* Introduzione
* Java realtime: specifica e implementazione da parte di Sun
* Utility per java realtime
  + BusyWait
  + Logging
* Uno scheduler EDF per java realtime
* La gestione dei deadline miss in Java Realtime
* Una nuova strategia di gestione dei deadline miss: la politica skip
* Un ulteriore miglioramento: la politica skipStop
* Conclusioni e sviluppi futuri

Sommario

[Introduzione 3](#_Toc298170535)

[Limiti di Java come piattaforma per sistemi in tempo reale 3](#_Toc298170536)

[Lazy initialization 4](#_Toc298170537)

[Just In Time Compilation 4](#_Toc298170538)

[Garbage Collection 5](#_Toc298170539)

[Scheduling 6](#_Toc298170540)

[Sincronismo 6](#_Toc298170541)

[Java realtime: specifica e implementazione da parte di Sun 7](#_Toc298170542)

[La specifica Java Real Time 8](#_Toc298170543)

[Storia 9](#_Toc298170544)

[Principi guida 9](#_Toc298170545)

[Gestione della memoria e garbage collection 10](#_Toc298170546)

[Tempo ed orologi 12](#_Toc298170547)

[Scheduling 14](#_Toc298170548)

[Parametri di scheduling 15](#_Toc298170549)

[RealtimeThread 18](#_Toc298170550)

[Eventi asincroni 20](#_Toc298170551)

[Trasferimento asincrono di controllo 22](#_Toc298170552)

[Sincronizzazione 24](#_Toc298170553)

[L’implementazione di Sun 28](#_Toc298170554)

[Inizializzazione e compilazione 28](#_Toc298170555)

[Garbage Collector 29](#_Toc298170556)

[Prime estensioni a Java Real-Time 31](#_Toc298170557)

[BusyWait 32](#_Toc298170558)

[Logging 35](#_Toc298170559)

[Periodic Thread 38](#_Toc298170560)

[Uno scheduler EDF per java Realtime 39](#_Toc298170561)

[L’implementazione dello scheduler 41](#_Toc298170562)

[La classe EDFScheduler 42](#_Toc298170563)

[EDFSchedulingParameters 44](#_Toc298170564)

[Modifiche a PeriodicThread 44](#_Toc298170565)

[Risultati sperimentali 45](#_Toc298170566)

[La gestione dei deadline miss in Java Realtime 46](#_Toc298170567)

[Parametri associati ad un processo periodico 47](#_Toc298170568)

[Comportamento in assenza di handler 47](#_Toc298170569)

[Comportamento in presenza di un handler associato al thread 48](#_Toc298170570)

[Criticità della Politica ASAP 49](#_Toc298170571)

[Una nuova strategia di gestione dei deadline miss: la politica SKIP 50](#_Toc298170572)

[Presentazione della politica SKIP 50](#_Toc298170573)

[L’implementazione della politica SKIP. 51](#_Toc298170574)

[L’interfaccia IPendigJobManager 52](#_Toc298170575)

[Modifiche alla classe PeriodicThread 53](#_Toc298170576)

[DeadlineMissedHandler 54](#_Toc298170577)

[SkipPolicyHandler 54](#_Toc298170578)

[Un ulteriore miglioramento: la politica skipStop 56](#_Toc298170579)

[La politica Stop 56](#_Toc298170580)

[BusyWait 57](#_Toc298170581)

[InterrumpiblePeriodicThread 57](#_Toc298170582)

[StopPolicyHandler 58](#_Toc298170583)

[Le politiche a soglia e la politica SkipStop 59](#_Toc298170584)

[Implementazione della politica 60](#_Toc298170585)

[ThresholdPolicyHandler 61](#_Toc298170586)

[SkipStopThresholdPolicyHandler 62](#_Toc298170587)

[Conclusioni e sviluppi futuri 63](#_Toc298170588)

[Sviluppi Futuri 65](#_Toc298170589)

[Bibliografia 67](#_Toc298170590)

[Glossario 67](#_Toc298170591)

Introduzione

I sistemi in tempo reale sono dei particolari sistemi che devono garantire una risposta entro determinati vincoli temporali. Gli esempi più lampanti sono i sistemi di controllo di centrali elettriche, quelli montati su aerei da trasporto o utilizzati per gestire gli scambi ferroviari. Le applicazioni in tempo reale, tuttavia, si stanno sempre più diffondendo anche in altri ambiti, quali l’automotive, l’automazione industriale, la gestione di video o di fonti multimediali.

Se è vero che l’hardware riveste comunque grande importanza in questo ambito, bisogna riconoscere come da solo non sia sufficiente a fornire le garanzie necessarie al sistema. I sistemi genaral purpose, infatti, sono, in media estremamente efficienti, ma, a causa di un approccio best-effort, non possono garantire il rispetto dei vincoli imposti nella totalità dei casi. Emerge, allora, la necessità di una piattaforma software che agisca in modo da garantire il sistema anche nel caso peggiore, spesso a scapito della pura efficienza.

Una peculiarità del software che deve eseguire su un sistema in tempo reale è che i processi in esso presenti hanno carattere periodico: questi processi eseguono lo stesso set di istruzioni ad intervalli regolari di tempo; ogni esecuzione ciclica prende il nome di job.

Inoltre, la violazione dei vincoli temporali può essere accettabile o meno a seconda del sistema. In ogni caso, è logico prevedere che il sistema reagisca in un qualche modo a fronte di questo genere di eventi.

Occorre notare come, la crescente complessità dei sistemi in tempo reale rende necessaria l’introduzione di una nuova generazione di linguaggi più strutturati. Il linguaggio Java è diffuso in ambiti molto diversificati, ma nella sua versione standard è sarebbe utilizzabile nei sistemi in tempo reale per una molteplicità di fattori che rendono estremamente instabile il tempo di esecuzione di una applicazione Java. Per ovviare questi problemi la piattaforma standard è stata modificata ed estesa, in modo da raggiungere l’obiettivo di avere un sistema Java in grado di fornire una solida base per lo sviluppo di applicazioni in tempo reale.

In questa tesi si è analizzata la specifica Real-Time Specification for Java (RTSJ) e la sua implementazione da parte di Sun-Oracle Real-Time Java System (RTJS) su sistema operativo Solaris 10.9. Su questa piattaforma sono stati sviluppati alcuni componenti in grado di migliorare le performance generali di un’applicazione in tempo reale.

Nel primo capitolo si analizzeranno tutti i fattori che non permettono di utilizzare la piattaforma Java standard nei sistemi in tempo reale

Il secondo capitolo parla di come queste criticità sono state affrontate dalla specifica Java Real-Time (Real-Time Specification for Java – RTSJ) e dalla sua implementazione da parte di Sun-Oracle (Real Time Java System - RTJS).

Il terzo capitolo tratta delle prime estensioni al sistema rese necessarie per carpire il suo funzionamento e per simulare il comportamento di un’applicazione in tempo reale: si tratta di un modulo che realizza una busyWait della durata desiderata in grado di calibrarsi sulle potenzialità computazionali della piattaforma hardware sottostante; di un modulo di logging che permette di monitorare le attività compiute dalle entità del sistema senza incorrere in rallentamenti dovuti alla sincronizzazione su risorse condivise; della modellazione di un thread periodico, che esegue ciclicamente delle esecuzioni di durata prestabilita

Il quarto capitolo affronta la tematica dello scheduling in Java Real-Time, mostrando il comportamento dello scheduler di default basato su priorità statica. Si mostra la realizzazione di uno scheduler EDF, che permette di avere delle performance migliori rispetto a quelle ottenibili con la schedulazione a priorità statica.

Il quinto capitolo affronta la tematica della violazione, da parte dei threads dei vincoli temporali specificati. Si mostra la politica adottata di default dal sistema evidenziandone le criticità maggiori: una tendenza a rincorrere le release perse, con la conseguenza di peggiorare il sovraccarico del sistema e l’assenza di intervento in caso di un’esecuzione che si protrae significativamente nel tempo, con il rischio di bloccare l’intero sistema.

Il sesto capitolo mostra l’implementazione di una politica di gestione della violazione dei vincoli temporali (deadline): la politica skip. Questa politica che non venga schedulato nessun altro job del thread che ha violato la deadline nel periodo in cui termina l’esecuzione del job che non ha rispettato la deadline stessa, in modo da alleggerire il carico computazionale del sistema.

Il settimo capitolo mostra la realizzazione di una politica che prevede di interrompere l’esecuzione di un job che viola consecutivamente un numero di deadline pari ad una soglia specificata, risolvendo la criticità legata all’assenza misure in grado di bloccare un thread che, con un’esecuzione troppo protratta nel tempo, rischia di bloccare l’intero sistema.

L’ottavo capitolo contiene le conclusioni ed i possibili sviluppi futuri nel sistema preso in esame.

Limiti di Java come piattaforma per sistemi in tempo reale

La piattaforma Java, nelle sue varie versioni è estremamente tanto da essere impiegata in ambiti molto diversificati, dalla gestione di transazione sicure a quella di flussi multimediali, senza dimenticare le piattaforme mobili ed i sistemi embedded.

La diffusione di Java nei sistemi real-time è stata fortemente limitata dal fatto che i tempi di esecuzione di un’applicazione Java sono difficilmente predicibili.

Anche qualora la Java Virtual Machine si appoggi su un sistema operativo adatto ai sistemi in tempo reale la predicibilità dei tempi di esecuzione di un programma Java è fortemente limitata da fattori interni alla virtual machine. Questi fattori sono principalmente collegabili alle attività interne alla virtual machine stessa, attività che possono causare la sospensione temporanea di un processo attivo. In questa sezione si presentano le principali cause che causano significative variazioni nei tempi di esecuzione per la stessa applicazione (jitter).

# Lazy initialization

La Java VM normalmente inizializza una classe quando il programma tenta per la prima volta di ottenere un'istanza di essa, o quando tenta di utilizzare un metodo statico o una variabile statica della classe stessa. Quando la classe è inizializzata, le sue variabili statiche sono inizializzate di conseguenza. Questa attività, ovviamente, richiede tempi di esecuzione che possono interferire con l'esecuzione di codice critico per l'applicazione in tempo reale. Poiché non è possibile prevedere quando le classi saranno inizializzate, non è possibile accertarsi di quando questo accadrà, né se la quantità di ritardo introdotto sarà accettabile.

# Just In Time Compilation

Just in Time (JIT) è la modalità di compilazione standard di una macchina virtuale Java. La compilazione JIT seleziona i metodi che devono essere compilate sulla base della frequenza della loro esecuzione. La compilazione di un metodo viene attivata in due situazioni:

Quando una chiamata interpretata viene eseguita, una delle prime operazioni effettuate dalla macchina virtuale è quella di incrementare il contatore di invocazione del metodo chiamato. Se il valore del contatore raggiunge la soglia di compilazione, il metodo viene compilato prima di essere eseguito.

Il numero di iterazioni di eventuali cicli all’interno del metodo è monitorato, a sua volta, da un contatore. La compilazione viene attivata anche quando questo contatore raggiunge un valore soglia. Questo processo è noto anche come On – Stack Replacement(OSR).

Il momento in cui avviene la compilazione è, quindi, totalmente sotto il controllo della macchina virtuale e non può essere previsto nella totalità dei casi. Per esempio, se una sezione di codice critica viene eseguita raramente, il contatore può raggiungere la soglia che fa scattare la compilazione dopo un tempo molto lungo.

Tuttavia non c'è quindi modo di sapere quando la compilazione si verificherà e questa incertezza non è accettabile in una applicazione real-time.

Tramite un’opzione si può istruire la Virtual Machine affinché esegua la compilazione in background. In questa modalità, non si verifica un’interruzione per permettere la compilazione del metodo che sta per eseguire: il thread esegue il metodo in modalità interpretata e la compilazione viene differita. Bisogna tuttavia sottolineare come, se il carico sulla cpu è elevato, la compilazione rischia di essere differita per un tempo indefinito, facendo sì che l’applicazione venga sempre eseguita in modalità interpretata, con grandissima inefficienza.

# Garbage Collection

Il servizio di garbage collection all’interno della Virtual Machine consente di recuperare aree di memoria precedentemente occupate, ma non più accedibili e di renderle, quindi, utilizzabili per l’allocazione di nuovi oggetti. La strategia per localizzare queste aree di memoria da considerare come riutilizzabili all’interno del java heap è, appunto, quella di individuare gli oggetti che non sono più accedibili tramite i thread attivi presenti all’interno della macchina virtuale. Per fare ciò si possono adottare varie metodologie, la maggior parte delle quali ha bisogno di lavorare su un’istantanea del sistema. Per questo motivo, nella Java Virtual Machine standard tutte le esecuzioni vengono fermate quando entra in azione il garbage collector (stop the word).

Appare evidente come l’attività di garbage collection, che prende il via ogni qual volta la memoria libera nell’heap scende sotto una determinata soglia, sia una forte fonte di indeterminismo. E’ infatti difficile prevedere quando il garbage collector entrerà in esecuzione, dal momento ciò è legato quota di memoria libera nell’heap, valore dell’andamento difficile da prevedere, specie considerando la pluralità di applicazioni che spesso sono in esecuzione sulla stessa virtual machine. Altro fattore di difficile stima è il tempo impiegato dal garbage collector per adempiere al suo compito: questo è infatti fortemente influenzato da fattori quali la quantità di memoria da riciclare ed altri fattori legati ai singoli algoritmi di garbage collecting, quali, ad esempio, la profondità e il numero di connessioni dei percorsi che partono dagli oggetti radice, o al numero di oggetti presenti nel sistema.

# Scheduling

Java mette a disposizione dello sviluppatore alcuni metodi della classe Thread che consentono di influenzare il comportamento dello scheduler interno alla virtual machine. Sono presenti dieci livelli di priorità da associare ai thread. Inoltre, tramite il metodo yeld è possibile posizionare il thread in fondo alla coda dei processi pronti relativa alla priorità del thread stesso.

Tuttavia, nell’ottica di sviluppo di applicazioni in tempo reale, questi strumenti non sono soddisfacenti. In particolare non è garantito che, tra i processi pronti, sia sempre in esecuzione quello a priorità maggiore. Questo rende difficile stabilire quanto tempo un processo possa impiegare per la sua esecuzione, dal momento che un thread può essere bloccato anche da un thread con priorità inferiore. Inoltre la virtual machine è libera di usare politiche di time-slicing per processi che eseguono alla stessa priorità. Questo fa sì che il tempo di esecuzione di un thread possa venire artificiosamente dilatato a causa della frammentazione dell’esecuzione stessa in più slice intervallati tra loro.

# Sincronismo

Gli strumenti che Java mette a disposizione per sincronizzare i Thread sono sostanzialmente basati su lock a mutua esclusione. Seppur dalla comprovata validità nel caso generale, queste soluzioni non sono da sole soddisfacenti per i sistemi in tempo reale, in quanto prone a fenomeni indesiderati quali l’inversione incontrollata di priorità di cui si mostra un esempio in figura.



Nell’ esempio quando il processo a priorità maggiore tenta di accedere alla risorsa R1 si sospende perché questa è già occupata dal processo Pl. Tuttavia, non è accettabile che il processo Pm eserciti indirettamente preemption su un processo a priorità maggiore come Ph, pur non condividendo con esso nessuna risorsa. Per evitare questo genere di fenomeni sono stati sviluppati in letteratura una serie di protocolli (tra i più noti si citano priority inheritance e priority ceiling) che la versione standard di Java non supporta.

