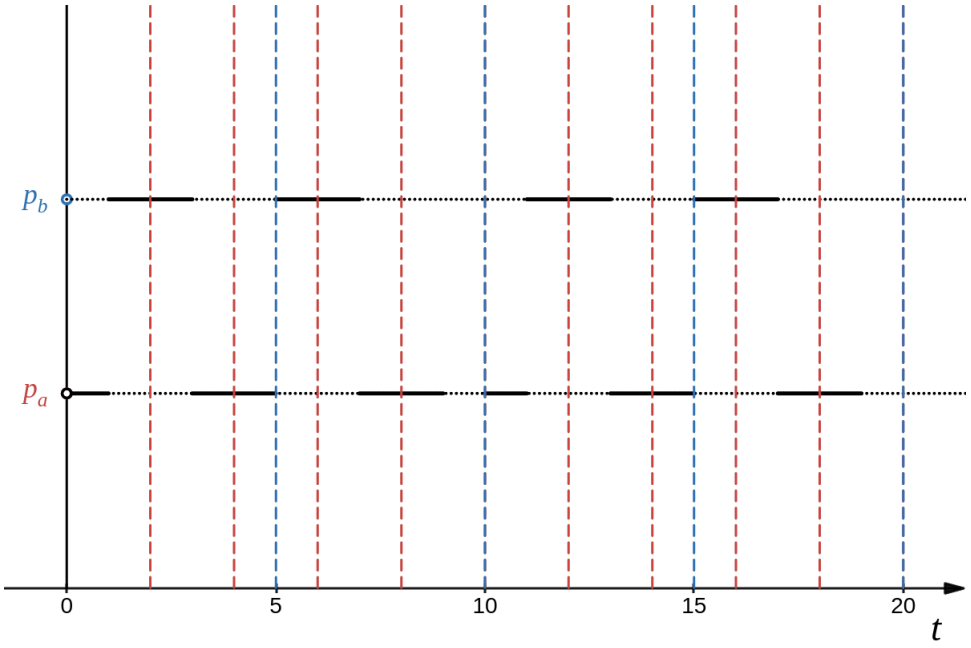
1 Lezione del 15-10-25

Continuiamo la discussione dell'algoritmo **RM** (*Rate Monotonic*).

Volevamo vedere gli effetti che si ottenevano quando si aumentava la pressione sulla CPU sfruttando questo algoritmo. Prendiamo allora gli stessi processi p_a e p_b della scorsa lezione, ma raddoppiamo il tempo di esecuzione del processo p_b:

Processo	**Δt periodo**	**C esecuzione**
p_a	2	1
p_b	5	2

Simulando l'esecuzione si ha, colorando le linee di periodo come nello scorso esempio:

