \end{cases}$$

**Dimostrazione dell'ottimalità di EDD:** consideriamo uno scheduler  $\sigma \neq \text{EDD}$  e un'altro scheduler  $\sigma'$  che è uguale a EDD fino all'istante  $f_a$ . Consideriamo due task:  $A$  e  $B$  che hanno *deadline*  $d_a$  e  $d_b$  dove nel caso dello scheduler  $\sigma$  ( $\neq \text{EDD}$ ) viene schedulato prima  $B$  e dopo  $A$  in quanto  $d_a < d_b$  mentre nel caso dello scheduler  $\sigma'$  ( $\simeq$

EDD) viene schedulato prima A e dopo B. Definiamo  $f_a$  e  $f'_a$  come l'istante di tempo di fine del task A e  $f_b$  e  $f'_b$  come l'istante di tempo di fine del task B. Siccome i tempi di esecuzione totale dei due task in entrambi gli scheduler sono uguali allora possiamo dire che  $f'_b = f_a$  e siccome il  $C'_a < C_b + C_a$  e  $s'_a < s_a$  avremo che  $f'_a < f_a$ . Possiamo andare a minimizzare la massima *lateness* utilizzando il seguente schema



$$L_{max} = L_a = f_a - d_a$$

$$L'_{max}(\sigma') < L_{max}(\sigma)$$

$$\begin{cases} L'_a = f'_a - d_a < f_a - d_a \\ L'_b = f'_b - d_b = f_a - d_b < f_a - d_a \end{cases}$$

Siccome  $\sigma'$  è equivalente ad EDD solo fino all'istante di tempo  $f_a = f'_b$  per poter iterare fino "all'infinito" è necessario andare a considerare uno scheduler  $\sigma \in \{\sigma', \sigma'', \dots, \sigma^*\}$ . Iterando il ragionamento del confronto fatto per uno scheduler  $\sigma$  e  $\sigma'$  con tutto l'insieme degli scheduler, in questo modo avremo che:  $L_{max}(\sigma') \geq L_{max}(\sigma'') \geq \dots \geq L_{max}(\sigma^*)$  andiamo a dimostrare che  $\sigma^* = \sigma_{EDD}$  dove  $L_{max}(\sigma_{EDD})$  è il minimo valore ottenibile da ogni algoritmo di scheduler.

Un *task set*  $\mathcal{T}$  è **fattibile** se  $\forall i f_i \leq d_i$  quindi avremo che il *finish time del task* è pari a  $f_i = \sum_{k=1}^i C_k$  ma quindi avremo il vincolo che  $\forall i \sum_{k=1}^i C_k \leq D_i$  ovvero che il **WCET** ovvero l'*worst case execution time* per ogni task sia minore della loro *deadline* assoluta.

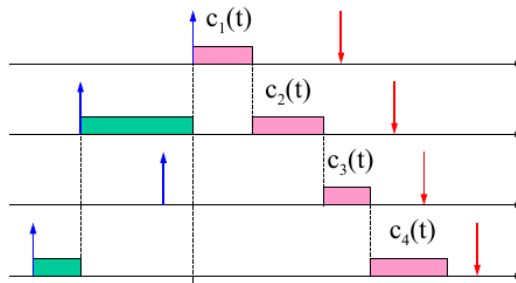
**Complessità:**

- per ordinare il *task set*  $\mathcal{O}(n \log n)$
- per garantire l'intero *task set*  $\mathcal{O}(n)$

## 3.2 Earliest Deadline First

Seleziona il task da eseguire considerando la *deadline assoluta* più imminente.

- i task possono arrivare in qualunque istante di tempo  $t_i$ .
- priorità **dinamiche**.
- *fully preemptive tasks*.
- minimizza la *lateness*  $L_{max}$  massima.



EDF garantisce la fattibilità della schedulabilità se  $\forall i \sum_{k=1}^i c_k(t) \leq d_i - t$ .

**Complessità:**

- per inserire un nuovo task all'interno del *task set* il *task set*  $\mathcal{O}(n)$
- per garantire un nuovo task  $\mathcal{O}(n)$

# Capitolo 4

## Periodic Task Scheduling

*Scheduling* di *tasks* **periodici** o **sporadici** (aperiodici). Definiamo un **task periodico**  $\tau_i(C_i, T_i)$  con  $C_i$  il *worst case execution time* e  $T_i$  il **periodo** per il quale il task  $\tau_i$  deve essere eseguito.

