Questo capitolo introduce le principali caratteristiche di 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 RTS(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 affinchè 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 che 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 che favoriscano 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 implementa tori 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 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 girare contemporaneamente 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

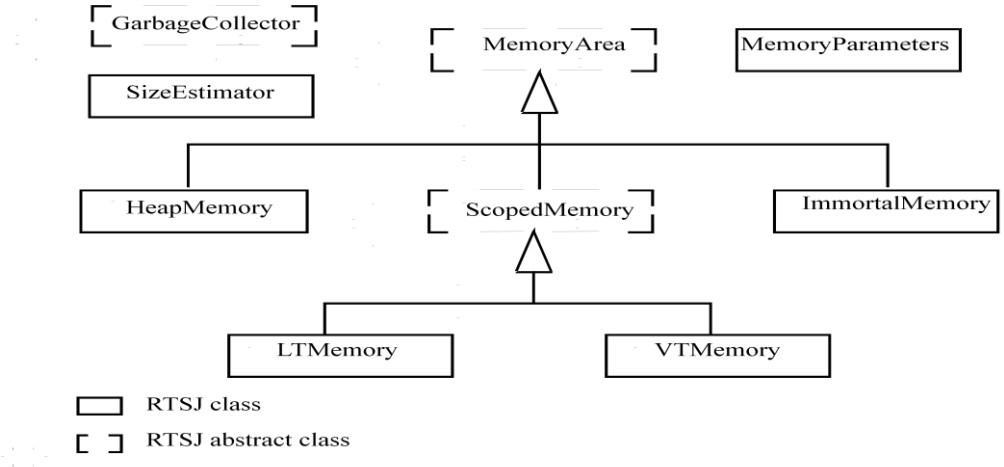
Estendibilità La specifica deve permettere l'utilizzo delle tecniche più conosciute per la gestione dei sistemi real time. Al contempo, deve permettere la futura implementazione di nuove tecniche e nuovi algoritmi

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 scopo o trade – off tra l'occupazione di memoria e velocità di esecuzione. In ogni caso la specifica si propone di non vincolare in nessun caso 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 tipi di ritardo 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.

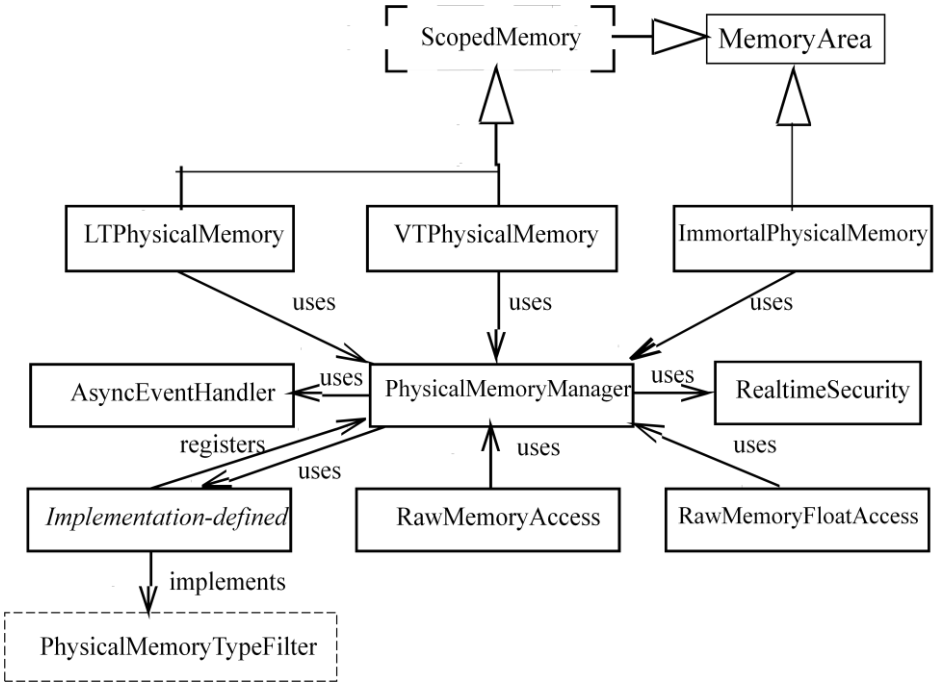


Java real - time permette di definire delle aree di memoria nelle quali allocare oggetti oltre all'heap tradizionale. Quando si entra in una area di memoria tutti gli oggetti vengono allocati all'interno di quest' ultima. Queste aree di memoria sono il già menzionato heap, la ScopedMemory e l'ImmortalMemory.

