WPI

CS 3013 - Operating Systems

Project 4 (65 points)
Assigned: Friday, September 30, 2022
Checkpoint: Friday, October 7, 2022 at 11:59:59pm
Due: Tuesday, October 11, 2022 at 11:59:59pm

Project 4: Exploring Job Allocation

Scheduling policies play an important role in operating systems by deciding what gets resources and when. In this project, you will gain hands-on experience in how different scheduling algorithms work and their respective pros and cons under various workloads. At the high level, you will be implementing three scheduling policies: first-in-first-out (FIFO), shortest job first (SJF), and round robin. All of the policies should be implemented in `scheduler.c` and should be compiled using a `Makefile` to an executable named `scheduler` when running `make`.

This project is divided into two phases:

1. **Scheduler Implementation:** The `scheduler` should implement the FIFO, SJF, and round robin policies. The `scheduler` takes three command line arguments: `policy name`, `job trace`, and `time slice`. For example, your implementation should support calling in the following format `./scheduler FIFO tests/1.in 0` for using the FIFO policy to schedule the jobs from the file `tests/1.in`.
2. **Policy Analysis:** The `scheduler` should output information necessary for analyzing the performance of the policies. This analysis should focus on three metrics: *response time*, *turnaround time*, and *wait time*. You will also develop five novel workloads that satisfy a given set of constraints.

All coding is to be done in the C programming language without using any non-standard libraries. The TAs will grade the project using a 64-bit Ubuntu 20.04 virtual machine. Projects that do not compile or run correctly on the TAs' virtual machines will be penalized.

We now describe each project phase in greater detail, along with how to leverage the testing framework to verify the scheduler's behavior.

Phase 1: Scheduler Implementation

Students will implement a simulated *workload scheduler*. It must accept three command line arguments:

1. the name of scheduling policy, either `FIFO`, `SJF`, or `RR`;
2. the name of the workload file, described below; and
3. the timeslice duration, which is only applicable to the round-robin policy.

In short, the scheduler will take as input a set of jobs that need to be simulated. This job set is called a *workload*. The scheduler will be responsible for determining the order in which the jobs are scheduled and the duration for which they scheduled. The order depends on the *scheduling policy*. You will be responsible for implementing three scheduling policies: first-in first-out (FIFO), shortest job first (SJF), and round robin (RR). Note that all jobs are simulated, so the simulator does not need to perform any real work.

Each workload is defined in a *workload file*. Each line of the workload file represents a different job in the workload and the provided number is the total amount of simulated time that job needs to run. For example the workload file `2.in` contains the following:

```
$ cat tests/2.in
3
10
16
```

```
15
15
9
17
11
2
10
```

Listing 1: An example input file

In this workload, there are a total of 10 jobs. The jobs are listed in the order of their arrival, so the job on the first line arrives first and has a run time of 3 units. Similarly, the second line corresponds to the job that arrives second and has a run time of 10 units.

The scheduler then uses the job file to initialize a *job list* data structure. In this list, each job should be assigned an `id` based on the line number in the workload file. For the above example, the job on the first line should be assigned an id of 0; the job on the second line should be assigned an id of 1; and so on.

More specifically, the job list should be implemented as a linked list of structs. Each struct should resemble the following:

```
struct job {
 int id;
 int length;
 // other meta data
 struct job *next;
};
```

After initialization, the scheduler will simulate the execution and scheduling of these jobs using the appropriate policy and printing output similar to the examples shown in this specification.

Policy Implementation: FIFO

The FIFO policy is one of the simplest scheduling policies, which makes it a good starting point. The FIFO policy states that jobs are scheduled in order of their arrival. Further, each job runs to completion. In other words, there is no preemption for this FIFO policy.

For example, the scheduler's output when using the FIFO policy with workload `2.in` should be:

```
$ ./scheduler FIFO tests/2.in 0
Execution trace with FIFO:
Job 0 ran for: 3
Job 1 ran for: 10
Job 2 ran for: 16
Job 3 ran for: 15
Job 4 ran for: 15
Job 5 ran for: 9
Job 6 ran for: 17
Job 7 ran for: 11
Job 8 ran for: 2
Job 9 ran for: 10
End of execution with FIFO.
```

Listing 2: Expected output when using the FIFO policy

Policy Implementation: SJF

The shortest job first (SJF) policy requires that the scheduler always pick the job with the shortest runtime to run next. Just like the FIFO policy, we again assume that all jobs will run to completion before the next job is started.

