

SYSC 5701 Operating System Methods for Real-Time Applications

Real-Time Java

Winter 2014

Real-Time Java <http://www.rtj.org/>

terminology
in slides

- relevant parts of “traditional” Java = Java
 - threads, monitors, memory access, garbage collection
- real-time Java solution = RTJava
 - “battle” between 2 proposals
- RTJava Specification = RTJS
 - the “winner” → 2000
- industrial status? – Mars rover?

Mar 20/14

2

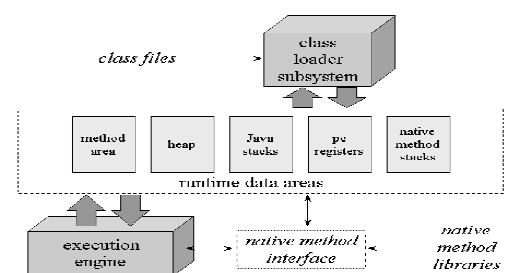
Warning!

- The following slides have Java details that were relevant at the time of the conception of RTJava (circa 2000)
- Java’s concurrency support made some improvements in Java 5 (circa 2004)
 - e.g. semaphores

Mar 20/14

3

Java Virtual Machine



Mar 20/14

4

Java

- **concurrency** → threads
- **synchronization** → monitors
- Java language spec → vague in many spots
 - under-specified → allows many possible implementations
- OK for soft real-time
 - not appropriate for hard real-time

Mar 20/14

5

Java Threads

- priority driven scheduler
 - unknown scheduling algorithm
 - may be time sliced
- unknown number of priorities
- allows mapping to a wide variety of native threading models (Windows, Unix, etc.)
- in general, not safe to explicitly transfer control from one thread to another
 - killing threads, asynchronous control transfer

under-specified!

Mar 20/14

6

Thread LifeCycle

- **new Thread** → an **empty Thread** object
- no system resources allocated to it
- in this state, can only **start** the thread,
- **start**:
 - creates system resources needed to run the thread
 - schedules the thread to run
 - calls the thread's **run** method
- return from **start** → thread is Runnable

thread
state

Mar 20/14

7

to Not Runnable

- A thread becomes **Not Runnable** when:
 - its **sleep** method is invoked
 - thread calls **wait** → wait for a specific **monitors!** condition to be satisfied
 - thread is blocked on I/O

Mar 20/14

8

back to Runnable

- if thread is sleeping and the specified number of milliseconds elapse
- if thread is waiting for a condition and another object notifies the thread of a change in condition
 - call to **notify** or **notifyAll** **monitors!**
- if thread is blocked on I/O and the I/O completes
- **stop** a thread: **run** method terminates

Mar 20/14

9

Thread Example (Sun Doc's)

```
public class SimpleThread extends Thread {  
    public SimpleThread(String str) {  
        super(str);  
    }  
    public void run( ) {  
        /* ... thread's execution code goes here ... */  
    }  
}
```

constructor

super class constructor

this example:
subclasses Thread and overrides run

Mar 20/14

10

Creating Instances of Threads

```
public class ThreadsDemo {  
    public static void main (String[] args) {  
        new SimpleThread("AThread").start();  
        new SimpleThread("BThread").start();  
    }  
}
```

Mar 20/14

11

Inheritance Problem

- how can a class **X** be extended with thread behaviour if it does not already have threads?
- Java does not support multiple class inheritance
 - e.g. inherit from X and Thread
- add runnable interface to class **X**

Mar 20/14

12

Runnable Example ☹️

```
public class ThreadedX extends X implements Runnable {
    private Thread MyXThread = null;
    public void start() { if (MyXThread == null) {
        MyXThread = new Thread(this, "MyXThread");
        MyXThread.start(); }
    }
    public void run() {
        Thread myThread = Thread.currentThread();
        while (MyXThread == myThread) {
            /* ... do stuff approx every second ... */
            try { Thread.sleep(1000); }
            (InterruptedException e)
            { /*stop sleep, get back to work */ } }
    }
    public void stop() {
        // stop MyXThread
        MyXThread = null;
    }
}
```

this example: implements runnable

what happens to "Thread" ?

