Programming Models for Concurrency and Real-time

Jan Vitek

based on joint work with
David Bacon and Josh Auerbach (IBM Research)
Jesper Spring and Rachid Guerraoui (EPFL)
Jason Baker, Toni Cunei, Tomas Kabilera, Filip Pizlo, Marek Prochazka (Purdue)

Personal history

- PhD on the Seal calculus (mobile processes)
- Started on Real-time Java in 2001, in DARPA project. At the time, no real RTSJ implementation
- Developed the Ovm virtual machine framework, a clean-room, open source RT Java virtual machine
- Fall 2005, first flight test with Java on a plane
- Collaboration with IBM Research
- Fiji systems LLC

Duke's Choice Award

Technical Aims of the Course

- To understand the basic requirements of concurrent and real-time systems
- To understand how these requirements have influenced the design of Java and the Real-Time Specification for Java
- To be able to program advanced concurrent real-time Java systems

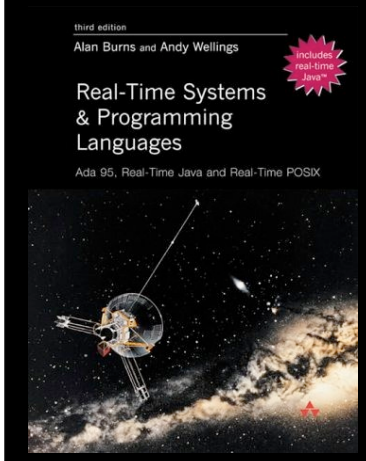
Course Contents

- Introduction to real-time processing
- Concurrent Programming in Java
- Overview of the Real-time Specification for Java
  - Memory Management / Clocks and Time / Scheduling and Schedulable Objects / Asynchronous Events and Handlers / Real-Time Threads / Asynchronous Transfer of Control / Resource Control / Schedulability Analysis
- Advanced Topics
  - Real-time garbage collection
  - Simple and flexible real-time programming with Flexotasks
  - Safety Critical Java

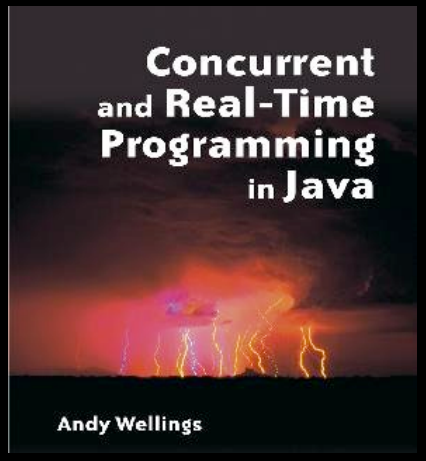
Introduction to Concurrent and Real-time Processing

Selected slides © Andy Wellings

Books

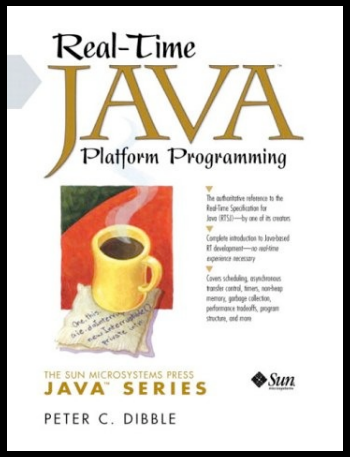


RTSJ Version 0.9

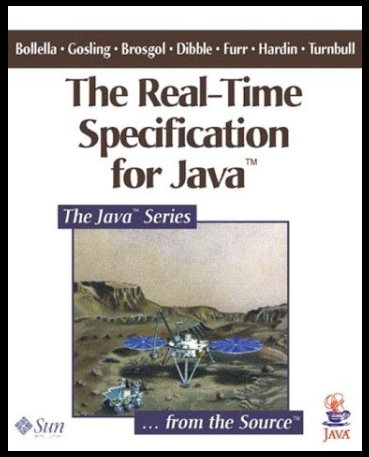


RTSJ Version 1.0.1

Other books



RTSJ Version 1.0

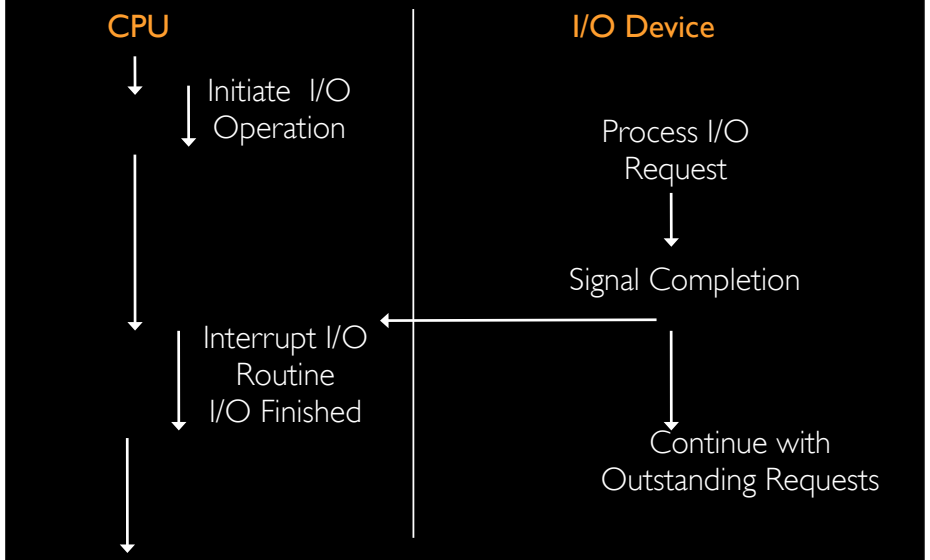


RTSJ Version 0.9

Concurrent Programming

- The name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems
- Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming
- Concurrent programming is important because it provides an abstract setting in which to study parallelism without getting bogged down in the implementation details
- A concurrent programming model is a set of abstractions exposed by the programming environment to specify and control concurrent activities

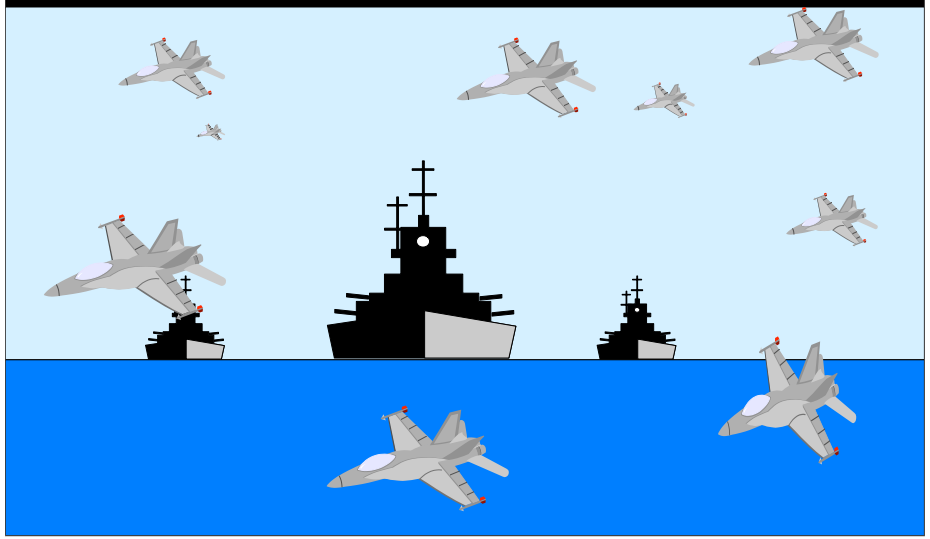
Parallelism Between CPU and I/O Devices



Why we need it

- To fully utilize computing resources
- To allow the expression of potential parallelism so that more than one computer can be used to solve the problem
- To model the parallelism in the real world
- Virtually all real-time systems are inherently concurrent — devices operate in parallel in the real world
- This is, perhaps, the main reason to use concurrency

Air Traffic Control



Why we need it

- Alternative: use sequential programming techniques
- We must construct the system as the cyclic execution of a program sequence to handle the various concurrent activities
- This complicates the task and involves considerations of structures irrelevant to the control of the activities in hand
- The resulting programs will be more obscure and inelegant
- Decomposition of the problem is more complex
- Parallel execution on >1 processors is more difficult to achieve
- The placement of code to deal with faults is more problematic

Terminology

- A concurrent program is a collection of autonomous sequential processes, executing (logically) in parallel
- Each process has a single thread of control
- The actual implementation (i.e. execution) of a collection of processes usually takes one of three forms.
  - Multiprogramming: processes multiplex execution on single processor
  - Multiprocessing: processes multiplex their executions on a multiprocessor system where there is access to shared memory
  - Distributed Processing: processes multiplex their executions on several processors which do not share memory

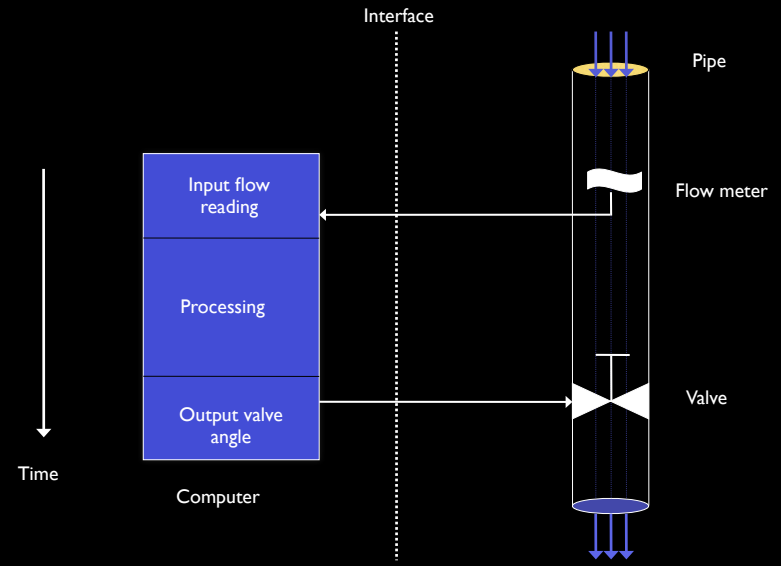
What is a real-time system?

- A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period
  - *correctness depends not only on logical result but also time it is delivered*
  - *failure to respond is as bad as the wrong response!*
- The computer is a component in a larger engineering system => **Embedded Computer System**
- 99% of all processors are for the embedded systems market

Terminology

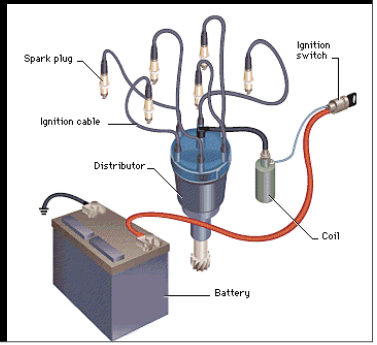
- **Hard real-time** — systems where it is absolutely imperative that responses occur within the required deadline. E.g. Flight control systems.
- **Soft real-time** — systems where deadlines are important but which will still function correctly if deadlines are occasionally missed. E.g. Data acquisition system.
- **Firm real-time** — systems which are soft real-time but in which there is no benefit from late delivery of service.
- A system may have all hard, soft and real real-time subsystems. Many systems may have a cost function associated with missing each deadline

A simple fluid control system

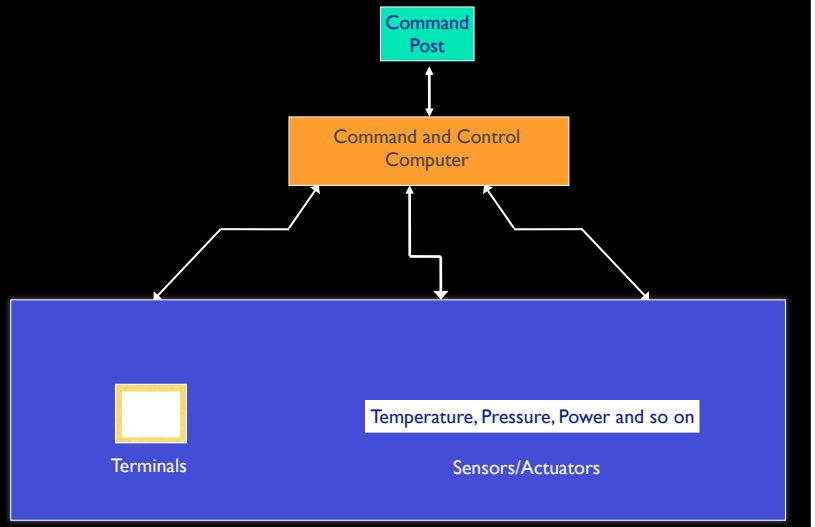


Distributor in Ignition System

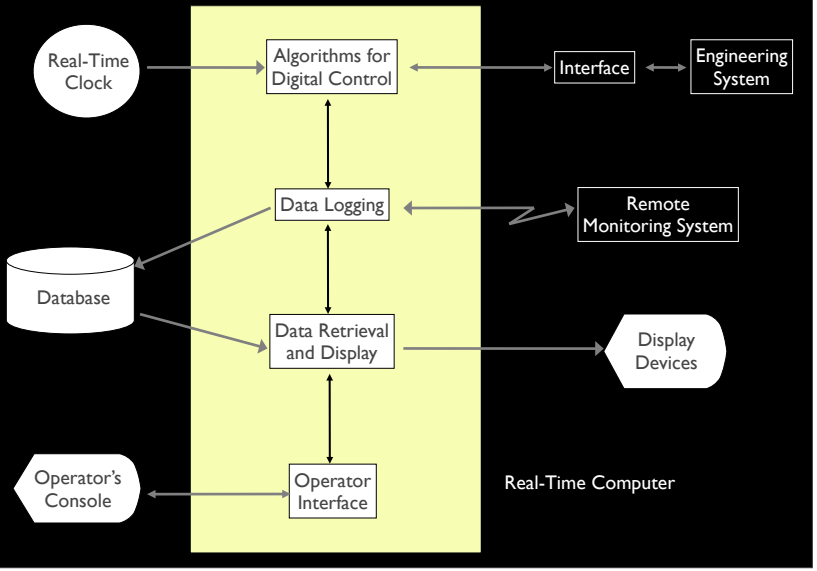
- Mechanical Distributor
  - Advance timing as function of load and speed
    - Vacuum advance—vacuum from intake manifold draws against a diaphragm under heavy loads
    - Centrifugal advance — weights swing out as engine speed increases
- Analog Computer
  - Same control laws as mechanical system
- Digital Computer
  - Equations
  - Look-up tables with interpolation
  - Digital filtering



A Command and Control System



A Typical Embedded System



Relevance of Embedded Systems

- "[…] estimate that each individual […] may unknowingly use more than 100 embedded computers daily"
- "The world market for embedded software will grow from about $1.6 billion in 2004 to $3.5 billion by 2009, at an average annual growth rate (AAGR) of 16%."
- "Embedded hardware growth will be at the aggregate rate of 14.2% to reach $78.7 billion in 2009, while embedded board revenues will increase by an aggregate rate of 10%"

http://www.bccresearch.com/comm/G229R.html, http://www.ecpe.vt.edu/news/ar03/embedded.html

Characteristics of a RTS

- Large and complex — vary from a few hundred lines of assembler or C to 20 million lines of Ada estimated for the Space Station Freedom
- Concurrent control of separate system components — devices operate in parallel in the real-world; better to model this parallelism by concurrent entities in the program
- Facilities to interact with special purpose hardware — need to be able to program devices in a reliable and abstract way

Characteristics of a RTS

- Extreme reliability and safe — embedded systems typically control the environment in which they operate; failure to control can result in loss of life, damage to environment or economic loss
- Guaranteed response times — we need to be able to predict with confidence the worst case response times for systems; efficiency is important but predictability is essential

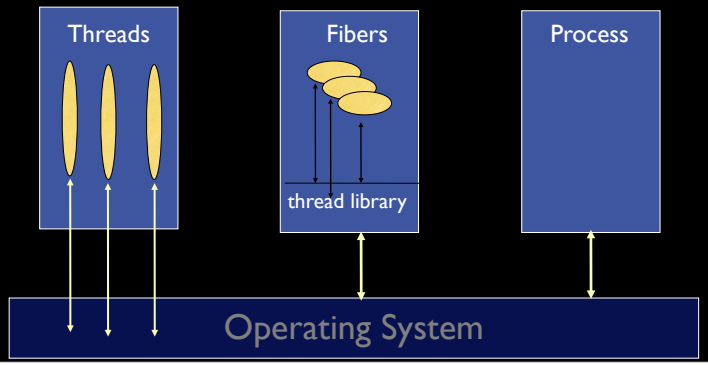
Real-time Programming Languages

- Assembly languages
- Sequential systems implementation languages
  - e.g. RTL/2, Coral 66, Jovial, C, C++.
- High-level concurrent languages. Impetus from the software crisis.
  - e.g. Ada, Chill, Modula-2, Mesa, Esterel, Java.
- We will focus on Java and the Real-Time Specification for Java
- See Burns, Wellings, *Real-Time Systems and Programming Languages*, for a discussion on languages and operating systems

Concurrent Programming in Java

Concurrency Models I

- Heavyweight tasks execute in their own address space
- Lightweight tasks run in the same address space
- A task is disjoint if it does not communicate with or affect the execution of any other task



Concurrency Models II

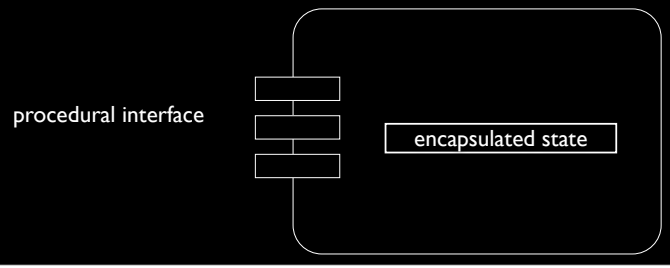
- Java supports threads
  - Threads execute within a single JVM
  - **Native threads** map a single Java thread to an OS thread
  - **Green threads** adopt the thread library approach
  - **M-on-N** threads are a mixture of the above (M green threads scheduled on N native threads)
  - On a multiprocessor, native threads are needed to get true parallelism

Concurrency Models III

- There are various ways in which concurrency can be introduced
  - API for explicit thread creation or thread forking
  - high-level language construct, e.g. PAR (occam), tasks (Ada), or processes (Modula)
- Integration with object-oriented programming, various models:
  - asynchronous method calls
  - early return from methods
  - futures
  - active objects
- Java adopts the active object approach

Concurrency Models IV

- Communication and Synchronization
  - approaches broadly classified as shared-variable or message passing
  - many different models, a popular one is a monitor
  - a monitor can be considered as an object where each of its operation executes in **mutual exclusion**



Concurrency Models V

- **Condition Synchronization**
  - expresses a constraint on the ordering of execution of operations, e.g., data cannot be taken from a buffer until data has been put in it
- Monitors provide condition variables with three operations which can be called when the lock is held
  - `wait`: an unconditional suspension of the calling thread (the thread is placed on a queue associated with the condition variable)
  - `notify`: one thread is taken from the queue and re-scheduled for execution (it must reclaim the lock first)
  - `notifyAll`: all suspended threads are re-scheduled
  - `notify` and `notifyAll` have no effect if no threads are suspended on the condition variable

Kinds of Synchronization

- **Cooperative**
  - Task A must wait for task B to complete some activity before A can continue executing, e.g., producer-consumer
  - `wait/notify`
- **Competitive**
  - Two or more tasks must use a resource that cannot be simultaneously accessed, e.g., a shared counter
  - `synchronized`

Liveness and Deadlock

- **Liveness** is a characteristic that a program may or may not have
  - In sequential code, it means the program will eventually complete
  - In a concurrent environment, a task can easily lose its liveness
  - If all tasks lose their liveness, it is called **deadlock**

Concurrency in the real world...

- Concurrency is a also a source of problems
  - Windows 2000: Concurrency errors most common defects among detectable errors
  - Windows 2000: Incorrect synchronization and protocol errors most common coding errors
  - Windows 2003: Synchronization errors second only to buffer overruns
  - Race conditions create security vulnerabilities
  - Concurrent programs are hard to test because they are non-deterministic
    - *Bugs are hard to reproduce because there are exponentially many possible interleavings that often only manifest on deployed*

Data Races

- A *race condition* occurs if two threads access a shared variable at the same time and at least one of the accesses is a write
- Consider the following program when multiple threads are call `deposit()` in parallel:

```
class Account {
  private int bal;

  void deposit(int n) {
     int j = bal;
     bal = j + n;
  }
}
```


Lock-based Synchronization

- Monitors must be used to protect every shared location access

```
a = x.a; b = x.b;  ||  x.a = 1; x.b = 2;
```

- Locks must be held before every read/write to a shared location

```
synchronized(x) {a = x.a; b = x.b;}
||
synchronized(x) {x.a
```

- When multiple locks are used, a lock acquisition protocol must be adhered to by the application to avoid deadlocks

```
synchronized(x){ synchronized(y) {...}}
||
synchronized(y){ synchronized(x) {...}}
```


Puzzlers #1

- How many different values can there be for local variables a and b?

```
a = x.a; b = x.b;   ||   x.a = 1; x.b = 2;
```


Puzzlers #2

- Do the following programs have data races? Are they non-deterministic?

```
long a = x.a;        ||   long b = x.a;
```

```
x.i=1;int i=x.i;     ||   x.i=1;int i=x.i;
```

```
                x.a = MAX_LONG
x.a = 0         ||  long a = x.a;  x.a = (a==MAX_LONG)? 0 : a;
```

Puzzlers #3

- Does the following program have data race?

```
interface INC { void inc(); }

class Int implements INC { int i;  void inc() { i++; } }

class SyncInt implements INC {
   INC val;
   SyncInt(INC v){ val = v; }
   synchronized inc() { val.inc(); }
}

...
INC i = new Int();
INC[] arr=new INC[]{new SyncInt(i),new SyncInt(i),new SyncInt(i)};



arr[0].inc();   ||   arr[1].inc();   ||   arr[2].inc();
```


Puzzlers #3

- Does the following program have concurrency problem?

```
class LL {
    int i;
    LL next;
    LL(LL n,int v){next=n;i=v;}
    synchronized swap() {
       synchronized(next) { int t=next.i; next.i= i; i=t; }
    }
}


   LL a=new LL(null,0), b=new LL(a,1); a.next=b;

        a.swap();   ||  b.swap();
```


Puzzlers #3

- A fix?

```
class LL {
    static int K;
    private final int id = K++;
    int i;
    LL next;
    LL(LL n,int v){next=n;i=v;}
    private void _swap() {int t=next.i; next.i= i; i=t;}
    void swap() {
       if (this.id > next.id)
         synchronized (this) {synchronized(next) { _swap(); }
       else
         synchronized (next) {synchronized(this) { _swap(); }
    }
}
```


Puzzlers #3

- A fix?

```
class LL {
    static int K;
    private final int id;
    int i;LL next;


    LL(LL n,int v){
       next=n;i=v;
       synchronized (LL.class) { id = K++; }
    }

    private void _swap() {int t=next.i; next.i= i; i=t;}
    void swap() {
       if (this.id > next.id)
         synchronized (this) {synchronized(next) { _swap(); }
       else
         synchronized (next) {synchronized(this) { _swap(); }
    }
```

Puzzlers #4

- The following idioms occurs frequently in library code. Why is it?

```
final class SyncTree {

   Node left, right;

   private final Object lock = new Object();

   public balance() {

       synchronized (lock) {

           ....
```


Puzzlers #5

- Is this a correct solution to synchronization problems?

```
class Big {
    static final Object Lock = new Object();
}


class LL {
     int i;
     LL next;
     LL(LL n,int v){next=n;i=v;}
     void swap() {
        synchronized(Big.Lock) { int t=next.i; next.i= i; i=t; }
     }
 }



   LL a=new LL(null,0), b=new LL(a,1); a.next=b;

         a.swap();   ||  b.swap();
```


Puzzlers #6

- Does the following program have a data race? A concurrency problem?

```
class Big { static final Object Lock = new Object(); }


class LL {
     int i;  LL next; LL(LL n,int v){next=n;i=v;}


     void swap() {
       int t = 0;
       synchronized(Big.Lock) { t=next.i; }
       synchronized(Big.Lock) { next.i= i; }
       synchronized(Big.Lock) { i=t; }
     }
 }
```


Lessons

- To reason about concurrency one must understand all interleaving of operations performed by each thread
- Not all high-level commands are atomic
- Aliasing makes it hard to determine which values are shared, thus one may fail to acquire the right lock (or locks)
- Lock acquisition protocols must be followed to avoid deadlocks
- Library classes must protect themselves from clients by implementing their own synchronization
- Oversynchronization is always safe but decreases concurrency (possibly to the point of making it meaningless?)
- Definition of data race too low level to catch all errors

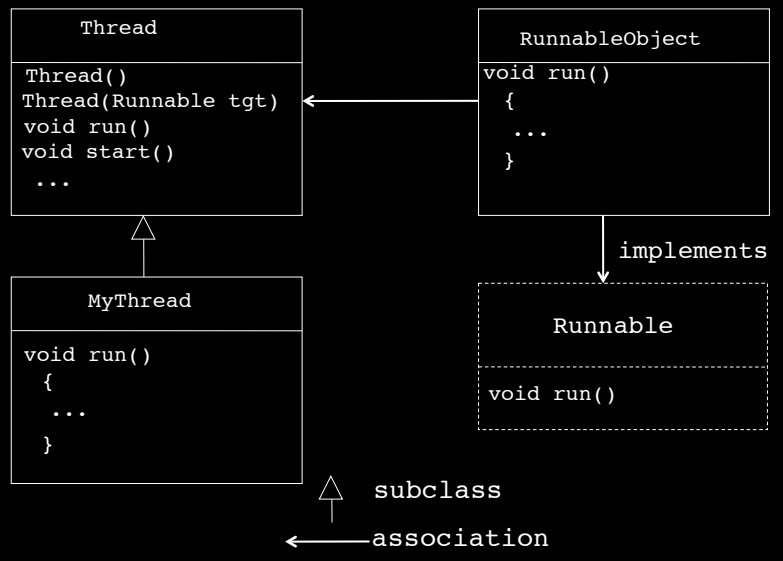
Concurrency in Java

- Java has a predefined class `java.lang.Thread` which provides the mechanism by which threads are created
- However to avoid all threads extending `Thread`, it also has a standard interface

```
public interface Runnable {

  public void run();

}
```

- Hence, a class which wishes to express concurrent execution implements this interface
- Threads do not begin their execution until `start` is called

Threads in Java



Communication in Java

- Via reading and writing to data encapsulated in shared objects protected by monitors
- Every object is implicitly derived from the `Object` class which defines a mutual exclusion lock
- Methods in a class can be labeled as `synchronized`, this means that they can only be executed if the lock can be acquired (this happens automatically)
- The lock can also be acquired via a synchronized statement which names the object
- A thread can `wait` and `notify` on a single anonymous condition variable

The Thread Class

```
public class Thread implements Runnable {
  public Thread();
  public Thread(String name);
  public Thread(Runnable target);
  public Thread(Runnable target, String name);
  public Thread(Runnable target, String name,
                long stackSize);
  public void run();
  public void start();
  ...
```

Thread Identification

- The identity of the currently running thread can be found using the `currentThread` method
- This has a static modifier, which means the method can always be called using the `Thread` class

```
public class Thread implements Runnable {
  ...
  public static Thread currentThread();
```


A Thread Terminates:

- when it completes execution of its `run` method either normally or as the result of an unhandled exception
- via a call to its `stop` method — the `run` method is stopped and the thread class cleans up before terminating the thread (releases locks and executes any finally clauses)
  - the thread object is now eligible for garbage collection.
  - `stop` is inherently unsafe as it releases locks on objects and can leave data in inconsistent states; (deprecated; should not be used)
- via a call to its `destroy` method — `destroy` terminates the thread without any cleanup (deprecated)

Daemon Threads

- Threads can be of two types: user threads or daemon threads
- Daemon threads provide general services and never terminate
- When all user threads have terminated, daemon threads will be terminated by the virtual machine on shutdown
- The `setDaemon` method must be called before calling `start`

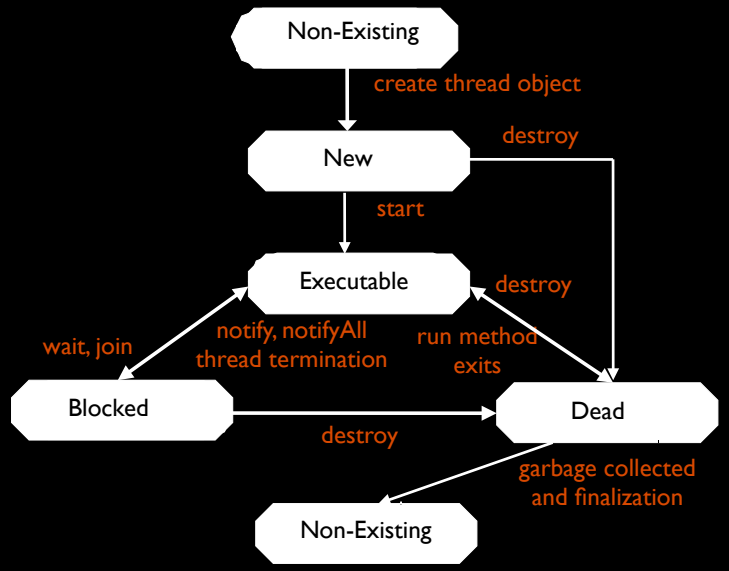
```
public class Thread implements Runnable {
  public void destroy();       // DEPRECATED
  public final boolean isDaemon();
  public final void setDaemon();
  public final void stop();  // DEPRECATED
```


Joining

- One thread can wait (with or without a timeout) for another thread (the target) to terminate by issuing the `join` method call on the target's thread object
- The `isAlive` method allows a thread to determine if the target thread has terminated

```
public class Thread implements Runnable {
  public final native boolean isAlive();
  public final void join() throws InterruptedException;
  public final void join(long ms) throws InterruptedExcept
  public final void join(long millis, int nanos) throws In
```

Java Thread States



Summary

- A thread is created when a **`Thread`** object is created
- At this point, the thread is not executable, it is in the `new` state
- Once `start` has been called, the thread is eligible for execution
- If the thread calls **`wait`** on an Object, or calls **`join`** on another thread object, the thread becomes blocked and no longer eligible
- It becomes executable if an associated **`notify`** is called by another thread, or if the target thread of the **`join`** is dead
- it enters the dead state, if the **`run`** method exits (normally or unhandled exception) or because **`destroy`** has been called. In the latter case, the thread will not execute any `finally` clauses; it may leave other objects locked

Synchronized Methods

- A mutual exclusion lock is associated with each object. It can't be accessed directly by the application but is affected by
  - the method modifier synchronized
  - block synchronization
- When a method is labeled as `synchronized`, the method can only execute once the system has the lock
- Hence, synchronized methods have mutually exclusive access to the data encapsulated by the object, if that data is only accessed by other synchronized methods
- Non-synchronized methods do not require the lock and, therefore, can be called at any time

Example of Synchronized Methods

```
class SharedInteger {
  private int data;
  SharedInteger(int val) { data = val; }
  synchronized int read() { return data;}
  synchronized void write(int val) { data = val; }
  synchronized void incrementBy(int by) { data += by; }
}

SharedInteger shi = new SharedInteger(42);
```

Block Synchronization

- A mechanism where a block can be labeled as synchronized
- The `synchronized` keyword takes as a parameter an object whose lock the system needs to obtain before it can continue
- Synchronized methods are effectively implementable as

```
public int read() {
  synchronized(this) {
      return theData;
  }
}
```

- `this` is the Java mechanism for obtaining the current object

Warning

- In its full generality, **synchronized blocks** can undermine one of the advantages of monitor-like mechanisms, that of encapsulating synchronization constraints associate with an object into a single place in the program
- This is because it isn't possible to understand the synchronization associated with an object by just looking at the object itself when other objects can name that object in a synchronized statement
- However with careful use, this facility allows more expressive synchronization constraints to be programmed

Accessing Synchronized Data

- Consider a simple class which implement a two-dimensional coordinate that is to be shared between two or more threads
- This class encapsulates two integers
- Writing is simple, the `write` method can be synchronized
- The constructor can be assumed not to have any synchronization

```
class Coord {
  Coord(int x,int x) { x_=x; y_=y; }
  synchronized void write(int x,int y) {x_=x; y_=y;}
  private int x_, y_;
}
```


Shared Coordinate

- How to read the value of the coordinates?
- Methods return a single value, parameters are passed by value
- Consequently, it is not possible to have a single read method which returns both the **x** and the **y** values
- If two synchronized functions are used, `readX` and `readY`, the value of the coordinate can be written between calls. The result will be an inconsistent value of the coordinate

Solution 1

- Return a new coordinate, which can be accessed freely

```
class Coord {
  synchronized Coord read() { return new Coord(x, y); }
  int readX() {return x;}    int readY() { return y; }
}
```

- The result is only a snapshot of the shared `Coord`, which might be changed by another thread right after the **`read`** has returned
- The individual field values will be consistent
- Once the coordinate has been used, it can be discarded and made available for garbage collection
- If efficiency is a concern, avoiding unnecessary object creation and garbage collection is appropriate

Solution 2

- Assume the client thread will use synchronized blocks to obtain atomicity

```
class Coord {
  ...
  synchronized void write(int x, int x) { … }

  int readX() { return x; } // not synchronized
  int readY() { return y; } // not synchronized
}
Coord p = new Coord(0,0);


synchronized(p) {
  Coord p2 = new Coord(p.readX(), p.readY());
}
```


Waiting and Notifying

- To obtain conditional synchronization requires the methods provided in the predefined object class
- These methods require the current thread to hold the object lock
- If called without the lock, the unchecked exception **`IllegalMonitorStateException`** is thrown
- The **`wait`** method always blocks the calling thread and releases the lock associated with the object

```
class Object {
  final void notify();
  final void notifyAll();
  final void wait()throws InterruptedException;
  final void wait(long millis) throws InterruptedException;
  final void wait(long millis, int nanos)throws InterruptedE
```


Important Notes

- The **`notify`** method wakes up one waiting thread; the one woken is not defined by the Java language
- **`notify`** does not release the lock; hence the woken thread must wait until it can obtain the lock before proceeding
- To wake up all waiting threads requires use of **`notifyAll`**
- If no thread is waiting, then **`notify`**/**`notifyAll`** are no-ops

Thread Interruption

- A waiting thread can also be awoken if it is interrupted by another thread
- In this case the **`InterruptedException`** is thrown

Condition Variables I

- There are no explicit condition variables in Java
- When a thread is awoken, it cannot assume that its condition is true, as all threads are potentially awoken irrespective of what conditions they were waiting on
- For some algorithms this limitation is not a problem, as the conditions are mutually exclusive,
- E.g., the bounded buffer traditionally has two condition variables: BufferNotFull and BufferNotEmpty
- If a thread is waiting for one condition, no other thread can be waiting for the other condition
- One would expect that the thread can assume that when it wakes, the buffer is in the appropriate state

Condition Variables II

- This is not always the case; Java makes no guarantee that a thread woken from a wait will gain immediate access to lock
- Another thread could call the put method, find that the buffer has space and inserted data into the buffer
- When the woken thread eventually gains access to the lock, the buffer will again be full
- Hence, it is usual for threads to re-evaluate their guards

Bounded Buffer

```
class BoundedBuffer  {
  private final int buffer[];
  private int first, last, numberInBuffer;
  private final int size;
  BoundedBuffer(int len){ buffer = new int[size = len]; }

  synchronized void put(int i)throws InterruptedException{
      while (numberInBuffer == size) wait();
      numberInBuffer++;
      buffer[last = (last+1)%size] = i;
      notifyAll();
  }

  synchronized int get()throws InterruptedException {
      while (numberInBuffer == 0) wait();
      numberInBuffer--;
      notifyAll();
      return buffer[first = (first+1)%size];
  }
}
```

Class Exercise

- How would you implement a semaphore using Java?

