Queuing Lab

In this lab, you will implement an online scheduler for a system that supports periodic real-time hard real-time tasks, sporadic hard real-time jobs, and aperiodic jobs.

Your scheduler will consist of admission control for these three classes of tasks, an EDF-based scheduling core for periodic and sporadic jobs, and several different kinds of servers for aperiodic jobs.

You will develop and evaluate your scheduler in the context of a simple event-driven simulator (discrete event simulator) that is driven with a trace of job/task arrivals and other events.

For this lab, admission control, preemption, and scheduling costs are (unrealistically) modeled as being zero.

Getting the Code

The queuing lab is designed to be run under Linux. To fetch it:

```
cd ~/real-time/code
svn update
```

You will see a new subdirectory “queuelab”.

You will want to look at and understand the following files right away:

```
scheduler.h
scheduler.cc   You will write your scheduler in these two

job.h
job.cc
jobqueue.h
jobqueue.cc    The abstraction for a queue of jobs

event.h
event.cc
eventqueue.h
eventqueue.cc  The event queue abstraction

make_arrivals.pl
               Generate aperiodic jobs with Poisson
               arrivals and exponential or power-law
               sizes
```

You'll want to compile and use queuesim on Linux. It should also work on cygwin under Windows, but there are no guarantees there. Also, queuesim makes use of gnuplot to show graphical output at times.

To compile for the first time, execute "touch .dependencies". Next, execute "make depend". Finally, execute "make all". This will build a single executable "queuesim".

Here is how to run queuesim:

```
./queuesim schedspec aperiodic_util eventfile [single]
```

A schedspec is a string explains what scheduling policy to use. Out of the box, queuesim only proveides "fifo_all". This policy runs all jobs in first-come-first-served order, ignoring all distinctions between them. The aperiodic utilization is the fraction of the time to reserve for running aperiodic jobs. For example, it is the size of a total bandwidth server. The eventfile (e1.txt is provided) is a file listing the startup events to feed the simulator. If the "single" argument is given, the simulator will single-step, printing a great deal of detail on every event.

At the end of the run, the simulator will print summary statistics. You can ask for these and other statistics at any time by creating the appropriate event, either at run-time or in the event file. You can also create an event that will cause the history of queue depths in the system to be shown.

Event-driven Simulation

Queuesim is an event-driven simulator. What this means is that instead of simulating the passage of time directly, it instead jumps from event to event. For example, suppose that you want to start a job running. Instead of simulating the entire interval before it finishes, you simply post an event that indicates that it is finished at the end of the interval. The simulator will immediately jump to that point in time if there are no other events in the interval.

Event-driven simulators are very powerful tools that are widely used in science and engineering. Queuesim is implemented in the usual manner for event-driven simulators. There is a priority queue, called the event queue, which stores the events in time order. The simulator main loop simply repeatedly pulls the earliest event from the queue and passes it to a handler until there are no more events in the queue. The handler for an event may insert one or more new events into the event queue. A handler can also update or delete events in the queue.

Events in Queuesim

Events in queuesim come from the event file, and from handler functions in your scheduler that are executed in response to events.

In event files, lines that are blank or whose first character is a '#' are ignored.

Here are events that can occur in a topology file. The first three are self-explanatory:

```
arrival_time PERIODIC_TASK_ARRIVAL period slice numiters
arrival_time SPORADIC_JOB_ARRIVAL size absolute_deadline
arrival_time APERIODIC_JOB_ARRIVAL size
```

The remainder are for generating output:

```
arrival_time PRINT_STATS
arrival_time PRINT_JOB_QUEUES
arrival_time PRINT_EVENT_QUEUE
arrival_time PRINT_ALL
arrival_time DISPLAY_QUEUE_DEPTHS
```

The last event pops up a gnuplot window with a graph showing the history of the depths of your EDF and aperiodic queues over time. The simulation will stall until you hit enter.

In addition to the events that can appear in the file, your scheduler may add these three kinds of events to the queue:

```
JOB_DONE
JOB_BLOCKED
TIMER             Timer interrupt
```

Your scheduler must implement the following corresponding methods:

```
AcceptanceTestOutput PeriodicTaskArrival(Job *job, SimulationContext *c);
AcceptanceTestOutput SporadicJobArrival(Job *job, SimulationContext *c);
AcceptanceTestOutput AperiodicJobArrival(Job *job, SimulationContext *c);

void JobDone(Job *job, SimulationContext *c);
void JobBlocked(Job *jon, SimulationContext *c);
void TimerInterrupt(SimulationContext *c);
```

The output functions are handled for you.

The simulation context gives you access to the event queue (for inserting/modifying/deleting events) and the EDF and aperiodic job queues (which you must maintain).

What to do

We want you to implement the following:

- EDF-based scheduling core
- Admission control for periodic tasks
- Admission control for sporadic jobs
- Total bandwidth server for aperiodic jobs
- Different scheduling policies within the total bandwidth server

We recommend that you approach the project in that order. The EDF core is relatively straightforward, so it'll give you the opportunity to climatize yourself to the simulator.

Admission control for periodic and sporadic tasks, which we suggest you think of together, is where the first real challenge of the lab is. Notice that a periodic task in this system does not repeat forever. It arrives and repeats for a fixed number of times.

Recall that the total bandwidth server runs as an EDF job. However, it also needs to keep track of aperiodic job completions internally.

We want you to implement the following policies for the aperiodic jobs scheduled by the total bandwidth server:

- FCFS or First Come, First Served. This is the default non-preemptive policy described in your book. A job runs to completion. This policy, combined with the EDF framework for the RT jobs, is given on the command-line as "edf_fcfs".
- Random. This is a non-preemptive policy. When a job arrives, it is placed in the queue at a random position. This is "edf_random".
- SJF or Shortest Job First. This is a non-preemptive policy as well. However, when a job arrives, it is placed into the queue according to its size. This is "edf_sjf".
- SRPT or Shortest Remaining Processing Time. This is a preemptive policy. Here, the queue is ordered according to the remaining processing time. Preemption (among the aperiodic jobs – preemption to run an RT job will occur with any policy) occurs only when a new aperiodic job arrives that has a smaller remaining processing time than the currently executing job. This is "edf_srpt".
- Round-robin. This is a preemptive policy. The processor is shared equally among all the jobs in the queue. A timer is used to switch consecutively from job to job. Processor sharing (PS) is achieved as the timer interrupt interval goes to zero. This is "edf_ps".
- Lottery scheduling. This is a preemptive policy. The processor is shared equally among all the jobs in the queue. A timer is used to switch to a random job. This is "edf_lottery".