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' immortal memory 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 immortal memory 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.

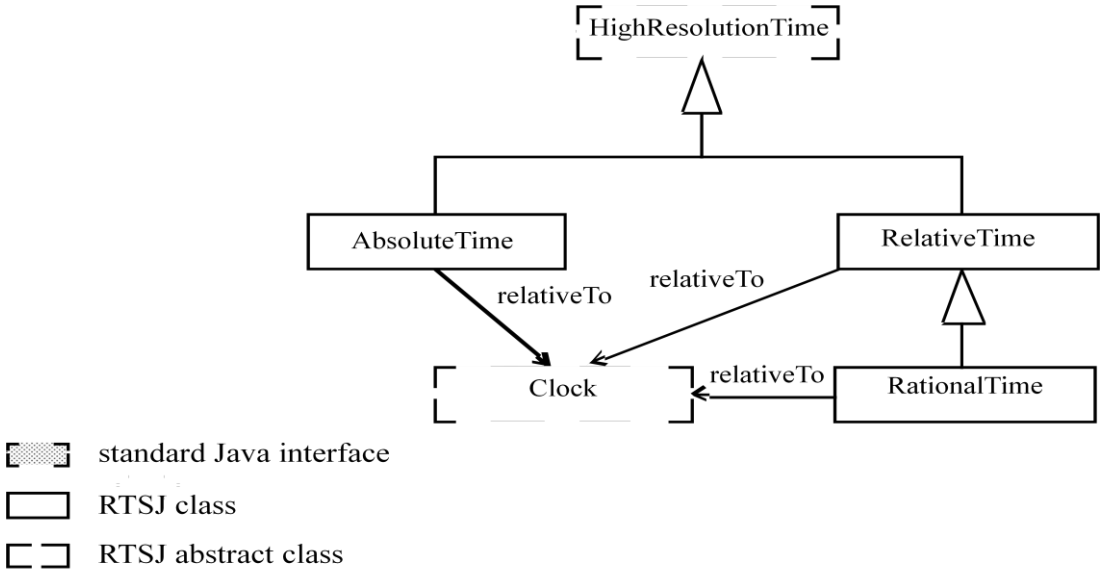
Per ottimizzare il sistema in caso di memorie con differenti caratteristiche può essere necessario lasciare alo sviluppatore la possibilità di decidere dove gli oggetti debbono essere memorizzati. La specifica prevede un'apposita estensione dei tre nuovi tipi di MemoryArea (LinearTime, VariableTime e ImmortalMemory) chiamati rispettivamente LTPhysicalMemory, VTPhysicalMemory, ImmortalPhysicalMemory. La gestione dell'accesso in lettura e scrittura alle zone di memoria è affidato ad un PhysicalMemory-Manager. 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) che 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 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 le 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 differenti. 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 inizializzare la con un valore che esprimesse il numero di iterazioni per secondo.

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.

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 con accesso mutuamente esclusivo.

