

Appunti sistemi embedded  
Scheduling in sistemi real-time

2025

# Contents

<b>1</b>	<b>Introduzione</b>	<b>4</b>
<b>2</b>	<b>Real time</b>	<b>5</b>
2.1	Limiti dei sistemi real-time correnti . . . . .	5
2.2	Feature desiderabili di un sistema real-time . . . . .	6
<b>3</b>	<b>Fonti di imprevedibilità</b>	<b>8</b>
3.1	DMA . . . . .	8
3.2	Cache . . . . .	8
3.3	Interrupts . . . . .	9
3.4	Semafori . . . . .	10
3.5	Gestione della memoria . . . . .	10
3.6	Linguaggi di programmazione . . . . .	10
<b>4</b>	<b>Concetti di base di scheduling</b>	<b>12</b>
4.1	Introduzione . . . . .	12
4.2	Tipi di vincoli di scheduling . . . . .	13
4.3	Classificazione di algoritmi di scheduling . . . . .	17
<b>5</b>	<b>Scheduling di task aperiodici</b>	<b>20</b>
5.1	Garanzie . . . . .	20
5.2	Earliest Deadline First (EDF) . . . . .	21
5.3	EDF con vincoli di precedenza . . . . .	21
<b>6</b>	<b>Scheduling di task periodici</b>	<b>22</b>
6.1	Fattore di utilizzo del processore . . . . .	23
6.2	Timeline scheduling . . . . .	23
6.3	Rate Monotonic Scheduling (RMS) . . . . .	24
6.4	RM vs EDF . . . . .	26
<b>7</b>	<b>Server a priorità fissa</b>	<b>27</b>
7.1	Background scheduling . . . . .	27
7.2	Polling server . . . . .	28
7.3	Deferrable server . . . . .	29
7.4	Priority exchange . . . . .	29
7.5	Sporadic server . . . . .	29
7.6	Slack stealing . . . . .	30
7.7	Summary . . . . .	30

---

<b>8</b>	<b>Server a priorità dinamica</b>	<b>31</b>
8.1	Dynamic priority exchange server . . . . .	31
8.2	Dynamic sporadic server . . . . .	32
8.3	Total bandwidth server . . . . .	32
8.4	Earliest deadline late server . . . . .	32
8.5	Improved priority exchange server . . . . .	32
8.6	Constant bandwidth server . . . . .	33
<b>9</b>	<b>Protocolli di accesso alle risorse condivise</b>	<b>35</b>
9.1	Inversione di priorità . . . . .	36
9.2	Terminologia e assunzioni . . . . .	37
9.3	Non-preemptive protocol (NPP) . . . . .	38
9.4	Highest locker priority (HLP) . . . . .	38
9.5	Priority inheritance protocol (PIP) . . . . .	38
9.5.1	Blocco a catena . . . . .	39
9.5.2	Deadlock . . . . .	40
9.6	Priority ceiling protocol (PCP) . . . . .	41
9.7	Stack resource policy (SRP) . . . . .	41
9.7.1	Definizione . . . . .	43

# Chapter 1

## Introduzione

I sistemi real time sono sistemi di computazione dove la reazione a eventi dell'ambiente deve avvenire entro precisi requisiti temporali. Dunque la correttezza della computazione non dipende unicamente dal risultato prodotto, ma anche dal tempo in cui il risultato è prodotto. Una reazione ritardata potrebbe essere inutile o addirittura dannosa. Molto spesso questi sistemi vengono implementati attraverso la programmazione in assembly o con driver a basso livello per la gestione delle periferiche, manipolazione delle task e la priorità degli interrupt.

Questo approccio presenta degli svantaggi

- Programmazione tediosa
- Codice poco comprensibile
- Difficile manutenibilità
- Difficile verifica dei vincoli temporali

La conseguenza maggior di questo approccio è che il software costruito tramite tecniche empiriche può essere imprevedibile.

Se tutti i vincoli temporali critici non possono essere verificati a priori e il sistema operativo non include meccanismi specifici per maneggiare i task real-time il sistema potrebbe apparentemente funzionare correttamente per un certo periodo di tempo, ma potrebbe fallire in situazioni particolari. Il testing, pur essendo importante, non può essere una soluzione per ottenere la predicibilità in un sistema real-time. Questo vale perché il flusso di controllo dipende anche da eventi esterni dell'ambiente, che non possono essere interamente inclusi nel testing.

# Chapter 2

## Real time

La correttezza del sistema non dipende solamente dal risultato logico prodotto dalla computazione ma anche dal tempo in cui il risultato è prodotto. Il termine reale indica che la reazione del sistema a eventi esterni deve avvenire **durante** la loro evoluzione. Di conseguenza il tempo interno al sistema deve essere misurato usando la stessa scala utilizzata per misurare il tempo dell'ambiente controllato. Al livello di processo la principale differenza tra un task real-time e non è la presenza di una **deadline**, ovvero il tempo massimo in cui una task deve essere completata. In un sistema critico una task che non rispetta la deadline non solo è in ritardo, è anche un errore! In base alle conseguenze a cui porta il non rispettare la deadline le task possono essere suddivise in:

- **Hard** se il non rispetto della deadline porta a conseguenze catastrofiche, ad esempio il fallimento di un sistema di controllo di un aereo.
- **Firm** se il non rispetto della deadline porta a un output inutile al sistema ma non produce conseguenze catastrofiche.
- **Soft** se il non rispetto della deadline porta a una degradazione della qualità del servizio ma non produce conseguenze catastrofiche.

Un sistema in grado di gestire task hard real-time è detto **hard real-time system**. Alcuni di questi sistemi sono tendenzialmente sistemi **safety critical**.

### 2.1 Limiti dei sistemi real-time correnti

Molti dei sistemi attualmente in uso sono soluzioni basate su kernel, che sono versioni modificate di sistemi operativi a time sharing. Una conseguenza di questo è che condividono le stesse feature di base dei sistemi a time sharing, che non sono adatte a sistemi real-time. Le caratteristiche principali sono:

- **Multitasking**  
Supporto per la programmazione concorrente offerta attraverso un insieme di chiamate a sistema per la gestione dei processi. Molte di queste primitive non prendono in considerazione il tempo e, peggio ancora, introducono ritardi senza limiti ai tempi di esecuzione, il che potrebbe portare le task hard a non rispettare le deadline in maniera non predicibile.

- **Scheduling basato su priorità**

Meccanismo flessibile di scheduling, dato che permette di implementare diverse strategie in base alla maniera in cui assegno le priorità. Nonostante ciò, mappare i constraint a un insieme di priorità è difficile e non sempre possibile, specialmente in un sistema dinamico. Il problema principale è che i kernel permettono un **numero limitato di livelli di priorità**, mentre le deadline possono variare molto di più. In sistemi dinamici l'arrivo di una nuova task può comportare un remappaggio dell'intero insieme di priorità.

- **Capacità di rispondere velocemente agli interrupt**

Solitamente questa feature viene ottenuta tramite la possibilità di impostare una priorità per gli interrupt, più alta rispetto alla priorità dei processi e riducendo al minimo le porzioni di codice eseguite con le interruzioni disabilitate.

Nota che questo approccio aumenta la reattività del sistema agli eventi esterni, ma introduce dei ritardi non limitati all'esecuzione dei processi. Infatti, un processo di un'applicazione sarà sempre interrotto da un interrupt di un driver, anche se è più importante che venga servita la periferica.

- **Meccanismo di base per la comunicazione tra processi e sincronizzazione**

Tipicamente semafori binari per la sincronizzazione e la mutua esclusione su risorse condivise. Però, se nessun protocollo d'accesso è utilizzato per entrare in sezioni critiche i semafori possono generare una serie di problemi, ad esempio la **priority inversion** o deadlock.

- **Small kernel e context switch veloce**

Questa feature riduce l'overhead del sistema, dunque migliora il response time medio dell'insieme di task. Ciò nonostante non è sufficiente per garantire il rispetto delle deadline. D'altro canto un kernel piccolo implica limitate funzionalità, il che ha effetto sulla predicibilità del sistema.

- **Supporto a clock real-time come reference di tempo interna**

Feature essenziale di ogni kernel real-time che gestisce task hard real-time che interagiscono con l'ambiente esterno. Nella maggior parte dei kernel commerciali questo è l'unico meccanismo di temporizzazione disponibile. In molti casi non vi sono primitive per controllare esplicitamente i constraint temporali delle task, e non c'è un supporto per l'attivazione automatica delle task periodiche.

La mancanza di una qualsiasi garanzia preclude l'utilizzo di questi sistemi in casi in cui i requisiti temporali siano stringenti.

## 2.2 Feature desiderabili di un sistema real-time

La predicibilità può essere ottenuta apportando pesanti modifiche ai tipici sistemi a time sharing. Alcune proprietà importanti sono:

- **Timeliness**

I risultati non devono essere corretti solo nel loro valore ma anche nel dominio temporale. Di conseguenza il sistema operativo deve fornire degli specifici meccanismi del kernel per la gestione del tempo e per la gestione dei task con constraint temporali espliciti e criticità diverse.

- **Predicibilità**

Per raggiungere il livello di performance desiderato, il sistema deve essere analizzabile per prevedere le conseguenze di ogni decisione di scheduling. Nelle applicazioni safety critical tutti i requisiti temporali devono esser garantiti a priori, prima di mettere il sistema in azione. Se qualche vincolo non può essere garantito, il sistema deve essere in grado di informare l'utente e delle decisioni alternative devono essere prese.

- **Efficienza**

La maggior parte dei sistemi real-time sono embedded, dunque l'efficienza è un requisito fondamentale.

- **Robustezza**

Il sistema deve essere in grado di continuare a funzionare a qualsiasi condizioni di carico. La gestione dell'overload e il comportamento di adattamento sono feature essenziali per sistemi con molta varianza di carico.

- **Tolleranza agli errori**

Singoli errori hardware o software non devono portare a un fallimento del sistema.

- **Manutenibilità**

L'architettura di un sistema real-time deve essere progettata in maniera modulare per permettere di eseguire modifiche facilmente.

# Chapter 3

## Fonti di imprevedibilità

Un sistema dovrebbe essere in grado di prevedere l'evoluzione delle task e garantire in anticipo che tutti i vincoli temporali critici vengano rispettati. L'affidabilità della garanzia dipende da un insieme di fattori, che includono anche le caratteristiche fisiche dell'hardware e i meccanismi/politiche del kernel.

La prima componente che influisce sulla predicibilità è il processore. Le caratteristiche del processore, ad esempio il prefetch delle istruzioni, pipelining, cache, DMA sono fonti di **non determinismo**.

Pur aumentando le performance medie del sistema, introducono fattori di non determinismo che prevengono una precisa stima dei worst-case-execution-time (WCETs). Altre caratteristiche che influenzano la predicibilità sono le caratteristiche interne del kernel, come l'algoritmo di scheduling, i meccanismi di sincronizzazione, i tipi di semafori, le politiche di gestione della memoria e le politiche di gestione degli interrupt.

### 3.1 DMA

Uno dei modi più comuni per implementarlo è tramite cycle stealing, il DMA controller prende il controllo del bus per un ciclo di clock e poi lo rilascia al processore. Durante l'operazione di DMA il trasferimento e l'esecuzione del codice sul processore avvengono in parallelo. Se la CPU e il DMA controller richiedono un ciclo di memoria allo stesso tempo, il bus viene assegnato al DMA controller e la CPU aspetta finché il DMA non ha finito. In questo modo non è possibile prevedere il tempo di esecuzione di un task, dato che il tempo di esecuzione dipende da quante volte il DMA controller ha bisogno del bus. Una possibile soluzione al problema è il time slicing, dove un ciclo di memoria è diviso in due slot di tempo adiacenti, uno per il DMA e uno per la CPU. Questa soluzione è più costosa ma più prevedibile di cycle stealing, dato che il response time di una task non è influenzato dal DMA.

### 3.2 Cache

Nei sistemi real-time la cache è una fonte di non determinismo, dato che il tempo di accesso alla memoria dipende dalla presenza o meno del dato in cache. Quando avviene un cache miss il tempo di accesso alla memoria è molto più lungo, e questo non è prevedibile. Quando si effettua una scrittura in memoria l'utilizzo della cache risulta costoso perché il tempo di accesso alla memoria è più lungo, dato che deve essere garantita la consistenza.



### 3.3 Interrupts

Un grosso problema per la predicibilità è la gestione degli interrupt provenienti dalle periferiche I/O. Se non gestite correttamente possono introdurre ritardi non limitati durante l'esecuzione di processi.

In quasi tutti i sistemi operativi la gestione di un interrupt prevede l'esecuzione di una service routine dedicata per la gestione del device. Il vantaggio di questo metodo è che tutti i dettagli hardware sono contenuti nel driver.

