Chapter 7: The Real-time Specification for Java

Overview

Introduction and chapter structure

Since its inception in the early 1990s, there is little doubt that Java has been a great success. However, the language does have weaknesses both in its overall model of concurrency and in its support for real-time systems. Section 4.7 has summarized some of the main problems in the concurrency area. This chapter considers real-time systems.

First, the chapter reviews the activities from which the original motivation for the RTSJ developed and examines the National Institute of Standards and Technology (NIST) requirements for "real-time Java". Then the main enhancements to the Java platform are introduced. These take the form of an additional package `javax.realtime`, which defines various classes and interfaces that provide extra functionality for the Java programmer and that require modification to the semantics of the Java virtual machine. The chapter presents an overview of these classes and interfaces in each of the main enhanced areas.

Team LiB ◂ PREVIOUS NEXT ▸

Team LiB ◂ PREVIOUS NEXT ▸

7.1 Background and NIST Requirements

Java's success has led to several attempts to extend the language so that it is more appropriate for a wide range of real-time systems. Much of the early work in this area was fragmented and lacked clear direction. In the late 1990s, under the auspices of the US National Institute of Standards and Technology (NIST), approximately 50 companies and organizations pooled their resources and generated several guiding principles and a set of requirements for real-time extensions to the Java platform (Carnahan and Ruark, 1999). Among the guiding principles was that Real-Time Java (RTJ) should take into account current real-time practices and facilitate advances in the state of the art of real-time systems implementation technology. The following facilities were deemed necessary to support the current state of real-time practice (Carnahan and Ruark, 1999).

- Fixed priority and round-robin scheduling.
- Mutual exclusion locking (avoiding priority inversion).
- Inter-thread communication (e.g. semaphores).
- User-defined interrupt handlers and device drivers – including the ability to manage interrupts (e.g. enabling and disabling).
- Timeouts and aborts on running threads.

These facilities will be described in detail in the remaining chapters of this book.

The NIST group recognized that profiles (subsets) of RTJ were necessary in order to cope with the wide variety of possible applications. These included safety critical, no dynamic loading and distributed real-time profiles.

There was also an agreement that any implementation of RTJ should provide the following.

- ¿ A framework for finding available profiles.
- ¿ Bounded preemption latency on any garbage collection.
- ¿ A well-defined model for real-time Java threads.
- ¿ Communication and synchronization between real-time and non-real-time threads.
- ¿ Mechanisms for handling internal and external asynchronous events.
- ¿ Asynchronous thread termination.
- ¿ Mutual exclusion without blocking.
- ¿ The ability to determine whether the running thread is real-time or non-real-time.
- ¿ A well-defined relationship between real-time and non-real-time threads.

Following on from the NIST requirements, there were two main efforts to define a "real-time" Java (J-Consortium, 2000; Bollella *et al.*, 2000). Perhaps the most high-profile attempt is the one backed by Sun and produced by The Real-Time for Java Expert Group (Bollella *et al.*, 2000). It is this that is the focus of the remaining chapters of this book.

The RTSJ in its defining document makes only a passing reference to the NIST requirements and instead defines its own "guiding principles". These include requirements to

- ¿ be backward compatible with non-real-time Java programs,
- ¿ support the principle of "Write Once, Run Anywhere" but not at the expense of predictability,
- ¿ address current real-time system practices and allow future implementations to include advanced features.
- ¿ give priority to predictable execution in all design trade-offs,
- ¿ require *no* syntactic extensions to the Java language,
- ¿ allow implementers flexibility.

Warning The requirement for no syntactic enhancements to Java has had a strong impact on the manner in which the real-time facilities can be provided. In particular, all facilities have to be provided by a library of classes and interfaces. In places, this has had a marked effect on the readability of real-time applications.

Team LiB ◄ PREVIOUS NEXT ►

7.2 Overview of Enhancements

The RTSJ enhances Java in the following areas:

¿ memory management

¿ time values and clocks

¿ schedulable objects and scheduling

¿ real-time threads

¿ asynchronous event handling and timers

¿ asynchronous transfer of control

¿ synchronization and resource sharing

¿ physical and raw memory access.

Important note It should be stressed that the RTSJ is only intended to address the execution of real-time Java programs on single-processor systems. It attempts not to preclude execution on shared-memory multiprocessor systems, but it has no facilities directly to control, for example, allocation of threads to processors.

The remainder of this chapter will present an overview of the RTSJ and introduce the various classes and interfaces. Later chapters will consider each of the above enhancements in detail.

7.3 Memory Management

Many real-time systems (particularly those that form part of an embedded system) have only a limited amount of memory available; this is either because of the cost or because of other constraints associated with the surrounding system (for example, size, power or weight constraints). It may, therefore, be necessary to control how this memory is allocated so that it can be used effectively. Furthermore, where there is more than one type of memory (with different access characteristics) within the system, it may be necessary to instruct the compiler to place certain data types at certain locations. By doing this, the program is able to increase performance and predictability as well as interact more effectively with the outside world.