For example, the scheduler's output when using the SJF policy with workload `2.in` should be:

```
$ ./scheduler SJF tests/2.in 0
Execution trace with SJF:
Job 8 ran for: 2
Job 0 ran for: 3
Job 5 ran for: 9
Job 1 ran for: 10
Job 9 ran for: 10
Job 7 ran for: 11
Job 3 ran for: 15
Job 4 ran for: 15
Job 2 ran for: 16
Job 6 ran for: 17
End of execution with SJF.
```

Listing 3: Expected output for using the SJF policy

If two jobs need the same amount of time to execute, e.g., Job 1 and Job 9, the SJF policy breaks the tie by favoring the job that arrived earlier. In the above example, the SJF policy will first run Job 1 and then Job 9.

Policy Implementation: Round Robin

Like FIFO, the round robin policy schedules jobs in the order of arrival. Unlike FIFO, the round robin policy (RR) dictates that each job will only be run for a fixed duration of time, called a *time slice*, before another job is scheduled. If the remaining duration of job exceeds the length of the time slice, than that job must be preempted. This use of preemption distinguishes the round robin policy from the FIFO and SJF, both of which run jobs to completion once started.

The length of the time slice is passed to the scheduler as a command line argument. For example, the scheduler's output when using the RR policy with workload `2.in` and a timeslice of 9 units should be:

```
$ ./scheduler RR tests/2.in 9
Execution trace with RR:
Job 0 ran for: 3
Job 1 ran for: 9
Job 2 ran for: 9
Job 3 ran for: 9
Job 4 ran for: 9
Job 5 ran for: 9
Job 6 ran for: 9
Job 7 ran for: 9
Job 8 ran for: 2
Job 9 ran for: 9
Job 1 ran for: 1
Job 2 ran for: 7
Job 3 ran for: 6
Job 4 ran for: 6
Job 6 ran for: 8
Job 7 ran for: 2
Job 9 ran for: 1
End of execution with RR.
```

Listing 4: Expected output for using the Round Robin policy (time slice is 9)

Phase 2: Policy Analysis

In this second phase of this project, students will add code to their scheduler to help them evaluate the performance of the previously implemented policies using three metrics: *response time*, *turnaround time*, and *wait time*. Without loss of generality, we assume that all jobs arrived at the system at the same time $T = 0$.

If a job starts and completes its execution at time T_s and T_c respectively, this job's *response time* can be calculated as $T_s - T$ and its *turnaround time* can be calculated as $T_c - T$. The *wait time* describes the time the job spent waiting instead of running after its arrival. For the FIFO and SJF policies, wait time and response time are the same. For RR policy, wait time also records to time that jobs spend waiting for their next time slice.

The modified scheduler should output both the per-job and the average performance for the workload. For example, the scheduler should output the following when running the FIFO policy with workload `2.in`:

```
$ ./scheduler FIFO tests/2.in 0
Execution trace with FIFO:
Job 0 ran for: 3
Job 1 ran for: 10
Job 2 ran for: 16
Job 3 ran for: 15
Job 4 ran for: 15
Job 5 ran for: 9
Job 6 ran for: 17
Job 7 ran for: 11
Job 8 ran for: 2
Job 9 ran for: 10
End of execution with FIFO.
Begin analyzing FIFO:
Job 0 -- Response time: 0  Turnaround: 3  Wait: 0
Job 1 -- Response time: 3  Turnaround: 13  Wait: 3
Job 2 -- Response time: 13  Turnaround: 29  Wait: 13
Job 3 -- Response time: 29  Turnaround: 44  Wait: 29
Job 4 -- Response time: 44  Turnaround: 59  Wait: 44
Job 5 -- Response time: 59  Turnaround: 68  Wait: 59
Job 6 -- Response time: 68  Turnaround: 85  Wait: 68
Job 7 -- Response time: 85  Turnaround: 96  Wait: 85
Job 8 -- Response time: 96  Turnaround: 98  Wait: 96
Job 9 -- Response time: 98  Turnaround: 108  Wait: 98
Average -- Response: 49.50  Turnaround: 60.30  Wait: 49.50
End analyzing FIFO.
```

