CSC3002F 2022
Operating Systems Part 2
Assignment: Process Scheduling

Introduction

This assignment aims to increase your understanding of CPU process scheduling and to give you some experience with testing and implementing scheduling using a simulator. Computer simulation is used to model the behaviour of a complex system. The intention is that the model has predictive properties i.e. it tells us something about the real world system. In assignment, we model the behaviour of an operating system kernel from the perspective of process scheduling. We're not interested in memory allocation, file handing, and device drivers and so on.

You are given a software **framework** for simulating process scheduling and **two working kernels** for scheduling: a **First Come First Served** (FCFS) kernel and a **Round Robin** (RR) kernel. Using this framework, you have **three tasks** to complete, as follows (in increasing order of difficulty).

(a) For an FCFS scheduler, to identify four workloads that optimise, respectively, turnaround time, waiting time, throughput and CPU utilisation. [10 marks]
(b) To profile **RR scheduling** using a range of time quanta. [15 marks]
(c) To build a working pre-emptive **Shortest Job First** (SJF) scheduler that runs correctly within the framework [25 marks].

Preparation

The framework may be downloaded from the assignment page.

The '`.zip`' file contains a '`src`' directory/folder, a directory called `processgenerator`, a directory/folder called '`tests`', a file called '`makefile`', a framework overview document called '`Framework.pdf`', and a document on creating simulation processes called '`ProcessGenerator.pdf`'.

The `src` directory contains a main simulation program called '**`Simulate`**', the working kernels, named '**`FCFSKernel`**' and '**`RRKernel`**', and the framework classes themselves in a folder called '**`simulator`**'.

Your first step should be to compile the materials. Use the command '**`make all`**':

```
$ make all
javac -g -d bin -cp bin:../lib/junit/junit-
4.12.jar:../lib/junit/hamcrest-core-1.3.jar src/simulator/*.java
javac -g -d bin -cp bin:../lib/junit/junit-
4.12.jar:../lib/junit/hamcrest-core-1.3.jar
src/processgenerator/*.java
javac -g -d bin -cp bin:../lib/junit/junit-
4.12.jar:../lib/junit/hamcrest-core-1.3.jar src/*.java
$
```

Continued

The overview document contains an example of running a simulation using **`Simulate`**, the **`RRKernel`**, and a test workload from the **`tests`** directory.

For question three you will also want API documentation. To obtain this, use the command '**`make doc`**'. The results will be placed in a new sub directory called '`doc`'.

Assignment Specifications

1. FCFS Metrics [10 Marks, automarked]

The performance of the **FCFS scheduling** strategy can vary greatly, according to the load on the system, the type of job running on the system (i.e., the mix of CPU or IO bound processes) and the metrics used to measure performance. This first part of the assignment aims to give you some experience with the simulation framework (in preparation for the tasks ahead) as well as to give you some insight into different scheduler metrics.

Your task is to construct four **simulator workloads** that show optimal performance for four different metrics of scheduler performance: **turnaround** time, **waiting** time, **throughput** and **CPU utilisation**.

You are given four processes: `procA.prg`, `procB.prg`, `procC.prg` and `procD.prg`. Your job is to create four **configuration files** describing a simulator workload for the **FCFSKernel scheduler** of **all of these processes** as follows.

(a) `config1.cfg` produces the **minimum *average* turnaround time** for the four processes.

(b) `config2.cfg` produces the **minimum average waiting time** for the four processes. Note that *waiting time* is the total time spent in the ready queue waiting to execute on a CPU (this is not what the Framework profiler calls "WAITING").

(c) `config3.cfg` produces the **maximum average process throughput per unit time** for the four processes.

(d) `config4.cfg` produces the **maximum CPU utilisation** for the four processes.

Use the **`Simulate`** program along with the **`FCFSKernel`**. See the framework overview document for specifications on how to create a configuration file and for a description of the profiling output produced from the simulator.

2. Profiling FCFS and RR [15 marks]

The size of the **time quantum** has a large effect on RR scheduling: it must neither be too large, nor too small. We want to ensure that:

a. the time quantum is **large enough** relative to the time for a context switch, to ensure good **CPU utilisation**; and
b. the time quantum is **small enough** to ensure **good turnaround time** (i.e., most processes finish their next CPU burst within a single time quantum).

If the time quantum is too large, RR scheduling degenerates to an FCFS policy.

Your task is to use the **simulator** and the **program generator** to create various workloads for the **`RRKernel`** and **FCFSKernel** schedulers, compare the two and to identify the **optimal time quantum** for **`RRKernel`**. You need to generate **graphs** of **CPU utilisation** and **turnaround time** with 10 different workloads of lamba from 1-9 (you will use the program generator to do this). You should assume a **context switch time of 5** time units, and **system call time of 1**.

Continued

Explain what you see in these graphs and **draw conclusions** – where you see best CPU utilization, where is the best turnaround time and when RR becomes FCFS (note that just the graphs is not sufficient). Submit your work as a '**FCFCvsRR.pdf**' document. It should be about 1-2 pages in length, assuming that you have two graphs: one of **CPU% vs lambda** and one of **turnaround time vs lambda**, each graph containing a single plot for FCFS and several plots for RR (one for each quantum size).

Use the **Simulate** program along with the **FCFSKernel** and **RRKernel.** See the framework overview document for specifications on how to create a configuration file and for a description of the profiling output produced from the simulator. Refer to the **ProcessGenerator** document.