Java real-time: specifica e implementazione da parte di Sun-Oracle

Questo capitolo ha lo scopo di fornire un introduzione a java real-time. In primo luogo viene analizzata la specifica RTSJ (Real–Time Specification for Java), successivamente vengono mostrate le caratteristiche distintive dell’implementazione della specifica utilizzata in questa tesi ossia Java RTJS(Java Real-Time System) sviluppata da Sun-Oracle.

# La specifica Java Real-Time

Il crescente sviluppo di applicazioni in tempo reale ha richiesto l’introduzione di linguaggi di programmazione di più alto livello anche per questi sistemi, con il fine di poter gestire meglio la loro crescente complessità.

Java, considerando anche la sua larga diffusione nei sistemi embedded, è uno dei linguaggi più appetibili per la nuova generazione di sistemi in tempo reale. Non essendo nato per questi peculiari scenari applicativi, tuttavia, occorre apportare pesanti modifiche alla sua macchina virtuale affinché sia efficacemente utilizzabile in questi sistemi.

La specifica RTSJ (Real-Time Specification for Java), successivamente implementata da vari vendors, propone estensioni e modifiche alla Java Virtual Machine standard per renderla adatta ad sistemi in tempo reale, oltre ad un insieme di classi contenute nel package javax.realtime di aiuto allo sviluppatore per la creazione di applicazioni in tempo reale.

Nella sezione storia si mostrano le tappe che hanno portato dalla prima stesura della specifica alle attuali implementazioni. Successivamente si mostreranno i principi che stanno alla base della realizzazione della specifica. Si procederà quindi ad illustrare le problematiche affrontate nella specifica e le soluzioni adottate con particolare riferimento alle classi del package javax.realtime da essa introdotte. Si partirà con l’illustrare la gestione della memoria, per poi passare alla rappresentazione del tempo. Si entrerà quindi nel cuore della specifica, ossia nei meccanismi di scheduling, iniziando dai parametri di scheduling, per passare alla descrizione dei thread real-time, alla gestione di eventi asincroni, al trasferimento asincrono di controllo e finire con l’analizzare le tematiche di sincronizzazione ed i due protocolli previsti dalla specifica: priority inheritance e priority ceiling.

## Storia

Alla fine degli anni novanta il NIST (National Institute for Standards and Tecnologies), un'agenzia del governo degli Stati Uniti d'America che si occupa della promozione delle tecnologie attraverso un lavoro coordinato con l'industria per sviluppare standard, tecnologie e metodologie allo scopo di la produzione e il commercio, ha coordinato un gruppo di esperti con lo scopo di raccogliere le linee guida ed un primo insieme di requisiti per le estensioni real-time di Java.

Successivamente, la prima release della specifica RTSJ (Real Time Specification for Java) venne pubblicata nel 2002.

La seconda versione della specifica viene rilasciata nel giugno del 2005.

L’implementazione di IBM, parte di WebSphere, è stata rilasciata nel 2006

La prima versione di Sun venne lanciata nel 2005. La versione attuale, la 2.2, risale al 2009.

## Principi guida

La specifica RTJS si ispira ai seguenti principi:

**Applicabilità a particolari ambienti:** La specifica non include caratteristiche che ne limitino l’utilizzo ad ambienti particolari come Java Micro Edition o particolari versioni del JDK, al contrario lo scopo degli implementatori deve essere quello di allargare il più possibile l’applicabilità dei loro prodotti. La specifica si riferisce ad un ambiente monoprocessore, sebbene non preclude l’esecuzione in un ambiente con più unità di elaborazione non offre funzionalità di supporto in tal senso quali il controllo dell’allocazione dei thread sulle cpu.

**Compatibilità**  La specifica non impedisce l’esecuzione di programmi Java non real-time sulla nuova virtual machine: su di essa devono poter essere contemporaneamente presenti sia applicazioni in tempo reale sia applicazioni java standard.

**Write once, run everywere**  Pur riconoscendo l’importanza dell’approccio write once, run everywere, data la difficoltà di mantenerlo su sistemi real-time, si è deciso di non sacrificare la predicibilità dell’esecuzione in favore della portabilità

**Estendibilità** La specifica deve permettere l’utilizzo delle tecniche più conosciute per la gestione dei sistemi real-time. Al contempo, deve permettere la futura implementazione di nuove tecniche e nuovi algoritmi

**Nessuna estensione sintattica** Allo scopo di semplificare il lavoro degli sviluppatori non sono stati introdotte nuove keyword e non è stata fatta alcuna estensione sintattica al linguaggio Java.

**Libertà di implementazione** Con lo scopo di permettere alle aziende implementatrici di venire incontro alle specifiche esigenze dei propri clienti,la specifica riconosce che le varie implementazioni possono divergere in un gran numero di decisioni, quali, ad esempio, l’uso di diversi algoritmi per eseguire lo stesso compito, o trade – off tra l’occupazione di memoria e velocità di esecuzione. In ogni caso, la specifica si propone di non vincolare mai all’uso di uno specifico algoritmo, ma si limita a descrivere i requisiti semantici che le operazioni devono rispettare.

## Gestione della memoria e garbage collection

Molti sistemi in tempo reale hanno a disposizione solo una quantità limitata di memoria, ciò è dovuto a considerazioni relative a problematiche di costo o a vincoli di natura fisica (dimensioni, potenza, peso…). Inoltre, essendo disponibili tipi diversi di memoria (con differenti caratteristiche d'accesso) può essere necessario far sì che certi dati risiedano esclusivamente in una certa zona di memoria. In certi scenari è quindi necessario controllare come la memoria viene allocata in modo da poterla usare efficientemente.

Java real-time permette di definire delle aree di memoria, oltre all’heap tradizionale, nelle quali allocare oggetti. Quando si entra in una area di memoria tutti gli oggetti vengono allocati all'interno di quest’ultima. Queste aree di memoria sono il già menzionato heap, la ScopedMemory e l’ImmortalMemory.

La zona di memoria ScopedMemory è stata concepita come una zona di memoria con un tempo di vita ben limitato: un reference counter è associato ad ogni ScopedMemory e tiene traccia di quante entità real-time la stanno attualmente utilizzando; quando il suo valore passa da uno a zero tutti gli oggetti residenti nella ScopedMemory vengono distrutti e la memoria viene liberata. Java real - time prevede due tipi di ScopedMemory: VtMemory, dove le allocazioni di memoria richiedono un tempo variabile e LTMemory, dove queste richiedono un tempo proporzionale alla dimensione in byte dell’oggetto da allocare.

L’ ImmortalMemory è una zona di memoria che non è mai coinvolta nelle operazioni di garbage collecting. Questa zona di memoria deve il suo nome al fatto che viene liberata solamente quando termina l’esecuzione dell’applicazione. Esiste una sola zona di ImmortalMemory nel sistema che, quindi, è condivisa da tutti i thread dell’applicazione. La gestione di questa zona di memoria è affidata all’utente, che deve provvedere a finalizzare gli oggetti non più utilizzati in modo da liberare la memoria da essi occupati.

La classe MemoryParameters fa parte di quell'insieme di classi che definiscono i parametri di tutti gli oggetti schedulabili (a tal fine si rimanda la sezione sullo scheduling). È deputata a specificare, per un oggetto schedulabile

* la quantità massima di memoria che l'oggetto può consumare in una zona di memoria
* la quantità massima di memoria che può essere occupata nell'ImmortalMemory
* un limite di allocazione nell'heap (in termini di byte al secondo)

I MemoryParameters possono essere utilizzati dallo scheduler nel controllo di feseabylity, o dal garbage collector per ottimizzare la sua attività.

Per ottimizzare il sistema in caso di memorie con differenti caratteristiche può essere necessario lasciare alo sviluppatore la possibilità di decidere dove gli oggetti debbono essere memorizzati. La specifica prevede un’apposita estensione dei tre nuovi tipi di MemoryArea (LinearTime, VariableTime e ImmortalMemory) chiamati rispettivamente LTPhysicalMemory, VTPhysicalMemory, ImmortalPhysicalMemory. La gestione dell’accesso in lettura e scrittura alle zone di memoria è affidato ad un PhysicalMemoryManager. La specifica fornisce, attraverso le classi ImmortalPhysicalMemory e RawMemoryFloatAccess, dei metodi per leggere e scrivere variabili Java o array di dati primitivi nelle zone di memoria fisica. La figura sottostante mostra le classi coinvolte nella gestione diretta della memoria

Per quanto riguarda il garbage collector la specifica non impone nessuna strategia o nessun algoritmo in particolare. Si limita a prevedere che il garbage collector del sistema real - time, qualunque esso sia, abbia una funzionalità tale da non compromettere la predicibilità dell'esecuzione dei processi real - time. Prevede inoltre l'utilizzo di una classe di supporto (SizeEstimator) la quale ha lo scopo di prevedere quanto spazio sarà necessario per garantire l'allocazione di un insieme di oggetti.

## Tempo ed orologi

In un sistema real - time, misurare con precisione, efficacia ed efficienza il tempo è fondamentale. La specifica Java real - time viene incontro allo sviluppatore fornendo una serie di classi di supporto per esprimere misure temporali. In figura vengono mostrate le principali classi del package javax.realtime coinvolte nel misurare il trascorrere del tempo.

La classe Clock è la classe base per tutti gli orologi presenti nel sistema: in ogni sistema, infatti, è sempre presente almeno un orologio: l'orologio di sistema con precisione al nanosecondo, tuttavia, per ragioni di efficienza, la specifica prevede che possono esistere anche altri orologi con precisioni differenti. Ogni misura di tempo, sia relativa che assoluta, è associata ad un orologio. Se non specificato diversamente, l'orologio associato ad essa è quello di sistema.

La classe HighResolutionTime è la classe madre di tutte le misure di tempo. La rappresentazione del tempo è incapsulata in due valori: un long per rappresentare i millisecondi ed un int per rappresentare i nanosecondi. Questa classe, in sostanza, fornisce le funzionalità per comparare misure di tempo e per estrarre da esse i valori di millisecondi e nanosecondi.

La classe AbsoluteTime è deputata a rappresentare uno specifico istante nell'asse temporale, ossia un riferimento temporale assoluto. Oltre alle funzionalità ereditate dalla classe madre, questa classe espone una serie di metodi che permettono di sommare o di sottrarre ad un AbsoluteTime sia un RelativeTime che un altro AbsoluteTime, ottenendo come risultato un RelativeTime o un AbsoluteTime.

La classe RelativeTime serve per esprimere un tempo relativo. Di conseguenza viene usata per esprimere periodi o differenze tra istanti di tempo. Anche la classe RelativeTime, oltre ai metodi ereditati dalla classe madre, espone una serie di metodi che consentono di sommare o di sottrarre ad essa altre misure temporali.

La classe RationalTime, deprecata dalla versione1.0.1 della specifica, era stata originariamente pensata per esprimere frequenze: infatti era possibile inizializzarla con un valore che esprimesse il numero di iterazioni per secondo.

## Scheduling

In letteratura con il termine scheduling si intende l'attività di ordinare l'esecuzione dei processi in modo da ottenere uno sfruttamento efficiente ed efficace delle risorse hardware. La specifica Java real - time prevede in questo ambito delle funzionalità obbligatorie e delle funzionalità accessorie.

Tra le prime rientrano la realizzazione di uno scheduler basato su priorità (che, assegnata una priorità ad un thread, ponga in esecuzione il thread a priorità maggiore tra quelli pronti); il controllo del rispetto da parte del thread delle deadline specificate, se presenti, ed il protocollo di priority inheritance per la gestione dell'accesso a sezioni di codice mutuamente esclusive.

Tra le funzionalità accessorie rientrano il controllo dei tempi di esecuzione effettiva dei thread (cost enforcement) ed il protocollo priority ceiling per la gestione delle zone di codice con accesso mutuamente esclusivo.

Nel corso di questo paragrafo si illustreranno tutte le classi e tutte le funzionalità presenti nella specifica partendo dai parametri che permettono di caratterizzare i flussi di esecuzione real - time; successivamente si analizzeranno i due tipi di oggetti schedulabili partendo dai thread real - time per concludere con i gestori di eventi asincroni; si passerà quindi ad un'analisi sul trasferimento asincrono di controllo, ossia la possibilità offerta sistema di interrompere in modo asincrono l'esecuzione di un oggetto schedulabile per fargli eseguire dell'altro codice; concludono il paragrafo le funzionalità che regolano l’accesso a sezioni ad accesso mutuamente esclusivo.

### Parametri di scheduling

L’attività di scheduling di java real-time si basa sull’idea di associare ad ogni oggetto schedulabile una serie di parametri in grado di caratterizzarne l’esecuzione. Più oggetti schedulabili possono, quindi, condividere gli stessi parametri. Quando tali parametri vengono modificati o un oggetto fa riferimento ad una nuova istanza di questi parametri, la modifica deve immediatamente propagarsi nel sistema. Ad esempio, se si associa ad un processo real-time una nuova istanza di PriorityParameters il processo dovrà immediatamente cambiare priorità; la stessa cosa deve accadere anche qualora si modifichi il valore numerico deputato ad indicare la priorità all’interno dei PriorityParameters stessi.

Nella figura sottostante si mostrano i parametri coinvolti nell'attività di scheduling , la classe astratta Scheduler e PriorityScheduler, ossia l'unico scheduler previsto dalla specifica.

La classe ReleaseParameters è una classe astratta che modella i generici parametri di release di un oggetto schedulabile: permette di indicare la deadline relativa (mediante un RelativeTime) di un particolare oggetto schedulabile e un'eventuale handler da richiamare qualora questa non venga rispettata. Analogamente è possibile specificare il costo (ossia il tempo massimo di esecuzione) e un handler da richiamare qualora il processo occupi l’unità di elaborazione per un tempo maggiore rispetto a quello dichiarato. Nel caso in cui il sistema di cost enforcement non sia implementato, questi ultimi due valori non avranno alcun effetto sul funzionamento del sistema.

La classe PeriodicParameters estende ReleaseParameters e modella i parametri di release di un processo periodico. E’ possibile specificare il periodo (con un relative time) di un oggetto schedulabile; si può anche indicare un tempo di StartTime in modo relativo o assoluto (se il tempo specificato è anteriore a quello presente il processo viene lanciato immediatamente).

La classe AperiodicParameters modella un processo aperiodico, consente di accordare le release in una coda, di dimensione settabile, e di attuare politiche differenti nel caso in cui arrivi una release con coda piena: è possibile ignorare la release, lanciare un'eccezione, oppure aumentare la dimensione della coda ed accettare anche quest'ultima release.

La classe SporadicParameters modella un processo sporadico, ossia un processo aperiodico tale che due release non possano presentarsi con distanza inferiore ad un tempo chiamato minimum interarrival time (MIT). Oltre a permettere di specificare il MIT, è possibile applicare politiche differenti nel caso si presentino al sistema due release a distanza inferiore del MIT: è possibile ignorare la seconda release, eseguirla solo dopo il MIT, o lanciare un'eccezione.

