

Java for Real-Time

Part 1 – General Introduction

Christos Kloukinas

School of Informatics
City University London

Copyright ©2012 Christos Kloukinas  School of Informatics 1/44



Copyright ©2012 Christos Kloukinas  School of Informatics 3/44

Why OO is Interesting/Difficult

- Focuses on designing APIs.
- API design is difficult – appropriate abstractions.
- Watch the Google Tech Talk: “How To Design A Good API and Why it Matters” by Joshua Bloch
 - Video: <http://bit.ly/X0Fmw>
(video.google.com/videoplay?docid=-3733345136856180693)
 - Slides: <http://bit.ly/UdcO>
(lcsd05.cs.tamu.edu/slides/keynote.pdf)

Copyright ©2012 Christos Kloukinas  School of Informatics 5/44

Contents

Object-Oriented
Concurrency Basics

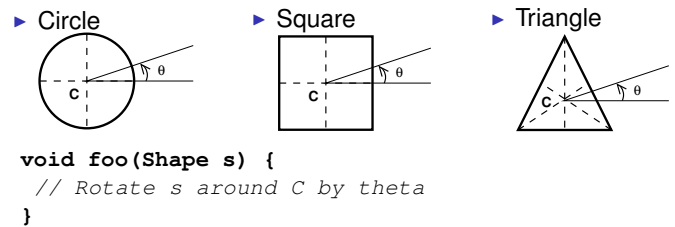
Threads
Monitors
Events

Conclusions

What we've covered
What we haven't covered
Study Material

Copyright ©2012 Christos Kloukinas  School of Informatics 2/44

Rotating by θ around C



Copyright ©2012 Christos Kloukinas  School of Informatics 4/44

OO Inheritance in Non-OO Languages

```
void fubar(shape &s) {  
  
    // CASE A:  
    if (circle_p(s)){  
        // do circle  
    } else if (square_p(s)){  
        // do square  
    } else if (triangle_p(s)){  
        // do triangle  
    } ...  
  
    }  
  
    // CASE B:  
    switch (s.shape_type){  
    case 0: // do circle  
        break;  
    case 1: // do square  
        break;  
    case 2: // do triangle  
        break;  
    ...  
}
```

If your code looks like that, then you need sub-classes or extra virtual methods...

Copyright ©2012 Christos Kloukinas  School of Informatics 6/44

Skip translations list

Hello World!

How the way people code “Hello World” varies depending on their age and job:

High School/Jr. High

```
10 PRINT "HELLO WORLD"  
20 END
```

First year in College

```
program Hello(input, output)  
begin  
  writeln('Hello World')  
end.
```

Senior year in College

```
(defun hello  
  (print  
    (cons 'Hello (list 'World))))
```

New professional

```
#include <stdio.h>  
  
void main(void)  
{  
  char *message[] = {"Hello ", "World"};  
  int i;  
  for(i = 0; i < 2; ++i)  
    printf("%s", message[i]);  
  printf("\n");  
}
```

Seasoned professional

```
#include <iostream.h>  
#include <string.h>  
class string  
{  
private:  
  int size;  
  char *ptr;  
public:  
  string() : size(0), ptr(new char('\0')) {}  
  string(const string &s) : size(s.size)  
  {  
    ptr = new char[size + 1];  
    strcpy(ptr, s.ptr);  
  }  
}
```

```

    }
    ~string()
    {
        delete [] ptr;
    }
    friend ostream &operator <<(ostream &, const string &);
    string &operator=(const char *);
};

ostream &operator<<(ostream &stream, const string &s)
{
    return(stream << s.ptr);
}
string &string::operator=(const char *chrs)
{
    if (this != &chrs)
    {
        delete [] ptr;
        size = strlen(chrs);
        ptr = new char[size + 1];
        strcpy(ptr, chrs);
    }
    return(*this);
}
int main()
{
    string str;
    str = "Hello World";
    cout << str << endl;
    return(0);
}

```

System Administrator

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    char *tmp;
    int i=0;
    /* on y va bourin */
    tmp=(char *)malloc(1024*sizeof(char));
    while (tmp[i]="Hello Wolrd"[i++]);
    /* Ooops y'a une infusion ! */
    i=(int)tmp[8];
    tmp[8]=tmp[9];
    tmp[9]=(char)i;
    printf("%s\n",tmp);
}

```

Apprentice Hacker

```

#!/usr/local/bin/perl
$msg="Hello, world.\n";
if ($#ARGV >= 0) {
    while(defined($arg=shift(@ARGV))) {
        $outfilename = $arg;
        open(FILE, ">" . $outfilename) || die "Can't write $arg: $!\n";
        print (FILE $msg);
        close(FILE) || die "Can't close $arg: $!\n";
    }
} else {
    print ($msg);
}

```

```
1;
```

Experienced Hacker

```
#include <stdio.h>
#include <string.h>
#define S "Hello, World\n"
main(){exit(printf(S) == strlen(S) ? 0 : 1);}
```

Seasoned Hacker

```
% cc -o a.out ~/src/misc/hw/hw.c
% a.out
Hello, world.
```

Guru Hacker

```
% cat
Hello, world.
```

New Manager (do you remember?)

```
10 PRINT "HELLO WORLD"
20 END
```

Middle Manager

```
mail -s "Hello, world." bob@b12
Bob, could you please write me a program that prints "Hello, world."?
I need it by tomorrow.
^D
```

Senior Manager

```
% zmail jim
I need a "Hello, world." program by this afternoon.
```

Chief Executive

```
% letter
letter: Command not found.
% mail
To: ^X ^F ^C
% help mail
help: Command not found.
% damn!
!: Event unrecognized
% logout
```

Research Scientist

```
PROGRAM HELLO  
PRINT *, 'Hello World'  
END
```

Older research Scientist

```
WRITE (6, 100)  
100 FORMAT (1H ,11HELLO WORLD)  
CALL EXIT  
END
```

Other humor in the GNU Humor Collection.

The joke on this page was obtained from the FSF's email archives of the GNU Project.

The Free Software Foundation claims no copyright on this joke.

Something That Runs


```
interface Runnable {  
    void run();  
}
```

- ▶ Method `run` is to a thread what `main` is to a program
- ▶ No arguments
- ▶ No return value
- ▶ No worries

Copyright ©2012 Christos Kloukinas  School of Informatics 7/44

Hello World! – Extending Thread

```
public class MyThread1 extends Thread {  
    public void run() {  
        System.err.println("Hello World!");  
    }  
  
    static public void main(String args[]) {  
        Thread runner = new MyThread1();  
        runner.start();  
    }  
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 9/44

Hello World! – Showing Off...

```
public class MyThread3 {  
    static public void main(String args[]) {  
        Thread runner = new Thread(  
            new Runnable() {  
                public void run() {  
                    System.err.println("Hello World!");  
                }  
            }  
        );  
        runner.start();  
    }  
}
```

Q: What's the highlighted stuff?!?

A: An Anonymous Inner Java Class.

Copyright ©2012 Christos Kloukinas  School of Informatics 11/44

Who Runs It?

```
public class Thread implements Runnable {  
    Thread();  
    Thread(Runnable target);  
  
    void run(); // RUN ME! ME! ME! ME!!!  
    void start(); // WHEN THIS IS CALLED!!!  
  
    ...  
}
```

- ▶ Don't run `run` directly – it'll be executed sequentially.

Copyright ©2012 Christos Kloukinas  School of Informatics 8/44

Hello World! – Implementing Runnable

```
public class MyThread2 implements Runnable {  
    public void run() {  
        System.err.println("Hello World!");  
    }  
  
    static public void main(String args[]) {  
        MyThread2 target = new MyThread2();  
        Thread runner = new Thread(target);  
        runner.start();  
    }  
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 10/44

Anonymous Inner Classes

- ▶ Just like normal classes but without a name
- ▶ They implement some interface
- ▶ Their code is provided directly to the `new` operator
- ▶ Why this madness?
 - ▶ Not (just) to show off...
- ▶ Consider Map/Reduce: We get a set, map each element to some value and then reduce the values to a single result.
 - ▶ $\sum_n x_n^2$
Map `int square(int)`
Reduce `int sum(List<int>)`

Copyright ©2012 Christos Kloukinas  School of Informatics 12/44

Anonymous Classes for $\Sigma_n x_n^2$ – Part I

```
import java.util.*;
interface Map { int mapIt(int datum); };
interface Reduce {
    int reduceIt(Map mapper, List<Integer> set); };

public class MapReduce {
    static public void main(String args[]) {
        List<Integer> theData = new ArrayList<Integer>();
        for (int i=0; i < 10; ++i)
            theData.add(new Integer(i));

        Map squareIt = new Map() {
            public int mapIt(int x) { return x * x; } };

        Reduce result = new Reduce() {
            public int reduceIt(Map mapper,
                                List<Integer> set) {
                int r = 0;
                Iterator<Integer> itr = set.iterator();
                while (itr.hasNext())
                    r += mapper.mapIt(itr.next());
                return r; }
            .reduceIt(squareIt, theData);

        System.err.println("Result is: " + result);
    }
};
```

Copyright ©2012 Christos Kloukinas  School of Informatics 13/44

Or In Fewer Words...

- Lisp: Created in 1958 – still unbeatable

```
(REDUCE #' + ; Reduce:  $\Sigma$ 
  (MAPcar #' (lambda (x) (* x x)) ; Map:  $x^2$ 
    '(0 1 2 3 4 5 6 7 8 9))) ; -> 285.
```

“car” – the head of a list; “cdr” – the tail of a list

- Anonymous classes are anonymous/lambda/λ functions
- Learn Lisp to improve your:
 - Java
 - C++ STL
 - Boost C++
 - C++11

- Ain't kidding – learn Lisp.

Seibel “Practical Common Lisp”

<http://www.gigamonkeys.com/book/>

Yes, we'll use *them*...

Copyright ©2012 Christos Kloukinas  School of Informatics 15/44

Closures

- We have `add(a, b)` and we've got the special case `inc(a) = a + 1`

Q How can we obtain adders that add some number N?

A Closures

```
(defun make-adder (n)
  #'(lambda (x) (+ x n))) ; returns a function where
                           ; n is bound to some value

(let ((add3 (make-adder 3))

      (add7 (make-adder 7)))

  (list (funcall add3 5)
        (funcall add7 5)
        (funcall add3 8))) ; -> (8 12 11)
```

Copyright ©2012 Christos Kloukinas  School of Informatics 17/44

Anonymous Classes for $\Sigma_n x_n^2$ – Part II

```
int result = new Reduce() {
    public int reduceIt(Map mapper,
                        List<Integer> set) {
        int r = 0;
        Iterator<Integer> itr = set.iterator();
        while (itr.hasNext())
            r += mapper.mapIt(itr.next());
        return r; }
    .reduceIt(squareIt, theData);

    System.err.println("Result is: " + result);
}
};
```

► Result is: 285

Copyright ©2012 Christos Kloukinas  School of Informatics 14/44

Lisp in One Slide

```
(defun my-sum (a b)
  (let ((result (+ a b)))
    (- result 3))) ; take my cut...
```

```
(defun my-abs (x)
  (cond ((< 0 x) 1)
        ((> 0 x) -1)
        (t 0)))
```

- But no syntax – everything's a list! No more parsing!!!
- No stupid XML `<my-sum> 42 13 </my-sum>`
- Instead: `(my-sum 42 13) ; -> 52`
- Less to type & it runs!
- Can use it with Java (abcl, clojure)
- Will definitely become better programmers
- There's more in Lisp...

Copyright ©2012 Christos Kloukinas  School of Informatics 16/44

Closures – We Need Them Badly


- It's how the memory allocation in RTSJ is controlled
- So how's it done in Java? (Anonymous) Inner classes...

```
interface Add { int doIt( int i ); };

public class MakeAdder1 {
    static public Add makeAdder( final int j ) {
        return new Add() {
            public int doIt (int i) {
                return i+j;
            }
        };
    }

    public static void main(String args[]) {
        Add add3 = MakeAdder1.makeAdder(3);
        Add add7 = MakeAdder1.makeAdder(7);

        System.err.println( add3.doIt(5) + " " + add7.doIt(5)
                            + " " + add3.doIt(8) );
    }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 18/44

Shared Closures

```
; shared environments are possible too
(let ((scrt "none"))
  (defun bar ()
    scrt)

  (defun foo (m)
    (setq scrt m))
)

scrt ; causes an error "Var scrt has no value"
; scrt no longer exists - it's out of scope. But:
(format t "~A~%" (bar)) ; prints "none"

(foo "listen very carefully, I will say it only once")

(format t "~A~%" (bar)) ; prints "listen very ..."

;; Now you even know how to print in Lisp!   :-)
```

Copyright ©2012 Christos Kloukinas  School of Informatics 19/44

Exaggerating a Bit

- ▶ RTSJ uses closures so as to:
 1. Set the memory before executing some code,
 2. Execute the code, and
 3. Then reset the memory to what it was before.
- ▶ It tries to emulate "Block allocation"


```
{ int i = 0;
  { int j = i + 1;
    { int n = i + j;
      } /* no n */
    } /* no j */
  } /* no i */
```
- ▶ The memory setting/resetting is done automatically.
- ▶ All you need to do is provide the code that should be executed in that closure.
- ▶ So your closures will read a bit like inline code.
- ▶ It's just that it takes a bit of getting used to it. So:
- ▶ Now, back to Threads!!!



Copyright ©2012 Christos Kloukinas  School of Informatics 21/44

Translating FSP models to Threads

```
public class CountDown implements Runnable {
  Thread counter; int i; final static int N = 10;

  public void begin() {          // begin ->
    counter = new Thread(this);
    i = N; counter.start();      // CD[N]
  }
  public void end() {           // end ->
    counter = null;
  }
  public void run() {           // CD[i]
    while(true) {              // Recursion
      if (null == counter) return; // STOP (after "end")
      if (0<i) { tick(); -i;}    // when(0<i) tick->CD[i-1]
      if (0==i) { beep(); return;} // when(0==i) beep->STOP
    }
  }                               // STOP if run() returns
}
```

bar & foo in Java?