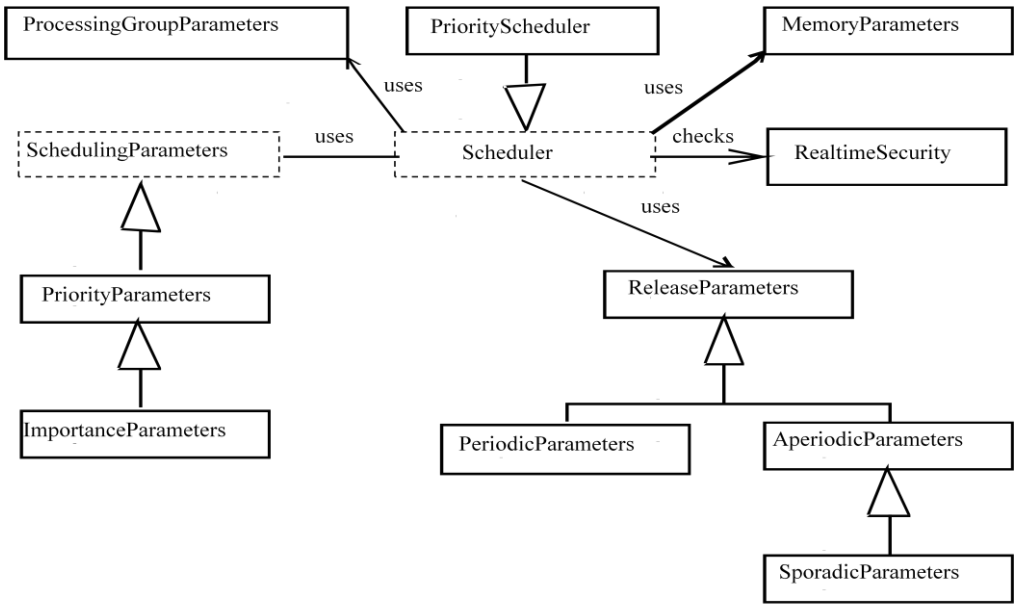
Tra le funzionalità accessorie rientrano il controllo dei tempi di esecuzione effettiva dei thread (cost enforcement) ed il protocollo priority ceiling per la gestione delle zone di codice con accesso mutuamente esclusivo.

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 scheduler abili 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 per concludere il paragrafo con 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 PriorityParamenters il processo dovrà immediatamente cambiare priorità; la stessa cosa deve accadere anche qualora si modifichi il valore numerico che indica la priorità all'interno dei PriorityParamentes stessi.

Nella figura sottostante si mostrano i parametri coinvolti nell'attività di scheduling , la classe astratta Scheduler e PriorityScheduler, ossia l'unico scheduler obbligatorio 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 un 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 con la 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 illustata 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 Schedueler. 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 schedulabilità; 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 anche con i nuovi parametri il sistema, alla luce delle potenziali nuove modifiche, resta feasible. Nonostante l’enfasi sull’analisi di feseability la specifica non richiede di realizzare nessuna analisi nel concreto, suggerisce, come algoritmo di default, di restituire sempre true per qualunque insieme di oggetti schedulabili. Inoltre, 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.

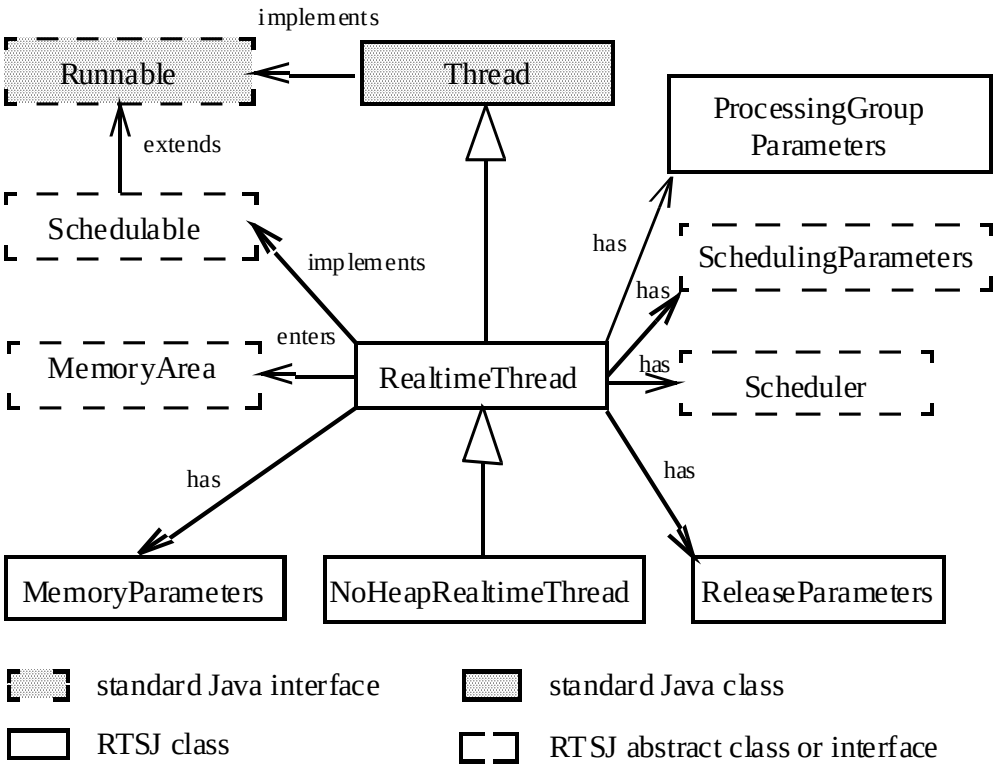
Lo scheduler di default per il sistema, ossia uno scheduler basato su priorità fissa, con almento 28 differenti livelli di priorità, è rappresentato dalla classe PriorityScheduler che estende direttamente la classe Scheduler. E’ un singleton, ciò significa che esiste una sola istanza di PriorityScheduler nel sistema, recuperabile attraverso un getter statico. Oltre a questo metodo ed a quelli ereditati dalla classe madre,