Summary I

- Errors in communication and synchronization cause working programs to suddenly suffer from deadlock or livelock
- The Java model revolves around controlled access to shared data using a monitor-like facility
- The monitor is represented as an object with synchronized methods and statements providing mutual exclusion
- Condition synchronization is given by the wait and notify method
- True monitor condition variables are not directly supported by the language and have to be programmed explicitly

Java 1.5 Concurrency Utilities

- Comprehensive support for general-purpose concurrent programming; partitioned into three packages:
  - `java.util.concurrent` — support common concurrent programming paradigms, e.g., various queuing policies such as bounded buffers, sets and maps, thread pools
  - `java.util.concurrent.atomic` — lock-free thread-safe programming on simple variables such as atomic integers, atomic booleans
  - `java.util.concurrent.locks` — framework for various locking algorithms, e.g., read -write locks and condition variables.

Locks I

```
package java.util.concurrent.locks;
public interface Lock {
  public void lock(); // Wait for the lock to be acquired
  public Condition newCondition();
  // Create a new condition variable for use with the Lock
  public void unlock();
  …
}

public class ReentrantLock implements Lock {
  public ReentrantLock();
  public void lock();
  public Condition newCondition();
  public void unlock();
}
```

Locks II

```
package java.util.concurrent.locks;
public interface Condition {
  public void await() throws InterruptedException;
  //Atomically releases associated lock and cause thread to wait
  public void signal(); // Wake up one waiting thread
  public void signalAll();// Wake up all waiting threads
}
```


Generic Bounded Buffer I

```
class BoundedBuffer<Data> {

  private final Data buffer[];
  private int first, last, numberInBuffer;
  private final int size;
  private final Lock lock = new ReentrantLock();
  private final Condition notFull  = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();

  public BoundedBuffer(int length) {
    buffer = (Data[]) new Object[size = length];
  }
```


Generic Bounded Buffer II

```
  public void put(Data item) throws InterruptedException {
    lock.lock();
    try {
      while (numberInBuffer == size) notFull.await();
      last = (last + 1) % size;
      numberInBuffer++;
      buffer[last] = item;
      notEmpty.signal();
    } finally { lock.unlock(); }
  }
  public Data get()throws InterruptedException {
     lock.lock();
     try {
       while (numberInBuffer == 0) notEmpty.await();
       first = (first + 1) % size ;
       numberInBuffer--;
       notFull.signal();
       return buffer[first];
     } finally { lock.unlock();}
  }
```


Asynchronous Thread Control

- Early versions of Java allowed one thread to asynchronously effect another thread through

```
public class Thread {
  ...
  public final void suspend();
  public final void resume();
  public final void stop();
  public final void stop(Throwable except)
         throws SecurityException;
```

All of the above methods are now obsolete and therefore should not be used

Thread Interruption

```
public class Thread …
  public void interrupt();
    // Send an interrupt to the associated thread
  public boolean isInterrupted();
  // Returns true if associated thread has been
  // interrupted, interrupt status is left unchanged

  public static boolean interrupted();
    // Returns true if the current thread has been
    // interrupted and clears the interrupt status
```


Thread Interruption

When a thread interrupts another thread:

- If the interrupted thread is blocked in `wait`, `sleep` or `join`, it is made runnable and the `InterruptedException` is thrown
- If the interrupted thread is executing, a flag is set indicating that an interrupt is outstanding; there is no immediate effect on the interrupted thread
- Instead, the called thread must periodically test to see if it has been interrupted using the `isInterrupted` or `interrupted` methods
  - If the thread doesn't test but attempts to blocks, it is made runnable immediately and the `InterruptedException` is thrown

Summary

- True monitor condition variables are not directly supported by the language and have to be programmed explicitly
- Communication via unprotected data is inherently unsafe
- Asynchronous thread control allows thread to affect the progress of another without the threads agreeing in advance as to when that interaction will occur
- There are two aspects to this: suspend and resuming a thread (or stopping it all together), and interrupting a thread
- The former are now deemed to be unsafe due to their potential to cause deadlock and race conditions
- The latter is not responsive enough for real-time systems

Completing The Java Model

- Aims:
  - To introduce thread priorities and thread scheduling
  - To show how threads delay themselves
  - To summarises the strengths and weaknesses of Java model
  - To introduce Bloch's safety levels

Thread Priorities

- Although priorities can be given to Java threads, they are only used as a guide to the underlying scheduler when allocating resources
- An application, once running, can explicitly give up the processor resource by calling the `yield` method, placing the thread to the back of the run queue for its priority level

```
public class Thread ...
  public static final int MAX_PRIORITY  = 10;
  public static final int MIN_PRIORITY  = 1;
  public static final int NORM_PRIORITY = 5;

  public final int getPriority();
  public final void setPriority(int newPriority);
  public static void yield();
```


Warning

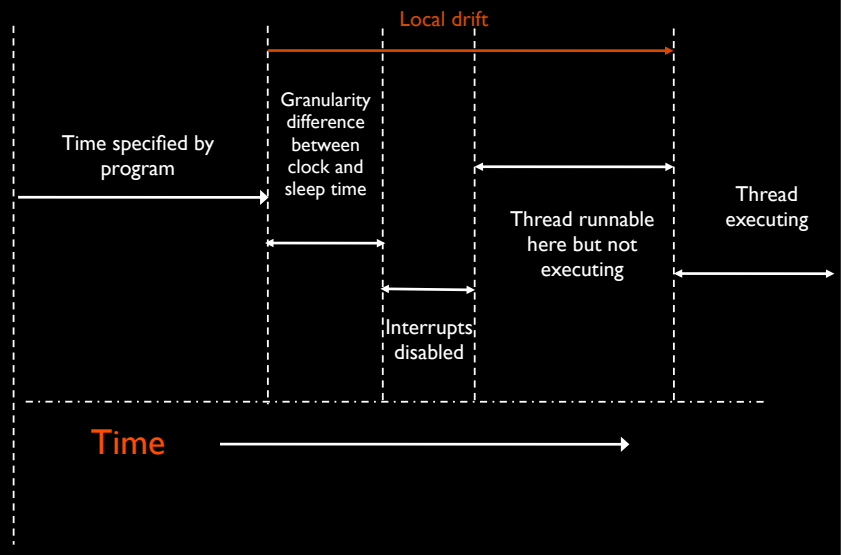
- From a real-time perspective, Java's scheduling and priority models are weak; in particular:
  - no guarantee is given that the highest priority runnable thread is always executing
  - equal priority threads may or may not be time sliced
  - where native threads are used, different Java priorities may be mapped to the same operating system priority

Delaying Threads: Clocks

- Java supports the notion of a wall clock
- `System.currentTimeMillis` returns the number of milliseconds since 1/1/1970 GMT and is used by used by `java.util.Date`
- However, a thread can only be delayed from executing by calling the `sleep` methods in the `Thread` class
- `sleep` provides a relative delay (sleep from now for some time), rather than sleep until 15th December 2003

```
class Thread...  ...
  static void sleep(long ms) throws InterruptedException;
  static void sleep(long ms,int nanoseconds)throws Interrup
```


Sleep Granularity



Absolute Delays I

- Consider an embedded system where the software controller needs to invoke two actions
- The second action must occur a specified period (say 10 seconds) after the first action has been initiated
- Simply sleeping for 10 seconds after a call to the first action will not achieve the desired effect for two reasons
  - The first action may take some time to execute. If it took 1 second then a sleep of 10 would be a total delay of 11 seconds
  - The thread could be pre-empted after the first action and not execute again for several seconds
- This makes it extremely difficult to determine how long the relative delay should be

Absolute Delays II

```
try{
  long start = System.currentTimeMillis();
  action_1();
  long end = System.currentTimeMillis();
  Thread.sleep(10000-(end-start));
} catch (InterruptedException ie) {...};
action_2();
```

What is wrong with this approach?

Timeout on Waiting I

- In many situations, a thread can wait for an arbitrary long period time within synchronized code for an associated **`notify`**
- The absence of the call, within a specified period of time, sometimes requires that the thread take some alternative action
- Java provides two methods for this situation both of which allows the **`wait`** method call to timeout
- There are two important points to note
  - As with sleep, the timeout is a relative time and not an absolute time
  - It isn't possible to know if the thread is woken by timeout or notify

Timeouts on Waiting

```
public class TimeoutException extends Exception {}

public class TimedWait {
  public static void wait(Object lock, long millis)
      throws InterruptedException, TimeoutException{
          // assumes the lock is held by the caller
    long start = System.currentTimeMillis();
    lock.wait(millis);
    if(System.currentTimeMillis() >= start + millis)
        throw new TimeoutException();
  }
}
```

What is wrong with this approach?

Strengths of the Java Concurrency Model

- Main strength is simplicity and direct support by the language
- Many of errors that potentially occur with uses of an operating system interface for concurrency do not exists in Java
- Language syntax + strong type checking gives some protection e.g., it is not possible to forget to end a synchronized block
- Portability is enhanced as the concurrency model is the same irrespective of the operating system on which the program runs

Weaknesses I

- Lack of support for condition variable
- Poor support for absolute time and time-outs on waiting
- No preference given to threads continuing after a notify over threads waiting to gain access to the monitor for the first time
- Poor support for priorities

Weaknesses II

- Synchronized code should be kept as short as possible
- Nested monitor calls should be avoided because the outer lock is not released when the inner monitor waits; this can lead to deadlocks
- It is not always obvious when a nested monitor call is made:
  - non-synchronized methods can still contain a synchronized block
  - non-synchronized methods can be overridden with a synchronized method; method calls which start off unsynchronized may be used with a synchronized subclass
  - interface methods cannot be labelled as synchronized

Bloch's Thread Safety Levels

- Immutable — Objects are constant and cannot be changed
- Thread-safe — Objects are mutable but they can be used safely in a concurrent environment as the methods are synchronized
- Conditionally thread-safe — Objects either have methods which are thread-safe, or have methods which are called in sequence with the lock held by the caller
- Thread compatible — Instances of the class provide no synchronization. However, instances of the class can be safely used in a concurrent environment, if the caller provides the synchronization by surrounding each method with the appropriate lock
- Thread-hostile — Instances should not be used in a concurrent environment even if the caller provides external synchronization. Typically because accessing static data or the external environment

Why care about Real-time?

- Real-time systems are inherently concurrent with multiple tasks of different priorities running on the same machine
- Most concurrent systems have implicit timeliness and responsiveness expectations
- There is much research in concurrent programming abstractions (e.g. transactional memory, data flow, actors,…)
- Yet, there has been, historically, little interest in Real-time from the programming language community and the real-time community tends to focus on scheduling and operating systems

Why care about Real-time?

- The **programming model** for most real-time systems is 'defined' as a function of the hardware, operating system, and libraries.
- Consequently real-time systems are not portable across platforms
- **Good news:** programming languages, such as Java and C#, are wrestling control from the lower layers of the stack and impose well-defined semantics (on threads, scheduling, synchronization, memory model)

What programming model?

- There are many dimensions:
  - Imperative vs. **Functional**
  - Shared memory vs. **Message passing**
  - Explicit lock-based synchronization vs. **Higher-level abstractions** (data-centric synchronization, transactional memory)
  - Time-triggered vs. **synchronous / logic execution time**
- And multiple languages, systems:
  - C, C++, Ada, SystemC, Assembler, Erlang, Esterel, Lustre, Giotto …

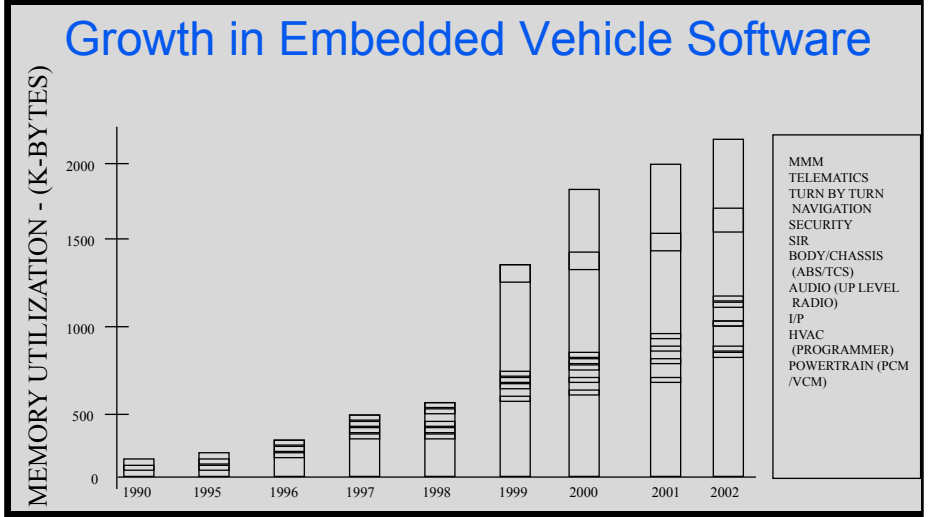
What programming model?

- "Real-time systems require fine grained control over resources and thus the language of choice is C, C++ or assembly"
- ...entails the software engineering drawbacks of low-level code
- Consider the following list of defects that have to be eradicated
  (c.f. "Diagnosing Medical Device Software Defects" Medical DeviceLink, May 2009):
  - Buffer overflow and underflow (does not occur in a HLL)
  - Null object dereference (checked exception in a HLL)
  - Uninitialized variable (does not occur in a HLL)
  - Inappropriate cast (all casts are checked in a HLL)
  - Division by zero (checked exception in a HLL)
  - Memory leaks (garbage collection in a HLL)

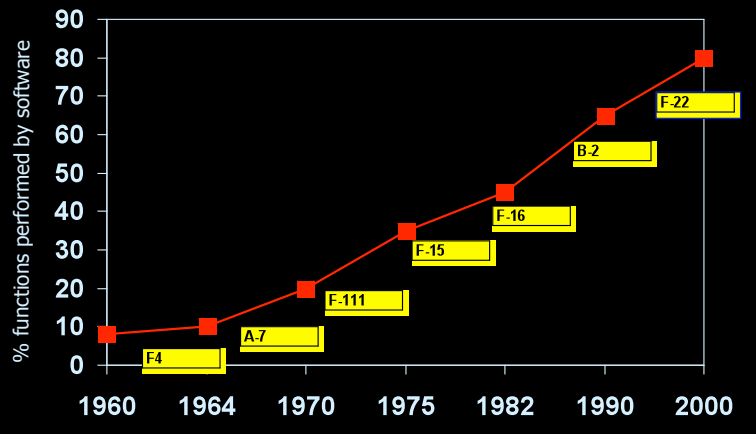
What programming model?

- Development time, code quality and certification are increasingly criteria. For instance in the automotive industry:
  - 90% of future innovation in the auto industry will be driven by electronics and software — *Volkswagen*
  - 80% of car electronics in the future will be software-based — *BMW*
  - 80% of our development time is spent on software — *JPL*
- Worst, software is often the source of missed project deadlines.

What programming model?



What programming model?



DoD Defense Science Board Task Force on Defense Software, November 2000

What programming model?

- Software and money
- Typical productivity
  - 5 Lines of Code (LoC) per person day ⇒ 1 kloc/person year
  - From requirements to end of module test
- Typical avionics "box"
  - 100 kloc ⇒ 100 person years of effort
  - $500M on a modern aircraft?

What programming model?

- The important metrics are thus
  - Reusability
  - Software quality
  - Development time

What programming model?

Java?

- **Productivity**

  Software-intensive systems require high-level languages
  C/C++ are less safe, and less portable
  Ada is struggling for developers
  Millions of Java programmers

- **What about performance?**

  For hand-tuned code, Java is ~2x slower than C,
  but large systems are often coded defensively
  it is often easier to analyze Java code

- **Is Java too dynamic?**

  Class loading and Just in time compilation need not be used
  Indeed many real-time JVM are compiled to C and statically linked

Java?

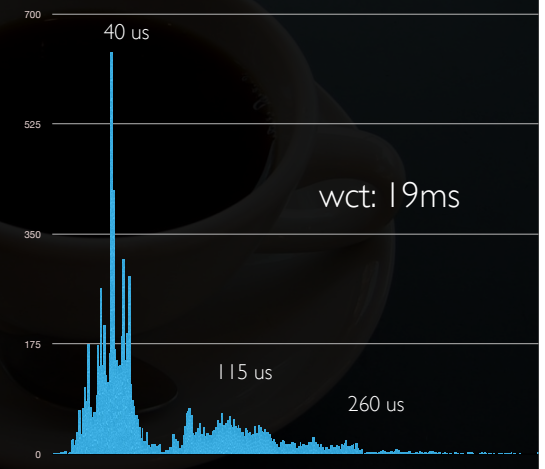
- Object-oriented programming helps software reuse
- Mature development environment and libraries
- Memory-safe high-level language
- Portable, little implementation-specific behavior
- Concurrency built-in, support for multiprocessors, memory model
- Garbage collected

Java?

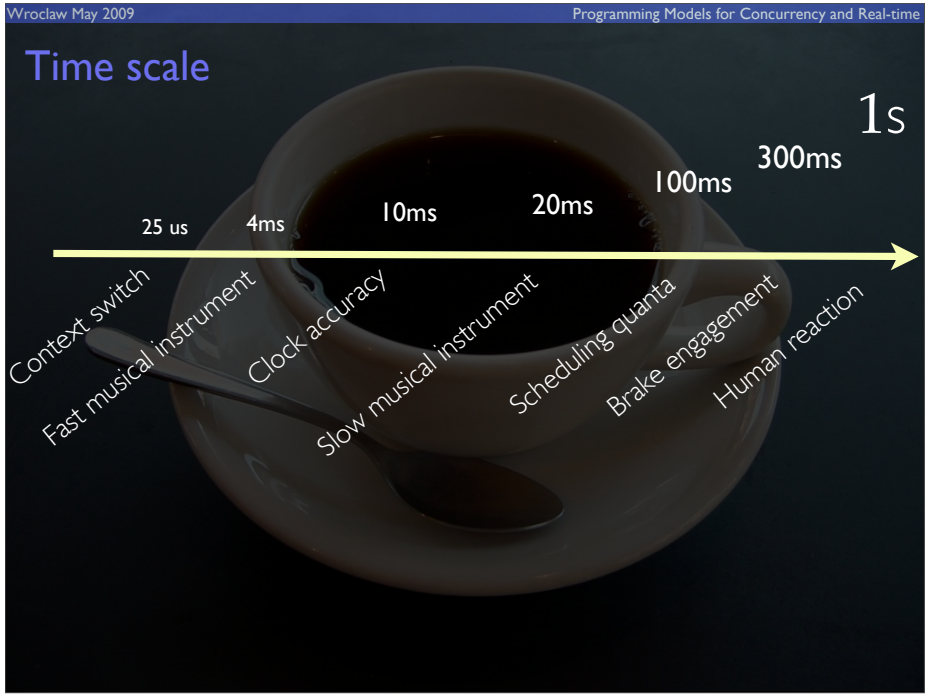
- Predictable?
- *Not really.*

Call sleep(10ms) and get up 20 milis.sec. variability.

Hard real-time often requires microsecond accuracy.

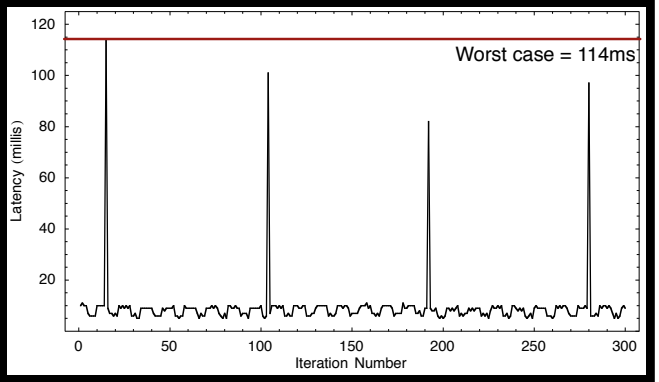


Time scale



Java?

- Predictable?



- **Java Collision Detector** running at 20Hz.
  - *Bartlett's Mostly Copying Collector. Ovm. Pentium IV 1600 MHz, 512 MB RAM, Linux 2.6.14, GCC 3.4.4*
- GC pauses cause the collision detector to miss up to three deadlines… *and this is not a particularly hard problem should support KHz periods*

Java?

- Java for **hard** real-time?

Case studies

The Real-time Specification for Java

(S³) PURDUE UNIVERSITY Fiji Systems LLC IBM

Tuesday, May 26, 2009

RTSJ Programming Model

- Java-like:
  - Shared-memory,
  - lock-based synchronization,
  - first class threads.
- Main real-time additions:
  - Real-time threads + Real-time schedulers
  - Priority avoidance protocols: PIP or PCE
  - Region-based memory allocation (to avoid GC pauses)

Tuesday, May 26, 2009

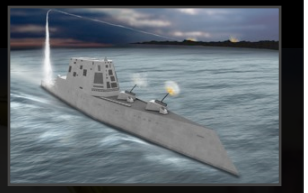
RTSJ

- Commercial implementations:
  - PERCS (AONIX), JamaicaVM (AICAS) ahead-of-time compiler
  - McKinack (SUN) based on Hotspot, JIT, SMP, RTGC
  - Websphere (IBM) based on J9, ahead/JIT, SMP, RTGC

Tuesday, May 26, 2009

Real-time applications

- Shipboard computing
  - US navy Zumwalt-class Destroyer, Raytheon / IBM
    5 mio lines of Java code
    Real-time GC key part of system.
- Avionics
  - Zedasoft's Java flight simulator
  - IBM Scaneagle UAV
- Financial information systems

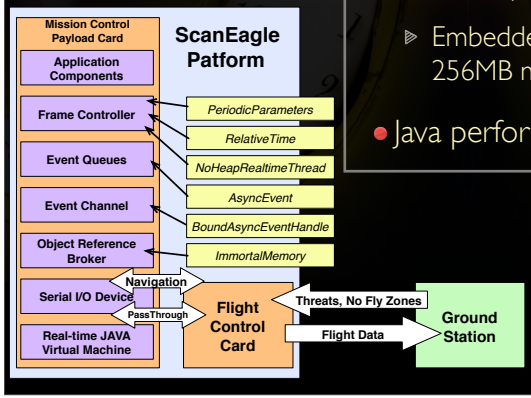


Case Study: ScanEagle

ScanEagle

ScanEagle

- Flight Software:
  - 953 Java classes, 6616 methods.
    Multiple Priority Processing:
    - High (20Hz) - Communicate with Flight Controls
    - Medium (5 Hz) - Computation of navigation data
    - Low (1 Hz) - Performance Computation
  - Embedded Planet 300 Mhz PPC, 256MB memory, Embedded Linux
- Java performed better than C++



Roadmap

- **Overview of the RTSJ**
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control
- Schedulability Analysis
- Conclusions

The Real-Time Specification for Java

Lecture aims

- To give the background of the RTSJ and the NIST requirements
- To provide an introduction to
  - Memory management
  - Time values and clocks
  - Schedulable objects and scheduling

Background and NIST Requirements

- In the late 1990s, the US National Institute of Standards and Technology (NIST) coordinated the derivation of guiding principles and requirements for real-time extensions to Java
- Requirements
  - Fixed priority and round robin scheduling
  - Mutual exclusion locking (avoiding priority inversion)
  - Inter-thread communication (e.g. semaphores)
  - User-defined interrupt handlers and device drivers — including the ability to manage interrupts (e.g., enabling and disabling)
  - Timeouts and aborts on running threads
  - Profiles to cope with the variety of applications, these included: safety critical, no dynamic loading, and distributed real-time

RTSJ Guiding Principles

- Be backward compatible with non real-time Java programs
- Support the principle of "Write Once, Run Anywhere" but not at the expense of predictability
- Address the current real-time system practice and allow future implementations to include advanced features
- Give priority to predictable execution in all design trade-offs
- Require no syntactic extensions to the Java language
- Allow implementers flexibility

Overview of Enhancements

- The RTSJ enhances Java in the following areas:
  - memory management
  - time values and clocks
  - schedulable objects and scheduling
  - real-time threads
  - asynchronous event handling and timers
  - asynchronous transfer of control
  - synchronization and resource sharing
  - physical memory access

Warning

The RTSJ only addresses the execution of real-time Java programs on single processor systems. It attempts not to preclude execution on a shared-memory multiprocessor systems but it has no support for, say, allocation of threads to processors.

Memory Management

- Many RTS have limited amount of memory available because of
  - the cost
  - constraints associated with surrounding system (size, power, weight)
- It is necessary to control how memory is allocated to use it effectively
- Where there is more than one type of memory (with different access characteristics), it may be necessary to instruct the compiler to place certain data types at certain locations
- By doing this, the program is able to increase performance and predictability as well as interact with the outside world

Heap Memory

- The JVM is responsible for managing the heap
- Problems are how much space is required and when to release it
- The latter can be handled in several ways, including
  - require the programmer to return the memory explicitly — this is error prone but is easy to implement
  - require the JVM to monitor the memory and release chunks which are no longer being used

Real-Time Garbage Collection

- From a RT perspective, the approaches have an impact on the ability to analyze the program's timing properties
- Garbage collection may be performed either when the heap is full or by an incremental activity
- In either case, running the garbage collector may have a significant impact on the response time of a time-critical thread
- Objects in standard Java are allocated on the heap and the language requires garbage collection
- The garbage collector runs as part of the JVM

Memory Areas

- RTSJ provides memory management which is not affected by GC
- It defines **memory areas**, some of which exist outside the traditional Java heap and never suffer garbage collection
- RTSJ requires that the GC be preemptible by real-time threads and that be a bounded latency for preemption
- The `MemoryArea` class is an abstract class from which for all RTSJ memory areas are derived
- When a particular memory area is entered, all object allocation is performed within that area

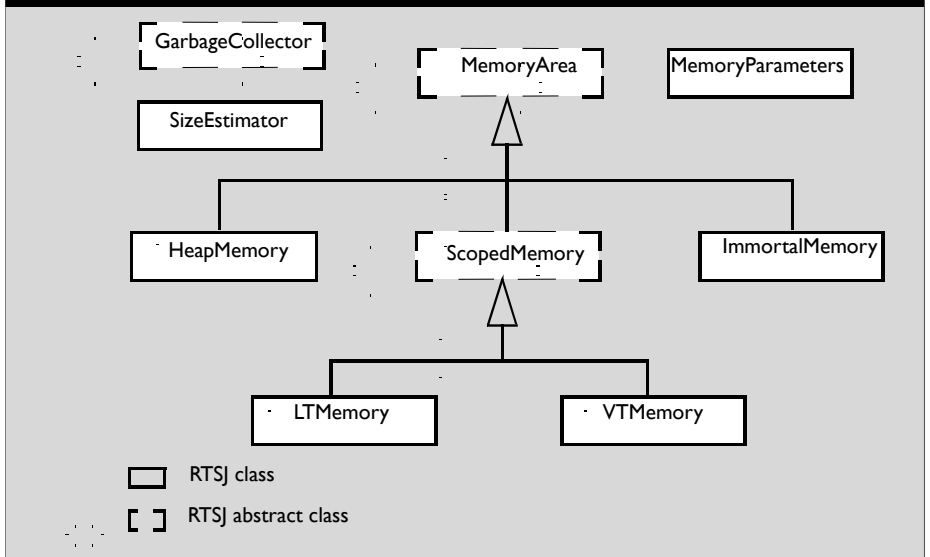
Subclasses of MemoryArea

- `HeapMemory` allows objects to be allocated in the heap
- `ImmortalMemory` is shared among all threads; objects created here are never subject to GC
- `ScopedMemory` is for objects that have a well-defined lifetime; with each scoped memory is a reference count which keeps track of how many real-time entities are currently using it
  - When the reference count goes from 1 to 0, all objects in the area have their finalization method executed and the memory is reclaimed
- `ScopedMemory` is abstract and has several subclasses
  - `VTMemory`: allocations may take variable amounts of time
  - `LTMemory`: allocations occur in linear time to the size of the object

Memory Parameters

- Can be given when real-time threads and asynchronous event handlers are created; they specify
  - maximum amount of memory a thread/handler can use in an area,
  - max. amount of memory that can be consumed in immortal memory
  - a limit on the rate of allocation from the heap (in bytes per second),
- Can be used by the scheduler as part of an admission control policy and/or for the purpose of ensuring adequate GC

Memory Management Classes



Time Values

- `HighResolutionTime` for time values with nanosecond granularity represented by a 64 bits milliseconds and a 32 bits nanoseconds component
- The class is abstract with subclasses:
  - `AbsoluteTime`: expressed as a time relative to some epoch. This epoch depends of the associated clock. It might be 1/1/970 for the wall clock or system start-up time for a monotonic clock
  - RelativeTime
- Time values are also relative to particular clocks

Clocks

- `Clock` is the abstract class from which all clocks are derived
- RTSJ allows many types of clocks; eg, there could be a CPU execution-time clock (although this is not required)
- At least one real-time clock which advances monotonically
- A static method `getRealtimeClock` returns this clock

Summary

- The RTSJ originates from the desire to use Java in real-time
- Java's main problems include unsuitable memory management because of GC and poor support for clocks and time
- The RTSJ stems from the NIST requirements
- Memory management is augmented by memory areas
- Clocks are augmented by a real-time clock and by high resolution absolute and relate time types

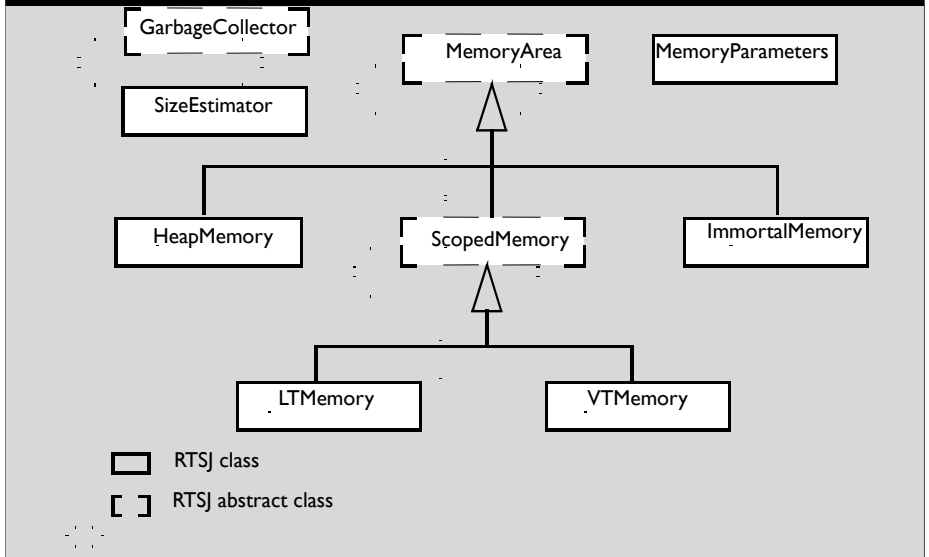
Roadmap

- **Overview of the RTSJ**
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

The RTSJ Overview Continued

- Lecture aims
- To complete the overview of the RTSJ by considering
  - Schedulable objects and scheduling
  - Real-time threads
  - Asynchronous events and timers
  - Asynchronous transfer of control
  - Synchronization and resource sharing
  - Physical memory access

Memory Management Classes



Scheduling in Java

- Java offers no guarantees that the highest priority runnable thread will always be the one executing
- This is because the OS may not support preemptive priority-based scheduling
- Java only defines 10 priority levels and an implementation is free to map these priorities onto a more restricted host operating system's priority range if necessary
- The weak definition of scheduling and restricted range of priorities means Java lacks predictability and, hence, Java's use for RT systems implementation is severely limited

Schedulable Objects

- RTSJ generalizes the entities that can be scheduled from threads to the notion of schedulable objects
- A **schedulable object** implements the `Schedulable` interface
- Each schedulable object must also indicate its specific
  - release requirement (that is, when it should become runnable),
  - memory requirements (e.g., heap allocation rate)
  - scheduling requirements (e.g, priority at which it should be scheduled)

Release Parameters

- Scheduling theories often identify three types of releases:
  - periodic (released on a regular basis)
  - aperiodic (released at random)
  - sporadic (released irregularly but with a minimum time bw releases)
- All release parameters have:
  - cost: amount of cpu time needed every release
  - deadline: the time at which the current release must have finished
- `PeriodicParameters` also include the start time for the first release and the time interval (period) between releases.
- `SporadicParameter` include the minimum inter-arrival time between releases

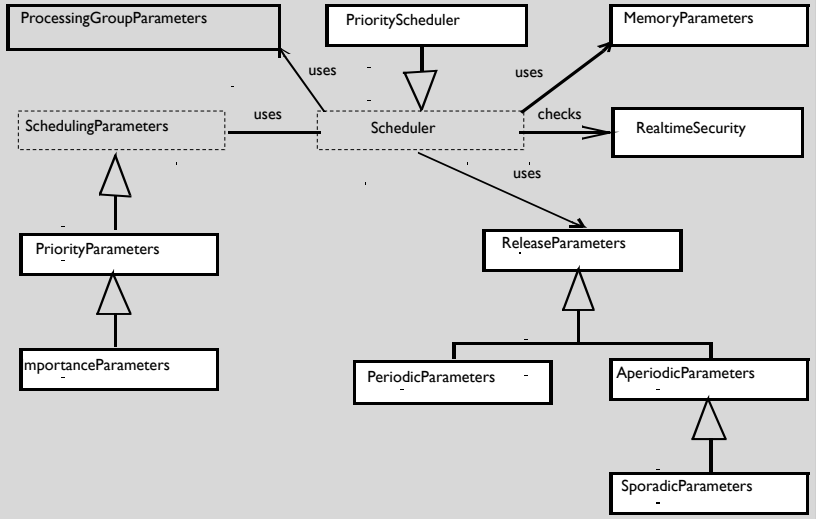
Scheduling Parameters

- Scheduling parameters are used by a scheduler to determine which object is currently the most eligible for execution
- The abstract class **SchedulingParameters** provides the root class from which a range of possible scheduling criteria can be expressed
  - The RTSJ defines only one criterion which is based on priority
  - High values for priority represent high execution eligibilities.