3. Shortest Job First (SJF) Simulator [25 marks, automarked]

Using the framework supplied, construct a preemptive SJF simulator as a **kernel class** called '**SJFKernel**'. You should use the implementations of the FCFS and RR kernels as guides on how to do this (**FCFSKernel.java** and **RRKernel.java**, respectively). You must construct a kernel class that can be used with the Simulate program to run an SJF simulation on a given workload (config file and process files).

The framework document provides an overview of the interfaces that a kernel must implement. Further framework details are available through the API documentation. To view this, you will need to generate it – see below.

Say that we have the following configuration files and process files:

Configuration file ('config.cfg'):

```
# Two programs, one with I/O.
DEVICE 1 disk
PROGRAM 0 0 programA.prg
PROGRAM 0 0 programB.prg
```

Process file ('programA.prg'):

```
CPU 1000
IO 1000 1
CPU 1000
```

Process file ('programB.prg'):

```
CPU 3000
```

When this workload is simulated using the **Simulate** program and an SJF kernel, the following behaviour will be exhibited:

```
*** Simulator ***
Configuration file name? config.cfg
Kernel name? SJFKernel
Enter kernel parameters (if any) as a comma-separated list:
Cost of system call? 1
Cost of context switch: 3
Trace level (0-31)? 0
Instantiating kernel with supplied parameters...
Building configuration...
Running simulation...
```

Continued

```
Done
System time: 5021
Kernel time: 21
User time: 5000
Idle time: 0
Context switches: 5
CPU utilization: 99.58
Write execution profile to CSV? Enter a file name or press return:
profile.csv
```

(User input in **bold**.)

The contents of `profile.csv` will be:

```
PID, STATE, MODE, START TIME, END TIME, PROGRAM
001, READY, N/A, 0000000001, 0000000005, programA.prg
001, RUNNING, USER, 0000000005, 0000001005, programA.prg
001, RUNNING, SUPERVISOR, 0000001005, 0000001006, programA.prg
001, WAITING, N/A, 0000001006, 0000002007, programA.prg
001, READY, N/A, 0000002007, 0000002010, programA.prg
001, RUNNING, USER, 0000002010, 0000003010, programA.prg
001, RUNNING, SUPERVISOR, 0000003010, 0000003011, programA.prg
001, TERMINATED, N/A, 0000003011, -, programA.prg
002, READY, N/A, 0000000005, 0000001009, programB.prg
002, RUNNING, USER, 0000001009, 0000002006, programB.prg
002, READY, N/A, 0000002006, 0000003014, programB.prg
002, RUNNING, USER, 0000003014, 0000005017, programB.prg
002, RUNNING, SUPERVISOR, 0000005017, 0000005018, programB.prg
002, TERMINATED, N/A, 0000005018, -, programB.prg
```

The **trace level** specifies the degree of detail shown of the simulated execution. In this example, no trace information is requested.

Tracing capability is built into the framework as an aid to kernel development. (See the simulator.TRACE class for details.) By way of example, if a trace level of 1 is specified for the same workload, a trace of context switches will be output.

```
*** Simulator ***
Configuration file name? config.cfg
Kernel name? SJFKernel
Enter kernel parameters (if any) as a comma-separated list:
Cost of system call? 1
Cost of context switch: 3
Trace level (0-31)? 1
Instantiating kernel with supplied parameters...
Building configuration...
Running simulation...
Time: 0000000001 Kernel: Context Switch {Idle}, process(pid=1,
state=READY, name="programA.prg")).
```

Continued

```
Time: 0000001006 Kernel: Context Switch process(pid=1,
state=WAITING, name="programA.prg"), process(pid=2, state=READY,
name="programB.prg")).
Time: 0000002007 Kernel: Context Switch process(pid=2, state=READY,
name="programB.prg"), process(pid=1, state=READY,
name="programA.prg")).
Time: 0000003011 Kernel: Context Switch process(pid=1,
state=TERMINATED, name="programA.prg"), process(pid=2, state=READY,
name="programB.prg")).
Time: 0000005018 Kernel: Context Switch process(pid=2,
state=TERMINATED, name="programB.prg"), {Idle}).
Done
System time: 5021
Kernel time: 21
User time: 5000
Idle time: 0
Context switches: 5
CPU utilization: 99.58
Write execution profile to CSV? Enter a file name or press return:
```

(User input in **bold**.)

The framework '`.zip`' file has a directory called '`tests`' with sub directories containing test workloads and associated outputs - trace and execution profiles. Traces have all been generated with a trace value of 31. Traces may be found in '`.log`' files and process execution profiles in '`.csv`' files. You can use these to check that your simulator functions correctly.

HINT: Besides the guidance provided by the FCFS and RR kernels, you should examine the features of the **`PCB`** and **`Instruction`** classes.

NOTE: You **must not** alter the contents of the framework **`simulator`** package in any way.

Marking and submission

You will submit your materials for assessment via the automatic marker within a single '.zip' file bearing your **student number**.

The file should contain the following:

- A folder called **`FCFSconfig`** that contains the four configuration files: `config1.cfg`, `config2.cfg`, `config3.cfg` and `config4.cfg` that form your solution to part 1 of the assignment.
- A '**`FCFCvsRR.pdf`**' document containing your solution to Part 2 of the assignment.
- A folder called **`SJF`** that contains your solution to Part 3 of the assignment.

The automatic marker will assess your solutions for Tasks 1 and 3.