Automazione e Strumentazione, Anno XLVIII, N. 5, pp. 107-116, Maggio 2000.

Sistemi Real-Time per il Controllo Automatico: Problemi e Nuove Soluzioni

Giorgio Buttazzo
Università di Pavia
Unità di ricerca INFM - Pavia
E-mail: giorgio@sssup.it

Sommario

Questo lavoro illustra alcune medotodologie per aumentare la prevedibilità dei sistemi real-time, ossia di quei sistemi di calcolo che richiedono il controllo stretto di vincoli temporali imposti sui processi applicativi. Al fine di poter garantire off-line il rispetto di tali vincoli, tutti i meccanismi del sistema operativo, dallo scheduling alla comunicazione tra i task, dalla sincronizzazione alla gestione degli interrupt, devono essere governati da algoritmi deterministici, dei quali si possa prevedere il comportamento mediante tecniche di analisi. In questo lavoro, saranno presi in esame alcuni algoritmi per la schedulazione di task (periodici e aperiodici) e per il controllo dell'accesso a risorse mutuamente esclusive. Infine, allo scopo di fornire un supporto efficiente allo sviluppo di sistemi real-time multimediali, verranno descritti dei nuovi approcci per la gestione dei sovraccarichi e la protezione temporale tra i processi.

1. Introduzione

Spesso si dice che un sistema di controllo si comporta in tempo reale quando è in grado di reagire velocemente ad un evento. Questa definizione, tuttavia, non è del tutto corretta, in quanto la velocità di elaborazione non fornisce alcuna informazione sulla effettiva capacità del sistema di reagire "in tempo" ad un evento esterno. Se non si considerano le caratteristiche dinamiche dell'ambiente da controllare, ha poco senso parlare di velocità di reazione.

Secondo una definizione più precisa, un'attività di calcolo è detta in tempo reale se la validità dei risultati non dipende solo dalla correttezza delle singole operazioni eseguite, ma anche dal tempo entro cui i risultati sono prodotti [Sta 88]. La differenza principale tra un processo real-time e un processo non real-time è che un processo real-time deve terminare entro una scadenza temporale prefissata, che prende il nome di *deadline*. La deadline è il tempo massimo entro cui un processo deve terminare la propria esecuzione. Nelle applicazioni real-time un risultato prodotto oltre la propria deadline non è solo in ritardo, ma è dannoso. A seconda delle conseguenze provocate da una mancata deadline, i processi real-time vengono solitamente distinti in due tipi: *hard* e *soft* [StRa 88]. Un processo real-time si dice di tipo *hard* se la violazione della propria deadline comporta un effetto catastrofico sul sistema, mentre si dice di tipo *soft* se la violazione della propria deadline è indesiderata, ma non compromette il corretto funzionamento del sistema. In tal caso, il processo non è caratterizzato da una scadenza "rigida" e può essere completato anche oltre il limite di tempo specificato dalla sua deadline. Un sistema operativo in grado di gestire dei processi real-time di tipo hard è detto *hard real-time*. Per una trattazione approfondita sui sistemi hard real-time si veda [But 95][Sta 95][But 97].

In generale, i sistemi hard real-time comprendono sia attività di tipo hard, che attività di tipo soft. Tipici processi di tipo hard che si incontrano in un'applicazione di controllo automatico riguardano in genere l'acquisizione di dati sensoriali, il rilevamento di condizioni critiche, l'asservimento di attuatori, la pianificazione di azioni senso-motorie, ed il controllo di dispositivi automatici. Tipiche attività di tipo soft riguardano invece l'interprete di comandi forniti dall'utente, l'ingresso di caratteri da tastiera, la visualizzazione di messaggi su monitor, la rappresentazione dello stato del sistema, oppure attività di grafica.

L'interesse per lo studio dei sistemi real-time è motivato dalla crescente diffusione che essi hanno nella nostra società in diversi settori applicativi. Le attività che maggiormente richiedono una elaborazione in tempo reale riguardano la regolazione di impianti chimici e nucleari, il controllo di processi produttivi complessi, i sistemi di controllo di volo sugli aerei, i sistemi di controllo del traffico (aereo, ferroviario, e marittimo), i sistemi di telecomunicazione, i sistemi di regolazione delle automobili, i sistemi di acquisizione e monitoraggio ambientale,

l'automazione industriale, i sistemi militari, le missioni spaziali, la teleoperazione, la realtà virtuale, e la robotica.

Nonostante la vastità dei settori applicativi interessati, la maggior parte delle applicazioni real-time, purtroppo, viene ancora oggi progettata empiricamente, senza l'ausilio di una metodologia scientifica consolidata. La gestione di eventi temporali viene di solito affrontata scrivendo grosse porzioni di codice in linguaggio assembler e manipolando la priorità delle interruzioni e dei processi di controllo. Sebbene il codice prodotto con queste tecniche possa essere molto efficiente, questo approccio comporta numerosi svantaggi. Innanzitutto, la realizzazione di programmi complessi in linguaggio assembler è estremamente impegnativa e, in genere, richiede tempi di sviluppo più lunghi rispetto alla programmazione di alto livello; inoltre, l'efficienza e l'affidabilità del software dipendono fortemente dall'abilità del programmatore. In secondo luogo, l'ottimizzazione del codice assembler peggiora la comprensibilità del software e complica la manutenzione di grossi programmi. Infine, senza l'ausilio di una metodologia informatica, risulta praticamente impossibile garantire il rispetto dei vincoli temporali dell'applicazione sotto tutte le possibili condizioni operative.

La conseguenza di tutto ciò è che le tecniche empiriche per la progettazione e l'implementazione di sistemi in tempo reale risultano estremamente inaffidabili. Ciò è ancora più grave quando il sistema deve essere utilizzato in applicazioni critiche, che comportano dei rischi per cose o persone. Una grossa percentuale degli incidenti che si verificano negli impianti nucleari, nei voli spaziali, o nei sistemi di difesa, è spesso causata da errori sul software del sistema di controllo. In alcuni casi, tali incidenti comportano ingenti perdite economiche, o addirittura hanno conseguenze catastrofiche.

Una garanzia più robusta del funzionamento di un sistema sotto tutte le possibili condizioni operative può essere ottenuta soltanto adottando tecniche di progetto più accurate, meccanismi di nucleo predisposti a gestire esplicitamente la variabile tempo e linguaggi di programmazione più adatti alla scrittura di processi real-time. Dunque, la proprietà più importante che un sistema real-time deve possedere non è la velocità, ma la prevedibilità di risposta.

