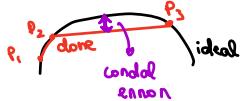


## Industrial Applications

- 1) Functional Requirements: funzionalità che l'app. deve offrire (cosa deve fare)

Condal Ennon: Ennone tra la "funzione svolta" e la "funzione ideale"



- 2) Non functional Requirements: Requisiti non implementabili da funzioni (proprietà che il sistema deve avere)

- a) Time : Execution Time of the Application

- Dead line: Tempo limite in cui l'esecuzione DEVE essere terminata

Non si deve verificare  
HAI

→ Hard Real time systems: Missing a Deadline può produrre effetti catastrofici (es: Airbag)

→ Soft Real time system: Missing a Deadline può non essere catastrofico

- b) Power Consumption : Tradeoff tra consumo e complessità d'implementazione

- Dissipazioni (cooling system): Necessario se richiesta molta energy consumption
- Sistemi di alimentazione complessi: Necessario se richiesta molta energy consumption

- c) Costo : Tradeoff tra costo e requisiti realtime dell'app.

- Risonse di computazione limitate
- Risonse di memorizzazione e I/O limitate

## Real-time Systems

I real-time systems sono computing systems che devono reagire entro precisi vincoli di tempo a eventi dell'environemnts.

Il loro comportamento dipende quindi non solo dal "valore" (risultato) della computazione ma anche dal tempo impiegato per produrla, questo perché una reazione in ritardo può essere inutile o addirittura dannosa.

In molti casi il realtime computer è embedded nel sistema che deve essere controllato,

sistemi embedded, cioè sistemi basati su microprocessori il cui software svolge funzionalità specifiche (dedicate).

### Caratteristiche Real-time system

Da quanto visto finora possiamo quindi escludere le seguenti caratteristiche:

- 1) REAL vuol dire che il sistema deve essere sincronizzato con il tempo dell'environment, di conseguenza REAL TIME non vuol dire FAST, la velocità del sistema è relativa a quella dello specifico environment e come esso evolve.  
Quindi, data una computer speed arbitraria, dobbiamo garantire che i timing constraints possano essere rispettati (TESTARE NON È SUFFICIENTE)
- 2) Un REAL-Time computing system deve essere prevedibile

### Traditional approach used to design Real-Time applications

la maggior parte delle RT applications sono disegnate usando tecniche empiriche, cioè si incarna osserva il fenomeno, sviluppa una soluzione e la testa. Nello specifico:

- 1) Programmazione assembly → Per rendere l'applicazione veloce (dovuto ai vincoli sul tempo)
- 2) Timing tramite Timers → Il tempo viene preso tramite appositi device HW
- 3) Control through driver Programming → Il flusso di controllo viene gestito tramite interrupts e drivers per la loro gestione.
- 4) Manipolazione della priorità in modo empirico, senza fondo matematico.

### Svantaggi di questo approccio:

- 1) Programmazione difficile dovuta all'assembly.
- 2) Difficoltà nel capire il codice per chi non l'ha sviluppato
- 3) Difficoltà nel mantenere il codice, capire il codice impiega più tempo che riscreverlo.  
Riscrivendolo è costoso e bug prone.

4) Difficoltà nel verificare i vincoli di tempo senza supporti dell'os e del linguaggio.

a) Il codice può essere imprevedibile

b) Il codice può apparentemente funzionare bene per un periodo, ma andare in tilt in alcune rare situazioni.

È difficile inolte capire perché è andato in fail il sistema.

OSS: Abbiamo quindi bisogno di un sistema che sia verificato apriori by hypothesis, invece che essere testato successivamente!

Questo perché i tests possono fornire solo una verifica parziale della software behaviour, relativa solo a un subset di dati forniti in input.

Non devo quindi scrivere il codice e poi verificare i timing constraints,

ma fare prima il design del sistema in modo che i real time constraints

siano rispettati e successivamente scrivere il codice.

Il sistema deve essere design under pessimistic Assumptions.

### Real-Time Tasks

Un Real-time task è caratterizzato da una deadline (massimo tempo per completare l'esecuzione). Un risultato prodotto dopo la deadline è SBAGLIATO.

#### Classificazione dei real-time tasks

wrong system behaviour

1) Hard: La produzione di un risultato dopo la deadline può causare effetti catastrofici sul sistema. → es: Sensory data Acquisition

Prima dell'esecuzione di questi task devono esserci meccanismi che assicurino che il task rispetti la deadline (Guaranteed offline)

2) Firm: La produzione di un risultato dopo la deadline è inutile per il sistema.

Se si nota che il task non rispetta la deadline vengono abbandonati per risparmiare risorse (Guaranteed online) → es: Video playing

3) Soft: La produzione di un risultato dopo la deadline ha qualche utilità per il sistema, anche se le performance sono degradate. → es: Graphical Activities  
Questi task dovrebbero essere gestiti per finire nel minore tempo possibile.

### RTOS - Real Time Operating system

RTOS è un OS responsabile di:

- 1) Managing concurrency
- 2) Time management
- 3) Scheduling
- 4) Mutual exclusion
- 5) Interrupt handling

Oltre a ciò: Implementa un meccanismo per garantire la predictable execution dei Task con real-time constraints

Quindi: Può fornire un supporto software predictabile per il design e implement delle real-time application.

### Predictability of a system

Una delle proprietà più importanti delle real-time systems è la predictability. Il sistema dovrebbe essere in grado di prevedere l'evoluzione del task e garantire in anticipo che tutti i critical timing constraints vengano rispettati.

Per fare ciò posso considerare il WORST CASE EXECUTION TIME (WCET), che se è minore della deadline posso schedularne il task.

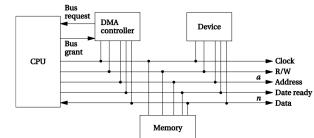
PROBLEMA: NON-DETERMINISMO

↓  
Dobbiamo gestire tutte quelle architectural features (HW/processor) e meccanismi/politiche adottate dal kernel, che causano non determinismo.

## Direct Memory Access

è una tecnica usata dalle periferiche per trasferire dati tra il device e la main memory.

CPU e DMA usano lo stesso bus.



PROBLEMA: Se la CPU e DMA richiedono un memory cycle (accesso alla mem.) nello stesso momento, il bus viene assegnato alla DMA e la CPU aspetta fin quando il DMA cycle non è completato.

→ Non c'è modo di prevedere quanto la CPU dovrà aspettare, quindi abbiamo un comportamento NON-DETERMINISTICO.

### 1) Time-slice Method

Ogni memory cycle è suddiviso in due time-slots :

a) Uno riservato alla CPU

b) Uno riservato al DMA device

redoppiato

BSS: Il costo per l'accesso in memoria è maggiore, ma non ha conflitti tra CPU e DMA → Determinismo.

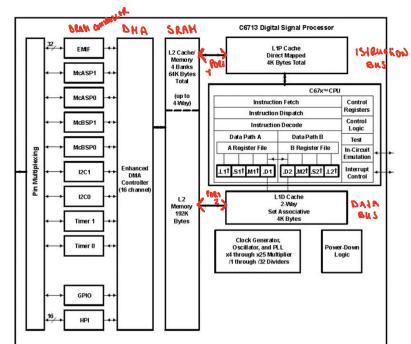
### 2) Multi-bus and dual port memory

La memoria viene "equipaggiata" con due ponti, in modo da poter supportare più accessi se non effettuali allo stesso bank o location.

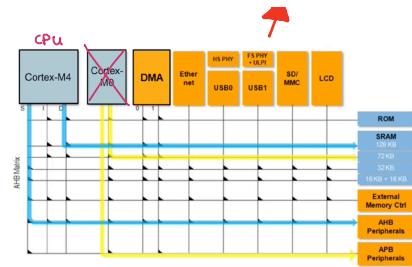
Vengono usati bus multipli.

Nel caso di conflitti sullo stesso bank/location la CPU deve aspettare che la DMA finisce per "leggere" i dati aggiornati, ma è comunque deterministico perché posso considerare questa attesa nella programmazione

considerando un limite superiore per questa attesa.



Matrix bus  
in ALU



Supponendo però che le variabili siano diverse per CPU e DMA,

se il linker è istruito a fare mapping delle variabili di

modo da non avere conflitto tra i due accessi (CPU e DMA),

allora non c'è questo problema. (sposta le variabili della CPU in un altro banco di memoria)

Ovviamente se il dato da accedere è lo stesso, la CPU deve comunque aspettare.

### 3) Dual Buffer



Ci sono due buffer, il DMA scrive "la prima volta" e la CPU aspetta per leggere. Dopo che la CPU sta leggendo, la DMA può scrivere sul secondo buffer in parallelo. Una volta finito, la CPU legge sul secondo buffer e la DMA scrive sul primo, e così via ...

### Cache

La cache è una piccola memoria veloce che contiene le copie di alcuni contenuti della main memory, velocizzando l'accesso a essi.

Problema: Se abbiamo un hit nella cache, l'accesso è veloce, se abbiamo un miss invece, l'accesso è lento (perché richiede l'accesso alla memoria principale)

Non-Deterministico

### Soluzioni possibili:

1) Disabilitare la cache o scegliere sistemi senza cache.

2) Il tempo di esecuzione di una istruzione (che usa la cache) viene aumentato di un fattore costante.

→ Problema: Non so se ho visto il tempo di esecuzione nel caso peggiore.

3) Viene considerato come  $\text{Worst execution time}$  il tempo come se ci fosse sempre un miss, cioè l'esecuzione senza cache.

4) Altre soluzioni più complesse tramite software tool:

Ho un program flow e una cache, sto per ogni istruzione quali hanno un miss e quali un hit.

A seconda dei path ho diversi miss e hit, li conto e più o meno capisco quant'è generata il programma per stimare il WCET.

Ovviamente, se cambiano le componenti del sistema, cache e cpu frequency devono rimanere costanti!

OSS: La soluzione 1 è predictable (perché si accede sempre in memoria), mentre nelle soluzioni 2,3,4 si cerca di sfruttare la cache (che in alcuni sistemi è necessaria, ad esempio self driving cars) e stabilire un safe margin nella deadline.

Soltanente, se riesco a rispettare i timing constraints senza cache allora posso non usarla (1), altrimenti devo usare sistemi non del tutto deterministici (2,3,4).

### Interruptions

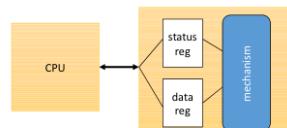
L'interrupt mechanism è utilizzato per gestire efficientemente operazioni di I/O.

Gli external interrupts permettono a un device di segnalare la CPU e forzare l'esecuzione di un particolare pezzo di codice (device driver, chiamato anche interrupt handler).

Problema: Essendo in generale difficile dare un limite superiore al numero di interrupts che un task potrebbe incontrare, il delay introdotto è imprevedibile.

→ Non-deterministico

Device :



OSS:

- Gli interrupt introducono delay, non influenzano l'execution time del task!
- Gli interrupt hanno priorità maggiore dei tasks → Task vengono preempted
- L'unico modo per non avere questo comportamento è disabilitarli.

**Soluzioni:**

1) **Managing I/O via Polling**

- a) Tutti gli **external interrupts** vengono **disabilitati** (tranne il timer, necessario per l'uso)
- b) Gli **peripheral devices** vengono **gestiti** dalle **application tasks**
  - Hanno accesso ai **registri** delle **interfacing boards** (registri **del device**)
  - Il **data transfer** viene fatto **tramite polling** (finché i dati non sono pronti, viene fatta una **busy active wait** da parte dell'application task).  
La "prontezza" dei dati viene segnalata dallo **status register** del device.

**Vantaggi:**

- **I/O interrupt eliminati**
- **tempo per i trasf. dei dati valutato precisamente** e a **causa** del application task.
- **il kernel non deve essere modificato anche se cambiamo gli I/O devices**

**Svantaggi:**

- **Busy active wait**  $\Rightarrow$  low processon efficiency
- Application task devono conoscere i **low level details** del device con cui interagiscono
  - ↳ risolvibile incapsulando le **device-dependent routines** in librerie usate dall'app.

2) **Polling in the Kernel**

- a) Tutti gli **external interrupts** vengono **disabilitati** (tranne il timer, necessario per l'uso)
- b) Gli **device** vengono **gestiti** da **kernel routines dedicate**, attivate periodicamente dal timer.

Questo ci permette di calcolare il computational load delle **kernel routines** una sola volta.

**Vantaggi:**

- **Delay limitato**
- le **kernel routines** possono essere incapsulate nelle **kernel procedures**, senza che debbano

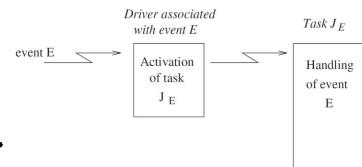
essere conosciute dall' application tasks

Svantaggi:

- Busy wait
- Overhead dovuto alla comunicazione tra le application tasks e le I/O kernel routines per lo scambio degli I/O data.
- Il kernel deve essere modificato quando un device viene rimpiazzato / aggiunto.

### 3) ISR with a server task

- a) Gli interrupt attivi.
- b) L'obiettivo dei vari drivers (interrupt handlers) è di attivare un task opportuno che si occupi della device management.  
Questo task è eseguito sotto il controllo dell'OS ed è schedulato come ogni altro application task.
- c) Un task di controllo in questo modo può avere una priorità maggiore del device handling task, ed essere eseguito in modo deterministico.  
Cioè i device handling task hanno una priorità "non normale".



Vantaggi:

- Busy wait eliminate durante l'I/O
- Delays introdotti dai drivers ridotti → Più predictability

Svantaggi:

- Un piccolo overhead non limitato dovuto all'exec di small drivers rimane nel sistema.  
Dovrebbero essere presi in considerazione nel guarantee mechanism.

### System Calls

La predictability di un sistema dipende anche da come sono impl. le system calls.

- Bisogna valutare il worst-case execution time di ognuna e caratterizzarla con un tempo

### bounded execution Time.

- Le system call dovrebbero essere preemptable (pre-nelasciabili) in modo da non ritardare l'attivazione / esecuzione di attività critiche che possono causare deadline miss.

### Semafoni

I semafoni introducono delay non deterministici.

Alcuni protocolli specifici possono essere usati per limitare il maximum blocking time dei task che condividono risorse critiche.

Questi protocolli possono richiedere modifiche al kernel (wait, signal calls, data structures, task management)

### Memory Management

I demand paging schemes (portare le pagine in memoria, page faults ecc...) non sono adatti per le real-time appl. perché introducono del delay.

Le soluzioni più usate sono la memory segmentation o static partitioning con un memory management scheme fisso (cioè non ho le pagine virtuali)

In RTOS:

- Viene allocata una parte di memoria ad ogni task/thread per stack e dati e se esso accede a una partizione che non gli "appartiene", viene lanciata un'eccezione.
- Per allocare memoria (memory pool) vengono usate delle primitive con tempi predictable.

Sono di grandezza fissata

### Programming

Alcuni linguaggi possono avere impatto sulla prevedibilità di un sistema, ad esempio linguaggi che permettono l'uso di: strutture dinamiche, loop non limitati, ricorsione.

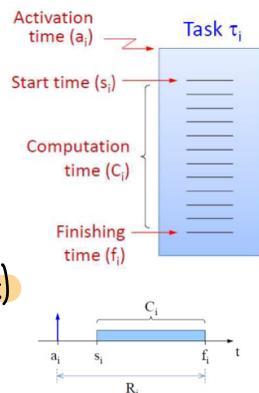
Real-time concurrent C è un linguaggio che fornisce facilitazioni per specificare i vincoli di periodicità e deadline, per garantire che i timing constraints siano rispettati e per poter svolgere azioni alternative nel caso non lo fossero.

## Task Definition and Characteristics

Task = Sequenza di istruzioni eseguite dal processore fino a completamento.

→ È composto da:

- 1) Activation time ( $a_i$ ): Quando il task viene attivato
- 2) Start Time ( $s_i$ ): Quando inizia l'esecuzione del task
- 3) Finishing Time ( $f_i$ ): Quando termina l'esecuzione del task
- 4) Computation time ( $C_i$ ): Tempo totale per l'esec. del task (netto, senza interruzioni)
 
$$f_i - s_i$$
- 5) Response time ( $R_i$ ): Intervallo tra la fine e l'attivazione del task.
 
$$f_i - a_i$$



## Task states

Un Task attivo può assumere tre differenti stati:

- 1) Ready: Sono task attivi che aspettano di essere messi in esecuzione

→ task che sono nello stato ready vengono contenuti in una ready queue, gestita da una scheduling policy.

Tramite l'operazione di dispatching, il processore viene assegnato al primo task della ready queue.

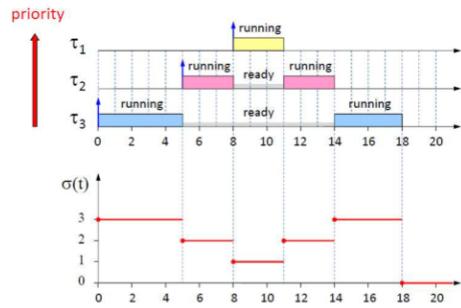
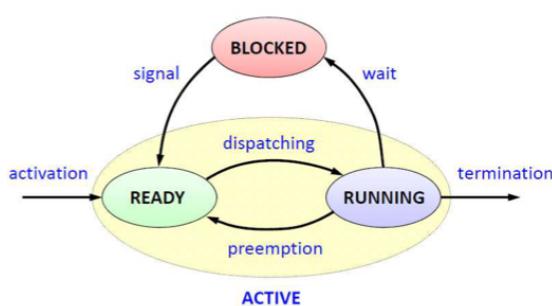
OSS: Possono esserci più ready queue, una per ogni tipologia di tasks.

