ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DISI

CORSO DI LAUREA IN INGEGNERIA INFORMATICA MAGISTRALE

TESI DI LAUREA

in

SISTEMI IN TEMPO REALE M

Studio e sviluppo di un'infrastruttura di supporto alla schedulazione di hard/soft real-time task in ambiente QNX

CANDIDATO:
Andrea Calafiore

RELATORE:
Chiar.mo Prof. Eugenio Faldella

Anno Accademico 2014/15

Sessione III

“Houston, we’ve had a problem here.”

John L. Swigert al JSC (Johnson Space Center)
durante la missione Apollo 13

Sommario

Capitolo 1 - Introduzione e obiettivi

I sistemi real-time sono quei sistemi di calcolo in cui la correttezza di funzionamento non dipende soltanto dalla validità dei risultati ottenuti ma anche dal tempo in cui i risultati sono prodotti [1]. Al giorno d'oggi, con l'aumento costante dell'uso di processori in ogni campo (ma particolarmente in ambito industriale), risultano un campo di studio particolarmente interessante.

I sistemi real-time sono necessari in numerose applicazioni quali possono essere:

- sistemi di controllo delle automobili;
- sistemi di regolazione di impianti industriali, in particolare in caso di impianti chimici o nucleari;
- la robotica;
- sistemi di telecomunicazione;
- sistemi in campo medico;
- sistemi militari;
- missioni spaziali;
- sistemi di controllo di volo sugli aeroplani;
- in ambito industriale, il controllo di processi produttivi.

Figura 1.1 Esempio di servizi gestiti dal sistema operativo real-time QNX Neutrino [1]

Solitamente in questi casi i sistemi sono formati da hardware, sistema operativo e software applicativo intrinsecamente legati per riuscire a ottimizzare le prestazioni.

L’utilizzo di un sistema in tempo reale permette anche di ridurre la quantità di processori necessari ad un impianto o sistema, riducendo di conseguenza anche i costi ed il consumo di energia. Soprattutto quest’ultima considerazione è tenuta in gran conto in quei sistemi come, per esempio, le missioni spaziali, in cui i problemi di consumo delle batterie o del carburante sono cruciali.

Al contrario di quello che potrebbe sembrare dalla definizione, i sistemi real-time non devono necessariamente essere veloci bensì devono essere prevedibili (deterministici). È importante che il sistema reagisca entro un certo tempo limite, non che il sistema completi le operazioni nel più breve tempo possibile.
Un sistema real-time deve quindi garantire che un *task* (elaborazione o processo) termini entro un dato vincolo temporale (detto in gergo *deadline*) tramite un’opportuna schedulazione.

I vari algoritmi di schedulazione si classificano differentemente in base ad alcuni aspetti: preemptive o non preemptive (a seconda che l’esecuzione di un processo possa essere sospesa o meno), statici o dinamici (se le decisioni si basano su parametri fissi assegnati ai processi prima della loro attivazione o se queste vengono prese sulla base di parametri che variano durante l’esecuzione dei processi), off-line oppure on-line (nel primo caso tutte le decisioni vengono prese prima dell’attivazione dei processi, in caso contrario la schedulazione viene stabilita durante l’esecuzione) e best-effort o guaranteed (a seconda che si ottimizzino le prestazioni medie dell’insieme di processi o che si punti a garantire il rispetto dei vincoli temporali di ogni processo).

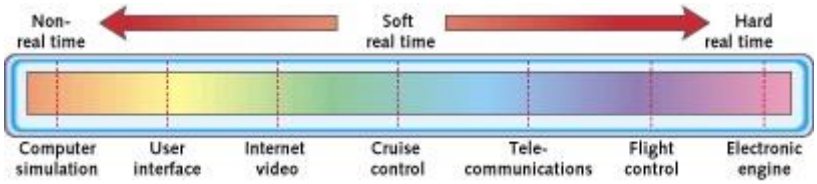

Figura 1.2 Rigidità dei vincoli temporali in relazione alla criticità dell'applicazione [2]

I processi di un sistema real-time ricadono in due categorie principali:

- hard real-time: è il caso in cui i vincoli temporali (cioè il completamento del processo entro la deadline) devono essere rispettati;

- soft real-time: è il caso in cui i vincoli temporali possono essere disattesi in condizioni di temporaneo sovraccarico.

In entrambe le categorie i processi possono avere una frequenza di esecuzione costante (nel qual caso sono definiti periodici) oppure svolgere attività in maniera impredicibile (nel caso dei processi hard real-time sono definiti sporadici, nel caso dei processi soft-real time invece vengono definiti aperiodici).

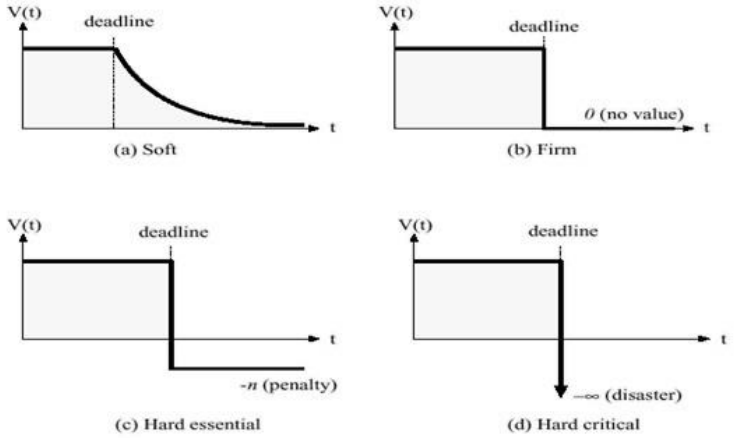

Figura 1.3 Funzioni di utilità di un processo [3]

Data l'importanza del fattore tempo in questo ambito è necessario fare chiarezza sui molti parametri temporali di un processo (o task):

- $\mathbf{a_i}$ ($\mathbf{r_i}$), **arrival (release) time**: tempo in cui il task diventa pronto per l'esecuzione;
- $\mathbf{d_i}$, **deadline**: tempo entro cui il task deve essere completato;
- $\mathbf{s_i}$, **start time**: tempo in cui al task viene assegnata la CPU per la prima volta;
- $\mathbf{f_i}$, **finishing time**: tempo di completamento dell'esecuzione del task;

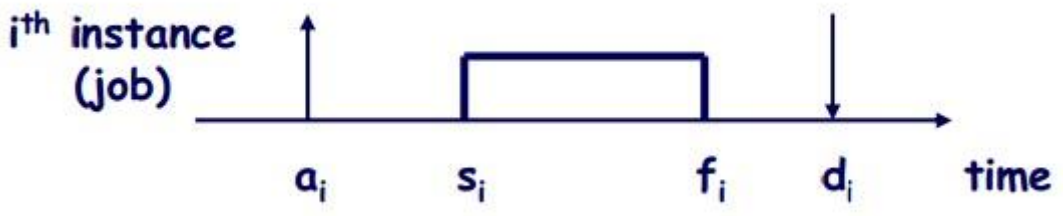

Figura 1.4 Parametri temporali di un processo [4]

A queste caratteristiche se ne aggiungono altre derivate:

- $C_i = f_i - s_i$, **computation time**: tempo necessario al processore per eseguire completamente il task senza interruzioni;
- $D_i = d_i - a_i$, **relative deadline**: massimo tempo entro il quale il task deve essere completato rispetto al suo rilascio;
- $R_i = f_i - a_i$, **response time**: tempo per cui il task ha richiesto l'uso della CPU;
- $L_i = f_i - d_i$, **lateness**: ritardo del completamento del task rispetto alla deadline;
- $E_i = \max(0, L_i)$, **tardiness (exceeding time)**: tempo in cui il task rimane attivo dopo la deadline;
- $X_i = D_i - C_i$, **laxity (slack time)**: massimo ritardo di attivazione che un task può subire per non eccedere la sua deadline.

Ogni istanza i-esima di uno stesso task viene definita job. Per un processo periodico il periodo (**T**) viene definito come la differenza tra il tempo di rilascio del processo i+1-esimo e il processo i-esimo ($a_{i+1} - a_i$).

1.1 Scopo

Lo scopo di questa tesi è verificare le funzionalità offerte dal sistema operativo commerciale real-time QNX, tramite l'uso di una licenza accademica. Per fare ciò sarà necessario uno studio dell'architettura del

sistema operativo e un’analisi approfondita dei servizi offerti (sia real-time che non).
La fase successiva consisterà nel creare una infrastruttura software in grado di simulare diversi processi real-time, creati sulla base di parametri temporali specifici, e verificare il funzionamento e le prestazioni del sistema.
Questo studio si concentrerà esclusivamente sulle funzioni e i servizi offerti direttamente dal sistema operativo, senza tenere in considerazione eventuali altri software applicativi, adibiti ad una integrazione delle funzioni real-time, sviluppati per particolari sistemi embedded.

1.2 Struttura

La tesi sarà suddivisa in sei capitoli.
Nel capitolo 2 verrà descritto brevemente il sistema operativo QNX e gli strumenti messi da esso a disposizione per lo sviluppo del simulatore, nel capitolo 3 invece saranno presentati gli strumenti utilizzati per la programmazione e il test del simulatore sviluppato. Il capitolo 4 descriverà nel dettaglio le politiche di schedulazione dei task periodici, le politiche di gestione delle richieste aperiodiche, la loro implementazione nel simulatore e il sistema di log interno al simulatore. Nel capitolo 5 verranno esposti le analisi di schedulabilità per i diversi algoritmi e i risultati ottenuti dal sistema. Il capitolo 6 sarà dedicato all’esposizione delle conclusioni e ad eventuali possibili sviluppi futuri del simulatore.

Capitolo 2 - QNX

QNX [2] è un sistema operativo real-time basato su Unix che implementa lo standard ***POSIX*** (***P****ortable* ***O****perating* ***S****ystem* ***I****nterface*). Basato su un'architettura a microkernel, è utilizzato prevalentemente in sistemi embedded in diversi settori. Viene infatti utilizzato in applicazioni automobilistiche, industriali, mediche ma anche nell'ambito delle telecomunicazioni e nel settore militare [3].

2.1 POSIX

POSIX è una famiglia di standard definita dall'***IEEE*** (***I****nstitute of* ***E****lectrical and* ***E****lectronic* ***E****ngineers*) per mantenere la compatibilità tra le diverse varianti dei sistemi operativi UNIX. POSIX definisce l'interfaccia utente standard, basata sulla shell Korn, come anche la standardizzazione di diversi software applicativi, servizi e programmi di utilità, tra cui *echo* e *ed*. Ha inoltre definito un gran numero di API, i cui servizi includono l'input/output (file, terminale, rete), funzioni per la gestione dei processi e dei thread e anche estensioni per i servizi real-time, tra cui semafori, schedulazione basata su priorità, timer ad alta risoluzione, ecc.

2.2 Architettura di QNX

QNX è un sistema operativo a microkernel. Si definisce sistema operativo a microkernel un sistema operativo in cui un piccolo kernel provvede ai servizi minimi necessari ad una serie di processi che, cooperando, provvedono a fornire i servizi di alto livello di un tipico sistema operativo (per esempio, nel caso di QNX, il microkernel non fornisce il servizio di file system che è delegato ad un altro processo). In questo modo è possibile per ogni utente abilitare o disabilitare tutti i servizi di cui necessita, personalizzando così il sistema operativo e migliorandone le prestazioni. Inoltre con questa architettura il sistema operativo risulta

facilmente estendibile tramite la creazione di semplici processi che integrino i servizi del kernel. Un'architettura a microkernel offre anche una protezione completa dell'accesso alla memoria, in quanto questo servizio (fornito da un'applicazione diversa) deve operare attraverso il microkernel.

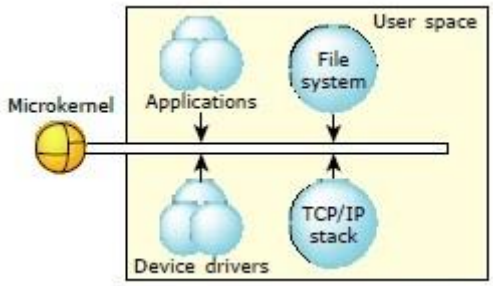

Figura 2.1 Protezione della memoria a opera del microkernel [5]

Il kernel, per essere più piccolo possibile, offre solo i servizi fondamentali:

- servizi per i thread, tramite le primitive di creazione thread specificate da POSIX;
- servizi per i segnali, tramite le primitive per i segnali POSIX;
- servizi di messaggistica: il microkernel gestisce lo smistamento di tutti i messaggi tra i thread dell'intero sistema;
- servizi di sincronizzazione, attraverso le primitive POSIX per la sincronizzazione dei thread;
- la schedulazione: il microkernel schedula i diversi thread sulla base delle politiche di schedulazione real-time POSIX;
- timer: il microkernel fornisce un ricco insieme dei servizi di timer definiti in POSIX;
- i servizi di gestione dei processi: il microkernel delega parte di questi servizi ad un processo per la gestione dei processi.

Al contrario dei diversi thread, il kernel non va mai in esecuzione. Il processore esegue codice del microkernel solo come risultato di una chiamata ad una esplicita kernel call, un'eccezione o in risposta ad un interrupt hardware.

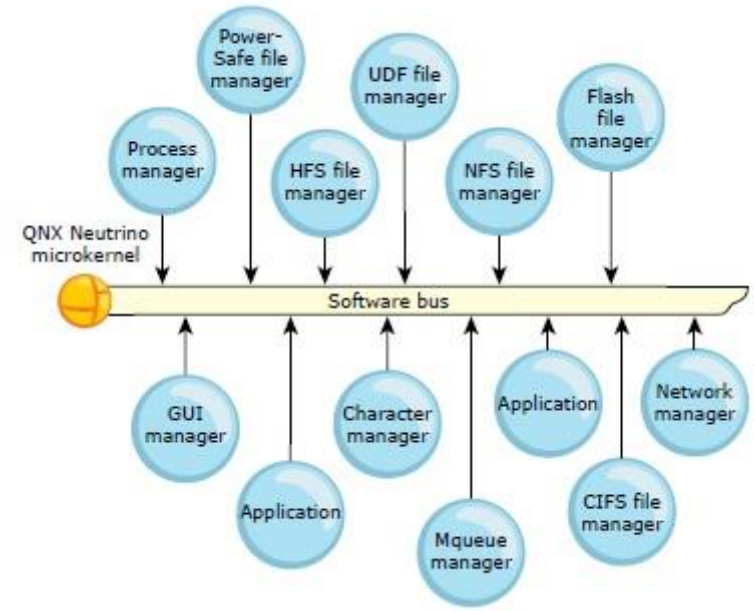

Figura 2.2 Architettura del sistema operativo QNX [5]

Tutti gli altri servizi del sistema operativo, sono gestiti attraverso processi standard.

Dopo questa introduzione sull'architettura di QNX, nelle due prossime sezioni verranno descritti più approfonditamente i servizi utilizzati nello sviluppo del simulatore in modo che la descrizione di quest'ultimo risulti chiara.

2.3 Il microkernel

Il microkernel implementa le funzioni POSIX e il sistema di comunicazione tra processi. Nello specifico fornisce le kernel call per supportare i seguenti servizi (in grassetto quelli utilizzati nel simulatore):

- I **thread**;
- il **sistema di messaggistica**;
- i segnali;
- la **gestione del clock**;
- i **timer**;

- la gestione delle interruzioni;
- i **semafori**;
- le variabili di mutua esclusione (mutex);
- le variabili condizione (condvar);
- le **barriere**.

2.3.1 Thread

Un'applicazione (e particolarmente il simulatore sviluppato per questa tesi) può aver bisogno di diversi algoritmi che eseguano concorrentemente al proprio interno. Utilizzando il modello a thread POSIX, un'applicazione (processo) contiene uno o più thread. Ogni thread esegue all'interno di un processo principale e dispone di attributi propri quali: identificatore, nome, priorità, signal mask, ecc.

Finché un thread è in esecuzione il suo stato può generalmente essere "pronto" o "bloccato". In QNX non esiste un unico stato bloccato, ma ad ogni possibile causa è assegnato uno stato specifico.

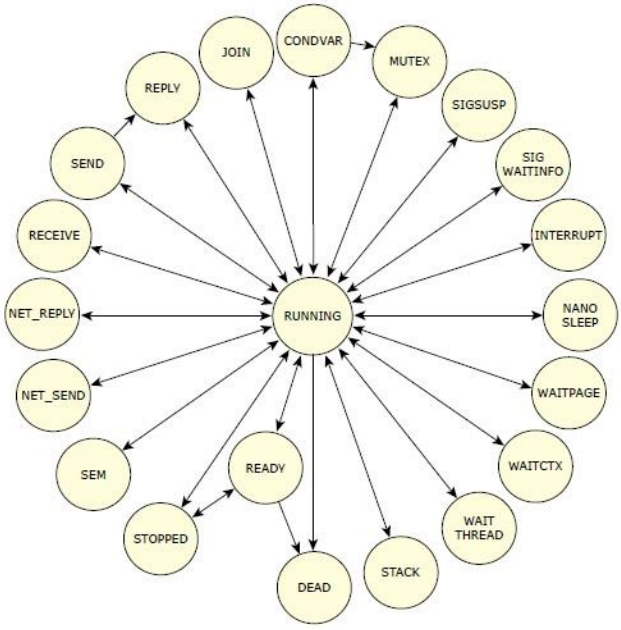

Figura 2.3 Possibili stati thread. Un thread può passare da qualsiasi stato, tranne DEAD, a READY [5]

L’esecuzione del thread corrente può essere temporaneamente sospesa in risposta ad una kernel call, un’eccezione o un interrupt hardware. Il microkernel effettua le scelte di schedulazione ogni volta che un thread cambia stato. Un cambio di contesto può quindi avvenire solo in tre casi. Il thread corrente infatti può perdere l’utilizzo della CPU se si blocca (per esempio su una mutex, se resta in attesa di un messaggio, ecc.), se un thread a priorità maggiore passa allo stato READY o se decide volontariamente di liberare il processore (*sched_yeld()*) nel qual caso viene posto alla fine della coda dei thread pronti.

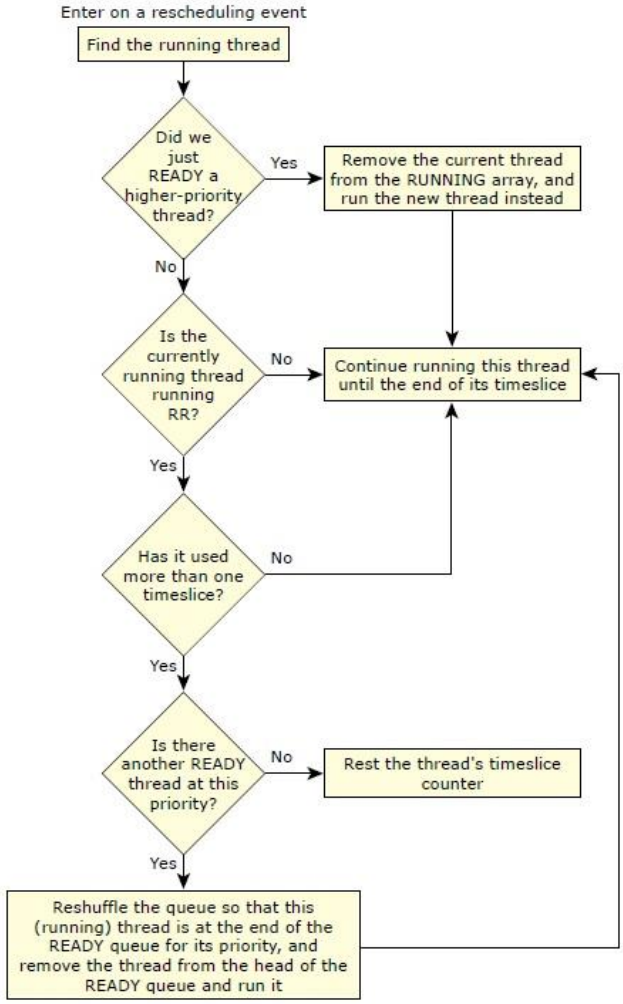

Figura 2.4 Regole per la schedulazione [6]

Ad ogni thread è assegnata una priorità indipendentemente dal processo di cui fa parte. Il sistema operativo supporta 256 livelli di priorità. I

processi senza privilegi possono modificare la propria priorità solo nell'intervallo da 1 (priorità minima) a 63. Solo i thread con privilegi di root possono utilizzare le priorità superiori a 63. La priorità di un thread è determinata alla creazione, ma può essere modificata in seguito.

Per prevenire l'inversione della priorità, il kernel può aumentare temporaneamente la priorità di un thread, questo nel caso di variabili condivise e processi sospesi tramite mutex.

QNX Neutrino fornisce tre diverse politiche di schedulazione:

- FIFO
- round-robin
- sporadic

Ogni thread può eseguire usando una qualsiasi di queste politiche, infatti non vengono applicate globalmente ma thread per thread.

Nella schedulazione FIFO un thread prosegue la sua esecuzione finché non rilascia "volontariamente" il controllo o finché non subisce preemption da un thread a priorità maggiore.

La schedulazione round-robin è identica al caso FIFO ma ad ogni thread è assegnato un intervallo temporale al termine del quale il processo viene inserito in fondo alla ready queue. L'intervallo temporale è pari a 4 volte il periodo di clock.

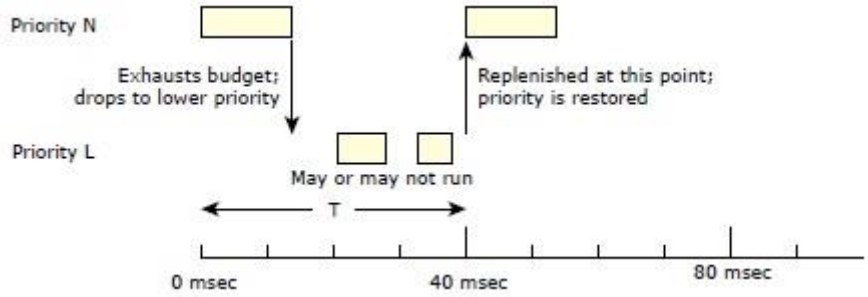

Figura 2.5 Esempio di schedulazione tramite sporadic scheduling [5]

Il caso dello sporadic scheduling è utilizzato solitamente per processi di controllo. Per ogni processo, oltre alla priorità base, vengono definiti altri quattro parametri. Un budget temporale iniziale, una seconda priorità

(inferiore alla priorità base del processo), un periodo di ripristino della capacità e il numero massimo di volte in cui la capacità potrà essere ripristinata.

Il processo esegue alla propria priorità base finché non termina la propria capacità, in quel momento la sua priorità verrà abbassata alla priorità di "riposo". Allo scadere del periodo la sua capacità verrà ripristinata e la sua priorità riportata al livello base. Nel caso in cui il processo venga bloccato durante l'esecuzione a priorità base il periodo di ripristino verrà calcolato a partire dall'inizio di ogni intervallo in base alla capacità consumata durante quell'intervallo.

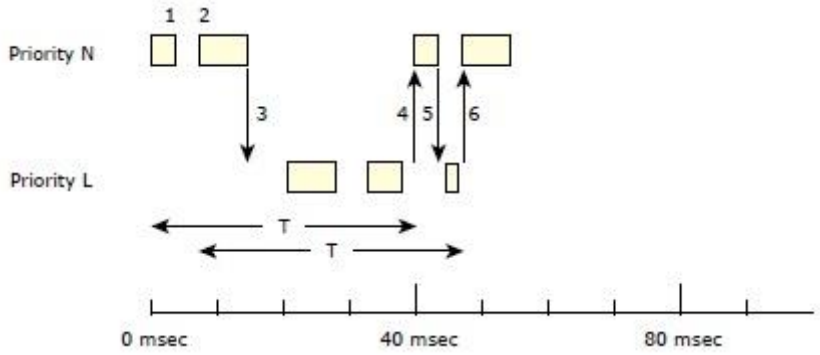

Figura 2.6 Esempio di schedulazione tramite sporadic scheduling, nel quale il thread non esegue in maniera continuativa. Questo causa un ripristino della capacità non completo ma dipendente dal tempo di esecuzione del thread [5]

2.3.2 Barriere

Le barriere sono uno strumento di sincronizzazione per i thread. Grazie ad esse è possibile sospendere un numero prefissato di thread facendo in modo che l'esecuzione prosegua solo dopo che tutti abbiano finito di operare.

Per prima cosa è necessario creare una barriera, specificando il numero di thread che dovranno attendere su di essa. Dopodiché ogni thread, al termine della sua parte di esecuzione, si sospenderà sulla barriera. Nel momento in cui l'ultimo thread arriverà a sospendersi sulla barriera questa si sbloccherà liberando tutti i processi che a quel punto potranno proseguire l'esecuzione in base alle proprie priorità.

2.3.3 Semafori

I semafori sono una delle più comuni forme di sincronizzazione. Si tratta di contatori a cui è possibile accedere con due operazioni. La post (*sem_post()*) incrementa il valore del semaforo, la wait (*sem_wait()*) lo decrementa. Nel caso in cui la wait venga effettuata su un semaforo con un valore non positivo il thread rimarrà bloccato finchè un altro thread non eseguirà una post sullo stesso semaforo.

2.3.4 Clock

Tramite diverse funzioni è possibile accedere al contatore del clock che mantiene il tempo del sistema. Il SO tende a limitare il clock in base all'hardware su cui esegue e per ridurre il peso degli interrupt. Attualmente il valore di default per il clock è pari a 1KHz, ma è possibile modificarlo tramite la kernel call *ClockPeriod()* nei limiti dell'hardware utilizzato (attualmente il periodo minimo impostabile è pari 10 µs). Il clock del sistema operativo determina anche la precisione massima dei timer.

2.3.5 Timer

Data la necessità di tener conto del trascorrere del tempo e della presenza di scadenze prefissate (deadline, periodi, ecc.) i timer sono uno strumento fondamentale per la gestione di un sistema real-time.

Prima che venga descritto il funzionamento si vuole ricordare che per quanto sia precisa la risoluzione temporale nominale di un timer, questa non può essere superiore alla periodo di clock. Nel caso di QNX i timer hanno una risoluzione pari al ns (nanosecondo), ma il periodo minimo impostabile è pari a 10 µs.

È possibile creare timer con una scadenza unica oppure che inviino una notifica allo scadere di ogni periodo. Nel primo caso la scadenza può essere indicata in maniera assoluta o relativa.

Una volta deciso che tipo di timer si vuole utilizzare bisogna scegliere come questo notificherà la scadenza. Ci sono tre tipi di notifica possibile:

invio di un messaggio particolare, detto pulse (v. par. 2.4.1), l'invio di un segnale o la creazione di un thread.
Nel primo caso sarà necessario che un server riceva il pulse (con parametri specificabili) e agisca in base a quanto ricevuto nel messaggio. In caso di invio di un segnale, il gestore dei segnali dovrà essere impostato in modo da intercettarlo ed eseguire la routine desiderata. Infine nel caso di avvio di un thread, come nel caso della creazione tramite *pthread_create()*, verrà avviato un nuovo thread (facente riferimento ad una funzione specifica) con i propri parametri e attributi impostati durante la creazione del timer.
Una volta avviati i timer possono essere fermati prima della scadenza e volendo è possibile modificare i loro parametri temporali.
Un'altra funzione per cui vengono utilizzati i timer riguarda il timeout delle operazioni bloccanti. È infatti possibile, prima di eseguire un'operazione bloccante (come per esempio una wait su un semaforo), assegnarle un timer allo scadere del quale il processo verrà sbloccato e l'operazione ritornerà un codice di errore specifico per indicare che è terminata a causa del timeout.

2.4 Comunicazione tra processi

Un servizio fondamentale per la coordinazione tra processi è la comunicazione tra questi (Interprocess Communication, IPC). QNX mette a disposizione i seguenti tipi di servizi (in grassetto quelli utilizzati nel simulatore): **passaggio di messaggi** e segnali (implementati dal kernel), code di messaggi POSIX, **memoria condivisa**, **pipe** e FIFO (implementate da processi esterni al kernel).

2.4.1 Messaggi

QNX mette a disposizione un classico sistema di messaggistica sincrono, in cui ogni thread che voglia inviare un messaggio resterà in attesa sulla

MsgSend() fino a quando il thread ricevente non avrà effettuato una *MsgReceive()*. Naturalmente, in condizioni normali, un processo che effettui una *MsgReceive()* senza che siano presenti messaggi in coda si sospenderà nell'attesa di un messaggio. Oltre a questo sistema sincrono, QNX mette a disposizione i **pulse**. Questi sono dei messaggi a dimensione fissa, formati da due campi: Code (8 bit) e Value (32 bit). La particolarità di questi messaggi è che il loro invio è asincrono, quindi il processo mittente non si sospenderà in attesa che il messaggio venga ricevuto. Inoltre si ricorda che i timer, al loro scadere, possono inviare un pulse, risultando quindi degli ottimi strumenti di notifica.