1.1 Prevedibilità di un sistema

Come si è più volte osservato, la qualità principale che deve possedere un sistema operativo real-time è la prevedibilità. In generale, per prevedibilità di un sistema si intende la capacità di determinare in anticipo se uno o più processi di calcolo potranno essere completati entro i propri vincoli temporali.

L'affidabilità della valutazione effettuata da un sistema sulla schedulabilità dei processi dipende da diversi fattori, che vanno dalle caratteristiche architetturali della macchina fisica, ai meccanismi di nucleo, fino al linguaggio di programmazione. I meccanismi interni del processore sono i primi ad influenzare il determinismo dell'esecuzione dei processi. Quelli più influenti sono le interruzioni, il DMA e la cache. Sebbene tali meccanismi migliorino le prestazioni medie del processore, essi introducono un non determinismo nell'esecuzione dei processi, che nel caso peggiore allunga i tempi di risposta di quantità indefinite. Altri fattori che influenzano l'esecuzione dei processi sono dovuti ai meccanismi di nucleo del sistema operativo, quali l'algoritmo di scheduling, il meccanismo semaforico, la gestione della memoria, e la gestione delle periferiche. Infine, anche il linguaggio di programmazione influisce sulla prevedibilità di una applicazione, a seconda dei costrutti che mette a disposizione per gestire più o meno esplicitamente le esigenze temporali dei processi. Fra tutti i fattori citati, il meccanismo di schedulazione è quello che maggiormente influenza la prevedibilità di risposta del sistema. Esso sarà pertanto oggetto dei successivi paragrafi.

2. Gestione di processi periodici

I processi periodici rappresentano la maggior parte delle attività di calcolo di un sistema di controllo real-time. Ad esempio, le attività di regolazione, acquisizione di segnali, filtraggio, elaborazione di dati sensoriali, pianificazione di azioni, monitoraggio, comando di attuatori, ecc., rappresentano processi che devono essere eseguiti con una certa periodicità, a frequenze definite dai vincoli e dai requisiti dell'applicazione.

Ogni attivazione di un processo periodico viene identificata con il termine di *istanza periodica* del processo. Per ogni istanza periodica vengono definiti l'*istante di richiesta* e la *deadline*. L'istante di richiesta $r(k)$ della k-esima istanza di un processo rappresenta l'istante in cui il processo diventa pronto per l'esecuzione per la k-esima volta. L'intervallo di tempo che intercorre tra due istanti di richiesta consecutivi corrisponde esattamente al periodo del processo. La deadline relativa alla k-esima istanza, indicata nel seguito con $d(k)$, corrisponde all'istante entro cui l'istanza deve essere completata.

2.1 Timeline scheduling

Il *Timeline Scheduling* (TS), noto anche come *cyclic executive*, è uno degli approcci maggiormente utilizzati nei sistemi di difesa militari, la navigazione automatica e il controllo di aerei. Il metodo consiste nel suddividere l'asse temporale in intervalli di tempo uguali (*time slices*) nei quali allocare opportunamente una o più procedure, in modo da rispettare le frequenze richieste da ogni procedura. Un timer sincronizza l'attivazione delle procedure all'inizio di ogni time slice. Al fine di illustrare il metodo, si consideri il seguente esempio, in cui tre procedure, A, B e C, devono essere eseguite con frequenze pari a 40, 20 e 10 Hz rispettivamente. Analizzando i periodi delle attività, è facile verificare che la lunghezza ottimale per le time slice risulta essere di 25 ms, ossia pari al Massimo Comun Divisore tra i periodi. Dunque, per rispettare le frequenze richieste, la procedura A dovrà essere eseguita in ogni time slice, la procedura B ogni due, e la procedura C ogni quattro. Una possibile soluzione dello scheduling è illustrata in Figura 1.

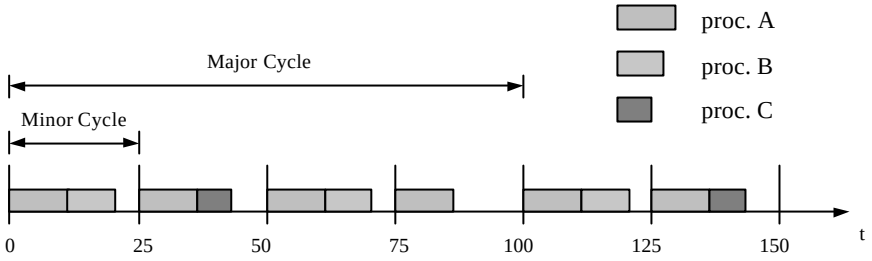

Figura 1: Esempio di timeline scheduling.

La durata del time slice è detta *Minor Cycle*, mentre il periodo minimo dopo il quale la schedulazione si ripete è detto *Major Cycle*. In generale, esso è pari al minimo comune multiplo tra i periodi (nell'esempio vale 100 ms).

Per garantire a priori che la schedulazione sia fattibile su un particolare processore, è sufficiente valutare i tempi di esecuzione delle procedure e verificare che la somma delle esecuzioni in ogni time slice sia minore o uguale del minor cycle. Nel nostro esempio, detti C_A, C_B e C_C i tempi di esecuzioni delle procedure, è sufficiente verificare che

$$C_A + C_B \leq 25 \text{ ms}$$

$$C_A + C_C \leq 25 \text{ ms}$$

Il pregio più evidente del timeline scheduling è dato dalla semplicità di realizzazione. Per implementare il metodo è sufficiente programmare un timer con un valore pari al minor cycle e scrivere un programma che richiami nell'ordine le procedure all'interno del major cycle, inserendo dei punti di sincronizzazione all'inizio di ogni minor cycle. Poiché la successione delle procedure non è dettata da un algoritmo di scheduling del nucleo, ma dipende dalle chiamate effettuate dal programma principale, non esistono commutazioni di contesto, per cui l'overhead introdotto è estremamente basso. Inoltre, la sequenza delle attività è deterministica, facilmente visualizzabile e, soprattutto, non è affetta da fenomeni di jitter.