In molti OS gli interrupt vengono serviti usando un sistema a priorità fissa, tramite i quali ogni driver viene schedulato in base a una priorità statica, questa priorità è più alta rispetto a quella di qualsiasi processo. In generale è molto difficile prevedere a priori il numero di interrupt che un task può ricevere, dunque il delay è imprevedibile.

Per ridurre l'interferenza dei driver sui task ed effettuare operazioni di I/O le periferiche devono essere gestite diversamente

#### Approccio A

La soluzione più radicale è disattivare tutti gli interrupt esterni, a eccezione di quelli generati dal timer. Tutte le periferiche vanno gestite dall'applicazione (controllo di programma), i quali hanno accesso ai registri delle periferiche. Dato che non ci sono più interrupt, il trasferimento deve essere effettuato tramite polling. L'accesso diretto alle periferiche consente alta flessibilità al livello di programma ed elimina i delay introdotti dai driver. Dunque il tempo necessario per un trasferimento può essere calcolato a priori. Un altro vantaggio è che il kernel non va modificato quando una nuova periferica viene aggiunta o ne viene rimossa una.

Lo svantaggio principale è che risulta poco efficiente dato la busy wait necessaria per il polling. Un grosso problema è che le applicazioni richiedono la conoscenza completa dei dettagli a basso livello delle periferiche che devono gestire. Questo può essere gestito incapsulando le routine che dipendono dai device in una libreria.

#### Approccio B

Come nella soluzione precedente tutti gli interrupt esterni sono disabilitati, tranne quelli dei timer.

In questa soluzione le periferiche non vengono gestite direttamente dall'applicazione, ma a turno da routine dedicate del kernel periodicamente attivate dal timer. Questo approccio elimina i ritardi causati dai driver e confina le operazioni di I/O a una o più routine periodiche del kernel. In alcuni sistemi i device I/O vengono suddivisi in due gruppi in base alla loro velocità, quelli più veloci vengono gestiti da task periodici con frequenza maggiore.

Il vantaggio di questo metodo è che tutti i dettagli hardware sono incapsulati nel kernel, dunque l'applicazione non ha bisogno di conoscere i dettagli a basso livello delle periferiche. Dato che gli interrupt sono disabilitati il maggior problema è dato dalla busy wait delle procedure di gestione dell'I/O del kernel. Questo metodo introduce un overhead maggiore rispetto all'approccio A, richiede modifiche al kernel quando una periferica viene aggiunta o rimossa.

## Approccio C

Lasciare gli interrupt abilitati e ridurre al minimo possibile i driver per le periferiche. L'unico scopo dei driver è quello di attivare una task che gestisce il device. Una volta attivato il task di gestione del device viene eseguito sotto diretto controllo del sistema operativo ed è schedato come un task normale. In questo modo la priorità che può essere assegnata al task di gestione del device è completamente indipendente dalle altre priorità e può essere impostata in base ai requisiti del programma.

Il vantaggio principale di questo metodo è che il busy wait viene eliminato e il tempo di esecuzione del driver di gestione del device è molto ridotto, di conseguenza il tempo di esecuzione del task è più prevedibile. Nella maggior parte delle applicazioni pratiche il piccolo overhead portato da questo tipo di driver viene ignorato.

## 3.4 Semafori

L'usuale meccanismo dei semafori utilizzato nei sistemi operativi tradizionali non è adatto per i sistemi real-time per via del fenomeno dell'**inversione di priorità**. L'inversione di priorità deve essere evitato a tutti i costi in un sistema real-time, dato che introduce non determinismo sull'esecuzione di task critiche. Per quanto riguarda il problema della mutua esclusione l'inversione di priorità può essere evitata usando protocolli specifici che devono essere usati ogni volta che si effettua un accesso alla sezione critica. L'implementazione di questo tipo di protocollo potrebbe richiedere estensive modifiche al kernel, che non comprendono solo le primitive *wait* e *signal*, ma anche strutture dati e meccanismi di task management.

## 3.5 Gestione della memoria

Come gli altri aspetti del kernel, la gestione della memoria deve essere progettata per evitare il non determinismo. Ad esempio sistemi di paging on-demand non sono adatti a sistemi con rigidi vincoli temporali a causa dei lunghi e imprevedibili ritardi causati dalle page fault e page replacement. Le soluzioni tipiche adottate nei sistemi real-time prevedono regole di segmentazione della memoria con uno schema fisso. Il partizionamento statico è particolarmente efficace quando i programmi richiedono una quantità di memoria simile tra loro. In generale gli schemi di allocazione statici aumentano la predicibilità del sistema, ma diminuiscono la flessibilità in sistemi dinamici, dunque la scelta progettuale va fatta bilanciando le due caratteristiche.

## 3.6 Linguaggi di programmazione

Con l'aumentare della complessità dei sistemi real-time aumenta anche la necessità di utilizzare astrazioni offerte dai linguaggi.

Purtroppo i linguaggi attualmente a disposizione non sono abbastanza espressivi da permettere di esprimere certi comportamenti temporali, dunque non sono adatti per la programmazione di sistemi real-time. L'esistenza di costrutti che modificano il flusso di controllo del programma in maniera non deterministica, come gli *if*, rende impossibile la stima affidabile dei WCETs in attività concorrenti. Un'altra fonte di imprevedibilità è

---

data dalla ricorsione, l'analizzatore di scheduling non è in grado di prevedere il numero di chiamate ricorsive e il tempo di esecuzione.

# Chapter 4

## Concetti di base di scheduling

### 4.1 Introduzione

Un **processo** è una computazione eseguita dalla CPU in maniera sequenziale, d'ora in poi useremo il termine processo come sinonimo di *task* o *thread*.

In realtà un processo è una entità più complessa composta da vari task (o thread) che condividono lo stesso spazio di indirizzamento e le stesse risorse. Una **politica di scheduling** è un criterio secondo il quale la CPU viene assegnata ai vari task che si sovrappongono. L'insieme di politiche che in ogni istante di tempo determinano l'ordine con il quale i task vengono eseguiti è detto **algoritmo di scheduling**.

L'operazione specifica di assegnare la CPU a un task selezionato dall'algoritmo di scheduling è detta **dispatching**. Quando un processo può essere potenzialmente eseguito dalla CPU, possiamo avere due casi:

- Il processo è in **running** perché è già stato selezionato dall'algoritmo di scheduling e la CPU è stata assegnata a lui.
- Il processo è in **ready** e aspetta di essere selezionato dall'algoritmo di scheduling.

In entrambi i casi il processo viene detto un task **attivo**. Tutti i processi in wait vengono tenuti in una **ready queue**. I vari sistemi operativi possono utilizzare anche più code dipendentemente dal tipo di task. In molti sistemi dove è concessa l'attivazione dinamica dei task, questi possono essere interrotti in qualsiasi momento, così che quando un task più importante entra nel sistema esso possa essere eseguito immediatamente senza aspettare nella ready queue. L'operazione di interrompere il task in esecuzione e reinserirlo nella

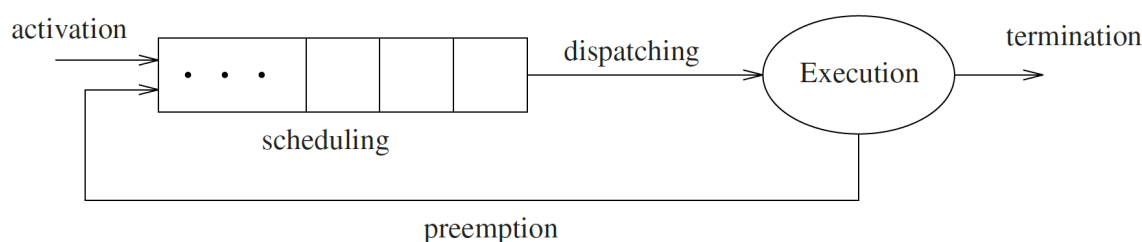


Figure 4.1: Coda di task in ready che aspettano di essere eseguiti

ready queue è detta **preemption**.

Nei sistemi real-time la preemption è importante per 3 motivi:

- I task che gestiscono le eccezioni potrebbero aver bisogno di fare preemption di task in esecuzione per garantire che le eccezioni siano gestite in tempo.
- Quando i task hanno livelli di priorità diversi, i task critici possono essere eseguiti immediatamente.
- Tipicamente lo scheduling con preemption permette maggiore efficienza, nel senso che permette l'esecuzione di task real time con una percentuale di utilizzo di CPU maggiore.

D'altro canto la preemption distrugge la località dei programmi e introduce overhead, di conseguenza limitare la preemption in un sistema real-time può essere benefico.

Uno schedule è detto **feasible** se tutti i task possono essere eseguiti rispettando l'insieme di vincoli a essi posti. Un insieme di task è detto **schedulabile** se esiste uno schedule che rispetta i vincoli di tutti i task.

## 4.2 Tipi di vincoli di scheduling

Tipicamente i vincoli applicabili a task real time rientrano in tre classi:

- Vincoli temporali
- Vincoli di precedenza
- Vincoli di mutua esclusione su risorse condivise

### Vincoli temporali

I sistemi realtime sono caratterizzati da attività computazionali con stringenti vincoli temporali che devono essere rispettati per ottenere il comportamento desiderato. Un requisito tipico è che un task deve essere completato entro un certo tempo, detto **deadline**. Se la deadline è rappresentata in rispetto a quando il task è stato attivato, essa è detta **relative deadline**, altrimenti è detta **absolute deadline**.

In base alle conseguenze a cui porta il non rispetto della deadline le task possono essere suddivise in:

- **Hard** se il non rispetto della deadline porta a conseguenze catastrofiche, ad esempio il fallimento di un sistema di controllo di un aereo.
- **Firm** se il non rispetto della deadline porta a un output inutile al sistema ma non produce conseguenze catastrofiche.
- **Soft** se il non rispetto della deadline porta a una degradazione della qualità del servizio ma non produce conseguenze catastrofiche.

In generale un task realtime  $\tau_i$  è caratterizzato da:

- **Tempo di arrivo**  $a_i$ , il momento in cui il task è pronto per essere eseguito, anche detto **release time** o **request time** e indicato da  $r_i$ .

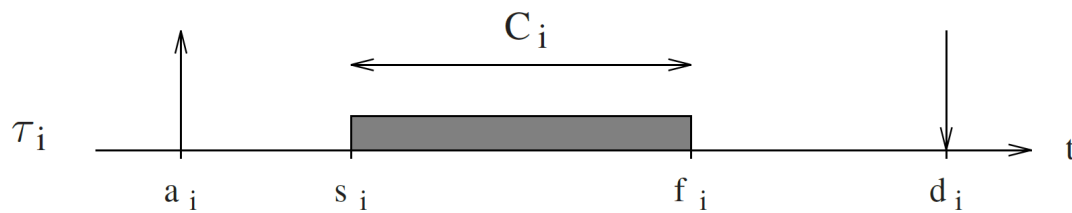


Figure 4.2: Parametri di un task

- **Tempo di computazione**  $c_i$ , il tempo necessario per completare il task senza interruzioni, l'upperbound è il WCET.
- **Deadline assoluta**  $d_i$ , il tempo entro il quale il task deve essere completato.
- **Deadline relativa**  $D_i$ , il tempo entro il quale il task deve essere completato rispetto al suo tempo di arrivo,  $D_i = d_i - r_i$ .
- **Tempo di inizio**  $s_i$ , il tempo in cui il task inizia a essere eseguito.
- **Tempo di completamento**  $f_i$ , il tempo in cui il task termina la sua esecuzione.
- **Tempo di risposta**  $T_i$ , il tempo di ritardo del task, ovvero  $T_i = f_i - r_i$ .
- **Criticità**, già discusso: hard, firm o soft.
- **Lateness**  $L_i$ , il tempo di ritardo del task rispetto alla deadline, ovvero  $L_i = f_i - d_i$ , in generale vogliamo che sia negativa, ovvero i task devono essere eseguiti entro le deadline.
- **Tempo di slack**  $X_i = d_i - a_i - c_i$ , il tempo massimo per cui l'attivazione di un task può essere ritardata senza violare la deadline.

Un'altra caratteristica temporale che riguarda la regolarità dell'attivazione di un task. Un task è detto **periodico** se consiste di una infinita sequenza di attività identiche, chiamate **job**, che si attivano a intervalli regolari. Per chiarezza di notazione notiamo i task periodici con  $\tau_i$  e quelli aperiodici con  $J_i$ . Il generico k-esimo job di un task periodico  $\tau_i$  è denotato con  $\tau_{i,k}$ . Il tempo di attivazione della prima istanza di un task periodico ( $\tau_{i,1}$ ) è detto **fase**, denotato con  $\phi_i$ . Il periodo di attivazione di un task periodico è denotato con  $T_i$ , che è il tempo tra due attivazioni consecutive. I task **aperiodici** sono caratterizzati da un tempo di attivazione non regolare, un task **sporadico** è un task aperiodico che ha un tempo di attivazione massimo, ovvero il tempo minimo tra due attivazioni consecutive.

