Real-time Java, Part 3: Threading and synchronization

Threading and synchronization considerations in the Real-time Specification for Java

Patrick Gallop
Mark Stoodley

April 24, 2007

This article, the third in a six-part series on real-time Java™, examines aspects of threading and synchronization that an implementation of the Real-time Specification for Java (RTSJ) must support. You'll also learn about related threading and synchronization concerns that are essential to keep in mind when you develop and deploy real-time applications.

View more content in this series

Threading and synchronization are core features of the Java programming language, described in the Java Language Specification (JLS). The RTSJ extends the core capabilities of the JLS in a number of ways. (See Related topics for links to the JLS and RTSJ.) For example, the RTSJ introduces new real-time (RT) thread types that must follow a stricter scheduling policy than a regular Java thread. Another example is priority inheritance, a locking policy that defines how lock synchronization is managed when a lock is contested.

Understanding the management of priorities and priority queues is useful in understanding the RTSJ changes to threading and synchronization. Priorities are also an important tool for use by RT applications. This article describes aspects of RTSJ threading and synchronization by discussing how thread priorities and priority queues are managed. The discussion includes considerations that you should take into account when developing, deploying, and executing RT applications, including those you build using IBM WebSphere® Real Time (see Related topics).

Understanding regular Java threads

Threads defined in the JLS are called regular Java threads. A regular Java thread is an instance of the java.lang.Thread class that has an integer priority in the range from 1 to 10. To accommodate a number of execution platforms, the JLS allows a lot of flexibility in how you implement, schedule, and manage priorities of regular Java threads.
WebSphere VMs on Linux®, including WebSphere Real Time, use native threading services that the Linux operating system provides. You can appreciate aspects of Java threading and synchronization by understanding aspects of Linux threading and synchronization.

**Linux threading and synchronization**

The Linux operating system has provided different user-level threading implementations throughout its history. The Native POSIX Thread Library (NPTL) (see Related topics) is the latest strategic threading direction for Linux and the one used by WebSphere VMs. The advantages of NPTL over its predecessors are POSIX compliance and better performance. POSIX services are available at compile time through system header files. POSIX services are available at run time through the `libpthread.so` dynamic library and underlying Linux kernel support. The Linux kernel performs thread scheduling based on static controls, such as thread priority levels, and on certain dynamic conditions of the threads executing in the system.

POSIX allows you to create POSIX threads (pthreads) with different thread scheduling policies and priorities to meet differing application needs. Three such scheduling policies are:

- SCHED_OTHER
- SCHED_FIFO
- SCHED_RR

The SCHED_OTHER policy is used for traditional user tasks such as program-development tools, office applications, and Web browsers. SCHED_RR and SCHED_FIFO are intended to be used by applications requiring more deterministic and timely needs. The primary difference between SCHED_RR and SCHED_FIFO is that SCHED_RR time slices a thread's execution, whereas SCHED_FIFO doesn't. The SCHED_OTHER and SCHED_FIFO policies are used by WebSphere Real Time and are described in more detail below. (We don't cover the SCHED_RR policy, which WebSphere Real Time doesn't use.)

POSIX provides locking and synchronization support through the pthread_mutex data type. Pthread mutexes can be created with different locking policies. A locking policy is intended to affect execution behavior when more than one thread wants to acquire the same lock simultaneously. Standard Linux versions support a single default policy, whereas RT Linux versions also support the priority inheritance locking policy. We'll describe the priority inheritance policy in more detail in this article's Synchronization overview section.

Linux scheduling and locking involve managing first in, first out (FIFO) queues.

**Thread scheduling with regular Java threads**

The RTSJ indicates that the behavior of regular Java threads is as defined in the JLS. In WebSphere Real Time, regular Java threads are implemented using the Linux's POSIX SCHED_OTHER scheduling policy. The SCHED_OTHER policy is intended for applications such as compilers and word processors, not for tasks requiring more determinism.

In the 2.6 Linux kernel, the SCHED_OTHER policy supports 40 priority levels. These 40 priority levels are managed on a per-processor basis, which means that:
• Linux attempts to execute a thread on the same processor for cache performance reasons.
• Thread scheduling predominantly uses per-processor locks instead of per-system locks.

If required, Linux migrates threads from one processor to another to balance workloads.

