Java

JavaOne℠

LAB-5538: The Real-Time Java™ Platform Programming Challenge: Taming Timing Troubles

David Holmes
Frederic Parain
Sun Microsystems

Instructor-Led Hands-on Labs

- Instructor(s) will provide background for exercises
- Exercises are self-paced
  - Hard-copy and online lab guides are available
  - Use suggested durations as a guide
- Raise your hand at any time for assistance
- To get the most out of the lab...
  - **Read** the lab guide, especially the background
  - **Don't** just copy and paste the solutions

Housekeeping

- Before you leave, please fill out a survey and hand it to someone before you leave
  - We really want to know what you think!
- Please log out of your machine when done
- Please look around to make sure you have all of your belongings
  - The hard copies of the lab guides are yours to keep

What is “Real-Time”?

- Simple definition:
  - The addition of temporal constraints to the correctness conditions of a program
    - “When” is as important as “what”
- Typical temporal constraint: deadline
  - Others are latency, jitter
- “real-time” does not mean “real-fast”
  - Predictability is the key
    - Are operations bounded in time?
- Classifications: hard-, soft-, non- real-time
  - How critical is timing to correctness

Where Could You Use Real-Time Java™ Technology?

- Military/Aerospace
  - Command & control systems
- Telecommunication Infrastructure
  - VoIP, PBX, IMS, new 3G services
- Banking/Finance
  - Meet customer QoS and regulatory requirements for pricing/trading
- Industry
  - Factory automation, process control
- ...

Real-Time Specification for Java™ (RTSJ) Mission Statement

[To extend] *The Java™ Language Specification* and *The Java™ Virtual Machine Specification* [to provide] an Application Programming Interface that will enable the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints (also known as real-time threads)

Key Functional Areas

- Thread Scheduling & Dispatching
  - Priority-preemptive scheduling
- Memory management
  - Allocation contexts without garbage collection
- Asynchronous Actions
  - Internal events, external “happenings”, and handlers
- Time, Clocks and Timers
- Synchronization
  - Priority inversion avoidance

Sun Java™ Real-Time System 2.2 Beta

- Based on Java™ 5.0 platform
  - Hotspot client VM (32-bit and 64-bit)
- Runs on:
  - Solaris™ 10 operating system (SPARC® and x86)
  - Linux with real-time POSIX extensions (x86)
- Implements RTSJ 1.0.2 plus:
  - Real-time garbage collection (RTGC)
    - Including additional monitoring via MXBeans
  - Multi-processor support
    - Including interrupt shielding and processor sets
  - Initialization-time Compilation (ITC)
    - Avoids unpredictable effects of JIT compilation
  - Tools to aid in analyzing timing problems

Exercises

Exercise 1: Creating Periodic Real-Time Threads

Expected duration: 30 minutes

Exercise 2: Debugging Deadline-misses using the Thread Scheduling Visualizer

Expected duration: 30 minutes

Exercise 3: Real-time Data Communication (almost!)

Expected duration: 20 minutes

Exercises (continued)

Exercise 4: More Debugging of Deadline-Misses using the Thread Scheduling Visualizer

