Real-Time Java

David Holmes
Senior Java Technologist
Java SE VM Real-Time Group
Sun Microsystems
What is Real-Time?

- Simple definition: The addition of \textit{temporal constraints} to the \textit{correctness} conditions of a program
  - “When” is as important as “what”
  - “A late answer is a \textit{wrong} answer”
- “real-time” \textbf{does not mean} “real-fast”
  - Going faster helps but ...
- \textbf{Predictability} is the key
- Non-real-time systems have many sources of unpredictable behaviour
  - Performance is based on the average-case
Example Temporal Constraints

- **Deadline**: started task must complete by a given time
  - Once a request for a trade is received, it must execute within 5ms

- **Latency**: difference between when an event happens and when it is seen to have happened
  - Stop button handler must respond within 500us of a press

- **Jitter**: Variance in the time interval between events
  - The input sensor must be sampled every 1ms +/- 100us
Latency and Jitter

**Latency** is the measure of how long it takes the system to respond to an event. **Jitter** is the variability of a measured value.

For both: lower is better.

Logic execution

---

Latency

Interval jitter = \( \frac{(\text{max(interval)} - \text{min(interval)})}{2} \)

Latency jitter = \( \text{max(latency)} - \text{min(latency)} \)
Why Real-Time Java?

- Same reasons as for using Java – but applied to real-time application domains
  - Traditional C/C++/assembler implementations difficult to write, debug, maintain

- Real-time software loads are evolving
  - Increase both in size and complexity
  - Traditional, low-level programming no longer provides the required level of abstraction

- Single solution for real-time and non-real-time code
  - Re-use of people, tools, knowledge
Real-Time Specification for Java (RTSJ) JSR-001

- Started in 1998. Experts from many communities
  - Real-time systems, embedded systems, Ada, Java, academia and industry
- The standard that defines how real-time behavior must occur within Java technology
  - Therefore, the only real-time Java technology!
- APIs and semantic enhancements which allow Java code developers to correctly reason about and control the temporal behavior of applications
  - Better, high-level, portable abstractions
  - 100% Java technology
JSR-1 Evolution

1998
Real-Time Specification for Java (JSR-001) proposal submitted

Many companies represented: IBM, Sun, Ajile, Apogee, Motorola, Nortel, QNX, Thales, TimeSys, WindRiver

2002
JSR-001 approved by the Java Community Process

TimeSys Reference Implementation

2005
RTSJ update proposal submitted (JSR-282)

Several JSR-1 compliant products (Apogee, IBM, Sun)

RTGC Available in IBM’s JVM

2007
RTGC added to Sun’s JSR1-compliant JVM

JSR-1 APIs added to RTGC enhanced JVMs

2008
New Sun/IBM JSR
Uses of RTSJ

- Industrial automation
- Aeronautic/Aerospace

Inverted pendulum control problem

- Telecommunications
- Banking/Financial

Boeing Scan-Eagle UAV

...
Sun's Java Real-Time System

- Sun's implementation of the RTSJ
  - 100% compliant with Java technology and RTSJ 1.0.2

- Java RTS 2.0 highlights
  - Based on Java Platform, Standard Edition 5
  - Runs on Solaris 10 OS,
    - SPARC® technology, and x86/x64 platforms
    - Relies on Solaris platform built-in real-time capabilities

- Java RTS 2.1 Early Access
  - Runs on real-time Linux on x86
    - SUSE Linux Enterprise Real Time 10
    - Red Hat Enterprise MRG 1.0 (beta)
Java RTS 2.x Platforms

- From embedded single-board computers
- To carrier-grade blade servers
- To enterprise servers
Java RTS Latency and Jitter Numbers

- Example of Java RTS 2.0 on Solaris 10 / SPARC
  - Maximum Jitter: < 5 microseconds
  - Maximum Latency: < 10 microseconds
- As good as the best commercial RTOS
  - And often better, particularly on faster processors with more memory
- But No Silver Bullet!
  - Enhanced predictability comes at expense of throughput
    - How much depends on platform, hardware, #CPUs, amount of memory, JVM configuration and the application itself
RTSJ System Model

Non Real-Time
java.lang.Threads

Soft Real-Time
Realtime Threads
Real-Time GC*

Hard Real-Time
NoHeapRealtime Threads
Highest priority
Tightly bounded jitter and latencies