Modelling a countdown timer in FSP

```
COUNTDOWN ( N=3 ) = ( begin -> CD[N] ),
  CD[i : 0..N] =
    ( when (0<i) tick -> CD[i-1]
      | when (0==i) beep -> STOP
      | end -> STOP
    ).
```

- ▶ A simple Labelled Transition System (finite too)
- ▶ Some actions are controlled "externally" (begin, end)
 - ▶ **Each gets its own public method**
- ▶ Some others are controlled "internally" (tick, beep)
 - ▶ **Each gets its own private method or block of statements**

Copyright ©2012 Christos Kloukinas  School of Informatics 22/44

The Thread Class API – Part I

```
public class Thread implements Runnable {
  Thread();          Thread(Runnable target);

  static Thread      currentThread();
  int                getPriority();
  void               setPriority(int newPriority);
  static int         MIN_PRIORITY; // NORM_/MAX_PRIORITY
                                     // just a hint... :-/

  boolean            isActive();
  // ! (NEW || TERMINATED)
  // = (RUNNABLE || BLOCKED || WAITING || TIMED_WAITING)

  void               run(); // RUN ME! ME! ME! ME!!!
  void               start(); // WHEN THIS IS CALLED!!!

  boolean            isDaemon();
  void               setDaemon(boolean on); //BEFORE start!

  ...
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 24/44

Daemon Threads

- ▶ An application runs until all its threads are no longer alive
- ▶ Some threads are used for auxiliary functions only, not really doing “real” work
- ▶ If the real worker threads finish the other ones will keep the application running
- ▶ Solution: We make auxiliary threads daemons
- ▶ Once all non-daemon threads are no longer alive, the application finishes execution

Copyright ©2012 Christos Kloukinas  School of Informatics 25/44

The Thread Class API – Part II


```
...
static void    yield(); // Make way for others - maybe..

static void    sleep(long millis);
static void    sleep(long millis, int nanos);

void          join();
void          join(long millis);

void          interrupt();
static boolean interrupted(); // DANGER!!!
boolean       isInterrupted();
...
}
```

- ▶ **static** means the current thread.
- ▶ **interrupted** – a very bad name (*API fail...*)

Copyright ©2012 Christos Kloukinas  School of Informatics 27/44

Sleeping Problems

- ▶ **sleep(millis)** causes a thread to sleep for **AT LEAST** so many milli-seconds.
 - ▶ The timer used may not be as accurate
 - ▶ Other threads may be running when this one is supposed to wake up (so further delays)
- ▶ The sleep duration is relative – cannot give an absolute time point “sleep till Christmas Eve 2025”
- ▶ Imagine we try to compute the corresponding relative time:

```
Calendar XmasEve2025 = ...;
long relative = XmasEve2025.getTimeInMillis()
               - System.currentTimeMillis();
// OK, so we now sleep for "relative".
sleep(relative);
```

- ▶ Why doesn't this work?

Copyright ©2012 Christos Kloukinas  School of Informatics 29/44

Be Careful When Running the Same Runnable

```
public class RunnableBis {
    static public void main(String args[]) {
        Runnable target = new Runnable() {
            int c = 0;
            public void run() {
                String me = Thread.currentThread().getName();
                int i = 0;
                for (i = 0; i < 2; ++i) {
                    c=c+1; System.err.println(me+"_c_is:"+c);
                }
            }
        };
        Thread runner1 = new Thread(target, "R1");
        Thread runner2 = new Thread(target, "R2");
        runner1.start(); //Possible results // R1 c is: 2
        runner2.start(); // R1 c is: 3
                        // R2 c is: 2
    }
}; // State is SHARED - PROTECT IT! // R2 c is: 4
```

Copyright ©2012 Christos Kloukinas  School of Informatics 26/44

Yielding the CPU

May not stop the thread

- ▶ If there's no other thread...
- ▶ If there are multiple cores...
- ▶ If the JVM's scheduler chooses to ignore it...

It is just a hint to the JVM scheduler

Copyright ©2012 Christos Kloukinas  School of Informatics 28/44

Interrupting

- ▶ In the Countdown timer we implemented the “end” signal by setting the **current** reference to **null**
- ▶ The thread had to check that reference from time to time to decide if it should stop or continue
- ▶ Another option would have been to send it an asynchronous interrupt signal

```
public void end() { counter.interrupt(); }
```
- ▶ What happens then?

Copyright ©2012 Christos Kloukinas  School of Informatics 30/44

Interrupting – Possible Outcomes

- ▶ If no appropriate security permissions we get a **SecurityException**!
 - ▶ **counter** that we wanted to interrupt, never is.
- ▶ If **counter** is blocked on sleep/wait/join then it receives an **InterruptedException**
 - ▶ So **sleep/wait/join** need to be inside a **try/catch** block (well wait for sure).
- ▶ If it's running, a flag is set in that thread by the JVM.
 - ▶ Nothing else happens until (if ever!) the thread tries to **sleep/wait/join** – then it gets an **InterruptedException**.
 - ▶ If it's not planning to do so, it can check from time to time if its flag is set:
`if (counter.isInterrupted()) return;`
- ▶ API fail – **interrupted()** returns a Boolean but it also clears the flag.
 - ▶ Only use it to clear your thread's flag, not to test for interruptions.

Copyright ©2012 Christos Kloukinas  School of Informatics 31/44

Modelling Shared State – A Car Park

```
CARPARKCONTROL(N=1)= SPACES[N],
SPACES[i:0..N]= ( when(i>0) arrive->SPACES[i-1]
                  | when(i<N) depart->SPACES[i+1]) .

ARRIVALS    = (arrive->ARRIVALS) .
DEPARTURES  = (depart->DEPARTURES) .

|| CARPARK =
  (ARRIVALS | | CARPARKCONTROL(4) | | DEPARTURES) .
```

From Magee & Kramer.

Copyright ©2012 Christos Kloukinas  School of Informatics 33/44

CarParkControl in Java

```
class CarParkControl {
    protected int spaces, capacity;
    CarParkControl(int n)
    {capacity = spaces = n;}

    synchronized void arrive()
    throws InterruptedException {
        while (spaces==0) wait();
        --spaces; notify();
    }

    synchronized void depart()
    throws InterruptedException {
        while (spaces==capacity) wait();
        ++spaces; notifyAll();
    }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 35/44

Dealing with an InterruptedException

- ▶ Don't catch it – declare it among the exceptions thrown by your method
- ▶ Catch it, clean up, then rethrow it

```
catch (InterruptedException e) { // ... clean up
    throw e; }
```
- ▶ If you cannot rethrow it, then reset the flag by interrupting yourself (the thrown exception has cleared it)

```
catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
```

For more: Goetz's article <http://ibm.co/LWMNxQ>

Copyright ©2012 Christos Kloukinas  School of Informatics 32/44

Translating FSP Models to Monitors – The Pattern

- ▶ **ARRIVE** is a “active” process
 - ▶ It gets translated into a Java **Thread**
- ▶ **CARPARKCONTROL** is a “passive” process
 - ▶ It gets translated into a Java **Monitor**

Copyright ©2012 Christos Kloukinas  School of Informatics 34/44

General Pattern

```
// FSP: when COND act -> NEW_STATE

// Java:
public synchronized void act()
    throws InterruptedException {
    while (! COND) wait();
    // modify monitor data
    notifyAll();
}
```

- ▶ Method is **synchronized**
- ▶ Repeated testing of condition with a **while**
We have no idea whether it holds when **wait()** returns
- ▶ We notify all waiting threads if we have modified the object in any way – maybe their waiting conditions are satisfied now.

Copyright ©2012 Christos Kloukinas  School of Informatics 36/44

Too many vs too little

- ▶ `notifyAll` may wake up too many threads.
- ▶ `notify` may resume the wrong thread.
- ▶ Design carefully
- ▶ Use `java.util.concurrent.locks`'s `ReentrantLock` and its `Condition` to declare different conditions per lock.
That way, you only **signal** those interested in a respective condition (notifier must know the conditions the others are **await**-ing on!).
- ▶ Dibble: *Avoid sharing in the first place!*

Copyright ©2012 Christos Kloukinas  School of Informatics 37/44

Lock & Condition – Pattern

```
lock.lock();
try {
    // Code to execute with the lock
} finally {
    lock.unlock();
}
```

Can you write a (Java) closure so that you only need to write the code part and not the lock/try/finally/unlock one?

Copyright ©2012 Christos Kloukinas  School of Informatics 39/44

Critical Region Length

- ▶ We need to keep critical regions short to minimise delayed event handlers
- ▶ Event handlers usually come in pairs – short and long
- ▶ Sending data over the network
- ▶ The short IH buffers messages and groups them into longer messages that are eventually sent by the long IH