The scheduler should output the following when running the SJF policy with workload `2.in`:

```
$ ./scheduler SJF tests/2.in 0
Execution trace with SJF:
Job 8 ran for: 2
Job 0 ran for: 3
Job 5 ran for: 9
Job 1 ran for: 10
Job 9 ran for: 10
Job 7 ran for: 11
Job 3 ran for: 15
Job 4 ran for: 15
Job 2 ran for: 16
Job 6 ran for: 17
End of execution with SJF.
Begin analyzing SJF:
Job 8 -- Response time: 0  Turnaround: 2  Wait: 0
Job 0 -- Response time: 2  Turnaround: 5  Wait: 2
Job 5 -- Response time: 5  Turnaround: 14  Wait: 5
Job 1 -- Response time: 14  Turnaround: 24  Wait: 14
Job 9 -- Response time: 24  Turnaround: 34  Wait: 24
Job 7 -- Response time: 34  Turnaround: 45  Wait: 34
Job 3 -- Response time: 45  Turnaround: 60  Wait: 45
Job 4 -- Response time: 60  Turnaround: 75  Wait: 60
Job 2 -- Response time: 75  Turnaround: 91  Wait: 75
Job 6 -- Response time: 91  Turnaround: 108  Wait: 91
Average -- Response: 35.00  Turnaround: 45.80  Wait: 35.00
End analyzing SJF.
```

The scheduler should output the following when running the RR policy with workload `2.in`:

```
$ ./scheduler RR tests/2.in 9
```

```
Execution trace with RR:
Job 0 ran for: 3
Job 1 ran for: 9
Job 2 ran for: 9
Job 3 ran for: 9
Job 4 ran for: 9
Job 5 ran for: 9
Job 6 ran for: 9
Job 7 ran for: 9
Job 8 ran for: 2
Job 9 ran for: 9
Job 1 ran for: 1
Job 2 ran for: 7
Job 3 ran for: 6
Job 4 ran for: 6
Job 6 ran for: 8
Job 7 ran for: 2
Job 9 ran for: 1
End of execution with RR.
Begin analyzing RR:
Job 0 -- Response time: 0  Turnaround: 3  Wait: 0
Job 1 -- Response time: 3  Turnaround: 78  Wait: 68
Job 2 -- Response time: 12  Turnaround: 85  Wait: 69
Job 3 -- Response time: 21  Turnaround: 91  Wait: 76
Job 4 -- Response time: 30  Turnaround: 97  Wait: 82
Job 5 -- Response time: 39  Turnaround: 48  Wait: 39
Job 6 -- Response time: 48  Turnaround: 105  Wait: 88
Job 7 -- Response time: 57  Turnaround: 107  Wait: 96
Job 8 -- Response time: 66  Turnaround: 68  Wait: 66
Job 9 -- Response time: 68  Turnaround: 108  Wait: 98
Average -- Response: 34.40  Turnaround: 79.00  Wait: 68.20
End analyzing RR.
```

Policy Analysis: Novel Workloads

Finally, you must design a set of workloads that meet the following conditions:

1. Design a workload of a least 5 jobs for the RR scheduler such that the wait time is the same as the response time for all jobs. Assume a time slice of 3 time units. Save this workload into a file named `workload_1.in`.
2. Design a workload of at least 5 jobs such that the average turnaround time of the FIFO scheduler is approximately 10 times that of the SJF scheduler. Save this workload into a file named `workload_2.in`.
3. Design a workload of at least 5 jobs such that FIFO, SJF, and RR produce the same average response time, turnaround time, and wait time. Assume a time slice of 3 time units. Save this workload into a file named `workload_3.in`.
4. Design a workload of at least 5 jobs such that RR produces an average wait time of less than 5 time units but a turnaround time greater than 100 time units. Assume a time slice of 3 time units. Save this workload into a file named `workload_4.in`.
5. Design a workload of exactly 3 jobs such that FIFO produces an average response time of exactly 5 time units and an average turnaround time of exactly 13 time units. Assume that the first job has a duration of 3 time units. Save this workload into a file named `workload_5.in`.

Test Cases

We will provide a bash script named `run_tests.sh` and a directory called `tests` that includes a selection of test workloads. You should not modify the existing files, but you are welcome to add additional test workloads if you see fit.

