# Capitolo 2

# Algoritmi di scheduling

## 2.1 Sistemi Real Time

In un sistema in *tempo reale* (real time) il tempo gioca un ruolo essenziale. Le applicazioni di tali sistemi sono molteplici e di larga diffusione. Ad esempio, un dispositivo lettore di compact disc, prende i bit in uscita dal drive e li converte in musica. Questa operazione deve compiersi in un intervallo di tempo ristretto, in quanto eventuali ritardi determinerebbero alterazioni nella riproduzione del suono. Sistemi di monitoraggio (ad esempio in una struttura ospedaliera) e di controllo (il pilota automatico di un aereo, il controllo robotizzato in una fabbrica, etc.) sono altri esempi comuni di sistemi che devono operare in tempo reale. In tutti questi casi, avere la risposta corretta, ma in ritardo, è come non averla affatto se non peggio.

I sistemi real time sono normalmente classificati in accordo a vincoli temporali che possono essere più o meno stringenti. In particolare:

- sistemi *hard real time*: sono sistemi che devono rispettare scadenze temporali in modo incondizionato. Non rispettare anche una sola scadenza può avere ripercussioni gravi sull'intero sistema;

- sistemi *soft real time*: sistemi in cui non rispettare occasionalmente una scadenza non è opportuno ma può essere tollerato.

In entrambi i casi, il comportamento real time è ottenuto dividendo un programma in processi il cui comportamento è prevedibile e noto in anticipo. Questi processi hanno normalmente una vita breve e possono essere eseguiti nella loro completezza in un tempo inferiore al secondo. Nel momento in cui viene rilevato un evento esterno, lo scheduler dei processi ha il compito di determinare un'opportuna sequenza di esecuzione dei processi tale da garantire il rispetto di tutte le scadenze.

Gli eventi a cui un sistema real time deve poter reagire possono essere classificati in:

- *periodici*: si verificano ad intervalli di tempo regolari;
- *aperiodici*: si verificano in modo imprevedibile.

Nel caso di eventi periodici, se esistono $m$ eventi e se l'evento $i$ arriva con periodo $P_i$ e richiede $C_i$ secondi di tempo di CPU per essere gestito, il carico può essere gestito solo se:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1. \tag{2.1}$$

Un sistema real time che rispetta questo vincolo è detto *schedulabile*. Ad esempio, consideriamo un sistema soft real time con tre eventi periodici di periodo 100, 200 e 500 millisecondi, rispettivamente. Se questi eventi richiedono rispettivamente 50, 30 e $100ms$ di tempo di CPU per la loro esecuzione, il sistema è schedulabile in quanto: $50/100+30/200+100/500 = 0,5 + 0,15 + 0,2 \leq 1$. Se un quarto evento, con periodo di 1 secondo, si aggiunge al sistema, l'insieme dei processi rimarrà schedulabile fino a quando questo evento non richiederà più di $150ms$ di tempo di CPU per occorrenza. Notare che nel calcolo precedente si suppone implicitamente che l'overhead per il cambio di contesto sia così piccolo da poter essere trascurato.

### 2.1.1 Scheduling dei Sistemi Real Time

Gli algoritmi di scheduling real time possono essere distinti in:

- *statici*: la decisione di schedulazione è presa prima che il sistema inizi l'esecuzione dei processi. Questi metodi richiedono che le informazioni complete circa il lavoro da fare e le scadenze da rispettare siano disponibili in anticipo rispetto all'esecuzione dei processi;
- *dinamici*: la decisione di schedulazione è presa durante l'esecuzione dei processi. Non hanno restrizioni circa la conoscenza anticipata sui tempi di esecuzione e le scadenze da rispettare.

Nel seguito verranno analizzate alcune politiche di scheduling real time, facendo riferimento al particolare contesto delle applicazioni multimediali. Infatti, i sistemi operativi che supportano applicazioni multimediali, differiscono da quelli tradizionali per tre aspetti principali: la schedulazione dei processi, il file system e la schedulazione del disco.

#### Schedulazione di processi omogenei

È la situazione che si presenta quando più processi con uguali richieste e vincoli temporali devono essere serviti in modo efficiente dalla politica di scheduling. Ad esempio, una tale situazione si presenta per un server video che deve supportare la visualizzazione di un numero fisso di video tutti

caratterizzati dalla stessa frequenza dei frame (*frame rate*), risoluzione video, frequenza di trasmissione dati, etc. In questa situazione una semplice ma efficace politica di scheduling è il round-robin. Infatti, tutti i processi sono ugualmente importanti, hanno la stessa quantità di lavoro da svolgere e si bloccano quando hanno terminato l'elaborazione del frame corrente. L'algoritmo di schedulazione può essere ottimizzato aggiungendo un meccanismo di temporizzazione per assicurare che ogni processo sia eseguito alla frequenza corretta.

