


SERT - 02/10/2020 - Modello di riferimento per i sistemi real-time - R03

Di cosa parliamo in questa lezione?

In questa lezione esaminiamo una classe di algoritmi di schedulazione dal comportamento deterministico e facili da validare

- 1 Algoritmi clock-driven
- 2 Cyclic executive
- 3 Suddivisione del tempo in frame
- 4 Gestione dei job aperiodici

Schedulazione clock-driven
Marco Cesati



Schema della lezione
Algoritmi clock-driven
Cyclic executive
Job aperiodici soft r.-t.
Job aperiodici hard r.-t.

SERT'20 R3.2

Oggi parliamo di scheduler clock-driven. Sono fondamentalmente una classe di algoritmi di schedulazione molto semplici come formulazione, molto semplici da implementare.

Tipologie di algoritmi per la schedulazione real-time

Esistono e vengono utilizzati un gran numero di differenti algoritmi per la schedulazione nei sistemi real-time

La maggior parte di essi possono essere ricondotti a tre grandi famiglie:

- 1 Algoritmi **clock-driven**
- 2 Algoritmi **weighted round-robin**
- 3 Algoritmi **priority-driven**

In questa lezione parliamo di algoritmi **clock-driven**

**Schedulazione
clock-driven**
Marco Cesati



Schema della lezione
Algoritmi clock-driven
Cyclic executive
Job aperiodici soft r.-t.
Job aperiodici hard r.-t.

SERT'20 R3.3

In effetti esistono tanti tipi di algoritmi di schedulazione. Gli algoritmi di tipo **clock-driven** che vediamo oggi. Gli algoritmi **round-robin pesati**, che sono tipicamente utilizzati nei sistemi operativi general purpose ma che effettivamente sono una classe di algoritmi più grande. In generale, per i sistemi RT moderni, si tende oggi ad usare algoritmi clock-driven. In effetti oggi parliamo di algoritmi clock-driven.

Algoritmi clock-driven

Un algoritmo di schedulazione è detto essere *clock-driven* se le decisioni riguardanti i job da eseguire e gli intervalli di tempo in cui questi devono rimanere in esecuzione sono determinate in anticipo (off-line) e adottate in istanti di tempo predefiniti

Tipicamente, in un sistema schedulato con un algoritmo *clock-driven*:

- Gli istanti in cui lo scheduler interviene sono fissati una volta per tutti
- L'insieme dei task periodici, con tutti i loro parametri funzionali ed i vincoli temporali, sono conosciuti e costanti
- Una schedulazione opportuna può essere calcolata "off-line" dal progettista del sistema ed è seguita fedelmente dallo scheduler a "run-time"

Spesso i sistemi che adottano una schedulazione *clock-driven* utilizzano un componente hardware chiamato *clock* o *timer* in grado di generare interruzioni ad intervalli di tempo regolari

Schedulazione clock-driven

Marco Cesati



Schema della lezione

Algoritmi clock-driven

Cyclic executive

Job aperiodici soft r.-t.

Job aperiodici hard r.-t.

SERT'20

R3.4

Come sono definiti? Le decisioni di un algoritmo di schedulazione *clock-driven* sono prese dallo scheduler soltanto limitatamente a controllare quali job effettivamente siano disponibili per essere eseguiti. In un algoritmo *clock-driven* è il progettista del sistema che decide quale, tra tutti i job che possono essere eseguiti, debbono effettivamente essere messi in esecuzione. E' il progettista del sistema, quando progetta il sistema, che decide la schedulazione. Lo scheduler, in realtà, non decide chi deve essere messo in esecuzione. Si limita a controllare che i job che effettivamente il progettista ha deciso di mettere in esecuzione, quei job siano presenti. Altrimenti lo scheduler predispone l'esecuzione di altri job, ma sempre tra quelli che il progettista ha detto che devono essere eseguiti.

