Software Design III 3BB4

Assignment IV

Jadees Anton	Connor Hallett	Spencer Lee	Nicolas Lelievre
1213386	*1158083*	*1224941*	*1203446*

March 31, 2015

Round Robin Scheduler

This document elaborates the round-robin scheduler Java implementation and how it functions in order to schedule a series of processes for eventual execution.

Overview

The following points demonstrate how the applets user interface is to be used in order for the round-robin scheduler to function properly.

First, in order to generate a single process, press the *run* button found in the *Generator* panel which will in turn display an additional 10 degrees to the cyclical counter. Note that multiple processes may be generated, during or in between executions, always increasing the cyclical counter by 10 degree increments.

Once the desired amount of processes have been inserted into the queue, or before any processes are added, pressing *run* on the *CPU* panel will commence execution of the processes found within the queue. Similarly, the *CPU* thread panel will rotate 360 degrees for each cycle of execution. This works by removing the process from the queue when loaded into `CPU`, executing the process for a single cycle, then rescheduling the process to the tail of the queue if there is any remaining execution time. Otherwise, the completed process is not re-inserted into the queue. A series of messages found within the applet also help with understanding the current state of the program as well as the execution order for all processes found within the queue.

Below is the according LTS model.

```
1  const MaxIF=6
2  range IF=0..MaxIF-1
3  const MaxCPU=5
4  range CPUT = 0..MaxCPU
5
6  GENERATOR=GENERATOR[0],
7  GENERATOR[num:IF] = (enqueue[num] -> GENERATOR[(num+1)%MaxIF]).
8
9  QUEUE = (enqueue[num:IF] -> QUEUE[num]),
10 QUEUE[n0:IF] = (
11 enqueue[num:IF] -> QUEUE[n0][num]
12 | cpu.load[n0] -> QUEUE),
13 QUEUE[n0:IF][n1:IF] = (
14 enqueue[num:IF] -> QUEUE[n0][n1][num]
15 | cpu.load[n0] -> QUEUE[n1]),
16 QUEUE[n0:IF][n1:IF][n2:IF] = (
17 enqueue[num:IF] -> QUEUE[n0][n1][n2][num]
18 | cpu.load[n0] -> QUEUE[n1][n2]),
19 QUEUE[n0:IF][n1:IF][n2:IF][n3:IF] = (
20 enqueue[num:IF] -> QUEUE[n0][n1][n2][n3][num]
21 | cpu.load[n0] -> QUEUE[n1][n2][n3]),
22 QUEUE[n0:IF][n1:IF][n2:IF][n3:IF][n4:IF] = (
23 enqueue[num:IF] -> QUEUE[n0][n1][n2][n3][n4][num]
24 | cpu.load[n0] -> QUEUE[n1][n2][n3][n4]),
25 QUEUE[n0:IF][n1:IF][n2:IF][n3:IF][n4:IF][n5:IF] = (
26 cpu.load[n0] -> QUEUE[n1][n2][n3][n4][n5]).
27
28 CPU = CPU[0],
29 CPU[t:CPUT] = (
30 when (t==0) load[c:IF] -> quantums[i:1..MaxCPU] -> CPU[i]
31 | when (t!=0) tick -> CPU[t-1]).
32
33 ||RR_SCHEDULER=(cpu:CPU || GENERATOR || QUEUE).
```

Java Classes & Implementation

The following will elaborate on the roles of each implemented java classes as well as how they execute the desired tasks.

ReadyQueue.java

The `ReadyQueue.java` class utilises a *ConcurrentLinkedList* Java library in order to implement a standard queue in a first-in-first-out (FIFO) manner.

`ReadyQueue` therefore holds the processes within a queue, ready for impending execution.

Design Assumption I

New processes (ie. processes with larger IDs) are initially stored sequentially at the tail of the queue, preserving a first-in-first-out order of primary execution. Concurrent executions may change later orders of queue IDs as completion times may vary.

Generator.java

The `Generator.java` class implements the *random* Java library in order to randomise the length of execution time for every process between a pre-defined minimum and maximum value of 1 to 5 cycles respectively.

Whereas the `ReadyQueue` stores the processes, `Generator` generates the process to be stored, assigning unique IDs for every process created. This allows for processes to be followed during execution inside and outside of the queue.

