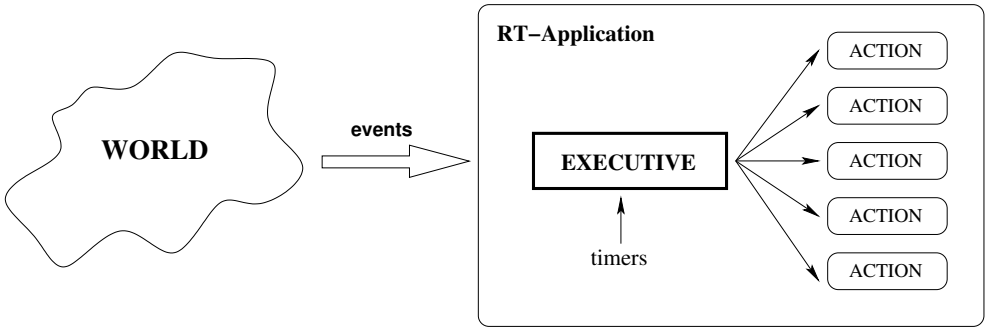
UNIVERSITÀ DEGLI STUDI DI PARMA
Dipartimento di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria Informatica

CORSO DI SISTEMI OPERATIVI E IN TEMPO REALE
Esercitazione *Real-Time* n.4

1 Design di una applicatione Real Time

Il funzionamento di una applicazione *real-time* è tipicamente influenzato da eventi esterni, che possono giungere ad esempio dal mondo fisico, ai quali dovranno corrispondere azioni di controllo intraprese dal software entro un tempo massimo.



Per garantire il rispetto dei tempi di esecuzione richiesti (*deadline*) ed una risposta rapida agli eventi esterni (*latenza*) è opportuno suddividere il compito del programma RT in un certo numero di task che devono essere posti in esecuzione allo scadere di intervalli di tempo regolari (*task periodici*) oppure all'arrivo di un evento asincrono dall'esterno (*task aperiodici*); i singoli task previsti dallo schedule possono quindi essere mappati su differenti thread di sistema, sfruttando le funzionalità proprie dello scheduler del sistema operativo.

Dato un insieme di task da eseguire si può progettare un sistema centralizzato per gestirne l'attivazione, delegando ad un *executive* il compito di decidere quale task attivare, istante per istante: questo design semplifica notevolmente la gestione delle temporizzazioni e le funzioni di supervisione, necessarie per prevedere azioni di recupero in caso di errori imprevisti.

Considerazioni:

- Nella realizzazione di un'applicazione RT i task e l'executive dovranno essere implementati facendo uso degli strumenti messi a disposizione dal sistema operativo: processi, thread, mutex, condition variables, eccetera;
- Creare un thread ogni volta che un task viene rilasciato non è efficiente, dato che la creazione è molto spesso un'operazione *lenta*: è preferibile creare a priori un insieme di thread che si mettono in attesa di un *segnale di attivazione*;
- Il supervisore (executive) dovrà possedere una propria trama di esecuzione, che verrà attivata con priorità sugli altri thread che si trovano eventualmente in esecuzione. Dopo aver attivato il nuovo task l'executive può rimanere inattivo fino al prossimo istante di attivazione.

2 Scheduling Clock Driven

Lo scopo di questa esercitazione è la costruzione di un sistema di scheduling real time che consenta di porre in esecuzione un insieme di task *periodici* secondo un approccio *clock driven*, utilizzando un algoritmo di scheduling statico che fornisce al sistema uno schedule *fattibile*, se esiste, prima che il sistema vada in esecuzione. Il sistema inoltre deve essere in grado di eseguire task *aperiodici* secondo le modalità precisate nel paragrafo 4.

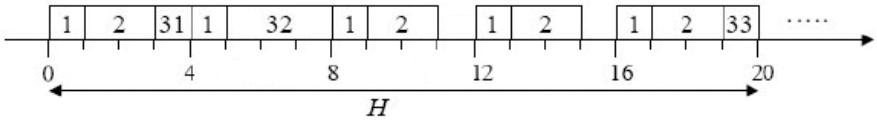
Se per ogni task τ_i sono noti (C_i, D_i, T_i), mediante l'algoritmo di scheduling è possibile calcolare a priori uno schedule statico caratterizzato da:

- Il valore dell'iperperiodo H;
- Il periodo di frame m;
- La sequenza di task da eseguire in ogni frame.

Lo schedule ciclico così definito può essere utilizzato per porre in esecuzione nell'ordine giusto i task all'interno dei frame, mediante un executive che ne governa l'attivazione.

Esempio (dai lucidi del corso): Siano dati i task τ_1, τ_2, τ_3, di cui τ_3 è stato partizionato in 3 sotto-task: $\tau_1 = (4, 1, 4)$, $\tau_2 = (5, 2, 7)$, $\tau_{3,1} = (20, 1, 20)$, $\tau_{3,2} = (20, 3, 20)$, $\tau_{3,3} = (20, 1, 20)$.

