A Comparison of the Mutual Exclusion Features in Ada and the Real-Time Specification for Java™

Benjamin M. Brosgol

AdaCore
Belmont, Massachusetts USA
brosgol@adacore.com

Abstract. A concurrent program generally comprises a collection of threads[1] that interact cooperatively, either directly or through shared data objects. In the latter case the sharing needs to be implemented by some mechanism that ensures mutually exclusive access, or possibly "concurrent read / exclusive write". Ada and the Real-Time Specification for Java have taken different approaches to mutual exclusion. This paper analyzes and compares them with respect to programming style (clarity, encapsulation, avoidance of errors such as deadlock), priority inversion management, expressibility/generality, and efficiency. It also looks at interactions with exceptions and asynchronous transfer of control.

1 Introduction

Mutual exclusion is a fundamental requirement in a concurrent program, and over the years a number of different approaches have been proposed, studied, and implemented, ranging from low-level primitives to higher-level features. This paper shows how Ada [1] and the Real-Time Specification for Java ("RTSJ")[2] [2] [3] approach this issue. The basic problem that each must address, especially with respect to meeting real-time requirements, is how to provide mutual exclusion in a way that offers sufficient generality, supports sound software engineering practice, prevents unbounded priority inversion, and allows efficient implementations.

In brief, mutual exclusion in Ada is obtainable via three mechanisms, listed in order of increasing generality:

- Atomically accessible data objects, so designated via a pragma
- Protected objects, with an associated locking policy
- "Passive tasks" sequentializing accesses via rendezvous

The RTSJ captures mutual exclusion in two ways:

- Atomically accessible variables so marked via the `volatile` modifier

[1] We use the term "thread" generically to refer to a concurrent activity. When discussing a particular language's mechanism we use that language's terminology (e.g., "task" in Ada).

[2] The RTSJ is the product of the Java Community Process's Java Specification Request 00001. It adds real-time functionality to the Java platform through the `javax.realtime` class library together with constraints on certain behaviors that are implementation dependent in Java.

– Methods or blocks that are "synchronized" on an object, with priority inversion management based on the object's "monitor control policy"

The second mechanism basically extends and makes more deterministic the semantics of Java's "synchronized" facility.

It is also possible to simulate Ada's passive task style in Java (and thus in the RTSJ), but this is not a typical idiom.

The following sections discuss and contrast the Ada and RTSJ approaches, with a particular focus on priority inversion management. Given the context for this paper, it is assumed that the reader is more familiar with Ada than with Java or the RTSJ. Thus more background information and technical detail are provided for Java and especially the RTSJ than for Ada.

2 Mutual Exclusion in Java

Since the RTSJ builds directly on the Java [4] mechanisms, this section summarizes and evaluates Java's facilities for mutual exclusion. A more comprehensive analysis of Java's concurrency model may be found in [5], [6], and [7].

2.1 `volatile` variables

In simple cases two threads may need to communicate via a shared variable, but Java's memory model [8] allows the compiler to cache the value of the variable in each thread's local memory. To prevent this unwanted optimization, the programmer can declare the variable as `volatile`. Because of Java's pointer-based semantics, `volatile` never applies to an entire object, but only to either a scalar variable or a reference to an object. The intent is that any access to the variable is atomic with respect to thread context switches. Indeed, when `volatile` is specified then atomicity is required, but with two exceptions: variables of type `long` or `double`, which take 64 bits. In these cases the language rules encourage but do not require atomic accesses. Further, `volatile` can not be specified for array components. It can be specified for an array variable, but then it applies only to the array reference and not to the components.

2.2 `synchronized` code

Java's mutual exclusion facility is based on the concept of "object locks". Conceptually, each object (including class objects[3]) has an associated lock. The program can attempt to acquire a lock on an object referenced by a variable `ref` either by executing a statement `synchronized(ref){...}` or by invoking an instance method `ref.func(...)` where `func` is declared as `synchronized`. (A static method may also be declared as `synchronized`, in which case the lock in question applies to the corresponding class object.)

[3] A *class object* is an instance of class `Class` and serves as a run-time manifestation of a Java class.

When a thread holds a lock on an object, any other thread attempting to synchronize on that object will be prevented from running [4] until the lock has been released, at which point it becomes ready and competes with other threads that were similarly stalled. Java does not dictate how the stalling is to be implemented.