Nonostante i vantaggi descritti, apprezzabili soprattutto nelle applicazioni critiche in cui i processi non sono soggetti a troppe variazioni e aggiornamenti, la tecnica del timeline scheduling presenta anche dei grossi inconvenienti, tra cui una fragilità nei sovraccarichi. Se una procedura non termina entro il tempo previsto possiamo farla continuare oppure abortirla; in entrambi i casi, però, rischiamo di mettere il sistema in una situazione di pericolo. Infatti, lasciando la procedura "anomala" in esecuzione rischiamo di superare il limite imposto dal Minor Cycle, causando un effetto domino su tutte le attività successive (*timeline break*). Se invece decidiamo di abortire la procedura rischiamo che questa rimanga in uno stato inconsistente, compromettendo il corretto funzionamento del sistema.

Un altro grosso problema di questa tecnica è la scarsa flessibilità. Se l'aggiornamento di una procedura causa una variazione del tempo di calcolo o della frequenza di attivazione, c'è il rischio che l'intera sequenza di schedulazione debba essere riprogettata da zero. Facendo riferimento all'esempio precedente, se la procedura B viene aggiornata e si ha che $C_A + C_B > 25$ ms, occorre suddividere B in due pezzi, B1 e B2, da allocare opportunamente negli spazi disponibili del Timeline. Un aggiornamento delle frequenze può causare cambiamenti ancora più radicali nella schedualzione. Ad esempio, se la frequenza di B dovesse passare da 20 Hz a 25 Hz, il precedente Timeline verebbe ad essere completamente stravolto, poiché nella nuova condizione il Minor Cycle risulterebbe pari a 10 ms e il Major Cycle pari a 200 ms. Infine, un'ultima limitazione del Timeline Scheduling consiste nella difficoltà di gestire efficientemente le attività aperiodiche attivate da eventi esterni.

I problemi sopra descritti possono essere superati elegantemente per mezzo di algoritmi a base prioritaria.

2.2 Rate Monotonic (RM)

L'algoritmo Rate-Monotonic (RM) consiste nell'assegnare ad ogni processo una priorità direttamente proporzionale alla propria frequenza, in modo che ai processi con periodi più brevi venga assegnata una priorità più elevata. Poiché i periodi dei processi sono assunti costanti, l'algoritmo RM è un algoritmo di tipo statico, nel senso che le priorità assegnate ai processi sono fissate all'inizio e rimangono invariate per tutta la durata dell'applicazione. Inoltre RM è un algoritmo intrinsecamente di tipo preemptive.

Nel 1973, Liu e Layland [Liu 73] hanno dimostrato che l'algoritmo RM è ottimo, nel senso che se un insieme di processi non è schedulabile con RM, allora l'insieme non è schedulabile con nessun'altra regola di assegnazione di priorità fisse. Un altro risultato importante ricavato dagli stessi autori è che un insieme $\Gamma = \{\tau_1, \ldots, \tau_n\}$ di n task periodici risulta schedulabile con RM se:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

dove C_i e T_i rappresentano rispettivamente il tempo di esecuzione massimo e il periodo del processo τ_i. La quantità

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

viene detta *utilizzazione del processore* e rappresenta la percentuale di tempo impiegata dal processore ad eseguire l'insieme di task. La Tabella 1 mostra i valori di $n(2^{1/n} - 1)$ per n che va da 1 a 10. Come si può notare, l'espressione assume valori decrescenti al crescere di n e, per n tendente all'infinito, ha un limite pari a:

$$\lim_{n \to \infty} n\left(2^{1/n} - 1\right) = \ln 2 \cong 0.69$$

Si noti che, essendo il test di schedulabilità solo sufficiente, l'insieme di task può essere schedulabile con RM anche se la condizione su U non è soddisfatta. Tuttavia, possiamo affermare che un insieme di task periodici è sicuramente non schedulabile se risulta U > 1.

n	$n(2^{1/n} - 1)$
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
7	0.729
8	0.724
9	0.721
10	0.718

Tabella 1: Utilizzazione massima del processore con RM.

Uno studio statistico effettuato da Lehoczky, Sha, e Ding [Leh 89] ha dimostrato che su insiemi di processi generati casualmente il fattore di utilizzazione di insiemi schedulabili vale mediamente 0.88, e diventa unitario quando i periodi dei processi sono legati tra loro da relazioni armoniche.

Nonostante tale limitazione, l'algoritmo RM è ampiamente utilizzato nelle applicazioni real-time, sia per la sua semplicità che per la facilità con cui i risultati illustrati possono essere estesi a insiemi di processi più generali di quelli finora trattati. Al tempo stesso, essendo RM un algoritmo di schedulazione che opera con priorità statiche, può essere facilmente implementato in sistemi esistenti in cui i processi sono gestiti su base prioritaria. Per questo motivo, presso il Software Engineering Institute di Pittsburgh è stato predisposto un vero e proprio prontuario da utilizzare come sorgente di informazioni e come guida rivolta a chiunque debba analizzare o progettare un sistema real-time che operi con la strategia Rate-Monotonic [Kle 93].

Poiché l'algoritmo RM è ottimo tra tutti gli algoritmi a priorità fissa, il limite superiore minimo del fattore di utilizzazione del processore può essere migliorato solo attraverso un'assegnazione dinamica delle priorità.

2.3 Earliest Deadline First (EDF)

L'algoritmo Earliest Deadline First (EDF) consiste nel selezionare dalla lista dei processi pronti quello la cui deadline assoluta è più imminente. L'algoritmo EDF è di tipo *preemptive*, nel senso che, se arriva un processo con una deadline minore di quella del processo in esecuzione, quest'ultimo viene sospeso e la CPU viene assegnata al processo con deadline più corta appena arrivato.

Se il sistema operativo non supporta esplicitamente la gestione di vincoli temporali, anche questo algoritmo (come RM) può essere implementato in un sistema a base prioritaria, dove le priorità sono assegnate ai processi in modo dinamico. Un processo che ha fatto richiesta riceve la massima priorità se la sua deadline è la più vicina tra tutte quelle dei processi pronti, mentre riceve la minima priorità se la sua deadline è la più lontana. Ciò equivale ad assegnare ad un processo una priorità inversamente proporzionale alla sua deadline assoluta.

A differenza dell'algoritmo RM, EDF può essere utilizzato per schedulare sia un insieme di processi periodici, che un insieme di processi aperiodici, poiché la selezione di un processo si basa unicamente sul valore della sua deadline. Un processo ciclico che ha terminato in tempo la sua esecuzione, si sospende fino alla propria deadline, che coincide con la fine del periodo corrente, dopodiché ridiventa pronto per il periodo successivo. Dertouzos [Der 74] ha dimostrato che EDF è un algoritmo ottimo fra tutti gli algoritmi on line, e Liu e Layland [Liu 73] hanno provato che un insieme $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ di *n* processi periodici è schedulabile con EDF *se e solo se*:

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

E' importante notare che, in questo caso, la condizione di schedulabilità risulta necessaria e sufficiente. Ciò significa che, se il fattore di utilizzazione dell'insieme di task è maggiore di uno, non esiste alcun algoritmo in grado di produrre una schedulazione fattibile.