### Schedulazione generale in tempo reale

Il semplice modello precedente si presente raramente nella pratica. Un modello più realistico prevede la presenza di più processi che competono per l'uso della CPU, ciascuno con il proprio carico di lavoro e le proprie scadenze temporali. Nel seguito supporremo che il sistema conosca la frequenza con cui eseguire ciascun processo, quanto lavoro debba compiere ogni processo e la successiva scadenza temporale.

Come esempio di ambiente in cui lavori uno schedulatore multimediale in tempo reale si considerino i tre processi $A$, $B$ e $C$ di Fig.2.1. Il processo $A$ viene eseguito ogni $30ms$ e ogni frame richiede $10ms$ di tempo di CPU. In assenza di competizione sarebbe eseguito nei periodi $A1$, $A2$, $A3$ etc., ciascuno $30ms$ dopo il precedente. Ogni periodo di CPU gestisce un frame e ha una scadenza, cioè deve terminare prima che inizi il successivo. I processi $B$ e $C$ in Figura sono eseguiti rispettivamente 25 e 20 volte al secondo, con tempi di calcolo di $15ms$ e $5ms$.

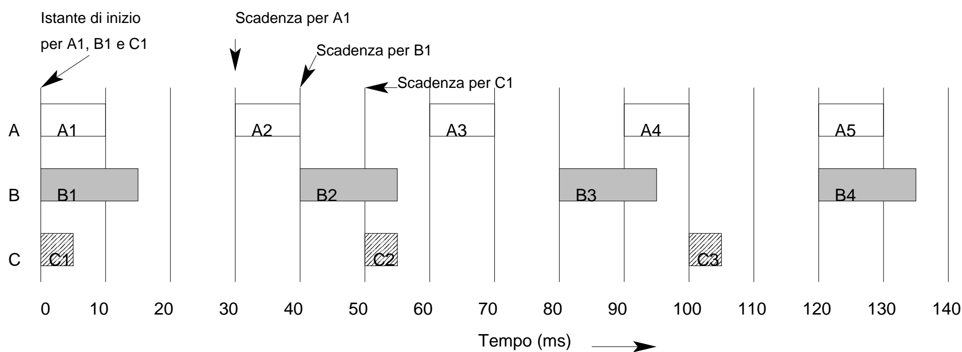


Figura 2.1: Tre processi periodici che visualizzano ciascuno un film. Le frequenze dei frame e i tempi di elaborazione per frame sono diversi per ciascun film.

Il problema diventa quello di schedulare $A$, $B$ e $C$ per essere certi che rispettino ciascuno le proprie scadenze temporali. Prima di cercare un algoritmo di schedulazione, è necessario valutare se questo insieme di processi sia effettivamente schedulabile. A questo fine è possible utilizzare l'Eq.(2.1), che

nel caso dei processi $A$, $B$ e $C$ dell'esempio produce: $10/30+15/40+5/50 = 0.808$ del tempo di CPU, ed il sistema dei processi è quindi schedulabile.

Esistono sistemi in tempo reale in cui i processi possono subire o meno prelazione. Nei sistemi multimediali i processi sono generalmente prelazionabili: un processo la cui scadenza temporale sia a rischio può interrompere il processo in esecuzione prima che esso completi l'elaborazione del proprio frame; quando ha terminato, il processo precedente può continuare. In aggiunta, gli algoritmi di schedulazione in tempo reale possono essere sia *statici* che *dinamici*. Quelli statici assegnano a ciascun processo una priorità determinata in precedenza ed effettuano una schedulazione con prelazione e con priorità utilizzando le priorità stesse. Gli algoritmi dinamici non hanno priorità prefissate.

**Schedulazione a frequenza monotòna** Il classico algoritmo statico di schedulazione in tempo reale per processi periodici e prelazionabili è RMS (*Rate Monotonic Scheduling*, schedulazione a frequenza monotòna). È utilizzabile per processi che soddisfano le seguenti condizioni:

1. ogni processo periodico deve essere completato entro il suo periodo di tempo;
2. nessun processo è dipendente dagli altri;
3. ogni processo necessita della stessa quantità di tempo di CPU per ogni periodo di esecuzione;
4. i processi non periodici non hanno scadenze temporali;
5. la prelazione dei processi avviene istantaneamente e senza sovraccarico di lavoro per il sistema.

