3BB4 Assignment 4 Documentation
RRScheduler – Java Implementation

Seung Woo Cho	1205098
Peter Hoang	1208271
Tim Li	1205961
Mitchell Spector	1208006

General Description of Code

This package of java files compose a program that adheres to Round Robin Scheduler from the following description from the assignment 4 pdf posted on Avenue to Learn, in particular, as follows:

> "When a process is created it is put on a ready queue stating that the process is ready for execution. During its life-time a process alternates its CPU usage time with other processes (also waiting for their CPU utilization time) until it stops its existence and hence is destroyed.
>
> The process of selecting a process waiting for execution is called scheduling. The scheduler selects among the processes that are waiting in the ready queue, and allocates the CPU (for some time) to one of them. This pattern continues while there are processes to be executed.
>
> There exists several scheduling mechanisms and different criteria of how to measure their efficiency, each one of them has its pros and cons. Among them, the round-robin (RR) scheduling algorithm proven useful in time-sharing systems.
>
> The behavior of a RR scheduler can be roughly described as follows. First a time unit (called quantum) of CPU time is defined. Then, a process waiting for execution is selected from the ready queue in a first-in first-out (FIFO) manner, and it is allowed to use the CPU for a time slice in accordingly to quantum. Now, two things may happen: either the process has a CPU usage time less than quantum and hence finishes its execution during its allocated CPU time (and it is later on destroyed); or it has a CPU usage time greater than quantum and therefore its CPU usage is pre-empted by the scheduler and is moved back the ready queue (indicating that it still have some pending tasks to perform)."

In this case, the program simulates the RR scheduler and CPU interactions by generating a set of processes with random executions times and using a quantum number to determine if a process is cycled back into the queue. All processes are run through the CPU and destroyed once complete. Once all processes are complete, the program ends.
This program will take in two inputs: the number of processes to be generated and the quantum number for the CPU. Output will be in the form of console prompts that first show the processes created with their respective execution times. The console will then show processes being extracted from the queue (in standard FIFO queue), simulate the runtimes (also showing if it will be returned to the queue or be destroyed by the GrimReaper), and indicate when the operation on the process is done.

*For lines detail explanation about each line of code please check the comments

Explanation of Classes/Connection of LTSA Model

-RRScheduler.java
According to LTSA:

```
RR_SCHEDULER = (GENERATOR || AQUEUE || DISPATCHER || CPU || GRIMREAPER).
```

This is done according to that model, as the java file itself has not state logic components (besides the user prompts for testing numbers). **This file merely creates an instance of each of** Generator, ReadyQueue (corresponding to AQUEUE), Dispatcher, CPU, and GrimReaper as described by the FSP

-CPU.java
According to FSP:

```
CPU(N=QUANTUM) = (start->CPU[N]), //similar to COUNTDOWN code in chapter two
CPU[i:0..N] = (when(i>0) working->CPU[i-1]|done_task -> death ->CPU[QUANTUM] |
                  when(i==0) notdone_task->CPU[QUANTUM]).
```

The execution time is simulated by putting the Thread to sleep, so the sleep time must be pre-determined. It is provided, by the processes identities. If it is determined that quantum will be exceeded, quantum time of execution is simulated by sleep, the process is returned to queue with the remainder of the execution time, and CPU is set back to ready state. Otherwise, if execution time is below quantum, the entire execution time is simulated by sleep, the process is sent to GrimReaper, and the CPU is set back to ready state.

-Dispatcher.java
According to LTSA:

```
DISPATCHER = (start-> dequeue ->DISPATCHER).
```

The Dispatcher has been changed slightly from the model. In the java implementation, it is now responsible for re-enqueueing processes into queue as opposed to GrimReaper doing so. It also dequeues one process at a time, constantly going back to the ready state. It continues dequeuing until there are no processes in the queue left. Note that there may be another change in the LTSA model, as Dispatcher ends once there is an empty queue as opposed to going back to ready state.

-Generator.java

According to LTSA:

```
                    GENERATOR = GENERATOR[2], //start state is 4 PROCESSES LEFT TO
GENERATE
                    GENERATOR[i:T] =
                    if(i>0) then
                                        ( generate->enqueue->GENERATOR[i-1]) // REDUCE THE
AMOUNT GENERATOR CAN                                        GENERATE
                    else (enqueueOverflow ->GENERATOR[i]).
```

The java implementation simply generates all the processes to be enqueued in a Process class array, and theoretically it is concurrently throw into the ReadyQueue class to be enqueued. Please note that overflow is eliminated by specifying how many processes are to be tested at once before the program starts, and a counter check is placed upon enqueue/dequeue commands in the entire program.

-GrimReaper.java

According to LTSA:

```
GRIMREAPER = (done_task-> death -> GRIMREAPER| notdone_task ->
enqueue -> GRIMREAPER).
```

The GrimReaper has been changed slightly from the model. The Dispatcher is now responsible for re-enqueueing. This change was necessary, as it is difficult to synchronize the Dispatcher Process queue loop with grimreaper unloading from CPU. It is also redundant to bring the GrimReaper object into Dispatcher as they lead to each other through CPU. Otherwise, the front half remains the same, as it still puts a stop to processes that the CPU deems has finished.

-Process.java

This class that extends Thread is simply an instance of a process. It contains the properties of ID and execution time. It also contains the methods necessary to run and stop.

-ReadyQueue.java

Note that the Queue has a limit of five items inside the Queue

```
const N=5 /*5 processes max enqueued*/
range P = 0..N

range M = 0..1
set   S = {[M]} //changes for two spots
```

```
QUEUE           //empty state, only send permitted
  = (enqueue[x:M]->QUEUE[x]),
QUEUE[h:M]       //one message queued to QUEUE
  = (enqueue[x:M]->QUEUE[x][h]
    |dequeue[h]->QUEUE
    ),
QUEUE[t:S][h:M]  //two messages sent to QUEUE
   = (enqueue[x:M]->QUEUE[x][t][h] // three states indicate overflow this leads to error detection
     |dequeue[h]->QUEUE[t]
     ).
||AQUEUE = QUEUE/{enqueue/enqueue[M],dequeue/dequeue[M]}.
```

The java implementation performs the same functionality, as a standard FIFO queue that accepts Process instances. Throughout the code, there are also checks for overflow/underflow, the code check occurs at the prompt.

Compile instructions

-Place all java files within same package in Java Eclipse, similar to the textbook example:
-This includes ReadyQueue, Generator, Dispatcher, CPU, GrimReaper, Process, RRScheduler.

-Run RRScheduler class, as it contains the main method

-Before continuing, please note that runtimes are randomized between 1 to 8 seconds for optimal testing. If these numbers must absolutely be changed, please change the class level integer variables High and Low located in Generator.java.

-Please also note that the queue process limit is currently set to 5. Checks are done when user input of number of processes generated is entered. To change this number, change the queueLimit integer constant in the RRScheduler.

-From here, test cases can be run by entering number of processes to be generated and quantum to their respective console prompts, and observing the output prompts showing queue and CPU operations that result.