Programming Assignment 2 (PA2)

- 截止 2月27日 由 23:59 编辑
- 得分 100
- 可用 2月3日 9:00 之后

Programming Assignment 2: Scheduling

Due: Friday, February 27, 2026 at 11:59 pm ET
You are to do this project in a team of one, two, three, or four students.
This project must be implemented in C (and not C++ or anything else)

Code for this assignment is found at
https://github.khoury.northeastern.edu/sav/CS5600_ClassRepo.git
(https://github.khoury.northeastern.edu/sav/CS5600_ClassRepo.git)

Objectives

There are four objectives to this assignment:

- To learn about scheduling processes on a system.
- To experiment with POSIX scheduling code in C.
- To explore the various aspects of scheduling.
- To understand scheduling implementation.

Overview

One piece of software that every modern operating system must contain is a scheduler: without one, only one task could be run at a time. In this Programming Assignment, your team will be writing a library to perform basic scheduling of tasks. Rather than interacting directly with the operating system, we have provided for you a discrete event simulator: we will simulate time, jobs arriving, and jobs running. Your library will inform the scheduler which job should be run next.

What you must do...

We have provided three files for you:

- `simulator.c`: **You should not edit this file.** This file is the discrete event simulator that will be run to interact with your library. You can find more information on how to run this at the end of this page.
- `libscheduler/libscheduler.c` and `libscheduler/libscheduler.h`: These two files you are free to edit to implement your scheduler.

In `libscheduler/libscheduler.h`, you'll find there is one shared variable between the simulator and the scheduler:

```
int scheme;
```

...the `scheme` variable will be set by the simulator to one of the six enum values provided: `{FCFS, SJF, PSJF, PRI, PPRI, RR}`. You should use this variable to determine what job should be run next, based on the scheduling policy that has been set. *(This variable will be set before any of your scheduler functions are called.)*

To complete this Programming Assignment, your team must implement the seven functions defined in `libscheduler/libscheduler.c`. These functions are self-descriptive, but a full function outline is provided for you below for each function:

- **scheduler_new_job()**:

  ```
  int scheduler_new_job(int job_number, int time, int running_time, int priority)
  ```

  This function is called when a new job arrives.

  Parameters:
  - `job_number`: A globally unique identification number of the job arriving.
  - `time`: The current time of the simulator.
  - `running_time`: The total number of time units this job will run before it will be finished.
  - `priority`: The priority of the job. *(The lower the value, the higher the priority.)*

  Returns:
  - if any job should be scheduled to run, the job number to run on the scheduler during the next time unit.
  - if no jobs are scheduled to run, return -1. *(Given this function adds a job, _new_job() should never return -1.)*

  Assumptions:
  - You may assume that this function will be the first function called in your library.
  - You may assume that every job will have a unique arrival time.
- **scheduler_job_finished()**:

  ```
  int scheduler_job_finished(int job_number, int time)
  ```

  This function is called when a job has completed execution.

  Parameters:
  - `job_number`: A globally unique identification number of the job arriving.
  - `time`: The current time of the simulator.

  Returns
  - if any job should be scheduled to run, the job number to run on the scheduler during the next time unit.

- if no jobs are scheduled to run, return -1.

Notes:

- The `job_number` and `time` parameters are provided for convenience. You may be able to calculate the values with your own data structure.

- **scheduler_quantum_expired()**:

```
int scheduler_quantum_expired(int time)
```

When the `scheme` is set to RR, this function is called when the round robin quantum has expired.

Parameters:

- `time`: The current time of the simulator.

Returns

- if any job should be scheduled to run, the job number to run on the scheduler during the next time unit.
- if no jobs are scheduled to run, return -1.

- **scheduler_average_waiting_time()**:

```
double scheduler_average_waiting_time()
```

This function should return the average waiting time of all jobs scheduled by your scheduler.

Assumptions:

- This function will only be called after all scheduling is complete (all jobs that have arrived will have finished and no new jobs will arrive).

- **scheduler_average_turnaround_time()**:

```
double scheduler_average_turnaround_time()
```

This function should return the average turnaround time of all jobs scheduled by your scheduler.

Assumptions:

- This function will only be called after all scheduling is complete (all jobs that have arrived will have finished and no new jobs will arrive).

- **scheduler_average_response_time()**:

```
double scheduler_average_response_time()
```

This function should return the average response time of all jobs scheduled by your scheduler.

Assumptions:

- This function will only be called after all scheduling is complete (all jobs that have arrived will have finished and no new jobs will arrive).

- **scheduler_clean_up()**:

```
void scheduler_clean_up()
```

This function should free any memory associated with your scheduler.

Assumptions:

- This function will be the last function called in your library.