- 2) Running: Sono task che hanno assegnato il processore e sono attualmente in esecuzione.

Tramite l'operazione di preemption è possibile sospendere l'esecuzione del running task in favore di un task più importante.

Il task sospeso torna in ready queue.

3) **Blocked**: Sono Task che sono stati bloccati da una wait operation e aspettano un segnale (signal operation) per essere sbloccati ed andare in ready queue.

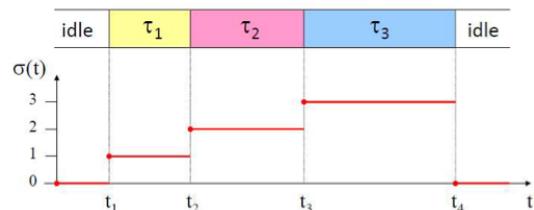


### Task scheduling

La schedule (ottenuta dallo scheduling) è un assegnamento particolare dei task al processore che determina la sequenza di esecuzione dei task.

Formalmente, dato un insieme di task  $T = \{t_1, \dots, t_n\}$ , una schedule è una funzione che associa un intero  $k$  ad ogni intervallo di tempo  $[t_i, t_{i+1}]$ , t.c.:

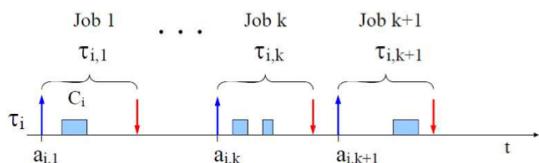
- $K = 0 \Rightarrow$  Nell'intervallo  $[t_i, t_{i+1}]$  il processore è idle
- $K > 0 \Rightarrow$  Nell'intervallo  $[t_i, t_{i+1}]$  il processore esegue  $t_k$  (task)



- Ogni intervallo  $[t_i, t_{i+1}]$  è chiamato time-slice
- Un  $t_i, t_2, t_3, t_4$  abbiamo un context switch

### Task's Jobs

Un task eseguito più volte su differenti dati di input genera una sequenza di istanze (Jobs).

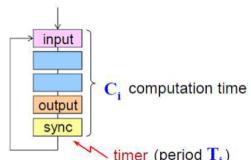


## Task activation mode

Le task possono essere attivate in due modi:

- 1) Time driven (Periodic Tasks)

Sono task attivati dal sistema operativo in modo periodico (con rate costante).



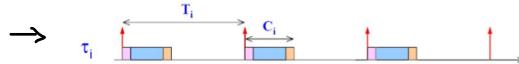
$$U_i = \frac{C_i}{T_i}$$

Utilization Factor

Se  $U > 1$  non posso schedularlo il task perché il computation time è maggiore del periodo.

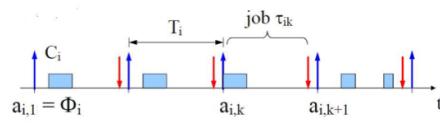
Mi dà una misura del load della cpu

Un task periodico genera infiniti jobs  $\rightarrow$



Un task periodico è caratterizzato da:

- Phase ( $\Phi_i$ ) : activation time della prima istanza
- Computation time ( $C_i$ )
- Period ( $T_i$ )
- Deadline ( $D_i$ ) - caso RT tasks



$$\begin{aligned} a_{i,1} &= \Phi_i \\ d_{i,k} &= a_{i,k} + D_i \end{aligned}$$

Often  $D_i = T_i$

Deve finire l'esecuzione del task prima della sua prossima attivazione

## Jitter

Il jitter è una misura della time variation di un evento periodico,

cioè di quanto è stata ritardata l'attivazione di un evento periodico:



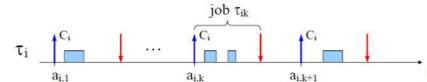
- Absolute:  $\max_k (t_k - a_k) - \min_k (t_k - a_k)$
- Relative:  $\max_k |(t_k - a_k) - (t_{k-1} - a_{k-1})|$

- 2) Event Driven (Aperiodic tasks)

Un task viene attivato quando arriva un evento (da un interrupt o un altro task tramite system call)

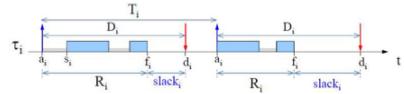
• Aperiodic:  $a_{i,k+1} \geq a_{i,k}$

Ci interessa solo che l'activation time dell'istante successivo è maggiore della precedente



## Panoramica Summary

Le parametri visibili finora possono essere di due tipi:



1) Known offline (specificati dal programmatore)

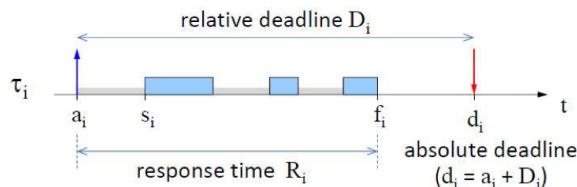
- Computation time ( $C_i$ )
- Period ( $T_i$ )
- Relative deadline ( $D_i$ )
- Arrival time ( $a_i$ )

2) Known at run-time (dipendono dallo scheduler e dall'esecuzione attuale)

- Start time ( $s_i$ )
- Finishing time ( $f_i$ )
- Response time ( $R_i$ )
- Slack and Lateness
- Jitter

## Real-Time Tasks

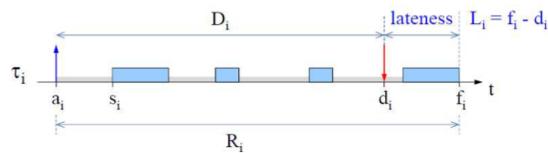
Sono tasks caratterizzati da un timing constraint (sul response time  $R_i$ ), chiamato deadline, il quale rappresenta il tempo massimo in cui un task debba essere terminata.



Feasible RT-task: Task la cui esecuzione finisce sicuramente prima della deadline ( $f_i \leq d_i$  o equiv.  $R_i \leq D_i$ )

Latency: Differenza di tempo tra il completamento del task e la sua deadline.

È negativa se il task finisce prima della sua deadline.



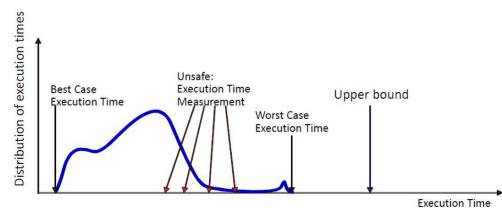
OSS: Se dimostra che latenza < 0, allora posso dire che un task è schedutabile.

**Slack Time**: Tempo massimo di ritardo sull'attivazione di un task per far sì che venga completato entro la sua deadline.

$$\text{SLACK}_i = d_i - a_i - C_i$$

Nei real-time task vogliamo assicurarsi che il tempo necessario al processore per eseguire il task  $t_i$  (Computation time -  $C_i$ ) sia minore della deadline per tutti i possibili casi.

Il computation time può variare ad ogni esecuzione, dovuto al path eseguito, cache hit/miss, operando usato (int, float, long...) ecc...



L'avg computation time è importante perché mi fa capire se un programma è efficiente.

Noi però siamo interessati al worst-case execution time (WCET), cioè il tempo di esecuzione più lungo data una qualsiasi input sequence.

### Tecniche per determinare il task execution time

Per fare una stima accurata di  $C_i$ , il task code e la CPU architecture devono essere conosciute.

Abbiamo visto due tecniche:

- By Analysis (Formal o semi-formal way e con l'utilizzo di tools)
  - Limita i cicli loop
  - Computa il path più lungo (che genera il maggior  $C_i$ )
  - Limita il numero di cache miss
  - Calcola il computation time per ogni istn. nel longest path.

Il programma non viene eseguito, ma viene fatta un'analisi del control flow diagram per capire il computation time.

OSS: Buono per simple CPU (composable CPU), nelle quali il delay è accumulato a run-time.

Non funziona nelle CPU più complesse perché:

- 1) Gli hit potrebbero avere effetti più rischiosi dei miss
- 2) I meccanismi come "out-of-order execution" (le ist. non vengono eseguite sequentialmente come vengono ricevute, ma in ordine diverso in modo da sfruttare la parallelizzazione) nascondono gli effetti dei miss e non li rendono visibili all'execution time.

OSS: Quando posso utilizzare l'analisi conviene perché otengo sempre un upper bound al WCET e un lower bound al best case execution time.

## 2) By Measurements

- a) Esegui il task più volte usando differenti dati di input
- b) Collezione le statistiche sull'execution times ( $c_i$ )

Non garantisce un upper bound a tutte le possibili esecuzioni tranne se non faccio un'esecuzione esaustiva con tutti i possibili initial system e inputs possibili. L'esecuzione esaustiva in genere non è possibile per via della grandezza dell'input space e initial states space.

OSS: Con il measurement posso ottenere il worst case execution time ma anche un valore minore che non mi fa da upper bound!

## Come sceglie il WCET

- 1) Task hand  $\Rightarrow$  Safe choice

Trovare il bound con l'analisi (scelta migliore per task veramente rischioso) oppure prendo il  $c_i^{\max}$  misurato e ci aggiungo un "tot" sperando che sia safe.

## 2) Soft-Firm tasks

Dei deadline miss sono tollerati, quindi posso scegliere  $c_i \gg c_i^{\text{avg}}$  essendo che migliora l'efficienza del sistema.

### Ideal e real periodic task execution

In una esecuzione ideale, chiamata zero execution time (ZET), non abbiamo delay

o Jitter, cioè l'esecuzione del task è istantanea e il ritardo dell'attivazione nullo.

Detto diversamente, in e out accadono instantaneamente all'inizio del periodo.)



Nelle reali esecuzioni però abbiamo sia il delay che il jitter.



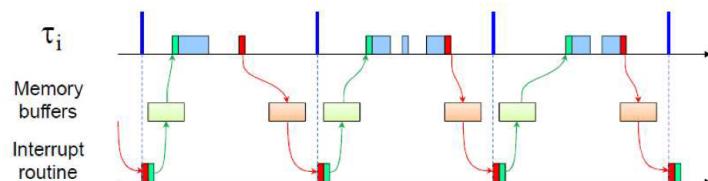
### Solution to deal with jitter

Per gestire il jitter usiamo il logical execution time (LET) model, nel quale in e out accadono simultaneamente alla fine di un periodo.



Viene usata una service routine che esegue l'input e output, ad ogni periodo  $T_i$ , usando due buffers (input e output buffers).

- La service routine acquisisce l'input e lo salva nell'input buffer per il task
- Il task scrive l'output nel output buffer per la routine
- Viene usato un delay costante  $T_i$  tra l'input e l'output



Non ho jitter perché l'input e output viene gestito dalla service routine, inoltre ho un delay costante "di un periodo"  $T_i$ .

Quindi non ho jitter e ho costant delay indipendentemente da come il task è schedulato.

Quindi posso tenere in considerazione solo il delay costante!

OSS: Ho sempre un input nel sistema, ne viene generato almeno uno.

Perché i periodic task sono utili?

Perché ci permettono di fare operazioni di controllo e calcoli periodici, ma evitando un polling continuo, cioè permettendo altre operazioni.

## Task constraints

Ci sono tre tipi di task constraints:

- 1) Timing constraints : activation, period, completion, jitter
- 2) Precedence constraints : impongono un ordine nell'esecuzione
- 3) Resource constraints : impongono la sincronizzazione nell'accesso mutualmente esclusivo delle risorse.

- Examples
  - open the valve in 10 seconds
  - send the position within 40 ms
  - Read the altimeter every 200 ms
  - Read the temperature every .5 seconds
  - acquire image from the camera every 20 ms

### Timing constraints

Possono essere esplicativi o impliciti.

- 1) Esplicativi: Sono inclusi nelle system specifications (descrizione del sistema)
- 2) Impliciti: Non compaiono nelle system specifications ma devono soddisfare i system requirements (non sono stati specificati esplicitamente ma "esistono")

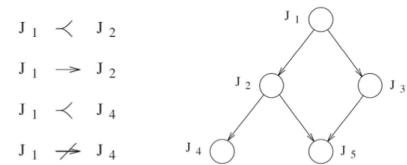
OSS: Sono derivati dal "sistema fisico"

### Precedence constraints

In alcune applicazioni le attività non possono essere svolte in ordine arbitrario ma devono rispettare alcune relazioni di precedenza.

Queste relazioni vengono solitamente descritte tramite un **Directed Acyclic Graph (DAG)**, nel quale i task sono rapp. dai nodi e le precedenze dalle frecce.

- The notation  $J_a \prec J_b$  specifies that task  $J_a$  is a *predecessor of task  $J_b$*   
•  $G$  contains a directed path from node  $J_a$  to node  $J_b$
- The notation  $J_a \rightarrow J_b$  specifies that task  $J_a$  is an *immediate predecessor of  $J_b$*   
•  $G$  contains an arc directed from node  $J_a$  to node  $J_b$
- Tasks with no predecessors are called *beginning tasks*
- Tasks with no successors, are called *ending tasks*



OSS: Derivati i task dalle specifiche, i precedenze constraints sono derivati dalla natura dell'applicazione.

### Resource Constraints

Una risorsa è definita come una qualsiasi software structure che può essere usata da un processo per avviare la sua esecuzione (data structures, variabili, main memory ...)

Esistono due tipi di risorse:

- Private: risorsa dedicata a un processo in particolare
- Shared: risorsa usata da più processi

Per mantenere la consistenza, molte risorse permettono esclusivamente l'accesso mutualmente esclusivo ed il codice che gestisce questo accesso è chiamata "sezione critica".

Per gestire l'accesso, l'OS fornisce un synchronization mechanism.

Resource constraints: Due o più task condividono risorse che vengono accedute tramite synchronization.

Come meccanismo di sincronizzazione vengono usati i semafoni (gestiti con le operazioni di wait e signal) i quali però introducono delay extra in high priority tasks.

## Scheduling Anomalies

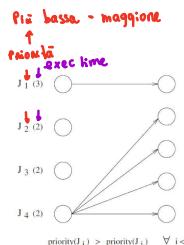
Le scheduling Anomalies sono schedules con comportamenti controintuitivi ottenuti dallo scheduler. Task set particolari (con priorità o vincoli di precedenza) in condizioni differenti (aumentando il n° di CPU, indebolendo i vincoli di precedenza...).

**Teorema (Graham):** Se un task set è schedulato su un multi-processore con priority assignment, numero fissato di processori, tempi di esecuzione fissati e vincoli di precedenza, allora aumentare il numero di processori, ridurre i tempi di esecuzione o indebolire i vincoli di precedenza può aumentare la durata della schedule.

Il teorema ci dice che introdurre possibili miglioramenti funzionanti in un sistema general purpose spesso non migliora le performance nei sistemi real-time (addirittura potrebbe peggiorarne).

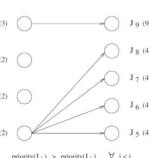
### Test case

- The execution is on a parallel machine
- The highest priority task is assigned to the first available processor
- The tasks have precedence constraints
- Computation times are indicated in parentheses



### a) Increased processors

- N=4
  - Tc=15
- Aumentando il n° di processori non otengo un risultato migliore!

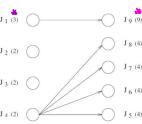


P1	J1	J8	J9	
P2	J2	J5		J9
P3	J3	J6		
P4	J4	J7		

### Schedule of task set J on a 3 processor machine

- Tc=12
  - Global completion time
- Tc=12 è ottimale perché le precedenze controllate. J1 > J9 <math>\forall i < j</math>

P1	J1		J9	
P2	J2	J4	J5	J7
P3	J3		J6	J8

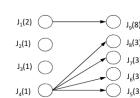


### b) Computation time reduced

- After some programming effort, the computation time is reduced by one unit

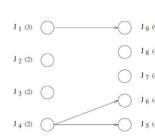
Riducendo l'exec time (ad es. usando una CPU migliore) non otengo un risultato migliore!

P1	J1	J5	J8	J9	
P2	J2	J4	J6		J9
P3	J3		J7		



### c) Released constraints

- Tc=16
- Anche riducendo i vincoli otengo un exec. time peggiore!



P1	J1	J8	J9	
P2	J2	J4	J5	J9
P3	J3	J7	J6	

### d) All constraints released

- T1: 3
- T2: 2
- T3: 2
- T4: 2
- T5: 4
- T6: 4
- T7: 4
- T8: 9
- T9: 4

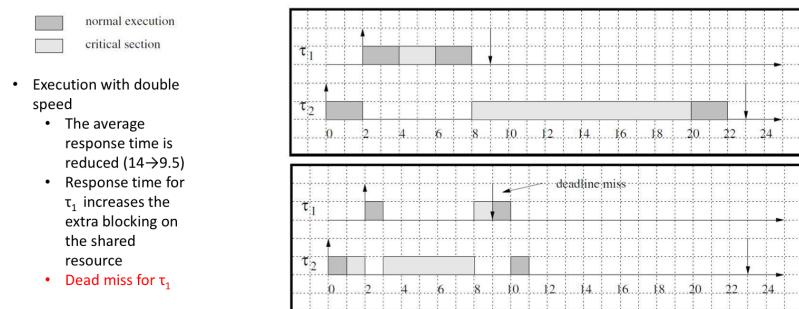
P1	T1	T6	T9	
P2	T2	T4	T7	T9
P3	T3	T5	T8	

t<sub>r</sub> = 16

lezione imparata: Una soluzione intuitiva spesso non porta ad un vantaggio, bisogna sempre valutare se effettivamente ha portato dei miglioramenti.