La classe PriorityParameters estende la classe astratta SchedulingParameters ed esprime i parametri relativi all'unico protocollo di scheduling previsto dalla specifica: lo scheduling basato su priorità statica con almeno 28 differenti livelli di priorità. Consente quindi di esprimere un valore numerico di priorità: i thread più prioritari sono quelli con valore numerico di priorità più alta.

La classe ImportanceParameters estende PriorityParameters introducendo un valore di importanza: l'idea alla base di questa classe è che il valore di importanza possa essere utilizzato dallo scheduler per dirimere casi di “tie” tra oggetti schedulabili con la stessa priorità: in questo caso il sistema può sfruttare il valore di importanza suggerito dall'utente per mettere in esecuzione il processo con importanza maggiore tra quelli dotati della stessa priorità.

È possibile raggruppare più oggetti schedulabili in un unico gruppo. È possibile specificare, per ogni gruppo, una deadline ed un costo (ossia il tempo massimo di cpu a disposizione dei membri del gruppo) con i relativi gestori da invocare qualora questi non vengano rispettati. Se i membri non terminano tutti la propria esecuzione entro la deadline specificata per il gruppo viene invocato il relativo handler. Analogamente, se la somma dei tempi di esecuzione dei membri del gruppo supera il costo specificato, viene rilasciato l’ handler specificato per il superamento di costo. Fanno parte di un determinato gruppo tutti gli oggetti schedulabili che condividono la stessa istanza della classe ProcessingGroupParameters, nella quale si possono specificare la deadline del gruppo, il costo massimo complessivo di tutti i processi del gruppo ed i relativi handler da invocare qualora questi parametri non siano rispettati. Dal momento che l’appartenenza ad un gruppo è legata solamente alla condivisione della stessa istanza di ProcessingGroupParameters e che tale riferimento è modificabile in qualsiasi momento ne segue che l’appartenenza ad un gruppo di processi è dinamica ed il sistema deve potersi adattare all’inserimento ed alla rimozione di partecipanti al gruppo di processi durante l’esecuzione.

La classe MemoryParameters,già illustrata nella sezione dedicata alla gestione della memoria, permette di indicare il consumo di memoria dell’oggetto schedulabile in termini di

* quantità massima di memoria che l'oggetto può consumare in una zona di memoria
* quantità massima di memoria che può essere occupata nell'ImmortalMemory
* limite di allocazione nell'heap (in termini di byte al secondo)

Qualora l’oggetto schedulabile violi quanto dichiarato in questi parametri il sistema solleva un OutOfMemoryException.

Lo scheduler base è rappresentato dalla classe astratta Scheduler. Questa classe definisce i metodi per ottenere ed impostare lo scheduler di default; per inserire e rimuovere un oggetto schedulabile all’interno dell’insieme di oggetti di cui valutare la feasibility; per ottenere un valore booleano che indica se l’insieme di oggetti è schedulabile o meno e per settare i parametri di scheduling di un oggetto schedulabile solo se con i nuovi parametri il sistema l’applicazione resta schedulabile. Nonostante l’enfasi sull’analisi di schedulabilità, la specifica non richiede di realizzare nessuna analisi nel concreto, al contrario suggerisce, come algoritmo di default, di restituire sempre true per qualunque insieme di oggetti schedulabili. In merito alla classe Scheduler occorre infine sottolineare come, al contrario di quanto ci si sarebbe potuti aspettare, non è previsto nessun metodo di risposta ad eventi quale la release, il termine dell’esecuzione, o il cambiamento dei parametri di un oggetto schedulabile.

Lo scheduler di default per il sistema, ossia uno scheduler basato su priorità fissa, con almeno 28 differenti livelli di priorità, è rappresentato dalla classe PriorityScheduler che estende direttamente la classe Scheduler. E’ un singleton, ciò significa che esiste una sola istanza di PriorityScheduler nel sistema, recuperabile attraverso un getter statico. Oltre a questo metodo ed a quelli ereditati dalla classe madre, PriorityScheduler offre la possibilità di ottenere i valori numerici di priorità minima, di quella massima e di quella normale (ossia il valore intermedio tra quello minimo e quello massimo).

### RealtimeThread

La classe RealtimeThread è la classe centrale della specifica. Formalmente è un’estensione della classe java.lang.Thread e implementa l’interfaccia SchedulableObject, che estende a sua volta l’interfaccia Runnable. Ha molte proprietà che ne caratterizzano l’esecuzione.

* Le prime da evidenziare sono i parametri di scheduling visti in precedenza. Se non specificati attraverso il costruttore ogni nuovo thread real-time utilizza gli stessi parametri del thread che lo crea o, se questo non è un thread real-time, dei parametri di default.
* Ha un riferimento all’area di memoria nella quale opera (per maggiori dettagli sulle aree di memoria si veda la sezione a loro dedicata)
* Ha un riferimento allo scheduler dal quale è controllata. Si fa notare come la specifica preveda per i thread real-time un riferimento allo scheduler, mentre lo scheduler non ha nessun riferimento esplicito ai thread che deve gestire. Si può quindi già ipotizzare come, nelle idee degli autori della specifica, lo scheduler non sia un oggetto attivo e siano gli stessi thread, attraverso la chiamata di opportuni metodi, a sincronizzare la propria esecuzione.



I metodi senza dubbio più significativi della classe RealtimeThread sono quelli coinvolti qualora il processo sia periodico ossia waitForNextPeriod, schedulePeriodic e deschedulePeriodic.

Il metodo waitForNextPeriod, qualora il thread sia periodico (ossia collegato ad un’istanza di PeriodicParameters), sospende il thread fino all’inizio del nuovo periodo. Questo metodo permette di controllare se, nel corso della sua esecuzione, il thread ha violato la sua deadline in quanto, in questo caso, il metodo ritorna true permettendo di diversificare il comportamento del thread per eventuali azioni di recupero.

Il metodo deschedulePeriodic fa sì che alla prossima invocazione del metodo waitForNextPeriod il thread rimanga bloccato finché non viene chiamato il metodo schedulePeriodic

Il metodo run contiene il codice che il thread deve eseguire . Si fa notare come il metodo run venga invocato una volta sola durante l’esecuzione del thread, ossia quando viene avviato. Ciò accade anche per i thread periodici. Deve essere lo sviluppatore che, all’interno del metodo run, crea un ciclo che si sincronizza con il periodo del thread chiamando il metodo waitForNextPeriod alla fine di ogni iterazione, in modo da sincronizzare l’inizio della prossima iterazione con quella del prossimo periodo.

Come già detto, analizzando Java standard da un punto di vista real-time, uno dei problemi maggiori è il fatto che i thread possono essere ritardati indiscriminatamente a causa di azione del garbage collector. Una delle soluzioni a questo problema è l'introduzione di zone di memoria (ScopedMemory ed ImmortalMemory) nelle quali il garbage collector non interviene. La classe NoHeapRealtimeThread rappresenta un thread che non accede mai all'heap, neanche in maniera transitiva. Di conseguenza, se ne può garantire l'esecuzione anche quando il garbage collector sta eseguendo. I suoi costruttori accettano come parametro un’ area di memoria e controllano che questa non sia l'heap, in caso contrario viene lanciata un'eccezione. Questo controllo viene fatto anche quando il thread viene lanciato attraverso il metodo start.

### Eventi asincroni

Un sistema in tempo reale deve potere reagire tempestivamente ad eventi inaspettati, quali sollecitazioni dal mondo esterno, errori di componenti software o semplicemente ad eventi che si vuole gestire con una strategia push. Nella figura sottostante vengono mostrate le classi coinvolte nel definire gli eventi asincroni e quelle coinvolte nella loro gestione.



In nucleo del sistema è composto dalla coppia AsyncEvent e AsyncEventHandler. Essi rappresentano rispettivamente l’evento asincrono ed il gestore ad esso associato. Come si può carpire dalla relazione molti a molti un evento può essere collegato a più gestori, così come lo stesso gestore può rispondere a più eventi.

La classe AsyncEventHandler rappresenta il gestore di un evento asincrono ed implementa l’interfaccia Schedulable, quindi ha associata ad essa tutti i parametri di scheduling che ne permettono l’esecuzione in maniera controllata. Quando si verifica un evento vengono schedulati tutti i gestori ad esso associati. Questi eseguono il metodo handleAsyncEvent. Si sottolinea come gli handler possano non entrare subito in esecuzione, ad esempio perché possono subire preemption da parte di processi più prioritari e nel frattempo si possono verificare altri eventi a cui sono associati. Per questo motivo, ogni volta che si verifica un evento a loro associato, tutti gli handler vedono incrementata la proprietà fireCount, che indica quanti eventi si sono verificati prima della loro attivazione. Ad ogni esecuzione del metodo handleAsyncEvent il valore di fireCount viene decrementato finchè non raggiunge il valore zero. Per garantire una maggiore flessibilità nel suo utilizzo, la classe fornisce anche una serie di metodi protetti per gestire il fireCount (incrementarlo, decrementarlo, azzerarlo).

Il sistema mantiene un pool di thread real-time deputati ad eseguire i gestori. E’ quindi necessario effettuare un binding dinamico tra i thread servitori e gli eventi da eseguire. In certe situazioni l’overhead di questa operazione non è tollerabile. Per questi casi la specifica propone la classe BoundAsyncEventHandler che lega al gestore un thread servitore dedicato.

In un sistema real-time gli eventi temporizzati (legati cioè non ad un evento interno o esterno al sistema, ma al trascorrere del tempo) rivestono grande importanza. La specifica Java real-time propone a tale scopo la classe astratta Timer e le sue due estensioni concrete OneShotTimer (un timer che genera un evento solo) e PeriodicTimer (un timer che lancia eventi ad intervalli regolari di tempo). Tutti i timer si basano su un Clock, se non viene specificato diversamente utilizzano quello di sistema. Ogni Timer ha un riferimento temporale (relativo o assoluto) a quando deve lanciare l’evento, se non ci sono handler associati al timer quando questo scatta l’evento viene perso. Attraverso una serie di metodi si può controllare il timer: con il metodo start lo si fa partire; il metodo disable fa sì che il timer continui il conto alla rovescia ma non lanci eventi; il metodo enable riabilita il timer al lancio di eventi; infine, il metodo reschedule permette di modificare quando scatterà il timer.

La classe POSIXSignalHandler consente di gestire l’interazione con altri componenti attraverso i segnali POSIX, consentendo di richiamare un handler specificato in risposta all’arrivo di un segnale POSIX.

### Trasferimento asincrono di controllo

Con il termine trasferimento asincrono di controllo in Java real - time si intende la possibilità di interrompere in modo asincrono un thread e di fargli eseguire dell'altro codice. Per ottenere questa importante funzionalità esistono sostanzialmente due approcci, uno basato sulla dichiarazione di metodi interrompibili, uno basato sull'esecuzione di oggetti interrompibili.

Il primo approccio si basa sulla possibilità di dichiarare certi metodi come interrompibili. Se si chiama il metodo interrupt di un thread real time mentre questo è in esecuzione, verrà lanciata una AsyncronouslyInterruptedException che il sistema deve raccogliere e, nel relativo catch, si può inserire del codice per rispondere all'interruzione aggiunta dall'esterno. Il comportamento è quindi sostanzialmente diverso da quanto accadeva con i normali thread nei quali bisognava verificare con una strategia a polling se il thread era stato interrotto meno.

Per poter gestire questa funzionalità occorre, come già detto, specificare i metodi che supportano un'interruzione asincrona e quelli che non lo fanno. I primi metodi vengono chiamati metodi interrompibili asincronicamente (AIMetods); tali metodi si differenziano dagli altri per il fatto dichiarano una AsyncronouslyInterruptedException nella loro throw list. I secondi metodi, quelli ordinari e le sezioni di codice synchronized, vengono definite ATCDeferred: se viene invocato il metodo interrupt mentre un thread sta eseguendo questa regione di codice l'eccezione verrà inviata solamente una volta che il thread entrerà in una sezione interrompibile.

Dal momento che questo approccio è a tutti gli effetti analogo alla gestione di un'eccezione non è possibile rientrare all'interno del catch nel quale l’eccezione stessa è stata generata. Ciò comporta che, una volta gestita l'interruzione, non è possibile riprendere l'esecuzione dove la si era lasciata.

Il secondo approccio si basa sulla classe AsyncronouslyInterruptedException e sull'interfaccia Interrumpible. In figura si mostrano le classi coinvolte in questo secondo sistema di gestione del trasferimento asincrono di controllo.



L'interfaccia Interrumpible contiene due metodi: il metodo run che contiene il codice dell'esecuzione “normale” ed il metodo interruptAction che viene eseguito quando c'è l'interruzione. Tutto è gestito dalla classe AsyncronouslyInterruptedException: l'oggetto interrompibile viene passato come parametro al metodo doInterruptible che non fa altro che eseguire il metodo run dell'oggetto passatogli. Se si vuole interrompere l'esecuzione del thread che sta eseguendo questo metodo si può chiamare dall'esterno il metodo fire di AsyncronouslyInterruptedException. Questo causa l'interruzione dell'esecuzione del metodo run e fa eseguire al thread interrotto il metodo interruptAction dell’oggetto Interrumpible. Anche questo approccio ha il difetto di non permettere il ritorno nel punto del metodo run nel quale si è verificata l'interruzione.

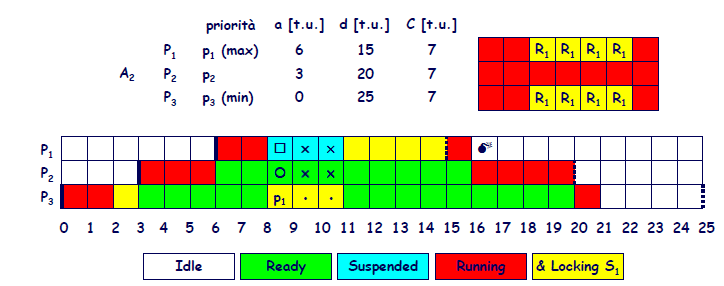
### Sincronizzazione

Gli strumenti che Java mette a disposizione per sincronizzare i Thread sono sostanzialmente basati su lock a mutua esclusione. Seppur dalla comprovata validità nel caso generale, queste soluzioni non sono da sole soddisfacenti per i sistemi in tempo reale, in quanto prone a fenomeni indesiderati quali l’inversione incontrollata di priorità. In letteratura sono presenti numerosi protocolli per migliorare la sincronizzazione tra processi in tempo reale. La specifica Java Real Time supporta due tra i protocolli più noti per la sincronizzazione di processi in ambienti real-time: priority inheritance (di default) e immediate priority ceiling. Di conseguenza, i meccanismi di accesso alle sezioni ad accesso mutuamente esclusivo (come i blocchi synchronized) sono stati modificati per realizzare le politiche in questione, alterando le priorità correnti dei processi che vi entrano. Nella figura sottostante si mostrano le due classi che rappresentano l’implementazione di queste due strategie



#### Priority Inheritance

Il protocollo priority inheritance prevede che quando un processo a priorità maggiore è bloccato nell’accesso ad una risorsa posseduta da un processo a priorità minore quest’ultimo erediti la priorità del processo a priorità maggiore, cosa che gli consente di esercitare preemption su processi a priorità intermedia. Riprendiamo l’esempio mostrato nel primo capitolo per mostrare come, utilizzando priority inheritance, il fenomeno dell’inversione incontrollata di priorità venga evitato. Come è facile intuire il protocollo funziona senza sapere a priori quali processi utilizzeranno quali risorse ed è quindi molto adatto ad ambienti dinamici.