Heap memory

The run-time implementations of most programming languages provide a large amount of memory (called the *heap*) so that the programmer can make dynamic requests for chunks to be allocated (for example, to contain an array whose bounds are not known at compile time). An allocator (the *new* operator in Java) is used for this purpose. It returns a reference to memory within the heap of adequate size for the program's data structures (classes). The run-time system (JVM) is responsible for managing the heap. Key problems are deciding how much space is required and deciding when allocated space can be released and reused. The first of these problems requires application knowledge (the `SizeEstimator` class in RTSJ helps here). The second can be handled in several ways, requiring

¿ the programmer to return the memory explicitly – this is error prone but is easy to implement;

¿ the run-time support system (JVM) to monitor the memory and determine when it can logically no longer be accessed – the scope rules of a language allows its implementation to adopt this approach; when a reference type goes out of scope, all the memory associated with that reference type can be freed;

¿ the run-time support system (JVM) to monitor the memory and release chunks that are no longer being used (*garbage collection*) – this is, perhaps, the most general approach as it allows memory to be freed even though its associated reference type is still in scope.

From a real-time perspective, the above approaches have an increasing impact on the ability to analyze the timing properties of the program. In particular, garbage collection may be performed either when the heap is full (there is no free space left) or incrementally (either by an asynchronous activity or on each allocation request). In either case, running the garbage collector may have a significant impact on the response time of a time-critical thread.

All objects in Java are allocated on the heap, and the language requires garbage collection for an effective implementation. The garbage collector runs as part of the JVM. Although there has been much work on real-time garbage collection and progress continues to be made, there is still a reluctance to rely on these techniques in time-critical systems.

Memory areas

The RTSJ recognizes that it is necessary to allow memory management, which is not affected by the vagaries of garbage collection. To this end, it introduces the notion of *memory areas*, some of which exist outside the traditional Java heap and never suffer garbage collection. It also requires that the garbage collector can be preempted by real-time threads and that the time between a real-time thread wishing to preempt and the time it is allowed to preempt is bounded (there should be a bounded latency for preemption to take place).

The `MemoryArea` class is an abstract class from which all RTSJ memory areas are derived. When a particular memory area is entered, all object allocation is performed within that area. Using this abstract class, the RTSJ defines various kinds of memory including the following:

¿ `HeapMemory` – Heap memory allows objects to be allocated in the standard Java heap.

¿ `ImmortalMemory` – Immortal memory is shared among all threads in an application. Objects created in immortal memory are never subject to garbage collection delays and behave as if they are freed by the system only when the program terminates.

¿ `ScopedMemory` – Scoped memory is a memory area where objects with a well-defined lifetime can be allocated. A scoped memory may be entered explicitly or implicitly by attaching it to a real-time entity (a real-time thread or an asynchronous event handler) at its creation time. Associated with each scoped memory is a reference count. This keeps track of how many real-time entities are currently using the area. When the reference count reaches 0, all objects resident in the scoped memory have their finalization method executed, and the memory is available for reuse. The `ScopedMemory` class is an abstract class that has several subclasses.

¿ `VTMemory` – A subclass of `ScopedMemory` where allocations may take variable amounts of time.

- `LTMemory` – A subclass of `ScopedMemory` where allocations occur in linear time (that is, the time taken to allocate the object is directly proportional to the size of the object).

Memory parameters can be given when real-time threads and asynchronous event handlers are created. They can also be set or changed while the real-time threads and asynchronous event handlers are active. Memory parameters specify

- the maximum amount of memory a thread/handler can consume in its default memory area,
- the maximum amount of memory that can be consumed in immortal memory,
- a limit on the rate of allocation from the heap (in bytes per second),

and can be used by the scheduler as part of an admission control policy and/or for the purpose of ensuring adequate garbage collection.

Figure 7.1 illustrates the various RTSJ classes that support memory management.

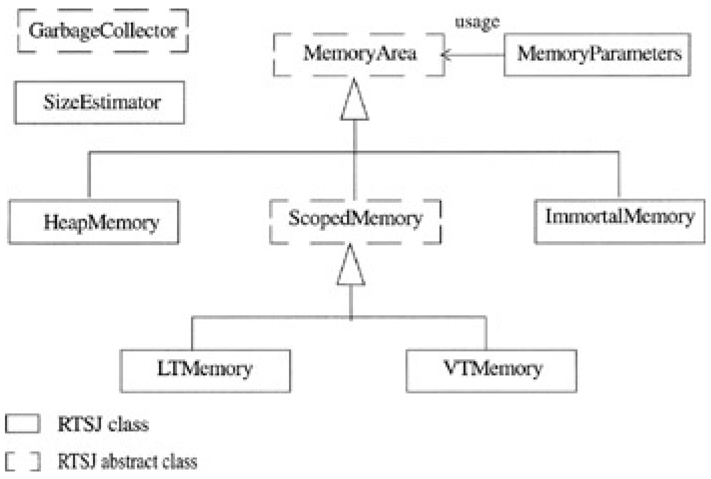

Figure 7.1: Classes Supporting Memory Management

Further control over the allocation of memory, in particular, specifying memory of a particular type (for example, flash memory) or specifying the actual machine address where memory is to be allocated, is given by the physical and raw memory access facility, which is considered in Section 7.10.

