Università degli studi di Modena e Reggio Emilia

Dipartimento di Ingegneria Enzo Ferrari

Real Time Embedded System

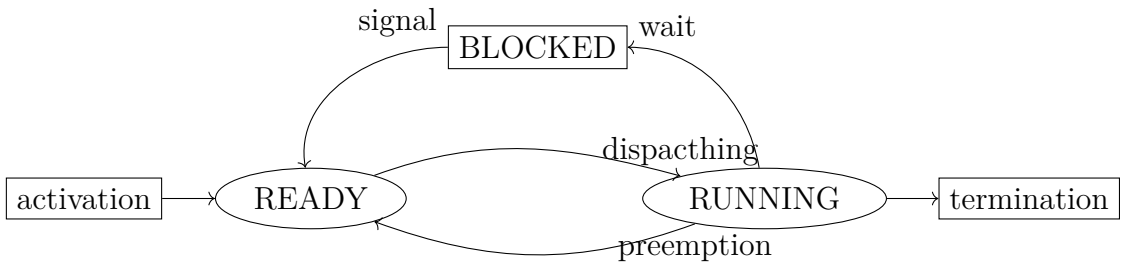
Anno Accademico 2023/24

Indice

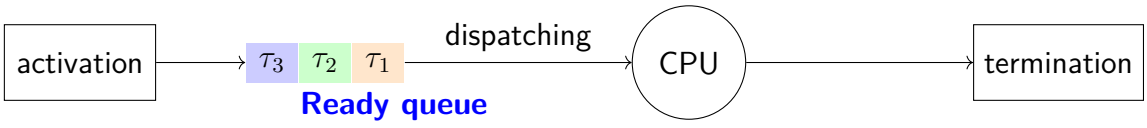
Capitolo 1

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***.

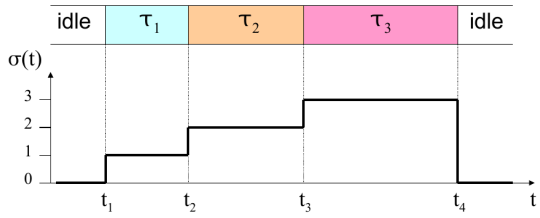


Scheduling può essere definito ***preemptive*** ovvero se il task in esecuzione in un certo istante di tempo t_i può essere temporaneamente sospeso per eseguire un task con importanza maggiore, mentre si dice ***non-preemptive*** se il task in esecuzione non può essere sospeso finché non viene completato.

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

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

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***.

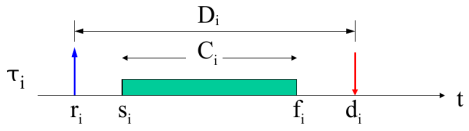


Figura 1.1: Real-time tasks

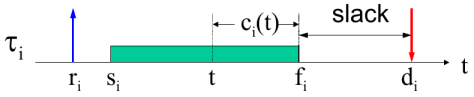


Figura 1.2: Real-time tasks

- $\mathbf{r}_i$ è il ***request time***.
- $\mathbf{s}_i$ è lo ***start time*** ovvero il tempo in cui il task inizia l'esecuzione.
- $\mathbf{C}_i$ è il tempo di esecuzione in caso peggiore (***WCET***).
- $\mathbf{d}_i$ è la ***deadline*** **assoluta**, mentre $\mathbf{D}_i$ è la ***deadline*** **relativa**.
- $\mathbf{f}_i$ 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 ≤ 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 $\mathbf{T_i}$ è il periodo a cui quel task viene invocato.

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

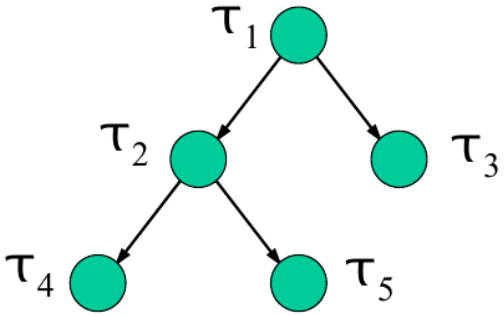
2. ***event driven***: anche noti come **tasks aperiodici**, ovvero il task viene attivato all'arrivo di un evento o per un'invocazione esplicita della sua primitiva di invocazione. A loro volta possono dividersi in:
   - ***aperiodic***: $r_{i,k+1} > r_{i,k}$
   - ***sporadic***: $r_{i,k+1} \geq r_{i,k} + T_i$

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

- ***timinig constraints***: ovvero dei vincoli sul tempo di esecuzione [*deadline*, *activation*, *completition* 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*.

