Assignment #2: Multi-process Scheduling

Due: March 6, 2023 at 23:59

1. Assignment Description

This is the second of a series of three assignments that build upon each other. In this assignment, you will extend the simulated OS to support **running concurrent processes.**

This assignment is significantly longer than the first one, so please plan your time wisely,

and don't hesitate to ask questions **on Ed** if you get stuck.

1.1 Starter files description:

You have three options:

- **[Recommended]** Use your solution to Assignment 1 as starter code for this assignment. If your solution passes the Assignment 1 testcases, it is solid enough to use as a basis for the second assignment.
- Use the official solution to Assignment 1 provided by the OS team as starter code. The solution will be released on **February 16**, so you will have to wait for another week to start coding. You can use this time to go over the assignment instructions carefully and sketch your solution.
- Ask a friend, or a team for their solution to Assignment 1. **In this case, you have to give credit to your peers in the README. Failure to do so will be considered plagiarism.**

1.2 Your tasks:

Your tasks for this assignment are as follows:

- Implement the scheduling infrastructure.
- Extend the existing OS Shell syntax to create concurrent processes.
- Implement different scheduling policies for these concurrent processes.

On a high level, in this assignment you will run concurrent processes via the `exec` command, and you will explore different scheduling strategies and concurrency control. `Exec` can take up to three files as arguments. The files are scripts which will run as concurrent processes. For each `exec` argument (i.e., each script), you will need to load the full script code into your shell memory. For this assignment, you can assume that the scripts will be short enough to fully fit into the shell memory – this will change in Assignment 3.

You will also need to implement a few data structures to keep track of the code execution for the scripts, as the scheduler switches the processes in and out of the "running" state. After this infrastructure is set up, you will implement the following scheduling policies: FCFS, SJF, and RR.

More details on the *behavior* of your scheduler follow in the rest of this section.

Even though we will make some recommendations, you have **full freedom for the implementation**. In particular:

- Unless we explicitly mention how to handle a corner case in the assignment, you are **free to handle corner cases as you wish**, without getting penalized by the TAs.
- You are free to craft your own error messages (please keep it polite).
- Just make sure that your output is the same as the expected output we provide in the test cases in Section 2.
- Formatting issues such as tabs instead of spaces, new lines, etc. **will not be penalized.**

Let's start coding! ☺

1.2.1. Implement the scheduling infrastructure

We start by building the basic scheduling infrastructure. For this intermediate step, you will modify the `run` command to use the scheduler and run `SCRIPT` as a process. Note that, even if this step is completed successfully, you will see no difference in output compared to the `run` command in Assignment 1.

However, this step is crucial, as it sets up the scaffolding for the `exec` command in the following section. As a reminder from Assignment 1, the `run` API is:

`run SCRIPT` *Executes the commands in the file SCRIPT*

`run` Assumes that a file exists with the provided file name, *in the current directory*. It opens that text file and then sends each line one at a time to the interpreter. The interpreter treats each line of text as a command. At the end of the script, the file is closed, and the command line prompt is displayed once more. While the script executes, the command line prompt is not displayed. If an error occurs while executing the script due a command syntax error, then the error is displayed, and the script continues executing.

You will need to do the following to run the `SCRIPT` as a process:

1. **Code loading.** Instead of loading and executing each line of the `SCRIPT` one by one, you will load *the entire source code* of the `SCRIPT` file into the OS Shell memory. It is up to you to decide how to encode each line in the Shell memory.
   - *Hint: If you solved Section 1.2.1 in Assignment 1 correctly, it might come in handy for loading the source code into the Shell memory.*
2. **PCB.** Create a data-structure to hold the `SCRIPT` PCB. PCB could be a `struct`. In the PCB, at a minimum, you need to keep track of:
   - The process PID. Make sure each process has a unique PID.
   - The spot in the Shell memory where you loaded the `SCRIPT` instructions. For instance, if you loaded the instructions contiguously in the Shell memory (highly recommended), you can keep track of the start position and length of the script.
   - The current instruction to execute (i.e., serving the role of a program counter).
3. **Ready Queue**. Create a data structure for the ready queue. The ready queue contains the PCBs of all the processes currently executing (in this case, there will be a single process). One way to implement the ready queue is to add a *next* pointer in the PCB (which points to the next PCB in the ready queue), and a pointer that tracks the head of the ready queue.
4. **Scheduler logic.** If steps 1—3 were done correctly, we are now in good shape to execute `SCRIPT` through the scheduler.