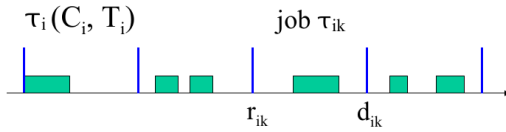


Figura 4.1: *periodic task*

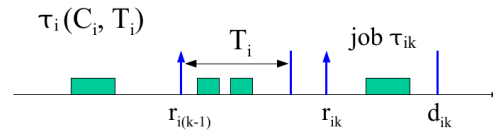


Figura 4.2: *sporadic task*

Per ogni task periodico, bisogna garantire che:

- ogni job  $\tau_{ik}$  venga attivato in  $r_{ik} = (k - 1) \cdot T_i$ .
- ogni job  $\tau_{ik}$  completi la sua esecuzione entro  $d_{ik} = r_{ik} + D_i$ .

Anche nel caso di **task aperiodici** possiamo definirli  $\tau_i(C_i, T_i)$ , in questo caso però  $T_i$  non è il periodo nel quale per il quale si ripete il task ma indica il *delay* minimo di attivazione tra un task  $\tau_{ik}$  e un task  $\tau_{i(k+1)}$ , infatti bisogna garantire per ogni task sporadico che:

- ogni job  $\tau_{ik}$  viene attivato in un istante  $r_{ik} \geq r_{i(k+1)} + T_i$ .
- ogni job  $\tau_{ik}$  completi la sua esecuzione entro una *deadline* relativa  $d_{ik} = r_{ik} + D_{ik}$

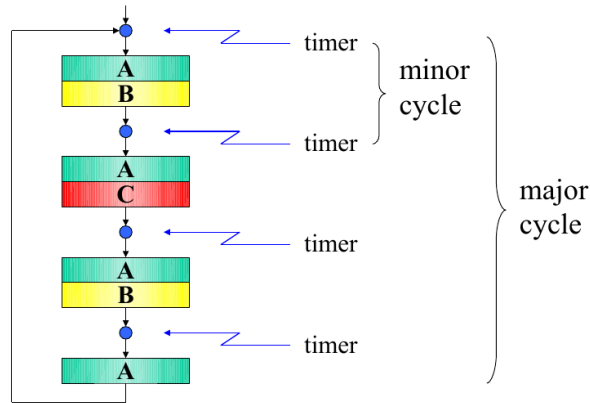


## 4.1 Timeline Scheduling

È una tipologia di *scheduling offline* è stata utilizzata per anni nei contesti in cui era richiesto un *hard real time* per via della delicatezza delle circostanze di uso (sistemi militari, navigazioni e sistemi di monitoraggio). Può essere chiamato anche ***cycle executive*** o ***cyclic scheduling***.

Il funzionamento era tale che l'asse del tempo venisse divisa in intervalli con lunghezza uguale, anche chiamati ***time slots***, ogni task viene allocato staticamente in un certo slot e in un certo ordine per venire incontro ai *request rate* desiderati. L'esecuzione per ogni slot viene attivato tramite un ***timer***. Consideriamo un task set  $\mathcal{T} = \{\tau_1, \tau_2, \dots, \tau_k\}$  e che  $\forall \tau_i \in \mathcal{T} \exists (C_i, T_i)$ , in questo caso  $T_i \equiv D_i$  ovvero che il periodo del task corrisponde con la sua *deadline* assoluta. Definiremo:

- il ***minor cycle*** come  $\Delta = \gcd(T_i, T_j) \forall T_i, T_j \in \mathcal{T}, i \neq j$
- il ***major cycle*** come  $\mathbf{T} = \text{lcm}(T_i, T_j) \forall T_i, T_j \in \mathcal{T}, i \neq j$ 
  - nel caso in cui ci siano task sporadici come andiamo a valorizzare il *major cycle*



**Vantaggi**

- implementazione semplice (non viene richiesto alcun sistema operativo *real-time*)
- ogni procedura condivide un *address space* comune
- basso *overhead* a tempo di esecuzione
- permette di controllare i *jitter*