Copyright ©2012 Christos Kloukinas  School of Informatics 41/44

Lock & Condition

```
public class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

See the pattern Copyright ©2012 Christos Kloukinas  School of Informatics 38/44

Events

- ▶ For the Car Park Control, arrivals and departures are events it needs to react to
- ▶ Each arrives at an unknown time and sometimes even the arrival rate is unknown
- ▶ Each is treated **asynchronously**
- ▶ Who treats it? Who runs the code of the **arrive** and **depart** methods?
- ▶ In our example, that's the thread that called them
- ▶ Concurrent calls to **arrive** are serialised because of the methods being **synchronized** – they **wait**
- ▶ What if the events cannot wait?
Phone rings while you're busy – is it going to wait until you're not busy?

Copyright ©2012 Christos Kloukinas  School of Informatics 40/44

What we've covered

- ▶ OO
 - ▶ Don't do it yourself – **Delegate!**
 - ▶ Watch out for inheritance done "by hand"...
 - ▶ It's difficult – designing APIs
- ▶ Threads, Monitors
 - ▶ Directly translatable from automata.
 - ▶ Model first, avoid direct coding!
 - ▶ Difficult to get a concurrent system right
 - ▶ Need to model and analyse it first
 - ▶ This is even more true for R-T systems
- ▶ Events
 - ▶ Asynchronous computation
 - ▶ Short and Long Interrupt Handlers

Copyright ©2012 Christos Kloukinas  School of Informatics 42/44

What we haven't covered

- ▶ Thread joining
 - ▶ Thread Groups
 - ▶ Thread Private Data
- Bloch's talk discusses their API (re-)design

Study Material


- ▶ Bloch's talk
 - ▶ Video: <http://bit.ly/X0Fmw>
(video.google.com/videoplay?docid=-3733345136856180693)
 - ▶ Slides: <http://bit.ly/UdcO>
(lcsd05.cs.tamu.edu/slides/keynote.pdf)
- ▶ Magee & Kramer "Concurrency: State Models and Java Programs"
<http://www.doc.ic.ac.uk/~jnm/book>
Try the online applets.
Get their LTSA tool as well – it includes all the models of the book and allows you to simulate and verify them.
- ▶ Wellings "Concurrent and Real-Time Programming in Java"
- ▶ Dibble "Real-Time Java Platform Programming"

Java for Real-Time

Part 2 – Real-Time, RTSJ Principles

Christos Kloukinas

School of Informatics
City University London

Copyright ©2012 Christos Kloukinas  School of Informatics 1/34

Deadlines

- ▶ The simplest way to identify if a system is R-T or not:
- ▶ Do tasks have *deadlines*?
- ▶ In R-T systems, individual response times matter in deciding whether they work or not
- ▶ Editor is slow
 - ▶ Annoying
- ▶ ABS (anti-lock breaking system) is slow
 - ▶ It doesn't work

Copyright ©2012 Christos Kloukinas  School of Informatics 3/34

High Throughput?

- ▶ R-T \neq High Throughput
- ▶ H-T means that we can treat a high number of requests per time unit
- ▶ That is, on *average* we treat each request fast and/or can treat requests concurrently
- ▶ But there may be requests that get treated in time (much) longer than the average
- ▶ In a R-T system, these violate their deadlines

High Throughput \neq Fast

- ▶ One couple can produce a baby in nine months
- ▶ Cannot speed this up
- ▶ We can increase the throughput by having more couples
- ▶ Nine couples \rightarrow throughput = one baby per month

Copyright ©2012 Christos Kloukinas  School of Informatics 5/34

Contents

Real-Time System Concepts

- What it is
- What it isn't
- Characteristics
- Types
- Constraints
- Black Arts – Controlling The Forces of Darkness... R-T Scheduling

RTSJ

- Some Java Issues for R-T
- NIST R-T Java Requirements
- RTSJ Guiding Principles
- RTSJ 7 Enhanced Areas

Conclusions

- What we've covered
- What we haven't covered
- Study Materials

Fast?

- ▶ R-T \neq Fast!
- \rightarrow An R-T system may be quite slow
 - ▶ A 10 year loan is a R-T Task – it fails if not repaid on time
- \leftarrow A fast system may not always meet its deadlines
 - ▶ Fast means low *average* response time

Copyright ©2012 Christos Kloukinas  School of Informatics 4/34

Three Main Characteristics of a Good R-T System

1. Predictability
 2. Predictability
 3. Predictability
- ▶ A R-T system needs to be predictable in its behaviour
 - ▶ Deterministic behaviour makes our job easier
 - ▶ Deterministic behaviour means that the program controls its behaviour
 - ▶ It'll behave as predicted, no matter what the environment does

Copyright ©2012 Christos Kloukinas  School of Informatics 6/34

Predictability: How to Achieve it

We need

- ▶ To have some idea of the type(s) of tasks
 - ▶ How often they are required to execute
 - ▶ How long they take to execute in the worst case (WCET)
 - ▶ What is the deadline for completing each ($D \neq \text{WCET}$)
 - ▶ What are their dependencies (shared resources, other tasks – CPU)
- ▶ Have some idea of the computation platform
 - ▶ Number of shared resources
 - ▶ Characteristics of these resources (size, access time, ...)
 - ▶ Number of processing units (processors, cores, ...)
- ▶ Have some idea of our design for meeting the deadlines given the resources available
 - ▶ We really need to *design & analyse* it – cannot hack it
- ▶ Know what it means to miss a deadline

Copyright ©2012 Christos Kloukinas  School of Informatics 7/34

R-T and Embedded Systems

R-T systems are often embedded – part of a larger system

- ▶ Resources are constrained
- ▶ Not much memory
- ▶ Memory not fast enough
- ▶ Not many processors
- ▶ Not much CPU power
- ▶ Not much power – running on batteries
- ▶ Little, if any, disk space
- ▶ Programming closely to / at the hardware level (DMA, interrupts, ...)
- ▶ Expected to run continuously
- Your `printf("Log: _val=%03d\n", i)` will fail at some point... (Why?)

Austerity programming...

Copyright ©2012 Christos Kloukinas  School of Informatics 9/34

R-T vs Mainstream Computing

- ▶ R-T development is difficult because mainstream computing strives to optimise the *average* case
 - ▶ Out-of-order instruction execution
 - ▶ Speculative execution (e.g., pre-fetching, branch pred.)
 - ▶ Different levels of caches and memories
 - ▶ Virtual memory
 - ▶ ...
- ▶ All of these make the platform more unpredictable
- ▶ R-T system environments are sometimes too harsh for modern hardware
 - ▶ Aeroplanes, satellites: Radiation can cause run-time HW errors
 - ▶ Need specialised, hardened components, error checks, ...
- ▶ Older, more predictable components get discontinued quickly
 - ▶ Aeroplane manufacturers need to stock large quantities of CPUs, etc. to ensure they'll have spare parts for the whole life-time of their planes
- ▶ Technology advancement can work against us sometimes

Copyright ©2012 Christos Kloukinas  School of Informatics 11/34

R-T Types – Missing Deadlines

There are generally three types of R-T systems

- ▶ Hard (ABS): Missing the deadline is a failure
Response is no longer of any use after the deadline
- ▶ Soft (data acquisition): Missing the deadline is unfortunate
Response after the deadline still worth something
Or acceptable to miss k out of n deadlines (but late response useless)
- ▶ Isochronous (motion control applications): response must be almost at the deadline (to coordinate robots)

Copyright ©2012 Christos Kloukinas  School of Informatics 8/34

R-T and Critical Systems

- ▶ R-T systems are often *Critical* systems
- ▶ When they fail something else fails badly
- ▶ Types of Criticality
 - ▶ Safety Critical – ABS, trains, planes, nuclear plants
Failure → People die
 - ▶ Mission Critical – Mars Rover robot, missile
Failure → Mission fails, equipment is lost
 - ▶ Business Critical – High Frequency Trading
Failure → Business fails
- ▶ Need to guarantee (non-)behaviours
 - ▶ The system shall do X
 - ▶ The system shall not do Y
- ▶ Cannot do so without careful design, analysis, and verification/validation (*verification vs validation?*)

Copyright ©2012 Christos Kloukinas  School of Informatics 10/34

WCET Analysis

- ▶ Measure and Estimate (Guesstimate...)
- ▶ Consider reading a value from memory
 - ▶ Good case: Value exists in cache
Super-fast!
 - ▶ Bad case: Value needs to be read from memory into cache
Slow
 - ▶ Worst case: Cache slot has a "dirty" value.
Need to write that to memory (1 write),
Then read new value into slot (1 read)
Super-slow
- ▶ In many R-T systems developers disable caches
Each value read/write takes 1 memory access
Much slower average access but WCET decreased by half
- ▶ Real story is even worse...

Copyright ©2012 Christos Kloukinas  School of Informatics 12/34

Reading from Memory (Dibble, p. 39) – I

- ▶ Consider `ld r0, r7, 12` – load the value at “12 bytes off the address in register `r7`” into `r0` (`r0 = *(r7 + 12)`)
- ▶ Processor: 100 MIPS
- ▶ Instruction should take around $1/(100 * 10^6)$ -th of a sec, i.e., 10 nanoseconds
- ▶ But can take up to 10^8 nanoseconds...
- ▶ 10,000,000 times more? Why???

Copyright ©2012 Christos Kloukinas  School of Informatics 13/34

Reading from Memory (Dibble, p. 39) – III

Developers:

- ▶ Disable caches, or
Allocate parts of it to specific tasks/memory regions
- ▶ Disable paging (pin memory to RAM)
- ▶ Tune DMA to limit its bandwidth
- ▶ Pre-load TLB (when possible) at task start-up and context switches
- ▶ Manage interrupts: predict their effects, disable or mask them, ...

Open dragon's mouth, tie its tongue and it'll not be able to breathe fire on you any more – simples!

Copyright ©2012 Christos Kloukinas  School of Informatics 15/34

Task Types

- ▶ Periodic tasks
 - ▶ Phase φ $C \leq D$
 - ▶ Completion duration / cost C $D = P$ **easy**
 - ▶ Deadline D $\vee D < P$ **peasy**
 - ▶ Period P $\vee D > P$ **lemon squeezy**
 - ▶ N^{th} arrival $A(N)$ $A(N) = \varphi + (N - 1) * P$
- ▶ Sporadic tasks
 - ▶ Completion duration / cost C $C \leq D$
 - ▶ Deadline D $D = M$
 - ▶ Minimum Interarrival Time M $\vee D < M$
 - ▶ N^{th} arrival $A(N)$ $\vee D > M$
 - ▶ $A(N) \geq A(N - 1) + M$
- ▶ Aperiodic tasks
 - ▶ Completion duration C $C \leq D$
 - ▶ Deadline D
 - ▶ N^{th} arrival $A(N) = ?$ $A(N) \geq A(N - 1)$
- ▶ Non-R-T tasks


Copyright ©2012 Christos Kloukinas  School of Informatics 17/34

Reading from Memory (Dibble, p. 39) – II

The true, horrid story (summarised):

- ▶ CPU reads instruction first (WC: from memory)
- ▶ Instruction page not in the address translation cache (TLB)
- ▶ Find address page table entry (depends on CISC/RISC & OS)
- ▶ Memory page may be paged out – read it from disk
- ▶ CPU can now read data – same story as for instruction but
- ▶ If cache slot is dirty, write dirty value to memory first
More TLB/demand paging faults
- ▶ External interrupts occur
- ▶ Memory is slow because of a long DMA

Cannot analyse the code – must analyse the system

Copyright ©2012 Christos Kloukinas  School of Informatics 14/34

R-T Scheduling

- ▶ Static
 - ▶ Task execution order fixed at design time
 - ▶ A “cyclic” schedule: T1, T2, T3, T1, T2, T3, ...
 - ▶ Environment state, interrupts, etc. make no difference
 - ▶ WCETs need to be increased substantially
 - ▶ Resource budget increased too (to reduce WCETs a bit)
 - ▶ Slow and expensive BUT VERY PREDICTABLE!
 - Preferred for highly critical systems
- ▶ Dynamic
 - ▶ Dispatch rule selects next task at run-time
Usually based on priority
 - ▶ Fixed priorities, e.g., RMA (Rate Monotonic Analysis)
More predictable, even when system overloaded – robust
 - ▶ Dynamic priorities, e.g., EDF (Earliest Deadline First)
“Faster”, needs fewer resources, unpredictable under overload

Copyright ©2012 Christos Kloukinas  School of Informatics 16/34

Easy, Peasy, Lemon Squeezy

- ▶ $D = P$ RMA (Rate Monotonic Analysis)
- ▶ $D < P$ DMA (Deadline ...)
- ▶ $D > P$ This essentially changes the problem...
We no longer have N tasks, since a task can start a new period while its previous instance is still running.
Dibble, p. 162, footnote 3:
“The RTSJ does not support deadline greater than period.”

Copyright ©2012 Christos Kloukinas  School of Informatics 18/34

RMA / DMA Assumptions

RMA/**DMA** Assumptions:

1. all tasks are periodic **or sporadic**;
2. all tasks are released at the beginning of period and have a deadline equal **or less** to their period **or MIT**;
3. all tasks are independent, i.e., have no resource or precedence relationships;
4. all tasks have a fixed computation time, or at least a fixed upper bound on their computation times, which is less than or equal to their period/**deadline**;
5. no task may voluntarily suspend itself;
6. all tasks are fully preemptible;
7. all overheads are assumed to be 0;
8. there is just one processor.

RMA: $P_i < P_j \rightarrow \text{Priority}_i > \text{Priority}_j$

DMA: $D_i < D_j \rightarrow \text{Priority}_i > \text{Priority}_j$

Copyright ©2012 Christos Kloukinas  School of Informatics 19/34

Task Hardness and Error Handlers

- ▶ The hardness of the tasks defines what is the utility of continuing their execution after a deadline miss
- ▶ Essentially, it's a *guide* for error handling
- ▶ The error handling itself is performed by the deadline miss handler of the task
- ▶ Another indication of softness: $C \leq D$ in the average case but $C > D$ in the worst case
- ▶ Another handler: Cost overrun handler

Copyright ©2012 Christos Kloukinas  School of Informatics 21/34


Not Measuring Cost – Why?

- ▶ Sounds rather silly – we've got quite precise clocks, so why not?
- ▶ Consider the case where task A preempts task B
- ▶ Should the OS count the context switch cost on A's budget, on B's, half on each, on neither?
- ▶ The OS has no idea how you've done your analysis – it cannot read your mind
- ▶ So the vast majority of OS don't even try to measure "execution duration"

Copyright ©2012 Christos Kloukinas  School of Informatics 23/34

Task Types

- ▶ Periodic tasks
 - ▶ Phase φ
 - ▶ Completion duration / cost C **Hard?**
 - ▶ Deadline D
 - ▶ Period P **Soft?**
 - ▶ N^{th} arrival $A(N)$
- ▶ Sporadic tasks
 - ▶ Completion duration / cost C **Hard?**
 - ▶ Deadline D
 - ▶ Minimum Interarrival Time M **Soft?**
 - ▶ N^{th} arrival $A(N)$
- ▶ Aperiodic tasks
 - ▶ Completion duration C **Hard?**
 - ▶ Deadline D
 - ▶ N^{th} arrival $A(N) = ?$ **Soft?**
- ▶ Non-R-T tasks

Copyright ©2012 Christos Kloukinas  School of Informatics 20/34

Overrunning Cost vs Missing Deadline

- ▶ You take a shirt to the dry cleaners on Monday for ironing
 - ▶ They tell you come back on Thursday – $D = \text{Thursday}$
 - ▶ They're not gonna spend all that time to iron it. . .
 - ▶ Most probably $C = 5 \text{ min}$
 - ▶ Spending more time on it means they don't make money
 - ▶ Overrunning cost also means that their WCET estimation is wrong – the whole analysis is wrong!!!
 - ▶ Most OS don't even try to measure the time spent computing something
- Your overrun handlers will most probably never be called, even if there's an overrun. . . :- (

Copyright ©2012 Christos Kloukinas  School of Informatics 22/34

Some Java Issues for R-T

- ▶ GC runs at top priority
- ▶ GC runs whenever it wants – cannot control it
- ▶ Java doesn't really obey thread priorities (hints)
- ▶ No support for priority inversion avoidance protocols
- ▶ Yield may not yield. . .
- ▶ No way to specify a scheduling policy
- ▶ WORA (unrealistic in R-T systems – platform matters)
- ▶ No direct access to hardware (interrupt handlers, dev. drivers)
- ▶ Time support for milliseconds, no absolute time-points
- ▶ No way to safely interrupt/stop another thread immediately

Copyright ©2012 Christos Kloukinas  School of Informatics 24/34


NIST R-T Java Core Requirements – I

1. The specification must include a framework for the lookup and discovery of available profiles.
2. Any garbage collection that is provided shall have a bounded preemption latency.
3. The RTJ specification must define the relationships among real-time Java threads at the same level of detail as is currently available in existing standards documents.
4. The RTJ specification must include APIs to allow communication and synchronization between Java and non-Java tasks.

Copyright ©2012 Christos Kloukinas  School of Informatics 25/34

NIST Goals – I

1. RTJ should allow any desired degree of real-time resource management for the purpose of the system operating in real-time to any desired degree.
2. Support for RTJ specification should be possible on any implementation of the complete Java programming language.
3. Subject to resource availability and performance characteristics, it should be possible to write RTJ programs and components that are fully portable regardless of the underlying platform.
4. RTJ should support workloads comprised of the combination of real-time tasks and non-real-time tasks.
5. RTJ should allow real-time application developers to separate *concerns* between negotiating components. (deadlines, periods)
6. RTJ should allow real-time application developers to automate resource requirements analysis either at runtime or off-line.
7. RTJ should allow real-time application developers to write real-time constraints into their software.

Copyright ©2012 Christos Kloukinas  School of Informatics 27/34

RTSJ Guiding Principles

1. No Restriction to Particular Java Environments
2. Backward Compatibility
3. Write Once, Run Anywhere
Write Once Carefully, Run Anywhere Conditionally
4. Current Practice (YES) and Advanced Features (ENABLE)
5. Predictable Execution: The RTSJ shall hold predictable execution as first priority in all trade-offs
6. No Syntactic Extension (led to using closures)
7. Allow Variation in Implementation Decisions

Copyright ©2012 Christos Kloukinas  School of Informatics 29/34

NIST R-T Java Core Requirements – II

5. The RTJ specification must include handling of both internal and external asynchronous events.
6. The RTJ specification must include some form of asynchronous thread termination.
7. The RTJ core must provide mechanisms for enforcing mutual exclusion without blocking.
8. The RTJ specification must provide a mechanism to allow code to query whether it is running under a real-time Java thread or a non-real-time Java thread.
9. The RTJ specification must define the relationships that exist between real-time Java and non-real-time Java threads.

Copyright ©2012 Christos Kloukinas  School of Informatics 26/34

NIST Goals – II

8. RTJ should allow resource reservations and should enforce resource budgets. [CPU time, memory, and memory allocation rate.]
9. RTJ should support the use of components as “black boxes”; including such use on the same thread. † [Open – no consensus]
10. RTJ must provide real-time garbage collection when garbage collection is necessary. GC implementation information must be visible to the RTJ application
11. RTJ should support straightforward and reliable integration of independently developed software components (including changing hardware).
12. RTJ should be specified in sufficient detail to support (and with particular consideration for) targeting by other languages, such as Ada.
13. RTJ should be implementable on operating systems that support real-time behavior.

† Not to be understood by mere mortals.

Copyright ©2012 Christos Kloukinas  School of Informatics 28/34

RTSJ Seven Enhanced Areas

1. Thread Scheduling and Dispatching
 2. Memory Management
 3. Synchronization and Resource Sharing
 4. Asynchronous Event Handling
 5. Asynchronous Transfer of Control
 6. Asynchronous Thread Termination
 7. Physical Memory Access
- 7 bis Time values and clocks

WARNING: RTSJ addresses uni-processor systems. It tries not to preclude multi-processor ones but offers no mechanisms for these, e.g., thread allocation to processors.

Copyright ©2012 Christos Kloukinas  School of Informatics 30/34

What we've covered – I

- ▶ R-T System Concepts
- ▶ $R-T \neq \text{Fast}$ and $R-T \neq \text{High Throughput}$
- ▶ $R-T = \text{Predictable}$
- ▶ Types: Hard, Soft, Isochronous
- ▶ R-T & Embedded Systems
- ▶ R-T & Critical Systems
- ▶ R-T vs Mainstream Computing
- ▶ WCET blues
- ▶ R-T Tasks: Periodic, Sporadic, Aperiodic
- ▶ Error handlers for missing deadlines, overrunning cost
- ▶ Cost vs Deadline – both matter, can only observe the latter

Copyright ©2012 Christos Kloukinas  School of Informatics 31/34

What we haven't covered

- ▶ R-T Scheduling algorithms.
See: Liu & Layland (RMA + EDF), and Sha et al.
- ▶ Finding the appropriate analysis method in the literature is an NP-complete problem. . .
- ▶ You may have to do to your system what old boy Procrustes did to travellers. . .

Copyright ©2012 Christos Kloukinas  School of Informatics 33/34

What we've covered – II

- ▶ Java issues for R-T
- ▶ NIST R-T Java Core Requirements
- ▶ NIST Goals
- ▶ RTSJ Guiding Principles
- ▶ RTSJ Enhanced Areas

Copyright ©2012 Christos Kloukinas  School of Informatics 32/34

Study Material

- ▶ Wellings "Concurrent and Real-Time Programming in Java"
- ▶ Dibble "Real-Time Java Platform Programming"
- ▶ Bruno & Bollella "Real-Time Java Programming with Java RTS"
- ▶ Liu & Layland "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment" (introduced RMA and EDF)
- ▶ Sha et al. "Real Time Scheduling Theory: A Historical Perspective"
- ▶ NIST Special Publication 500-243 "Requirements For Real-time Extensions For the Java Platform"
<http://1.usa.gov/hkSM4o>
(<http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf>)
- ▶ R-T Specification for Java: www.rtsj.org

Copyright ©2012 Christos Kloukinas  School of Informatics 34/34

Java for Real-Time

Part 3 – Using RTSJ

Christos Kloukinas

School of Informatics
City University London

Copyright ©2012 Christos Kloukinas  School of Informatics 1/74

Contents

Threads in RTSJ

RealtimeThread – The API
Simple Example (Dibble)
Example Handling Deadline Misses (Dibble)
Another Example Handling Deadline Misses (Wellings)
Wellings on Deadlines
RTSJ and Aperiodic/Sporadic R-T Threads
Wellings' AperiodicThread
RTT & NHRTT

Handling Events

AsyncEvent & AsyncEventHandler
Events vs Threads
RTSJ Timers

Conclusions

What we've covered
What we haven't covered
Study Material

Copyright ©2012 Christos Kloukinas  School of Informatics 3/74

Time

- ▶ Improved support for time and clocks (`javax.realtime.`)
 - ▶ **HighResolutionTime** allows nano-second granularity
 - ▶ Abstract class with three concrete sub-classes:
 - ▶ **RelativeTime** what we had (aka "Time Duration"...) (`Duration`: NUMBER UNIT, e.g., 3 sec)
 - ▶ **AbsoluteTime** (YEAAAAH!!!) (aka "Time Point"...) Defined with respect to some point in time, e.g., Jan 1 1970 for the wall clock or system start-up for a monotonic clock (`Point`: REFERENCE-POINT NUMBER UNIT, e.g., (Jan 1, 1970) 1 month -> Feb 1, 1970)
 - ▶ **RationalTime**: sub-class of **RelativeTime** with an associated frequency for representing rates
- Deprecated. As of RTSJ 1.0.1.**

CAUTION: These classes (Rel/Abs) are explicitly unsafe in multithreaded situations when they are being changed. No synchronization is done.

Copyright ©2012 Christos Kloukinas  School of Informatics 5/74

Contents

Some "Easy" Parts

The "Not So Easy" Parts

Release Parameters
Scheduling Parameters
Scheduler
MonitorControl
Wait-Free
Remaining Schedulable Parameters

Memory in RTSJ

Non-GC Memory Areas
Problems with Non-GC Memory
Constraints on Referencing
Memory Classes
MemoryParameters
Memory Area API

Copyright ©2012 Christos Kloukinas  School of Informatics 2/74

RTSJ Seven Enhanced Areas

1. Thread Scheduling and Dispatching
2. Memory Management
3. Synchronization and Resource Sharing
4. Asynchronous Event Handling
5. Asynchronous Transfer of Control
6. Asynchronous Thread Termination
7. Physical Memory Access

7 bis Time values and clocks

WARNING: No explicit support for multi-core/processor systems.

(It doesn't preclude them either)

Copyright ©2012 Christos Kloukinas  School of Informatics 4/74

Clocks