### Anomalies under Resource Constraints

Se dei task condividono risorse mutualmente esclusive, le scheduling anomalies possono avvenire quando  $\Rightarrow$  modifica la velocità del processore.



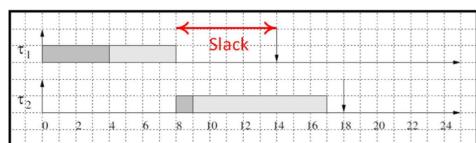
### Anomalies Using a delay primitive

Le Timing Anomalies possono avvenire quando i task che usano risorse condivise  $\Rightarrow$  sospendono esplicitamente tramite una system call, chiamata delay( $T$ ).

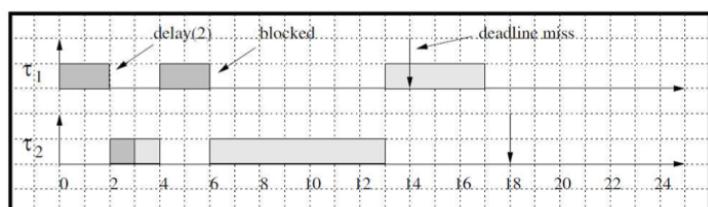
**Delay( $T$ )**: System call che sospende l'esecuzione del "calling task" per  $T$  unità di tempo.

DSS:

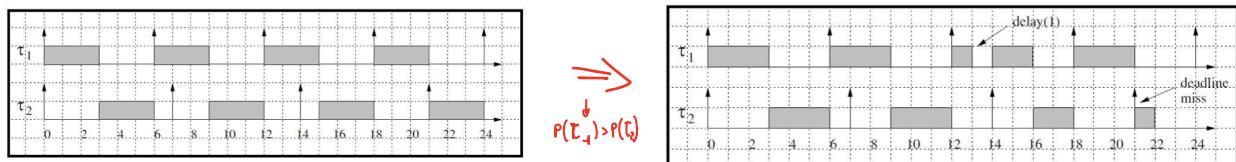
1) Un delay  $T$  può causare un delay maggiore di  $T$ .



- $\tau_1$  is feasible and has a slack time of 6 units when running at the highest priority
- It seems that it could tolerate a delay of two units.
- if  $\tau_1$  executes a  $\text{delay}(2)$  at time  $t = 2$ ,  $\tau_1$  misses its deadline



2) Un delay in un Task può aumentare il response Time di un altro task, anche senza condividere nessuna risorsa.

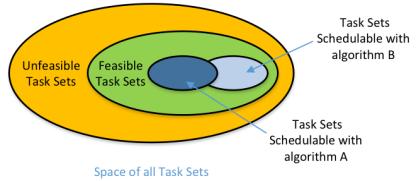


## Aperiodic Task scheduling

Lo scheduling, come visto precedentemente, indica come vengono assegnati i task ai vari processori.

### Feasible scheduling

Una schedule ( $\sigma$ ) è detta feasible se tutti i task possono essere completati in accordo con i constraints specificati.



### Feasible Task set

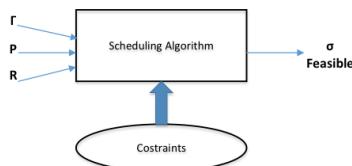
Un task set ( $\Gamma$ ) è detto feasible se esiste un algoritmo che genera una feasible schedule per  $\Gamma$ .

### Schedulable Task set

Un task set è detto schedulabile se esiste almeno un algoritmo che può produrre una feasible schedule per quel task set.

### Scheduling Problem

Dato un task set ( $\Gamma$ ) di  $n$  task, un insieme  $P$  di  $p$  processori, un insieme  $R$  di risorse e un insieme di constraints, trovare un assegnamento di  $P$  e  $R$  a  $\Gamma$  tale che produce un feasible schedule sotto l'insieme dei constraints.



Problema: Il problema descritto è NP hard, cioè il tempo per trovare un feasible schedule cresce esponenzialmente con il numero di tasks.

Per poter trovare un algoritmo in complessità polinomiale, dobbiamo fare delle ipotesi semplificate:

- Viene usato un singolo processore

↳ Importante per real-time system dinamici

Perché il task set è schedulato dinamicamente!

- Il task model è preemptive
- Non ci sono vincoli di precedenza
- Non ci sono resource constraints
- Simultaneous task activation
- Task set omogeneo (solo task periodici o solo task aperiodici)

Le assunzioni che vengono fatte tra queste, classificano i diversi tipi di scheduling algorithms.

### Classificazione degli scheduling Algorithms

- 1) Uniprocessor vs multiprocessor
- 2) Preemptive vs Non preemptive
  - a) Preemptive: Il task in exec può essere interrotto in qualsiasi momento per assegnare il processore a un altro task attivo.
  - b) Non-preemptive: Viceversa. Preemptive è più semplice ma il context switch causato da una preemption può causare ritardi dovuti ai cache miss.
- 3) Static vs Dynamic
  - a) Static: le scheduling decisions sono basate su parametri fissati assegnati staticamente ai task prima della loro attivazione.
  - b) Dynamic: le scheduling decisions sono basate su parametri dinamici che possono cambiare durante la system evolution.
- 4) Offline vs Online
  - a) Offline: Tutte le scheduling decisions vengono prese prima dell'attivazione del task. La schedula viene salvata in una tabella ed eseguita successivamente da un dispatcher.  
Oss: Precedence e resource constraints sono risolti offline
  - b) Online: Scheduling decisions vengono prese a runtime sul set di task attivi.

## 5) Optimal vs Heuristic

a) Optimal : Possiamo definire due tipi di ottimalità per gli scheduling algorithm

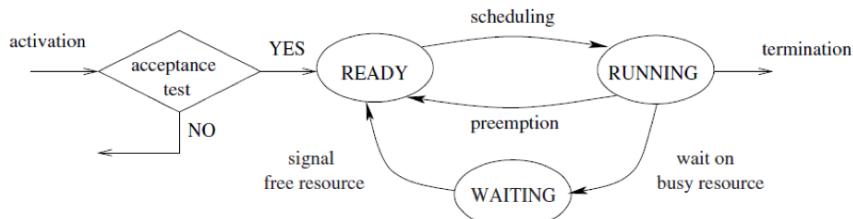
- Optimality according to a function , ad esempio minimizzare la latenza o il numero di deadline missate.
- Optimality according to feasibility, cioè che se esiste una feasible schedule ( $\sigma$ ) per un task set ( $\Gamma$ ), allora l'algoritmo lo troverà.

b) Heuristic : Un algoritmo heuristico è guidato da una funzione di costo heuristica che prova a soddisfare i criteri di ottimalità quando prende le sue scheduling decisions.  
Un algoritmo heuristico tende verso una optimal schedule, ma non garantisce di trovarla essendo che cerca di ridurre la complessità computazionale.

## Guarantee - Based systems

Nei sistemi real-time dinamici, i task vengono creati a runtime e c'è il bisogno di garantire, a real-time, che attivando il nuovo task, il nuovo task set sia ancora feasible (tramite un acceptance test).

Se il nuovo task set risulta non feasible, il nuovo task non viene accettato (to avoid a deadline miss).



## Graham's Notation

È una notatione introdotta da Graham per individuare i main behaviors di uno scheduling algorithm e classificarlo velocemente.

La notazione consiste in  $\alpha | \beta | \gamma$  dove:

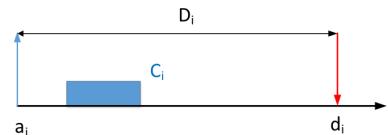
- $\alpha$  denota il numero di processori
- $\beta$  denota le caratteristiche dei task e risorse
- $\gamma$  denota i criteri di ottimalità che devono essere seguiti dalla schedula

- 1 | prec |  $L_{\max}$   
denotes a uniprocessor algorithm for a set of tasks with precedence constraints that minimizes the maximum lateness.  
• If no additional constraints are indicated in the second field, preemption is allowed at any time, and tasks can have arbitrary arrivals
- 2 | no-preem |  $R_{avg}$   
denotes a dual-core algorithm. Preemption is not allowed and the objective is to minimize the average response time.  
• Since no other constraints are indicated in the second field, tasks do not have precedence nor resource constraints but have arbitrary arrival times
- 2 | sync |  $\sum f_i$   
denotes a dual-core algorithm. Tasks have synchronous (the same) arrival times and do not have other constraints. The objective is to minimize the sum of the finishing times

## Algoritmi per il real-time Aperiodic task scheduling

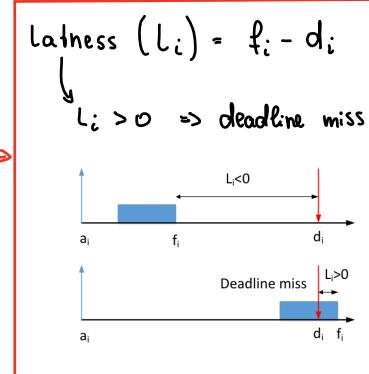
Questi algoritmi possono schedulare i tasks in relazione a:

- relative deadlines ( $D_i$  - statiche)
- absolute deadlines ( $d_i$  - dinamiche)



### i) Earliest Due Date (EDD - Jackson's algorithm)

- Problema:  $t | sync | L_{\max}$   
n°cpu      sync arrival time per task      minimizza maximum Lateness
- Algoritmo: Ordina la ready queue rispetto all' increasing deadline
- Assunzioni:



- Tutti i tasks costituiscono in un singolo job e hanno arrival time synchroni
- Tasks sono indipendenti (no precedence e resource constraints)
- Preemption not needed → perché sono sync, quindi arrivano nello stesso momento
- Static, cioè  $D_i$  fixed e scheduling decision prese offline

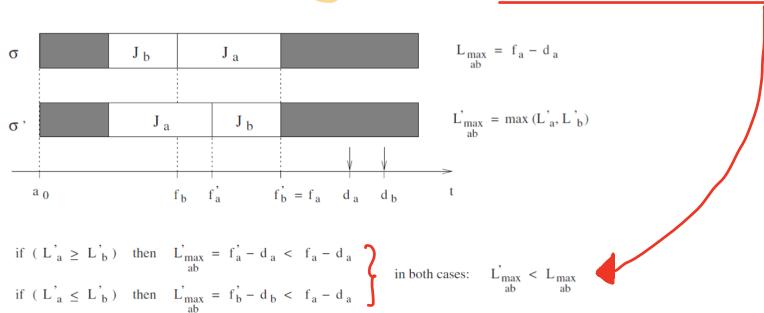
## Ottimalità di $\sigma_{EDD}$

Sia  $\sigma$  una schedule processata da un qualsiasi algoritmo.

Se  $\sigma \neq \sigma_{EDD}$ , vuol dire che esistono due task  $J_a$  e  $J_b$  (con  $d_a < d_b$ ) tale che  $J_b$  precede immediatamente  $J_a$  in  $\sigma$  (cioè  $J_a$  è schedulato dopo  $J_b$ )

Sia ora  $\sigma'$  una schedule ottenuta intercambiando  $J_a$  con  $J_b$  in  $\sigma$ , cioè tale che  $J_a$  precede immediatamente  $J_b$ .

l'intercambio  $J_a$  e  $J_b$  in  $\sigma$  non può aumentare la massima latenza del task set.



Otteniamo quindi che:

- Usando un numero finito di trasposizioni ( $\sigma \rightarrow \sigma'$ ),  $\sigma$  può essere trasformato in  $\sigma_{EDD}$
- Essendo che ad ogni trasposizione la latenza massima non può aumentare,  $\sigma_{EDD}$  è ottimo.

Attention: l'ottimalità di EDD garantisce che se una feasible schedule esiste per un task set, EDD la troverà. Inoltre, se un task set non è schedulabile con EDD esso non è schedulabile con nessun altro algoritmo.

Guarantee test (capire se un task set è schedulabile senza eseguire lo scheduling) - offline

Per garantire che un task set possa essere feasibly scheduled da EDD, dobbiamo dimostrare che nel caso peggiore tutti i task possono essere completati prima delle loro deadlines.

Siano i tasks ordinati by deadline crescenti:  $J_1, \dots, J_n$ . Dobbiamo quindi dimostrare

che per ogni task, il worst case finishing time  $f_i$  è minore o uguale alla sua deadline. ( $f_i \leq d_i$ )

$$f_i = \sum_{k=1}^i c_k \rightarrow \begin{array}{l} \text{Computation time dei task} \\ \text{che precedono } T_i, \text{ più se stesso} \end{array}$$

Quindi:

- Caso  $a_i = 0$ :  
activation time  
 $\sum_{k=1}^i c_k \leq d_i$
- Caso  $a_i \neq 0$ :  
 $\sum_{k=1}^i c_k \leq d_i$   
oss:  $a_i$  uguale per ogni task perché sono sincroni

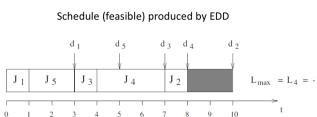
### EDD examples

- 5 tasks, simultaneously activated at time  $t = 0$

- $L_{\max}$  is equal to 1 and it is due to task  $J_4$

- Since the maximum lateness is negative, all tasks have been executed within their deadlines (the schedule is feasible).

Tasks parameters				
	$J_1$	$J_2$	$J_3$	$J_4$
$C_i$	1	1	1	3
$d_i$	3	10	7	8

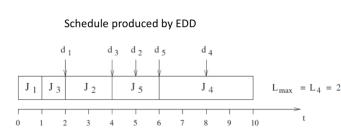


- 5 tasks, simultaneously activated at time  $t = 0$

- $L_{\max}$  is equal to 2 and it is due to task  $J_4$

- Since the maximum lateness is positive, the schedule is unfeasible.

Tasks parameters				
	$J_1$	$J_2$	$J_3$	$J_4$
$C_i$	1	2	1	4
$d_i$	2	5	4	8



## 2) Earliest Deadline First (EDF) - Horowitz's algorithm

- Problema: 1 | preem. |  $L_{\max}$   
n°cpu  
Preemption allowed  
minimize maximum Lateness
- Algorithm: Ordina la ready queue by absolute deadline crescente
- Assunzioni:
  - Tutti i tasks convivono in un singolo job e possono essere attivati dinamicamente (no sync)
  - Task sono indipendenti (no precedence e resource constraints)
  - Preemption allowed
  - Dynamic, cioè  $d_i$  dipende da  $a_i$  e scheduling decision prese online
- Proprietà: Minimizza la latenza massima ( $L_{\max}$ )

### Ottimalità di EDF

Prima di discutere l'ottimalità di EDF, analizziamo due proprietà degli algoritmi ottimali:

- ! [ • Se un task set non è schedulabile da un algo ottimale, allora non è schedulabile da nessun altro algoritmo]
- Se un algoritmo A minimizza  $L_{\max}$ , allora A è ottimo in termini di feasibility.  
Il contrario non è vero però! → Se A è ottimo in termini di feas.  $\not\Rightarrow$  min latenza massima ( $L_{\max}$ )

L'ottimalità di EDF (in termini di feasibility) può essere dimostrata mostrando che data una feasible schedule arbitraria, essa può essere trasformata in EDF senza incrementare la lateness.

Per fare ciò useremo la Dentoutos transformation.

### Dentoutos Transformation

- Assunzione:**
- Senza perdita di generalità, la schedule  $\sigma$  può essere divisa in time slices di una unità di tempo.
  - Un task viene eseguito per almeno un'unità di tempo.

! La Dentoutos transformation mostra che, con al più  $D$  trasposizioni (dove  $D$  è l'ultima deadline),  $\sigma_{EDF}$  può essere ottenuta da una feasible schedule  $\sigma$  prodotta da un algoritmo generico A in al più  $D$  trasposizioni.