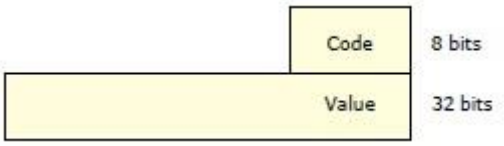

Figura 2.7 Pulse [5]

Ogni messaggio è caratterizzato da una priorità e il server li riceve ed elabora in ordine di priorità. Inoltre, se si è deciso di utilizzare l'ereditarietà della priorità, il server eseguirà il codice di risposta ad un determinato messaggio alla priorità del messaggio, cioè alla priorità del thread mittente.

In QNX il passaggio dei messaggi avviene attraverso i canali e le connessioni invece che essere diretto da thread a thread.

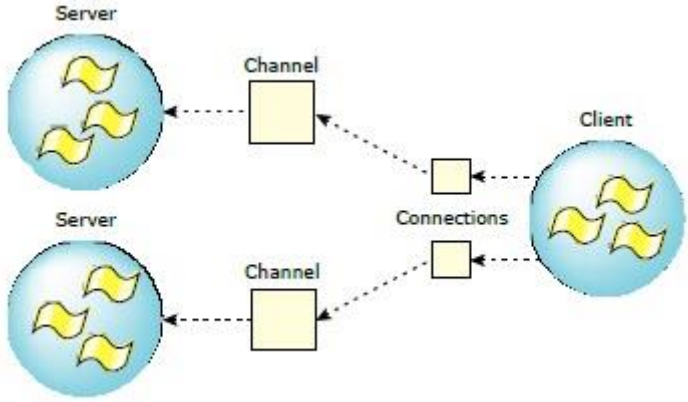

Figura 2.8 Canali e connessioni [5]

Un thread che desideri ricevere messaggi deve come prima cosa creare un canale (*ChannelCreate()*); un altro thread che desideri inviare un messaggio al thread "server" deve prima creare una connessione (*ConnectAttach()*) collegata al suo canale. Il server effettuerà le proprie *MsgReceive()* riferendosi al proprio canale, mentre i client invieranno i messaggi (*MsgSend()*) tramite le connessioni. Le connessioni vengono direttamente mappate tra i descrittori dei file, in questo modo si elimina un livello di indirettezza, migliorando l'efficienza. Più connessioni possono collegarsi ad uno stesso canale. Se più thread di uno stesso processo si collegano allo stesso canale tramite più connessioni il kernel per efficienza le mapperà sullo stesso oggetto.

2.4.2 Memoria condivisa

Il simulatore è costituito da un unico processo in cui vengono creati più thread che quindi condividono lo spazio di indirizzamento.

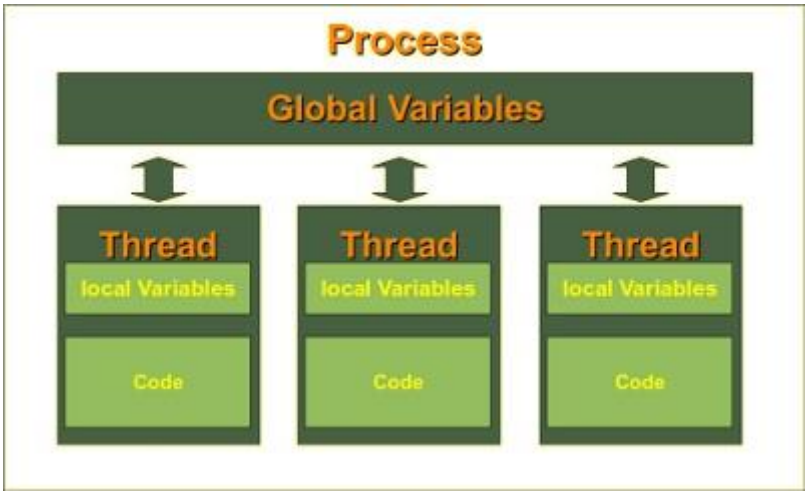

Figura 2.9 Processi e thread [7]

2.4.3 Pipe

Le pipe sono delle code che collegano i processi. Dato che non sono gestite dal sistema operativo è necessario che il loro gestore sia avviato perché possano essere utilizzate (*pipe*).

Le pipe sono dei file che servono come canale di comunicazione tra due processi cooperanti: un processo scrive mentre l'altro legge. Una volta

che entrambe le estremità di una pipe vengono chiuse questa viene rimossa.

Capitolo 3 - Strumenti utilizzati

In questo capitolo verranno descritti brevemente gli strumenti applicativi e di ricerca utilizzati per la programmazione e il test del simulatore di un sistema real-time, obiettivo di questa tesi.

3.1 VMware Player

Per comodità, si è deciso di utilizzare una macchina virtuale (questa scelta è consigliata anche dal produttore). La scelta del software da utilizzare è ricaduta su VMware per due motivi: è un software gratuito ed è stato utilizzato in altri progetti in passato facendo quindi risultare semplice il suo utilizzo.

3.1.1 Macchina virtuale

Seguendo le impostazioni consigliate dal produttore, e tenendo conto dell'utilizzo primario del sistema operativo QNX, sistemi embedded, si è utilizzata per l'installazione una macchina virtuale con i seguenti parametri:

- 256 MB di RAM;
- 1 processore;
- hard disk da 8 GB.

L'utilizzo della macchina virtuale si è rivelato estremamente comodo in fase di debug in quanto è stato spesso necessario riavviare il sistema a causa di deadlock imprevisti.

3.2 QNX Momentics Tool Suite

Come in ogni caso di programmazione, sarebbe possibile programmare tramite un editor di testo e utilizzare direttamente il compilatore presente sul sistema operativo per cui si vuole programmare per ottenere l'eseguibile. Per fortuna però, agli sviluppatori QNX viene fornito un ***IDE*** (**I**ntegrated **D**evelopment **E**nvironment) basato su Eclipse estremamente

completo che, oltre alle funzionalità base di ogni IDE, mette a disposizione un gran numero di strumenti extra che facilitano il debug e la verifica del sistema su cui si sta lavorando. L'IDE esiste sia in versione Windows che Linux. Nel caso di questa tesi si è lavorato con la versione Windows.

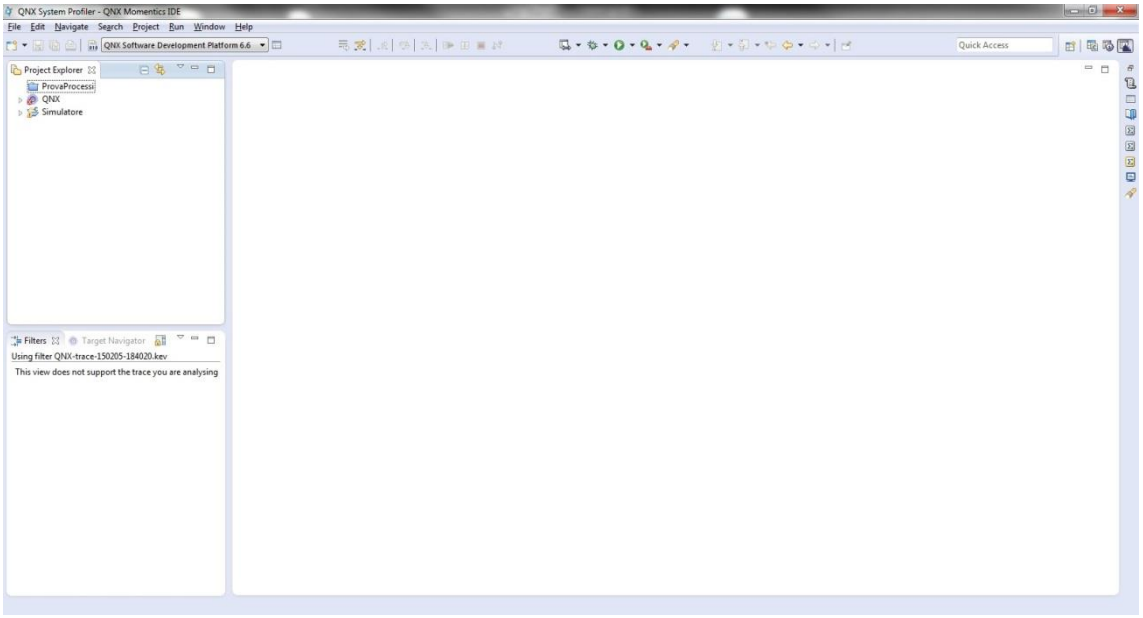

Figura 3.1 QNX IDE

3.2.1 Sistema di test

All'interno dell'IDE è possibile designare uno o più sistemi a cui collegarsi tramite rete. In questo modo la compilazione e l'avvio della applicazione verrà svolto dall'IDE, ma direttamente sul sistema preso in esame. Anche l'output verrà intercettato e trasferito sulla console dell'IDE. In questo modo lo sviluppo può procedere più velocemente, senza bisogno di passare ogni volta il codice sulla macchina di test (nel nostro caso una macchina virtuale) per compilarlo e testarlo, ma è sufficiente che il sistema di test sia collegato in rete. Una volta che il sistema è stato collegato è possibile accedere agli strumenti di visualizzazione dello stato del sistema.

Il sistema è caratterizzato dai processi in esecuzione su di esso. Questi vengono divisi tra i processi utente e i processi di sistema (cioè quelli

necessari ai servizi del sistema operativo). Sempre attraverso l'IDE è possibile visualizzare lo stato corrente della memoria e della CPU del sistema in esame. È inoltre possibile visualizzare e operare direttamente sul file system, tramite una comoda interfaccia grafica senza aver bisogno di accedere direttamente alla macchina con installato sopra QNX.

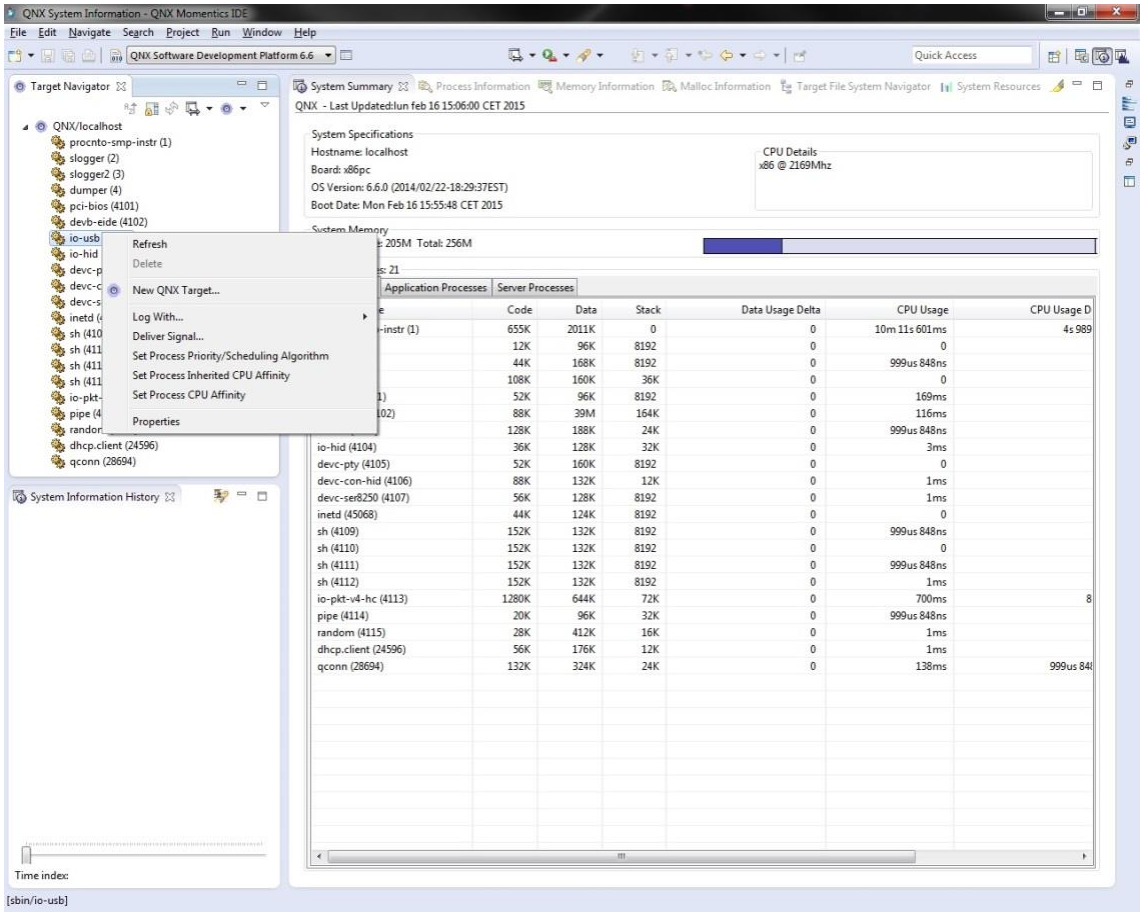

Figura 3.2 Informazioni di sistema visualizzate nell'IDE

Oltre a poter lavorare sul file system è possibile anche agire sui vari processi in esecuzione. Dall'IDE è possibile inviare segnali e modificare la priorità o l'algoritmo di schedulazione di ogni processo.

Oltre alle informazioni sull'intero sistema si possono ottenere informazioni estremamente dettagliate su ogni singolo processo. Dopo aver selezionato il processo desiderato si può verificare il suo stato completo. Questo include l'insieme dei thread di cui è composto, il loro stato e, in

caso di sospensione, il motivo del blocco. Inoltre per ogni thread viene visualizzato il tempo totale dedicatogli dalla CPU, o da ogni processore in caso di sistemi multiprocessore. Oltre a queste informazioni è possibile anche visualizzare l'uso che ogni processo fa della memoria di sistema.

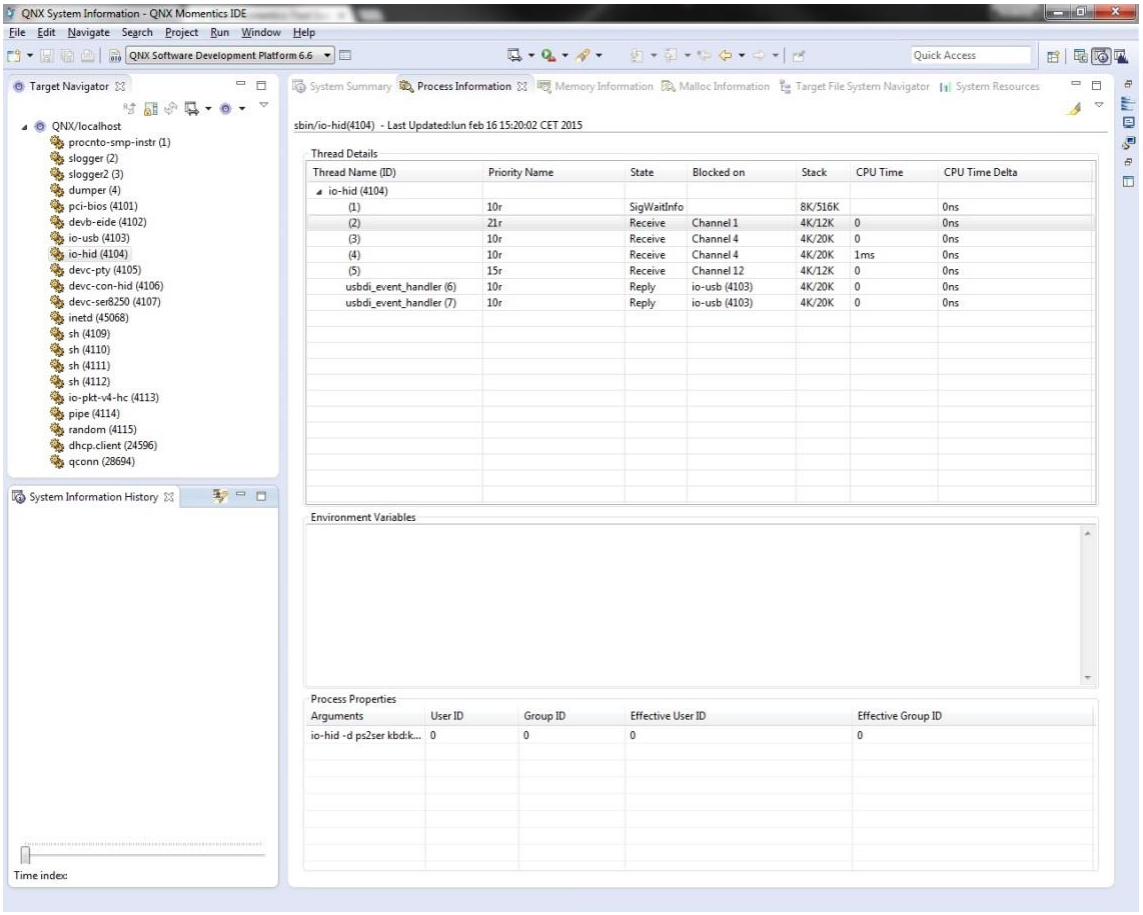

Figura 3.3 Informazioni processo

3.3 Sistema di log

QNX mette a disposizione un sistema di log estremamente accurato. Sempre attraverso l'IDE o agendo direttamente dalla shell di sistema, lo si può avviare per un tempo specificato, al termine del quale, verrà salvato su un file il log del sistema per il periodo specificato. È possibile modificare i parametri del sistema di log per raccogliere solo le informazioni desiderate, scegliendo tra gli eventi di sistema e i processi, solo quelli che si vogliono analizzare. Inoltre ogni evento è analizzabile in

due diversi modi: fast o wide. Nel primo caso si raccolgono solo le informazioni essenziali quali il tempo di arrivo, mentre nel secondo caso vengono salvate più informazioni. Bisogna tenere presente che il sistema di log necessita comunque di un minimo delle risorse di sistema e bisogna stare attenti a non sovraccaricarlo. In caso contrario i log ottenuti rischiano di non essere completi o di non essere correttamente ordinati temporalmente. Per esempio, se non viene rilevato un evento di thread running, un thread in esecuzione non viene visualizzato nel file di log.

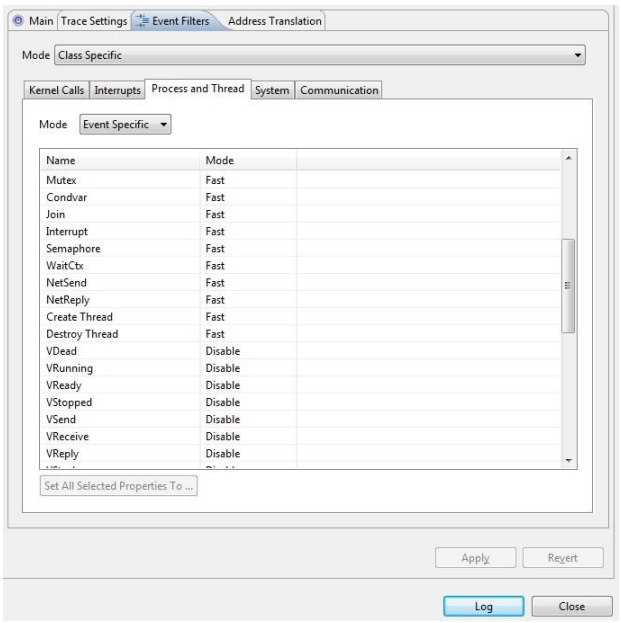

Figura 3.4 Opzioni configurabili del log di sistema

Una volta ottenuto il file di log tramite la rete, l'IDE permette di accedervi attraverso un sistema proprio di visualizzazione dei dati. Oltre a un sommario, in cui è possibile visualizzare l'insieme dei processi analizzati (e dei loro thread) con tutti i tempi relativi (running, ready, blocked) e altre informazioni, quali il numero di kernel call effettuate e di messaggi inviati, esistono altre possibili visualizzazioni.

La CPU activity permette di visualizzare tramite grafico l'andamento dell'utilizzo della CPU nel tempo. Sempre in riferimento all'utilizzo del microprocessore, per ognuno è possibile visualizzare esattamente come è stato impiegato nel corso del periodo di log. Nel caso di sistemi

multiprocessore si possono visualizzare le comunicazioni tra i diversi processori e la migrazione dei thread da una CPU all'altra, ma nel caso in analisi il lavoro si è svolto utilizzando un sistema monoprocessore.

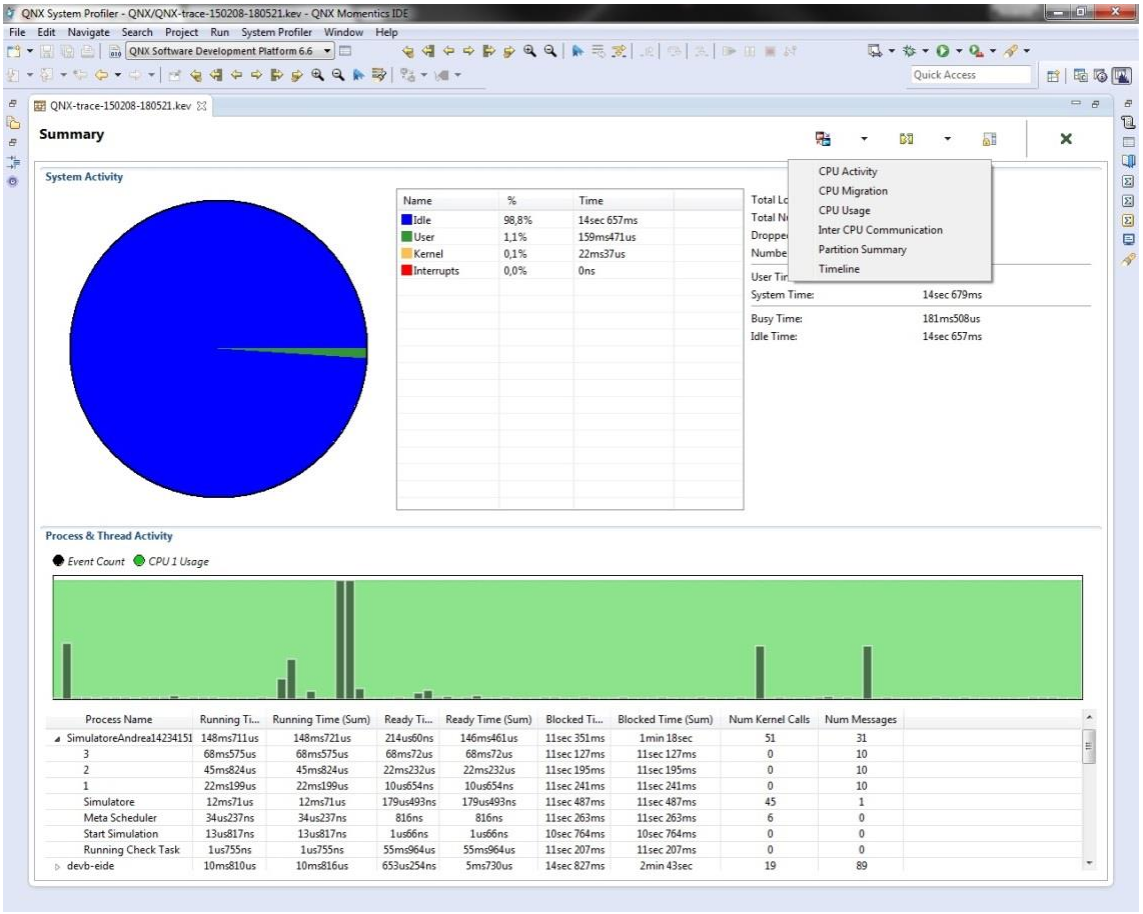

Figura 3.5 Sommario log

La visualizzazione che è risultata più utile per analizzare i dati nel caso di questa tesi è la cronologia (timeline). Tramite questa è possibile rendere visibile l'andamento di tutti i processi e dei loro thread nel tempo. Tra i filtri, oltre a poter selezionare i processi e i thread da rappresentare, è possibile anche selezionare i tipi di evento che si vogliono visualizzare. L'IDE indica con colori diversi i vari stati dei thread e utilizza delle barre nere verticali per gli eventi. Ad ogni evento sono associati i suoi parametri, ma questi non vengono esposti direttamente per non rendere illeggibile la cronologia.

Una volta deciso cosa visualizzare si possono aggiungere in automatico delle etichette specifiche per indicare la modifica della priorità dei thread, il cambio di stato o eventi specifici. Queste ultime etichette sono da configurare associando ad ogni evento un nome ed eventualmente i parametri propri (tid, pid, ecc.) dell'evento che si vogliono vedere scritti nell'etichetta. Nello specifico per poter visualizzare i risultati della tesi in maniera chiara, sono stati associati alcuni eventi a determinate condizioni durante l'esecuzione del simulatore al fine di visualizzare nella cronologia gli eventi di simulazione verificatisi.

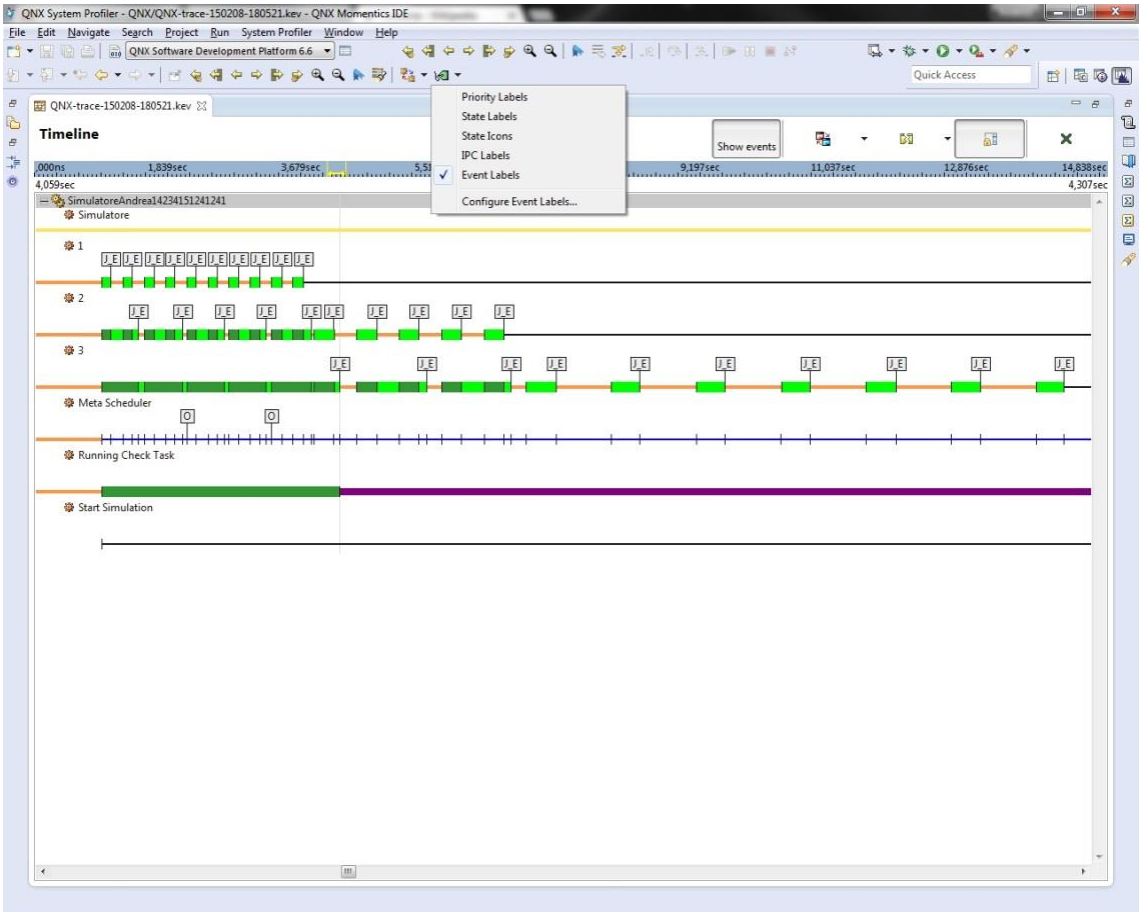

Figura 3.6 Esempio di cronologia ottenuta col sistema di log

Naturalmente la cronologia rappresenta lo scorrere del tempo tramite una linea temporale in cima alla schermata, ma non sempre il tempo assoluto (o comunque relativo all'inizio del log) risulta facilmente interpretabile. È

quindi possibile selezionare un intervallo per avere in automatico il tempo relativo.

