Real-Time Java

Ron K. Cytron and Christopher D. Gill
Washington University in Saint Louis
Department of Computer Science

Funded by
The National Science Foundation under grant 0081214
DARPA under contract F33615-00-C-1697

Thanks to
Angelo Corsaro, Morgan Deters, Nicholas Leidenfrost Washington University
David Sharp and Jim Urnes The Boeing Company

Center for Distributed Object Computing
Department of Computer Science
Washington University

June 2001
Tutorial Objectives

• Real-time programming issues
• Context for RT Java
  – Why is plain Java not good enough?
  – Why should real-time programmers use Java?
• RT Java
  – Specification
  – Use
  – Examples
• State of RT Java implementations
• Research issues for RT Java
Tutorial Outline

• Real-time applications
  – Cuckoo Clock

• Real-Time Specification for Java (RTSJ)
  – Guiding principles
  – Specification
    • Threads
    • Memory Model
    • Asynchronous Events
    • ATC
Example: Cuckoo Clock

• Synchronous events
  – Time annunciation (periodic)
  – Wake-up alarm (aperiodic)

• Asynchronous events
  – Reset to 12 noon
  – What time is it?
  – Pendulum adjustment
Digital Cuckoo Clock

Sweep-second hand ticks every second
Digital Cuckoo Clock
Digital Cuckoo Clock
Digital Cuckoo Clock
Digital Cuckoo Clock

Quail chirps 4 times at the hour
Digital Cuckoo Clock

Quail chirps 4 times at the hour
Then cuckoo sounds the hour
Digital Cuckoo Clock
Digital Cuckoo Clock

At quarter-past, quail chirps once
Digital Cuckoo Clock
Digital Cuckoo Clock

At half-past, quail chirps twice
Digital Cuckoo Clock

At half-past, quail chirps twice
Then cuckoo sounds once
Digital Cuckoo Clock
At quarter-till, quail chirps thrice
Digital Cuckoo Clock
Digital Cuckoo Clock
Digital Cuckoo Clock

Chirp
Chirp
Chirp
Chirp
Digital Cuckoo Clock

Cuckoo
Cuckoo
Cuckoo
Cuckoo
Digital Cuckoo Clock
Digital Cuckoo Clock

Push the red button to get most recent sequence

What time is it?
Digital Cuckoo Clock

Chirp
Chirp
Chirp
Chirp

What time is it?
Digital Cuckoo Clock

What time is it?
Digital Cuckoo Clock

Push the green button to reset to noon

What time is it?
Reset to 12 PM
Digital Cuckoo Clock

What time is it?

Reset to 12 PM
Java for Real Time

• Threads with priorities—for activities with
  – Different time scales
  – Different importance

• Synchronization
  – Safety
  – Sequencing

• Efficient exception mechanism
Threads and Real-Time

- Multi-threading is useful to decouple different activities
  - Active objects, request queues, synch/async
- However, work in different threads competes for CPU time and memory resources
- Must ensure resource usage by non-critical activities does not interfere with needs of critical activities
Java Threads

Thread seconds = new Thread() {
    public void run() {
        while (true) {
            waitOneSec();
            advanceSecondHand();
        }
    }
};

Thread alarm = new Thread() {
    public void run() {
        waitUntilAlarmtime();
        soundAlarm();
    }
};

seconds.start(); alarm.start();

• Subclass Thread to create a Runnable object
• Invoke start() → run() executes asynchronously
public void SecondHand()
{
    speaker.utter("tick");
}

public void soundAlarm()
{
    speaker.utter("Wach’ auf!!");
}

public synchronized void utter(String s) /* says s on the speaker */
{
Java Synchronization

synchronized (obj) {
    /* code block */
}

synchronized (obj) {
    /* code block */
}

- Alternatively, code outside an object can synchronize by obtaining a lock on that object

- The two blocks now have mutual exclusion
Synchronization and Real-Time

- Risk of unbounded priority inversions
  - Canonical high, low, middle scenario

- Priorities can uncover or exacerbate “bad” executions of existing race conditions
  - Horstmann & Cornell, *Core Java 2*

- Define lock priority semantics using
  - `setMonitorControl`

priority key: red high    middle    low
running outside block
blocked at guard
waiting (blocked) on a condition variable
running inside block

Real-Time Java Tutorial © 2000 Ron Cytron and Chris Gill
Design of the Cuckoo Clock

• Design perspectives
  – From the outside, we see three independent activities
  – Looking inside, we see one mechanism driving the whole system

• Design strategies
  – Bottom-up, liveness-based using \textit{thread adapters}
  – Top-down, safety-based using \textit{synchronization}

\textbf{Lea, Concurrent Programming in Java}
Cuckoo Clock Design from the “Outside”

• Independently acting entities
  – Sweep second-hand
  – Quail
  – Cuckoo

• Leads to three recurring tasks
  – Every second
    • Generate second-hand advance
  – Every 15 minutes
    • Generate quail action
  – Every 30 minutes
    • Generate cuckoo action
In Java, Each Task is a Thread