An object lock is not simply a boolean, it needs to have a non-negative count associated with it to handle situations when a thread acquires a lock on an object for which it already holds the lock. When a thread exits from synchronized code, either normally or through exception propagation, the count is decreased. If/when it becomes zero, the lock is released.

The `synchronized` mechanism is rather general; there are no restrictions on the statements that can be executed inside synchronized code. In particular, a thread can block (e.g., via the `sleep()` method) while holding the lock. Some blocking methods (specifically `wait`) release the lock; others do not.

Java's `synchronized` mechanism has several problems:

- It is susceptible to the "nested monitors" problem: if two threads attempt to synchronize on different objects in different orders, they may deadlock.
- The language semantics do not specify how priorities affect lock acquisition. If low- and high-priority threads are competing for the same object lock, there is no guarantee that preference is given to the high-priority thread. If a low-priority thread owns a lock that is needed by a high-priority thread, there is no requirement for priority inheritance, and thus unbounded priority inversion may result.
- Java's `synchronized` construct is not fully integrated with state-based mutual exclusion (i.e., when a thread not only needs mutually exclusive access to an object but also needs to wait until the object is in a particular state). The `wait` / `notify` / `notifyAll` mechanism basically supports just one condition variable per object, which makes it somewhat error-prone to express classical idioms such as the bounded buffer. Further, there is a run-time check to ensure that when one of these methods is invoked, the calling thread holds the lock on the object.
- Methodologically, the presence of synchronized methods in a class does not mean that uses of objects in that class are "thread safe". For example, even though one thread has "locked" an object through a synchronized method, another thread can corrupt the object by invoking a non-synchronized method or by directly accessing the object's fields.

These last two issues imply that Java's use of the term "monitor" to denote the object lock mechanism is somewhat of a misnomer, since the classical monitor construct includes condition variables and data encapsulation. Indeed, Brinch Hansen's rather scathing critique of Java's thread model [9] was due to a large extent to Java's failure to provide a safe mutual exclusion mechanism as the basis for its concurrency model.

Java has no notion of synchronized code being immune to asynchronous interruption. This caused some anomalous interactions with several methods in the `Thread` class:

[4] We use the term *stalled* to denote the state of a thread that has attempted to synchronize on an object that has been locked by some other thread. It is useful to differentiate the stalled state from the *blocked* state that results from invocation of methods such as `sleep` and `wait`.

– If `t.stop()` is invoked while `t` is inside code that is synchronized on some object `obj`, then `t` will suffer the throwing of a `ThreadDeath` exception. If not handled inside the synchronized code, the exception will be propagated and the lock on `obj` will be released, possibly leaving `obj` in an inconsistent state.
– If `t.destroy()` is invoked while `t` is inside code that is synchronized on some object `obj`, then `t` is terminated "immediately" without propagating any exceptions or releasing the lock on `obj`. Thus threads that later attempt to synchronize on `obj` will be deadlocked.

Because of such anomalies, the `Thread.stop` and `Thread.destroy` methods have been deprecated.

In summary, Java's `synchronized` mechanism can best be regarded as a low-level building block. It is a flexible construct that offers quite a bit of generality, but needs to be used with care. The Java designers made no pretext of attempting to support real-time requirements, and indeed the semantics are too loose to be depended upon.

More recently, an attempt has been made to enhance the basic facilities with higher-level constructs: the concurrency utilities from JSR-166 [10]. Their development was in parallel with the RTSJ and had different objectives; JSR-166 did not attempt to address real-time requirements but rather sought to define a common set of idioms for general-purpose concurrent programming. In the interest of minimizing its assumptions about the underlying Java platform, the RTSJ makes no use of the JSR-166 facilities, and thus they will not be further considered in this paper.

3 A Note on Priority Inversion

A *priority inversion*[5] occurs when a ready or stalled thread is prevented from running while a lower priority thread is running. Some priority inversions are necessary and desirable; for example, stalling a high priority thread that attempts to acquire a lock owned by a lower priority thread. A major issue for real-time programming, which is affected by both language semantics and programming style, is to ensure that priority inversions are anticipated and that their durations are predictable and sufficiently short / bounded.

4 RTSJ Summary

The RTSJ needed to address several major issues that make Java unsuitable for real-time applications:

– The implementation-dependent nature of the thread semantics, making it impossible to write portable code that manages priority inversions and ensures that deadlines will be met
– The reliance on heap allocation and garbage collection for storage reclamation, resulting in unpredictable space and/or time behavior

[5] See [11] for further background information.