Lo scheduler interviene, prende queste decisioni sui job che però ha deciso il progettista, ad intervalli di tempo ben definiti. Il tempo viene partizionato, viene suddiviso in tanti istanti, ed in quei istanti lo scheduler interverrà. Lo scheduler interviene in istanti prefissati. Non è uno scheduler che interviene quando succede qualcosa nel sistema. Lo scheduler non ha la libertà di prendere un altro job eseguibile, fuori dall'insieme definito, ed eseguirlo. Per poter realizzare un sistema *clock-driven*, il progettista deve conoscere l'insieme dei task che rappresentano il carico del sistema. Deve conoscere i parametri funzionali come l'istante di arrivo, il tempo di esecuzione, le mutue dipendenze. Tutto deve essere conosciuto dal progettista. E' lui che decide l'ordine con cui i job si avvicenderanno sul processore. E' veramente difficile costruire un sistema in cui qualche cosa non è conosciuto nel momento in cui lo progetto. Si tratta di una schedulazione *off-line*, costruita a priori, quando già tutto è noto sui task che caratterizzano il

sistema.

Si tratta di un paragone denigratorio, però possiamo pensare al progettista del sistema come al compositore di un'opera orchestrale. Ed allo scheduler come a quel componente che prende la battitura, l'opera, e la esegue, la fa eseguire all'orchestra. Si tratta di un esempio denigratorio e totalmente falso dal punto di vista della logica, ma come esempio può aiutare a capire che il direttore d'orchestra non ha la libertà di prendere scelte se il compositore non lo aveva previsto. Non può prendere iniziative.

Spesso, lo scheduler deve intervenire ad intervalli di tempo prefissati, che sono stabiliti in fase di progetto. Spesso per realizzare fisicamente questo sistema ci si appoggia a dei componenti hardware che sono dei chip che possono generare dei segnali hardware ad un certo istante di tempo, come i clock ed i timer. Il nome di questa classe di algoritmi deriva proprio da questo.

Perché sono utilizzati?

Quali sono i vantaggi più evidenti degli scheduler clock-driven?

L'algoritmo implementato dallo scheduler è molto semplice, quindi:

- lo scheduler è efficiente (ha un piccolo overhead)
- è facile validare il sistema nel caso di hard real-time

Qual è lo svantaggio più evidente degli scheduler clock-driven?

Lo scheduler è poco flessibile; ad esempio, è difficile gestire insieme di task non periodici, oppure la creazione di nuovi task a run-time

In passato, la maggior parte dei sistemi embedded hard real-time erano basati su uno scheduler di tipo **clock-driven**

La tendenza generale oggi è quella di adottare quando possibile scheduler **priority-driven**

Schedulazione clock-driven
Marco Cesati



Schema della lezione
Algoritmi clock-driven
Cyclic executive
Job aperiodici soft r.-t.
Job aperiodici hard r.-t.

SERT'20 R3.5

Quali sono i vantaggi più evidenti degli scheduler *clock-driven*? Non è difficile capire che il loro vantaggio principale è l'essere molto semplici. Fondamentalmente lo scheduler è così semplice che è piccolo, occupa poca memoria ed è efficiente. Non deve prendere decisioni o fare calcoli complicati, deve semplicemente *fare l'appello*. E' molto semplice e quindi poca memoria.

L'altro enorme vantaggio è che siccome la schedulazione è fatta a priori quando progetto il sistema, non c'è più questione sul rispettare o meno le scadenze. Se eseguo il job in quel preciso momento allora avrò il rispetto della scadenza.

Se non è arrivato non può rispettare la scadenza. Da parte del progettista, convincere l'ente certificatore che il sistema rispetta le scadenze è facile. Non può succedere qualcosa di differente dal punto di vista del software che non faccia rispettare le scadenze. E' facile validare un sistema hard-RT.

D'altra parte però questi scheduler hanno degli svantaggi. Quello più evidente è la mancanza di flessibilità. Se io devo progettare un sistema, ma devo conoscere tutto il sistema, non posso aspettarmi che il tutto possa gestire dei task non previsti in fase di progetto. La differenza è tra costruire un sistema RT, specializzato, tale per cui devo definire in modo completo qualunque task che quel sistema dovrà portare avanti. Altrimenti non posso fare la schedulazione off-line. Oppure un sistema che sì, è progettato per un certo ambito, ha certe caratteristiche, ma ho una certa libertà nel definire i parametri fondamentali del carico che andrò ad eseguire. Il progettista ha progettato il sistema in modo tale che certi parametri possano essere decisi, entro certi limiti, quando il sistema verrà utilizzato. Questa seconda possibilità offre molta più flessibilità.