- Each thread executes independently
public SecondHand extends Thread {

    public void run() {

        while (true) {

            /* tick—advance the second hand */

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }

        }

    }

}
Problems?

```java
public SecondHand extends Thread {

    public void run() { 

        while (true) { 

            /* tick */
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {} 

        }
    }
}
```

- Accuracy
- Drift from missed deadline
Context for RTSJ

• Big investment already in Java
  – Application developers
  – Java implementers

• Needs of the RT community
  – Predictable execution
  – Time to market
Guiding Principles

- **Backward compatibility**

  **Pros**
  - If you know Java you know RT Java
  - Wider acceptance

  **Cons**
  - RT features introduced by new classes, methods, instead of clear syntax
  - Can write programs that require runtime tests
Guiding Principles

- Backward compatibility
- Appropriate for any Java environment

Pros
- J2ME for small platforms
- Enterprise edition for servers

Cons
- Can a large range of platforms be addressed by one standard?
Guiding Principles

Pros

- Backward compatibility
- Appropriate for any Java edition
- Support current practice and leading edge

Cons

- Nice idea
- Do they really do this?
Guiding Principles

- Backward compatibility
- Appropriate for any Java edition
- Support current practice and leading edge
- Predictable execution

Pros
- RTSJ’s “prime directive”
- Necessary for real-time applications

Cons
- Baby thrown out with bath water (no gc)
- Not the same as write-once run-anywhere
Guiding Principles

- Backward compatibility
- Appropriate for any Java edition
- Support current practice and leading edge
- Predictable execution
- Implementation details left vague

Pros
- Allows for invention

Cons
- Surprisingly underspecified for Java
- Write once carefully, run anywhere conditionally (Dave Hardin)
Guiding Principles

• Backward compatibility
• Appropriate for any Java edition
• Support current practice and leading edge
• Predictable execution
• Implementation details left vague

I expect the RTSJ to become the first real-time programming language to be commercially and technologically successful.

Doug Jensen, 2000
RTSJ Areas

- Storage Management
- Threads
- Time
- Interruptible I/O
- Scheduler
- Timers
- Asynchronous Transfer of Control
- Synchronization and Sharing
- Asynchronous Event Handler
RT Threading Issues

- Threads compete for CPU
- Some activities are more important than others
- Java thread priorities are a good start, but
  - Relationship to the garbage collector is uncertain
  - Priority inversion allowed by the specification
- Priority may be the wrong abstraction
  - Concept of time is fundamental
RT Threading Issues

- Threads compete for CPU
- Some activities are more important than others
- Java thread priorities are a good start, but
  - Relationship to the garbage collector is uncertain
  - Priority inversion allowed by the specification
- Priority may be the wrong abstraction
  - Concept of time is fundamental

```java
SecondHand
    seconds = new SecondHand();

Quail
    quail = new Quail();

Cuckoo
    cuckoo = new Cuckoo();

seconds.start();
quail.start();
cuckoo.start();
```
RT Scheduling Issues

- Priorities
  - Need sufficient *unique* priority levels

- Preemptive scheduling
  - Need well defined and appropriate semantics

- Fairness among threads is not *usually* a Real-Time concern (FIFO vs. RR)
  - But may be useful

- Feasibility
  - Admission control, certification/testing
Cuckoo Clock Design from the “Outside”

• Independently acting entities
  – Sweep second-hand
  – Quail
  – Cuckoo

• Leads to three recurring tasks
  – Every second
    • Generate second-hand advance
  – Every 15 minutes
    • Generate quail action
  – Every 30 minutes
    • Generate cuckoo action
In RTSJ, Each Task is a **RealtimeThread**

- Like a **Thread**, **RealtimeThread** executes independently
- Can associate real-time parameters
  - **SchedulingParameters** (for priority)
  - **ReleaseParameters** (for periodic execution)
Independent Realtime Threads

- How do we coordinate these threads?
  - Quail should sound before Cuckoo
  - Second Hand should advance before Quail
Independent RealtimeThreads

Second Hand

High Priority

Quail

Medium Priority

Cuckoo

Low Priority

• How do we coordinate these threads?
  • Quail should sound before Cuckoo
  • Second Hand should advance before Quail

• Use priorities to order the three threads’ events?
Independent Realtime Threads

- Won’t work!
  - Multiple processors would schedule threads concurrently
  - Preemption may take time
- Solution?
Independent RealtimeThreads

- Second Hand—release on the second
- Quail—release 10 ms after SecondHand
- Cuckoo—release 20 ms after SecondHand

Use PeriodicParameters
- SecondHand—release on the second
- Quail—release 10 ms after SecondHand
- Cuckoo—release 20 ms after SecondHand
Class **PeriodicParameters**

- Specify time properties
  - Start (relative or absolute)
Class **PeriodicParameters**

- Specify time properties
  - Start
  - Period (relative)
Class **PeriodicParameters**