Within each of the 40 priority levels, Linux manages an active queue and an expiration queue. Each queue contains a linked list of threads or is empty. The active and expiration queues are used for efficiency, load balancing, and other purposes. Logically, you can view the system as managing a single FIFO queue, termed a run-queue, for each of the 40 priorities. A thread is dispatched from the front of the nonempty run-queue with the highest priority. That thread is removed from the queue and executes for a period of time called a time quantum or time slice. When an executing thread expires its time quantum, it's placed at the end of the run-queue for its priority and assigned a new time quantum. By dispatching from a queue's head and placing expired threads at a queue's tail, execution happens in a round-robin fashion within a priority.

The time quantum given to a thread depends on the thread's assigned priority. Threads with higher assigned priorities are given longer time quanta to execute. To prevent threads from hogging the CPU, Linux dynamically raises or lowers a thread's priority based on factors such as whether the thread is I/O or CPU bound. A thread can voluntarily relinquish its time slice by yielding (such as by calling Thread.yield()), or a thread can relinquish control by blocking, at which point the thread waits for an event to occur. One such event could be triggered by a lock being released.

The VM in WebSphere Real Time doesn't explicitly assign the 10 regular Java thread priorities across the range of the 40 SCHED_OTHER Linux thread priorities. All regular Java threads, regardless of their Java priority, are assigned the default Linux priority. The default Linux priority is in the middle of the range of the 40 SCHED_OTHER priorities. By using the default, regular Java threads execute with fairness in the sense that, regardless of dynamic priority adjustments that Linux may make, every regular Java thread in a run-queue eventually executes. This assumes a system with only regular Java threads executing and not, for example, a system that's executing RT threads.

Note that both the VM in WebSphere Real Time and the non-RT version of the WebSphere VM use the SCHED_OTHER policy and default priority assignment for regular Java threads. By using the same policy, both JVMs have similar, but not identical, thread scheduling and synchronization characteristics. Changes in the WebSphere Real Time class libraries, changes in the JVM, and changes in the JIT compiler in support of the RTSJ as well as the introduction of the RT Metronome garbage collector (see Related topics) make it impossible for applications to run with identical timing and performance characteristics in both JVMs. During IBM WebSphere Real Time testing, timing differences surfaced race conditions (in other words, bugs) in test programs that had run fine for years on other JVMs.

A note about this article's code examples

The code examples in the following sections use spin loops with yielding. This is done for illustrative purposes only; such practices should not be used in an RT application.
Code example using regular Java threads

Listing 1 shows a program using regular Java threads that determines how many iterations of a loop can be executed by each of two created threads in a five-second interval:

Listing 1. Regular Java threads

class myThreadClass extends java.lang.Thread {
    volatile static boolean Stop = false;

    // Primordial thread executes main()
    public static void main(String args[]) throws InterruptedException {
        // Create and start 2 threads
        myThreadClass thread1 = new myThreadClass();
        thread1.setPriority(4);    // 1st thread at 4th non-RT priority
        myThreadClass thread2 = new myThreadClass();
        thread2.setPriority(6);    // 2nd thread at 6th non-RT priority
        thread1.start();           // start 1st thread to execute run()
        thread2.start();           // start 2nd thread to execute run()

        // Sleep for 5 seconds, then tell the threads to terminate
        Thread.sleep(5*1000);
        Stop = true;
    }

    public void run() { // Created threads execute this method
        System.out.println("Created thread");
        int count = 0;
        for (;Stop != true;) {    // continue until asked to stop
            count++;
            Thread.yield();   // yield to other thread
        }
        System.out.println("Thread terminates. Loop count is " + count);
    }
}

The program in Listing 1 has three user threads that are regular Java threads:

- The primordial thread:
  - It's the main thread that is implicitly created on process startup and executes the main() method.
  - main() creates two regular Java threads: one thread with priority 4 and another thread with priority 6.
  - This main thread intentionally blocks itself by sleeping for five seconds by making a call to Thread.sleep().
  - After the five-second sleep, this thread directs the other two threads to terminate.
- A thread at priority 4:
  - This thread is created by the primordial thread, which executes the run() method containing a for loop.
  - The thread:
    1. Increments a count in each loop iteration.
    2. Voluntarily relinquishes its time slice by making a call to Thread.yield().
    3. Terminates when the main thread has requested. Just prior to termination, the thread prints out the loop count.
• A thread at priority 6: This thread performs the same actions as the thread at priority 4.

If this program is run on a uniprocessor or unloaded multiprocessor system, each thread prints out close to the same for loop iteration count. In one run, the program printed:

Created thread
Created thread
Thread terminates. Loop count is 540084
Thread terminates. Loop count is 540083

If the call to `Thread.yield()` is removed, the loop counts for the two threads can be close but are highly unlikely to be identical. Both threads are assigned the same default priority in the `SCHED_OTHER` policy, so both threads are given the same time slice to execute. Because the threads execute identical code, they should have similar dynamic priority adjustments and execute in a round-robin fashion from the same run-queues. But, because the thread at priority 4 executes first, it should have a slightly larger share of the five-second execution interval and print out a higher loop count.