La proprietà di variare dinamicamente la priorità dei processi, consente a EDF di utilizzare appieno il processore, sfruttando il 100% del tempo disponibile. Anche se i processi hanno una utilizzazione minore di uno, la frazione di tempo residua può essere efficientemente impiegata per gestire richieste aperiodiche attivate da eventi esterni. Inoltre, EDF genera un minor numero di cambi di contesto rispetto a RM e, quindi, un minore overhead a runtime. Per contro, RM risulta più semplice da realizzare su un sistema operativo a priorità fisse e, inoltre, RM è più prevedibile nelle situazioni di sovraccarico, poichè i task a maggiore priorità sono sempre i più privilegiati.

2.4 Processi con deadline minore del periodo

Utilizzando l'algoritmo Rate Monotonic, o Earliest Deadline First, un processo periodico può essere eseguito in qualunque intervallo di tempo all'interno del periodo. L'unica garanzia che si ha effettuando il test di schedulabilità è che ciascun processo riesca a completare la sua esecuzione entro il proprio periodo. In alcune applicazioni, tuttavia, si desidera che i processi ciclici, oltre ad essere attivati con un periodo prefissato, siano eseguiti entro un intervallo di tempo minore del proprio periodo.

L'algoritmo Deadline Monotonic (DM), ideato da Leung e Whitehead nel 1982 [Leu 82], rappresenta un'estensione di Rate Monotonic, in quanto consente la schedulazione di processi periodici con deadline indipendente dal periodo.

Secondo l'algoritmo DM, il processo che in ogni istante viene schedulato è quello con la deadline relativa più corta. Nei sistemi a base prioritaria, ciò equivale ad assegnare ad ogni processo una priorità p_i inversamente proporzionale alla propria deadline relativa:

$$P_i \ \mu \ 1/D_i$$

Essendo D_i fissata per ciascun processo, l'algoritmo DM è un algoritmo statico di tipo preemptive.

Un metodo per trovare una condizione necessaria e sufficiente che garantisca la schedulabilità di un insieme di processi periodici con l'algoritmo DM è stato proposto da Audsley [Aud 92]. Tale metodo si basa sul calcolo del tempo di risposta R_i di ciascun task periodico, dato dal tempo di calcolo proprio del task (C_i) sommato all'interferenza dei task a priorità più elevata:

$$R_i = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

dove *hp(i)* rappresenta l'insieme dei processi a priorità più elevata di τ_i.
Essendo l'espressione di tipo ricorsivo, essa si risolve in modo iterativo, partendo con $R_i^{(0)} = C_i$ e terminando quando $R_i^{(s)} = R_i^{(s-1)}$. Se poi risulta $R_i^{(s)} > D_i$, allora vuol dire che la schedulazione non è fattibile.

Sotto EDF, l'analisi di schedulabilità per task periodici con deadline minore del periodo è basata sul criterio della domanda di calcolo (*processor demand criterion*), ideato da Baruah [Bar 90]. Secondo tale metodo, un insieme di task periodici è schedulabile con EDF se e solo se, in ogni intervallo L, la domanda di calcolo non supera il tempo disponibile, ossia se e solo se:

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i \;\leq\; L$$

3. Gestione di processi aperiodici

Sebbene in un sistema real-time la maggior parte dei processi di acquisizione e di controllo sia di tipo periodico, esitono delle attività che devono essere svolte solo al verificarsi di eventi esterni (in genere segnalati mediante interruzione), che possono verificarsi con cadenze irregolari, aperiodiche. Quando nel sistema arrivano delle richieste di calcolo di tipo aperiodico, ci si trova di fronte a due esigenze contrastanti: da un lato, infatti, si desidera rispondere all'evento esterno nel più breve tempo possible, dall'altro però non si vuole compromettere la schedulabilità dei processi periodici consentendo che uno o più task vengano ritardati oltre la loro deadline.

Se le attività aperiodiche sono di tipo soft, l'obiettivo dell'algoritmo di scheduling è in genere quello di minimizzare il loro tempo di risposta, garantendo che tutti i task periodici (pur essendo ritardati dal servizio aperiodico) completino la loro esecuzione entro le rispettive deadline. Se alcune attività aperiodiche sono di tipo hard, si cerca di garantire off-line che la loro esecuzione termini entro le deadline specificate. Tale garanzia può essere fatta solo ipotizzando che le richieste, pur arrivando ad intervalli irregolari, non superino una frequenza massima prestabilita, ovvero siano intervallate da un minimo tempo di interarrivo (*minimum interarrival time*). Un processo aperiodico caratterizzato da un minimo tempo di interarrivo è detto *sporadico*.

La Figura 2 illustra un esempio in cui una richiesta aperiodica J_a di 3 unità di tempo debba essere eseguita insieme a due task periodici, con tempi di calcolo $C_1 = 1$, $C_2 = 3$ e periodi $T_1 = 4$, $T_2 = 6$, schedulati con RM. Come mostrato in figura, se la richiesta aperiodica viene servita immediatamente (ossia con priorità superiore a quella dei task periodici), il task τ_2 supera la deadline di due unità di tempo.

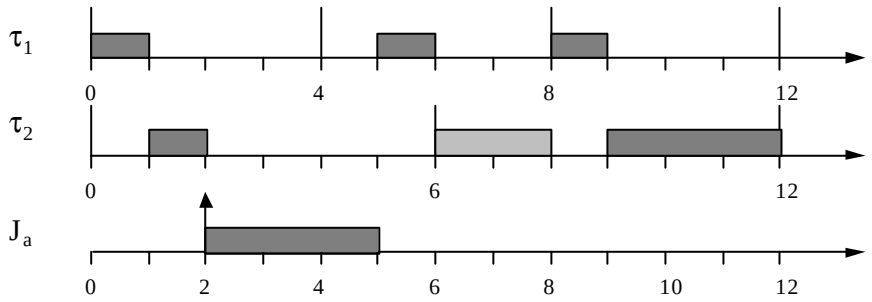

Figura 2: Gestione immediata di un task aperiodico.

