Assignment #2: Multi-Process Scheduling

Due: February 28, 2025 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 can become longer than Assignment 1 if your solution duplicates a lot of code, so **plan your solution in advance** to be well-factored and don't hesitate to ask questions **on Discord** 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 **approximately February 19**, so you will have to wait to start programming. You can use this time to go over the assignment instructions carefully and sketch your solution.

To obtain a local copy of this documentation, you can get the files from our git repository, assuming you've added our remote when you completed A1:

```
$ git fetch staff
$ git checkout main
$ git merge staff/main
# This file is at A2/Assignment_2_Winter2025.{docx,pdf}
```

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.

To complete this assignment, you will need to implement several data structures to manage code execution for scripts as the scheduler transitions processes in and out of the "running" state. These data structures should be defined in their own header and implementation files. Once the infrastructure is established, you will implement the following scheduling policies: FCFS, SJF, and RR. You are encouraged to add new files to the project and modify the Makefile as needed, as your Makefile will be used for evaluation.

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.
- 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.
- Some of the test cases in the assignment have both a *_result. txt and a *_result2. txt expected result. This is because we didn't tell you how to break ties when adding processes to your queue, and depending on which way you break ties you may get different results. You only **have to** match one, not both.
- Formatting issues such as tabs instead of spaces, new lines, etc. in the output **will not be penalized.**

Let's start programming! ☺

1.2.1. Implement the scheduling infrastructure

We start by building the basic scheduling infrastructure. For this intermediate step, you will modify the `source` 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 `source` 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 `source` API is:

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

`source` assumes that a file exists with the provided path, absolute or relative to 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: We highly recommend defining a new data structure in shellmemory.c for storing program lines, separate from the variable memory. Each program line would be a string.*
   - *Hint: While you* could *store the program lines in the PCB, or have a per-process structure in shellmemory.c, that approach would not work for Assignment 3 where the programs will share their*

code lines. Therefore, we recommend using a structure shared by all processes. This will require an allocator to allocate space in the structure, but it can be very simple.

- *Hint: Alternatively, you may wish to ignore the looming future of A3 for now, and keep the program code separated from each other. That is a perfectly valid approach.*

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 a shared data structure in shellmemory.c (highly recommended), you can keep track of the start position and length of the script.
   - The current instruction to execute. If following the recommendation, this is probably an index into an array of program lines (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 then your queue structure will have a pointer that tracks the head of the ready queue.
   - Note: you will only ever need one queue at a time, so it is OK to have a single global queue. However, keeping separate queues for different scheduling policies, or for other purposes, can lead to nice solutions, so you may want a complete queue interface with create/destroy functions. It is up to you.
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 `source` 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 (specifically the highest priority process, in case your queue is not sorted by priority), 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 shell memory can hold 1000 lines (and is shared by all processes); so, no more than 1000 lines of code will be loaded at the same time. If you implemented your shell from scratch, please use the same limit.
- 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 `source` 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*

- For now, `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 (for now) always the last parameter of `exec.`
- `POLICY` can take the following four 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.
    - Recommendation: the policies are different in two ways: when they interrupt the processes, and how they choose which process to run. You will have the easiest time if you structure your code around these two differences. Have only one (a bit tricky) or two (we found this easier) loops executing program code. If using two loops, one is for policies that are non-preemptive, and the other is for preemptive policies. Use configurable variables to decide when to preempt processes, and to decide which policy's enqueue/dequeue functions should be used. (The reference solution sets the enqueue/dequeue functions using variables with function pointers. However, that is not required, and there are plenty of other reasonable ways to do this.)
    - If it would take more than a few minutes to add a new scheduling policy to your code, the structure is probably not very good, and this will cause headaches later in the assignment.