**Oss:** Ogni trasposizione non può aumentare la latenza massima ( $L_{\max}$ )

### Algoritmo

Siano:

- $\sigma(t)$  identifica il task in esecuzione nello slice  $[t, t+1]$
- $E(t)$  identifica il ready task che a tempo  $t$  ha la earliest deadline
- $t_{E(t)}$  è il tempo ( $\geq t$ ) nel quale il prossimo slice del task  $E(t)$  inizia la sua esecuzione nella schedule corrente

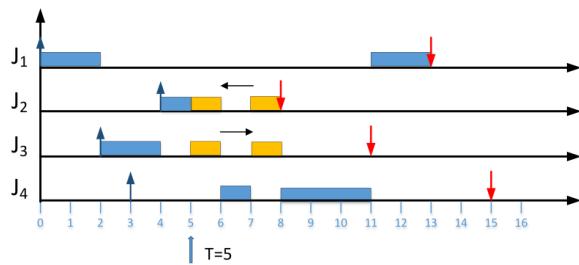
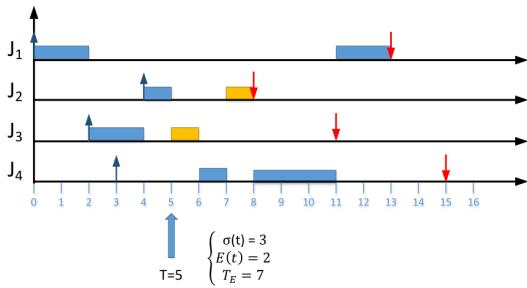
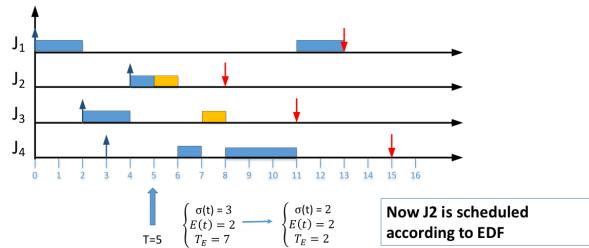
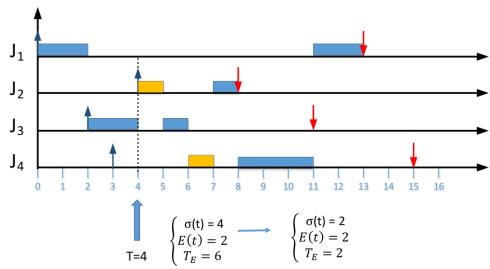
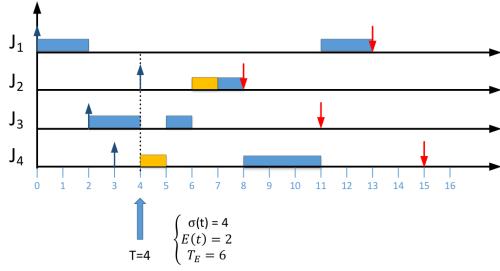
Abbiamo che se  $\sigma \neq \sigma_{EDF}$  allora in  $\sigma$  esiste un tempo  $t$  tale che  $\sigma(t) \neq E(t)$ , cioè il task scheduled non è quello con earliest deadline.

La Dentoutos transformation consiste nello spostare il task eseguito a quel timeslice  $t$  al timeslice  $t_{E(t)}$  ed eseguire il task con earliest deadline nel timeslice  $t$ .

```

for (t=0 to D-1) {
    if ( $\sigma(t) \neq E(t)$ ) {
        Task
        timeslice
        t
        ↑
        task con
        earliest
        deadline
         $\sigma(t_E) = \sigma(t);$ 
         $\sigma(t) = E(t);$ 
    }
}
Invece:
due tasks
  
```

$\sigma(t)$  = task executing at time t  
 $E(t)$  = task with min d at time t  
 $t_E$  = time at which E is executed



le trasformazioni preservano la schedulabilità

- Ovvio per i task anticipati perché anticipa task con deadline più vicina
- Il postponed task ha un deadline maggiore dell'anticipated task, quindi se era schedulabile prima, lo è anche dopo lo scambio.

Inoltre, con la stessa dimostrazione usata per EDD, questo scambio non porta un aumento di latenza, quindi l'algoritmo è ottimo in termini di feasibility e minima  $L_{max}$ .

### EDF Guarantee test

Nel caso di EDF il guarantee test deve essere fatto dinamicamente (online) man mano che nuovi task entrano nel sistema.

Sia  $J$  il set di active task corrente, e sia  $J_{new}$  un nuovo task, per accettare  $J_{new}$  nel sistema dobbiamo garantire che il nuovo task set  $J'$  sia schedulabile. Come in EDD, dobbiamo dimostrare che nel caso peggiore tutti i task in  $J'$  finiscono prima della deadline ( $d_i$ ).

Siano:

- I task in  $J'$  vengono ordinati per deadline crescente
- Essendo i task preemptable, quando  $J_{new}$  arriva a tempo  $t$  alcuni task potrebbero essere parzialmente eseguiti:
  - $C_i(t)$  è il rimanente worst-case execution time del task  $J_i$
  - $C_i(t)$  ha un valore iniziale uguale a  $C_i$  e può essere aggiornato ogni volta che  $J_i$  è preempted
- A tempo  $t$ , il worst-case finishing time del task  $J_i$  può essere calcolato come

$$f_i = \sum_{k=1}^i C_k(t) + t$$

↓  
rimanente running time dei task con priorità più alta

Dai punti sopra descritti, la schedulabilità può essere garantita dalle seguenti condizioni:

$$\forall i=1 \dots n \quad \sum_{k=1}^i C_k(t) \leq d_i - t$$

OSS: Il dynamic guarantee test può essere fatto in  $O(n)$  essendo  $f_i = f_{i-1} + C_i(t)$  con  $f_0 = 0$  per definizione

### Example

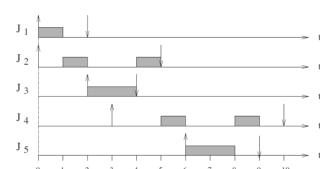
- 5 tasks, with different arrival time

Tasks parameters					
	$J_1$	$J_2$	$J_3$	$J_4$	$J_5$
$a_i$	0	0	2	3	6
$C_i$	1	2	2	2	2
$d_i$	2	5	4	10	9

Schedule (feasible) produced by EDF

$L_{max} = L2 = L3 = 0$

- All tasks meet their deadlines (the schedule is feasible).



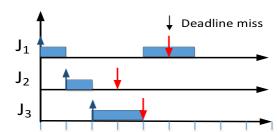
### Example 1

- 3 tasks, with different arrival time

	$J_1$	$J_2$	$J_3$
$a_i$	0	1	2
$C_i$	3	1	2
$d_i$	5	3	4

$L_{max} = L1 = 1$

- $J_1$  misses the deadline (the schedule is unfeasible).



## Complessità EDF

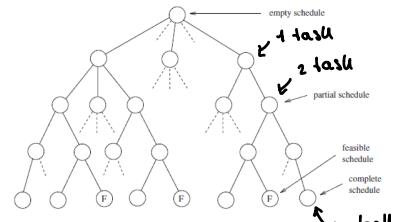
- scheduler (inserimento nella coda):  $O(n)$  → Devo vedere quale tra tutti i task ha la deadline più vicina
- guarantee test:  $O(n)$  → è seguito ogni volta che un task entra nel sistema

## 3) Non Preemptive Scheduling algorithms (Bratley's algorithm)

Quando la preemption non è permessa e i task non sincronizzati (dynamic), il problema del minimizzare la latenza massima e di trovare un feasible schedule diventa NP-hard.

Quando gli arrival time dei task sono conosciuti a priori, non-preemptive scheduling problems vengono gestiti con algoritmi branch-and-bound, i quali performano bene in media ma diventano peggiori verso una complessità esponenziale nel caso peggiore.

Questi algoritmi lavorano su un search tree space, dove la root è l'empty schedule, un vertice intermedio è una partial schedule e le foglie una complete schedule.



Oss: l'obiettivo degli algoritmi è cercare una foglia che corrisponde a una feasible schedule (non tutte le foglie lo sono)

Siano  $n$  i task, un path dalla root alla foglia è lungo  $n$  e il numero di foglie è  $n!$ .

Una ricerca esaustiva richiederebbe, nel caso peggiore, una complessità di  $O(n \cdot n!)$ .

Gli algoritmi proposti introducono tecnologie ad-hoc per limitare il search space e ridurre la complessità.

### Bratley's Algorithm

- Problema:  $\downarrow$  | no-preemptive | feasible  
 $\downarrow$        $\downarrow$        $\rightarrow$  cerca una feasible schedule  
n°cpu    preemption  
not allowed
- Algoritmo: Riduce la complessità media della ricerca tramite una pruning rule.

→ **Pruning Rule:** Non espande una partial schedule se non è **strongly feasible**, cioè aggiungendo uno qualsiasi dei nodi rimanenti la partial schedule rimane feasible.

Devo verificare se eseguendo uno specifico task, tutti gli altri rimangono feasible.

- **Assumptions:**

- Tutti i tasks consistono in un singolo Job e possono essere attivati dinamicamente (no sync)
- Task sono indipendenti (no precedence e resource constraints)
- Preeemption not allowed

- **Property:** Trovare una feasible schedule

Example

	J <sub>1</sub>	J <sub>2</sub>	J <sub>3</sub>	J <sub>4</sub>
a <sub>i</sub>	4	1	1	0
C <sub>i</sub>	2	1	2	2
d <sub>i</sub>	7	5	6	4

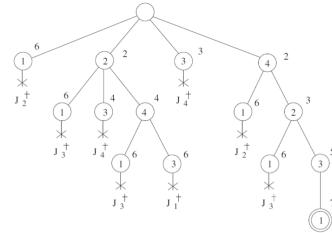
Number in the node = scheduled task

Number outside the node = finishing time

J<sub>i</sub><sup>+</sup> = task that misses its deadline

○ = feasible schedule

•  $\sigma = \{4, 2, 3, 1\}$



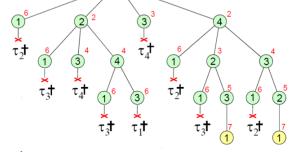
example

• To find all the feasible scheduler (and minimize L<sub>max</sub>)

	J <sub>1</sub>	J <sub>2</sub>	J <sub>3</sub>	J <sub>4</sub>
a <sub>i</sub>	4	1	1	0
C <sub>i</sub>	2	1	2	2
d <sub>i</sub>	7	5	6	4

•  $\sigma_1 = \{4, 2, 3, 1\}$

•  $\sigma_2 = \{4, 3, 2, 1\}$  L<sub>max</sub>=0 for both



Posso fermarmi quando trovo la prima feasible schedule oppure trovo tutte le soluzioni e scelgo quella con lateness migliore.

• **Complessità:** Nell' average case le pruning technique sono effettive, ma il worst case rimane comunque  $O(n \cdot n!)$ .

Per questo motivo il Bratley's algorithm può essere usato solo offline quando tutti i task parameters (includendo gli arrival times) sono conosciuti. La schedule risultante dal Bratley's algorithm può essere salvata in una struttura dati (**task activation list**) e a run time un dispatcher esegue il prossimo task dalla activation list e lo mette in esecuzione.

## 4) Scheduling Algorithms with precedence constraints

Il problema di trovare una schedule ottimale per un insieme di task con precedence relations è

NP-hard.

Assumendo però synchronous activations o preemptive scheduling, possiamo trovare algoritmi ottimali che risolvono il problema in tempo polinomiale.

Latest Deadline First (LDF) [Lawler 73] → Synchronous activation

- Problema: 4 | prec sync. |  $L_{\max}$ 
  - $n^o$  CPU
  - precedence constraints
  - sync task
  - minimize maximum lateness

- Algoritmo: Costruire la scheduling queue dalla coda alla testa.

- Tra i task senti successori o con successori che sono già stati selezionati, sceglie il task con latest deadline ( $D_i$ : maggiore)
- La procedura viene ripetuta fin quando tutti i task non vengono selezionati.
- Finita la procedura, a nun time i task vengono estraatti dalla testa alla coda

- Assunzioni:

- Tutti i tasks costituiscono un singolo job e hanno arrival time syncroni
- g task hanno precedence constraints
- Preeemption not allowed

- Property: Minimizzare la massima lateness ( $L_{\max}$ )

example

Tasks parameters

	$J_1$	$J_2$	$J_3$	$J_4$	$J_5$	$J_6$
$C_i$	1	1	1	1	1	1
$d_i$	2	5	4	3	5	6

DAG

```

graph TD
    J1((J1)) --> J2((J2))
    J1 --> J3((J3))
    J2 --> J4((J4))
    J2 --> J5((J5))
    J3 --> J6((J6))
  
```

complexity  $O(n^2)$ : For each of the  $n$  steps it needs to visit the precedence graph to find the subset with no successors.

LDF scheduler

$J_1$	$J_2$	$J_4$	$J_3$	$J_5$	$J_6$
0	1	2	3	4	5

$L_{\max} = 0$

Example: EDF is not optimal with prec. const.

Tasks parameters

	$J_1$	$J_2$	$J_3$	$J_4$	$J_5$	$J_6$
$C_i$	1	1	1	1	1	1
$d_i$	2	5	4	3	5	6

DAG

```

graph TD
    J1((J1)) --> J2((J2))
    J1 --> J3((J3))
    J2 --> J4((J4))
    J2 --> J5((J5))
    J3 --> J6((J6))
  
```

The EDF schedule is build by selecting the task with the earliest deadline among the current eligible tasks.

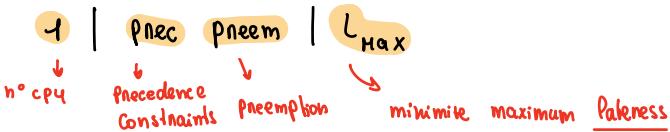
EDF scheduler Not feasible

$J_1$	$J_3$	$J_2$	$J_4$	$J_5$	$J_6$
0	1	2	3	4	5

$L_{\max} = L_4 = 1$

$J_4$  non rispetta la deadline

## EDF\* (Chotto e Chotto 89) → Preemption

- Problem: 

- Algorithm:

- Trasforma i Precedence constraints in timing constraints modificando gli arrival times e deadline basandosi sul precedence graph.
- Applica EDF sul task set modificato  
oss: se task set originale è schedulabile  $\Leftrightarrow$  se task set modificato è schedulabile

- Assumptions:

- Tutti i tasks consistono in un singolo Job e hanno arrival time dinamici
- g task hanno precedence constraints
- Preemption allowed
- Arrival times conosciuti a priori

### Arrival time modifications (release time ( $r$ ) = arrival time)

Se  $J_a$  precede  $J_b$  ( $J_a \rightarrow J_b$ ) allora una schedule per essere valida deve avere:

- $s_a \geq r_b$  :  $J_b$  deve iniziare la sua exec. dopo il suo release time
- $s_b \geq r_a + c_a$  :  $J_b$  deve iniziare la sua exec. non prima che  $J_a$  finisca

Quindi il release time  $r_b$  di  $J_b$  può essere sostituito con  $\max(r_b, (r_a + c_a))$

### Deadlines Modifications

Se  $J_a$  precede  $J_b$  ( $J_a \rightarrow J_b$ ) allora una schedule per essere valida deve avere:

- $f_a \leq d_a$  :  $J_a$  deve terminare la sua execution prima della deadline
- $f_a \leq d_b - c_b$  :  $J_a$  deve terminare la sua execution prima del maximum start time di  $J_b$

Quindi il deadline time  $d_a$  di  $J_a$  può essere sostituito con  $\min(d_a, (d_b - c_b))$

### Algorithm: arrival time modification

1. For all root nodes, set  $r_i^* = r_i$



2. Select a task  $J_i$  such that its release time has not been modified but the release times of all immediate predecessors  $J_h$  have been modified. If no such task exists, exit.

3. Set  $r_i^* = \max [r_i, \max_{J_h \rightarrow J_i} [r_h^* + C_h]]$

$$r_i^* = \max [r_i, \max_{J_h \rightarrow J_i} [r_h^* + C_h]]$$

$$r_c^* = \max [r_A^* + C_A, r_B^* + C_B]$$

4. Return to step 2.

non finito

1 Per il root node non cambio il release time.

2 Se ho un task in cui tutti i predecessori hanno il release time modificato a \* ma lui no allora esco (esempio RC)

3 ... il max tra ri e il massimo tra tutte le somme considerando i predecessori

Deadline modification algorithm:

### Algorithm: deadline modification

1. For all leaves (terminal nodes), set  $d_i^* = d_i$

$$d_i^* = \min [d_i, \min_{J_i \rightarrow J_k} [d_k^* - C_k]]$$

$$d_E^* = \min [d_A^* - C_A, d_B^* - C_B, d_F^* - C_F]$$

2. Select a task  $J_i$  such that its deadline has not been modified but the deadline of all immediate successors  $J_k$  have been modified. If no such task exists, exit.

3. Set  $d_i^* = \min [d_i, \min_{J_i \rightarrow J_k} [d_k^* - C_k]]$

4. Return to step 2.

Simile agli arrival time.

## Periodic Task scheduling

4) task periodici (sensory data acq., signal filtering ...) sono attività che devono essere eseguite ciclicamente a specifici rate che possono essere derivate dagli appl. requirements.

Quando ci sono timing constraints, il sistema deve garantire che ogni periodic instance sia regolarmente attivata (nel suo rate) e completata entro la sua deadline.

Assumptions di base:

- 1) le istanze di un task periodico  $T_i$  sono regolarmente attivate a rate costante. L'intervallo  $T_i$  tra due activations consecutive è detto **Task period**.
- 2) Tutte le istanze di un task periodico hanno lo stesso worst-case execution time ( $C_i$ )
- 3) Tutte le istanze hanno la stessa relative deadline  $D_i$ , che equivale al periodo  $T_i$ .
- 4) Tutti i task nel task set sono indipendenti (No precedenze e resource constraints)
- 5) Tutti i task sono preemptive e possono essere sospesi/attivati in un qualsiasi momento
- 6) Tutti gli overhead nel kernel sono assunti di essere 0.