**Svantaggi**

- non è robusto contro *overload* del sistema
- nel caso di aggiunta di un nuovo task è molto difficile l'espansione dello *scheduler*
- non è facile gestire task aperiodici
- tutti i processi devono avere periodo multiplo del *minor cycle*
- è difficile includere processi con un periodo lungo
- difficile da costruire e da mantenere
- tutti i processi con un *WCET* variabile devono essere *splittati* in procedure con lunghezza fissa. (il determinismo non è richiesto, ma la predicibilità sì)

Durante un ***overload*** si possono “considerare” due vie: la prima è quella di lasciar finire il task, che però comporta un **effetto domino** che va a portare delle ripercussioni anche su tutti gli altri task e che potrebbe portare ad un ***timeline break***; il secondo caso è gestire l'*overload* con l'interruzione del task, in questo caso il sistema potrebbe rimanere in uno stato **inconsistente**. In un altro caso si ha necessità di incrementare il *WCET* di un task, ma se la somma dei *WCET* dei task in esecuzione nel  $\Delta$  è maggiore del  $\Delta$ , allora sarà necessario dividere uno degli  $n$  task in quel  $\Delta$  di tempo in modo da evitare un *timeline break*.

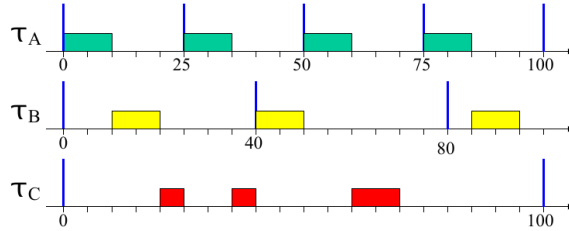
task	$T$	$T'$
$A$	25 ms	25 ms
$B$	50 ms	40 ms
$C$	100 ms	100 ms
<i>minor cycle</i>	$\Delta = 25\text{ms}$	$\Delta = 5\text{ms}$
<i>major cycle</i>	$\mathbf{T} = 100\text{ms}$	$\mathbf{T} = 200\text{ms}$

## 4.2 Priority Scheduling

Ad ogni task viene assegnata una priorità basata sui suoi vincoli temporali, è possibile verificare la fattibilità di uno *schedule* usando tecniche analitiche. I task sono eseguiti su un *priority-based kernel*.

### 4.2.1 Rate Monotonic

Ad ogni task viene assegnata una **priorità fissa** in maniera proporzionale alla sua frequenza. In caso di *overhead* sull'esecuzione di singoli job l'**RM** è più solido del *timeline schedule*.



Definiamo l'utilizzazione della *CPU* da parte di un task come  $U_i = \frac{C_i}{T_i}$ , in questo modo possiamo andare a calcolare l'utilizzazione della *CPU* su tutti i task definiti:

$$U_p = \sum_{i=1}^n \frac{C_i}{T_i} \quad \leftarrow U_p \text{ processor load}$$

In questo modo riusciamo a valutare il **carico del processore** che però **non** è una condizione **sufficiente** per garantire la **schedulabilità** di un *task set*, ma solo **necessaria**, infatti ci può dire se il *task set* non è schedulabile in quanto  $U_p > 1$  indica che il processore è *overloaded*, ma nel caso in cui  $U_p < 1$  ci possono essere dei casi in cui il *task set* non può essere schedulato tramite *RM*.

È possibile però identificare un punto, noto come  $U_{lub}$  [*least upper bound*] tale per cui sia possibile avere un **test di schedulabilità** sia **necessario** che **sufficiente**. Infatti nel caso in cui  $U_p \leq U_{lub}$  il *task set* è sicuramente schedulabile tramite *RM*. Mentre nel caso in cui  $U_{lub} < U_p \leq 1$  non possiamo dire niente di certo sulla fattibilità del *task set*.

Per *Rate Monotonic*  $U_p^{RM} = n \cdot (2^{\frac{1}{n}} - 1)$  quindi avremo che:

$$\lim_{n \rightarrow \infty} U_{lub} = \log_2 2$$

Definiamo il **Critical Instant** come l'istante di tempo in cui è presente il *response time* maggiore, è stato dimostrato che equivale, considerando un *task set*  $\mathcal{T}$  all'istante in cui arrivano in corrispondenza tutti i task a più alta priorità.

Dal punto di vista della schedulabilità un task sporadico può essere considerato come un task periodico e quindi è possibile calcolarsi il suo  $U_p$  e confrontarlo con l' $U_{lub}$  del *task set*.