Team LiB
Team LiB
◄ PREVIOUS NEXT ►
◄ PREVIOUS NEXT ►

7.4 Time Values and Clocks

As mentioned in Section 4.2, Java supports the notion of a wall clock (calendar time). The `Date` class is intended to reflect UTC (Coordinated Universal Time), however, accuracy depends on the host system [Gosling, Joy and Steele, 1996].

The RTSJ introduces clocks with high-resolution time types. The associated classes are illustrated in Figure 7.2.

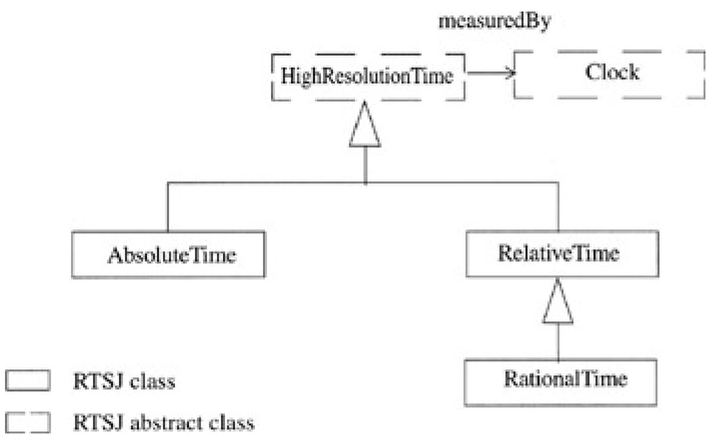

Figure 7.2: Time Classes

`HighResolutionTime` encapsulates time values with nanosecond granularity and an associated clock. A value is represented by a 64-bits millisecond and a 32-bits nanosecond component. There are methods to read, write and compare time values, as well as methods to get and set the clock. The class is an abstract class that has three subclasses: one that represents relative time, one that represents absolute time, and one that represents "rational" time. Relative time is a duration measured by a particular clock. Absolute time is actually expressed as a time relative to some epoch. This epoch depends on the associated clock. It might be January 1, 1970, GMT for the wall clock or system start-up time for a monotonic (nondecreasing and no added leap seconds) clock. Rational time is a relative-time type, which has an associated frequency. It is used to represent the rate at which certain events occur (for example, periodic thread execution).

The RTSJ `Clock` class defines the abstract class from which all clocks are derived. The specification allows many different types of clocks; for example, there could be a CPU execution-time clock (although this is not required by the RTSJ). There is always one real-time clock that advances monotonically. A static method `getRealtimeClock` allows this clock to be obtained.

Countdown clocks are called *timers* by the RTSJ. They will be considered in Section 7.7.

Team LiB ◀ PREVIOUS NEXT ▶
Team LiB ◀ PREVIOUS NEXT ▶

7.5 Schedulable Objects and Scheduling

Scheduling of threads is a key aspect for all real-time systems. Java allows each thread to have a priority that can be used by the JVM when allocating processing resources. However, as discussed in Section 4.1, Java offers no guarantees that the highest priority runnable thread will be the one executing at any point in time. This is because a JVM may be relying on a host operating system to support its threads. Some of these systems may not support preemptive priority-based scheduling. Furthermore, Java only defines 10 priority levels, and an implementation is free to map these priorities onto a more restricted host operating system's priority range if necessary.

The weak definition of scheduling and the restricted range of priorities means that Java programs lack predictability and, hence, Java's use for real-time systems' implementation is severely limited. Consequently, this is a major area that needs to be addressed. The RTSJ attacks these problems on several fronts. Firstly, it generalizes the entities that can be scheduled away from threads toward the notion of ***schedulable objects***. A schedulable object is one that implements the `Schedulable` interface. Each schedulable object must also indicate its specific

¿ release requirement (that is, when it should become runnable),

- ¿ memory requirements (for example, the rate at which it will allocate memory on the heap),
- ¿ scheduling requirements (for example, the priority at which it should be scheduled).

Parameters affecting scheduling

Release requirements are specified via the `ReleaseParameters` class hierarchy. Scheduling theories often identify three types of releases: periodic (released on a regular basis), aperiodic (released at random) and sporadic (released irregularly, but with a minimum time between each release). These are represented by the `PeriodicParameters`, `AperiodicParameters` and `SporadicParameters` classes respectively. All release parameter classes encapsulate a `cost` and a `deadline` (relative time) value. The `cost` is the maximum amount of CPU time (execution time) needed to execute the associated schedulable object every time it is released. The `deadline` is the time at which the object must have finished executing the current release; it is specified relative to the time the object was released. `PeriodicParameters` also include the `start` time for the first release and the time interval (`period`) between releases. `SporadicParameters` include the minimum inter-arrival time between releases.