PrioritySchedueur offre la possibilità di ottenere i valori numerici della 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 Runnable. Ha molte proprietà che ne caratterizzano l'esecuzione.

- L e 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, gli vengono attribuiti 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 anche 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 si che alla prossima invocazione del metodo waitForNextPeriod il thread rimanga bloccato finché non viene chiamato il metodo scheudulePeriodic

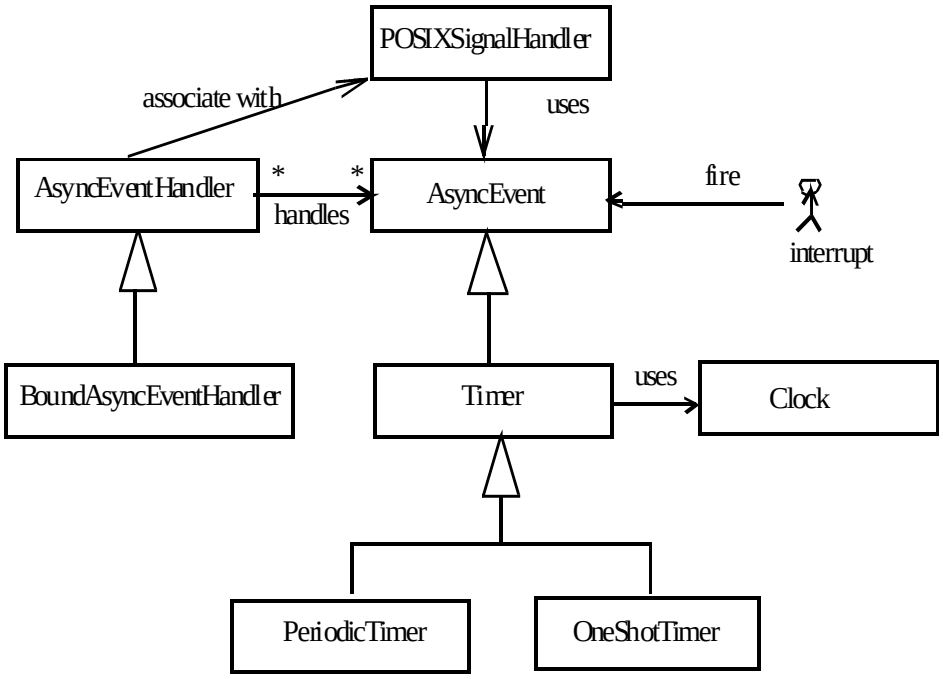
Il metodo run contiene il codice che il thread deve eseguire . Si fa notare come il metodo run venga invocato una volta sola durante l'esecuzione del thread, ossia quando viene avviato. Ciò accade anche per i thread periodici. Deve essere lo sviluppatore che, all'interno del metodo run, crea un ciclo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

Il funzionamento dei metodi della classe RealtimeThread, con specifica enfasi sul caso in cui un thread periodico viola la deadline stabilita, verrà discussa approfonditamente nella sezione relativa agli esperimenti condotti su java real – time.