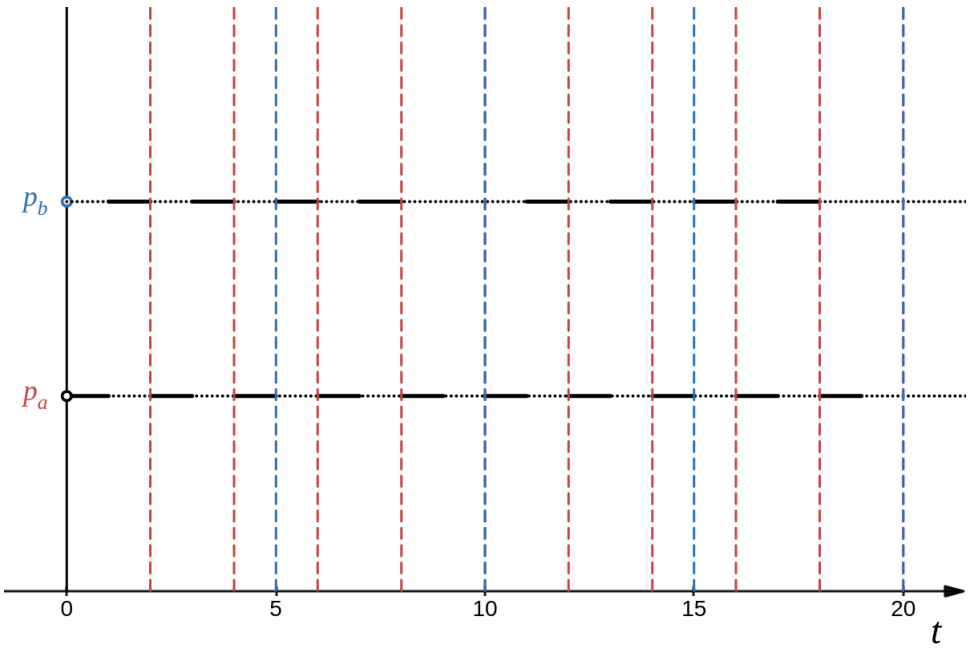


Vediamo quindi come siamo arrivati al 90% di utilizzo della CPU, e come il fatto che l'algoritmo è non preemptive significa che quando p_b accede alla CPU, la tiene anche oltre la linea di periodo di p_a (che ha comunque tempo di eseguire prima della linea successiva).

1.0.1 Algoritmo RM "preemptive"

Potremmo introdurre la preemption nell'algoritmo RM. In questo caso, ad ogni periodo riportiamo in esecuzione il processo con priorità più alta.

Vediamo quindi la timeline che otteniamo applicando questa versione con preemption all'esempio della scorsa sezione:



Notiamo che questo algoritmo di scheduling introduce un overhead maggiore della versione non preemptive, dato dai maggiori cambi di contesto. Inoltre, almeno in questo caso, non varia particolarmente in utilizzo CPU o in efficacia generale.

Comunque, possiamo osservare che mantiene i processi ancora più lontani dalla deadline, che è generalmente un comportamento desiderabile.

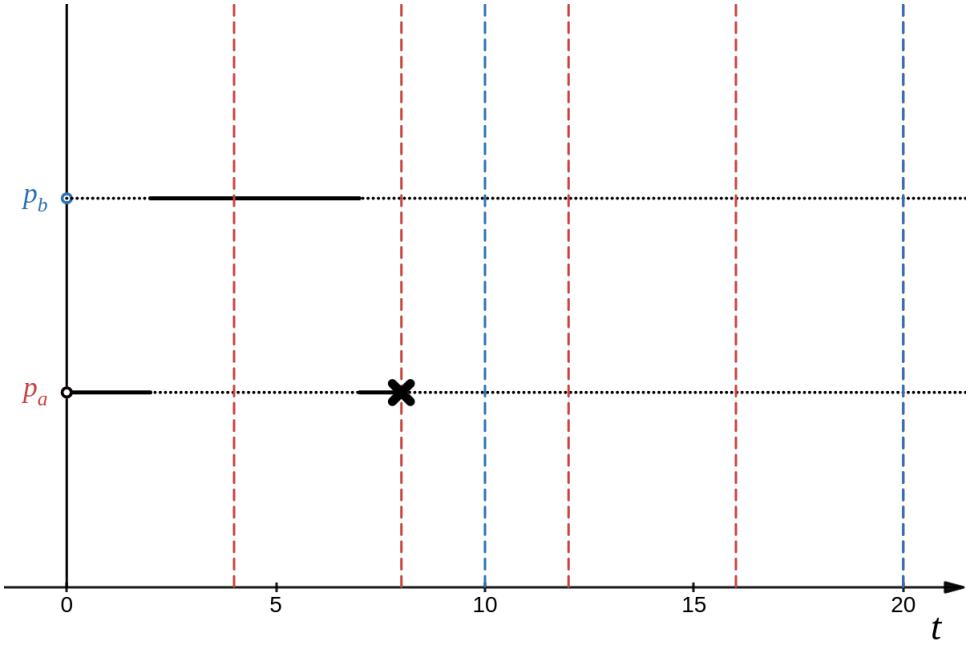
Abbiamo quindi che per gli algoritmi a priorità *static*, il RM è ottimo: se un insieme di processi è schedulabile a priorità statica in real-time, allora lo è con RM. Di contro, se non è schedulabile con RM, non esiste nessun algoritmo a priorità statica che può schedularlo.