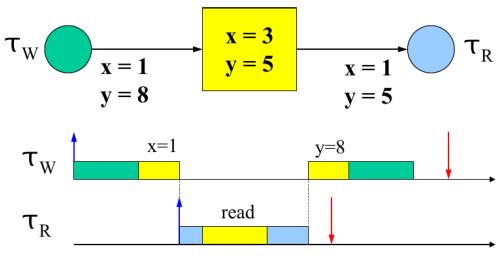


Figura 1.3: *no mutual exclusion*

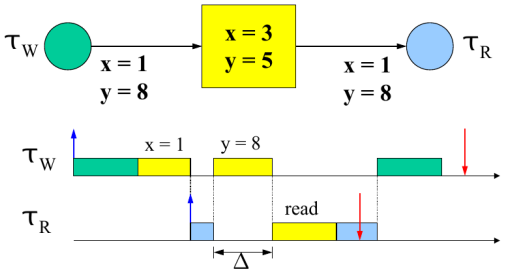


Figura 1.4: *mutual exclusion*

Mentre si analizza un *tasks set* e si cerca che il tempo di esecuzion 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'apporccio 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 **predicibitilà** (*real-time system*) o la ***fairness***.
Tra i processi possiamo differenziare anche il tipo di risorsa che viene utilizzanta: ***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* σ 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, ricordurci ad un algoritmo *polynomial time*.
Per il ora consideriamo:

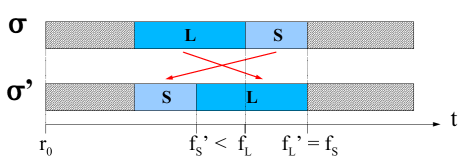
- processore singolo
- *fully preemptive tasks*
- attivazione simultanea
- nessun vincolo di precedenza
- nessun vincolo sulle risorse

Algorithm taxonomy	
preemptive	***non-preemptive***
off line: tutte le decisioni sullo scheduling vengono prese prima dell'attivazione dei task, normalmente lo *schedule* viene salvato in una tabella (***table-driven scheduling***)	***on line***: le decisioni di scheduling vengono prese *runtime* sul set dei tasks attivi
static: le decisioni di scheduling vengono prese basandosi su parametri fissati, staticamente assegnati al task prima dell'attivazione	***dynamic***: le decisioni di scheduling vengono prese su parametri che possono variare nel tempo
best effort: trova sempre uno *schedule* fattibile, se **esiste**	***optimal***: fa del suo meglio per trovare uno *schedule* 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 $\sigma \neq$ SJF e un'altro scheduler σ' 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* σ schedula il task L prima (non conforme con SJF), mentre σ' 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 σ' infatti avremo $f'_S < f_L$.
Avremo perciò $f'_S + f'_L \leq f_S + f_L$

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

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

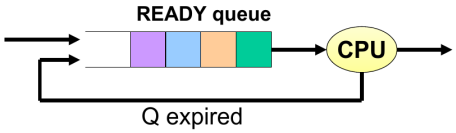
3. ***Priority Scheduling***: ad ogni task viene assegnata una **priorità**, il task con la priorità maggiore viene eseguito come primo, mentre per i task con pari priorità siene eseguito uno scheduler o FCFS o RR. Può essere utilizzato per fini *real-time* se le priorità sono assegnate seguendo specifiche 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 τ_i non può eseguire più un **Q** unità

di tempo. Quando **Q** scade, τ_i viene riaccodato nella *ready queue*.



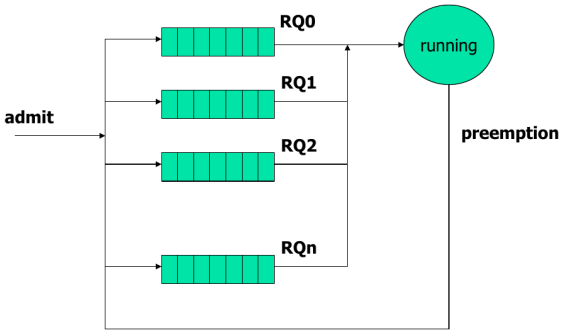
Introduce il concetto di ***time sharing***: ovvero che ogni task esegue in solitaria su un "processore virtuale" n volte più lento rispetto a quello reale.

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

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 δ allora avremo che:

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

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 Earlies Due Date

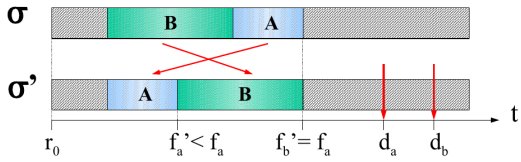
Esegue come primo task quello con la *deadline* **relativa** più imminente.

- tutti i taks arrivano simulataneamente.
- 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 $\sigma \neq$ EDD e un'altro scheduler σ' 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 σ ($\neq$ EDD) viene schedulato prima B e dopo A in quanto $d_a < d_b$ mentre nel caso dello scheduler σ' ($\simeq$

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 $f'_a < f_a$. Possimao 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 σ' 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 $\sigma \in \{\sigma', \sigma'', ..., \sigma*\}$. Iterando il ragionamento del confronto fatto per uno scheduler σ e σ' con tutto l'insieme degli scheduler, in questo modo avremo che: $L_{max}(\sigma') \geq L_{max}(\sigma'') \geq ... \geq L_{max}(\sigma*)$ 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.