Lo schedule ciclico che viene proposto si può rappresentare graficamente come segue:



Le informazioni necessarie all'executive per gestire l'attivazione dei task sono le seguenti:
$H = 20$; $m = 4$;
$F_1 = \{\tau_1, \tau_2, \tau_{3,1}\}$, $F_2 = \{\tau_1, \tau_{3,2}\}$, $F_3 = \{\tau_1, \tau_2\}$, $F_4 = \{\tau_1, \tau_2\}$, $F_5 = \{\tau_1, \tau_2, \tau_{3,3}\}$.

2.1 L'executive

Realizzare un executive che sia in grado di governare l'esecuzione dei task *attivandosi* soltanto all'inizio di ogni frame per compiere le seguenti azioni:

- Porre in esecuzione i task *periodici* del prossimo frame, nel giusto ordine;
- Controllare che i task *periodici* del frame precedente abbiano terminato;
- Gestire l'esecuzione di task *aperiodici* evitando interferenze con la schedule ciclica;
- Rilevare e segnalare eventuali deadline miss.

3 Il software

Il software che vogliamo sviluppare dovrà fornire un *motore* il cui ruolo sia quello di semplificare la realizzazione dell'applicazione real-time da parte di un ipotetico utente finale che deve poter scrivere il codice relativo ai singoli task, specificare lo schedule e farlo eseguire all'executive senza conoscere i dettagli implementativi che sono stati utilizzati per realizzare la temporizzazione, il controllo delle deadline, ecc.

Il progetto deve funzionare su sistema operativo GNU/Linux, perciò sarà necessario utilizzare le API per gestire la programmazione multithread fornite nativamente dal linguaggio C++11 e le estensioni POSIX per lo scheduling real-time.

3.1 Alcune considerazioni:

- **Task:** Ogni task ha una propria trama di esecuzione, pertanto si richiede la creazione di un thread per ogni task, la cui esecuzione deve essere *controllata* dall'executive, che ne governa l'attivazione all'inizio di ogni frame e controlla che abbia terminato in tempo, allo scadere della deadline. Per fare questo si dovranno utilizzare meccanismi di sincronizzazione basati su *variabili condizione* e *mutex*.

- **Executive:** L'executive deve essere in grado di attivarsi all'inizio di ogni frame, con la minor latenza possibile, pertanto deve essere garantito che il protrarsi dell'esecuzione dei task non ritardi l'attivazione successiva dell'executive. Per fare questo l'executive dovrà essere eseguito su di un apposito thread e le priorità dei thread coinvolti nel sistema dovranno essere adeguatamente calibrate mediante scheduling real-time *FIFO*. L'executive dovrà inoltre controllare con precisione l'esecuzione periodica dello schedule, utilizzando API di temporizzazione di tipo *assoluto*.

- **Deadline miss:** La procedura di recovery da eseguire in caso di deadline miss sarà l'emissione di un segnale di errore sullo standard error (`std::cerr`) e l'abbassamento della priorità del task al minimo valore realtime disponibile, per non interferire con l'esecuzione dei task successivi. In questa situazione, per ridurre momentaneamente il carico di esecuzione, si considera lecito cancellare una o più esecuzioni successive del medesimo task che ha generato la deadline miss.

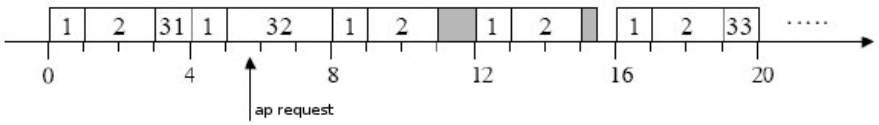
- **Test & Debug:** Per verificare la reale durata dei C_i dei singoli task e per verificare i reali istanti di attivazione dei thread, si dovranno utilizzare i *timer* resi disponibili dal linguaggio C++ sull'HW utilizzato.

4 Scheduling di task aperiodici

Il progetto, oltre all'implementazione di una politica di scheduling *Clock-Driven* per l'esecuzione dei task *periodici*, deve consentire la schedulazione di un task *aperiodico*, il cui rilascio avvenga in modo sporadico durante il funzionamento del sistema.

- L'esecuzione del task aperiodico deve fare uso dello *slack time* presente nei frame immediatamente successivi a quello di rilascio, senza interferire con il rispetto delle deadline dei task periodici.
- La richiesta di rilascio del task aperiodico può avvenire solo da parte di uno o più dei task periodici, mediante l'invocazione di una apposita funzione `ap_task_request()`.
- Il funzionamento del task aperiodico si considera corretto quando la sua esecuzione termina entro la successiva richiesta di rilascio, pertanto deve essere effettuato un controllo di *deadline miss* che emetta un segnale di errore sullo standard error (`std::cerr`) quando l'executive, attivando una nuova istanza del task aperiodico, verifica che la sua precedente esecuzione è ancora in atto. In questa situazione si potranno saltare una o più esecuzioni successive del task aperiodico.
- Per semplicità si assume che non sia lecito richiedere più di una attivazione del task aperiodico all'interno dello stesso frame.

Esempio



In questo caso l'esecuzione del task aperiodico, indicata in grigio, viene richiesta da $\tau_{3,2}$ durante il frame n.2, può iniziare solo durante lo slack time presente nel frame n.3 e si conclude durante il frame n.4.