This may be observed on the user interface by a 10 degree increment for each generated process. However, the process is not executed until `CPU` is called.

Design Assumption II

Generated process IDs are assigned numerically in increasing order, starting at 0, and are unbounded allowing for a theoretical infinite amount of generated processes.

Dispatcher.java

Possibly the simplest class within this round-robin scheduler, `Dispatcher.java` loads the process at the head of the queue into the `CPU` for execution.

CPU.java

As previously mentioned, `CPU.java` calls the above `Dispatcher.java` to load a single process for execution. This execution is scheduled for one cycle.

If the process has not completed it's full execution after one run cycle, it is rescheduled to the tail of the queue for eventual completion. Otherwise, if there is no remaining run time, the process is removed by `GrimReaper`.

This process may be observed in the user interface under the *CPU* thread panel. Pressing *run* executes `CPU` indefinitely until there are either no remaining processes in the queue for execution, or the *pause* button is pressed, temporarily halting execution.

A processes execution is graphically represented using a cyclical counter, similar to the *Generator* panel. A single cycle of execution is displayed using a completed circle. `CPU` may be paused at any time during execution.

Design Assumption III

The process is not executed by `CPU` *inside the queue. Instead, it is removed from the queue by* `Dispatcher` *and is either eliminated or rescheduled in the queue by the* `CPU` *depending on the execution's outcome.*

GrimReaper.java

The `GrimReaper.java` class removes the process if, after execution by the `CPU`, there is no remaining run time for any given process after a cycle of execution (it is not restored into the queue).

On the other hand, if the process requires more execution time before completion, `GrimReaper` reschedules the process by placing it at the tail of the queue.

Design Assumption IV

Completed processes are immediately eliminated without the need to be reintroduced into the queue.

Process.java

The `Process.java` class implements the processes that are scheduled and run by decrementing they're remaining execution time by 1 after the execution of a full `CPU` cycle.

Note that each process has a total runtime (random number between 1 and 5 cycles) as well as a unique process ID for easy tracking during execution.

These processes are stored in `ReadyQueue` until they reach a point for execution.

RRScheduler.java

Arguably the largest of the classes, `RRScheduler.java` incorporates the *Applet* Java library in order to create a basic user interface for the round robin scheduler. This allows it to display both the *Generator* and *CPU* animated thread panels as well as a series of messages (labels) that help to understand the execution process employed by the program.

Messages displayed include:

- What process is currently loaded into the CPU
- The number of cycles that have been executed by the CPU for a unique process
- What process has completed it's execution
- What process has yet to complete it's execution and must be rescheduled

Design Assumption V

`RRScheduler` *does not perform any tasks other than calling aforementioned classes and methods at the appropriate times.*

Desirable Properties

This implementation of the round robin scheduler ensures that the system's below desirable properties are all correct and present.

Many Generated Processes

`Generator` allows the creation (or "generation") of many processes. Simply press the *run* button found on the *Generator* thread panel multiple times before, during or after the execution of `CPU`.

The "generation" of a new process is indicated by an increased 10 degrees in the cyclical counter, also found in the *Generator* thread panel.

Deadlock Free

We have ensured that this system is completely deadlock free since all processes in the queue will eventually be serviced without fail.

There is no precise notion of priority in execution between processes since they are executed by the `CPU` in the order defied by the queue.

No Incomplete Process Destruction

No process with any remaining execution time is ever destroyed.

The `GrimReaper` class will always reschedule a process back into the queue if there is any execution time left for any uniquely identified process.

No Overuse of CPU Time

No process stored within the queue uses the `CPU` time more than is allowed by the system.

In every execution case, `CPU` will execute the process at the head of the queue for exactly one cycle before `GrimReaper` either eliminates or reschedules said process.

No Incorrect Process Execution

No process with null remaining time is ever scheduled for execution.

When loaded to the `CPU`, the process is removed from the queue and is rescheduled into the queue if and only if, after 1 execution cycle, the process still has more than 0 execution cycles remaining. Otherwise, the process is not returned to the queue, ensuring that only incomplete processes remain for.

Testing

Proper functioning of the system was ensured using the animated *Generate* and *CPU* thread panels as well as the message labels found within the user interface.