Expected duration: 15 minutes

Exercise 5: Real-Time Data Communication

Expected duration: 15 minutes

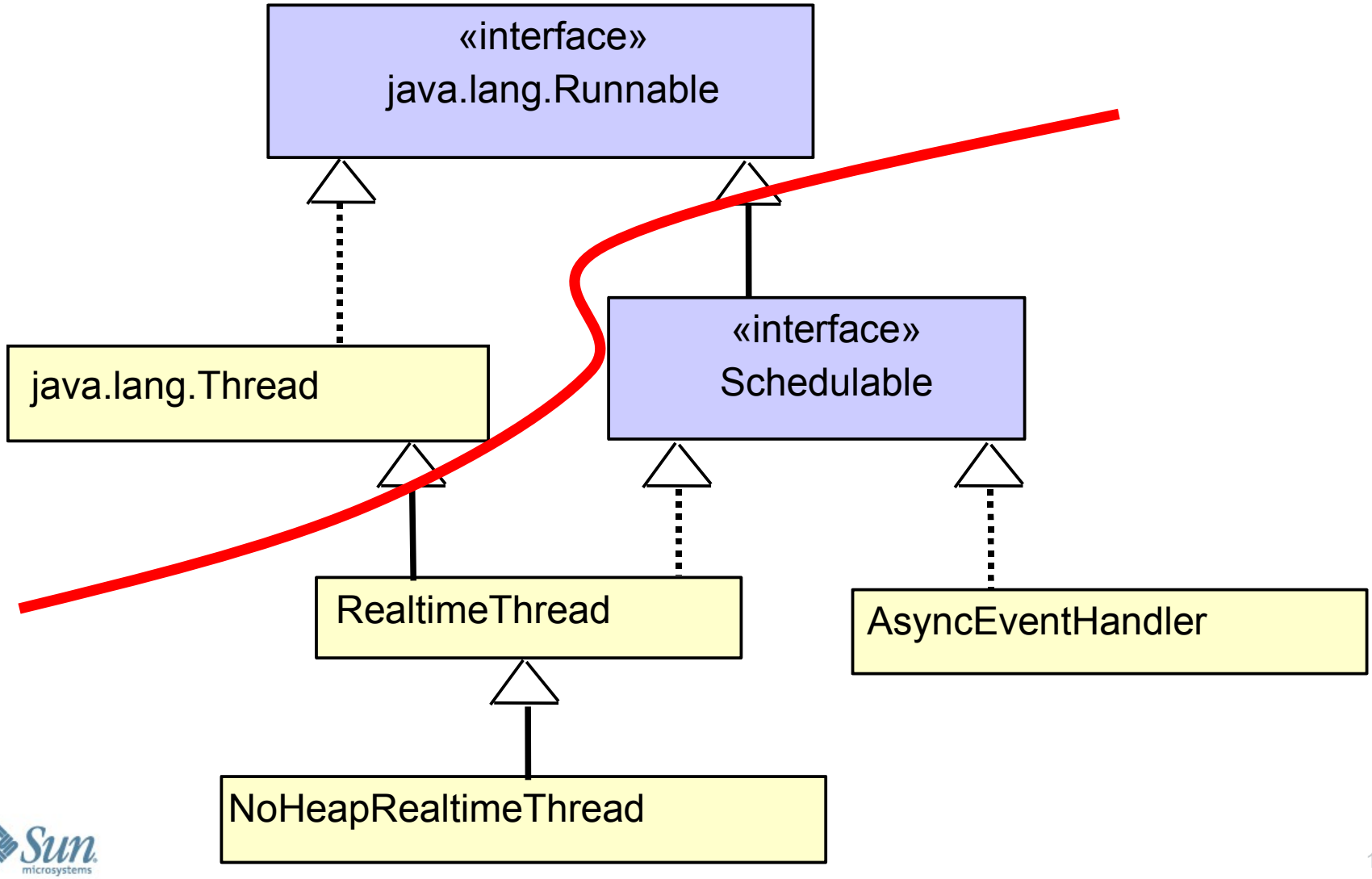
Getting Started

- If you have not logged in, log in with
  - username: lab5538
  - password: hol009
- Online lab guide will open in a browser window
- All necessary software and lab files are already installed on your lab machine
- Start from exercise 1

Background for Exercise 1

- Threads, Scheduling and Parameter Objects
- Clocks and Time objects
- Memory Areas

Introducing Schedulable Objects



Real-Time Threads

- Most features of RTSJ only apply to real-time threads or asynchronous event handlers
- **`RealtimeThread`** extends **`java.lang.Thread`**
  - The base class for real-time threads
  - Specializes the semantics of some **`java.lang.Thread`** methods
    - **`setPriority`**, **`interrupt`**
- Writing a periodic real-time thread:

```
while(workToBeDone) {
    // do work
    waitForNextPeriod();
}
```

For start time, S, and period T, periodic threads are released according to: S + nT (n=0, 1, 2 ...)

NoHeapRealtimeThreads

- Extends **`RealtimeThread`**
- Forbidden from accessing objects in the regular Java heap
  - Must operate in *scoped* or *immortal* memory
  - Run-time checks to enforce this rule
- Execution eligibility should be higher than GC
  - Requires execution eligibility > all heap-using threads & GC threads
  - No synchronizing on an object shared with heap-using threads
    - Or sharing scoped memory areas
  - Programmers responsibility to get it right!