  - The PCB for `SCRIPT` is added at the tail of the ready queue. Note that since the `run` command only executes one script at a time, `SCRIPT` is the only process in the ready queue (i.e., it is both the tail and the head of the queue). This will change in Section 1.2.2 for the `exec` command.
  - The scheduler runs the process at the head of the ready queue, by sending the process' current instruction to the interpreter.
  - The scheduler switches processes in and out of the ready queue, according to the scheduling policy. For now, the scheduling policy is FCFS, as seen in class.
  - When a process is done executing, it is cleaned up (see step 5 below) and the next process in the ready queue starts executing.
5. **Clean-up.** Finally, after the `SCRIPT` terminates, you need to remove the `SCRIPT` source code from the Shell memory.

Assumptions

- The shell memory is large enough to hold three scripts and still have some extra space. In our reference solution, the size of the Shell memory is 1000 lines; each script will have at most 100 lines of source code. If you implemented your shell from scratch, please use the same limits.
- You can also assume that each command (i.e., line) in the scripts will not be larger than 100 characters.

If everything is correct so far, your `run` command should have the same behavior as in Assignment 1. You can use the existing unit tests from Assignment 1 to make sure your code works correctly.

1.2.2. Extend the OS Shell with the exec command

We are now ready to add concurrent process execution in our shell. In this section, we will extend the OS Shell interface with the `exec` command:

`exec prog1 prog2 prog3 POLICY` *Executes up to 3 concurrent programs, according to a given scheduling policy*

- `exec` takes **up to four arguments**. The two calls below are also valid calls of `exec`:
  - `exec prog1 POLICY`
  - `exec prog1 prog2 POLICY`
- `POLICY` is always the last parameter of `exec.`
- `POLICY` can take the following three values: `FCFS`, `SJF`, `RR,` or `AGING`. If other arguments are given, the shell outputs an error message, and `exec` terminates, returning the command prompt to the user.

Exec behavior for single-process. The behavior of `exec prog1 POLICY` is the same as the behavior of `run prog1,` regardless of the policy value. *Use this comparison as a sanity check.*

Exec behavior for multi-process. Exec runs multiple processes concurrently as follows:

- The entire source code of each process is loaded into the Shell memory.
- PCBs are created for each process.
- PCBs are added to the ready queue, according to the scheduling policy. For now, implement only the `FCFS` policy.
- When processes finish executing, they are removed from the ready queue and their code is cleaned up from the shell memory.

Assumptions

- For simplicity, we are simulating a single core CPU.
- We will not be testing recursive `exec` calls. That is, you can assume that a program does not contain other `exec` calls.
- Each `exec` argument is the name of a **different** script filename. If two `exec` arguments are identical, the shell displays an error (of your choice) and `exec` terminates, returning the command prompt to the user (or keeps running the remaining instructions, if in batch mode).
- If there is a code loading error (e.g., running out of space in the shell memory), then no programs run. The shell displays an error, the command prompt is returned, and the user will have to input the `exec` command again.

