University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	Networks and Operating Systems Essentials 2
Coursework Number	2
Deadline	**Time:** 16:30 **Date:** Fri 23 Nov 2018
% Contribution to final course mark	10%
Solo or Group ✓	Solo ☐ Group ✓ (2 members)
Anticipated Hours	10 per student (20 overall)
Submission Instructions	Submit two files (Report.docx, Source.zip) via Moodle
Please Note: This Coursework cannot be Re-Done	

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

  (i)   in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.
  (ii)   work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an "Own Work" form via
https://webapps.dcs.gla.ac.uk/ETHICS for all coursework
UNLESS submitted via Moodle

Networks & Operating Systems Essentials 2 (NOSE 2)
Assessed Exercise 2: Scheduling

The aim of this exercise is to have students use the knowledge they've acquired so far in the second part of the course, in the context of implementing a set of scheduling algorithms. You will be using and extending a simple discrete event simulator written in Python, by creating classes that simulate the scheduling and dispatching of processes performed by four scheduling algorithms: First-Come-First-Served (FCFS), Shortest-Job-First (SJF), Round-Robin (RR) and Shortest-Remaining-Time-First (SRTF). This assessed exercise will also act as a gentle introduction to such concepts as queuing theory, discrete event simulation, and object-oriented software engineering.

Queueing Theory Primer

Queuing Theory is a discipline of mathematics (specifically, of probability theory) that studies waiting lines, also known as queues. Queues (sometimes also referred to as *queuing nodes*) are usually described using a notation of the form X/Y/Z, where X denotes the distribution between arrivals of successive jobs to the system, Y denotes the distribution of job sizes, and Z indicates the number of servers for the queue. Given a queue, a model is then constructed to allow for prediction of statistics on the queue length and waiting times.

One of the simplest and most frequently used models is the M/M/1 queue. In this case, the inter-arrival time follows a Poisson process with rate parameter λ (i.e., on average λ jobs will arrive per time unit), the job sizes follow an exponential distribution with rate parameter μ (i.e., on average each job will require $1/\mu$ time units), and there is only one server for the queue.

The system is said to be *stable* if $\lambda < \mu$; that is, if jobs arrive less frequently than they are completed. In this case, one can compute the *stationary distribution* – that is, the limiting distribution over an infinite amount of time – and therefore deduce various performance measures. Let $\rho = \lambda/\mu$. Then, for example, the average number of jobs in the queue is given by $\lambda/(1-\lambda)$, the average waiting time is given by $\rho/(\mu-\lambda)$ and the average response time is given by $1/(\mu-\lambda)$.

The simple discrete event simulator you are provided with already features an implementation of a M/M/1 queue. That is, new processes are added to the system following a Poisson process, process service times follow an exponential distribution, and there is supposed to be only one CPU (i.e., only one process should be active/running at any time).

Discrete Event Simulation Primer

A discrete event simulation (DES) simulates the operation of a system via a series of discrete events ordered in time. That is, given a set of events ordered by time of occurrence from earliest to latest, the internal clock of the simulator does not take on a continuum of values but rather jumps from one event to the next. The simulated system is considered to be in a constant state in between any pair of successive events,

while every event triggers a possible change in the system's state. The DES, in essence, implements the following algorithm:

```
Queue event_queue # Assume a queue of events, ordered by time

system_time = 0
while event_queue is not empty:
    current_event = event_queue.remove_first_event()
    if current_event.time > system_time:
        system_time = current_event.time
    service(current_event)
```

where `service(current_event)` may end up adding new events to the queue.

In the simple discrete event simulator provided for this assessed exercise, we have three types of events:
- PROC_ ARRIVES: A new process arrives to the system.
- PROC_CPU_REQ: An existing process requests access to the CPU.
- PROC_CPU_DONE: A process has used up its service time and thus terminates.

Similarly, simulated processes can be in one of the following states:
- NEW: A process that will arrive to the system at some point in the future.
- READY: A process that is waiting for the CPU; note that this can be a new process that just arrived, or an older process that was never scheduled or pre-empted.
- RUNNING: A process currently executing on the CPU.
- WAITING: A process waiting for some IO or other blocking operation to finish (only used in the "plus" version of the assignment, to be discussed shortly).
- TERMINATED: A process that has used up its service time and is thus terminated.

Originally the simulator creates a list of processes to be simulated. Each process has the following attributes:
- Process ID: A number uniquely identifying the process in the system. As all processes are added to a table (aka the *process table*), their ID is simply the table index for the cell at which their info is stored, starting from 0.
- Arrival time: The time of arrival of the process.
- Service time: The total amount of CPU time required by the process.
- State: The state in which the process currently is, as discussed above.

Each process keeps track of its execution time and further offers a set of utility functions, comprising:
- A function that returns its remaining time.
- A function that returns its departure time, if the process has terminated.
- A function that computes and returns its total waiting time.
- A function that computes and returns its turnaround time.
- A function that executes the process for up to a specified amount of time.