**Understanding RT threads**

An RT thread is an instance of `javax.realtime.RealtimeThread`. The RTSJ requires that an implementation of the specification must provide a contiguous range of at least 28 priorities for RT threads. These priorities are referred to as real-time priorities. The starting value of the RT priority range is not mandated by the specification other than it must be numerically higher than 10 -- the highest priority value given to a regular Java thread. For portability reasons, application code should use the new `PriorityScheduler` class's `getPriorityMin()` and `getPriorityMax()` methods to determine the range of available RT priority values.

**Motivation for RT threads**

Thread scheduling in the JLS is imprecise and provides only 10 priority values. The POSIX `SCHED_OTHER` policy, as implemented by Linux, meets the needs of a variety of applications. But the `SCHED_OTHER` policy has some undesirable characteristics. Dynamic priority adjustments and time slicing can happen at unpredictable times. The number (40) of `SCHED_OTHER` priorities is not that large, and a range of these priorities is already assumed by applications with regular Java threads and for dynamic priority adjustments. The JVM also requires priorities for internal threads for special purposes such as garbage collection (GC).

The lack of determinism, the need for more priority levels, and the desire for compatibility with existing applications motivated the need for extensions that would provide new scheduling capabilities to Java programmers. Classes in the `javax.realtime` package, as described in the RTSJ, provide these capabilities. In WebSphere Real Time, the Linux `SCHED_FIFO` scheduling policy addresses RTSJ scheduling needs.

**Thread scheduling with RT Java threads**

In WebSphere Real Time, 28 RT Java priorities are supported and have a range of 11 to 38. APIs of the `PriorityScheduler` class should be used to retrieve this range. This section describes more
details of thread scheduling as described in the RTSJ and aspects of the Linux SCHED_FIFO policy that more than meet the RTSJ's requirements.

The RTSJ views RT priorities as being logically implemented by the run-time system maintaining a separate queue for each of the RT priorities. The thread scheduler must dispatch from the head of the highest-priority queue that is not empty. Note that if no threads are in any of the queues for RT priorities, a regular Java thread is dispatched that executes as described in the JLS (see Thread scheduling with regular Java threads).

A dispatched thread with an RT priority is allowed to execute until it blocks, voluntarily relinquishes control by yielding, or is preempted by a thread with a higher RT priority. A thread with an RT priority that voluntarily yields is placed at the back of the queue for its priority. The RTSJ also requires that such scheduling be done in constant time and cannot vary based on factors such as the number of RT threads that are executing. The 1.02 version of the RTSJ applies these rules to uniprocessor systems; the RTSJ doesn't mandate how the scheduling happens on multiprocessor systems.

Linux provides all the proper RTSJ scheduling requirements with the SCHED_FIFO policy. The SCHED_FIFO policy is intended for RT and not for user tasks. SCHED_FIFO differs from the SCHED_OTHER policy by providing 99 priority levels. SCHED_FIFO does not time slice threads. Also, the SCHED_FIFO policy doesn't dynamically adjust the priority of RT threads except through the priority inheritance locking policy, which the Synchronization overview section describes. Because of priority inheritance, priority adjustments are required by the RTSJ.

Linux provides constant time scheduling for both RT threads and regular Java threads. On multiprocessor systems, Linux tries to emulate the behavior of a single global queue of RT threads that are dispatched to available processors. This is closest to the spirit of the RTSJ but it does differ from the SCHED_OTHER policy used for regular Java threads.

### Problematic code example using RT threads

Listing 2 modifies the code from Listing 1 to create RT threads instead of regular Java threads. The use of java.realtime.RealtimeThread instead of java.lang.Thread indicates the distinction. The first thread is created at the 4th RT priority level, and the second thread is created at the 6th RT priority level, as determined by the getPriorityMin() method.

```java
import javax.realtime.*;

class myRealtimeThreadClass extends javax.realtime.RealtimeThread {
    volatile static boolean Stop = false;

    // Primordial thread executes main()
    public static void main(String args[]) throws InterruptedException {

        // Create and start 2 threads
        myRealtimeThreadClass thread1 = new myRealtimeThreadClass();
        thread1.setPriority(PriorityScheduler.getMinPriority(null)+ 4);
        myRealtimeThreadClass thread2 = new myRealtimeThreadClass();
        thread2.setPriority(PriorityScheduler.getMinPriority(null)+ 6);
```
thread1.start();           // start 1st thread to execute run()
thread2.start();           // start 2nd thread to execute run()