Le prime quattro condizioni sono ragionevoli, mentre l'ultima rende più semplice la modellazione del sistema.

RMS assegna a ciascun processo una priorità prefissata uguale alla frequenza con cui deve essere eseguito. Ad esempio, un processo che debba essere eseguito ogni $30ms$ (33 volte/s) acquisisce priorità 33; un processo da eseguire ogni $40ms$ (25 volte/s) acquisisce priorità 25, mentre un processo da eseguire ogni $50ms$ (20 volte/s) acquisisce priorità 20. Dato che le priorità variano linearmente con la frequenza (numero di volte al secondo in cui il processo è eseguito), il metodo è detto a frequenza monotòna. Durante l'esecuzione, lo schedulatore esegue sempre il processo pronto a più alta priorità, prelazionando, se necessario, il processo in esecuzione. È stato dimostrato che RMS è ottimale rispetto alla classe di algoritmi di schedulazione statici. La Fig.2.2 illustra il funzionamento dell'algoritmo di schedulazione a frequenza monotòna relativamente ai processi dell'esempio di Fig.2.1. I processi

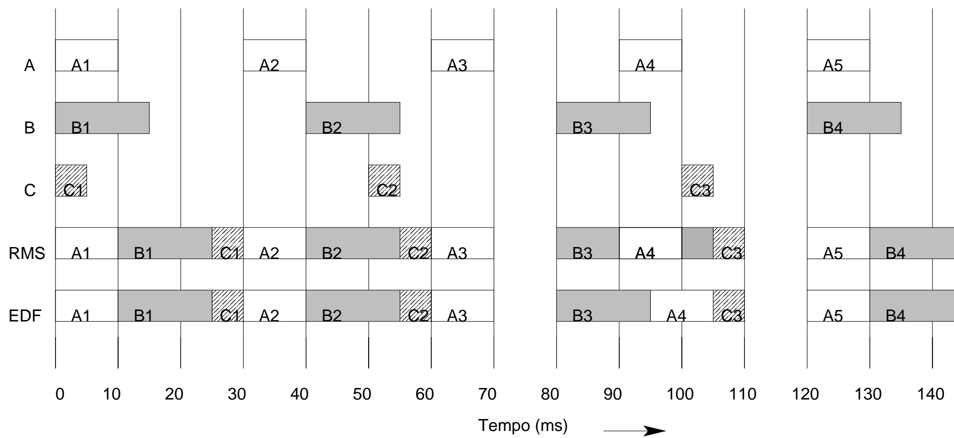


Figura 2.2: Esempio di schedulazione in tempo reale con RMS e EDF.

$A$, $B$ e $C$ hanno rispettivamente priorità statiche 33, 25 e 20. Come conseguenza, quando $A$ deve andare in esecuzione prelaziona ogni altro processo; $B$ può prelazionare $C$, mentre $C$ per andare in esecuzione deve attendere fino a quando la CPU non è libera.

**Schedulazione con priorità alla scadenza più vicina** L'algoritmo EDF (*Earliest Deadline First*, schedulazione con priorità alla scadenza più vicina), è un algoritmo dinamico e pertanto non richiede né che i processi siano periodici, né che abbiano lo stesso tempo di esecuzione per periodo di CPU. Con questo approccio, è sufficiente che un processo che ha bisogno della CPU annunci la sua presenza e la scadenza temporale. Lo schedulatore mantiene una lista dei processi eseguibili, ordinata rispetto alla scadenza temporale; l'algoritmo esegue il primo processo della lista, cioè quello con scadenza temporale più vicina. Quando un nuovo processo è pronto, il sistema controlla se la sua scadenza preceda quella del processo correntemente in esecuzione; in caso affermativo il nuovo processo prelaziona quello corrente. La Fig.2.2 presenta un esempio di schedulazione con EDF. Un secondo esempio che confronta RMS e EDF è mostrato in Fig.2.3.