Schedulers

- Responsible for scheduling associated schedulable objects
- RTSJ supports priority-based scheduling via the `PriorityScheduler` (a fixed preemptive priority-based scheduling with 28 unique priority levels)
- `Scheduler` is an abstract class with `PriorityScheduler` a defined subclass
- An implementation could provide an EarliestDeadlineFirst scheduler
- Attempt by the application to set the scheduler for a thread has to be checked for the appropriate security permissions

Scheduling-related Classes



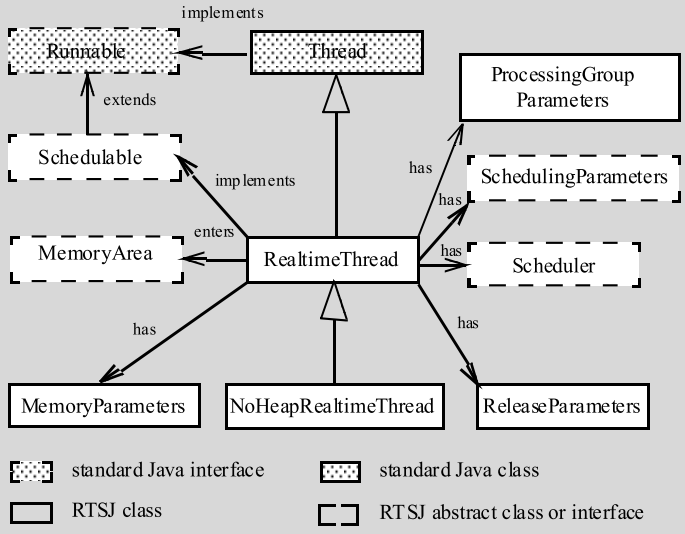
Meeting Deadlines

- A real-time system needs to
  - predict whether a set of application objects will meet their deadlines
  - report a missed deadline, a cost overrun, or min. inter-arrival violation
- For some systems it is possible to predict offline whether the application will meet its deadline
- For other systems, some form of on-line analysis is required
- The RTSJ provide the hooks for on-line analysis
- Irrespective of how prediction is done, it is necessary to report overruns
- The RTSJ provides an asynchronous event handling mechanism for this purpose

Real-Time Threads

- A schedulable object
- More than an extension of `java.lang.thread`
- No heap version ensure no access to the heap and therefore independence from garbage collection

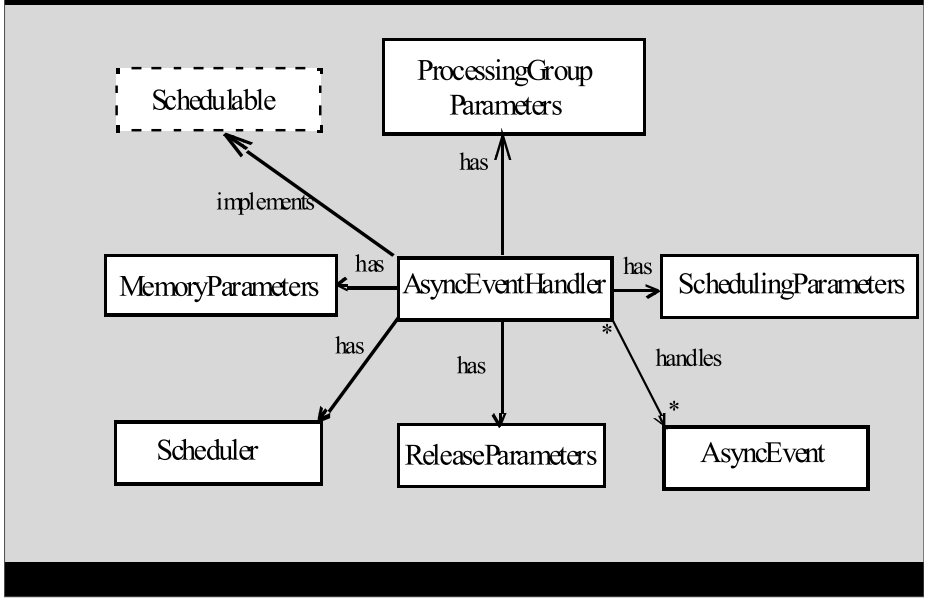
Real-time Threads



Asynchronous Event Handling

- Threads and RT threads are the abstractions to use to represent concurrent activities that have a significant life history. But it is also necessary to respond to events that happen asynchronously to a thread's activity
- Events may be happenings in the environment of an embedded system or notifications received from within the program
- It possible to have threads wait for them but this is inefficient
- From a RT perspective, events may require their handlers to respond within a deadline; hence, more control is needed over the order in which events are handled
- RTSJ generalizes event handlers to be schedulable entities

Async Events and their Handlers



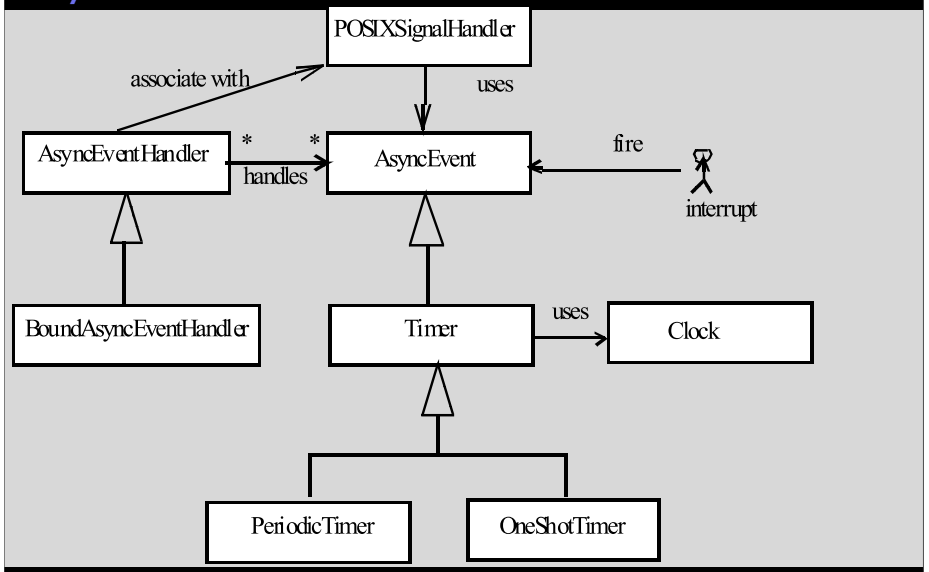
Handlers and Real-Time Threads

- In practice, an event handler will be associated dynamically with a RT thread when the handler is released for execution
- To avoid this overhead, it is possible to specify that the handler must be permanently bound to a RT thread
- Each `AsyncEvent` can have one or more handlers and the same handler can be associated with more than one event
- When the event occurs, all the handlers associated with the event are released for execution according to their `SchedulingParameters`

More on Async Events

- Asynchronous events can be associated with interrupts or POSIX signals or they can be linked to a timer
- The timer will cause the event to fire at a specified time
- This can be a one shot firing or a periodic firing

AsynEvent related Classses



Asynchronous Transfer of Control

- Asynchronous events allow the program to respond in a timely fashion to a condition
- They do **not** allow a particular schedulable object to be directly informed
- In many apps, the only form of asynchronous transfer of control that a real-time thread needs is a request for it to terminate itself
- Consequently, languages and OSs provide a kill or abort facility
- For RT systems, this is too heavy handed; instead what is required if for the schedulable object to stop what it is doing and to execute an alternative algorithm

ATC I

- In standard Java, it is the interrupt mechanism which attempts to provide a form of asynchronous transfer of control
- The mechanism does not support timely response to the "interrupt"
- Instead, a running thread has to poll for notification
- This delay is deemed unacceptable for real-time systems
- For these reasons, the RTSJ provides an alternative approach for interrupting a schedulable object, using asynchronous transfer of control (ATC)

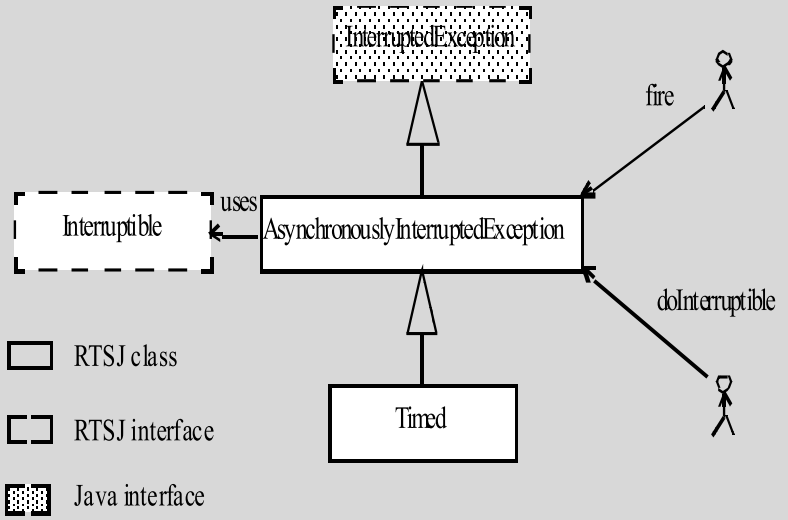
ATC II

- The ATC model is based on the following principles
  - A schedulable object must explicitly indicate that it is prepared to allow an ATC to be delivered
  - By default, schedulable object will have ATCs deferred
  - The execution of synchronized methods and statements always defers the delivery of an ATC
  - An ATC is a non-returnable transfer of control

ATC III

- The RTSJ ATC model is integrated with the Java exception handling facility
- An `AsynchronouslyInterruptedException` (AIE) class defines the ATC event
- A method that is prepared to allow an AIE indicate so via a throws `AsynchronouslyInterruptedException` in its declaration
- The `Interruptible` interface provides the link between the AIE class and the object executing an interruptible method

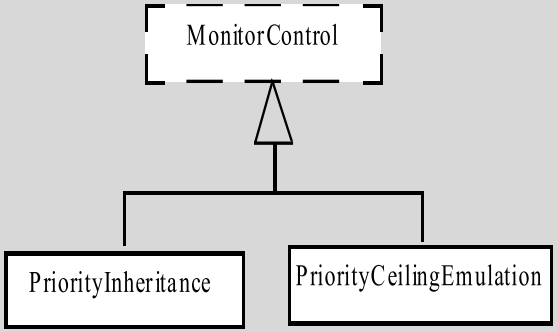
The ATC Classes and Interface



Synchronization

- Key to predicting the behavior of concurrent programs is understanding how threads communicate and synchronize
- Java provides mutually exclusive access to shared data via a monitor-like construct
- All synchronization mechanisms which are based on mutual exclusion suffer from priority inversion
- The problem of priority inversion is well-researched
- There are many priority inheritance algorithms; RTSJ support two: simple priority inheritance and priority ceiling emulation inheritance

RTSJ Classes for Priority Inheritance



Priority Inheritance and GC

- If RT threads want to communicate with plain threads then interaction with GC must be considered
- It is necessary to try to avoid the situation where a plain thread has entered into a mutual exclusion zone shared with a RTthread
- The actions of the plain thread results in garbage collection
- The RT thread then preempts GC but is unable to enter the mutual exclusion zone
- It must now wait for GC to finish and the plain thread to leave

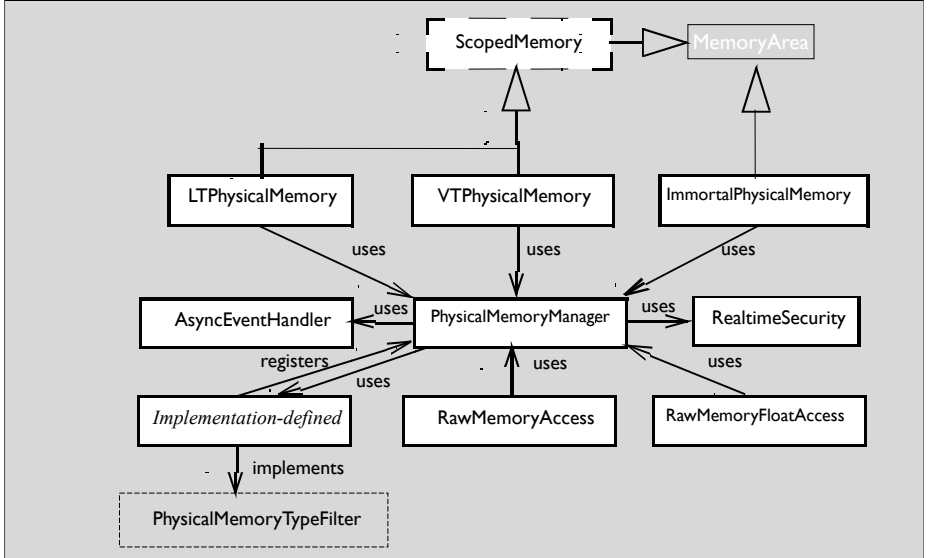
Wait Free Communication

- One way of avoiding unpredictable interactions with GC is to provide a non-blocking communication mechanism for use between plain and RT threads
- RTSJ provides three wait-free non blocking classes:
  - `WaitFreeWriteQueue`: a bounded buffer; the read operation is synchronized; the write operation is not synchronized
  - `WaitFreeReadQueue`: a bounded buffer; the write operation on the buffer is synchronized; the read operation is not; the reader can request to be notified (via an asynchronous event) when data arrives
  - `WaitFreeDequeue`: a bounded buffer which allows both blocking and non-blocking read and write operations

Physical and Raw Memory Classes

- Allow objects to be placed into parts of memory with particular properties or access requirements; eg DMA memory, shared memory
  - Extensions of `MemoryArea` are the physical memory counterparts to the linear-time, variable-time and immortal memory classes
- Allow the programmer to access raw memory locations that are being used to interface to the outside world; e.g memory-mapped I/O device registers
  - Classes which can access raw memory in terms of reading and writing Java variables or arrays of primitive data types (int, long, float etc.)

Physical and Raw Memory Classes II



Roadmap

- Overview of the RTSJ
- **Memory Management**
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

Memory Management

- Lecture aims
  - To provide an introduction to memory management in the RTSJ
  - To give an overview of MemoryAreas
  - To show an example of Scoped Memory Usage
  - To consider how to estimating the size of objects

Introduction I

- An advantage of using a high-level language is that it relieves the programmer of the burden of dealing with many of the low-level resource allocation issues
- Issues such as assigning variables to registers or memory locations, allocating and freeing memory for dynamic data structures, etc., all detract the programmer from the task at hand
- Java remove many of these troublesome worries and provide high-level abstract models that the programmer can use

Introduction II

- For real-time/embedded systems programming there is a conflict
  - Use of high-level abstractions aid in software engineering
  - Yet embedded and real-time systems often only have limited resources which must be carefully managed
- Nowhere is this conflict more apparent than in the area of memory management
- Embedded systems usually have a limited amount of memory available; this is because of cost, size, power, weight or other constraints imposed by the overall system requirements
- It may be necessary to control how this memory is allocated so that it can be used effectively

Stack versus Heap Memory

- Most languages provide two data structures to help manage dynamic memory: the stack and the heap
- The stack is typically used for storing variables of basic data types (such as int, boolean and references) local to a method
- All objects, which are created from class definitions, are stored on the heap and Java requires GC
- Much work has been done on real-time GC, yet there is still a reluctance to rely on it in time-critical systems

Real-time and Garbage Collection

- GC may be performed either when the heap is full or incrementally (asynchronous garbage collection)
- GC may have significant impact on the response time of a RT thread
- Consider a time-critical periodic thread which has had all its objects pre-allocated
- Even though it may have a higher priority than a plain thread & will not require any new memory, it may still be delayed if it preempts the plain thread during GC
- It is not safe for the time-critical thread to execute until GC has finished (particularly if memory is compacted)

The RTSJ and Memory Management

- The RTSJ recognizes that it is necessary to allow memory management which is not affected by the problems of garbage collection
- It does this via the introduction of immortal and scoped memory areas
- These are areas of memory which are logically outside of the heap and, therefore, are not subject to garbage collection
- If a schedulable object is active in a memory area, all calls to new create the object in that memory area

The Basic Model

- RTSJ provides two alternatives to using the heap: immortal memory and scoped memory
- The memory associated with objects allocated in immortal memory is never subject to GC
- Objects allocated in scoped memory have a well-defined life time
- Schedulable objects may enter and leave a scoped memory area
- Whilst they are executing within that area, all memory allocations are performed from the scoped memory
- When there are no schedulable objects active inside a scoped memory area, the allocated memory is reclaimed

Memory Areas I

```
public abstract class MemoryArea {
  protected MemoryArea(long sizeInBytes);
  protected MemoryArea(long sizeInBytes, Runnable logic);
  public void enter();
```

Associate this memory area to the current schedulable object for the duration of the run method passed as a parameter to the constructor

```
  public void enter(Runnable logic);
```

Associate this memory area to the current schedulable object for the duration of the run method of the object passed as a parameter

```
  public static MemoryArea getMemoryArea(Object object);
```

Get the memory area associated with the object

```
  public long memoryConsumed();
```

Returns the number of bytes consumed

```
  public long memoryRemaining();
```

Returns the number of bytes remaining

```
  public long size();
```

Returns the current size of the memory area

Immortal Memory

- There is only one `ImmortalMemory` area, hence the class is defined as final and has only one additional method (`instance`) which will return a reference to the immortal memory area
- Immortal memory is shared among all threads in an application
- Note, there is no public constructor for this class. Hence, the size of the immortal memory is fixed by the JVM

```
public final class ImmortalMemory extends MemoryArea {
  public static ImmortalMemory instance();
}
```


Allocating Objects in Immortal Memory

- The simplest method for allocating objects in immortal memory is to use the enter method in the `MemoryArea` class and pass an object implementing the `Runnable` interface

```
ImmortalMemory.instance().enter(new Runnable(){
      public void run() {
```

any memory allocation performed here using the allocator will occur in Immortal

```
      }
    });
```

Although memory allocated by the **`run`** method will occur from immortal memory, memory needed by the object implementing the **`run`** method will be allocated from the current memory area at the time of the call to enter

Linear Time Memory

```
public class LTMemory extends ScopedMemory {
  public LTMemory(long initialSizeInBytes,
                  long maxSizeInBytes);
  public LTMemory(long initialSizeInBytes,
                  long maxSizeInBytes, Runnable logic);
  ...
  public int getMaximumSize();
```

Linear time refers to the time it takes to allocate an object, and not the time it takes to run the constructor

Reference Counts

- Each scoped memory object has a reference count which indicates the number of times the scope has been entered
- When that reference count goes from 1 to 0, the memory allocated in the scoped memory area can be reclaimed (after running any finalization code associated with the allocated objects)

Example: Scoped Memory

- Consider a class that encapsulates a large table which contains (two by two) matrices
- The match method takes a two by two matrix and performs the dot product of this with every entry in the table
- It then counts the number of results for which the dot product is the unity matrix

Matrix

```
class MatrixExample {
  MatrixExample(int Size) { /* initialize table */ }

  int match(final int with[][])
  {/*check if "with" is in the table*/}

  private int[][] dotProduct(int[][] a, int [][] b)
  { /* calculate dot product return result */ }

  private boolean eq(int[] [] a, int[] [] b)
  { /* returns true if the matrices are equal */ }

  private int [][][] table; // 2 by 2 matrices
  private int [][] unity = {{1,1},{1,1}};
}
```


Match Method: with GC

```
public int match(final int with[][]) {
  int found = 0;
  for(int i=0; i < table.length; i++) {
    int[][] product = dotProduct(table[i], with);
    if(eq(product, unity)) found++;
  }
  return found;
}
```

- Each time around the loop, a call to `dotProduct` is made which creates and returns a new matrix object
- This object is then compared to the unity matrix
- After this comparison, the matrix object is available for GC

Match Method: LTMemory

```
public int match(final int with[][]) {// first attempt
  LTMemory myMem = new LTMemory(1000, 5000);
  int found = 0;
  for(int i=0; i < table.length; i++) {
    myMem.enter(new Runnable(){
      public void run() {
        int[][] product = dotProduct(table[i], with);
        if(eq(product, unity)) found++;
       }
      });
    }
  }
  return found;
}
```


Problems

- Accessibility of the loop parameter `i`; only `final` local vars can be accessed from within a class nested within a method
- To solve this: create a new `final` variable, `j`, inside the loop which contains the current value of `i`
- `found` must become a field and initialized to `0` on each call
- The anonymous class is a subclass of `Object` not the outer class, consequently, `eq` is not in scope
- This can be circumvented by naming the method explicitly

Matrix Version 2

```
public class MatrixExample    {
  public MatrixExample(int Size)
  { /* initialize table */ }

  public int match(final int with[][])
  {  /* check if "with" is in the table */ }

  private int[][] dotProduct(int[][] a, int [][] b)
  { /* calculate dot product return result */ }

  private boolean equals(int[] [] a, int[] [] b)
  { /* returns true if the matrices are equal */ }

  private int [][][] table; // 2 by 2 matrices
  private int [][] unity = {{1,1},{1,1}};
  public int found = 0;
  private LTMemory myMem;
}
```


Match Version 2

```
public int match(final int with[][]) {
  found = 0;
  for(int i=0; i < table.length; i++) {
    final int j = i;
    myMem.enter(new Runnable(){
      public void run() {
        int[][] product = dotProduct(table[j], with);
        if(MatrixExample.this.equals(product, unity))
          found++;
       }
      });
  }
  return found;
}
```

Problem

- We now reclaim temporary memory every time `enter()` returns
- However creating instance of the anonymous class still use memory in the surrounding memory area
- There will be a few bytes of objects header (e.g. a monitor, class table, hash code) and a copy of the `j` variable and a reference to the `with` array (this is how the compiler implements the access to the method's data)
- Although this memory usage cannot be eliminated, it can be bounded by caching the object in a field

Matrix Final Version

```
public class MatrixExample {
  ...
  Product produce = new Product();

  private class Product implements Runnable {
    int j;
    int withMatrix[][];
    int found = 0;
    public void run() {
      int[][] product=dotProduct(table[j],withMatrix);
      if(matrixEquals(product, unity)) found++;
    }
  }
  ...
}
```


Match Final Version

```
public int match(final int with[][])  {
  produce.found = 0;
  for(int i=0; i < table.length; i++) {
    produce.j = i;
    produce.withMatrix = with;
    myMem.enter(produce);
  }
  return produce.found;
}
```

- Now, there is just the initial cost of creating the produce object (performed in the constructor of `MatrixExample`)
- Note that the only way to pass parameters to the run method is via setting attributes of the object (in this case directly)
- Note also that only one thread may call match at a time

Size Estimator

```
public final class SizeEstimator {
  public SizeEstimator();
  public long getEstimate();
  public void reserve(Class c, int n);
  public void reserve(SizeEstimator s);
  public void reserve(SizeEstimator s, int n);
```

- Allows the size of a Java object to be estimated
- Again, size is the object itself, it does not include an objects created during the constructor

```
SizeEstimator s = new SizeEstimator();
s.reserve(javax.realtime.PriorityParameters.class, 1);
System.out.println("size of PP is "+s.getEstimate());
```

Summary

- The lack of confidence in RTGC is one of the main inhibitors to the widespread use of Java in real-time and embedded systems
- The RTSJ has introduced an alternative memory management facility based on the concept of memory areas
- There are two types on non-heap memory areas
  - a singleton immortal memory which is never subject to GC
  - scoped memory in to which schedulable objects can enter and leave. When there are no schedulable objects active in a scoped memory area, all the objects are destroyed and their memory reclaimed

Roadmap

- Overview of the RTSJ
- **Memory Management**
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

Memory Management

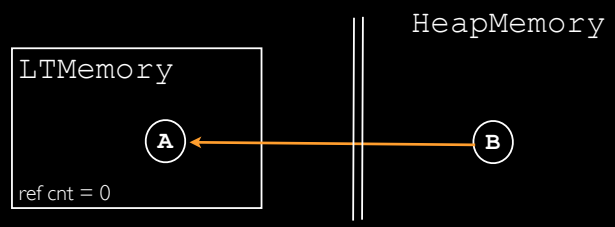
Lecture aims:

- To explain the RTSJ assignment Rules
- To illustrate the single parent rule
- To consider the sharing of memory areas between `Schedulable` objects

Memory Assignment Rules

- In the RTSJ there are four types of memory
  - heap memory: collected by the garbage collector
  - local variables: collected automatically when methods exit
  - immortal memory: never collected
  - scoped memory: collected when reference count equals zero
- Given the different collection mechanism, it is necessary to be careful when accessing memory
- Otherwise dangling references may occur
- *A dangling reference is a references to an already collected object*

Dangling Reference Example



- **A**, has been created in a scoped memory region
- A reference to **A** has been stored in object, **B**, in the heap
- The lifetime of a scoped memory is controlled by its reference count, if it goes to 0, the there will be a reference to a non-existent object

The Reference Count

The reference count of a scoped memory area is the count of the number of active calls to its **`enter`** method. It is **not** a count of the number of objects that have references to the objects allocated in the scoped memory.

The reference to object, **A**, from **B** becomes invalid when **A**'s memory area is reclaimed. Without something like GC there is no support for detecting this invalid reference, so the safety of the Java program would be compromised.

Memory Assignment Rules

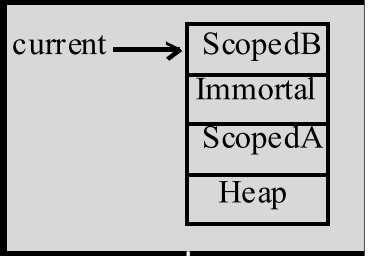
From	To Heap Memory	To Immortal Memory	To Scoped Memory
Heap Memory	allowed	allowed	forbidden
Immortal Memory	allowed	allowed	forbidden
Scoped Memory	allowed	allowed	allowed is to same scope or outer scope forbidden if to an inner scope
Local Variable	allowed	allowed	generally allowed

Note

- If the program violates the assignment rules, the unchecked exception `IllegalAssignmentError` is thrown
- One of the requirements for RTSJ was that there should be no changes to the Java language and that existing compilers can be used to compile RTSJ programs
- So these rules are enforced on every assignment at run-time by the JVM
- An RTSJ-aware compiler may optimize some check at compile-time or at class loading-time, but there is likely to be some residual checks
- In practice, the overhead of the checks is 10 - 20% of execution time if implemented efficiently

Nested Memory Areas

- The JVM keeps track of the currently active memory areas of each schedulable object
- Usually by keeping a stack of areas
- Every time a schedulable object enters an area, the identity of that area is pushed onto the stack
- When it leaves the memory area, the identity is popped off



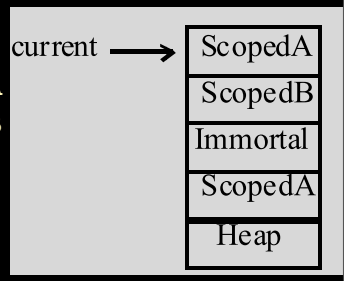
stack grows upwards

Implementation of Assignment Rules

- The stack can be used to check for invalid memory assignment to and from scoped memory areas
- Creating a reference from an object in a scoped area to an object in another scoped are below the first area in the stack is allowed
- Creating a reference from an object in one scoped memory area to an object in another area above the first area is forbidden

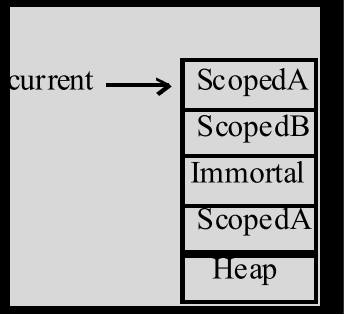
Consider

- Assume a schedulable enters the same memory area twice
- an object could be created in **`ScopedA`** which references an object in **`ScopedB`**
- when the current **`ScopedA`** memory is exited, its reference count is still **>0**. so its objects are not reclaimed
- when **`ScopedB`** is exited, it objects are reclaimed creating a dangling reference in **`ScopedA`**



The Single Parent Rule

- To avoid this problem, the RTSJ requires that each scoped memory area has a single parent
- The parent of an active scoped memory area is
  - If the memory is the first scoped area on the stack, its parent is termed the primordial scope area
  - For all other scoped memory areas, the parent is the first scoped area below it on the stack



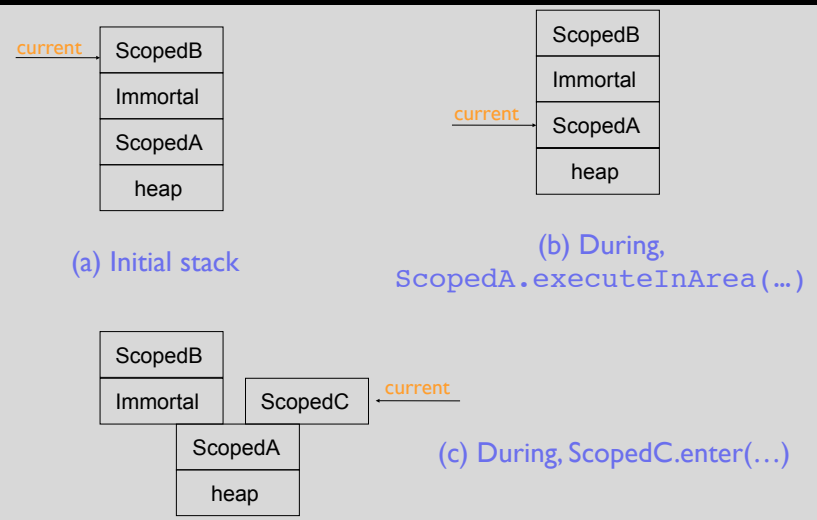
The example violates the single parent rule, therefore `ScopedCycleException` is thrown

Moving Between Memory Areas

- As it is not possible to re-enter an active scoped memory area, it is necessary to provide alternative mechanisms for moving between active memory areas

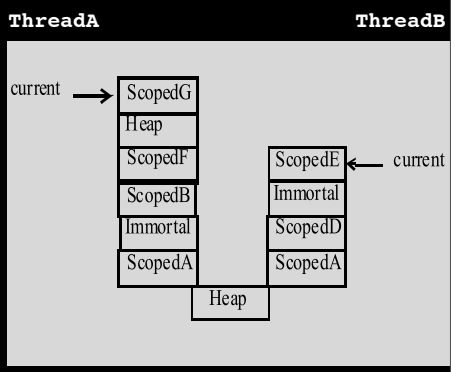
```
public abstract class MemoryArea {

  public void executeInArea(Runnable logic);

  public Object newArray(Class type, int number)throws …
  public Object newInstance(Class type) throws …
  public Object newInstance(Constructor c, Object[] args)
    throws …
```


Cactus Stack Needed



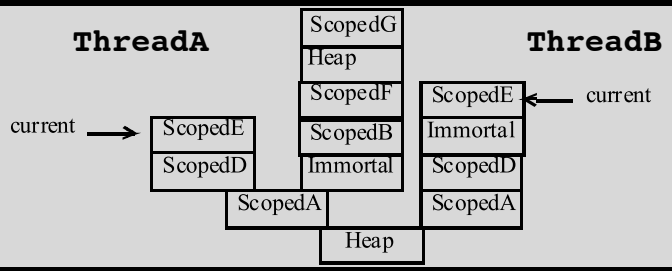
Sharing Memory Areas

- Multiple schedulable objects can access the same memory areas
- Consequently, the cactus stacks for each schedulable objects are linked together
- Here, two real-time threads (**ThreadA** and **ThreadB**) have active scoped memory stacks



Sharing Memory Areas

- Suppose **ThreadA** wishes to enter **ScopedE**; it cannot do so directly because **ScopedE** is active and has parent **ScopedD**
- To access **ScopedE**, **ThreadA** must move to **ScopedA** (using **executeInArea**), then enter **ScopedD** and **ScopedE**



Entering and Joining Scoped Memory Areas

- Memory areas can be used cooperatively or competitively
- In cooperative use, the schedulable objects aim to be active in a scoped memory simultaneously
  - they use the area to communicate shared objects
  - when they leave, the memory is reclaimed
- In competitive use, they aim for efficient use of memory
  - the schedulable objects are trying to take their memory from the same area but are not using the area for the communication
  - it is usually required for only one schedulable to be active in the area at a time
  - the intention is that the memory can be reclaimed when each of the schedulable objects leave the area

Competitive Use

- To ensure that the area becomes inactive use join.
- If timeout expires, the memory area is entered
- It is difficult to know if the timeout did expire

```
public abstract class ScopedMemory extends MemoryArea {
  public int getReferenceCount();
  public void join() throws InterruptedException;
  public void join(HighResolutionTime time) throws …
  public void joinAndEnter() throws InterruptedException;
  public void joinAndEnter(HighResolutionTime time) throws…
  public void joinAndEnter(java.lang.Runnable logic) throws…
  public void joinAndEnter(java.lang.Runnable logic,
                           HighResolutionTime time) throws…
```


Summary

- The lack of confidence in RTGC is one of the main inhibitors to the widespread use of Java in real-time and embedded systems
- The RTSJ has introduced an additional memory management facility based on the concept of memory areas
- There are two types on non-heap memory areas
  - immortal memory which is never subject to garbage collection
  - scoped memory in to which schedulable objects can enter and leave; when there are no schedulable objects active in a scoped memory area, all the objects are destroyed and their memory reclaimed.
- Due to the variety of memory areas, the RTSJ has strict assignment rules between them in order to ensure that dangling references do not occur

Roadmap

- Overview of the RTSJ
- **Memory Management**
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control
- Schedulability Analysis
- Conclusions

Memory Management

- Lecture aims:
- To motivate the need for portal objects and provide an example of their use
- To consider the real-time issues of scoped memory areas

Recall from previous lecture

- Scoped memory areas can be used two modes of operation: cooperative or competitive
- In cooperative use, the SOs are active in a scoped memory area simultaneously
  - they use the area to communicate shared objects
  - when they leave, the memory is reclaimed
- In competitive use, the goal is to make the most efficient use of memory
  - only one schedulable is active in the memory area at one time
  - the intention is that the memory can be reclaimed when each of the schedulable objects leave the area

Portals

- For cooperative sharing where there is no relationship between the SOs, how can SOs share objects created in a memory area?
- To share an object requires each SO to have a reference to it
  - A reference to an object can only be stored in an object in the same scoped area or in an object in a nested scoped area
  - It cannot be stored in the immortal or heaped memory area
- Consequently, unless there is some relationship between the SOs, one SO cannot pass a reference to an object it has just created to another SO

Portals

- Portals solve this problem; each memory area can have one object which can act as a gateway into that memory area
- SOs can use this mechanism to facilitate communication

```
public abstract class ScopedMemory extends MemoryArea {
  ...
  public Object getPortal();
  public void setPortal(Object o);
}
```

Portal Example

- Consider, an object which controls the firing of a missile
- For the missile to be launched two independent RT threads must call its fire method each with its own authorization code

```
public class FireMissile {
   public FireMissile();
   public boolean fire1(final String authorizationCode);
   public boolean fire2(final String authorizationCode);
}
```


Portal Example

- The two threads call `fire1` and `fire2` respectively
- Whichever calls in first has it authorization code checked and is held until the other thread calls its fire method
- If both threads have valid authorization codes, the missile is fired and the methods returns true, otherwise the missile is not fired and false is returned.
- In order to implement the fire methods, assume that objects need to be created in order to check the authorization

```
class Decrypt {
  boolean confirm(String code){/*check authorization*/}
}
```


Portal Example

- To synchronize, the two RT threads must communicate

```
class BarrierWithParameter {
  BarrierWithParameter(int participants);
  synchronized boolean wait(boolean go);
}
```

The threads call wait indicating whether they wish to fire. If both pass true, wait returns true

Portal Example

- The goal is to implement `FireMissile` without using extra memory other than that required to instantiate the class
- All memory needed by the fire methods should be created in scoped memory which can be reclaimed when the methods are inactive
- In order to implement the required firing algorithm, the class needs two `Decrypt` objects and one `BarrierWithParameter` object

Portal Example

```
public class FireMissile {
  public FireMissile() {
    ...
    SizeEstimator s = new SizeEstimator();
    s.reserve(Decrypt.class, 2);
    s.reserve(BarrierWithParameter.class, 1);
    shared = new LTMemory(s.getEstimate(),
                          s.getEstimate());
  }
  private LTMemory shared;
  ...
}
```


Portal Example

- Both threads need to access a `BarrierWithParameter` object, they therefore must enter into the same memory area
- The shared memory area must have a single parent
- As this class is unaware of the scoped stacks of the calling threads, it needs to create new scoped memory stacks
- It does this by using the `executeInArea` method to enter into immortal memory
- The **`run`** method can now enter into the scoped memory area.