La tecnica più semplice per gestire le attività aperiodiche preservando la garanzia sui processi periodici è quella di schedulare le richieste aperiodiche in background. La gestione in background comporta che un task aperiodico sia schedulato solo quando il processore è libero (*idle*), ossia quando non ci sono processi periodici in esecuzione. Ciò significa che, se il carico computazionale relativo ai processi periodici è elevato, il tempo residuo per l'esecuzione delle attività aperiodiche può essere insufficiente a soddisfare le esigenze dei processi, causando tempi di risposta troppo lunghi.

La Figura 3 illustra la schedualzione sull'insieme di task dell'esempio precedente nel caso in cui il task aperiodico sia servito in background.

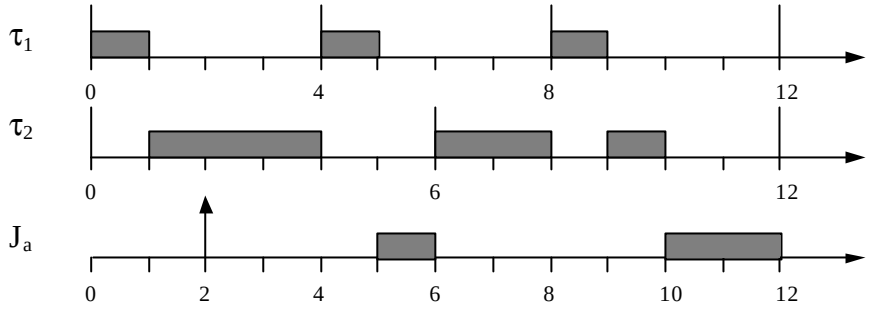

Figura 3: Gestione in background di un task aperiodico.

Il tempo di risposta dei processi aperiodici può essere migliorato associando un processo periodico (*server*) dedicato al servizio delle richieste aperiodiche. Come un qualsiasi altro processo periodico, un server è caratterizzato da un periodo T_s e da un tempo di esecuzione C_s, detto *capacità* del server.

In generale, il server viene schedulato con lo stesso algoritmo utilizzato per i processi periodici e, una volta attivato, esso effettua il servizio delle richieste aperiodiche pendenti, entro i limiti della sua capacità C_s. L'ordine di servizio delle richieste aperiodiche è indipendente dall'algoritmo di schedulazione usato per i processi periodici, e può essere deciso in funzione del tempo di arrivo, del tempo di calcolo, oppure della deadline.

In letteratura, sono stati proposti numerosi algoritmi di servizio per task aperiodici, caratterizzati da prestazioni e complessità differenti. Tra gli algoritmi basati su priorità fisse ricordiamo il *Polling Server* e il *Deferrable Server* [Leh 87][Str 95], lo *Sporadic Server* [Spr 89], e lo *Slack Stealer* [Leh 92]. Tra i server a priorità dinamica (mediamente più efficienti) ricordiamo il *Dynamic Sporadic Server* [Spu 94][Gha 95], il *Total Bandwidth Server* [Spu 96], il *Tunable Bandwidth Server* [BuSe 97] ed il *Constant Bandwidth Server* [Abe 98].

Per chiarire l'idea alla base del servizio aperiodico mediante server, la Figura 4 illustra la schedulazione EDF prodotta da un *Dynamic Deferrable Server* con capacità $C_s = 1$ e periodo $T_s = 4$. Si noti che, a parità di deadline assoluta tra il server e un task periodico, la precedenza viene data al server per favorire le richieste aperiodiche. Si noti che lo stesso insieme di task non sarebbe schedulabile utilizzando uno schema a priorità fisse.

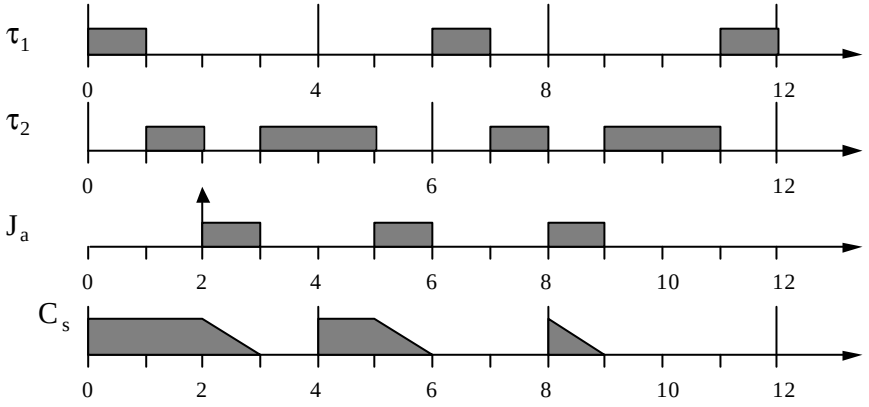

Figura 4: Servizio aperiodico con Dynamic Deferrable Server.

Sebbene il tempo di risposta del task aperiodico gestito con il server sia minore di quello ottenuto con il servizio in background, esso non rappresenta il minimo tempo possibile. Il tempo di risposta minimo può essere ottenuto con un algoritmo ottimo (TB*) che assegna alla richiesta aperiodica una opportuna deadline, la quale, sotto EDF, consente di minimizzare il tempo di risposta del task rispettando la garanzia dei processi periodici [BuSe 97]. La schedulazione generata dall'algoritmo ottimo è illustrata in Figura 5, dove si vede che il tempo di risposta minimo per la richiesta J_a è di 5 unità di tempo.

Come in tutte le soluzioni efficienti, la migliore prestazione ottenuta dall'algoritmo ottimo si paga in termini di complessità computazionale (e quindi di overhead). Tuttavia, utilizzando un *Tunable Bandwidth Server* [BuSe 97] è possibile bilanciare costo e prestazioni, in modo da adattare il criterio di servizio alle esigenze della specifica applicazione real-time. Una rassegna dei principali algoritmi di servizio per attività aperiodiche (sia a priorità statica che a priorità dinamica) è riportata in [But 97].

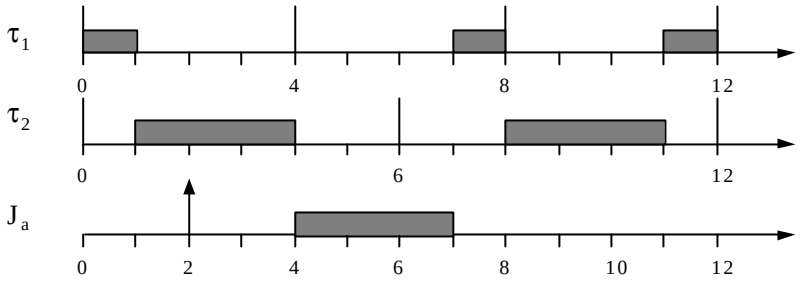

Figura 5: Servizio aperiodico ottimo.