Example execution

prog1 code	prog2 code	prog3 code

`echo helloP1`
`set x 10`
`echo $x`
`echo byeP1`
	`echo helloP2`
	`set y 20`
	`echo $y`
	`print y`
	`echo byeP2`
		`echo helloP3`
		`set z 30`
		`echo byeP3`

Execution:

```
$ exec prog1 prog2 prog3 FCFS
helloP1
10
byeP1
helloP2
20
20
byeP2
helloP3
byeP3
$                                   //exec ends and returns command prompt to user
```

1.2.3. Adding Scheduling Policies

Extend the scheduler to support the Shortest Job First (SJF) and Round Robin (RR) policies, as seen in class.

- For **SJF**, use the **number of lines of code** in each program to estimate the job length.
- For **RR**, schedulers typically use a timer to determine when the turn of a process ended. In this assignment, we will use a fixed number of instructions as a time slice. **Each process gets to run 2 instructions before getting switched out**.

Example execution (prog1, prog2, prog3 code is the same as in Section 1.2.2)

Example SJF	**Example RR**
`$ exec prog1 prog2 prog3 SJF`	`$ exec prog1 prog2 prog3 RR`
`helloP3`	`helloP1`
`byeP3`	`helloP2`
`helloP1`	`helloP3`
`10`	`10`
`byeP1`	`byeP1`
`helloP2`	`20`
`20`	`20`
`20`	`byeP3`
`byeP2`	`byeP2`
`$`	`$`

1.2.4. SJF with job Aging

One of the important issues with SJF is that short jobs continuously preempt long jobs, leading to starvation. Aging is a common technique that addresses this issue. In this final exercise, you will implement a simple aging mechanism to promote longer running jobs to the head of the ready queue.

The aging mechanism works as follows:

- Instead of sorting jobs by estimated job length, we will sort them by a "job length score". You can keep track of the job length score in the PCB.
- In the beginning of the exec command, the "job length score" of each job is equal to their job length (i.e., the number of lines of code in the script) like in Section 1.2.3.
- The scheduler will re-assess the ready queue every time slice. For this exercise, we will use a time slice of **1 instruction.**
  - After a given time-slice, the scheduler "ages" all the jobs that are in the ready queue, apart from the current head of the queue.
  - The aging process decreases a job's "job length score" by 1. The job length score cannot be lower than 0.
  - If after the aging procedure there is a job in the queue with a score that is lower than the current running job, the following happens:
    - The current running job is preempted
    - The job with the new lowest job length score is placed at the head of the running queue. In case of a tie, the process closer to the head of the running queue has priority.
    - The scheduler runs the new process in the head of the ready queue.
  - If after the aging procedure the current head of the ready queue is still the job with the lowest "job length score", then the current job continues to run for the next time slice.

prog1 code	**prog2 code**	**prog3 code**

`echo helloP1`
`set x 10`
`echo $x`
`echo byeP1`
	`echo helloP2`
	`set y 20`
	`echo $y`
	`print y`
	`echo byeP2`
		`echo helloP3`
		`set z 30`
		`echo byeP3`

Execution of SJF with aging and a time slice of 1 instruction; the state of the ready queue shown in comments:

`$ exec prog1 prog2 prog3 AGING`	
`helloP3`	// (P3, 3), (P1, 4), (P2, 5) → aging (P3, 3), (P1, 3), (P2, 4) → no promotion
//Nothing printed for set z 30	// (P3, 3), (P1, 3), (P2, 4) →aging (P3, 3), (P1, 2), (P2, 3) →promote P1
`helloP1`	// (P1, 2), (P2, 3), (P3, 3) →aging (P1, 2), (P2, 2), (P3, 2) →no promotion
//Nothing printed for set x 10	// (P1, 2), (P2, 2), (P3, 2) →aging (P1, 2), (P2, 1), (P3, 1) →promote P2
`helloP2`	// (P2, 1), (P3, 1), (P1, 2) →aging (P2, 1), (P3, 0), (P1, 1) →promote P3
`byeP3`	// (P3, 0), (P1, 1), (P2, 1) →aging (P3, 0), (P1, 0), (P2, 0), →promote P1
`10`	// (P1, 0), (P2, 0), no more aging possible
`byeP1`	// (P1, 0), (P2, 0), no more aging possible
//Nothing printed for set y 20	// (P2, 0), no more aging possible
`20`	// (P2, 0), no more aging possible
`20`	// (P2, 0), no more aging possible
`byeP2`	// (P2, 0), no more aging possible
`$`	

1.2.5. Multithreaded scheduler

So far, the scheduler is single-threaded. In this final exercise you will transform the scheduler from single-thread to multi-threaded.

