CS3103: Operating Systems

Spring 2021

Programming Assignment 3

1 Introduction

The purpose of this assignment is to help you better understand Uni-processor Scheduling and scheduling algorithms. To achieve this goal, you will use C/C++ to implement a single-threaded program to simulate a scheduling system of a computer. The scheduling system should be able to admit new processes to ready queue(s), select a process from ready queue to run using a particular scheduling algorithm, move the running process to a blocked queue when it has to wait for an event like I/O completion or mutex signals and put them back to ready queue(s) when the event occurs.

2 Requirements

2.1 Special Notes

In this assignment, you don't need to actually schedule processes on a real CPU, which requires deeper knowledge about Linux system and is not suitable for this assignment. The scheduling system you implement would work in a simulated way. Both processes and processor are simulated by your program. So, when we say a process is dispatched to CPU or I/O device in your program, no new process or thread would be created or actually be dispatched to the device, you could just simply mark the process's status as "running on CPU" or "using I/O device".

2.2 Overall Work Flow

You are required to implement a simulated scheduling system of a computer with only one CPU, one keyboard and one disk. Each of these devices provides service to one process at a time, so scheduling is required to coordinate the processes. Your scheduling system should work in a loop to emulate the of behavior the CPU. Each loop is called a **tick**. In each loop iteration, the tick is incremented by one to emulate the elapsed internal time of the CPU.

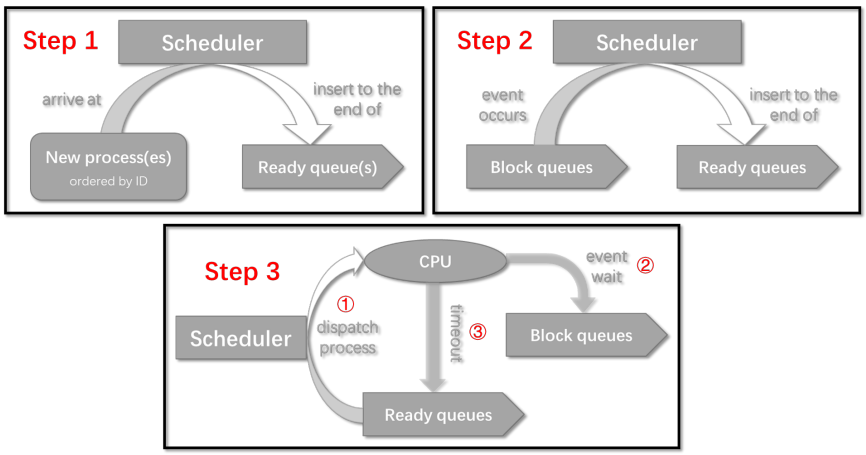

Figure 1: Overall work flow of your scheduler in each tick

As shown in Fig.1, the overall work flow of your scheduler in each tick can be divided into three steps:

Step 1. At the beginning of each tick, the scheduler would check whether there are new processes to be admitted to the ready queue. If multiple processes arrive at this tick, they should be enqueued in ascending order of their process IDs.

Step 2. There is one block queue for each I/O device and each mutex. Processes waiting for an I/O device or a mutex are inserted into the corresponding block queue. The event handler would dispatch processes from block queues in FCFS manner to the I/O devices they are waiting for if the devices become available. If a process is done with the device, it would be re-inserted to the ready queue. Besides, the event handler would also check processes waiting for mutexes. Similar to I/O, only the first process waiting in the mutex block queue is able to lock the mutex after the mutex is unlocked by another process.

Step 3. If there is no process running on the CPU, your scheduling system would dispatch a process to CPU from the ready queue according to one of the following scheduling algorithms: First-Come-First-Served (FCFS), Round Robin (RR) and Feedback Scheduling (FB). At the end of the tick, your scheduler should look ahead the service requested by the currently running process in the next tick to determine the action required. For example, if the service next tick is disk I/O, then blocking of the current process is required. If mutex operations (lock & unlock), which are assumed to have a time cost of zero tick, are encountered while looking ahead, they should be executed immediately and the scheduler would keep finding the next service which is not a mutex operation. Finally, if the process on CPU uses up its time quantum, it should be preempted and placed at the end of the ready queue (RR & FB only).

To make the implementation simpler, when a process is dispatched to CPU or I/O devices, it can be kept in its current ready queue or block queue. In other word, you don't need to move the processes somewhere else in your program to simulate that they are in the CPU or I/O devices.

2.3 Processes

The information of processes is given in a text file with the following format.

```
# process_id arrival_time service_number
(followed by service_number lines)
service_type service_description
service_type service_description
… … … …
service_type service_description
```

There are 3 integers in the first line and the # marks the beginning of a process. The first number is the process ID. The second number in the arrival time of the process in number of ticks. The third number is the number of services requested by the process. Services are the resources the process needs or some particular operations that the process executes. The following $service\_number$ lines are the services requested and the description of the service. There are 5 different types of services, namely **C** (CPU), **K** (keyboard input), **D** (disk I/O), **L** (mutex lock) and **U** (mutex unlock).