```
public abstract class javax.realtime.Clock {  
  
    abstract RelativeTime getEpochOffset();  
    // ThisOrigin - Epoch (Jan 1, 1970) †  
    static Clock getRealtimeClock(); // X*X*X*X*X*X*X  
    // There is always at least one clock object  
    // available: the system real-time clock.  
    abstract RelativeTime getResolution(); // tick2tick  
    abstract AbsoluteTime getTime();  
    // Gets the current time IN A NEWLY ALLOCATED OBJECT.  
    abstract AbsoluteTime getTime(AbsoluteTime dest)  
    // Gets the current time IN AN EXISTING OBJECT.  
    abstract void setResolution(RelativeTime res  
    // Set the resolution of this clock.  
} // BE CAREFUL ABOUT OBJECT ALLOCATION!!!!
```

† Cause Dennis Ritchie & Ken Thompson say so

Copyright ©2012 Christos Kloukinas  School of Informatics 6/74

System Info

```
public final class javax.realtime.RealtimeSystem {
    static byte BIG_ENDIAN, BYTE_ORDER, LITTLE_ENDIAN;

    static GarbageCollector      currentGC();
    static int  getConcurrentLocksUsed();
    static int  getMaximumConcurrentLocks();
    static void setMaximumConcurrentLocks(int numLocks);
    static void setMaximumConcurrentLocks(int n,
                                             boolean hard);
    static MonitorControl getInitialMonitorControl();
    static RealtimeSecurity getSecurityManager();
    static void setSecurityManager(RealtimeSecurity mgr);
}
//MonitorControl is about synchronisation; more later.
```

Copyright ©2012 Christos Kloukinas  School of Informatics 7/74

The Not So “Easy” Parts

- ▶ So far we’ve considered features that are:
 - ▶ More or less independent from each other
AbsoluteTime, RealtimeSystem, Clock
 - ▶ Improvements of existing ones
HighResolutionTime
- ▶ The remaining ones are interdependent.
- ▶ In order to get “R-T” threads one needs to consider:
 - ▶ Temporal behaviour
 - ▶ Memory behaviour
 - ▶ Error handlers and asynchronous transfer of control

So from now on we need to tread carefully...

Copyright ©2012 Christos Kloukinas  School of Informatics 9/74

Reminder: Task Types

- ▶ Periodic tasks
 - ▶ Phase φ $C \leq D$
 - ▶ Completion duration / cost C $D = P$ **easy**
 - ▶ Deadline D $\vee D < P$ **peasy**
 - ▶ Period P $\vee D > P$ **lemon squeezey**
 - ▶ N^{th} arrival $A(N)$ $A(N) = \varphi + (N - 1) * P$
- ▶ Sporadic tasks
 - ▶ Completion duration / cost C $C \leq D$
 - ▶ Deadline D $D = M$
 - ▶ Minimum Interarrival Time M $\vee D < M$
 - ▶ N^{th} arrival $A(N)$ $\vee D > M$
 - ▶ $A(N) \geq A(N - 1) + M$
- ▶ Aperiodic tasks
 - ▶ Completion duration C $C \leq D$
 - ▶ Deadline D
 - ▶ N^{th} arrival $A(N) = ?$ $A(N) \geq A(N - 1)$
- ▶ Non-R-T tasks

Copyright ©2012 Christos Kloukinas  School of Informatics 11/74

OS (POSIX) SIGNALS

```
public final class POSIXSignalHandler {
    static void set/add/removeHandler(int signal,
                                       AsyncEventHandler handler);

    static int SIGABRT, SIGALRM, SIGBUS, SIGCHLD, SIGCLD,
               SIGCONT, SIGEMT, SIGFPE, SIGHUP, ...;
};
// Use like:
POSIXSignalHandler.addHandler(SIGINT, intHndlr);

// Only supported when OS has POSIX signals!!!
AsyncEventHandler ??? (for later...)
```

Copyright ©2012 Christos Kloukinas  School of Informatics 8/74

Schedulable Objects – Not Just Runnable

```
public interface Schedulable
    extends java.lang.Runnable {
    void setScheduler(Scheduler scheduler,
                     SchedulingParameters scheduling,
                     ReleaseParameters release,
                     MemoryParameters memoryParameters,
                     ProcessingGroupParameters group);

    // ...
}
// All Known Implementing Classes:
// AsyncEventHandler, BoundAsyncEventHandler,
//
// RealtimeThread, NoHeapRealtimeThread
```

Copyright ©2012 Christos Kloukinas  School of Informatics 10/74

ReleaseParameters and PeriodicParameters

```
public abstract class ReleaseParameters;
protected ReleaseParameters(RelativeTime cost,
                             RelativeTime deadline,
                             AsyncEventHandler overrunHandler,
                             AsyncEventHandler missHandler);
// Extended by: PeriodicParameters, AperiodicParameters

PeriodicParameters(HighResolutionTime phase,
                   RelativeTime period,
                   RelativeTime cost,
                   RelativeTime deadline,
                   AsyncEventHandler overrunHandler,
                   AsyncEventHandler missHandler);

// CAUTION: An implementation is not required to
// ensure that each AsyncEventHandler with periodic
// parameters is released periodically. No idea...

// CAUTION: Cost monitoring is an optional facility
// in an implementation of the RTSJ. See? I told you so...
```

Copyright ©2012 Christos Kloukinas  School of Informatics 12/74

PeriodicParameters Defaults

Attribute	Default Value
phase	new RelativeTime(0,0)
period	No default. A value must be supplied
cost	new RelativeTime(0,0)
deadline	new RelativeTime(period)
overrunHandler	None (null)
missHandler	None (null)

NOTE – The first release time is:

- ▶ If phase is a `RelativeTime` then
SUM(Time you invoked the `start()` method , phase)
- ▶ If phase is an `AbsoluteTime` then
MAX(Time you invoked the `start()` method , phase)

Copyright ©2012 Christos Kloukinas  School of Informatics 13/74

AperiodicParameters Defaults

Attribute	Default Value
cost	new RelativeTime(0,0)
deadline	new RelativeTime(Long.MAX_VALUE, 999999)
overrunHandler	None
missHandler	None
Arrival time queue size	0
Queue overflow policy	SAVE arrivalTimeQueueOverflowSave

Copyright ©2012 Christos Kloukinas  School of Informatics 15/74

SporadicParameters Defaults

Attribute	Default Value
minInterarrival time	No default. A value must be supplied
cost	new RelativeTime(0,0)
deadline	new RelativeTime(mit)
overrunHandler	None
missHandler	None
MIT violation policy	SAVE mitViolationSave
Arrival queue overflow policy	SAVE arrivalTimeQueueOverflowSave
Initial arrival queue length	0

Copyright ©2012 Christos Kloukinas  School of Informatics 17/74

AperiodicParameters

```
AperiodicParameters(RelativeTime cost,
                    RelativeTime deadline,
                    AsyncEventHandler overrunHandler,
                    AsyncEventHandler missHandler);

static String arrivalTimeQueueOverflowSave; // DEFAULT!!!
// Increase queue length on overflow.
static String arrivalTimeQueueOverflowExcept;
// throw ArrivalTimeQueueOverflowException if overflow.
static String arrivalTimeQueueOverflowIgnore; // Drop it.
static String arrivalTimeQueueOverflowReplace;
// "this arrival replaces _a_ previous arrival."
// "_a_"? - Which one?

String getArrivalTimeQueueOverflowBehavior();
void setArrivalTimeQueueOverflowBehavior(String bhvr);
int getInitialArrivalTimeQueueLength();
void setInitialArrivalTimeQueueLength(int initial);
// Sub-classed by SporadicParameters
```

Copyright ©2012 Christos Kloukinas  School of Informatics 14/74

SporadicParameters

```
SporadicParameters(RelativeTime minInterarrival,
                  RelativeTime cost,
                  RelativeTime deadline,
                  AsyncEventHandler overrunHandler,
                  AsyncEventHandler missHandler);

static String mitViolationSave; // DEFAULT!!!
// Minimum interarrival time violations -- the arrival
// time for any instance of Schedulable which has this
// as its instance of ReleaseParameters is not compared
// to the specified minimum interarrival time.
static String mitViolationExcept;
// throw MITViolationException
static String mitViolationIgnore;
static String mitViolationReplace;
// "[the] arrival replaces _a_ previous arrival." - "a"?