// Sleep for 5 seconds, then tell the threads to terminate
Thread.sleep(5*1000);
Stop = true;
}

public void run() { // Created threads execute this method
    System.out.println("Created thread");
    int count = 0;
    for (;Stop != true;) {    // continue until asked to stop
        count++;
        // Thread.yield();   // yield to other thread
    }
    System.out.println("Thread terminates. Loop count is "+count);
}

The modified code in Listing 2 has a few problems. If the program is run in a uniprocessor environment, it never terminates and prints only the following:

Created thread

This result can be explained by the behavior of RT thread scheduling. The primordial thread remains a regular Java thread and runs with a non-RT (SCHED_OTHER) policy. As soon as the primordial thread starts the first RT thread, the RT thread preempts the primordial thread and the RT thread runs indefinitely because it’s not bound by time quanta nor does the thread block. After the primordial thread is preempted, it’s never allowed to execute, so it never starts the second RT thread. Thread.yield() has no effect on allowing the primordial thread to execute -- because yielding logically places the RT thread at the end of its run-queue -- but the thread scheduler dispatches this same thread again because it’s the thread at the front of the run-queue with the highest priority.

The program also fails on a two-processor system. It prints the following:

Created thread
Created thread

The primordial thread is allowed to create both RT threads. But after creating the second thread, the primordial thread is preempted and is never allowed to tell the threads to terminate because the two RT threads execute on the two processors and never block.

The program runs to completion and produces a result on a system with three or more processors.

**RT code example that runs on a single processor**

Listing 3 shows the code modified so that it runs correctly on a uniprocessor system. The main() method's logic is moved to a "main" RT thread that has the 8th RT priority. This priority is higher than the priority given to either of the two other RT threads that the main RT thread creates. Having the highest RT priority allows this main RT thread to create the two RT threads successfully and also lets the main RT thread preempt the currently running thread when it wakes up from its five-second sleep.
Listing 3. Modified RT thread example

```java
import javax.realtime.*;

class myRealtimeThreadClass extends javax.realtime.RealtimeThread {
    volatile static boolean Stop = false;

    static class myRealtimeStartup extends javax.realtime.RealtimeThread {
        public void run() {
            // Create and start 2 threads
            myRealtimeThreadClass thread1 = new myRealtimeThreadClass();
            thread1.setPriority(PriorityScheduler.getMinPriority(null) + 4);
            myRealtimeThreadClass thread2 = new myRealtimeThreadClass();
            thread2.setPriority(PriorityScheduler.getMinPriority(null) + 6);
            thread1.start();  // start 1st thread to execute run()
            thread2.start();  // start 2nd thread to execute run()

            // Sleep for 5 seconds, then tell the threads to terminate
            try {
                Thread.sleep(5*1000);
            } catch (InterruptedException e) {
            }
            myRealtimeThreadClass.Stop = true;
        }
    }

    // Primordial thread creates real-time startup thread
    public static void main(String args[]) {
        myRealtimeStartup startThr = new myRealtimeStartup();
        startThr.setPriority(PriorityScheduler.getMinPriority(null) + 8);
        startThr.start();
    }

    public void run() { // Created threads execute this method
        System.out.println("Created thread");
        int count = 0;
        for (;Stop != true;) { // continue until asked to stop
            count++;
            // Thread.yield(); // yield to other thread
        }
        System.out.println("Thread terminates. Loop count is "+ count);
    }
}
```

When this program is run on a uniprocessor, it prints the following:

```
Created thread
Thread terminates. Loop count is 32767955
Created thread
Thread terminates. Loop count is 0
```

The program's output shows that all threads run and terminate, but only one of the two threads executes an iteration of the `for` loop. This output is explained by considering the priorities of the RT threads. The main RT thread runs until it blocks by the call to `Thread.sleep()`. The main RT thread creates the two RT threads, but only the second RT thread at the 6th RT priority is allowed to run while the main RT thread sleeps. This thread runs until the main RT thread wakes up from its sleep and tells the threads to terminate. Once the main RT thread has itself terminated, the thread at the 6th priority is allowed to execute and terminate. It does so and prints out the loop count with nonzero value. After this thread terminates, the thread at 4th RT priority is allowed to
run, but it merely bypasses the for loop because it has been directed to terminate. That thread prints out the zero loop count value before it terminates.