For service type C, K and D, the service description is an integer which is the number of ticks required to complete the service. It is worth noting that the number of ticks required to complete service K or D does NOT include the waiting time in the block queue. The keyboard and the disk provide I/O service to processes in a FCFS manner. In other words, only the first process in each

block queue can receive I/O service. For service type L (mutex lock) and U (mutex unlock), the service description is a string representing the name of the mutex. Similarly, if multiple processes are waiting for a mutex to unlock, after other process unlocks it, the first process waiting in the queue would get the mutex and lock it. Note that we assume the lock and unlock operation take **0 tick**, which means they should be execute immediately.

```
# 0 0 8
C 2
D 6
C 3
K 5
C 4
L mtx
C 5
U mtx
```

Above is an example of a short process. This process with ID 0 arrives at your scheduling system at tick 0. It requires 8 services in total. To complete its task, you need to schedule 2 ticks on CPU, 6 ticks for disk I/O, then another 3 ticks on CPU, 5 ticks to wait for keyboard input, then 4 ticks on CPU, 0 tick to lock the mutex *mtx*, 5 ticks on CPU, and finally 0 tick to unlock the mutex *mtx*.

For simplicity and reasonability, the possible service type at the end of every service sequences could only be C or U. And the process IDs are assumed to be consecutive integers from 0 to *N*-1, if there are *N* processes.

2.4 Blockings

Several events may lead to the blocking of current process, for example, I/O interrupts or mutex lock. You should implement **multiple blocked queues** for efficiently storing processes that are blocked by such events. To be more exact, every kind of I/O interrupt (K and F) and every mutex should have its own blocked queue. The event handler of your scheduling system should then handle the processes in blocked queues in a FCFS manner: only the first process at each blocked queue can receive the I/O service or mutex signals.

2.4.1 I/O Interrupts

You are required to implement two I/O interrupts in your scheduling system: keyboard input and disk I/O. When an I/O interrupt occurs, your scheduling system should block the current process, putting it into the corresponding blocked queue and after I/O completes, put the process back to ready queue.

2.4.2 Mutexes

Since your scheduling system is a single-threaded program, the mutexes from *pthread.h* cannot be applied to your system. Instead, you should implement your own mutex structure to allow mutual exclusion in this simulated system. Your mutex should at least have one variable to store its status (locked or unlocked) and have two functions *mutex_lock* and *mutex_unlock* to lock or unlock the mutex, just like the ones in *pthread.h.* All mutexes used by the given processes would be inferred from the given text file. Additionally, when several processes are waiting for a mutex in block queue, only the one at the front can get the mutex once it is unlocked.

2.5 Scheduling Algorithms

In this assignment, you should implement 3 scheduling algorithms: FCFS, RR and Feedback scheduling.

2.5.1 FCFS

First-Come-First-Served (FCFS) scheduling is the simplest scheduling algorithm. When the current process ceases to execute, FCFS selects the process that has been in the ready queue the longest.

2.5.2 RR

Round Robin (RR) algorithm uses preemption based on the tick of your scheduling system. Clock interrupts would be generated every K (K=5) ticks. When a clock interrupt occurs, the currently running process is placed in the ready queue, select next ready job on a FCFS basis.

2.5.3 Feedback Scheduling

Feedback Scheduling (FB) is a complicated algorithm and has many implementations. In this assignment, you will implement a simple version of Feedback Scheduling using a three-level feedback ready queue, namely RQ_0, RQ_1 and RQ_2. Different queues are of different level of priority. RQ_0 has the highest priority and RQ_2 has the lowest. Specifically, your FB should follow the four rules below:

1. A new process is inserted at the end of the top-level ready queue RQ_0.
2. When current process ceases to execute, select the first process from the first non-empty ready queue that has the highest priority.
3. Each ready queue works similarly to RR with a time quantum of K (K=5) ticks. If a process all up its quantum time, it is preempted by other processes in ready queue. The processes preempted are demoted to the next lower priority queue. (i.e. from RQ_i to RQ_{i+1}) There is no process demotion for those in RQ_2 since RQ_2 has the lowest priority.
4. If a process blocks for I/O interrupts or mutexes, after it is ready again, it is re-inserted to the tail of the same ready queue where it is dispatched earlier.

3 Input & Output samples

Your program must accept 3 arguments from command line. For example, your program should be able to be compiled by the following command:

```
g++ 5xxxxxxx.cpp -o 5xxxxxxx
```

And executes with:

```
./5xxxxxxx FCFS processes.txt outputs.txt
```

The first argument is the name of scheduling algorithm, namely FCFS, RR and FB. The second argument is the path of the process text file. Please refer to **2.3** for the format of process text file. The last argument is the name of the output file.