For aperiodic schedulable objects, it is possible to limit the amount of time the scheduler gives them in a particular period. This is achieved by `ProcessingGroupParameters` that have associated start, period and cost times. Where the same `ProcessingGroupParameters` class is associated with more than one schedulable object, the limitations apply to the group as a whole.

Scheduling parameters are used by a scheduler to determine which object is currently the most eligible for execution. The abstract class `SchedulingParameters` provides the root class from which a range of possible scheduling criteria can be expressed. The RTSJ defines only one criterion that is based on priority (as would be expected, it is silent on how those priorities should be assigned). In common with most other languages and operating systems, high numerical values for priority represent high execution eligibilities. `ImportanceParameters` allow an additional numerical scheduling metric to be assigned; it is a subclass of the `PriorityParameters` class.

Schedulers

The scheduler is responsible for scheduling its associated schedulable objects. The RTSJ is silent on how many schedulers might exist within a single real-time JVM but typically only a single scheduler will be present. Although RTSJ explicitly supports priority-based scheduling via the `PriorityScheduler` (a fixed preemptive priority-based scheduler with 28 unique priority levels), it acknowledges that an implementation might provide other schedulers. Consequently, `Scheduler` is an abstract class with `PriorityScheduler` a defined subclass. This allows an implementation to provide, say, an Earliest-Deadline-First scheduler. Any attempt by the application to set the scheduler for its threads has to be checked to ensure that it has the appropriate security permissions.

The relationship between the above classes is illustrated in Figure 7.3.

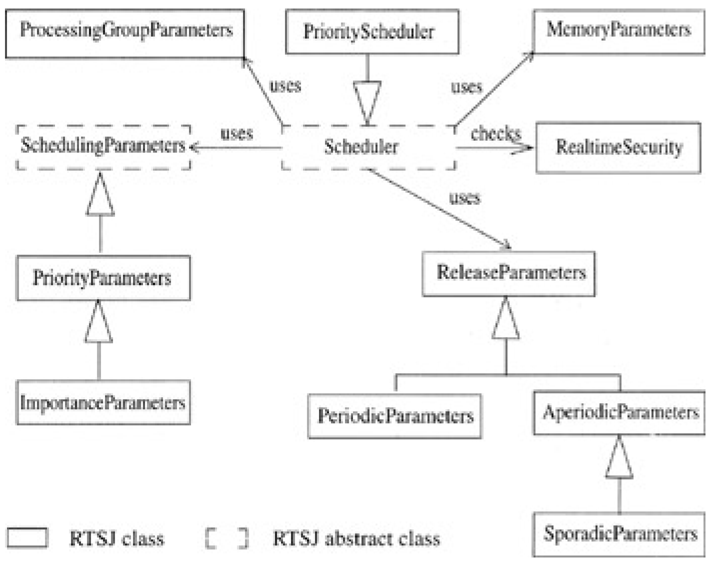

Figure 7.3: Scheduling and its Parameter Classes

Meeting deadlines

Once a system accepts that schedulable objects have deadlines and a cost associated with their execution, there is an obligation on the system to undertake the following activities (Burns and Wellings, 2001):

- provide a means by which it is possible to predict whether a set of application objects will meet their deadlines, and
- provide mechanisms whereby the system can report that an application object has missed its deadline, consumed more resources than indicated by the cost value or has been released more often than indicated by its minimum inter-arrival time.

For some systems it is possible to predict offline whether the application will meet its deadline – for example, via a form of schedulability analysis (see Burns and Wellings, 2001). For other systems, some form of on-line analysis is required. The RTSJ does not require that an implementation support on-line analysis, but it does provide the hooks that can be used if needed.

Irrespective of whether or how prediction has been performed, it is necessary to report overruns, etc. The RTSJ provides an asynchronous event-handling mechanism for this purpose. All release parameters also specify which event handlers should be released as a consequence of a deadline miss or a cost overrun[1]. Of course, a program can indicate that it is not concerned with a missed deadline, etc., by setting a null handler.

[1]However, the RTSJ does not require an implementation to support execution-time monitoring.

PREVIOUS NEXT
PREVIOUS NEXT

7.6 Real-time Threads

One type of schedulable object is a real-time thread represented by the `RealtimeThread` class. This is an extension of the Java `Thread` class, which also implements the `Schedulable` interface. A real-time thread's parameters are illustrated in Figure 7.4.