Hint: Part 1 and Part 2 of this exercise are preparation for adding the multi-threading. You do not need to implement multiple threads until Part 3.

Part 1. Execution in the background. Before implementing the multi-threaded scheduler, we need to add one more option, i.e., the `#` option, to the `exec` command:

`exec prog1 [prog2 prog3] POLICY #`

- The semantics of exec are the same as described in 1.2.2.
- *# is an optional parameter that indicates execution in the background (similar to the & command in the Linux terminal). If exec is run with #, exec will be run in the background and the control in the shell returns immediately to the batch script; the following instruction will be executed normally.*
- This is achieved by converting the rest of the Shell code into a program and running it, as you are running programs in the `exec` command.
- All the programs, including the Shell code program, are run according to `POLICY.`

Example execution

Commands (prog1, prog2, prog3 same as in Section 1.2.2; RR policy is the same as in Section 1.2.3)	

`exec prog1 RR`
`echo progDONE`
`echo progDONE2`
`echo progDONE3`
	`exec prog1 RR #`
	`echo progDONE`
	`echo progDONE2`
	`echo progDONE3`

Execution	

`helloP1`
`10`
`byeP1`
`progDONE`
`progDONE2`
`progDONE3`
	`prog1DONE`
	`progDONE2` // batch script has priority
	`helloP1` // Only 1 line printout, as the set command does not have an output
	`progDONE3`
	`10`
	`byeP1`

Assumptions

- Regardless of the scheduling policy, you can assume that the main shell program will be placed at the head of the running queue.
- You can assume that only one `exec` command will be run with the `#` option in each testcase.
- You can assume that the `#` option will only be used in batch mode.
- You can assume that if an `exec` command with the `#` option is launched with a `POLICY` P, then all following `exec` commands will use the same `POLICY` P. We will not be testing different policies in the same testcase.

Part 2. RR policy with extended time slice.

Add a new `RR30` policy, where each process gets to run for **30 instructions** before it is switched out. The rest of the implementation is identical to the RR policy described in Section 1.2.3. Note that the multi-threaded scheduler (Part 3 below) will only be tested with `RR` and `RR30`.

Part 3. Multithreaded scheduler.

To enable multi-threaded scheduling, we will add one more option to the `exec` command:

`exec prog1 [prog2 prog3] POLICY [#] MT`

If `MT` appears at the end of the `exec` command, the multi-threaded scheduler is enabled. Once the multi-threaded scheduler has been enabled by one of the `exec` commands, it remains enabled for the entire duration of the testcase (i.e., the threads are terminated only when `quit` is called).

Your multi-threaded scheduler will consist of a pool of **two worker threads, created by using the pthreads library**. The two worker threads will handle the requests (i.e., programs that are ready to be run in the running queue). Note that, up to this point, your scheduler is single-threaded. Therefore, the programs are executed sequentially, according to the POLICY, and the instructions interleaving is deterministic. With a multi-threaded scheduler that uses two worker threads, two programs can run concurrently, leaving room for non-determinism in the output.

If the `#` option is used in `exec`, the remainder of the main program is treated as a separate program and placed at the top of the ready queue. Then, the execution resumes normally, according to the policy.

If `quit` is called and the ready queue is not empty, the quit implementation needs to join with the scheduler threads.

