Machine Problem 1: Cyclic Executive in Java

(Due Date: To Be Announced)

Introduction

In this machine problem, we will acquire an understanding of the actual operation of scheduling subsystems for real-time systems. Following the approach used in Jane Liu's textbook, we will use a language with threading support (in our case Java) as the underlying platform.

In this particular exercise, we will be implementing a cyclic executive scheduler. First, we will implement a basic cyclic executive scheduler, which handles periodic real-time tasks only. We will then add capabilities to handle aperiodic tasks, with and without slack stealing. As a stretch goal, we will implement a sporadic task server, based on an EDF scheduler.

We will not be implementing the whole system from scratch. Instead, we will be making use of a significant amount of skeleton code and of pre-defined interfaces. This will (a) make sure that we all proceed roughly in the right direction, (b) allow us to do most of the work without detailed knowledge of Java, and (c) significantly reduce the amount of work required.

The skeleton code and the interfaces somewhat mimic a simplified version of the Real-Time Java Specification [www.rtj.org], which is likely to become the industry standard for Real-time Java platforms.

Note: It is unlikely that a real-life system would be implemented the way we do it here. The mechanisms that we use carry much overhead. For example, we will be realizing all tasks (periodic, aperiodic, and sporadic) as extensions to Java threads. (For one, Java itself uses a priority-driven scheduler for its threads; this means that we are running a cyclic executive over a priority-driven scheduler. Also, some Java Virtual Machines create separate OS threads to handle Java threads; this means that we have scheduling happening at three layers in total: our real-time scheduler, the Java scheduler, and the OS scheduler.) In particular for periodic tasks this is not appropriate, where a real cyclic executive would handle individual chunks of code instead of blocking and unblocking existing threads.

Real-Time Thread Model

The thread model rather closely follows the traditional Java thread model, except that all threads are under the control of a scheduler thread. Instead of using **Thread** classes, we use **RealTimeThread** classes, which inherit the **java.lang.Thread**. The class diagram in Figure 1 illustrates the relationship between the various classes.

Each system contains a single scheduler object, which starts, dispatches, and generally manages the schedulable objects in that system. The scheduler becomes aware of a new schedulable object whenever such an object gets created. The scheduler then starts executing the object when the application invokes the `start()` method of the schedulable object.

Aperiodic and Sporadic threads provide a `triggerInvocation()` call, which is called by the application, and simulates the asynchronous invocation of the threads.