**Threading considerations for RT applications**

This section discusses features of RT threading that you need to take into account when porting applications to use RT threads or when writing new applications to exploit RT threading.

**New extensions for RT threads**

The RTSJ specifies facilities for creating RT threads to start at a specific or relative time. You can create a thread to run certain logic at specified time intervals or periods. You can define a thread to have an `AsynchronousEventHandler` (AEH) execute (fire) when this logic does not complete within the specified period. You can also define limits on the type and amount of memory that a thread can consume such that an `OutOfMemoryError` is thrown if the thread consumes more than that limit. These facilities are available only to RT threads, not to regular Java threads. You can find more information about these facilities in the RTSJ.

**Thread.interrupt() and pending exceptions**

`Thread.interrupt()` behavior is extended for RT threads. This API interrupts blocked threads as described in the JLS. This exception is also raised in a method that a user has explicitly marked as interruptible by adding the `.Throws AsynchronouslyInterruptedException` clause to the method declaration. The exception is also sticky to the thread in the sense that the user must explicitly clear the exception; otherwise it remains bound -- termed pending -- to the thread. If the user doesn't clear the exception, a thread can terminate with this sticky exception still bound. This error is benign if the thread terminates in the "normal" fashion but not in applications that do their own form of pooling of RT threads. That is, a thread could be returned to a pool with an `InterruptedException` still bound. In this case, the code that does thread pooling should explicitly clear the exception; otherwise, the exception might be spuriously thrown when the pooled thread, with the bound exception, is reassigned.

**Primordial thread and application dispatch logic**

The primordial thread is always a regular Java thread -- not an RT thread. The first RT thread is always created by a regular Java thread. That RT thread would immediately preempt the regular Java thread if there were insufficient available processors to run both the RT thread and the regular Java thread simultaneously. Preemption could prevent the regular Java thread from creating further RT threads or other logic to get the application into an appropriately initialized state.

You can avoid this problem by performing application initialization from a high-priority RT thread. This technique might be required for applications or libraries that do their own form of thread pooling and thread dispatching. That is, thread-dispatch logic should be run either at high priority or within a high-priority thread. Appropriate priority selection for execution of thread-pooling logic can help prevent problems encountered in thread enqueueing and dequeueing.
Runaway threads
Regular Java threads execute with time quanta, and the dynamic priority adjustments the scheduler performs, based on CPU usage, allow all regular Java threads to execute eventually. In contrast, RT threads are not bound by time quanta, and the thread scheduler does not do any form of dynamic priority adjustments based on CPU usage. These differences in the scheduling policies between regular Java threads and RT threads create the chance that runaway RT threads can occur. Runaway RT threads can take control of the system and prevent any other applications from running, prevent users from signing on to the system, and so on.

During development and testing, one technique that can help lessen the impact of runaway threads is to set a limit on the amount of CPU a process can use. On Linux, limiting CPU consumption causes a runaway thread to be killed when it exhausts the CPU limit. Also, programs that monitor system state or provide system login should run at high RT priority so that the program is allowed to preempt the problematic threads.

Mapping from Java priorities to operating-system priorities
On Linux, the POSIX SCHED_FIFO policy provides 99 RT priorities in the integer range of 1 to 99. From this system range, priorities 11 through 89 are used by the WebSphere VM, and a subset of this range is used to implement the 28 RTSJ priorities. The mapping of 28 RT Java priorities to this range of POSIX system priorities is described in IBM WebSphere Real Time documentation. But application code shouldn't make a dependence on this mapping and should rely only on the relative ordering, at the Java level, of the 28 RT priorities. This lets the JVM remap this range and provide improvements in future WebSphere Real Time releases.

Applications could use SCHED_FIFO priority 1 or priority 90 to implement daemons or other RT processes if they require a RT priority above or below those used in WebSphere Real Time.

JNI AttachThread()
The Java Native Interface (JNI) lets you use the JNI AttachThread() API to attach threads created in C code to a JVM, but the RTSJ does not make changes or provisions to the JNI interface for attaching RT threads. Accordingly, applications should avoid creating POSIX RT threads in C code that they intend to attach to the JVM. Instead, such RT threads should be created in the Java language.

Forked processes and RT priorities
A thread can fork another process. On Linux, the primordial thread of the forked process inherits the priority of the parent thread that forked it. If the forked process is a JVM, the primordial thread of the JVM is created with an RT priority. This would violate the order that regular Java threads, such as the primordial thread, have lower scheduling preference to RT threads. To prevent this situation, the JVM forces the primordial thread to have a non-RT priority -- that is, have a SCHED_OTHER policy.