– Inadequate functionality in areas such as low-level programming and asynchrony

The RTSJ provides a class library, together with implementation constraints, that are designed to overcome these problems.

It offers a flexible scheduling framework based on the `Schedulable` interface and the `Thread` subclass `RealtimeThread` that implements this interface. The latter class overrides various `Thread` methods with versions that add real-time functionality, and supplies new methods for operations such as periodic scheduling. The `Schedulable` interface is introduced because certain schedulable entities (in particular, handlers for asynchronous events) might not be implemented as dedicated threads.

The RTSJ mandates a default POSIX-compliant preemptive priority-based scheduler – the so-called *base scheduler* – that supports at least 28 distinct priority levels beyond the 10 that are defined by Java's thread model. The implementation can provide other schedulers (e.g., Earliest Deadline First). For priority inversion management the RTSJ provides Priority Inheritance (required) and Priority Ceiling Emulation (optional).

To deal with Garbage Collection issues, the RTSJ defines various "memory areas" that are not subject to Garbage Collection: *immortal memory*, which persists for the duration of the application; and *scoped memory*, which is a generalization of the run-time stack. Restrictions on assignment, enforced in general at run time, prevent dangling references. The RTSJ also provides a `NoHeapRealtimeThread` class; instances of this class never reference the heap, may preempt the Garbage Collector at any time (even when the heap is in an inconsistent state), and thus do not incur Garbage Collector latency except in specialized circumstances as described below. A `NoHeapRealtimeThread` can reference objects in memory areas not subject to garbage collection (immortal or scoped memory).

Java's asynchrony issues are addressed through two main features. First, the RTSJ allows the definition of asynchronous events and asynchronous event handlers – these are basically a high-level mechanism for handling hardware interrupts or software signals. Secondly, the RTSJ extends the effect of `Thread.interrupt()` to apply not only to blocked threads, but also to real-time threads and asynchronous event handlers whether blocked or not.

The RTSJ supports absolute and relative high-resolution time, as well as one-shot and periodic timers. It also provides several classes for low-level programming. "Peek" and "poke" facilities for integral and floating-point data are available for *raw memory*, and *physical memory* may be defined with particular characteristics (such as flash memory) and used for general object allocation.

The RTSJ does not provide any specialized support for multiprocessor architectures.

5 Mutual Exclusion in the RTSJ

It would have been outside the scope of the RTSJ to introduce a new mutual exclusion facility, so the approach was to make the standard Java mechanism suitable for real-time applications. This entailed addressing two main issues:

– Managing priority inversions
– Dealing with asynchronous interruptibility

5.1 Managing Priority Inversions

The RTSJ offers a general, extensible, and somewhat ambitious approach to solving the priority inversion problem. It provides an abstract class `MonitorControl` and non-abstract subclasses `PriorityInheritance` and `PriorityCeilingEmulation`. `PriorityInheritance` is a singleton class; `PriorityCeilingEmulation` has distinct instances, one per ceiling level. The program can assign a `MonitorControl` instance (referred to as a *monitor control policy*) to any object, and can dynamically change the assignment. (Thus dynamic ceiling changes are allowed.) A default policy can also be established system-wide, so that it governs all objects subsequently constructed. The initial default policy is Priority Inheritance, but this can be overridden at system startup.

At any point in time a thread has a set of *priority sources*, namely its *base priority* (which reflects explicitly-invoked dynamic priority changes) and also other values depending on the monitor control policies governing the objects that the thread has locked. For example the ceiling of a `PriorityCeilingEmulation` instance is a priority source for any thread that has locked an object governed by this policy. A thread's *active priority* is the maximum of the values of its priority sources. Entering synchronized code adds a priority source; leaving synchronized code removes a priority source. Thus both actions affect the thread's active priority. Priority sources may be added/removed either synchronously or asynchronously.

The integration of both Priority Inheritance and Priority Ceiling Emulation into a common framework, with well-defined semantics, is new. (Posix includes both mechanisms but in an underspecified manner.) As will be pointed out below, the interactions between the two protocols led to an interesting formalization of the Priority Ceiling Emulation policy.

Under the base scheduler, access to synchronization locks is controlled by priority ordered queues, FIFO within priority. Thus a thread attempting to acquire a lock that is in use goes to the tail of the "stalled" queue associated with that lock.