**Rate Monotonic** è **ottimo**, infatti se esiste un assegnamento a priorità fisse che permette la fattibilità di uno *schedule* per un *task set*  $\mathcal{T}$  allora l'assegnamento *RM* è fattibile per il *task set*  $\mathcal{T}$ , al contrario se un *task set* non è schedulabile con *RM* allora non esiste nessun altro assegnamento a priorità fissa che riesca a rendere fattibile la schedulazione del *task set*.

**Hyperbolic Bound:**

$$\prod_{i=1}^n (U_i + 1) \leq 2 \quad \text{vs.} \quad \sum_{i=1}^n U_i \leq n \cdot (2^{\frac{1}{n}} - 1)$$

### 4.3 Earliest Deadline First

Ogni *job* riceve una *deadline assoluta*  $d_{i,k} = r_{i,k} + D_i$ , in ogni istante di tempo il processore viene assegnato al *job* con la *earliest absolute deadline*, con EDF, qualsiasi set di attività può utilizzare il processore fino al 100%.

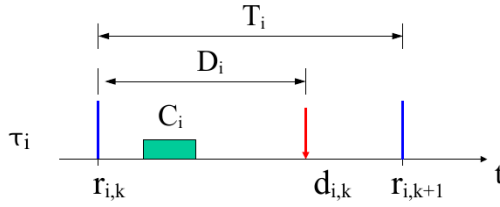
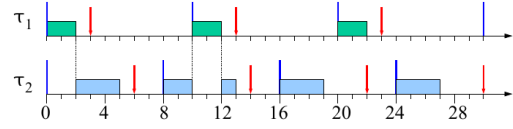
**EDF** è **ottimale** ovvero se esiste uno *schedule* per  $\mathcal{T}$  allora **EDF** genererà uno *schedule* fattibile, viceversa se  $\mathcal{T}$  non è schedulabile con **EDF** allora non sarà schedulabile per nessun altro algoritmo di scheduling. Nel caso di **EDF** il test **necessario** e **sufficiente** affinché un *task set* sia schedulabile è che  $U_p \leq 1$ .

EDF	RM
- è molto più efficiente	- è più semplice implementarla su un sistema operativo commerciale
- riduce i <i>context switches</i>	- è più predicibile durante <i>overload</i>

## 4.4 Task $D_i \leq T_i$

Andiamo ora a considerare il caso in cui un task  $\tau_i(C_i, D_i, T_i)$  ovvero in cui la *deadline* assoluta non coincide con il *request time* ovvero in cui  $D_i < T_i$ . In questi casi possiamo utilizzare due tipologie diverse di *scheduler*:

- **fixed priority: Deadline Monotonic**  $p_i \propto \frac{1}{D_i}$
- **dynamic priority: Earliest Deadline First**  $p_i \propto \frac{1}{d_i}$

Figura 4.3: task con  $D_i \leq T_i$ Figura 4.4: *deadline monotonic*

## 4.5 Response Time Analysis

Siccome nel caso di uno scheduler *deadline monotonic* ricavare l'*utilization bound* non è utile in quanto sono fattibili anche *task set* con  $U_p > 1$ .

Il *response time analysis* è un test **sufficiente** e **necessario** per la schedulabilità di un *task set*  $\tau_i$ , per ogni task  $\tau_i$  calcolare l'*interference*  $I_i$  che può essere causata da task a più alta priorità, possiamo quindi ora calcolare il *response time* come  $R_i = C_i + I_i$  e possiamo verificare che questo sia minore della *deadline* relativa  $R_i \leq D_i$ .

Per calcolare l'interferenza di un task  $\tau_k$  (a priorità alta) su  $\tau_i$  (a priorità minore) nell'intervallo  $[0, R_i]$ :

$$I_{ik} \lceil \frac{R_i}{T_k} \rceil \cdot C_k \rightarrow I_k = \sum_{k=1}^{i-1} \lceil \frac{R_i}{T_k} \rceil \cdot C_k$$

Il calcolo del **response time** è invece iterativo:

$$\begin{cases} R_i^0 = C_i \\ R_i^{(s+1)} = C_i + \sum_{k=1}^{i-1} \lceil \frac{R_i^{(s)}}{T_k} \rceil \cdot C_k \end{cases} \rightarrow R_i^{(s+1)} = R_i^{(s)}$$

**Esercizio**

Consideriamo un *task set* del tipo: 
$$\begin{cases} \tau_1 = (3, 6) \\ \tau_2 = (7, 28) \\ \tau_3 = (5, 28, 30) \end{cases}$$

andiamo prima a valutare la *schedulabilità* andando a calcolare l'*utilitation bound* modificata, visto che c'è almeno un task che ha  $T_i \neq D_i$ .

$$\begin{aligned} U_p^* &= \frac{C_1}{T_2} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \\ &= \frac{3}{6} + \frac{7}{28} + \frac{5}{28} = 0.93 \quad \stackrel{?}{\leq} \quad U_p^* = n \cdot (\sqrt[n]{2} - 1) = 0.78 \end{aligned}$$

- $\tau_1 \rightarrow R_i = C_i - I_i = 3 - 0 = 3 \stackrel{?}{\leq} 6$  **OK**.

Il task  $\tau_1$  è **schedulabile**, siccome siamo in priorità fisse.

- il task  $\tau_2$  è **schedulabile**

$$R_2 = C_2 + \lceil \frac{R_2}{T_1} \rceil \cdot C_1 = 7 + \lceil \frac{7}{6} \rceil \cdot 3 = 13 \stackrel{?}{\leq} 7 \text{ **NO**}$$

$$R'_2 = 7 + \lceil \frac{13}{6} \rceil \cdot 3 = 16 \stackrel{?}{\leq} 13 \text{ **NO**}$$

$$R''_2 = 7 + \lceil \frac{16}{6} \rceil \cdot 3 = 16 \stackrel{?}{\leq} 16 \text{ **SI} \stackrel{?}{\leq} 28 \text{ **OK**}**$$

- il task  $\tau_3$  è **schedulabile**

$$R_3 = C_3 + \lceil \frac{R_3}{T_1} \rceil \cdot C_1 + \lceil \frac{R_3}{T_2} \rceil \cdot C_2$$

$$= 5 + \lceil \frac{5}{6} \rceil \cdot 3 + \lceil \frac{5}{28} \rceil \cdot 7 = 15 \stackrel{?}{\leq} 5 \text{ **NO**}$$

$$R'_3 = 5 + \lceil \frac{15}{6} \rceil \cdot 3 + \lceil \frac{15}{28} \rceil \cdot 7 = 21 \stackrel{?}{\leq} 15 \text{ **NO**}$$

$$R''_3 = 5 + \lceil \frac{21}{6} \rceil \cdot 3 + \lceil \frac{21}{28} \rceil \cdot 7 = 24 \stackrel{?}{\leq} 21 \text{ **NO**}$$

$$R'''_3 = 5 + \lceil \frac{24}{6} \rceil \cdot 3 + \lceil \frac{24}{28} \rceil \cdot 7 = 24 \stackrel{?}{\leq} 24 \text{ **SI} \stackrel{?}{\leq} 28 \text{ **OK**}**$$

Possiamo quindi affermare che l'intero *task set*  $\mathcal{T}$  è **schedulabile**, ed ha una complessità pari a:  $\mathcal{O}(n \cdot D_{max})$ .

## 4.6 Processor Demand Criterion

In ogni intervallo di tempo la *computation demanded* dal *task set* non deve essere maggiore del tempo disponibile. Nel caso di **EDF** abbiamo priorità dinamiche ed è quindi impossibile utilizzare il *response time analysis* per dire se un *task set* è schedulabile.

La **demand** in  $[t_1, t_2]$  è il *WCET* di quei task che hanno  $r_i \geq t_1 \cap d_i \leq t_2$ :

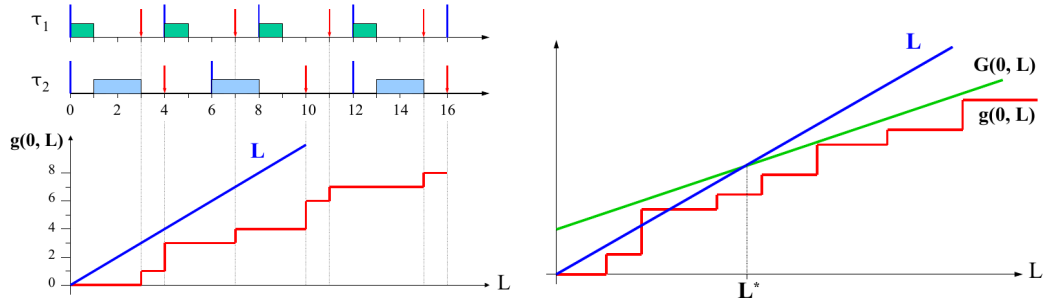
$$g(t_1, t_2) = \sum_{\substack{d_i \leq t_2 \\ r_i \geq t_1}} C_i$$