Thread.yield()
Thread.yield() yields execution only to a thread at the same priority and never yields execution to a thread with a higher or lower priority. Yielding only to a thread of the same priority means that
Thread.yield() is of questionable use in an RT application that uses more than one RT priority. You should avoid Thread.yield() unless it's absolutely necessary.

**NoHeapRealtimeThreads**

javax.realtime.NoHeapRealtimeThread (NHRT) is another new thread type in the RTSJ that is a subclass of javax.realtime.RealtimeThread. An NHRT has the same scheduling characteristics as RT threads as we’ve described them, except that an NHRT isn't preempted by GC and an NHRT can't read or write to the Java heap. NHRTs are an important aspect of the RTSJ and will be discussed in a later article in this series.

**AsynchronousEventHandlers**

AsynchronousEventHandler (AEHs) are new with the RTSJ and can be viewed as a form of RT thread that executes when an event occurs. For example, you can set an AEH to fire at a specific or relative time. AEHs also have the same scheduling characteristics as RT threads and come in heap and no-heap flavors.

**Synchronization overview**

Many Java applications use Java threading features directly, or the applications being developed use libraries that involve more than one thread. A primary concern in multithreaded programming is to make sure the program runs correctly -- thread safe -- in a system where multiple threads are executing. Making a program thread safe could require serializing access to data that is shared between more than one thread using synchronization primitives such as locks or atomic machine operations. Programmers of RT applications are often challenged to make the programs execute within certain time constraints. To meet the challenge, they may need to know implementation details, implications, and performance attributes of the components they’re using.

The remainder of this article discusses aspects of the core synchronization primitives that the Java language provides, how these primitives change in the RTSJ, and some implications that RT programmers need to be aware of when they use these primitives.

**Java language synchronization overview**

The Java language provides three core synchronization primitives:

- Synchronized methods and blocks allow a thread to lock an object on entry and unlock the object on exit (to the method or block).
- `Object.wait()` releases the object lock and the thread waits.
- `Object.notify()` unblocks a thread waiting on the object. `notifyAll()` unblocks all waiters.

Threads doing `wait()` and `notify()` must currently have locked the object.

*Lock contention* occurs when a thread tries to lock an object that is already locked by another thread. When this happens, the thread failing to acquire the lock is put into a logical queue of lock contenders for the object. Similarly, several threads could have done `Object.wait()` on the same object, and so there’s a logical queue of waiters for the object. The JLS doesn't specify how these queues are managed, but the RTSJ does stipulate the behavior.
Priority-based synchronization queues

The spirit of the RTSJ is that all queues of threads are FIFO and priority based. Priority-based FIFO behavior -- as in the earlier synchronization example where the highest-priority thread is chosen to execute next -- also applies to the queues of lock contenders and lock waiters. From a logical viewpoint, there is a FIFO priority-based queue of lock contenders similar to the execution queue of threads waiting to execute. There’s also a similar queue of lock waiters.

When a lock is released, the system chooses the thread from the front of the highest-priority queue of contenders to try to lock the object. Similarly, when a `notify()` is done, the thread from the front of the highest-priority queue of waiters is unblocked from its wait. A lock release or lock `notify()` operation is analogous to a scheduling dispatch operation in the sense that the thread at the head of the highest-priority queue is acted on.

To support priority-based synchronization, modifications needed to be made to RT Linux. Changes were also necessary to the VM in WebSphere Real Time to delegate to Linux the responsibility of selecting which thread is unblocked when a `notify()` operation is performed.

Priority inversion and priority inheritance

**Priority inversion** is a condition where a high-priority thread is blocked on a lock held by a low-priority thread. A medium-priority thread might preempt the low-priority thread while it holds the lock and run in preference to the low-priority thread. The inversion of priorities delays both the low- and the high-priority thread from making progress. The delays caused by priority inversion could prevent critical deadlines from being met. This condition is shown in the first timeline of Figure 1.

**Priority inheritance** is a technique for avoiding priority inversion. Priority inheritance is mandated by the RTSJ. The idea behind priority inheritance is that at the point of lock contention, the lock holder's priority is boosted to that of the thread wishing to acquire the lock. The lock holder's priority is "deboosted" back to its base priority when it releases the lock. In the scenario just described, the low-priority thread runs at high priority when lock contention occurs until the point in time that it releases the lock. On the lock release, the high-priority thread locks the object and continues execution. The medium-priority thread is prevented from delaying the high-priority thread. The second timeline in Figure 1 shows how the locking behavior of the first timeline changes when priority inheritance is in effect.
Figure 1. Priority inversion and priority inheritance

It's possible that, at the point the high-priority thread tries to acquire the low-priority thread's lock, the low-priority thread itself is blocked on a different lock held by another thread. In this case, the low-priority thread and that other thread are boosted. That is, priority inheritance could require priority boosting and deboosting of a group of threads.