Priority Inheritance If an object `obj` is governed by the `PriorityInheritance` instance and is currently locked by a thread `t1`, and a thread `t2` attempts to synchronize on `obj`, then `t2` becomes a priority source for `t1`. If `t1` has an active priority less than `t2`'s, then `t1`'s active priority will be boosted to that of `t2`. When `t1` releases the lock on `obj`, `t2` ceases serving as a priority source for `t1`, and `t1`'s active priority is adjusted accordingly.

Full (recursive) priority inheritance is required by the RTSJ. In the above description, if `t1` is stalled, waiting for an object locked by thread `t0`, then `t0`'s active priority will be boosted to that of `t2` as a result of `t2` attempting to synchronize on `obj`.

An interesting issue arises when a `NoHeapRealtimeThread t2` at high-priority `p2` attempts to synchronize on a Priority Inheritance-governed object (in immortal or scoped memory) locked by a heap-using thread `t1` at low priority `p1` while the Garbage Collector ("GC") is in progress. (We assume that the GC is running at a priority higher than `p1` but lower than `p2`). Ordinarily, `t1` would have its priority boosted to `p2`, but if this were the sole effect then `t1` would preempt the GC, thus leaving the heap inconsistent. The solution is to postulate a Priority Inheritance-governed lock on the heap. When `t1`, executing at inherited priority `p2`, attempts to access the heap, it fails because the

lock on the heap is in use by the GC. The GC inherits `t1`'s active priority `p2` and then runs until it reaches a safe preemption point, at which time it relinquishes the lock, allowing `t1` to continue. The effect is to induce a GC-induced latency for `t2` (and also for `NoHeapRealtimeThread`s executing at priorities higher than the GC's base priority and lower than `p2`). Since this somewhat defeats the purpose of `NoHeapRealtimeThread`s, the RTSJ provides "wait-free queues" as the recommended mechanism for communication between heap-using threads and `NoHeapRealtimeThread`s.

A *wait-free queue* is a data structure that allows concurrent "writers" and "readers" to store / retrieve items without interference but without blocking (on one side). The RTSJ supplies two classes to obtain this effect:

- A `WaitFreeWriteQueue` is intended for access by a single writer (generally a `NoHeapRealtimeThread`) and multiple readers (arbitrary threads). The write operations are non-synchronized and non-blocking (one method returns a status value, another method overwrites an existing element when the queue is full). The read operation is synchronized and will block when the queue is empty.
- A `WaitFreeReadQueue` is intended for access by a single reader (generally a `NoHeapRealtimeThread`) and multiple writers (arbitrary threads). The read operation is non-synchronized and non-blocking (if the queue is empty a special value is returned). The write operation is synchronized and will block when the queue is full.

Priority Ceiling Emulation Informally, Priority Ceiling Emulation (also known as *Highest Lockers Protocol*) is a technique in which a thread holding a lock executes at a *ceiling priority* associated with the lock. The ceiling value, which the application must define for the lock, is the highest priority of any thread that could hold that lock. The benefits of Priority Ceiling Emulation, compared with Priority Inheritance, are that it reduces the blocking time from priority inversions, and it prevents "nested monitor" deadlocks. Moreover, as exemplified by protected objects in Ada, in specialized circumstances (when blocking cannot occur while holding a lock) an especially efficient implementation is possible; this point will be further addressed below.

In order for Priority Ceiling Emulation to have the desired effect in terms of avoiding unbounded priority inversion, the priority of the locking thread must be no higher than the lock's ceiling, a condition that in general requires a run-time check. (On the other hand, once a lock is acquired, increasing the priority of the locker above the ceiling does not risk priority inversion, and indeed nested locking where the ceiling of the inner lock exceeds the ceiling of the outer lock will result in the thread's executing the inner code at a priority higher than the ceiling of the outer lock.)

This informal description omits an important detail: when an application assigns a ceiling value to a lock (and when the implementation checks that a thread's priority does not exceed the ceiling), which priority should be used: the *active priority*, or the *base priority*? Historically, and in fact in the initial release of the RTSJ, it is the active priority. However, this led to some subtle interactions between Priority Inheritance and Priority Ceiling Emulation. As an example, suppose low-priority thread `t` holds a Priority Inheritance lock and (asynchronously) inherits priority `p` from another thread that attempts to acquire that lock. If `t` then attempts to lock an object governed by Priority Ceiling Emulation with ceiling `c`, and `p` exceeds `c`, then `t` will suffer a ceiling violation

exception. If `t` does not provide a handler, then synchronized code (for the lock governed by Priority Inheritance) will be abruptly terminated, possibly leaving the object in an inconsistent state. This may be regarded as a design error – the programmer needs to understand the global locking behavior and assign priority ceilings accordingly – but the asynchronous nature of priority inheritance makes this difficult to solve in practice.