- Specify time properties
  - **Start**
  - **Period**
  - **Cost** (relative)
Class PeriodicParameters

- Specify time properties
  - Start
  - Period
  - Cost
  - Deadline (relative)
Using **PeriodicParameters**

- **Second Hand**
- **Quail**
- **Cuckoo**

- 3:00:00
- 3:00:01
public SecondHand extends RealtimeThread {
    public SecondHand() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0), /* start */
                new RelativeTime(1000,0), /* period */
                new RelativeTime(5,0), /* cost */
                new RelativeTime(500,0), /* deadline */
                null, null /* handlers */
            )
        );
    }
}
Release Failures

- **Release parameters** advertise how threads are *projected* to behave
- However, differences between projected and actual behavior can lead to unexpected failures
- Need to be able to detect (and if possible handle) release failures
  - Cost overruns
  - Deadline misses
Class **PeriodicParameters**

- Specify time properties
  - **Start**
  - **Period**
  - **Cost**
  - **Deadline**

- Specify “oops” handlers
  - **OverrunHandler**
  - **MissHandler**
Specifying **PeriodicParameters**

```java
public SecondHand extends RealtimeThread{
    public SecondHand( AsyncEventHandler overrun, AsyncEventHandler miss) {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0),    /* start    */
                new RelativeTime(1000,0), /* period   */
                new RelativeTime(5,0),    /* cost     */
                new RelativeTime(500,0),  /* deadline */
                overrun, miss              /* handlers */
            )
        );
    }
}
```
Can Success Be Guaranteed?

Use **PeriodicParameters** and **PriorityScheduler**

- Each RealTimeThread is submitted to a Scheduler
- *Feasible* system $\rightarrow$ scheduling deadlines guaranteed
SecondHand  seconds  = new SecondHand();
Quail        quail    = new Quail();
Cuckoo       cuckoo  = new Cuckoo();

Scheduler boss   = Scheduler.getDefaultScheduler();

boss.addToFeasibility(seconds);
boss.addToFeasibility(quail);
boss.addToFeasibility(cuckoo);

if (!boss.isFeasible())
    throw new Error("Cannot accommodate threads");
Feasibility Testing

• Success is a function of
  – Tasks’ computational requirements
  – Scheduler policy
  – Platform speed

• Cannot get the same result everywhere

• What if the system is infeasible?
  – Give up
  – If running locally, apply more resources
  – Adapt
    • Lower the cost or frequency
    • Increase flexibility (**RationalTime** vs. **RelativeTime**)
Implementing Second Hand

```java
public SecondHand extends RealtimeThread {

    public SecondHand() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0),    /* start */
                new RelativeTime(1000,0), /* period */
                new RelativeTime(5,0),    /* cost */
                new RelativeTime(500,0),  /* deadline */
                null, null                /* handlers */
            )
        );
    }
}
```
Implementing Second Hand

```java
public SecondHand extends RealtimeThread {
    public SecondHand() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0),    /* start    */
                new RelativeTime(1000,0), /* period   */
                new RelativeTime(5,0),    /* cost     */
                new RelativeTime(500,0),  /* deadline */
                null, null                /* handlers */
            )
        );
    }

    public void run() {
        while (true) {
            /* tick */
            /* tick */
            waitForNextPeriod();
        }
    }
}
```
Synchronization

• How do we coordinate the output of the three threads?
  – Ordering is dictated by release times
  – What if clock is “quailing” when cuckoo wants to speak?
• RT Java provides a wait-free queue
• We adapt `WaitFreeDequeue` to make an `EventSequencer`
Design

- **EventSequencer**
  - Runs in its own thread
  - Thread-safe, built upon [WaitFreeDequeue](#)
  - Blocks until an event arrives
- Half-Sync / Half-Async pattern [Schmidt and Cranor, 1996]
public class EventSequencer extends RealtimeThread {
    private WaitFreeDequeue queue;

    public EventSequencer() {
        queue = new WaitFreeDequeue();
        start();
    }

    private Event dequeue() {
        return (Event) queue.blockingRead();
    }

    public void enqueue(Event e) {queue.nonBlockingWrite (e);}

    public void run() {
        while (true) {
            dequeue().doEvent();
        }
    }
}
Cuckoo Clock Design from the “Inside”

- One independently acting activity (pendulum)
- One component drives another
  - Pendulum drives second hand
  - Second hand drives quail (period=900)
  - Quail drives cuckoo (period=2)
- Leads to one recurring task
  - 10 ms pendulum
Design

- **Cog**
  - Connects to another Cog
  - Turns period times then turns the connected cog once
Design

- **Cog**
  - Connects to another **Cog**
  - Turns period times then turns the connected cog once
Design

- **Cog**
  - Connects to another **Cog**
  - Turns period times then turns the connected cog once
Design

- **Cog**
  - Connects to another Cog
  - Turns period times then turns the connected cog once
Design

- **Cog**
  - Connects to another Cog
  - Turns period times then turns the connected cog once
Design

