Assignment #2: Multi-process Scheduling

Due: March 7, 2022 at 23:59 on myCourses

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.

You will use the **C programming language** in your implementation. The assignment is presented from a Linux point of view using the SOCS server `mimi.cs.mcgill.ca`, which you can reach remotely using `ssh` or `putty`. Please make sure your assignment runs on `mimi.cs.mcgill.ca` since these are the computers the TAs will be using for testing your code.

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 public unit tests, 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. Finally, you will add simple concurrency control, where the scripts can access and modify a shared variable in the shell memory.

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, extra command line prompts, etc. **will not be penalized.**

Let's start coding! ☺

1.2.1. Implement the scheduling infrastructure

We will 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.
    - *Hint1: 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 a bit of extra space. In our reference solution, the size of the Shell memory is 1000 lines, thus each script will have at most 300 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 do not support threading**.**
- We will not be testing recursive `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`
`helloP3`
`byeP3`
`helloP1`
`10`
`byeP1`
`helloP2`
`20`
`20`
`byeP2`
`$`
	`$ exec prog1 prog2 prog3 RR`
	`helloP1`
	`helloP2`
	`helloP3`
	`10`
	`byeP1`
	`20`
	`20`
	`byeP3`
	`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
$
```

2. TESTCASES

IMPORTANT: The TAs will use batch mode when testing your code, so make sure that your program produces the expected outputs when testcases run in batch mode. You can assume that the TAs will run one test at a time in batch mode.

We provide you with 5 testcases and expected outputs for your code. Please run the testcases to ensure your code runs as expected.

3. WHAT TO HAND IN

The assignment is **due on March 7, 2022 at 23:59, no extensions.** You need to submit the following:

- All the files in your project, including the testcases and a README, in one zipped file, named as follows: **lastname_firstname_mcgillid.zip**
- If you are working in a team, the zipped file name is **lastname1_firstname1_mcgillid1_ lastname2_firstname2_mcgillid2.zip**
  If the name is too long, only include the last names and McGill IDs.
- The README should mention 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 give credit to starter code borrowed from your peers.

- The project must compile on mimi by running `make clean; make mysh`
- The project must run in batch mode, i.e. `./mysh < testfile.txt`

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 assignment is graded out of **20 points**.

- **IMPORTANT: Your program must compile and run on mimi to be graded**. If the code does not compile/run using the commands in Section 3, you will receive **0 points** for the entire assignment.

- **Known testcases: 10 points.** You were provided 5 testcases, with expected outputs. If your code matches the expected output, you will receive 2 points 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, extra command line prompts, etc. **will not be penalized.**

- **Surprise testcases: 10 points.** The OS team also has 5 surprise testcases, similar to the ones already provided to you. The surprise testcases are graded the same as the known testcases.

- 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 5 points, as they see fit.**
    - **You must write this assignment in the C Programming language**, otherwise the assignment will receive 0 points.