The RTSJ is being updated (in early 2005) to address such issues. The likely approach, inspired by suggestions from [12], consists of several main points: [6]

1. In the ceiling violation check, use the thread's base priority (or the ceiling of the most recently acquired Priority Ceiling Emulation lock, if there is such a lock) rather than the active priority
2. Treat a busy Priority Ceiling Emulation lock with Priority Inheritance semantics when a thread that owns a Priority Inheritance lock attempts to acquire it

Here's an example that shows the effect of these rules. Suppose thread `t1` at priority 10 locks an object `objPI` governed by Priority Inheritance. It then locks an object `objPCE15` governed by Priority Ceiling Emulation, with ceiling 15. The active priority for `t1` is 15. Another thread `t2`, at priority 20, attempts to lock `objPI`. This causes `t1`'s active priority to be boosted to 20. Suppose `t1` then attempts to lock object `objPCE17`, governed by Priority Ceiling Emulation with ceiling 17. Since `t1`'s base priority is used in the ceiling check, `t1` is allowed to obtain the lock, and it is still running at priority `20`. If it then attempted to acquire a Priority Ceiling Emulation lock with ceiling 16 it would fail, since it is currently holding a lock at a higher ceiling.

Now suppose thread `t3` at base priority 11 and active priority 30 (via Priority Inheritance) preempts `t1` and attempts to acquire the lock on `objPCE15`. Since `t3`'s base priority is less than the ceiling, there is no ceiling violation. If the rules simply provided for enqueueing `t3` on `objPCE15`'s lock, then we could have an unbounded priority inversion, since threads at priorities in the range 21 through 29 could preempt `t1` and run to completion while `t3` is waiting for the lock on `objPCE15`. This is where the 2nd rule above comes in. Since a thread that holds a Priority Inheritance lock is attempting to obtain the lock on `objPCE15`, the latter lock has Priority Inheritance semantics. In particular, the active priority of the thread `t1` that owns the Priority Ceiling Emulation lock is boosted to 30, thus avoiding the priority inversion.

Dynamic ceiling changes are permitted – this is a special case of the general principle that an object's monitor control policy may be updated – by assigning to the object a monitor control policy with a different ceiling value. This is only permitted for the thread that currently owns the lock on the object. There are some subtleties lurking in the details – for example, a thread may acquire a lock at ceiling `c1`, block on a call of `wait()`, and then be awakened to reacquire the lock after the ceiling has been lowered to `c2`. Should the ceiling violation check be performed? This issue is currently under discussion.

Support for the `PriorityCeilingEmulation` class is optional. The RTSJ designers felt that it was not as prevalent as Priority Inheritance in existing RTOSes or in real-time practice.

[6] These are captured more formally in the rules for a thread's priority sources and in the semantics for the `PriorityCeilingEmulation` class.

Example The following fragment illustrates several concepts:

- Defining a class's constructor to provide a Priority Ceiling Emulation policy for the new object, with the ceiling value passed as a constructor parameter
- Changing an object's monitor control policy dynamically, to be Priority Inheritance

```
public class Resource{
   public Resource( int ceiling ){
      synchronized(this){
         MonitorControl.setMonitorControl(
           this,
           PriorityCeilingEmulation.instance(ceiling) );
      }
   }
   ...
}

class MyRealtimeThread extends RealtimeThread{
   public void run(){
      Resource r = new Resource(20);
        // r is governed by Priority Ceiling Emulation with ceiling 20
      ...
      synchronized(r){
         MonitorControl.setMonitorControl(
                        r,
                        PriorityInheritance.instance() );
      }
      // r is now governed by Priority Inheritance
      ...
   }
}
```

Note that the invocation of `setMonitorControl` needs to be in code that is synchronized on the target object.