Consideriamo ora  $t_1 = 0$  (ovvero il *critical instance*) e  $t_2 = \mathcal{L}$  e andiamo a calcolarci la **demand** nell'intervallo di tempo  $[0, \mathcal{L}]$  allora avremo che:

$$g(0, \mathcal{L}) = \sum_{\substack{d_i \leq \mathcal{L} \\ r_i \geq 0}} C_i = \sum_{i=1}^n \underbrace{\left\lfloor \frac{\mathcal{L} - D_i + T_i}{T_i} \right\rfloor}_{\text{Quando i task hanno } r_i \text{ e } T_i \text{ all'interno di } \mathcal{L}} \cdot C_i$$

L'*execution time demanded* dai singoli job che hanno *deadline*  $d_i \leq \mathcal{L}$  su un intervallo di lunghezza  $\mathcal{L}$  non può essere maggiore di  $\mathcal{L}$  ovvero

$$\forall \mathcal{L} > 0 \quad g(0, \mathcal{L}) \leq \mathcal{L} \quad \leftarrow \text{è inapplicabile } \mathcal{L} \rightarrow \infty$$



Controllare la schedulabilità utilizzando la funzione  $g(0, \mathcal{L})$  che è continua nell'intervallo  $[0, \mathcal{L}]$  **non** è **fattibile**. Possiamo però rendere la funzione una *staircase function* con dei salti (discontinuità) nei punti di *deadline* assoluta  $d_{i,k} = (k-1) \cdot T_i + D_i$ , perché possiamo notare dalla figura che i punti in cui la *demand* rischia di superare  $\mathcal{L}$  sono nei punti di **discontinuità**. Possiamo identificare un **hyperperiod** **H** dopo il quale il comportamento della nostra funzione discontinua si ripeterà,  $\mathbf{H} = \text{lcm}(T_i, T_j) \quad \forall T_i, T_j \in \mathcal{T} \text{ con } i \neq j$  ovvero il minimo comune multiplo tra i periodi del *task set*.

$$g(0, \mathcal{L}) = \sum_{i=1}^n \lfloor \frac{\mathcal{L} + T_i - D_i}{T_i} \rfloor \cdot C_i$$

$$G(0, \mathcal{L}) = \sum_{i=1}^n \left( \frac{\mathcal{L} + T_i - D_i}{T_i} \right) \cdot C_i$$

Per definizione del *floor* avremo che  $g(0, \mathcal{L}) \leq G(0, \mathcal{L})$ .

$$\begin{aligned}
G(0, \mathcal{L}) &= \sum_{i=1}^n \left( \frac{\mathcal{L} + T_i - D_i}{T_i} \right) \cdot C_i \\
&= \sum_{i=1}^n \frac{\mathcal{L}}{T_i} \cdot C_i + \left( \frac{T_i - D_i}{T_i} \right) \cdot C_i \\
&= \sum_{i=1}^n \mathcal{L} \cdot \underbrace{\frac{C_i}{T_i}}_U + (T_i - D_i) \cdot \underbrace{\frac{C_i}{T_i}}_{U_i} \\
&= \mathcal{L}U \cdot \sum_{i=1}^n (T_i - D_i) \cdot U_i
\end{aligned} \tag{4.1}$$

Ora vogliamo trovare il punto di intersezione  $\mathcal{L}^*$  tra  $\mathcal{L}$  e  $G(0, \mathcal{L})$ , poniamo quindi:

$$\begin{aligned}
\mathcal{L} &= G(0, \mathcal{L}) \\
\mathcal{L} &= \mathcal{L}U \cdot \sum_{i=1}^n (T_i - D_i) \cdot U_i \\
\mathcal{L} - \mathcal{L}U &= \sum_{i=1}^n (T_i - D_i) \cdot U_i \\
\mathcal{L} \cdot (1 - U) &= \sum_{i=1}^n (T_i - D_i) \cdot U_i \\
\mathcal{L}^* &= \frac{\sum_{i=1}^n (T_i - D_i) \cdot U_i}{1 - U}
\end{aligned} \tag{4.2}$$