Nella maggior parte dei casi, in passato si progettavano sistemi hard-RT basati su scheduler clock-driven. Per tutti questi motivi. Ma ultimamente le cose sono cambiate. Ultimamente la tendenza è quella di progettare sistemi RT che siano **priority-driven**. Cioè in cui la schedulazione non è scelta dal progettista ma dall'algoritmo mentre il sistema è in funzione. Perché? Sono più flessibili. L'altro fattore che ha contribuito alla diffusione dei sistemi *priority-driven* è che oggi i sistemi embedded hanno potenze calcolo superiori rispetto al passato. Quindi il vantaggio dell'avere uno scheduler semplice si sta attenuando. Ultimamente i sistemi embedded acquisiscono sempre più potenza di calcolo. Oggi la tendenza è di usare sistemi *priority-driven*. Non è un caso che noi ai sistemi *clock-driven* dedichiamo una singola lezione.

Perché sono utilizzati? (2)

- Gli scheduler di tipo **clock-driven** sono caratterizzati dal prendere le decisioni ad intervalli di tempo prefissati e costanti
- Sono adatti per sistemi con alto grado di determinismo in cui i parametri di (quasi) tutti i job sono conosciuti a priori
- È possibile calcolare la migliore schedulazione possibile una volta per tutte (*off-line*): **schedulazione statica**
- Se applicata a task periodici, viene anche chiamata schedulazione **ciclica**

Per contrasto, gli algoritmi priority-driven:

- determinano la schedulazione ad ogni occorrenza di eventi dinamici come il completamento di un job o la creazione di un nuovo task
- sono quindi algoritmi **on-line** che effettuano una schedulazione **dinamica**

Ricapitolando, gli scheduler di tipo *clock-driven* prendono le decisioni ad intervalli di tempo regolari prefissati, costanti. E le decisioni sono: “E’ arrivato o no un job?”, ma non quale job devo eseguire. Quindi sono adatti per sistemi con un grado di parallelismo abbastanza alto in cui i parametri di quasi tutti i job sono conosciuti a priori già dalla fase di progetto. E quindi, in fase di progetto, posso calcolare quale è la miglior schedulazione possibile una volta per tutte (**schedulazione statica**). Quando applico questi sistemi *clock-driven* al modello a task periodici, questo tipo di schedulazione viene chiamata **schedulazione ciclica**. Invece, gli algoritmi *priority-driven*, ad ogni invocazione dello scheduler ricalcolano quale è il miglior task da eseguire. Ed intervengono sugli eventi, non conosciuti dal progettista.

**Schedulazione
clock-driven**

Marco Cesati



Schema della lezione

Algoritmi clock-driven

Cyclic executive

Job aperiodici soft r.-t.

Job aperiodici hard r.-t.

SERT'20 R3.6

Modello a Task Periodici Ristretto

Per ragionare sugli algoritmi di [schedulazione ciclica](#) è utile fare riferimento ad una restrizione del [modello a task periodici](#):

- Il numero n di task nel sistema è fissato
- I parametri di tutti i task periodici sono conosciuti a priori
- Ogni job può essere eseguito dal suo istante di rilascio (niente vincoli di precedenza o conflitti sulle risorse)
- Possono esistere job aperiodici con vincoli temporali soft e hard real-time

Notazioni per indicare i parametri di un task periodico T_i :

- (ϕ_i, p_i, e_i, D_i) : fase ϕ_i , periodo p_i , tempo d'esecuzione e_i , scadenza relativa D_i
- (p_i, e_i, D_i) : fase uguale a 0
- (p_i, e_i) : fase 0, scadenza relativa uguale al periodo p_i

Schedulazione
clock-driven

Marco Cesati



Schema della lezione

Algoritmi clock-driven

Cyclic executive

Job aperiodici soft r.-t.

Job aperiodici hard r.-t.

SERT'20

R3.7

Come possiamo ragionare sui sistemi di schedulazione ciclica? E' utile fare riferimento ad una restrizione del modello a task periodici. Nel modello a task periodici abbiamo detto che ci sono tanti task. In realtà il numero di task può cambiare. Nel modello a task periodici ristretto abbiamo un numero n di task periodici fissati. Inoltre, i parametri di tutti i task periodici sono conosciuti a priori. Posso costruire un algoritmo di schedulazione a priorità, anche non conoscendo ad esempio il tempo di esecuzione di un job. Per validarlo lo devo in qualche modo conoscere, ma posso progettare un algoritmo che funziona a prescindere dai tempi di esecuzione dei job. Non devo necessariamente conoscerli in anticipo.