Students can use the `run_tests.sh` to test their implementation against the provided workloads. The option `-t` specifies the test ID to run.

```
$ ./run_tests.sh -t 1
test 1: passed
$ ./run_tests.sh -t 2
test 2: passed
$ ./run_tests.sh -t 3
test 3: passed
```

Listing 5: The test cases for FIFO implementation

If a test fails, students can get more information about the specifics by passing the verbose option `-v`:

```
$ ./run_tests.sh -v -t 1
running test 1: eval FIFO policy with one job
test:       ./scheduler FIFO tests/1.in 0
test 1: out incorrect
  what results should be found in file: tests/1.out
  what results produced by your program: tests-out/1.out
  compare the two using diff, cmp, or related tools to debug, e.g.:
  prompt> diff tests/1.out tests-out/1.out
```

Listing 6: An example of getting verbose output from a failed test case

Students may also find it useful to directly inspect the files associated with each test. Below explains what each file is used for in the `tests` directory:

- *.rc: Program return code (usually 0 or 1)
- *.out: The expected standard output from running the test
- *.err: The expected standard error from running the test
- *.run: The command line to run the test
- *.desc: A short description of the test
- *.out.tmp: A temporary file for comparing the output

Finally, students may find Unix command line utilities such as `diff` useful in debugging the mismatched output formats. Please consult the corresponding `man` pages for more information.

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any assignment in which the FIFO policy implementation is completed will be considered as making substantial progress. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

When submitting the project, please include the following:

- The file `scheduler.c` that containing the scheduler code, including the phase 2 metrics.
- The novel workload files, named `workload_[1-5].in`.

- A file called `Makefile` that can be used by the `make` command for building the executable. It should support the "`make clean`" command.
- A document called `README.txt` explaining your project, each novel workload, and anything that you feel the teaching staff should know when grading the project. In particular, describe the data structure and algorithm you used to complete both the policy implementation and analysis parts. Only plaintext write-ups are accepted.

Please compress all the files together as a single .zip archive for submission. As with all projects, please **only use standard zip files** for compression; **.rar, .7z, and other custom file formats will not be accepted**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/request_teammate.php`),
2. Submit the project code and documentation via InstructAssist (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/files.php`), and
3. Complete a Partner Evaluation (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/evals.php`).

A grading rubric has been provided at the end of this specification to give you a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback. Students in teams are expected to contribute equally; unequal contributions may yield different grades for the team members.

Project 4 – Job Allocation – Grading Sheet/Rubric

				Evaluation?
Grader:		Student Name:		
Date/Time:		Student Name:		
Team ID:		Student Name:		
Late?:				
Checkpoint?:			Project Score:	/ 65

Earned	Points	Task ID	Description
_____	8	1	FIFO policy implementation – Passing test 1.
_____	3	2	FIFO policy implementation – Passing test 2.
_____	3	3	FIFO policy implementation – Passing test 3.
_____	3	4	SJF policy implementation – Passing test 4.
_____	3	5	SJF policy implementation – Passing test 5.
_____	3	6	SJF policy implementation – Passing test 6.
_____	8	7	RR policy implementation – Passing test 7.
_____	3	8	RR policy implementation – Passing test 8.
_____	3	9	RR policy implementation – Passing test 9.
_____	3	10	RR policy implementation – Passing test 10.
_____	3	11	RR policy implementation – Passing test 11.
_____	4	12	FIFO policy analysis – Passing test 12. Prerequisites: Task 1, 2, 3.
_____	3	13	FIFO policy analysis – Passing test 13. Prerequisites: Task 1, 2, 3.
_____	3	14	SJF policy analysis – Passing test 14. Prerequisites: Task 4, 5, 6.
_____	3	15	SJF policy analysis – Passing test 15. Prerequisites: Task 4, 5, 6.
_____	6	16	RR policy analysis – Passing test 16. Prerequisites: Task 7,8,9,10,11.
_____	3	17	RR policy analysis – Passing test 17. Prerequisites: Task 7,8,9,10,11.

Note that simply passing all tests is not itself sufficient to earn all the indicated points since the correctness of the implementation is judged at the discretion of the grader. For example, any assignment in which the program is hardcoded to produce the expected output will lead to point deductions. The project must have a Makefile and will receive a 4 point deduction if one is not provided.

Grader Notes: