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.

Object-Oriented Software Engineering Primer

You should already be familiar with the basic concepts of object-oriented programming in Java from the JP2 course. Specifically, you should already know:

- What a class is and what member variables and functions/methods are.
- What an object or instance of a class is.
- How to create (instantiate) an object of a class.
- How to refer to member variables/methods using dot notation (e.g., ClassX.func()).
- What the meaning of 'this' is in Java.
- What inheritance is and how you can override class methods through creating a subclass/child class.
- What encapsulation is – the act of bundling together data/variables and functions/methods that operate on said data/variables in a single unit: the class.

Python also offers an object-oriented API. In Python a class is defined using the keyword 'class', like so:

```
class Animal:
    …
```

The equivalent of Java's '`this`' in Python is '`self`'. As Python is not a pure object-oriented programming language, all class methods need to have '`self`' as their first argument. All variables and methods defined in this class (i.e., indented one level) are class variables and class methods. Also, the constructor of a class does not have the same name as the class but is rather called '`__init__`' in Python. In Python it is also customary to create instance variables in a class's `__init__` method.

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 (a.k.a. ancestor class or superclass). In Python subclassing (inheritance) is defined by including the name of the parent class in parentheses after the name of the new child class. 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. 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.

Putting it all together, have a look at the example code below:

```
class Bird:
    def __init__(self, species):
        self.species = species

    def what_am_i(self):
        print("I am a " + self.species)

    def can_fly(self):
        print("I can't fly")

    def can_swim(self):
        print("I can't swim")

class Parrot(Bird):
    def __init__(self):
        self.species = "Parrot"

    def can_fly(self):
        print("I can fly")

class Penguin(Bird):
    def __init__(self):
        self.species = "Penguin"

    def can_swim(self):
        print("I can swim")

birds = [Eagle(),Penguin()]
for bird in birds:
    bird.what_am_i()
    bird.can_fly()
    bird.can_swim()
```

If you save it in a file and execute it, output should be:

```
I am a Parrot
I can fly
I can't swim
I am a Penguin
I can't fly
I can swim
```

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.
- 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.

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.

Method overriding 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 (as briefly discussed in last week's tutorial). 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:

```
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:

```
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**. These are interesting in the sense that the respective workloads make some algorithms win over others. What you should do in these cases, is execute the program with debug logging on, examine the sequence of events/scheduling decisions and address the points laid out in the spec below.

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'])
```

Keep in mind, though, that executing your code from the command line (as in AE1) is probably both faster and easier.

Please also note that:

- You shouldn't need to add any new queue or other state to the simulator. All the state required by the simulator is already there in SchedulerDES.
- If you need to walk through the list of processes, use `self.processes`. Given a process ID (say, `id`), you can query its state via `self.processes[id].X`, where `X` is either an attribute of the Process class or one of its methods.
- To peak ahead at the time of the next event after the current time point, use `self.next_event_time()`.
- If you want direct access to the events' queue (although you shouldn't need to), use `self.events_queue`.
- You can also get the currently executing process (`self.process_on_cpu`) and the current simulator time (`self.time`).
- The scheduler function is called with a single event as an argument, but it also has full access to the process table and other internal data structures of the simulator (e.g., events queue, etc.). In other words, the scheduler doesn't need to decide based solely on the process that created the current event; some scheduling algorithms do make use of this information while others completely ignore it.

The simulator is written in such a way to facilitate abstracting out the scheduler and dispatcher functions. As such, the code you need to write is rather minimal. As a yardstick, the sample solution comprises less than 50 lines of code in schedulers.py including whitespace (56 including imports) – that is, less than 7-8 new lines of code per simulator... Your mileage may vary, of course, but these figures should give you a rough idea of how much coding effort should go into this assessed exercise.

Sample results (random seed, timings, etc.) are provided below. In the meanwhile, if you want to verify that your code works correctly, you should turn debug logging on (-vv), fix the random seed to a value (e.g., -S 0) and walk your way through the trace. If the execution trace you get by hand (i.e., process A runs first for X time units, then process B for Y time units, etc.) does not agree with the simulator's output, then there's something wrong; revisit your hand trace and code, rerun with the same random seed, and see if the two agree; rinse and repeat as appropriate. **Try running the simulator with different context switch times as well, as the default value in the simulator is set to 0.0.**

What to submit

For this assessed exercise, you are asked to work in teams of up to two (2) students (**i.e. 2 students is the MAXIMUM – groups of more than 2 will not be considered for marking**). Submit your amended `schedulers.py` file, as well as a short report in pdf format, via the course's Moodle page (look for the "Assessed Exercise 2" assignment link). Your report should include a heading stating **your full names, matriculation numbers and lab groups**, and 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). 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 100 (perfect in every way). These numerical marks will then be converted to a band (C2 etc.)

The marking scheme is given below:

- 15 marks for each of FCFS and SJF (total: 30): 7 marks for correct selection of the next process to execute (scheduler function), 8 marks for correct state keeping and process execution (dispatcher function).
- 20 marks for each of RR and SRTF (total: 40): 7 marks for correct selection of the next process to execute (scheduler function), 13 marks for correct state keeping and process execution (dispatcher function).
- 30 marks for the report: 10 marks for the discussion of the results of each of the "interesting" seed values.

Sample Outputs