Inoltre, facciamo un'assunzione. Non ci sono vincoli di precedenza, non ci sono conflitti sulle risorse. Ogni job può essere eseguito dal suo istante di rilascio. Questo per studiare questi algoritmi. In effetti non è difficile tenere in considerazione questi vincoli sulle risorse quando progetto il sistema. Però, per fissarci le idee, pensiamo che non ce ne siano.

L'esistenza di job aperiodici, cioè job che possono arrivare in qualsiasi momento, può essere presa in considerazione. Vedremo come è possibile cercare di realizzare scheduler *clock-driven* in cui altri job, oltre a quelli periodici, possono arrivare quando vogliono. Saranno abbastanza semplici da gestire per i job soft-RT, in cui il rispetto delle scadenze è secondario. Saranno molto più complicati da gestire quando dovrò gestire hard-RT, in cui dovrò rispettare le scadenze.

Vediamo un po di introdurre qualche notazione per ragionare su questi sistemi.

Quando parliamo di task periodico lo definiamo con parametri che sono:

- Fase
- Periodo p_i
- Tempo di esecuzione e_i
 - Il tempo massimo di esecuzione di tutti i job del task
- Scadenza relativa
 - Ciò che determina, per ogni rilascio di un job, la scadenza assoluta. La scadenza assoluta è istante di rilascio più scadenza relativa

Molte volte indico una terna di numeri. Se abbiamo la terna di numeri, indica che la fase è uguale a 0. Una coppia di numeri indica fase uguale a 0 e scadenza relativa uguale al periodo, cioè la scadenza esplicita.

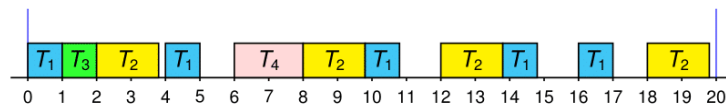
Esempio di schedulazione ciclica

Consideriamo un sistema con un processore e quattro task
 $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$ e $T_4 = (20, 2)$

Come derivare una schedulazione fattibile?

- Lunghezza dell'iperperiodo: $\text{mcm}(4, 5, 20, 20) = 20$
- Troviamo una schedulazione fattibile in un iperperiodo
- Ripetiamo la schedulazione all'infinito

Una possibile soluzione:



Schedulazione
clock-driven

Marco Cesati



Schema della lezione

Algoritmi clock-driven

Cyclic executive

Job aperiodici soft r.-t.

Job aperiodici hard r.-t.

SERT'20 R3.8

Consideriamo un sistema con 4 task ed un solo processore. Questi task hanno:

- $T_1 = (4, 1)$
- $T_2 = (5, 1.8)$
- $T_3 = (20, 1)$
- $T_4 = (20, 2)$

Io sono il progettista e devo progettare l'algoritmo *clock-driven* di questo sistema di 4 task. Come si può fare? La prima cosa da fare è calcolare l'iperperiodo:

- **Iperperiodo:** $\text{mcm}(4, 5, 20, 20) = 20$

Cerco all'interno dell'iperperiodo una schedulazione fattibile, stando attendo

però. In realtà devo anche tenere in considerazione le fasi dei task. Tutti questi task stanno in fase, tutti cominciano all'istante 0 i primi rilasci, quindi questo non è un problema.

Quando ho trovato una schedulazione fattibile, che rispetta le scadenze implicite, poi è facile. Prendo questa schedulazione che funziona in questo iperperiodo e la ripeto all'infinito. L'iperperiodo è il blocco che ripeto all'infinito.

Quindi, ad esempio, nell'immagine è presente una possibile soluzione. Quello che esce fuori è che in realtà, se vado a controllare, tutti i task rispettano le scadenze. Come facciamo ad implementare via software questa schedulazione?