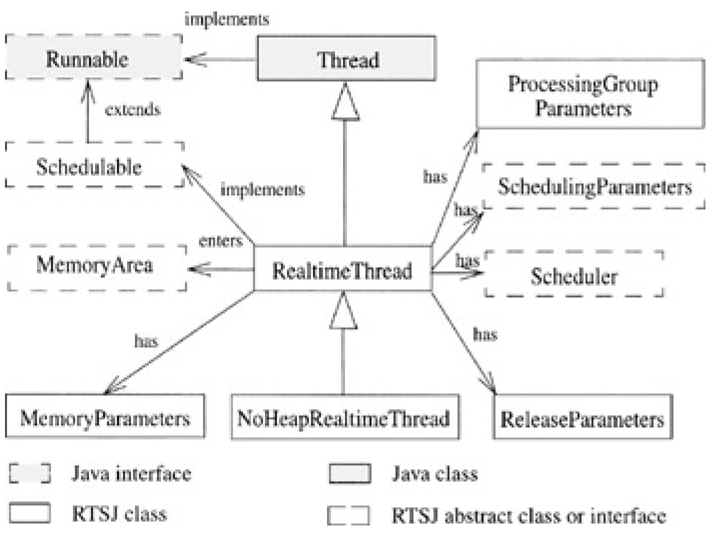

Figure 7.4: Real-time Thread and its Parameters

A periodic real-time thread is a real-time thread that has periodic release parameters. Similarly, an aperiodic (or sporadic) real-time thread is one that has aperiodic (or sporadic) release parameters. A `NoHeapRealtimeThread` is one that guarantees not to create or reference any objects on the heap. Hence, its execution is totally independent of the garbage collector.

Team LiB PREVIOUS NEXT
Team LiB PREVIOUS NEXT

7.7 Asynchronous Event Handling and Timers

Threads and real-time threads are the appropriate abstractions to use when representing concurrent activities that have a significant life history. However, it is also often necessary to respond to events that happen asynchronously to a thread's activity. These events may be happenings in the environment of an embedded system or notifications received from internal activities within the program. It is always possible to have extra threads that wait for these events, but this is inefficient and may result in an explosion of the number of threads in a program.

In Java, standard classes can be programmed that multiplex events onto a single thread that handles them in a particular order. For example, the Abstract Windows Toolkit has an event-handling thread for interacting with user interface windowing events. The Java `Timer` and `TimerTask` (see Section 4.4) classes provide similar functionality for time-triggered events.

From a real-time perspective, events may require their handlers to respond within deadlines. Hence, more control is needed over the order in which events are handled. The RTSJ, therefore, generalizes Java event handlers to be schedulable entities. Like real-time threads, they have a variety of parameters (as illustrated in Figure 7.5).

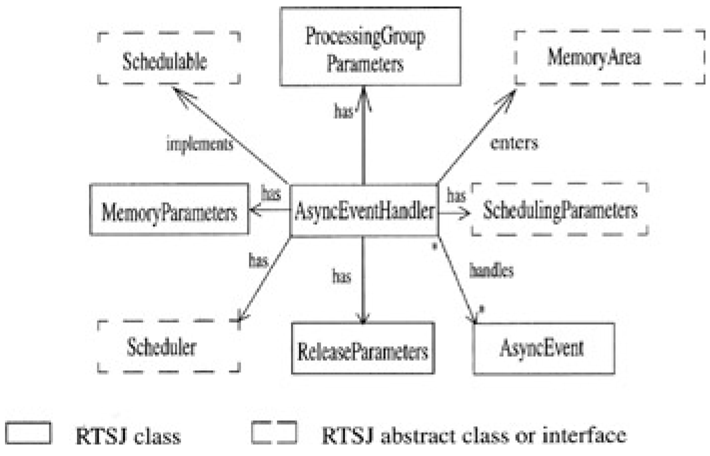

Figure 7.5: Asynchronous Event Handler Parameters

In practice, the real-time JVM will (usually) dynamically associate an event handler with a real-time thread when the handler is released for execution (some may even delay the association until the handler has the highest execution eligibility). To avoid this overhead, it is possible to specify that the handler must be permanently bound to a real-time thread. Each `AsyncEvent` can have one or more handlers, and the same handler can be associated with more than one event. When the event occurs, all the handlers associated with the event are released for execution according to their `Scheduling-Parameters`.

Asynchronous events can be associated with interrupts or POSIX signals (if supported by the underlying operating system) or they can be linked to a timer. The timer will cause the event to fire when a specified time (relative to a particular clock) expires. This can be a one-shot firing or a periodic firing. Figure 7.6 illustrates the classes associated with events and their handlers.

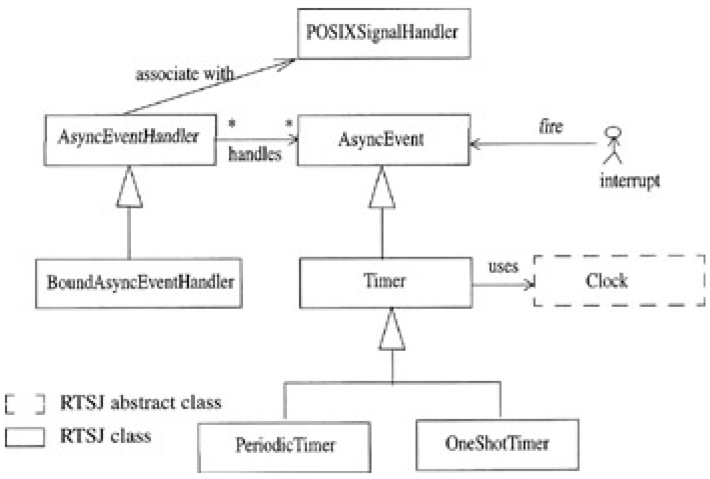

Figure 7.6: Asynchronous Events and Timer Classes

Team LiB ◀ PREVIOUS NEXT ▶

7.8 Asynchronous Transfer of Control

Asynchronous events allow the program to respond in a timely fashion to a condition that has been detected by the program or the environment. However, they do not allow a particular schedulable object to be directly informed. In many applications, the only form of asynchronous notification that a real-time thread needs is a request for it to terminate itself. Consequently, languages and operating systems typically provide a kill or abort facility. Unfortunately, for real-time systems this approach is too heavy-handed; instead what is required is for the schedulable object to stop what it is currently doing and begin executing an alternative algorithm.