* RTGC not specified by RTSJ
Key RTSJ Features

- Scheduling and Dispatching
  - Managing schedulable objects removes unpredictable scheduling
- Synchronization
  - Priority inversion avoidance removes unpredictable delays
- Memory Management
  - Alternatives to the heap to removes unpredictable GC
- Asynchronous events and handlers
  - Event driven programming model
- Time, Clocks and Timers
Threads and Schedulable Objects

```
   «interface»
  java.lang.Runnable
    ↑
  java.lang.Thread
     ↑
    RealtimeThread
       ↑
      NoHeapRealtimeThread
       ↓
     AsyncEventHandler
    ↑
   «interface»
    Schedulable
```

- java.lang.Runnable
- java.lang.Thread
- RealtimeThread
- NoHeapRealtimeThread
- AsyncEventHandler
- Schedulable
RTSJ Scheduling

- Schedulable Object managed by a Scheduler
- One defined scheduler: PriorityScheduler
  - Singleton: PriorityScheduler.instance()
  - Execution eligibility based on an integer priority value
    - Higher the value the higher the priority
  - Minimum of 28 unique, consecutive priority levels
    - Distinct from (and >) the 10 java.lang.Thread priorities
  - getMinPriority(), getMaxPriority(), getNormPriority()
Fixed Priority Preemptive Scheduling

• Highest priority schedulable object always runs
  • Higher priority SO preempts lower priority one
• Schedulable object runs until it blocks
  • No time-slicing! No “fairness”
  • Caution: “greedy” real-time threads can “hang” your system!
• PriorityScheduler doesn’t change priority except for priority inversion avoidance
  • Contrast with dynamic scheduler: Earliest Deadline First
• All internal system queues maintained in priority order;
  • Run queue, monitor entry queue, monitor wait-set
• Necessary for predictability, but not sufficient ...
Java RTS Predictability on Solaris OS

- Uses Solaris Real-Time (RT) scheduling class
  - 60 priority levels
  - Highest range of thread priorities in the system
- JVM locked into memory
  - No page swap in/swap out
- Processor set bindings
  - RTTs and NHRTs
- Class pre-loading and initialization
- Initialization Time Compilation (ITC)
  - No runtime execution variance
Characterizing Schedulable Objects

- Schedulable objects have execution characteristics
  - Scheduling behaviour, release pattern, memory constraints
- Characteristics represented by “parameter” objects:
  - `SchedulingParameters`, `ReleaseParameters`
- Parameter objects “tag” the SO and contain data
  - Priority, deadline, deadline-miss handler, cost
- An SO can link to one parameter object of a given kind
  - Initially set at SO construction, can be modified later
- A parameter object can be associated with many SO’s
  - Any change to the parameter object affects all the SO’s
Scheduling Parameters

- SchedulingParameters
  - PriorityParameters
  - ImportanceParameters

Not used by PriorityScheduler
Release Parameters

- **Periodic Parameters**
  - Periodic release pattern
  - Start time
  - Period

- **Aperiodic Parameters**
  - Unknown release pattern

- **Sporadic Parameters**
  - Unknown release pattern, but with minimum inter-arrival time (MIT)

- **Execution cost**
- **Deadline**
- **Cost overrun handler**
- **Deadline miss handler**
Example: Periodic Real-time Thread

```
RelativeTime period = new RelativeTime(5,0); // 5ms period

AbsoluteTime start =
    Clock.getRealtimeClock().getTime().add(50,0); // now+50ms

PeriodicParameters pp = new PeriodicParameters(start, period);

int prio = PriorityScheduler.instance().getNormPriority();

PriorityParameters priop = new PriorityParameters(prio);

RealtimeThread rtt = new RealtimeThread(priop, pp) {
    public void run() {
        while (workToDo) {
            // do work
            if (!RealtimeThread.waitForNextPeriod())
                throw new Error("Deadline missed");
        }
    }

    ...
}

rtt.start();
```
Synchronization: Priority Inversion

Tries to acquire same lock - thread blocked

High priority task is blocked indefinitely by low priority task

Medium priority tasks prevent low level task from completing and releasing lock

Priority

Lock Acquired
Priority Inversion
A Real World Example: Mars Pathfinder

- Spacecraft had two high priority periodic tasks
  - Data distribution, bus scheduling
  - Data distribution must complete before bus scheduling starts
- Low priority data gathering task acquired internal lock via call to IPC mechanism
- Distribution task got blocked trying to acquire same lock
- Other medium priority tasks prevented data gathering task from completing and releasing lock
- Bus scheduler task detects distribution task has not completed in required time and takes action
  - Reboots spacecraft!
Priority Inversion Avoidance in RTSJ

- Priority Inheritance Protocol (Required)
  - Thread holding lock gets priority boosted to that of blocking thread until lock is released
  - No application code changes required

