

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

---

# Real Time Embedded System

---

Anno Accademico 2023/24

# Indice

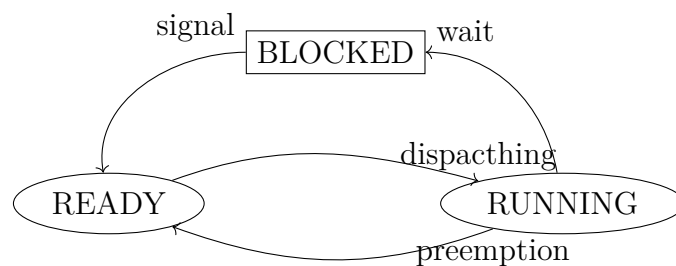
<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Non Real-time scheduling algorithms</b>	<b>6</b>
<b>3</b>	<b>Real-time scheduling algorithms</b>	<b>11</b>

# Capitolo 1

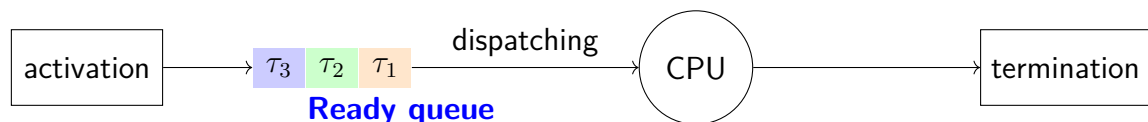
## Introduzione

**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*. La strategia con cui vengono scelti i task dalla coda per essere eseguiti sulla *CPU* sono gli **scheduling algorithms**.

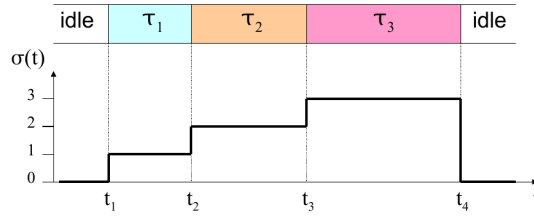


**Scheduling** può essere definito **preemptive** ovvero se il task in esecuzione in un certo istante di tempo  $t_i$  può essere temporaneamente sospeso per eseguire un task con importanza maggiore, mentre si dice **non-preemptive** se il task in esecuzione non può essere sospeso finché non viene completato.

**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**.

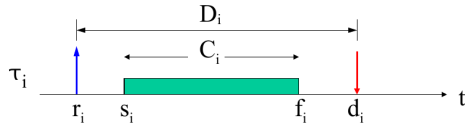


Figura 1.1: Real-time tasks

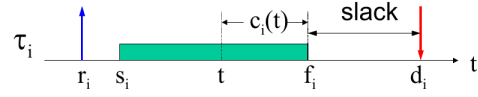


Figura 1.2: Real-time tasks

- $r_i$  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.

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

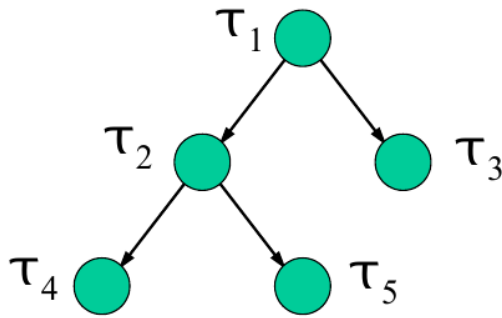
2. **event driven**: anche noti come **tasks aperiodici**, ovvero il task viene attivato all'arrivo di un evento o per un'invocazione esplicita della sua primitiva di invocazione. A loro volta possono dividersi in:

- **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*.

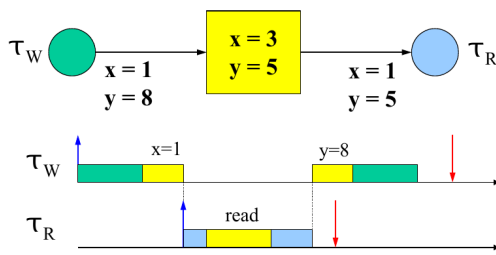


Figura 1.3: *no mutual exclusion*

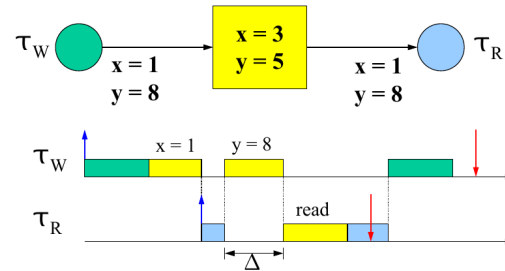


Figura 1.4: *mutual exclusion*

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:

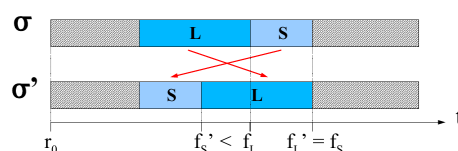
- processore singolo
- *fully preemptive tasks*
- attivazione simultanea
- nessun vincolo di precedenza
- nessun vincolo sulle risorse

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

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.

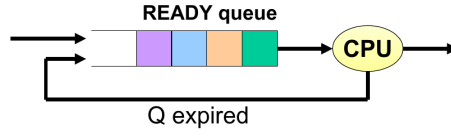
- può avere problemi di **starvation** infatti dei task a bassa priorità possono accumulare ritardo per via della *preemption* dovuto a task con più alta priorità.
- Una possibile **soluzione** è quella dell'**aging** ovvero che la priorità viene incrementata con il passare del tempo:

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

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

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*.



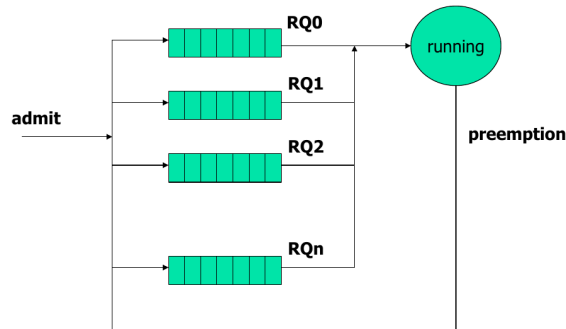
Introduce il concetto di **time sharing**: ovvero che ogni task esegue in solitaria su un “processore virtuale”  $n$  volte più lento rispetto a quello reale.

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

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