The alternative approach would be to assume the calling threads have empty (or the same) scope stacks

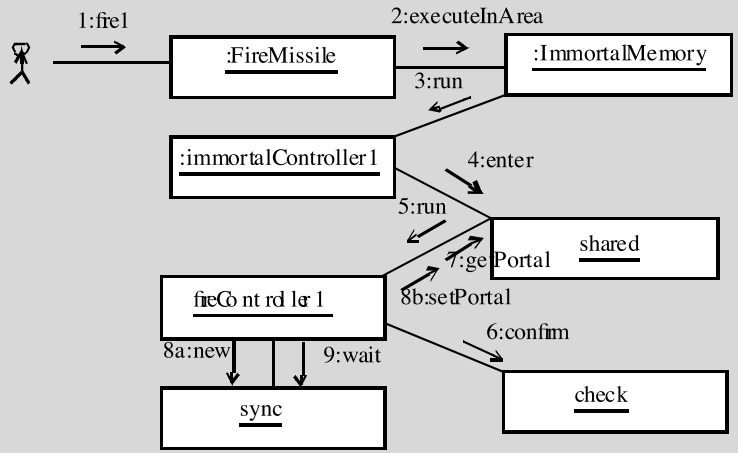
Portal Example

```
class FireMissile {
  private LTMemory shared;
  private FireAction fireController1, fireController2;
  private ImmortalAction immController1,
                         immController2;
  FireMissile() {
    fireController1 = new FireAction();
    fireController2 = new FireAction();
    immController1 = new ImmortalAction();
    immController2 = new ImmortalAction();
    SizeEstimator s = new SizeEstimator();
    s.reserve(Decrypt.class,2);
    s.reserve(BarrierWithParameter.class, 1);
    shared = new LTMemory(s.getEstimate(),
                          s.getEstimate());
  }
```


Portal Example

```
class FireMissile {
  class FireAction implements Runnable {
    String authorization;  boolean result;
    public void run() { /*coordinate missile firing */ }
  }
  class ImmortalAction implements Runnable {
    FireAction fireController;
    public void run(){ shared.enter(fireController);  }
  }
  boolean fire1(final String authorizationCode) {
    try {
      immortalController1.fireController = fireController1;
      fireController1.authorization = authorizationCode;
      ImmortalMemory.instance().executeInArea(immController1);
      return fireController1.result;
    }
    catch (InaccessibleMemoryArea ma)  {}
  }   // similarly for fire2
```

Portal Example



Portal Example

- Once in the memory area, the required objects can be created
- However, there is a problem with the shared `BarrierWithParameter` object.
  - In order for both threads to access it they must have a reference to it
  - Usually, the reference would be stored in a field declared at the class level
  - However, this is not possible, as it would break the RTSJ assignment rules as the object is in an inner scope

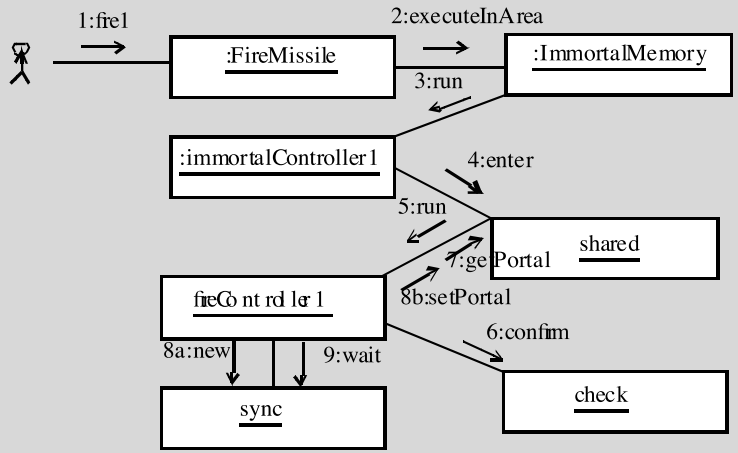
Portal Example

- The first thread to arrive attempts to obtain the portal object for the shared memory region
  - This is null, so it creates the `BarrierWithParameter` object and sets it up as the portal object
- The next thread to arrive can then obtain the required reference
- When both threads leave the shared memory area, all the memory is reclaimed
- However, there is a race condition between checking for the existence of the portal object and setting a newly created one
- Consequently, a lock is needed. One lock that is available at this stage is the lock associated with the memory object itself!

Portal Example

```
class FireAction implements Runnable {
  String authorization;  boolean result;
  public void run() {
    BarrierWithParameter sync;
    Decrypt check = new Decrypt();
    boolean confirmed = check.confirm(authorization);
    synchronized(RealtimeThread.getCurrentMemoryArea()) {
      sync = (BarrierWithParameter)shared.getPortal();
      if(sync == null)
        shared.setPortal(sync = new BarriewWithParameter(2));
    }
    result = sync.wait(confirmed);
  }
```

Portal Example



Using Scoped Memory Areas

- When a class `C` needs temporary memory should it take responsibility for creating and entering a scoped area?
- If `C` is intended for sequential access and is used concurrently, a `ScopedCycleException` may be thrown
  - The client SOs must either ensure that the scoped memory stack is empty or, ensure that all clients call with same stack. Clients need to have details of when a class is using scoped areas
- If `C` targets concurrent access:
  - Clients must either have the same scoped memory stacks, or `C` must "execute in" a heap or an immortal area first
  - if `C` is in the heap, it can't be used by a no-heap client; if `C` chooses immortal memory, named inner classes must be used, otherwise every time it instantiate an anonymous class, memory will be created in immortal memory and this will not be reclaimed

Real-Time Issues

- Entry to scoped memory
  - On entry into a scoped memory region, a SO may be blocked if it uses one of the `join` or `joinAndEnter` methods
  - The duration of this blocking may be difficult to bound (unless timeouts are used, but these have their own problems)
  - If the SOs using the scoped memory do not themselves block when they have entered the area, the blocking time will be the maximum time that lower priority SOs take to execute the associated `run` methods
  - When a SO attempts to enter into a previously active scoped memory area, it may also have to wait for the memory to be reclaimed (finalizers run, etc.)

Real-Time Issues

- Predictable scoped memory allocation
  - Memory allocation in a scoped memory should be predictable from the details of the implementation and the objects being created
  - It will consist of two components: the time taken to allocate the space for the objects (which will be proportional to their sizes, for LTMemory), and the time taken to execute the constructors for the objects created

Real-Time Issues

- Exit from scoped memory
  - The RTSJ gives a great deal of freedom on when the memory used in a scoped memory area is reclaimed
  - An implementation might decide to reclaim the memory immediately the reference count becomes zero or sometime after it becomes zero, but no later than when a new SO wishes to enter it
  - Hence, a SO on one occasion may leave the scoped memory and suffer no impact; on another occasion, it is the last object to leave and reclamation occurs immediately; alternatively it may suffer when it first enters the scope

Real-Time Issues

- Object finalization
  - Whenever scoped memory is reclaimed, the objects that have been created must have their finalization code executed before reclamation
  - The time for this to occur must be accounted for in any analysis.
- Garbage collector scans
  - The impact, if any, of the GC scanning the scoped memory area looking for heap references when those memory areas are being reclaimed concurrently

Summary

- Portal objects can be used to facilitate communication between schedulable objects using the same scoped memory area
- There is little doubt that non-heap memory management is one of the most complicated areas of the RTSJ, and one that has a major impact on the overheads of the virtual machine

Roadmap

- Overview of the RTSJ
- Memory Management
- **Clocks and Time**
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

Clocks and Time

Lecture aims

- To provide some background on clocks and time
- To explain absolute and relative time values
- To consider the support for real-time clocks
- To give some simple examples

Introduction I

- Java only supports the notion of a wall clock (calendar time); for many applications, a clock based on UTC is sufficient
- However real-time systems often require
  - A monotonic clock which progresses at a constant rate and is not subject to the insertion of extra ticks to reflect leap seconds (as UTC is). A constant rate is needed for control algorithms which want to executed on a regular basis. Many clocks are also relative to system startup and used to measure the passage of time, not calendar time
  - A count down clock which can be paused, continued or reset (e.g the clock which counts down to the launch of the Space Shuttle)
  - A CPU execution time clock which measures the amount of CPU time that is being consumed by a particular thread or object

Introduction II

- All of the above clocks need a resolution which is finer than the millisecond level
- They may all be based on the same underlying physical clock, or be supported by separate clocks
- Where more than one clock is provided, issues of the relationship between them may be importance; in particular, whether their values can drift

Basic Model

- A hierarchy of time classes rooted at **`HighResolutionTime`**
- This abstract class has three subclasses:
  - one which represents absolute time
  - one which represents relative time
  - and one which represents rational time
- The intention is to allow support for time values down to the nanosecond accuracy
- Clocks are supported through an abstract **`Clock`** class

High Resolution Time I

```
public abstract class HighResolutionTime
       implements Comparable, Cloneable {

  public abstract AbsoluteTime absolute(Clock clock);
  public int compareTo(HighResolutionTime time);
  public boolean equals(HighResolutionTime time);
  public final long getMilliseconds();
  public final int getNanoseconds();
  public abstract RelativeTime relative(Clock clock);
  public void set(HighResolutionTime time);
  public void set(long millis, int nanos);
  public static void waitForObject(Object target,
    HighResolutionTime time) throws InterruptedException;
```


High Resolution Time II

- The abstract methods allow time types that are relative to be re-expressed as absolute time values and vice versa.
- They allow clocks associated with the values to be changed
  - *absolute to absolute* — value returned has the same millisecond and nanosecond components as the encapsulated time value
  - *absolute to relative* — value returned is the value of the encapsulated absolute time minus the current time as measured from the given clock parameter
  - *relative to relative* — value returned has the same millisecond and nanosecond components as the encapsulated time value
  - *relative to absolute* — value returned is the value of current time as measured from the given clock parameter plus the encapsulated relative time

High Resolution Time III

- Changing the clock associated with a time value is potentially unsafe, particularly for absolute time values
  - This is because absolute time values are represented as a number of milliseconds and nanoseconds since an epoch
  - Different clocks may have different epochs.
- The `waitForObject` does not resolve the problem of determining if the schedulable object was woken by a notify method call or by a timeout
  - It does allow both relative and absolute time values to be specified.

Absolute Time

```
public class AbsoluteTime extends HighResolutionTime {
  public AbsoluteTime();
  public AbsoluteTime(AbsoluteTime time);
  public AbsoluteTime(Date date);
  public AbsoluteTime(long millis, int nanos);

  public AbsoluteTime absolute(Clock clock);

  public AbsoluteTime add(long millis, int nanos);
  public final AbsoluteTime add(RelativeTime time);

  public java.util.Date getDate();
  public RelativeTime relative(Clock clock);
  public void set(Date date);

  public RelativeTime subtract(AbsoluteTime time);
  public AbsoluteTime subtract(RelativeTime time);
```

Note that an absolute time can have either a positive or a negative value and that, by default, it is relative to the epoch of the real-time clock

Relative Time

```
public class RelativeTime extends HighResolutionTime {
  public RelativeTime();
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);

  public AbsoluteTime absolute(Clock clock);
  public RelativeTime add(long millis, int nanos);
  public RelativeTime add(RelativeTime time);
  public RelativeTime relative(Clock clock);
  public RelativeTime subtract(RelativeTime time)
```

A relative time value is an interval of time measured by some clock; for example 20 milliseconds

Clocks

- `Clock` is the abstract class from which all clocks are derived
- RTSJ allows many different types of clocks; eg, an execution-time clock which measures the amount of execution time consumed
- There is real-time clock which advances monotonically
  - can never go backwards
  - should progress uniformly and not experience insertion of leap ticks

```
public abstract class Clock {
  public Clock();
  public static Clock getRealtimeClock();
  public abstract RelativeTime getResolution();
  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);
  public abstract void setResolution(RelativeTime resolutn);
```


Example: measuring elapse time

```
Clock clock = Clock.getRealtimeClock();
AbsoluteTime oldt = clock.getTime();
...
AbsoluteTime newt = clock.getTime();
RelativeTime inter = newt.subtract(oldt);
```

This would only measure the approximate elapse time, as the schedulable object executing the code may be pre-empted after it has finished the computation and before it reads the new time

Example

- A launch clock is a clock which is initiated with a relative and an absolute time value
- The absolute time is the time at which the clock is to start ticking; the relative time is the duration of the countdown
- The count down can be stopped, restarted, or reset
- The class extends the `Thread` class
- The constructor saves the start time, the duration and the clock
- The resolution of the count down is one second

LaunchClock I

```
class LaunchClock extends Thread {
  private AbsoluteTime start;
  private RelativeTime remaining, tick;
  private Clock clock;
  private boolean counting = true;
  private boolean go;

  LaunchClock(AbsoluteTime at,RelativeTime count) {
    start = at; remaining = count;
    clock = Clock.getRealtimeClock();
    tick = new RelativeTime(1000,0);
  }
  RelativeTime getResolution(){ return tick; }
  synchronized AbsoluteTime getCurrentLaunchTime(){
    return new AbsoluteTime(clock.getTime().add(remaining))
    // assumes started and ticking
  }
```


LaunchClock II

```
synchronized void stopCountDown(){
  counting = false;  notifyAll();
}
synchronized void restartCountDown() {
  counting = true;  notifyAll();
}
synchronized void resetCountDown(RelativeTime to){
  remaining = to;
}

synchronized void launch() throws Exception{
  while(!go)
      try {  wait(); }
      catch(InterruptedException ie) {
          throw new Exception("Launch failed");
      }// Launch is go
}
```


LaunchClock V

```
public void run(){
  try  {
    synchronized(this) {
      while(clock.getTime().compareTo(start) < 0)
         HighResolutionTime.waitForObject(this, start);
      while(remaining.getMilliseconds() > 0) {
          while(!counting) wait();
          HighResolutionTime.waitForObject(this, tick);
          remainin.set(remaining.getMilliseconds() -
                       tick.getMilliseconds());
      }
      go = true;
      notifyAll();
    }
  } catch(InterruptedException ie) {}
}
```


Summary

- Clocks and time are fundamental to any real-time system
- The RTSJ has augmented the Java facilities with high resolution time types and a framework for supporting various clocks
- A real-time clock is guaranteed to be supported by any compliant RTSJ implementation
- This is a monotonically non-decreasing clock

Roadmap

- Overview of the RTSJ
- Memory Management
- Clocks and Time
- **Scheduling and Schedulable Objects**
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

Scheduling and Schedulable Objects

Lecture aims:

- To give an overview of fixed priority schedule
- To present the RTSJ Basic Scheduling Model

Introduction

- Real-time systems must be able to interact with their environment in a timely and predictable manner
- Designers must engineer analyzable systems whose timing properties can be predicted and mathematically proven correct
- Scheduling is the ordering of thread executions so that the underlying hardware resources and software resources (shared data objects) are efficiently and predictably used

Scheduling

- In general, scheduling consists of three components
  - an algorithm for ordering access to resources (scheduling policy)
  - an algorithm for allocating the resources (scheduling mechanism)
  - a means of predicting the worst-case behaviour of the system when the policy and mechanism are applied (schedulability analysis or feasibility analysis)
- Once the worst-case behavior of the system has been predicted, it can be compared with the system's timing requirements to ensure that all deadlines will be met

Fixed Priority Scheduling: Policy

- FPS requires
  - statically allocating schedulable objects to processors
  - ordering the execution of schedulable objects on a single processor according to a priority
  - assigning priorities to schedulable objects at their creation time — although no particular priority assignment algorithm is mandated, usually the shorter the deadline, the higher the priority
  - priority inheritance when accessing resources

FPS: Mechanism and Analysis

- **Mechanism**: FPS requires preemptive priority-based dispatching of processes — the processing resource is always given to the highest priority runnable schedulable object
- **Feasibility analysis**: There are many different techniques for analyzing whether a fixed priority-based system will meet its deadlines. Perhaps the most flexible is response time analysis

Information Needed for Analysis

- View the system as consisting of a number of schedulable objects
- Each schedulable object is characterized by its
  - release profile
  - processing cost per release
  - other hardware resources needed per release
  - software resources per release
  - deadline
  - value

Release Profile

- Typically after a schedulable object is started, it waits to be **released**
- When released it performs some computation and then waits to be released again (its completion time)
- The release profile defines the frequency with which the releases occur; they may be time-triggered (periodic) or event triggered
- Event triggered releases are further classified into sporadic (meaning that they are irregular but with a minimum inter-arrival time) or aperiodic (meaning that no minimum inter-arrival assumptions can be made)

Processing cost per release

- This is a measure of how much time is required to execute the computation associated with the schedulable object's release
- This may be a worst-case value or an average value depending on the feasibility analysis

Other resources

- Hardware: (other than the processor)
  - For networks, it is usually the time needed or bandwidth required to send messages
  - For memory, it is the amount of memory required by the schedulable objects (and types of memory)
- Software resources: a list of the non shareable resources that are required and the cost of using each resource
  - Access to non-shareable resources is a critical factor when performing schedulability analysis
  - Non shareable resources are usually non preemptible. Consequently when a schedulable tries to acquire a resource it may be blocked if the resource is already in use
  - This blocking time has to be taken into account

Deadline

- The time which the schedulable object has to complete the computation associated with each release
- As usually only a single deadline is given, the time is a relative value rather than an absolute value

Value

- A metric which indicates the schedulable objects contribution to the overall functionality of the application. It may be
  - a very coarse indication (e.g, safety critical, mission critical, non critical)
  - a numeric value giving a measure for a successful meeting of a deadline
  - a time-valued function which takes the time at which the schedulable object completes and returns a measure of the value (for those systems where there is no fixed deadline)

Online versus Off-line Analysis

- A key characteristic of schedulability (feasibility) analysis is whether the analysis is performed off-line or on-line
- For safety critical systems, where the deadlines associated with schedulable objects must always be met, off-line analysis essential
- Other systems do not have such stringent timing requirements or do not have a predictable worst case behavior; on-line analysis may be appropriate or, the only option available
- These systems must be able to tolerate schedulable objects not being schedulable and offer degraded services
- Furthermore, they must be able to handle deadlines being missed or situations where the assumed worst-case loading scenario has been violated

Basic Model

- The RTSJ provides a framework from within which on-line feasibility analysis of priority-based systems can be performed for single processor systems
- The specification also allows the real-time JVM to monitor the resources being used and to fire asynchronous event handlers if those resources go beyond that specified by the programmer
- The RTSJ introduces the notion of a **schedulable object** rather than considering just threads
- A schedulable object is any object which implements the `Schedulable` interface

Schedulable Objects Attributes I

- **`ReleaseParameters`**
  - the processing cost for each release
  - its deadline
  - if the object is periodic or sporadic then an interval is also given
  - event handlers can be specified for the situation where the deadline is missed or the processing resource consumed is greater than the cost specified
  - There is no requirement to monitor the processing time consumed by a schedulable object

Schedulable Objects Attributes II

- **`SchedulingParameters`**
  - an empty class
  - subclasses allow the priority of the object to be specified and, potentially, its importance to the overall functioning of the application
  - although the RTSJ specifies a minimum range of real-time priorities (28), it makes no statement on the importance parameter

Schedulable Objects Attributes III

- **MemoryParameters**
  - the maximum amount of memory used by the object in an associated memory area
  - the maximum amount of memory used in immortal memory
  - a maximum allocation rate of heap memory.

The Schedulable Interface

- Three groups of methods
  - Methods which communicate with the scheduler and result in the scheduler adding/removing the schedulable object from the list of objects it manages, or changing the parameters associated with it
    - the scheduler performs a feasibility test on the objects it manages
  - Methods which get/set the parameters associated with the schedulable object
    - If the parameter object set is different from the one currently associated with the schedulable object, the previous value is lost and the new one will be used in any future feasibility analysis
  - Methods which get/set the scheduler

Schedulable Interface

```
public interface Schedulable extends Runnable {
  public boolean addIfFeasible();
  public boolean addToFeasibility();
  public boolean removeFromFeasibility();
  public boolean setIfFeasible(ReleaseParameters release,
                 MemoryParameters memory);
  public boolean setReleaseParametersIfFeasible(
                 ReleaseParameters release);
  public boolean setSchedulingParametersIfFeasible(
                 SchedulingParameters sched);
 public MemoryParameters getMemoryParameters();
  public void setMemoryParameters(MemoryParameters memory);
  public ReleaseParameters getReleaseParameters();
  public void setReleaseParameters(ReleaseParameters rels);
  public SchedulingParameters getSchedulingParameters();
  public void setSchedulingParameters(SchedulingParameters s)
  public Scheduler getScheduler();
  public void setScheduler(Scheduler s);
```


Scheduler

```
public abstract class Scheduler  {
  protected Scheduler();
  public abstract boolean setIfFeasible(Schedulable s,
         ReleaseParameters r, MemoryParameters m);
  public abstract boolean setIfFeasible(
         Schedulable s, ReleaseParameters r,
         MemoryParameters m, ProcessingGroupParameters g);
  public abstract void fireSchedulable(Schedulable s);
  public static Scheduler getDefaultScheduler();
  public abstract String getPolicyName();
  public static void setDefaultScheduler(Scheduler s);
  protected abstract boolean addToFeasibility(Schedulable s)
  public abstract boolean isFeasible();
  protected abstract boolean removeFromFeasibility(
         Schedulable schedulable);
}
```

The Priority Scheduler: Policy

- Orders the execution of schedulable objects on a single processor according to a priority
- Supports a real-time priority range of at least 28 unique priorities (the larger the value, the higher the priority)
- Allows the programmer to define the priorities (say according to the relative deadline of the schedulable object)
- Allows priorities may be changed at run time
- Supports priority inheritance or priority ceiling emulation inheritance for synchronized objects (covered later)

The Priority Scheduler: Mechanism

- Supports pre-emptive priority-based dispatching of schedulable objects — the processing resource is always given to the highest priority runnable schedulable object
- Does not defined where in the run queue (associated with the priority level), a pre-empted object is placed; however, a particular implementation is required to document its approach
- Places a blocked schedulable object which becomes runnable, or has its priority changed at the back of the run queue associated with its (new) priority
- Places a thread which performs a yield operation at the back of the run queue associated with its (new) priority

The Priority Scheduler: Feasibility Analysis

- Requires no particular analysis to be supported
- Default analysis assumes an adequately fast machine

The PriorityScheduler Class

```
public class PriorityScheduler extends Scheduler {
  public static final int MAX_PRIORITY,MIN_PRIORITY;
  protected PriorityScheduler();

  protected boolean addToFeasibility(Schedulable s);
  public boolean isFeasible();
  protected boolean removeFromFeasibility(Schedulable s);
  public boolean setIfFeasible(Schedulable s,
     ReleaseParameters r, MemoryParameters m);
  public void fireSchedulable(Schedulable s);
  public int getMaxPriority();
  public static int getMaxPriority(Thread thread);
  public int getMinPriority();
  public static int getMinPriority(Thread thread);
  public int getNormPriority();
  public static int getNormPriority(Thread thread);
  public String getPolicyName();
```

Summary

- Scheduling is the ordering of thread execution so that hardware and software resources are efficiently and predictably used
- The only scheduler mandated is a fixed priority scheduler (FPS)
- Scheduling policy: FPS requires
  - statically allocating schedulable objects to processors
  - ordering the execution of schedulable objects according to a priority
  - assigning priorities to schedulable objects at their creation time
  - priority inheritance when accessing resources.
- Scheduling mechanism: FPS requires preemptive priority-based dispatching of processes
- Feasibility analysis: RTSJ does not mandate any schedulability analysis technique

Roadmap

- Overview of the RTSJ
- Memory Management
- Clocks and Time
- **Scheduling and Schedulable Objects**
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

The Parameter Classes

- Each schedulable objects has several associated parameters
- These parameters are tightly bound to the schedulable object and any changes to the parameters can have an immediate impact on the scheduling of the object or any feasibility analysis performed by its scheduler
- Each schedulable object can have only one set of parameters associated with it
- However, a particular parameter class can be associated with more than one schedulable object
- In this case, any changes to the parameter objects affects all the schedulable objects bound to that parameter

Release Parameters I

- Release parameters characterize
  - how often a schedulable object runs
  - the worst case processor time needed for each run
  - a relative deadline by which each run must have finished
- There is a close relationship between the actions that a schedulable object can perform and its release parameters
- E.g., the `RealtimeThread` class has a method called `waitForNextPeriod`; however a RT thread can only call this methods if it has `PeriodicParameters` associated with it
- This allows a thread to change its release characteristics and hence adapt its behavior

The ReleaseParameter Class

```
public class ReleaseParameters {

  protected ReleaseParameters();
  protected ReleaseParameters(RelativeTime cost,
              RelativeTime deadline,
              AsyncEventHandler overrunHandler,
              AsyncEventHandler missHandler);

  public RelativeTime getCost();
  public AsyncEventHandler getCostOverrunHandler();
  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler();
  public boolean setIfFeasible(RelativeTime cost,
                RelativeTime deadline);
```


Release Parameters II

- The minimum information that a scheduler will need for feasibility analysis is the **cost** and **deadline**
- **cost**: a measure of how much CPU time the scheduler should assume that the scheduling object will require for each release
  - This is dependent on the processor on which it is being executed; consequently, any programmer-defined value will not be portable
- **deadline**: the time from a release that the scheduler object has to complete its execution
- `overrunHandler`: the asynchronous event handler that should be released if the schedulable object overruns
- `missHandler` is released if the schedulable object is still executing when its deadline arrives

Cost Enforcement

- If cost monitoring is supported:
  - the RTSJ requires that the priority scheduler gives a schedulable object a CPU budget of no more than its cost value on each release
  - if a schedulable objects overrun its cost budget, it is automatically descheduled immediately
  - it will not be re-scheduled until either its next release occurs (in which case its budget is replenished) or its cost value is increased

The PeriodicParameters Class

- For schedulable objects which are released on a regular basis
- The start time can be an `AbsoluteTime` or a `RelativeTime` and indicates when the schedulable should be first released
- For a real-time thread, the actual first release time is given by
  - if start is a RelativeTime value
    - Time of invocation of start method + start
  - if start is an AbsoluteTime value
    - Max of (Time of invocation of start method, start)
- A similar formula can be given for an async event handler
- The deadline for a schedulable with periodic parameters is measured from the time it is released not when it is started/fired

The PeriodicParameters Class

```
public class PeriodicParameters extends ReleaseParameters {
  public PeriodicParameters(
         HighResolutionTime start, RelativeTime period,
         RelativeTime cost, RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);
  public RelativeTime getPeriod();
  public HighResolutionTime getStart();
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);
  public boolean setIfFeasible(RelativeTime period,
          RelativeTime cost, RelativeTime deadline);
}
```


The AperiodicParameters Class I

```
class AperiodicParameters extends ReleaseParameters {
  public AperiodicParameters(
         RelativeTime cost, RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);
  public boolean setIfFeasible(RelativeTime cost,
                               RelativeTime deadline);
```

The deadline of an aperiodic schedulable object can never be guaranteed, why? Hence, the deadline attribute should be interpreted as a target completion time rather than a strict deadline.

The AperiodicParameters Class II

- Schedulable object with aperiodic parameters are released at irregular intervals
- When an aperiodic schedulable object is released for the first time, it becomes runnable and is scheduled for execution.
- Before it completes its execution, it may be released again.
- It is necessary for an implementation to queue the release events to ensure that they are not lost
- The programmer is able to specify the length of this queue and the actions to be taken if the queue overflows

Roadmap

- Overview of the RTSJ
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- **Asynchronous Events and Handlers**
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

Asynchronous Event and their Handlers

Lecture aims:

- To motivate the needs for asynchronous events
- To introduce the basic RTSJ model
- To show how to implement Asynchronous events with parameters
- To consider event handlers and program termination

Time Triggered Systems

- Within embedded system design, the controllers for real world objects such as conveyor belts, engines and robots are usually represented as threads
- The interaction between the real world objects and their controllers can be either time-triggered or event-triggered
- In a time-triggered systems, the controller is activated periodically; it senses the environment in order to determining the status of the real-time objects it is controlling
- It writes to actuators which affect the behavior of the objects
- *E.g, a robot controller determines the position of the robot via a sensor and decide that it must cut the power to a motor thereby bringing the robot to a halt*

Event Triggered Systems

- In an event triggered system, sensors in the environment are activated when the real world object enters into certain states
- The events are signaled to the controller via interrupts
- *Eg, a robot may trip a switch when it reaches a certain position; this is a signal to the controller that the power to the motor should be turn off, thereby bringing the robot to a halt*
- Event triggered systems are often more flexible whereas time triggered systems are more predictable
- In either case, the controller is usually represented as a thread

Threads Considered Harmful

- There are occasions where threads are not appropriate:
  - the external objects are many and their control algorithms are simple and non-blocking
    - using a thread per controller leads to a proliferation of threads along with the associated per-thread overhead
  - the external objects are inter-related and their collective control requires significant communication/synchronization between controllers
    - complex communication and synchronization protocols are needed which can be difficult to design correctly and may lead to deadlock or unbounded blocking

Event-based Programming

- An alternative to thread-based programming
- Each event has an associated handler; when events occur, they are queued and one or more server thread takes an event from the queue and executes it associated handler
- When the handler has finished, the server takes another event from the queue, executes the handler and so on
- There is no need for explicit communication between the handlers as they can simply read/write from shared objects without contention

Disadvantages of Events

- Disadvantages of controlling all external objects by event handlers:
  - it is difficult to have tight deadlines associated with event handlers as a long-lived and non-blocking handler must terminate before the server can execute any newly arrived high-priority handler, and
  - it is difficult to execute the handlers on a multiprocessor system as the handlers assume no contention for shared resources

The RTSJ Approach

- Attempt to provide the flexibility of threads and the efficiency of event handling via the notion of real-time asynchronous events (AE) and associated handlers (AEH)
- AEs are data-less happenings either fired by the program or associated with the occurrence of interrupts in the environment
- One or more AEH can be associated with a single event, and a single AEH can be associated with one or more events
- The association between AEHs and AEs is dynamic
- Each AEH has a count of the number of outstanding firings. When an event is fired, the count is atomically incremented
- The handler is then released

The AE Class

```
public class AsyncEvent {

  public void addHandler(AsyncEventHandler handler);
  public void removeHandler(AsyncEventHandler handler);
  public void setHandler(AsyncEventHandler handler);
  public boolean handledBy(AsyncEventHandler target);
  public void bindTo(String happening)
          throws UnknownHappeningException;
  public void unBindTo(String happening)
          throws UnknownHappeningException;
  public ReleaseParameters createReleaseParameters();
  public void fire();
}
```

The AEH Class

```
public class AsyncEventHandler implements Schedulable {
  public AsyncEventHandler();
  public AsyncEventHandler(Runnable logic);
  public AsyncEventHandler(boolean nonheap);
  public AsyncEventHandler(boolean nonheap, Runnable logic);
  public AsyncEventHandler(SchedulingParameters s,
     ReleaseParameters r, MemoryParameters m, MemoryArea a);
  ... // various other combinations

  protected final int getAndClearPendingFireCount();
  protected int getAndDecrementPendingFireCount();
  protected int getAndIncrementPendingFireCount();
  protected final int getPendingFireCount();

  public void handleAsyncEvent();
  public final void run();
```


ASEH

- A set of protected methods allow the fire count to be manipulated
- They can only be called by creating a subclass and overriding the `handlerAsyncEvent` method
- The default code for `handleAsyncEvent` is null unless a `Runnable` object has been supplied with the constructor, in which case, the `run` method of the `Runnable` object is called
- The `run` method of the `AsyncEventHandler` class itself is the method that will be called by the underlying system when the object is first released
- It will call `handleAsyncEvent` repeatedly whenever the fire count > 0

Bound Event Handlers

- Both event handlers and threads are schedulable objects
- Threads provide the vehicles for execution of event handlers
- Therefore, an event handler is to a server real-time thread
- For `AsyncEventHandler` objects binding is dynamic
- There is some overhead with doing this and `BoundEvent-Handler` objects are supplied to eliminate this overhead
- Bound event handlers are permanently associated with a dedicated server real-time thread

BASEH

```
class BoundAsyncEventHandler extends AsyncEventHandler {
  public BoundAsyncEventHandler();
  public BoundAsyncEventHandler(
      SchedulingParameters scheduling,
      ReleaseParameters release, MemoryParameters memory,
      MemoryArea area, ProcessingGroupParameters group,
      boolean nonheap);
}
```

Timers I

- The abstract `Timer` class defines the base class from which timer events can be generated
- All timers are based on a clock; a null clock values indicates that the `RealtimeClock` should be used
- A timer has a time at which it should fire; that is release its associated handlers
- This time may be an absolute or relative time value
- If no handlers have been associated with the timer, nothing will happen when the timer fires