- **Cog**
  - Connects to another Cog
  - Turns period times then turns the connected cog once
Design

• **Cog**
  – Connects to another **Cog**
  – Turns period times then turns the connected cog once
Design

- **ClockAction**—abstract
  - Represents something connected to a Cog.
  - Can perform an action when the cog moves
  - Uses the `EventSequencer` to sequence output
public class SecondHand extends ClockAction {
    public SecondHand (EventSequencer seq) {
        super (seq);
    }

    public void perform () {
        appendEvent (new Event (this) {
            public void doEvent () {
                System.out.println("Tick");
            }
        });
    }
}
public class Quail extends ClockAction {
    int quarterHours = 0;
    public Quail (EventSequencer seq) {super (seq);}

    public void perform () {
            appendEvent (new Event (this) {
                public void doEvent () {
                    for (int i=0; i<quarterHours+1; ++i)
                        System.out.print ("Chirp! ");
                    System.out.print(‘\n’);
                    quarterHours = (quarterHours + 1) % 4;
                }
            });
    }
}
RTSJ Areas

- Storage Management
- Threads
- Time
- Interruptible I/O
- Scheduler
- Timers
- Asynchronous Transfer of Control
- Synchronization and Sharing
- Asynchronous Event Handler
Time, Timers, and Handlers

• Precision issue
• Kinds of time
  – Absolute
  – Relative
  – Rational
• Can associate timers with handlers
  – AsyncEventHandler (AEH) invocation
  – Another way to engineer periodicity
• Handlers can also be used other ways
  – Cost overrun and deadline miss handlers
  – One-shot “what time is it” button
  – One-shot “reset to noon” button
Threads vs. Handlers in Design

- Threads can be made periodic, but then...
  - More like co-routines
  - One-shot buttons: how?
- Can unify temporally coupled actions in one thread (e.g., pendulum model)
  - Can handle one-shot buttons
  - But, no automatic feasibility test (must provide separately)
- Async Event Handlers
  - Intermediate design alternative
  - Offers looser coupling of objects
  - Supports RTSJ bells & whistles (e.g., scheduler feasibility test)
Now Each Object Extends `AsyncEventHandler` (which *also* implements `Schedulable`)

- Async Event Handlers (AEHs) are also dispatched independently
- AEHs can associate real-time parameters, too
  - `SchedulingParameters` (for priority)
  - `ReleaseParameters` (for periodic execution)
Second Hand (Using Threads)

public SecondHand extends RealtimeThread {
    public SecondHand() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0), /* start */
                new RelativeTime(1000,0), /* period */
                new RelativeTime(5,0), /* cost */
                new RelativeTime(500,0), /* deadline */
                null, null /* handlers */
            )
        );
    }
    public void run() {
        while (true) {
            /* tick */
            waitForNextPeriod();
        }
    }
}
public SecondHand extends AsyncEventHandler {
    public SecondHand() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0),    /* start */
                new RelativeTime(1000,0), /* period */
                new RelativeTime(5,0),    /* cost */
                new RelativeTime(500,0),  /* deadline */
                null, null                /* handlers */
            )
        );
    }

    public void handleAsyncEvent () {
        // while (true) {
        /* tick */
        //     waitForNextPeriod();
        // }
    }
}

public Quail extends AsyncEventHandler {
    public Quail() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(10, 0), /* start  */
                new RelativeTime(900000, 0), /* period */
                new RelativeTime(50, 0), /* cost  */
                new RelativeTime(500, 0), /* deadline */
                null, null /* handlers */
            )
        );
    }
    public void handleAsyncEvent() {
        /* Chirp */
    }
}
Cuckoo (Using Handlers)

```java
public Cuckoo extends AsyncEventHandler {
    public Cuckoo() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(20, 0), /* start */
                new RelativeTime(1800000, 0), /* period */
                new RelativeTime(100, 0), /* cost */
                new RelativeTime(500, 0), /* deadline */
                null, null /* handlers */
            )
        );
    }

    public void handleAsyncEvent() {
        /* Cuckoo */
    }
}
```
Associating Handlers with Timers

```java
public class CuckooClock {
    private PeriodicTimer littleTimer;
    private PeriodicTimer bigTimer;
    private Quail littleBird;
    private Cuckoo bigBird;

    public void init (HighResolutionTime startTime) {
        // . . . Start second hand
        PeriodicParameters birdParams =
            (PeriodicParameters)
                littleBird.getReleaseParameters ();

        littleTimer =
            new PeriodicTimer ( startTime.add(new RelativeTime(10,0)),
                                birdParams.getPeriod (),
                                littleBird);

        // ...
    }
}
```
Handling External Events

• Want to add two more capabilities to our clock
  – “what time is it now?”
  – “reset to noon”

• We’ll re-use AEH approach
  – Associate AEHs with events
  – Bind events to button pushes