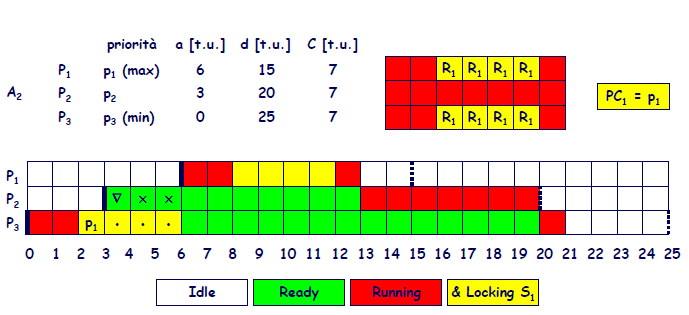


Come si può notare il processo p2 subisce preemption da p3 che ha ereditato la priorità da p1 e non può più bloccare indirettamente quest’ultimo.

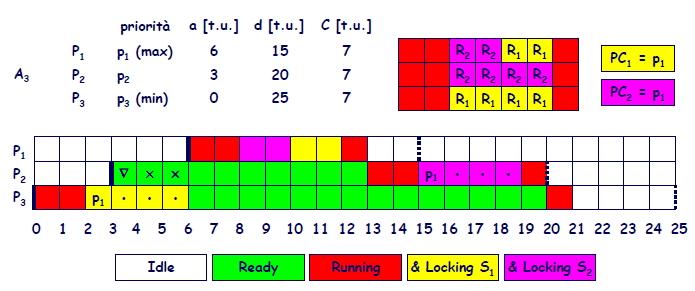
Come già detto priority inheritance è il protocollo di default del sistema, la classe PriorityInheritance è quindi pensata per reimpostare questa politica attraverso il metodo setMonitorControl della classe madre MonitorControl.

#### Priority Ceiling

La specifica prevede, come funzionalità dalla realizzazione facoltativa, la politica immediate priority ceiling o priority ceiling emulation. Questo protocollo prevede di associare ad ogni risorsa un valore che esprime il suo tetto di priorità. Questo valore è pari alla priorità maggiore tra quelle dei thread che possono accedere alla risorsa stessa. Quando un thread accede ad una risorsa la sua priorità diventa pari al massimo tetto di priorità delle risorse in suo possesso. Inoltre, le code dei processi pronti devono essere gestiti secondo una politica FIFO. L’effetto finale è che un thread può subire un blocco solo all’inizio della sua esecuzione. Come mostrano gli esempi seguenti il protocollo priority ceiling, oltre ad evitare l’inversione incontrollata di priorità, risolve un altro comportamento indesiderato in un sistema in tempo reale noto come concatenazione di blocchi, ossia il fatto che un processo possa essere bloccato più volte nell’accesso a risorse distinte.



Nell’esempio riportato nella figura soprastante è possibile osservare come il protocollo previene il fenomeno dell’inversione incontrollata di priorità dal momento che, eseguendo il blocco di codice mutuamente esclusivo con priorità p1, il processo p3 esercita preemption su p2 che non può ritardare indebitamente p1.



In questo esempio si mostra come immediate priority ceiling permetta di evitare la concatenazione di blocchi in quanto un processo (p2, nel caso dell’esempio) può essere bloccato solamente all’inizio della sua esecuzione.

Il maggior limite di priority ceiling è che occorre conoscere quali processi possono accedere a quali risorse al fine di determinare i tetti di priorità delle risorse stesse. La classe PriorityCeilingEmulation serve per rappresentare questa politica ed occorre quindi un attento studio di ogni applicazione prima di poterlo impiegare.

Per ogni oggetto coinvolto è necessario specificare il tetto di priorità tramite il metodo setMonitorControl. Se a run-time un processo accede ad una risorsa con una priorità maggiore rispetto al tetto di priorità della risorsa stessa, viene lanciata un’eccezione per indicare che la risorsa non è stata configurata correttamente.

# L’implementazione di Sun

La versione utilizzata durante in questa tesi è Java Real-Time System 2.2 rilasciata da Sun nel 2009. Nello specifico si è utilizzata la piattaforma riservata ad uso accademico su sistema operativo Solaris 10.9. Java Real-Time system recepisce la specifica RTJS (Real Time Specification for Java). Non implementa nessuna delle funzionalità facoltative della specifica, in particolare:

* Non supporta la funzionalità di controllo del tempo di esecuzione di un oggetto schedulable (cost enforcement).
* Non supporta il protocollo priority ceiling, priority inversion è l’unico protocollo per l’accesso a risorse condivise ad essere implementato.
* Non fornisce altri clock a parte quello di sistema. Questo, se il sistema operativo sottostante è in grado di supportarlo, ha precisione al nanosecondo.
* Estende il numero di livelli di priorità riservati ai processi real – time rispetto ai 28 richiesti dalla specifica: sono presenti, infatti, 58 distinti livelli di priorità.

L’implementazione di Sun fornisce una Java Virtual Machine modificata al fine di aumentare il determinismo delle applicazioni java real – time. Si illustra come questa virtual machine permette di controllare le tre fonti di jitter principali di un’applicazione java: l’inizializzazione, la compilazione e le interferenze dovute al garbage collector

## Inizializzazione e compilazione

Come spiegato nel capitolo sui limiti di Java come piattaforma per i sistemi in tempo reale, i meccanismi di lazy initialization e di just in time compilation previsti per la java virtual machine standard sono inadatti per i sistemi in tempo reale in quanto possono introdurre dilatazioni dei tempi di esecuzione di un’applicazione in momenti impredicibili.

La soluzione ideata per il sistema Java real – time consiste nel fornire alla virtual machine una lista degli elementi da inizializzare e da compilare prima di far partire un’applicazione. In questo modo questa non sarà più ritardata da eventuali fasi di compilazione o di inizializzazione durante la sua esecuzione.

E’possibile indicare un file che contiene la lista delle classi che devono essere inizializzate prima che inizi l’esecuzione dell’applicazione tramite l’opzione -XX:PreInitList=<preinit-file-name> . E’ anche possibile far creare il file automaticamente dal sistema grazie all’opzione -XX:+RTSJBuildClassInitializationList. Questa opzione fa si che la Virtual Machine generi un file che contiene la lista delle classi referenziate durante l’esecuzione. Il file viene generato in maniera incrementale, ciò significa che , se in una successiva esecuzione dell’applicazione viene referenziata una classe non utilizzata in precedenza, questa viene inserita in fondo al file già formato.

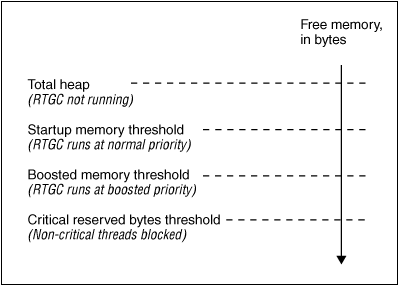
Analogamente, per quanto riguarda la compilazione è possibile specificare tramite l’opzione -XX:CompilationList = <precompile–file-name> un file che contiene la lista dei metodi da compilare prima dell’esecuzione del programma Java. E’ importante che in questa lista siano presenti i metodi utilizzati dai thread real-time in modo che questi vengano eseguiti in modo compilato e non interpretato, con conseguente beneficio in termini di velocità, ma senza l’overhead altrimenti dovuto alla compilazione just in time. Come per la lista di inizializzazione anche questo file si può far generare automaticamente ed in maniera incrementale attraverso l’opzione -XX:+RTSJBuildCompilationList .Con questa opzione la virtual machine inserisce nel file tutti i metodi che vengono eseguiti dai thread real time.

## Garbage Collector

Occorre ricordare come l’attività di Garbage Collection sia fondamentale per preservare la funzionalità di un sistema Java: permette, infatti, di riciclare la memoria altrimenti occupata da oggetti non più utilizzati e di metterla a disposizione del sistema per l'allocazione di nuovi oggetti.

Per quanto importante, in un sistema real - time occorre che questa attività non vada a modificare sensibilmente i tempi di esecuzione delle applicazioni in corso, con il rischio che queste sfornino i vincoli temporali a loro assegnati.

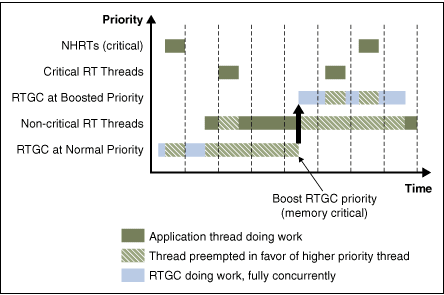
Per questo motivo Java real – time system prevede l'introduzione di un nuovo garbage collector in grado di coesistere con le applicazioni real - time: può essere infatti impostato affinché non eserciti preemption nei confronti dei processi real - time e, in caso di sistemi multiprocessore, affinché lavori su una sola cpu senza bloccare l'esecuzione dei processi di Java sugli altri processori.

Entrando nello specifico, come mostrato nella figura a fianco, il garbage collector ha tre modalità di funzionamento, basate sulla quantità di memoria libera residua:

* Finché la quantità di memoria libera nell'heap resta sopra la soglia di startup memory non viene eseguita nessuna azione di garbage collecting.
* Quando la memoria libera scende sotto la soglia di startup mermory il garbage collector esegue alla priorità normale: quando l'opera a questa priorità il garbage collector blocca solamente i thread non real - time.
* Se la memoria libera scende oltre la soglia di BoostedMemory, il garbage collector esegue ad una priorità maggiore rispetto a quella normale detta BoostedPrioriy. Con il garbage collector a questa priorità vengono bloccati anche alcuni il thread real – time (quelli con priorità inferiore alla boosted priority) che il sistema, di conseguenza, considera non critici.
* Se la memoria libera scende ulteriormente fino ad occupare anche la quantità riservata ai processi critici il garbage collector entra nella modalità deterministica. In questa modalità il garbage collector lavora sempre con priorità boosted, tuttavia le richieste di memoria di thread non critici vengono bloccate finché la quota di memoria libera non torna sopra alla soglia. In questo modo si cerca di garantire la funzionalità dei processi più importanti per il sistema, in quanto solo i processi che eseguono a livello critico sono in grado di allocare memoria proveniente dalla quota riservata appositamente per loro.

Tutti i parametri descritti finora (le priorità e le soglie di memoria) vengono calcolate automaticamente dalla virtual machine; in alternativa si possono indicare manualmente tramite opzioni.

La figura sottostante mostra un esempio di funzionamento del garbage collector, in particolare si evidenzia come, quando la memoria libera scende sotto la soglia di boosted memory threshold, il garbage collector esegue a priorità maggiore esercitando preemption anche su alcuni thread real - time non critici per il sistema.



Prime estensioni a Java Real-Time

In questo capitolo si illustrano i moduli sviluppati col fine di fornire i servizi di busyWait (utile per modellare l’esecuzione di un processo in un sistema real time) e di logging (necessario per fornire allo sviluppatore i risultati dell’esecuzione senza rallentare la stessa con scritture su console). Infine, si espone il contenuto della classe PeriodicThread, necessaria per simulare il comportamento di un thread periodico.

# BusyWait

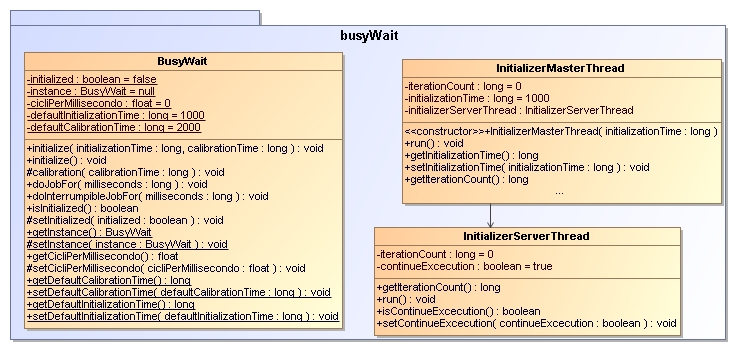
Fin dall’inizio della tesi si è palesata la necessità di modellare l’esecuzione di un processo in un sistema real time. Il modulo di busy wait sviluppato fornisce proprio questo servizio: permette di modellare un’esecuzione di durata specificata.

Una strategia percorribile sarebbe quella di realizzare un metodo che sblocchi il thread dopo il tempo di esecuzione desiderato. Tuttavia, considerando la possibilità che il thread subisca preemption, non è possibile basarsi su i riferimenti temporali del sistema, ci sarebbe quindi la necessità di tenere traccia del tempo di esecuzione effettivo del processo.

Se, quindi, la funzione di cost enforcement fosse supportata, il modulo in questione si risolverebbe in un handler che provvede a sbloccare il thread una volta richiamato a causa del raggiungimento del tempo di esecuzione specificato nei ReleaseParameters. Non essendo questo il caso di Java Real Time System, si è deciso di utilizzare un approccio differente, in cui il thread esegue effettivamente del codice occupando la cpu per la frazione di tempo desiderata.

La busy wait sviluppata è sostanzialmente formata dall’esecuzione di un ciclo che compie delle operazioni senza alcuna utilità pratica, con il solo scopo di occupare la cpu. La durata del singolo ciclo deve essere abbastanza breve, in modo da poter specificare finemente i tempi di esecuzione. Grazie alle modifiche introdotte alla virtual machine, in particolare nel campo della precompilazione, è possibile, a parità di numero di iterazioni da eseguire, ottenere tempi di esecuzione estremamente stabili. Il problema si sposta quindi nel determinare correttamente il numero di iterazioni da compiere per avere un’esecuzione della lunghezza desiderata. Con l’obbiettivo di rendere il modulo utilizzabile su più macchine possibile, si è stabilito che, prima dell’esecuzione vera e propria, il modulo in questione debba essere inizializzato, cosa che permette di ottenere un numero veritiero di quante iterazioni il sistema riesce a svolgere in un certo periodo di tempo; una volta fatto ciò, si può dualmente ottenere un valore affidabile indicante quante iterazioni svolgere per realizzare un’esecuzione della durata specificata.

Seguendo una logica di servizio si è deciso che la busy wait dovesse essere un modulo separato dai thread che lo eseguivano. Si mostrano a figura le classi che permettono la realizzazione di questo servizio.



La classe principale, **BusyWait** è un singleton. Come già detto, per poter essere utilizzata deve essere inizializzata tramite il metodo initialize.

Ha un attributo privato, float, che indica il numero di iterazioni compiute mediamente in un millisecondo

Ha come attributo due long. Il primo di questi indica di quanti millisecondi deve essere lunga la fase di inizializzazione di default, se non settato diversamente, è pari a 1000 (1 secondo). E' consigliabile, per non perdere precisione che tale valore non scenda sotto il decimo di secondo.

Un altro long indica quanto deve essere lunga la fase di calibrazione di default. Se non settato diversamente è pari a due secondi, per non perdere in precisione è consigliabile sia comunque almeno di mezzo secondo

La classe BusyWait esibisce il metodo *doJobFor(long millis)* che non fa altro che eseguire lo stesso ciclo usato per la inizializzazione un numero di volte pari al prodotto del numero di cicli per milliscondo per il numero di millisecondi richiesto. Se la classe non è inizializzata il metodo stesso esegue una inizializzazione rapida

Esibisce anche il metodo *doInterumpibleJobFor(log millis*). Questo metodo è identico a doJobFor, con la differenza che, quando viene eseguito, può essere interrotto asincronicamente in quanto può lanciare una AsynchronouslyInterruptedException.