3.4 Strumenti di ricerca

Per poter scrivere il codice del simulatore è stato necessario imparare il funzionamento delle varie funzioni e librerie offerte dal sistema operativo QNX. L'azienda mette a disposizione degli sviluppatori due importanti strumenti: un motore di ricerca e un portale.

3.4.1 Documentazione

Tutta la documentazione sul sistema operativo, comprese le guide e gli esempi, è accessibile tramite un motore di ricerca dedicato fruibile da tutti [4].

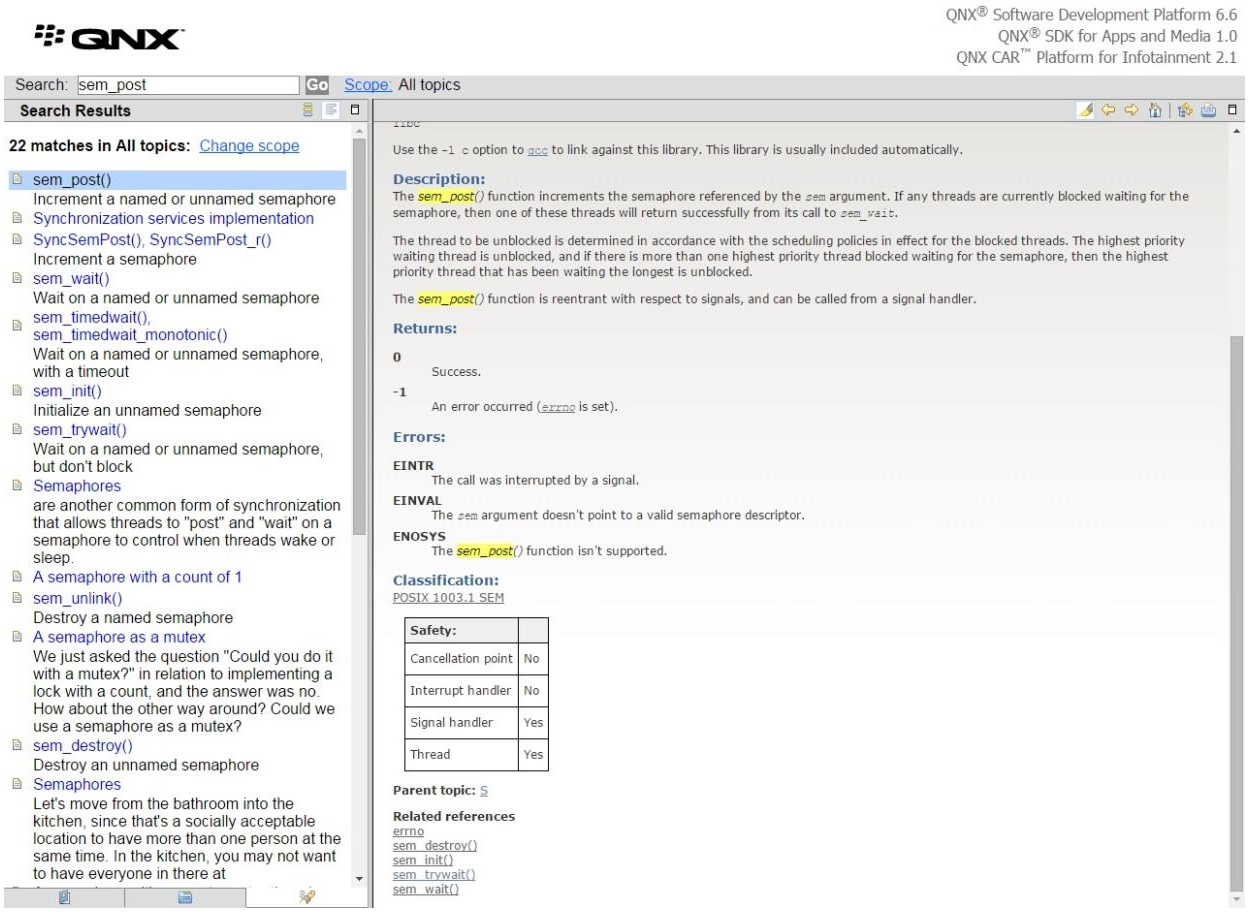

Figura 3.7 Documentazione QNX

3.4.2 Foundry27

Foundry27 [5] è un portale mantenuto dall'azienda comprensivo di un forum, una wiki, un blog, diversi articoli, pubblicazioni e altro materiale relativo a QNX a cui possono accedere gli sviluppatori registrati. In questo modo c'è uno spazio di confronto privilegiato nel quale discutere degli algoritmi da implementare, chiedere conferma di e segnalare eventuali bug, chiedere aiuto se necessario, ecc.

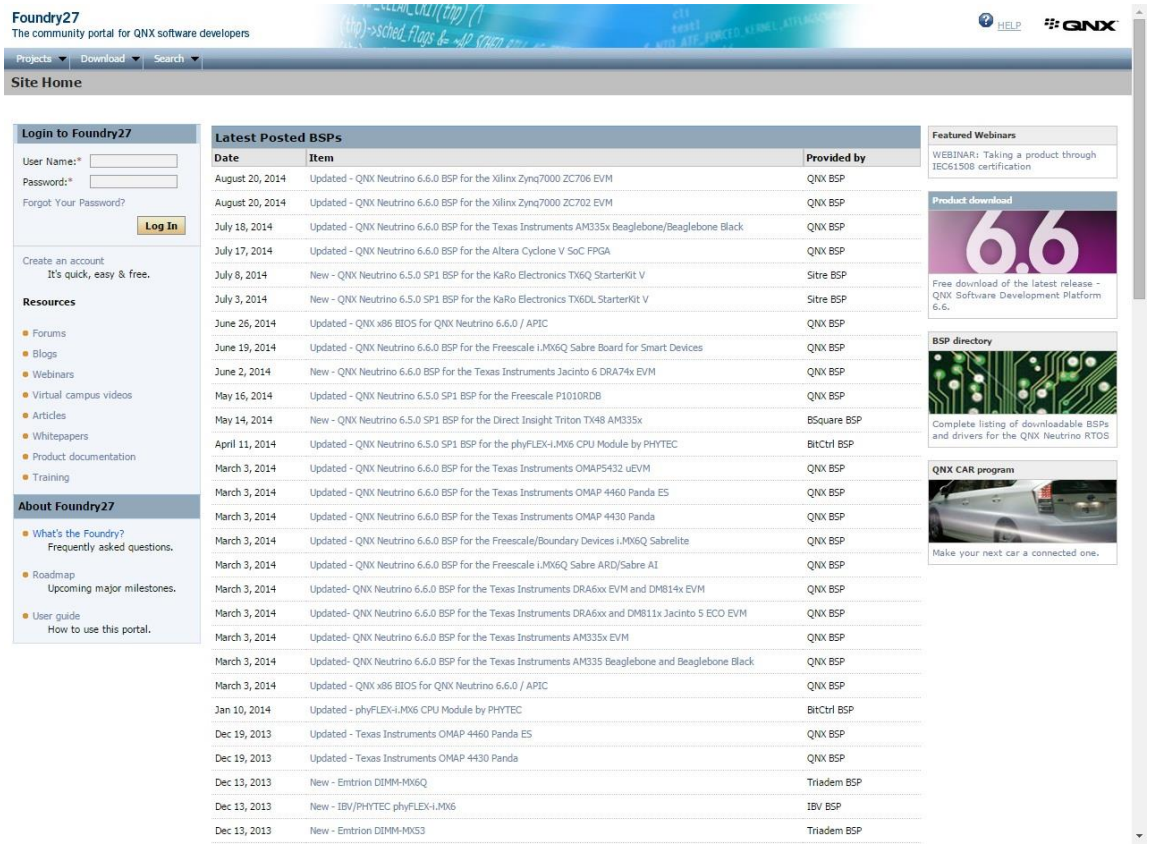

Figura 3.8 Home page di Foundry27 [8]

Capitolo 4 – Simulatore

Il simulatore sviluppato per permettere di studiare e mettere alla prova il sistema operativo QNX Neutrino è in grado di gestire contemporaneamente task periodici e richieste aperiodiche. Dato che le due componenti interagiscono solo tramite la priorità di schedulazione e risultano quindi indipendenti dal punto di vista della programmazione, verranno trattate separatamente. Nelle sezioni di questo capitolo verranno dunque descritte le politiche di schedulazione per i task periodici, la gestione delle richieste aperiodiche tramite server e la loro implementazione. Alcune parti del programma interagiscono sia con i task periodici che con quelli aperiodici e verranno trattate all'inizio del capitolo.
Ad ogni thread, per avere una visualizzazione facilmente comprensibile nel log, viene, come prima operazione, modificato il nome in funzione del suo scopo.

4.1 Parti comuni

Le parti comuni alla gestione dei task periodici e delle richieste aperiodiche sono quattro:

- busy wait;
- thread per il controllo della terminazione della simulazione;
- main;
- thread per l'inizio della simulazione.

Ognuno di questi verrà descritto singolarmente.

4.1.1 Busy wait

Nei sistemi in tempo reale (e più in generale in un sistema generico) i processi (o task) hanno uno scopo che cercano di portare a termine durante la loro esecuzione. Nel nostro caso, dovendo simulare questo comportamento, si è dovuto utilizzare una funzione che mantenesse in esecuzione i task per il tempo di esecuzione previsto (C), la ***Busy_Wait()***.

Oltre alla funzione vera e propria, richiamata dai vari task e dal server di gestione delle richieste aperiodiche, utilizzando come parametro i nanosecondi di esecuzione desiderati (*Busy_Wait_Time_ns)*, è necessaria una funzione di calibrazione (***Busy_Wait_Automatic_Tuning()***). Questa funzione, invocata solo una volta prima dell'avvio della simulazione, esegue per un numero di volte prefissato (*Reference_Time_Evaluations_Number*) la Busy_Wait(), utilizzando come parametro un tempo di riferimento prefissato (*Reference_Time_Iterations_Number*) e ad ogni iterazione viene tenuto conto del tempo effettivo di esecuzione. Al termine delle iterazioni viene calcolato un tempo di riferimento (*Reference_Time_ns*) dato dal rapporto tra il tempo totale necessario al completamento delle Busy_Wait() e il tempo teorico previsto per la singola Busy_Wait(). Infine il tempo di riferimento può essere ulteriormente ridotto di una certa percentuale (*Execution_Time_Dumping_Percentage*) per migliorarne la precisione sulla base della sperimentazione su un sistema specifico.
Quando richiamata, la Busy_Wait(), esegue quindi per un numero di volte dato dal prodotto del parametro di ingresso per il rapporto tra il numero di iterazioni svolte durante la calibrazione e il tempo di riferimento ottenuto.

4.1.2 Controllo terminazione simulazione

Come sarà descritto più avanti la durata teorica della simulazione è definita dai task periodici, ma l'effettiva durata può variare a causa di sovraccarichi dei diversi task o dell'arrivo di richieste aperiodiche. Per evitare che la simulazione venga interrotta prima del tempo e per non aggiungere compiti non previsti a nessuno dei task destinati alla simulazione è stato deciso di aggiungere un task a priorità minima, il cui scopo è verificare il termine della simulazione. Questo thread (***Running_Check_Task_Entry_Point()***) esegue una verifica continua dello stato del server e di ogni task periodico. Nel caso in cui il server sia

terminato (o non sia presente) e ogni singolo task abbia completato tutti i job previsti, il thread termina, risvegliando il main sospeso tramite join (*pthread_join()*) in attesa della terminazione di questo thread.
Avendo il thread una priorità di schedulazione (*LOWEST_PRIORITY*, 200) inferiore a quella di ogni altro thread del simulatore, la sua esecuzione non interferisce in alcun modo con quella della simulazione, risultando così trasparente.

4.1.3 Main

Il main si occupa principalmente dell'avvio dei thread necessari alla simulazione e del settaggio dei parametri.
Come già indicato precedentemente, il sistema operativo, all'avvio, imposta un clock di 1 KHz per ridurre il peso degli interrupt e per avere una compatibilità maggiore con i sistemi embedded su cui viene prevalentemente utilizzato. Per ottenere prestazioni temporali maggiori è necessario migliorare la frequenza di clock. Questo è possibile tramite la kernel call *ClockPeriod()*. Nel caso in questione il periodo viene portato a 10000 nanosecondi (10 µs), il minimo impostabile dal sistema operativo.
Dopodiché il main si occupa della lettura dei parametri riguardanti i task periodici e le richieste aperiodiche. Una volta fatto ciò, in base alla politica di schedulazione prevista (v. par. 4.2) i task periodici vengono riordinati per priorità decrescente. Al termine di queste operazioni viene effettuata la calibrazione della *Busy_Wait()*. Una volta completata questa operazione viene creata la pipe destinata alle richieste aperiodiche. Si arriva quindi alla impostazione delle priorità dei diversi thread.
È stato deciso di utilizzare una schedulazione di tipo FIFO per ogni thread della simulazione e di sfruttare le priorità maggiori per ridurre al minimo le interferenze da parte dei servizi del sistema operativo. Sono state quindi definite diverse priorità. *QNX_HIGHEST_PRIORITY* pari a 255, è stata lasciata libera per poterla eventualmente sfruttare per altri servizi in

futuro. La priorità immediatamente inferiore, definita *META_SCHEDULER_PRIORITY*, è stata riservata al meta-scheduler, il cui scopo è la gestione di tutti i task periodici, dall'avvio alla terminazione. La priorità minima definita è la *LOWEST_PRIORITY* (200), utilizzata per il thread di controllo della terminazione della simulazione. La priorità subito superiore, definita *BACKGROUND_SERVER_PRIORITY* viene, come dice il nome stesso, utilizzata per il server di gestione delle richieste aperiodiche a priorità minima. Il simulatore dispone quindi dei livelli di priorità compresi tra 254 e 201 per la gestione delle richieste aperiodiche e dei task periodici.

Nel caso in cui la simulazione richieda la gestione di richieste aperiodiche e il server non sia di tipo background le priorità maggiori vengono riservate ai thread relativi alla gestione delle richieste. Oltre all'impostazione delle priorità, in questo frangente vengono anche impostati i semafori necessari alla gestione dei server e si effettua la conversione dei parametri temporali da millisecondi, utilizzati nell'impostazione dei parametri, a nanosecondi.

A seguito dell'eventuale impostazione del server, si passa alla gestione dei task periodici. Partendo dal task a cui è associata la priorità maggiore, ad ogni task viene assegnata una effettiva priorità di esecuzione. Anche in questo caso si effettua la conversione dei diversi parametri temporali da millisecondi a nanosecondi, unità di misura temporale in QNX. Per ogni task viene inoltre inizializzato un semaforo.

Prima dell'avvio dei thread necessari alla simulazione e per fare in modo che l'avvio sia simultaneo viene impostata una barriera (*pthread_barrier_init()*) con valore pari alla somma del numero di task periodici, il numero di task necessari alla gestione dei server (compreso tra 2 e 4) e 3. Questi ultimi tre thread sono il meta-scheduler, il thread per il controllo della terminazione della simulazione e il thread incaricato di sbloccare la barriera al tempo di avvio della simulazione (*Start_Time_ns*).

Quest'ultimo viene calcolato aggiungendo al tempo corrente un valore preimpostato (*Start_Delay_ms)* per fare in modo che tutti i thread siano in grado di completare le operazioni preliminari necessarie alla simulazione.
Fatto ciò il main comincia a creare i diversi thread: i thread per la gestione delle richieste aperiodiche, i thread corrispondenti ai task periodici, il meta-scheduler e il thread per il controllo della terminazione della simulazione. Infine imposta un timer assoluto con valore pari al tempo di avvio, allo scadere del quale verrà creato il thread di avvio.
A questo punto il main si sospende tramite join, in attesa del completamento della simulazione.

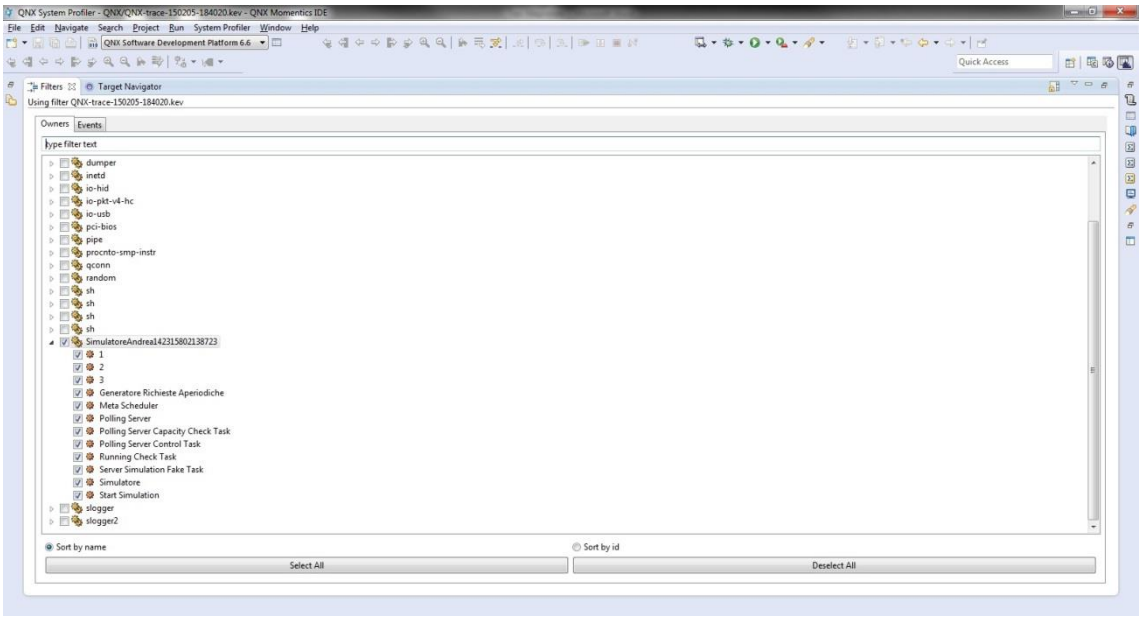

Figura 4.1 Esempio di simulazione, da notare il numero di thread del processo

Al termine di questa invia un messaggio (*MsgSendPulse()*) al meta-scheduler per segnalare la fine della simulazione e rilascia tutte le risorse precedentemente allocate: la pipe per le richieste aperiodiche, la barriera, il timer per l'inizio della simulazione e i semafori utilizzati dal server e dai task periodici.

4.1.4 Inizio della simulazione

Per garantire che l'avvio della simulazione sia sincronizzato, ogni thread si sospende sulla barriera creata per questo scopo (*Starting_Barrier*). Allo

scadere del timer impostato col tempo di avvio viene avviato un thread (***Starting_Simulation_Task_Entry_Point()***) che come unica operazione si sospende sulla barriera (*pthread_barrier_wait()*). In questo modo, dato che tutti gli altri thread sono già sospesi, la barriera si sblocca e la simulazione ha inizio.

4.2 Gestione dei task periodici

Nella letteratura ogni task periodico ha associati una serie di parametri:

- il nome, che serve ad identificarlo;
- il tempo di esecuzione, solitamente costante, indicato con C (computation time);
- il periodo tra l'avvio di un job e l'avvio del successivo, indicato con T;
- la priorità di esecuzione, dipendente dalla politica di schedulazione in uso, indicata con P;
- la deadline, il tempo rispetto all'avvio di un job entro il quale questo deve essere completato, indicata con D, se non specificata corrisponde al periodo T.

È possibile anche che ogni task per svolgere i propri compiti richieda l'accesso a risorse condivise, ma il simulatore sviluppato non prevede al momento questa ipotesi.

I parametri di inizializzazione corrispondono esattamente a questi e sono espressi in millisecondi, ma per eseguire la simulazione ad ogni task sono necessari più parametri. Oltre ai corrispondenti parametri in nanosecondi, ad ogni task sono associati: un contatore indicante il numero di job correnti, il numero di job totali da eseguire, lo stato, l'identificatore del thread associato, un timer usato per il periodo, un semaforo. Inoltre durante l'esecuzione di ogni job vengono registrati il tempo di rilascio e il tempo di completamento, utilizzati nel meta-

scheduler e la durata del job più lungo, utilizzata come informazione per il log interno del simulatore.

Il numero di job da eseguire è determinato all'interno del main in base a quanto specificato nel file dei parametri. È possibile procedere in due modi: tramite un numero predefinito di job per ogni task, oppure eseguire la simulazione per un tempo predefinito. Nel secondo caso ad ogni task viene associato il numero di job necessari in base al proprio periodo di esecuzione.

Un task, durante la simulazione, può trovarsi in 5 stati diversi:

- *RUNNING*, il task è in esecuzione o è sospeso perché un thread a priorità maggiore impegna il microprocessore;
- *MISSED_DEADLINE*, come RUNNING ma il task ha superato la propria deadline, quindi non è riuscito a completare la sua esecuzione in tempo;
- *JOB_OVERRUN*, come RUNNING ma è stato superato il periodo e non è stato possibile avviare il job previsto in quanto quello corrente non è ancora terminato;
- *IDLE*, il task ha completato la sua esecuzione ed attende lo scadere del periodo per tornare in esecuzione;
- *END_OF_EXECUTION*, il task ha completato tutti i suoi job e il thread associato è terminato.

L'aggiunta dello stato ad ogni task periodico è stata decisa solo in un secondo momento. Inizialmente le informazioni necessarie al meta-scheduler venivano ricavate esclusivamente dai dati temporali, ma il tutto risultava poco leggibile. In questo modo invece il codice risulta leggibile ed è possibile costruire uno schema temporale dell'evoluzione dei diversi task utilizzando i cambiamenti di stato.

L'implementazione del task risulta estremamente semplice in quanto tutta le gestione dei cambiamenti di stato e in generale dei fattori temporali è delegata al meta-scheduler che si occupa di implementare ed applicare le

diverse politiche di schedulazione possibili. Prima di descrivere le diverse politiche e la loro implementazione tramite meta-scheduler verrà descritta nel dettaglio l'implementazione dei task periodici.

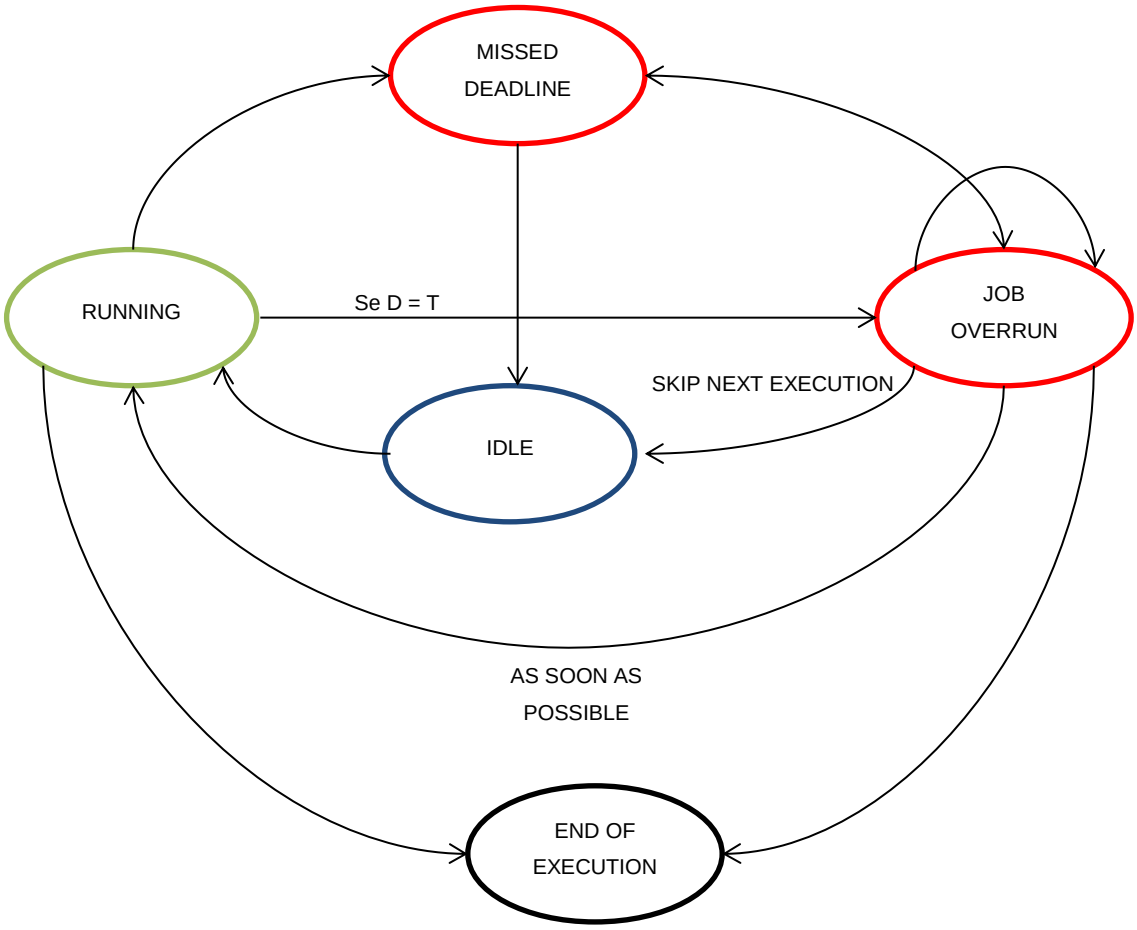

Figura 4.2 Stati di un task periodico e possibili transizioni

4.2.1 Implementazione task periodico

Per ogni task periodico viene creato un thread a cui viene associato il task specifico. Ogni thread dopo aver impostato il proprio nome sulla base del task associato, si sospende sulla barriera in attesa dell'inizio della simulazione.

Una volta che la simulazione ha avuto inizio, il thread esegue ripetutamente una *Busy_Wait()* per una durata pari al tempo di esecuzione specificato nei parametri e letto dal main. Subito prima e appena terminata l'esecuzione vengono notificati l'inizio e la fine dell'esecuzione di un job (v. par. 4.4 e successivi). Una volta terminata l'esecuzione della *Busy_Wait()* il task invia un pulse di tipo *END_JOB_PULSE_CODE* inserendo come parametro il proprio ID (1, 2, 3, ecc.) al meta-scheduler. Infine si sospende tramite una wait (*sem_wait()*) sul proprio semaforo.
Il thread prosegue ciclicamente la propria esecuzione fino a quando il suo stato non viene modificato dal meta-scheduler in *END_OF_EXECUTION* ad indicare che il task ha terminato il numero di job a lui assegnati e il thread associato può terminare.

Prima della descrizione del meta-scheduler verranno descritte le politiche di schedulazione da esso implementate.

4.2.2 Politiche di schedulazione

Nel caso in esame le politiche implementate sono tutte "priority-driven", cioè basate sull'attribuzione ad ogni task di una priorità di esecuzione specifica. Questo può avvenire staticamente o dinamicamente. L'esecuzione di un task è sospesa nel caso in cui un altro task a priorità superiore sia pronto per l'esecuzione (preemption).
I tre algoritmi di schedulazione implementati sono: ***RMPO*** (**R**ate **M**onotonic **P**riority **O**rdering), ***DMPO*** (**D**eadline **M**onotonic **P**riority **O**rdering) e ***EDF*** (**E**arliest **D**eadline **F**irst).
Il primo algoritmo assegna staticamente ad ogni processo una priorità direttamente proporzionale alla corrispondente frequenza di esecuzione: $p(P_j) = p_j = 1 / T_j$. Solitamente nel caso di schedulazione di tipo RMPO

non è prevista nessuna deadline specifica. Ogni processo deve terminare la sua esecuzione entro il rilascio del job successivo.

Se un insieme di processi periodici è schedulabile (ogni task riesce sempre a concludere la sua esecuzione prima del rilascio del job successivo) con un qualche algoritmo che preveda un'assegnazione statica di priorità, allora tale insieme è schedulabile anche utilizzando RMPO. Nel caso questo non sia vero, cioè se l'insieme non è schedulabile tramite RMPO, allora tale insieme non è schedulabile con nessun altro algoritmo ad attribuzione statica di priorità.