In Java, it is the interrupt mechanism (see Section 3.5) that attempts to provide a limited form of asynchronous notification. Lamentably, the mechanism is synchronous and does not support timely response to the "interrupt". Instead, a running thread has to poll for notification. This delay is deemed unacceptable for real-time systems. For these reasons, the RTSJ provides an alternative approach for interrupting a schedulable object, using asynchronous transfer of control (ATC). The ATC model is based on the following principles (Bollella *et al.*, 2000).

- A schedulable object (real-time thread or event handler) must explicitly indicate that it is prepared to allow an ATC to be delivered. By default, a sehedulable object will have ATCs deferred. The rationale for this approach is that the schedulable object may be executing a method inside an object that is unaware that an ATC may be delivered. If the schedulable object is forced to abandon execution of the method, it might compromise the integrity of the object's state.
- The execution of synchronized methods and statements always defers the delivery of an ATC.
- ATCs have termination semantics; this means that the real-time thread (or event handler) does not resume execution at the point in its code where the ATC was delivered. An ATC is a nonreturnable transfer of control (it is, therefore, more analogous to a goto statement than a procedure call).

The RTSJ ATC model is integrated with the Java exception handling facility. An `AsynchronouslyInterruptedException` (AIE) class defines the ATC event. Every method that is prepared to allow the delivery of an AIE must indicate so via a `throws AsynchronouslyInterruptedException` in its declaration. ATCs are deferred until the thread is executing within such a method. The `Interruptible` interface provides the link between the AIE class and the object executing an interruptible method. A subclass of AIE, called `Timed`, allows an ATC to be generated at a point in time (absolute or relative). The relationship between the classes is illustrated in Figure 7.7.

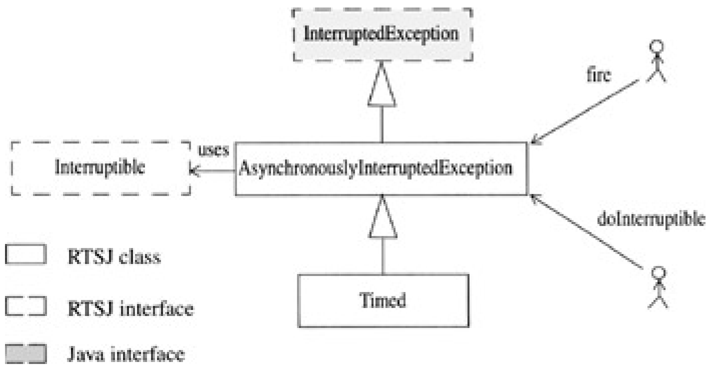

Figure 7.7: Asynchronous Transfer of Control

The ATC facilities (possibly used in conjunction with the asynchronous event-handling mechanisms) also allow real-time threads to be terminated in a controlled and safe manner.

Team LiB ◄ PREVIOUS NEXT ►
Team LiB ◄ PREVIOUS NEXT ►

7.9 Synchronization and Resource Sharing

The key to predicting the behavior of multithreaded real-time programs is understanding how threads (and other schedulable objects) communicate and synchronize with each other. Java provides a mechanism that is based on mutually exclusive access to shared data via a monitor-like construct (see Chapter 3). Unfortunately, all synchronization mechanisms that are based on mutual exclusion suffer from *priority inversion*.

The problem of priority inversion and its solution *priority inheritance* is now a well-researched area of real-time systems (see (Burns and Wellings, 2001)). There are a variety of priority inheritance algorithms; the RTSJ explicitly supports two: *simple priority inheritance* and *priority ceiling emulation inheritance* (sometimes called *immediate ceiling priority inheritance, priority protect inheritance protocol* or the *highest locker protocol*).

The RTSJ and priority inheritance

The way in which the RTSJ supports priority inheritance algorithms is through the `MonitorControl` class hierarchy. The abstract root class has several static methods that allow

¿ the default inheritance property for all monitor locks to be set and queried

¿ a specific object's inheritance property to be set.

The actual policies are defined by subclasses as illustrated in Figure 7.8.

The actual policies are defined by subclasses as illustrated in Figure 7.8.