For instance, for the following example:

<table>
<tr><th>pA code</th><th>pB code</th><th>pC code</th></tr>
<tr><td>echo A
echo A
echo A
echo A</td><td>echo B
echo B
echo B
echo B</td><td>echo C
echo C
echo C
echo C</td></tr>
<tr><td colspan="3">Commands (RR policy is the same as in Section 1.2.3)</td></tr>
<tr><td>//single-thread scheduler
exec pA pB pC RR</td><td colspan="2">//multi-thread scheduler
exec pA pB pC RR MT</td></tr>
<tr><td colspan="3">Execution (ready queue RQ shown in comments)</td></tr>
<tr><td>A // RQ: pA pB pC
A // RQ: pA pB pC
B // RQ: pB pC pA
B // RQ: pB pC pA
C // RQ: pC pA pB
C // RQ: pC pA pB
A // RQ: pA pB pC
A // RQ: pA pB pC
B // RQ: pB pC pA
B // RQ: pB pC pA
C // RQ: pC pA pB
C // RQ: pC pA pB</td><td colspan="2">// RQ: pA pB pC → pA and pB picked up by the 2 workers W1, W2
A // RQ: pC . pC is alone in RQ
B
B // pB at end of time slice. Is put at the end of RQ.
//RQ: pC pB → W2 picks up pC
A // pA at end of time slice. Is put at the end of RQ.
//RQ: pB pA →W1 picks up pB
C
B
C // pC at end of time slice. Is put at the end of RQ.
//RQ: pA pC →W2 picks up pA
B // pB done
//RQ: pC →W1 picks up pC
C
A
A // pA done
C // pC done
// the order above is non-deterministic; the deterministic part is
// that four As, four Bs, and four Cs are printed.</td></tr>
</table>

2. TESTCASES

We provide 20 testcases and expected outputs in the starter code repository. Please run the testcases to ensure your code runs as expected, and make sure you get similar results in the automatic tests.

IMPORTANT: The grading infrastructure uses batch mode, so make sure your program produces the expected outputs when testcases run in batch mode. You can assume that the grading infrastructure will run one test at a time in batch mode, and that there is a fresh recompilation between two testcases.

3. WHAT TO HAND IN

The assignment is **due on March 6, 2023 at 23:59, no extensions.**

Your final grade will be determined by running the code in the GitLab repository that is crawled by our grading infrastructure. We will take into account the most recent commit that happened before the deadline, on the master branch of your fork.

In addition to the code, please include a README mentioning the author name(s) and McGill ID(s), any comments the author(s) would like the TA to see, and mention whether the code uses the starter code provided by the OS team or not.

The project must compile on the DISCS server by running `make clean; make mysh`

The project must run in batch mode, i.e. `./mysh < testfile.txt`

Feel free to modify the Makefile to add more structure to your code, but make sure that the project compiles and runs using the commands above.

Note: You must submit your own work. You can speak to each other for help but copied code will be handled as to McGill regulations. Submissions are automatically checked via plagiarism detection tools.

4. HOW IT WILL BE GRADED

Your program must compile and run on the DISCS lab to be graded. If the code does not compile/run using the commands in Section 3, in our grading infrastructure you will receive **0 points** for the entire assignment. If you think your code is correct and there is an issue with the grading infrastructure, contact sebastian.rolon@mail.mcgill.ca.

Your assignment is graded out of 20 points. You were provided 20 testcases, with expected outputs. If your code matches the expected output, you will receive 1 point for each testcase. You will receive 0 points for each testcase where your output does not match the expected output. Formatting issues such as tabs instead of spaces, new lines, etc. **will not be penalized.** The TA will look at your source code only if the program runs (correctly or not). The TA looks at your code to verify that you implemented the requirement as requested. Specifically:

- **Hardcoded solutions will receive 0 points for the hardcoded testcase**, even if the output is correct.
- **Programming expectations.** Your code needs to meet the programming style posted on myCourses. **If not, your TA may remove up to 6 points, as they see fit.**
- **You must write this assignment in the C Programming language**, otherwise the assignment will receive 0 points.