• Differences between buttons
  – Query simply reads current time
  – However, reset changes state asynchronously
  – We’ll extend our solution using ATC to address the latter issue
Registering External Events

```
AsyncEvent

addHandler(…)

bindTo(String)
```
Responding to External Events

```
AsyncEvent

fire()
schedule()
```

© 2000 Ron Cytron and Chris Gill
Registering External Events

```java
public class CuckooClock {
    // ...
    private AsyncEvent currentTimeRequest;
    private AsyncEvent resetRequest;
    private CurrentTime currentTimeHandler;
    private ResetTime resetHandler;

    public void init (HighResolutionTime startTime) {
        // ... SecondHand, Quail, Cuckoo setup
        currentTimeRequest.addHandler (currentTimeHandler);
        currentTimeRequest.bindTo ("currentTimeButtonPush");
        resetRequest.addHandler (resetHandler);
        resetRequest.bindTo ("resetButtonPush");
        // ...
    }
}
```
RT Issues: Asynch Transfer of Control (ATC)

- Asynchronous
  - Quail chirps, advances
  - Reset quail’s counter
- Ensure quail is reset in a controlled way
  - Mid-chirp is rude
  - Mid-advance is wrong
- ATC refines stop and suspend semantics
  - Quail throws **AIE** when it can be interrupted
  - ResetHandler calls

```
button push
```

```
ResetHandler
```

```
Quail
```

```
reset
timer
```
ATC: Inter-Thread Exceptions
(AsynchronouslyInterruptedException)

- Special ATC semantics
  - When safe to throw must be declared (default: unsafe)
  - Deferred throw and propagation behavior (‘exception tunneling’)
  - Nesting and replacement of exceptions governed by scope
Java Storage Management

- **Automatic storage allocation**
- **Pointer references are type-checked (safe)**
  - Mostly at compile-time
  - Run-time for narrowing casts
- **Garbage collection**
  - Automatic
  - Conceptually precise
  - Guaranteed correct

```java
ClockAction part = new Quail();
(Quail) part).chirp();
part = new SecondHand();
```
Real-Time Storage Management Issues

nuclearReactor.on();

new A();

nuclearReactor.off();

• Predictable execution required
• Allocation time for Java
  • Average case
  • Worst case
• What happens if there is insufficient free storage to satisfy the request?
RT Java Storage Management

- We’ll examine each topic in turn
- RTSJ tries to balance the ideals of Java with the needs of RT applications

- Automatic storage allocation
- Pointer references are type-checked (safe)
- Garbage collection
RT Java Storage Management

• Automatic storage allocation
• Pointer references are type-checked (safe)
• Garbage collection
Storage Allocation—Free List

• Linked list of free blocks

• Search for desired fit

• Worst case $O(n)$ for $n$ blocks in the list
Solution: Bounded Allocators

- RT Java introduces the notion of a `MemoryArea`
Solution: Bounded Allocators

- RT Java introduces the notion of a `MemoryArea`
- The traditional allocator is represented by the `HeapMemory` class
Solution: Bounded Allocators

- RT Java introduces the notion of a `MemoryArea`
- The traditional allocator is represented by the `HeapMemory` class
- Other allocators are now possible, including
  - `VTMemory`—variable time taken for allocation; this is what Java already gives us—no guarantees
Solution: Bounded Allocators

- RT Java introduces the notion of a MemoryArea
- The traditional allocator is represented by the HeapMemory class
- Other allocators are now possible, including
  - VTMemory—variable time taken for allocation; this is what Java already gives us—no guarantees
  - LTMemory—linear in the size of the allocated object (used to be CTMemory)
LTMemory

- Linear in the size of the request
  - Free list sorted by size
LTMemory

• Linear in the size of the request
  – Free list sorted by size
  – Sorted inversely by size
LTMemory

- Linear in the size of the request
  - Free list sorted by size
  - Sorted inversely by size
- Cost of maintaining structure must be included
LTMemory

- Linear in the size of the request
  - Free list sorted by size
  - Sorted inversely by size
- Cost of maintaining structure must be included
- Random doesn’t work
- Knuth’s Buddy System does
Knuth’s Buddy System

- Free-list segregated by size (radix sort)
  - All requests rounded up to a power of 2
Knuth’s Buddy System (1)

- Begin with one large block
- Suppose we want a block of size 16
Knuth’s Buddy System (2)

- Begin with one large block

- Recursively subdivide
Knuth’s Buddy System (3)

- Begin with one large block

  - Recursively subdivide

  256
  128
  64
  32
  16
  8
  4
  2
  1
Knuth’s Buddy System (4)

- Begin with one large block
  - 256
  - 128
  - 64
  - 32

- Recursively subdivide
  - 16
  - 8
  - 4
  - 2
  - 1
Knuth’s Buddy System (5)

- Begin with one large block
  - 256
  - 128
  - 64
  - 32
  - 16
  - 8
  - 4
  - 2
  - 1
- Yield 2 blocks size 16
Knuth’s Buddy System (6)