Per verificare quindi se il *task set* è schedulabile dovremo valutare se  $\forall \mathcal{L} \in D, g(0, \mathcal{L}) \leq \mathcal{L}$  dove avremo che:

$$D = \{d_k | d_k \leq \min(\mathbf{H}, \mathcal{L}^*)\} \left\{ \begin{array}{l} \mathbf{H} = lcm(T_i, T_j) \forall T_i, T_j \in \mathcal{T}, i \neq j \\ \mathcal{L}^* = \frac{\sum_{i=1}^n (T_i - D_i) \cdot U_i}{1 - U} \end{array} \right.$$



**Esercizio**

Consideriamo il seguente *task set*:

$$\begin{cases} \tau_1(1, 2, 3) \\ \tau_2(2, 5.5, 7) \\ \tau_3(2, 6, 10) \end{cases}$$

Identifichiamo come prima cosa la nostra  $D$ :

$$D = \{d_k | d_k \leq \min(\mathbf{H}, \mathcal{L}^*)\} \begin{cases} \mathbf{H} = lcm(T_1, T_2, T_3) = lcm(3, 7, 10) = 210 \\ \mathcal{L}^* = \frac{\sum_{i=1}^n (T_i - D_i) \cdot U_i}{1 - U} \\ = \frac{[(3-2) \cdot \frac{1}{3}] + [(7-5.5) \cdot \frac{2}{7}] + [(10-6) \cdot \frac{2}{10}]}{1 - \underbrace{(\frac{1}{3} + \frac{2}{7} + \frac{2}{10})}_{U_{lub} = 0.8190}} \\ = 8.63 \end{cases}$$

- consideriamo  $\mathcal{L} = 2$

$$\lfloor \frac{\mathcal{L} + T_1 - D_1}{T_1} \rfloor \cdot C_1 = \lfloor \frac{2+3-2}{3} \rfloor \cdot 1 = 1 \stackrel{?}{\leq} 2 \quad \leftarrow \text{OK}$$