"asynchronous
stop" ? ☹️ ?

Thread Synchronization

- **synchronized** → **monitor** well ... sort of
- only one thread at a time can be executing a synchronized block in an object
 - managed by runtime environment
- not really a monitor ?
 - managing other threads in monitor?

Mar 20/14

14

Synchronized Methods

```
public class SharedQ {
    public synchronized int get() { ... }
    public synchronized void put(int value) { ... }
}
```

- SharedQ object locked automatically during **get** & **put** calls
- prevents interference
- what about **put** when full (**get** when empty)?

Mar 20/14

15

Wait / Notify Conditions

- **wait** – allows thread to block self
- **notify** – signals one thread that is waiting on object
 - But choice of thread is arbitrary !?!
- **notifyAll** – signals all threads that are waiting on object

Mar 20/14

16

SharedQ Revisited

```
public class SharedQ {
    ... synchronized ... as before
    boolean SpaceAvailable = true;
    boolean DataAvailable = false;

    public synchronized int get() {
        while ( DataAvailable == false ) {
            try {
                // wait for data
                wait();
            } catch (InterruptedException e) { ... }
        }
        here ... get value from queue and adjust state variables
        notifyAll(); // let others in!
        return value;
    }
}
```

why not just
"notify" ?

release **all**? busy
wait? ugly!?

not really
monitor!?

Mar 20/14

17

revisit on slide 44

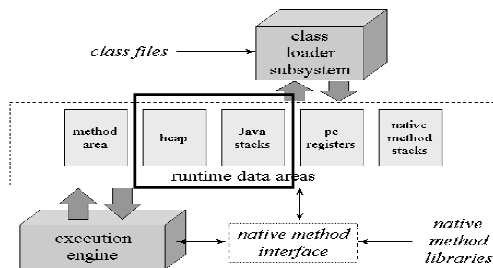
InterruptedException

- thrown when a thread is waiting, sleeping, or "otherwise paused for a long time" (?) and another thread interrupts it using the **interrupt** method in class **Thread**
- maybe this could be used to implement thread management in "monitors" ?!
 - backdoor? using "exception" mechanism for planned "normal" operation ☹️

Mar 20/14

18

Memory Issues



Mar 20/14

19

Lifetime of Variables

- Java allows variables to be:
 - Globally available for the life of the program
 - Local to functions: exist for the life of a function call
 - Local to objects: exist for the life of an object

Mar 20/14

20

Java Memory Access

- no “physical memory” access by address
 - no pointers !!!
- abstract references only
- **objects** created in heap
 - create objects using **new**
 - cannot explicitly delete!
 - automatic garbage collection (GC)
 - reclaim object when no longer in use

Mar 20/14

21

Java Stack

- each thread has its own “Java stack”
- Java stack **frames** for local data needs:
 - local variables, arguments, return values, intermediate calculations
 - unavailable to Java programmers
 - managed by virtual machine

Mar 20/14

22

Garbage Collection

- run by virtual machine when free heap space goes below some “low” level
- all details are managed by virtual machine
- garbage collection thread is often non-preemptible by other threads
- a significant problem for real-time deadlines!

overhead vs. program robustness

Mar 20/14

23

Garbage Collection Implementation

- find at least one reference to object
- if no references, object “dead” → collect object
- possible sources of references:
 - from variable in stack frame
 - from a static variable
 - from a field in a live object
 - from a virtual machine internal variable
- search for references? → no time guarantees!

Mar 20/14

24

Real-Time Java Battle

the armies:

1. backward compatible with Java
 - minimize expansion of language
 - extend existing Java classes
2. new language
 - target real-time systems
 - don't require backward compatibility

taste's great!

less filling!

Mar 20/14

25

The Winner: backward compatible

- RTJ Experts Group
- started work ~ 1998
- RTJ Specification v1.0, 2000
- November 2001, reference implementation
 - TimeSys

Mar 20/14

26

Guiding Approach

- general applicability
- backward compatibility with Java
- **Write Once** Carefully, **Run Anywhere** Conditionally (WOCRAC ?) **WORA ?**
- reflect current practice
- predictable execution
- no syntactic extensions
- allow implementation variation/customization
 - documented!