Priority inheritance implementation

Priority inheritance is provided through Linux kernel functionality that is exported to user space through POSIX locking services. A solution entirely in user space is undesirable because:

- The Linux kernel can be preempted and is subject to priority inversion. Priority inheritance is also needed for certain system locks.
- Attempting a solution in user space causes difficult-to-solve race conditions.
- Priority boosting requires a kernel call anyway.

The POSIX lock type is a `pthread_mutex`. A POSIX API for creating a `pthread_mutex` lets the mutex implement the priority-inheritance protocol. There are POSIX services for locking a `pthread_mutex` and unlocking a `pthread_mutex`. These are the points where priority-inheritance support takes effect. Linux performs all locking in user space in cases without contention. When lock contention occurs, priority boosting and synchronization-queue management is done in kernel space.

The WebSphere VM uses the POSIX locking APIs to implement the core Java language synchronization primitives we described earlier and for priority-inheritance support. User-level C code can also use these POSIX services. At the point of a Java level lock operation, a unique `pthread_mutex` is allocated and bound to a Java object using an atomic machine operation. At the point of a Java level unlock operation, the `pthread_mutex` is unbound from the object using an atomic operation, provided there was no contention for the lock. With contention, a POSIX lock and unlock operation would trigger Linux kernel priority inheritance support to occur.

To help minimize mutex allocation and locking time, the JVM manages a global lock cache and a per-thread lock cache where each cache contains unallocated `pthread_mutex`es. A mutex in the thread-specific cache is acquired from the global lock cache. Before it is put into the thread lock cache, the mutex is prelocked by the thread. Uncontested unlock operations return a locked
mutex to the thread lock cache. The assumption here is that uncontested locking is the norm, and POSIX-level locking is both reduced and amortized by reusing prelocked mutexes.

The JVM itself has internal locks used to serialize access to critical JVM resources such as the thread list and the global lock cache. These locks are based on priority inheritance and are held for short time intervals.

**Synchronization considerations for RT applications**

This section covers some features of RT synchronization that can help developers porting applications to use RT threads or writing new applications to exploit RT threading.

**Lock contention between regular Java threads and RT threads**

An RT thread can be blocked on a lock held by a regular Java thread. When this happens, priority inheritance takes over so that the regular Java thread is priority boosted to the RT thread's priority for as long as it holds the lock. The regular Java thread inherits all the scheduling characteristics of an RT thread:

- The regular Java thread runs with `SCHED_FIFO` policy and so the thread does not time slice.
- Dispatch and yielding happen from the RT run-queue of the boosted priority.

This behavior reverts to `SCHED_OTHER` when the regular Java thread releases the lock. If either of the threads created in **Listing 1** were to run while holding a lock required by an RT thread, that program would not terminate and would exhibit the same problems we described in **Problematic code example using RT threads**. Because this situation is possible, doing spin loops and yielding is not advisable for any threads executing within a real-time JVM.

**Lock contention between NHRTs and RT threads**

An NHRT could block on a lock held by an RT thread (or a regular Java thread for that matter). While the RT thread holds the lock, GC could preempt the RT and indirectly preempt the NHRT. The NHRT needs to wait until the RT is no longer preempted by GC and then release the lock before the NHRT has a chance to execute. Preemption of an NHRT by GC can be a serious problem if the NHRT is performing a time-critical function.

The deterministic garbage collector in WebSphere Real Time keeps pause times under one millisecond, making the NHRT preemption more deterministic. If such pauses are not tolerable, you could bypass the problem by avoiding lock sharing between NHRTs and RT threads. If locking is mandatory, you could consider having RT- and NHRT-specific resources and locks. For example, applications that implement thread pooling could consider separate pools and pool locks for NHRTs and RT threads.

Also, the `javax.realtime` package provides the following:

- The `WaitFreeReadQueue` class for the primary purpose of passing objects from an RT thread to NHRT.
• The `WaitFreeWriteQueue` class for the primary purpose of passing objects from an NHRT to an RT thread.

These classes guarantee that an RT thread, which could be blocked by GC while performing a wait-free operation, won't hold a lock that an NHRT requires in performing a wait-free operation.