Timer II

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

  public ReleaseParameters createReleaseParameters();
  public void destroy();
  public void disable();
  public void enable();
  public Clock getClock();
  public AbsoluteTime getFireTime();
  public void reschedule(HighResolutionTime time);
  public void start();
  public boolean stop();
}
```


Timers III

- Once created a timer can be explicitly destroyed, disabled (which allows the timer to continue counting down but prevents it from firing) and enabled (after it has been disabled)
- If a timer is enabled after its firing time has passed, the firing is lost
- The `reschedule` method allows the firing time to be changed
- Finally the `start` method, starts the timer going
- Any relative time given in the constructor is converted to an absolute time at this point; if an absolute time was given in the constructor, and the time has passed, the timer fires immediately

One Shot Timer

```
public class OneShotTimer extends Timer {
  public OneShotTimer(HighResolutionTime fireTime,
                      AsyncEventHandler handler);
  // assumes the default real-time clock

  public OneShotTimer(HighResolutionTime fireTime,
                      Clock clock, AsyncEventHandler handler);
  // fireTime is based on the clock parameter
```

Periodic Timer

```
public class PeriodicTimer extends Timer {
  public PeriodicTimer(HighResolutionTime start,
       RelativeTime interval, AsyncEventHandler handler);
  public PeriodicTimer(HighResolutionTime start,
       RelativeTime interval, Clock clock,
       AsyncEventHandler handler);

  public ReleaseParameters createReleaseParameters();
  public void fire(); // deprecated
  public AbsoluteTime getFireTime();
  public RelativeTime getInterval();
  public void setInterval(RelativeTime interval);
```


Example: A Panic Button I

- Consider a computerized hospital intensive care unit
- A patient's vital signs are automatically monitored and if there is cause for concern, a duty doctor is paged automatically
- There is also a bed-side "panic" button which can be pushed by the patient or a visitor should they feel it is necessary
- The "panic" button is mainly for the patient/visitor's benefit; if the patient's life is really in danger, other sensors will have detected the problem

Panic Button II

- To be on the safe side, the system responds to a press of the panic button in the following way:
  - if there is no paging of the doctor in the last five minutes, test to see if the patient's vital signs are strong, if they are weak, the duty doctor is paged immediately
  - if the vital signs are strong and a nurse has been paged in the last ten minutes, the button is ignored
  - if the vital signs are strong and a nurse has not been paged in the last ten minutes, the duty nurse is paged

Panic Button III

- The press of the "panic" button is an external happening
- It is identified by the string "PanicButton"
- A pager is represented by an asynchronous event; `dutyDoctor` and `dutyNurse` are the events for the doctor's and nurse's pages respectively
- A firing of the appropriate event results in the associated handler initiating the paging call
- First the event handler for the "panic button" can be defined
- The constructor attaches itself to the "panic button" event
- Note also that the handler clears the fire count as it is possible that the patient/visitor has pressed the button multiple times

PanicButtonHandler I

```
import javax.realtime.*;
class PanicButtonHandler extends AsyncEventHandler {
  private AbsoluteTime lastPage;
  private Clock clock;
  private final long nursePagesGap = 600000; // 10 mins
  private final long doctorPagesGap = 300000;// 5 mins
  private AsyncEvent nursePager,doctorPager;
  private PatientVitalSignsMonitor patient;

  PanicButtonHandler(AsyncEvent button, AsyncEvent n,
                     AsyncEvent d, PatientVitalSignsMonitor s) {
    lastPage = new AbsoluteTime(0,0);
    clock = Clock.getRealtimeClock();
    nursePager = n;doctorPager = d; patient = s;
    button.addHandler(this);
  }
```


PanicButtonHandler II

```
void handleAsyncEvent() {

   RelativeTime lastCall =
          clock.getTime().subtract(lastPage);
   if(lastCall.getMilliseconds() > doctorPagesGap) {
     if(!patient.vitalSignsGood()) {
       lastPage = clock.getTime();
       doctorPager.fire();
     } else {
       if(lastCall.getMilliseconds() > nursePagesGap) {
         lastPage = clock.getTime();
         nursePager.fire(); }
       }
     }
   }
   getAndClearPendingFireCount();
}
```


Configuration

```
AsyncEvent nursePager = new AsynEvent();
AsyncEvent doctorPager = new AsynEvent();
PatientVitalSignsMonitor signs = new ... ;
PriorityParameters priority = new ...;
AsyncEvent panicButton;
ImmortalMemory im = ImmortalMemory.instance();
im.enter( new Runnable() { public void run(){
     panicButton  = new AsyncEvent();
     AsyncEventHandler handler = new PanicButtonHandler(
       panicButton, nursePager, doctorPager, signs);
     handler.setSchedulingParameters(
           new PriorityParameters(priority));
     handler.setReleaseParameters(
           panicButton.createReleaseParameters());
     if(!handler.addToFeasibility()) { outputwarning  }
     panicButton.bindTo("PanicButton");
       // start monitoring
} } );
```


Configuration II

- Note, as the asynchronous event is being bound to an external happening, the object is created in immortal memory
- The panicButton reference can be in any scope convenient for the program
- However, as the event will need to reference the handler, the handler too must be in immortal memory
- Given that the handler is a Schedulable object, its scheduling and release parameters must be defined
- The release parameters will be aperiodic in this case

Event Handlers and Termination

- Many real-time systems do not terminate. However, it is necessary to define under what conditions a program terminates
- In Java, threads are classified as being daemon or user threads; the program terminates when all user threads have terminated; the daemon threads are destroyed
- The server threads used to execute asynchronous event handlers are daemon threads
- This means that when all user real-time threads are terminated, the program will still terminate
- However, where events are bound to happenings in the environment or timers, the program may not execute as the programmer intended

Summary

- Event-based systems are supported by the `AsyncEvent` and `AsyncEventHandler` class hierarchies
- Event handlers are schedulable entities and consequently can be given release and scheduling parameters
- Periodic, one-shot timers along with interrupt handlers are supported
- Care must be taken as an implementation may use daemon server threads; these may, therefore, be terminated when the programmer does not expect it

Roadmap

- Overview of the RTSJ
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- **Real-Time Threads**
- Asynchronous Transfer of Control
- Resource Control

Real-Time Threads

Lecture aims:

- To introduce the basic RTSJ model for real-time threads
- To explain the concept of a NoHeapRealtimeThread
- To evaluate the support for periodic, sporadic and aperiodic threads
- Example: monitoring deadline miss

Introduction

- For real-time systems, it is necessary to model the activities in the controlled system with concurrent entities in the program
- Standard Java supports the notion of a thread, however, in practice Java threads are both too general and yet not expressive enough to capture the properties of real-time activities
- Real-time activities are also usually characterized by their execution patterns: being periodic, sporadic or aperiodic; representing these in standard Java can only be done by coding conventions in the program
- Such conventions are error prone and obscure the true nature of the application

The Basic Model

- Two classes: `RealtimeThread` and `NoHeapRealtimeThread`
- Real-time threads are schedulable objects and, therefore, can have associated release, scheduling, memory and processing group parameters
- A real-time thread can also have its memory area set

By default, a real-time thread inherits the parameters of its parent. If the parent has no scheduling parameters (because it was a plain Java thread), the scheduler's default value is used. For the priority scheduler, this is the normal priority.

The RealtimeThread Class I

```
class RealtimeThread extends Thread implements Schedulable {
  public RealtimeThread(SchedulingParameters s);
  public RealtimeThread(SchedulingParameters s,
                        ReleaseParameters r);
  public RealtimeThread(SchedulingParameters s,
      ReleaseParameters r, MemoryParameters m,
      MemoryArea a, ProcessingGroupParameters g,
      Runnable logic);
  ....
  public static MemoryArea getCurrentMemoryArea();
  public MemoryArea getMemoryArea();
  public static MemoryArea getOuterMemoryArea(int index);
  public static int getInitialMemoryAreaIndex();
  public static int getMemoryAreaStackDepth();
  public static void sleep(Clock clock, HighResolutionTime t)
         throws InterruptedException;
  public static void sleep(HighResolutionTime time) throws …
  public void start();
  public static RealtimeThread currentRealtimeThread();
```


The RealtimeThread Class II

```
public class RealtimeThread extends ...
  ...
  public boolean waitForNextPeriod()
         throws IllegalThreadStateException;
  public void deschedulePeriodic();
  public void schedulePeriodic();
```

The meaning of these methods depends on the thread's scheduler. The following definitions are for the base priority scheduler.

Support for Periodic Threads

- The **`waitForNextPeriod`** method suspends the thread until its next release time (unless the thread has missed its deadline)
- The call returns true when the thread is next released; if the thread is not a periodic thread, an exception is thrown
- The **`deschedulePeriodic`** method will cause the associated thread to block at the end of its current release (when it calls `wFNP`); it will then remain descheduled until **`schedulerPeriodic`** is called
- When a periodic thread is "rescheduled" in this manner, the scheduler is informed so that it can remove or add the thread to the list of schedulable objects it is managing

Support for Periodic Threads

- The **`ReleaseParameters`** associated with a real-time thread can specify asynchronous event handlers which are scheduled by the system if the thread misses its deadline or overruns its cost allocation
- For deadline miss, no action is immediately performed on the thread itself.
- It is up to the handlers to undertake recovery operations
- If no handlers have been defined, a count is kept of the number of missed deadlines

Support for Periodic Threads

- The **`waitForNextPeriod`** (wFNP) method is for use by real-time threads that have periodic release parameters
- Its behavior can be described in terms of the following attributes:
  - **`lastReturn`** — indicates the last return value from wFNP
  - **`missCount`** — indicates the how many deadlines have been missed (for which no event handler has been released)
  - **`descheduled`** — indicates the thread should be descheduled at the end of its current release
  - **`pendingReleases`**— indicates number of releases that are pending

Support for Periodic Threads

- Schedulable objects (SO) have 3 states:
  - Blocked means the SO cannot be selected to have its state changed to executing; the reason may be blocked-for-I/O-completion,
    - blocked-for-release-event
    - blocked-for-reschedule
    - blocked-for-cost-replenishment
  - Eligible-for-execution means the SO can have its state changed to executing
  - Executing means the program counter in a processor holds an instruction address within the SO

Support for Periodic Threads

- On each deadline miss:
  - if the thread has a deadline miss handler, `descheduled` := true and the deadline miss handler is released with a `fireCount` increased by `missCount+1`
  - Otherwise, the `missCount` is incremented

Support for Periodic Threads

- On each cost overrun:
  - if the thread has an overrun handler, it is released
  - if the next release event has not already occurred (`pendingReleases` == 0)
    - and the thread is Eligible-for-execution or Executing , the thread becomes blocked (blocked_for_cost_replenishment)
    - otherwise, it is already blocked, so the state transition is deferred
  - otherwise (a release has occurred), the cost is replenished

Support for Periodic Threads

- When each period is due:
  - if the thread is waiting for its next release (`blocked_for_release_event`)
    - if **`descheduled`** == true, nothing happens
    - otherwise, the thread is made eligible for execution, the cost budget is replenished and `pendingReleases` is incremented
  - Otherwise: `pendingReleases` is incremented, and
    - if the thread is blocked_for_cost_replenishment, it is made Eligible-for-execution and rescheduled

Support for Periodic Threads

- When the thread's `schedulePeriodic` method is invoked:
  - `descheduled` is set to false
  - if the thread is blocked-for-reschedule, `pendingReleases` is set to zero and it is made blocked-for-release-event
- When the thread's `deschedulePeriodic` method is invoked:
  - `descheduled` is set to true

Support for Periodic Threads

- When the thread's cost parameter changes:
  - if the thread's cost budget is depleted and the thread is currently eligible for execution, a cost overrun for the thread is triggered
  - if the cost budget is not depleted and the thread is currently blocked-for-cost-replenishment, the thread is made eligible for execution

Support for Periodic Threads

- The `waitForNextPeriod` method has three possible behaviors depending on the state of `missCount`, `descheduled` and `pendingReleases`
- If `missCount >0:`
  - `missCount--`
  - if `lastReturn` is false, `pendingReleases` is decremented and false is returned (this indicates that the next release has already missed its deadline), a new cost budget is allocated
  - `lastReturn` is set to false and false is returned (this indicates that the current release has missed its deadline)

Support for Periodic Threads

- else, if **`descheduled`** is true:
  - the thread is made blocked-for-reschedule until it is notified by a call to `schedulePeriodic`
  - then it becomes blocked-for-release-event, when the release occurs the thread becomes eligible for execution
  - `pendingReleases--`
  - `lastReturn` is set to true and true is returned
- Otherwise,
  - if **`pendingReleases>=0`**:
    - the thread becomes blocked-for-release-event, when the release occurs the thread becomes eligible for execution
  - `lastReturn` = true, `pendingReleases--`, true is returned

Periodic Threads Summary

- If there are no handlers, `wFNP` will not block the thread in the event of a deadline miss. It is assumed that in this situation the thread itself will take some corrective action
- Where the handler is available, it is assumed that the handlers will take some corrective action and then reschedule the thread by calling **`schedulePeriodic`**

NoHeapRealtimeThread

- One of the main weaknesses with standard Java, from a real-time perspective, is that threads can be arbitrarily delayed by the action of the garbage collector
- The RTSJ has attacked this problem by allowing objects to be created in memory areas other than the heap
- These areas are not subject to GC
- A no-heap real-time thread NHRT is a real-time thread which only ever accesses non-heap memory areas
- Hence, it can safely be executed even when GC is occurring

NoHeapRealtimeThread I

- The `NHRT` constructors contain references to a memory area; all memory allocation performed by the thread will be from within this memory area
- An unchecked exception is thrown if the `HeapMemory` area is passed
- The `start` method is redefined; its goal is to check that the `NHRT` has not been allocated on the heap and that it has obtained no heap-allocated parameters
- If either of these requirements have been violated, an unchecked exception is thrown

NoHeapRealtimeThread Class

```
public class NoHeapRealtimeThread extends RealtimeThread {
  public NoHeapRealtimeThread(
      SchedulingParameters scheduling, MemoryArea area);
  public NoHeapRealtimeThread(
      SchedulingParameters scheduling,
      ReleaseParameters release, MemoryArea area);
  public NoHeapRealtimeThread(
      SchedulingParameters scheduling,
      ReleaseParameters release, MemoryParameters memory,
      MemoryArea area, ProcessingGroupParameters group,
      java.lang.Runnable logic);
```


A Simple Model of Periodic Threads

```
class Periodic extends RealtimeThread {

  Periodic(PriorityParameters pp, PeriodicParameters P)
  { super(pp, p); };

  public void run() {
   boolean noProblems = true;
   while(noProblems) { // code to be run each period
     ...
     noProblems = waitForNextPeriod();
   }
   // a deadline has been missed, and there is no handler
   ...
  }
}
```

The Model of Sporadic/Aperiodic Threads

- Unlike periodic threads, their release parameters have no start time, so they can be considered to be released as soon as they are started
- However, how do they indicate to the scheduler that they have completed their execution and how are they re-released?
- There is no equivalent of wFNP (contrast this with sporadic and asynchronous event handlers which have a **`fire`** method and a **`handleAsyncEvent`** method executed for each call of **`fire`**)
- The answer to this question appears to be that you have to provide your own!

Monitoring Deadline Miss I

- In many soft real-time systems, applications will want to monitor any deadline misses and cost overruns but take no action unless a certain threshold is reached
- Consider: deadline miss monitor

```
import javax.realtime.*;
public class HealthMonitor{
  public void persistentDeadlineMiss(Schedulable s);
}
```


Monitoring Deadline Miss II

```
class DeadlineMissHandler extends AsyncEventHandler {
 private RealtimeThread myrt;
 private int missDeadlineCount, myThreshold 0;
 private HealthMonitor myHealthMonitor;

 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);
 }
```


Monitoring Deadline Miss

The deadline miss handler is as schedulable object. Consequently, it will compete with its associated schedulable object according to their priority. Hence, for the handler to have an impact on the errant real-time thread, it must have a priority higher than the thread. If the priority is lower, it will not run until the errant thread blocks.

Example

```
import javax.realtime.*;

class Main {
  static RealtimeThread rt;

  static class MyRTThread extends NoHeapRealtimeThread {
     MyRTThread() {
        super(new PriorityParameters(14), new PeriodicParameters(
              new RelativeTime(50, 0)), ImmortalMemory.instance());
        }

        Clock clock = Clock.getRealtimeClock();
        int maxiter = 500;
        RelativeTime[] times = new RelativeTime[maxiter];
        ScopedMemory area = new LTMemory(4000000,4000000);
```


Example

```
        public void run() {
          int i = 0, missed = 0;
          AbsoluteTime last = clock.getTime();
          while (true) {
              area.enter(new Runnable() {
                  public void run() {
                      for (int k = 0; k < 1000; k++)
                          int[] kk = new int[k];
              }});
              if (!waitForNextPeriod()) missed++;
              AbsoluteTime now = clock.getTime();
              RelativeTime interval = now.subtract(last);
              last = now;
              times[i] = interval;
              if (++i == maxiter) break;
          }
          System.err.println("missed " + missed);
          for (int j = 0; j < maxiter; j++)
            if (times[j].getMilliseconds() > 100)
              System.err.println(j+" "+times[j].getMilliseconds());
      }
  }
```


Example

```
      public static void main(String[] args) {
        new RealtimeThread() {
            public void run() {
                ImmortalMemory.instance().enter(new Runnable() {
                    public void run() {
                        rt = new MyRTThread();
                        System.out.println("starting");
                        rt.start();
                    }
                });
            }
        }.start();
    }
}
```


Summary

- The RTSJ supports the notion of schedulable objects with various types of release characteristics
- We have reviewed two type of schedulable objects: real-time threads and no-heap real-time threads
- We have illustrated that periodic activities are well catered for
- However, the support for aperiodic and sporadic activities is lacking
- It is currently not possible for the RTSJ to detect either deadline miss or cost overruns for these activities, as there is no notion of a release event
- Indeed, programmers are best advised to use event handlers to represent non-periodic activities

Roadmap

- Overview of the RTSJ
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- **Asynchronous Transfer of Control**
- Resource Control

Asynchronous Transfer of Control

- Lecture aims:
- To introduce the application requirements for ATC
- To explain the basic RTSJ Asynchronous Transfer of Control (ATC) model

Introduction

- An asynchronous transfer of control (ATC) is where the point of execution of one schedulable object is changed by the action of another schedulable object
- Consequently, a SO may be executing one method and then suddenly, through no action of its own, find itself executing another
- Controversial because
  - complicates the language's semantics
  - makes it difficult to write correct code as the code may be subject to interference
  - increases the complexity of JVM
  - may slow down the execution of code which doesn't use the feature

The Application Requirements for ATC

- Fundamental requirement: to enable a process to respond quickly to a condition detected by another process
- Error recovery — to support coordinated error recovery between real-time threads
  - Where several threads are collectively solving a problem, an error detected by one thread may need to be quickly and safely communicated to the other threads
  - These types of activities are often called atomic actions
  - An error detected in one thread requires all other threads to participate in the recovery
  - *E.g, a hardware fault detected by a thread may mean that other threads will never finish executing because the preconditions under which they started no longer hold; they may never reach reach their polling point*

Mode Changes

- Mode changes — where changes between modes are expected but cannot be planned
  - a fault may lead to an aircraft abandoning its take-off and entering into an emergency mode of operation
  - an accident in a manufacturing process may require an immediate mode change to ensure an orderly shutdown of the plant
- The processes must be quickly and safely informed that the mode in which they are operating has changed, and that they now need to undertake a different set of actions

Scheduling and Interrupts

- Scheduling using partial/imprecise computations — there are many algorithms where the accuracy of the results depends on how much time can be allocated to their calculation
  - numerical computations, statistical estimations and heuristic searches may all produce an initial estimation of the required result, and then refine that result to a greater accuracy
  - at run-time, a certain amount of time can be allocated to an algorithm, and then, when that time has been used, the process must be interrupted to stop further refinement of the result
- User interrupts — In an interactive environment, users often wish to stop the current processing because they have detected an error condition and wish to start again

Polling ands Aborts

Polling for the notification is too slow. One approach to ATC is to destroy the thread and allow another thread to perform some recovery. All operating systems and most concurrent programming languages provide such a facility. However, destroying a thread can be expensive and is often an extreme response to many error conditions. Furthermore, it may leave the system in an inconsistent state (for example, monitor locks may not be released). Consequently, some form of controlled ATC mechanism is required.

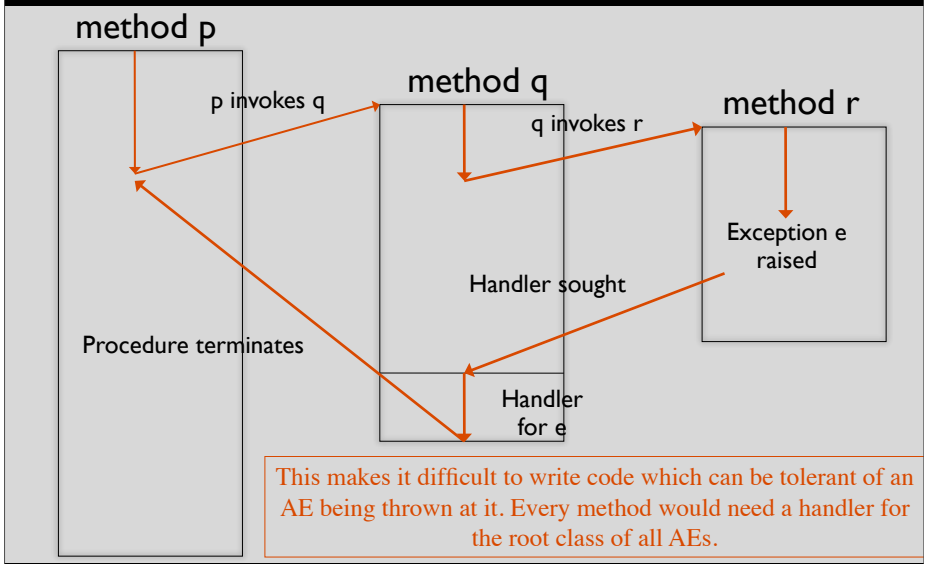
The Basic Model

- Brings together the Java exception handling model and an extension of thread interruption
- When a real-time thread is interrupted, an asynchronous exception is thrown at the thread rather than the thread having to poll for the interruption as with standard Java
- The notion of an asynchronous exception is not new and has been explored in previous languages

Problems with Asynchronous Exceptions

- The problem is how to program safely in their presence
- Most exception handling mechanisms have exception propagation with a termination model
- Consider a method `p`, which has called `q` which called `r`
- When an exception is raised within `r`, if there is no local handler, the call to `r` is terminated and a handler is sought in `q` (the exception propagates up the call chain)
- If no handler is found in `q`, the exception is propagated to `p`.
- When a handler is found, it is executed and the program continues to execute in the handler's context. There is no return to the context where the original exception was thrown.

The Termination Model



This makes it difficult to write code which can be tolerant of an AE being thrown at it. Every method would need a handler for the root class of all AEs.

ATC

- The RTSJ solution is to require that all methods, which are writen to handle an asynchronous exception, place the exception in their **`throws`** list
- These are called AI-methods (Asynchronously Interruptible)
- If a method does not do this then the asynchronous exception is not delivered but held pending until the thread is in a method which has the appropriate `throws` clause
- Hence, code written without being concerned with ATC can execute safely in an environment where ATCs are used

ATC

- To ensure that ATC can be handled safely, the RTSJ requires that
  - ATCs are deferred during the execution of synchronized methods or statements (to ensure that any shared data is left in a consistent state); these sections of code and methods which are not AI methods are called ATC-deferred sections
  - an ATC only be handled from within code that is an ATC-deferred section; this is to avoid an ATC handler being interrupted by another ATC being thrown

ATC

- Use of ATC requires
  - declaring an `AsynchronouslyInterruptedException` (AIE)
  - identifying methods which can be interrupted using a throws clause
  - signaling an AIE to a schedulable object
- Calling `interrupt` "throws" the system's generic AIE

AIE

```
public class AsynchronouslyInterruptedException extends
             InterruptedException {
  ...
  public boolean enable();
  public void disable();
  public boolean doInterruptible (Interruptible logic);

  public boolean fire();
  public boolean clear();

  public static AsynchronouslyInterruptedException getGeneric();
  // returns the AsynchronouslyInterruptedException which
  // is generated when RealtimeThread.interrupt() is invoked
```


Example of ATC

```
import nonInterruptibleServices.*;

public class InterruptibleService {
  public boolean Service() throws AIE {
    //code interspersed with calls to nonInterruptibleServices
  }
  ...
}

public InterruptibleService IS = new InterruptibleService();

// code of real-time thread, t
if(IS.Service()) { ... } else { ... };

// now another real-time thread interrupts t:

t.interrupt();
```


Semantics: when AIE is fired

- If `t` within an ATC-deferred section the AIE is marked as pending
- If `t` is in a method which does not declare AIE in its throws list, the AIE is marked as pending
- A pending AIE is thrown as soon as `t` returns to (or enters) a method with an AIE declared in its throws list
- If `t` is blocked inside a **`sleep`** or **`join`** method called from within an AI-method, `t` is rescheduled and the AIE is thrown.
- If `t` is blocked inside a **`wait`** method or the **`sleep`** or **`join`** methods called from within an ATC-deferred region, `t` is rescheduled and the AIE is thrown as a synchronous exception and it is also marked as pending

Semantics

- Although AIEs appear integrated into the Java exception handling mechanism, the normal Java rules to not apply
- Only the naming of the AIE class in a throw clause indicates the thread is interruptible. It is not possible to use a subclass. Consequently, catch clauses for AIEs must name the class AIE explicitly and not a subclass
- Handlers for AIEs do not automatically stop the propagation of the AIE. It is necessary to call the clear method in the AIE class
- Although catch clauses in ATC-deferred regions that name the `InterruptedException` or `Exception` classes will handle an AIE this will not stop the propagation of the AIE

Semantics

- Although AIE is a subclass of `InterruptedException` which is a subclass of `Exception`, catch clauses which name these classes in AI-methods will not catch an AIE
- Finally clauses that are declared in AI-methods are not executed when an ATC is thrown
- Where a synchronous exception is propagating into an AI-method, and there is a pending AIE, the synchronous exception is lost when the AIE is thrown

Catching an AIE

- Once an ATC has been thrown and control is passed to an exception handler, it is necessary to ascertain whether the caught ATC is the one expected by the interrupted thread
  - If it is, the exception can be handled.
  - If it is not, the exception should be propagated to the caller
- The `clear` method defined in the class `AsynchronouslyInterruptedException` is used for this purpose

Example Continued

```
import nonInterruptibleServices.*;

public class InterruptibleService {
  ...

  private AIE stopNow;

  public void useService() {
    stopNow = AIE.getGeneric();
    try {
      // call with calls to InterruptibleService
    } catch (AIE AI) {
      if(stopNow.clear()) { //handle the ATC }
      // No else clause, if the current exception is not
      // stopNow, it will remain pending
    }
  }
}
```

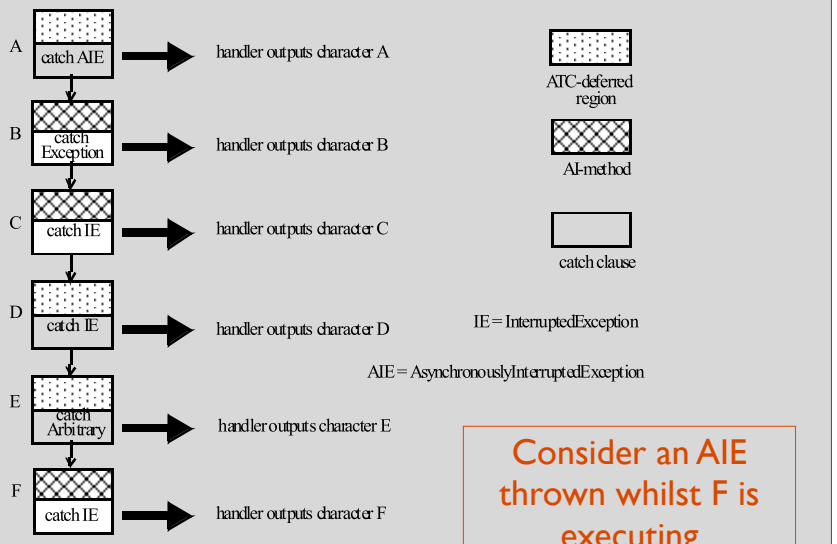
Alternative Handler

```
catch AIE AI {
  if(stopNow.clear()) {
    //handle the ATC
  } else {
    //cleanup
  }
}
```


Semantics of Clear

- If the AIE is the current AIE, the AIE is no longer pending; return true
- If the AIE isn't the current AIE, the AIE is still pending; return false

Example 1

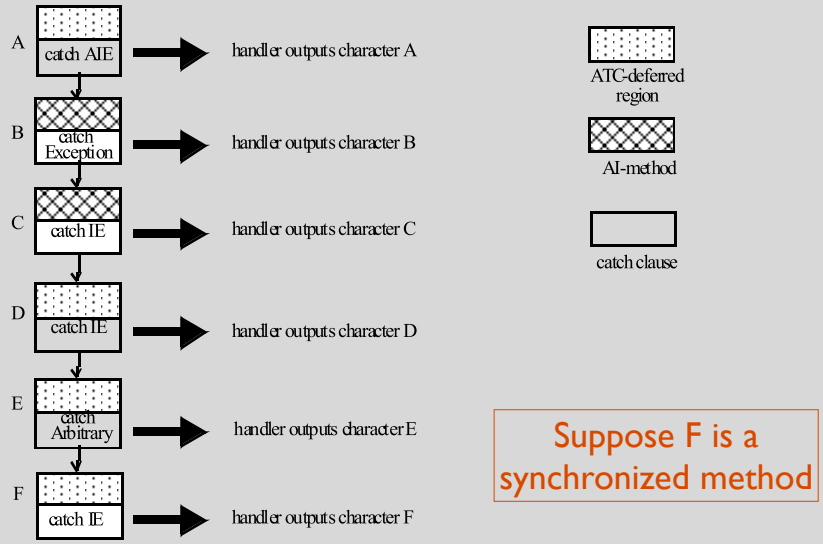


Consider an AIE thrown whilst F is executing

Example 1

- The method F is terminated
- E is ATC-deferred but it doesn't have an AIE handler; E is terminated
- D is ATC-deferred and has a handler for the IE; the handler executes, however, the AIE remains pending
- When control returns to C, the AIE is thrown, C is terminated
- B is an AI-method so the exception propagates through it
- Control returns to A which is ATC-deferred with a AIE catch clause; this executes; if A calls `clear` on the current AIE, control will return to whoever called A and the AIE will have been handled and no longer pending. If the handler for the AIE is for a different AIE, the current AIE will remain pending
- The output will be the character 'D' followed by 'A'

Example 2



Suppose F is a synchronized method

Example2

- When the ATC is thrown, the AIE is marked as pending
- The AIE remains pending until control returns to C
- The AIE is then thrown and C is terminated as is the call to B
- Finally, the handler in A is executed; the output is 'A'
- If method F had been blocked on a call to the wait method when the ATC was thrown, the AIE is marked as pending and is thrown immediately
- This would be handled by the local IE catch clause
- However, the AIE is still marked as pending and will be thrown when control returns normally to method C
- When the AIE is thrown in C, the nearest handler is in A
- Output will, be the characters 'F' and 'A'

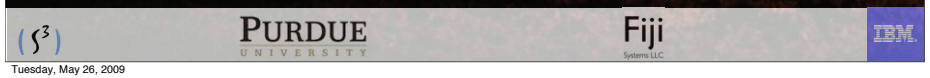
Summary I

- The ability to quickly gain the attention of a SO asynchronously is a requirement from the real-time community
- The modern approach to meeting this requirements is via an asynchronous transfer of control (ATC) facility
- RTSJ combines thread interruption with exception handling and introduces the notion of asynchronously interrupted exceptions
- A **`throws`** AIE in a method's signature indicates that the method is prepared to allow asynchronous transfers of control
- RTSJ requires that ATCs are deferred during the execution of `synchronized` methods or statements, and that an ATC only be handled from within code that is an ATC-deferred section

Summary II

- Java exception handling rules to not apply because:
  - Only AIE in a throw clause indicates the SO is interruptible
  - Handlers for AIE do not automatically stop the propagation of the AIE; it is necessary to call the clear method in the AIE class
  - Although `catch` clauses in ATC-deferred regions that name the IE or `Exception` classes handle an AIE this won't stop propagation
  - Although AIE is a subclass of IE and `Exception`, `catch` clauses which name these classes in AI-methods will not catch an AIE
  - **`finally`** clauses in AI-methods are not executed when an ATC is thrown
  - Where a synchronous exception is propagating into an AI-method, with a pending AIE, the synchronous exception is lost when the AIE is thrown

Lessons from the RTSJ



1/9/90 Rule

- The main reason of RTSJ's success is that it allows mixing freely real-time and timing-oblivious code in the same platform
- Rule of thumb: 1% hard real-time, 9% soft real-time, 90% non-RT
- RT Programming Model should:
  - Allow non-intrusive injection of real-time in a timing-oblivious system
  - Avoid paradigm switches---single semantic framework for RT/non-RT

Small Footprint

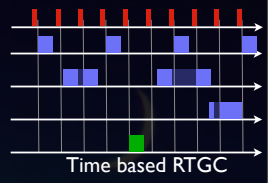
- The RTSJ requires sweeping changes in the Virtual Machine and has complex interaction with Java features.
  - The RTSJ changes the semantics of "normal" Java code:
    - any access to a reference variable may throw an exception,
    - meaning of "throws" changed to support asynchronous exception,
    - finalization interacts with memory management.
- Avoid complex APIs with ill-defined feature interactions
- Don't change the semantics of non-RT code

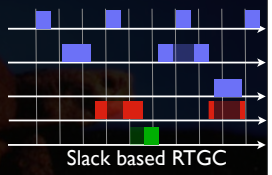
Schedule Flexibly

- Support for real-time scheduling is crucial with a clean integration in the language concurrency model
  - Priority Preemptive Scheduling for periodic tasks
  - Support PIP and PCE locks for priority inversion avoidance
- Provide facilities for writing non-blocking algorithms
- Allow for user-defined schedulers.

Don't Fear Garbage

- Real-time Garbage Collectors good enough for most RT apps
  - Time-based collectors (Metronome) can easily bound pauses to one millisecond.
  - Automatic defragmentation for long lived apps.
  - Slack-based collectors (SUN's) ensure RT code never preempted by GC.
  - Ensuring that the GC can keep up requires some understanding of whole-program allocation behavior.

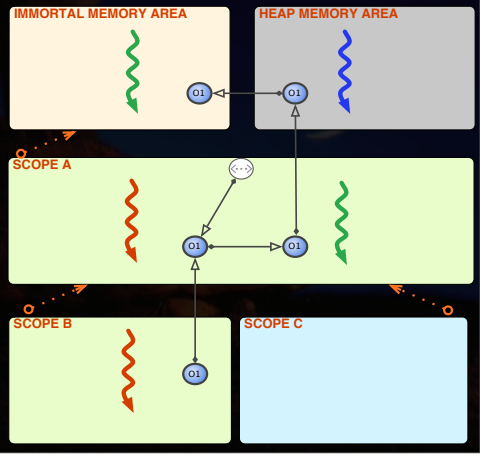




- Avoid manual memory management unless absolutely necessary.

Type-check Region-based Allocation

- Regions are the worst mistake of the RTSJ
  - Dynamic safety checks slow down programs and cause runtime errors that are hard to prevent
  - Distinction between heap-using and non-heap thread is tricky and error prone.
- Use static type system for assurance and efficiency



Shared-memory Concurrency is Hard

- Real-time presents a number of concurrency related challenges.
  - Critical sections must be short --- avoid over-synchronization
  - Avoid priority inversion between real-time and non-RT code
  - Avoid blocking operations in RT code
- Limit/localize communication between RT and non-RT.
- Support flexible isolation to encourage determinism.

Roadmap

- Overview of the RTSJ
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- **Asynchronous Transfer of Control**
- Resource Control

Asynchronous Transfer of Control

Lecture aims

- To explain the high level model and how to use the Interruptible interface
- To consider multiple **`AsynchronouslyInterruptedExceptions`** (AIE)
- To introduce the Timed class and show an example

ATC Summary

- All methods that are prepared to allow the delivery of an asynchronous exception must place the **`AIE`** exception in their **`throws`** list
- These methods are AI-methods (Asynchronously Interruptible)
- Other methods are called ATC-deferred and the asynchronous exception is not delivered but held pending until the thread is in a method which has the appropriate throws clause
- **`synchronized`** blocks are ATC deferred

AIE