Classes

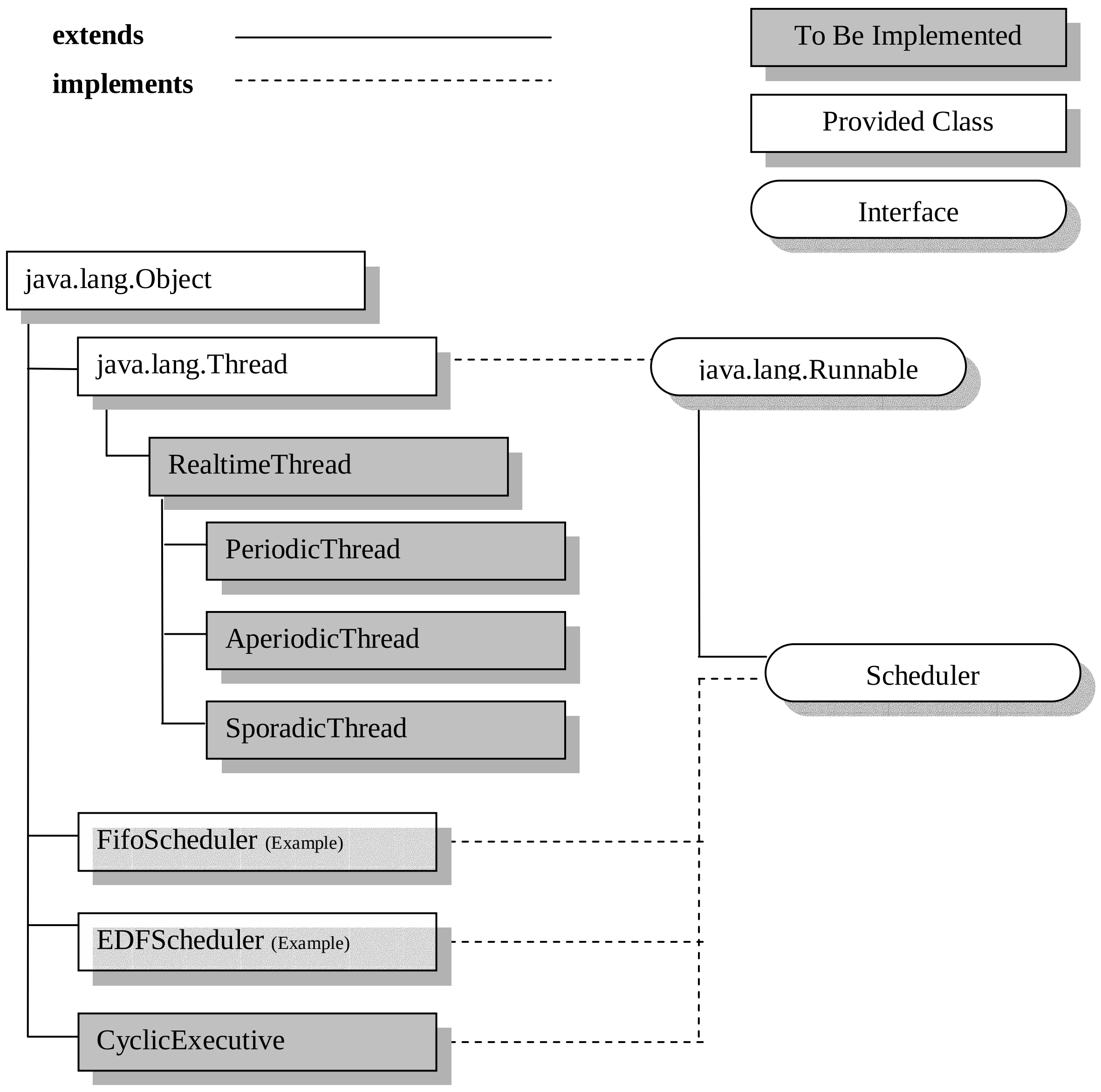

Figure 1 Class Diagram

Class `RealtimeThread`

Declaration
`public class` **`RealtimeThread`** `extends` **`java.lang.Thread`**

Implemented Interface: **java.lang.Runnable**

Description
The class **RealtimeThread** implements the interface **java.lang.Runnable**. The instances of **RealtimeThread** class are managed by the scheduler. The constructor of **RealtimeThread** informs the scheduler of the new schedulable task. The execution of the thread is then initiated upon invocation of the method start().

Constructor
`public RealtimeThread()`
Constructs an instance of class **RealtimeThread.**

Methods
`public void start()`
Start "this", the **RealtimeThread** instance.
This method overrides the method **java.lang.Thread**.start().
Because **java.lang.Thread**.start() method creates and starts an OS thread.

`public boolean waitForNextInvocation()`
This blocks the thread until the next invocation is released. For periodic threads, the next invocation is released by the scheduler. For aperiodic and sporadic threads, the next invocation is released by the application program, using the `triggerInvocation()` method. (In this way we simulate asynchronous invocations.) True when the thread is not in a deadline miss condition. False when the last invocation just missed the deadline. The return value is always True for aperiodic threads.

Class `PeriodicThread`

Declaration
`public class` **`PeriodicThread`** `extends` **`RealTimeThread`**

Implemented Interface: **java.lang.Runnable**

Description
The class **PeriodicThread** inherits **RealtimeThread**. The instances of **PeriodicThread** class (and their executions) are managed by the scheduler.

Constructor
`public PeriodicThread(long period, long execTime)`

Constructs an instance of class **PeriodicThread** with given parameters**.** The thread is added to the pool controlled by the periodic-tasks server by calling the **Scheduler**`.add()` method.

Methods
```
public long getPeriod();
public long getExecTime();
```

Class `AperiodicThread`

Declaration
`public class` **`AperiodicThread`** `extends` **`RealTimeThread`**

Implemented Interface: **java.lang.Runnable**

Description
This class implements aperiodic tasks.

Constructor
`public AperiodicThread(long execTime)`

Constructs an instance of class **AperiodicThread.** The thread is added to the pool controlled by the aperiodic-tasks server by calling the **Scheduler**`.add()` method. The execTime parameter is an estimate on the execution time, and allows for a better scheduling of aperiodic tasks.

Methods
`public void triggerInvocation()`

This method is called by the application, and simulates the asynchronous triggering of a new invocation of the task.

`public long getExecTime();`

Class `SporadicThread`

Declaration
`public class` **`SporadicThread`** `extends` **`RealTimeThread`**

Implemented Interface: **java.lang.Runnable**

Description
This class implements sporadic tasks.

Constructor
`public SporadicThread(long execTime, long deadline)`

> Constructs an instance of class **SporadicThread.** The thread is added to the pool controlled by the sporadic-tasks server by calling the **Scheduler**.`add()` method. The parameter execTime specifies the worst-case execution time, which is used to determine at invocation time whether the invocation can be feasibly scheduled.

Methods
`public boolean triggerInvocation()`

> This method is called by the application, and simulates the asynchronous triggering of a new invocation of the task. Returns True if the new invocation can be accepted, False otherwise.

```
public long getExecTime();
public long getDeadline();
```

Interface `Scheduler`

Declaration
`public interface` **`Scheduler extends java.lang.Runnable`**

Known Implementing Classes: **FIFOScheduler, EDFScheduler, CyclicExecutive**

Description
The interface **Scheduler** defines how to manage the execution of schedulable objects.