### EDF\* Example

Given seven tasks, A, B, C, D, E, F, and G, construct the precedence graph from the following precedence relations:

- A → C
- B → C B → D
- C → E C → F
- D → F D → G

Then, assuming that all tasks arrive at time  $t = 0$ , have deadline  $D = 25$ , and computation times 2, 3, 3, 5, 1, 2, 5, respectively, modify their arrival times and deadlines to schedule them by EDF.

solution

$$r_i^* = \max [r_i, \max_{J_h \rightarrow J_i} [r_h^* + C_h]]$$

$$d_i^* = \min [d_i, \min_{J_h \rightarrow J_i} [d_h^* - C_h]]$$

	$C_i$	$r_i$	$r_i^*$	$d_i$	$d_i^*$
A	2	0	0	25	20
B	3	0	0	25	15
C	3	0	3	25	23
D	5	0	3	25	20
E	1	0	3	25	23
F	2	0	8	25	25
G	5	0	8	25	25

Riassunto algoritmi per aperiodic Scheduling:

	sync. activation	preemptive async. activation	non-preemptive async. activation
independent	EDD (Jackson '55) $O(n \log n)$ Optimal	EDF (Horn '74) $O(n^2)$ Optimal	Tree search (Bratley '71) $O(n!)$ Optimal
	LDF (Lawler '73) $O(n^2)$ Optimal	EDF* (Chetto et al. '90) $O(n^2)$ Optimal	Spring (Stankovic & Ramamirtham '87) $O(n^2)$ Heuristic
precedence constraints			

Se le assunzioni 1, 2, 3, 4 negli  $t_i$ , un task periodico  $t_i$  è caratterizzato da 3 parametri:

- Phase  $\phi_i$ : Time della prima attivazione del task ( $1^{\text{st}}$  activation time)
- Periodo  $T_i$
- Worst case computation time  $C_i$

Inoltre:

- Un set di task periodici può essere denotato da  $\Gamma = \{t_i(\phi_i, T_i, C_i), i=1\dots n\}$
- Il release time  $r_{i,k}$  (o l'activation time  $a_{i,k}$ ) e l'absolute deadline  $d_{i,k}$  di una generica  $k^{\text{th}}$  instance sono:

$$\begin{aligned} a_{i,k} &= r_{i,k} = \phi_i + (k-1) \cdot T_i \\ b) \quad d_{i,k} &= r_{i,k} + T_i = \phi_i + k \cdot T_i \end{aligned}$$

### Hyperperiod

L'hyperperiod è il minimo intervallo di tempo dopo il quale

la scheduale si ripete.

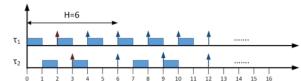
Per un insieme di task periodici sync attivati a tempo  $t=0$ ,

l'hyperperiod è dato dal least common multiple (lcm) dei periodi, cioè  $H = \text{lcm}(T_1, \dots, T_n)$   
 ↓ Minimo Comune Multiplo

L'hyperperiod ( $H$ ) è importante perché se i task periodici sono schedulabili nell'hyperperiod, allora lo saranno per sempre (perché dopo  $H$  la scheduale si ripete).

	$\Phi$	$T$	$C$
T1	0	2	1
T2	0	3	1

$\text{lcm}=3*2=6=H$



### Problem Formulation

Per ogni task periodico  $T_i$ , uno scheduling algorithm deve garantire che:

- Ogni Job  $T_{i,k}$  sia attivato a tempo  $a_{i,k} = \phi_i + (k-1) \cdot T_i$  (al rate stabilito)
- Ogni Job venga completato entro  $d_{i,k} = a_{i,k} + D_i$  (rispetti la deadline)

## Process utilization Factor ( $U(\tau)$ )

Dato un task set ( $\Gamma$ ) di  $n$  tasks, il processon utilization factor ( $U$ ) è la frazione di tempo del processo spesa nell'esecuzione del task set.

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \rightarrow \text{Quanto tempo del periodo } T \text{ il processo esegue un task?}$$

Da questa formula otteniamo che la cpu utilization può essere migliorata aumentando i tasks computation times o riducendo i tasks periods.

## Utilization Upper bound ( $U_{UB}(\Gamma, A)$ )

Esiste un valore chiamato Utilization Upper Bound ( $U_{UB}$ ) il quale è il valore massimo di  $U$  per il quale  $\Gamma$  rimane schedulabile:

- Se  $U \leq U_{UB} \rightarrow \Gamma$  schedulabile
- Se  $U > U_{UB} \rightarrow \Gamma$  not schedulable

$U_{UB}$  dipende sia dal task set  $\Gamma$ , sia dall'algoritmo di scheduling usato.

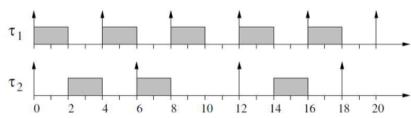
OSS: Se  $U = U_{UB}(\Gamma, A)$  il task set  $\Gamma$  utilizza completamente il processo.

### Example: Utilization Upper Bound ( $U_{ub}$ )

•  $U_{UB} = 2/4 + 2/5 = 1/2 + 2/5 = 9/10 = 0.9$

• The processor is fully utilized:

- if any  $C_i$  is increased, the first job of  $\tau_2$  misses the deadline (the task set becomes infeasible)

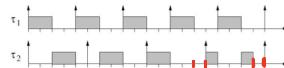


es: Se aumentassi  $c_1$  da 2 a 2.1 avrei che il primo Job di  $\tau_2$  verrebbe eseguito per 1.9 invece che di 2 (perché  $\tau_1$  ha una priorità maggiore) e quindi lo 0,1 di  $\tau_2$  verrebbe eseguito tra 6 e 6,1 che è oltre la sua deadline!

S stessa cosa se provassi ad aumentare  $c_2$ .

•  $U_{UB} = 2/4 + 2/5 = 1/2 + 2/5 = 9/10 = 0.9$

H = 20

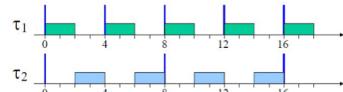


Note: The upper bound  $U_{UB}$  depends on the specific task set.

Posso anche vedere che se conto gli idle in un hyperperiod e lo divido per esso, otengo quanto "mi manca" per  $U_{UB} = 1$ .

In questo esempio,  $2/20 = 0.1$

•  $U_{UB} = 2/4 + 4/8 = 1 \rightarrow$  Migliore → CPU mai in idle



## least utilization upper bound ( $U_{LUB}$ )

Dato un algoritmo  $A$ ,  $U_{LUB}(A)$  oif processon utilization è il minimo dell'utilizzo su tutti i task sets che usano completamente il processore.

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

$U_{LUB}$  è utile per verificare la schedulabilità di un task set perché:

- $U_i \leq U_{ub,i} \rightarrow \Gamma$  schedulable
- $U_i \leq U_{LUB} \rightarrow \Gamma$  schedulable (perché  $U_{LUB}$  è il minimo dei  $U_{ub,i}$ )
- $U_i > 1 \rightarrow \Gamma$  non schedulable

Quindi,  $U_{LUB}$  è più facile da trovare di  $U_{ub}$ :

**TEOREMA:** Se  $U > 1$ , il task set non può essere schedulato da nessun algoritmo.

↓  
Dim:

1) Sia  $H$  l'hyperperiod per il task set

2) Abbiamo  $U > 1 \Rightarrow U \cdot H > H \Rightarrow \sum_{i=1}^n \frac{H}{T_i} \cdot C_i > H$

dove: •  $H/T_i$  è il numero di volte che il task  $T_i$  è eseguito nell'hyperperiod

•  $(H/T_i) \cdot C_i$  è il numero totale di computation time richiesto da  $T_i$  nell'hyperperiod.

•  $\sum_{i=1}^n \frac{H}{T_i} \cdot C_i$  rappresenta il computational time totale richiesto dal task set in  $[0, H]$

3)  $\sum_{i=1}^n \frac{H}{T_i} \cdot C_i$  non può superare il valore di  $H$ , altrimenti si va oltre l'hyperperiod.

Quindi se  $U > 1$ , non posso schedulare il task set.

## Time line scheduling

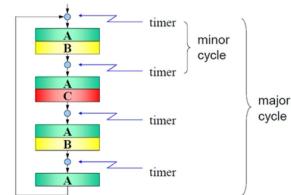
Il timeline scheduling, anche conosciuto come cyclic scheduling, è stato usato per 30 anni in sistemi militari, navigation e monitoring systems.

Questo metodo **consiste in**:

- 1) Dividere il tempo in slot di uguale lunghezza (time slots)
- 2) Allocare staticamente uno o più task in uno slot per l'esecuzione, rispettando le frequenze derivate dall'applic. requirements
- 3) L'esecuzione in ogni slot è attivata da un timer.

Data una tabella come la seguente:

Task	Ci	Ti
A	10 ms	25
B	10 ms	50
C	10 ms	100



• Minor Cycle (Time slot) = Massimo comune divisore  $\rightarrow$  In questo caso 25 ms

Per rispettare le frequenze:

- A deve essere eseguito ogni time slot
- B deve essere eseguito ogni due time slots
- C deve essere eseguito ogni quattro time slots
- Major Cycle (hyper period) = Minimo comune multiplo  $\rightarrow$  In questo caso 100 ms

Per implementare questo scheduling basta far arrivare ogni 25 ms un interrupt ed eseguire

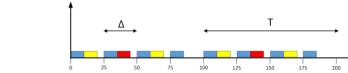
i task che devono essere eseguiti in quel time-slot.

Non ho bisogno del sistema operativo perché non ho preemption, concetto di task o context switch, quindi posso implementare questo meccanismo tramite bare-metal programming (cioè faccio girare il codice direttamente sull'HW).

example

Task	Ci	Ti
A	10 ms	25
B	10 ms	50
C	10 ms	100

- $\Delta = \text{minor cycle} = 25 \text{ ms}$
- $T = \text{major cycle} = 100 \text{ ms}$



- Guarantee (a priori)
- The sum of the execution time within each time slot is less than or equal to the minor cycle

$$\begin{cases} C_A + C_B \leq 25\text{ms} \\ C_A + C_C \leq 25\text{ms} \end{cases}$$

coding

```
#define MINOR 25 // minor cycle = 25 ms
configure_timer(MINOR); // interrupt every 25 ms
while (1) {
    wait_for_interrupt(); // block until interrupt // sync()
    Function_A();
    function_B();
    wait_for_interrupt(); // block until interrupt // sync()
    function_A();
    Function_C();
    wait_for_interrupt(); // block until interrupt // sync()
    function_A();
    function_B();
    wait_for_interrupt(); // block until interrupt // sync()
    function_A();
}
```

Vantaggi:

- Implementatione semplice (non richiede RTOS)
- Run-time overhead basso (non richiede context switches)
- Sillo basso per tutti i tasks (task start e response times non sono soggetti a grosse variazioni)

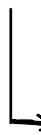
Svantaggi:

- È difficile espandere / cambiare la schedule (sensitivity to application changes)

sensitivity to application changes

Se uno o più tasks devono essere aggiornati o aggiunti, dobbiamo ridisegnare l'intero scheduling.

Caso 1) Aggiornare la computation time di un task ( $c_B = 20 \text{ ms}$ )

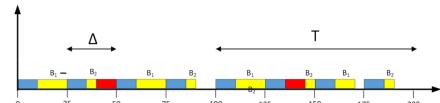


OSS: Abbiamo  $c_A + c_B > \Delta \Rightarrow$  unfeasible schedule

Task	Ci	Ti
A	10 ms	25
B	20 ms	50
C	10 ms	100

SOLUZIONE: Dividere il task B in due o più pezzi che saranno allocati negli

intervalli disponibili della time line



Caso 2) Aggiornare la task frequency di un task ( $T_2$  ridotto a 40 ms)

Aggiorno → OSS: Minor cycle = 5 e Major cycle = 200

PROBLEMA: Tutti i task devono essere suddivisi in piccoli pezzi per fitto; nuovi time slots.

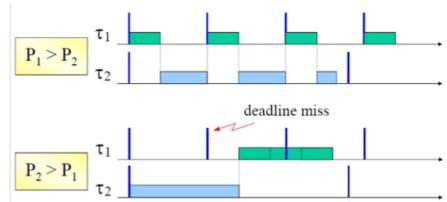
Task	Ci	Ti
A	10 ms	25
B	20 ms	40
C	10 ms	100

La soluzione al "sensitivity to application changes" problem è adottare tecniche di scheduling basate sulla priorità.

## Priority scheduling algorithms

Il priority scheduling consiste in:

- Assegnare le priorità ad ogni task basandosi sul suo timing constraint
- Verificare la feasibility della schedule usando analytical techniques
- Eseguire i task su un priority-based kernel (l'OS si occupa di generare lo scheduling)



oss: Diversi assegnamenti di priorità possono portare a diversi  $U_{UB}$  e deadline miss ...

### 1) Rate Monotonic scheduling algorithm (RM)

Questo algoritmo funziona con  $D_i = T_i$  e assegna le priorità  $P_i = \frac{1}{T_i}$ , quindi:

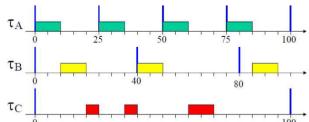
il task con periodo più piccolo ottiene priorità maggiore.

- Essendo i periodi costanti, RM priority assignment è fixed (offline assigned prima dell'exec).
- RM è un algo preemptive, il task in exec viene preempted se arriva un task con periodo minore.
- Arrival Time = Deadline = Period
- Il task con più alta priorità viene sempre eseguito quando arriva, quindi non ha jitter!

example

$\rightarrow P_A > P_B > P_C$

$P_A$  ha la priorità maggiore perché ha il periodo minore rispetto a tutti gli altri.



#### RM optimality

- 1) RM è ottimo se  $D_i = T_i$
- 2) Se esiste un fixed priority assignment che porta a una feasible schedule, allora l'RM schedule è feasible.

- 3) Se un Task set ( $\Gamma$ ) non è schedulabile da RM, allora non può essere scheduled da nessun fixed priority assignment.

Per dimostrare l'ottimalità abbiamo bisogno delle seguenti definizioni:

- Job response Time: Tempo che intercorre tra il release time del job e la fine di esso.

$$R_{i,k} = f_{i,k} - r_{i,k}$$

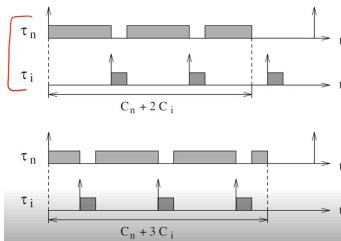
- Task response Time: Massimo response time tra tutti i Job.

$$R_i = \max_k R_{i,k}$$

- Critical instant of a task: è l'arrival time che produce il più grande task response time



Esempio:



Il response time di  $n$  è  $C_n + 2C_i$ . Quindi quando schedulo un task si ha che uno a maggior priorità può ritardare il finishing time di un altro e quindi aumentare il suo response time.  
Se muovo verso lo 0 il release time (arrival) di  $\tau_u$ , il response time di  $\tau_u$  peggiora, infatti si vede che nel secondo caso si hanno più preemption su  $\tau_u$ .  
La massima interferenza ce l'ha con  $\tau_u$  a 0!

Il critical instant (per task indip. preemptive con fixed priority) avviene quando un task arriva insieme a task che hanno priorità maggiore.

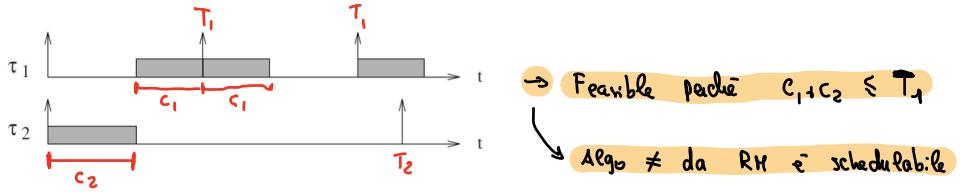
Quindi la task schedulability può essere controllata andando a vedere i critical instants di ogni task e se sono feasible nel critical instant allora il task set è schedulabile.

L'ottimalità di RM viene dimostrata mostrando che se un task set è schedulabile da un priority assignment arbitrario, allora è schedulabile anche da RM (vengono usati 2 task per la dim.).

1) Consideriamo 2 task periodici  $t_1$  e  $t_2$  con  $T_1 < T_2$

2) a) Se le priorità non sono assegnate in accordo a RM, allora  $t_2$  riceverà una priorità più alta.

In un critical instant, la schedule è feasible se la seguente inequality è soddisfatta:  $C_1 + C_2 \leq T_1$  (A)



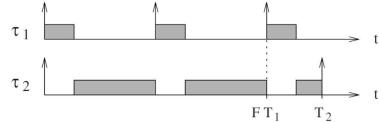
b) Se le priorità sono assegnate in accordo a RM,  $T_1$  riceverà un priorità maggiore.

Sia  $F = \lfloor \frac{T_2}{T_1} \rfloor$  il numero di periodi del task  $t_1$ , contenuti interamente in  $t_2$ .

Per garantire una feasible schedule due casi devono essere considerati:

Caso 1) Il tempo computazionale di  $t_1$  è abbastanza lungo per far sì che tutte le sue richieste sono completate prima della seconda richiesta di  $t_2$ , cioè

$$C_1 < T_2 - F T_1$$



Il task set è quindi schedulabile se  $(F+1) \cdot C_1 + C_2 \leq T_2$  (B)

Manipolando (A), vista sopra, e (B), otteniamo:

- Da (A):  $C_1 + C_2 \leq T_1 \Rightarrow F \cdot C_1 + F \cdot C_2 \leq F \cdot T_1$  quando  $F \geq 1$

- Da (B):  $F \cdot C_1 + C_2 \leq F \cdot C_1 + F \cdot C_2 \leq F \cdot T_1$  quando  $F \geq 1$

Aggiungendo  $C_1$  ad ogni membro abbiamo

- $(F+1) \cdot C_1 + C_2 \leq F \cdot T_1 + C_1$

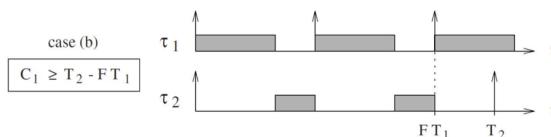
Essendo che assumiamo  $C_1 < T_2 - F \cdot T_1$ , abbiamo  $C_1 + F \cdot T_1 < T_2$

- $(F+1) \cdot C_1 + C_2 \leq F \cdot T_1 + C_1 \leq T_2$

che soddisfa (B)!