Il metodo Initialize crea un'istanza della classe InitializerMasterThread e la avvia. Questa classe è un thread realtime che esegue con priorità massima.

Ha, come attributo, un long che indica di quanti millisecondi deve essere lunga la fase di inizializzazione.

Un'altro attributo è secondo long che indica quanti cicli sono stati eseguiti nel tempo di inizializzazione.

Il metodo run di **InitializerMasterThread** non fa altro che chiamare il metodo omonimo della classe InitializerServerThread, sospendersi per il tempo della fase di inizializzazione e, al risveglio, interrompere InitializerServerThread per poi prendere nota di quanti cicli di esecuzione ha svolto.

La classe **InitializerServerThread** ha come proprietà il numero di cicli eseguiti fino al momento in cui è interrotta.

Il suo metodo run contiene lo steso ciclo utilizzato dalla busy wait, nel quale, oltre ad altre operazioni, si incrementa il conteggio del numero di cicli effettuati.

La necessità della fase di calibrazione emerge dai risulati sperimentali ottenuti, presentati nella tabella in fondo al paragrafo. Si evince come i tempi di esecuzione, se si esegue la sola inizializzazione, siano sensibilmente differenti da quelli specificati. E' quindi necessario fare una calibrazione, tramite un’invocazione al metodo calibration, della variabile cicli per millisecondo. In questo metodo si fa una busywait di durata pari al parametro di calibrationTime e se verifica quanto questa è durata in realtà. Si ottiene quindi un numero più corretto di iterazioni compiute dal sistema in un millisecondo dividendo il vecchio numero di cicli al millesimo per la durata effettiva e moltiolicando per quella nominale. In formula: **n iterazioni per millesimo = n° iterazioni per millesimo \* durata nominale ciclo di busyWait / durata effettiva ciclo di busyWait.**

|  |  |  |  |
| --- | --- | --- | --- |
| Tempo esecuzione Effettivo | Tempo teorico Esecuzione | Tempo Calibrazione | rapporto tempo effettivo /tempo nominale |
| 267 | 500 | 30000 | 53,40% |
| 267 | 500 | 1000 | 53,40% |
| 1067 | 2000 | 1000 | 53,35% |
| 5339 | 10000 | 1000 | 53,39% |
| 533 | 1000 | 100 | 53,30% |
| 530 | 1000 | 10 | 53,00% |
| 53 | 100 | 5000 | 53,00% |
| 534 | 1000 | 2000 | 53,40% |

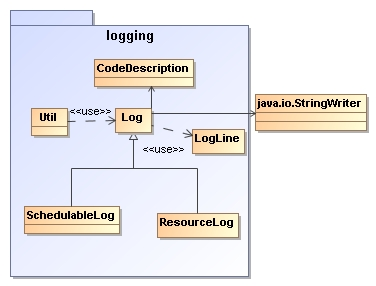
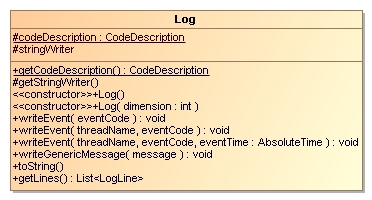
# Logging

La programmazione concorrente necessita di strumenti di debug che permettano allo sviluppatore di controllare l’esecuzione dei vari processi. In un sistema realtime, visti gli stringenti vincoli temporali, è importante tenere traccia delle varie entità coinvolte senza intaccare le performance del sistema.

Bisogna inoltre evitare assolutamente che queste attività, accedendo a risorse condivise del sistema blocchino i processi della cui esecuzione si vuole tenere traccia.

L’idea alla base del modulo di logging sviluppato è che ogni entità coinvolta nel sistema abbia una sua zona di memoria dedicata nella quale vengono registrati gli eventi che la coinvolgono. Le scritture di questi eventi debbono essere il più veloci e sintetiche possibili. Una volta terminata l’esecuzione, si possono raccogliere i vari log e rielaborarli in una forma user-freindly di facile comprensione per l’utente.

Il diagramma seguente fornisce un sunto delle classi sviluppate per fornire le funzionalità descritte sopra



La classe **Log** rappresenta il generico Log associato ad una entità real-time, come detto ha una zona di memoria dedicata rappresentata da uno stringWriter nel quale vengono registrati i vari eventi. Le scritture di eventi hanno il formato nomeThread;codice Evento;timestamp. La classe codeDescriprion contiene le coppie codice-descrizione dei vari eventi. Per aggiungere un nuovo tipo di evento non previsto basta aggiungere una nuova entry a questa tabella. Attualemente gli eventi previsti sono:

00: messaggio generico

10: creazione di un oggetto schedulabile (thread o handler)

20: inizio job

30: termine job

40: occupazione risorsa

50: rilascio risorsa

60: deadline non rispettata

70: job saltato per polica skip

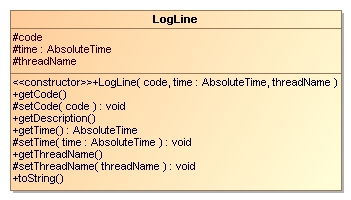
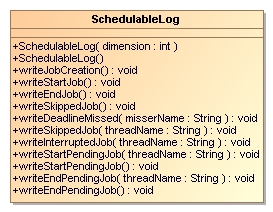
80: job interrottto per politica stop

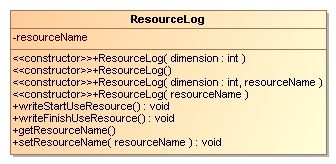
90: inizio job di recupero

100: fine job di recupero

La classe fornisce le funzionalità per scrivere un evento sul log specificando solo codice dell’evento, oppure fornendo anche il nome del thread e il timestamp. E’ anche possibile scrivere un generico messaggio sul log sfruttando il campo dedicato al nome del thread ed al pseudo evento messaggio generico (00).

Tramite il metodo getLines della classe Log è possibile ottenere l’insieme degli eventi registrati nel log sotto forma di oggetti LogLine.

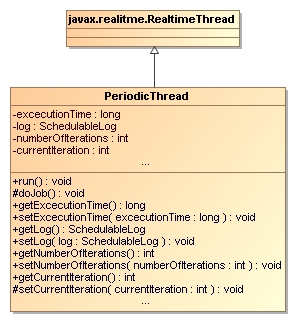
La classe **LogLine** è stata creata per contenere, in maniera strutturata tutti i campi presenti in una riga di log (codice evento, nome del thread che lo ha genereato e timestamp dell’evento stesso). Tutte queste informazioni sono incapsulate in proprietà della classe stessa.

La classe **SchedulableLog** è pensata per facilitare la scrittura degli eventi relativi agli oggetti schedulabili (derivati ,quindi, da RealtimeThread o AsyncEventHandler). In Particolare esibisce una serie di metodi che permettono di incapsulate la scrittura di eventi relativi all’inizio ed alla fine di un job, al verificarsi di un deadline miss, al salto di un job per alleviare il carico sul sistema.

Analogamente la classe **ResourceLog** è pensata per facilitare la scrittura di eventi (principalmente l’inizio e la fine dell’uso) di una risorsa ad accesso esclusivo.

La classe **Util** è pensata per aiutare l’utente nell’unificare i vari log nella fase di raccolta dei risultati. Il metodo simpleMerge è pensato per fondere i vari log fornendo in output una stringa con tutti gli eventi presenti nei log passati come parametro ordinati in ordine cronologico. Il metodo RelativeMerge si comporta in maniera analoga, ma i tempi forniti in output sono relativi ad un istante di tempo specificato, anziché gli AbsoluteTime forniti da simpleMerge che possono essere di difficile lettura .

# Periodic Thread



Sin dal principio è emersa la necessità di modellare un processo periodico. La classe **PeriodicThread**, che estende la classe RealtimeThread fornita da java real-time, svolge questo compito. Con lo scopo di non creare confusione nel lettore e di facilitare la lettura della tesi, in questo capitolo se ne mostrano le componenti basilari, mentre le parti connesse con le tecniche introdotte nei capitoli successivi (come ad esempio la parte correlata alla gestione dei deadline miss o il metodo run dedicato all’esecuzione con una scheduler edf) verranno introdotte e trattate nei rispettivi capitoli.

La classe possiede le seguenti proprietà:

* il log sul quale scrivere gli eventi che la riguardano
* il tempo di esecuzione di ciascun job
* il numero di iterazioni che deve eseguire (il numero di job)
* il numero dell'iterazione (del job) corrente

Il metodo *run* contiene il codice che viene eseguito una volta lanciato il thread. Come già detto, affichè un thread real-time abbia carattere periodico, oltre ad essere collegato con dei periodic parameters, occorre che abbia come corpo dell’esecuzione un ciclo e che, al termine di ogni ciclo il thread si blocchi in attesa del prossimo istante di release attraverso una chiamata al metodo waitForNextPeriod.

In ogni ciclo si eseguono, quindi, le seguenti operazioni:

1. si scrive l’inizio del job sul log
2. si incrementa la proprietà che indica il numero dell’iterazione corrente
3. si esegue una busyWait di durata pari all’apposita proprietà della classe
4. si scrive sul log del termine del job
5. si chiama il metodo waitForNextPeriod della classe madre PeriodicThread

Si può osservare che, qualunque sia il codice applicativo che il thread deve eseguire (nel caso della classe base PeriodicThread tutto si riduce ad una busyWait) esistono delle operazioni (come la scrittura sul log) che vengono in ogni caso sempre eseguite. Per separare il codice che contiene la business-logic di competenza del thread dal resto (che in ambito informatico viene spesso chiamato boilerplate code) si è racchiusa la parte relativa alla sola business logic nel metodo doJob. In questo modo, qualora si voglia creare un thread che esegue dei compiti differenti dalla sola busyWait di durata fissa sarà sufficiente estendere la classe PeriodicThread e inserire il codice relativo alla business logic facendo un override del solo metodo doJob, senza dover riscrivere, con la possibilità di commettere errori, tutto il metodo run.

Uno scheduler EDF per java Realtime

La strategia di scheduling di default di Java real-time si basa sulla schedulazione priority driven dei thread. Come già affermato, il sistema pone in esecuzione, tra tutti i processi pronti, quello che vanta la priorità maggiore.

L’approccio del sistema rende automatico realizzare la politica Rate Monotonic Priority Order (RMPO). Questo algoritmo di scheduling prevede di assegnare le priorità dei processi in base loro al periodo: più questo è breve, più alta è la priorità del processo.

RMPO viene definito come un algoritmo a priorità statica in quanto la priorità assegnata ai processi non cambia mai durante l’esecuzione. In particolare si può considerare RMPO come l’algoritmo a priorità statica ottimo in quanto se un insieme di processi non è schedulabile (ossia non se ne può garantire l’esecuzione senza che un processo violi una sua deadline) con RMPO, allora non può esserlo con nessun altro algoritmo a priorità statica. Dualmente, se un insieme di processi è schedulabile con un algoritmo a priorità statica, allora è schedulabile anche usando RMPO.

Dal momento che il processo a priorità maggiore esercita preemption su tutti gli altri, risulta evidente come questo non subirà mai interferenze (se si esclude il caso delle risorse ad accesso mutuamente esclusivo) dai processi a priorità minore. Questo aspetto è estremamente positivo infatti, considerando come spesso i processi che vengono eseguiti con frequenza maggiore sono quelli più importanti per il sistema, si garantisce a loro un’esecuzione prioritaria estremamente utile in caso di sovraccarico: saranno, infatti, solo i processi a priorità minore ad avere problemi di rispetto di vincoli temporali.

Di contro, l’attribuzione statica di priorità limita il range dell’insieme di applicazioni che sono schedulabili con RMPO, specialmente se confrontati con algoritmi che assegnano dinamicamente le priorità ai processi. Una prima misura di questa inefficienza deriva da uno dei più semplici criteri per verificare se si può garantire la schedulabilità dei un insieme di processi: il test di Liu – Layland.

Questo test di schedulabilità (feseability test) garantisce che un insieme di processi puramente periodici,in assenza di accessi a risorse condivise, è schedulabile se il carico sulla cpu (ossia la somma dei rapporti tra tempo di esecuzione e periodo dei vari processi) è inferiore a N\*(21/N-1), dove n è il numero di processi presenti nel sistema. Questa misura per N che tende ad un valore infinito vale circa 0,7.

Tra gli algoritmi a priorità dinamica il più celebre è EDF (Earliest Deadline First). La strategia di questo algoritmo è quella di porre in esecuzione il processo con deadline più imminente. Di conseguenza, ad ogni processo è associata dinamicamente una priorità tanto maggiore quanto più imminente è la prossima deadline assoluta.

Confrontato con RMPO ha performance sono migliori, in quanto si può garantire la schedulabilità di un insieme di processi puramente periodici, senza l’accesso a risorse condivise se il fattore di utilizzazione del processore è inferiore o pari al 100%.

Di contro, uno dei maggiori difetti di EDF deriva dalla sua natura di algoritmo a priorità dinamica. Difatti, in scenari di sovraccarico presta il fianco a critiche legate al fatto che, non essendoci una priorità esplicita tra i processi, spesso il sistema mette in esecuzione job di rilevanza secondaria, facendo violare la deadline a processi ritenuti sì più importanti dall’utente, ma con la deadline successiva.

**Se hai bisogno di fare massa puoi aggiungere qui il documento Funzionamento Scheduler**

# L’implementazione dello scheduler

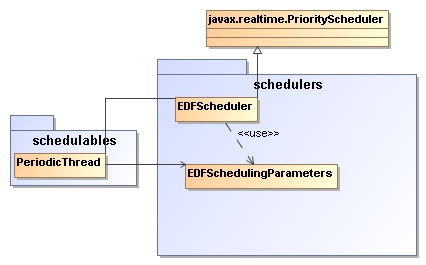
Si è realizzato uno scheduler che realizza la strategia EDF in java real-time. L’approccio utilizzato per la creazione dello scheduler è basato su una coda ordinata di processi in base alla priorità. Quando un thread entra in esecuzione chiama una funzione dello scheduler che lo inserisce nella coda in base alla sua deadline assoluta, quindi mette in esecuzione, ponendo a priorità maggiore rispetto agli altri, il processo con deadline più imminente.