"Lock-Free" Priority Ceiling Emulation During the maintenance phase of the RTSJ, the Technical Interpretations Committee[7] considered adding support for "lock free" (queueless) Priority Ceiling Emulation, along the lines of the Ada 95 model. The main idea was that a thread that was synchronized on an object governed by a lock-free Priority Ceiling Emulation policy would not be allowed to block. Several designs were considered. The simplest scheme was to introduce a `PriorityCeilingEmulation` subclass, say `LockOptimizedPCE`. A thread that blocks while holding such a lock would suffer the throwing of an exception. However, this scheme interacts poorly with the

[7] After the RTSJ was approved, it went into a maintenance phase administered by a group known as the Technical Interpretations Committee.

dynamic nature of the RTSJ's monitor control policies. An object of a class with synchronized code that was not written under the lock-free assumption (i.e., which could block) might be assigned a `LockOptimizedPCE` policy. The consequential exception propagation could leave the object in an inconsistent state; this was considered unacceptable.

An alternative approach was also contemplated: a "marker interface"[8] `LockOptimizable`. A class that implemented this interface would need to ensure that all synchronized methods were non-blocking; if it blocked, an exception would be thrown. (This is different from the situation above, since here the author of the lock optimizable class knows in advance that synchronized code should not block.) An implementation could optimize such a class, Ada style, by using priority instead of actual locks / mutexes to enforce mutual exclusion. However, this raises several issues:

- An implementation that did not want to bother with the lock-free optimization could not simply ignore the fact that a class implemented the `LockOptimizable` interface, since the semantics required throwing an exception on blocking in synchronized code.
- Capturing the optimization on a class-wide basis was judged too coarse; in practice, it might be desirable to specify the lock-free optimization on a per-instance basis.

Since there was no consensus on how to best model lock-free Priority Ceiling Emulation, it was omitted from the RTSJ.

5.2 Interactions with Asynchronous Transfer of Control

A complete discussion of asynchronous transfer of control ("ATC") in the RTSJ and Ada is given in [13]. Here we consider only the issues related to mutual exclusion.

The RTSJ solves the problem of ATC out of synchronized code by defining it out of existence: synchronized code is simply not asynchronously interruptible ("AI"). If `t.interrupt()` is invoked while `t` is (lexically) inside synchronized code, the interrupt stays pending until the next time `t` attempts to execute AI code. This may occur during a later invocation of an AI method from within the synchronized code. This invocation will throw an `AsynchronouslyInterruptedException` ("AIE"), but such an exception occurrence is considered to be synchronous in this context, since in general a method invoked from synchronized code may throw any exception identified in its `throws` clause. Indeed, it would be good RTSJ style for a synchronized block to provide a handler for AIE if it calls any AI methods, since that will explicitly show that it anticipates such situations and will provide the necessary cleanup. (Such style is required if the synchronized code is the body of a synchronized method, since AIE is a checked[9] exception.)

[8] A *marker interface* is an empty interface. A common style in Java is to use a marker interface to define a boolean property for a class: a class has the property if and only if it implements the interface.

[9] Recall that a "checked" exception in Java is one for which the throwing method must explicitly provide either a handler or a `throws` clause.

6 Mutual Exclusion in Ada

This section discusses Ada's approach to managing priority inversions and also summarizes its handling of the interaction between mutual exclusion and asynchronous transfer of control.

6.1 Managing Priority Inversions

Ada has a mixed approach to priority inversion, depending on whether protected objects or passive tasks are used. In the former situation, the `Ceiling_Locking` policy prevents unbounded priority inversions; indeed, a task attempting to invoke a protected operation is deferred at most once by a lower-priority task. The assumption that blocking does not occur within a protected operation allows an extremely efficient ("lock free") implementation.

The situation with rendezvous is different, however. The rule that an accept statement is executed at the higher of the priorities of the calling and the called tasks is only part of what would be required for priority inheritance. This was a well-known issue in Ada 83, but the solution (full, recursive priority inheritance) was judged to impose too high an overhead and was intentionally omitted from Ada 95 [14]. As a result, it is possible to incur unbounded priority inversions. For example, if a high-priority task `T2` calls an entry of a low-priority server task `S` while `S` is serving a task `T1` at priority lower than `T2`'s, then `T2`'s priority is not required to be inherited by the server task. Thus `T2` can suffer an unbounded priority inversion from intermediate-priority tasks (higher than `T1`'s but lower than `T2`'s). The typical programming style to deal with this issue is to assign to each server task a priority at least as high as that of any of its callers, which effectively simulates the Priority Ceiling Emulation policy.