String getMitViolationBehavior();
void setMitViolationBehavior(String behavior);
```

Copyright ©2012 Christos Kloukinas  School of Informatics 16/74


Timeout – Let's Ruminare a Bit. . .

- ▶ Theory vs Reality
 - ▶ Cost & Cost Overrun Handlers
 - ▶ You can define them – You **SHOULD** define them!
 - ▶ They'll be ignored. . . (almost guaranteed)
 - ▶ Why define them then? *To document your intentions.*
 - ▶ **Write programs for other people, not for the compiler.**
 - ▶ *Don't waste your work on a stupid machine. . .*
 - ▶ `arrivalTimeQueueOverflowSave/Except/...` and `mitViolationSave/...` are strings. Why? Why not enums?
 - ▶ Why some parts in 1.0.2 are already Deprecated?
 - ▶ Very difficult to design a system of such complexity
 - ▶ Need to produce a prototype and use it to see what works well and what needs improvement
 - ▶ If anything, the deprecated features show that RTSJ is really being used! :-)
- Only dead things don't change. . .

Copyright ©2012 Christos Kloukinas  School of Informatics 18/74

Timeout – II

- ▶ Shit happens – got to deal with it (*no one else will...*)
 - ▶ Deadline Miss Handler – Default: null (**EVIL!!! NOT Evel Knievel...**)
 - ▶ Arrival Queue Overflow – Default: Increase queue (is this OK?)
 - ▶ Do you have enough memory for the new queue?
 - ▶ For how long can you keep increasing it?
 - ▶ How about replacing a previous happening?
 - ▶ The specification does not specify which one will be replaced:
The first? The last? A random one? :-{
 - ▶ Maybe this is controllable by defining your own scheduler? No idea really...
 - ▶ Throwing an exception doesn't look very nice – you miss the info concerning the latest arrival
 - ▶ Ignore: even worse! You don't even find out that there's been a problem!
 - ▶ Exception with the new arrival attached? That'd be nice...
- ▶ MIT violations – same story... Oh, well...

Copyright ©2012 Christos Kloukinas  School of Informatics 19/74

Talking of the Devil – Scheduler


```
public abstract class Scheduler {
    // Subclasses of Scheduler are used for alternative
    // scheduling policies and should define an instance()
    // class method to return their default instance.

    static Scheduler getDefaultScheduler();
    static void setDefaultScheduler(Scheduler s);
    abstract String getPolicyName();

    abstract void fireSchedulable(Schedulable s);
    // Trigger the execution of s (like an AsyncEventHandler).

    protected abstract boolean addToFeasibility(Schedulable s);
    protected abstract boolean removeFromFeasibility(Schedulable s);
    abstract boolean isFeasible();

    abstract boolean setIfFeasible(Schedulable s,
        SchedulingParameters scheduling, ReleaseParameters release,
        MemoryParameters memory, ProcessingGroupParameters group);
    // Resets s if feasible, even if already in the schedule.
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 21/74

PriorityScheduler Behaviour

- ▶ Supports preemptive priority-based dispatching – highest priority wins always
- ▶ Particular implementations required to document where in the run queue (of priority p) a preempted Schedulable is placed
- ▶ A blocked Schedulable that becomes runnable placed at the back of its priority queue
- ▶ A thread that yields goes at the back of its priority queue
- ▶ Follows priority inheritance when accessing resources

Copyright ©2012 Christos Kloukinas  School of Informatics 23/74

SchedulingParameters

```
public abstract class SchedulingParameters; // Nothing here

public class PriorityParameters
    extends SchedulingParameters {
    PriorityParameters(int priority); //+ get/setPriority

public class ImportanceParameters
    extends PriorityParameters {
    ImportanceParameters(int priority, int importance);
    // Importance is an additional scheduling metric that
    // may be used by some priority-based scheduling
    // algorithms during overload conditions to
    // differentiate execution order among threads of the
    // same priority.

    // The base scheduling algorithm represented by
    // PriorityScheduler must not consider importance. :-)
```

Want to be fancy? Write your own scheduler!

Copyright ©2012 Christos Kloukinas  School of Informatics 20/74

What Exists for Sure? PriorityScheduler

```
public class PriorityScheduler extends Scheduler {
    static PriorityScheduler instance();
    String getPolicyName(); // "Fixed Priority"

    int getMaxPriority();
    int getMinPriority();
    // getMaxPriority() - getMinPriority() >= 28
    int getNormPriority();
    static int getMaxPriority(java.lang.Thread thread);
    // If a real-time thread, then maximum priority. If a Java
    // thread then the maximum priority of its thread group.
    static int getMinPriority(Thread thread);
    // If a real-time thread, then minimum priority. If a Java
    // thread then Thread.MIN_PRIORITY.
    static int getNormPriority(Thread thread);
    // "normal" R-T priority or Thread.NORM_PRIORITY

    // All of 'em throw an exception if thread is real-time but
    // not scheduled by this scheduler instance.
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 22/74

Controlling Monitor Synchronisation

```
javax.realtime.RealtimeSystem.getInitialMonitorControl();
public abstract class MonitorControl {
    static MonitorControl getMonitorControl();
    static MonitorControl setMonitorControl(MonitorControl p);
    static MonitorControl getMonitorControl(Object o);
    static MonitorControl setMonitorControl(Object o,
        MonitorControl policy);
}

public class PriorityInheritance extends MonitorControl {
    static PriorityInheritance instance(); // singleton
}

public class PriorityCeilingEmulation extends MonitorControl {
    static PriorityInheritance instance(); // singleton
    static PriorityCeilingEmulation getMaxCeiling();
    // Ceiling is PriorityScheduler.instance().getMaxPriority()
    int getCeiling();
} // Priority ceiling of this PriorityCeilingEmulation object
```

- ▶ PriorityInheritance is guaranteed to exist

- ▶ PriorityCeilingEmulation is not

Copyright ©2012 Christos Kloukinas  School of Informatics 24/74

The Problem: **Priority Inversion**

- ▶ I go to a small village with one taxi
- ▶ I take the taxi and leave
- ▶ Maradona is also at the village and needs a taxi now!
- ▶ I ask the taxi driver to stop so as to take a call from my significant other (signal is bad while travelling) – *she's more important than arriving at my destination on time*
- ▶ Maradona is left there waiting while I talk!!! He's not amused...

The problem:

- ▶ I = Low priority task
- ▶ Significant Other (SO) = Higher priority task
- ▶ Maradona (M) = Highest priority task
- ▶ Priority Inversion (BAD!) = M waiting for SO to finish, so that I can release the resource M needs

Copyright ©2012 Christos Kloukinas  School of Informatics 25/74

Avoiding Synchronisation

```
public class WaitFreeReadQueue;
public class WaitFreeWriteQueue;

// Encapsulates both
public class WaitFreeDequeue; // Deprecated. 1.0.1

// Just a teaser...
```

Copyright ©2012 Christos Kloukinas  School of Informatics 27/74

Avoiding the GC

- ▶ In C/C++, you can use static memory, the stack or the heap.
 - ▶ Static – fixed size, always there
 - ▶ Stack – dynamic size, automatic de-allocation when a function/block ends
 - ▶ Heap – dynamic size, manual de-allocation (or by GC)
- ▶ RTSJ does the same:
 - ▶ **ImmortalMemory** – fixed size, always there. GC doesn't touch it.
 - ▶ **ScopedMemory** – dynamic size, automatic de-allocation once all Schedulable objects inside it end
 - ▶ **HeapMemory** – the GC's playground.
- ▶ If you want R-T behaviour then you need to use either the **ImmortalMemory** or the **ScopedMemory**

Dibble Avoid the **ImmortalMemory** – very easy to abuse it and fill it up

Copyright ©2012 Christos Kloukinas  School of Informatics 29/74


The Solutions: **Priority Inheritance & Ceiling**

Priority Inheritance:

- ▶ When M needs one of my resources, my priority increases to be equal to M's
- ▶ SO cannot preempt me any more (yes, she'll have to wait)
- ▶ I release my resource as soon as possible and lower my priority to what it was before.
- ▶ M is driven away...

Priority Ceiling:

- ▶ Through some magic, each resource knows who's the highest priority task that will ever need it
- ▶ That task's priority becomes the ceiling C of the resource
- ▶ When I get a resource I change my priority to its C
- ▶ Messi will need to stop whatever he's doing while I travel with the taxi, in case Maradona is in the village and needs my taxi
- ▶ *One has to set some priorities in life...*

Copyright ©2012 Christos Kloukinas  School of Informatics 26/74

Remaining **Schedulable** Parameters


```
public interface Schedulable {
    void setScheduler(Scheduler scheduler, // Covered
        SchedulingParameters scheduling, // Covered
        ReleaseParameters release, // Covered
        MemoryParameters memoryParameters,
        ProcessingGroupParameters group);
}
```

- ▶ **ProcessingGroupParameters** – we'll ignore them
- ▶ **MemoryParameters** – we'll **NOT** ignore them...
- ▶ Memory is the main resource to control for predictability (once we've got a good scheduler for the CPU)
- ▶ Remember plain old Java's problem?
 - ▶ Everything gets allocated on the heap
 - ▶ GC can run whenever it likes
 - ▶ GC can run for as long as it likes
 - ▶ GC has top priority
- ▶ **Need to be able to avoid the GC**

Copyright ©2012 Christos Kloukinas  School of Informatics 28/74

The (A) Memory Problem in C++

```
#include <iostream>
using namespace std;    static int *fubar = 0;
void bar();
void fu() {
    int ar[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    fubar = ar+4;
    for (int i = -4; i < 6; ++i)
        cerr << "fu_i:_" << i << "_fubar:_"
                << fubar[i] << endl; // SAFE?
    bar();
}
void bar() {
    for (int i = -4; i < 6; ++i) fubar[i] += 5; // SAFE?
    for (int i = -4; i < 6; ++i)
        cerr << "bar_i:_" << i << "_fubar:_"
                << fubar[i] << endl; // SAFE?
}
int main() {
    fu(); cerr << "Once_more\n"; bar();
    cerr << "OK" << endl;    return 0; }
```

Copyright ©2012 Christos Kloukinas  School of Informatics 30/74

The Memory Problem in C++ – Results

```
fu i: -4 fubar: 0      bar i: -4 fubar: 4197193
fu i: -3 fubar: 1      bar i: -3 fubar: 0
fu i: -2 fubar: 2      bar i: -2 fubar: -1496366290
fu i: -1 fubar: 3      bar i: -1 fubar: 32532
fu i: 0 fubar: 4       bar i: 0 fubar: 32532
fu i: 1 fubar: 5       bar i: 1 fubar: 0
fu i: 2 fubar: 6       bar i: 2 fubar: 4196747
fu i: 3 fubar: 7       bar i: 3 fubar: 0
fu i: 4 fubar: 8       bar i: 4 fubar: 5
fu i: 5 fubar: 9       bar i: 5 fubar: 5
bar i: -4 fubar: 5     OK
bar i: -3 fubar: 6
bar i: -2 fubar: 7     // First bar went fine.
bar i: -1 fubar: 8     // Second one corrupted the memory.
bar i: 0 fubar: 9
bar i: 1 fubar: 10     // What's the problem?
bar i: 2 fubar: 11     //
bar i: 3 fubar: 12     // "Dangling Pointers"
bar i: 4 fubar: 13
bar i: 5 fubar: 14
Once more
```

Copyright ©2012 Christos Kloukinas  School of Informatics 31/74

The Memory Problem – II

	Reference To Heap	To Immortal	To Scoped
Heap	Permit	Permit	Forbid
Immortal	Permit	Permit	Forbid
Scoped	Permit	Permit	Only same OR predecessor scope
Local Var	Permit	Permit	Permit

Why?

- ▶ Immortal objects live for ever
- ▶ Heap ones can potentially live for ever
- ▶ So immortal ones can reference heap ones and vice-versa
- ▶ Local vars have the shortest life span, so can reference anything
- ▶ Scoped objects will be reclaimed before immortal ones – Forbid
- ▶ Scoped objects may be reclaimed before heap ones – Forbid
- ▶ Scoped objects in successor scopes will be reclaimed before ones in predecessor scopes – Forbid

Copyright ©2012 Christos Kloukinas  School of Informatics 33/74

RTSJ MemoryParameters

```
▶ SizeEstimator
void reserve(Class c, int number);
void reserve(SizeEstimator size);
void reserve(SizeEstimator estimator, int number);
void reserveArray(int length); //of references
void reserveArray(int length, Class type); //of type
long getEstimate();

▶ MemoryParameters
MemoryParameters(long maxBytesInitMem, // 0=no memory
    long maxBytesImmortal, // NO_MAX=unlimited mem
    long heapAllocRateBytesPerSec) // Same
// Limits may not be enforced. Specify them nevertheless.
```

Copyright ©2012 Christos Kloukinas  School of Informatics 35/74

The Memory Problem

- ▶ The problem lies in the use of the stack/scoped memory
- ▶ Compiler assumes that when the code goes out of scope, the scope's objects can be automatically collected
- ▶ If we allow objects in the static/immortal or heap area to point to the objects in the stack then we can end up pointing to garbage
- ▶ Same goes for parent/child stack frames
 - ▶ Object in a child frame can point to objects in a parent frame
 - ▶ Never the other way round!

	Reference To Heap	To Immortal	To Scoped
Heap	Permit	Permit	Forbid
Immortal	Permit	Permit	Forbid
Scoped	Permit	Permit	Only same OR predecessor scope
Local Var	Permit	Permit	Permit

Copyright ©2012 Christos Kloukinas  School of Informatics 32/74

RTSJ Memory Classes

- ▶ **MemoryArea**
 - ▶ HeapMemory
 - ▶ ImmortalMemory
 - ▶ ImmortalPhysicalMemory
 - ▶ ScopedMemory
 - ▶ LTMemory
 - ▶ LTPhysicalMemory
 - ▶ VTMemory
 - ▶ VTPhysicalMemory
- ▶ **RawMemoryAccess**
 - ▶ RawMemoryFloatAccess
- ▶ **SizeEstimator**
- ▶ **GarbageCollector** – Max wait for a preemption-safe point?


```
RelativeTime getPreemptionLatency();
```
- ▶ **MemoryParameters**

Copyright ©2012 Christos Kloukinas  School of Informatics 34/74

Using Memory Areas

```
static MemoryArea getMemoryArea(Object o);
MemoryArea(long size, Runnable logic);
MemoryArea(SizeEstimator size, Runnable logic);
void enter();
void enter(Runnable logic);
void executeInArea(Runnable logic);

long size();
long memoryConsumed();
long memoryRemaining();

Object newArray(Class type, int number);
Object newInstance(Class type);
Object newInstance(Reflect.Constructor c, Object[] args);
```

Copyright ©2012 Christos Kloukinas  School of Informatics 36/74

Reclaiming Memory Without GC

```
import javax.realtime.*;
public class ScopedMemTest {
    public static void main(String args[]) {
        ScopedMemory mem = new LTMemory(10000, 15000);

        mem.enter( new Runnable() { // Anonymous Java class
            public void run() { // Remember closures?
                int array[][] = new int[100][];
                for (int i = 0; i < 100; ++i) {
                    array[i] = new int[100];
                    for (int j = 0; j < 100; ++j)
                        array[i][j] = i+j;
                }
            }
        });
        // all objects in mem have been reclaimed here.
        System.err.println("Back_to_the_previous_memory_area");
    } // which is the previous memory area?
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 37/74

Memory Areas – Have we Forgotten Something?

```
void enter(); // [*]
void enter(Runnable logic); // [*]
void executeInArea(Runnable logic); // [*]
```

...

- ▶ **These take a Runnable but execute it sequentially**
- ▶ Just like the C/C++ block construct for allocation on the stack, they do not execute the block code concurrently
- ▶ Useful for controlling memory de-allocation but what about our Schedulable?