Mar 20/14

27

Key Advances

- thread scheduling and dispatching
- synchronization
- memory management
- asynchronous actions
- time, clocks & timers

Mar 20/14

28

Priority

- stricter notion of priority
 - (original) Java threads – “low” priority
 - garbage collector – dividing line
 - higher than garbage collector – real-time!
- priority inversion control!
 - default = priority inheritance
- 28 unique priorities

Mar 20/14

29

Realtime Threads

- **extend** Thread → RealtimeThread
 - priority overlaps with Java threads, garbage collector & “higher”
 - can be run higher than GC!
 - can access new memory types (later)
 - can also allocate in heap ☹
 - potential priority inversion with GC!

Mar 20/14

30

No Heap Realtime Threads

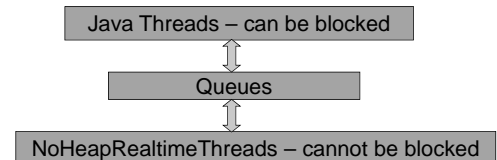
- **extend** RealtimeThread → NoHeapRealtimeThread
 - priority always higher than GC
 - cannot access heap or references to heap
 - never priority inversion with GC! 😊

Mar 20/14

31

Can NoHeapRealtime Threads Interact with Java Thread?

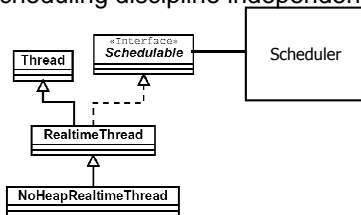
- cannot allow Java thread to block NoHeapRealtimeThread !
 - unbounded priority inversion!
- wait-free queuing to pass messages



32

Scheduling

- introduce:
 - Schedulable entities: RealTimeThreads
 - Scheduler – manages Schedulable objects
 - API is scheduling discipline independent



Mar 20/14

33

Scheduler

- methods for:
 - feasibility analysis
 - admission control
 - dispatching
 - asynchronous event handling
- **extend** Scheduler → PriorityScheduler
 - customize – override methods above, e.g.:
 - RMAScheduler extends Scheduler
 - EDFScheduler extends Scheduler

Mar 20/14

34

Schedulable Interface

- objects that **implement** **Schedulable** are scheduled by Scheduler
- RealTimeThreads and AsyncEventHandlers implement Schedulable
- Schedulable object includes reference to Scheduler to be used

Mar 20/14

35

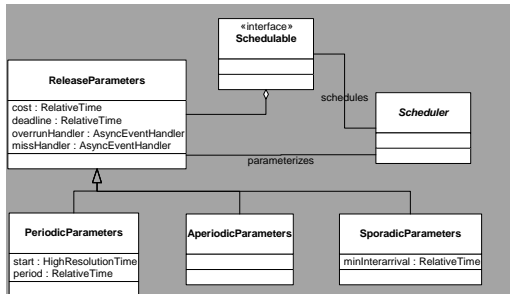
Scheduling Parameters

- priority & importance
- periodic, aperiodic, sporadic
- memory demands

Mar 20/14

36

Scheduling



Mar 20/14

37

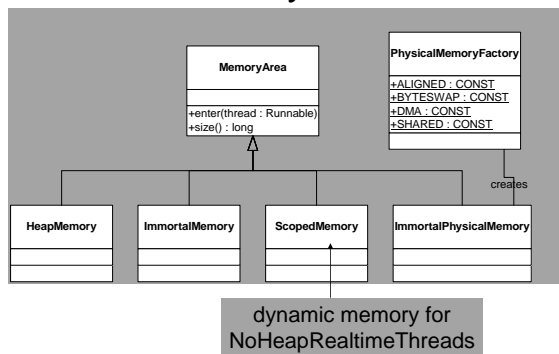
Memory Management

- object lifetime control:
 - manual → under explicit program control
 - automatic → visibility (scope)
- memory areas
 - not managed by garbage collector !
 - immortal → persistent for life of application
 - scoped → life of run()