Come già detto analizzando Java standard da un punto di vista real-time uno dei problemi maggiori è il fatto che il 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 (scoped ed immortal memory) 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. Nell'ottica degli autori della specifica, quindi, questa classe rappresenta thread con vincoli molto stringenti di rispetto delle deadline (hard real-time). I suoi costruttori accettano come parametro una aree 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 viene si verifica un evento vengono schedulati tutti i gestori ad esso associati i quali eseguono il metodo HandleAsyncEvent. Gli handler possono non entrare subito in esecuzione, ad esempio perché possono subire preemption da parte di processi più prioritari e nel frattempo possono verificarsi altri eventi a cui sono associati. Per questo motivo ad ogni verificarsi dell'evento a loro associato tutti gli handler vedono incrementato 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 implementazioni 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

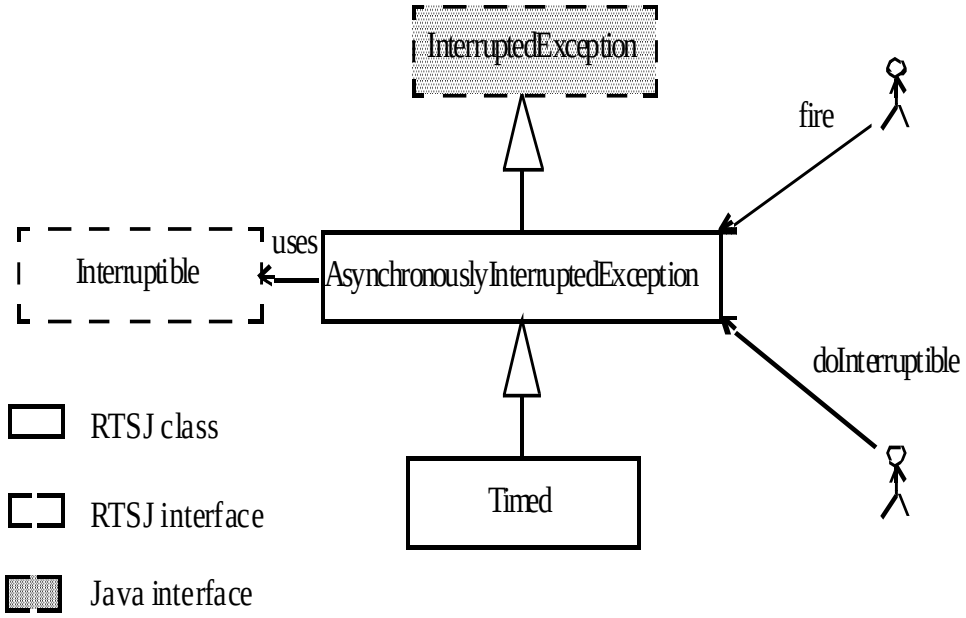
Trasferimento asincrono di controllo

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 sul poter 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 eseguire 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.

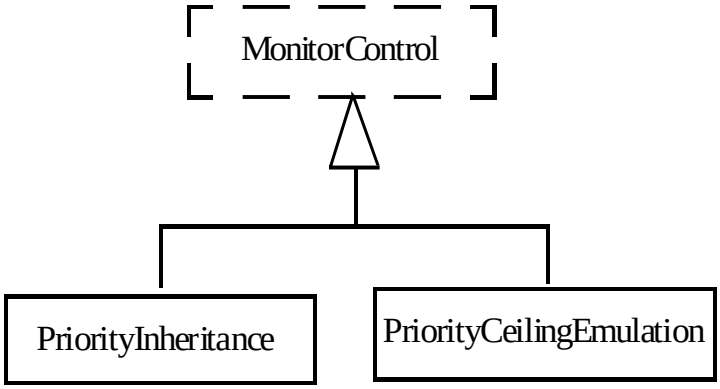
Il secondo approccio si basa sulla classe AsyncronouslyInterruptedException e sull'interfaccia Interrumpible In figura si mostrano le classi coinvolte in questo secondo sistema di gestione del trasferimento asincrono di controllo.