```
Using seed: 1523376833
Processes to be executed:
    [#0]: State: ProcessStates.NEW, Arrival: 0.8965518035211827,
Service: 0.4772854990859405, Remaining: 0.4772854990859405
    [#1]: State: ProcessStates.NEW, Arrival: 1.0476559314160219,
Service: 3.177651950380589, Remaining: 3.177651950380589
    [#2]: State: ProcessStates.NEW, Arrival: 1.0699502089969615,
Service: 0.6431423507594756, Remaining: 0.6431423507594756
    [#3]: State: ProcessStates.NEW, Arrival: 1.133575330296856,
Service: 0.21095976155023272, Remaining: 0.21095976155023272
    [#4]: State: ProcessStates.NEW, Arrival: 1.5419034712499409,
Service: 3.519113233731405, Remaining: 3.519113233731405
    [#5]: State: ProcessStates.NEW, Arrival: 2.268572897370114,
Service: 6.1209365033748995, Remaining: 6.1209365033748995
    [#6]: State: ProcessStates.NEW, Arrival: 2.622213160578937,
Service: 0.6057431641679517, Remaining: 0.6057431641679517
    [#7]: State: ProcessStates.NEW, Arrival: 2.8181529993918173,
Service: 0.603425465615895, Remaining: 0.603425465615895
    [#8]: State: ProcessStates.NEW, Arrival: 3.0484632504147853,
Service: 1.2052874280418702, Remaining: 1.2052874280418702
    [#9]: State: ProcessStates.NEW, Arrival: 3.213418227642897,
Service: 2.892823843272308, Remaining: 2.892823843272308
-----
FCFS [#Processes: 10, Avg arrivals per time unit: 3.0, Avg CPU burst
time: 2, Context switch time: 0.0]:
    Avg. turnaround time: 9.055465010768293
    Avg. waiting time: 7.109828090770234
-----
SJF [#Processes: 10, Avg arrivals per time unit: 3.0, Avg CPU burst
time: 2, Context switch time: 0.0]:
    Avg. turnaround time: 5.667330888524098
    Avg. waiting time: 3.721693968526041
-----
RR [#Processes: 10, Avg arrivals per time unit: 3.0, Avg CPU burst
time: 2, Context switch time: 0.0, Quantum: 0.5]:
    Avg. turnaround time: 8.6057126790477
    Avg. waiting time: 6.660075759049643
-----
SRTF [#Processes: 10, Avg arrivals per time unit: 3.0, Avg CPU burst
time: 2, Context switch time: 0.0]:
    Avg. turnaround time: 5.071064581670328
    Avg. waiting time: 3.1254276616722705
```

```
Using seed: 3672961927
Processes to be executed:
    [#0]: State: ProcessStates.NEW, Arrival: 0.9967930942538261,
Service: 1.7805632589480833, Remaining: 1.7805632589480833
    [#1]: State: ProcessStates.NEW, Arrival: 1.026222231111286,
Service: 1.7022393172998072, Remaining: 1.7022393172998072
    [#2]: State: ProcessStates.NEW, Arrival: 1.8916853352121672,
Service: 2.3302464408938315, Remaining: 2.3302464408938315
    [#3]: State: ProcessStates.NEW, Arrival: 2.0051880828926594,
Service: 5.220529617260626, Remaining: 5.220529617260626
    [#4]: State: ProcessStates.NEW, Arrival: 3.0218802368513096,
Service: 2.226205967193615, Remaining: 2.226205967193615
    [#5]: State: ProcessStates.NEW, Arrival: 3.255766032372033,
Service: 4.047087113201036, Remaining: 4.047087113201036
    [#6]: State: ProcessStates.NEW, Arrival: 3.3862860082130255,
Service: 3.0229686771601187, Remaining: 3.0229686771601187
    [#7]: State: ProcessStates.NEW, Arrival: 3.723246678844583,
Service: 0.3318700244392102, Remaining: 0.3318700244392102
    [#8]: State: ProcessStates.NEW, Arrival: 3.8174183504968853,
Service: 3.021590099896377, Remaining: 3.021590099896377
    [#9]: State: ProcessStates.NEW, Arrival: 4.056820014748351,
Service: 2.448209295878759, Remaining: 2.448209295878759
-----
FCFS [#Processes: 10, Avg arrivals per time unit: 3.0, Avg CPU burst
time: 2, Context switch time: 0.0]:
    Avg. turnaround time: 12.626963581749276
    Avg. waiting time: 10.01381260053213
-----
SJF [#Processes: 10, Avg arrivals per time unit: 3.0, Avg CPU burst
time: 2, Context switch time: 0.0]:
    Avg. turnaround time: 9.484330868807557
    Avg. waiting time: 6.871179887590411
-----
RR [#Processes: 10, Avg arrivals per time unit: 3.0, Avg CPU burst
time: 2, Context switch time: 0.0, Quantum: 0.5]:
    Avg. turnaround time: 16.262966521167005
    Avg. waiting time: 13.649815539949861
-----
SRTF [#Processes: 10, Avg arrivals per time unit: 3.0, Avg CPU burst
time: 2, Context switch time: 0.0]:
    Avg. turnaround time: 9.436993491606682
    Avg. waiting time: 6.823842510389535
```