Methods
(**Note:** Except for the constructor and the add() method, the scheduler is invisible to the application. You are therefore free to make some modifications to this interface. This means that you may make do with a subset of these methods or with modifications to some of these methods.)

`public static` **`Scheduler`** `getScheduler()`

> Returns the instance of current running instance of **Scheduler**.
> If none of **Scheduler** is running, this method invokes one instance of **Scheduler**.

`public boolean add(`**`RealtimeThread`** `task)`

> Inform the scheduler of this task, the instance of **RealtimeThread**.
> Returns true if the resulting system is feasible.
> Returns false if not.

`public void start(`**`RealtimeThread`** `task)`

> Start this task.

This method will typically create a Java thread.

`public void stop(`**`RealtimeThread`**` task)`
Stop this task, the instance of **RealtimeThread**.

`public void suspend(`**`RealtimeThread`**` task)`
Suspend this task, the instance of **RealtimeThread**.

`public void resume(`**`RealtimeThread`**` task)`
Resume this task, the instance of **RealtimeThread**.

`public long getCurrentTime()`
Return the current time in milliseconds.

Class `CyclicExecutive`

Declaration
`public class `**`CyclicExecutive`**` implements `**`Scheduler`**

Implemented Interfaces: **java.lang.Runnable, Scheduler**

Description
The **CyclicExecutive** class is for Timer-Driven scheduling. The following is a very much simplified realization – in pseudocode – of the `run()` method for the cyclic executive scheduler.

```
public final void run() {
    long waitingTime;
    long currentTime;

      synchronized(this) {
        while(true) {
            // Record current time.
              currentTime = getCurrentTime();
            // Take appropriate action if previous
            // job missed deadline.
              missedDeadlineHandler();
            // Wake up the periodic tasks server to execute
            // the job slices
              WakeupPeriodicTaskServer();
            // If time allows, run sporadic task server.
            // If time allows, run aperiodic task server.
            // Get waiting time for next timer interrupt.
              waitingTime = getNextInterrupt() - currentTime;
            // Sleep until the next timer interrupt.
              wait(waitingTime);
        }
      }
}
```

Example Program

```
import cpsc663;

class PeriodicHelloThread extends PeriodicThread {

    // constructor()
    public PeriodicHelloThread(long period, long execTime) {
        super(period, execTime);
    }

    public void run() {
            System.out.println("Hello Periodic World");
            System.out.flush();

            while (waitForNextInvocation()) {
                    System.out.println("HELLO AGAIN!");
                    System.out.flush();
                }
    } /* run() */

    public static void main(String[] args) {

        PeriodicHelloThread rtThread =
            new PeriodicHelloThread(1000L, 10L);
        // periodic task with period 1000ms and
        // worst execution time 10ms.

        (CyclicExecutive) defaultScheduler =
            CyclicExecutive.getScheduler();

        if (defaultScheduler.add(rtThread)) {
            rtThread.start();
        else {
            System.out.println(“Admission Denied!”);
        }
    }
}
```

Implementation Note

Your scheduler will be controlling the execution of Java threads. For this, you will probably have to make substantial use of synchronization primitives to block/unblock threads. In addition, you will have to play with priorities of Java threads (don’t go overboard with this,) in order to keep control with the cyclic executive and the periodic/aperiodic/sporadic task server threads. For a good example of how to play with priorities and schedulers, see the Jimy/Java URL below.

Evaluation / Test

You will be provided a test application. (Check the Web site!) The purpose of this evaluation is to check whether your scheduler works, and to visualize whether/how it guarantees deadlines for already accepted tasks. Also, this test will visualize the behavior of sporadic and aperiodic tasks.

What to Hand In

You can turn in this homework in two levels; with and without support for EDF-scheduled sporadic tasks.

Basic Level: At this level, you realize the Cyclic Executive with support for aperiodic tasks and for slack stealing. This means that you implement (a) the admission control and the scheduling for periodic tasks (b) the scheduling of aperiodic tasks (no admission control needed here) with the support for slack stealing.

Advanced Level: You add support for sporadic tasks. This means that you add an EDF-based server for sporadic tasks. Also, the method `TriggerInvocation()` invokes an admission control test to make sure that the new invocation can be feasibly scheduled. If no feasible scheduling is possible, the `TriggerInvocation()` method returns False to the application.

You have to develop the program in Java. Don't use JNI (Java Native Interface). Therefore you can use either Solaris or Windows NT.

Design Document

You have to hand in design document along with the source code.

Hard Copy of Java Source Code

Hand in a hard copy of all the code you **created**.

Soft Copy of Java Source Code

You have to turn in Java source code and Makefile.

Reference URLs

http://www.rtj.org

http://www-cad.eecs.berkeley.edu/~jimy/java/index.html