A_2	C	T
P_1	15	25
P_2	5	50
P_3	17.5	100

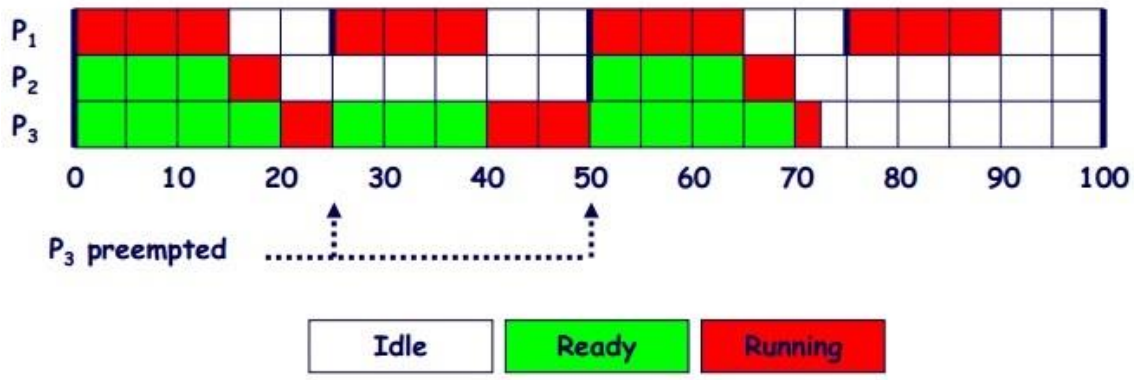

Figura 4.3 Esempio di schedulazione tramite RMPO, i task sono ordinati per priorità decrescente [9]

Alcuni processi, detti sporadici, caratterizzati solitamente da una bassa frequenza di esecuzione, impongono però vincoli stringenti riguardo al completamento della propria esecuzione. Da questa necessità nasce l'algoritmo DMPO, come una semplice estensione di RMPO.

Si procede quindi ad assegnare ad ogni task una deadline entro la quale l'esecuzione del task deve essere completata con $D_j <= T_j$. La deadline

sarà minore del periodo del task relativo nel caso di processi sporadici e uguale al periodo nel caso di task periodici.

La priorità assegnata staticamente ad ogni processo non è più in funzione della frequenza di esecuzione, bensì è inversamente proporzionale alla corrispondente deadline relativa: $p_j = 1 / D_j$.

Se un insieme di processi periodici e/o sporadici è schedulabile con un qualche algoritmo che preveda un'attribuzione statica di priorità, allora tale insieme è schedulabile anche con DMPO. Se un tale insieme non è schedulabile con DMPO, allora non è schedulabile con nessun altro algoritmo che preveda un'attribuzione statica di priorità.

Nel seguito si userà il termine task periodici sia per indicare i processi periodici che sporadici, poiché a livello di programmazione non esiste differenza.

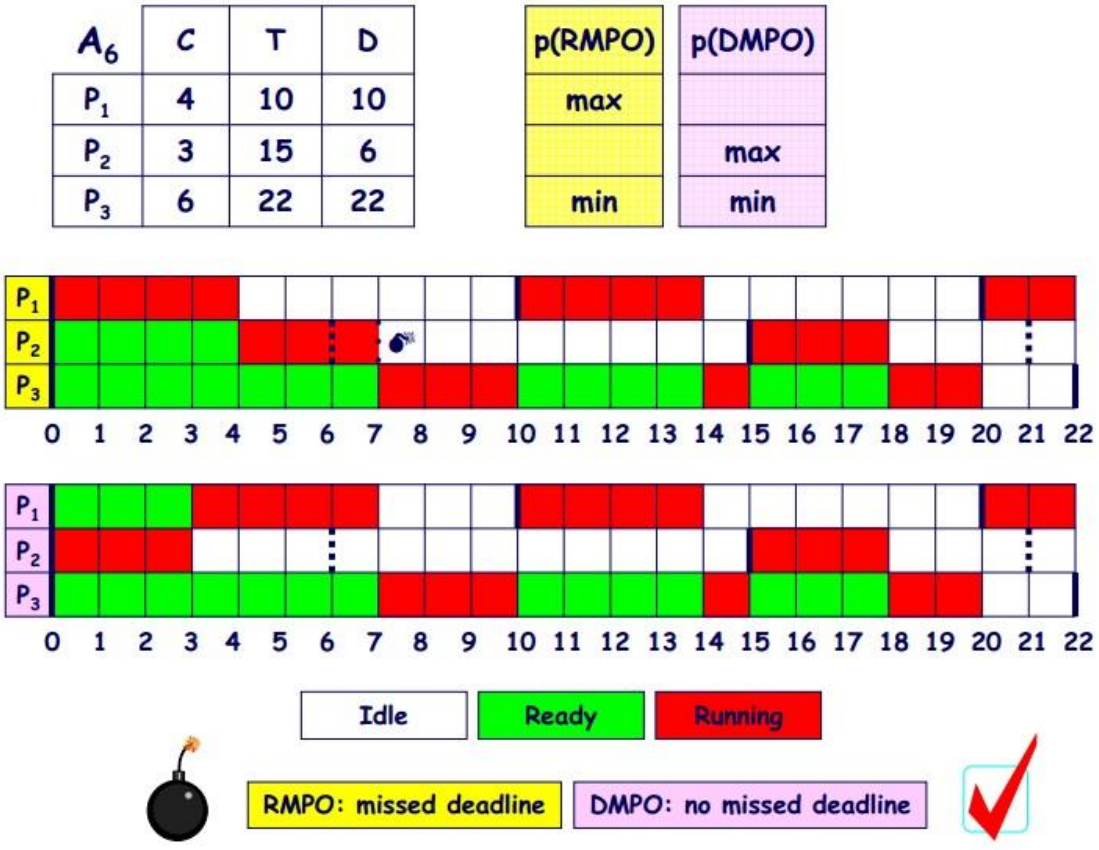

Figura 4.4 Confronto tra RMPO e DMPO [9]

L’ultimo dei tre algoritmi di schedulazione implementati, EDF, è anche l’unico ad attribuzione dinamica di priorità. Ad ogni task viene associata dinamicamente una priorità tanto maggiore quanto più è imminente la corrispondente deadline assoluta. Nello specifico si tratta un algoritmo con livelli di priorità dinamica a livello di task e priorità fissa a livello di job, in quanto l’assegnazione delle priorità avviene solo al rilascio di ogni nuovo job. Le priorità restano quindi costanti durante l’esecuzione fino a quando non “scade” il periodo di un task e viene rilasciato un nuovo job.

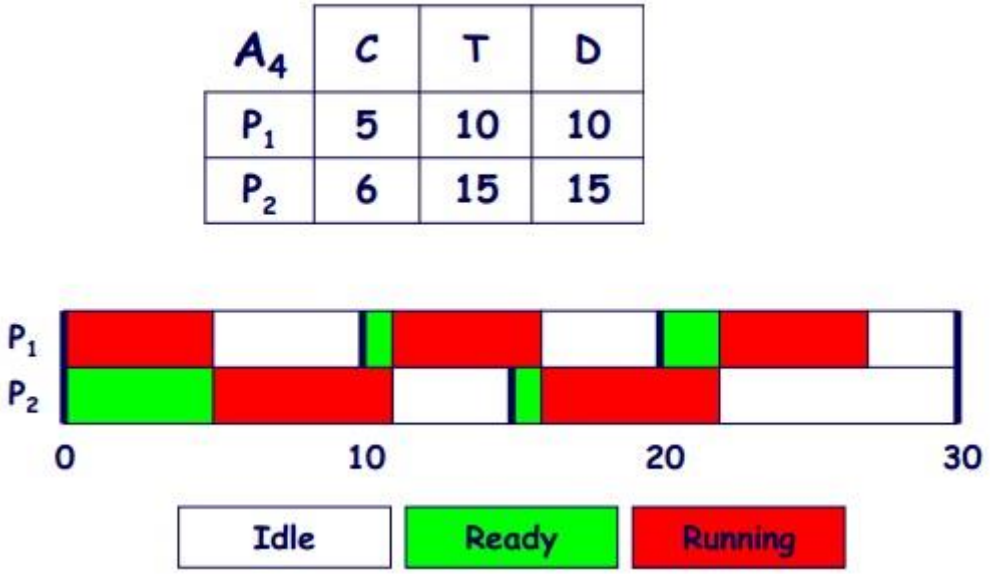

Figura 4.5 Esempio di schedulazione tramite EDF [9]

Un altro fattore di importanza rilevante è la politica per la gestione dei sovraccarichi. Finora sono stati presentati come esempi casi in cui la schedulazione con l’algoritmo giusto era sempre fattibile (riguardo alla fattibilità della schedulazione v. cap. 5) e quindi non esistevano casi in cui un job non riuscisse a completare la propria esecuzione prima del rilascio del job successivo. Nel caso in cui questo invece avvenga si parla di job overrun. Esistono due politiche principali per la gestione di questi casi:

- *SKIP_NEXT_EXECUTION_POLICY*;
- *AS_SOON_AS_POSSIBLE_EXECUTION_POLICY*.

Entrambe hanno una loro implementazione all'interno del meta-scheduler.

La prima politica è sicuramente la più semplice, in quanto non lascia adito ad interpretazioni sbagliate. Semplicemente in caso di job overrun, il task "ritardatario" prosegue nella sua esecuzione fino al completamento e durante la sua esecuzione non vengono avviati altri job. Una volta completato il job che ha causato il sovraccarico si aspetta fino alla prossima release (determinata dal periodo del task) per avviare un nuovo job. In questo modo si riduce il lavoro destinato al processore nel tentativo di eliminare le cause che hanno portato al sovraccarico.

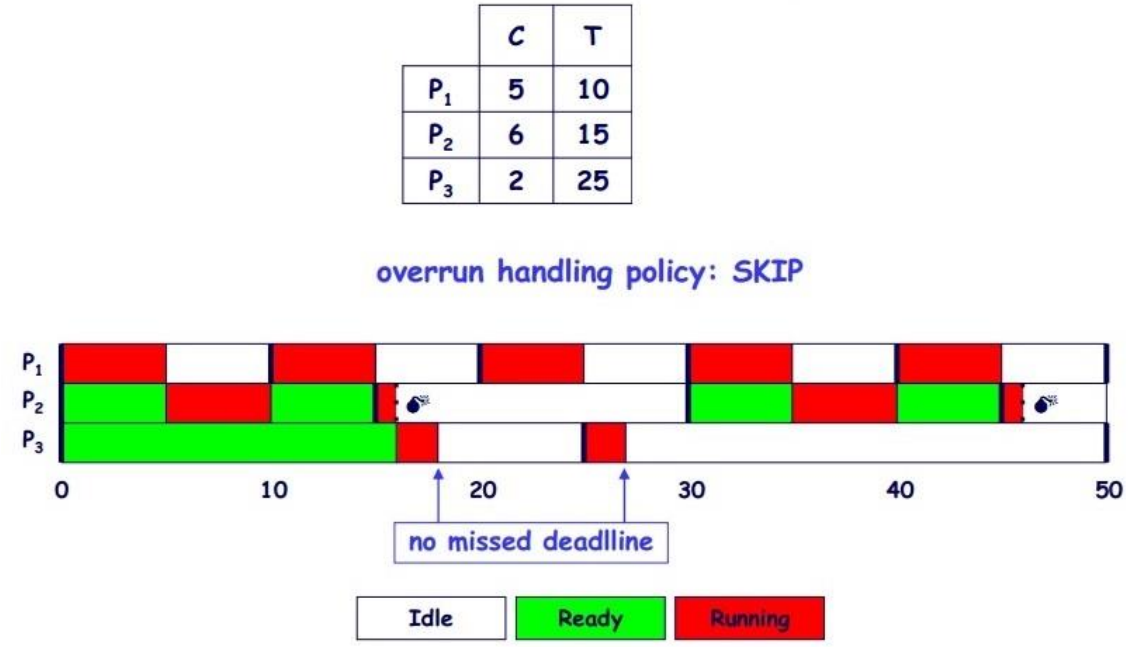

Figura 4.6 Esempio di job overrun con politica di schedulazione RMPO e gestione dei sovraccarichi tramite SKIP NEXT EXECUTION POLICY [9]

La politica ***ASAP*** (**A**s **S**oon **A**s **P**ossible) prevede in caso di sovraccarico che non appena il task ritardatario abbia completato il suo job, il job successivo venga immediatamente avviato. Per quanto possa apparire semplice l'implementazione di questa politica non è ovvia come sembra, soprattutto nel caso in cui avvengano più job overrun. Il caso più semplice prevede semplicemente di sbloccare l'esecuzione del task allo scadere di

ogni periodo. Utilizzando un semaforo questo avviene semplicemente richiamando la *sem_post()* sul semaforo del task interessato. Tuttavia questo approccio non tiene in considerazione la funzione tipica dei processi periodici. Solitamente infatti ai processi periodici è assegnato il controllo di alcuni parametri specifici dell'impianto o macchinario "sorvegliato" (una centrale nucleare, gli alettoni di un aereo, ecc.) dal sistema real-time. Queste operazioni vengono svolte da processi periodici proprio in virtù del fatto che i controlli devono essere effettuati a cadenze regolari. Una verifica con una frequenza maggiore, a seguito di un sovraccarico non solo risulta inutile, ma potrebbe addirittura essere controproducente inducendo altri sovraccarichi nel sistema. Tuttavia una situazione di job overrun indica un ritardo che, oltre a dover essere segnalato, il sistema deve cercare di recuperare.

	C	T
P_1	5	10
P_2	6	15
P_3	2	25

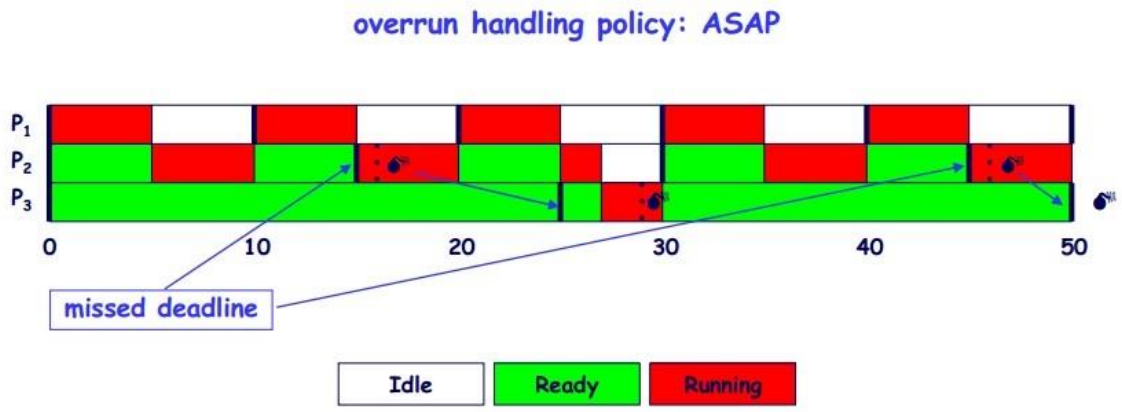

Figura 4.7 Job overrun con politica di schedulazione RMPO e gestione sovraccarichi tramite ASAP [9]

Durante lo svolgimento del progetto sono state ipotizzate due metodologie per risolvere il problema. Entrambe le metodologie prevedono il lancio del nuovo job appena il job in ritardo sia terminato.

Con il primo metodo una volta avviato il nuovo job si provvederebbe ad aggiornare il successivo tempo di rilascio a partire dall'istante effettivo di avvio del job successivo al job ritardatario. In questo modo l'intero task subirebbe uno sfasamento temporale in base al ritardo accumulato. Questo metodo riduce il rischio di un'accumulazione di job overrun successivi, ma comporta il riaggiornamento di tutti i parametri temporali (release e deadline). Procedendo in tal modo viene anche evitato il rilascio di job per recuperare quelli perduti durante il job overrun.

Con il metodo alternativo, al rilascio del nuovo job, per evitare lo sfasamento temporale del task, i valori della deadline e della nuova release vengono calcolati a partire dall'ultimo release previsto. Nel caso in cui il job abbia causato un solo job overrun, si tratta dell'istante determinato dalla somma tra il tempo di rilascio del job ritardatario e il periodo. Nel caso in cui il job abbia saltato più volte la release i parametri temporali verranno calcolati di conseguenza. Nell'implementazione del simulatore si è deciso di utilizzare questa seconda politica.

4.2.3 Meta-scheduler

Il meta-scheduler può essere considerato il cuore del simulatore. Si occupa infatti lui della gestione di tutti i task periodici. Ad esso sono delegate le operazioni di rilascio dei task, di verifica delle deadline ed eventualmente di applicazione delle politiche di schedulazione e/o di gestione dei sovraccarichi. Ad esclusione dei thread di gestione dei server, che saranno descritti nel paragrafo 4.3, è l'unica entità che durante la simulazione ha nozione dello scorrere del tempo.

Prima dell'inizio della simulazione il meta-scheduler deve svolgere diverse operazioni per poter svolgere il proprio lavoro.

La prima operazione consiste nel creare un canale per la ricezione dei messaggi e collegarvisi tramite una connessione. Questa servirà a inviare

i messaggi tramite i timer che si occupano di scandire lo scorrere del tempo.

Lo stato di tutti i task periodici viene modificato in *IDLE*, si imposta il job counter ad 1 e si imposta il release del primo job all'istante di inizio della simulazione (*Start_Time_ns*). Vengono inoltre allocate alcune strutture dati per effettuare il log della simulazione. La parte principale risulta la creazione dei timer. Per ogni task viene creato un timer periodico con intervallo di scadenza pari al periodo del task corrispondente. Alla scadenza di ogni periodo il timer invierà un pulse di tipo *RELEASE_JOB_PULSE_CODE* sul canale del meta-scheduler con valore pari all'identificatore del task relativo. Se la deadline e il periodo del task non coincidono (D_i != T_i) viene creato un altro timer, questa volta assoluto, con scadenza pari alla somma del tempo di avvio e la deadline relativa. Nel caso in cui deadline e periodo siano uguali non è necessario il secondo timer perché allo scadere del timer del release, se il task risulterà essere ancora in esecuzione (*RUNNING*) ovviamente avrà sforato la propria deadline. Da notare che in questo caso il non rispetto della deadline equivale anche ad un sovraccarico con conseguente job overrun. I timer utilizzati per le deadline sono assoluti non periodici per due motivi: se il task termina l'esecuzione entro la deadline il timer può essere disabilitato in modo da non generare un evento che risulterebbe inutile e in caso di job overrun è possibile che non sia necessario avviare il timer per la deadline in quanto questa potrebbe già essere stata mancata.

Al termine di queste operazioni preliminari il meta-scheduler si sospende in attesa dell'inizio della simulazione.

Allo sbloccarsi della barriera, lo stato di tutti i task periodici viene impostato in RUNNING. Dopodiché, il thread si sospende ciclicamente in attesa di un messaggio sul canale precedentemente creato. Il meta-scheduler al risveglio deve controllare lo stato del sistema per decidere

che operazioni svolgere, ma grazie all'uso dei pulse, invece di dover ogni volta verificare la situazione corrente (come avverrebbe se si risvegliasse costantemente allo scadere di un certo periodo), può agire essendo già a conoscenza delle operazione che deve svolgere. I pulse ricevuti infatti contengono tutte le informazioni necessarie. Ogni pulse identifica tramite il campo *value* il task periodico a cui si riferisce e tramite il campo *code* l'evento che ha causato il risveglio del meta-scheduler. Gli eventi possibili sono i seguenti:

- *RELEASE_JOB_PULSE_CODE*, inviato dal timer che indica il termine del periodo di ogni task;
- *DEADLINE_JOB_PULSE_CODE*, inviato dal timer relativo al task nel caso in cui la deadline sia scaduta e il task corrispondente sia ancora in esecuzione;
- *END_JOB_PULSE_CODE*, inviato dal task periodico al termine dell'esecuzione di ogni job;
- *END_OF_SIMULATION_PULSE_CODE*, inviato dal main al termine della simulazione, causa l'uscita del meta-scheduler dal ciclo di attesa dei messaggi.

La combinazione del codice ricevuto e dello stato del task corrispondente permette di discernere la causa del risveglio e di effettuare il minimo di operazioni necessarie. La figura 4.2 rappresenta le modifiche effettuate dal meta-scheduler allo stato dei task periodici in seguito alle sue operazioni. Si ricorda che i task periodici non effettuano nessuna operazione sul proprio stato e tutte le modifiche sono effettuate esclusivamente dal meta-scheduler. Alcune combinazioni di stato del task e messaggio ricevuto dal thread non sono possibili, in quanto la programmazione non permette il loro verificarsi.

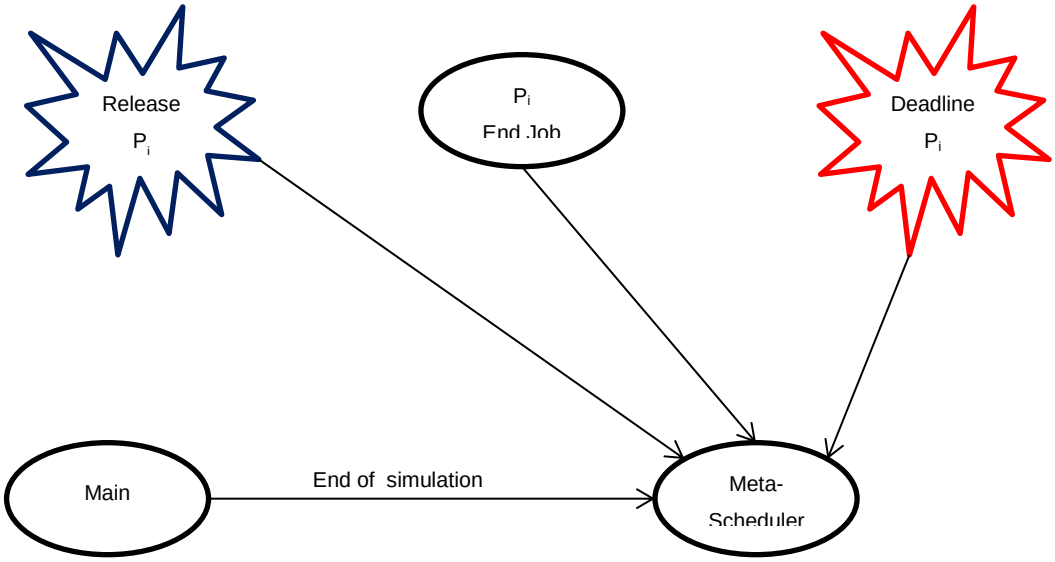

Figura 4.8 Schema dei messaggi ricevuti dal meta-scheduler. Si differenziano tra quelli generati da timer o da altri thread

Ad un task in stato *RUNNING* possono essere associati tutti i diversi pulse. Nel caso di fine di un job, il meta-scheduler controlla la durata del job appena terminato e nel caso sia maggiore di ogni altra precedente la salva (per il sistema di log). Dopodiché aggiorna il job counter. Se il task ha così raggiunto il numero di job previsti si procede alla distruzione dei timer associatigli (release e deadline) e il task viene posto nello stato *END_OF_EXECUTION*, in caso contrario lo stato del task viene modificato in *IDLE* e viene eliminato solo il timer relativo alla prossima deadline. In caso di ricezione di un messaggio di tipo *DEADLINE_PULSE_CODE* lo stato del corrispondente task viene modificato in *MISSED_DEADLINE* e viene aggiornato il contatore relativo al numero di deadline mancate dal task interessato. Infine la ricezione di un pulse di tipo *RELEASE_JOB_PULSE_CODE* comporta il passaggio del task nello stato *JOB_OVERRUN* e, oltre al contatore relativo ai job overrun, viene aggiornato anche il contatore relativo alle deadline mancate, poiché il verificarsi di questa combinazione (stato *RUNNING* e

pulse *RELEASE_JOB_PULSE_CODE*) implica che il task abbia deadline e periodo equivalenti ($D_i = T_i$).

L'unico messaggio associabile ad un task con stato *IDLE* è quello relativo ad una nuova release, dato che l'eventuale timer associato alla deadline è stato disattivato e il task corrispondente è sospeso. Alla ricezione del messaggio, il meta-scheduler avvia un nuovo timer relativo alla deadline (solo se D_i != T_i), modifica lo stato del task in *RUNNING* e sblocca il task tramite una *sem_post()* sul semaforo corrispondente.

Gli ultimi due stati in cui il task può trovarsi alla ricezione di un messaggio indicano che durante l'esecuzione si è verificato un ritardo più o meno grave.

Nel caso di un task che non ha rispettato la sua deadline (*MISSED_DEADLINE*) il meta-scheduler può ricevere solo messaggi di due tipi, *END_JOB_PULSE_CODE* o *RELEASE_JOB_PULSE_CODE*. Il primo caso è equivalente al caso del task in stato *RUNNING*, con l'esclusione della distruzione del timer relativo alla deadline (già scattato), mentre in caso di una nuova release il meta-scheduler registra il sovraccarico sul contatore corrispondente e modifica lo stato in *JOB_OVERRUN*.

In caso di sovraccarico (task in stato *JOB_OVERRUN*) possono verificarsi solo due eventi. Il task non riesce a completare l'esecuzione entro il periodo successivo, nel qual caso viene semplicemente aggiornato il contatore dei job overrun, oppure il task termina la sua esecuzione. Appena questo accade, come negli altri casi di terminazione dell'esecuzione di un job, viene aggiornato il contatore dei job e si controlla se il job terminato fosse l'ultimo previsto. In questo caso il task viene posto in stato *END_OF_EXECUTION* e il meta-scheduler si sospende nuovamente in attesa di un altro pulse. Se il task deve proseguire l'esecuzione con altri job è necessario applicare la politica di gestione dei sovraccarichi scelta per la simulazione corrente. In caso di

politica *SKIP*, il task viene semplicemente messo in stato *IDLE*, in attesa dello scadere del prossimo periodo. L'applicazione della politica *ASAP* risulta invece più complicata. Il task deve ripartire subito con un nuovo job, ma ci sono due possibilità. Queste dipendono dalla deadline relativa al nuovo job. Come descritto precedentemente nella descrizione delle politiche di gestione del sovraccarico, al termine del job ritardatario viene calcolato il tempo di rilascio atteso per il nuovo job. Questo equivale alla differenza tra il tempo relativo alla prossima scadenza del timer di rilascio di un nuovo job per il task corrente e il periodo del task. Da questo si calcola la deadline per il job corrente. Una volta fatto ciò, si verifica se il tempo di rilascio del job corrente è superiore alla nuova deadline. Se questo è vero il nuovo job ha già sforato la sua deadline, quindi si procede ad aggiornare il contatore e a modificare lo stato del task in *MISSED_DEADLINE*. In caso contrario lo stato del task diventa *RUNNING* e viene avviato il timer corrispondente alla nuova deadline. In entrambi i casi il task periodico viene sbloccato.

Al momento di ogni release, che sia o meno in seguito ad un sovraccarico, nel caso la politica di schedulazione scelta sia *EDF*, viene avviato anche il meta-scheduler EDF (*Meta_Scheduler_EDF()*).

Quest'ultimo si occupa di riassegnare le priorità ai diversi task in base all'imminenza delle corrispondenti deadline. Il meta-scheduler EDF crea un vettore contente gli indici dei diversi task periodici. Poi, tramite bubble sort, lo ordina sulla base delle deadline più imminenti. Una volta che gli indici sono stati ordinati, il meta-scheduler EDF scorre il vettore e, se diversa dalla priorità corrente, aggiorna la priorità di schedulazione di ogni task periodico.

Una volta che la simulazione è terminata il meta-scheduler viene sbloccato dal main (*END_OF_SIMULATION_PULSE_CODE*) e, dopo

aver chiuso la connessione e il canale utilizzati per la ricezione e l'invio dei pulse, termina la propria esecuzione.

Nel meta-scheduler sviluppato, trattandosi di un simulatore, in caso di deadline mancate o di sovraccarichi, non viene lanciato alcun allarme ma viene semplicemente registrato l'avvenimento. Al contrario, in un sistema real-time operativo è possibile che in questi casi sia necessario adottare delle contromisure dipendenti dal sistema specifico. È il meta-scheduler che in casi come questo si deve occupare della segnalazione di questi inconvenienti al sistema incaricato di gestirli.

4.3 Gestione delle richieste aperiodiche

Le richieste aperiodiche, solitamente associate a comandi utente, causano l'avvio di processi. Perché il sistema real-time funzioni correttamente è necessario decidere una strategia di schedulazione che tenga conto di queste richieste che possono avere a loro volta una deadline. Nel caso del simulatore sviluppato, le richieste aperiodiche sono caratterizzate dai seguenti parametri:

- nome, utilizzato per l'identificazione;
- tempo di arrivo, indicato con **a**;
- tempo di servizio, indicato con **s**.