È interessante notare come nell'esempio riportato in Fig.2.3 l'algoritmo RMS fallisca. Questo è dovuto al fatto che, utilizzando priorità statiche, l'algoritmo funziona solo se l'utilizzo della CPU non è troppo elevato. È possibile dimostrare che, per ogni sistema di processi periodici, se:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m \cdot (2^{1/m} - 1) \tag{2.2}$$

allora è garantito il funzionamento di RMS (condizione sufficiente). Per $m$ uguale a 3, 4, 5, 10, 20, 100 le massime utilizzazioni permesse sono 0.780, 0.757, 0.743, 0.718, 0.705 e 0.696. Per $m$ che tende all'infinito, l'utilizzo

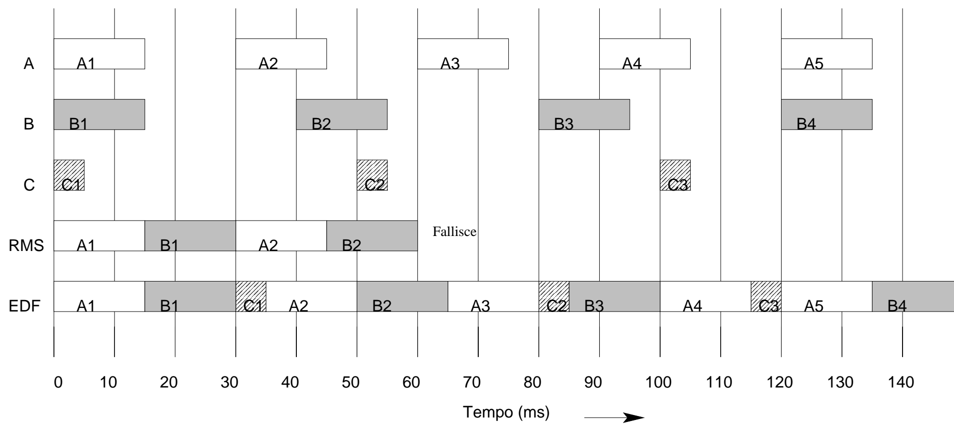


Figura 2.3: Esempio di schedulazione in tempo reale con RMS e EDF.

massimo della CPU tende in modo asintotico a $ln2 \simeq 0.69$. Questo significa che per $m = 3$, RMS funziona sempre se l'utilizzazione della CPU è uguale o minore di 0.780. Nell'esempio di Fig.2.3 l'utilizzo della CPU, calcolato con l'Eq.(2.1), è così elevato (0.975) da non permettere il funzionamento di RMS.

Il caso di Fig.2.2 è invece una situazione "fortunata," in quanto anche se l'utilizzazione della CPU è 0.808, quindi maggiore del limite imposto da RMS per tre processi (0.780), l'algoritmo riesce ugualmente a schedulare i processi.

Al contrario, EDF funziona sempre per qualunque insieme di processi schedulabile e può raggiungere il 100% di utilizzo della CPU. Il prezzo di questo è pagato in termini di una maggiore complessità dell'algoritmo EDF rispetto a RMS.

### 2.1.2 Inversione di priorità ed ereditarietà della priorità

Il fenomeno dell'inversione di priorità può essere introdotto attraverso un esempio.

Si consideri un sistema in cui sono in esecuzione 3 processi $P_A$, $P_B$ e $P_C$, con livelli di priorità, rispettivamente, 1 (minima), 2 e 3 (massima). La configurazione in cui i processi sono inizialmente osservati è la seguente:

| | | |
|---|---|---|
| $t_0$: | coda processi pronti a priorità 1: | $\mathbf{P_A^*}$ |
| | coda processi pronti a priorità 2: | *vuota* |
| | coda processi pronti a priorità 3: | *vuota* |
| | coda processi bloccati: | $P_B$, $P_C$ |

Il carattere "*" indica che il processo $P_A$ detiene una particolare risorsa. Questa potrebbe essere una stampante, il disco, una unità a nastro, ma anche una sezione critica a cui si accede in mutua esclusione. Il processo in

esecuzione è quindi $P_A$, essendo l'unico processo pronto (il processo in esecuzione è indicato in grassetto). Supponendo che successivamente il processo $P_B$ si sblocchi, la configurazione del sistema diventa:

| | | |
|---|---|---|
| $t_1$: | coda processi pronti a priorità 1: | $P_A^*$ |
| | coda processi pronti a priorità 2: | $\mathbf{P_B}$ |
| | coda processi pronti a priorità 3: | *vuota* |
| | coda processi bloccati: | $P_C$ |

ed il processo in esecuzione diviene $P_B$, avendo questo priorità maggiore di $P_A$. Se a questo punto anche il processo $P_C$ si sblocca, questo ottiene il processore in quanto processo pronto a più alta priorità:

| | | |
|---|---|---|
| $t_2$: | coda processi pronti a priorità 1: | $P_A^*$ |
| | coda processi pronti a priorità 2: | $P_B$ |
| | coda processi pronti a priorità 3: | $\mathbf{P_C}$ |
| | coda processi bloccati: | *vuota* |