- Begin with one large block
- Yield: 2 blocks size 16
- One of those blocks can be given to the program

<table>
<thead>
<tr>
<th>Size</th>
<th>256</th>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>
Worst-case free-list behavior

• The longer the free-list, the more pronounced the effect
• No a priori bound on how much worse the list-based scheme could get
• Average performance similar

<table>
<thead>
<tr>
<th>Number of objects allocated</th>
<th>Speedup of Buddy over List</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Worst</td>
</tr>
<tr>
<td>180</td>
<td>3.15</td>
</tr>
<tr>
<td>3000</td>
<td>72.84</td>
</tr>
</tbody>
</table>
Spec Benchmark Results

Speedup of Buddy over List

- compress
- jess
- raytrace
- db
- javac
- mpegaudio
- mtrt
- jack
RT Java Storage Management

- Java enables good garbage collection
- Unfortunately, current implementations do not embrace best practices
- As a result, RTSJ is skeptical of real-time garbage collection
  - Mandate to allow for cutting-edge research
  - No hooks currently for RT garbage collection
- Automatic storage allocation
- Pointer references are type-checked (safe)
- Garbage collection

Real-Time Java Tutorial © 2000 Ron Cytron and Chris Gill
### Storage Management Linked to Threads

<table>
<thead>
<tr>
<th></th>
<th>RT Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Guarantees</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

---

Real-Time Java Tutorial © 2000 Ron Cytron and Chris Gill
## Storage Management Linked to Threads

<table>
<thead>
<tr>
<th>RT Specifications</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thread</td>
<td>RealtimeThread</td>
</tr>
<tr>
<td>Guarantees</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>No</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Yes</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
## Storage Management Linked to Threads

<table>
<thead>
<tr>
<th>Guarantees</th>
<th>RT Specifications</th>
<th>Thread</th>
<th>RealtimeThread</th>
<th>NoHeapRealtimeThread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>RealtimeThread</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoHeapRealtimeThread</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RealtimeThread offers no guarantees

- Realtime threads get higher priority than the garbage collector
- RealtimeThreads can instantiate and reference the heap.
RealtimeThread offers no guarantees

- Realtime threads get higher priority than the garbage collector
- RealtimeThreads can instantiate and reference the heap.
  - Suppose a gc cycle is necessary to satisfy an allocation request
  - How much time will that take?
RealtimeThread offers no guarantees

- Realtime threads get higher priority than the garbage collector
- RealtimeThreads can instantiate and reference the heap.
  - Suppose a gc cycle is necessary to satisfy an allocation request
  - How much time will that take?
- **NoHeapRealtimeThread** avoids this
NoHeapRealtimeThread Avoids the Heap

- NHRT threads
  - Cannot allocate in heap
  - Cannot hold references to the heap
  - Cannot even manipulate references to the heap
- Non-collected memory
  - ImmortalMemory
  - ScopedMemory
    - VTMemory
    - LTMemory
Using **ImmortalMemory** temporarily

/* Suppose thread currently uses the heap */

MemoryArea undead = ImmortalMemory.instance();

SecondHand s =
   (SecondHand)
      undead.newInstance(
         Class.forName(“SecondHand”)
      );    /* immortal */

SecondHand t = new SecondHand();    /* heap */
Using **ImmortalMemory** by Default

```java
public SecondHand extends NoHeapRealtimeThread {
    public SecondHand() {
        super(null, ImmortalMemory.instance());
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0),     /* start    */
                new RelativeTime(1000,0),  /* period   */
                new RelativeTime(5,0),     /* cost     */
                new RelativeTime(500,0),   /* deadline */
                null, null                  /* handlers */
            )
        );
    }
}
```
Using **ScopedMemory**

```java
public SecondHand extends NoHeapRealtimeThread {
    public SecondHand() {
        super(null, new ScopedMemory(4096));  /* 4K */
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0),   /* start  */
                new RelativeTime(1000,0), /* period */
                new RelativeTime(5,0),    /* cost   */
                new RelativeTime(500,0),  /* deadline*/
                null, null                 /* handlers */
            )
        );
    }
}
```
public void enter(Runnable logic) throws RuntimeException {
    synchronized (referenceCountLock) {
        referenceCount++;
    }
    super.enter(logic);
    synchronized (referenceCountLock) {
        referenceCount--;
        resetIfRequired();
    }
}

• Any RealtimeThread can call enter() on a ScopedMemory
The Scope of **ScopedMemory**

```java
public void enter(Runnable logic) throws RuntimeException {
    synchronized (referenceCountLock) {
        referenceCount++;
    }
    super.enter(logic);
    synchronized (referenceCountLock) {
        referenceCount--;
        resetIfRequired();
    }

    oldMemoryArea = currentThread.getMemoryArea();
    this.activate();
    try {
        logic.run();
    } finally {
        oldMemoryArea.activate()
    }
}
```