## Vincoli di precedenza

In alcune applicazioni le attività non possono essere eseguite in ordine arbitrario, ma devono rispettare un ordine di precedenza stabilito durante la progettazione. Tali precedenza tendenzialmente vengono rappresentate tramite un grafo diretto aciclico (DAG), dove i nodi rappresentano le attività e gli archi rappresentano le precedenza. Un DAG

definisce un relazione di ordine parziale tra le attività, che può essere usata per determinare un ordine di esecuzione.

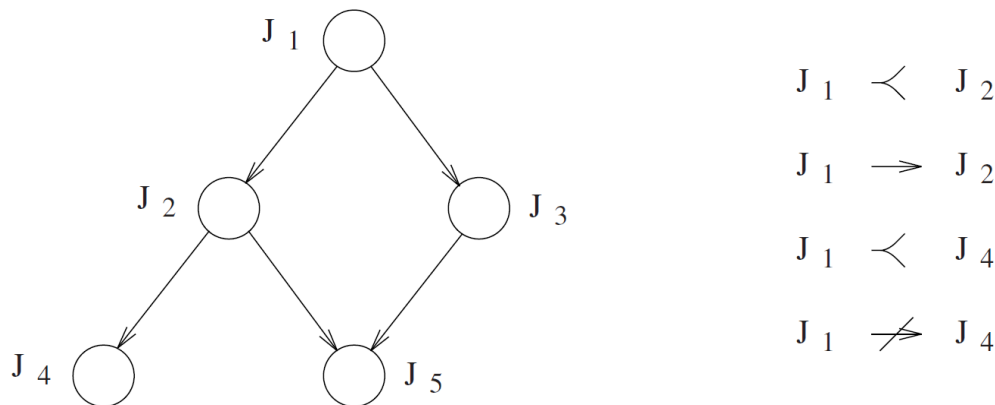


Figure 4.3: Relazione di precedenza tra 5 task

La notazione  $J_a \prec J_b$  indica che il task  $J_a$  è un predecessore di  $J_b$ , ovvero il grafo contiene un cammino orientato da  $J_a$  a  $J_b$ .

La notazione  $J_a \rightarrow J_b$  indica che  $J_a$  è un predecessore diretto di  $J_b$ , ovvero il grafo contiene un arco diretto da  $J_a$  a  $J_b$ .

I task senza predecessori vengono detti **task iniziali**, mentre quelli senza successori vengono detti **task finali**.

## Vincoli di mutua esclusione

Dal punto di vista di un processo, una risorsa è una struttura software che può essere utilizzata dal processo per avanzare la sua esecuzione. Una risorsa è detta *privata* se è dedicata a un processo, altrimenti se può essere utilizzata da più processi è detta **condivisa**. Per mantenere la correttezza del dato, molte risorse condivise non permettono l'accesso simultaneo da parte di task concorrenti, dunque richiedono la **mutua esclusione**. La parte di codice che accede alla risorsa condivisa in mutua esclusione è detta **sezione critica**.

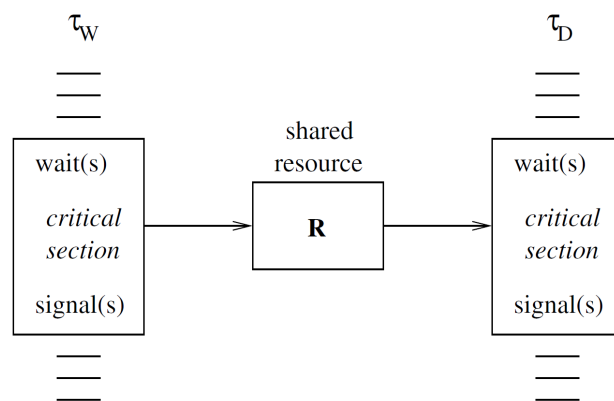


Figure 4.4: Struttura di due task che accedono a una risorsa condivisa in mutua esclusione, tramite meccanismo di semafori

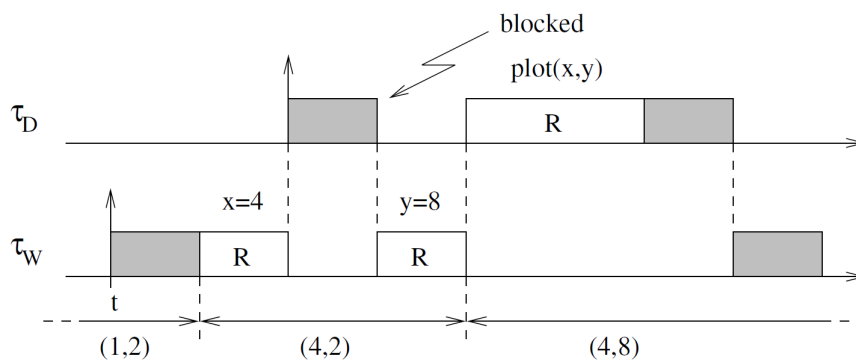


Figure 4.5: Esempio di scheduling quando la risorsa è protetta da un semaforo

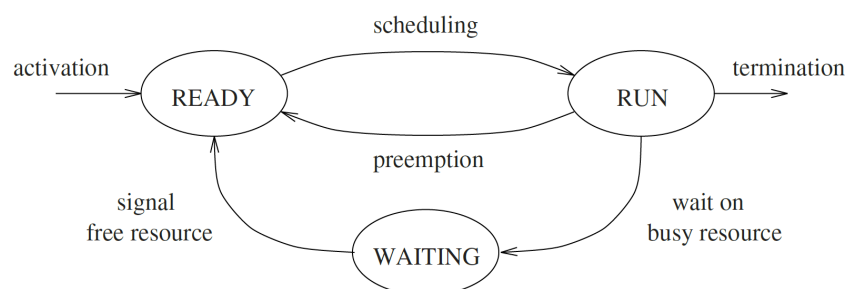


Figure 4.6: Stato di wait causato da un vincolo sulle risorse

Quando un task è in attesa di una risorsa condivisa, si dice che è *bloccato* su quella risorsa. Tutti i task bloccati su una stessa risorsa vengono messi in una **wait queue** associata a un semaforo che protegge la risorsa. Quando un task in esecuzione utilizza la primitiva *wait* su un semaforo bloccato, entra nello stato di **wait**, fino a quando un altro task non



esegue la primitiva *signal* che sblocca il semaforo. Nota che quando un task lascia lo stato di wait non entra immediatamente nello stato di running, ma entra nello stato di ready, cosicché la CPU possa essere assegnata a un task con priorità più alta dall'algoritmo di scheduling.

## 4.3 Classificazione di algoritmi di scheduling

Le classi principali di algoritmi di scheduling sono:

- **Preemptive vs. non preemptive**

- Preemptive: un task può essere interrotto in qualsiasi momento da un task con priorità più alta, secondo una policy di scheduling definita in precedenza.
- Non preemptive: un task una volta avviato deve essere eseguito fino al termine della sua esecuzione. Tutte le scelte di scheduling vengono effettuate quando il task termina la sua esecuzione.

- **Static vs. Dynamic**

- Static: le decisioni di scheduling avvengono in base a un insieme di parametri che vengono definiti prima dell'attivazione dei task.
- Dynamic: le decisioni di scheduling avvengono in base a un insieme di parametri che possono essere modificati durante l'evoluzione del sistema.

- **Off-line vs. online**

- Off-line: Le decisioni di scheduling vengono interamente prese per l'intero insieme di task prima dell'attivazione di essi. Questo scheduling è memorizzato in una tabella e successivamente viene eseguito dal dispatcher.
- Online: Le decisioni di scheduling vengono prese a runtime ogni volta che un nuovo task entra nel sistema, o quando un task running termina. Notare che questo genera un overhead.

- **Ottimale vs. euristico**

- Ottimale: Un algoritmo è detto ottimale se minimizza la "funzione costo" definita per l'insieme di task. Quando non è definita questa funzione costo il nostro unico obiettivo è ottenere uno schedule feasible (se esiste).
- Euristico: Un algoritmo è detto euristico se è guidato da una funzione euristica per prendere decisioni di scheduling. Un algoritmo euristico tende verso lo scheduling ottimale, ma non è garantito che lo raggiunga.

Inoltre un algoritmo è detto *chiaroveggente* se conosce il futuro, ovvero conosce in anticipo il tempo di arrivo di tutti i task. Questo algoritmo chiaramente non esiste, ma è utilizzato come benchmark per valutare gli altri algoritmi in confronto al caso perfetto.

## Algoritmi di scheduling basati sulla garanzia

Nelle applicazioni hard real-time sono richiesti comportamenti altamente predicibili, la feasibility di uno scheduling dovrebbe essere garantita in anticipo (prima dell'attivazione dei task). In questa maniera, così che se un task critico non può essere schedulato prima della sua deadline, il sistema è in tempo per eseguire un'azione alternativa che tenta di evitare conseguenze catastrofiche. Per controllare la feasibility di uno scheduling prima dell'esecuzione dei task il sistema deve pianificare le sue azioni assumendo il caso peggiore. Quindi vanno analizzati i parametri del task set per verificare che esista uno scheduling ammissibile. Nei sistemi statici, ovvero il task set è fisso e noto a priori, tutte le attivazioni dei task possono essere pre calcolate off-line, e tutto lo schedule può essere memorizzato in una tabella che contiene tutti i task garantiti. Poi a runtime il dispatcher può semplicemente eseguire lo schedule memorizzato. Il vantaggio principale dell'approccio statico è che l'overhead a runtime non dipende dalla complessità dell'algoritmo di scheduling. Questo permette di utilizzare algoritmi di scheduling sofisticati per risolvere problemi complessi. D'altro canto, il sistema è rigido rispetto a cambiamenti nell'ambiente. Nei sistemi dinamici (che tipicamente consistono di task *firm*) i task possono essere creati a runtime, dunque la garanzia di scheduling deve essere effettuata online ogni volta che un nuovo task viene creato.

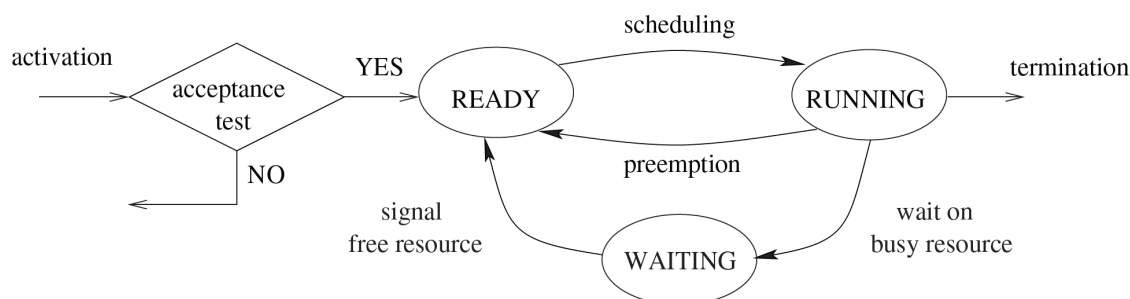


Figure 4.7: Meccanismo di garanzia in un sistema dinamico real time

Inoltre deve essere garantito che quando un task viene accettato, esso non deve interdire la feasibility di task garantiti in precedenza.

## Algoritmi di scheduling best effort

Nelle applicazioni real-time soft, i task non hanno vincoli temporali stringenti, ma devono essere eseguiti in un tempo ragionevole per soddisfare i requisiti si sistema. Dunque il mancato rispetto delle deadline non porta a conseguenze catastrofiche, ma porta a una degradazione delle performance. Un algoritmo best effort cerca di fare "del suo meglio" per rispettare le deadline, ma non garantisce di trovare uno scheduling feasible. Negli approcci best effort i task potrebbero essere messi in coda in base a delle politiche che considerano i requisiti temporali, ma non viene controllata la feasibility e quindi un task potrebbe venir abortito durante la sua esecuzione.

Mediante le performance di un algoritmo best effort sono migliori rispetto a quelle di un algoritmo basato sulla garanzia, infatti un algoritmo basato sulla garanzia potrebbe,

basandosi sul caso peggiore, rifiutare inutilmente un task che in realtà sarebbe riuscito a rispettare la deadline.

# Chapter 5

## Scheduling di task aperiodici

Gli algoritmi affrontati in questa sezione riguardano task aperiodici real-time, eseguiti su una macchina single core. Ogni algoritmo è una soluzione a un problema di scheduling espresso come insieme di assunzioni sulle task e criteri di ottimalità da utilizzare. Pur essendo pensati per task aperiodici su macchine monoprocesso, molti di questi algoritmi possono essere estesi per lavorare su sistemi multiprocessore o architetture distribuite per lavorare con task set più complessi.