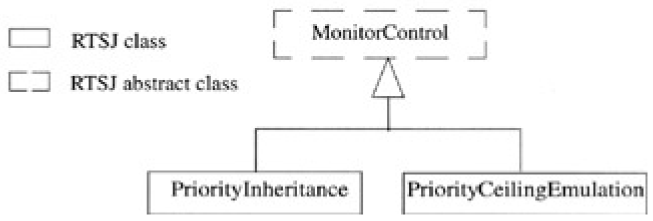

Figure 7.8: Priority Inheritance Classes

Priority inheritance and garbage collection

Priority inheritance algorithms allow the blocking suffered by schedulable objects to be bounded. However, if schedulable objects want to communicate with non-real-time threads, then interaction with garbage collection must be considered. It is necessary to try to avoid the situation where a non-real-time thread has entered into a mutual exclusion zone shared with a schedulable object. The actions of the non-real-time thread results in garbage collection being performed. The schedulable object then preempts the garbage collector, but is unable to enter the mutual exclusion zone. It must now wait for the garbage collection to finish and the non-real-time thread to leave the zone.

Wait-free communication

One way of avoiding unpredictable interactions with the garbage collector is to provide a nonblocking communication mechanism for use between non-real-time threads and schedulable objects. The RTSJ provides three wait-free nonblocking classes to help facilitate this communication:

¿ `WaitFreeWriteQueue` – This is a bounded buffer intended for the case where the schedulable object wishes to send an object to the non-real-time thread. The read operation on the buffer is synchronized and blocks if the buffer is empty. The write operation is not synchronized and indicates whether it has succeeded in writing to the buffer.

- `WaitFreeReadQueue` – This is a bounded buffer intended for the case where the non-real-time thread wishes to send an object to the schedulable object. The write operation on the buffer is synchronized. The read operation is not, and returns either an object or null if the buffer is empty. The reader can request to be notified when data arrives.

- `WaitFreeDequeue` – This is a bounded buffer, which allows both blocking and nonblocking read and write operations.

The RTSJ classes are illustrated in Figure 7.9.

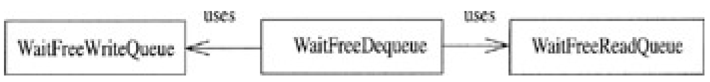

Figure 7.9: Wait-free Communication Classes

Team LiB
Team LiB
PREVIOUS NEXT
PREVIOUS NEXT

7.10 Physical and Raw Memory Access

As mentioned in Section 7.3, embedded real-time systems may support more than one type of memory (with different access characteristics); furthermore, devices may have their interface registers mapped into the virtual memory address space. There are, therefore, two main mechanisms that must be provided by the RTSJ.

1. Mechanisms that allow objects to be placed into areas of memory that have particular properties or access requirements; for example Direct Memory Access (DMA) memory, or shared memory.

2. Mechanisms that allow the programmer to access raw memory locations that are being used to interface to the outside world; for example memory-mapped input and output device registers.

Figure 7.10 summarizes the various classes.

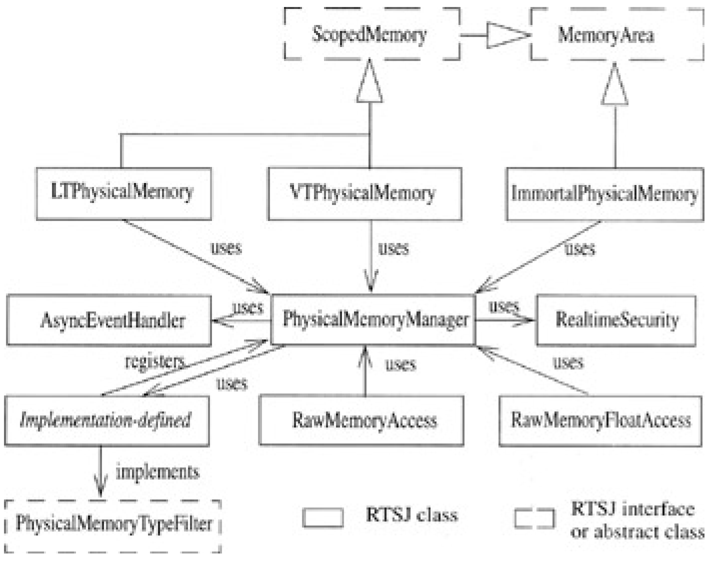

Figure 7.10: Physical and Raw Memory Access

To support the first requirement, the RTSJ provides extensions of the `ScopedMemory` areas discussed in Section 7.3. These extensions provide the physical memory counterparts to the linear-time and variable-time scoped memory classes. They enable the programmer to specify that objects should be created in memory with a particular characteristic (for example, shared memory) as well as the usual requirements for linear time allocation etc. Immortal physical memory is also provided.