4. Protocolli di accesso a risorse condivise

Quando più processi interagiscono mediante risorse comuni, la diretta applicazione dei meccanismi classici di sincronizzazione, come i semafori o i monitor, può provocare un fenomeno noto come *inversione di priorità*, secondo cui un processo ad alta priorità viene bloccato da un processo a più bassa priorità per un intervallo di tempo indefinito. Tale situazione crea dei seri problemi nei sistemi real-time, poiché può compromettere la schedulabilità del sistema.

Si considerino, ad esempio, tre processi τ_1, τ_2 e τ_3 con priorità decrescenti (τ_1 è il processo a maggiore priorità), e si assuma che τ_1 e τ_3 condividano una struttura dati protetta da un semaforo binario S. Con riferimento alla Figura 6, supponiamo che al tempo t_1 il processo τ_3 entri in sezione critica, occupando il semaforo S. Durante l'esecuzione di τ_3, all'istante t_2, arriva la richiesta di τ_1 il quale, essendo a priorità maggiore, effettua preemption su τ_3.

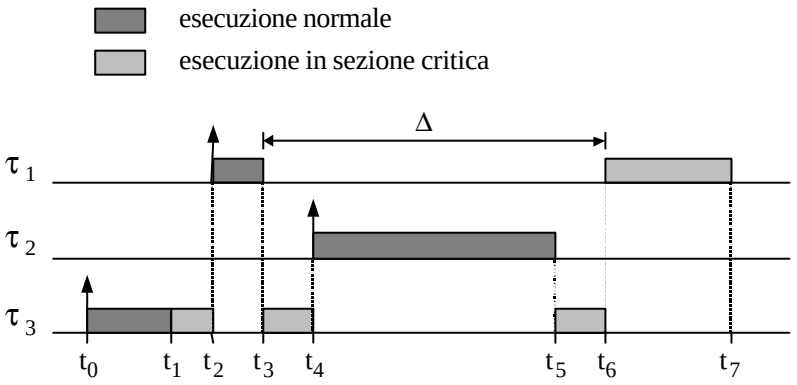

Figura 6: Esempio di inversione di priorità.

Nell'istante t_3 il processo τ_1 tenta di usare la risorsa condivisa, ma viene bloccato sul semaforo S poiché la risorsa è occupata da τ_3. Essendo τ_1 il processo a priorità più elevata, ci si aspetta che esso rimanga bloccato per un tempo non superiore al tempo necessario a τ_3 per completare la sezione critica. Purtroppo, invece, il tempo di bloccaggio di τ_1 può diventare imprevedibile!

Infatti, il processo τ_3 può subire preemption da parte del processo a priorità intermedia τ_2. Di conseguenza, la durata Δ del bloccaggio di τ_1 dipende non solo dalla lunghezza della sezione critica eseguita da τ_3, ma anche dall'esecuzione di τ_2 e di altri eventuali processi a priorità intermedia che possono inserirsi nel mezzo.

La situazione illustrata in Figura 6 può essere evitata se si impedisce la preemption sui processi che si trovano all'interno di sezioni critiche. Questa soluzione, tuttavia, può essere appropriata solo nel caso di sezioni critiche molto brevi, poiché altrimenti potrebbe dar luogo a inutili tempi di attesa. Ad esempio, un processo a bassa priorità dentro una sezione critica molto lunga, impedirebbe per lungo tempo l'esecuzione di un processo ad alta priorità, anche nel caso in cui questo non usi risorse condivise.

Una soluzione più efficiente consiste nel regolare l'accesso alle risorse condivise mediante l'uso di appositi protocolli di accesso ai quali viene demandato il compito di evitare condizioni di inversione di priorità.

4.1 Protocollo di Priority Inheritance

Una soluzione al problema di inversione della priorità causato dalla mutua esclusione tra processi viene offerta dal protocollo di *Priority Inheritance* (PI) [Sha 90]. Tale metodo consiste nel modificare dinamicamente la priorità dei processi che causano il bloccaggio. In particolare, quando un processo τ_a si blocca all'entrata di una sezione critica, trasmette la sua priorità al processo τ_b che si trova all'interno della stessa. In generale, τ_b eseguirà la sua sezione critica con una priorità pari alla massima fra le priorità di tutti i processi da esso bloccati. Si dice che τ_b *eredita* la massima priorità fra i processi da esso bloccati. L'ereditarietà della priorità è transitiva, per cui se un processo τ_c blocca τ_b, il quale blocca τ_a, allora τ_c eredita la priorità di τ_a attraverso τ_b.

La Figura 7 illustra come si modifica la schedulazione di Figura 6 nell'ipotesi in cui l'accesso alle sezioni critiche sia regolato dal protocollo PI. Fino all'istante t_3 l'evoluzione dei processi è identica a quella di Figura 6. All'istante t_3, il processo τ_1 ad alta priorità si blocca dopo aver tentato l'accesso alla sezione critica occupata da τ_3 (*direct blocking*). In questo caso, però, il protocollo impone che τ_3 erediti la massima priorità dei processi da esso bloccati, e quindi esegua la sua sezione critica alla priorità di τ_1. In queste condizioni, nell'istante t_4, il processo τ_2 non è più in grado di effettuare preemption su τ_3, per cui si blocca (*push-through blocking*).

In altre parole, pur avendo una priorità nominale maggiore di τ_3, τ_2 non può andare in esecuzione, poiché τ_3 ha ereditato la priorità di τ_1. Quando, al tempo t_5, τ_3 termina la sua sezione critica, esso riacquista la sua priorità nominale e sveglia τ_1 che era in attesa sul semaforo. Avendo una priorità maggiore di τ_3, il processo τ_1 va in esecuzione fino al suo completamento, che avviene all'istante t_6. Solo in questo istante τ_2 può riprendere l'esecuzione e terminare.