Mar 20/14

38

Memory Areas



Mar 20/14

39

Immortal Memory Example

```

import javax.realtime.*;

/** Example of the use of "Immortal" memory in a periodic
    processing context. No heap allocation, avoids garbage
    collection overhead! */

/** Class that performs processing in Immortal memory */
class Runner implements Runnable {
    public void run( ) {
        // Processing code goes here
    }
}
    
```

Mar 20/14

40

```

public static void main(String[] Args) {
    NoHeapRealtimeThread t = null;

    // Set up periodic processing
    PeriodicParameters timeParams = new PeriodicParameters();
    // 1 msec computation
    timeParams.cost = new RelativeTime(1, 0);
    // 10 msec period
    timeParams.period = new RelativeTime(10, 0);

    // Set up immortal memory; size given in RealtimeSystem
    MemoryParameters memParams = new
        MemoryParameters(ImmortalMemory.instance());
    // Processing is encapsulated in a Runnable object
    Runner r = new Runner();
}
    
```

Mar 20/14

41

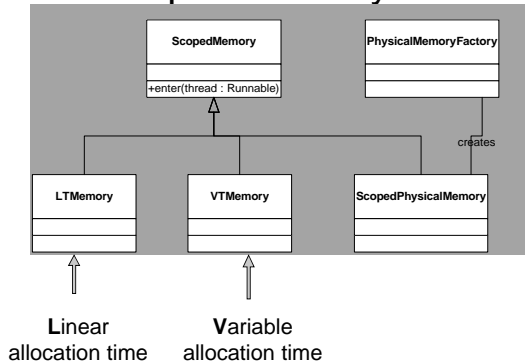
```

/* Create a NoHeapRealtimeThread with Periodic
    scheduling parameters and ImmortalMemory memory
    parameters. */
try {
    t = new NoHeapRealtimeThread( timeParams,
                                   memParams, r );
} catch (AdmissionControlException e) { }
// Start processing
t.start( );
} // end of main
    
```

Mar 20/14

42

Scoped Memory Area



Mar 20/14

43

Asynchronous Event Handling

- **AsyncEvent** objects
 - instance represents an event that can happen
- **AsyncEventHandler** implements **Schedulable**
 - logic to process AsyncEvent
 - execute with semantics of threads ← as if it was a thread
- handlers bound to events:


```
AsyncEvent.addHandler( AsyncEventHandler a );
```
- may be bound to external events, or invoked internally:


```
AsyncEvent.fire( )
```

Mar 20/14

44

Asynchronous Transfer of Control

- "throws" clause including **AsynchronouslyInterruptedException (A/E)**
 - exception raised by the JVM when the **interrupt()** method for thread is called
- mechanism extends the current semantics of the **interrupt()** method from only certain blocking calls to straight-line code
- can be used to fudge "killing" a thread ☹

recall slide 18

Mar 20/14

45

AsyncEvents & Interrupts Example

```

import javax.realtime.*;
/** Example of using Asynchronous Event/Event Handling facility
    to provide an interface to hardware events, i.e. interrupts. A
    hardware interrupt conceptually fires an AsyncEvent, which
    causes the associated handler to run. */
public class HardwareEventExample extends AsyncEvent {
    private int interruptNum;

    /** Construct a new Hardware Event for a given interrupt. */
    public HardwareEventExample(int num) { interruptNum = num; }

    /** Bind a handler to the interrupt. */
    public void setHandler(AsyncEventHandler h) {
        super.setHandler(h);
        Hardware.bind(interruptNum, h);
    }
}
    
```

define event

Mar 20/14

46

Interrupt Example (con't)

```

class HardwareEventHandler extends AsyncEventHandler {
    private int interruptCount = 0;
    /** Interrupt handler method. */
    public void handleAsyncEvent() {
        interruptCount++;
        // Driver code
    }
}
    
```

define handler

bind this handler to interrupt using
previous setHandler method

Mar 20/14

47

Industry Status?

- TimeSys has commercial RTJS compliant compiler product – built over RTLinux
 - Mars Rover
 - Team Jefferson in DAPRA Grand Challenge 2007
- still missing many of the higher-level real-time language features Halang & Stoyenko would like to see!
- long-term ?
- Distributed RTJ movement already in progress!

Mar 20/14

48