Come nel caso dei processi periodici, anche qui i tempi specificati nei parametri vengono espressi in millisecondi. In un ambiente reale le richieste aperiodiche possono arrivare in qualsiasi istante, ma trattandosi di una simulazione è necessario specificare il tempo di arrivo, a partire dall'inizio della simulazione, e il tempo di servizio. Quest'ultimo in un caso reale, dipende dal tipo di richiesta ricevuta dal sistema real-time.
Il simulatore oltre ai parametri caratterizzanti le richieste aperiodiche tiene traccia per ognuna dei tempi equivalenti in nanosecondi e, per verificare il funzionamento delle politiche di schedulazione implementate, del tempo

di inizio del servizio della richiesta e del tempo di completamento. Viene inoltre calcolato il ritardo con cui la richiesta è completata.

4.3.1 Generatore delle richieste aperiodiche

Non potendo ricevere le richieste dall'esterno non avendo a che fare con un sistema reale, è necessario delegare un thread alla generazione delle richieste aperiodiche.

Nel caso venga specificata una politica di gestione delle richieste aperiodiche, questo è il primo thread a venire creato dal main. La sua priorità (*HIGHEST_PRIORITY*, 253) è la massima del simulatore ad esclusione del meta-scheduler.

All'avvio il thread segnala tramite una variabile booleana che è in esecuzione, imposta il contatore delle richieste a 0 e una variabile temporale (*Aperiodic_Request_Relative_Arrival_Time_ns*) a 0. Poi si sospende in attesa dell'inizio della simulazione. Questo è uno dei due, tre o quattro processi necessari alla gestione delle richieste aperiodiche di cui il main tiene conto alla creazione della barriera.

Figura 4.9 Il processo P_g genera le richieste scrivendo in una FIFO, il processo P_e simula l'esecuzione della richiesta impegnando la CPU per il tempo necessario [10]

Una volta sbloccato, il thread entra all'interno di un ciclo in cui ottiene i dati relativi alla prossima richiesta aperiodica da generare. Fatto ciò, si sospende tramite una sleep fino al tempo di arrivo della richiesta. Al risveglio registra il tempo di arrivo della richiesta e inserisce quest'ultima in una pipe (gestione a FIFO), precedentemente creata dal main. Inoltre, se la gestione avviene tramite deferrable server, invia un pulse di tipo *SERVICE_REQUEST_PULSE* a uno dei thread di gestione del server.

Una volta terminate le richieste aperiodiche previste, il thread, dopo aver modificato la variabile booleana associata alla sua esecuzione, termina.

Prima della descrizione dell'implementazione dei diversi servizi implementati verranno descritte le diverse politiche di schedulazione utilizzate.

4.3.2 Politiche di schedulazione per processi aperiodici

L'algoritmo di schedulazione più semplice per la gestione delle richieste aperiodiche è il servizio in background. Utilizzabile nel caso le richieste aperiodiche abbiano vincoli di tipo soft o non real-time, implica che le richieste vengano servite solo nel caso in cui non vi siano processi hard real-time pronti per l'esecuzione. Si vengono così a creare due code a priorità diversa. Quella a priorità maggiore riservata ai task periodici o sporadici, mentre la coda a bassa priorità è utilizzata per le richieste aperiodiche. Un task periodico o sporadico causa la preemption dei task aperiodici.

Figura 4.10 Code a priorità diversa [10]

Le strategie di schedulazione per le due code risultano indipendenti. Per la coda dei processi periodici vengono solitamente utilizzati gli algoritmi RMPO, DMPO o EDF, mentre per la coda dei processi periodici si utilizza tipicamente una strategia di tipo ***FCFS*** (**F**irst **C**ome **F**irst **S**erved).

Questa strategia di gestione garantisce l'eventuale schedulabilità dei processi periodici, ma può causare tempi di risposta molto lunghi per le richieste aperiodiche.

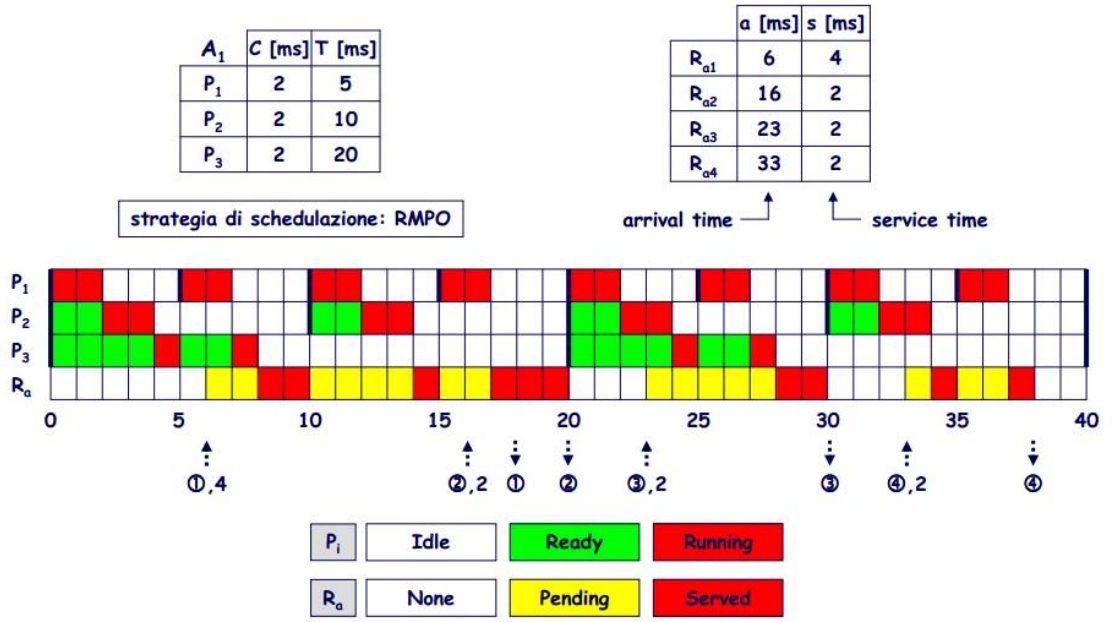

Figura 4.11 Esempio di schedulazione utilizzando per i processi periodici la strategia RMPO e il servizio in background per le richieste aperiodiche [10]

Gli altri due servizi implementati, al contrario del precedente, si focalizzano sul garantire tempi di risposta migliori alle richieste aperiodiche. Sono entrambi definiti servizi tramite server a priorità statica.

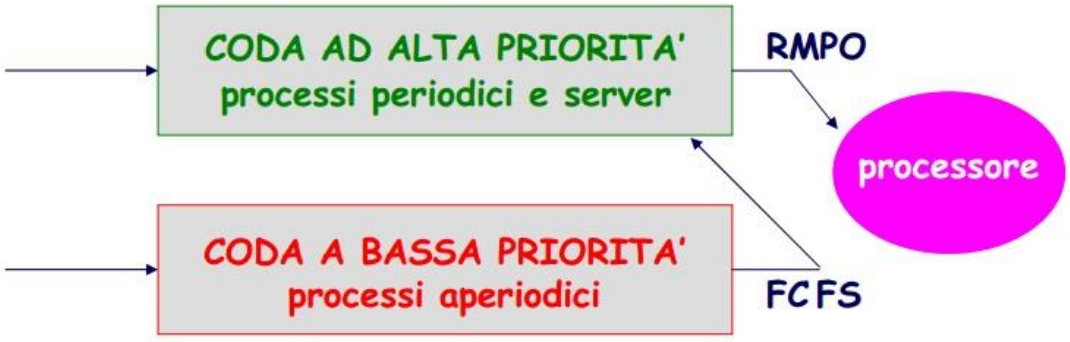

Figura 4.12 Con un servizio a server a priorità statica le richieste aperiodiche restano in una coda a bassa priorità, ma vengono servite da un processo server servito come un precesso periodico [10]

Il primo dei due servizi implementati è il polling server. Il server è un processo periodico, caratterizzato da un periodo **T_s** e una capacità (tempo massimo di esecuzione) **C_s** prefissati. Il processo server (**P_s**) viene considerato a tutti gli effetti un processo periodico e viene schedulato secondo la stessa strategia (tipicamente RMPO) in base alla priorità $p(P_s) = 1 / T_s$, di norma elevata, che gli compete.

Nel momento in cui viene schedulato il processo server, se esistono richieste periodiche in attesa di essere servite, il processo server comincia a servirle consumando la propria capacità (se la capacità scende a zero, il server si sospende). Al termine del periodo T_s la capacità viene ripristinata. Se in un qualsiasi momento il server è in esecuzione e non sono presenti richieste aperiodiche in coda, la capacità residua viene scartata.

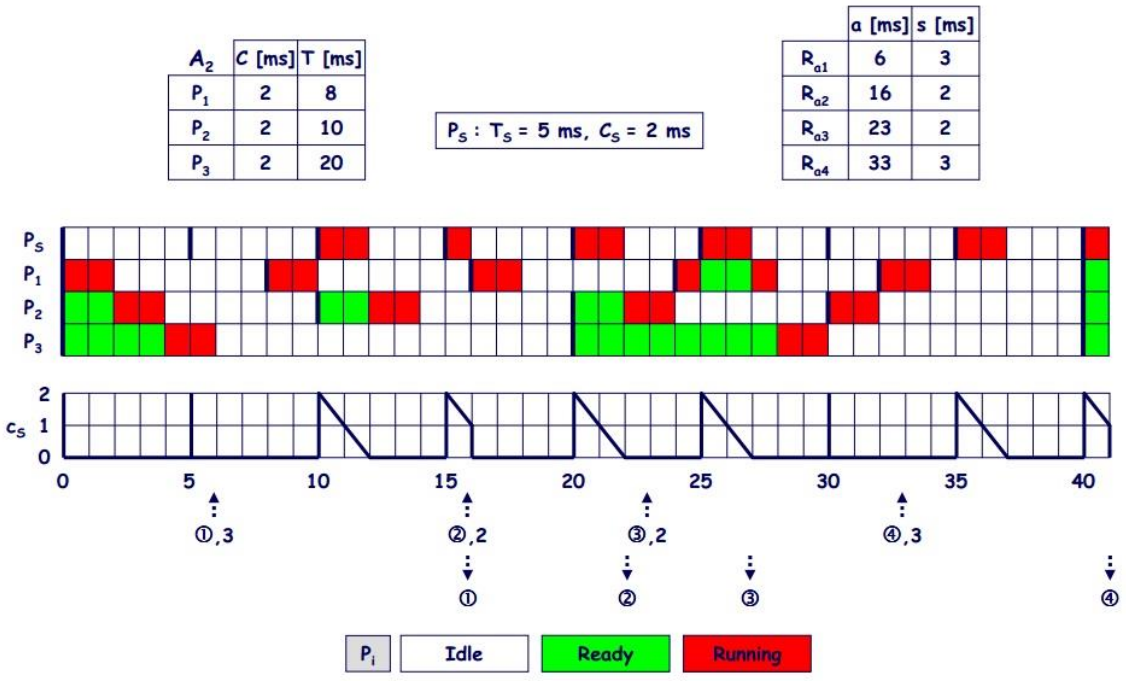

Figura 4.13 Esempio di schedulazione tramite Polling Server [10]

Il terzo servizio, sempre tramite server a priorità statica, è il deferrable server. Come il polling server viene schedulato insieme ai processi periodici sulla base del rapporto $1 / T_s$ e dispone anche lui di una certa capacità C_s. La differenza con il polling server consiste proprio nella

gestione della capacità. Al termine del periodo T_s la capacità viene ripristinata sempre al valore C_s, ma diversamente dal polling server, la capacità non viene esaurita in caso di assenza di richieste aperiodiche, ma viene conservata. In questo modo è possibile differire il servizio di gestione. Durante l'esecuzione di una richiesta aperiodica la capacità viene consumata normalmente.

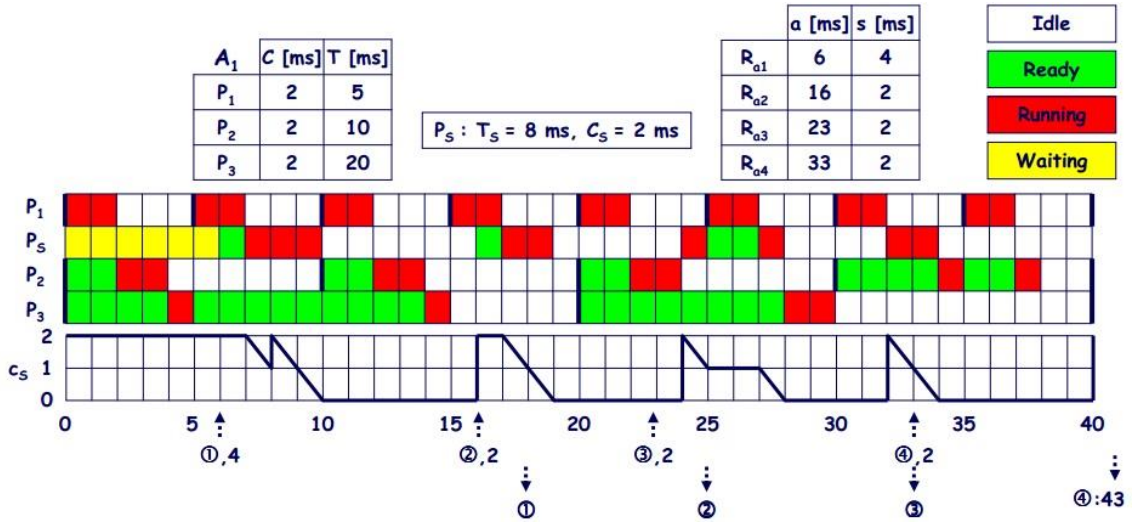

Figura 4.14 Esempio di schedulazione tramite deferrable server [10]

I servizi tramite server a priorità statica tendono ad avere tempi di risposta migliori rispetto al servizio in background, ma vanno a inficiare la schedulabità dei processi periodici. È quindi importante tenerne conto in fase di progettazione del sistema e calibrare il periodo e la capacità del server in maniera accurata.

Dopo questa descrizione teorica delle politiche di gestione delle richieste aperiodiche, verranno ora descritte le loro implementazioni all'interno del simulatore.

4.3.3 Servizio in background

Il servizio in background è il più semplice dei tre implementati. Consiste in un solo thread la cui priorità è al limite basso delle priorità utilizzate per la simulazione (*BACKGROUND_SERVER_PRIORITY*, 251). Il thread

semplicemente verifica che la pipe assegnata alle richieste aperiodiche sia vuota. Quando questo non si verifica, allora esegue una *Busy_Wait()* con durata pari al tempo di servizio richiesto dalla richiesta aperiodica. Prima e dopo l'inizio della busy wait vengono registrati il tempo di inizio del servizio e il tempo di fine. Il server continua questo ciclo finché il generatore di richieste aperiodiche è in esecuzione. Quando questo non è più vero, il server controlla un'ultima volta la pipe e serve l'eventuale ultima richiesta. Infine prima di terminare il thread modifica lo stato del server, segnalando che non è più in esecuzione a beneficio del thread per il controllo del termine della simulazione.

L'implementazione dei server a priorità statica richiede una struttura dati più complessa rispetto al servizio in background. Nella struttura dati che identifica il server, oltre a nome, priorità di esecuzione, l'identificatore del thread del server, le variabili booleane *Running* e *End_Of_Execution* utilizzati dal servizio in background, sono necessarie le variabili a cui associare il periodo e la capacità (in millisecondi e nanosecondi) e il semaforo su cui il thread del server si sospende quando ha esaurito la capacità.
I server nel caso del simulatore agiscono sempre a priorità massima indipendentemente dal periodo e dalla capacità per l'impossibilità di tenere conto del consumo della capacità in caso di preemption del server.
Dato che in QNX non esistono primitive che permettono di sospendere un thread dall'esterno per gestire i server come sono descritti nella letteratura, nel caso la politica di servizio delle richieste aperiodiche scelga di utilizzare un server a priorità statica, viene creato un processo a priorità subito superiore alla minima (252). Questo thread, nel caso il processore non sia utilizzato dai task periodici, impedisce al server di sfruttarlo nel caso che la sua capacità sia azzerata. Per fare ciò il thread esegue in continuazione una busy wait di tempo prefissato. Al termine

dell'esecuzione del server anche questo thread termina la sua esecuzione.

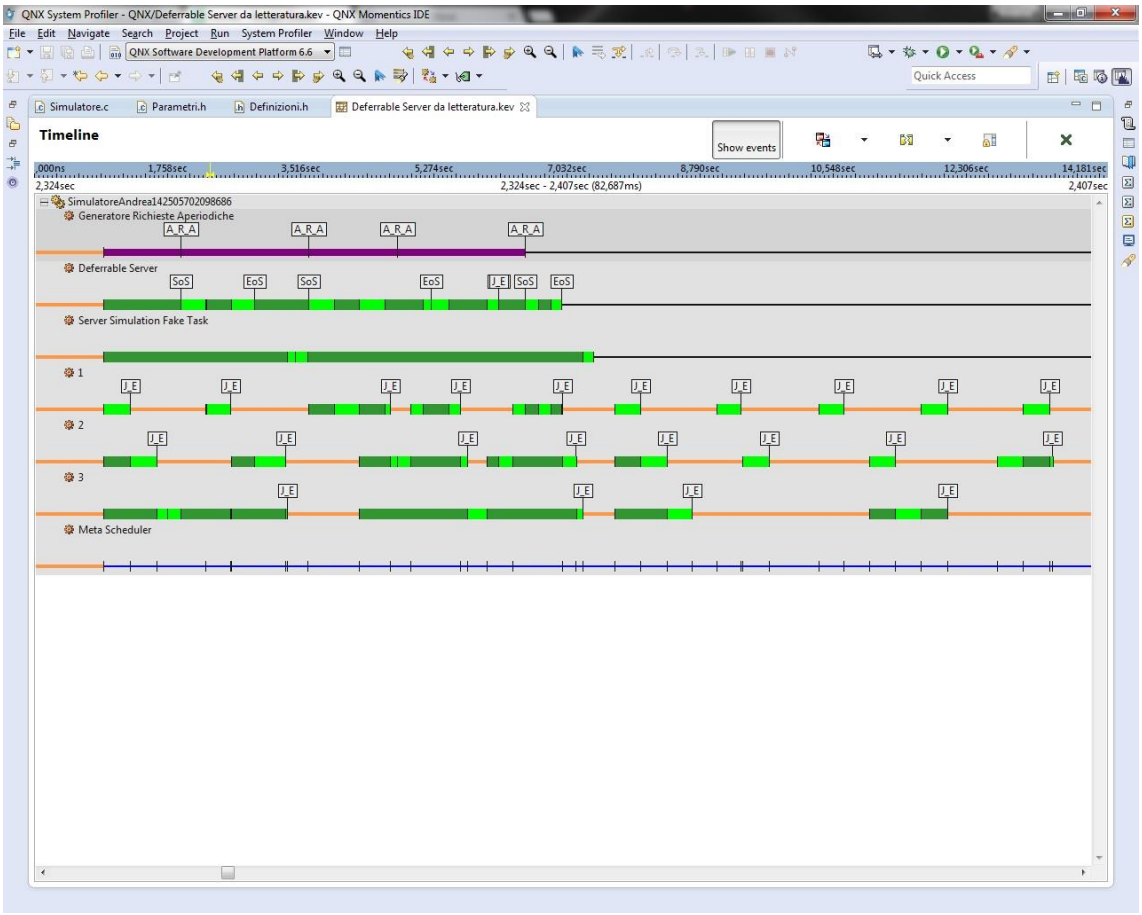

Figura 4.15 Esempio di deferrable server con fake task attivo. Nel caso questo fosse disattivo il tempo di esecuzione di quest'ultimo verrebbe invece dedicato al servizio delle richieste aperiodiche senza influire in alcun modo sulla schedulazione dei processi periodici

Per rendere più versatile il simulatore e aumentarne le possibilità di simulazione, la creazione di questo thread è subordinata ad una variabile booleana dei parametri. Nel caso si decida di non crearlo e il processore non debba servire task periodici e il server sia a capacità azzerata e siano presenti delle richieste aperiodiche ancora da servire, il processore porterà avanti il lavoro del server fino a quando un processo periodico non ne richieda l'utilizzo. In questo modo le richieste aperiodiche possono essere completate con tempi leggermente inferiori rispetto al caso di utilizzo del server "ufficiale".

4.3.4 Polling server

La gestione tramite polling server avviene attraverso 2 thread, uno dedicato al servizio delle richieste e l'altro incaricato della gestione della capacità. Il thread di controllo agisce al livello di priorità subito superiore rispetto al simulatore del servizio.

Il server vero e proprio è molto simile al background server, ma nel caso il server sia attivo e non ci siano richieste pendenti, invece di continuare a controllare la pipe, il server si sospende sul proprio semaforo. Questo avviene anche al termine delle operazioni di servizio di una richiesta, cioè al termine della *Busy_Wait()* corrispondente. Il semaforo può essere sbloccato solo dal thread di controllo. Il server termina la propria esecuzione nel caso in cui non ci siano richieste pendenti e il thread generatore di richieste sia terminato.

Il secondo thread si occupa della gestione della capacità e del suo ripristino al termine di ogni periodo. Per fare ciò, prima dell'inizio della simulazione (cioè prima di sospendersi sulla barriera), crea un canale a cui collega una connessione. In questo modo può rilevare lo scorrere del tempo tramite un timer, come il meta-scheduler. Allo scadere il timer, periodico, invia un pulse di tipo *CAPACITY_REPLANISHMENT_PULSE_CODE* tramite la connessione.

Durante la simulazione il thread si sospende in attesa di un messaggio. Allo scadere del periodo T_s, il thread va subito in esecuzione (salvo il caso in cui il meta-scheduler debba agire) grazie al livello di priorità elevato e innalza il livello di priorità del thread server alla sua priorità nominale, dopodiché si sospende tramite una sleep con una durata pari alla capacità C_s. In questo modo se ci sono richieste pendenti il server è in grado di agire. Al termine della sleep la priorità del thread server viene ridotta al minimo delle priorità utilizzate per la simulazione (*BACKGROUND_SERVER_PRIORITY*). Questa attesa ciclica prosegue fino a quando il server non termina la propria esecuzione, cioè dopo che il

generatore di richieste ha terminato la sua esecuzione e la pipe è vuota. Prima di terminare il thread di controllo distrugge il timer usato per notificare il trascorrere del periodo, la connessione e il canale corrispondente.

4.3.5 Deferrable server

Il deferrable server è il più complesso dei tre metodi implementati. Per utilizzare questo servizio sono necessari tre thread. È inoltre l'unico dei tre server che necessità di uno stato interno per la gestione della capacità. I possibili stati in cui il server può trovarsi sono:

- *SERVER_STARTING*, stato di partenza e indicante l'assenza di capacità e l'assenza di richieste in coda;
- *SERVER_WAITING*, stato in cui il server dispone di capacità, ma non sono presenti richieste da servire (il server attende una richiesta e mantiene la capacità);
- *SERVER_RUNNING*, il server è in esecuzione e consuma la sua capacità per servire una richiesta aperiodica;
- *SERVER_IDLE*, il server ha delle richieste da servire, ma ha esaurito la propria capacità ed è in attesa del ripristino.

Per coordinarsi i thread utilizzano anche in questo caso i pulse. Oltre ai tre thread specifici utilizzati per l'implementazione del deferrable server, al coordinamento partecipa anche il generatore di richieste aperiodiche, generando un messaggio specifico all'arrivo di una richiesta aperiodica.
Il server può ricevere quattro tipi di messaggio:

- *SERVICE_REQUEST_PULSE_CODE*, inviato dal thread generatore delle richieste aperiodiche all'arrivo di una nuova richiesta;
- *END_OF_SERVICE_PULSE_CODE*, inviato dal thread che serve le richieste al termine del servizio di una richiesta se non sono presenti ulteriori richieste in coda;

- *CAPACITY_REPLENISHMENT_PULSE_CODE*, inviato dal thread per il ripristino della la capacità allo scadere di ogni periodo;
- *CAPACITY_EXHAUSTED_PULSE_CODE*, questo messaggio non viene inviato mai, ma serve a indicare l'esaurirsi della capacità.

I tre thread dedicati al deferrable server sono, in ordine di priorità di schedulazione:

- *Deferrable_Server_Control_Task*;
- *Deferrable_Server_Capacity_Check_Task*;
- *Deferrable_Server_Task*, la priorità di quest'ultimo varia da quella nominale a quella minima (*BACKGROUND_SERVER_PRIORITY*);

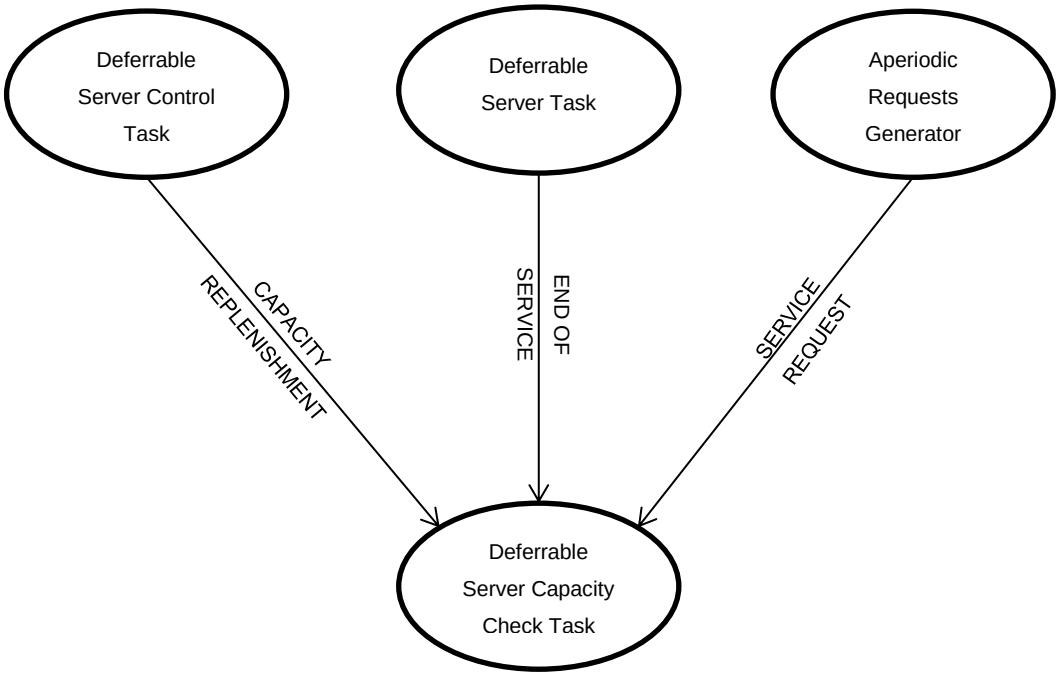

Figura 4.16 Schema dei thread e dei messaggi inviati con gestione delle richieste aperiodiche tramite deferrable server

Il thread dedicato al server vero e proprio è estremamente semplice. Una volta iniziata la simulazione entra in un ciclo, in cui riduce la propria priorità (*BACKGROUND_SERVER_PRIORITY*). All'interno del primo

ciclo ne è presente un altro che verifica se sono presenti richieste aperiodiche pendenti sulla pipe. Se ciò è vero, il server comincia a servirle tramite busy wait e tenendo traccia del tempo di inizio del servizio e del tempo di fine del servizio. Al termine dell'esecuzione della busy wait, se non sono presenti altre richieste sulla pipe, invia un messaggio sul canale dedicato alla gestione del deferrable server indicando il completamento del servizio (*END_OF_SERVICE_PULSE_CODE*), altrimenti estrae la nuova richiesta dalla pipe ed inizia a servirla. La sua esecuzione prosegue fino a quando tutte le richieste aperiodiche previste sono state servite. Prima della sua terminazione il thread indica ai due thread di controllo di terminare, tramite la modifica di una variabile di tipo booleano.

Al processo di controllo (*Deferrable_Server_Control_Task*) sono assegnati due compiti. Il primo consiste nel creare le strutture dati necessarie alla comunicazione tra i quattro thread di gestione delle richieste aperiodiche. Questo avviene prima dell'inizio della simulazione. L'altro compito consiste nel segnalare al thread di controllo della capacità, la scadenza dei periodi per permettere il ripristino. Per fare ciò, prima dell'inizio della simulazione, viene creato un timer periodico (a cui sono associati un altro canale e un'altra connessione), alla cui scadenza viene lanciato un pulse di tipo *CAPACITY_REPLENISHMENT_PULSE_CODE*.

Durante la simulazione il thread si sospende in attesa dello scadere del timer. Alla ricezione del messaggio previsto, il thread invia un altro messaggio sul canale dedicato alla gestione del deferrable server per ripristinare la capacità del server al valore C_s.