6.2 Interactions with Asynchronous Transfer of Control

The Ada model for ATC in code that is executed with mutual exclusion is similar in its basic approach to the RTSJ's – not surprising, since the RTSJ model was directly inspired by Ada's – but differs in detail. The similarity is that protected operations and accept statements are defined to be "abort deferred". The difference is that in Ada the abort deferral is "inherited" by invoked subprograms. In principle, this results in lower latency for RTSJ programs, since asynchronous interruptions are detected earlier. In practice this will likely not be an issue with protected objects, since protected operations are generally short.

7 Comparison

In the area of mutual exclusion it is useful to regard Ada 95 and the RTSJ as each addressing real-time issues that arose in the languages they were based on / extending, Ada 83 and Java, respectively. Ada 83 semantics were not strong enough to avoid unbounded priority inversions, and the "passive task" idiom was widely criticized by users as being inefficient and stylistically clumsy. The protected object / locking policy

mechanism was basically a completely new feature, though designed to fit in smoothly with existing Ada syntax. In contrast, the RTSJ introduced an API rather than a new language feature, constraining the Java semantics for synchronized code to help realize real-time behavior. In fact, such an approach was mandated by the Java Community Process, which prohibited syntactic extensions.

7.1 Generality

The RTSJ offers more generality than Ada's protected objects:

- There are no restrictions on what can be executed from synchronized code, whereas an Ada implementation may assume that potentially blocking operations are absent from protected operations.
- Both Priority Inheritance and Priority Ceiling Emulation are provided; Ada defines only the latter policy.

The RTSJ model is highly dynamic. Some objects may be governed by Priority Inheritance, others by Priority Ceiling Inheritance; indeed, the same object may be governed by different monitor control policies at different times. Ada's model is much more static; the object locking policy is established on a per-partition basis. An implementation may (but is not required to) provide locking control at a finer granularity.

In Ada 95, priority ceilings are constant. This is an inconvenient restriction, and one of the proposed revisions for Ada 2005 [15] provides additional generality by allowing a ceiling to be modified as a protected action on the affected object.

Both the RTSJ and Ada are extensible: the RTSJ through an API (subclassing `MonitorControl`) and Ada through pragmas.

Ada offers more generality in the area of volatile / atomic data, for example by allowing array components to be so specified and also by allowing whole arrays and records to be marked as volatile.

7.2 Software Engineering

Ada's approach enforces encapsulation: accesses to protected data are only permitted inside the implementation of protected operations. In contrast, the synchronized mechanism in Java and the RTSJ is independent of the encapsulation facility, and it is certainly possible to have unsynchronized access to an object's data even when all of the methods are synchronized.

The low-level nature of synchronized code and the `wait` / `notify` / `notifyAll` mechanism makes the expression of state-based mutual exclusion rather error prone in Java and thus also in the RTSJ. Ada's protected object/type model is a more reliable basis for mutual exclusion, with state notification automatic in the entry barrier evaluation semantics.

A drawback to Ada is that the error of executing a potentially blocking operation from protected code is a bounded error, not guaranteed to be caught at either compile time or run time. The looseness of the language standard thus results in implementation-dependent effects, although restricted profiles for high-integrity systems define deterministic behavior. For example, the Ravenscar profile [16] requires that `Program_Error` be raised.

A portability issue in the RTSJ is that Priority Ceiling Emulation is an optional feature.

7.3 Management of Priority Inversion

Both the RTSJ and Ada deal effectively with Priority Inversion, although some subtleties arise in both approaches. In the case of the RTSJ, an interaction between a `NoHeapRealtimeThread` and a heap-using thread attempting to synchronize on a shared object can result in GC-induced latencies for `NoHeapRealtimeThread`s. Further, the provision of both Priority Inheritance and Priority Ceiling Emulation, with specific semantics on their interactions, is an ambitious undertaking; some rules (for example the definition of priority inheritance semantics for priority ceiling emulation locks under some circumstances) may seem surprising. In Ada, the uses of server tasks may lead to priority inversion for callers unless a specific programming style is used (assigning high priorities to servers).

7.4 Efficiency

Comparing the performance of a specific feature in different languages is a challenge, since it is difficult to separate the implementation of that feature from other elements. Nevertheless it is possible to offer a qualitative analysis based on the anticipated runtime cost of the features and the practicality of optimizations.