- **Any RealtimeThread** can call **enter()** on a **ScopedMemory**
- **The enter()** method will call the thread back via its **run()** method.
public void enter(Runnable logic) throws RuntimeException {
    synchronized (referenceCountLock) {
        referenceCount++;
    }
    super.enter(logic);
    synchronized (referenceCountLock) {
        referenceCount--;
        resetIfRequired();
    }
}

- Any RealtimeThread can call enter() on a ScopedMemory
- The enter() method will call the thread back via its run() method.
- The scope is reference counted
- When the reference count reaches 0, the scope can be deleted
Entering a Scope

```java
thread.run()
area.enter()
run() {
}  
area
thread
```
How Scoped Memories Nest

• Each **ScopedMemory** is logically separate

[Diagram with boxes labeled 'Scope A' and 'Scope B']
How Scoped Memories Nest

- Each ScopedMemory is logically separate
- A RealtimeThread can enter any scope
How Scoped Memories Nest

- Each **ScopedMemory** is logically separate
- A **RealtimeThread** can enter any scope
- The reference count of each entered scope is bumped
How Scoped Memories Nest

- Each **ScopedMemory** is logically separate
- A **RealtimeThread** can enter any scope
- The reference count of each entered scope is bumped
How Scoped Memories Nest

- Each **ScopedMemory** is logically separate
- A **RealtimeThread** can enter any scope
- Scope B cannot be deallocated until the thread exits its `run()` method
- This occurs when the scope’s reference count becomes 0
How Scoped Memories Nest

- Each `ScopedMemory` is logically separate
- A `RealtimeThread` can enter any scope
- Scope B cannot be deallocated until the thread exits its run() method
- The thread must exit its run() method again for Scope A to be freed
- In a sense, Scope B was nested inside Scope A
How Scoped Memories Nest

- One ScopedMemory can be viewed as contained in another if its set of active threads is a proper subset of the other’s
However....
RT Java Storage Management

- Automatic storage allocation
- Pointer references are type-checked (safe)
- Garbage collection
Rules for scope references

14. Strict assignment rules placed on assignments to or from memory areas prevent the creation of dangling pointers, and thus maintain the pointer safety of Java. The restrictions are listed in the following table:

<table>
<thead>
<tr>
<th>Reference to Heap</th>
<th>Reference to Immortal</th>
<th>Reference to Scoped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Immortal</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Scoped</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Local Variable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Rules for scope references

14. Strict assignment rules placed on assignments to or from memory areas prevent the creation of dangling pointers and then maintain the integrity of code. 

**Thou shalt not reference any object whose lifetime could be shorter than thine own.**

<table>
<thead>
<tr>
<th>Scoped</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes, if same, outer, or shared scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Variable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, if same, outer, or shared scope</td>
</tr>
</tbody>
</table>
Why this rule?

- The thread could enter Scope B
Why this rule?

- The purple thread can enter Scope B
- The other threads may have deposited references in Scope B to objects in Scope A
Why this rule?

• The purple thread can enter Scope B
• The other threads may have deposited references in Scope B to objects in Scope A
• If those threads exit, then Scope A can go away
• References in the purple thread to Scope A are invalid and will throw an exception
Ensuring Safe Scope Accesses

• When a thread is about to enter a scope, the programmer can
Ensuring Safe Scope Accesses

• When a thread is about to enter a scope, the programmer can cause the thread first to enter the ancestors’ scopes
Ensuring Safe Scope Accesses

• When a thread is about to enter a scope, the programmer can cause the thread first to enter the ancestors’ scopes

• This guarantees safe access in the thread
Ensuring Safe Scope Accesses

• When a thread is about to enter a scope, the programmer can cause the thread first to enter the ancestors’ scopes

• This guarantees safe access in the thread

• Notion of
  • Kami
  • Baptism-by-Proxy
Status of an Implementation

- **J9 implementation from IBM**
  - Mostly there, but some pieces are missing
  - Newer and more complete systems on the way
- **Can develop using any JDE**
  - Uses `javax.realtime.*` package
- **Must run on an RTSJ-aware JVM**
  - Native calls to accomplish scheduling, storage, etc.
  - Mostly there, but some pieces are missing
  - New and better systems on the way
  - Requires OS support for RT features
How well does the JVM do RT?

• **Real-time determinism test cases**
  – RealtimeThread preemption handling
  – Priority inversion avoidance
  – Dispatching of AsyncEvents
  – Jitter in periodic event handling
  – Timing of periodic event handling
PreemptTest Scenario

• **Purpose:**
  – Measure whether priority preemption occurs correctly for multiple RealtimeThreads of different priorities

• **Method:**
  – Stagger the start of fixed-duration, processor-holding RealtimeThreads of increasing or decreasing priority. Using timestamp logging, see when threads enter and exit in relation to each other
PreemptTest Results

Starting at priority 7, start the threads every 2 seconds in *decreasing* priority order. Threads try to keep the processor for 7 seconds.