Per descrivere un problema di scheduling utilizziamo i parametri  $\alpha|\beta|\gamma$ :

- $\alpha$  è il tipo di macchina (uniprocessore, multiprocessore, distribuito, ecc.)
- $\beta$  descrive le task e le caratteristiche delle risorse (preemptive, indipendente vs vincoli di precedenza, attivazione asincrona, etc.)
- $\gamma$  è il criterio di ottimalità

### 5.1 Garanzie

Per garantire che esista uno scheduling feasible dobbiamo mostrare che, nel caso peggiore, tutti i task vengano completati prima delle rispettive deadline. Vale a dire che per ogni task dobbiamo mostrare che il tempo di completamento nel caso peggiore  $f_i$  è minore o uguale alla rispettiva deadline  $d_i$ .

$$f_i \leq d_i, \forall i \in \{1, 2, \dots, n\} \quad (5.1)$$

Se un task ha requisiti hard, quest'analisi di fattibilità deve essere effettuata prima dell'esecuzione dei task. Senza perdita di generalità possiamo assumere che i task  $J_1, J_2, \dots, J_n$  siano elencati in ordine di deadline crescente, ovvero  $J_1$  ha la deadline più vicina. In questo caso il tempo di completamento del task  $J_i$  nel caso peggiore è dato da:

$$f_i = \sum_{k=1}^i C_k \quad (5.2)$$

Dunque in un task set con  $n$  task il test di garanzia consiste nel verificare che:

$$\forall i \in \{1, 2, \dots, n\} \sum_{k=1}^i C_k \leq d_i \quad (5.3)$$

## 5.2 Earliest Deadline First (EDF)

Se le task non sono sincrone ma possono avere tempo di arrivo arbitrario, la preemption diventa un fattore importante. In generale un problema di scheduling dove la preemption è permessa è sempre più facile da risolvere rispetto allo stesso problema senza preemption. Una soluzione elegante a problemi del tipo  $1|preem|L_{max}$  è data dall'algoritmo **Earliest Deadline First (EDF)**.

**Theorem 1** *Dato un insieme di  $n$  task con tempo di arrivo arbitrario, ogni algoritmo che a ogni istante esegue il task con la deadline assoluta più vicina tra tutti i task attivi è ottimale rispetto alla minimizzazione della lateness.*

La complessità computazionale dell'algoritmo è  $O(n)$  per ogni task se la ready queue è implementata come lista, e  $O(\log n)$  se la ready queue è implementata come heap. Da notare che EDF non fa assunzioni esplicite sulla periodicità dei task, dunque può essere usato sia per task set aperiodici che per task set periodici.

### Garanzia

Quando i task hanno attivazioni dinamiche e il tempo di arrivo non è noto a priori, il test di garanzia deve essere fatto dinamicamente ogni volta che un task entra nel sistema. Sia  $\mathcal{T}$  l'insieme corrente di task attivi, che sono stati garantiti in precedenza, sia  $J_{new}$  il nuovo task che entra nel sistema. Per poter accettare  $J_{new}$  dobbiamo garantire che il nuovo task set  $\mathcal{T} \cup J_{new}$  sia schedulabile.

Sotto le assunzioni di 6 possiamo verificare la schedulabilità di un task set periodico con EDF attraverso il fattore di utilizzo del processore. In questo caso  $U_{lub} = 1$ , quindi un task potrebbe usare il 100% della CPU ed essere schedulabile.

**Theorem 2** *Un task set di task periodici è schedulabile da EDF se e solo se:*

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \quad (5.4)$$

## 5.3 EDF con vincoli di precedenza

Consideriamo problemi di scheduling del tipo  $1|prec, preem|L_{max}$ , dove i task sono soggetti a vincoli di precedenza. Il problema di schedulare un insieme di  $n$  task con vincoli di precedenza con attivazioni dinamiche può essere risolto in tempo polinomiale SOLO se i task sono preentable. L'idea alla base dell'approccio è quella di trasformare il set di task dipendenti  $\mathcal{T}$  in un insieme di task indipendenti  $\mathcal{T}^*$  tramite adeguate modifiche ai parametri temporali, dopodiché possiamo applicare l'algoritmo EDF. L'algoritmo di trasformazione garantisce che  $\mathcal{T}$  sia schedulabile se e solo se  $\mathcal{T}^*$  è schedulabile. In pratica tutti i release time e le deadline sono modificati in modo che ogni task non possa cominciare prima del predecessore e non possa fare preemption su i suoi successori.

# Chapter 6

## Scheduling di task periodici

In molti sistemi real-time di controllo, le attività periodiche compongono la maggior parte della computazione del sistema. Quando una applicazione di controllo consiste in molteplici task periodici concorrenti, con vincoli temporali indipendenti, il sistema operativo deve garantire che ogni istanza periodica venga regolarmente attivata con la giusta frequenza e che venga eseguita entro la deadline (che in generale è diversa dal suo periodo).

Per semplificare l'analisi di scheduling vengono assunte le seguenti ipotesi:

1. Le istanze di task periodici  $\tau_i$  sono regolarmente attivate a frequenza costante, ovvero ogni  $T_i$  (periodo del task).
2. Tutte le istanze di un task periodico  $\tau_i$  hanno lo stesso WCET  $C_i$ .
3. Tutte le istanze di un task periodico  $\tau_i$  hanno la stessa deadline relativa  $D_i$ , che è uguale a  $T_i$ .
4. Tutte le task di  $\mathcal{T}$  sono indipendenti.

Inoltre alcune assunzioni implicite sono:

- Nessun task si può bloccare da solo, ad esempio per operazioni di I/O.
- Tutti i task vengono rilasciati appena arrivano nel sistema.
- L'overhead del kernel viene considerato nullo.

Per quest'ultima assunzione potrei evitare problemi se considero solo una percentuale di "potenza di cpu" disponibile, così da lasciare dello spazio per l'overhead del kernel.

Nota che le prime due assunzioni non sono restrittive, dato che in molte applicazioni di controllo ogni attività periodica richiede l'esecuzione della stessa routine a intervalli regolari. Quando si analizzano casi realistici, le assunzioni 3 e 4 vengono rilassate.

Altri parametri importanti per i task periodici sono:

- **Hyperperiod**: minimo intervallo di tempo nel quale lo scheduling si ripete.
- **Critical instant** di un task: tempo di arrivo che produce il response time del task più lungo.

## 6.1 Fattore di utilizzo del processore

Dato un insieme di task periodici  $\Gamma$ , il *fattore di utilizzo* del processore  $U$  è la frazione del tempo di CPU che viene utilizzato per eseguire l'insieme di task. Dato che  $\frac{C_i}{T_i}$  è la frazione di tempo di CPU utilizzata da un task  $\tau_i$ , il fattore di utilizzo del processore è dato da:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \quad (6.1)$$

Questo valore ci fornisce una misura del carico computazionale sulla CPU. Nonostante possa essere migliorato aumentando il tempo di computazione dei task o diminuendo il loro periodo, esiste un valore massimo di  $U$  per il quale  $\Gamma$  è schedulabile. Tale limite dipende dalla relazione dei periodi dei task e dall'algoritmo di scheduling utilizzato. Sia  $U_{ub}(\Gamma, A)$  il limite superiore al fattore di utilizzo del processore per un insieme di task  $\Gamma$  schedulato con l'algoritmo  $A$ . Quando  $U = U_{ub}(\Gamma, A)$ , si dice che  $\Gamma$  *utilizza completamente* il processore. In questo caso  $\Gamma$  è schedulabile da  $A$ , ma un qualsiasi aumento nel tempo di computazione rende il task set non feasible. Per un algoritmo di scheduling  $A$ , il limite superiore minore  $U_{lub}(\Gamma, A)$  è il minimo fattore di utilizzo su tutti i task set che utilizzano completamente il processore.

$$U_{lub}(\Gamma, A) = \min_{\Gamma} U_{ub}(\Gamma, A) \quad (6.2)$$

Dato che  $U_{lub}(\Gamma, A)$  è il minimo di tutti gli upper bound, qualsiasi task set con fattore di utilizzo minore di  $U_{lub}(\Gamma, A)$  è sicuramente schedulabile da  $A$ . In questa maniera è facilmente determinabile se un task set è schedulabile o meno con un determinato algoritmo.

- Se  $U_{lub} < U < 1.0$ , la schedulabilità viene raggiunta solo se i periodi sono compatibili.
- Se  $U_{lub} > 1.0$ , il task set non è schedulabile da nessun algoritmo.

## 6.2 Timeline scheduling

Anche noto come *Cyclic Executive*, è uno degli approcci più adottati per gestire task periodici nei sistemi di difesa militari e di controllo di traffico. Il metodo consiste nel dividere l'asse temporale in slot di egual lunghezza, nei quali uno o più task vengono allocati per l'esecuzione; in modo da rispettare le frequenze date dai requisiti. Un timer sincronizza l'attivazione dei task all'inizio di ogni slot.

Consideriamo un esempio dove tre task  $A, B$  e  $C$  devono essere eseguiti con frequenze rispettivamente di  $40Hz$ ,  $20Hz$  e  $10Hz$ . Dunque i rispettivi periodi sono  $T_A = 25ms$ ,  $T_B = 50ms$  e  $T_C = 100ms$  (N.B.  $F = \frac{1}{T}$ ), si ricava che la lunghezza ottimale dello slot è di  $25ms$ , che è il massimo comune divisore dei periodi. Quindi il task  $A$  viene eseguito ogni slot, il task  $B$  ogni 2 slot e il task  $C$  ogni 4 slot.

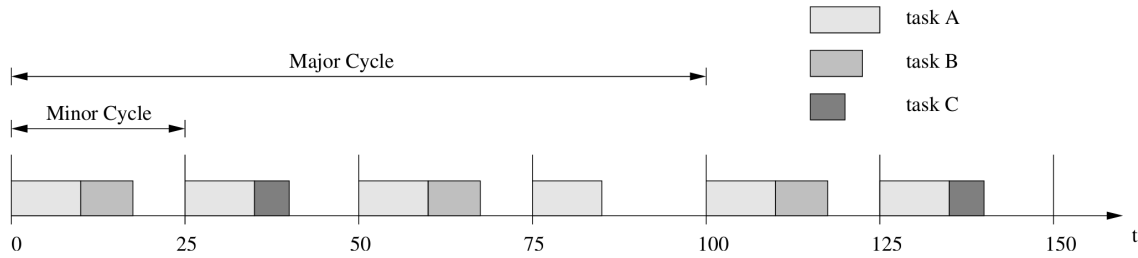


Figure 6.1: Esempio di timeline scheduling

La durata dello slot è detta *Minor cycle*, mentre l'intervallo minimo di tempo dopo il quale uno schedule si ripete è detto *Major cycle*.

In generale, il major cycle è uguale al minimo comune multiplo di tutti i periodi. Per poter garantire a priori che lo schedule sia feasible su un processore, è sufficiente sapere il WCET dei task e verificare che la somma all'interno di ogni slot sia minore o uguale al minor cycle.

Il **vantaggio** principale di timeline scheduling è la sua **semplicità**. Può essere implementato tra programmando un timer che interrompa con periodo uguale al minor cycle e scrivendo un ciclo main dove i task vengono chiamati nell'ordine dato dal major cycle, inserendo punti di sincronizzazione a ogni inizio del minor cycle. Dato che la sequenza di attivazione dei task non è determinata da un algoritmo di scheduling nel kernel, ma è determinata dalle chiamate effettuate dal main, non vi sono context switch, dunque l'overhead a runtime è molto basso. Nonostante questi vantaggi, vi sono dei problemi, ad esempio l'alta fragilità in situazioni di overload. Se un task non termina nel limite del minor cycle, può essere continuato o abortito. In entrambi i casi, il sistema entra in uno stato critico. Se il task fallimentare rimane in esecuzione esso può causare un effetto a domino sugli altri task, rompendo l'intero schedule (timeline break). Se un task viene abortito mentre compie azioni su risorse condivise, esso può lasciare le risorse in uno stato inconsistente e causare un comportamento non corretto. Inoltre è particolarmente sensibile a cambiamenti nei requisiti.

### 6.3 Rate Monotonic Scheduling (RMS)

Funziona tramite una semplice regola che assegna la priorità a un task in base alla sua frequenza di attivazione. In particolare, task con periodi più brevi hanno priorità più alta. Dato che i periodi sono costanti, RM è un sistema fixed priority, ovvero la priorità  $P_i$  di un task  $i$  non cambia durante l'esecuzione. RM è intrinsecamente preemptive, ovvero un task con priorità più alta può interrompere un task con priorità più bassa. È stato dimostrato da Liu e Layland che RM è ottimale rispetto a tutti gli assegnamenti di priorità fissi, nel senso che nessun altro sistema di scheduling a priorità fissa può schedulare un task set che non può essere schedulato da RM.

