Overview

- RTSJ – why not
- Simple RT profile
- Scheduler implementation
- User defined scheduling
Real-Time Specification for Java

- Real-time extension definition
- Sun JSR - standard
- Still not completely finished
- Implementations
  - Timesys RI
  - Purdue OVM
RTSJ Issues

- Large and complex specification
  - Implementation
  - Verification
- Scoped memory cumbersome
- Expensive longs (64 bit) for time values
- J2SE library
  - Heap usage not documented
  - OS functions can cause blocking
RTSJ Subset

- Ravenscar Java
  - Name from Ravenscar Ada
  - Based in Puschner & Wellings paper
- Profile for high integrity applications
- RTSJ compatible
- No dynamic thread creation
- Only NHRTT
- Simplified scoped memory
- Implementation?
Real-Time Profile

- Hard real-time profile
  - See Puschner paper
- Easy to implement
- Low runtime overhead
- No RTSJ compatibility
Real-Time Profile

- Schedulable objects
  - Periodic activities
  - Asynchronous sporadic activities
    - Hardware interrupt or software event
    - Bound to a thread

- Application
  - Initialization
  - Mission
Application Structure

- Initialization phase
  - Fixed number of threads
  - Thread creation
  - Shared objects in *immortal* memory

- Mission
  - Runs forever
  - Communication via shared objects
  - Scoped memory for temporary data
Schedulable Objects

- Three types:
  - RtThread, HwEvent and SwEvent
- Fixed priority
- Period or minimum interarrival time
- Scoped memory per thread
- Dispatched after mission start

```java
class RtThread {
    public RtThread(int priority, int period) {
        ...
        public RtThread(int priority, int period,
                        int offset, Memory mem) {
            ...
        }
    }
    public void enterMemory() {
        ...
    }
    public void exitMemory() {
        ...
    }
    public void run() {
        ...
    }
    public boolean waitForNextPeriod() {
        ...
    }
    public static void startMission() {
        ...
    }
}

class HwEvent extends RtThread {
    public HwEvent(int priority, int minTime,
                   Memory mem, int number) {
        ...
    }
    public void handle() {
        ...
    }
}

class SwEvent extends RtThread {
    public SwEvent(int priority, int minTime,
                   Memory mem) {
        ...
    }
    public final void fire() {
        ...
    }
    public void handle() {
        ...
    }
}
```
Scheduling

- Fixed priority with strict monotonic order
- Priority ceiling emulation protocol
  - Top priority for unassigned objects
- Interrupts under scheduler control
  - Priority for device drivers
  - No additional blocking time
  - Integration in schedulability analysis
Memory

- No GC: Heap becomes immortal memory
- Scoped memory
  - Bound to one thread at creation
  - Constant allocation time
    - Cleared on creation and on exit
  - Simple enter/exit syntax
Restrictions of Java

- Only WCET analyzable language constructs
- No static class initializer
  - Use a static init() function
- No finalization
  - Objects in immortal memory live forever
  - Finalization complicates WCET analysis of exit from scoped memory
- No dynamic class loading
Implementation

- Scheduler for a Java real-time profile
  - Periodic and sporadic threads
  - Preemptive
  - Fixed priority
- Microcode
- Java
Low-level Functions

- Access to JVM internals
- Exposed as special bytecodes
- In Java declared as native methods
- Translation
- Avoids non-standard class files
Interrupts in JOP

- Translation of JVM bytecodes to microcode
- Interrupts are special bytecodes
- Inserted in the translation stage
- Call of JVM internal Java method
Dispatching

- Scheduler is a Java method
- Context of task is on the stack
- Exchange stack
- Set stack pointer
- Simple return

```java
private static int newSp;

public static void schedule() {
    Native.wr(0, IO_INT_ENA);
    RtThread th = tasks[active];
    th.sp = Native.getSP();
    Native.int2extMem(...);

    // find ready thread and
    // new timer value
    newSp = tasks[ready].sp;
    Native.int2extMem(...);

    Native.wr(tim, IO_TIMER);
    Native.setSP(newSp);
    Native.wr(1, IO_INT_ENA);
}
```
Implementation Results

- Scheduler and Dispatch in Java
- Only one function in microcode
- Test JVM in C
  - JOP compatible JVM
  - Implements timer with timestamp counter
  - Scheduling in Java
  - No OS needed, just a 32-bit C compiler
User-Defined Scheduler

- Java is a safe OO Language
  - No pointers
  - Type-safety
- No kernel user space distinction
- Hooks for scheduling
- Scheduler in Java extended to a framework
  - Class Scheduler
  - Class Task
Schedule Events

- Timer interrupt
- HW interrupt
- Monitor
- Thread blocking
- SW Event
Interrupts

- Hook for HW interrupts
- Timer interrupt results in scheduler call
- Access to timer interrupt
- Generate interrupt for blocking
- SW Event is not part of the framework
Synchronization

- Part of the language
- Each object can be a monitor
- JVM instruction or method declaration

```java
synchronized void foo() {
    ...
}

synchronized(o) {
    ...
}
```
Synchronization cont.

- Called by framework:
  - monitorEnter(Object o)
  - monitorExit(Object o)

- Attach user data to an object:
  - attachData(Object obj, Object data)
  - getAttachedData(Object obj)
Services for the Scheduler

- Dispatch
- Time
- Timer
- Interrupts
Class Scheduler

- Extended for a user-defined scheduler
- User provides:
  - schedule()
  - Monitor handling
- Framework supplies:
  - Software interrupt for blocking
  - Dispatch function
  - Current time
  - Timer interrupt
Class Task

- Minimal (not j.l.Thread)
- Provides list of tasks
- Scoped memory
- Usually extended
class Work extends Task {

    private int c;
    Work(int ch) {
        c = ch;
    }

    public void run() {

        for (;;) {
            Dbg.wr(c); // debug output
            int ts = Scheduler.getNow() + 3000;
            while (ts - Scheduler.getNow() > 0)
                ;
        }
    }
}
public class RoundRobin extends Scheduler {

    public void schedule() {
        Task t = getRunningTask().getNext();
        if (t == null) t = Task.getFirstTask();
        dispatch(t, getNow() + 10000);
    }

    public static void main(String[] args) {
        new Work('a');
        new Work('b');
        RoundRobin rr = new RoundRobin();
        rr.start();
    }
}
Summary

- RTSJ is too complex
- System code in Java is possible
- No extra memory protection needed
- Dispatch is 20% slower in framework
- Missing C++ friend in Java
- JopVm in C
Garbage Collection?

- An essential part of Java
- Without GC it is a different computing model
- RTSJ does not believe in real-time GC
- Real-time collectors evolve
- Active research area
  - For You?
Further Reading