Una volta terminata la simulazione il thread elimina le risorse allocate, un timer, due connessioni e due canali, e termina.

Il più complesso dei tre thread dedicati al deferrable server è indubbiamente quello dedicato alla gestione della capacità e, più in

generale, alla modifica della priorità del server vero e proprio in modo da interromperne l'esecuzione grazie alla preemption.

Il funzionamento è molto simile al meta-scheduler per via della gestione tramite stato, ma invece di tenere conto dei diversi stati dei task periodici, in questo caso è necessario tenere traccia solo del proprio stato.

Fino al termine della gestione delle richieste aperiodiche il thread si sospende in attesa di un messaggio, all'arrivo del quale discerne le operazioni da svolgere combinando lo stato corrente con il valore del pulse ricevuto.

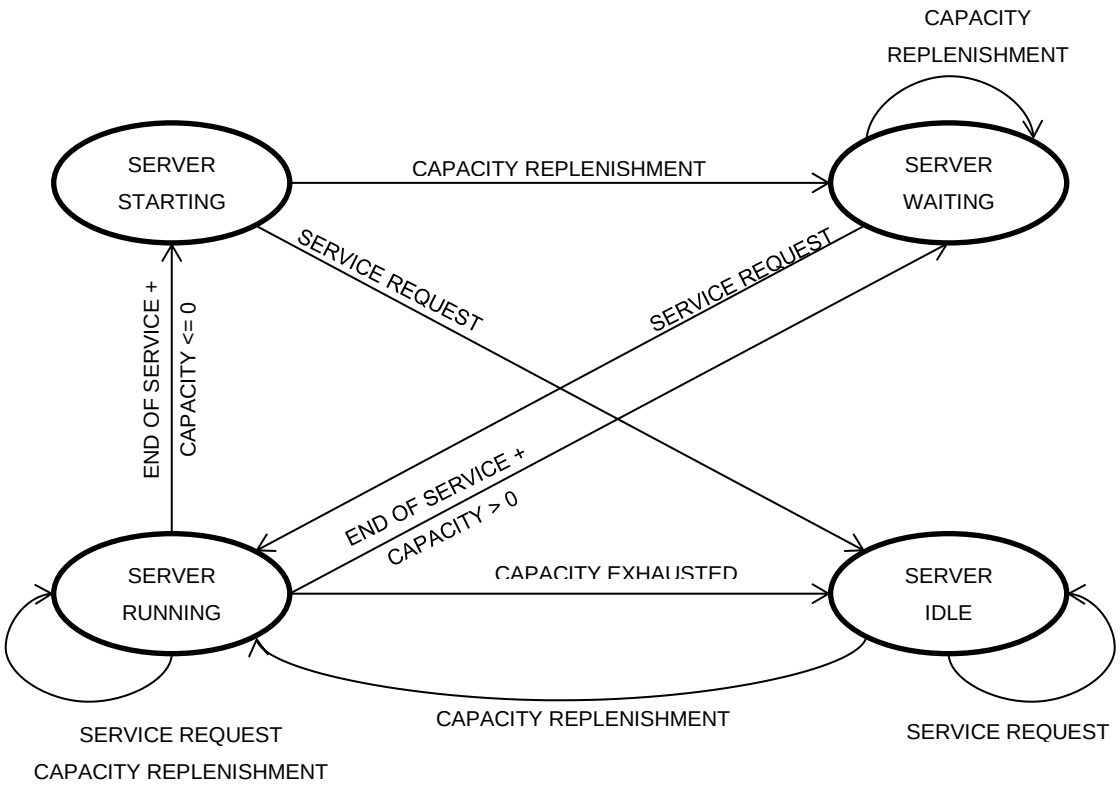

Figura 4.17 Cambiamenti di stato del deferrable server

Il server inizia la simulazione nello stato *SERVER_STARTING* e con una capacità iniziale pari a 0 millisecondi. In questa situazione il server può ricevere solo due tipi di notifiche. L'arrivo di una nuova richiesta aperiodica (*SERVICE_REQUEST_PULSE_CODE*) che comporta il passaggio del server allo stato *SERVER_IDLE* o la segnalazione di ripristinare la capacità (*CAPACITY_REPLENISHMENT_PULSE_CODE*).

In questo caso il thread ripristina la capacità al valore nominale C_s specificato nei parametri (reso disponibile dal main) e modifica il proprio stato in *SERVER_WAITING*. In questo stato il server attende l'arrivo di una richiesta aperiodica. L'arrivo di questa, segnalata dallo specifico pulse, causa il ripristino della priorità nominale del server (la più bassa prima di quelle assegnate ai task periodici) e il passaggio allo stato *SERVER_RUNNING*. Nel caso non arrivi alcuna richiesta, il server tiene comunque conto del ripristino della capacità allo scadere del periodo, segnalato tramite il pulse inviato dal thread di controllo.

Al passaggio allo stato *SERVER_RUNNING* il thread registra il tempo attuale e imposta nella struttura dati destinata ad accogliere i messaggi da ricevere, un pulse di tipo *CAPACITY_EXHAUSTED_PULSE_CODE* e, prima di sospendersi in attesa di un nuovo messaggio, imposta un timeout con durata pari alla capacità residua. Una volta sbloccato, prima di controllare il tipo di messaggio ricevuto, viene calcolato il tempo effettivo di esecuzione. Se il messaggio ricevuto è di tipo *SERVICE_REQUEST_PULSE_CODE* il server aggiorna la capacità residua e torna in esecuzione. In caso di scadenza del periodo T_s la capacità viene ripristinata al valore nominale C_s. Un messaggio ricevuto di tipo *END_OF_SERVICE_PULSE_CODE* causa l'aggiornamento della capacità residua e la modifica dello stato del server. A seconda che il server abbia ancora a disposizione della capacità o meno lo stato raggiunto sarà *SERVER_WAITING* o *SERVER_STARTING*. Nel caso in cui invece non sia stato ricevuto alcun messaggio, ma sia scattato il timeout, il server azzera la capacità residua, riduce la priorità del thread dedicato al servizio delle richieste (*Deferrable_Server_Task*) alla priorità di "sospensione" e modifica il proprio stato in *SERVER_IDLE*. Quest'ultimo stato indica che il server ha delle richieste da servire ma non dispone della capacità per poterlo fare. Quindi in caso di una nuova richiesta, continua l'attesa. Nel caso di ripristino della capacità, il valore

viene aggiornato al valore nominale, la priorità del thread di servizio viene riportata al valore effettivo e il server passa nello stato *SERVER_RUNNING*.

Qui si conclude la descrizione delle implementazioni degli algoritmi di schedulazione e gestione delle richieste aperiodiche. Però, senza un sistema per visualizzare i risultati (dati temporali) del simulatore, non è possibile effettuare una valutazione del sistema e verificare che questo funzioni come previsto. Per testare il sistema QNX è stato quindi necessario tenere traccia dei dati temporali durante l'esecuzione della simulazione per poi esporli in maniera chiara una volta che fosse terminata. La combinazione di questo con il sistema di log interno di QNX opportunamente configurato permette una facile comprensione dei dati del simulatore. Seguirà una breve descrizione di come è stato realizzato il sistema di log del simulatore.

4.4 Log simulatore

Durante l'esecuzione il simulatore elabora una gran quantità di dati riferiti all'evoluzione temporale dei processi periodici e delle richieste aperiodiche. Dato che la stampa di tutti questi dati risulterebbe poco leggibile e grazie alla presenza del sistema di log interno di QNX che permette di visualizzare la timeline degli eventi, al termine dell'esecuzione vengono stampati solo alcuni dati particolarmente significativi.
Per ogni processo periodico vengono riportati:

- il numero di deadline mancate, comprese quelle in seguito all'avvio di un task già in ritardo a causa di un precedente job overrun (con politica di gestione dei sovraccarichi *ASAP*);
- il numero di job overrun, uno per ogni periodo in cui non è stato possibile lanciare un nuovo job a causa dell'esecuzione del job ritardatario;

- la durata del job più lungo di ogni task, per una migliore interpretazione dei due dati precedenti.

In questo modo è possibile che ad un processo corrispondano più job overrun rispetto alle deadline mancate. Questo indica che almeno un job del task in questione ha bloccato almeno due release dei job successivi a causa del suo ritardo.

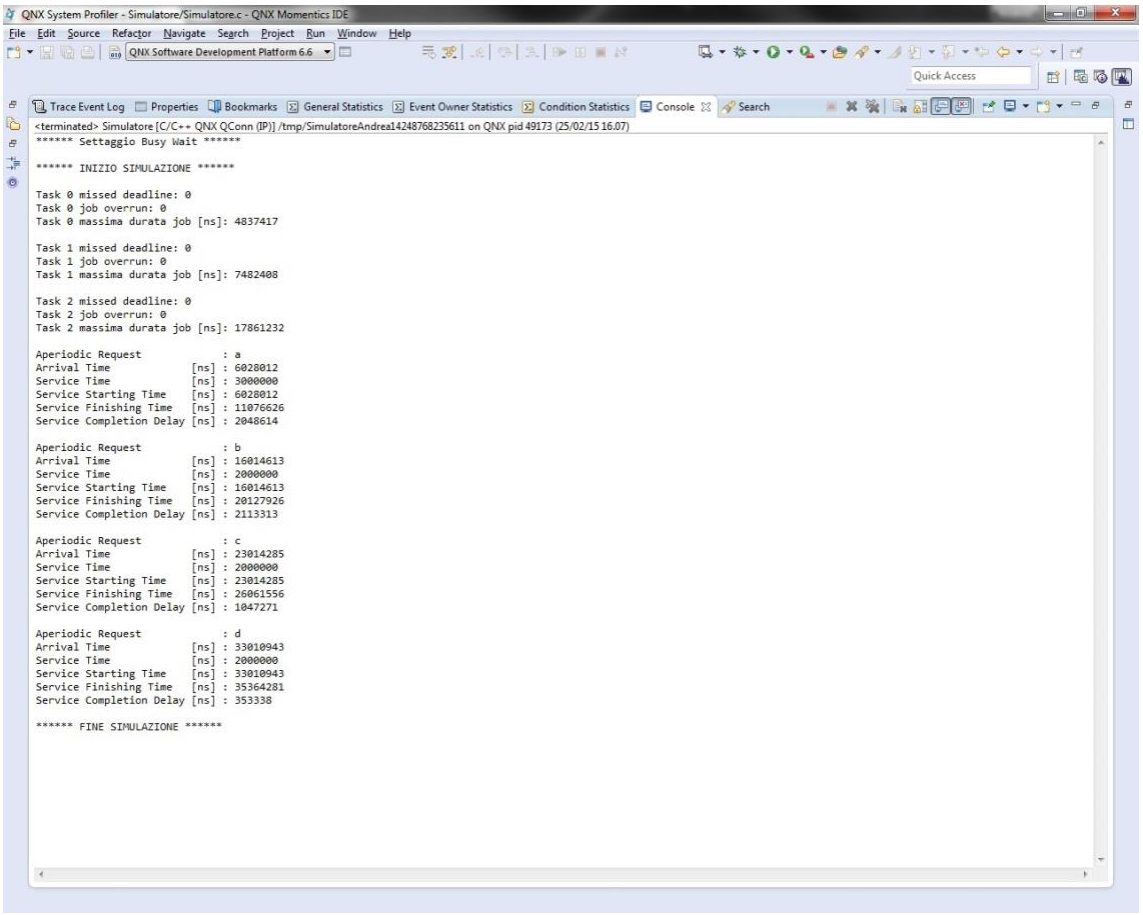

Figura 4.18 Esempio dell'output inviato tramite console al termine della simulazione

Per le richieste aperiodiche vengono registrati e stampati a video i seguenti dati:

- tempo di effettivo di arrivo della richiesta;
- tempo di servizio previsto;
- tempo di inizio del servizio della richiesta;
- tempo di fine del servizio della richiesta;

- ritardo nel completamento del servizio, calcolato sottraendo il tempo di completamento, il tempo di inizio del servizio e il tempo di servizio.

Con questi dati, in particolare attraverso l'analisi del ritardo, è possibile rendersi conto dell'efficienza effettiva dei diversi tipi di server implementati.

4.4.1 Configurazione sistema di log QNX

Dato che l'analisi di dati puramente numerici risulta di difficile interpretazione, si è deciso di sfruttare il più possibile il sistema di log di QNX per affiancare l'output vero e proprio del simulatore nell'analisi del sistema. Il sistema permette di tenere traccia di un gran numero di eventi, divisi per categorie: comunicazione, eventi di controllo, invocazione delle kernel call, eventi dei processi e dei thread, eventi utente. Per ottimizzare il log e renderlo il più leggero possibile per il sistema, è stato necessario decidere di quali eventi di sistema tenere traccia durante il logging. Inizialmente si è cercato di associare ad ogni evento significativo della simulazione, come per esempio l'arrivo di una nuova richiesta aperiodica, un evento registrabile durante il logging sulla base del programma già sviluppato. Questo non è stato però possibile nella totalità dei casi ma si è visto necessario sfruttare alcune kernel call specifiche per rilevare l'avvenimento delle operazione del simulatore.

Il sistema di log è stato configurato in maniera da tenere traccia dei seguenti eventi di sistema:

- per le kernel call, *ChannelDestroy Enter*, *ConnectAttach Enter*, *ConnectDetach Enter*, *ConnectFlags Enter*, *SchedGet Enter*;
- gli interrupt non sono stati registrati;
- per i processi e i thread sono stati registrati tutti gli eventi tranne quelli riguardati l'uso di variabili virtuali;
- gli eventi di sistema non sono stati registrati;
- per le comunicazioni è stato registrato solo l'invio dei pulse.

Il log di tutti questi eventi avviene in modalità fast.

Alcuni di essi vengono registrati anche durante la preparazione alla simulazione ma, dopo l'avvio, che avviene con un certo ritardo (sfruttando una barriera), si associano in maniera univoca a particolari eventi di simulazione, tramite l'esecuzione delle kernel call associate ad essi.

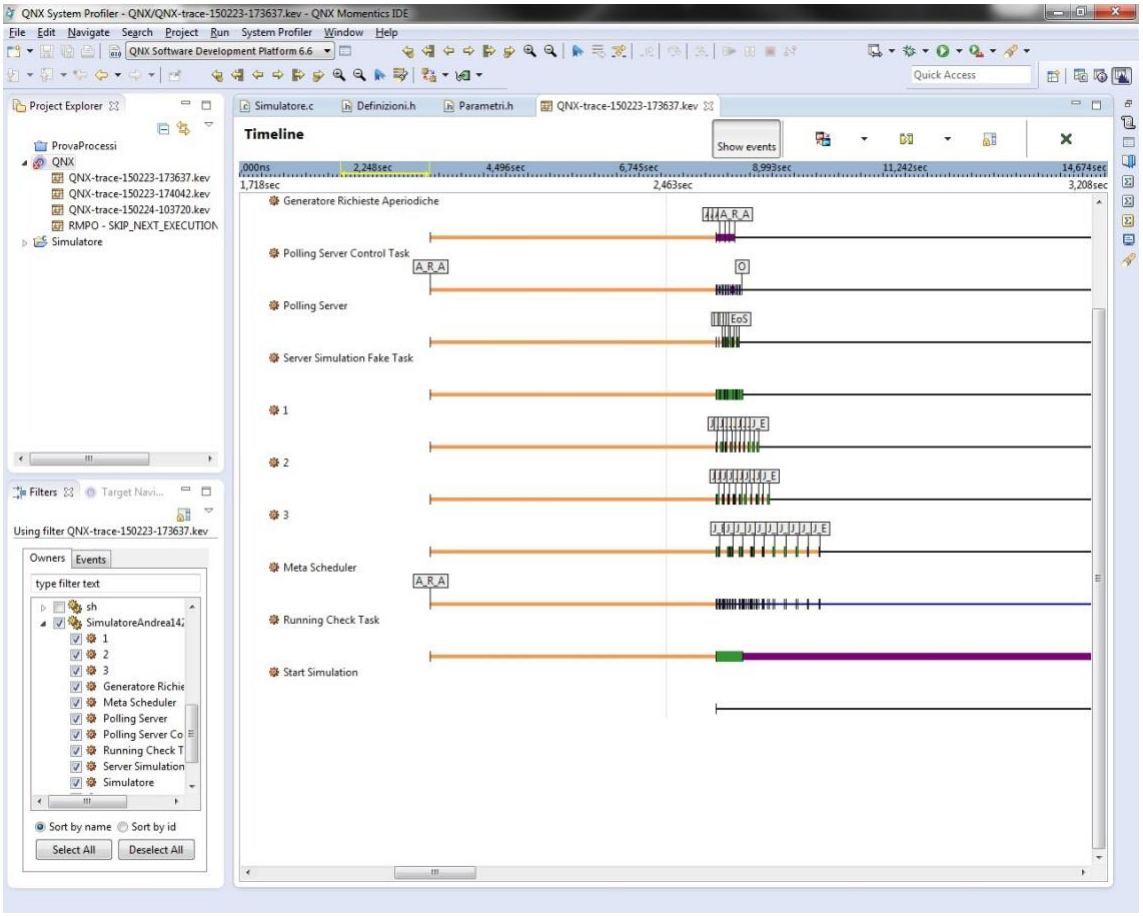

Figura 4.19 Esempio di simulazione con le etichette associate agli eventi specifici

Le kernel call vengono eseguite con tutti i parametri a *NULL*, in modo che non eseguano alcuna operazione, ma vengano comunque rilevate dal sistema di log.

La tabella delle associazioni è la seguente:

Evento simulatore	Kernel call	Etichetta
Job Start Of Execution		
Job End Of Execution	MsgSendPulse	J_E
Missed Deadline	ChannelDestroy	D
Job Overrun	ConnectDetach	O

Aperiodic Request Arrival	ConnectAttach	A_R_A
Aperiodic Request Start Of Service	ConnectFlags	SoS
Aperiodic Request End Of Service	SchedGet	Eos

Per l'inizio dell'esecuzione di un job si è deciso di non utilizzare nessuna kernel call, in quanto l'evento risulta già chiaramente visibile nella timeline e per il fatto che non sarebbe stato possibile segnalarlo in maniera chiara. Infatti all'inizio della simulazione ogni task avvia un nuovo job e al termine di ogni periodo (ad esclusione dei casi di sovraccarico) il task inizia un altro job, ma la kernel call verrebbe invocata solo se il task andasse immediatamente in esecuzione, cosa non certa ma dipendente dalla priorità dei task. Risulta comunque chiaro nella timeline che un nuovo job viene avviato non appena il task passa dallo stato di sospensione a quello di ready o running. Questo è facilmente visibile poiché ogni evento, compresi quelli specifici di thread e processi, è visualizzato tramite una linea nera verticale sulla linea temporale associata al thread specifico. Inoltre il sistema sfrutta un comodo sistema di colori per identificare lo stato di ogni thread: verde chiaro per lo stato running, verde scuro per un thread ready, arancione per un thread sospeso su una variabile (nel caso del simulatore un semaforo), viola per identificare una sleep in corso e blu per i la sospensione in attesa della ricezione di un messaggio.
È da tenere presente che mentre gli eventi dovuti alla mera esecuzione di un servizio sono inviati e quindi visualizzati dal thread specifico, questo non è possibile per i casi delle deadline e dei job overrun. Infatti è il meta-scheduler che si occupa del loro rilevamento e della loro segnalazione, ma incrociando la timeline con l'output del simulatore è possibile effettuare facilmente l'associazione deadline / job overrun e task periodico.

Nonostante gli accorgimenti per ridurre al minimo il peso del logging per il sistema, questo influisce sull'esecuzione del simulatore e non risulta utilizzabile nei casi in cui il processore abbia pochi o nessun tempo morto.

Capitolo 5 - Risultati sperimentali

Una volta concluso lo sviluppo del simulatore, si è potuto procedere al test per verificare le prestazioni del meta-scheduler e del sistema operativo QNX Neutrino. In questo capitolo saranno esposti gli esperimenti effettuati, i loro risultati e alcuni problemi riscontrati durante la fase di test con la relativa soluzione adottata.

5.1 Calibrazione busy wait

Il primo problema riscontrato ha riguardato il tempo di esecuzione effettivo dei task periodici e delle richieste aperiodiche. Utilizzando la funzione di calibrazione della busy wait si ottenevano tempi inferiori a quanto richiesto, rendendo difficile la verifica effettiva delle prestazioni del simulatore non essendo possibile effettuare una simulazione precisa. Per superare il problema era necessario ridurre il tempo di riferimento (Reference_Time_ns) calcolato durante la calibrazione per fare in modo che la busy wait durasse per un tempo maggiore. Al termine dei test la riduzione necessaria per ottenere i tempi desiderati è risultata pari al 17%. I tempi di esecuzione dei task risultano così molto simili a quelli richiesti, ma è importante tenere in considerazione che una modifica percentuale eseguita su ogni task indipendentemente dalla sua durata mal si rapporta con task con esecuzioni molto diverse tra loro.

5.2 Condizioni di schedulabilità

Per schedulabilità si intende la possibilità di eseguire un certo numero di task in maniera continuativa senza che si verifichino deadline mancate o job overrun. Per verificare che i processi siano schedulabili esistono metodi diversi a seconda dell’algoritmo di schedulazione utilizzato e dalla presenza o meno di richieste aperiodiche. I concetti base sono però condivisi.

Definiamo il fattore di utilizzazione del processore (**U)** come la frazione di tempo utilizzata dalla CPU per eseguire l'insieme di task.

Per un insieme di n task vale: $U = \sum_1^n \frac{C_i}{T_i}$, con C_i tempo di esecuzione e T_i periodo del task i-esimo. Il processore risulta "utilizzato completamente" se la schedulazione diventa infattibile in conseguenza di un aumento di uno dei C_i. Condizione sufficiente per cui un insieme di processi risulti non schedulabile è U > 1. Infatti prendendo $T = \prod T_i$ si ottiene $UT > T$ quindi $\sum \frac{T}{T_i} C_i > T$. Dato che $\frac{T}{T_i} C_i$ rappresenta il tempo di calcolo richiesto dall'i-esimo task nel tempo T, risulta che la domanda totale nel tempo [0,T] è superiore al tempo disponibile T, quindi la schedulazione è impossibile indipendentemente dall'algoritmo utilizzato.

Le analisi di schedulabilità che seguiranno non tengono conto dell'esecuzione del meta-scheduler e dei thread di gestione delle richieste aperiodiche, ma sono un buon punto di partenza per verificare le prestazioni del sistema.

5.3 Schedulazione task periodici

Nei paragrafi seguenti verranno analizzate le condizioni di schedulabilità dei diversi algoritmi di schedulazione implementati.

5.3.1 Rate Monotonic Priority Ordering (RMPO)

Condizione sufficiente per la schedulazione di un insieme di processi "semplicemente periodici" (cioè con periodi in relazione armonica) è U<=1.

Il test effettuato con i parametri della figura 5.1 sul simulatore (per una durata prevista di 500 ms) causa però un sovraccarico, con il terzo task incapace di concludere in tempo la propria esecuzione. In caso di uso della politica *ASAP* per la gestione dei sovraccarichi, il sistema non riesce a recuperare, mentre nel caso della politica *SKIP* il sistema riesce a

ridurre i sovraccarichi, ma questo è imputabile al numero limitato di job previsti per ogni task. Resta che la condizione di schedulazione non risulta valida per il sistema corrente.

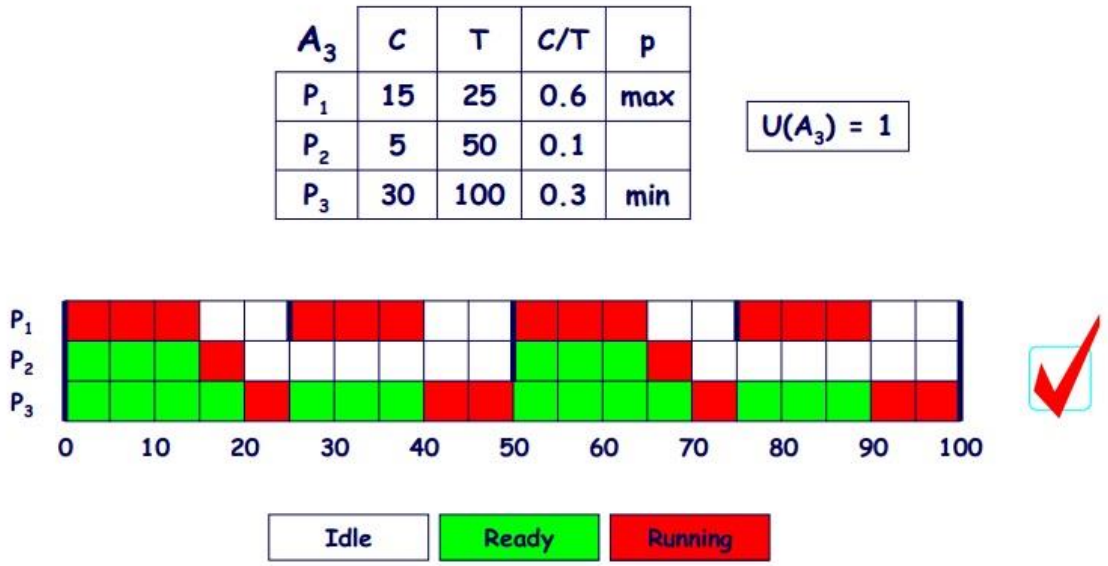

Figura 5.1 Analisi di schedulabilità di processi "semplicmente periodici" [9]

Riducendo di 10 ms il periodo di esecuzione del task numero 3 (U = 0,9) questo insieme di processi risulta schedulabile. Il meta-scheduler richiede quindi tra l'1 e il 10% del tempo del processore.

Ulteriori test hanno portato a stabilire il limite massimo per la schedulabilità di processi "semplicemente periodici" in U <= 0,9.

Ma i processi di questo tipo rappresentano solo una piccola parte dei casi reali ed è necessario trovare un limite massimo di U per la schedulabilità di processi a cui non siano stati applicati vincoli così restrittivi sul periodo.

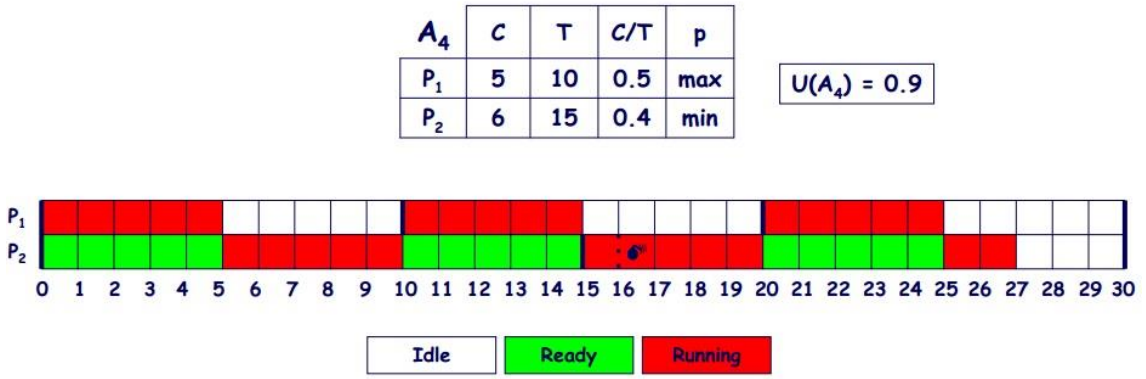

Figura 5.2 Esempio di schedulazione non fattibile con processi con U<=1 [9]

Un test basato completamente sul fattore di utilizzazione del processore è stato trovato nel 1973 da Liu & Layland [6].