- Priority Ceiling Emulation Protocol (Optional)
  - Each object lock is assigned a “ceiling” priority
    - Highest active priority of any thread that will acquire it
  - Thread sets its priority to the ceiling value when it acquires the lock, and drops it when lock released

- Applies to Java object monitors only
  - synchronized methods / blocks
Wait-Free Data Transfer Queues

- Allows non-blocking data exchange between no-heap SO’s and heap-using SO’s
  - If NHRTT synchronizes with RTT then GC can preempt it
- **WaitFreeReadQueue**
  - Single reader can perform non-blocking read
  - Multiple writers can perform synchronized/blocking writes
- **WaitFreeWriteQueue**
  - Single writer can perform non-blocking write
  - Multiple readers can perform synchronized/blocking reads
- **WaitFreeDequeue**
  - Combined WFRQ and WFWQ
Memory Management

- C/C++ memory management is completely under program control
  - `malloc()`, `free()`
  - Obvious disadvantages for memory leaks, invalid pointers

- Java uses automatic memory management
  - Eliminates problems of `free()`
  - Introduces non-deterministic behavior to application as normal GC cannot be controlled directly
RTSJ Memory Management

- Goal: “to not interfere with the ability of real-time code to exhibit deterministic behavior”
- Issues with normal heap-management in Java
  - Allocation times can vary dramatically
  - GC is unpredictable in its frequency and execution
- Real-time GC was not an option in 2000!
- RTSJ introduces the notion of allocation context
  - The memory area used when code executes `new`
- Each area has different access and GC properties
- Access to physical memory
RTSJ Memory Areas

MemoryArea

HeapMemory

ScopedMemory

ImmortalMemory

VTMemory

LTMemory

Linear Time allocation

Variable Time allocation

time

time

time

enter(Runnable r)
memoryConsumed()
memoryRemaining()
newArray(...)
newInstance(...)

Immortal Memory

- Shared amongst all threads
- Objects allocated here are never garbage collected
  - Live until end of application
  - Objects referred to can also not be garbage collected!
- Three mechanisms for allocating immortal memory
  - Implicit
    - Static initialization, interned strings, string literals, Class objects
  - Direct request
    - `newInstance()`, `newArray()`
  - Execute code with immortal as current allocation context
    - `enter()`, `executeInArea()`
Scoped Memory

- Lifetime of an object is determined by the scope
  - Objects exist as long as scope is “in use”
  - When scope no longer “in use” it can be cleared and so is “empty” the next time it becomes “in use”

- Scope usage is governed by complex rules
  - **Single parent rule**: all entry to a scope must be from the same parent scope (or else heap or immortal)
  - **Assignment rules**: an object in one memory area can not hold references to objects in a shorter-lived area
    - Scopes can't hold references to objects in a child scope
    - Heap/Immortal can never hold references to objects in scope
  - Run-time checks enforce the rules
Physical Memory

- Physical memory can be mapped to particular HW
  - `PhysicalMemoryManager`
  - `ImmortalPhysicalMemory`
  - Scoped physical memory
    - `VTPhysicalMemory, LTPhysicalMemory`
- Raw memory access allows read/write of any address
  - Primitive types only
  - `RawMemoryAccess`
  - `RawMemoryFloatAccess`
Java RTS Memory Management: Real-time Garbage Collection

- RTGC allows latency guarantees to be extended to RealtimeThreads (not just NHRTTs)
  - Under certain conditions
- Critical RTT can preempt the GC
  - And allocate from a reserved buffer
  - And avoid GC induced latencies
- Cost of RTGC is paid for by non-critical threads
  - No silver bullet!
- Very sophisticated, very configurable, very flexible
Event Based Programming

AsyncEvent
+ addHandler()
+ removeHandler()
+ fire()

AsyncEventHandler
+ handleAsyncEvent()

Schedulable

BoundAsyncEventHandler
Times and Clocks

Clock

HighResolutionTime

AbsoluteTime

RelativeTime

RationalTime

(millis, nanos) pair

Clock.getRealtimeClock()

- High resolution
- Monotonic
- Time zero == UNIX epoch
Timers

Time based release of async event handlers

AsyncEvent

Timer

OneShotTimer

PeriodicTimer
Resources

http://java.sun.com/javase/technologies/realtime/index.jsp
http://www.jcp.org (JSR-001, JSR-282)
http://www.rtsj.org
Real-Time Java

David Holmes
Senior Java Technologist
Java SE VM Real-Time
Sun Microsystems