Result: Priority 7 rightfully kept the processor until done. Priorities 6—4 then ran concurrently until completion, and then threads 3—1 did the same. Priorities were not respected.
PreemptTest Analysis

• Problem
  – The J9 implementation maps multiple Java priority levels to the same underlying RTOS thread level
  – RTSJ Requires at least 28 unique priority levels

<table>
<thead>
<tr>
<th>Java Thread Priority</th>
<th>QNX Neutrino Thread Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>28, 29, 30</td>
<td>15r</td>
</tr>
<tr>
<td>25, 26, 27</td>
<td>14r</td>
</tr>
<tr>
<td>22, 23, 24</td>
<td>13r</td>
</tr>
<tr>
<td>19, 20, 21</td>
<td>12r</td>
</tr>
<tr>
<td>16, 17, 18</td>
<td>11r</td>
</tr>
<tr>
<td>13, 14, 15</td>
<td>10r</td>
</tr>
<tr>
<td>10, 11, 12</td>
<td>9r</td>
</tr>
<tr>
<td>7, 8, 9</td>
<td>8r</td>
</tr>
<tr>
<td>4, 5, 6</td>
<td>7r</td>
</tr>
<tr>
<td>1, 2, 3</td>
<td>6r</td>
</tr>
</tbody>
</table>
PrilInvertTest Scenario

• Purpose:
  – Measure whether priority inversion is properly avoided.

• Method:
  1. A low-priority thread obtains a lock
  2. A medium-priority thread consumes CPU time
  3. A high-priority thread needs the lock
PriInvertTest Results

Result: The low-priority thread did NOT get elevated to high priority. Therefore, the medium-priority thread finished before the high-priority thread. Priority inversion occurred.
PrilInvertTest Analysis

• Problem
  – Priority Inheritance does not currently work
  – RTSJ specifies Priority Inheritance as the default priority inversion avoidance method for synchronized blocks
EventDispatchTest Scenario

• Purpose:
  – Measure the execution order for multiple AsyncEventHandlers of different priorities, when an AsyncEvent fires for which they are all registered.

• Method:
EventDispatchTest Results

The Java priorities are such that the resulting QNX priorities are different

Result: This is correct. The highest priority handler runs first, then middle one, and then lowest.
EventJitterTest Scenario

• Purpose:
  – Measure the variation between runs of a PeriodicTimer-driven AsyncEventHandler

• Method:
  – A PeriodicTimer fires an AsyncEventHandler at a fixed rate while lower-priority threads consume CPU time
EventJitterTest Results

In this 1 second test, our AsyncEventHandler runs at priority 30. Another RealtimeThread runs at priority 6. PeriodicTimer event fires every 50 msecs (20 Hz.).

Result: Quite good - jitter within RTOS timer resolution.
EventJitterTest Results (cont)

Same as before, but with 200 background threads at priority 6

Result: Not bad - some jitter (+/- 1.1 msec) between runs, but lower-priority threads do seem to affect jitter.
EventJitterTest Results (cont)

One background thread at priority 10; AsyncEventHandler threads still at priority 30

```
In the main: args = 6
In the main: mode = RT
In the main: handlerpri = 30
In the main: handlerper = 50
In the main: numThreads = 1
In the main: otherThreadpri = 10
Collecting: mode = RT approx. duration_secs = 1
In the main, leaving.
In the main, log results.
```

Result: Bad - the periodic events never get to execute, even though the handler has higher priority than the background thread
EventJitterTest Analysis

PeriodicTimer’s fire() method called

“Spawner” thread
QNX pri=8r

Handler thread
Java pri=30
QNX pri=15r

Another thread
Java pri = 10
QNX pri = 9r

handler thread of desired priority
spawned for each firing

Problem!
If this thread is using the processor,
the “Spawner” thread will not get
the opportunity to spawn the handler
thread. Priority inversion occurs.
Summary Of Experimental Results

• Java thread priority preemption must be maintained for each of the 28 required thread levels.

• AsyncEventHandler threads driven by PeriodicTimers must not be prevented from running by lower priority threads.

• Priority inheritance should work by default.

• Programmer needs better control of Java thread to underlying RTOS mapping.
Review: Java, RTSJ, Real-Time Issues

• Threads (Java, revisited in RTSJ)
• Release characteristics & failures
• Scheduling
• Synchronization (Java, revisited in RTSJ)
• Time and timers
• Asynchronous event handling
• Memory management
• Asynchronous transfer of control
Concluding Remarks

• The RTSJ extends Java to address real-time
  – Scheduling, Storage Management, Asynchrony
• The RTSJ largely stays within the existing programming model
  – Some new idioms to master, but much is preserved
  – ATC in particular illustrates the trade-offs
• Stay tuned, more evolution is on the horizon
  – Reference implementations and benchmarking
  – New specification efforts, e.g., the DRTSJ (JSR 50)
• We wish we had
  – Hooks for user-defined gc and scheduling
Effectiveness of Real-Time GC on Spec (white portion is uncollected)

size 10 Absolute
You’ve seen the movie, now read the book