Liu e Layland hanno inoltre derivato che  $U_{lub}$  per RM su un task set di  $n$  task è dato da:

$$U_{lub} = n(2^{\frac{1}{n}} - 1) \quad (6.3)$$



n	$U_{lub}$	n	$U_{lub}$
1	1.000	6	0.73
2	0.828	7	0.729
3	0.780	8	0.724
4	0.757	9	0.721
5	0.743	10	0.718

(a)
(b)

Table 6.1: Valori di  $U_{lub}$  per RM al variare di  $n$ 

Il bound diminuisce all'aumentare di  $n$ . Per valori alti  $U_{lub}$  converge a  $U_{lub} = \ln 2 \approx 0.69$ .

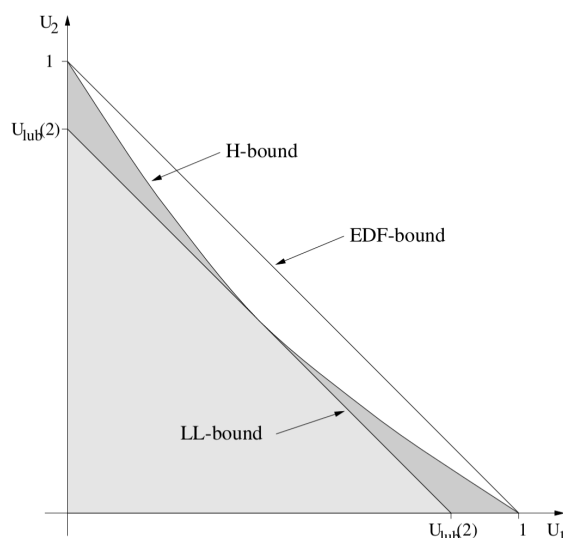
### Bound iperbolico per RM

L'analisi di fattibilità su RM può anche essere effettuata utilizzando un bound iperbolico. Il test ha la stessa complessità computazionale di quello di Liu e Layland, ma è meno pessimistico, dato che accetta task set che verrebbero rifiutati dal test LL. Il seguente teorema offre una condizione sufficiente per testare la schedulabilità di un task set con RM.

**Theorem 3** *Sia  $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$  un insieme di task periodici dove ogni task  $\tau_i$  è caratterizzato dal fattore di utilizzo  $U_i$ . Allora  $\Gamma$  è schedulabile da RM se:*

$$\prod_{i=1}^n (U_i + 1) \leq 2 \quad (6.4)$$

Rappresentiamo in un piano  $n$  dimensionale il bound LL che interseca gli assi in  $U_{lub} = n(2^{\frac{1}{n}} - 1)$ . Tutti i punti al di sotto la superficie di RM rappresentano task set periodici schedulabili da RM. Il nuovo bound viene rappresentato da una superficie iperbolica tangente al piano RM e che interseca gli assi in  $U_i = 1$ . Rappresentiamo in figura il bound iperbolico per  $n = 2$ . Da notare che gli asintoti dell'iperbole sono a  $U_1 = -1$ . Dal plot possiamo notare che la regione di ammissibilità è più grande per il test H-bound che per il test LL.

Figure 6.2: Bound iperbolico per RM con  $n = 2$ 

## 6.4 RM vs EDF

Il problema di schedulare un insieme di task periodici indipendenti e preemptable è stato risolto sia con sistemi a priorità fissa (RM) che con sistemi a priorità dinamica (EDF). Il vantaggio principale di sistemi a priorità fissa è la semplicità di implementazione, inoltre se la ready queue è implementata come una queue multi livello con  $P$  livelli di priorità, sia l'inserimento che la rimozione di un task dalla ready queue richiede  $O(1)$  tempo.

D'altro canto in uno scheduler deadline driven, la soluzione migliore è l'implementazione tramite heap, dove l'inserimento e la rimozione richiedono  $O(\log n)$  tempo. Per quanto riguarda l'utilizzo del processore, EDF è in grado di utilizzare completamente la CPU, mentre RM può garantire la fattibilità di task set con fattore di utilizzo minore al 69% nel caso peggiore.

Nonostante la computazione maggiore richiesta da EDF per aggiornare le deadline assoluta a ogni attivazione di task, EDF introduce meno overhead a runtime quando vengono considerati i context switch. Infatti, per applicare l'ordine a priorità fissa, il numero di preemption che tipicamente si verifica con RM è *molto maggiore* rispetto a EDF.

Una proprietà interessante di EDF durante overload permanenti è che viene performata in automatico un riscalaggio dei periodi, così che si comportino come se venissero eseguiti a frequenza più bassa, questo fenomeno viene chiamato **graceful degradation** e deve essere evitato. Come verrà discusso in seguito, un altro grande vantaggio dello scheduling dinamico rispetto ai sistemi a priorità fissa è una migliore responsività nel gestire task aperiodici. Questa proprietà deriva dal fattore di utilizzo del processore maggiore in EDF, infatti il bound minore di RM limita l'utilizzo massimo che può essere assegnato a un server per garantire la schedulabilità di un task set periodico. Di conseguenza l'utilizzo di processore che avanza che non può essere assegnata al server viene sprecata come esecuzione di background. Questo non avviene con EDF.

# Chapter 7

## Server a priorità fissa

Gli algoritmi di scheduling trattati in precedenza lavorano su task set omogenei, dove tutti le attività computazionali sono periodiche o aperiodiche. Nella realtà dei fatti nelle applicazioni di controllo real time i due tipi si possono mischiare, avendo anche diversi livelli di criticità. Tipicamente i task periodici sono time-driven ed eseguono attività di controllo critiche con vincoli di temporizzazione hard con lo scopo di garantire la attivazione regolare dei task. Quando i task set sono ibridi lo scopo principale del kernel è di garantire la schedulabilità di tutti i task critici nel caso peggiore, e di offrire buoni response time per gli altri. Tutti gli algoritmi che verranno trattati in questa sezione sono basati sulle seguenti assunzioni:

- I task periodici sono schedulati tramite RM (fixed priority).
- Tutti i task periodici iniziano a  $t = 0$  e le loro deadline relative sono uguali al loro periodo.
- Il tempo di arrivo dei task aperiodici è arbitrario.
- Quando non viene specificato, il minimo tempo di "interarrivo" di un task periodico è uguale alla sua deadline.
- **Tutti** i task sono preemptable.

### 7.1 Background scheduling

È il metodo più semplice per gestire task aperiodici soft in presenza di task periodici, consiste nel schedulare i task aperiodici quando non ci sono istanze di task periodici in stato di ready. Il problema principale di questo approccio è che, per carichi periodici elevati, il response time delle richieste aperiodiche può essere troppo lungo per alcune applicazioni. Per questo motivo background scheduling può essere adottato solo quando le attività aperiodiche non hanno vincoli di temporizzazione stringenti e il carico di lavoro periodico è basso.

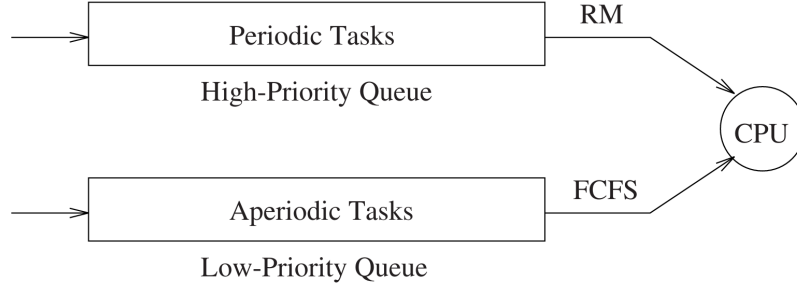


Figure 7.1: Esempio di ready queue per background scheduling

Il vantaggio principale di background scheduling è la sua semplicità. Da notare che come mostrato in figura 7.1, le due strategie di attesa dei task sono indipendenti e possono essere realizzate con algoritmi differenti. L'attivazione di un nuovo task periodico causa la preemption di un qualsiasi task aperiodico.

## 7.2 Polling server

Il tempo di risposta medio di task aperiodici può essere migliorato rispetto a background scheduling attraverso l'utilizzo di un **server**, ovvero, un task periodico il quale scopo è di servire task aperiodici non appena possibile. Come qualsiasi task periodico, il server ha un periodo  $T_s$  e un tempo di computazione  $C_s$ , detto anche *capacità* o *budget*. In generale, il server viene schedulato con lo stesso algoritmo usato per gli altri task periodici e, una volta attivo, serve i task aperiodici nel limite del suo budget. Da notare che l'ordine in cui vengono serviti i task aperiodici non è determinato dall'algoritmo usato per quelli periodici.

Il polling server a intervalli regolari  $T_s$  diventa attivo ed esegue task aperiodici nel limite del suo budget  $C_s$ . Se nessun task aperiodico è in attesa, il server si sospende fino al prossimo periodo e il budget viene lasciato a task periodici.

### Dare una dimensione a un server polling

Dato un set di task periodici, come possiamo computare  $T_s$  e  $C_s$  in maniera da avere uno schedule feasible? Prima di tutto calcoliamo il fattore massimo di utilizzo del server  $U_s^{max}$ , che garantisce la feasibility del task set. Dato che il response time non è facile da manipolare a causa di arrotondamenti, possiamo derivare  $U_s^{max}$  dal test iperbolico 6.4, che è più restrittivo rispetto al test di LL. definiamo:

$$P \stackrel{\text{def}}{=} \prod_{i=1}^n (U_i + 1) \quad (7.1)$$

allora dobbiamo avere

$$P \leq \frac{2}{U_s + 1} \quad (7.2)$$

ovvero

$$U_s \leq \frac{2 - P}{P} \quad (7.3)$$

dunque

$$U_s^{max} = \frac{2 - P}{P} \quad (7.4)$$

Quindi  $U_s$  deve essere minore di  $U_s^{max}$ . Per un dato  $U_s$  ci sono infinite coppie  $(C_s, T_s)$ , quindi ne dobbiamo selezionare una. Una soluzione semplice è assegnare al server la priorità più alta (e quindi in RM con il periodo più breve), però non è utile avere  $T_s < T_1$ . dato che un piccolo  $T_s$  implica un minore  $C_s$  e questo causerebbe una maggiore frammentazione (quindi runtime overhead per i cambi di contesto).

### 7.3 Deferrable server

Pensato per migliorare i response time per task aperiodici rispetto polling server. La differenza è che se nessun task aperiodico è in attesa, il server non si sospende e rimane in attesa (continuando a consumare il budget) fino a quando un task aperiodico non entra nella ready queue e, compatibilmente con il budget rimanente, lo esegue. La capacità viene ricaricata all'inizio di ogni periodo. DS offre una risposta molto più rapida rispetto al PS, dato che preserva la capacità fino a quando server. Tempi di risposta più rapidi possono essere ottenuti dando a DS la priorità più alta tra i task periodici.

### 7.4 Priority exchange

Pensato per servire un insieme di task aperiodici soft in presenza di task periodici hard. Rispetto a PS e DS è leggermente meno performante in termini di risposta a task aperiodici, ma offre migliori bound di schedulabilità per insiemi di task periodici. Come DS e PS utilizza un server periodico (tendenzialmente con alta priorità), con la differenza rispetto DS che riserva la sua capacità ad alta priorità scambiandola con tempo di esecuzione di task periodici a priorità più bassa. All'inizio di ogni periodo del server la capacità viene ricaricata. Se dei task aperiodici sono in attesa e il server è il task ready con priorità più alta, allora le richieste vengono servite consumando la capacità, altrimenti  $C_s$  viene scambiato con tempo di esecuzione di un task periodico con la priorità più alta.

**Quando uno scambio di priorità avviene il task periodico viene eseguito con il livello di priorità del server mentre il server accumula capacità al livello di priorità del task periodico.**

Quindi il task periodico avanza la sua esecuzione e la capacità del server viene conservata a una priorità più bassa. Se nessuna richiesta aperiodica arriva ad utilizzare la capacità del server, la sua priorità continua ad abbassarsi fino a quando o viene utilizzata o raggiunge la priorità di processi di background.

Dato che l'obiettivo di PE è offrire alta responsività per task aperiodici, tutte le priorità vengono "rotte" a favore dei task aperiodici.

### 7.5 Sporadic server

Permette miglioramenti su i tempi di risposta medi su task aperiodici, senza degradare i bound di utilizzo di task set periodici. Viene creato un task ad alta priorità per servire i task aperiodici e, come DS, preserva la capacità alla sua priorità alta fino a quando arriva una richiesta aperiodica. La differenza rispetto a DS è nel ripristino della capacità, SS ricarica la sua capacità solo dopo che è stata consumata dall'esecuzione di task aperiodici.

## 7.6 Slack stealing

Offre miglioramenti sostanziali in termini di response time rispetto a PE, DS e SS. A differenza di questi ultimi, non utilizza un server periodico, ma crea un task passivo chiamato *slack stealer*, che cerca di utilizzare i periodi di slack dei task periodici (il tempo tra la fine dei task e le loro deadline). L'idea fondamentale è che, tipicamente, non vi è beneficio nel completare in anticipo task periodici.