**Synchronization in the `javax.realtime` package**

Certain `javax.realtime` methods are intentionally not synchronized because overheads in synchronization occur even when a lock is uncontested. If synchronization is required, the caller is responsible for wrapping the required `javax.realtime` methods in synchronized methods or blocks. A programmer must consider adding such synchronization when the `java.realtime` package's methods are used.

**Synchronization in the core JLS packages**

In contrast, core JLS services such as `java.util.Vector` are already synchronized. Also, certain core JLS services can do some internal locking to serialize certain shared resources. Because of this synchronization, when using core JLS services, you must exercise care to avoid the problem of NHRT preemption by GC (see Lock contention between NHRTs and RT threads).

**Uncontested locking performance**

Benchmarking and instrumentation of non-RT applications have shown that locking is predominantly uncontested. Uncontested locking is also believed to be the predominant case in RT applications, particularly when existing components or libraries are reused. It can be desirable to have a small and deterministic cost for locking that is known to be uncontested but where synchronization directives are hard to avoid or remove.

As we described earlier, an uncontested lock operation involves some setup and an atomic machine instruction. An unlock operation involves an atomic machine operation. The setup for a lock operation involves allocation of a prelocked mutex. The allocation is considered the largest variable cost in an uncontested lock operation. `RealtimeSystem.setMaximumConcurrentLocks()` can help control this variable cost.

`RealtimeSystem.setMaximumConcurrentLocks(int numLocks)` cause the VM in WebSphere Real Time to preallocate `numLocks` mutexes to the global lock cache. The global lock cache feeds the per-thread lock caches. By using this `RealTimeSystem` API, you can reduce the chance that lock initialization happens within a time-critical code region.

`RealtimeSystem.getMaximumConcurrentLocks()` can be used to help decide what number should be used in the `setMaximumConcurrentLocks()` call, but note that `getMaximumConcurrentLocks()` gives lock usage at the point of the call, not the high-water mark. A future RTSJ version may provide an API to provide the high-water mark. Do not provide ridiculously large values for the value of `numLocks` because the call to `setMaximumConcurrentLocks()` could take an inordinate amount of time and memory to create that many locks. Also note that this API is defined to be JVM specific, so other JVMs might ignore the call or provide different behavior.
Contested locking performance

A thread can simultaneously hold more than one lock, and these locks might have been acquired in a certain order. The set of all such locking patterns forms a lock hierarchy. Priority inheritance can mean the boosting and deboosting of a group of threads. The number of threads in the group should be no larger than the depth of the deepest lock hierarchy possible in the system. By keeping lock hierarchies shallow, which translates to locking as few objects as possible, you can affect the maximum number of threads that need to be priority adjusted.

Times in synchronization operations

Object.wait(long timeout, int nanos) provides nanosecond granularity for a relative wait operation. The HighResolutionTime.waitForObject() API is similar to Object.wait() and provides relative and absolute times that can be specified with nanosecond granularity. In WebSphere Real Time, both APIs are implemented with underlying POSIX locking wait services. These underlying services provide, at best, microsecond granularity. When required, and for portability, the getResolution() method of the javax.realtime package's Clock class should be used to retrieve the resolution for the executing platform.

Summary

The RTSJ extends and tightens threading and synchronization capabilities for the Java programmer through new RT classes and APIs in the javax.realtime package. In WebSphere Real Time, these capabilities are implemented by the RT version of the Linux kernel, modifications to POSIX threading, and modifications to the JVM itself. The deeper understanding you now have of RTSJ threading and synchronization can help you avoid problems as you write and deploy RT applications.
Related topics

- **Real-time Java series**: Read the other parts in this series.
- **The Java Language Specification, Third Edition**: The JLS is the complete specification of the Java language's syntax and semantics.
- **JSR 1: Real-time Specification for Java**: You'll find the RTSJ at the Java Community Process site.
- "**IBM WebSphere Real Time V1.0 delivers predictable response times using Java standards**": Read the product announcement for WebSphere Real Time.
- **WebSphere Real Time**: WebSphere Real Time lets applications dependent on a precise response time take advantage of standard Java technology without sacrificing determinism.
- **Real-time Java technology**: Visit the authors' IBM alphaWorks research site to find cutting-edge technologies for real-time Java.
- **Metronome**: Learn more about Metronome, the GC technology incorporated in WebSphere Real Time.

© Copyright IBM Corporation 2007

[Trademarks](www.ibm.com/developerworks/ibm/trademarks/)