Se durante la sua esecuzione il processo $P_C$ cerca di ottenere l'accesso alla risorsa detenuta dal processo $P_A$, esso si blocca nuovamente, ed il processore viene assegnato al processo $P_B$:

| | | |
|---|---|---|
| $t_3$: | coda processi pronti a priorità 1: | $P_A^*$ |
| | coda processi pronti a priorità 2: | $\mathbf{P_B}$ |
| | coda processi pronti a priorità 3: | *vuota* |
| | coda processi bloccati: | $P_C$ *in attesa della risorsa* $*$ |

Generalizzando l'esempio precedente, è possibile affermare che il fenomeno dell'inversione di priorità (*priority inversion*) si verifica quando un processo ad alta priorità—al limite un processo real-time (il processo $P_C$ nel caso dell'esempio)—si trova bloccato in attesa di una risorsa detenuta da un processo a bassa priorità (il processo $P_A$ nell'esempio). A causa della sua bassa priorità, il processo potrebbe rilasciare la risorsa con notevole ritardo in quanto altri processi possono andare in esecuzione (il processo $P_B$ dell'esempio). In una situazione del genere il processo ad elevata priorità risulta accodato ad un processo a bassa priorità, e quindi procede tutt'altro che speditamente (contrariamente a quanto ci si aspetterebbe da un processo cui è stata assegnata un'elevata priorità).

La soluzione al problema appena descritto prende il nome di ereditarietà della priorità (*priority inheritance*) e consiste nell'assegnare temporaneamente al processo che detiene la risorsa, la priorità del processo a maggiore priorità che la richiede (il processo a bassa priorità viene promosso); nel momento in cui il processo rilascia la risorsa, ad esso viene riassegnata la priorità originaria (viene cioè retrocesso). In questo modo si accelera l'esecuzione del processo che detiene la risorsa, riducendo il periodo di tempo durante il quale il processo a maggiore priorità risulta essere bloccato.

Vediamo la soluzione con inversione di priorità applicata all'esempio precedente.

Non appena il processo $P_C$ si blocca in attesa della risorsa detenuta da $P_A$, a quest'ultimo viene temporaneamente assegnata una priorità pari a 3, per cui gli viene assegnato il processore. In accordo al meccanismo di ereditarietà della priorità, lo stato dei processi all'istante $t_3$ è modificato nel modo seguente:

| $t_3$: | coda processi pronti a priorità 1: | *vuota* |
|---|---|---|
| | coda processi pronti a priorità 2: | $P_B$ |
| | coda processi pronti a priorità 3: | $\mathbf{P_A^*}$ |
| | coda processi bloccati: | $P_C$ *in attesa della risorsa* $*$ |

Quando $P_A$ rilascia la risorsa, il sistema gli assegna nuovamente la priorità originaria, ed il processo $P_C$ viene sbloccato e riprende l'esecuzione:

| $t_4$: | coda processi pronti a priorità 1: | $P_A$ |
|---|---|---|
| | coda processi pronti a priorità 2: | $P_B$ |
| | coda processi pronti a priorità 3: | $\mathbf{P_C^*}$ |
| | coda processi bloccati: | *vuota* |

Notare che, in assenza del meccanismo di ereditarietà della priorità, nella peggiore delle ipotesi la situazione si sarebbe sbloccata solo nel momento in cui il processo $P_B$ avesse completato la propria esecuzione. Infatti, nell'ipotesi in cui $P_B$ non si bloccasse mai in attesa di una risorsa, $P_B$ sarebbe il processo pronto a maggiore priorità fino al suo completamento (anche in presenza di un meccanismo di scheduling tipo round-robin).

La soluzione descritta è di interesse per la realizzazione di sistemi operativi tempo-reale, ed in particolare di quelli soft real-time. Infatti, essa consente ad un processo ad elevata priorità di entrare velocemente in possesso di una risorsa di cui necessita. Ad esempio, utilizzando questo approccio un programma per la riproduzione di sequenze video può consentire una fruizione sufficientemente "fluida" del filmato.

La tecnica può comunque essere utilizzata anche in altri tipi di Sistemi Operativi in quanto consente di realizzare una gestione della priorità che si avvicina a quella attesa dagli utenti del sistema. L'ereditarietà della priorità è ad esempio presente su molte Virtual Machine Java.