```
public class AIE extends InterruptedException {
  ...
  public synchronized void disable();
  public boolean doInterruptible (Interruptible logic);

  public synchronized boolean enable();
  public synchronized boolean fire();

  public boolean clear ();

  public static AIE getGeneric();
  // returns the AIE which
  // generated when RealtimeThread.interrupt() is invoked
```


Interruptible

```
public interface Interruptible{
  public void interruptAction(AIE exception);
  public void run(AIE exception) throws AIE;
}
```

- An object which wishes to provide an interruptible method does so by implementing the **`Interruptible`** interface
- The **`run`** method is the method that is interruptible; the **`interruptedAction`** method is called by the system if the **`run`** method is interrupted

Interruptible

- Once this interface is implemented, the implementation can be passed as a parameter to the `doInterruptible` method in the AIE class
- The method can then be interrupted by calling the `fire` method in the AIE class
- Further control over the AIE is given by
  - `disable`
  - `enable`
  - `isEnabled`

Only one SO can be executing a `doInterruptible` at once

Warning

Note, the firing of an AIE has no effect if there is no currently active `doInterruptible`.

The firing is NOT persistent. Hence, care must be taken as there may be a race condition between one thread calling a doInterruptible and another thread calling fire on the same AIE.

To help cope with this race condition, fire will return false, if there is no currently active doInterruptible or the AIE is disabled.

Mode Change Example I

```
public class ModeA implements Interruptible {

  public void run(AIE aie) throws AIE  {
    // operation performed in Mode A
  }

  public void interruptAction(AIE aie) {
    // reset any internal state, so that when Mode A
    // becomes current, it can contine
  }
}

// similary for ModeB
```


Mode Change Example II

```
class ModeChanger extends AIE {
  static final int MODE_A = 0,MODE_B = 1;
  private int mode;

  ModeChanger(int initial) {
    mode = (initial==MODE_A)? MODE_A : MODE_B;
  }

  synchronized int currentMode(){ return mode; }
  synchronized void setMode(int nextMode) {
    if(nextMode == MODE_A | nextMode == MODE_B)
      mode = nextMode;
    else throw new IllegalArgumentException();
  }
  synchronized void toggleMode() {
    mode = (mode==MODE_A)?MODE_B:MODE_A;
  }
}
```

Mode Change Example IV

```
ModeChanger modeChange = newModeChanger(ModeChanger.MODE_A);

public void run() { // inside a RT thread with PeriodicParameters
  ModeA modeA = new ModeA();
  ModeB modeB = new ModeB();
  boolean ok = true;
  while(ok) {
    boolean _ =(modeChange.currentMode()==ModeChanger.MODE_A)?
         modeChange.doInterruptible(modeA):
         modeChange.doInterruptible(modeB); //throw away result
    ok = waitForNextPeriod();
  }
}
```

signaller:
```
modeChange.toggleMode();
modeChange.fire();
```


A Persistent AIE

```
public class PersistentAIE extends AIE {
  private volatile boolean outstandingAIE;

  public boolean fire() {
    if(super.fire()) return true;
    outstandingAIE = true;
    return false;
  }
 public boolean doInterruptible(Interruptible logic) {
    if(outstandingAIE) {
      outstandingAIE = false;
      logic.interruptAction(this);
      return true;
    }
    return super.doInterruptible(logic);
  }
}
```

Why does this fail?

A Persistent AIE

```
public boolean doInterruptible(Interruptible logic) {
  if(outstandingAIE) {
    outstandingAIE = false;
    logic.interruptAction(this);
    return true;
  }
  // fire comes in here!!
  return super.doInterruptible(logic);
}
```


A Persistent AIE

```
class PersistentAIE  extends AIE implements Interruptible {

  boolean fire(){ /* As before */ }
  public boolean doInterruptible(Interruptible logic) {
    this.logic = logic;
    return super.doInterruptible(this);
  }
  public void run(AIE aie) throws AIE {
    if(outstandingAIE) {
      outstandingAIE = false;
      super.fire();
    } else { logic.run(aie); }
  }
  public void interruptAction(AIE aie)
  { logic.interruptAction(aie); }

  private volatile boolean outstandingAIE;
  private Interruptible logic;
}
```

Nested ATCs

- Once the ATC-deferred region is exited, the currently pending AIE is thrown
- This AIE will be caught by the interruptAction of the inner most nested AIE first
- However, it will typically propagate the ATC

Nested ATC Example

```
class NestedATC {
  AIE AIE1 = new AIE();
  // similarly for AIE2 and AIE 3
  void method1()  { /* ATC-deferred region */ }

  void method2() throws AIE {
    AIE1.doInterruptible(new Interruptible(){
      public void run(AIE e) throws AIE {  method1(); }
      public void interruptAction( AIE e) {/* recovery*/}
    });
 }

 // similarly for method3, whose run method calls method2,
 // and for method4, whose run method calls method3
```


The Timed Class

- With Real-Time Java, there is a subclass of `AIE` called `Timed`
- Both absolute and Relative times can be used

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


The Timed Class

- When an instance of the `Timed` class is created, a timer is created and associated with the time value passed as a parameter
- A timer value in the past results in the AIE being fired immediately the `doInterruptible` is called.
- When a time value is passed which is not in the past, the timer is started when the `doInterruptible` is called
- The timer can be reset for the next call to doInterruptible by use of the `resetTime` method

Imprecise Computation

- The algorithm consists of a mandatory part which computes an adequate, but imprecise, result
- An optional part then iteratively refines the results
- The optional part can be executed as part of a `doInterruptible` attached to a `Timed` object
- The `run` method updates the result from within a `synchronized` statement so that it is not interrupted

Imprecise Computation III

```
class ImpreciseResult {
  int value; // the result
  boolean precise; //is the value precise
}

class ImpreciseComputation {
private HighResolutionTime CompletionTime;
  private ImpreciseResult result;
  ImpreciseComputation(HighResolutionTime T) { {
    CompletionTime = T; result = new ImpreciseResult();
  }
  private int compulsoryPart() { … }
```


Imprecise Computation III

```
class ImpreciseComputation {
  ImpreciseResult service() { // public service
    result.precise = false;
    result.value = compulsoryPart();
    Interruptible I = new Interruptible() {
      public void run(AIE exception) throws AIE {
        // optional part which improves on the compulsory part
         boolean canBeImproved = true;
         while(canBeImproved) synchronized(result) {…}
         result.precise = true;
      }
      public void interruptAction(AIE exp) {
        result.precise = false; }
    });
    Timed t = new Timed(CompletionTime);
    boolean res = t.doInterruptible(I));
    // execute optional part, throw away result of doInterruptible
    return result;
  }
```


Summary

- The RTSJ combines thread interruption with exception handling and introduces the notion of an asynchronously interrupted exception (AIE)
- The presence of a throws `AsynchronouslyInterruptedException` in a method's signature indicates that the method is prepared to allow asynchronous transfers of control
- The high-level approach involves creating objects which implement the Interruptible interface
- Note, the firing of an `AsynchronouslyInterrupted-Exception` has no effect if there is no currently active `doInterruptible`
- The firing is NOT persistent

Roadmap

- Overview of the RTSJ
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- **Resource Control**

Resource Sharing

Lecture aims:

- To introduce the notion of priority inversion
- To illustrate simple priority inheritance and priority ceil emulation inheritance
- To show the RTSJ Basic Model

Introduction and Priority Inversion

- In Java, communication and synchronization is based on a monitor-like construct. Synchronization mechanisms based on mutual exclusion suffer from priority inversion
- This is where a low-priority thread, **`L`**, enters into a mutual-exclusion zone shared with a high-priority thread, **`H`**. A medium-priority thread, **`M`**, then becomes runnable, pre-empts **`L`** and performs a computationally-intensive algorithm

Priority Inversion Continued

- At the start of this algorithm, a high-priority thread preempts **`M`** and tries to enter the mutual-exclusion zone
- As **`L`** currently occupies the zone, **`H`** is blocked
- **`M`** then runs, thereby indirectly blocking the progression of **`H`** for a potentially unbounded period of time
- It is this blocking that makes it very difficult to analyze the timing properties of SOs

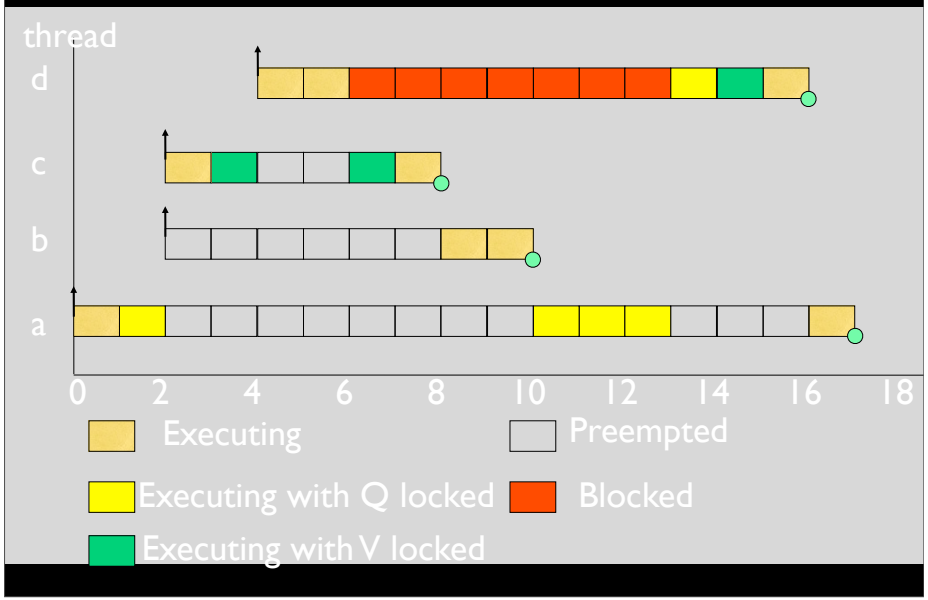
Priority Inversion

- To illustrate an extreme example of priority inversion, consider the executions of four periodic threads: a, b, c and d; and two resources (synchronized objects) : Q and V

thread	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

- Where E is executing for one time unit, Q is accessing resource Q for one time unit, V is is accessing resource V for one time unit

Example of Priority Inversion



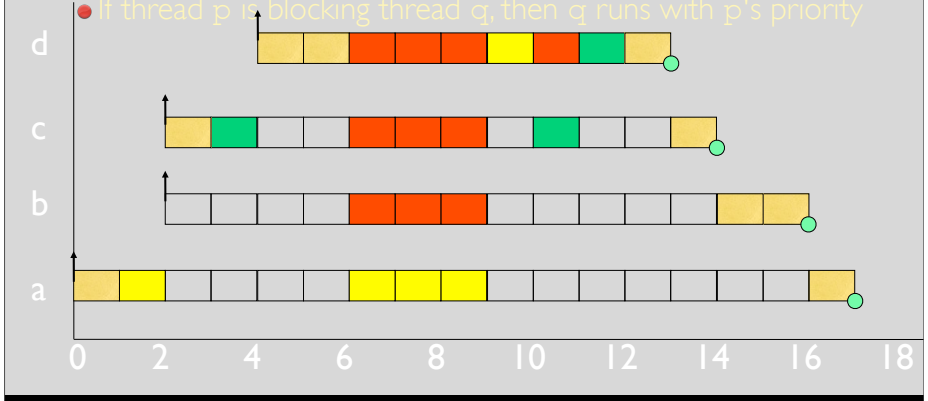
Solutions to Priority Inversion

- Priority Inheritance
- Non-blocking communication

Priority Inheritance

Process

- If thread p is blocking thread q, then q runs with p's priority



The RTSJ Basic Model

- Priority inversion can occur whenever a schedulable object is blocked waiting for a resource
- In order to limit the length of time of that blocking, the RTSJ requires the following:
  - All queues maintained by the system to be priority ordered (e.g. the queue of schedulable objects waiting for an object lock
    - Where there is more than one schedulable object in the queue at the same priority, the order should be first-in-first-out (FIFO)
    - Similarly, the queues resulting from calling the wait method in the Object class should be priority ordered
  - Facilities for the programmer to specify different priority inversion control algorithms
    - By default, the RTSJ requires simple priority inheritance to occur

Monitor Control

```
public abstract class MonitorControl {
  protected MonitorControl();
  public static void setMonitorControl(MonitorControl
policy);
  public static void setMonitorControl(Object monitor,
                                       MonitorControl policy);
}
```


Priority Inheritance

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

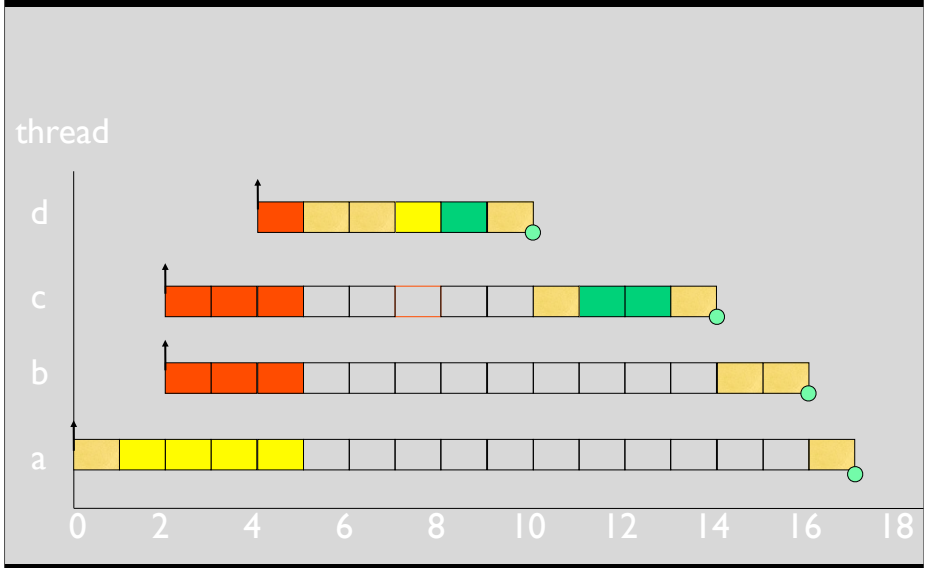

Blocking and Priority Inheritance

- If a thread has m critical sections that can lead to it being blocked then the maximum number of times it can be blocked is m
- Priority ceiling emulation attempts to reduce the blocking

Priority Ceiling Emulation

- Each thread has a static (base) default priority assigned (perhaps by the deadline monotonic scheme).
- Each resource has a static ceiling value defined, this is the maximum priority of the threads that use it.
- A thread has a dynamic (active) priority that is the maximum of its own static priority and the ceiling values of any resources it has locked
- As a consequence, a thread will only suffer a block at the very beginning of its execution
- Once the thread starts actually executing, all the resources it needs must be free; if they were not, then some thread would have an equal or higher priority and the thread's execution

Priority Ceiling Emulation Inheritance



Priority Ceiling Emulation

```
public class PriorityCeilingEmulation extends
MonitorControl {
  public static PriorityCeilingEmulation instance(int
ceiling);
  public int getCeiling();
  ...
}
```


NOTE

- Generally, the code used inside a synchronized method (or statement) should be kept as short as possible, as this will dictate the length of time a low priority schedulable object can block a high one
- It is only possible to limit the block if the code doesn't contain:
  - unbounded loops,
  - arbitrary-length blocking operations that hold the lock, for example an arbitrarily-length sleep request

Ceiling Violations

- Whenever a schedulable object calls a synchronized method (statement) in an object which has the Priority-CeilingEmulation policy in force, the real-time virtual machine will check the active priority of the caller
- If the priority is greater than the ceiling priority, the uncheck CeilingViolationException is thrown

Communication between heap-using and no-heap

- One of the main driving forces behind the RTSJ was to make real-time Java applications more predictable
- Hence, schedulable objects which need complete predictability can be defined not to reference the heap
- This means that they can pre-empt any garbage collection that might be occurring when the schedulable objects are released.

Communication between heap-using and

- Most large real-time systems will consists of a mixture of heap-using and no-heap schedulable objects
- There will be occasions when they need to exchange information
- To ensure that no-heap schedulable objects are not indirectly delayed by garbage collection requires
  - all no-heap schedulable object should have priorities greater than heap-using schedulable objects,
  - priority ceiling emulation should be used for all shared synchronized objects,
  - all shared synchronized object should have their memory requirements pre-allocated (or dynamically allocated from scoped or immortal memory areas), and

Summary

- Priority inversion is where a high priority thread is blocked by the execution of a lower priority thread
- Priority inheritance results in the lower priority thread inheriting the priority of the high priority thread whilst it is blocking the high priority thread
- Whilst priority inheritance allows blocking to be bounded, a block still occurs for every critical section
- Priority ceiling emulation allows only a single block
- The alternative to using priority inheritance algorithms is to use wait free communication (see book)

Roadmap

- Introduction
- Concurrent Programming
- Communication and Synchronization
- Completing the Java Model
- Overview of the RTSJ
- Memory Management
- Clocks and Time

- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control
- Schedulability Analysis
-

Scheduling

- Lecture aims
  - To understand the role that scheduling and schedulability analysis plays in predicting that real-time applications meet their deadlines
  - Education onlu - not examinable
- Topics
  - Simple process model
  - The cyclic executive approach
  - Process-based scheduling
  - Utilization-based schedulability tests
  - Response time analysis for FPS and EDF
  - Worst-case execution time

Scheduling

- In general, a scheduling scheme provides two features:
  - An algorithm for ordering the use of system resources (in particular the CPUs)
  - A means of predicting the worst-case behaviour of the system when the scheduling algorithm is applied
- The prediction can then be used to confirm the temporal requirements of the application

Simple Process Model

- The application is assumed to consist of a fixed set of processes
- All processes are periodic, with known periods
- The processes are completely independent of each other
- All system's overheads, context-switching times and so on are ignored (i.e, assumed to have zero cost)
- All processes have a deadline equal to their period (that is, each process must complete before it is next released)
- All processes have a fixed worst-case execution time

Standard Notation

B	Worst-case blocking time for the process (if applicable)
C	Worst-case computation time (WCET) of the process
D	Deadline of the process
I	The interference time of the process
N	Number of processes in the system
P	Priority assigned to the process (if applicable)
R	Worst-case response time of the process
T	Minimum time between process releases (process period)
U	The utilization of each process (equal to C/T)
a - z	The name of a process

Cyclic Executives

- One common way of implementing hard real-time systems is to use a cyclic executive
- Here the design is concurrent but the code is produced as a collection of procedures
- Procedures are mapped onto a set of minor cycles that constitute the complete schedule (or major cycle)
- Minor cycle dictates the minimum cycle time
- Has the advantage of being fully deterministic

Properties

- No actual processes exist at run-time; each minor cycle is just a sequence of procedure calls
- The procedures share a common address space and can thus pass data between themselves. This data does not need to be protected (via a semaphore, for example) because concurrent access is not possible
- All "**process**" periods must be a multiple of the minor cycle time

Problems with Cycle Executives

- The difficulty of incorporating processes with long periods; the major cycle time is the maximum period that can be accommodated without secondary schedules
- Sporadic activities are difficult (impossible!) to incorporate
- The cyclic executive is difficult to construct and difficult to maintain — it is a NP-hard problem
- Any "process" with a sizable computation time will need to be split into a fixed number of fixed sized procedures (this may cut across the structure of the code from a software engineering perspective, and hence may be error-prone)
- More flexible scheduling methods are difficult to support

Process-Based Scheduling

- Scheduling approaches
  - Fixed-Priority Scheduling (FPS)
  - Earliest Deadline First (EDF)
  - Value-Based Scheduling (VBS)

Fixed-Priority Scheduling (FPS)

- This is the most widely used approach and is the main focus of this course
- Each process has a fixed, static, priority which is computer pre-run-time
- The runnable processes are executed in the order determined by their priority
- In real-time systems, the "priority" of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity

FPS and Rate Monotonic Priority Assignment

- Each process is assigned a (unique) priority based on its period; the shorter the period, the higher the priority
- I.e, for two processes `i` and `j`,
- This assignment is optimal in the sense that if any process set can be scheduled (using pre-emptive priority-based scheduling) with a fixed-priority assignment scheme, then the given process set can also be scheduled with a rate monotonic assignment scheme
- Note, priority 1 is the lowest (least) priority

Example Priority Assignment

Process	Period, T	Priority, P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

Utilisation-Based Analysis

- For D=T task sets only
- A simple sufficient but not necessary schedulability test exists

Utilization Bounds

N	Utilization bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

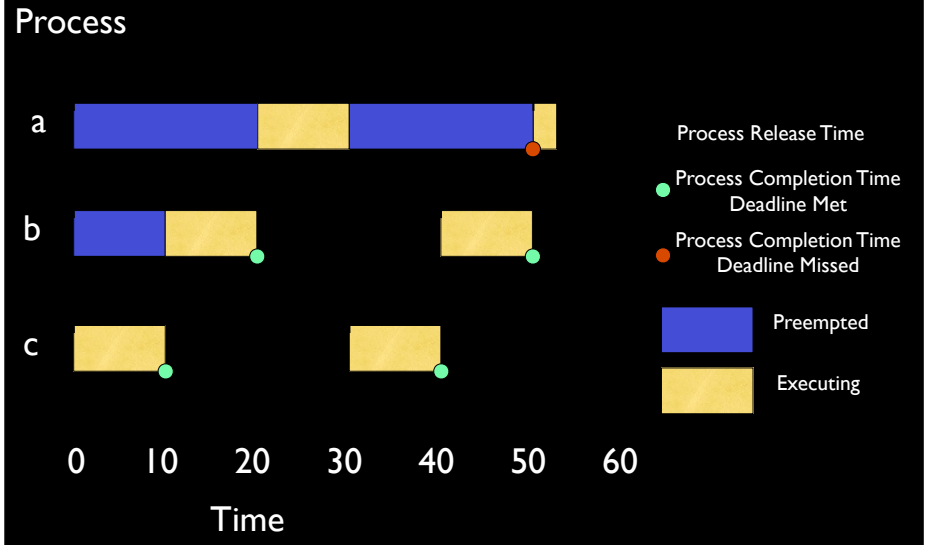
Approaches 69.3% asymptotically

Process Set A

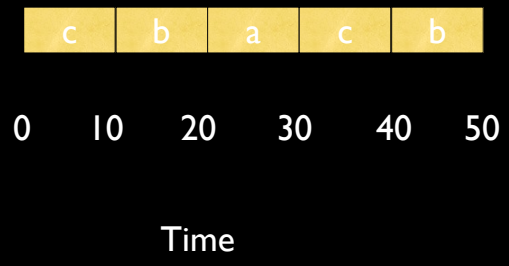
Process	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

- The combined utilization is 0.82 (or 82%)
- This is above the threshold for three processes (0.78) and, hence, this process set fails the utilization test

Time-line for Process Set A



Gantt Chart for Process Set A



Process Set B

Process	Period T	Computation Time C	Priority P	Utilization U
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

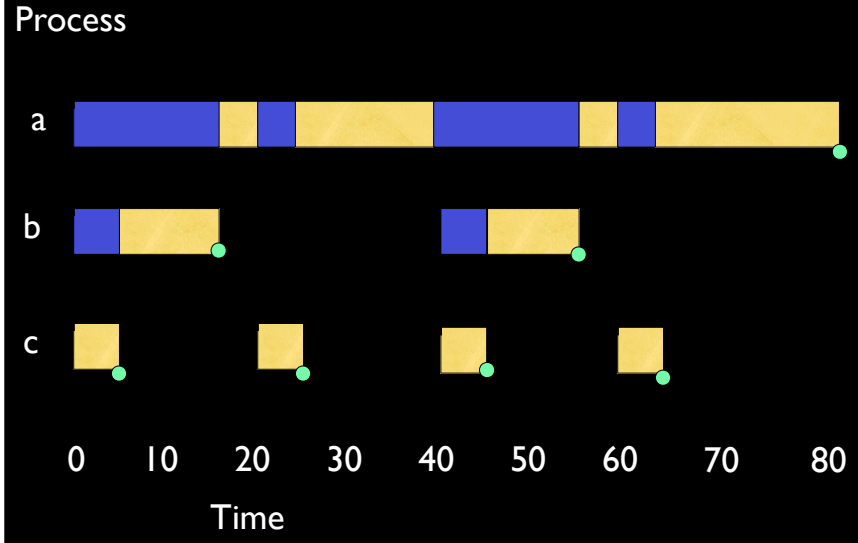
- The combined utilization is 0.775
- This is below the threshold for three processes (0.78) and, hence, this process set will meet all its deadlines

Process Set C

Process	Period T	Computation Time C	Priority P	Utilization U
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

- The combined utilization is 1.0
- This is above the threshold for three processes (0.78) but the process set will meet all its deadlines

Time-line for Process Set C



Criticism of Utilisation-based Tests

- Not exact
- Not general
- BUT it is O(N)

The test is said to be sufficient but not necessary

Response-Time Analysis

- Here task i's worst-case response time, R, is calculated first and then checked (trivially) with its deadline

$$R_i \leq D_i$$

Where I is the interference from higher priority tasks

Calculating R

During R, each higher priority task j will execute a number of times:

The ceiling function gives the smallest integer greater than the fractional number on which it acts. So the ceiling of 1/3 is 1, of 6/5 is 2, and of 6/3 is 2.

Total interference is given by:

Response Time Equation

Where $hp(i)$ is the set of tasks with priority higher than task i

Solve by forming a recurrence relationship:

The set of values is monotonically non decreasing

When the solution to the equation has been found, must not be greater that (e.g. 0 or)

Process Set D

Process	Period T	Computation Time C	Priority P
a	7	3	3
b	12	3	2
c	20	5	1

Revisit: Process Set C

Process	Period T	Computation Time C	Priority P	Response time R
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

- The combined utilization is 1.0
- This was above the ulilization threshold for three processes (0.78), therefore it failed the test
- The response time analysis shows that the process set will meet all its deadlines

Response Time Analysis

- Is sufficient and necessary
- If the process set passes the test they will meet all their deadlines; if they fail the test then, at run-time, a process will miss its deadline (unless the computation time estimations themselves turn out to be pessimistic)

Worst-Case Execution Time - WCET

- Obtained by either measurement or analysis
- The problem with measurement is that it is difficult to be sure when the worst case has been observed
- The drawback of analysis is that an effective model of the processor (including caches, pipelines, memory wait states and so on) must be available

WCET— Finding C

Most analysis techniques involve two distinct activities.

- The first takes the process and decomposes its code into a directed graph of basic blocks
- These basic blocks represent straight-line code
- The second component of the analysis takes the machine code corresponding to a basic block and uses the processor model to estimate its worst-case execution time
- Once the times for all the basic blocks are known, the directed graph can be collapsed

Need for Semantic Information