To support the second requirement, the RTSJ provides classes which can access raw memory in terms of reading and writing Java variables or arrays of primitive data types (int, long, float etc.). The implementation of both these physical and raw memory classes can assume the existence of a `PhysicalMemoryManager` class, which must be provided by the real-time Java virtual machine. This class can also make use of implementation-dependent classes, called *filters*, which help support and police the various memory categories. All these classes must support the `PhysicalMemoryType-Filter` interface. The memory manager can also check that the program has the necessary security permissions before allowing access to physical and raw memory.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

7.11 System-wide Properties

The overall properties of an RTSJ virtual machine can be acquired via the `Realtime-System` class. Operations on this class also allow the system's performance to be tuned. For example, the total number of concurrent object locks can be specified. The byte ordering of the machine can also be determined.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

7.12 Synchronization and the RTSJ

Most of the classes provided by the RTSJ will be used in a concurrent environment. However, in some cases there will be no concurrent use of a particular object. For example, consider the `AsyncEvent` class. In general, an asynchronous event may be fired by more than one real-time thread. In practice, only one firer will exist for many events. There is, therefore, a dilemma for the RTSJ designers of whether to make the `fire` method synchronized or not. Indeed, the initial version of the RTSJ has `fire` as synchronized (Bollella *et al.*, 2000). Version 1.0 has it nonsynchronized!

The advantage of making a method synchronized is that the class designer is forced to consider the use of the class in a concurrent environment (making it "thread safe" or "thread conditionally safe", using Bloch's terminology – see Section 4.8). The disadvantage is that the program incurs a run-time overhead in calling synchronized methods irrespective of whether the actual object is subject to concurrent calls.

The alternative approach is to declare the class as being "thread compatible". This means that the class has paid no considerations to concurrent calls of its methods but does guarantee that it is not completely "thread hostile". It is, therefore, up to the programmer to provide the required synchronization if an object is called concurrently. This is a burden to the programmer, but the run-time overhead is only incurred when it is really needed. It is this latter approach that is generally (but not universally) taken by the RTSJ. Methods are only made synchronized when absolutely necessary; for instance, because there is some condition synchronization required. On occasions, the synchronization is hidden in the class.

There are many ways to make a "thread compatible" class "thread safe". The simplest is to create a subclass and override the methods that can be called concurrently (directly or indirectly). The overridden methods can then be made synchronized. The subclass can be used in all places where the superclass can be used, and run-time dispatching will ensure the correct method is called.

If this approach is not appropriate (perhaps because the methods are `final` methods), an alternative is to lock the object when an unsynchronized method is called. However, this is more error prone, as it requires all users of the object to first obtain the lock.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

7.13 Summary

This chapter has introduced the majority of classes and interfaces that can be found in the Real-Time Specification for Java and has attempted to illustrate the overall motivation for their inclusion. The details of the semantic model behind these classes and full descriptions of their methods (and how they can be used) are given in the following chapters.

At the beginning of this chapter, the NIST core requirements for real-time Java extensions were identified. It is now possible to review these requirements and see how closely the RTSJ has met them. Firstly, the facilities needed to support the current state of real-time practice:

¿ Fixed priority and round-robin scheduling – the RTSJ supports a fixed priority scheduler and allows implementations to provide other schedulers.

¿ Mutual exclusion locking (avoiding priority inversion) – the RTSJ supports priority inheritance algorithms of synchronized objects and requires that all RTSJ implementations avoid unbounded priority inversion.

¿ Inter-thread communication (e.g. semaphores) – schedulable objects can communicate using the conventional Java mechanisms augmented with priority queues and priority inversion avoidance algorithms.

¿ User-defined interrupt handlers and device drivers (including the ability to manage interrupts; e.g. enabling and disabling) – the RTSJ allows interrupts to be associated with asynchronous events. Memory-mapped device registers can be accessed by the raw memory facilities.

¿ Timeouts and aborts on running threads – the RTSJ allows asynchronous transfers of control via asynchronous exceptions; they can be event-triggered or time-triggered.

In terms of implementation requirements:

¿ A framework for finding available profiles – the RTSJ does not explicitly address the issues of profiles other than by allowing an implementation to provide alternative scheduling algorithms (e.g. EDF) and allowing the application to locate the scheduling algorithms. There is no identification of, say, a safety critical systems profile or a profile that prohibits dynamic loading of classes. Distributed real-time systems are not addressed, but there is another Java Expert Group that is considering this issue (Wellings, Clark, Jenson, and Wells, 2002).

¿ Bounded preemption latency on any garbage collection – supported by the `GarbageCollector` class.

¿ A well-defined model for real-time Java threads – supported by the `RealtimeThread` and

`NoHeapRealtimeThread` classes.

¿ Communication and synchronization between real-time and non-real-time threads – supported by the wait-free communication classes.

¿ Mechanisms for handling internal and external asynchronous events – supported by the `AsyncEvent`, `AsyncEventHandler` and `POSIXSignalHandler` classes.

¿ Asynchronous thread termination – supported by the `AsynchronouslyInterruptedException` class and the `Interruptible` interface.

¿ Mutual exclusion without blocking – supported by the wait-free communication classes.

¿ The ability to determine whether the running thread is real-time or non-real-time – supported by the `RealtimeThread` class.

¿ A well-defined relationship between real-time and non-real-time threads — supported by the real-time thread, the scheduling and memory management models.

Overall, it can be seen that the RTSJ addresses all the NIST top level requirements in some form or other. It is, however, a little weak in its support for profiles.

Team LiB PREVIOUS NEXT