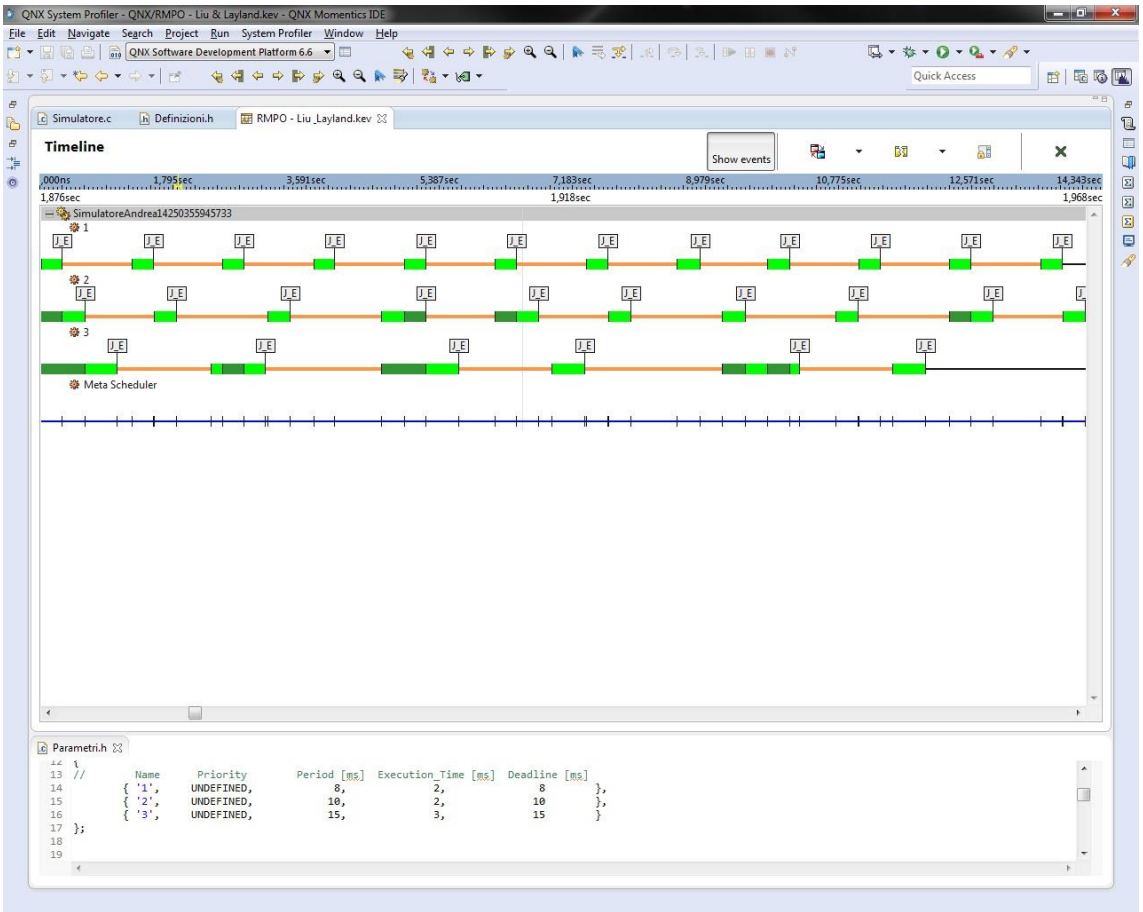

Figura 5.3 Esempio di schedulazione tramite RMPO con U = 0,65. Ogni task riesce a soddisfare la propria deadline per tutta la durata della simulazione

Lo studio asserisce che condizione sufficiente, ma non necessaria, perché un insieme di N processi sia schedulabile con l'algoritmo RMPO è

che $U \leq U_{RMPO}(N) = N\left(2^{\frac{1}{N}} - 1\right)$. U_{RMPO} risulta 0.8284 per 2 task. Procedendo con un numero di processi tendenti all'infinito si arriva a $\lim_{n\to\infty} n\left(\sqrt[n]{2} - 1\right) = \ln 2 \approx 0.693147$ .... Quindi un insieme di processi può rispettare tutte le proprie deadline se l'utilizzo della CPU richiesto è minore del 69,32%. L'altro 30,7% della CPU può essere utilizzato per la schedulazione di processi a bassa priorità non real-time.

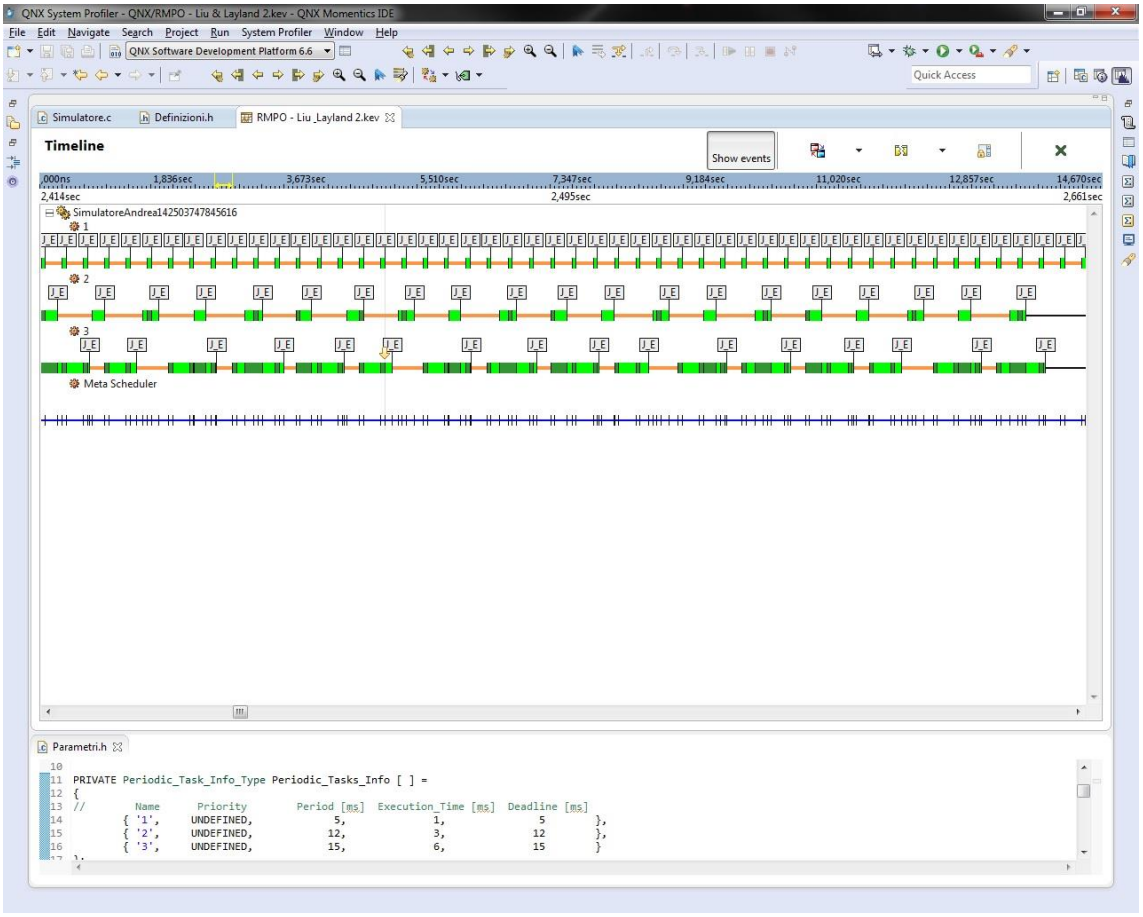

Figura 5.4 Esempio di schedulazione tramite RMPO con U = 0,85 . Nonostante il limite teorico sia di 0,8 per 3 processi la schedulazione risulta fattibile

Questo test risulta però estremamente restrittivo, infatti è possibile che, a seconda dei parametri dei diversi task, un insieme di processi risulti schedulabile anche se il test asserisce il contrario. Nella maggior parte dei casi, però i test più accurati risultano troppo vincolati ad un sistema matematico, che non tiene in considerazione i tempi "tecnici", tra cui anche l'esecuzione del meta-schedueler, necessari all'elaborazione dei

diversi processi. Infatti in alcuni dei diversi esperimenti effettuati è stato sufficiente avvicinarsi al limite indicato dal teorema di Liu & Layland perché i processi non riuscissero più a rispettare le proprie deadline.

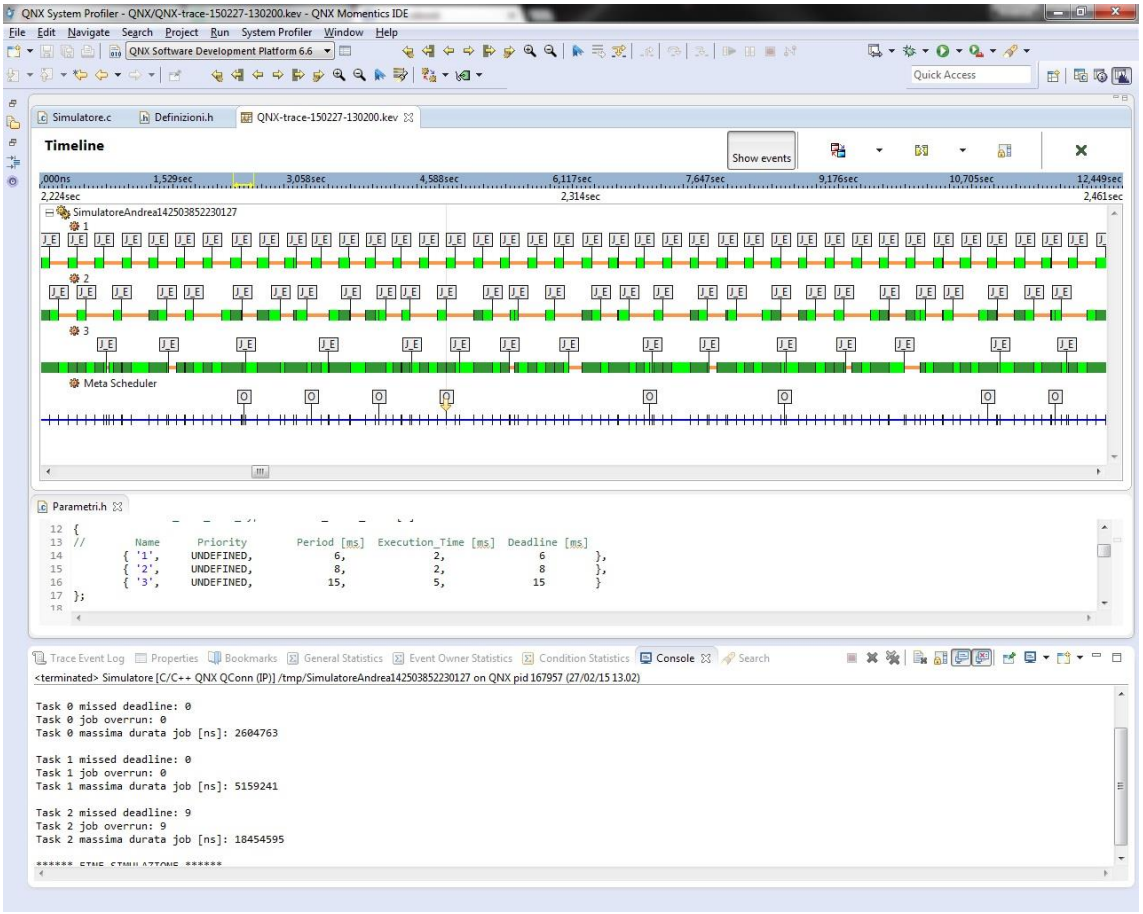

Figura 5.5 Esempio di RMPO con schedulazione fallita con politca di gestione dei sovraccarichi ASAP

5.3.2 Deadline Monotonic Priority Ordering (DMPO)

Trattandosi anche DMPO di un algoritmo ad assegnazione di priorità statica, i test per verificare la schedulabilità risultano simili o estensioni di quelli utilizzati per RMPO. Il più semplice, ma quindi anche il più restrittivo, risulta il test basato sulla “densità di utilizzazione” del processore.

Se $\Delta = \sum_{j=1}^{N} \frac{C_j}{D_j} \leq U_{RMPO}(N) = N\left(2^{\frac{1}{N}} - 1\right)$ è rispettata allora l’insieme di N processi periodici e sporadici è schedulabile con l’algoritmo DMPO.

Anche in questo caso esistono diversi algoritmi meno restrittivi e più efficaci, ma lo scopo di questo simulatore consiste anche nel verificare la schedulabilità di eventuali insiemi di processi senza dover ogni volta applicare i diversi test, che come già detto non tengono conto dei tempi necessari agli strumenti di gestione.

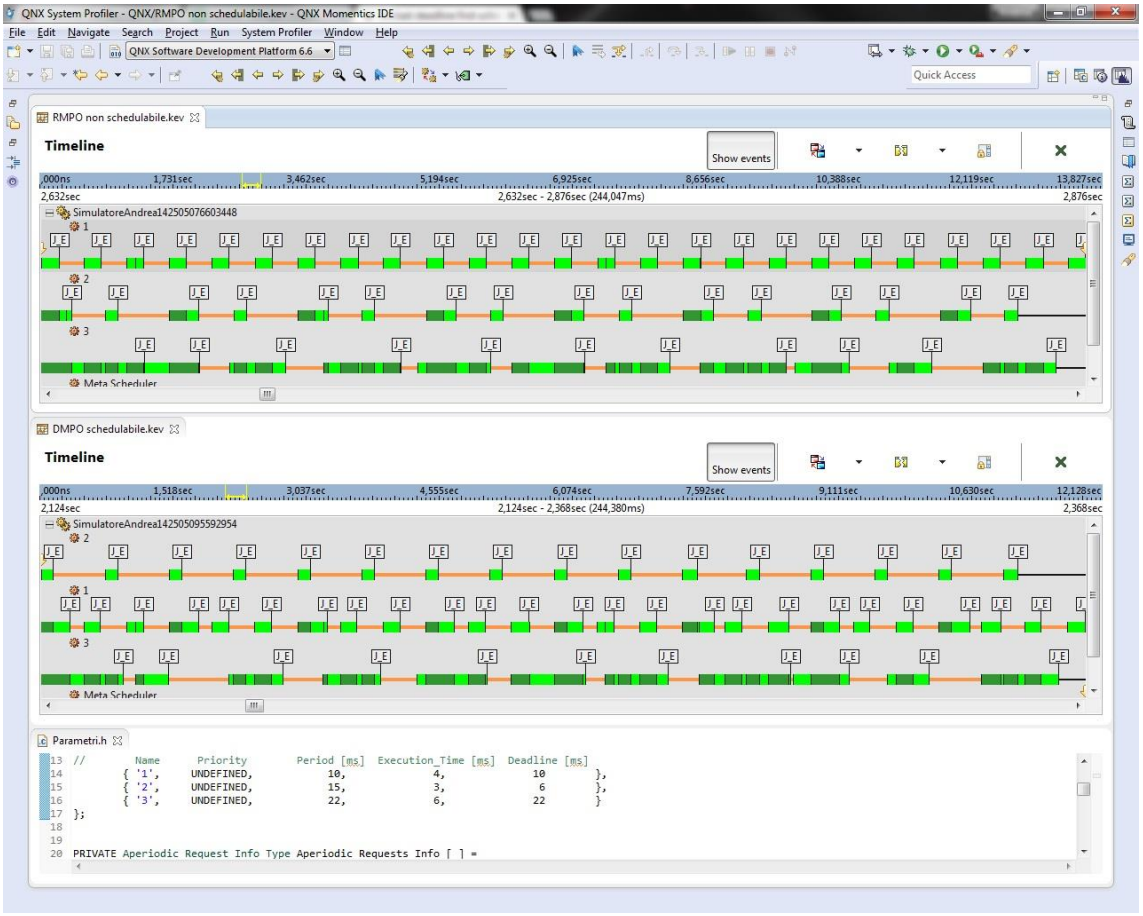

Figura 5.6 Esempio di schedulazione tramite RMPO e DMPO. RMPO non è in grado di garantire la schedulazione al contrario di DMPO.

Una simulazione della durata di 500 ms, richiede solo una quindicina di secondi tra il tempo per l'avvio del log, il completamento della simulazione e la ricezione del log. In questo modo è anche possibile verificare che l'insieme di processi sia schedulabile per l'ambiente in questione, cioè il sistema operativo QNX Neutrino con il sistema di gestione del simulatore.

L’implementazione di quest’algoritmo permette comunque la schedulazione di processi sporadici e periodici che solo attraverso RMPO non sarebbero possibili.

5.3.3 Earliest Deadline First (EDF)

L’algoritmo di schedulazione più interessante per valutare le prestazioni del sistema è senza dubbio EDF. Grazie alla modifica dinamica delle priorità dei task periodici e sporadici è infatti capace di sfruttare al meglio la CPU e risulta l’algoritmo meno dipendente dai parametri specifici dei diversi task per verificare la schedulabilità a priori di un insieme di task.

Dato un insieme di N processi periodici l’unica condizione necessaria (e sufficiente) per garantire la schedulabilità è che $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq U_{EDF} = 1$.

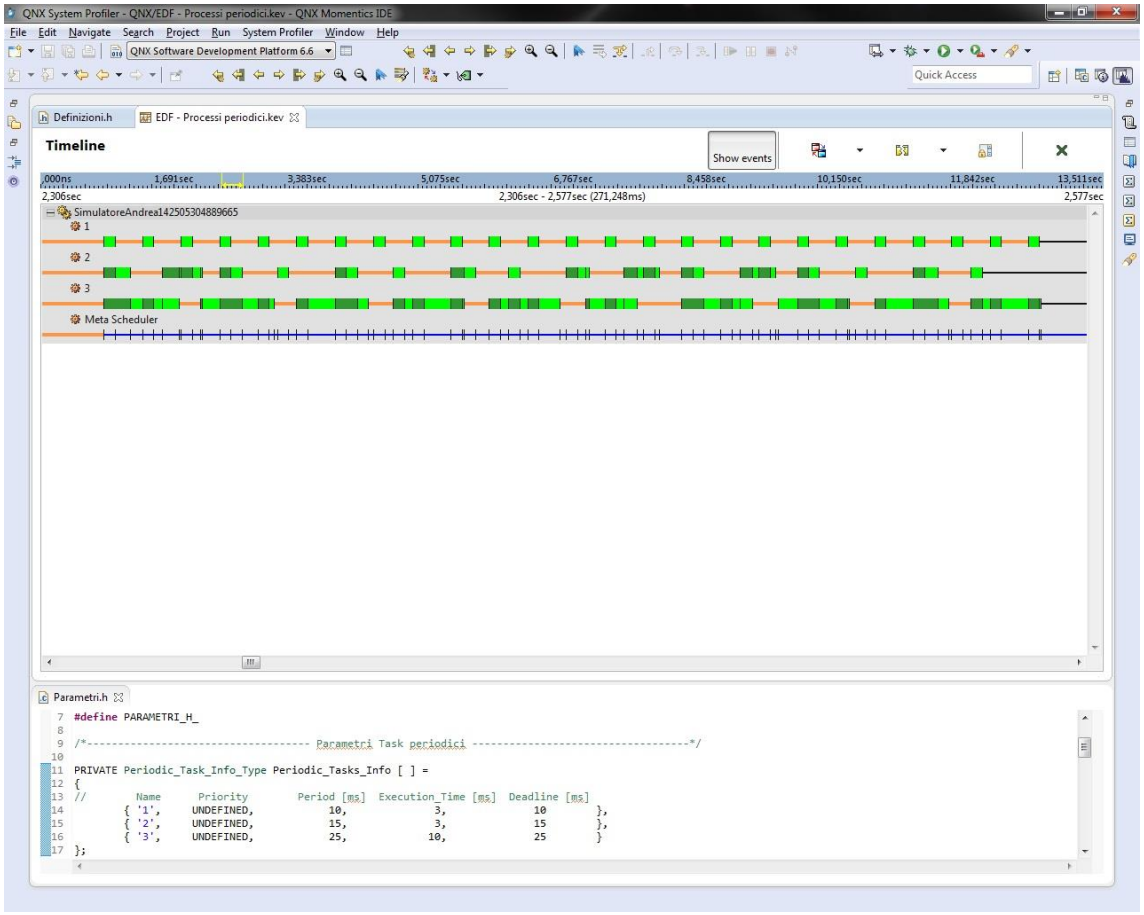

Figura 5.7 Esempio di schedulazione di processi periodici tramite EDF. U = 0,9

In questi casi il simulatore è in grado di schedulare senza errore anche un insieme di processi aventi U = 0.98. Nei casi con un fattore di utilizzazione così alto non è però possibile effettuare il log del sistema per ottenere una timeline. Il processore non ha infatti tempo sufficiente a gestire il sistema di log che termina in seguito allo scadere di un timeout. Il tentativo di avviare il logging causa inoltre una riduzione dell'efficienza del sistema rendendo non schedulabile un insieme di processi altrimenti schedulabile correttamente.

Per la schedulazione di processi periodici e sporadici è necessario seguire un approccio diverso, tenendo conto delle differenti deadline. Un insieme di N processi periodici e sporadici è schedulabile con EDF se è rispettata la seguente condizione: $\Delta = \sum_{i=1}^{N} \frac{C_i}{D_i} \leq U_{EDF} = 1$.

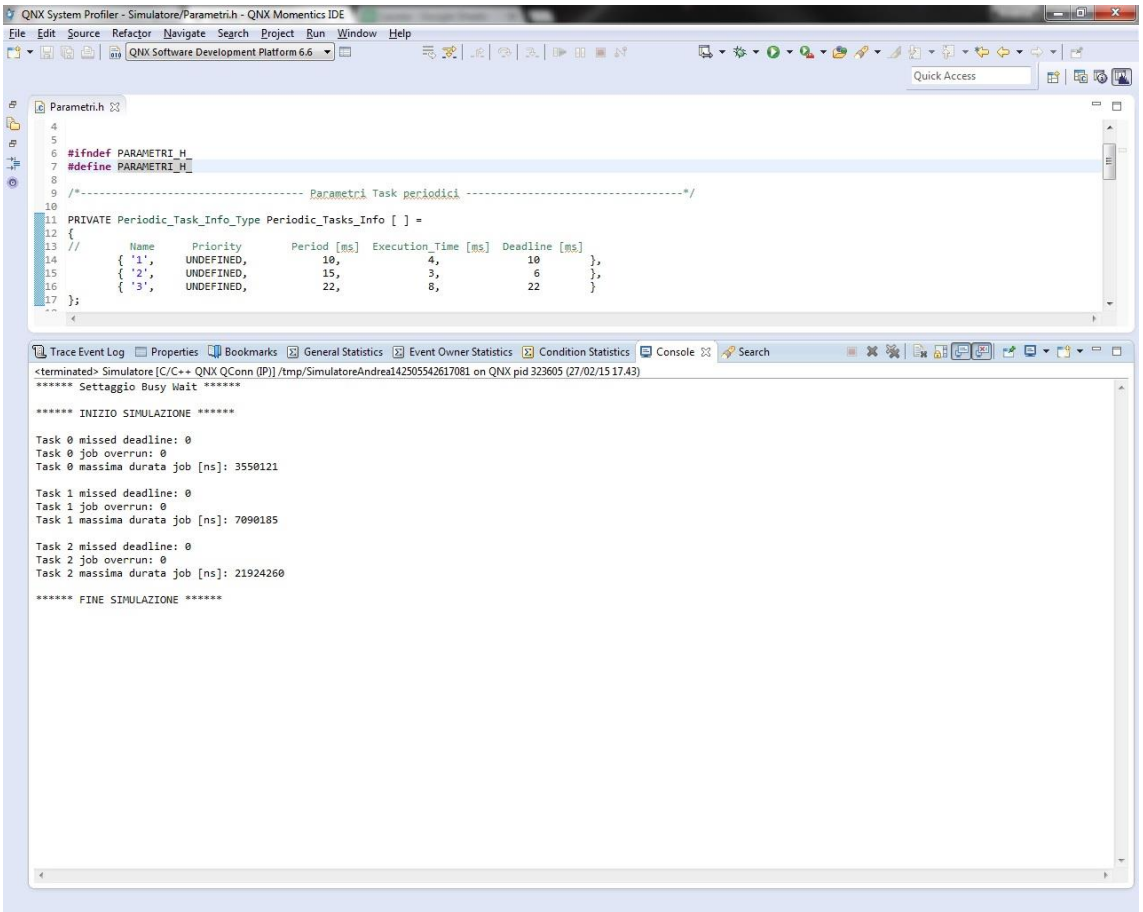

Figura 5.8 Esempio di schedulazione tramite EDF. In questo caso abbiamo U = 0,92 e Δ=1,22 ma l'insieme di processi risulta schedulabile. Non è tuttavia effettuabile il log a causa dell'elevato utilizzo della CPU

È possibile che in alcuni casi, anche di fronte ad un Δ > 1 la schedulazione teorica risulti comunque possibile.

Per verificare questo è necessario utilizzare l'approccio "Process Demand" [7], che dato un insieme di N processi periodici e sporadici, contraddistinti da un fattore di utilizzazione U<=1 ed attivati contemporaneamente all'istante 0, ne garantisce la schedulabilità se in ogni intervallo [0,t] il tempo di elaborazione cumulativamente richiesto per completare l'esecuzione di tutti i job aventi deadline <= t non eccede il tempo disponibile t:

$$C_p(0,t) = \sum_{i=1}^{N} C_i(0,t) = \sum_{i=1}^{N} \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor - 1 \right) C_i \leq t$$

Tuttavia come nel caso di RMPO e DMPO anche qui non viene tenuto conto del meta-scheduler. È quindi necessaria una simulazione di prova sul sistema, tramite il simulatore sviluppato per accertarsi che la schedulazione sia fattibile.

5.4 Meta-scheduler

Come è stato visto il meta-scheduler influisce sulla schedulabilità dei task periodici. Tuttavia come è stato indicato precedentemente è stato possibile schedulare insiemi di processi con un fattore di utilizzazione del processore pari al 98%. Le richieste del meta-scheduler (e del sistema operativo) risultano quindi minime. Nel valutare la schedulabilità di un algoritmo nel momento di considerare l'influenza che può avere il meta-scheduler bisogna tenere in considerazione la modalità di esecuzione di quest'ultimo. Questa dipende dai parametri dei diversi processi periodici e sporadici in esecuzione. Il meta-scheduler esegue infatti solo in presenza di istanti critici relativi ai diversi processi. Questi istanti sono: la release di un nuovo job, la scadenza di una deadline e il termine di un job. A parità di fattore di utilizzazione quindi processi con periodi (e quindi deadline)

minori tendono a richiamare più spesso il meta-scheduler e a causare un peggioramento delle prestazioni del sistema.
Gli eventi che causano il risveglio del meta-scheduler hanno un diverso peso temporale.
I diversi casi che possono verificarsi con annessi tempi medi di esecuzione sono visibili nella tabella 1.

Evento \ Stato task	*RUNNING*	*MISSED_DEADLINE*	*JOB_OVERRUN*	*IDLE*
RELEASE JOB	900 ns	400 ns	900 ns	900 ns
DEADLINE	600 ns	X	X	X
END JOB	450 ns	300 ns	450 ns (SKIP) 450 ns (ASAP)	X

Tabella 1 Tempi di esecuzione del meta-scheduler. I casi indicati con X non possono verificarsi

I tempi di esecuzione del meta-scheduler risultano estremamente ridotti, sforando solo occasionalmente il microsecondo. Incrociando i tempi esposti nella tabella 1 con i parametri dei diversi task periodici è possibile ricavare il peso del meta-scheduler in base allo specifico insieme di processi che si vuole schedulare. È da tenere in considerazione che in caso di sforamento delle deadline e/o di sovraccarichi il meta-scheduler andrà in esecuzione più frequentemente, peggiorando ulteriormente le prestazioni della simulazione.

5.5 Schedulazione richieste aperiodiche

Per garantire la schedulabilità dei task periodici in presenza di richieste aperiodiche è necessario utilizzare test diversi rispetto a quanto visto finora.
L'unico fattore comune a tutti e tre gli algoritmi implementati consiste nel generatore di richieste aperiodiche. Questo non ha un'influenza calcolabile sulla base di parametri specifici, ma ad ogni "arrivo" di richiesta aperiodica entra in esecuzione per un tempo medio di

esecuzione pari a 1,9 microsecondi. Questo può causare problemi nel caso il fattore di utilizzazione del processore per l'insieme di processi in schedulazione sia estremamente elevato, in caso contrario risulta quasi trascurabile.