## 7.7 Summary

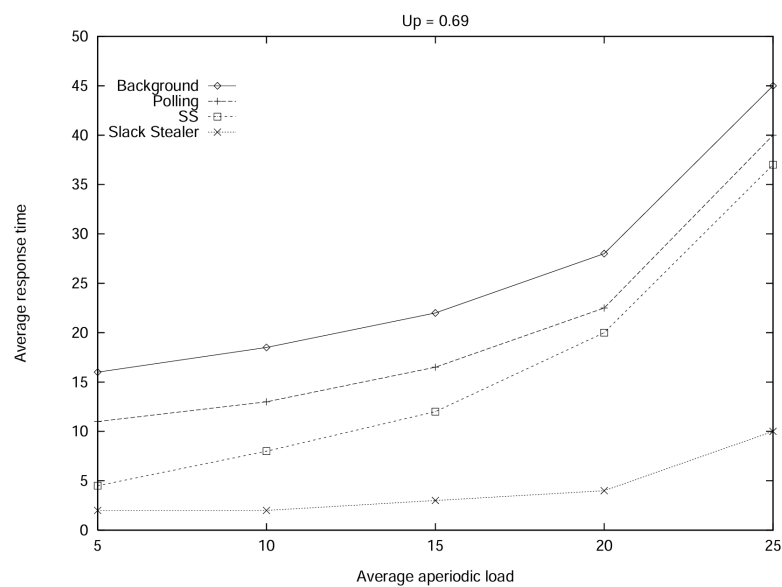


Figure 7.2: Performance dei server




























				
	excellent	good	poor	
	performance	computational complexity	memory requirement	implementation complexity
Background Service				
Polling Server				
Deferrable Server				
Priority Exchange				
Sporadic Server				
Slack Stealer				

Figure 7.3: Confronto overall dei server

# Chapter 8

## Server a priorità dinamica

In questa sezione si discute il problema dello schedare task soft aperiodici e task hard periodici in un sistema a priorità dinamica. In particolare l'obiettivo è ridurre il tempo di risposta medio delle richieste aperiodiche, senza degradare la schedabilità di task hard periodici. I task periodici vengono schedati con EDF (dunque il fattore di utilizzo massimo del processore è 1). L'aumentare del fattore di utilizzo massimo rispetto a RM (0.69) è vantaggioso perché ci permette di aumentare la dimensione massima dei server i.e.  $U_s = 1 - U_p$ . Per semplicità facciamo le seguenti assunzioni:

- Tutti i task periodici hanno deadline hard e la loro schedabilità deve essere garantita off-line.
- Tutti i task aperiodici non hanno deadline e devono essere schedati il prima possibile, senza compromettere la schedabilità dei task periodici.
- La deadline dei task periodici è uguale al loro periodo.
- Tutti i task periodici vengono attivati a tempo  $t = 0$ .
- Tutti i task aperiodici hanno tempo di computazione noto, ma tempo di arrivo arbitrario.

### 8.1 Dynamic priority exchange server

Può essere visto come un'estensione del priority exchange server affrontato nella sezione 7.4. L'idea principale è di lasciare che il server scambi il suo runtime con il runtime di task periodici di priorità più bassa (in EDF ciò significa deadline più lontana) nel caso non ci siano richieste aperiodiche in attesa. In questo modo il runtime del server viene scambiato con il runtime di task periodici, ma mai sprecato perché verrà poi riscattato quando un task aperiodico arriva. Quando l'entità con priorità più alta nel sistema è il server con capacità  $C$  allora:

- Se ci sono richieste aperiodiche nel sistema allora queste vengono servite fino a quando vengono completate o la capacità del server viene esaurita.
- Se non ci sono richieste aperiodiche da soddisfare allora viene eseguito il task periodico con la deadline relativa più vicina, viene aggiunta alla capacità del server il tempo di esecuzione
- e viene sottratta alla deadline del server.

### Utilizzare lo spare time

Nei sistemi hard real-time la garanzia viene effettuata sulla base del worst case scenario, ma quando non viene raggiunto il worst case vi è un tempo di esecuzione minore del WCET. Usando un server DPE, il tempo che avanza può essere facilmente utilizzato per eseguire task aperiodici. Questo viene fatto aggiungendo lo spare time dei task periodici alla capacità del server corrispondente. Da notare che utilizzare lo spare time in questo modo non influisce sulla schedulabilità del sistema.

## 8.2 Dynamic sporadic server

Estensione di sporadic server, trattato nella sezione 7.5. A differenza di altri server la sua capacità non viene ricaricata a ogni periodo, ma solo quando viene completamente consumata. La differenza con lo sporadic server classico è nella priorità del server. SS ha una priorità fissa e viene schedulato con RM, mentre DSS ha una priorità dinamica assegnata in base alla deadline adatta.

## 8.3 Total bandwidth server

Osservando l'approccio del dynamic sporadic server è facilmente notabile che, quando il server ha un periodo lungo, l'esecuzione di task aperiodici può essere ritardata per un lungo periodo di tempo. Una soluzione è assegnare una deadline più vicina a ogni task aperiodico, questo assegnamento deve essere fatto in maniera che l'utilizzo complessivo del processore non superi mai  $U_s$ . Il nome viene dal fatto che quando una richiesta aperiodica entra nel sistema, l'intera bandwidth del server viene assegnata a essa, se possibile. Una volta che la priorità viene assegnata, la richiesta viene inserita nella ready queue del sistema e schedulata con EDF come ogni altro task periodico, di conseguenza l'overhead di questo algoritmo è risibile.

## 8.4 Earliest deadline late server

Versione dinamica dello slack stealer trattato nella sezione 7.6. Utilizziamo i tempi morti per schedulare i task aperiodici il prima possibile. L'idea di base è di utilizzare i tempi morti per eseguire i task aperiodici il prima possibile. Quando non ci sono attività aperiodiche nel sistema, i task periodici vengono schedulati secondo EDF. Quando una nuova richiesta aperiodiche entra nel sistema (e nessuna richiesta aperiodica è ancora attiva) i tempi morti del task set periodico corrente vengono calcolati e usati per schedulare i task aperiodici.

## 8.5 Improved priority exchange server

Nonostante sia ottimo, EDL ha troppo runtime overhead per essere considerato nella pratica. La pesante computazione può essere evitata utilizzando il meccanismo di priority exchange. I tempi morti di EDL possono essere pre calcolati off-line e il server può usarli per schedulare richieste aperiodiche (quando ci sono), o per eseguire task periodici. Nel secondo caso il tempo morto può essere considerato come capacità del server per task



aperiodici al livello di priorità del task periodico eseguito. Il server non risulta quindi più periodico, ma viene sempre eseguito con la priorità più alta del sistema.

## 8.6 Constant bandwidth server

Come DSS garantisce che, se  $U_s$  è la frazione di tempo di CPU utilizzata dal server (bandwidth), il contributo totale del server al fattore utilizzo è minore di  $U_s$  anche in caso di overload. Con CBS si ottengono performance molto migliori rispetto a DSS, confrontabili con quelle ottenibili con TBS.

L'idea di base è che quando un nuovo task entra nel sistema gli venga assegnata una deadline adatta e venga inserito nella ready queue di EDF. Se il task prova a eseguire per più del tempo che abbiamo assegnato la sua deadline viene spostata in avanti (ovvero la priorità viene abbassata) per diminuire l'interferenza con altri task. Nota che abbassando la priorità del task comunque rimane possibilmente il task da eseguire. In questa maniera CBS si comporta come un algoritmo conservativo, sfruttando lo slack disponibile in maniera efficiente (in maniera deadline driven). Se un sottoinsieme di task viene gestito da un solo server, tutti i task di quel sottoinsieme condividono la stessa bandwidth, quindi non c'è isolamento tra loro. Tutti gli altri task sono protetti rispetto a sforamenti interni a quel sottoinsieme.

Definiamo CBS:

- Un CBS è caratterizzato da un budget  $c_s$  e da una coppia ordinata  $(Q_s, T_s)$ , dove  $Q_s$  è il budget massimo e  $T_s$  è il periodo del server. Invece la bandwidth del server è definita come  $U_s = \frac{Q_s}{T_s}$ . A ogni istante una deadline fissa  $d_{s,k}$  viene associata al server. All'inizio  $d_{s,0} = 0$ .
- A ogni job servito  $J_{i,j}$  viene associata una deadline dinamica  $d_{i,j}$  uguale alla deadline corrente del server  $d_{s,k}$ .
- Quando un job del server viene eseguito, il budget  $c_s$  viene diminuito della durata del job.
- Quando il budget  $c_s$  viene esaurito, viene ricaricato a  $Q_s$  e una nuova deadline viene calcolata come  $d_{s,k+1} = d_{s,k} + T_s$ . (nota che non esistono intervalli finiti in cui il budget è zero).
- Un CBS è detto attivo se ci sono jobs in attesa di essere eseguiti, altrimenti è detto inattivo.
- Quando un job  $J_{i,j}$  arriva e il server è attivo la richiesta viene inserita nella queue dei job in attesa, che rispetta una disciplina arbitraria.
- Quando un job  $J_{i,j}$  arriva e il server è inattivo, se  $c_s \geq (d_{s,k} - r_{i,j})U_s$  il server genera una nuova deadline  $d_{s,k+1} = r_{i,j} + T_s$  e il budget viene ricaricato a  $Q_s$ . Altrimenti il job viene servito con deadline  $d_{s,k}$  usando il budget corrente.
- Quando un job termina, il prossimo job in coda (se esiste) viene servito usando il budget e deadline correnti. Se non ci sono jobs in coda, il server diventa inattivo.
- A ogni istante, a un job viene assegnato l'ultima deadline generata dal server.

Una proprietà interessante è quella dell'isolamento, ovvero, possiamo usare una strategia di preservamento della bandwidth per allocare una frazione del tempo di CPU per soft task per cui il tempo di esecuzione non può essere facilmente delimitato. La conseguenza più importante è che i task soft possono essere schedulati assieme a task hard senza avere effetto sulla garanzia a priori di schedulabilità, anche nel caso in cui i tempi di computazione dei task soft effettivi superino quelli previsti.

## Chapter 9

# Protocolli di accesso alle risorse condivise

Una *risorsa* è una struttura software che può essere utilizzata da un processo per avanzare la sua esecuzione. Una risorsa dedicata a un processo in particolare è detta *risorsa privata*, mentre una *risorsa condivisa* è una risorsa che può essere utilizzata da più processi. Una risorsa condivisa protetta rispetto all'accesso concorrente da parte di più processi è detta *risorsa protetta*. Una porzione di codice che accede a una risorsa condivisa è detta *sezione critica*. Quanto un task deve entrare in una sezione critica deve aspettare fino a quando nessun altro task sta accedendo a quella risorsa. Un task che aspetta una risorsa condivisa è detto *bloccato*. Quando un task esce dalla sezione critica, la risorsa viene rilasciata e viene detta *free*.

I sistemi operativi tipicamente offrono un meccanismo di sincronizzazione, detto *semaforo*. Un semaforo è una struttura dati kernel che può essere utilizzata solo attraverso primitive del kernel, tipicamente `wait` e `signal`. Una risorsa  $R_k$  viene protetta da un semaforo  $S_k$ , ogni sezione critica che accede a  $R_k$  deve essere racchiusa tra le primitive `wait( $S_k$ )` e `signal( $S_k$ )`. Tutti i task bloccati da una risorsa vengono messi in una coda associata al semaforo che protegge la risorsa. Quando un task running esegue una `wait` su un semaforo bloccato esso viene messo in wait, fino a quando un altro task esegue una `signal` su quel semaforo. Quando un task lascia lo stato di wait non va in running, ma nello stato di ready, cosicché la CPU possa essere assegnata al task con priorità più alta dall'algoritmo di scheduling.

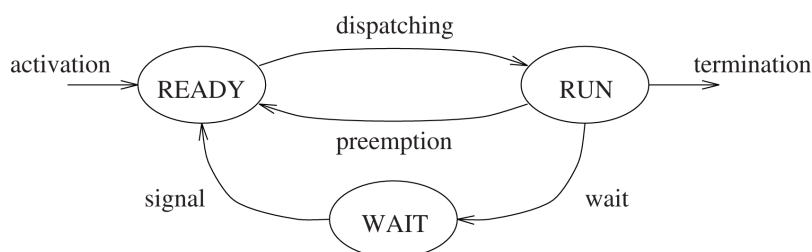


Figure 9.1: Wait causato da una risorsa condivisa

Andremo a descrivere i principali problemi che sorgono in sistemi monoprocesore quando task concorrenti accedono a risorse condivise in mutua esclusione. Presenteremo alcuni

protocolli di accesso alle risorse pensati per evitare questi problemi e limitare il tempo massimo di blocco di ogni task.