```
for I in 1.. 10 loop
  if Cond then
    -- basic block of cost 100
  else
    -- basic block of cost 10
  end if;
end loop;
```

- Simple cost 10*100 (+overhead), say 1005.

Sporadic Processes

- Sporadics processes have a minimum inter-arrival time
- They also require D<T
- The response time algorithm for fixed priority scheduling works perfectly for values of D less than T as long as the stopping criteria becomes
- It also works perfectly well with any priority ordering — hp(i) always gives the set of higher-priority processes

Hard and Soft Processes

- In many situations the worst-case figures for sporadic processes are considerably higher than the averages
- Interrupts often arrive in bursts and an abnormal sensor reading may lead to significant additional computation
- Measuring schedulability with worst-case figures may lead to very low processor utilizations being observed in the actual running system

General Guidelines

Rule 1 — all processes should be schedulable using average execution times and average arrival rates

Rule 2 — all hard real-time processes should be schedulable using worst-case execution times and worst-case arrival rates of all processes (including soft)

- A consequent of Rule 1 is that there may be situations in which it is not possible to meet all current deadlines
- This condition is known as a transient overload
- Rule 2 ensures that no hard real-time process will miss its deadline
- If Rule 2 gives rise to unacceptably low utilizations for "normal execution" then action must be taken to reduce the worst-case execution times (or arrival rates)

Aperiodic Processes

- These do not have minimum inter-arrival times
- Can run aperiodic processes at a priority below the priorities assigned to hard processes, therefore, they cannot steal, in a pre-emptive system, resources from the hard processes
- This does not provide adequate support to soft processes which will often miss their deadlines
- To improve the situation for soft processes, a server can be employed.
- Servers protect the processing resources needed by hard processes but otherwise allow soft processes to run as soon as possible.
- POSIX supports Sporadic Servers

Process Sets with D < T

- For D = T, Rate Monotonic priority ordering is optimal
- For D < T, Deadline Monotonic priority ordering is optimal

D < T Example Process Set

Process	Period T	Deadline D	ComputationTime C	Priority P	Response time R
a	20	5	3	4	3
b	15	7	3	3	6
c	10	10	4	2	10
d	20	20	3	1	20

Process Interactions and Blocking

- If a process is suspended waiting for a lower-priority process to complete some required computation then the priority model is, in some sense, being undermined
- It is said to suffer priority inversion
- If a process is waiting for a lower-priority process, it is said to be blocked

Response Time and Blocking

Dynamic Systems and Online Analysis

- There are dynamic soft real-time applications in which arrival patterns and computation times are not known **a priori**
- Although some level of off-line analysis may still be applicable, this can no longer be complete and hence some form of on-line analysis is required
- The main task of an on-line scheduling scheme is to manage any overload that is likely to occur due to the dynamics of the system's environment
- EDF is a dynamic scheduling scheme that is an optimal
- During transient overloads EDF performs very badly. It is possible to get a cascade effect in which each process misses its deadline but uses sufficient resources to result in the next process also missing its deadline

Admission Schemes

- To counter this detrimental domino effect, many on-line schemes have two mechanisms:
  - an admissions control module that limits the number of processes that are allowed to compete for the processors, and
  - an EDF dispatching routine for those processes that are admitted
- An ideal admissions algorithm prevents the processors getting overloaded so that the EDF routine works effectively

Values

- If some processes are to be admitted, whilst others rejected, the relative importance of each process must be known
- This is usually achieved by assigning value
- Values can be classified
  - Static: the process always has the same value whenever it is released.
  - Dynamic: the process's value can only be computed at the time the process is released (because it is dependent on either environmental factors or the current state of the system)
  - Adaptive: here the dynamic nature of the system is such that the value of the process will change during its execution
- To assign static values requires the domain specialists to articulate their understanding of the desirable behaviour of the system

Summary

- A scheduling scheme defines an algorithm for resource sharing and a means of predicting the worst-case behaviour of an application when that form of resource sharing is used.
- With a cyclic executive, the application code must be packed into a fixed number of minor cycles such that the cyclic execution of the sequence of minor cycles (the major cycle) will enable all system deadlines to be met
- The cyclic executive approach has major drawbacks many of which are solved by priority-based systems
- Simple utilization-based schedulability tests are not exact

Summary

- Response time analysis is flexible and caters for:
  - Periodic and sporadic processes
  - Blocking caused by IPC
  - Cooperative scheduling (not covered)
  - Arbitrary deadlines (not covered)
  - Release jitter (not covered)
  - Fault tolerance (not covered)
  - Offsets (not covered)
- RT Java supports preemptive priority-based scheduling
- RT Java addresses dynamic systems with the potential for on-line analysis

Flexible Task Graphs

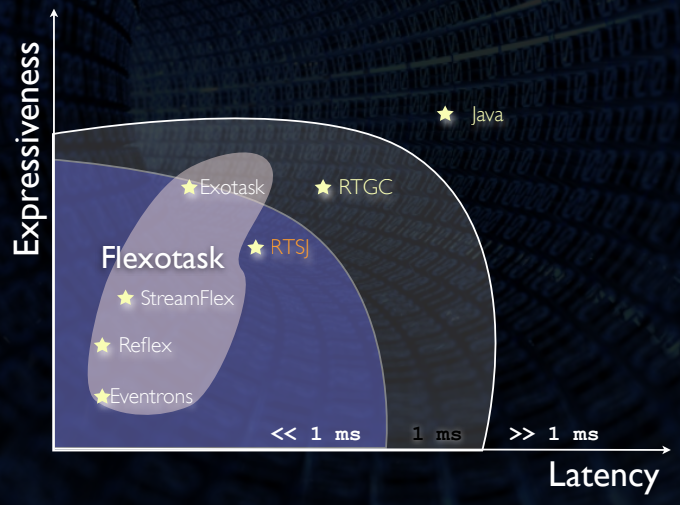
Goals

- Design a new real-time programming model that allows embedding hard real-time computations in timing-oblivious Java applications
- Principle of Least Surprise
  - Semantics of non-real-time code unchanged
  - Semantics of real-time code unsurprising
- Limited set of new abstractions that compose flexibly
- No cheating
  - Run efficiently in a production environment

Unification of previous work

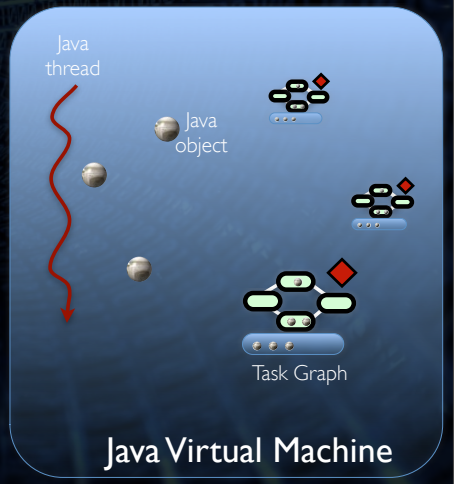
- **Eventrons** [PLDI'06] (IBM)
- **Reflexes** [VEE'07] (Purdue/EPFL)
  - Inspired by RTSJ and Eventrons
- **Exotasks** [LCTES'07] (IBM)
  - Inspired by Giotto, and E-machine
- **StreamFlex** [OOPSLA'07] (Purdue/EPFL)
  - Inspired by Reflexes, StreamIt and dataflow languages

Design space



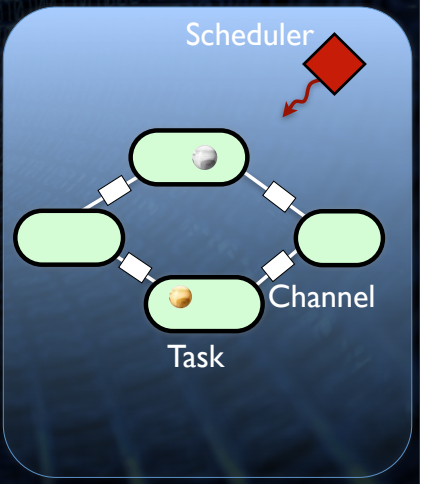
Flexible Task Graphs

- A *FlexoTask Graph* is a set of *concurrently* executing, *isolated*, tasks communicating through non-blocking *channels*
- Semantics of legacy code is unaffected
- Real-time code has restricted semantics, enforced by compile and start-up time static checks



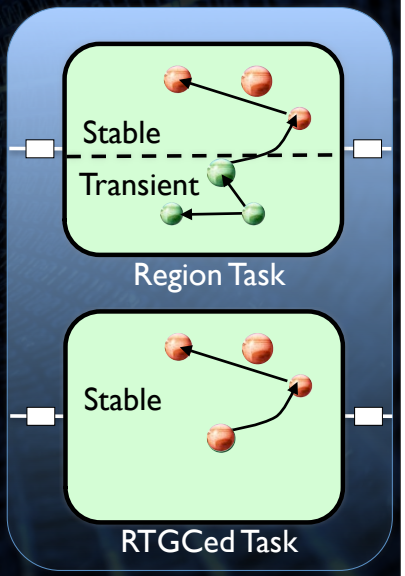
Task Graph

- A *FlexoTask Graph* is a set of *concurrently* executing, *isolated*, tasks communicating through *channels*
- Schedulers control the execution of tasks with user-defined policies (eg. logical execution time, data driven)
  - atomically update task's in ports
  - invoke task's `execute()`
  - update the task's output ports



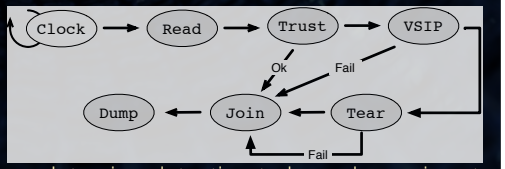
Memory management

- Either garbage collected with a real-time GC, or a region allocator for sub-millisecond response times.
- Region-allocated tasks preempt task RTGC and Java GC
- Region tasks are split between
  - Stable objects
  - Transient (per invocation) objects



Task communication: Channels

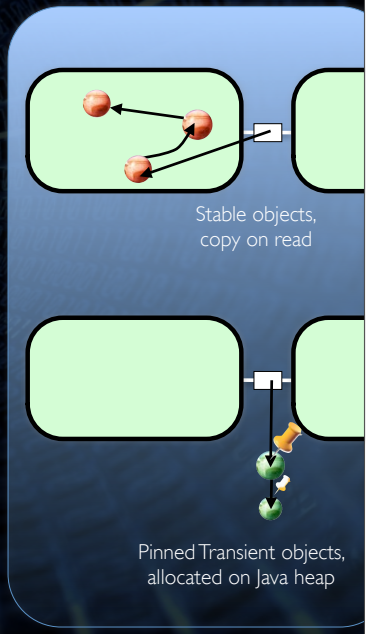
- Design goals: scalability and expressiveness
- Rule #1: isolation to allow local, per task GC (avoid dependencies on global allocation rates)
- Rule #2: allow for decomposition of problems in stages



e.g. Intrusion detection task graph running at 750Mb per sec [OOPSLA07]

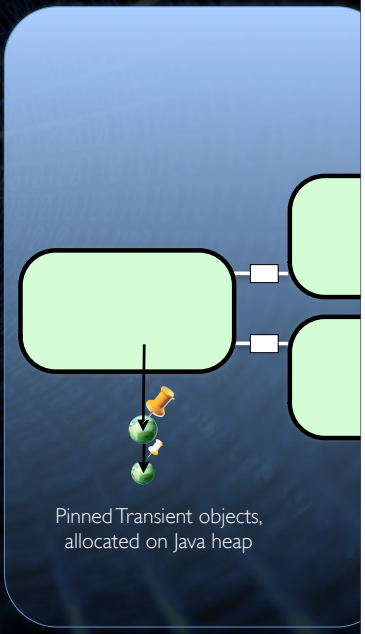
Channels

- Stable channels
  - Can refer to any stable object (complex structures)
  - Deep copy on read (atomic)
- Transient channels
  - Can refer to Capsules (transient objects, arrays)
  - Zero-copy (linear reference)



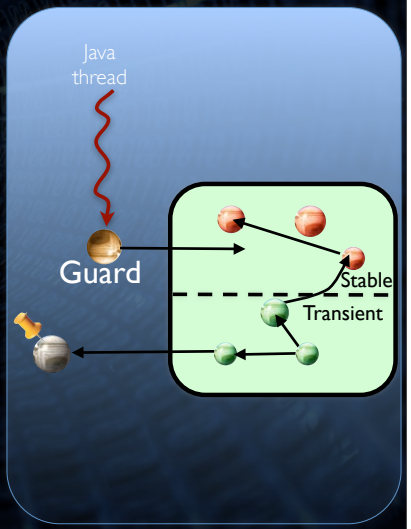
Channels

- Allow double send of a capsule?
- Immutable capsules
  - *will force users to perform copies of the data*
- Mutable capsules
  - Zero-copy
    - *mutable capsules are shared introducing data races*
  - Copy
    - *copies are done atomically when the task execute() method returns*



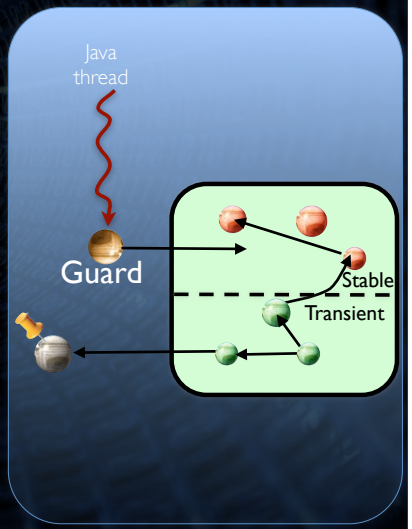
Communication with Java

- Challenges:
  - Avoiding priority inversion
  - Enforcing isolation



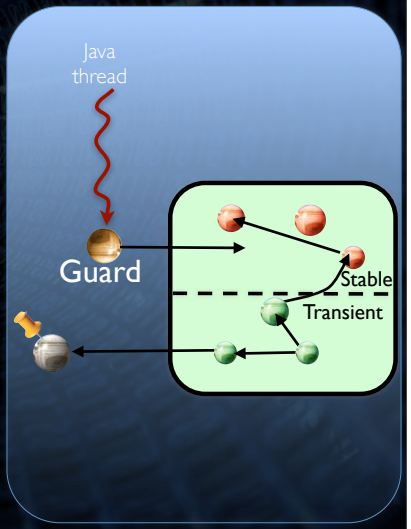
Communication with Java

- Every task has an automatically generated proxy-object
- User-defined atomic methods can be called from Java with transactional semantics
- Arguments are reference-immutable pinned objects



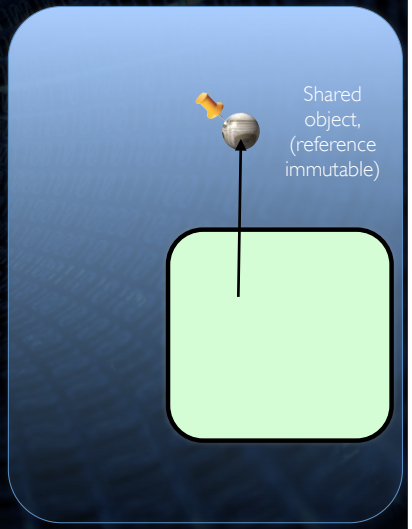
Communication with Java

- Atomic Methods:
  - acquire a lock on guard & pin all reference-immutable arguments
  - start transaction;
  - execute method
  - commit transaction
  - reclaim transient memory
  - unpin all arguments & release lock on guard
- If during execution of the method the Task is scheduled, the transaction is immediately aborted.



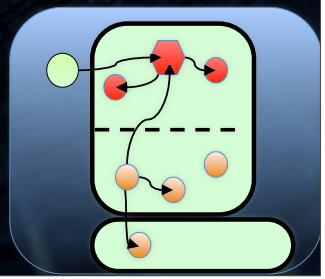
Shared Objects

- Shared objects can be given as startup argument to a task
- They must be pinned
- Restricted to reference-immutable objects to prevent breaking isolation
- *This is a loophole! Breaks determinism guarantees, but sometimes necessary.*



Static safety

- Safety checks must prevent references to transient objects after they have been deallocated and to capsules once they have been sent.
- A simple form of ownership types is used where **`Stable`** is a marker interface for data allocated in the stable heap and **`Capsule`** for messages. Some polymorphism is needed for arrays.
- Checking is done statically, no dynamic tests are need.



Main program

```
// Obtain the graph template:
in =  getResourceAsStream("Detector.ftg");
spec = FlexotaskXMLParser.parseStream(in);


// To instantiate graph with sharing:
sharedArray = new RawFrameArray();
params.put("DetectorTask", sharedArray);
graph = spec.validate("TTScheduler", parameters);


// To obtain reference to Detector
detector = graph.getGuardObject("DetectorTask");
graph.start();


// To communicate a new frame:
int frameIndex = detector.getFirstFree();
frames.get(frameIndex).copy(lengths, callsigns, posns);
detector.setNextToProcess(frameIndex);
```

Detector task

```
interface DetectorGuard implements ExternalMethods {
  void setNextToProcess(int idx) throws AtomicException;
  ...
}

class StateTable implements Stable { … }

class DetectorTask extends AtomicFlexotask
                   implements DetectorGuard {
  private StateTable state;
  private RawFrameArray frames;
  private int nextToProcess;

  void initialize(...,  Object param) {
    frames = (RawFrameArray) param;
    state = new StateTable();
  }
```

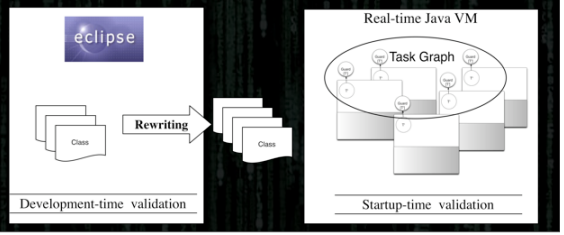

Detector task (ctd)

```
void execute() {
  if (nextToProcess != firstFree) {
    cd = new Detector(state, Constants.GOOD_VOXEL_SIZE);
    cd.setFrame(frames.get(nextToProcess));
    cd.run();
    nextToProcess = firstFree = increment(nextToProcess);
  }
}

int getFirstFree() throws AtomicException {
  int check = increment(firstFree);
  if (check == nextToProcess) return -1;
  int ans = firstFree;
  firstFree = check;
  return ans;
}

void setNextToProcess(int ntP) throws AtomicException {
  this.nextToProcess = ntP;
}
```


Implementation

Implementation Overview

- Eclipse
  - IDE support for task graph construction
  - Static checker for development time warnings
  - Byte-code rewriter for transactions and memory management
- IBM's J9 production real-time VM
  - Start up time verifier
  - Runtime support



Static checking

- Validation performed at compile-time and at start-up time
- Enforce many restrictions for isolation (e.g. limits on `getstatic`), memory safety (ownership and reference immutability), programming model compliance (e.g. no thread operations, or synchronized statements or finalization).
- Inference of 'effectively' final fields.
- Use Rapid Type Analysis with value information at initialization time.
- Can infer Stable classes.

Atomics

- Use Rapid Type Analysis to approximate call graph of atomic methods. Generate a transactional version of all reachable methods.
- Use a single roll-forward transaction log per task.
  Only log changes to stable objects.

Evaluation

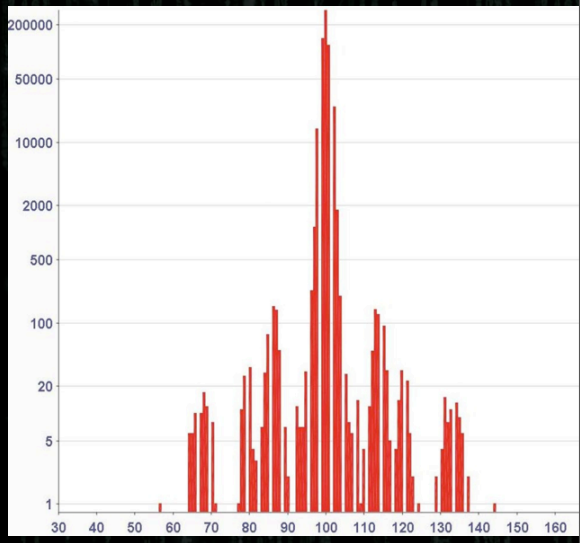
- Experimental setup
  - IBM Websphere Real Time (WRT) VM
  - RHEL 5 Linux, kernel 2.6.21.4 (real-time config)
  - IBM Blade server, 4 dual-core AMD Opteron 64 2.4 GHz, 12GB RAM

Predictability

- High Frequency Reader, 540 LOC.
  - Scheduled for 100us periods, plain Java thread invoking transactional method every 20ms
  - Noise maker thread allocating 2MB per second using 48 byte objects, maintaining live set of 40,000 objects

Predictability

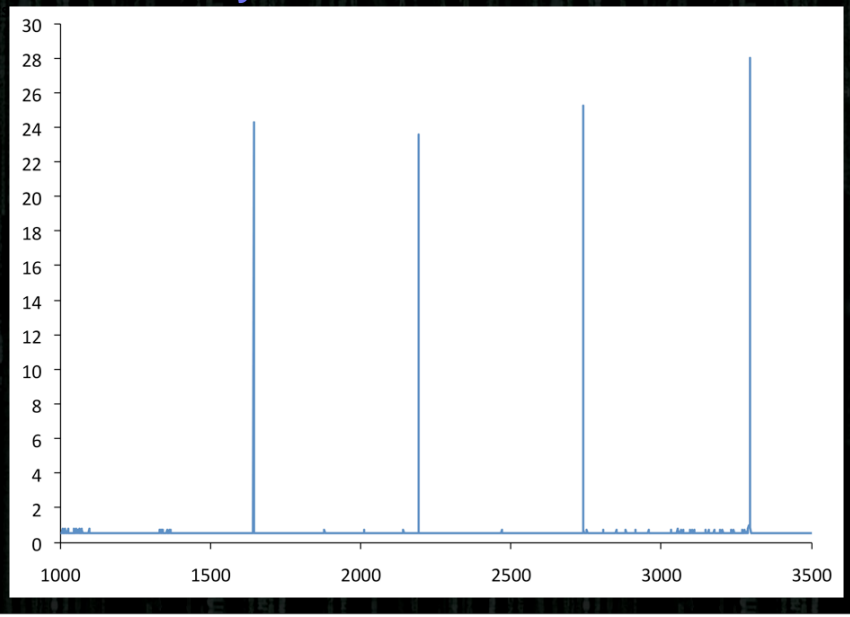
- Summary:
  - 600,000 periodic invocations
  - Inter-arrival time between 57 and 144us
  - 516 aborts of the atomic method



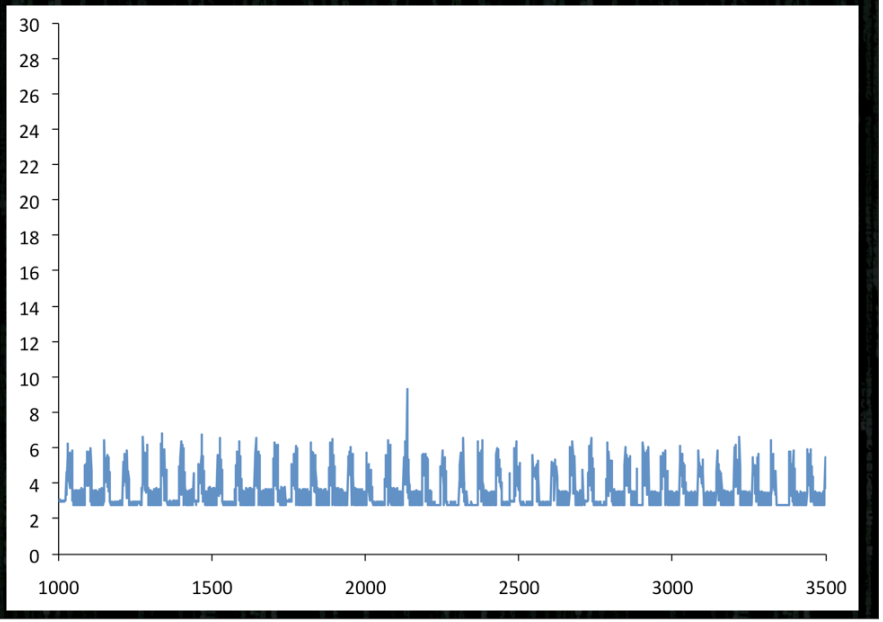
Performance

- Avionics Collision Detector, 30KLOC
  - Flexotask (detector) scheduled for 20ms periods. Plain Java thread (simulator) communicating with Detector every 20ms
  - Noise maker thread allocating 2MB per second using 48 byte objects, maintaining live set of 150,000 objects
  - Four variants: Plain Java, Plain Java/RTGC, (RTSJ/NHRT) and Flexotasks

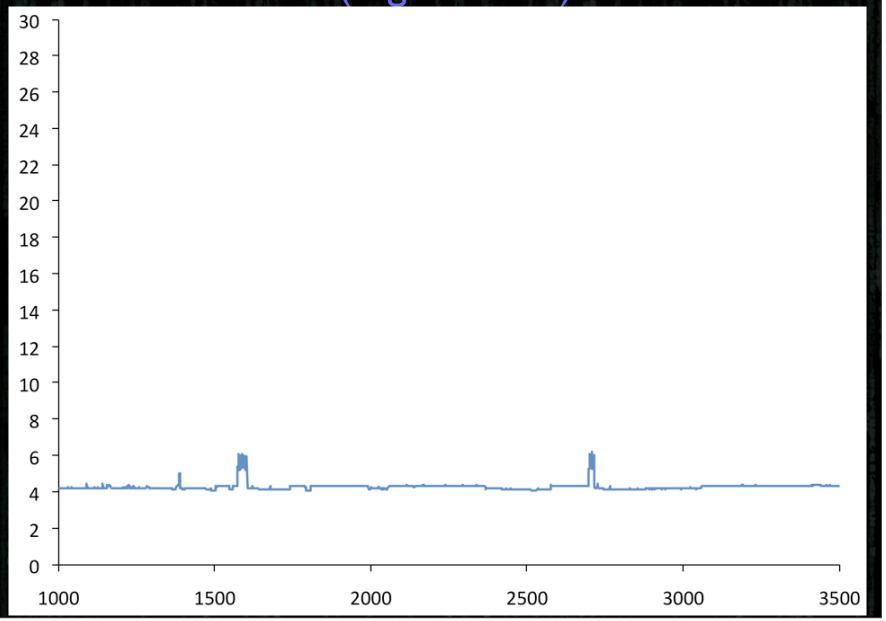
CD with Plain Java



CD with RTGC



CD with Flexotasks (region tasks)

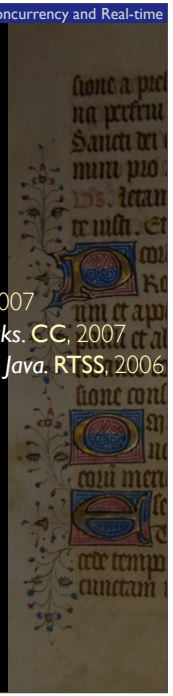


Conclusions

- Flexible tasks graph have a simple semantics and an efficient implementation
- Integration between real-time and non-real-time code allows for gradual adoption
- Implementation in a production quality virtual machine shows good performance

Paper trail

- *A Unified Restricted Thread Programming Model for Java.* LCTES, 2008
- *StreamFlex: High-throughput Stream Programming in Java.* OOPSLA, 2007
- *Garbage Collection for Safety Critical Java.* JTRES, 2007
- *Hierarchical Real-time Garbage Collection.* LCTES, 2007
- *Reflexes: Abstractions for Highly Responsive Systems.* VEE, 2007
- *Scoped Types and Aspects for Real-time Java Memory management.* RTS, 2007
- *Accurate Garbage Collection in Uncooperative Environments with Lazy Stacks.* CC, 2007
- *An Emprical Evaluation of Memory Management Alternatives for Real-time Java.* RTSS, 2006
- *Scoped Types and Aspects for Real-Time Systems.* ECOOP, 2006
- *A New Approach to Real-time Checkpointing.* VEE, 2006
- *A Real-time Java Virtual Machine for Avionics.* RTAS, 2006
- *Preemptible Atomic Regions for Real-time Java.* RTSS,2005
- *Transactional lock-free data structure for Real Time Java.* CSJP, 2004
- *Real-Time Java scoped memory: design patterns, semantics.* ISORC, 2004
- *Subtype tests in real time.* ECOOP, 2003
- *Engineering a customizable intermediate representation.* IVME, 2003



JMM

The Java Memory Model (JMM)

- Java has a complex memory model which gives implementers considerable freedom
- As long as programmers ensures that all shared variables are only accessed by threads when they hold an appropriate monitor lock, they need not be concerned with issues such as multiprocessor implementations, compiler optimizations,
- However, synchronization can be expensive, and there are times when we might want to use shared variables without an locks
- *One example is the so-called double-checked locking idiom. In this idiom, a singleton resource is to be created; this resource may or may not be used during a particular execution of the program. Furthermore, creating the resource is an expensive operation and should be deferred until it is required*

Intuitive Implementation of Resource

```
public class ResourceController {
  private static Resource resource = null;
  public static synchronized Resource getResource() {
    if(resource == null) resource = new Resource();
    return resource;
  }
}
```

- The problem with this solution is that a lock is required on every access to the resource
- In fact, it is only necessary to synchronize on creation of the resource, as the resource will provide its own synchronization when the threads use it

Double-checked Locking Idiom

```
public class ResourceController {
  private static Resource resource = null;
  public static Resource getResource() {
    if(resource == null) {
      synchronized (ResourceController.class){
        if(resource == null)
          resource = new Resource();
      }
    }
    return resource;
  }
}
```

Why is this broken?

The JMM Revisited I

- In the JMM, each thread is considered to has access to its own working memory as well as the main memory which is shared between all threads
- This working memory is used to hold copies of the data which resides in the shared main memory
- It is an abstraction of data held in registers or data held in local caches on a multi-processor system

The JMM Revisited II

- It is a requirement that:
  - inside a synchronized method or statement any read of a shared variable must read the value from main memory
  - before a synchronized block finishes, any variables written to during the method or statement must be written back to main memory
- Data may be written to the main memory at other times as well, however, the programmer just cannot tell when
- Code can be optimized and reorder as long as long as it maintains "as-if-serial" semantics
- For sequential programs, we will not be able to detect these optimizations and reordering. In concurrent systems, they will manifest themselves unless the program is properly synchronized

Double-checked Locking Idiom Revisited I

- Suppose that a compiler implements the `resource = new Resource()` statement logically as follows

```
tmp = create memory for the Resource class
    // tmp points to memory
Resource.construct(tmp)
   // runs the constructor to initialize
resource = tmp // set up resource
```


Double-checked Locking Idiom Revisited II

- Now as a result of optimizations or reordering, suppose the statements are executed in the following order

```
tmp = create memory for the Resource class
    // tmp points to memory
resource = tmp
Resource.construct(tmp)
   // run the constructor to initialize
```

There is a period of time when the `resource` reference has a value but the `Resource` object has not been initialized!

Warning

- The double-checked locking algorithm illustrates that the synchronized method (and statement) in Java serves a dual purpose
- Not only do they enable mutual exclusive access to a shared resource but they also ensure that data written by one thread (the writer) becomes visible to another thread (the reader)
- The visibility of data written by the writer is only guaranteed when it releases a lock that is subsequently acquired by the reader

Memory Models

- Many programming languages do not deal with concurrency, instead a library provides a set of API calls.
- This is brittle in the presence of optimizing compilers and modern architectures
- Java is the first programming language to specify a *memory model* that must be enforced by the compiler
- The memory model gives semantics to concurrent programs, a must if we want to program multicores
  - C++ is getting one -- it must be good

Motivation: Double Checked Locking

- Double checked locking is an idiom that tries to avoid paying the cost of synchronization when not needed
  - Important for lazy initalization
- The prototypical example:

```
// Single threaded version
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
        helper = new Helper();
    return helper;
    }
  // other functions and members...
  }
```

  - See Bill Pugh's web page for the details.

Broken Double Checked Locking

- Avoids cost of synchronization when helper is already initialized:

```
private Helper helper;
Helper getHelper() {
  if (helper == null)
    synchronized(this){if(helper == null) helper = new Helper();}
  return helper;
}
```

- Problems:
  - The writes that initialize the Helper object and the write to the helper field can be out of order. A thread invoking getHelper() could see a non-null reference to a helper object, but see the default values for its fields, rather than the values set in the constructor.
  - If the compiler inlines the call to the constructor, then the writes that initialize the object and the write to the helper field can be reordered
  - Even if the compiler does not reorder, on a multiprocessor the processor or memory system may reorder those writes, as perceived by a thread running on another processor.

More on why it's broken

- A test case showing that it doesn't work (by Paul Jakubik). When run on a system using the Symantec JIT:

```
singletons[i].reference = new Singleton();
```

to the following (note that the Symantec JIT using a handle-based object allocation system).

```
0206106A   mov        eax,0F97E78h
0206106F   call       01F6B210                 ; allocate space for
                                                ; Singleton, return result in eax
02061074   mov        dword ptr [ebp],eax      ; EBP is &singletons[i].reference
                                                ; store the unconstructed object here.
02061077   mov        ecx,dword ptr [eax]      ; dereference the handle to
                                                ; get the raw pointer
02061079   mov        dword ptr [ecx],100h     ; Next 4 lines are
0206107F   mov        dword ptr [ecx+4],200h   ; Singleton's inlined constructor
02061086   mov        dword ptr [ecx+8],400h
0206108D   mov        dword ptr [ecx+0Ch],0F84030h
```

As you can see, the assignment to singletons[i].reference is performed before the constructor for Singleton is called. This is completely legal under the existing Java memory model, and also legal in C and C++ (since neither of them have a memory model).

The Java™ Memory Model: the building block of concurrency

Jeremy Manson, Purdue University
William Pugh, Univ. of Maryland
http://www.cs.umd.edu/~pugh/java/memoryModel/

TS-1630

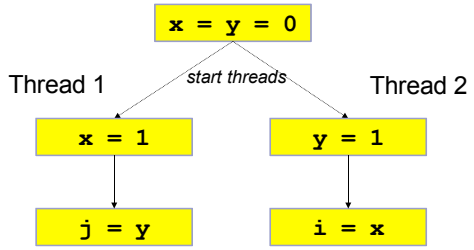
Synchronization is needed for Blocking *and* Visibility

- Synchronization isn't just about mutual exclusion and blocking
- It also regulates when other threads *must* see writes by other threads
  - When writes become visible
- Without synchronization, compiler and processor are allowed to reorder memory accesses in ways that may surprise you
  - And break your code

Don't Try To Be Too Clever

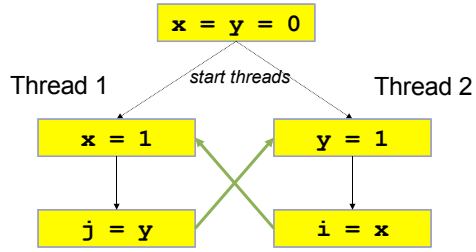
- People worry about the cost of synchronization
  - Try to devise schemes to communicate between threads without using synchronization
    - locks, volatiles, or other concurrency abstractions
- Nearly impossible to do correctly
  - Inter-thread communication without synchronization is not intuitive

Quiz Time



Can this result in i = 0 and j = 0?

Answer: Yes!

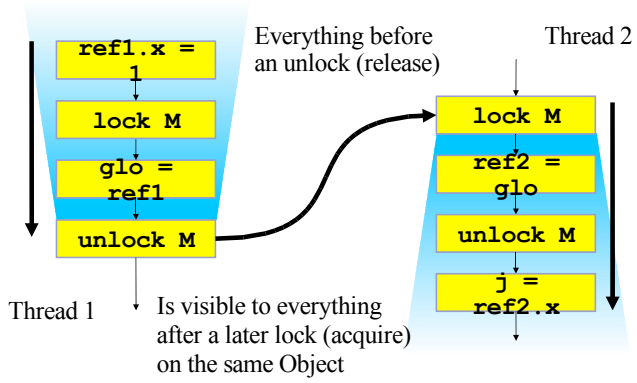


How can i = 0 and j = 0?

How Can This Happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized to global memory
- The memory model is designed to allow aggressive optimization
  - including optimizations no one has implemented yet
- Good for performance
  - bad for your intuition about insufficiently synchronized code

When Are Actions Visible to Other Threads?



Release and Acquire

- All memory accesses before a release
  - are ordered before and visible to
  - any memory accesses after a matching acquire
- Unlocking a monitor/lock is a release
  - that is acquired by any following lock of *that* monitor/lock

JavaOne

Happens-before ordering

- A release and a matching later acquire establish a *happens-before* ordering
- execution order within a thread also establishes a happens-before order
- happens-before order is transitive

JavaOne

Data race

- If there are two accesses to a memory location,
  - at least one of those accesses is a write, and
  - the memory location isn't volatile, then
- the accesses *must* be ordered by happens-before
- Violate this, and you may need a PhD to figure out what your program can do
  - not as bad/unspecified as a buffer overflow in C

Volatile fields

- A field marked **`volatile`** will be treated specially by the compiler
- read/writes go directly to memory and are never cached in registers
- volatile long/double are atomic
- volatile operations can't be reordered by the compiler
- The following example will only be guaranteed to work with volatile because this ensure that stop is not stored in a register:

```
class Animator implements Runnable {
  private volatile boolean stop = false;
  public void stop() { stop = true; }
  public void run() {
    while (!stop)
      oneStep();
      try { Thread.sleep(100); } …;
  }
  private void oneStep() { /*...*/ }
}
```


Volatile fields

- Volatile fields induce happens-before edges, similar to locking
- Incrementing a volatile is not atomic
  - if threads threads try to increment a volatile at the same time, an update might get lost
- volatile reads are very cheap; volatile writes cheaper than synchronization
- atomic operations require compare and swap; provided in JSR-166 (concurrency utils)

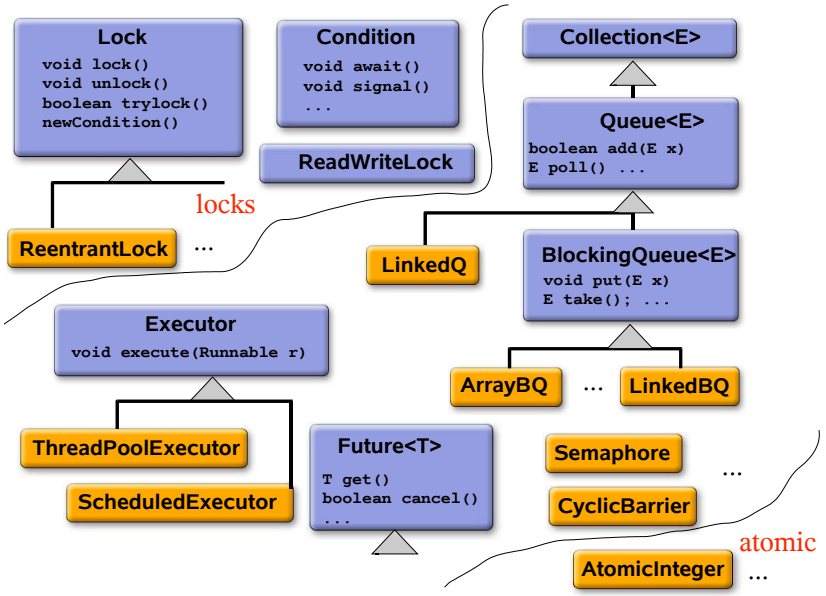
Correct Double Checked Locking

- This avoids the cost of synchronization when helper is already initialized:

```
class Foo {
  private volatile Helper helper;
  Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null) helper = new Helper();
      }
    return helper;
  }
```


Concurrency Utilities and Atomics

- The JSR-166 (designed by Doug Lea at SUNY Oswego) has introduced a rich family of API for concurrent programming



http://gee.cs.oswego.edu

Atomic Variables

- Classes representing scalars supporting
  `boolean compareAndSet(expectedValue, newValue)`
  - Atomically set to `newValue` if currently hold `expectedValue`
  - Also support variant: `weakCompareAndSet`
    - May be faster, but may spuriously fail (as in LL/SC)
- Classes: { *int, long, reference* } X { *value, field, array* } plus boolean value
  - Plus AtomicMarkableReference, AtomicStampedReference
    - (emulated by boxing in J2SE1.5)
- JVMs can use best construct available on a given platform
  - Compare-and-swap, Load-linked/Store-conditional, Locks

Example: AtomicInteger

```
class AtomicInteger {
  AtomicInteger(int initialValue);
  int get();
  void set(int newValue);
  int getAndSet(int newValue);
  boolean compareAndSet(int expected, int newVal);
  boolean weakCompareAndSet(int expected, int newVal);
  //     prefetch              postfetch
  int getAndIncrement();   int incrementAndGet();
  int getAndDecrement();   int decrementAndGet();
  int getAndAadd(int x);   int addAndGet(int x);
}
```

- Integrated with JSR133 memory model semantics for volatile
  - `get` acts as volatile-read
  - `set` acts as volatile-write
  - `compareAndSet` acts as volatile-read and volatile-write
  - `weakCompareAndSet` ordered wrt other accesses to same var

Treiber Stack

```
interface LIFO<E> { void push(E x); E pop(); }

class TreiberStack<E> implements LIFO<E> {
  static class Node<E> {
    volatile Node<E> next;
    final E item;
    Node(E x) { item = x; }
  }