Nello specifico il sistema funziona con quattro livelli di priorità:

1. Il primo, quello a priorità maggiore, è quello di un job quando esegue i metodi dello scheduler all’inizio ed alla fine del job (releasePriority)
2. Il secondo è dedicato agli handler associati ai thread (handlerPriority)
3. Il terzo è quello di un job durante la sua esecuzione (excecutionPriority)
4. Il quarto è quello dei job pronti che sono sospesi perché è in esecuzione un job con una deadline più imminente (readyPriority)

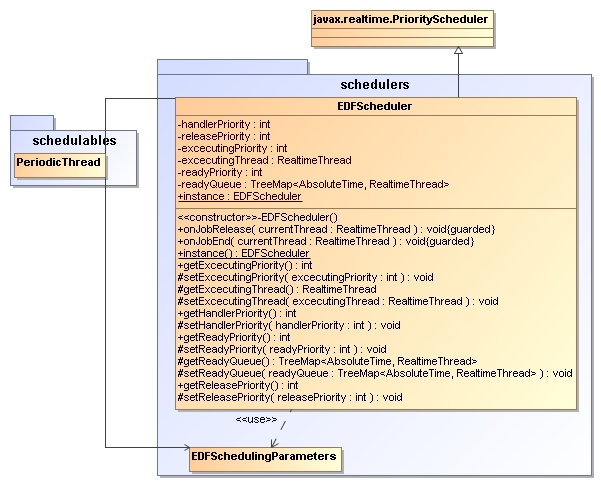
Si fa notare come lo scheduler ideato sia robusto anche in nel caso in cui il processo in esecuzione si blocchi nell’accesso ad una risorsa mutuamente esclusiva: dal momento che il sistema fornisce di default il protocollo priority inheritance, il thread che possiede la risorsa (che, per forza di cose si trova tra i processi pronti) si vede aumentare la priorità a quella del thread in esecuzione che sta bloccando. Può quindi terminare l’esecuzione della sezione di codice ad accesso mutualmente esclusivo liberando la risorsa e sbloccando il processo con deadline più imminente che era bloccato sulla risorsa stessa.

lo scheduler è ideato per funzionare in un ambiente monoprocessore, tuttavia nella sua progettazione si è tenuto conto della possibilità di estenderlo in futuro per ambienti dotati di più di una CPU. Si mostra un diagramma statico che formalizza l’archittettura generale dei componenti riguardanti lo scheduler EDF, successivamente ci si soffermerà sulle singole parti e sul loro funzionamento.



## La classe EDFScheduler

Si mostra il diagramma statico della classe EDFScheduler e si chiariscono le funzionalità da esso esposte.



Lo scheduler EDF è un'estensione di PriorityScheduler, ciò implica che è sempre in esecuzione il processo a priorità maggiore. Lo scopo dello scheduler è affidare questa priorità al processo con deadline in più imminente tra quelli pronti per l’esecuzione. A tale scopo, lo scheduler tiene ordinati i job che sta gestendo per deadline assoluta. Questa struttura dati è, per comodità di implementazione, una tabella ordinata in cui la chiave è la deadline assoluta contenuta negli EDFSchedulingParameters associati ai thread real-time, mentre questi ultimi sono il valore della tabella in questione.

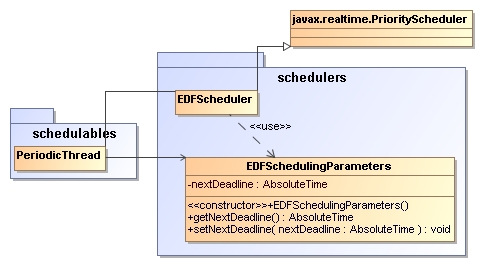
Lo scheduler, oltre alla tabella ordinata dei processi pronti,contiene anche un riferimento al processo attualmente in esecuzione. Fornisce, inoltre, i metodi per ottenere i valori numerici delle quattro priorità.

Il nucleo dello scheduler sono i due metodi statici onJobRelease e onJobEnd. Questi due metodi vengono eseguiti come preambolo e come epilogo da tutti thread real - time. Per una loro spiegazione dettagliata si rimanda alla sezione modifiche a periodicThread.

Il metodo *onJobRelease* , eseguito dai thread come preambolo di ogni nuovo job, provvede a calcolare la prossima deadline assoluta e, basandosi su di essa, ad inserire il thread nella tabella ordinata dei processi pronti oppure, se la deadline è più imminente di quella del processo attualmente in esecuzione, a porlo direttamente in esecuzione, inserendo il processo che stava eseguendo nella coda dei processi pronti. In questo modo sarà sempre in esecuzione, a meno di vincoli sincronizzazione, il processo con la deadline più imminente.

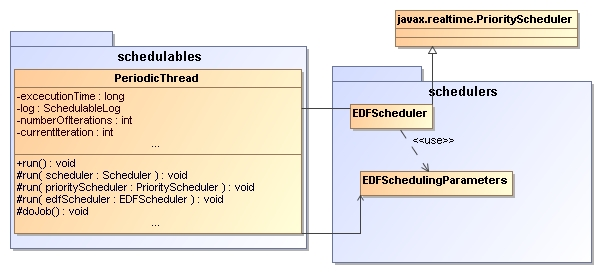
Quando il thread termina l'esecuzione di un job deve chiamare come epilogo il metodo **onJobEnd** dello scheduler. Questo metodo reimposta la priorità del thread a quella massima, in modo che al sua prossimo istante di release non subisca preemption da nessuno ed esegua tempestivamente il preambolo. Infine, estrae il primo processo dalla coda dei processi pronti per metterlo in esecuzione.

## EDFSchedulingParameters



Ad ogni thread gestito con politiche EDF deve essere associata un'istanza della classe EDFSchedulingParameters. Questa classe, che estende la classe PriorityParameters, ha come unico campo un riferimento temporale assoluto alla prossima deadline. I job vengono ordinati proprio in base a questo valore.

## Modifiche a PeriodicThread

Dal momento che lo scheduler in Java real-time non riceve eventi nè quando c’è una nuova release nè quando il job termina la sua esecuzione e che è a tutti gli effetti un oggetto passivo, è necessario che sia la classe PeriodicThread ad eseguire un preambolo ed un epilogo per ogni job.

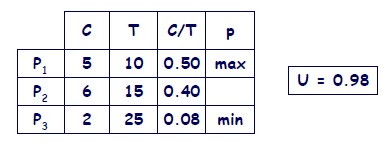
Al fine di poter prevedere l’implementazione di una molteplicità di scheduler, si è fatto sì che nella classe PeriodicThread esistano tanti metodi run privati quanti gli scheduler supportati. Il metodo run pubblico, quello ereditato dalla classe Thread, ha lo scopo di ridirigere l'esecuzione verso il metodo run privato appropriato in base allo scheduler associato al thread.

Il metodo run per PriorityScheduler non verrà, quindi, in alcun modo modificato.

Il metodo run per EDFScheduler contiene, per ogni iterazione, oltre alle scritture sul log, le chiamate al preambolo ed all’epilogo dello scheduler . Il preambolo onJobRelease e l’epilogo onJobEnd circondano la chiamata del metodo doJob che contiene la business logic che il job deve eseguire. Dal momento che si era deciso di separare la business logic dal resto del codice del metodo run questo può rimanere inalterato indipendentemente dallo scheduler utilizzato.

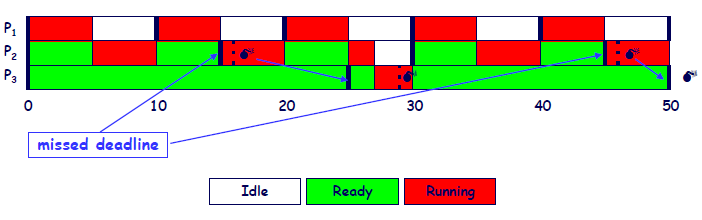
## Risultati sperimentali

Per confrontare le performance dello scheduler EDF con quelle dello scheduler basato su priorità statica fornito dal sistema si prende la seguente applicazione di esempio:

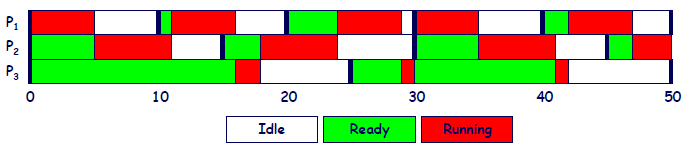


L’unità di misura temporale è il centesimo di secondo (10 millisecondi). Il parametro C indica la durata dell’esecuzione di ogni job dei vari thread; il parametro T indica la durata dei periodi; p indica la priorità (P1 è il processo più prioritario con a seguire P2 e P3); C/T indica il fattore di utilizzazione del processore imputabile al singolo processo, la cui somma è 0,98.

Come mostra il grafico sottostante,con la politica basata su priorità statica offerta dal sistema l’applicazione non è schedulabile in quanto si verificano numerose violazione di deadline. A tale proposito si anticipa che la politica di gestione dei casi di deadline miss di defalut per il sistema prevede di schedulare immediatamente il job successivo a quello che ha violato la deadline.



Con la politica EDF attuata dallo scheduler creato non c’è alcuna violazione di deadline, come mostrato nel grafico seguente ed in accordo con la regola che vuole che ogni insieme di processi periodici con fattore di utilizzazione non superiore a 1 sia schedulabile con EDF



La gestione dei deadline miss in Java Realtime

Questo capitolo ha l’obiettivo di analizzare nello specifico il comportamento di defalut di Real Time Java System in caso di violazione di deadline, con specifica enfasi sui processi periodici. Ne verranno sottolineati le maggiori criticità, le quali verranno affrontate nei capitoli successivi.

# Parametri associati ad un processo periodico

Come già spiegato nell’analisi della specifica RTSJ, è possibile caratterizzare un RealtimeThread come un processo periodico associandogli dei PeriodicParameters. I periodicParameters permettono di specificare:

* **Start time** un RealtiveTime che indica con quanto ritardo il processo deve essere lanciato rispetto al momento in cui se ne invoca il metodo start(); oppure un AbsoluteTime che indica il momento esatto in cui il processo deve essere avviato. Se non specificato il suo valore è zero, il che significa che il thread viene immediatamente avviato una volta chiamato il metodo start()
* **Cost:** un RelativeTime che indica il massimo tempo stimato di esecuzione in ogni ciclo del thread. Dal momento che l’implementazione RTJS non supporta le funzioni di cost eforcement, questo valore non viene utilizzato.
* **Overrun handler** l’handler da richiamare se il thread esegue per più tempo rispetto a quanto specificato. Dal momento che l’implementazione RTJS non supporta le funzioni di cost eforcement, l’handler non viene mai richiamato.
* **Period**  un RelativeTime che indica il periodo del thread. Deve essere specificato obbligatoriamente
* **Deadline** un RelativeTime che indica la deadline del thread. Deve essere pari o minore al periodo. Se non specificata viene posta pari al periodo
* **missHandler** l’handler che viene richiamato in caso di violazione della deadline. Se non specificato il sistema non richiama nessun handler e si limita a rischedulare il thread.

# Comportamento in assenza di handler

Se non viene specificato alcun handler il sistema adotta una politica AsSoonAsPossible che prevede di schedulare immediatamente il prossimo job. Ciò deriva dal comportamento del metodo *waitForNextPeriod* che ha il compito di bloccare il thread fino all’inizio del prossimo periodo. Se questo istante è passato (e di conseguenza il thread ha sforato la deadline) la chiamata al metodo è passante. *waitForNextPeriod* ritorna false se il thread ha violato la deadline, permettendo di diversificare l’esecuzione in base a ciò e di eseguire eventuali azioni correttive all’interno del thread.

Come mostrato dalla figura sottostante, qualora l’esecuzione sia particolarmente lunga, tanto da violare consecutivamente più deadline (due nel caso in figura), l’esecuzione del job non viene mai interrotta e, appena questa termina, vengono recuperati tutti i job “pendenti” fino a tornare ad una situazione ordinaria.

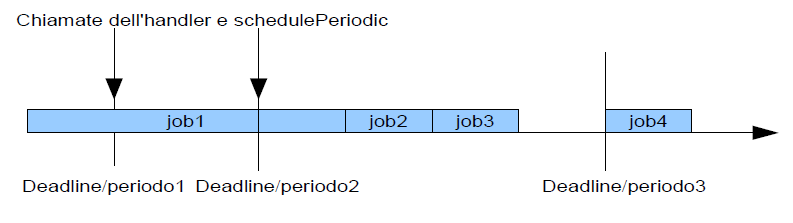


# Comportamento in presenza di un handler associato al thread

Quando si verifica un deadlineMiss,, evento che viene lanciato quando un thread che è attivo (ossia in esecuzione, pronto per l'esecuzione o bloccato nell'accesso ad una risorsa) supera la deadline specificata nei suoi ReleaseParameters, viene schedultato, se presente, l'handler associato per quel thread (se entra in esecuzione o meno dipende dalla priorità).

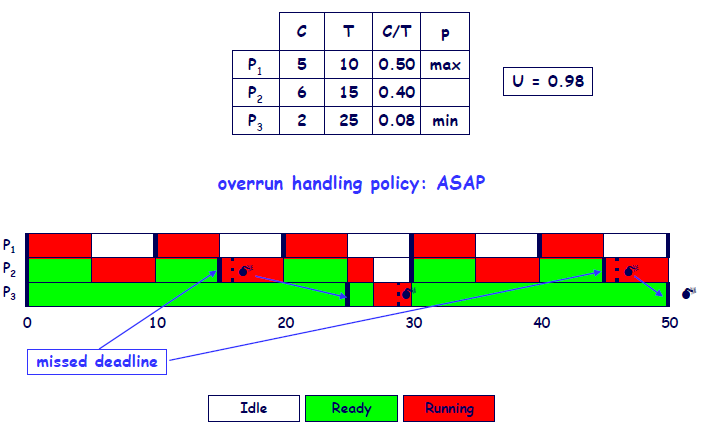
Se non viene chiamato il metodo *schedulePeriodic* sul thread che ha violato la deadline quest’ultimo rimane bloccato sulla chiamata al metodo *waitForNextPeriod.* La differenza di comportamento è motivata dal fatto che, dal momento che si presuppone che sia l’handler a dovere assumere azioni correttive, l’handler stesso provvedere a sbloccare il thread solo dopo che queste azioni siano terminate.

Se l’handler provvede correttamente a rischedulare il thread, il comportamento osservabile è analogo al caso in cui non ci sia alcun handler specificato, come mostrato dalla figura seguente.



# Criticità della Politica ASAP

Si ripresenta l’applicazione di esempio già utilizzata nel capitolo dedicato alla realizzazione di uno scheduler EDF al fine di esaminare un importante aspetto negativo della politica ASAP.



Si può notare come le violazioni di deadline di P2 causino un effetto domino che fa sì che anche P3 non rispetti i suoi vincoli temporali. Infatti, dopo che un job di P2 termina in ritardo la sua esecuzione, viene immediatamente schedulato il job successivo che causa anch’egli preemption suP3, ritardandone ulteriormente l’esecuzione.

Astraendo dall’esempio si può dichiarare che la politica ASAP, nel tentativo di “rincorrere” i job persi, spesso causa molto sovraccarico nel sistema con un effetto domino che porta al verificarsi di altre violazioni di deadline.

Un altro aspetto critico della politica di default implementata dal sistema è che non esiste nessun controllo sulla durata di un job. Nel caso in cui un thread a massima priorità entri in deadlock, questo occuperà la cpu per un tempo indefinito di tempo, interrompendo, dal momento che esercita preemption su di essi, l’esecuzione di ogni altro processo.