Copyright ©2012 Christos Kloukinas  School of Informatics 39/74

RealtimeThread

```
public class RealtimeThread
    extends Thread implements Schedulable {
    public RealtimeThread(SchedulingParameters scheduling,
        ReleaseParameters release,
        MemoryParameters memory,
        MemoryArea area,
        ProcessingGroupParameters group,
        Runnable logic)
        throws IllegalAssignmentError;
    public static RealtimeThread currentRealtimeThread();
    public static MemoryArea getCurrentMemoryArea();
    public MemoryArea getMemoryArea(); // area

    public static void sleep(Clock c, HighResolutionTime t);

    public boolean waitForNextPeriod(); // true if no miss
    public boolean waitForNextPeriodInterruptible()
        throws InterruptedException;
    public void deschedulePeriodic()/schedulePeriodic();
    // Block at next WFNP until schedulePeriodic is called
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 41/74

In C/C++

```
int main() {
    {
        int array[100][100]; // allocated on the stack
        for (int i = 0; i < 100; ++i)
            for (int j = 0; j < 100; ++j)
                array[i][j] = i+j;
    } // array de-allocated!
    cerr << "Back_to_the_previous_memory_area\n";
    return 0;
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 38/74

The Story so Far

- ▶ We can create new Schedulable objects (instead of Runnable)
- ▶ We can specify in quite a lot of detail their characteristics
- ▶ We can even create new memory areas for them
- ▶ But what's the point if all we can do is run them sequentially?
- ▶ So it's time to look at what RTSJ has to offer for R-T threads!
- ▶ Previously we've classified tasks as Periodic, Sporadic, Aperiodic
- ▶ Orthogonally we've classified them as Hard and Soft
- ▶ RTSJ classifies them according to their predictability (so more like Hard and Soft)
- ▶ This predictability is mainly a result of the memory area they execute in, i.e., how much they'll be disturbed by the GC

Threads RealtimeThread, NoHeapRealtimeThread
Event Handlers AsyncEventHandler, BoundAsyncEventHandler

Copyright ©2012 Christos Kloukinas  School of Informatics 40/74

RTT Example (Dibble, p. 158)

```
import javax.realtime.*; // Dibble, p. 158
public class Periodic1 {
    public static void main(String [] args) {
        SchedulingParameters sched=
            new PriorityParameters(
                PriorityScheduler.getMinPriority(null)+10);
        ReleaseParameters rel=
            new PeriodicParameters(
                new RelativeTime(), // Start at .start()
                new RelativeTime(1000, 0), // 1 second period
                null, // cost
                new RelativeTime(500,0), // deadline=period/2
                null, // no overrun handler
                null); // no miss handler

        RealtimeThread rt= new RealtimeThread(sched, rel);
        rt.start();
        try { rt.join(); } catch (Exception e) { };
    }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 42/74

RTT Example, More Complex (Dibble, p. 161)

```
import javax.realtime.*;
public class MyThread extends RealtimeThread {
    static final RelativeTime DELAY_INC=new RelativeTime(80,0)
    static final RelativeTime DELAY_DEC=new RelativeTime(60,0)
    public MyThread(SchedulingParameters sched,
                    ReleaseParameters release) {
        super(sched, release);
    }
    public void run() {
        RelativeTime delay = new RelativeTime(DELAY_INC);
        int misses = 0;

        while (true) {
            do {
                try { sleep(delay); // Use some time
                } catch (InterruptedException ie) { }
                delay.add(DELAY_INC, delay); // Use more next time
                System.out.println("Ding:_" + delay.getMilliseconds());
            } while (waitForNextPeriod());
        }
    }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 43/74

Catching Deadline Misses – Wellings, p. 258–260

```
public class HealthMonitor {
    public void persistentDeadlineMiss(Schedulable s); }
class DeadlineMissHandler extends AsyncEventHandler {
    public DeadlineMissHandler(HealthMonitor mon,
                              int threshold) {
        super(new PriorityParameters(
            PriorityScheduler.MAX_PRIORITY),
            null, null, null, null, null);
        myHealthMonitor = mon; myThreshold = threshold;
    }
    public void setThread(RealtimeThread rt) { myrt = rt; }
    public void handleAsyncEvent() {
        if(++missDeadlineCount < myThreshold)
            myrt.schedulePeriodic();
        else myHealthMonitor.persistentDeadlineMiss(myrt);
    }
    private RealtimeThread myrt;
    private int missDeadlineCount = 0;
    private HealthMonitor myHealthMonitor;
    private final int myThreshold; }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 45/74

More Notes on Deadlines, etc.– Wellings, p. 252

- ▶ Upon a deadline miss the handler is released, the fireCount of the handler is increased by the deadlineMiss count + 1, and the thread's deadlineMiss count is set to zero
- ▶ WFPN returns immediately if there's a deadline miss, even if a deadline miss handler is active
- ▶ Programmers are responsible for handling any race conditions that might occur as a result of these :-)

Copyright ©2012 Christos Kloukinas  School of Informatics 47/74

RTT Example, More Complex (Dibble, p. 161) – II

```
// Recover from miss
System.out.println("Miss!");
delay.subtract(DELAY_DEC, delay); // Reduce load
if (++misses > 3) break; // We've had enough

while (waitForNextPeriod() == false) // Eat errors
    System.out.print(".");
System.out.println();
} // while (true)
}
```

- ▶ Without handlers cannot know what the problem is
- ▶ Good Citizen: try to reduce the load a bit
- ▶ Eat up any remaining missed periods (WFPN == false) so as to start with a fresh period that stands a chance of succeeding

Copyright ©2012 Christos Kloukinas  School of Informatics 44/74

Some Notes on Deadlines, etc.– Wellings, p. 250–251

- ▶ With a deadline miss handler, RTSJ assumes that the handler will eventually call schedulePeriodic (if it wants to reschedule the thread). So WFPN returns at the next release time after schedulePeriodic. **All releases in between are lost.**
- ▶ Without a appropriate handler, WFPN will not deschedule the real-time thread in the event of a deadline miss. RTSJ assumes that the thread itself will try to correct the situation and then call WFPN again. The thread may overrun its cost if the time to handle the condition has not been accounted for in the cost parameter.
- ▶ Application-level calls to deschedulePeriodic take effect when the current release completes and all deadline misses have been accounted for (even if the next release has already occurred).

Copyright ©2012 Christos Kloukinas  School of Informatics 46/74

RTT and Aperiodic/Sporadic Threads

- ▶ Aperiodic/Sporadic threads cannot use WFPN
- ▶ Dibble, p. 155:
"Aperiodic and sporadic scheduling make little sense for threads. The thread can have aperiodic or sporadic release parameters, but there is no way for the scheduler and thread to communicate about scheduling events. The [WFPN makes no] sense here, since aperiodic threads have no period by definition. The nonperiodic release parameters are intended for async event handlers. The AEH system includes suitable APIs for asynchronous and sporadic events. Perhaps RTSJ version 1.1 will include new methods for real-time threads that let them loop on aperiodic releases."
- ▶ **RTSJ only supports one-shot aperiodic/sporadic threads**
- ▶ Wellings (p. 254–258) provides a user implementation of a waitForNextRelease for aperiodic/sporadic threads and expects RTSJ 1.1 to include it.

Copyright ©2012 Christos Kloukinas  School of Informatics 48/74

Using Wellings' AperiodicThread

```
public class MyAperiodic extends AperiodicThread {
    public void run() {
        boolean noProblems = true;
        while(noProblems) {
            myLogic.run();
            noProblems = waitForNextRelease();
        }
        // Deal with deadline overrun.
    }
}
```

- ▶ First release happens through thread's `start()`
- ▶ Thread blocks on `waitForNextRelease()`
- ▶ Subsequent releases occur when `release()` is called
- ▶ Who will call `release()`?

Copyright ©2012 Christos Kloukinas  School of Informatics 49/74

Wellings' AperiodicThread – II

```
public boolean waitForNextRelease() {
    synchronized(this) {
        first = (first + 1) % size;
        // Check for miss deadline.
        AbsoluteTime lastDeadline =
            releaseTimes[first].add(myDeadline);
        if(myClock.getTime().compareTo(lastDeadline) == 1)
            return false; // deadline miss
        try {
            while(((first + 1) % size) == last) wait();
            return true;
        } catch(Exception e) {
            return false;
        }
    }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 51/74

RealtimeThread & NoHeapRealtimeThread

- ▶ RealtimeThread can allocate memory anywhere
- ▶ The GC may
 - ▶ Either preempt it (if it has a higher priority)
 - ▶ Or delay it for some short period while it tries to release the heap in a proper state
- ▶ If we don't want to allow the GC to have any effect at all then we need to use:

```
public class NoHeapRealtimeThread extends RealtimeThread
```

- ▶ **WARNING** A NHRTT may be preempted by the GC if its priority is lower than that of a RTT that uses the heap!!!
- ▶ A NHRTT cannot allocate on the heap.
- ▶ **It cannot even reference an object on the heap**
- ▶ A NHRTT may require significant run-time checks to ensure that it doesn't reference the heap

Copyright ©2012 Christos Kloukinas  School of Informatics 53/74

Wellings' AperiodicThread

```
import javax.realtime.*;
public class AperiodicThread extends RealtimeThread {

    public AperiodicThread(PriorityParameters scheduling,
        AperiodicParameters release, int queueSize) {
        super(scheduling, release);
        releaseTimes = new AbsoluteTime [queueSize];
        first= queueSize -1; last= 0; size=queueSize;
        myClock = Clock.getRealtimeClock();
        myDeadline = release.getDeadline();
        MonitorControl.setMonitorControl(this,
            PriorityInheritance.instance());
    }

    public void start() { // Save the release time.
        synchronized(this) {
            releaseTimes [last] = myClock.getTime();
            last = (last + 1) % size;
        }
        super.start();
    }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 50/74

Wellings' AperiodicThread – III

```
public void release() {
    synchronized(this) {
        // Throws ResourceLimitError.
        if(first == last)
            throw new ResourceLimitError();
        releaseTimes[last] = myClock.getTime();
        last = (last + 1) % size;
        notify();
    }
}

private AbsoluteTime releaseTimes[];
private int first, last, size;
private Clock myClock;
private RelativeTime myDeadline;
}
```

- ▶ If you plan to use it, make sure you read the respective chapter for the assumptions made by the code, e.g., that calls to `release` are buffered.

Copyright ©2012 Christos Kloukinas  School of Informatics 52/74

Treating Events

- ▶ R-T systems are for responding to internal and external events
- ▶ Some events can be treated by periodic threads
- ▶ Every N sec we check if an event occurred and treat it
- ▶ Some events cannot wait – they need to be treated **asynchronously** as they occur
- ▶ RTSJ represents them with the **AsyncEvent** class
- ▶ These are treated by the **AsyncEventHandler** and the **BoundAsyncEventHandler** handlers
- ▶ RTSJ calls external events “happenings”

Copyright ©2012 Christos Kloukinas  School of Informatics 54/74

AsyncEvent

```
public class AsyncEvent {
    public AsyncEvent();
    // All throw IllegalArgumentException if null arg
    public void addHandler(AsyncEventHandler handler);
    public void removeHandler(AsyncEventHandler handler);
    // Does target handle AE?
    public boolean handledBy(AsyncEventHandler target);
    // Clear all handlers and set h as the sole handler.
    public void setHandler(AsyncEventHandler h);


    // Both throw UnknownHappeningException.
    public void bindTo(String happening);
    public void unBindTo(String happening);

    public ReleaseParameters createReleaseParameters();
    // Throws MITViolationException and
    // ArrivalTimeQueueOverflowException
    public void fire();
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 55/74

AsyncEvent & AsyncEventHandler

- ▶ Either subclass AEH or provide it with a Runnable
- ▶ One AE can be handled by many AEHs
- ▶ One AEH can handle more than one AE
- ▶ A handler may be called again while previous releases are still running
- ▶ By default all AEH are daemons
 - ▶ If you don't like this, "exorcise" it before attaching it to an AE
- ▶ An RTSJ program terminates when and only when
 - ▶ all non-daemon threads (either regular Java threads or real-time threads) are terminated, and
 - ▶ the fireCounts of all non-daemon Bound AEHs or non-daemon AEHs are zero and all releases are completed, and
 - ▶ there are no non-daemon Bound AEHs or AEHs attached to timers or async events associated with happenings.
- ▶ So non-daemon AEHs will not allow the program to terminate

Copyright ©2012 Christos Kloukinas  School of Informatics 57/74

Event Handlers vs Threads

- ▶ Threads are for long-running code – sleeping, waiting, etc.
- ▶ Handlers are for short and quick code – normally no sleeping, waiting
- ▶ Usually one internal JVM thread will be servicing many AEHs
- ▶ Don't overdo it with BAEHs – the cost is too high (and they may be slower to start with. ... – *measure!!!*)

Copyright ©2012 Christos Kloukinas  School of Informatics 59/74

AsyncEventHandler


```
public class AsyncEventHandler implements Schedulable {
    public AsyncEventHandler(SchedulingParameters scheduling,
        ReleaseParameters release, MemoryParameters memory,
        MemoryArea area, ProcessingGroupParameters group,
        boolean nonheap, Runnable logic);
    protected int getPendingFireCount();
    protected int getAndClearPendingFireCount();
    protected int getAndDecrementPendingFireCount();
    protected int getAndIncrementPendingFireCount();
    public void handleAsyncEvent();
    public final void run();
    public MemoryArea getMemoryArea();
}
```

- ▶ Extend AEH to redefine **get*PendingFireCount**

1 AE increases the **fireCount** of each associated AEH and causes the AEH's **run** to be called

2 AEH's **run** calls **handleAsyncEvent** repeatedly while the **fireCount** is greater than zero

3 **handleAsyncEvent** calls your **Runnable**'s **run** each time

Copyright ©2012 Christos Kloukinas  School of Informatics 56/74

BoundAsyncEventHandler

```
public class BoundAsyncEventHandler
    extends AsyncEventHandler
```

- ▶ An AEH that is permanently bound to a dedicated real-time thread (provided by the JVM)
- ▶ BAEH are for situations where the added timeliness is worth the overhead of dedicating an individual real-time thread to the handler.
- ▶ Individual server real-time threads can only be dedicated to a single bound event handler.