1.1 Processi non schedulabili staticamente

Approfondiamo cosa significa, per un insieme di processi, essere *schedulabili a priorita statica*. Prendiamo la tabella di processi:

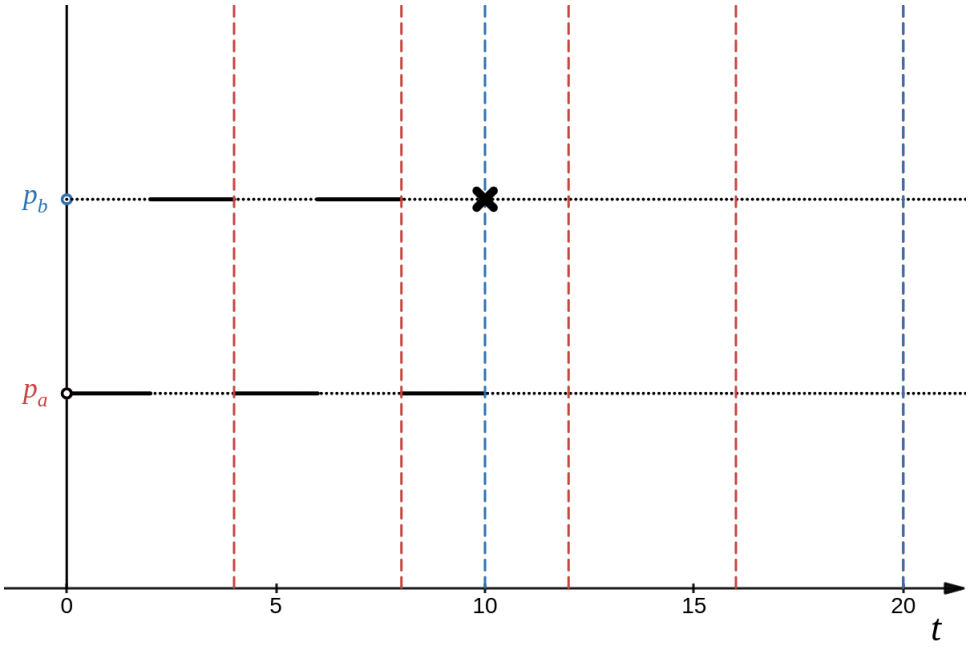
Processo	Δt periodo	C esecuzione
p_a	4	2
p_b	10	5

Vediamo come ogni processo chiede di eseguire per metà del suo periodo.
Se usiamo la schedulazione RM in questo caso, otteniamo:



Dove all'istante $t = 8$ non siamo riusciti a completare il CPU burst di p_a, e si è quindi giunti ad un *overflow*: un fallimento della schedulazione che non ha rispettato la deadline.

Pensiamo allora di utilizzare l'algoritmo RM "preemptive" visto nella scorsa szione. In questo caso, si avrà:



A questo punto è p_b ad andare in overflow! All'istante $t = 10$ infatti non siamo riusciti a completare le sue 5 unità temporali per completare l'esecuzione.

Abbiamo chiaramente incontrato un insieme di processi non schedulabili con priorità statica, e nemmeno introducendo la preemption abbiamo risolto il problema: dovremo trovare una qualche altra soluzione.

1.1.1 Trattazione matematica

Abbiamo che, nel caso dei due processi p_a e p_b, il minimo che dobbiamo rispettare per poter in primo luogo eseguire i processi nel periodo di sistema è:

$$n_a C_a + n_b C_b \leq T$$

dove ricordiamo T è il m.c.m. fra i periodi t_a e t_b, e n_a e n_b sno rispettivamente il numero di volte in cui i processi p_a e p_b entrano in esecuzione per periodo di sistema. In particolare, questi valori si possono calcolare dai perodi dei processi Δt_a, Δt_b, come:

$$n_a = \frac{T}{\Delta t_a}, \quad n_b = \frac{T}{\Delta t_b}$$

Sostituendo, si ha quindi:

$$\frac{T}{\Delta t_a} C_a + \frac{T}{\Delta t_b} C_b \leq T \implies \frac{C_a}{\Delta t_a} + \frac{C_b}{\Delta t_b} \leq 1$$

Possiamo quindi generalizzare quanto trovato alla (ovvia) legge:

$$U = \sum_{i=0}^{n-1} \frac{C_i}{T_i} \leq 1$$

per n processi arbitrari, dove U viene detto **fattore di utilizzazione**.