Scheduling

- Notion of a **`Schedulable`** object
  - Real-time threads and async event handlers
  - Interface extends **`java.lang.Runnable`**
- **`Scheduler`** manages scheduling/dispatching of schedulable objects
- Default scheduler is the **`PriorityScheduler`**
  - Enforces priority preemptive scheduling
  - Supports periodic release of real-time threads
    - **`RealtimeThread`**: **`boolean waitForNextPeriod()`**
  - Supports sporadic and aperiodic async event handlers

Parameter Objects

- Parameter objects abstract the properties of schedulable objects related to
  - Release characteristics (periodic, sporadic, aperiodic)
  - Scheduling (priority-based, importance-based)
  - Memory (allocation rates and limits)
- Can be set at construction time or explicitly, e.g.
  - `thread.setReleaseParameters(myParams)`
- **`PriorityParameters`** hold priority value
  - `PriorityParameters(int priority)`
- **`PeriodicParameters`** hold release information
  - Period, start-time, deadline, deadline-miss handler ...
  - `PeriodicParameters(RelativeTime period)`

Memory Areas

- `MemoryArea` represents an allocation context
  - `enter(Runnable r)`
    - Executes `r` with this area as the allocation context
  - `newInstance(Class c, ...)`
    - Reflectively creates an object in this memory area
- Three kinds of memory areas
  - Heap: `HeapMemory.instance()`
    - The normal Java heap
  - Immortal: `ImmortalMemory.instance()`
    - Memory that is never garbage-collected
  - Scope: `ScopedMemory`
    - Lifetime of objects determined by use of scope
    - When no-longer in-use objects are reclaimed; next use sees empty scope
    - Strictly enforced rules to avoid “dangling references”

Clocks and Times

- `Clock` represents a time-source
  - `AbsoluteTime getTime(AbsoluteTime dest)`
  - `RelativeTime getResolution()`
- Real-time clock is a nanosecond-precision clock
  - `Clock.getRealtimeClock()`
  - Resolution is platform-dependent
  - Only pre-defined clock in RTSJ
- Time objects are a (millisecond,nanosecond) pair associated with some clock
  - `long millis`, `int nanos`

Clocks and Times (cont)

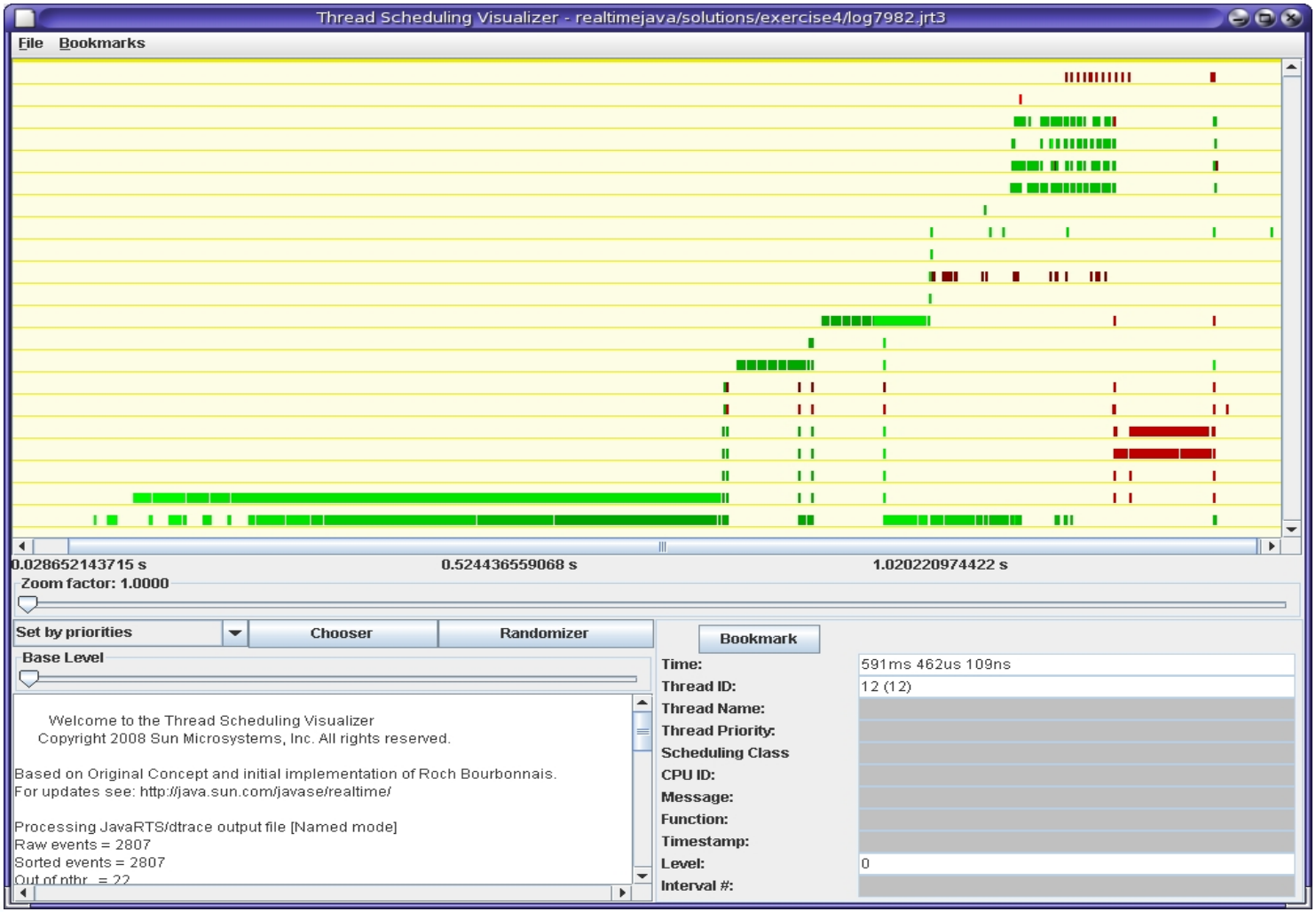
- **`HighResolutionTime`**: base class
  - Supports simple set/get methods
    - `void set(HighResolutionTime other)`
- **`AbsoluteTime`**: An actual time for a given clock
  - `AbsoluteTime(long millis, int nanos)`
  - Supports time arithmetic: add/subtract
    - `RelativeTime subtract(AbsoluteTime other, RelativeTime dest)`
    - Subtract `other` time from this time, store in `dest` and return it
- **`RelativeTime`**: A time interval on a given clock
  - Similar methods to **`AbsoluteTime`**