Copyright ©2012 Christos Kloukinas  School of Informatics 58/74

Time-Related Events

- ▶ Sometimes instead of an action we need to respond to a time event
- ▶ Examples: Alarm, Birthday, Anniversary, Watchdog, etc.
- ▶ RTSJ provides support for such time-triggered events through Timers
- ▶ Timer is an abstract class
- ▶ There are two concrete sub-classes: **OneShotTimer** & **PeriodicTimer**

Copyright ©2012 Christos Kloukinas  School of Informatics 60/74

Timer API

```
public abstract class Timer extends AsyncEvent {
    protected      Timer(HighResolutionTime time,
                        Clock clock,
                        AsyncEventHandler handler);

    void    setHandler(AsyncEventHandler handler);
    void    addHandler(AsyncEventHandler handler);
    void    removeHandler(AsyncEventHandler handler);
    boolean handledBy(AsyncEventHandler handler);

    void    start();
    void    start(boolean disabled);
    boolean stop(); // stops if target's a RelativeTime
    void    disable(); // Doesn't fire
    void    enable();
    void    destroy(); // stop+disable+removeHandlers
    boolean isRunning();
```

Copyright ©2012 Christos Kloukinas  School of Informatics 61/74

A Watchdog Timer (Dibble, p. 170)

```
public class Dog {
    static final int TIMEOUT=2000; // 2 seconds
    public static void main(String [] args){
        double d; long n;
        AsyncEventHandler handler = new AsyncEventHandler() {
            public void handleAsyncEvent(){
                System.err.println("Emergency_reset!!!");
                System.exit(1); } };
        RelativeTime timeout = new RelativeTime(TIMEOUT, 0);
        OneShotTimer dog = new OneShotTimer(
            timeout, /* Watchdog interval */ handler);
        dog.start();
        while(true){
            d= java.lang.Math.random(); n= (long) (d*TIMEOUT +400);
            System.out.println("Running_t=" + n);
            try { Thread.sleep(n); } catch(Exception e){} // XXXX
            dog.reschedule(timeout);
        }
    } // XXXX: Why the sleep in there?
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 63/74

Dibble on Masking Timers

- ▶ Choices:
 1. Disable the timer.
Events are lost
 2. Set a variable to tell the timer whether to handle events or ignore them (but still count them).
Can also treat them later on (events are still counted)
 3. Hold a lock in the AEH (thus blocking it!!!).
You don't treat interrupts now but treat them later when you unblock
- ▶ Third choice is functionally similar to the second one but uses existing JVM mechanisms to achieve the goal.

Copyright ©2012 Christos Kloukinas  School of Informatics 65/74

Timer API – II

```
Clock    getClock();
AbsoluteTime    getFireTime();
AbsoluteTime    getFireTime(AbsoluteTime dest);

};
public class OneShotTimer / PeriodicTimer
    extends Timer;
public PeriodicTimer(HighResolutionTime phase,
                    RelativeTime interval,
                    AsyncEventHandler handler);
```

NOTE – The first release time is:

- ▶ If **phase** is a **RelativeTime** then
SUM(Time you invoked the **start()** method , **phase**)
- ▶ If **phase** is an **AbsoluteTime** then
MAX(Time you invoked the **start()** method , **phase**)

Copyright ©2012 Christos Kloukinas  School of Informatics 62/74

A OneShot Timer (Dibble, p. 172)

```
import javax.realtime.*;
public class OSTimer {
    static volatile /*XXXX*/ boolean stopLooping = false;
    public static void main(String [] args){
        AsyncEventHandler handler = new AsyncEventHandler() {
            public void handleAsyncEvent(){ stopLooping = true; }
        };
        OneShotTimer timer = new OneShotTimer(
            new RelativeTime(10000, 0), // Fire in 10 seconds
            handler);
        timer.start();
        while(!stopLooping){ //non-volatile could make this true
            System.out.println("Running");
            try { Thread.sleep(1000); } catch(Exception e){}
        }
        System.exit(0);
    }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 64/74


Empty

Slide for hire!

Copyright ©2012 Christos Kloukinas  School of Informatics 66/74

Periodic Timer with Masking (Dibble, p. 173–174)

```
public class PTimer2 extends RealtimeThread {
    PTimer2(){ super(null, null, null, ImmortalMemory.instance
        null, null); }
    public static void main(String [] args){
        PTimer2 rt = new PTimer2();
        rt.start();
        try{rt.join();} catch(Exception e){}
        System.exit(0);
    }
    public static class MaskableAEH extends AsyncEventHandler
    public Object mask = new Object();
    public void maskedAEH(){ }
    public void handleAsyncEvent(){
        synchronized (mask) { maskedAEH(); }
    }
}

public void run(){
    final int MIN_MS = 1500; // Lower limit for random times
    final int MAX_MS = 5000; // Upper limit for random times
    boolean maskIt = false; int timeAccum = 0;
    Copyright ©2012 Christos Kloukinas  School of Informatics 67/74
```

What we've covered

- ▶ Time values and clocks
- ▶ System (R-T) info
- ▶ POSIX signals
- ▶ Schedulable – the new Runnable
- ▶ ReleaseParameters: PeriodicParameters, AperiodicParameters, SporadicParameters
 - ▶ Violations: arrivalTimeQueueOverflow..., mitViolation...
- ▶ SchedulingParameters: PriorityParameters, ImportanceParameters
- ▶ Scheduler: PriorityScheduler
- ▶ Priority Inversion and Priority Inheritance/Ceiling


Copyright ©2012 Christos Kloukinas  School of Informatics 69/74

What we haven't covered

- ▶ **ProcessingGroupParameters** – used for grouping non-periodic threads (*whatever...*)
- ▶ **WaitFreeReadQueue**, **WaitFreeWriteQueue** (:- ()
- ▶ The *real* story about memory areas... (:- (()
- ▶ Asynchronous transfer of control (:- ((()
- ▶ Physical memory access (:-))

Copyright ©2012 Christos Kloukinas  School of Informatics 71/74

Periodic Timer with Masking – II

```
MaskableAEH handler = new MaskableAEH() {
    public void maskedAEH(){ System.out.println("tick"); }
};
PeriodicTimer timer = new PeriodicTimer(
    null, // Start now
    new RelativeTime(1000, 0), // Tick every 1 second
    handler);
timer.start();
while(timeAccum < 30000) { // Run for 30 seconds
    double d = java.lang.Math.random();
    int msDelay = (int)(d * (MAX_MS - MIN_MS) + MIN_MS);
    timeAccum += msDelay;
    if(maskIt) {
        synchronized (handler.mask) {
            try { Thread.sleep(msDelay); } catch(Exception e){ }
        } else
            try { Thread.sleep(msDelay); } catch(Exception e){ }
        maskIt = ! maskIt;
    }
    timer.removeHandler(handler);
}
Copyright ©2012 Christos Kloukinas  School of Informatics 68/74
```

What we've covered – II

- ▶ The GC problem
- ▶ The MemoryArea solution
- ▶ Types of memory areas: Immortal/Heap/Scoped
- ▶ Objects in one cannot always reference objects in another
 - ▶ So as to avoid dangling pointers by construction
- ▶ MemoryParameters
- ▶ R-T Threads: RealtimeThread, NoHeapRealtimeThread
- ▶ Catching Deadlines
- ▶ RTT and Aperiodic/Sporadic Threads
- ▶ Wellings' AperiodicThread
- ▶ Treating Events: AsyncEvent & (Bound)AsyncEventHandler
- ▶ Timer events: OneShotTimer & PeriodicTimer
- ▶ Masking timers

Copyright ©2012 Christos Kloukinas  School of Informatics 70/74

Study Material

- ▶ R-T Specification for Java: www.rtsj.org
RTSJ Spec version 1.0.2 http://www.rtsj.org/specjavadoc/book_index.html
Next version will be 1.1 – the group for it has been formed already and the proposal is under JSR 282:
 - ▶ JSR-282 <http://jcp.org/en/jsr/detail?id=282>
 - ▶ Dibble and Wellings “JSR-282 status report”
<http://dx.doi.org/10.1145/1620405.1620431>
 - ▶ “JSR-282 Early Draft Review version 6”: https://edelivery.oracle.com/otn-pub/jcp/rtsj_1.1-6-edr-oth-JSpec/rtsj-6-edr-spec.pdf
- ▶ Dibble “Real-Time Java Platform Programming”
- ▶ Wellings “Concurrent and Real-Time Programming in Java”
- ▶ Bruno & Bollella “Real-Time Java Programming with Java RTS”

Copyright ©2012 Christos Kloukinas  School of Informatics 72/74

Study Material – Extras

- ▶ Javolution <http://javolution.org/>
 - ▶ High performance and time-deterministic (real-time) util / lang / text / io / xml base classes.
 - ▶ Context programming in order to achieve true separation of concerns (logging, performance, etc).
 - ▶ A testing framework addressing not only unit tests but also performance and regression tests as well.
 - ▶ Straightforward and low-level parallel computing capabilities with ConcurrentContext.
 - ▶ Struct and Union base classes for direct interfacing with native applications (e.g. C/C++).
 - ▶ World's fastest and first hard real-time XML marshalling/unmarshalling facility.
 - ▶ Simple yet flexible configuration management of your application.
- ▶ Haven't used it but it seems interesting.
- ▶ May help you get more predictable behaviour from your programs

Study Material – Extras – II

- ▶ Safety Critical Java
 - ▶ <http://jcp.org/en/jsr/detail?id=302>
 - ▶ Zhao et al. "A Technology Compatibility Kit for Safety Critical Java" <http://www.cs.purdue.edu/homes/jv/pubs/jtres09b.pdf>
 - ▶ Henties et al. "Java for Safety-Critical Applications"
 - ▶ Open Safety-Critical Java
<http://code.google.com/p/scj-jsr302/>
 - ▶ Atego Research and Development – Safety Critical Java Specification Initiative (an earlier attempt):
<http://research.aonix.com/jsc/>

Java for Real-Time

Part 4 – Using RTSJ Part II

Christos Kloukinas

School of Informatics
City University London

Copyright ©2012 Christos Kloukinas  School of Informatics 1/31

RTSJ Seven Enhanced Areas

1. Thread Scheduling and Dispatching
 2. Memory Management
 3. Synchronization and Resource Sharing
 4. Asynchronous Event Handling
 5. **Asynchronous Transfer of Control**
 6. **Asynchronous Thread Termination**
 7. Physical Memory Access
- 7 bis *Time values and clocks*

Copyright ©2012 Christos Kloukinas  School of Informatics 3/31

Scenarios for ATC

- ▶ A computation can produce an ever more precise result if left running more time – stopping it when no more time is available or precision is good enough
- ▶ Calling a blocking method, e.g., open a file – ensure that you'll not block for more than X sec
- ▶ Having a thread that has gone out of control – another thread wants to stop it
- ▶ A deadline miss handler wants to change the logic of a thread

Copyright ©2012 Christos Kloukinas  School of Informatics 5/31

Contents

Interruptions

ATC – Some Scenarios For It
ATC Design Goals
ATC API
Using ATC
Timed
Asynchronous Termination

Wait-Free Queues

Conclusions

What we've covered
What we haven't covered
Study Material

Copyright ©2012 Christos Kloukinas  School of Informatics 2/31

Interruptions – Reminder

- ▶ Java threads support **interrupt()** (which they can ignore)
- ▶ They also supported **stop()** for stopping abruptly (has been deprecated)
- ▶ What's the general problem these are supposed to solve?
- ▶ Thread is working on X when Y happens
- ▶ Now thread must stop working on X and go back to some previous state Z
- ▶ In C, this was done with **setjmp/longjmp**
- ▶ Powerful mechanism, not loved by all – go to sleep on Mars, wake up back on Earth (a bit too drastic for some)
- ▶ RTSJ introduced Asynchronous Transfer of Control (ATC) through the **AsynchroneouslyInterruptedException** (AIE)


Copyright ©2012 Christos Kloukinas  School of Informatics 4/31

Some Design Goals for ATC

- ▶ “Protect the innocent” – unlike **stop()**, ATC requires cooperation
If a method does not throw AIE then it cannot be interrupted
- ▶ Synchronized blocks are not interrupted either
- ▶ Methods that throw AIE can be interrupted when they're not executing synchronised code
- ▶ A pending AIE is thrown immediately once it's safe to do so
- ▶ A pending AIE remains so until one calls its **clear** method or it executes its **Interruptible.run()**
- ▶ An AIE can be fired while another is pending. The one aiming at the least deeply nested method wins.

Dibble “The AIE was invented because some R-T programmers consider it crucial. Others find it repulsive. If you are in the latter class, don't use ATC. The RTSJ does not force you to learn to use ATC in order to use its other features.”

- ▶ It's nasty, it's brutal but it might prove just the tool one day

Copyright ©2012 Christos Kloukinas  School of Informatics 6/31

AIE API

```
public interface Interruptible {
    void run(AsynchronouslyInterruptedException e);
    void interruptAction(AsynchronouslyInterruptedException e)
    // what to do if run is interrupted
}

public class AsynchronouslyInterruptedException
    extends InterruptedException {
    AsynchronouslyInterruptedException();
    boolean doInterruptible(Interruptible logic);
    boolean clear(); // true if it was pending, otherwise false
    boolean fire(); // generate this exception if its
    // doInterruptible has been invoked and not completed.
    static AsynchronouslyInterruptedException getGeneric();
    // Singleton system generic AIE, generated when
    // RealtimeThread.interrupt() is invoked.
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 7/31

Using AIE – II

```
public static void main(String [] args) {
    final One rt1 = new One();
    RealtimeThread rt2 = new RealtimeThread() { // C starts
    public void run() {
        PriorityParameters oldPp = (PriorityParameters)
        getSchedulingParameters();
        PriorityParameters newPp = new PriorityParameters(
            oldPp.getPriority() + 1);
        try{setSchedulingParameters((SchedulingParameters)newPp);
        } catch (Exception e) { }
        try { Thread.sleep(1000);
            if (rt1.myaie.fire()) // rt2 fires rt1's AIE here!!!
                System.out.println("Fire_returned_true");
            else
                System.out.println("Fire_returned_false");
        } catch (InterruptedException e) { } } }; // C ends
    rt2.start(); rt1.start();
    try{rt1.join();rt2.join();}catch(InterruptedException e){}
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 9/31

AIE – Obfuscating the Picture...

- ▶ “An AIE can be fired while another is pending. The one aiming at the least deeply nested method wins.”
- ▶ Scenario: One AIE (X) is currently “in flight”. Another one (aie2) is fired. Which should the system treat? X or aie2?