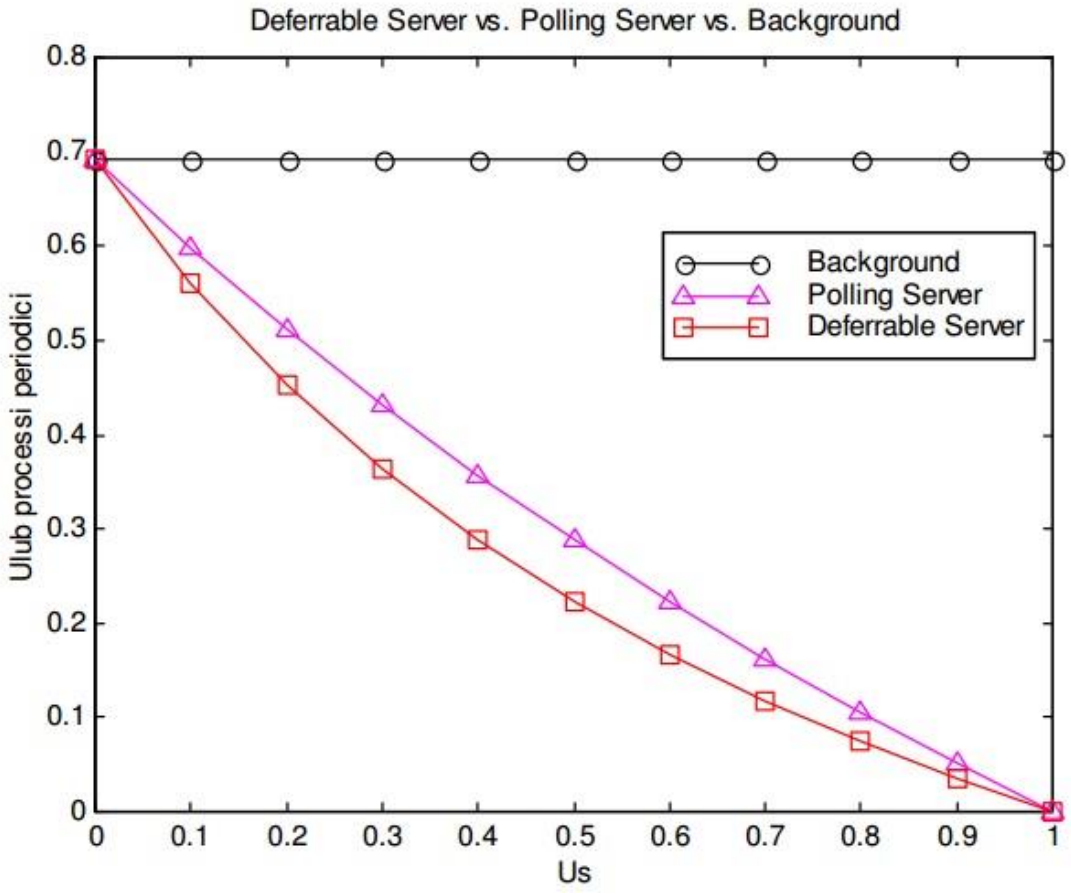

Figura 5.9 Comparazione tra i diversi servizi di gestione delle richieste aperiodiche implementati in relazione al massimo fattore di utilizzazione attribuibile ai task periodici sulla base del fattore di utilizzazione specifico del servizio utilizzato (schedulazione tramite RMPO) [10]

5.5.1 Servizio in background

Il servizio in background teoricamente non dovrebbe causare alcun problema alla schedulazione dei processi periodici e sporadici, poiché la gestione delle richieste avviene solo nei tempi in cui il processore non è utilizzato dai task periodici o sporadici. Tuttavia il servizio di generazione delle richieste può causare un ritardo nella schedulazione di detti processi e causare deadline mancate o sovraccarichi.

Per verificare quanto influisce questo fattore, si è deciso di usare come algoritmo di schedulazione dei task periodici EDF, poiché permette di sfruttare al meglio il processore. Al contrario di quanto avviene senza la gestione delle richieste aperiodiche, un insieme di processi con U = 0,98 non è schedulabile, ma il peso del servizio di gestione e generazione delle richieste risulta pari o inferiore ad un 1%. Infatti un insieme di processi con fattore di utilizzazione del processore pari al 97% è risultato schedulabile.

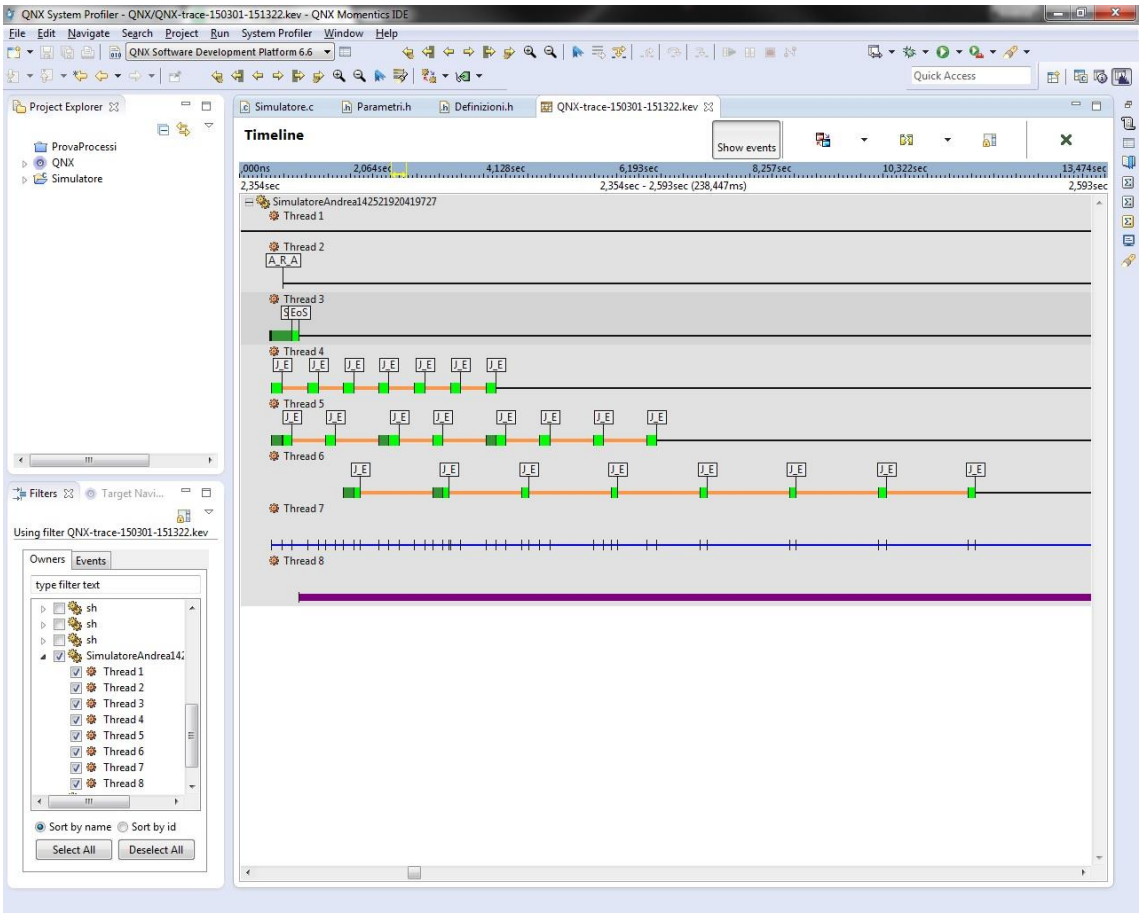

Figura 5.10 Esempio di file di log generato durante l'utilizzo del servizio in background. Da notare la mancanza della maggior parte degli eventi e i thread senza nome

Purtroppo non è possibile effettuare il log del sistema durante l'esecuzione del servizio in background (o nei casi in cui il server a priorità statica in assenza di capacità agisca come questo), perché l'elevato numero di eventi generati dalla lettura continua della pipe in

attesa di nuove richieste rende impossibile al sistema la generazione di un file di log affidabile, dato che i buffer adibiti alla raccolta dati vengono persi.

Il risultato è un file di log in cui la maggior parte degli eventi non compaiono e/o sono in ordine diverso da quanto avvenuto in realtà (v. fig. 5.10).

5.5.2 Polling server

Nel caso di gestione delle richieste aperiodiche tramite server a priorità statica, la schedulazione dei processi periodici avviene tipicamente utilizzando l'algoritmo RMPO. È possibile anche in questo caso utilizzare il testi di schedulazione di Liu & Layland considerando il processo server un processo periodico. Nel caso specifico del polling server il massimo carico computazionale indotto dal server è pari a $U_s = C_s / T_s$. Questo rappresenta il caso peggiore, in cui ad ogni periodo del server e al termine del servizio di ogni richiesta aperiodica sia presente nella pipe un'ulteriore richiesta da servire. Quindi, condizione sufficiente ma non necessaria, per la schedulazione di un insieme di N processi periodici, contraddistinti da un fattore di utilizzazione del processore U_p è:

$$U_P + U_s \leq U_{RMPO}(N+1) = (N+1)(2^{\frac{1}{N+1}} - 1).$$

Il server deve quindi avere un fattore di utilizzazione U_s tale che:

$$U_s \leq \frac{2}{\left(1+\frac{U_P}{N}\right)^N} - 1 =_{N\to\infty} \frac{2}{e^{U_P}} - 1.$$

Questo sempre nel caso teorico in cui i servizi di gestione non richiedano l'uso del processore. Nel caso del polling server, oltre al sempre presente generatore delle richieste aperiodiche, è presente anche il thread per la gestione della capacità che riduce le prestazioni effettive del sistema. Bisogna tenere in considerazione che il test risulta però molto restrittivo e durante i test è stato possibile ottenere una corretta schedulazione di un insieme di processi con $U_P + U_s > 0{,}9$. È stato inoltre riscontrato che

anche in seguito all'arrivo di richieste aperiodiche tali da impegnare costantemente il server, l'arrivo delle richieste all'inizio della simulazione tende a peggiorare le prestazioni.

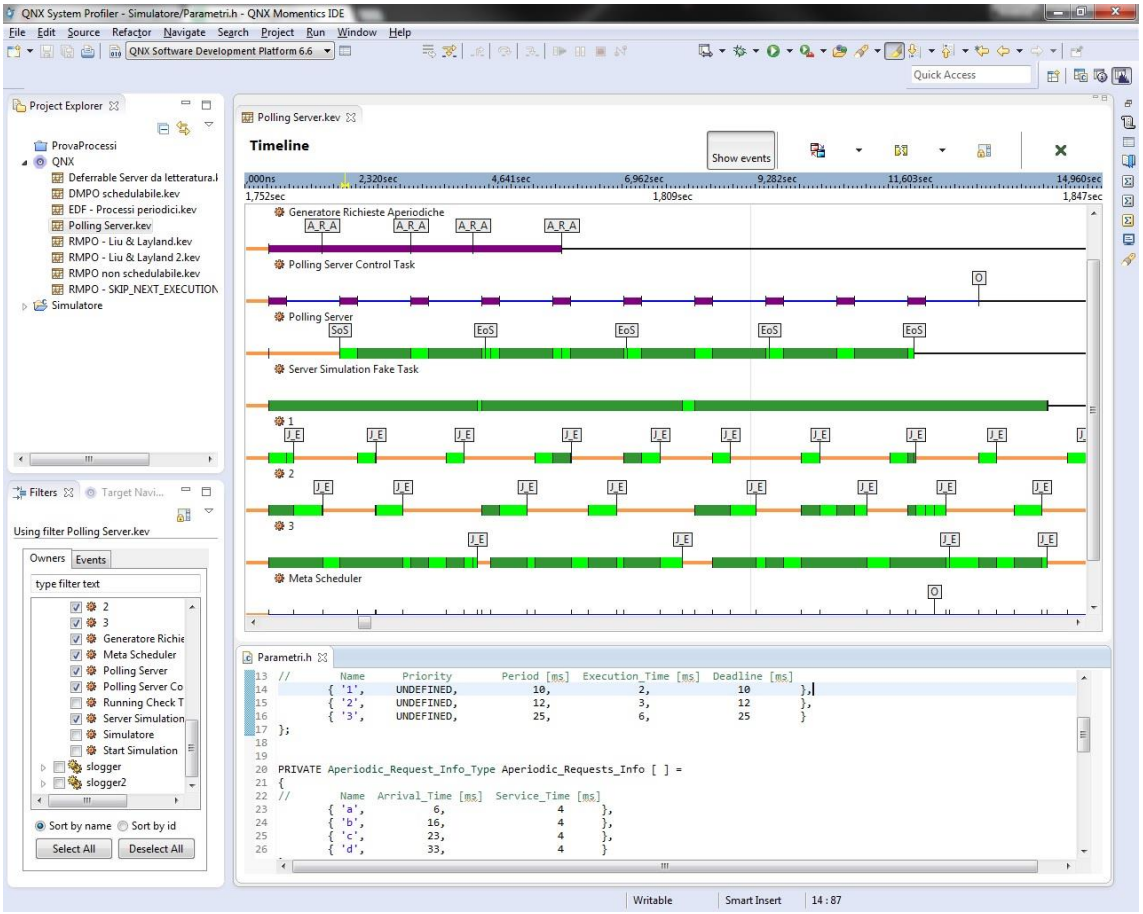

Figura 5.11 Esempio di schedulazione RMPO con gestione delle richieste aperidiche tramite polling server. C_s=2 ms, T_s=8 ms

Il thread per la gestione della capacità del polling server risulta praticamente trasparente, infatti il tempo effettivo di esecuzione (escludendo la sleep) è pari a circa un microsecondo, cioè lo 0,1% della scala su cui operano i diversi thread (i parametri dei diversi task e delle richieste aperiodiche sono specificati in millisecondi).

5.5.3 Deferrable Server

Nell'effettuare i test per la schedulabilità nel caso di utilizzo del deferrable server, bisogna tenere presente che il fattore di utilizzazione del

processore imputabile al server risulta maggiore rispetto al caso del polling server. In questo caso infatti il fattore di utilizzazione U_s risulta maggiore rispetto al rapporto tra la capacità del server C_s e il periodo di ripristino T_s.

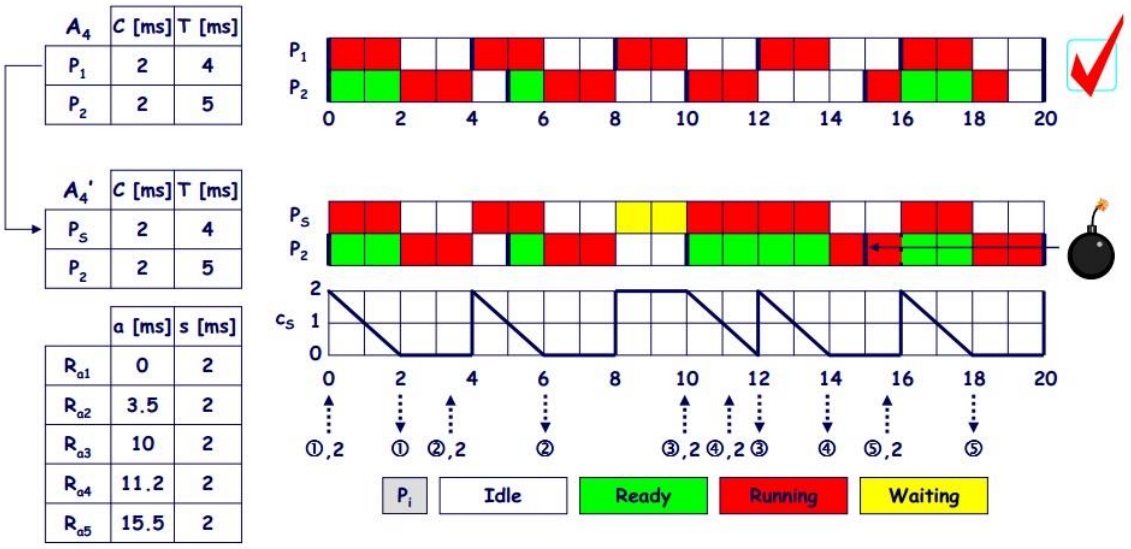

Figura 5.12 Differenza nei fattori di utilizzazione del processore tra un processo periodico e un deferrable server con capacità e periodo equivalenti [10]

Questo è dovuto alla possibilità che all'arrivo di una nuova richiesta il server (se dispone di capacità residua) può causare la preemption di un task periodico a priorità inferiore.

Per garantire la schedulabilità teorica, il fattore di utilizzazione del server deve quindi rispettare la seguente relazione:

$$U_S \leq \frac{2-(1+\frac{U_P}{N})^N}{2(1+\frac{U_P}{N})^N-1} =_{N\to\infty} \frac{2-e^{U_P}}{2e^{U_P}-1}.$$

Durante i test, il server ha comunque avuto prestazioni simili al polling server, ma permette di garantire tempi di risposta migliori alle richieste aperiodiche.

Anche in questo caso il thread per il ripristino per la capacità occupa il processore per un tempo poco inferiore al millisecondo, allo scadere di ogni periodo.

La gestione dello stato del server è ancora più efficiente. Infatti il thread esegue per un tempo pari a circa 120-150 nanosecondi al verificarsi di ogni evento (ripristino capacità, fine capacità, arrivo richiesta aperiodica, fine servizio di una richiesta).

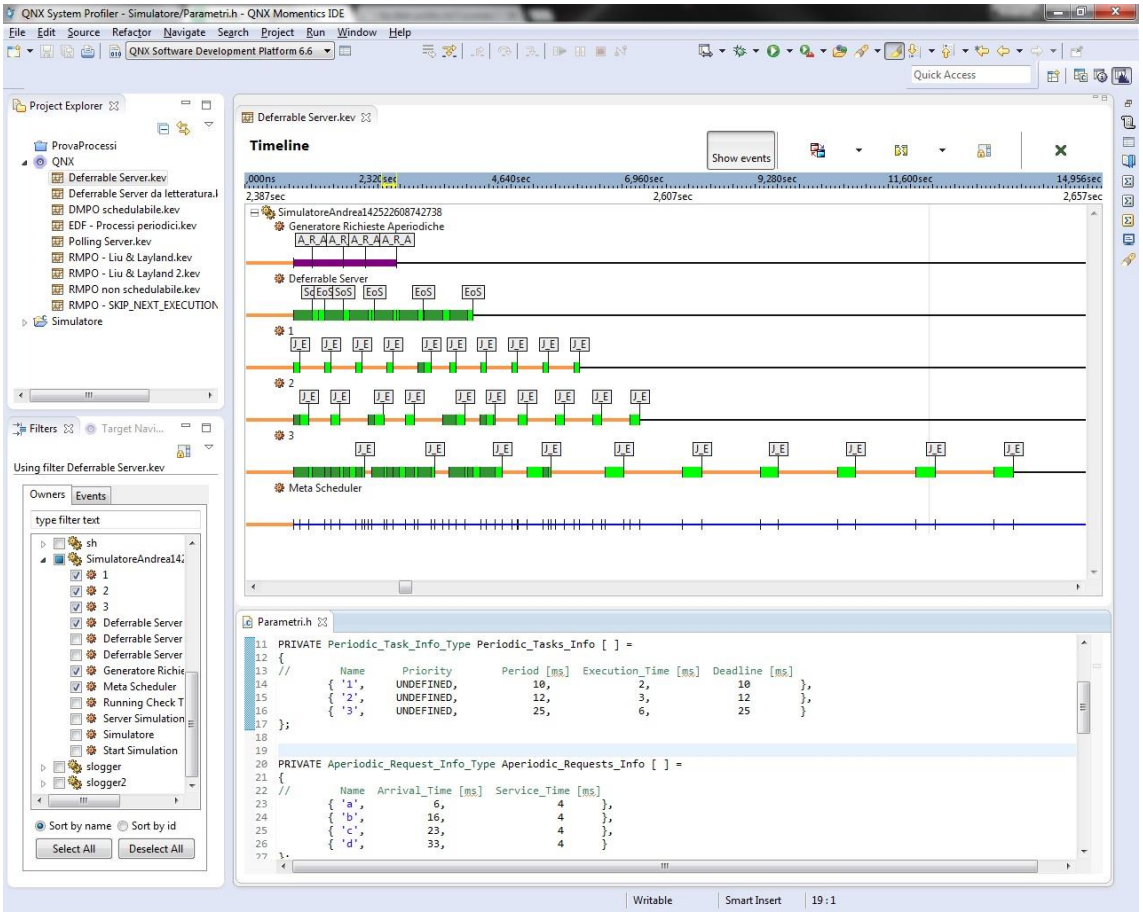

Figura 5.13 Esempio di gestione delle politiche aperiodiche tramite deferrable server. Politica di schedulazione: RMPO. C_S=2 ms, T_S=8 ms

Il simulatore, come è stato descritto nel capitolo 4, necessita di diversi altri thread per funzionare correttamente. Questi però non sono coinvolti nella simulazione real-time e quindi non si è ritenuto necessario effettuare alcuna raccolta dati specifica.

La valutazione dei risultati ottenuti verrà effettuata nel prossimo capitolo.

Capitolo 6 – Conclusioni

Lo scopo di questa tesi era sviluppare un simulatore di sistema in tempo reale con lo scopo di valutare gli strumenti messi a disposizione dal sistema operativo QNX Neutrino e le sue prestazioni in ambito real-time.

Come è stato visto finora il simulatore è stato realizzato, ma il sistema operativo, definito real-time dal produttore, si è rivelato carente nel fornire strumenti adatti alla gestione di task real-time.

Il sistema non fornisce alcun supporto alla gestione di task periodici, la loro simulazione avviene infatti attraverso l'utilizzo di semafori e timer. Una possibilità di gestione, ma con caratteristiche non conformi ai sistemi real-time, consiste nell'utilizzare processi in esecuzione costante che al termine dell'esecuzione dei loro compiti si sospendono tramite sleep, al termine della quale rieseguono il loro compito per poi tornare a sospendersi. In questo modo però l'esecuzione non è soggetta ad alcun vincolo temporale, tuttavia dalla documentazione esaminata sembra che questo sia il metodo più utilizzato nella pratica per gestire processi che debbano eseguire compiti periodici. Un'alternativa consiste nell'avvio di nuovi thread (rappresentanti i task) allo scadere di un timer periodico, ma questo, pur rispettando le tempistiche di rilascio in maniera accurata, può causare, in caso di errori, una sovrabbondanza di thread da gestire che può portare al blocco del sistema, pertanto questa pratica è fortemente sconsigliata dal produttore.

Oltre a non gestire il rilascio periodico dei processi eliminando quindi il concetto di sovraccarico (job overrun), il sistema operativo non ha, per le stesse motivazioni, alcuna nozione del concetto di deadline. Questo comporta che la generazione di un eventuale allarme o segnalazione dovuta al ritardo di un'operazione venga gestita dal processo stesso o da un altro agente (come nel caso del simulatore). Non avendo il concetto di periodo e di deadline, anche le modalità di scheduling fornite dal sistema operativo non corrispondono a quelle richieste dalla letteratura per un

sistema real-time. Per sopperire a queste mancanze è stato necessario implementare un servizio di gestione esterno per i task periodici, il meta-scheduler. L'implementazione è stata possibile utilizzando i diversi strumenti, prevalentemente presenti nella specifica POSIX, messi a disposizione dal sistema operativo, timer e semafori prevalentemente. Durante la fase di test è emerso che la gestione dei processi tramite agente esterno risulta fattibile grazie ai tempi di esecuzione estremamente ridotti ottenuti dal meta-scheduler.

La gestione delle richieste aperiodiche deve essere analizzata in maniera diversa. Infatti in questo caso non si tratta solo di servizi offerti dal sistema operativo, ma anche dell'implementazione di un sistema per la ricezione di messaggi esterni. Da questo punto di vista QNX Neutrino si è rivelato un sistema estremamente versatile e di facile utilizzo. Per quanto non sia stato utilizzato un sistema vero e proprio di ricezione di messaggi, ma una scadenza a timer per generare le diverse richieste, è stato necessario utilizzare ampiamente il sistema di messaggistica, in grado di intercettare i messaggi (pulse) lanciati dai diversi timer allo scadere. L'utilizzo della messaggistica si è rivelato efficiente e di semplice implementazione. Se questo è un punto a favore del sistema operativo, soprattutto nel caso di una gestione in background, per l'implementazione dei server a priorità statica si sono verificati problemi simili a quanto successo con i task periodici, peggiorati dal dover gestire la capacità del server. I server a priorità statica risultano a tutti gli effetti dei processi periodici per quanto riguarda la schedulazione e, non disponendo il sistema operativo di politiche di schedulazione conformi ai sistemi real-time, la gestione dev'essere effettuata tutta tramite un gestore esterno, come nel caso del meta-scheduler. Non è stato però possibile delegare la gestione dei server al meta-scheduler, trattandoli come semplici processi periodici. In questo caso infatti non sarebbe stato possibile tenere conto del consumarsi della capacità (ma soprattutto del non consumo di questa

nel caso in cui il server subisse preemption da parte di un task periodico a priorità maggiore). Per questo la simulazione, indipendentemente dai parametri dei task e del server, avviene sempre con un server a massima priorità. In realtà il sistema operativo offre una politica di scheduling (*SCHED_SPORADIC*), che può essere utilizzata per un server di gestione delle richieste aperiodiche, ma non è conforme a nessuno degli standard presenti in letteratura e si è deciso di non utilizzarla.
Anche in questo caso però, indipendentemente dal metodo implementato, il sistema operativo ha dato prova di essere estremamente efficiente, riducendo al minimo i tempi di gestione.

In conclusione, il sistema operativo QNX Neutrino non può essere definito un sistema operativo real-time secondo la definizione ufficiale, poiché non è in grado di garantire la schedulazione di processi sulla base del loro periodo e delle loro deadline, e in caso di sforamento dei limiti temporali segnalarlo.
Tuttavia dispone di strumenti temporali molto precisi (i timer) e può vantare un overhead estremamente ridotto dovuto alla sua architettura a micro-kernel. Infatti la rimozione dei servizi non richiesti permette di ottimizzare le prestazioni a seconda delle richieste del sistema specifico. Tenendo in considerazione questi due fattori, può essere considerato un ottimo punto di partenza per applicazioni integrate specifiche grazie all'elevato grado di personalizzazione possibile. È però necessario implementare ogni volta un sistema di gestione specifico sfruttando gli strumenti messi a disposizione dal sistema operativo.

6.1 Possibili sviluppi futuri

Il simulatore sviluppato offre diverse possibilità di sviluppo futuro.
La prima riguarda indubbiamente l'utilizzo del simulatore stesso in un altro sistema operativo conforme a POSIX, per verificarne le prestazioni.

Infatti tutti i servizi utilizzati, ad esclusione dei pulse, appartengono alle specifiche POSIX. Basterebbe apportare solo alcune piccole modifiche per rendere compatibile il progetto con un qualsiasi altro sistema operativo POSIX-compliant.

Rimanendo invece all'interno del sistema operativo QNX sarebbe possibile verificare la reale utilità della schedulazione tramite sporadic scheduling, utilizzabile per la gestione di un server a priorità statica, eventualmente anche a priorità di esecuzione intermedia invece che solo a priorità massima.

Sempre nell'ambito dei server a priorità statica potrebbe essere interessante valutare se sia possibile intercettare gli eventi generati dal sistema operativo al cambio di stato di un thread. In questo modo dovrebbe essere possibile creare dei server a priorità intermedia conformi alla letteratura. Una volta verificatasi questa possibilità sarebbe allora possibile l'utilizzo di server a priorità dinamica, ma come anche nel caso precedente, bisognerebbe valutare gli eventuali benefici sulla base dell'overhead generato dalla gestione del server.

Bibliografia

[1] John A. Stankovic and K. Ramaritham, editors. *Tutorial on Hard Real-Time Systems*.

[2] *QNX Neutrino RTOS System Architecture*

[3] http://www.qnx.com/

[4] http://www.qnx.com/developers/docs/660/index.jsp

[5] http://community.qnx.com/sf/sfmain/do/home

[6] Liu, C. L.; Layland, J. (1973), *Scheduling algorithms for multiprogramming in a hard real-time environment*

[7] Baruah et al. (1990), *Processor demand criterion*

Get Programming with the QNX Neutrino RTOS

A Quickstart Guide

Immagini

Vengono qui riportate le fonti delle immagini utilizzate. Le immagini senza fonte sono state acquisite direttamente tramite screenshot durante lo sviluppo della tesi.

[1] http://www.qnx.com/solutions/industries/automotive/

[2] http://www.embedded.com/electronics-blogs/beginner-s-corner/4023859/Introduction-to-Real-Time

[3] http://blog.softwareinsider.org/2011/06/20/mondays-musings-real-time-versus-right-time-and-the-dawn-of-engagement-apps/

[4] http://lia.deis.unibo.it/Courses/SistRT/contenuti/2%20Materiale%20didattico/1%20Introduzione.pdf

[5] QNX Neutrino RTOS System Architecture

[6] Get Programming with the QNX Neutrino RTOS

[7] http://www.python-course.eu/threads.php

[8] http://community.qnx.com/sf/sfmain/do/home

[9] http://lia.deis.unibo.it/Courses/SistRT/contenuti/2%20Materiale%20didattico/2%20Scheduling%20di%20processi%20hard%20real-time.pdf

[10] http://lia.deis.unibo.it/Courses/SistRT/contenuti/2%20Materiale%20didattico/3%20Scheduling%20di%20processi%20soft%20real-time.pdf