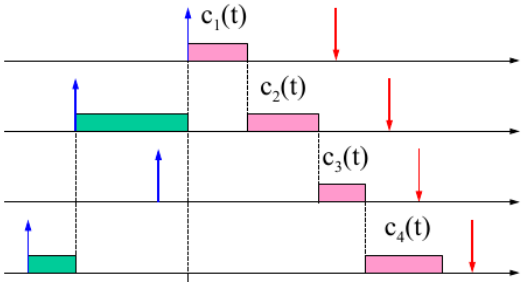
Complessità:

- 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 τ_i deve essere eseguito.

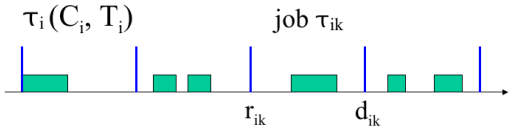


Figura 4.1: *periodic task*

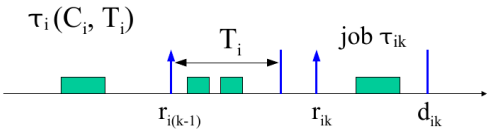


Figura 4.2: *sporadic task*

Per ogni task periodico, bisogna garantire che:

- ogni job τ_{ik} venga attivato in $r_{ik} = (k-1) \cdot T_i$.
- ogni job τ_{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 *deelay* minimo di attivazione tra un task τ_{ik} e un task $\tau_{i(k+1)}$, infatti bisogna garantire per ogni task sporadico che:

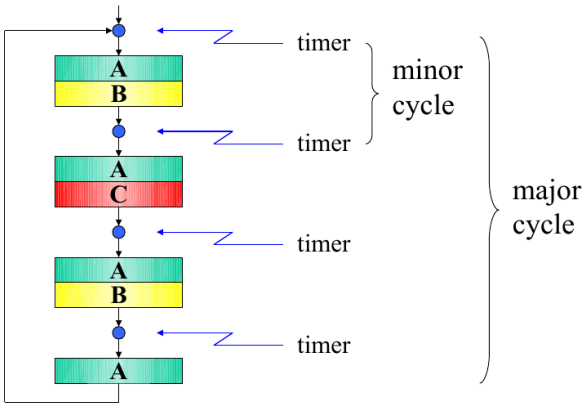
- ogni job τ_{ik} viene attivato in un istante $r_{ik} \geq r_{i(k+1)} + T_i$.
- ogni job τ_{ik} completi la sua esecuzione entro una *deadline* relativa $d_{ik} = r_{ik} + D_{ik}$

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 $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_k\}$ 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 $\mathbf{T} = lcm(T_i, T_j)\ \forall T_i, T_j \in \mathcal{T},\ i \neq j$
  - nel caso in cui ci siano task sporadici come andiamo a valorizzare il *major cycle*



Vantaggi	**Svantaggi**
- implementazione semplice (non viene richiesto alcun sistema operativo *real-time*)	- non è robusto contro *overload* del sistema
- ogni procedura condivide un *address space* comune	- nel caso di aggiunta di un nuovo task è molto difficile l'espansione dello *scheduler*
- basso *overhead* a tempo di esecuzione	- non è facile gestire task aperiodici
- permette di controllare i *jitter*	- 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 si)

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 al somma dei *WCET* dei task in esecuzione nel Δ è maggiore del Δ, allora sarà necessario dividere uno degli n task in quel Δ di tempo in modo da evitare un *timeline break*.

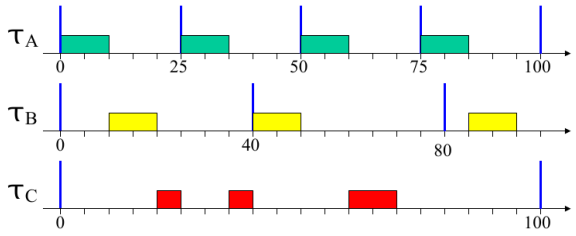
task	T	T'
A	25 ms	25 ms
B	50 ms	40 ms
C	100 ms	100 ms
minor cycle	$\Delta = 25$ms	$\Delta = 5$ms
major cycle	$\mathbf{T} = 100$ms	$\mathbf{T} = 200$ms

4.2 Priority Scheduling

Ad ogni task viene assegnata una priorità basata sui sui 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'esecuzioni 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:

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

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:

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

Definiamo il ***Critical Instant*** come l'istante di tempo in cui è presente il *response time* maggiore, è stato dimostrato che equivale, considernado 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:

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

4.3 Earliest Deafline First