L'interfaccia Interrumpible contiene due metodi: il metodo run che contiene la 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.

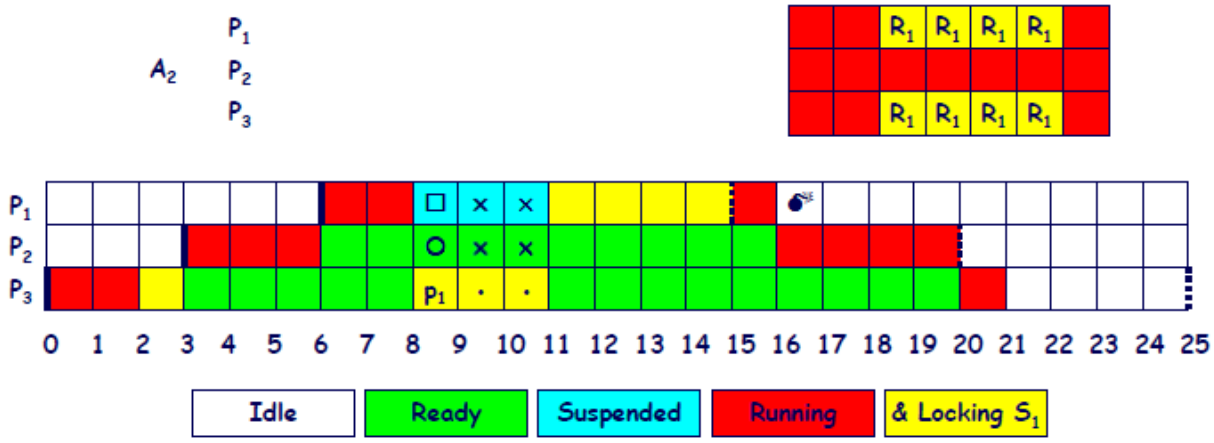
Sincronizzazione

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 capitolo "Perché java non è una piattaforma valida per i sistemi real-time" 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.

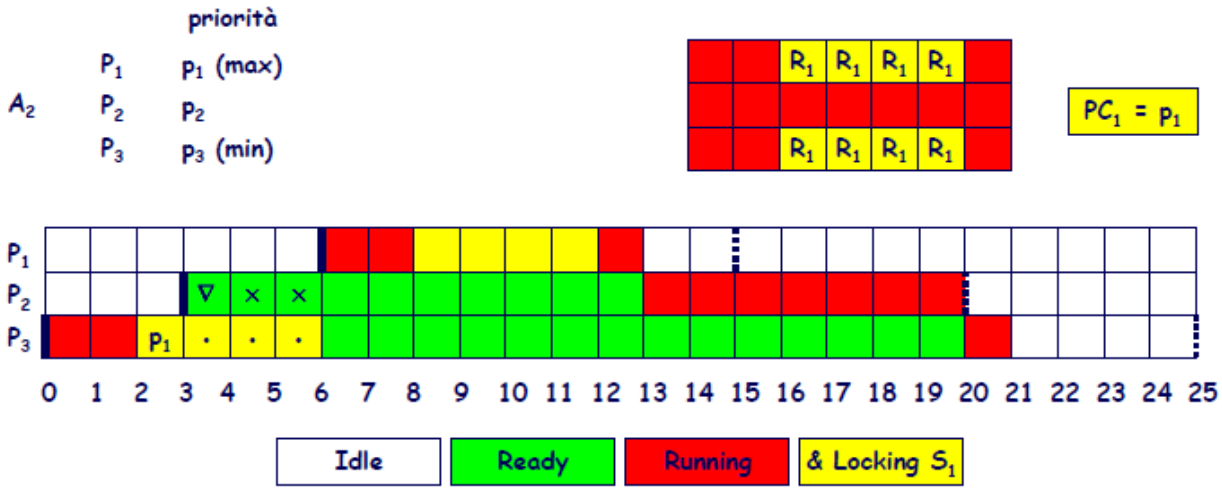


Come si può notare il processo p2 subisce preemption da p3 che ha ereditato la priorità da p1 e non può più bloccare indirettamente quest'ultimo.