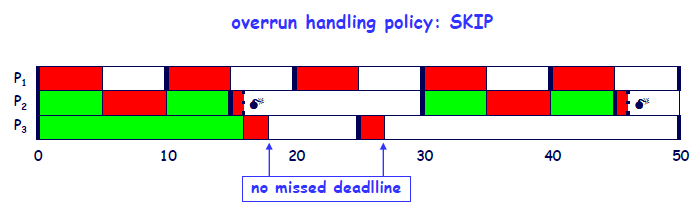
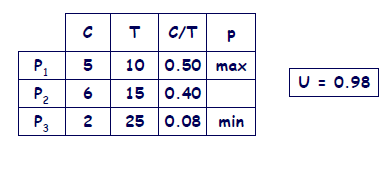
Nei capitoli successivi saranno presentate e realizzate due strategie che risolvono le criticità di ASAP qui sottolineate. Va comunque ricordato come non esista la politica perfetta di gestione dei deadline miss, in quanto ognuna presenta pregi e difetti e sia più adatta ad alcuni scenari applicativi piuttosto che ad altri.

Una nuova strategia di gestione dei deadline miss: la politica SKIP

Col fine di evitare che la violazione di una deadline da parte di un processo causi un effetto domino dovuto all’immediata release dei job pendenti, si è realizzata la politica SKIP, che prevede di saltare i job pendenti. In questo capitolo si analizzerà la politica in questione e si illustrerà come sia stata realizzata sull’infrastruttura Java real – time.

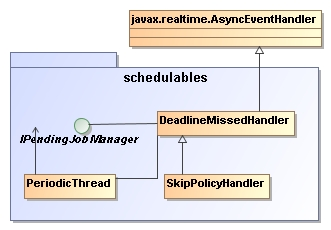
# Presentazione della politica SKIP

La politica SKIP prevede di saltare i job “pendenti” a causa di un deadline miss e di ritornare alla schedulazione dei job soltanto a partire dal periodo successivo rispetto al termine del job che ha causato un deadline miss, col vantaggio di non causare ulteriore sovraccarico nel sistema con il conseguente effetto domino di deadline miss. Riprendendo l’applicazione già utilizzata nell’analisi della politica ASAP si può osservare come la mancata schedulazione dei job di P2 successivi alle due violazioni di deadline permetta al thread P3 di eseguire regolarmente senza nessun deadline miss



# L’implementazione della politica SKIP.

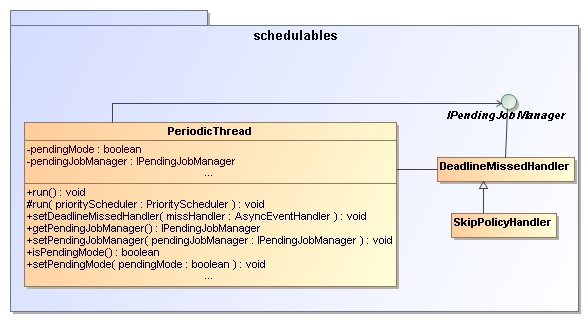
Dal momento che il sistema impone una politica ASAP ci sono due momenti in cui inserire delle misure di controllo: nel momento stesso in cui il thread sfora la deadline e nelle release dei job pendenti del thread schedulate dal sistema secondo la politica ASAP, che la politica skip prevede di saltare brutalmente, ma per i quali, in un caso più generale, si può prevedere comunque di prevedere un comportamento differenziato. Come già affermato nel capitolo di analisi della specifica Java Real Time, la responsabilità di intervenire nel momento in cui si verifica lo sforamento della deadline è dell'handler di deadlineMiss associato al thread, il quale viene appositamente risvegliato dal sistema. Si è deciso che la seconda fase sia responsabilità di un oggetto che implementa un'interfaccia (IpendingJobManager) e che contiene la logica da eseguire nei job di recupero. Nella implementazione attuale questo oggetto è l’handler stesso. Questo approccio ha il vantaggio di racchiudere in una sola entità tutta la gestione dei deadline-miss. Di conseguenza si può attribuire una certa politica ad un thread semplicemente associandogli un determinato handler. Come nei capitoli precedenti, si mostra un diagramma statico che formalizza l’architettura generale del sistema sviluppato. Di seguito, verranno esaminate nel dettaglio le entità coinvolte.



## L’interfaccia IPendigJobManager

Le classi che implementano questa interfaccia sono deputate a contenere la logica da eseguire nei di job “di recupero”. L'interfaccia è fatta dal solo metodo solo doPendingJob. Questo metodo è pensato perché sia invocato dal thread nei job pendenti, svolgendo un ruolo analogo a quello svolto dal metodo doJob interno alla classe thread nei job ordinari

## Modifiche alla classe PeriodicThread



La classe PeriodicThread è stata modificata modificata per accogliere le nuove strategie di gestione dei deadline miss. Ha, di conseguenza, un riferimento all’oggetto che implementa l’interfaccia IPendingJobManager che rappresenta il gestore dei job pendenti.

Il thread deve è ora in grado discriminare tra l’esecuzione di un job ordinario e l’esecuzione di uno di recupero. Per fare ciò si basa sul flag *pendingMode* che viene settato al valore appropriato dall’handler di deadline miss.

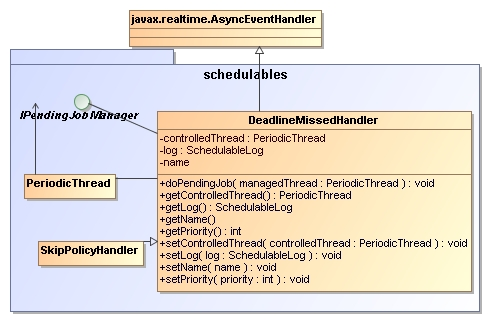
Si è reso, quindi necessario modificare in maniera analoga i metodi *run* relativi ai due scheduler finora supportati: EDFScheduler e PriorityScheduler. Per ogni ciclo di esecuzione (job), se il valore del flag della release di recupero è pari a false (quindi si tratta di un'esecuzione “normale”), viene scritto l'inizio e la fine del job sul log e viene eseguito il metodo *doJob* deputato a contenere contiene la businness logic dell'esecuzione del thread. Questo metodo, si ricorda, non fa altro che eseguire una busyWait di durata pari al tempo di esecuzione indicato nella relativa proprietà.

Se, al contrario il flag è false, si richiama il metodo opportuno del gestore (pendingReleaseJob), se questo è presente. In caso contrario, presupponendo che sia da adottare la politica di default per il sistema (ASAP), si invoca il metodo *doJob*.

Come ultima operazione, indipendentemente se si è eseguito un job di recupero o uno ordinario viene invocato il metodo *waitForNextPeriod*.

Infine, sono stati inseriti due metodi di utilità per settare o reperire il gestore di deadline miss che operano provvedendo ad invocare gli opportuni metodi dei ReleaseParameters associati al thread.

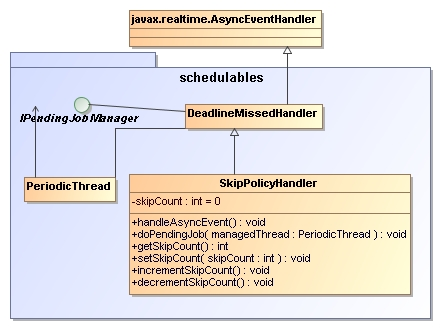
## DeadlineMissedHandler



E’ la classe base che rappresenta un generico handler per la gestione di deadline miss. Questa classe estende direttamente la classe AsyncEventHandler, classe di sistema deputata a gestire gli eventi asincroni, quali sono gli sforamenti di deadline da parte dei thread real-time. Le proprietà introdotte sono il nome, il log ed il riferimento al thread che deve gestire. Dal momento che si è deciso di accorpare nei gestori di deadline miss la totale implementazione della politica, questa classe implementa l’interfaccia **IPendingJobManager** ed esibisce il metodo *doPendingJob*. Per lo stesso motivo, il metodo *setThread*, provvede anche ad indicare al thread stesso che l'handler è anche il gestore per i job di recupero chiamandone il relativo setter.

Infine, sono previsti due metodi, *GetPriority* e *setPriority*, per rendere più comodo la gestione della priorità dell'handler. Questi metodi provvedono rispettivamente a restituire o ad impostare la priorità alla quale eseguirà l’handler quando viene invocato in caso di violazione della deadline invocando gli opportuni metodi dei ReleaseParameters associati all’handler stesso.

## SkipPolicyHandler



La classe SkipPolicyHandler rappresenta l’handler che implementa la politica Skip.

Questo gestore di deadline miss estende la classe deadlineMissedHandler illustrata in precedenza.

Dal momento che il sistema di default implementerebbe la politica ASAP, rischedulando immediatamente un numero di thread pari al numero di deadline violate consecutivamente, l’handler deve memorizzare questo valore. La proprietà skipCount espleta questo compito: viene incrementata quando si verifica un deadline miss (nel metodo handlerAsyncEvent) e decrementata al termine di ogni job di recupero.

Entrando nel particolare, quando viene richiamato il metodo HandleAsyncEvent a fronte di uno sforamento di deadline, oltre a scrivere l'evento sul log ed a rischedulare il thread, l'handler imposta il flag pendigMode del thread a true e incrementa il suo contatore skipCount, che tiene traccia di quanti job in futuro si dovranno saltare.

Dualmente, il metodo doPendingJob, oltre a scrivere sul log che si è saltato il job, decrementa il contatore dei job da saltare e, se questo è arrivato a zero, imposta il flag pendingMode del thread controllato a false, in modo che il thread possa riprendere la sua esecuzione ordinaria .

Un ulteriore miglioramento: la politica skipStop

La politica SKIP presentata nel caso precedente aiuta ad alleviare il sovraccarico nel sistema in presenza di thread che non rispettano le deadline stabilite. Non risolve la seconda criticità riscontrata nella politica ASAP implementata di default in Java real-time: la possibilità di un thread a massimo livello di eseguire per un tempo indefinito, violando un numero teoricamente infinito di deadline, e di bloccare di conseguenza tutto il sistema. Sorge dunque la necessità di creare un’entità che sia in grado di comportarsi da watch dog nei confronti dell’esecuzione di un thread, interrompendola se questa presenta una durata eccessiva e pericolosa per il sistema.

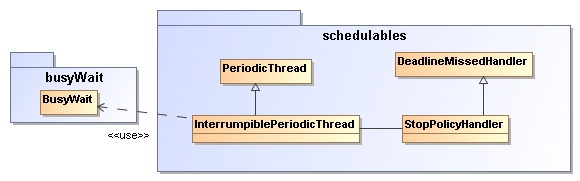
In questo capitolo si mostra come si è soddisfatta questa necessità sfruttando il meccanismo del trasferimento asincrono di controllo. Si partirà presentando la politica di stop, se ne analizzeranno i limiti e si introdurranno i gestori a soglia, in particolare verrà illustrata la politica SkipStop con la relativa implementazione.

# La politica Stop

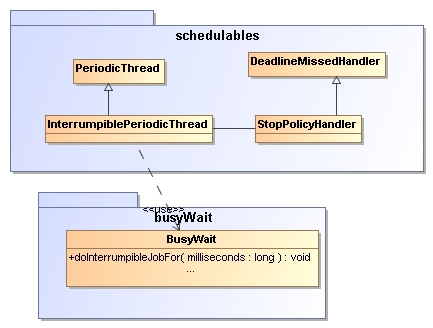
Al fine di non permettere alcuna violazione di deadline, si può prevedere una politica che interrompa, preferibilmente consentendo un epilogo che garantisca un’interruzione sicura, ogni job che violi la propria deadline. In questo modo nessun processo sarà bloccato a causa della preemption esercitata da un altro thread a priorità superiore che ha violato la deadline.

Come già detto nel capitolo di presentazione della specifica, il trasferimento asincrono di controllo permette di interrompere un thread che sta eseguendo un metodo interrompibile tramite l’invocazione del metodo interrupt. A questo punto il thread riceve una AsyncronouslyInterruptedException, che deve raccogliere e gestire di conseguenza.

Come in precedenza, si mostra prima un’overview generale dei componenti del sistema implementato, per poi entrare nello specifico.

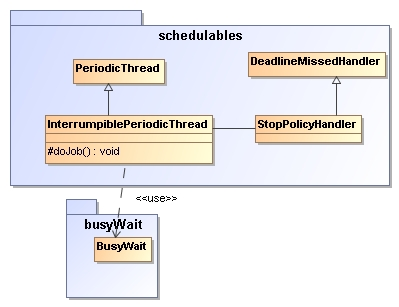


## BusyWait



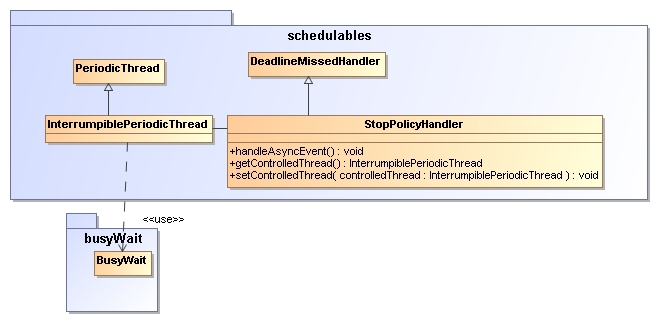
Il metodo *doInterrumpibleJobFor* è stato pensato appositamente per permettere l’esecuzione di una busy wait interrompibile di durata desiderata. Questo metodo è identico al metodo doJobFor, la sola differenza è che dichiara una AsyncrnonouslyInterruptedException nella sua thow-list e può, di conseguenza essere interrotto asincronicamente.

## InterrumpiblePeriodicThread



La classe InterrumpiblePeriodicThread estende PeriodicThread e modella un thread periodico la cui esecuzione può essere interrotta. Ciò è fatto eseguendo, nel metodo do job, una busy wait interrompibile al posto di una “normale”. L’ azioni da eseguire nel caso di interruzione, inserita nella catch della AsynchronouslyInterruptedException, è semplicemente la scrittura sul log dell’ avvenuta interruzione.

## StopPolicyHandler



La classe stopPolicyHandler modella la politica di gestione dei deadline miss secondo la quale, quando un job viola la sua deadline, questo debba essere interrotto. Perché ciò sia possibile occorre che il thread controllato possa ricevere un'interruzione dall'handler quando quest'ultimo viene invocato dal sistema in occasione del deadline miss. Occorre, quindi che il thread controllato da questo tipo di handler sia un InterrumpiblePeriodicThread. A tale scopo si sono modificati sia il costruttore, sia la proprietà controlledThread al fine di supportare solo un InterrumpiblePeriodicThread e non il generico PeriodicThread.

Il metodo handleAsyncEvent, oltre a registrare sul log la violazione della deadline, interrompe il lavoro del thread invocando metodo interrupt del thread controllato, il quale sta eseguendo una busyWait Interrompibile, al fine di causare l'interruzione del job. Occorre, infine, rischedulare il thread affinché un nuovo job possa partire nel periodo successivo.

La politica stop, interrompendo i thread non appena questi violano la deadline, non prevede l’esistenza di job pendenti, di conseguenza il metodo doPendingJob non è implementato, né è mai sfruttato il flag pendingMode del thread controllato.

# Le politiche a soglia e la politica SkipStop

La politica Stop presentata nel paragrafo precedente ha il difetto di essere estremamente severa nei confronti dei thread che violano le deadline. Occorre osservare come, nella maggior parte dei sistemi in tempo reale, la violazione dei vincoli temporali da parte di un thread, a patto che si tratti di episodi saltuari e di dimensione limitata, è un fenomeno tollerabile. In ogni caso, il fenomeno del blocco dell’intero sistema a causa dell’esecuzione prolungata del sistema resta non accettabile.