Your program should output a text file which stores the scheduling details of each processes. For each process, you should write two lines. The first line is the process ID of that process, i.e. process 0. The second line includes the time period that the process runs on CPU. Specifically, if a process is scheduled to run on CPU from the a-th tick to b-th tick and from c-th tick to d-th tick, then second line should be "$a\ b\ c\ d$"

```
[processes.txt]
# 0 0 7
C 3
K 9
C 5
L mtx
C 1
U mtx
C 3
# 1 0 5
```

```
C 1
L mtx
D 12
C 4
U mtx
# 2 2 5
C 3
K 5
C 5
D 7
C 5
# 3 5 3
C 6
D 8
C 5
```

The above are an example of input file, *processes.txt*. Assume that FCFS is employed to schedule these processes. The execution sequence can be deducted and the output file *outputs.txt* of your program should be something like:

```
[outputs.txt]
process 0
0 3 13 18 27 31
process 1
3 4 18 22
process 2
4 7 22 27 36 41
process 3
7 13 31 36
```

More test cases would be uploaded to Canvas later. Those example cases are for debugging purposes and may not cover all marking points. You could also design your own example for testing. Besides, a helper program is provided to help you visualize the output result. You can download it from the Canvas attachment (*vis_schedule.zip*), unzip it and open *vis.html*. This is a simple webpage on which you click the open button and upload the output text file of your program, then a scheduling diagram would be drawn on the page. The visualization diagram can help you check scheduling results and debug your program. A visualization of the above example output is as follows:

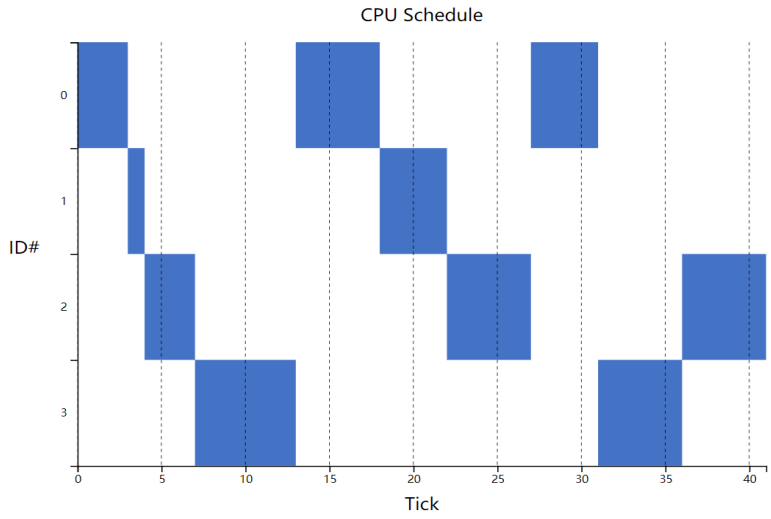

Figure 2: example visualization of outputs.txt

The blue rectangles represent the time periods during which the processes are running in the CPU.

4 Marking Scheme

Important Note: You program will be tested on our CSLab Linux servers (cs3103-01, cs3103-02, cs3103-03.cityu.edu.hk). You should describe clearly how to compile and run your program in a readme text file. If an executable file cannot be generated or the executable file cannot run on our Linux servers, your program will be considered as incorrect.

- Design of scheduling system (10%)
- Design of mutexes (10%)
- Design of blocking and blocking events (15%)
- Design of scheduling algorithms (FCFS: 10% + RR: 15% + FB: 20%)
- Program correctness (10%)
- Programming style (10%)

5 Submission

- This assignment is to be done by a group of two students, or individually if you are confident enough. You are encouraged to discuss the high-level design of your solution with your classmates but you must implement the program on your own. Academic dishonesty such as copying another student's work or allowing another student to copy your work, is regarded as a serious academic offence.
- Each submission consists of two files: a source program file (.cpp file) and a readme text file, if necessary, and all possible outputs produced by your program (.txt file).
- Write down your name(s), EID(s) and student ID(s) in the first few lines of your program as comment.
- Use your student ID(s) to name your submitted files, such as 5xxxxxxx.cpp, 5xxxxxxx.txt for individual submission, or 5xxxxxxx_5yyyyyyy.cpp, 5xxxxxxx_5yyyyyyyy.txt for group submission. Only ONE submission is required for each group.
- The deadline is 11:00 a.m., 10-April-2021 (Saturday). No late submission will be accepted.

6 Hints & Notes

- The execution order of different operations is crucial for your scheduling system. Read the requirements very carefully to make sure there are no mistakes.
- You are allowed to use C++ STL data structures like *vector* or *queue* to implement the ready queue and block queues.
- Since the workload for building the whole scheduling system from scratch is too heavy for an assignment, we provide you with a compilable and executable demo code of a simple scheduling system which only supports FCFS and disk I/O blockings (it does not include other I/O blockings or mutex implementation). You can download it from Canvas and build your scheduling system based on the given code. The way to compile and execute the demo is written in *README.txt*. But remember: you need to implement a complete scheduling system based on the demo, **submitting only the demo code or something similar would NOT get any credit**.

7 Questions?

- This is not a programming course. You are encouraged to debug the program on your own first.
- If you have any question, please submit your question to Mr WEN Zihao via the Discussion

board “Programming Assignment #3” on Canvas. To avoid possible plagiarism, do not post your source code on the Discussion board.

- If you have some specific problems about the assignment, you may contact Mr WEN Zihao at zihaowen2-c@my.cityu.edu.hk. Naïve problems like how to write the program or whether your program is correct would NOT be answered.