Exercise 1

Background for Exercise 2

- Running and understanding the Thread Scheduling Visualizer

The Thread Scheduling Visualizer



What is the Thread Scheduling Visualizer?

- A tool that presents a graphical time-line of per-thread events
  - Originally only scheduling events (on-cpu, off-cpu) hence TSV
- Events are recorded using DTrace scripts
  - VM defined events
  - Application defined using `com.sun.rtsjx.DTraceUserEvent`
- What can you do?
  - Inspect the timeline of events
  - Measure the elapsed time between events
  - Zoom in/out to observe different timescales
- Popup displays of certain event information:
  - User-defined events
  - Call-stacks

TSV Basics

- One row of event data per-thread
  - Ordered by thread creation time – oldest at bottom
    - Main thread on bottom
    - Application threads on top (generally)
    - VM & library threads in between
- Default color scheme:
  - Green: Time Sharing/Interactive/Fair Scheduling
  - Blue: System
  - Red: Real-Time
  - The lighter the shade, the higher the priority
- Thread information in bottom right pane
- Bookmarks can be used to measure intervals between events
  - Click first event, click second -> elapsed time shown in bottom left pane

Exercise 2

Background for Exercise 3

- Asynchronous Events
- Asynchronous Event Handlers
- Timers

Asynchronous Events

- Many real-time systems are *reactive*
  - They respond to events occurring in their environment
- RTSJ defines three explicit kinds of events with which handlers can be associated
  - Programmatic events: `AsyncEvent`
    - Triggered by invoking `fire()`
  - Timers: `OneShotTimer`, `PeriodicTimer`
    - Triggered by the passage of time
  - POSIX Signal handler:
    - Triggered by the delivery of a signal to the process
- Plus implicit: deadline-miss handlers, cost overrun handlers
- External named events: "*happenings*" attach to `AsyncEvents`
  - Implementation defined events triggered externally

Asynchronous Event Handlers

- Defines the work of the event handler but is not a thread itself
  - `handleAsyncEvent()` is the method to override
    - Or pass a `Runnable` to the constructor, like threads
- Handlers can be associated with one or more events
- When the event is triggered the handler is released as-if in its own real-time thread of control
  - `BoundAsyncEventHandler` has a dedicated thread
  - "server threads" acquire all the attributes of the handler
    - Priority, deadline, cost, miss handler, memory constraints
- Each trigger increases the *fire count* of the handler
  - System calls `handleAsyncEvent` in a loop while fireCount > 0
    - Fire count is decremented each time
    - Sporadic MIT constraints need to be accounted for