In generale, è positivo dare ad un sistema la possibilità di intervenire in modo diverso in risposta ad una violazione estemporanea dei vincoli temporali piuttosto che ad un thread che, con una esecuzione sensibilmente più lunga del previsto, rischia di bloccare l’intero sistema.

Si è quindi pensato a dei gestori in grado di cambiare strategia in base al numero di deadline violate consecutivamente dallo stesso job. Nel caso particolare sviluppato e presentato in questo capitolo si mostra il gestore SkipStop. Questo gestore prevede di interrompere un job quando viola consecutivamente un numero di deadline pari o superiore ad una soglia prestabilita. Analogamente a quanto accade con la politica skip, non viene recuperato nessun job perso al fine di limitare la congestione del sistema . Gli esempi mostrati di seguito aiutano a comprendere la politica. In entrambi la soglia di interruzione è di tre violazioni di deadline consecutive da parte dello stesso job

Job1

Job2

Job3

Deadline / periodo1

Deadline / periodo2

Deadline / periodo3

Deadline / periodo4

Nell’esempio il job1 viola due deadline consecutive, mentre la soglia è di tre. Di conseguenza il job non viene interrotto; tuttavia, in accordo con la politica skip, non viene schedulato nessun altro job nel periodo nel quale job1 termina l’esecuzione. L’esecuzione periodica riprende regolarmente nel periodo successivo.

Job1

Job2

Deadline / periodo1

Deadline / periodo2

Deadline / periodo3

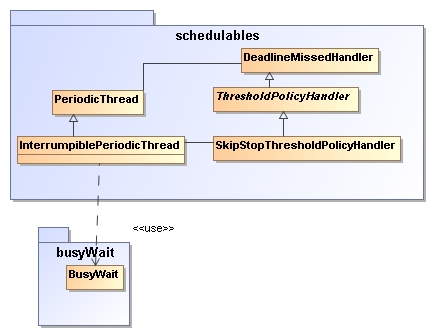
Deadline / periodo4

Chiamata handler ed interruzione thread

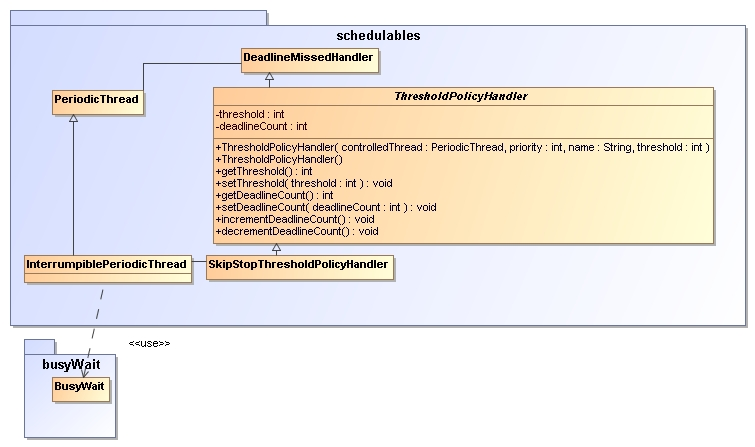
In quest’esempio, invece, l’esecuzione di job1 supererebbe anche la terza deadline. Alla terza violazione consecutiva da parte dello stesso job, tuttavia, l’handler provvede ad interromperlo. Si fa notare come, dal punto di vista degli altri thread ,se fosse immediatamente schedulato il prossimo job dello stesso thread non ci sarebbe alcun beneficio, visto che il thread continuerebbe ad esercitare preemption come se continuasse l’esecuzione del vecchio job. Per questo motivo si è previsto che non vengano rilasciati altri job del thread che è stato interrotto nel periodo successivo all’interruzione stessa. L’esecuzione periodica riprende regolarmente nel periodo successivo.

## Implementazione della politica

Si mostrano i componenti che permettono la realizzazione della politica SkipStop. Si mostra prima un diagramma statico in grado di offrire una panoramica sull’architettura di questi componenti, successivamente gli stessi componenti verranno illustrati nel dettaglio.



### ThresholdPolicyHandler

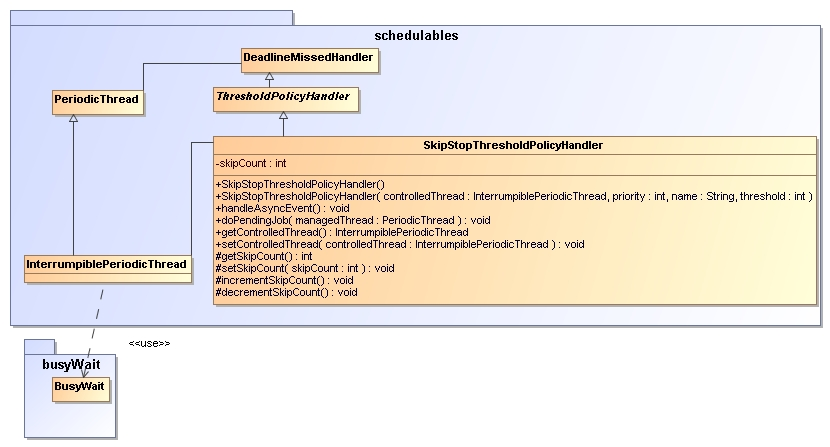


La classe astratta ThresholdPolicyHandlerè la classe base di tutti i gestori a soglia ed estende direttamente la classe madre DeadlineMissedHandler.

Ha una proprietà pubblica threshold: è un intero che indica dopo quante violazioni consecutive di deadline l’handler deve cambiare strategia.

Ha un'altra proprietà intera: deadlineCount, ossia il contatore che tiene traccia di quante volte la deadline è stata violata consecutivamente, se questo valore raggiunge o supera la soglia, si deve usare la seconda politica

### SkipStopThresholdPolicyHandler



Questa classe estende ThresholdPolicyhandler e modella un gestore che, finchè un job non sfora consecutivamente più di un valore prestabilito di deadline, adotta la politica skip, non eseguendo alcun job nel periodo in cui termina quello che aveva sforato la deadline. Tuttavia, qualora il job sfori almeno tante deadline quante la soglia, allora si decide di interromperlo.

Dal momento che il sistema adotta nativamente una politica asap, bisogna assicurarsi che, una volta interrotto il job, non si eseguano i job di recupero, ma li si salti, in maniera analoga a quanto avviene con la skip.

Occorre quindi memorizzare in una apposita proprietà (skipCount) il numero di release da saltare. Si fa notare come non sia possibile utilizzare la proprietà deadlineCount della classe madre in quanto questa, in caso di interruzione del thread controllato, deve essere azzerata al fine di garantire che il job successivo venga interrotto solo dopo avere violato anch’esso un numero di deadline pari o superiore alla soglia.

E' necessario che il thread controllato sia un InterrumpiblePeriodicThread, di conseguenza, è necessario adeguare i parametri del costruttore, del getter e del setter di tale proprietà in modo che accettino e restituiscano istanze di questa classe.

Nel metodo handleAsyncEvent si scrive sul log l'evento di deadline miss e si mette il flag pendingMode del thread controllato a true; poi si deve incrementare deadlineCount e skipCount. Infine si controlla se il numero di violazioni consecutive (memorizzato in deadlineCount) è superiore alla soglia, in questo caso si azzera il contatore deadlineCount e si interrompe il thread

Dal momento che la stop di per se non prevede release pendenti e cha la gestione è comunque affidata secondo una tecnica skip, nel metodo doPendingJob si scrive sul log che il job è stato saltato, si decrementa il valore di skipCount e, se questo ha raggiunto il valore zero, si setta il flag pendingMode del thread controllato al false in modo che riprenda la sua esecuzione “normale”.

Conclusioni e sviluppi futuri

I moderni sistemi in tempo reale affrontano nuove problematiche, dovute ad una crescente complessità. Emerge quindi la necessità di ricorrere a linguaggi più strutturati che, grazie ad un maggiore livello di astrazione, aiutano lo sviluppo di sistemi di maggiori dimensioni.

La piattaforma Java e le sue diverse versioni sono estremamente diffuse, tanto da essere impiegate in ambiti molto diversificati, dalla gestione di transazione sicure a quella di flussi multimediali, senza dimenticare le piattaforme mobili ed i sistemi embedded.

La diffusione di Java nei sistemi real-time è stata fortemente limitata dal fatto che i tempi di esecuzione di un’applicazione Java sono difficilmente predicibili.

La specifica Java Real Time propone una serie di modifiche a Java standard in grado di renderla una piattaforma efficace anche per sistemi in tempo reale.

In questa tesi si è analizzata la specifica RTSJ e si è valutata la piattaforma Real Time Java System (RTJS) sviluppata da Sun-Oracle per il sistema operativo Solaris 10.9. In seguito sono state sviluppate una serie di funzionalità atte a migliorare le performance del sistema.

In primo luogo sono stati creati dei moduli che fornissero delle funzionalità necessarie per capire a fondo il comportamento del sistema. In particolare si è sviluppato un componente in grado di eseguire una busyWait dalla durata desiderata calibrandosi sulle capacità computazionali del sistema sul quale sta eseguendo. E’ stato, inoltre, creato un sistema di logging che permette di tenere traccia delle attività del sistema, senza che quest’attività intralci in modo significativo le performance dello stesso. Ciò è stato realizzato associando ad ogni entità coinvolta una zona di memoria nella quale le entità stesse possono scrivere liberamente senza incorrere in problemi di sincronizzazione su risorse condivise. Ad esecuzione terminata una serie di utility permettono di aggregare le varie scritture in memoria e di presentarle all’utente in una forma user-friendly.

Il resto lavoro è concentrato principalmente sull’attività di scheduling e di gestione della violazione di deadline da parte dei thread real-time.

Nel primo campo si è confermato come l’unico algoritmo di schedulazione supportato dal sistema sia la schedulazione priority driven, si è verificato quindi come spetta all’utente specificare staticamente il livello di priorità ed il sistema provvede a porre in esecuzione il thread a priorità maggiore. Con riferimento ad un ambiente di esecuzione monoprocessore, si è creato uno scheduler che opera secondo la strategia EDF, la quale prevede di porre in esecuzione il thread che ha la prossima deadline assoluta più imminente. Lo scheduler lavora mantenendo i thread pronti per l’esecuzione in una struttura dati ordinata per deadline assoluta e ponendo il thread con deadline più imminente a priorità superiore, in modo da garantirne l’esecuzione. Si è dimostrato come la schedulazione, grazie al protocollo priority inheritance fornito da java real-time, sia robusta a blocchi dell’esecuzione del thread a priorità più imminente dovuti all’accesso a sezioni di codice ad accesso mutuamente esclusivo. Si è anche dimostrato come, in accordo con la teoria, lo scheduler sviluppato permetta la schedulazione senza violazione di deadline di un insieme di applicazioni più ampio rispetto a quanto consentito dal protocollo Rate Motonic Priority Order derivabile dall’utilizzo delle funzionalità di default del sistema.

Nel campo della gestione della violazione dei vincoli temporali da parte dei processi di java real-time si è analizzato il potere espressivo delle classi fornite dalla specifica e si è mostrato come il sistema implementi nativamente una politica che prevede di schedulare immediatamente il job successivo a quello che ha violato la deadline (As Soon As Possible – ASAP). Si è mostrato come questo approccio rischi di sovraccaricare ulteriormente il sistema, inducendo altri thread a non rispettare la propria deadline. Inoltre si è evidenziato come Java Real-Time System non preveda alcun meccanismo di watch dog, con il rischio che un thread, con una esecuzione indefinitamente lunga, possa bloccare l’intero sistema. Si è provveduto, sfruttando le funzionalità messe a disposizione dal sistema, a creare una serie di gestori che permettono di risolvere queste criticità. In particolare si è creato un handler che implementa la politica skip, la quale prevede che il thread non possa eseguire nessun altro job nel periodo nel quale termina l’esecuzione del job che ha violato la deadline. Questa politica aiuta, quindi, ad alleggerire il carico computazionale, generalmente assai alto, nei casi di deadline miss, ma non risolve il problema della mancanza di controllo sul tempo di esecuzione massima di un job, con i conseguenti rischi di deadlock. Sfruttando il meccanismo dell’interruzione asincrona di controllo si è provveduto a creare un gestore che interrompe in modo sicuro l’esecuzione di un thread non appena questo viola una deadline. Dal momento che questo approccio è troppo poco permissivo per la maggior parte delle applicazioni, si è creato un secondo gestore che opera secondo una politica a soglia: viene tollerato il superamento di un numero di deadline inferiore ad una soglia stabilita dall’utente. Se il job viola tante deadline quante la soglia stabilita, allora l’esecuzione viene interrotta. Anche secondo questa politica non viene schedulato nessun job nel periodo successivo al termine di quello che non ha rispettato la deadline.

# Sviluppi Futuri

Prima di partire commentando i possibili sviluppi del lavoro portato avanti in questa tesi occorre sottolineare come si possa lavorare in due direzioni: per estendere le funzionalità sviluppate a scenari più generali di quello preso in esame, oppure per si introdurre nuove funzionalità oltre a quelle già implementate.

Si può pensare di adattare i componenti realizzati per sistemi dotati di più cpu, in questa tematica il componente maggiormente esposto è chiaramente lo scheduler EDF che dovrà essere in grado di mettere contemporaneamente in esecuzione tanti thread quante le unità di elaborazione presenti nel sistema.

Dal momento che tutte le politiche di gestione della violazione delle deadline sono pensate per un insieme di processi periodici, la più logica estensione è quella ai processi sporadici, ossia processi nei quali la deadline è antecedente all’inizio del periodo successivo. Occorre prestare particolare attenzione al caso in cui il processo viola sì la deadline, ma termina prima del prossimo istante di release. In questi casi sia la politica Skip, che la politica skipStop, debbono schedulare correttamente il job nel nuovo periodo.

Si possono prevedere, inoltre, una serie di funzionalità non ancora implementate. In primo luogo non si è affrontato l’argomento dell’analisi di schedulabilità, ossia fornire a priori indicazioni in merito al fatto che un insieme di processi sia schedulabile o meno. Si ricorda che la specifica prevede questo tipo di analisi, che deve essere svolta dallo scheduler, ma che nello scheduler di base non è implementata. In letteratura esiste un gran numero di algoritmi per questo compito, ognuno con caratteristiche distinte. Non è dunque da escludere che, in una futura realizzazione, per fornire la risposta più corretta sia necessario analizzare l’insieme di oggetti da schedulare, individuare in base alle caratteristiche di essi l’algoritmo più adatto ed infine applicarlo. Si può anche prevedere un supporto per i processi soft real-time. Attualmente questi vengono semplicemente eseguiti ad una priorità inferiore rispetto ad i thread real time con conseguente limitazione in termine di performance. In letteratura esistono diversi protocolli in grado di migliorare le performance dei processi non real-time senza intaccare il funzionamento dei processi real-time. Si possono implementare questi protocolli, tra i quali si citano Polling Server, Deferrable Server, Priority Excange Server, SporadicServer per gli scheduler a priorità statica. Constant Utilization Server e Total Bandwith Server Per gli scheduler a priorità dinamica, quali EDF.

Bibliografia

Glossario