Additionally, we provide one function to help you with debugging:

- **scheduler_show_queue()**:

  ```
  void scheduler_show_queue()
  ```

  This function may print out any debugging information you choose. This function will be called by the simulator before the simulator makes any other calls to the scheduler.

  In our provided output, we have implemented this function to list the jobs in the order they are to be scheduled. For example, if job(id=2) should be run, then job(id=4), and finally job(id=1), our output will be:

  ```
  2 4 1
  ```

  This function is not required and will not be graded. You may leave it blank if you do not find it useful.

Scheduler Details

The simulator will always follow a few, very specific rules. It's not important to understand the specifics of the simulator, but we provide these to help you with debugging:

- All execution of tasks will happen **at the very end of a time unit**.
- If a job's last unit of execution occurred in the previous time unit, a `job_finished()` call will be made as the first call in the new time unit.
- If a job has finished, the quantum clock will be reset. (Therefore, `quantum_expired()` will never be called at the same unit that a job has finished and the next job will get a full quantum.)
- If no job has finished and the scheme is RR, if the quantum clock has expired a `quantum_expired()` will be called.
- Finally, if any job arrives for the time unit, the `new_job()` function will be called.

There are a few specific cases where a scheduler needs to define behavior based on the scheduling policy provided. In this Programming Assignment, you should apply the following rules:

- In `PSJF`, if the job has been partially run, you should schedule the job based on its **remaining time** (not the full running time).
- In all schemes, if two or more jobs are tied (e.g.: if in `PRI` multiple jobs have the priority of 1), you should use the job with the **earliest arrival time**. In `new_job()`, we provided the assumption that all jobs will have a unique arrival time.
- In RR, when a new job arrives, it must be placed at the end of the cycle of jobs. Every job must run some amount of time before the new job should run.

- If a job was scheduled to run but was never run, its response time should not be calculated by when it was scheduled. This may happen if a `job_finish()` call returned and job A should run next, but `new_job()` was called in the same time unit and job B preempted job A before job A had a chance to run.

Examples

Consider the following simple schedule:

job number	arrival time	running time	priority
0	0	8	1
1	1	8	1
2	3	4	2

If the scheme was set to `FCFS`, the flow of execution between the simulator will be:

```
new_job(job_number = 0, time = 0, running_time = 8, priority = 1)
    --> returns 0, indicating job 0 should run next

new_job(job_number = 1, time = 1, running_time = 8, priority = 1)
    --> returns 0, indicating job 0 should continue running

new_job(job_number = 2, time = 3, running_time = 4, priority = 2)
    --> returns 0, indicating job 0 should continue running

job_finished(job_number = 0, time = 8)
    --> returns 1, indicating job 1 should run next

job_finished(job_number = 1, time = 16)
    --> returns 2, indicating job 2 should run next

job_finished(job_number = 2, time = 20)
    --> returns -1, indicating no job should run next

...
```

If the scheme was set to RR with a quantum of 2, the flow of execution for the same scheduler would be:

```
new_job(job_number = 0, time = 0, running_time = 8, priority = 1)
    --> returns 0, indicating job 0 should run next

new_job(job_number = 1, time = 1, running_time = 8, priority = 1)
    --> returns 0, indicating job 0 should continue running

quantum_expired(time = 2)
    --> returns 1, indicating job 1 should run next

new_job(job_number = 2, time = 3, running_time = 4, priority = 2)
    --> returns 1, indicating job 1 should continue running

quantum_expired(time = 4)
    --> returns 0, indicating job 0 should run next

quantum_expired(time = 6)
    --> returns 2, indicating job 2 should run next

quantum_expired(time = 8)
```

```
    --> returns 1, indicating job 1 should run next

...
```

...it's important to note that even though the jobs arrive in the order `0 1 2`, the order of round robin execution ends up being `0 2 1`.

Compile and Run

To compile this code, run:

```
make clean
make
```

To run the simulator, run:

```
./simulator <scheme> <schedule>
```

The acceptable values for `scheme` are:

- `FCFS`
- `SJF`
- `PSFJ`
- `PRI`
- `PPRI`
- RR#, where # indicates any numeric value

We provide two sample schedules: `examples/proc1.csv` and `examples/proc2.csv`. We also provide the expected output of those schedules in the `examples` directory. **It's only important that the last five lines of the output match.** We will not grade any output except the last five lines, as `show_queue()` is not required to be implemented in the same way as we did.

Grading, Submission, and Other Details

Grading Rubric

- README file clarity 5 points
- Code format clarity 5 points
- Makefile functional 5 points
- Test Cases for each of the six scheduling algorithms correctly function 10 points each x 6 = 60 points
- Miscellaneous Issues 25 points