Real-Time Java

by Benjamin M. Brosolg

Under the auspices of Sun Microsystems’ Java Community Process an effort to improve Java’s real-time capabilities began in 1999. This effort culminated in the Real-Time Specification for Java (RTSJ), which adds constraints to the Java Virtual Machine spec and defines a new class library, javax.realtime. RTSJ-compliant JVMs can be considered Real-Time Java Virtual Machines (RTJVMs).

Preemption

The RTSJ’s RealtimeThread class extends java.lang.Thread. It enlarges the priority range (at least 28 levels) and supports a variety of scheduling approaches through a flexible multi-threading framework. The default scheduler is preemptive and priority based. Prioritization applies also to threads that are waiting for locks and queues [BB: ARE THERE EXPLICITLY SUCH THINGS AS LOCKS AND QUEUES? - MB]. Specific RTJVMs may provide additional alternative scheduling algorithms, such as earliest deadline first.

A real-time thread’s behavior is established by several parameters passed to the thread’s constructor. Scheduling parameters establish the thread’s priority and can also include other characteristics. Release parameters identify whether the thread is periodic, aperiodic (releasable based on events that occur at unpredictable times), or sporadic (aperiodic with a minimum interarrival time). Any periodic thread has a period and a deadline. To assist with feasibility analysis and computation overrun detection, you may specify a cost (maximum computation time per period) and also handlers that are invoked (as callbacks) in a failsoft fashion when the cost is exceeded or a deadline is missed.

The RTSJ provides a general mechanism for preventing priority inversion. By default, synchronized code is managed via priority inheritance. Additionally, the RTSJ allows the user to specify priority ceiling emulation. These policies can be applied globally or on an object-by-object basis, using semantics consistent with the POSIX standard. RTJVM implementations are free to add alternative policies.

Determinism

The RTSJ does not include a garbage collection algorithm that meets the predictability and performance requirements of real-time applications, and it doesn’t require the RTJVM to implement one. Rather, the RTSJ provides for program-definable memory areas that are not subject to garbage collection.

Objects that will never need to be freed and reusable global pools of various kinds can be allocated in the immortal memory area. Objects that are only needed within a particular method or code block can be allocated from a scoped memory area, which is a generalization of C’s run-time stack. Scoped areas are automatically created upon entry to that code and freed (as a whole) on the way out.

The garbage collector never need run in the immortal or any scoped memory area. Restrictions on assignment prevent dangling references; for example, neither heap nor immortal objects can reference scoped objects.

Another innovation is the introduction of a special kind of real-time thread that doesn’t access the general heap and, therefore, can preempt the garbage collector at any time. Instances of the NoHeapRealtimeThread class can only allocate objects in immortal and scoped memory areas. Since they never reference the general heap, such threads incur no latency when preempting the garbage collector. The garbage collection process will not interfere with their timely and responsive computation.

Peeking, poking, and more

Conveniently, the RTSJ also adds to Java a mechanism for accessing memory-mapped I/O. A new RawMemory class allows peeking and poking of integer or floating-point data at program-specified addresses. Physical memory areas may be defined with given characteristics (such as RAM [BB: Or? -MB]) and used for object allocation.

A programmer can now also write
event handlers for asynchronous events, including interrupts. The definition of asynchronous event handlers is quite general, so each can have priority, deadline, cost, and other related parameters.

Furthermore, RTSJ programs can also initiate asynchronous transfers of control via a generalization of the `interrupt()` method in `java.lang.Thread`. The effect is essentially to force an exception in another thread; this exception will either be handled immediately or deferred, depending on the state of the target thread. Timeouts and thread termination are achieved through this mechanism.

Other RTSJ functionality includes a set of classes for dealing with time and timers. Several kinds of time (absolute, relative) are provided, and you can express periodic behavior as a frequency (say, 30 times per second). Extreme resolutions down to 1ns are possible, if supported by the implementation.

**Multitasking made easy**

The program in Listing 1 illustrates some of the basic capabilities of the RTSJ. Its behavior is the same as that of the example in my previous piece on Ada (January 2003, p. 39).

Although Java is by design not up to the task of real-time programming, suitable extensions can provide the needed functionality and predictability. The Real-Time Specification for Java is one such approach to these issues. More information about the RTSJ can be found at [www.rtj.org](http://www.rtj.org).

---

**Listing 1  A real-time Java multitasking program**

```java
import javax.realtime;

class Outputter extends RealtimeThread {
  (Outputter(PriorityParameters priorityParams, PeriodicParameters
  periodParams)
  { super(prioParams, periodParams);
  }

  volatile boolean finished = false;

  public void run() {
    int count = 1;
    while (!finished) {
      System.out.println(count++);
      this.waitForNextPeriod();
    }
  }
}

class Tasking_Example {
  public static void main(String[] args) throws InterruptedException {
    Outputter out = new Outputter(
      new PriorityParameters(PriorityScheduler.instance().getMinPriority() + 15),
      new PeriodicParameters(null, new RelativeTime(1000, D), null, null, null, null));

    out.start();
    Thread.sleep(20000); // 20 seconds
    out.finished = true;
    out.join();
    System.out.println("Terminating");
  }
}
```

Ben Brosgol is a senior member of the technical staff at Ada Core Technologies. He was a primary member of the team that designed the Real-Time Specification for Java and is currently on the Technical
Interpretation Committee. He has a PhD in applied mathematics from Harvard. Contact him at brosgol@gnat.com.