All processes start off in the NEW state. Along the same lines, the simulator populates the event queue with PROC_ARRIVES events for all processes to be simulated. The simulator's service function then looks like so (in abstract terms):

```
service(current_event):
    for process in list_of_processes:
        if process.arrival_time <= system_time:
            process.state = READY
    process_to_run = scheduler_func(current_event)
    if process_to_run != currently_executing_process:
        system_time += context_switch_time
        currently_executing_process = process_to_run
    new_event = dispatcher_func(process_to_run)
    if new_event.type != PROC_CPU_DONE:
        event_queue.insert(new_event)
    system_time = new_event.time
```

The service function makes use of two more functions, printed in italics above: `scheduler_func(event)` and `dispatcher_func(process)`. The former takes the current event into consideration and selects the next process to be given access to the CPU, based on the scheduling policy/algorithm. The latter takes as argument the process selected by the scheduler and executes it on the CPU; this translates to transitioning the process to the RUNNING state, allowing it to run for a specific amount of time (dependent on the scheduling algorithm used), then transitioning it to either the READY or TERMINATED state (depending on whether it used up its service time), generating and returning an appropriate event (PROC_CPU_REQ in the former case, PROC_CPU_DONE in the latter case). If the event is a PROC_CPU_REQ one, it is also added to the events queue, as it will require further processing in the future. Finally, the simulator's internal (simulated) clock is updated to the time of the returned event. With this in hand, you should be able to implement most simple process scheduling algorithms, as outlined below.

Object-Oriented Software Engineering Primer

The design of the simple discrete event simulator provided for this assessed exercise, follows an object-oriented approach. The information and methods required for events and processes are implemented in matching classes (Event, EventTypes, Process, ProcessStates). The basic simulator logic is also implemented in a class of its own – namely, SchedulerDES.

Then, four new scheduler-specific classes are provided; namely, FCFS, SJF, RR and SRTF, one for each of the scheduling algorithms to be implemented. The classes are defined as *subclasses* of SchedulerDES. Subclassing, or *inheritance*, is one of the mainstays of object-oriented programming. Each subclass S (sometimes also referred to as a child or heir class) of a class C has access to all members and methods of its parent class (aka ancestor class or superclass). Python supports *multiple inheritance* (i.e., having multiple parent classes) but this is beyond the scope of this assessed exercise/course.

While other object-oriented programming languages offer mechanisms for controlling the access to a parent class's members and methods, Python doesn't really have such

a feature. However, by convention subclasses shouldn't use and/or access members/methods whose name starts with one or more underscores.

When a subclass S has a method M with the exact same name and input parameters as those of a superclass C, we say that S's M *overrides* (in essence, provides a new default implementation of) C's M. In other words, if one creates an object of S and calls M on it, it will be S's implementation code that will be executed instead of M's. This is used in the provided source code to allow you to implement the various scheduling algorithms without having to touch the base discrete event simulator implementation. You will notice that the source code of the latter includes skeleton definitions for the scheduler and dispatcher functions discussed above, and that these same functions are defined in the subclasses as well. You only need to edit the latter, as it's these implementations that will be used by the main function of the simulator. Remember that you still have full access to all methods/members of SchedulerDES, but you should only really use those without one or more leading underscores.

Scheduling Algorithms

As part of this assessed exercise, you are requested to implement the following scheduling algorithms:

- FCFS/FIFO (non-pre-emptive): Processes should be executed in the order in which they arrived at the system. Conceptually, when a process arrives at the system, it is added to a queue. The scheduling algorithm will always pick the first process in the queue and will execute it to completion. It will then proceed with the next process in the queue, and so on.
- SJF (non-pre-emptive): Processes are prioritised based on their service time. Conceptually, on arrival, processes are added to a priority queue, which is sorted in ascending order of service time. The scheduling algorithm will then always pick the first process in the queue and will execute it to completion. It will then proceed with the next process in the queue, and so on.
- RR (pre-emptive): On arrival, processes are added to a queue. Conceptually, the algorithm will pick the first process in the queue, execute it for a specified amount of time (also known as a time-slice or quantum), and if the process needs more time it will then be added to the end of the queue.
- SRTF (pre-emptive): This is a pre-emptive version of the SJF algorithm above. Conceptually, whenever a change occurs in the system (i.e., a new process arrives, a running process terminates, etc.), the scheduler is called to select the process among those in the READY state with the minimum remaining execution time. This process will then be pre-empted when a new change occurs that results in some other process having a lower remaining execution time than the currently executing one.