Timers

- A `Timer` is an `AsyncEvent` triggered by the passage of time
  - Methods `fire()` and `bindTo()` are disabled
- Timers have a complex lifecycle
  - `start()` tells the timer to start tracking time – *active*
  - `stop()` tells the timer to stop tracking time – *in-active*
  - `destroy()` prevents the timer from being used any more
    - All handlers are removed
    - All methods throw `IllegalStateException`
  - `enable()` allows the timer to fire
  - `disable()` prevents the timer from firing
    - If the fire time passes this is known as a *skipped-firing*
  - `isRunning()` returns true if the timer is *active* and *enabled*
  - `getFireTime()` returns absolute time when next firing is due

Periodic Timers

- Re-usable periodic timer: fires repeatedly as the period falls due
- `PeriodicTimer(HighResolutionTime start, RelativeTime period, A…E…H handler)`
  - `start` indicates when the timer should initially fire
  - `period` defines the interval between firings
    - A zero period means “act like a one-shot”
  - `handler` is initial handler
- Absolute start
  - First firing when start is reached and timer is active and enabled; If start is in the past when timer is started, fires immediately
- Relative start
  - First firing is start units after `start()` is called
- `void reschedule(HighResolutionTime time)`
  - Sets a new firing time for the initial firing

Exercise 3

Background for Exercise 4

- Synchronization & Priority-Inheritance

Semantics for Monitors

- Two issues addressed by the RTSJ with respect to monitors
- All queues are ordered by priority with FIFO when equal
  - Monitor entry queue
  - Wait-set notification queue
    - Order of notifications is now defined
- Monitor lock acquisition must avoid unbounded priority inversion
  - Priority inversion isn't addressed with any other form of synchronization e.g. `java.util.concurrent.locks.Lock`
  - RTSJ refers to the more general "execution eligibility inversion" but we'll stick with priority for simplicity
- What does the programmer have to do?
  - Be aware of how monitors are used
  - Be aware of what other forms of synchronization are used

Priority Inversion and its Avoidance

- When thread A tries to acquire a lock held by thread B, it blocks
- If A is high priority and B is low priority then having B execute when A would like to is a *priority-inversion*
- Worse, if a medium priority thread C preempts B, then A is blocked until C completes and then B releases the lock!
  - This is potentially an unbounded priority inversion
- Predictable execution time requires predictable locking time
  - And preferably a short as possible
- Priority inversion is not just concerned with locks – any means by which a thread A has to wait for an action in a thread B, is a potential priority inversion
- *Priority inversion avoidance*
  - Techniques to minimise the time that A has to wait for the lock

Priority Inheritance

- Priority Inheritance Protocol (PIP)
  - When A blocks acquiring the lock held by B then B is boosted to A's priority until it releases the lock
  - This prevents C from preempting B
  - Worst-case for A is the longest critical section for each lock it shares with other threads
- The *base priority* of a SO is the value in its scheduling parameters
- The *active priority* of a SO is the current priority
  - Which can be different to base due to priority inheritance
    - No means to query the active priority
- Even plain Java threads have an active priority
  - Java threads can execute at the highest real-time priority!
    - Avoid explicit lock sharing with Java threads

Exercise 4

Background for Exercise 5

- Wait-free Queues

Wait-Free Queues

- Primary goal is to allow non-blocking data exchange between no-heap SO's and heap-using SO's within a given memory area
  - If NHRT synchronizes with RTT then GC can preempt the NHRT
- `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`
  - A combined `WaitFreeReadQueue` and `WaitFreeWriteQueue`
- Simple API but construction can be complex and confusing
  - The internals of the queue can be created in a different memory area

Exercise 5

Real-Time Sessions at JavaOne

- TS-4807:
  - Easily Tuning Your Real-Time Application
- TS-5059:
  - Real Time: Understanding the Trade-Offs Between Determinism and Throughput
- TS-6735:
  - Building a Java™ Technology-Based Automation Controller: What, Why, How
- TS-6989:
  - Building Real-Time Systems for the Real World

Congratulations!

- You should now have completed this lab
- If you would like more time to continue working, please consider taking the lab exercises with you
- Discs containing all of the labs offered this year are available for you to take home
- To save your work, please copy it to a USB drive or email it to yourself
- The lab guide will tell you where to get help with this lab after JavaOne
- Thank you for attending this hands-on lab!

JavaOne℠

Thank You

David Holmes
Frederic Parain
Sun Microsystems