1 Lezione del 14-10-25

1.1 Schedulazione real-time

Veniamo quindi a come implementare la schedulazione nei sistemi in **tempo reale**. Avevamo detto che questi erano sistemi principalmente di tipo *embedded*, cioè incorporati, non general-purpose ma *special-purpose* atti a governare sistemi esterni (sistemi di controllo per veicoli, macchinari industriali, ecc...).

1.1.1 Esecuzione ciclica

Abbiamo che la caratteristica principale di sistemi di questo tipo è il tipo di periferiche con cui interagiscono: invece di periferiche multiple e variabili (come nei sistemi general-purpose), avremo un insieme fisso di dispositivi di ingresso (detti *sensori*) e di uscita (detti *attuatori*).

Questo porta ad un paradigma di esecuzione fortemente periodico: si campiona il sistema esterno attraverso i sensori, compie una qualche elaborazione, e aggiornano gli attuatori per rispondere a quanto rilevato.

Ciò significa che i processi messi in esecuzione devono rispettare il periodo dell'esecuzione ciclica, e produrre il loro risultato entro date *scadenze* date dal periodo corrente e il numero di altri processi in esecuzione. Occorre allora avere un controllo preciso e granulare sul tempo che impiegano a terminare.

1.1.2 Deadline

In questo caso prevederemo, dopo l'istante di richiesta r di un processo, una certa deadline d, calcolata come:

$$d = r + \Delta d$$

dove Δd è il tempo massimo di esecuzione del processo.

- In un sistema *soft real-time* si cerca di fare il possibile per assicurare che il processo termini prima di d;
- In un sistema *hard real-time* la terminazione del processo prima di d è prerogativa dell'integrità dell'intero sistema.

In particolare, riguardo al paradigma di esecuzione ciclica accennato nello scorso paragrafo, avremo che per un processo che deve eseguire ciclicamente con periodo Δt, per ogni istante di richiesta r_i l'istante di richista successivo sarà calcolato come:

$$r_{i+1} = r_i + \Delta t$$

In questo caso sarà fondamentale rispettare la diseguaglianza:

$$d_i < r_{i+1} \;\Leftrightarrow\; \Delta d_i < \Delta t$$

data d_i come deadline dopo la richiesta r_i.

Estendo il concetto a sistemi multiprogrammati, avremo che dati n processi il periodo T di aggiornamento simultaneo di ogni processo in esecuzione sarà:

$$T = \text{MCM}\left(\Delta t_i\right)$$

con t_i i periodi di ogni processo: semplicemente si prende il minimo comune multiplo.

Ritornando all'idea dei CPU burst, se il processo si svolge in più CPU burst C_1, C_2, ecc... dovrà quindi essere che:

$$T_e = \sum C_i < \Delta d_i$$

cioè che almeno il tempo di esecuzione del processo sia minore del tempo massimo di esecuzione per rispettare la deadline corrente.

1.1.3 Algoritmo RM

Iniziamo quindi a vedere alcuni algoritmi di scheduling per sistemi realtime. Il primo che vediamo è l'**RM** (*Rate Monotonic*). Questo consiste semplicemente ad assegnare una priorità statica *monotonica crescente* ai processi in base al *rate*, cioè la frequenza, del loro ciclo di esecuzione. Questo equivale ad assegnare una proprietà inversamente proporzionale al periodo t del processo:

$$p \propto \frac{1}{t} = f$$

Visto che la proprietà è statica, chiaramente l'algoritmo è non preemptive.

Facciamo quindi l'esempio dell'esecuzione dell'algoritmo, ipotizzando due processi p_a e p_b:

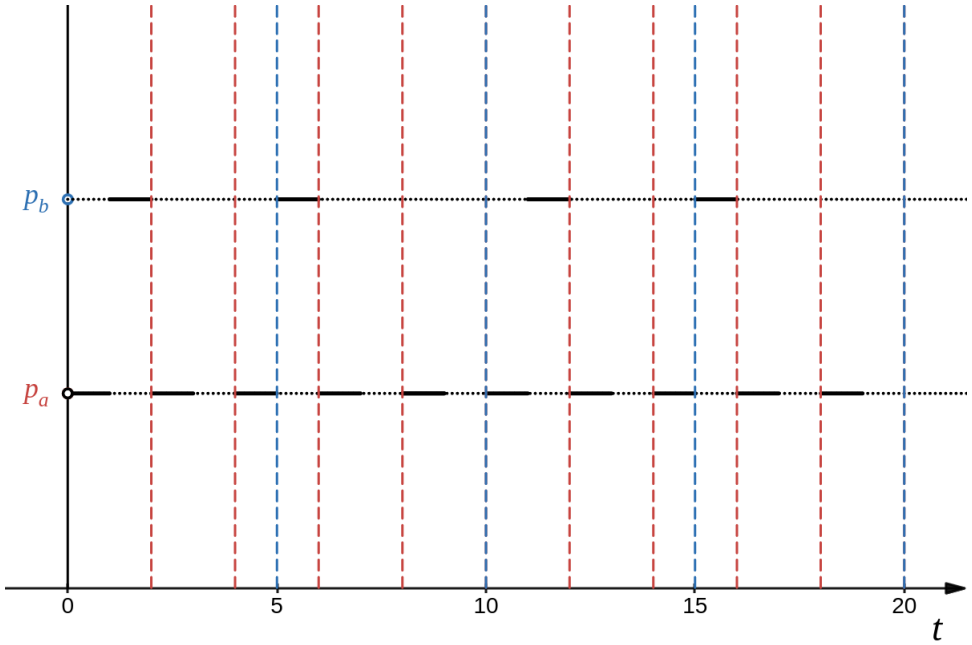
Processo	Δ**t periodo**	$\mathbf{\Delta T}$ **esecuzione**
p_a	2	1
p_b	5	1

Da questo è fra l'altro immediato che, con $\Delta t_a = 2$ e $\Delta t_b = 5$, il periodo complessivo di sistema T è:

$$T = \text{MCM}(\Delta t_a, \Delta t_b) = \text{MCM}(2, 5) = 10$$

Vedremo come questo periodo determina anche il periodo dell'attività dello scheduler.

Simulando l'esecuzione si ha, colorando in rosso le deadline di p_a e in blu quelle di p_b:



Vediamo quindi come riusciamo a rispettare tutte le deadline. Un problema è che, ad esempio all'istante 10, si sono fatti 3 cicli da un unità temporale a vuoto, cioè l'efficienza E è:

$$E = \frac{10 - 3}{10} = 70\%$$

Questo non è immediatamente sbagliato: significa solo che il sistema ha abbastanza risorse da soddisfare ampiamente le richieste in arrivo. Potrebbe diventare un problema quando vogliamo *"stringere"* le temporizzazioni in modo da far fronte ad un maggior numero di processi, o processi con CPU burst più consistenti.