The Real-time Specification for Java

PURDUE UNIVERSITY

Fiji Systems LLC

IBM

Roadmap

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

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

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

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 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());
```

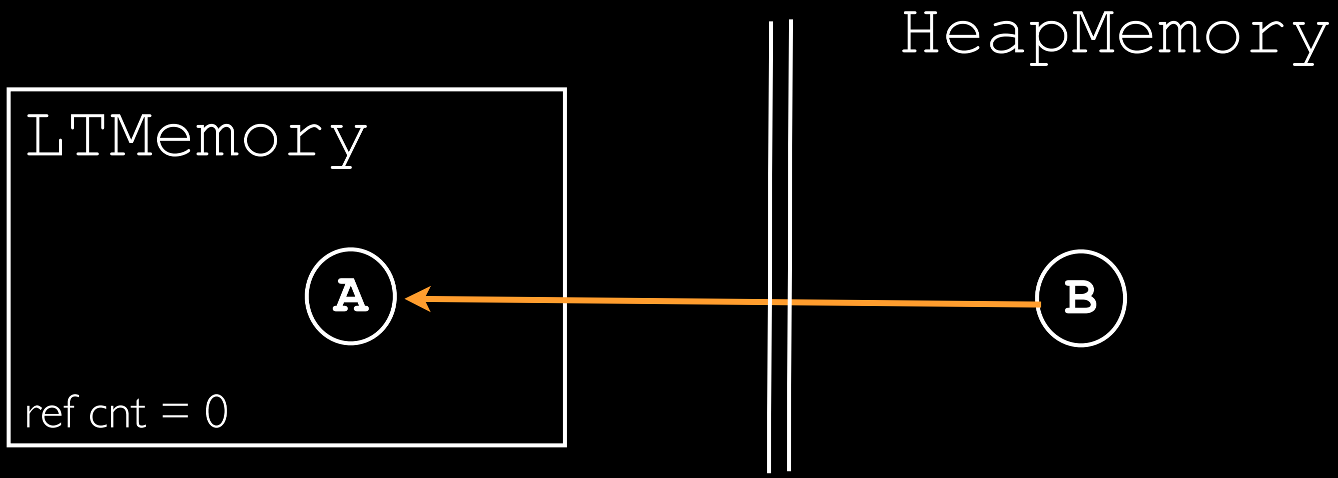
Roadmap

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

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

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

Roadmap

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

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;
```

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

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);
```

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

- 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 behavior 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 priority assignment algorithm is mandated, usually shorter deadline ⇒ higher 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)

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 r,
                 MemoryParameters mem);
  public boolean setReleaseParametersIfFeasible(
                 ReleaseParameters r);
  public boolean setSchedulingParametersIfFeasible(
                 SchedulingParameters s);
  public MemoryParameters getMemoryParameters();
  public void setMemoryParameters(MemoryParameters mem);
  public ReleaseParameters getReleaseParameters();
  public void setReleaseParameters(ReleaseParameters r);
  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 s);
}
```

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 to be changed at run time
- Supports priority inheritance or priority ceiling emulation inheritance for synchronized objects (covered later)

The Priority Scheduler: Mechanism

- Supports preemptive priority-based dispatching of schedulable objects —always run highest priority runnable SO
- Does not defined where in the run queue (associated with a priority level), a preempted object is placed; however, a particular implementation is required to document its approach
- Places a blocked SO which becomes runnable, or has its priority changed at the back of the run queue associated with its 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

The PriorityScheduler Class

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

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

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);
}
```

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 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 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 bound 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

- 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

```
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

- 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);
```

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

- 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

- 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

- 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; `duty-Doctor` 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

```
class PanicButtonHandler extends AsyncEventHandler {
  private AbsoluteTime lastPage = new AbsoluteTime(0,0);
  private Clock clock = Clock.getRealtimeClock();
  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, PatientMonitor s) {
    nursePager = n;doctorPager = d; patient = s;
    button.addHandler(this);
  }
```

PanicButtonHandler II

```
void handleAsyncEvent() {

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

Configuration

```
AsyncEvent nursePager = new AsynEvent();
AsyncEvent doctorPager = new AsynEvent();
PatientMonitor 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
} } );
```

Roadmap

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

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

```
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

- 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

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

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
   ...
  }
}
```

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

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*

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

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

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

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 Basic Model

- Priority inversion can occur whenever a SO blocks 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 SOs waiting for an object lock)
    - Where there is more than one SO in the queue at the same priority, the order should be first-in-first-out (FIFO). Similarly, queues resulting from calling the Object.wait method 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 whenever a schedulable object is blocked waiting for a resource

Monitor Control

```
public abstract class MonitorControl {
  protected MonitorControl();
  public static void setMonitorControl(MonitorControl plcy);
  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 max of its static priority and the ceiling values of any resource it locked
- Thus, a thread will only block at the beginning of its execution
- Once the thread starts actually executing, all the resources it needs are free; otherwise some thread would have an equal or higher priority and the thread's execution would be postponed

Priority Ceiling Emulation

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

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 unchecked CeilingViolationException is thrown