  AtomicReference<Node<E>> head =
        new AtomicReference<Node<E>>();

  public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
      oldHead = head.get();
      newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
  }
```


TreiberStack(2)

```
public E pop() {
   Node<E> oldHead;
   Node<E> newHead;
   do {
     oldHead = head.get();
     if (oldHead == null) return null;
     newHead = oldHead.next;
   } while (!head.compareAndSet(oldHead,newHead));

   return oldHead.item;
 }

}
```


Java for Safety-Critical Applications

Java Community Process Expert Group members (JSR 302)

Thomas Henties	Siemens
James J. Hunt	aicas
Doug Locke	Locke Consulting
Kelvin Nilsen	Aonix
Martin Schoeberl	Vienna University of Tech.
Jan Vitek	Purdue University

In Real-time Systems…

… it is mission-critical that –besides computing correct results– these results be available in a given time frame.

If the deadline is missed:

soft real time ➔ Loss of quality

- Example: video streaming, circuit switching

hard real time ➔ Engine failure or damage

- Example: robot control; Transrapid voltage control

safety critical ➔ Human beings may die

- Example: aviation, military, medicine, power plants

Java real-time issues

Java Technology was not really designed for real-time systems

- **Supports threads, but provides inadequate scheduling control**
  - Synchronization delays unpredictable
  - No predictable event processing
  - No "safe" asynchronous transfer of control
- **Requires garbage collection**
- **Very coarse timer support** (until Java 5)
- **Limited hardware access**

Real Time Specification for Java

Short overview of RTSJ

javax.realtime

JSR 001 & JSR 282

RTSJ
Memory allocation regions

Heap Memory:

- *Objects remain until no references exist, then reclaimed by GC*
  - Can contain references to Immortal Memory
  - Garbage collector may cause delays, depending on GC performance

Immortal Memory:

- *Objects remain until the application terminates*
  - Is never Garbage Collected
  - Can contain references to Heap Memory

Scoped Memory:

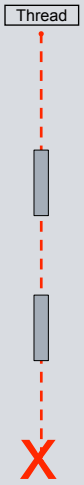
- *Objects remain until last thread leaves it, then it is totally reclaimed*
  - Is never Garbage Collected
  - Can contain references to Immortal, Heap and enclosing scopes

RTSJ
Threads

Normal Java Threads *(java.lang.Thread)*
- No restrictions on its access, but cannot reach Scoped memory
- Cannot get predictable performance

RealtimeThread *(javax.realtime.RealtimeThread)*
- Can access all memory regions
- May be delayed by Garbage Collector
- Depends on the Garbage Collector behavior and performance

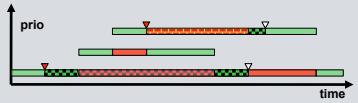
NoHeapRealtimeThread *(javax.realtime.NoHeapRealtimeThread)*
- Can access only Scoped and Immortal memory
- Can always preempt Garbage Collector
- Never delayed by Garbage Collector



RTSJ
Priorities and Synchronization

Priorities:
- At least 28 real-time priority levels plus the 10 standard priority-like levels
- Real-time priorities taking precedence over standard priorities

Priority Inversion avoidance
- Priority inheritance
- Priority ceiling



Synchronization
- *javax.realtime* methods are intentionally not synchronized
- Priorities are used to control scheduling
- *synchronized* may be used on application level

RTSJ
Scheduling and ATC

Scheduling:
- Priority scheduler is minimally required
- Flexible interface for more advanced custom scheduling policies
- Dramatic effects on
  - RealtimeThreads
  - NoHeapRealtimeThreads
  - AsyncEventHandlers

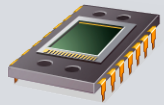
Asynchronous Transfer of Control
- Asynchronous Interrupt Exceptions can be raised by any thread
- Interruption by Asynchronous Events can be deferred (default)
- Events can be triggered arbitrarily by:
  - *fire()*
  - timers
  - signals
  - happenings

RTSJ
Timers and Memory Access

High Resolution Timers
- Specified in nanoseconds
- Actual resolution is implementation-defined

Raw Memory & Physical Memory Access

Provides direct access to memory areas, accessible through the OS
- Memory mapped areas
- Consider endianess (BYTE_SWAP)
- DMA areas
- Flash memory areas
- … <implementation defined>



Safety Critical Specification for Java

SCSJ

javax.safetycritical

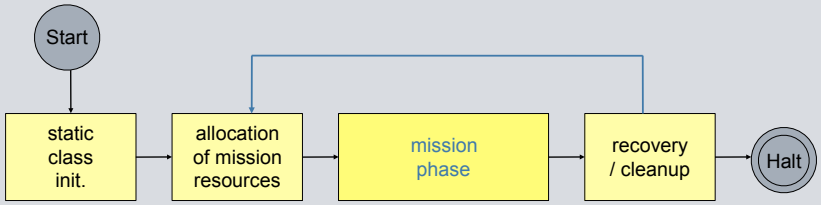
JSR 302

SC Java
Goal

A specification for Safety Critical Java capable of being certified under DO-178B Level A

- Implies small, reduced complexity infrastructure (i.e. JVM)
- Emphasis is on defining a minimal set of capabilities required by implementations
- Based on HIJA – High-Integrity Java Application (EU project)

SC Java
Phases of a Safety Critical Java Application (safelet)

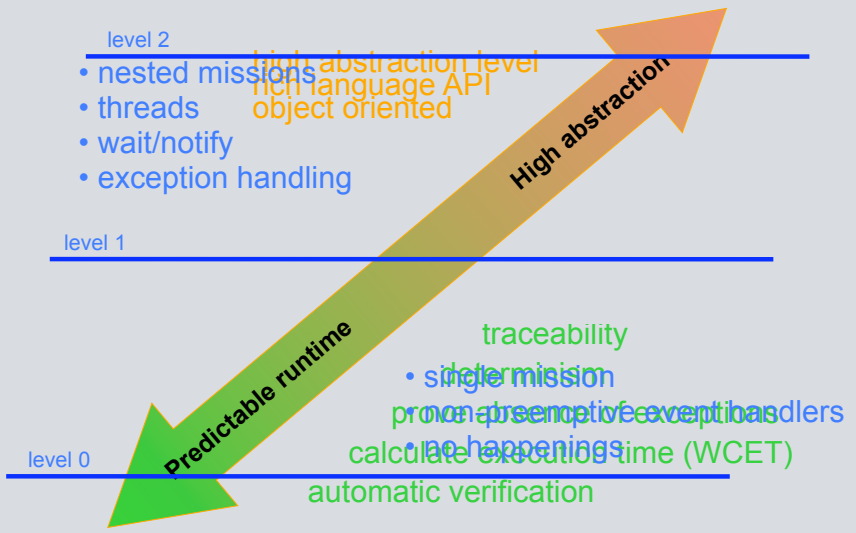


- All classes are loaded and initialized in advance
- Resources are allocated and initialized
  - Application startup will not require heap memory
  - Data structures created and initialized in Mission Memory
  - Additional NHRT-threads may be created and started
- Mission phase
  - Scoped memory areas used for limited dynamic allocation

SC Java
Main points to meet certification demands

- **No Heap**
  - Garbage Collector not necessary
  - Predictable allocation time
  - → Lots of *java.lang* classes cannot be used
- **Scoped Memory**
  - Nesting restricted
  - Reference safety to be statically analyzable
- **Scheduling**
  - All Schedulable Objects will be non-heap
  - Fully controlled deterministic scheduling

Safety Critical Java compliance levels



Vendors of Real Time and SC Java VMs

Vendors

Real time VMs

Available Java Virtual Machines with Real Time capabilities

- Sun: Java Real Time System
- IBM: WebSphere Real Time
- Aonix: PERC Pico
- Aicas: JamaicaVM

Past, present and future of Real-Time related JSRs

- **JSR-1: The Real-Time Specification for Java:**
  Proven Real-Time Java flying to the Mars
- **JSR-50: Distributed Real-Time Specification for Java**
  Support end-to-end application timeliness
- **JSR-282: RTSJ version 1.1**
  Adding few new features. Ease use of Scoped Memory
- **JSR-302: Safety Critical Java**
  Simplify JSR-1 / 282 to enable Static Analysis
- **JSR-xxx (?)**
  Real-Time and Safety Critical Java utilize Multi-Core CPUs

Memory Management for Real-time Java

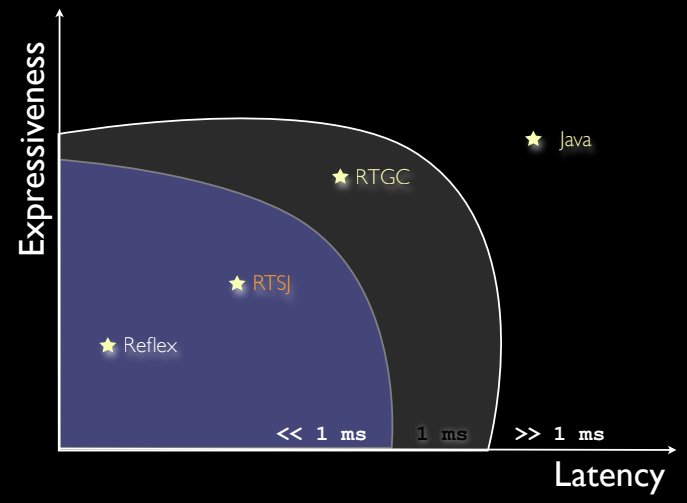
Jan Vitek

PURDUE UNIVERSITY IBM

Overview

- Memory management & Java
- Scoped Memory
- Real-time Collectors:
  - AICAS Jamaica, Sun McKinak, IBM Metronome
- Lock-free Collectors
- Conclusions

Design space



Background: Manual Memory

- Memory management is the source of many errors (dangling pointers, memory leaks) in traditional software systems due to the non-local nature of the allocation/deallocation protocol
- One can sometimes prevent such errors by avoiding allocation, at the cost of over provisioning, but not in general.
- Programmers often use custom 'allocators' in conjunction with object pools
- *... but rich data structures and concurrency make manual management hard*
- *... requires synchronization on multicores*
- *... must deal with fragmentation*

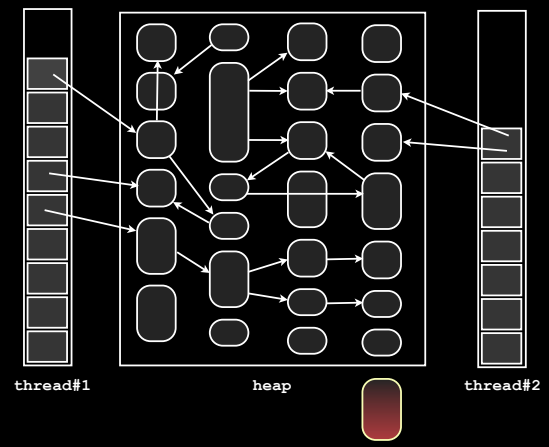
Garbage Collection

PURDUE UNIVERSITY Fiji Systems LLC IBM

Garbage Collection

- A Garbage Collector (GC) is an algorithm that automatically finds unused objects in the memory of an application and prepares them for reuse
- GC frees programmers from worrying about the exact lifetime of objects and ensures that the heap will not be corrupted by access to previously freed data
- *... but introduces pauses that may be **O**(heap) and can increase the memory required.*
  Moreover, pauses occur at unpredictable times, especially in concurrent programs

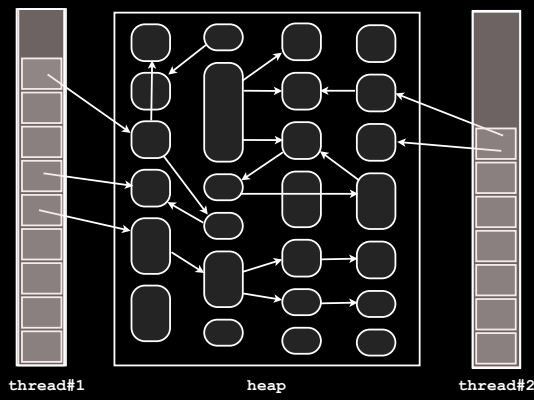
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

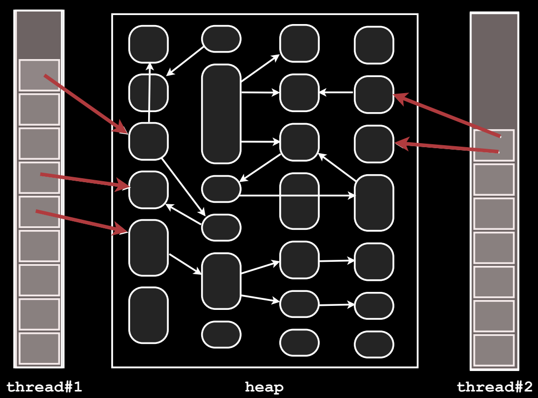
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

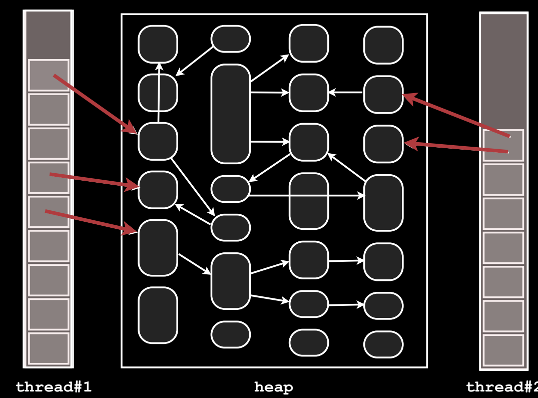
Garbage Collection



Phases

- Mutation
- Stop-the-world
- **Root scanning**
- Marking
- Sweeping
- Compaction

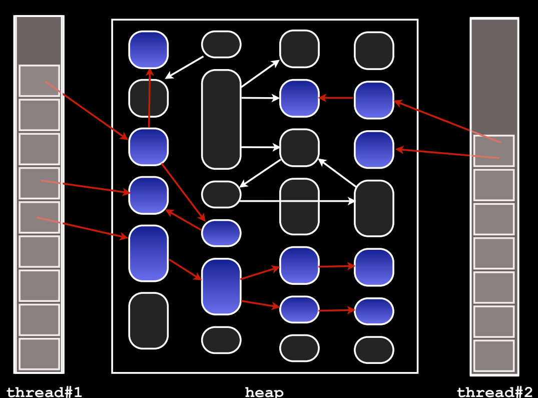
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- **Marking**
- Sweeping
- Compaction

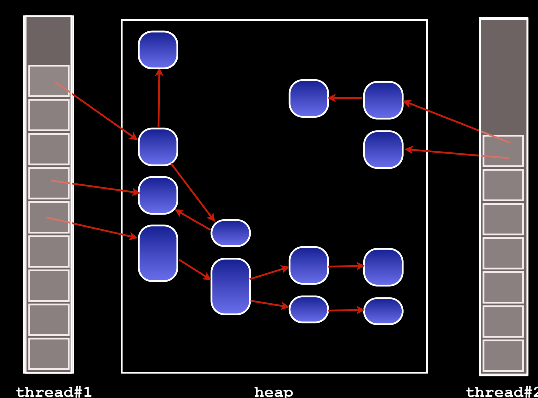
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- **Marking**
- Sweeping
- Compaction

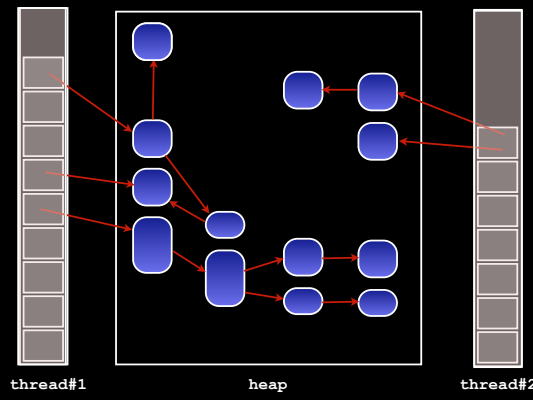
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- **Sweeping**
- Compaction

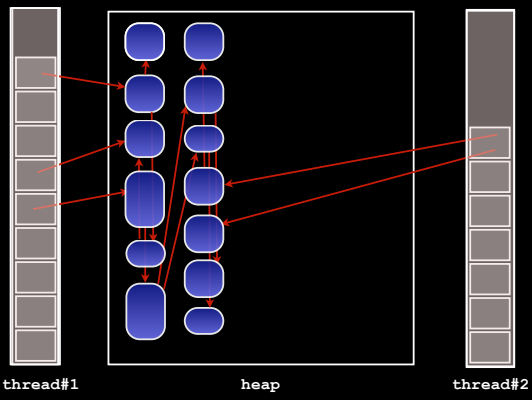
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

GC is Easy

- If responsiveness is not an issue,
  the GC can complete in one long pause under the assumption that there is no interleaved application activity
- Marking is easy
  if the graph does not change while you are searching it.
- Copying/compacting objects and fixing up the heap is easy
  if the application is prevented from accessing the heap

Real-time Garbage Collection

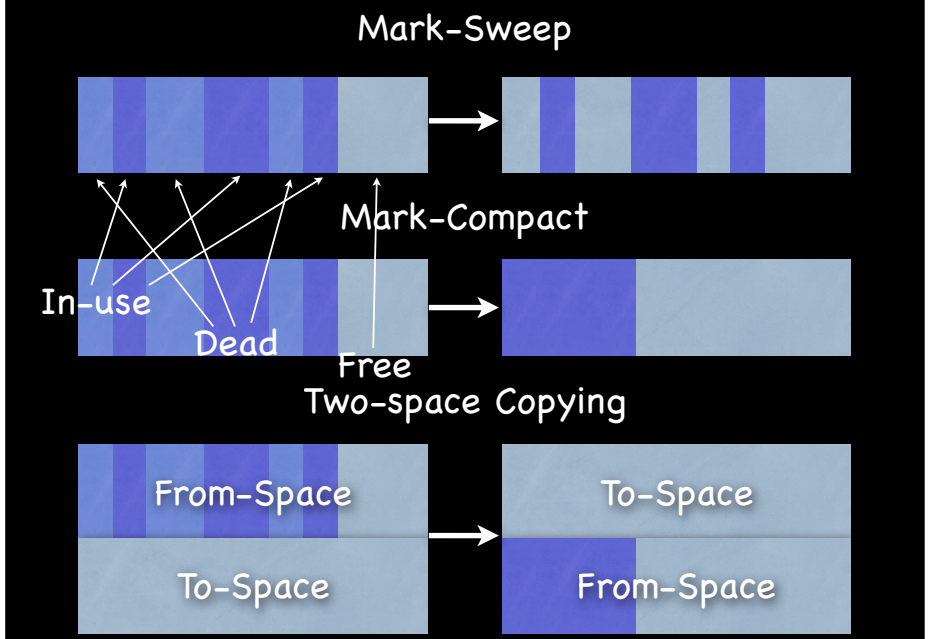
(S³) PURDUE UNIVERSITY Fiji Systems LLC IBM

Real-time GC

- A Real-time GC must provide time and space predictability
  - provide a performance model that can be used to guarantee that programs do not run out of memory or experience pauses that violate their timing constraints
- A Real-time-GC must support defragmentation of the heap if it is to be used with long-lived applications
- Multi-processor support is unavoidable
- Throughput should not degrade overly

Main Collector Types

- Reference counting
  - keep count of incoming references. Requires auxiliary GC to reclaim cycles.
- Mark-Sweep
  - one phase to mark used objects, another to sweep unused space. No copying is performed.
- Mark-Compact
  - one phase to mark used objects, another to copy all used objects to one side of the heap.
- Copying
  - one phase to simultaneously mark and evacuate all used objects; the space that previously held all objects is then completely free.



RTGC: state of the art

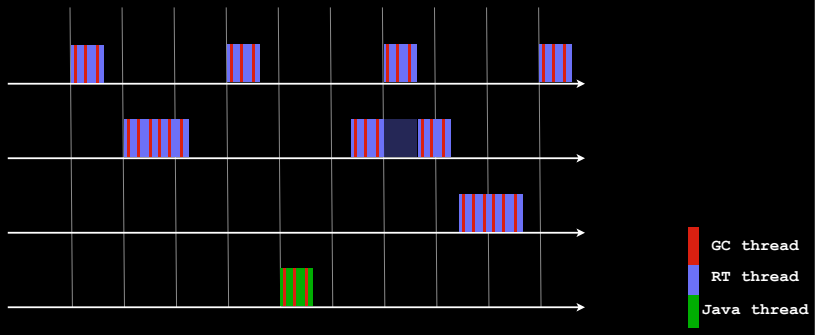
- Sort-of Real-time GCs:
  - Too many to list, Baker (1978) was first
- Really Real-time GCs:
  - JamaicaVM (1999) - *work-based*
  - Henriksson (1998) - *slack-based*
  - Metronome (2003) - *time-based*

Work-based RTGC

Baker'78, Siebert'99

(S^3) PURDUE UNIVERSITY Fiji Systems LLC IBM

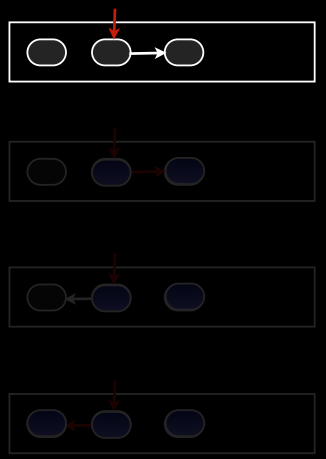
Work-based scheduling



Baker

- First attempt at RTGC by Baker '78
- For LISP where all objects are the same, small, size
- Copying collector that instruments the mutator to maintain:
  - To-space invariant:
    - mutator uses to-space, never sees from-space, objects are copied on first read
  - Steady state:
    - collector keeps up with mutator allocations, collection work performed on each allocation

Incrementalizing marking



Dissecting Baker

- The chief problem with Baker is the to-space invariant
  - Reads are frequent, a heavy read barrier leads to poor performance
  - Variable sized objects in Java make costs harder to bound
- What RT programmers fear:
  - a burst of copying reads, causing substantial slow-downs

Siebert

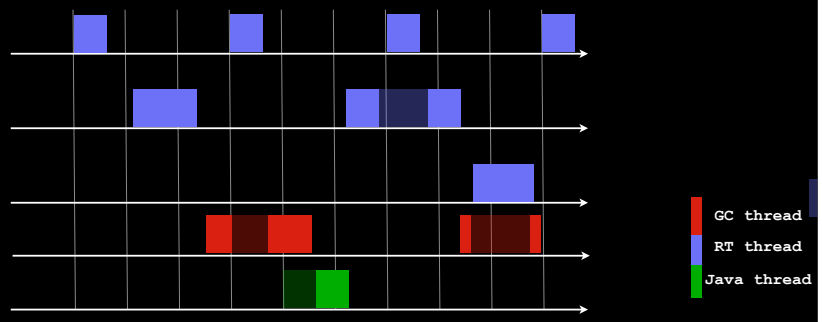
- Modern work-based collector in the JamaicaVM:
  - Mark-Sweep, no copying
  - All objects are same size, 32 bytes
    (large objects⇒ lists, arrays⇒ tries) Stack is logically an object
  - Root scanning is fully incremental
  - Write barrier to mark objects (including the stack)
  - Allocation triggers a bounded amount of collector work

Slack-based RTGC

Henriksson'98

(S³) PURDUE UNIVERSITY Fiji Systems LLC IBM

Slack-based GC Scheduling

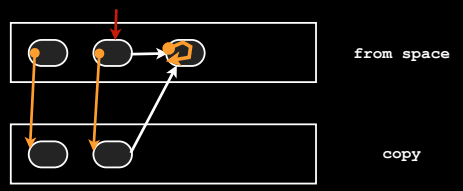


Slack-based GC

- Henriksson takes a different approach to real-time guarantees:
  - tries to guarantee real-time tasks never experience GC interference
- How is this done?
  - RT tasks can always preempt GC
  - Objects have normal representation.
  - Read & write barriers are used to ensure marking/copying soundness.
  - Copying collector, with roll backs when the mutator preempts GC

Incrementalizing copying/compaction

- Forwarding pointers are used to refer to the current version of the object.
- Every access must start with a derefence



Dissecting Henriksson

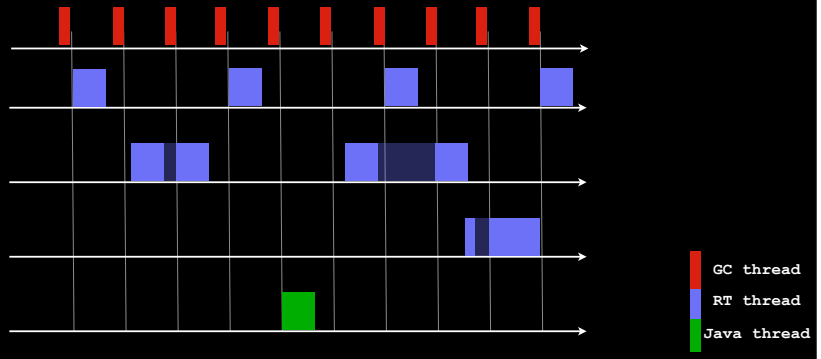
- Pros:
  - Gives the programmer full control over the GC schedule
  - SUN uses a variant of this collector in their RTSJVM
- Cons:
  - Read barrier are expensive
  - Mutator induced copy aborts make collector progress hard to ensure
  - Programmer must know allocation rate, and to budget for GC work
  - Priority inversion can lead to GC interference

Time-based RTGC

Metronome'03

PURDUE UNIVERSITY Fiji Systems LLC IBM

Time-based GC Scheduling



Time-based GCs

- Metronome runs periodically for a bounded amount of time
  - Amount of collector interference is easy to understand
  - Progress is easier to guarantee than in Henriksson
- A Mark-Sweep-and-sometimes-Copy collector:
  - Write barrier to ensure marking soundness
  - Light read barrier to ensure copying soundness
  - Almost-empty pages evacuated in response to fragmentation
  - Arrays are split into tries with page-sized leaves
  - Collector increments lower-bounded by time to copy a page-size objects

Suming up

Siebert

Local ref. assignment	mark(b) a = b
Primitive heap load	a = b->f
Reference heap load	a = b->f mark(a)
Primitive heap store	a->f = b
Reference heap store	mark(b) a->f = b

Henriksson

Local ref. assignment	a = b
Primitive heap load	a = b->forward->f
Reference heap load	a = b->forward->f mark(a)
Primitive heap store	a->forward->f = b
Reference heap store	mark(b) a->forward->f = b->forward

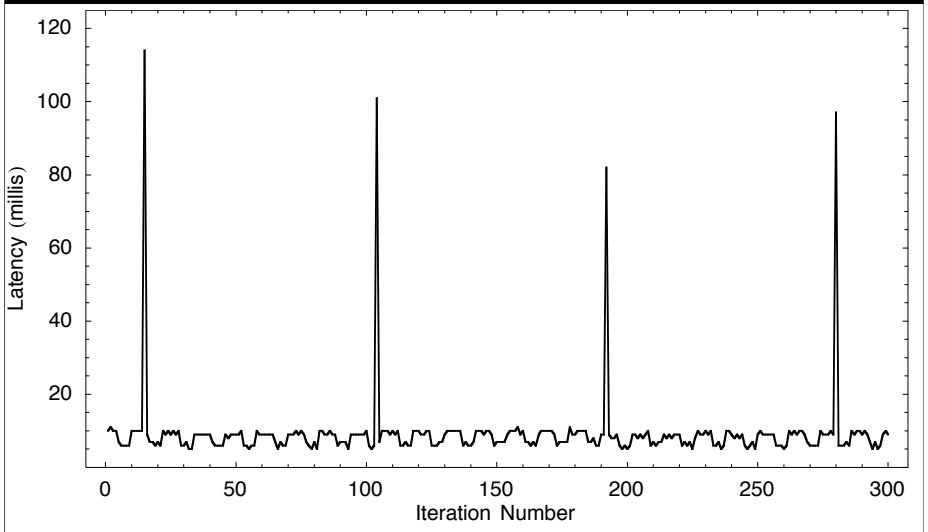
Metronome

Local ref. assignment	a = b
Primitive heap load	a = b->forward->f
Reference heap load	a = b->forward->f
Primitive heap store	a->forward->f = b
Reference heap store	mark(a->forward->f) a ->forward->f = b->forward

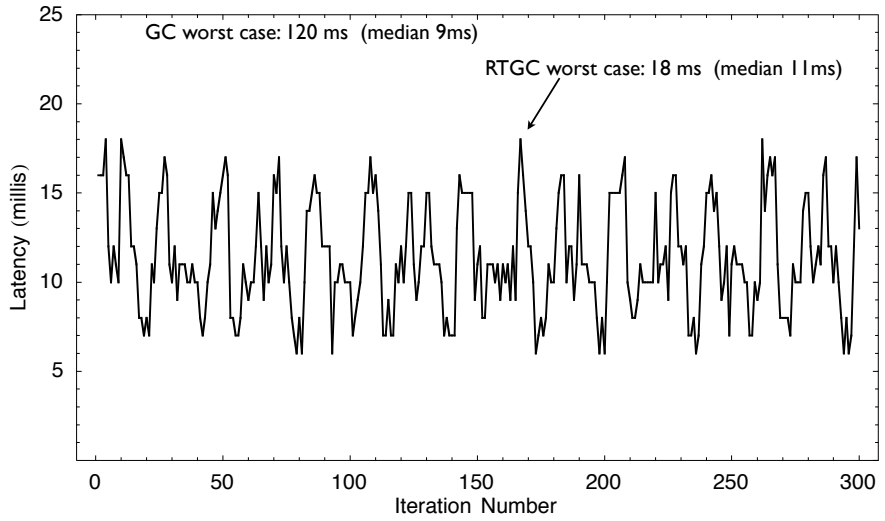
Collision Detector

- Experiment:
  - Pentium IV 1600 MHz, 512 MB RAM, Linux 2.6.14, GCC 3.4.4
  - Application: Real-time Java collision detector (20Hz)
  - Virtual machine: Ovm

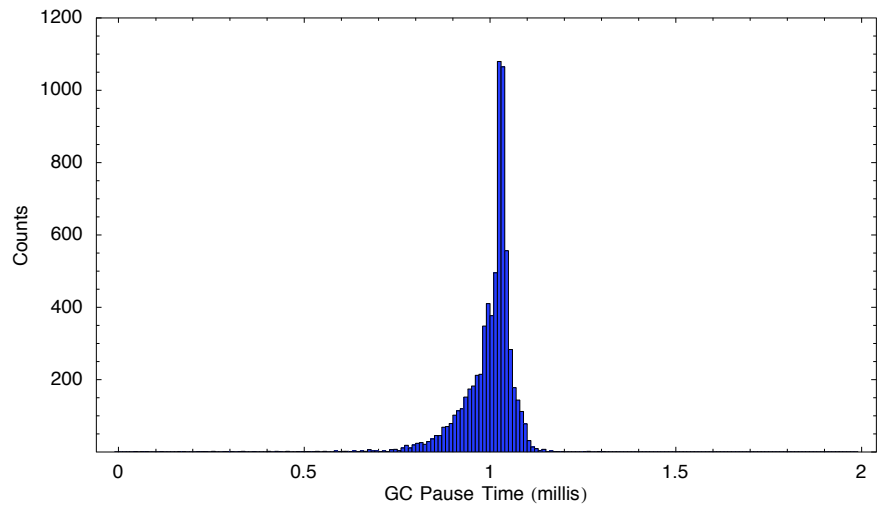
CD with GC



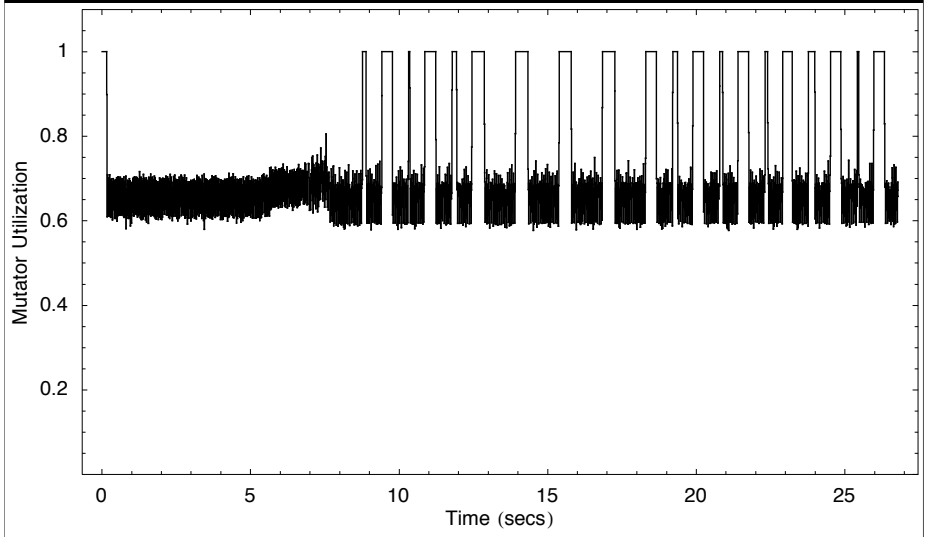
CD with RTGC



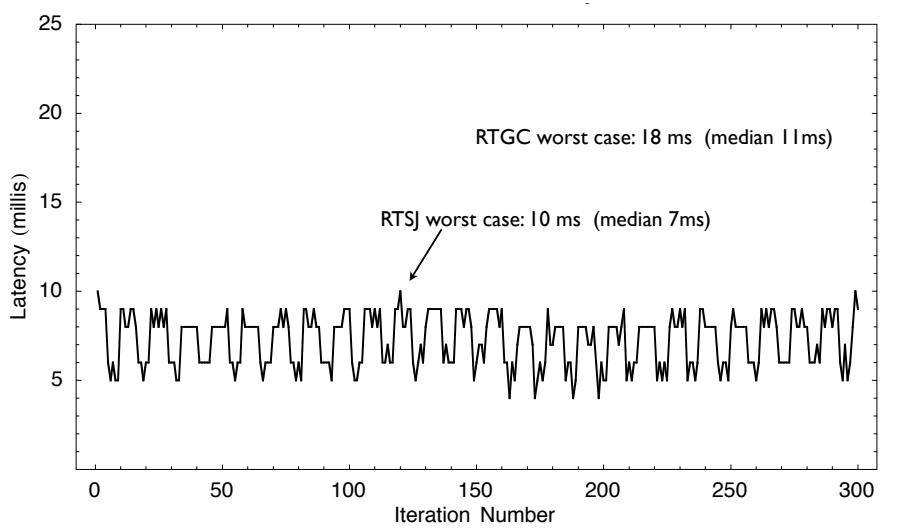
RTGC Pause time distribution



Utilization



CD with RTSJ



CD on J9

- The implementation seems to favor GC over scopes