## 9.1 Inversione di priorità

Consideriamo due task  $\tau_1$  e  $\tau_2$  che condividono una risorsa  $R_k$  protetta da un semaforo  $S_k$ . Per garantire la mutua esclusione, le operazioni su questa risorsa condivisa devono essere contenute in una sezione critica racchiusa tra le primitive `wait( $S_k$ )` e `signal( $S_k$ )`. Se la preemption è permessa e  $\tau_1$  ha priorità più alta di  $\tau_2$ , allora  $\tau_1$  può essere bloccato da  $\tau_2$  in una situazione raffigurata in figura 9.2. In questo esempio il tempo massimo

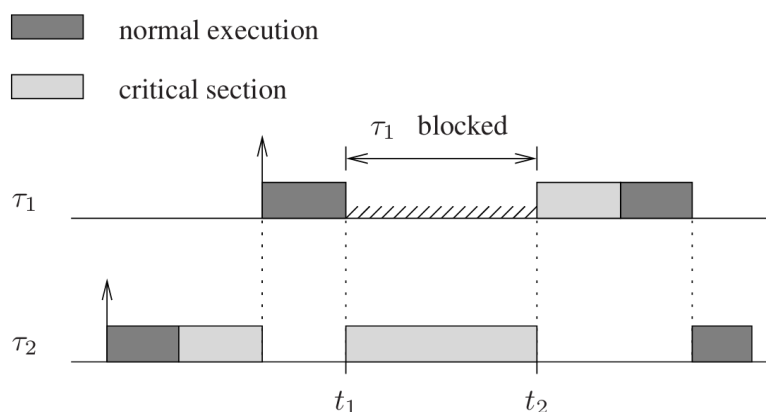


Figure 9.2: Esempio di bloccaggio su una risorsa condivisa

di bloccaggio che  $\tau_1$  può subire è uguale al tempo necessario a  $\tau_2$  per completare la sua sezione critica. Questo bloccaggio non può essere evitato perché è diretta conseguenza della mutua esclusione necessaria per proteggere la risorsa condivisa. Sfortunatamente, in generale, il tempo di blocco di un task su una risorsa occupata non può essere limitato dalla durata della sezione critica eseguita da un task con priorità più bassa. Consideriamo l'esempio in figura 9.3, dove  $\tau_1$ ,  $\tau_2$  e  $\tau_3$  hanno priorità decrescenti e  $\tau_1$  e  $\tau_3$  condividono una risorsa protetta dal semaforo  $S$ .

Se  $\tau_3$  inizia a tempo  $t_0$ ,  $\tau_1$  potrebbe arrivare a tempo  $t_2$  e fare preemption di  $\tau_3$  nella sua sezione critica. Al tempo  $t_3$   $\tau_1$  prova a usare la sua risorsa, ma viene bloccato dal semaforo  $S$ , dunque  $\tau_3$  continua la sua esecuzione della sezione critica. Se  $\tau_2$  arriva al tempo  $t_4$ , preempts  $\tau_3$  (dato che ha priorità più alta) e aumenta il tempo di bloccaggio di  $\tau_1$  per la sua intera durata. Quindi si denota che il tempo di bloccaggio massimo di  $\tau_1$  non dipende solo dai task che condividono una risorsa con esso, ma anche da altri task! Se questa situazione avviene con altri task di priorità intermedia può portare a un blocco senza controllo che può causare il mancato rispetto delle deadline. Dunque in generale il tempo di blocco di un task non può essere limitato.

Diversi approcci sono stati proposti per evitare l'inversione di priorità, sia in contesti a priorità fissa che dinamica. Tutti i metodi sviluppati nel contesto di priorità fissa consistono nell'aumentare la priorità di un task quando accede a un risorsa condivisa, aderendo a un protocollo per entrare e uscire dalle sezioni critiche. Invece nei sistemi a priorità dinamica (quindi con EDF) vengono modificati i parametri in base alle deadline

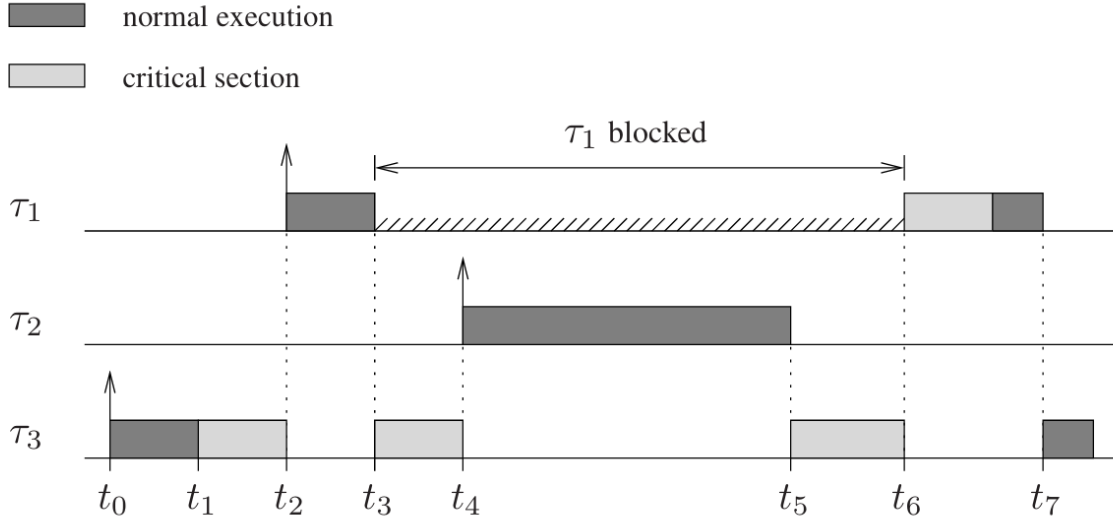


Figure 9.3: Esempio di inversione di priorità

relative dei task.

I protocolli che affrontiamo sono:

1. Non-preemptive protocol (NPP)
2. Highest locker priority (HLP), anche noto come Immediate priority ceiling (IPC)
3. Priority inheritance protocol (PIP)
4. Priority ceiling protocol (PCP)
5. Stack resource policy (SRP)

I primi quattro sono pensati per sistemi a priorità statica, mentre stack resource policy è pensato per entrambi i contesti.

## 9.2 Terminologia e assunzioni

Consideriamo un insieme di  $n$  task periodici  $\tau_1, \tau_2, \dots, \tau_n$ , dove ogni task  $\tau_i$  ha un periodo  $T_i$  e un tempo di computazione  $C_i$ , che cooperano attraverso l'uso di  $m$  risorse condivise  $R_1, R_2, \dots, R_m$ . Ogni risorsa  $R_k$  è protetta da un semaforo  $S_k$ , dunque ogni sezione critica che accede a  $R_k$  deve essere racchiusa tra le primitive `wait( $S_k$ )` e `signal( $S_k$ )`. Dato che un protocollo modifica la priorità di un task, ogni task è caratterizzato da una *priorità nominale*  $P_i$  (decisa ad esempio tramite rate monotonic) e da una *priorità attiva*  $p_i$  ( $p_i \geq P_i$ ) che è dinamica e viene inizialmente impostata uguale a  $P_i$ .

Utilizzeremo le seguenti notazioni:

- $z_{i,k}$  denota una sezione critica generica del task  $\tau_i$  che è protetta da  $S_k$ .
- $Z_{i,k}$  denota la sezione critica più lunga del task  $\tau_i$  che è protetta da  $S_k$ .
- $z_{i,h} \subset z_{i,k}$  indica che la sezione critica  $z_{i,h}$  è una completamente contenuta in  $z_{i,k}$ .

- $\sigma_i$  denota l'insieme di semafori che possono bloccare  $\tau_i$ .

Inoltre, i protocolli proposti funzionano sotto le seguenti assunzioni:

- I task  $\tau_1, \tau_2, \dots, \tau_n$  hanno diverse priorità e sono ordinati in ordine decrescente di priorità nominale, quindi  $\tau_1$  ha priorità nominale più alta.
- I task non si sospendono da soli per operazioni di I/O o per primitive esplicite di sincronizzazione (eccetto per semafori bloccati).
- Le sezioni critiche di ogni sono propriamente innestate, ovvero data una coppia qualsiasi  $z_{i,k}$  e  $z_{i,h}$  allora  $z_{i,h} \subset z_{i,k}$  o  $z_{i,k} \cap z_{i,h} = \emptyset$ .

### 9.3 Non-preemptive protocol (NPP)

Una semplice soluzione per evitare l'inversione di priorità è di disabilitare la preemption durante l'esecuzione di una qualsiasi sezione critica. Questo può essere implementato semplicemente aumentando al massimo la priorità di un task quando entra in una sezione critica.

### 9.4 Highest locker priority (HLP)

Migliora NPP aumentando la priorità di un task che utilizza una risorsa  $R_k$  alla priorità più alta tra i task che condividono quella risorsa. La priorità dinamica viene cambiata quando entra la sezione critica e viene ripristinata a  $P_i$  quando il task esce dalla sezione critica. La computazione online della priorità dinamica può essere semplificata assegnando a ogni risorsa  $R_k$  un *priority ceiling*  $C(R_k)$ , che viene computato offline, uguale alla priorità più alta tra i task che condividono quella risorsa.

Pur migliorando NPP, contiene una sorgente di pessimismo che potrebbe produrre blocchi non necessari. Infatti, un task viene bloccato nel momento in cui prova a fare preemption, prima che voglia effettivamente utilizzare una risorsa. Se una sezione critica è contenuta in un solo ramo della computazione, allora il blocco potrebbe non essere necessario, dato che il task potrebbe non utilizzare la risorsa eseguendo un altro ramo.

### 9.5 Priority inheritance protocol (PIP)

Evita l'inversione di priorità senza limiti modificando la priorità dei task che causano il blocco. In particolare, quando un task  $\tau_i$  blocca uno o più task con priorità più alta, assume temporaneamente la priorità più alta tra i task che sono bloccati da esso.

I task vengono schedulati secondo le loro priorità attive, task con la stessa priorità attiva vengono schedulati in ordine di arrivo. Quando un task  $\tau_i$  cerca di entrare nella sezione critica  $z_{i,k}$  e la risorsa  $R_k$  è utilizzata da un task  $t_j$  con priorità più bassa, allora  $\tau_i$  viene bloccato. Quando il task  $t_j$  è bloccato da un semaforo, trasmette la sua priorità attiva al task  $\tau_i$  che lo ha bloccato. In generale un task assume la priorità più alta tra i task che sono bloccati da esso. Quando  $\tau_i$  esce dalla sezione critica sblocca il semaforo, e il task con priorità più alta che è bloccato da esso (se c'è) viene sbloccato.

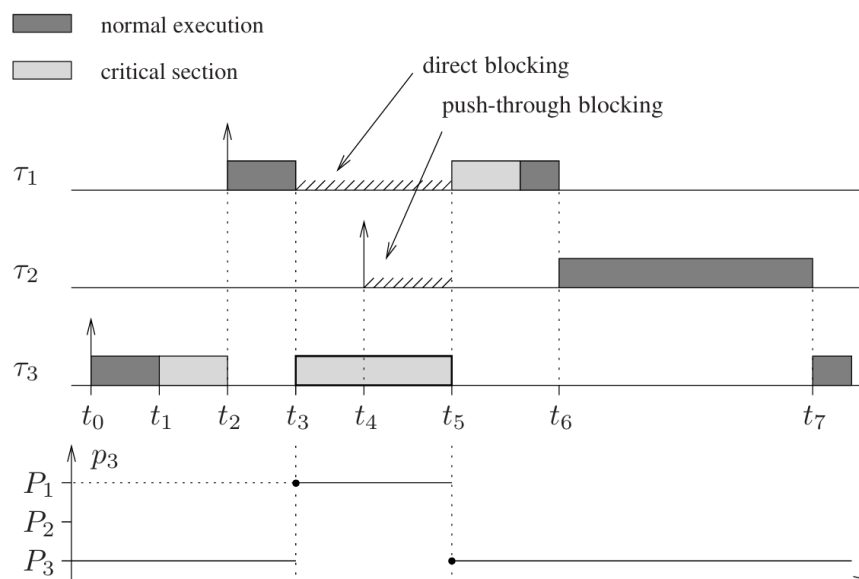


Figure 9.4: Esempio di priority inheritance

Dalla figura 9.4 possiamo notare che esistono due modi in cui un task a priorità più alta può essere bloccato:

- **Bloccaggio diretto:** quando un task con priorità più alta entra in una sezione critica e trova la risorsa occupata da un task con priorità più bassa. Necessario per garantire la consistenza della risorsa condivisa.
- **Bloccaggio push-through:** quando un task a priorità media è bloccato da un task con priorità più bassa che ha ereditato la priorità di un task con priorità più alta che ha bloccato. Necessario per evitare l'inversione di priorità.