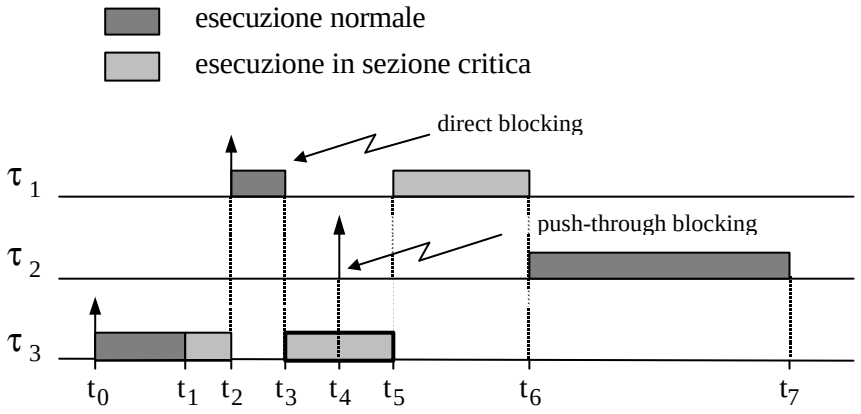

Figura 7: Esecuzione dei processi dell'esempio precedente con l'uso del protocollo di Priority Inheritance.

Il protocollo PI gode della seguente proprietà [Sha 90]:

> Se *n* è il numero di processi a priorità più bassa di τ, ed *m* è il numero di semafori che possono bloccare τ, allora τ può essere bloccato al massimo per la durata di *min(n,m)* sezioni critiche.

Nonostante il protocollo PI limiti il fenomeno di inversione di priorità, esso presenta ancora due problemi rilevanti. Il primo è che il protocollo non previene le condizioni di stallo (*deadlock*) tra processi. Il secondo è che la durata del bloccaggio, se pur limitata, può essere significativa a causa di un possibile bloccaggio a catena.

4.2 Protocollo di Priority Ceiling

Il protocollo di *Priority Ceiling* (PC) [Sha 90] consente di risolvere il problema dell'inversione di priorità tra processi che accedono a sezioni critiche condivise ed inoltre riesce ad evitare sia le condizioni di stallo che le catene di bloccaggi.

L'idea di base di questo protocollo è quella di assicurare che, quando un processo τ entra in una sezione critica, la sua priorità sia più alta di tutte le priorità ereditabili dai processi correntemente sospesi in sezioni critiche. Se questa condizione non può essere soddisfatta, il protocollo impedisce al processo τ di entrare in sezione critica, e il processo che blocca τ eredita la priorità di τ.

Questa idea viene realizzata assegnando a ciascun semaforo un tetto di priorità (*priority ceiling*) pari alla più alta priorità dei processi che possono usare quel semaforo. Quindi si consente ad un processo τ di entrare in

sezione critica solo se la sua priorità è strettamente maggiore di tutti i tetti di priorità associati ai semafori bloccati dai processi diversi da τ. Come per PI, il meccanismo di ereditarietà della priorità è transitivo.

Il procollo di PC, oltre a prevenire lo stallo e le catene di bloccaggi, gode della seguente proprietà: un processo può essere bloccato al massimo per la durata di una sola sezione critica.

4.3 Analisi della schedulabilità

L'importanza dei protocolli di accesso a risorse condivise nei sistemi hard real-time risiede nel fatto che essi sono in grado di fornire un limite superiore sul tempo di bloccaggio di un processo. Ciò è importante per analizzare la schedulabilità di un insieme di processi real-time che interagiscono mediante sezioni critiche.

Allo scopo di verificare la schedulabilità del processo τ_i si devono considerare, oltre al fattore di utilizzazione proprio del processo, sia gli effetti della preemption causata dai processi a più alta priorità, sia gli effetti del bloccaggio causato dai processi a più bassa priorità. Detto B_i il tempo massimo di bloccaggio che può subire il processo τ_i, si ha che con l'algoritmo Rate Monotonic, la somma degli effetti dovuti alla preemption, all'utilizzazione del processore e al bloccaggio non deve superare il fattore di utilizzazione limite imposto dall'algoritmo, ovvero deve rispettare la condizione:

$$\forall i = 1,\ldots,n \qquad \frac{C_i}{T_i} + \sum_{k \in hp(i)} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i\left(2^{1/i} - 1\right)$$

dove *hp(i)* rappresenta l'insieme dei processi a priorità più elevata di τ_i. Si noti che questo metodo di garanzia è valido per entrambi i protocolli considerati, la differenza risiede soltanto nel tempo massimo di bloccaggio B_i che ogni processo può subire.

5. Nuove problematiche

Oltre ai sistemi di controllo critici, esistono numerose applicazioni real-time caratterizzate da vincoli temporali meno stringenti, in cui il superamento occasionale di una deadline viene generalmente tollerato. Esempi di sistemi con queste caratteristiche sono i sistemi di monitoraggio, i sistemi multimediali, i simulatori di volo o, in generale, i giochi di realtà virtuale. In queste applicazioni, il superamento di una deadline non provoca effetti catastrofici, ma causa solo un degrado di prestazioni. Dunque, più che una garanzia assoluta, in questi sistemi si richiede una *Qualità di Servizio* (QoS) accettabile. Poiché i vincoli temporali devono essere comunque gestiti, un sistema operativo non real-time, tipo Linux o Windows, risulta inadeguato per diversi motivi. Innanzitutto, in tali sistemi manca un isolamento temporale tra i task, per cui un picco di carico su un task si ripercuote sugli altri in modo incotrollato. Inoltre, la mancanza di meccanismi che prevengano l'inversione di priorità rende tali sistemi inadeguati a garantire un livello desiderato di QoS.

D'altra parte, un sistema hard real-time può risultare inadeguato per lo spreco di risorse che deriva dalla preallocazione statica di risorse (necessaria per avere un'elevata prevedibilità). Inoltre, nelle applicazioni multimediali, i processi sono caratterizzati da una grande variabilità dei tempi di esecuzione (si pensi ad esempio ad un player mpeg), pertanto fornire una stima accurata dei tempi di calcolo e dei tempi di interarrivo di processi aperiodici risulta praticamente impossibile, a meno di non utilizzare valori estremamente pessimistici.

Allo scopo di fornire un supporto efficiente, ma pur sempre prevedibile, a questo tipo di applicazioni real-time, sono stati proposti nuovi modelli di processo e nuove metodologie di scheduling in grado di rendere il sistema più flessibile e adattivo alle variazioni on-line. Ad esempio, nuovi meccanismi di protezione temporale sono stati proposti per isolare gli overrun dei task e ridurre l'interferenza reciproca fra i processi [Sto 96][Abe 98]. Tecniche statistiche di analisi sono state introdotte per fornire una garanzia probabilistica mirata a migliorare l'efficienza del sistema [Abe 98].