Scheduler a tabella

Implementazione di uno scheduler clock-driven tramite tabella di voci $(t_k, T(t_k))$:

- t_k indica l'istante in cui una "decisione" è presa
- $T(t_k)$ indica il nome del job o task da eseguire oppure I se il processore è inutilizzato (*idle*)

La schedulazione precedente è rappresentata dalla tabella:

t_k	0	1	2	3.8	4	5	6	8	9.8
$T(t_k)$	T_1	T_3	T_2	I	T_1	I	T_4	T_2	T_1

10.8	12	13.8	14.8	16	17	18	19.8
I	T_2	T_1	I	T_1	I	T_2	I

- Job aperiodici soft real-time: schedulati negli intervalli " I ", ma interrotti se non completati entro il t_k successivo
- Job aperiodici hard real-time: non previsti (in questa versione)

Schedulazione clock-driven

Marco Cesati



Schema della lezione

Algoritmi clock-driven

Cyclic executive

Job aperiodici soft r.-t.

Job aperiodici hard r.-t.

SERT'20
R3.9

Potrei costruire una tabella, in cui ho due voci. Una voce indica t_k , l'istante in cui una decisione è presa, e $T(t_k)$, che rappresenta il nome del job o del task che deve essere eseguito in quell'istante. Oppure I , che indica che il processore non viene utilizzato. La figura di prima in realtà la possiamo descrivere con questa tabella. Questa è esattamente la descrizione vettoriale della figura di prima.

Dopo di che, quando posso gestire i job aperiodici soft-RT? Ovviamente in tutti gli intervalli con I il processore non sta facendo nulla, e posso sfruttarlo per eseguire job aperiodici soft-RT. In realtà però, quando arrivo al t_k successivo, i job vengono interrotti se non completati. I job aperiodici devono essere interrompibili, sia quelli soft che hard RT. In questa versione però, non possiamo tenere conto dei job aperiodici per i job hard-RT.

Pseudo-codice per scheduler clock-driven

Procedura SCHEDULER:

Input: Tabella di schedulazione $(t_0, T(t_0)), \dots, (t_{N-1}, T(t_{N-1}))$

$i = 0, k = 0$

imposta il timer a t_k (tempo assoluto)

ripeti:

accetta interruzioni di timer

interrompi un eventuale job aperiodico

job corrente $J = T(t_k)$

$i = i + 1, k = i \bmod N$

imposta il timer a $\lfloor i/N \rfloor H + t_k$ (tempo assoluto)

se $J = \mathcal{I}$

attiva il primo job nella coda aperiodica

altrimenti

attiva il job J

sospendi l'esecuzione dello scheduler

Fine SCHEDULER

Schedulazione
clock-driven

Marco Cesati



Schema della lezione

Algoritmi clock-driven

Cyclic executive

Job aperiodici soft r.-t.

Job aperiodici hard r.-t.

SERT'20 R3.10

A causa di questa tabella, questi sono anche chiamati scheduler a tabella. Fondamentalmente questo è lo pseudocodice. Ho come input la tabella di schedulazione, che sarebbe la *partitura orchestrale*, e successivamente abbiamo l'esecuzione dello scheduler. Abbiamo degli indici i (contatore che viene incrementato per ogni intervallo), mentre k è l'intervallo all'interno dell'iperperiodo. Abbiamo detto che lo scheduler deve intervenire ad intervalli di tempo prefissati. Possiamo farlo andando ad impostare il timer ad un determinato istante di tempo.

Schedulazioni cicliche strutturate

In generale è preferibile lavorare con schedulazioni che soddisfano certe proprietà strutturali, ad esempio:

- Attivazione dello scheduler ad intervalli regolari
- Distribuzione regolare degli intervalli “ I ” (processore **idle**) nell'iperperiodo

Che vantaggi portano queste proprietà?

- Lo scheduler può essere attivato da un dispositivo hardware che genera interruzioni periodiche (ad es. il PIT, Programmable Interval Timer)
- I job aperiodici possono essere eseguiti in modo regolare in corrispondenza degli intervalli “ I ”
- Possibilità di monitorare e/o forzare il rispetto dei vincoli temporali in caso di job che allungano la loro esecuzione

Una procedura che implementa un algoritmo di schedulazione ciclica “strutturata” è chiamata *cyclic executive*

Schedulazione
clock-driven

Marco Cesati



Schema della lezione

Algoritmi clock-driven

Cyclic executive

Job aperiodici soft r.-t.

Job aperiodici hard r.-t.