Caso 2) Il computation time di  $t_1$  è abbastanza lungo per sovrapporsi con

la seconda richiesta di  $t_2$ , cioè  $C_1 \geq T_2 - F \cdot T_1$



Il task è schedulabile se  $F \cdot C_1 + C_2 \leq F \cdot T_1$  (c)

• Da (a) :  $C_1 + C_2 \leq T_1 \Rightarrow F \cdot C_1 + F \cdot C_2 \leq F \cdot T_1$  quando  $F \geq 1$

Essendo  $F \geq 1 \rightarrow F \cdot C_1 + C_2 \leq F \cdot C_1 + F \cdot C_2 \leq F \cdot T_1 \rightarrow$  che soddisfa (c)

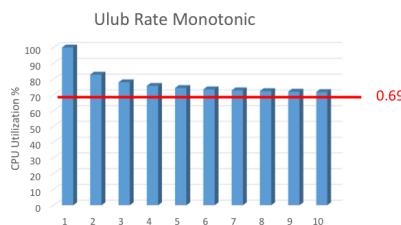
Dimostrando (A) e (B) abbiamo ottenuto che se il task set è schedulabile con un algo diverso da RM, allora il task set è schedulabile anche per RM. (OTTIMALITÀ)

### RM Schedulability Test (Guarantee Test)

#### 1) Liu & Layland (LL) bound

Per un insieme di  $n$  task periodici è stato dimostrato che :  $U_{LUB} = n(2^{1/n} - 1)$

abbiamo quindi che per valori grandi ( $n \rightarrow \infty$ ) il least upper bound ( $U_{LUB}$ ) vale 0.69



Se, dato un task set,  $U_p \leq U_{LUB}$ , allora il task set è schedulabile. **Guarantee Test**

#### Caso speciale

Se i task hanno "harmonic periods" (ogni periodo è un multiplo degli altri), allora  $U_{LUB} = 1$ . es:  $T_1 = 2, T_2 = 4, T_3 = 8 \dots$

#### 2) Hyperbolic Bound

Un insieme di  $n$  task periodici è **schedulabile** da RM se :  $\prod_{i=1}^n (U_i + 1) \leq 2$

È un test meno restruttivo di LL, cioè se LL "fallisce", questo può essere aspettato.

Il Guarantee test per RM viene fatto unendo i due test :

1) Se  $U_p > 1$  allora la schedule non è feasible

2) Se i due test falliscono, cioè  $U_p > HB$  e  $U_p > LL$ , non possiamo dire nulla

3) Se  $U_p \leq HB$  o  $U_p \leq LL \Rightarrow$  schedule feasible

## early deadline first (EDF)

EDF è un dynamic algorithm che seleziona i task in base alle loro absolute deadlines, cioè task con earlier deadline verranno eseguiti con priorità maggiori.

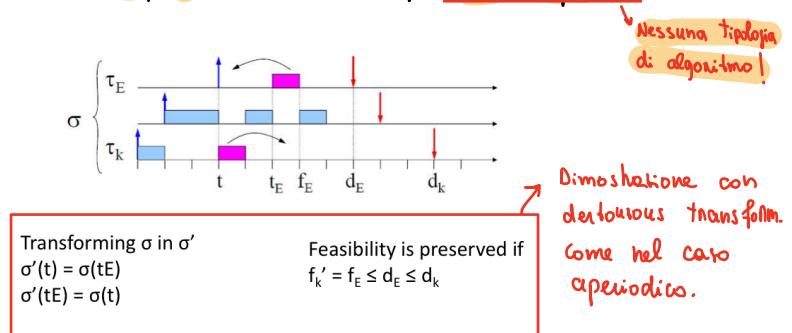
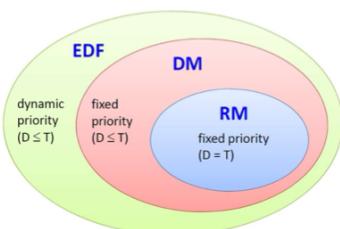
$$P_i = 1/d_{i,u} \text{ dove } d_{i,u} = \pi_{i,u} + D_i$$

- EDF è detto dinamico perché le absolute deadline dipende dall' istanza corrente del task.
- EDF è preemptive (task con periodi minori possono preempt i task in exec)
- EDF non fa assunzioni specifiche sulla periodicità del task, per questo può essere usato sia per schedulare task periodici, sia task aperiodici. (a differenza di RT che richiede  $D_i = T_i$ )
- EDF genera meno context switch di RM

## EDF Optimality

EDF è ottimo tra tutti gli algoritmi (Non solo con priority assignment fixed/dynamic)

- Se esiste una feasible schedule per un task set  $\Gamma$ , allora EDF genera una feasible schedule
- Altrimenti, se  $\Gamma$  non è schedulabile da EDF, allora non lo è per nessun algoritmo.



## EDF Schedulability

Per un insieme di  $n$  task periodici :  $\frac{\text{EDF}}{U_{LUB}} = t \Leftarrow \text{SOTTO LE ASSUNZIONI A1-A6!}$

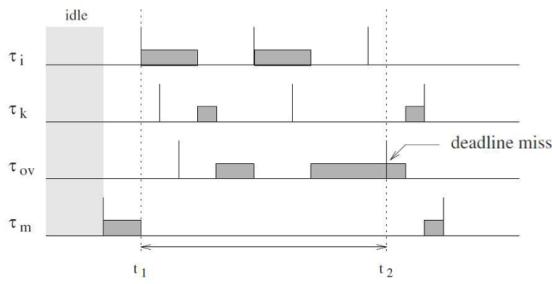
Questo vuol dire che un insieme di task periodici è schedulabile con EDF, sse:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \quad \downarrow \quad U \leq \frac{\text{EDF}}{U_{LUB}}$$

Quindi, dato EDF :

- Un task set non può essere scheduled se  $U > 1$  (come visto in precedenza)
- Se  $U_p \leq t = \frac{\text{EDF}}{U_{LUB}}$  allora  $\Gamma$  è schedulabile

### Dimostrazione per contraddizione



Assumiamo che  $U_p \leq 1$  e  $\Gamma$  non è schedulabile.

- Sia che avviene un deadline miss a tempo  $t_2$ . no idle
- Sia  $[t_1, t_2]$  il più lungo intervallo di utilizzo continuo, prima di  $t_2$ , t.c solo le istante con deadline  $\leq$  di  $t_2$  siano eseguite in  $[t_1, t_2]$ .
  - Se la CPU fosse idle, potrei schedulare la parte di  $\tau_{ov}$  che non è completata e causa un deadline miss, quindi questa condizione serve ad avere un miss.
- Sia  $C_p(t_1, t_2)$  il computation time totale richiesto dai task periodici in  $[t_1, t_2]$ .  
È dato dalla somma dei computation time dei task che hanno release time maggiore di  $t_1$  e una deadline  $\leq$  di  $t_2$  (che stanno all'interno di  $[t_1, t_2]$ )

$$C_p(t_1, t_2) = \sum_{r_k \geq t_1, d_k \leq t_2} C_k \leq \sum_{i=1}^n \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^n \frac{t_2 - t_1}{T_i} C_i$$

essendo  $\sum_{i=1}^n \frac{C_i}{T_i} = U_p$  otteniamo che:

$$C_p(t_1, t_2) \leq (t_2 - t_1) U_p$$

- Siccome c'è stata una deadline miss, l'ampiezza dell'intervallo dovrebbe essere minore del computation time richiesto in quell'intervallo, cioè:

$$(t_2 - t_1) < C_p(t_1, t_2)$$

Ottieniamo quindi :

$$(t_2 - t_1) < U_p(t_1 - t_2)$$

cioè che  $U_p > 1$  e quindi abbiamo la condizione rispetto a  $U_p \leq 1$ , cioè la nostra ipotesi.

Di conseguenza :

- 1) EDF permette di raggiungere il 100% di CPU utilization factor.
- 2) La feasibility di un insieme di task periodici può essere facilmente determinata.

### Svantaggi EDF

Quando un task entra nel sistema, devo ric算olare la priorità di EDF, perché la deadline assoluta cambia. Questo procedimento lo devo fare per ogni sua nuova istanza.

l'OS deve quindi conoscere le deadline, fase iniziale, periodo ed a ogni step di scheduling trovare l'assoluta deadline e ric算olare le priorità dei processi.

Con gli scheduler a priorità fissa non ho questo overhead.

### Extension to tasks con $D < T$

Gli algoritmi visti precedentemente fanno affidamento sull'assunzione (A3) la quale impone un relative deadline uguale al periodo, cioè  $D = T$ .

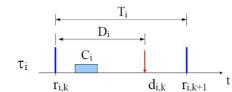
Nelle real-time applications questa condizione deve essere rilassata e avere  $D \leq T$ .

Parametri:

- $\phi_i$  = fase
- $C_i$  = Worst case computation time
- $D_i$  = Relative Deadline
- $T_i$  = Periodo

Caratteristiche:

- $C_i \leq D_i \leq T_i$
- $R_{i,u} = \phi_i + (u-1)T_i$
- $d_{i,u} = R_{i,u} + D_i$



Gli scheduling algorithms sono:

1) Deadline monotonic:  $P_i \approx \frac{1}{D_i}$  (statico)

2) EDF:  $P_i \approx \frac{1}{d_i}$  (dinamico)

### Deadline Monotonic (DM)

Sia  $P_i = \frac{1}{D_i}$ , con  $D_i \leq T_i$ , la Deadline-monotonic priority è ottimale, cioè un task è schedutabile da un fixed priority algorithm, è schedutabile anche da DM.

### Schedulability test

La schedulability non può essere testata con:

- Utilization Based set: Sarebbe pessimistico essendo che il workload sul processore

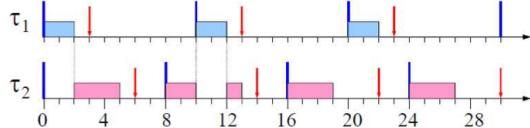
$$\sum_{i=1}^n \frac{C_i}{D_i} \leq n(2^{1/n} - 1).$$

sarebbe sovraffollato (perché  $T_i > D_i$ , è come se stessi considerando  $D_i$  come periodo!)

→ ottenuto riducendo i tasks' period to relative deadlines

- Utilization bound: Anche se Up viene maggiore di 1 il task set potrebbe essere schedutabile perché la CPU load viene sovraffollata (per lo stesso motivo precedente)

$$U_p = \sum_{i=1}^n \frac{C_i}{D_i} = \frac{2}{3} + \frac{3}{6} = 1.16 > 1$$



- Il worst case processor demand accade quando tutti i tasks vengono attivati

simultaneamente:

- Per task  $(\tau_i)$  la somma dei suoi processing time e interferenze imposte dai task con priorità maggiore deve essere  $\leq D_i$
- Assumiamo che i tasks siano ordinati per increasing relative deadlines, cioè:

$$i < j \Rightarrow D_i < D_j (P_i > P_j)$$

- Ottieniamo che  $\forall i: 1 \leq i \leq n \quad C_i + I_i \leq D_i$

dove  $I_i$  è la misura dell'interferenza su  $T_i$ , calcolata come:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_j}{T_j} \right\rceil C_j.$$

$\Rightarrow$  Somma dei processing time di tutti i task con priorità più alta rimasti prima di  $D_i$ .  
(attivati)

Anche questo test non garantisce la schedulabilità perché l'interferenza "reale" può essere minore di  $I_i$ , poiché  $T_i$  potrebbe finire prima.

### Response Time Analysis

È un sufficient e necessary schedulability test per DM.

1) Per ogni task  $T_i$  calcola l'interferenza ( $I_i$ ) dovuta ai task con più alta priorità.

$$I_i = \sum_{k=0}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k \Rightarrow$$

L'interferenza è calcolata sull'intervallo  $[0, R_i]$



2) Calcolo dei response time :

$$R_i = \sum_{k=0}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil \cdot C_k + C_i \quad (2)$$

$\Rightarrow$  Il worst-case responsive time del task  $T_i$  è dato dal più piccolo valore di  $R_i$  che soddisfa l'equazione (2).  $\Rightarrow$  Può essere trovato tramite iterazioni

1.  $R_i^{(0)} = \sum_{j=1}^i C_j$ ,  $k = 0$  (the first instant in which  $T_i$  can complete)
2.  $I_i^{(k)} = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j$  (is the interference in  $[0, R_i^{(k)}]$ )
3. If  $I_i^{(k)} + C_i = R_i^{(k)}$ ,  $R_i^{(k)}$  is the actual worst case response time of  $T_i$ , i.e.  
 $R_i = R_i^{(k)}$   
else  $R_i^{(k+1)} = I_i^{(k)} + C_i$   
go to step 2

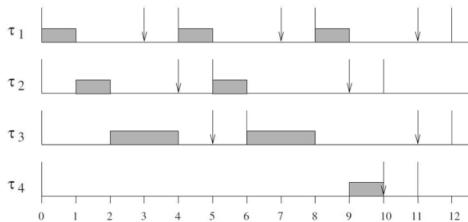
3) Conosciuto  $R_i$ , la feasibility di un task  $T_i$  è garantita se  $R_i \leq D_i$ .

Valutando l'algoritmo  $A_i$ , verifico se tutti i task sono feasible e di conseguenza se il task set  $\Gamma$  è schedulabile.

Attention: Il Response time Analysis può essere applicato anche a RM!

Conviene soprattutto quando i "sufficient tests" fallano!

Esempio :



	$C_i$	$T_i$	$D_i$
$\tau_1$	1	4	3
$\tau_2$	1	5	4
$\tau_3$	2	6	5
$\tau_4$	1	11	10

Poniamo da  $\tau_4$  che è il task con minore priorità :

- 1) Step 0:  $R_4^{(0)} = \sum_{i=1}^4 C_i = 5$ , but  $I_4^{(0)} = 5$  and  $I_4^{(0)} + C_4 > R_4^{(0)}$   
hence  $\tau_4$  does not finish at  $R_4^{(0)}$ .
- Step 1:  $R_4^{(1)} = I_4^{(0)} + C_4 = 6$ , but  $I_4^{(1)} = 6$  and  $I_4^{(1)} + C_4 > R_4^{(1)}$   
hence  $\tau_4$  does not finish at  $R_4^{(1)}$ .
- Step 2:  $R_4^{(2)} = I_4^{(1)} + C_4 = 7$ , but  $I_4^{(2)} = 8$  and  $I_4^{(2)} + C_4 > R_4^{(2)}$   
hence  $\tau_4$  does not finish at  $R_4^{(2)}$ .
- Step 3:  $R_4^{(3)} = I_4^{(2)} + C_4 = 9$ , but  $I_4^{(3)} = 9$  and  $I_4^{(3)} + C_4 > R_4^{(3)}$   
hence  $\tau_4$  does not finish at  $R_4^{(3)}$ .
- Step 4:  $R_4^{(4)} = I_4^{(3)} + C_4 = 10$ , but  $I_4^{(4)} = 9$  and  $I_4^{(4)} + C_4 = R_4^{(4)}$   
hence  $\tau_4$  finishes at  $R_4 = R_4^{(4)} = 10$ .
- Per calcolare  $I_4^{(0)}$  uso la formula,  
il valore di  $R_4^{(0)}$  è 5.  
Se come  $I_4^{(0)} + C_4 > R_4^{(0)}$  dobbiamo continuare.
  - Continuo finché non ho  
 $I_i^{(u)} + C_i = R_i^{(u)}$   
dove  $I_i^{(u)}$  corrisponde al worst case response time  
per  $\tau_4$ .
- 2) Verifico se  $R_4^{(4)} \leq D_4$ , essendo  $R_4^{(4)} = 10 \leq 10 = D_4$ , ho che  $\tau_4$  è schedulabile.

Ripeto queste due operazioni per tutti gli altri task.

La complessità del response time analysis è pseudo polinomiale e dipende dal numero di passi fatti per il calcolo di  $R_i$  per ogni task.

EDF con  $T_i < D_i$

EDF è ottimale.

### Schedulability Analysis

Con  $D_i = T_i$  è possibile applicare l'utilization bound test, che è semplice.