Altre tecniche sono state ideate per gestire i sovraccarichi in modo controllato, al fine di aumentare il carico computazionale medio del sistema e sfruttare al meglio le risorse della macchina. Una di queste tecniche consiste nell'assorbire i sovraccarichi abortendo uniformemente alcune istanze di task periodici, senza superare un limite massimo specificato dall'utente mediante un parametro, che descrive la minima QoS accettabile per quel task in termini di numero minimo di istanze tra due aborti [Kor 95][But 99]. Un'altra tecnica consiste nel gestire il sovraccarico attraverso un allungamento opportuno dei periodi, in modo da abbassare l'utilizzazione del processore e degradare le prestazioni in modo controllato [But 98].

6. Conclusioni

In questo lavoro sono state descritte alcune metodologie di nucleo mirate a migliorare l'affidabilità e la prevedibilità delle applicazioni di controllo real-time. Sono state descritte tecniche di analisi e algoritmi di schedulazione per task periodici, meccanismi di servizio per attività aperiodiche e protocolli di accesso a risorse condivise che limitano il fenomeno di inversione di priorità. Ciascuna tecnica descritta ha la proprietà di essere analizzabile, in modo da poter fornire una garanzia off-line sulla fattibilità della schedulazione entro i vincoli temporali imposti dall'applicazione.

Per sistemi real-time di tipo soft, quali i sistemi multimediali o i simulatori, le metodologie hard real-time possono risultare troppo rigide e inefficienti, specialmente quando l'applicazione è caratterizzata da parametri soggetti ad una certa variabilità. In questi casi, esistono tecniche per migliorare l'efficienza nel caso medio, garantendo un adeguato livello di qualità del servizio e un degrado controllato delle prestazioni nei sovraccarichi.

Nonostanze la ricerca continui ad investigare offrendo nuove soluzioni a problemi sempre più complessi, al fine di ottenere un reale aumento di affidabilità nei sistemi real-time futuri, è neccessario che le nuove metodologie vengano poi integrate nei sistemi operativi della nuova generazione, definendo nuovi standard per lo sviluppo di sistemi real-time. Infine, affiché si abbia un aumento effettivo della qualità e dell'affidabilità dei futuri sistemi, è altresì importante che i programmatori vengano educati all'utilizzo appropriato delle nuove tecnologie.

Bibliografia

[Abe 98] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems", Proceedings of the IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998.

[Aud 92] N. C. Audsley, A. Burns, M. Richardson, and A. Wellings: "Hard Real-Time Scheduling: The Deadline Monotonic Approach", *IEEE Workshop on Real-Time Operating Systems*, 1992.

[Bar 90] S. K. Baruah, R. R. Howell, and L. E. Rosier, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor," *Real-Time Systems*, 2, 1990.

[But 95] Giorgio Buttazzo: "Sistemi in Tempo Reale", *Pitagora Editrice*, Bologna, 1995.

[But 97] G. Buttazzo, "HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications", Kluwer Academic Publishers, Boston, 1997.

[BuSe 97] G. C. Buttazzo and F. Sensini, "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments", Proceedings of the 3rd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Como, Italy, September 1997.

[But 98] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control", Proceedings of the IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998.

[But 99] G. Buttazzo and M. Caccamo, "Minimizing Aperiodic Response Times in a Firm Real-Time Environment", IEEE Transactions on Software Engineering, Vol. 25, No. 1, pp. 22-32, January/February 1999.

[Der 74] M. L. Dertouzos: "Control Robotics: the Procedural Control of Physical Processes", *Information Processing 74*, North-Holland Publishing Company, 1974.

[Gha 95] T. M. Ghazalie and T. P. Baker: "Aperiodic Servers In A Deadline Scheduling Environment". *The Journal of Real-Time Systems*, 1995.

[Kle 93] M.H. Klein, et al. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer Academic Publishers, 1993.

[Kor 95] G. Koren and D. Shasha, "Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips", *Proceedings of IEEE Real-Time System Symposium*, December 1995.

[Leh 87] J. P. Lehoczky, L. Sha, and J. K. Strosnider: "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 261-270, San Jose, CA, December 1987.

[Leh 89] J. P. Lehoczky, L. Sha, and Y. Ding: "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour", *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166-171, 1989.

[Leh 92] J. P. Lehoczky and S. Ramos-Thuel: "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems", *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[Leu 82] J. Leung and J. Whitehead: "On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks", *Performance Evaluation*, 2(4), pp. 237-250, 1982.

[Liu 73] C. L. Liu and J. W. Layland: "Scheduling Algoritms for Multiprogramming in a Hard-Real-Time Environment", *Journal of ACM*, Vol. 20, No. 1, January 1973.

[Raj 91] R. Rajkumar, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishing, 1991.

[Sha 90] L. Sha, R. Rajkumar, and J. P. Lehoczky: "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990.

[Spr 89] B. Sprunt, L. Sha, and J. Lehoczky: "Aperiodic Task Scheduling for Hard Real-Time System", *Journal of Real-Time Systems*, 1, pp. 27-60, June 1989.

[Spu 94] M. Spuri and G. C. Buttazzo: "Efficient Aperiodic Service under Earliest Deadline Scheduling", *Proceedings of the 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, 1994.

[Spu 96] M. Spuri and G. C. Buttazzo: "Scheduling Aperiodic Tasks in Dynamic Priority Systems", Journal of Real-Time Systems, Vol. 10, No. 2, pp. 1-32, 1996.

[StRa 88] John Stankovic e Krithi Ramamritham, *Tutorial on Hard Real-Time Systems*, IEEE Computer Society Press, 1988.

[Sta 88] J. Stankovic: "A Serious Problem for Next-Generation Systems", *IEEE Computer*, pp. 10-19, October 1988.

[Sta 95] J. Stankovic, M. Spuri, M. Di Natale, G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems", *IEEE Computer*, Vol. 28, No. 6, pp. 16-25, June 1995.

[Sto 96] I. Stoica, H-Abdel-Wahab, K. Jeffay, S. Baruah, J.E. Gehrke, and G. C. Plaxton, "A Proportional Share Resource Allocation Algorithm for Real-Time Timeshared Systems", *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1996

[Str 95] J. K. Strosnider, J. P. Lehoczky and L. Sha: "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", *IEEE Transactions on Computers*, Vol. 44, No. 1, pp. 73-91, January 1995.