Nonostante PIP limiti il fenomeno di inversione di priorità, la durata del blocco di un task può essere sostanziale a causa di un blocco a catena che si viene a formare. Un altro problema è la possibilità di avere un deadlock.

### 9.5.1 Blocco a catena

Consideriamo tre task  $\tau_1$ ,  $\tau_2$  e  $\tau_3$  con priorità decrescente, che condividono due semafori  $S_a$  e  $S_b$ . Supponiamo che  $\tau_1$  richieda di accedere sequenzialmente a  $S_a$  e  $S_b$ ,  $\tau_2$  acceda a  $S_b$  e  $\tau_3$  acceda a  $S_a$ . Inoltre consideriamo che  $\tau_3$  blocchi  $S_a$  e che sia preempted da  $\tau_2$  mentre si trova nella sezione critica.

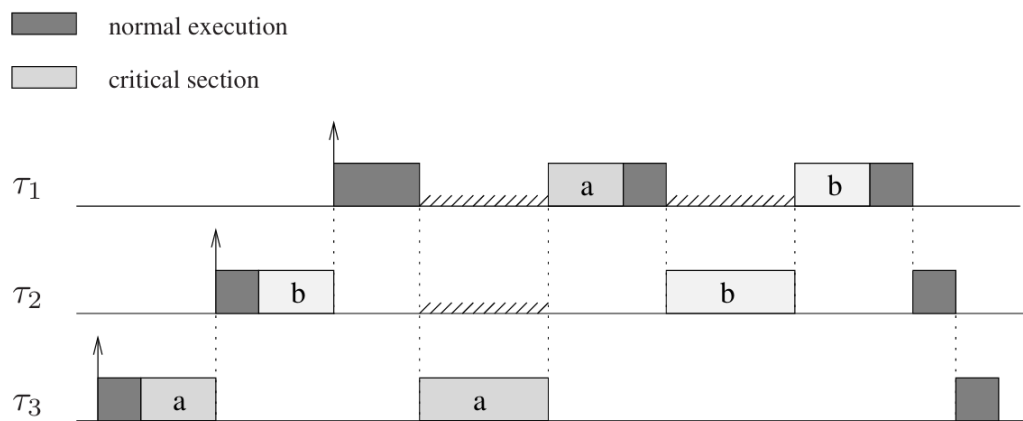


Figure 9.5: Esempio di blocco a catena

Come mostrato in figura 9.5, quando  $\tau_1$  prova a usare le sue risorse viene bloccato per un tempo uguale alla durata di due sezioni critiche, una volta aspetta che  $\tau_3$  rilasci  $S_a$  e una volta aspetta che  $\tau_2$  rilasci  $S_b$ . Nel caso peggiore, se  $\tau_1$  accede a  $n$  semafori distinti che sono stati bloccati da  $n$  task di priorità più bassa,  $\tau_1$  verrà bloccato per il tempo di  $n$  sezioni critiche.

## 9.5.2 Deadlock

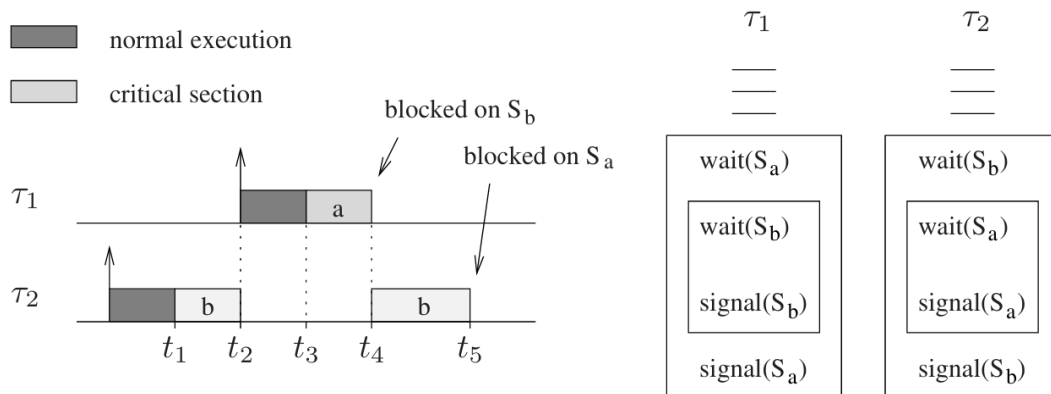


Figure 9.6: Esempio di deadlock

Considera due task che usano due semafori in maniera innestata, ma in ordine inverso, come raffigurato in figura 9.6. Ora supponi che al tempo  $t_1$  il task  $\tau_2$  blocchi il semaforo  $S_b$  e che entri nella sezione critica. Al tempo  $t_2$ ,  $\tau_1$  preempta  $\tau_2$  prima che possa bloccare  $S_a$ . Al tempo  $t_3$ ,  $\tau_1$  blocca  $S_a$ , che è libero, ma poi viene bloccato da  $S_b$  al tempo  $t_4$ . A questo punto  $\tau_2$  riprende l'esecuzione alla priorità di  $\tau_1$ . PIP non previene il deadlock, che avviene al tempo  $t_5$ , quando  $\tau_2$  prova a bloccare  $S_a$ . Nota che il deadlock non è causato da PIP in sé, ma da un errato utilizzo dei semafori.



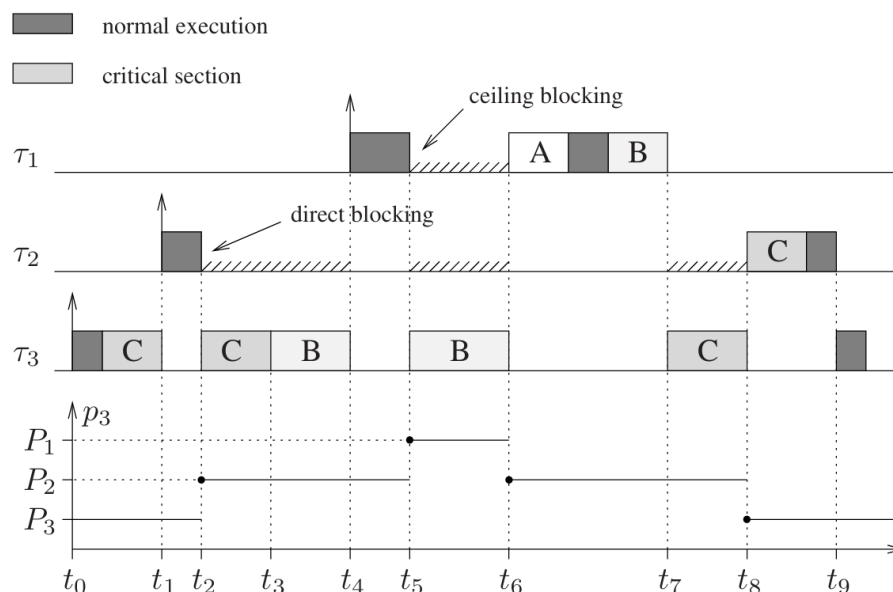


Figure 9.7: Esempio di priority ceiling protocol

## 9.6 Priority ceiling protocol (PCP)

Creato per evitare l'inversione di priorità, il blocco a catena e il deadlock. L'idea di base è di estendere PIP con una regola per garantire una richiesta di lock su un semaforo libero. Per evitare blocchi multipli, questa regola deve vietare che un task entri in una sezione critica se c'è un semaforo che potrebbe bloccarlo. Questo vuol dire che, una volta che entra nella sua prima sezione critica, un task non può mai essere bloccato da un task di priorità più bassa fino al suo completamento. Quindi l'idea in generale è di assegnare a ogni semaforo un *priority ceiling*, uguale alla priorità più alta tra i task che possono bloccare quel semaforo. Allora un task  $\tau_i$  può entrare in una sezione critica solo se la sua priorità è maggiore di tutti i priority ceiling dei semafori al momento bloccati da altri task oltre  $\tau_i$ . Da notare in figura 9.7 che viene aggiunta una terza forma di blocco, detto *ceiling blocking*, in aggiunta a quello diretto e push-through causati da PIP. Necessario a evitare il deadlock e il blocco a catena.

## 9.7 Stack resource policy (SRP)

Estensione di PCP, i tre punti essenziali sono:

1. Permette l'uso di risorse multi-unità (?)
2. Supporta lo scheduling a priorità dinamica.
3. Permette la condivisione di risorse basate sullo stack a runtime (??)

La differenza sostanziale dal punto di vista dello scheduling tra SRP e PCP sta nel momento in cui un task viene bloccato. In PCP un task viene bloccato nel momento in cui richiede per la prima volta una risorsa, in SRP un task è bloccato nel momento in cui prova a fare preemption. Questo blocco prematuro riduce leggermente la concorrenza, ma riduce i cambi di contesto non necessari, semplificando l'implementazione.

Prima di procedere è necessario definire alcune notazioni.

### Priorità

A ogni task  $\tau_i$  viene assegnata una priorità  $p_i$ , che indica la sua importanza (ovvero l'urgenza) rispetto agli altri task del sistema. La priorità di un task può essere statica (fissa) o dinamica (cambia durante l'esecuzione). A ogni momento  $t$ ,  $p_a > p_b$  indica che l'esecuzione di  $\tau_a$  è più urgente di quella di  $\tau_b$ : dunque  $\tau_b$  può essere ritardato in favore di  $\tau_a$ .

### Livello di preemption

Un task è caratterizzato anche da un livello di preemption  $\pi_i$ . Il livello di preemption è statico e viene assegnato al momento della creazione del task, viene associato a ogni istanza di quel task. Un task  $\tau_a$  può preemptare un task  $\tau_b$  se e solo se  $\pi_a > \pi_b$ . Questo è vero anche considerando le priorità.

In generale, se  $\tau_a$  arriva dopo  $\tau_b$  e  $\tau_a$  ha una priorità più alta di  $\tau_b$ , allora  $\tau_a$  deve avere un livello di preemption più alto di  $\tau_b$ .

Con EDF, la condizione precedente è soddisfatta se i livelli di preemption sono ordinati in maniera inversa rispetto alle deadline relative dei task, ovvero:

$$\pi_i > \pi_j \iff D_i < D_j$$

### Resource units

Ogni risorsa  $R_k$  può avere  $N_k$  unità che possono essere utilizzate in maniera concorrente da più task. In generale  $n_k$  denota il numero di unità al momento disponibili di  $R_k$ , quindi  $N_k - n_k$  risorse sono bloccate.

### Resource requirements

Quando si entra nella sezione critica  $z_{i,k}$ , protetta dal semaforo  $S_k$ , un task  $\tau_i$  deve specificare il numero di unità di cui necessita nella chiamata a `wait( $S_k, r$ )`. Quando esce dalla sezione critica, il task rilascia tutte le unità con `signal( $S_k$ )`.

Il massimo numero di unità che possono essere simultaneamente richiesta da  $\tau_i$  di  $R_k$  è denotato da  $\mu_i(R_k)$ , che può essere calcolato offline analizzando il codice del task.

### Resource ceiling

A ogni risorsa  $R_k$  viene associato un *resource ceiling*  $C_{Rk}(n_k)$ , uguale al massimo livello di preemption dei task che possono essere bloccati su  $R_k$  (assumendo la richiesta massima). Quindi è una funzione dinamica rispetto al numero di unità disponibili  $n_k$ .

### System ceiling

Il massimo dei ceiling delle risorse, ovvero:

$$\Pi_s = \max_k \{C_{Rk}\}$$

Nota che è un parametro dinamico che cambia ogni volta che una risorsa viene rilasciata o richiesta.

### 9.7.1 Definizione

Dunque l'idea chiave di SRP è che quando un task ha bisogno di una risorsa che non è disponibile esso venga bloccato al momento in cui prova a fare preemption. Inoltre, per prevenire l'inversione di priorità, un task non può iniziare fino a quando le risorse al momento disponibili sono sufficienti per rispettare la massima richiesta di ogni task che potrebbe farlo preemptare.

Possiamo effettuare il seguente test:

Un task non può fare preemption fino a che la sua priorità non è la più alta tra tutti i task che sono in stato di ready, e il suo livello di preemption è più alto del system ceiling.

Se la ready queue è ordinata in base a priorità decrescente, allora il preemption test può essere fatto semplicemente confrontando il livello di preemption  $\pi_i(\tau)$  del task ready con priorità più alta (quello in testa alla ready queue) con il system ceiling. Se  $\pi_i(\tau) > \Pi_s$  allora il task viene eseguito, altrimenti rimane nella ready queue fino a quando  $\Pi_s < \pi_i(\tau)$ . La condizione va testata ogni volta che  $\Pi_s$  diminuisce, ovvero quando una risorsa viene rilasciata.