Con  $D_i \leq T_i$  invece, la schedulability analysis può essere fatta tramite Processor Demand  
↓  
Pseudo-polynomial

Criterion:

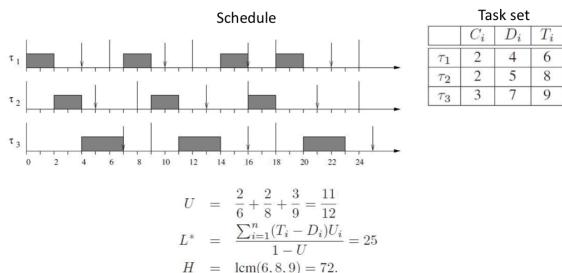
Un insieme sincrono di task periodici con le  $D_i \leq T_i$  è schedulabile da EDF se e solo se:

$$\forall L \in D \quad g(0, L) = \sum_{i=1}^n \left\lceil \frac{L+T_i-D_i}{T_i} \right\rceil C_i \leq L \quad \text{dove: } D = \{dk \mid dk \leq \min[H, L^*]\} \quad \text{e} \quad L^* = \frac{\sum_{i=1}^n (T_i - D_i) U_i}{1 - U}$$

Prendi come valori possibili di  $L$  quelli appartenenti all'insieme delle absolute deadlines, la richiesta computazionale  $g(0, L)$  deve essere minore di  $L$ .

le deadline che devo verificare sono le deadline distinte nell'hyperperiod e questo valore equivale al minimo tra  $H$  e  $L^*$ .

Esempio:



summary

- Three scheduling approaches:

- Off-line construction (Timeline)
- Fixed priority (RM, DM)
- Dynamic priority (EDF)

- Three analysis techniques:

- Processor Utilization Bound  $U \leq U_{\text{UB}}$
- Response Time Analysis  $\forall i \quad R_i \leq D_i$
- Processor Demand Criterion  $\forall L \quad g(0, L) \leq L$

complexity

- Utilization based analysis  $U \leq U_{\text{UB}}$

- $O(n)$  complexity

- Response time analysis  $\forall i \quad R_i \leq D_i$

- Pseudo-polynomial complexity

- Processor demand analysis  $\forall L \quad g(0, L) \leq L$

- Pseudo-polynomial complexity

$$D = \{4, 5, 7, 10, 13, 16, 21, 22\}$$

L	$g(0, L)$	result
4	2	OK
5	4	OK
7	7	OK
10	9	OK
13	11	OK
16	16	OK
21	18	OK
22	20	OK

↓  
Task set schedulable

## Resource Access Protocols

Una risorsa è una software structure che può essere usata da un processo in esecuzione e può essere:

- Privata: Dedicata a un processo in particolare

- Condivisa: Usata da più processi

Se protetta da accessi concorrenti è chiamata "exclusive resource" e dovrebbe essere acceduta con degli appropriati Resource Access Protocols per garantire la mutual exclusion.

Un pezzo di codice acceduto in mutua esclusione è chiamato "critical section".

### Semafoni

Per poter costruire una critical section viene usato il semaforo, il quale è una Kernel data structure che permette due operazioni: `wait()` e `signal()`.

Ogni sezione critica deve iniziare con la `wait()` e terminare con la `signal()`.

Un task che prova ad accedere a una critical section, deve verificare che nessun altro task stia usando quella risorsa:

- 1) Se è libera → Accede direttamente e svolge le operazioni.
- 2) Se è occupata → Il task viene "bloccato" dalla `wait()` operation e viene inserito in una coda.

Esso può essere risvegliato da una operazione di `signal()` e tornare nella Ready queue.

OSS: la CPU viene assegnata a un task in base allo scheduling algorithm.

### Resource Access Protocol di un semaforo

- 1) **Access Rule**: "Entra nella sezione critica se libera, altrimenti bloccati se occupata."
- 2) **Progress Rule**: "Esegue la sezione critica con nominal priority (priorità del task)".
- 3) **Release Rule**: "Sveglia i task con priorità maggiore" o "sveglia i task in FIFO order"

## Blocking time

Anche se siamo in un sistema preemptive, una sezione critica non può essere rilasciata finché non è terminata per assicurare lo stato consistente della risorsa.

Questo blocking time non può essere bounded (non possiamo sapere nemmeno se finisce), di conseguenza è un problema per i real-time systems.

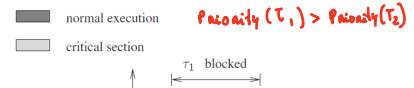
Quando un task con altra priorità è bloccato da task con meno priorità per un unbounded interval time, questo fenomeno viene detto priority inversion.

Soluzione: Usare specifici resource access protocols.

## Resource access protocols for Real-time applications

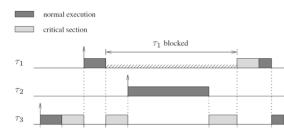
Terminologia e assunzioni:

- Insieme di  $n$  task periodici  $T_1, \dots, T_n$  che cooperano su  $m$  risorse  $R_1, \dots, R_m$
  - Ogni Task ha un periodo  $T_i$  e il worst case computation time  $C_i$
  - Ogni risorsa ha un semaforo assegnato  $S_k$  con  $k = 1, \dots, m$
  - Ogni task ha una fixed nominal priority  $P_i$  e una active priority  $p_i$  (dinamica e inizialmente settata uguale a  $P_i$ ).  $\Rightarrow p_i \geq P_i$
  - I task sono ordinati in ordine decrescente di  $P_i$ .
  - I task con stessa priorità vengono schedulati in base al loro ordine di arrivo, cioè FCFS (First come First served).
  - I task non si sospendono su op. di I/O o synchronizzazione (hanno i semafori).
  - le critical section possono essere nested, ma devono esserlo correttamente  $\Rightarrow$
- Notazione:  $T_B \subset T_A$  indica che  $T_B$  è contenuta interamente in  $T_A$ .



Rilasciata cpu ma la risorsa rimane a  $T_2$  finché non termina!

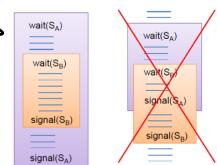
example



- if  $T_2$  arrives at time  $t_3$ , it preempts  $T_1$  (because it has a higher priority) and increases the blocking time of  $T_1$  by its entire duration.

As a consequence, the maximum blocking time that  $T_1$  may experience does depend not only on the length of the critical section executed by  $T_3$  but also on the worst-case execution time of  $T_2$ .

- In general, the duration of the blocking time of  $T_1$  is unbounded, since any intermediate-priority task that can preempt  $T_3$  will indirectly block  $T_1$ .



- $B_i$  = max blocking time di  $T_i$
- $Z_{i,u}$  = generic critical section di  $T_i$  rispetto al semaforo  $S_u$
- $Z_{i,u}$  = longest critical section di  $T_i$  rispetto al semaforo  $S_u$
- $\delta_{i,u}$  = Duration of  $Z_{i,u}$
- $y_i$  = insieme delle critical section più lunghe che possono bloccare  $T_i$

### Non Preemptive Protocol (NPP)

Ogni volta che un task  $T_i$  entra nella risorsa  $R_k$ , la sua priorità dinamica viene aumentata al massimo possibile  $p_i(R_k) = \max_h\{P_h\}$

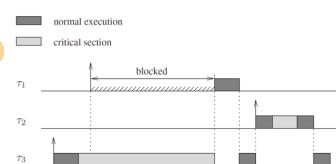
Questo permette che non si verifichino priority inversion problems perché sto "eliminando" la possibilità del preemptive.

#### Vantaggi:

- Tasks never blocks on wait perché il task in esecuzione è quello con maggiore priorità, quindi nessun task viene eseguito prima che esso termini e di conseguenza nessun task può chiamare la wait.  $\rightarrow$  Coda semaforo non necessaria
- Previene deadlocks perché il task in esecuzione è quello con maggiore priorità, quindi nessun task può bloccare la risorsa necessaria al task in esecuzione.

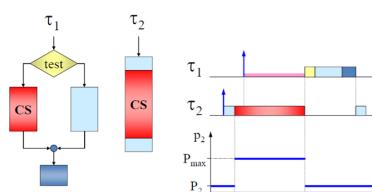
#### Svantaggi:

- Un task con priorità nominale maggiore vengono bloccati anche se non usano la sezione critica.



- Un task può bloccare, anche se non necessario, una sezione critica.

In questo esempio,  $T_1$  può accedere o meno la sezione critica (a seconda di come va il test), ma viene bloccato a prescindere da  $T_2$  perché assume una priorità più alta.



### Blocking Time computation

Il maximum blocking time per  $\tau_i$  in NPP è dato dalla durata della longest critical section tra tutte quelle assegnate a task di priorità minore.

$$B_i = \max_{j,k} \{ \delta_{j,k} - 1 \mid Z_{j,k} \in \gamma_i \}.$$

Oss: il  $-1$  è dovuto dal fatto che  $Z_{j,k}$  deve iniziare prima dell'arrivo di  $\tau_i$  per bloccarlo!

### Higher Locker Priority Protocol (HLPP)

L' HLP protocol aumenta la priorità del task quando entra in una risorsa  $R_k$  assegnandogli la massima priorità tra tutti i task che condividono quella risorsa.

$$p_i(R_k) = \max_h \{ P_h \mid \tau_h \text{ uses } R_k \}.$$

In realtà, la massima priorità per ogni risorsa viene calcolata offline (Priority ceiling  $C(R_k)$ ) e poi viene assegnata a un task quando entra all'interno della risorsa.

$$C(R_k) \stackrel{\text{def}}{=} \max_h \{ P_h \mid \tau_h \text{ uses } R_k \}$$

Naturalmente, per fare il calcolo offline devo conoscere quali task accederanno a un determinato risorsa.

Questa soluzione ci permette di evitare sia la priority inversion, sia che i task con alta priorità facciano delay inutili per risorse non condivise!

### Blocking Time Computation

Un Task  $\tau_i$  può essere bloccato da una sezione critica che appartiene a un lower priority task con un resource ceiling  $\geq p_i$ .

$$\gamma_i = \{ Z_{j,k} \mid (P_j < P_i) \text{ and } C(R_k) \geq P_i \}$$

Inoltre, può essere bloccato al più una volta, come descritto dal seguente teorema:

Sotto HLP, un task  $\tau_i$  può essere bloccato, al più, per la durata di una

singola sezione critica appartenente al set  $\gamma_i$ .

Di conseguenza, il maximum blocking time di  $T_i$  è dato dalla durata della critical section più lunga che può bloccare  $T_i$ .

$$B_i = \max_{j,k} \{\delta_{j,k} - 1 \mid Z_{j,k} \in \gamma_i\}$$

Dimostrazione per contraddizione:

- Assumiamo che  $T_i$  sia bloccato da due sezioni critiche  $\tau_{1,a}$  e  $\tau_{2,b}$  che devono appartenere a due task,  $T_1$  e  $T_2$ , con priorità minore di  $P_i$  e devono avere una ceiling  $\geq P_i$ .  
 $P_1 < P_i \leq C(R_a)$ ;  
 $P_2 < P_i \leq C(R_b)$ .
- $T_i$  può essere bloccato due volte solo se  $T_1$  e  $T_2$  sono entrambi all'interno delle risorse quando arriva  $T_i$  e può succedere solo se uno dei due (ad esempio  $T_1$ ) è preempted dall'altro all'interno della sezione critica.
- Ma se  $T_1$  preemps  $T_2$ , vuol dire che  $P_1 > C(R_b)$ , la quale è una contraddizione. Quindi, ricordando  
 $P_1 < P_i \leq C(R_a) \rightarrow P_1 < C(R_b)$ .

Vantaggi:

- Semaphores queue not needed
- Ogni task blocca al più una sezione critica
- Previene i deadlocks

Problemi:

- Il blocking di una sezione critica può essere non necessario (stesso "pessimism di NPP").  
 Un task può bloccare, anche se non necessario, una sezione critica.

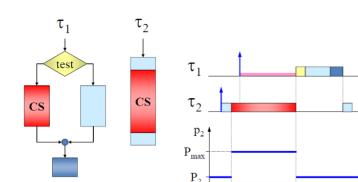
In questo esempio,  $T_1$  può accedere o meno la sezione critica (a seconda di come va il test), ma viene bloccato a prescindere da  $T_2$  perché assume una priorità più alta e condividono la risorsa.

#### HLPP: implementation notes

- Each task  $t_i$  is assigned a nominal priority  $P_i$  and a dynamic priority  $p_i$ .
- Each semaphore  $R_k$  is assigned a resource ceiling  $C(R_k)$ :

$$C(R_k) \stackrel{\text{def}}{=} \max_h \{P_h \mid \tau_h \text{ uses } R_k\}$$

- the protocol can be implemented by changing the behavior of the wait and signal primitives:
  - wait(S);  $p_i = C(S)$
  - signal(S);  $p_i = P_i$
- Full definition:**
  - wait(S);  $p_i = \max(p_i, C(S))$ , as  $p_i$  can be higher than  $C(S)$  in nested section
  - signal(S);  $p_i = \text{prev } p_i$  -- value before entering the critical section (if no critical  $P_j$ )



- Non è trasparente al programmatore (perché deve sapere i ceiling di ogni semaforo)

Anche se HLPP migliora NPP, potrebbe comunque bloccare i task in modo non necessario. Analizzeremo ora il "Priority Inheritance Protocol" che risolve questo problema postponendo la blocking condition all'entrata della sezione invece che all'attivazione.

### Priority Inheritance Protocol

Quando un task  $T_i$  blocca uno o più task con priorità maggiore, temporaneamente ottiene la priorità più alta tra i task bloccati.

In questo modo previene che Task con priorità "media" preemptino  $T_i$  e prolungano la durata del blocco dei Task con alta priorità.

Oss: Task con stessa priorità sono eseguiti in FCFS.

- 1) Quando  $T_i$  prova ad accedere a una sezione critica  $S_{i,k}$  e la risorsa  $R_k$  è già occupata da un task  $T_j$  con priorità minore, allora  $T_i$  viene bloccato e trasmette la sua priorità a  $T_j$  (modificando  $P_j$ ).
- 2)  $T_j$  continua la sua esecuzione
- 3) Quando  $T_j$  esce dalla sezione critica, sblocca il semaforo e sveglia il task con priorità maggiore.

La priorità di  $T_j$  viene aggiornata come segue:

- Se nessun altro task è bloccato da  $T_j$   $\Rightarrow P_j = P_j$
- Altrimenti  $P_j$  è impostato uguale alla più alta priorità fra essi.

Dovuto a ciò la priority inheritance è transittiva:

Se  $P_3 < P_2 < P_1 \rightarrow T_3$  ottiene la priorità di  $T_1$  tramite  $T_2$

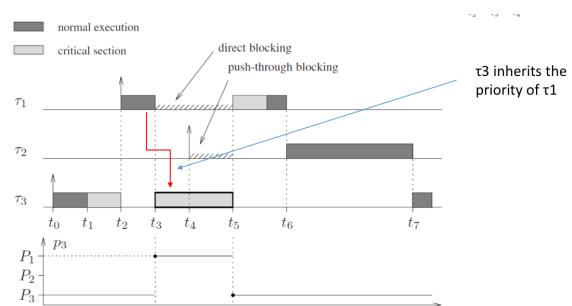
Oss: Se ci sono sezioni critiche annidate ( $A, B, B, A$ ), il task non deve tornare alla priorità nominale quando esce da B, ma a quella di A.

task annidate

1)  $T_3$  accede alla critical section  $([t_1, t_2])$

2)  $T_1$  arriva, preemps  $T_3$ , esegue la normal exec.

Foi però gli serve anche ad esso la sezione critica detenuta da  $T_3$ , quindi  $T_1$  da la sua priorità a  $T_3$ .



4)  $T_3$  continua l'esecuzione nella sezione critica e termina tornando alla priorità nominale

5)  $T_1$  esegue la sezione critica ...

### Type of Blocking

1) Direct Blocking : Avviene quando un higher-priority task prova ad acquisire una risorsa già detenuta da un lower-priority task.



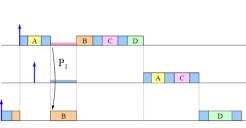
- $T_1$  can be blocked once by  $T_2$  (on A<sub>2</sub> and C<sub>2</sub>) and once by  $T_3$  (on B<sub>3</sub> and D<sub>3</sub>)
- $T_2$  can be blocked once by  $T_3$  (on B<sub>3</sub> and D<sub>3</sub>)
- $T_3$  cannot be blocked

Il direct blocking è necessario per garantire la consistenza delle risorse condivise.

2) Push-Through Blocking : Occorre quando un medium-priority task è bloccato da un low priority task che ha ereditato una higher priority da un task che blocca.

- $T_2$  is blocked on B<sub>3</sub> by push-through

Il push-through block è necessario per evitare la unbounded priority inversion.



Il push-through block è necessario per evitare la unbounded priority inversion.

### Proprietà di PiP

Lemma 1) Un semaforo  $S_k$  può causare un push-through blocking sul task  $T_i$  solo se è acceduto sia da un task con priorità  $< P_i$ , sia da un task con priorità  $> P_i$ .

→ Dimostrazione:

Supponiamo che  $S_k$  sia acceduto da un task  $T_e$  con priorità  $< P_i$ , ma non sia acceduto da un task con priorità  $> P_i$ .

Allora,  $T_2$  non può ereditare una priorità  $> p_i$  e di conseguenza  $T_2$  sarà preemped da  $T_i$ .

Lemma 2) L'eredità transitiva della priorità può avvenire solo in presenza di critical section intercalate.

→ Dimostrazione:

Una eredità transitiva avviene quando un high-priority task ( $T_H$ ) è bloccato da un medium priority Task ( $T_M$ ) che a sua volta è bloccato da un low-priority task ( $T_L$ ).

Essendo  $T_H$  bloccato da  $T_M$ ,  $T_M$  deve avere un semaforo ( $S_a$ ), e essendo  $T_M$  bloccato da  $T_L$ ,  $T_L$  deve avere un diverso semaforo ( $S_b$ ).