SERT'20

R3.11

In realtà quando si progettano questo tipo di scheduler conviene lavorare con scheduler progettati in maniera un po' più sofisticata. Vorrei che le schedulazioni vadano a soddisfare certe proprietà, come che lo scheduler fosse attivato ad intervalli di tempo regolari. L'attività di programmare questi timer è costosa in termini di tempo. Programmare un timer costringe ad arrivare a programmare l'hardware. Sarebbe meglio se riuscissi a farlo una volta per tutte. E' più semplice fare in modo che il clock sia regolare, e quindi che lo scheduler sia attivato ad intervalli regolari.

Inoltre vorrei che, se possibile, questi intervalli I siano distribuiti in modo regolare all'interno dell'iperperiodo. Questa cosa posso farla se ho una certa regolarità del modo in cui vengono generati i clock.

Queste proprietà che vantaggi portano? Posso usare un dispositivo che è disponibile in tutti i sistemi hardware, il PIT (*timer interval programmabile*). I job aperiodici posso eseguirli in maniera regolare. Non devo aspettare quasi tutto un iperperiodo per arrivare al primo intervallo in cui il processore non fa nulla. Posso cercare di sparpagliare gli intervalli I in tutto l'iperperiodo, e fare in modo che i job aperiodici vadano avanti con una certa regolarità.

E poi, soprattutto, se progetto bene lo scheduler posso fare in modo che faccia un controllo. Oltre a capire quali job arrivano e quali no, lo scheduler assume un altro ruolo. Un ruolo di controllo sul fatto che i job schedulati precedentemente abbiano davvero rispettato la loro scadenza. Questo sembra un po' un controsenso, in quanto se progettiamo bene il sistema allora il sistema deve rispettare le scadenze.

Come dicevamo però, noi progettiamo il software di un sistema hard-RT, ma non tutto è sotto il controllo il controllo del software. Eventi hardware che per esempio rallentano l'esecuzione del processore, possono portare a mancato rispetto delle scadenze. Anche se a regola d'arte il software questo non lo faceva succedere. E' buona norma, è bene costruire anche il software in modo da tener conto che qualcosa possa andar male.

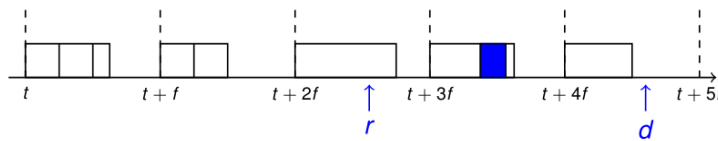
Quindi avere almeno la possibilità di rilevare il fatto che le scadenze siano mancate. Questo tipo di scheduler regolari, esecuzioni cicliche strutturate, permettono di fare questo lavoro. Una procedura che implementa gli algoritmi di schedulazioni cicliche strutturate è chiamato in inglese **cyclic executive**. L'esempio che abbiamo visto alla prima lezione sul flight controller dell'elicottero, quello che è un tipico esempio di **cyclic executive**. C'era un apparato che eseguiva diversi job in un ordine prefissato, ad intervallo di tempo regolari.

Frame

Gli istanti di tempo in cui uno scheduler ciclico strutturato prende decisioni partizionano la linea temporale in intervalli regolari chiamati *frame*

- La lunghezza f dei *frame* è prefissata
- In ogni frame è definita una lista di job da eseguire in sequenza (*blocco di schedulazione*)
- All'interno dei *frame* non si possono interrompere i job: se un job inizia in un frame, deve terminare entro lo stesso frame
- La fase di ogni task periodico è un multiplo intero non negativo della lunghezza del *frame*:

$$\forall i \in \{1, \dots, n\}, \exists k \in \mathbb{N} : \phi_i = k \times f$$



Gli istanti di tempo in cui lo **scheduler ciclico strutturato** prende decisioni, partizionano la linea temporale in intervalli regolari chiamati frame. Un problema critico è capire quanto è lungo il frame. La lunghezza del frame è fissata dal progettista. Quando devo far lungo questo frame? Consideriamo che dentro ad ogni frame devo definire la lista dei job che vanno eseguiti nel frame che sta partendo. Questa lista è chiamata **blocco di schedulazione**. Il compito dello scheduler all'inizio del frame è di dire: "Di tutti i job che il progettista ha definito dover essere eseguiti dentro questo frame che arriva, quali effettivamente sono stati rilasciati?". Questi che sono stati rilasciati vengono messi nel blocco di

Schedulazione
clock-driven

Marco Cesati



Schema della lezione

Algoritmi clock-driven

Cyclic executive

Job aperiodici soft r.-t.