Exec behavior for single-process. The behavior of `exec prog1 POLICY` is the same as the behavior of `source prog1,` regardless of the policy value. *Note: this is not a special case – a correctly implemented exec should have this property. 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, as in 1.2.1.1.
- 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.
- Each `exec` argument is the name of a **different** script filename. If two `exec` arguments are identical, the shell has to display an error (of your choice) and `exec` must terminate, returning the command prompt to the user (or keep 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, or file does not exist), then none of the programs should run. The shell should display an error, and then display the prompt again. The user will have to input the `exec` command again.
- For now, when `exec` completes, the ready queue should be empty and any space used by the program lines should be reset, so that another `exec` command can be used independently of the first. If the "background" mode is used (see 1.2.5), this might not be true, as there may still be other processes in the queue that were added by different `exec` commands.

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
```

Assumptions

- For simplicity, we are simulating a single core CPU. Do **not** use real threads to run the different programs as this will almost certainly result in out-of-order output.
- You can assume that a program containing an exec call does not include other nested exec calls, except that the “background script” (see section 1.2.5) might contain more exec calls. There is more information about how to handle this in that part of the description.

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 policy, 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, but not the job that was executing during that time slice. (You may find it easiest to remove jobs from the queue while they are executing via a dequeue operation, and enqueue them back to the queue only after their time slice and any aging step is complete. This goes for all policies.)
  - 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 score of the job that just ran, then the job with the lower score will run next.
    - Ultimately, this is up to you, but you should keep the queue sorted by job length score when using this policy. The easiest way to do this is for your enqueue function to behave like the "insert" part of insertion sort. That way, after each aging step, when you enqueue the job that just ran back to the queue, it will be inserted in the right place. The other jobs will necessarily be sorted, because they were sorted before and all of their scores decreased by 1 (or are 0).
  - If after the aging procedure the current head of the ready queue is still the job with the lowest (or tied for lowest) "job length score", then the current job will continue 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. Background Mode

In this final exercise, you will approximate the behavior of a bash shell when the `&` modifier is placed after a command. (If you are not familiar, experiment with commands like `sleep 10 && echo “done!”` vs `sleep 10 && echo done &`.)

Part 1. 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. If your code is well-structured, this part should take only a few minutes. (Adding this to the reference solution touched only 8 lines of code.)

Part 2. Execution in the background. We will now add 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 #, control will appear to return to the batch script, and the batch script will appear to continue running **with the scheduler**; it will be swapped out with the other programs given to `exec`.
- This is achieved by converting the rest of the Shell input into a program and running it, as you are running programs in the `exec` command. That is, read the rest of the user input as if it were another program `prog0`, and then schedule it as such. Call this the “batch script process.” The batch script process begins with the first line **after** the exec that used #.
- All the programs, including the batch script process, are run according to `POLICY`.
- Regardless of the scheduling policy, the batch script process must run **first**. This gives the batch script process a chance to invoke additional `exec` commands before other programs run (and possibly finish, which would complicate your code line allocation). Once the batch script has been given a time slice and preempted, it will be scheduled normally after that. In other words, this condition only affects the first time that the batch script process is scheduled.
- While none of the test programs invoke the `exec` command, the batch script process might. **This is the only way that `exec` commands will be invoked while your scheduler is in-use.** This is sort of a weird “exec recursion,” and requires special care. When this happens, you should enqueue the newly `exec`’d programs onto the **same queue** that is being used for the `exec` command that used #. For example, see test case T_RR30_2.txt. Notice that P_quit runs before any of the programs in the first exec command are given a second timeslice.

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

Assumptions

- 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. It is difficult to test in interactive mode, so we recommend doing your own testing with batch mode as well.
- 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. More generally, we will not be testing different policies in the same testcase.

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 the same results as 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 February 28, 2025 at 23:59.** As is the late policy, you have up to 4 late days to use across the entire term, but you must fill out the late submission form on MyCourses **before the deadline** if you wish to use late days.

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, adjusted by any late days requested, on the main 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 mimi 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. (We will use **your** Makefile.)

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 our grading server 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 mo.danesh@mcgill.ca.

We will check your code for style, as in A1. The penalty for any style mismatches will be 2 points. (Total, not per mismatch.)

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. in the output **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.
- **You must write this assignment in the C Programming language**, otherwise the assignment will receive 0 points.