Pseudo-Random Number Generation

The main simulator code makes use of a pseudo-random number generator (PRNG) to generate the process arrival and service times, following the M/M/1 queue equations. The "pseudo" prefix in PRNG denotes that these generators don't really produce completely random data. Instead, they compute a new "random" output

based on a deterministic computation on state kept internally by the PRNG implementation. Consequently, if one could control the original internal state, they would be able to force the PRNG into producing repeatable sequences of "random" outputs. Most implementations of PRNGs support this functionality through *seeding*; in other words, they provide a method that allows the user to define an initial *seed* (usually taking the form of an integer), based on which the PRNG initialised its internal state in a deterministic manner. For example, execute the following code in your favourite Python environment:

```python
for repetition in range(3):
    print("Repetition {}:".format(repetition))
    for i in range(10):
        random.random()
```

You should get three sequences of ten random floats, each different to the next. Then, execute the following code:

```python
for repetition in range(3):
    print("Repetition {}:".format(repetition))
    random.seed(1)
    for i in range(10):
        random.random()
```

You should now get three sequences of ten random floats, but all three sequences are identical to each other. This is due to the fact that we have reset the seed to the same value (1) before the generation of each of these sequences.

This is used in the provided implementation to allow for repeatable experiments. The PRNG defaults to a random initial seed. This seed is printed to the screen when the simulator starts. The program also features a command-line argument that allows you to set the seed to a user-defined value, as well as arguments to enable verbose logging of the internal state of the simulator. You can use this whenever a run of the simulator produces counter-intuitive results, to allow you to dig into the problem and identify the cause of the "discrepancy". For your consideration, here is a set of "interesting" seeds: **1797410758**, **2688744162**, **3399474557**.

AE2 Plus

You will notice that the assessed exercise comes with two versions of the associated source code: ae2.zip and ae2+.zip. The latter contains an advanced version of the simulator core. Whereas the "standard" (ae2.zip) version simulates processes with a single CPU burst, the "plus" (ae2+.zip) version simulates processes with a succession of multiple CPU and IO bursts. In the latter case, it is assumed that each process starts its execution with a CPU burst, followed by one or more IO-CPU burst pairs, each with their own duration (generated using similar logic as above). Students who think that the standard version is easy and would like to be challenged, are welcome to work on the plus version. In this case, you will still only need to change the code in schedulers.py, with the same restrictions as in the standard case. However, please make sure to read the source code documentation carefully, as you will need to take some more aspects of the simulation into account. Also, please be advised that "plus" version submissions will *not* have a higher weight/percentage than standard version submissions. Last, keep in mind that results for the same random seed are not

comparable across the two versions, due to the way in which random numbers are drawn to initialise the simulator.

Miscellanea

Start by reading and trying to understand the provided code. Please feel free to ask us if you have any questions that cannot be answered by a simple online search. You shouldn't need to read up on the concepts outlined earlier (queuing theory, discrete event simulation, object-oriented programming), but feel free to do so if you deem it necessary to understand the provided code (or have an interest in the field).

You should only need to change the code in schedulers.py. The list of imports in said file includes all imports that you should need for the assessed exercise. Please ask us if you think more imports are necessary, as it might be an indication of a misunderstanding.

Please make sure that your code is well formatted and documented, and that an appropriate function/variable naming scheme has been used.

The main function (main.py) provides a number of command-line options that allow you to change various parameters of the system. Executing the program with the '-h' or '--help' command-line arguments will print a help message explaining all supported options. A set of sane defaults is provided in the source code, but feel free to play around with other values. You can do so by either executing the simulator on the command line and providing different arguments, or by changing the defaults in the source code (main.py), or by providing your own input parameters as input to the `parser.parse_args(…)` function; as an example of the latter, to execute the program with the command-line arguments `-S 851898649`, do the following:

```
args = parser.parse_args(['-S', '851898649'])
```

to print the help message, do the following:

```
args = parser.parse_args(['-h'])
```

to increase the verbosity level to full debug output, do the following:

```
args = parser.parse_args(['-v', '-v'])
```

and for combinations of the above, simply add more arguments to the list, like so:

```
args = parser.parse_args(['-S', '851898649', '-v', '-v'])
```

What to submit

For this assessed exercise, you are asked to work in teams of up to two (2) students. Submit a zip file with your Python source code files, as well as a short report, via the course's Moodle page (look for the "Assessed Exercise 2" assignment link). Your report should include a heading stating your full names and matriculation numbers, a discussion of your findings for the "interesting" seed values outlined earlier (i.e., what was the relative performance of the four algorithms, whether it deviated from your expectations, and what you believe is the cause of the deviation based on the internal state of the simulator), as well as a discussion of any hardships you faced and how you

addressed them. Feel free to also include any feedback you may have on the assessed exercise. Only one submission should be done per team.

How Exercise 2 will be marked

Following timely submission on Moodle, the exercise will be given a numerical mark, between 0 (no submission) and 25 (perfect in every way). These numerical marks will then be converted to a band (C2 etc.)

The marking scheme is given below:
- 4 marks for each of FCFS and SJF (total: 8): 2 marks for correct selection of the next process to execute (scheduler function), 2 marks for correct state keeping and process execution (dispatcher function).
- 5 marks for each of RR and SRTF (total: 10): 2 marks for correct selection of the next process to execute (scheduler function), 3 marks for correct state keeping and process execution (dispatcher function).
- 7 marks for the report: 2 marks for the discussion of the results of each of the "interesting" seed values (total: 6), 1 mark for discussion of hardships and solutions.