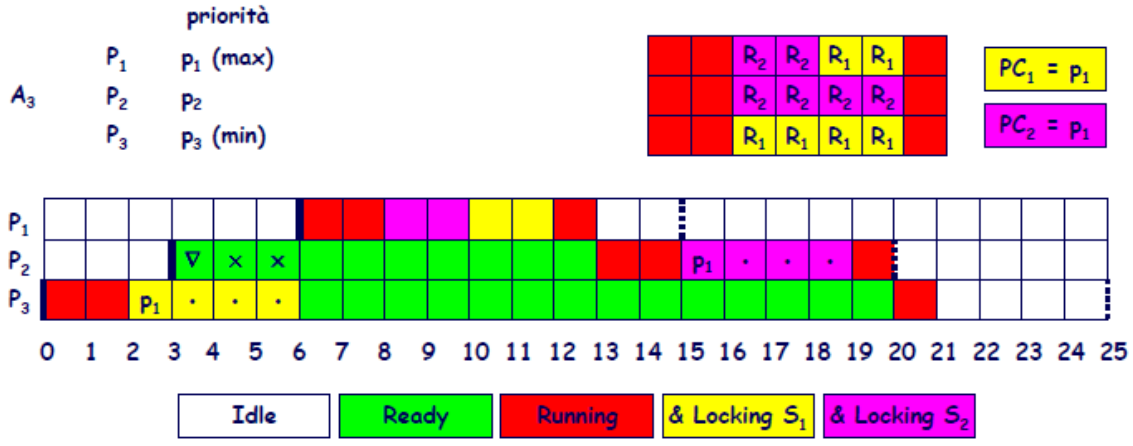
Come già detto priority inheritance è il protocollo di default del sistema, la classe Priorityinheritance è quindi pensata per reimpostare questa attraverso il metodo setMonitorControl della classe madre MonitorControl.

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 pari alla priorità maggiore tra quelle dei thread che vi possono accedere. 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. Per ogni oggetto coinvolto è necessario specificare il tetto di priorità tramite il metodo setMonitorControl. Se a run-time un processo che 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. 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 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

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 "perché Java non è una piattaforma valida per i sistemi real time" 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. A titolo di esempio, si mostra uno stralcio del file in questione

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, senza tuttavia 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 vengon eseguiti dai thread real time . A titolo di esempio, si mostra uno stralcio del file in questione

Garbage Collector

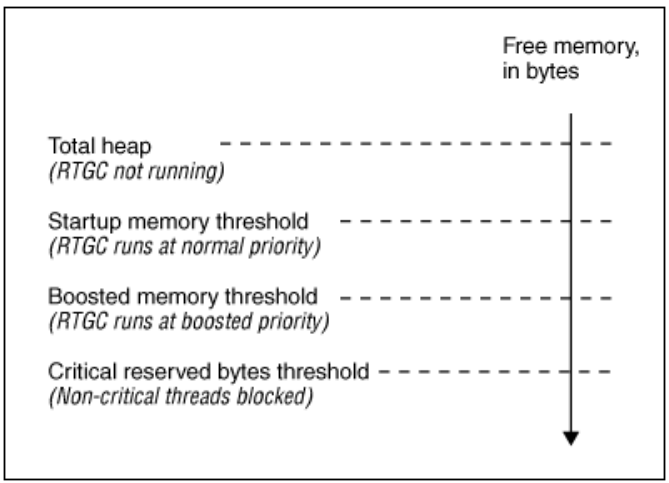
Occorre ricordare come l'attivita di Grabage Collection sia fondamentale per preservare l a funzionalità di un sistema Java: permette, infatti, di riciclare la memoria altrimenti occupata da oggetti non più utilizzati e di metterla a disposizione del sistema per l'allocazione di nuovi oggetti.

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 su i 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:

- Finchè la quantità di memoria libera nell'heap resta sopra la soglia di StartupMemory 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 (con priorità inferiore a quella critica) 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 - timea non critici per il sistema.