Several factors give an efficiency advantage to Ada:

- Its static (per partition) approach to locking policies
- Its potential for lock-free priority ceiling emulation
- Its ability to specify that array components are atomically accessible
- Its efficiency in accessing protected data, which are always declared in protected specifications rather than bodies and thus can be referenced without a level of indirection

The much more dynamic Java model makes optimizations difficult, as evidenced by the problems in trying to capture lock-free priority ceiling emulation.

8 Conclusions

Both Ada and the RTSJ can be regarded as solving the underlying issues with mutual exclusion: arranging safe accesses, providing well defined semantics, and allowing priority inversion management and predictable performance. They achieve these goals rather differently, however, with both languages consistent with their respective underlying philosophies. Ada, especially through its protected type mechanism, offers encapsulation, freedom from certain kinds of deadlock, and the opportunity for efficient implementation; it achieves these at the expense of generality. The RTSJ trades off in the other direction. It reflects Java's highly dynamic nature by providing complete flexibility (for example, dynamic replacement of monitor control policies). On the other hand, optimization will likely be more difficult. Further, the low-level nature of

the RTSJ approach (defined in terms of lock acquisition and release) comes somewhat at the expense of program understandability.

Interestingly, both the RTSJ and Ada approaches to mutual exclusion have benefited from "cross fertilization". This is perhaps more evident in the case of the RTSJ design, which has directly borrowed some Ada ideas such as abort deferral in synchronized code. Also, the success of the Priority Ceiling Emulation policy in Ada was one of the reasons that it has been included in the RTSJ. The influence in the other direction has been more subtle, but several facilities proposed for Ada 2005 (such as dynamic ceiling priorities) may be due to the realization that flexibility and generality are often important in real-time systems, a fact that is one of the underpinnings of the RTSJ.

Acknowledgements

Anonymous referees provided many useful suggestions that helped improve this paper. I am also grateful to my colleagues on the RTSJ Technical Interpretations Committee for their many stimulating discussions of priority inversion management issues in the RTSJ: Rudy Belliardi, Greg Bollella, Peter Dibble, David Holmes, Doug Locke, and Andy Wellings.

References

1. S.T. Taft, R.A.Duff, R.L. Brukardt, and E. Ploedereder; *Consolidated Ada Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*; Springer LNCS 2219; 2000
2. Java Community Process; *JSR-001: Real-Time Specification for Java*; March 2004; `www.jcp.org/en/jsr/detail?id=1`
3. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull; *The Real-Time Specification for Java*, Addison-Wesley, 2000
4. J. Gosling, B. Joy, G. Steele, G. Bracha; *The Java Language Specification (2nd ed.)*; Addison Wesley, 2000
5. B. Brosgol; *A Comparison of the Concurrency and Real-Time Features of Ada 95 and Java*; Ada UK Conference; Bristol, UK; 1998.
6. S. Oaks and H. Wong; *Java Threads (3rd Edition)*; O'Reilly, 2004.
7. A. Wellings; *Concurrent and Real-Time Programming in Java*; John Wiley & Sons; 2004.
8. Java Community Process; *JSR-133: Java Memory Model and Thread Specification*; March 2004; `www.jcp.org/aboutJava/communityprocess/review/jsr133/`
9. P. Brinch Hansen; "Java's Insecure Parallelism"; *ACM SIGPLAN Notices*, V.34(4), April 1999.
10. Java Community Process; *JSR-166: Java Concurrency Utilities*; December 2003; `www.jcp.org/aboutJava/communityprocess/review/jsr166/`
11. L. Sha, R. Rajkumar, and J. Lehoczky; "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transaction on Computers*; Vol.39, pp.1175-1185; 1990.
12. A. Wellings and A. Burns; Informal communication; December 2004.
13. B. Brosgol and A. Wellings; "A Comparison of the Asynchronous Transfer of Control Facilities in Ada and the Real-Time Specification for Java", *Proc. Ada Europe 2003*, June 2003, Toulouse, France.
14. Intermetrics, Inc.; *Ada 95 Rationale*; January 1995.
15. ISO/IEC JTC1 / SC22 / WG9; *AI-327, Dynamic Ceiling Priorities*; November 2004; `www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00327.TXT?rev=1.13`
16. A. Burns; "The Ravenscar Profile", *Ada Letters*, XIX (4), pp.49-52, 1999.