Top of Thread Stack

aie3
aie2
aie1

- ▶ If X = aie3, then aie2 replaces it and gets treated instead
- ▶ If X = aie1, then X (aie1) wins and aie2 is ignored
- ▶ **The least deeply (i.e., furthest from the top of the stack) nested method wins**

Copyright ©2012 Christos Kloukinas  School of Informatics 11/31

Using AIE

```
public class One extends RealtimeThread {
    AsynchronouslyInterruptedException myaie;
    public void run() { //run A starts
        One.this.myaie=new AsynchronouslyInterruptedException();
        One.this.myaie.doInterruptible(
            new Interruptible() {
                private volatile int n;
                public void interruptAction(
                    AsynchronouslyInterruptedException aie) {
                    System.out.println("Interrupted_at_" + n); }
            }
        );
        public void run() { //run B starts
            AsynchronouslyInterruptedException aie)
            throws AsynchronouslyInterruptedException {
                for (int i = 0; i < 10000000; ++i) {
                    try{Thread.sleep(1);}catch(InterruptedException ie){}
                    n += 1;
                }
                System.out.println("Something's_wrong.");
            }
        } // run B ends
    }
} // run A ends
```

Copyright ©2012 Christos Kloukinas  School of Informatics 8/31

AIE – Clarifying the Picture

- ▶ Never throw an AIE with **throw new AIE();**
- ▶ This will not throw an Asynchronous exception but a synchronous one (to you)
- ▶ Instead, call its **fire()** method
- ▶ You can think of the AIE objects as establishing some rollback points
- ▶ You place them around in your code to have somewhere to land when things turn nasty
- ▶ Code runs the AIE's **doInterruptible()** method until something bad happens (Thread.interrupt() or aie.fire())
- ▶ Then the code unwinds the stack and starts running the aie's Interruptible **interruptAction()**
- ▶ *Not for the fainthearted...*
- ▶ *Need to study carefully Dibble's code examples (in atc/)*

Copyright ©2012 Christos Kloukinas  School of Informatics 10/31

A Special AIE – Timed

- ▶ Many cases require a task to run until a point in time or for a fixed duration then do something else if it's not finished
- ▶ RTSJ supports this through the **Timed** class

```
public class Timed
    extends AsynchronouslyInterruptedException {
    Timed(HighResolutionTime time);
    boolean doInterruptible(Interruptible logic);
    void resetTime(HighResolutionTime time);
}
```

- ▶ Use the **doInterruptible** to run the code in logic until **time** is no longer on your side
- ▶ Then you receive an AIE

Copyright ©2012 Christos Kloukinas  School of Informatics 12/31

Doing Time...

```
class AIE extends AsynchronouslyInterruptedException{}
public class TimedTest extends RealtimeThread {
    public void run() {
        Timed tm = new Timed(new RelativeTime(1000, 0));
        tm.doInterruptible(new Interruptible() {
            private volatile int n;
            public void interruptAction(AIE e) {
                System.out.println("Interrupted_at_" + n); }
            public void run(AIE e) throws AIE {
                System.out.println("Entering_Timed_run");
                while (n < 100000000) n += 1;
                System.out.println("Wow!_Finished_normally."); }
        });
    }
    public static void main(String [] args) {
        TimedTest rt = new TimedTest();
        rt.start();
        try {rt.join();} catch (InterruptedException e) {
            System.out.println("Interrupted_in_join"); }
    }
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 13/31

Too Easy!

- ▶ Not all **finally** clauses will be executed!!!
- ▶ So much for them being “always” executed (not even true in normal Java BTW - put a `System.exit(0)` inside the try to see...)
- ▶ Which **finally** clauses will be executed?
 - ▶ The ones inside methods that don't throw AIE
 - ▶ The ones inside synchronized methods/blocks
- ▶ **So, finally clauses that are inside a method that throws an AIE and are not inside a synchronized block will be ignored.**
- ▶ *Code carefully – very carefully...*

Copyright ©2012 Christos Kloukinas  School of Informatics 15/31


Wait-Free Queues

- ▶ Queues whose one end blocks but the other never blocks
- ▶ The non-blocking end is used by the NHRTs
- ▶ The blocking end is used by the RTTs/JTs
- ▶ **WaitFreeWriteQueue**: Writing is non-blocking
- ▶ **WaitFreeReadQueue**: Reading is non-blocking

Copyright ©2012 Christos Kloukinas  School of Informatics 17/31

Asynchronous Thread Termination

- ▶ Short story: The target thread must cooperate!
- ▶ If you call `somethread.interrupt()` it should do the job (if it cooperates):
 - ▶ If somethread was written to expect AIEs then it'll visit code that allows them
 - ▶ There the generic AIE created by `interrupt()` will be thrown
 - ▶ It'll not match any AIE on somethread's stack
 - ▶ So it'll propagate all the way through, running any **finally** and **catch** clauses and any **interruptAction** methods, until it reaches somethread's **run** method, where it'll terminate
- BUT!** If somethread never calls a method that can throw an AIE, never polls its interrupted flag, and never responds to exceptions from blocking calls that signify interrupts, then **NOTHING** happens when you try to interrupt it.
- ▶ Many legacy applications fall in this uncooperative class.
- ▶ So calling `interrupt()` on a thread might not do the job...

Copyright ©2012 Christos Kloukinas  School of Informatics 14/31

Reminder – Critical Region Length

- ▶ We need to keep critical regions short to minimise delayed event handlers
- ▶ **Event handlers usually come in pairs – short and long**
- ▶ Sending data over the network
- ▶ The short IH buffers messages and groups them into longer messages that are eventually sent by the long IH
- ▶ Makes sense to have the short part in a NHRT or an AIH and the long part in a normal Java thread (or RTT) that can also access the heap (and be stopped by the GC)
- ▶ How will these communicate?
- ▶ If the NHRT waits for the RTT then it might be delayed by the GC as well!
- ▶ Solution: Wait-Free Queues

Copyright ©2012 Christos Kloukinas  School of Informatics 16/31

WaitFreeWriteQueue

```
public class WaitFreeWriteQueue {
    WaitFreeWriteQueue(int maximum, MemoryArea memory);
    WaitFreeWriteQueue(int maximum); //in ImmortalMemory
    int size();
    void clear();
    boolean isEmpty();
    boolean isFull();
    Object read(); //synchronized, blocks if empty
    boolean write(Object object); //non-blocking
    boolean force(Object object); //overwrite last
}
```

- ▶ **force** returns true when the queue was full and it replaced the last entry
- ▶ If the queue is empty, it'll write the object and return false
- ▶ If the queue is neither full nor empty, it'll add the object and return false
- ▶ So false means it wrote into a vacant slot, true means it wrote into an occupied slot

Copyright ©2012 Christos Kloukinas  School of Informatics 18/31

WaitFreeReadQueue

```
public class WaitFreeReadQueue {
    WaitFreeReadQueue(int max, MemoryArea m, boolean notify);
    WaitFreeReadQueue(int max, boolean ntf); // in ImmortalMemory
    int    size();
    void    clear();
    boolean isEmpty();
    boolean isFull();
    void    write(Object obj); // synchronized, blocks if full
    Object  read(); // non-blocking, null if empty
    void    waitForData(); // block if no data
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 19/31

Memory Issues

- ▶ There's the queue object (QO), the queue itself (Q), and the queue elements (EQs)
- ▶ Each of them can be in a different memory area (*trouble...*)
- ▶ If all of them in the ImmortalMemory then we put a bit of a strain on the immortal memory (*not very good*)
- ▶ If all of them in scoped memory then they're freed once all threads leave that scope

Copyright ©2012 Christos Kloukinas  School of Informatics 21/31

Memory Issues – Wellings – II

- 2 If a wait-free queue is being used to communicate between a heap-using SO and a no-heap using SO, then:
 - ▶ The queue can be in immortal or scoped memory
 - ▶ Any objects passed must be in immortal or scoped memory

Clueless guy (yours truly):

So maybe always have a no-heap using SO communicating with a heap-using SO through a WF queue in scoped memory?

If a conventional Java thread needs to communicate it can pass through the heap-using SO (not necessarily through another WF queue).

(*maniacal laughter heard in the distance...*)

Copyright ©2012 Christos Kloukinas  School of Informatics 23/31

Patterns for Wait-Free Queues (Dibble, p. 283–284)

```
WaitFreeReadQueue(QueueSize,
    RealtimeThread.getCurrentMemoryArea(),
    false);
WaitFreeWriteQueue(QueueSize,
    RealtimeThread.getCurrentMemoryArea());
```

- ▶ The default area (ImmortalMemory) cannot hold queue elements allocated in scoped memory
- ▶ Wait-Free queues can be shared safely at their blocking end
 - ▶ Many readers for a WaitFreeWriteQueue
 - ▶ Many writers for a WaitFreeReadQueue

Copyright ©2012 Christos Kloukinas  School of Informatics 20/31

Memory Issues – Wellings

- 1 If a wait-free queue is being used to communicate between a no-heap schedulable object and a conventional Java thread, then:

The queue must be in immortal memory as:

- ▶ A Java thread cannot enter scoped memory
- ▶ Immortal memory cannot point to scoped memory
- ▶ A no-heap thread cannot access the heap

Any objects passed must be in immortal memory, otherwise an exception will be thrown if:

- ▶ A writer Java thread passes a heap object – the reader no-heap SO will attempt to access the heap
- ▶ A writer no-heap SO passes a scoped memory object – the reader Java thread will try to store the reference of that scoped memory object on the heap.

Copyright ©2012 Christos Kloukinas  School of Informatics 22/31

Recycling Memory (Dibble, p. 287 – Algorithm 21–1)

```
// Object recycling using two queues - q and r.
// Create a wait-free queue q, and n objects
// of type T to go in it
// 1-Construct the wait-free queue q as required
// 2-Construct the recycling queue, r. An n-entry wait-free
// queue with its wait on the end opposite to q's.
// 3-Initialise queue r with n objects of type T.
for(i = 0; i < n; ++i)
    r.write(new T());

// 4-Use the recycling queue after every q.read
obj = q.read();
// copy obj locally
r.write(obj); // make obj available for q

// 5-And before every q.write
obj = r.read(); // What if r returns null?
// copy data into obj
q.write(obj);
```

Copyright ©2012 Christos Kloukinas  School of Informatics 24/31

Recycling Memory – WaitFreeReadQueue

(Dibble maketh, Clueless Guy breaketh... :-)

```
// 1-3- NHRT side
myMem = RealtimeThread.getCurrentMemoryArea();
q = new WaitFreeReadQueue(n, myMem, false);
r = new WaitFreeWriteQueue(n, myMem);
for(i = 0; i < n; ++i) r.write(new T());
// 4- NHRT side
obj = q.read();
if (obj) {
    // copy obj locally
    r.write(obj); // make obj available for q
} else {} // nothing to do if no data
// 5- RTT side
obj = r.read(); // may block
// copy data into obj
q.write(obj);
```

Copyright ©2012 Christos Kloukinas  School of Informatics 25/31

What we've covered

- ▶ Asynchronous transfer of control
- ▶ Timed computations
- ▶ Asynchronous thread termination
- ▶ Clause finally not always executed
- ▶ **WaitFreeReadQueue, WaitFreeWriteQueue**

Copyright ©2012 Christos Kloukinas  School of Informatics 27/31

Study Material

- ▶ R-T Specification for Java: www.rtsj.org
RTSJ Spec version 1.0.2 http://www.rtsj.org/specjavadoc/book_index.html
Next version will be 1.1 – the group for it has been formed already and the proposal is under JSR 282:
 - ▶ JSR-282 <http://jcp.org/en/jsr/detail?id=282>
 - ▶ Dibble and Wellings “JSR-282 status report”
<http://dx.doi.org/10.1145/1620405.1620431>
 - ▶ “JSR-282 Early Draft Review version 6”: https://edelivery.oracle.com/otn-pub/jcp/rtsj_1.1-6-edr-oth-JSpec/rtsj-6-edr-spec.pdf
- ▶ Dibble “Real-Time Java Platform Programming”
- ▶ Wellings “Concurrent and Real-Time Programming in Java”
- ▶ Bruno & Bollella “Real-Time Java Programming with Java RTS”

Copyright ©2012 Christos Kloukinas  School of Informatics 29/31

Recycling Memory – WaitFreeWriteQueue

```
// 1-3- NHRT side
myMem = RealtimeThread.getCurrentMemoryArea();
q = new WaitFreeWriteQueue(n, myMem);
r = new WaitFreeReadQueue(n, myMem, false);
lastobj=new T(); r.write(lastobj);
for(i = 1; i < n; ++i) r.write(new T());
// 4- RTT side
obj = q.read(); // may block
// copy obj locally
r.write(obj); // make obj available for q
// 5- NHRT side
obj = r.read();
if (obj) {
    // copy data into obj
    q.write(obj); lastobj = obj;
} else { // r is empty, q is full
    // copy data into lastobj
    q.force(lastobj); // superfluous? Don't blame Dibble...
}
```

Copyright ©2012 Christos Kloukinas  School of Informatics 26/31

What we haven't covered

- ▶ The *real* story about memory areas... (:- (()

Copyright ©2012 Christos Kloukinas  School of Informatics 28/31

Study Material – Extras

- ▶ RTSJ implementations (<http://rtjava.blogspot.co.uk/2009/07/real-time-java-vms.html>)
Closed JamaicaVM, Aonix PERC, aJ100, JRockit Real-Time, IBM/Apogee Aphelion FijiVM
Open OVM, jRate, LJRT
- ▶ <http://www.ovmj.net/oscj/> oSCJ is an open-source prototype implementation of a restricted subset of Safety-Critical Java developed at Purdue University. This project is originating from the JSR-302.
<http://jcp.org/en/jsr/detail?id=302>
- ▶ OVM <http://www.ovmj.net/> implements RTSJ, used for oSCJ too
- ▶ jRate <http://jrate.sourceforge.net/> RTSJ through gcj
- ▶ LJRT <http://www.robot.lth.se/java/>

Copyright ©2012 Christos Kloukinas  School of Informatics 30/31

Miscellaneous

- ▶ To use Linux, add the `rt-linux` package
- ▶ Also edit `/etc/security/limits.conf` to add:

```
@realtime    soft    cpu      unlimited
@realtime    -       rtprio   100
@realtime    -       nice     40
@realtime    -       memlock  unlimited
```
- ▶ Add your user(s) to the `realtime` group
- ▶ Install some RTSJ implementation (and then show us how it's done... :- ()