Quindi,  $T_M$  blocca  $S_b$  all'interno della critical section controllata da  $S_a$ .

→ Consequenza lemma 2:

Se non ci sono nested critical section  $\Rightarrow$  Non ci sono eredità transitive di priorità e blocking.

Importante perché sappiamo che se non ci sono sezioni critiche intercalate, non dobbiamo

considerare la transitive priority nel calcolo del blocking time!

Lemma 3) Se ci sono  $l_i$  lower-priority tasks che bloccano  $T_i$ , allora  $T_i$  può essere bloccato per al più la durata di  $l_i$  critical sections, indipendentemente dal numero di semafori usati da  $T_i$ .

Quindi, un task  $T_i$  può essere bloccato al più una volta da un lower-priority task.

→ Dimostrazione:

Siano  $B_{i,j}$  l'insieme delle sezioni critiche di un low-priority task  $T_j$  che possono bloccare  $T_i$  e  $Z_{j,u}$  una sezione critica di  $T_j$  con semaforo  $S_u$ .

- $T_j$  può bloccare  $T_i$  solo se  $T_j$  è in esecuzione in una critical section  $Z_{j,u} \in B_{i,j}$  quando  $T_i$  viene initializzato.

- Quando  $T_j$  esce da  $Z_{s,u}$ , può essere preempted da  $T_i$  e quindi  $T_i$  non può essere bloccato da  $T_j$  nuovamente.
- Questo può accadere per ognuno degli  $l_i$ , quindi  $T_i$  può essere bloccato al più  $l_i$  volte.

**Lemma 4)** Se ci sono  $s_i$  semafori distinti che bloccano  $T_i$ , allora  $T_i$  può essere bloccato al più per la durata di  $s_i$  sezioni critiche, indipendentemente dal numero di sezioni critiche usate da  $T_i$ .  
Quindi, un task  $T_i$  può essere bloccato al più una volta su un semaforo  $S_u$ .

→ **Dimostrazione:**

Essendo i semafori binari, solo uno dei lower priority tasks ( $T_j$ ) può essere all'interno di una critical section bloccata corrispondente al semaforo  $S_u$ . Quando  $S_u$  viene sbloccato,  $T_j$  può essere preempted e non blocca più  $T_i$ . Se tutti i semafori (che bloccano  $T_i$ ) sono bloccati da  $s_i$  lower-priority tasks, allora  $T_i$  può essere bloccato al più  $s_i$  volte.

**Teorema** (ottenibile dai precedenti lemma):

Sotto P:P, un task  $T_i$  può essere bloccato al più per la durata di  $d_i = \min(n_i, m_i)$  critical sections, dove:

- $n_i$  = n° tasks con priorità minore di  $T_i$
- $m_i$  = n° semafori che bloccano  $T_i$

**Terminologia P:P**

- Ogni risorsa  $R_u$  ha un semaforo distinto  $S_u$ .
- $Z_{i,u}$  = generic critical section di  $T_i$  rispetto al semaforo  $S_u$
- $Z_{i,u}$  = longest critical section di  $T_i$  rispetto al semaforo  $S_u$
- $S_{i,u}$  = worst-case duration of  $Z_{i,u}$

- $\sigma_i$  = insieme dei semafori usati da  $T_i$
- $\sigma_{i,j}$  = insieme di semafori che possono bloccare  $T_i$  usati dal lower priority task  $T_j$ .
- $\gamma_{i,j}$  = insieme delle longest critical sections che possono bloccare  $T_i$ , accedute da un lower priority task  $T_j$ .  $\gamma_{i,j} = \{Z_{j,k} \mid (P_j < P_i) \text{ and } (S_k \in \sigma_{i,j})\}$
- $\gamma_i$  = insieme di tutte le longest critical sections che possono bloccare  $T_i$ .
 
$$\gamma_i = \bigcup_{j:P_j < P_i} \gamma_{i,j}$$
- $B_i$  = Massimo (worst-case) blocking time che  $T_i$  può riscontrare.
- $a_i$  = Massimo numero di critical sections che possono bloccare  $T_i$

### Blocking Time Computation in RP

Quando non ci sono nested critical sections (no eredità transitiva) :

- l'insieme dei semafori del lower priority task  $T_j$  che possono causare un direct blocking a  $T_i$ 

$$\sigma_{i,j}^{dir} = \sigma_i \cap \sigma_j$$

dove :

$\sigma_i$  = semafori che possono bloccare  $T_i$

$\sigma_j$  = semafori che possono bloccare  $T_j$

- l'insieme dei semafori del lower priority task  $T_j$  che possono causare un push-through blocking a  $T_i$ 

$$\sigma_{i,j}^{pt} = \bigcup_{k=1}^{i-1} \sigma_k \cap \sigma_j$$

dove :

- $\bigcup_{k=1}^{i-1} \sigma_k$  = semafori che possono bloccare i task con priorità maggiore di  $T_i$

- $\sigma_j$  = semafori che possono bloccare  $T_j$

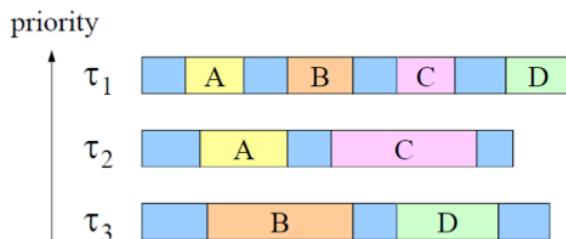
- l'insieme dei semafoni del lower priority task  $T_j$  che possono bloccare  $T_i$  è dato dall'intersezione tra l'insieme dei semafoni che direct-blocking  $T_i$  e l'insieme dei semafoni che push-through blocking  $T_i$ :

$$\sigma_{i,j} = \sigma_{i,j}^{dir} \cap \sigma_{i,j}^{pt} = \bigcup_{k=1}^i \sigma_k \cap \sigma_j$$

Esempio:

- 1)  $T_1$  può essere bloccato da:

- $\sigma_{1,2} = A, C$
  - $\sigma_{1,3} = B, D$
- $\left. \begin{matrix} \\ \end{matrix} \right\}$  Direct blocks



- 2)  $T_2$  può essere bloccato da:

- $\sigma_{2,3} = \{A, B, C, D\} \cap \{B, D\} = B, D$
- $\left. \begin{matrix} \\ \end{matrix} \right\}$  Push-through blocking

- 3)  $T_3$  Non può essere bloccato

Quindi, ritornando al calcolo del blocking Time, siano:

- l'insieme delle longest critical sections usate da  $T_j$  che possono bloccare  $T_i$ .

$$\gamma_{i,j} = \{Z_{j,k} \mid (P_j < P_i) \text{ and } (R_k \in \sigma_{i,j})\}.$$

- l'insieme di tutte le critical sections che possono bloccare  $T_i$

$$\gamma_i = \bigcup_{j=i+1}^n \gamma_{i,j}$$

$B_i$  è dato dalla più grande somma delle lunghezze delle  $a_i$  critical sections in  $\gamma_i$

Algoritmo:

- 1) Identifica  $\gamma_{i,j}$  per ogni lower priority task  $j$
- 2) Identifica  $\gamma_i$
- 3) Calcola  $a_i$

- 4) Calcola  $B_i$ : come la più grande somma delle  $a_i$  durations  $\sum_{j,u} \in \gamma_i$   
 le  $a_i$  selezionate devono appartenere a task differenti (lemma 1) e far riferimento a  
 semafori differenti (lemma 2)

Example

	Ci	Ti	A	B	C
$\tau_1$	5	25	1	2	0
$\tau_2$	15	60	0	9	3
$\tau_3$	20	100	8	7	0
$\tau_4$	20	200	6	5	4

#### Identification of $\gamma_1$

	Ci	Ti	A	B	C
$\tau_1$	5	25	1	2	0
$\tau_2$	15	60	0	9	3
$\tau_3$	20	100	8	7	0
$\tau_4$	20	200	6	5	4

- $\gamma_1 = \{B_2, A_3, B_3, A_4, B_4\}$

- $\tau_1$  can only experience direct blocking because it is the highest priority task.

#### Identification of $\gamma_2$

	Ci	Ti	A	B	C
$\tau_1$	5	25	1	2	0
$\tau_2$	15	60	0	9	3
$\tau_3$	20	100	8	7	0
$\tau_4$	20	200	6	5	4

- $\gamma_2 = \{A_3, B_3, A_4, B_4, C_4\}$

- $\tau_2$  can be blocked directly by  $B_3, B_4, C_4$
- $\tau_2$  can be blocked indirectly by  $A_3, B_3, A_4, B_4$

#### Identification of $\gamma_3$

	Ci	Ti	A	B	C
$\tau_1$	5	25	1	2	0
$\tau_2$	15	60	0	9	3
$\tau_3$	20	100	8	7	0
$\tau_4$	20	200	6	5	4

- $\gamma_3 = \{A_4, B_4, C_4\}$

- $\tau_3$  can be blocked directly by  $A_4, B_4$
- $\tau_3$  can be blocked indirectly by  $A_4, B_4, C_4$

#### Identification of $\gamma_4$

	Ci	Ti	A	B	C
$\tau_1$	5	25	1	2	0
$\tau_2$	15	60	0	9	3
$\tau_3$	20	100	8	7	0
$\tau_4$	20	200	6	5	4

- $\gamma_4 = \{\}$

- the lowest priority task can never be blocked

#### Identification of $\alpha_i$

	A	B	C	$\gamma_i$	$n_i$	$m_i$	$\alpha_i$
$\tau_1$	1	2	0	$\{B_2, A_3, B_3, A_4, B_4\}$	3	2	2
$\tau_2$	0	9	3	$\{A_3, B_3, A_4, B_4, C_4\}$	2	3	2
$\tau_3$	8	7	0	$\{A_4, B_4, C_4\}$	1	3	1
$\tau_4$	6	5	4	$\{\}$	0	0	0

- $\alpha_i = \min(n_i, m_i)$

- $n_i$  = Number of tasks with priority less than  $\tau_i$  that can block  $\tau_i$
- $m_i$  = Number of semaphores that can block  $\tau_i$ 
  - Directly or indirectly

#### Identification of $B_i$

	A	B	C	$\gamma_i$	$\alpha_i$	durations	$B_i$
$\tau_1$	1	2	0	$\{B_2, A_3, B_3, A_4, B_4\}$	2	$B_2 + A_3$	17
$\tau_2$	0	9	3	$\{A_3, B_3, A_4, B_4, C_4\}$	2	$A_3 + B_4$	13
$\tau_3$	8	7	0	$\{A_4, B_4, C_4\}$	1	$A_4$	6
$\tau_4$	6	5	4	$\{\}$	0	0	0

- For  $\tau_2$ , we cannot select  $A_3$  and  $B_3$  because each semaphore can block only once (Lemma 4)

- For  $\tau_2$ , we cannot select  $A_3$  and  $A_4$  because each task can block only once (Lemma 3)
  - Hence, we select  $A_3$  and  $B_4$

OSS:

- Questa somma dovrebbe contenere solo termini  $\sum_{j,u} \in \gamma_i$  che fanno riferimento a diversi task e differenti semafori (perché un task può essere bloccato solo una volta da un task/sempaforo)

- Anche senza considerare nested critical sections, la computazione esatta di  $B_i$  richiede una licenza combinatoriale per trovare la più grande somma fra tutte le possibili  $a_i$ . È possibile però trovare un upper bound con un algoritmo semplificato.

### Simple Algorithm - Blocking time Computation

Per individuare le sezioni critiche che possono bloccare un task:

- Per ogni semaforo  $S_k$ , il ceiling  $C(S_k)$  è uguale alla più alta priorità dei Task che usano  $S_k$   $C(S_k) \stackrel{\text{def}}{=} \max_i \{P_i \mid S_k \in \sigma_i\}$ .
- Una sezione critica  $\tau_{j,k}$  di  $T_j$  con semaforo  $S_k$  può bloccare  $T_i$  se e solo se:  $P_j < P_i \leq C(S_k)$ .

Quindi abbiamo che:

$$\begin{aligned} B_i^l &= \sum_{j=i+1}^n \max_k \{\delta_{j,k} - 1 \mid C(S_k) \geq P_i\} \\ B_i^s &= \sum_{k=1}^m \max_{j>i} \{\delta_{j,k} - 1 \mid C(S_k) \geq P_i\}. \end{aligned} \quad \left\{ \begin{array}{l} \text{prendo} \\ B_i = \min(B_i^l, B_i^s) \end{array} \right.$$

DSS:

- Richiede  $O(m \cdot n^2)$
- Trova un upper bound per  $B_i$  (è più pessimistico)

example

	A (1)	B (1)	C (2)
$\tau_1$	1	2	0
$\tau_2$	0	9	3
$\tau_3$	8	7	0
$\tau_4$	6	5	4

- Semaphore ceilings are indicated in parentheses

Computation of  $B_1^l$

	A (1)	B (1)	C (2)
$\tau_1$	1	2	0
$\tau_2$	0	9	3
$\tau_3$	8	7	0
$\tau_4$	6	5	4

iteration for the search of the max

$$B_1^l = 9 + 8 + 6 = 23$$

sezione critica massima per ogni task

Computation of  $B_1^s$  and  $B_1$

	A (1)	B (1)	C (2)
$\tau_1$	1	2	0
$\tau_2$	0	9	3
$\tau_3$	8	7	0
$\tau_4$	6	5	4

iteration for the search of the max

massimo sui lower priority task per ogni semaforo \*

- $B_1^s = 9 + 8 = 17$
- $B_1 = \min(B_1^l, B_1^s) = 17$

### Computation of $B_2$

	A (1)	B (1)	C (2)
$T_1$	1	2	0
$T_2$	0	9	3
$T_3$	8	7	0
$T_4$	6	5	4

- $B_2^I = A_3 + A_4 = 8+6=14$
- $B_2^S = A_1 + B_3 + C_4 = 8+7+4=19$
- $B_2 = 14$
- the algorithm is pessimistic:  $B_2^I$  is computed by adding the duration of two critical sections both guarded by semaphore  $A_i$ , which can never occur in practice.

### Computation of $B_3$ and $B_4$

	A (1)	B (1)	C (2)
$T_1$	1	2	0
$T_2$	0	9	3
$T_3$	8	7	0
$T_4$	6	5	4

- $B_3^I = A_4 = 6$
- $B_3^S = A_1 + B_4 + C_4 = 6+5+4=15$
- $B_3 = 6$
- $B_4^I = B_4^S = 0$

OSS: Prima si provano i test con l'algoritmo più semplice e se ci da "schedulable", allora si provava anche l'algoritmo più complesso.

### Schedulability Analysis

Tutti gli schedulability test presentati nella sezione dei task periodici per task indipendenti possono essere estesi per includere blocking terms, i quali valori dipendono sullo specifico "concurrency control protocol" adottato nella schedule.

Si va ad includere il blocking term nel calcolo:  $C_i + B_i < \text{deadline}$

Tutti i guarantee tests che erano necessari e sufficienti sotto il preemptive schedule diventano solo sufficienti con la presenza dei blocking factors.

### Analysis Under fixed priority assignments

Sia il Rate Monotonic (RM) algorithm lo scheduling algorithm utilizzato:

1) Liu and Layland test ( $\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$ )

Un insieme di task periodici con blocking factors e relative deadlines (uguali ai periodi,  $D_i = T_i$ ) è schedulabile da RM se:

$$\forall i, 1 \leq i \leq n, \quad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \underbrace{\frac{C_i + B_i}{T_i}}_{\substack{\text{contribute task} \\ \text{con più alta priorità}}} \leq i(2^{1/i} - 1)$$

2) Hyperbolic Test

Un insieme di task periodici con blocking factors e relative deadlines (uguali ai periodi) è schedulabile da RM se:

$$\forall i, 1 \leq i \leq n, \quad \prod_{k=1}^{i-1} \left( \frac{C_k}{T_k} + 1 \right) \left( \frac{C_i + B_i}{T_i} + 1 \right) \leq 2$$

DSS: i test (1) e (2) valgono per ogni resource access protocol visto, cambierà solo il calcolo di  $B_i$ .

### Response Time Analysis

Considerando le blocking conditions, il response time di un generico task  $T_i$  (con priorità fissata) può essere calcolato dalla recurrent relation:

$$R_i = \sum_{k=1}^{i-1} \left[ \frac{R_i}{T_k} \right] C_k + C_i + B_i \quad \xrightarrow{\text{risolvibile così:}}$$

$\begin{cases} R_i^0 = C_i + B_i \\ R_i^{(s)} = C_i + B_i + \sum_{k=1}^{i-1} \left[ \frac{R_i^{(s-1)}}{T_k} \right] C_k \end{cases}$	Iterative solution: iterate while $R_i^s > R_i^{(s-1)}$
--	---

### Vantaggi: P:P

- 1) Rimuove il pessimismo di NPP e HLP (un task è bloccato solo quando necessario)
- 2) Trasponibile al programmatore

### Problemi di P:P

- 1) Chained blocking: Se un task ( $T_i$ ) accede a  $n$  semafori distinti bloccati da  $n$  lower-priority tasks,  $T_i$  sarà bloccato per la durata di  $n$  critical sections.
- 2) Deadlocks: L'ereditarietà della priorità non prevede i deadlocks perché dipendono da un uso erroneo dei semafori