- consideriamo  $\mathcal{L} = 5$

- consideriamo  $\mathcal{L} = 5.5$

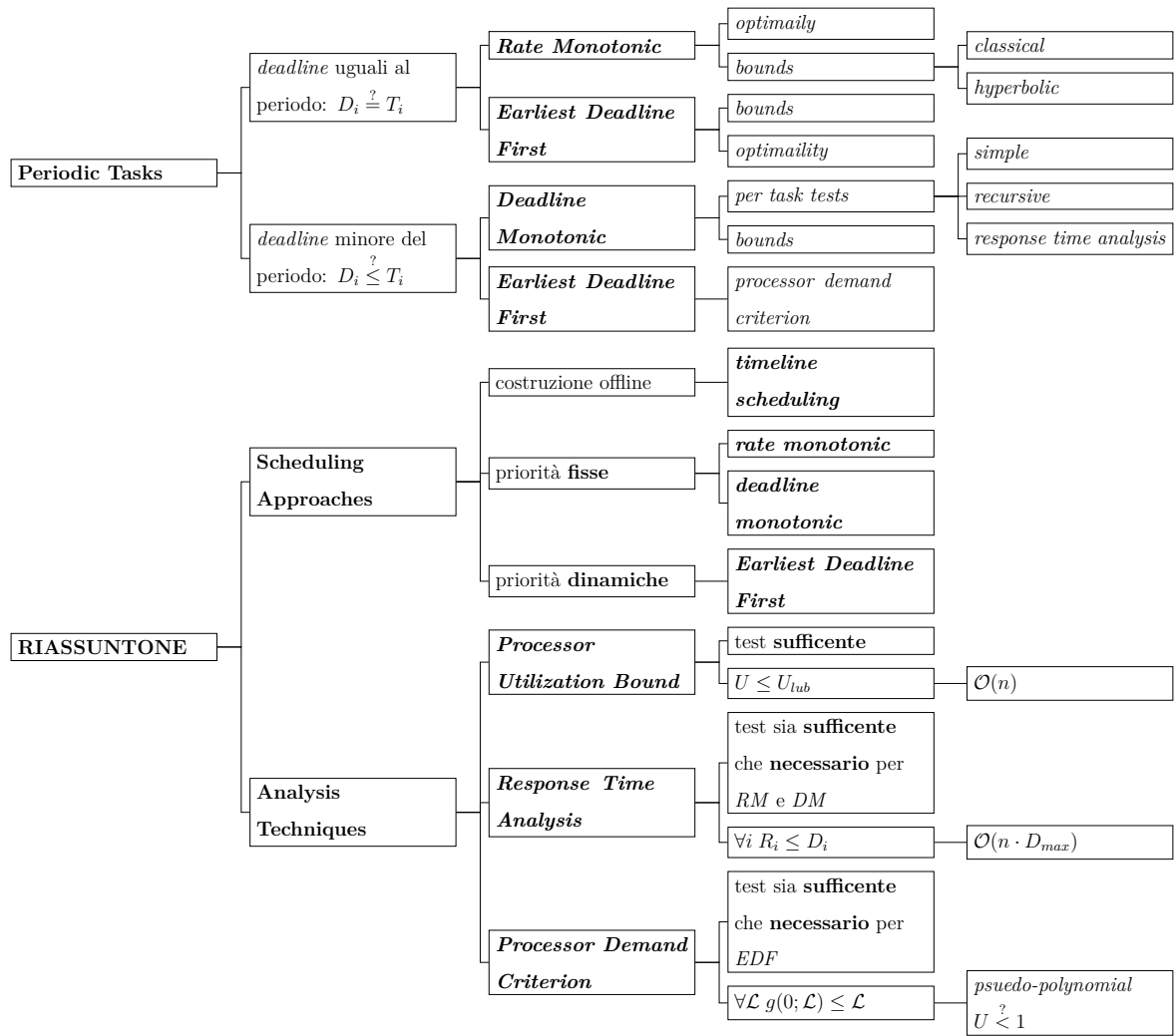
$$\begin{aligned} g(0, \mathcal{L}) &= \lfloor \frac{\mathcal{L} + T_1 - D_1}{T_1} \rfloor \cdot C_1 + \lfloor \frac{\mathcal{L} + T_2 - D_2}{T_2} \rfloor \cdot C_2 \\ &= \lfloor \frac{5.5 + 3 - 2}{3} \rfloor \cdot 1 + \lfloor \frac{5.5 + 7 - 5.5}{7} \rfloor \cdot 2 \\ &= 2 + 2 = 4 \stackrel{?}{\leq} 5.5 \quad \leftarrow \text{OK} \end{aligned}$$

- consideriamo  $\mathcal{L} = 6$

$$\begin{aligned} g(0, \mathcal{L}) &= \lfloor \frac{\mathcal{L} + T_1 - D_1}{T_1} \rfloor \cdot C_1 + \lfloor \frac{\mathcal{L} + T_2 - D_2}{T_2} \rfloor \cdot C_2 + \lfloor \frac{\mathcal{L} + T_3 - D_3}{T_3} \rfloor \cdot C_3 \\ &= \lfloor \frac{6 + 3 - 2}{3} \rfloor \cdot 1 + \lfloor \frac{6 + 7 - 5.5}{7} \rfloor \cdot 2 + \lfloor \frac{6 + 10 - 6}{10} \rfloor \cdot 2 \\ &= 2 + 2 + 2 = 6 \stackrel{?}{\leq} 6 \quad \leftarrow \text{OK} \end{aligned}$$

- consideriamo  $\mathcal{L} = 8$

$$\begin{aligned} g(0, \mathcal{L}) &= \lfloor \frac{\mathcal{L} + T_1 - D_1}{T_1} \rfloor \cdot C_1 + \lfloor \frac{\mathcal{L} + T_2 - D_2}{T_2} \rfloor \cdot C_2 + \lfloor \frac{\mathcal{L} + T_3 - D_3}{T_3} \rfloor \cdot C_3 \\ &= \lfloor \frac{8 + 3 - 2}{3} \rfloor \cdot 1 + \lfloor \frac{8 + 7 - 5.5}{7} \rfloor \cdot 2 + \lfloor \frac{8 + 10 - 6}{10} \rfloor \cdot 2 \\ &= 3 + 2 + 2 = 7 \stackrel{?}{\leq} 8 \quad \leftarrow \text{OK} \end{aligned}$$



## Capitolo 5

# Scheduling con Shared Memory