Job aperiodici hard r.-t.

SERT'20

R3.12

schedulazione del prossimo frame di questo frame. Quindi verranno eseguiti uno dopo l'altro senza che lo scheduler intervenga più dentro al frame. Finito il frame, lo scheduler si risveglia. Di questo blocco di schedulazione precedente, quali frame hanno effettivamente rispettato le scadenze e quali le hanno mancate? E poi va a vedere il prossimo frame. Chi dovrebbe essere eseguito? Costruisco un nuovo blocco di schedulazione e così via.

All'interno dei frame i job non si possono interrompere. Se un job inizia in un frame, deve terminare entro lo stesso frame. A meno che io non abbia un modo per suddividere il lavoro di un job in due sotto job che posso eseguire in due frame. Questo si può fare, ma di per se non si può interrompere un unico job. Se il progettista ha un insieme di job prefissati, che non può più suddividere, allora questi job devono essere eseguiti ciascuno dentro ad un frame. Non si può scavalcare il frame.

Un altro vincolo è che la fase di ogni task periodico deve essere un multiplo intero non negativo della lunghezza del frame. La fase è l'istante di rilascio del primo job di un task. Questo istante di rilascio deve cadere esattamente su un frame. In altre parole, per ogni task t_i , la fase del task i deve essere uguale esattamente ad un $k \cdot f$, dove k è un qualsiasi numero naturale, mentre f è la lunghezza del frame. Perché abbiamo questo vincolo? Lo vediamo in figura.

Abbiamo un frame che inizia al tempo t , un altro al tempo $t + f$ e così via. Ad ogni frame è presente un insieme di job che in qualche modo sono arrivati e che devono essere eseguiti. Il progettista assume che tutti i job che possono arrivare siano arrivati. Per il progettista questi sono esattamente tutti job che devono essere eseguiti all'interno del frame. Se qualche job non arriva, vuol dire che questo frame non avrà il corrispondente job e quindi non avrà più tempo libero. Il punto è che dentro ad ogni frame, il progettista ha piazzato un po di job. Considerando il job in blu, eseguito tra $t + 3f$ e $t + 4f$. Il punto è che se arriva all'istante r non può essere eseguito appena arriva, ma bisogna eseguirlo nel frame successivo a quello di arrivo. Se proprio arriva sulla scadenza di un frame e l'inizio del successivo, anche in quello successivo, ma non certamente se arriva in mezzo ad un frame come in questo caso.

Il job ha una scadenza, d . Questa scadenza, se cade in mezzo ad un frame, implica che il job non può essere eseguito in questo frame. Mettiamo che il job sia eseguito nel frame e termini entro la scadenza. Il problema è che chi deve fare il controllo che lui abbia terminato, cioè lo scheduler, lo farebbe entro un tempo post scadenza stessa. Quindi lo scheduler dice che il job è completato, ma non sa dire se è completato entro la scadenza oppure no. L'unico modo è se lo scheduler interviene prima della scadenza del job. Dunque ogni job deve essere terminato nel frame che precede la sua scadenza altrimenti lo scheduler non può controllare.

Vincoli sulla dimensione dei frame

- (1) Poiché non è possibile interrompere un job all'interno di un frame, il frame deve essere abbastanza lungo da garantire la completa esecuzione di ciascun job:

$$f \geq \max\{e_1, \dots, e_n\}$$

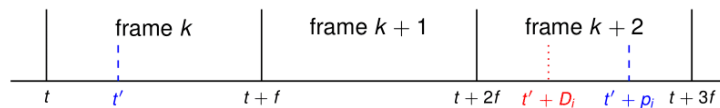
- (2) La dimensione del frame deve dividere la lunghezza dell'iperperiodo H :

$$H/f - \lfloor H/f \rfloor = 0$$

Condizione sufficiente è che f divida il periodo p_i di almeno uno dei task.

- (3) Il frame deve essere abbastanza piccolo così che tra l'istante di rilascio e la scadenza di ogni job ci sia sempre almeno un frame; condizione sufficiente:

$$2 \cdot f - \gcd(p_i, f) \leq D_i, \quad \forall i \in \{1, \dots, n\}$$



38.57