Nell'esempio considerato finora, questo valore è:

$$U = \frac{2}{4} + \frac{5}{10} = 1$$

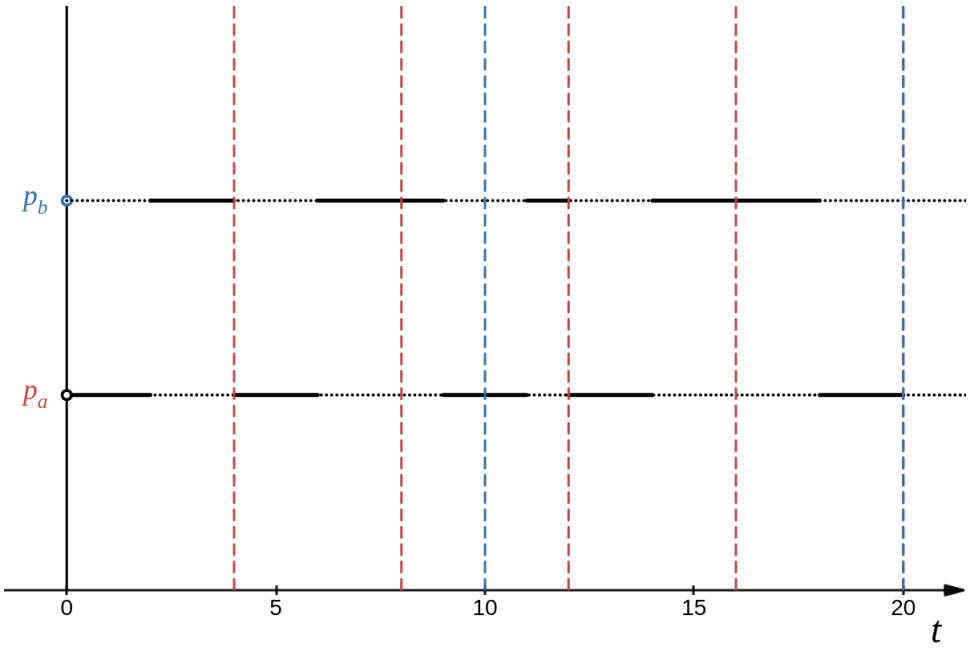
per cui i processi sono schedulabili. Ci manca da trovare un algoritmo che li sappia schedulare.

1.2 Algoritmo EDF

L'algoritmo **EDF** (*Earliest Deadline First*) è un algoritmo di schedulazione real-time, preemptive e a priorità dinamica.

Consiste nel mandare in esecuzione il processo che è più vicino alla sua deadline. Come in tutti gli algoritmi a priorità dinamica, lo scheduler viene messo in esecuzione al cambio dei criteri di scelta (quindi quando un processo entra in coda pronti). In ogni caso, se due processi so trovano ugualmente vicini alla deadline al momento dell'esecuzione dello scheduler, si opta per ridurre i cambi di contesto al minimo e mantenere in esecuzione quello che sta già eseguendo.

Vediamo come questo algoritmo si applica all'esempio schedulabile visto finora:



Vediamo come otteniamo il 100% dell'utilizzazione CPU, e riusciamo a schedulare i processi senza overflow.

All'istante $t = 8$, infatti, il processo p_b è più vicino di p_a alla deadline, e quindi viene mantenuto in esecuzione (fino a $t = 9$ dove termina). In questo modo si riesce ad evitare che il suo CPU burst venga *"tagliato"* prima che esso possa rispettare la sua deadline.

Abbiamo quindi trovato un'algoritmo che risolve i problemi che avevamo incontrato con RM: possiamo anticipare che questo algoritmo è ottimo fra gli algoritmi di schedulazione in real-time a priorità dinamica.

Una considerazione può essere fatta sull'overhead che introduciamo, almeno per l'esempio sopra. Abbiamo detto che sì, si cerca di mantenere al minimo i cambi di contesto, ma c'è comunque un certo overhead dato dall'esecuzione dello scheduler ad ogni creazione di processo.

In questo, potremmo aggiornare il modello introdotto in 9.1.1 come segue:

$$U = ... \leq 1 - O_v$$

dove O_v è un fattore temporale che tiene conto dell'overhead.

1.3 Thread

Introduciamo semplicemente il concetto di **thread** (o *processo leggero*). Avevamo detto che un processo è al contempo:

- Un elemento che possiede delle *risorse*;
- Un elemento a cui viene *assegnata* la CPU (si conserva lo *stato* e si usa uno *scheduler* per decidere quando caricarlo).

Possiamo separare questi due aspetti:

- Definiamo **processo leggero**, o *thread*, l'elemento a cui viene assegnata la CPU;
- Di contro, definiamo **processo pesante**, o *task*, l'elemento che possiede le risorse.

Un processo pesante può essere composto da più thread, ognuno dei quali rappresenta effettivamente un flusso di esecuzione a sé stante. Tutti i thread possono però accedere alle risorse del loro processo pesante (incluso *spazio di indirizzamento*, file aperti, ecc...).