CS342 Operating Systems – Spring 2023
Project #2 –Multiprocessor Scheduling, Threads, Synchronization

Assigned: Mar 28, 2023.
Due date: April 20, 2023, 23:59. Document version: 1.6

This project will be done in groups of two students. You can do it individually as well. You will program in C/Linux. Programs will be tested in Ubuntu Linux. Please start as soon as possible.

Part A (60 pts):

In this project you will implement a program that will simulate **multiprocessor scheduling**. The program will simulate **two approaches** for multiprocessor scheduling: single-queue approach, and multi-queue approach (Figures 1 and 2). In single-queue approach, we have a single common ready queue (runqueue) that is used by all processors. In multi-queue approach, each processor has its own ready queue. Your program will simulate the following scheduling **algorithms**: FCFS (first-come first-served), SJF (shortest job first - nonpreemptive), and RR(Q) (round robin with time quantum Q). The program will be called as **mps**. It will take the following arguments (options).

```
mps [-n N] [-a SAP QS] [-s ALG Q] [-i INFILE] [-m OUTMODE]
[-o OUTFILE] [-r T T1 T2 L L1 L2 PC]
```

The `-n N` option is used to specify the number of processors (default is `2`). The `-a SAP QS` option is to specify the scheduling approach. `SAP` can be either `S` (single-queue approach), or `M` (multi-queue approach). If `SAP` is set to be `M`, we should have `N` to be greater than `1`. The default value for `SAP` is `M`. `QS` indicates the queue selection method for multi-queue approach. It can be either `RM` or `LM` (explained later). Default value is `RM`. For single-queue approach, it will be set to `NA` (not applicable). The `-s ALG Q` option specifies the scheduling algorithm that will be simulated. `ALG` can be one of the following values: `FCFS`, `SJF`, `RR` (default is `RR`). `Q` is the time quantum (ms) for RR scheduling (default value is `20` ms). For FCFS and SJF algorithms `Q` has to be set to `0`. The `-i INFILE` option, if specified, gives the name of an input file from which burst information (workload information) is to be taken. The `-m OUTMODE` flag specifies what will be printed to screen while the program is running (explained later) (default is `1`). The `-o OUTFILE` option specifies the name of an output file into which all output of the simulator should go (nothing will be printed to screen in this case). The `-r T T1 T2 L L1 L2 PC` option, if specified, indicates that burst information will be generated randomly inside the program using the respective argument values (explained later).

Since we will have default values for program arguments, they are optional to specify when we invoke the program. We can specify the arguments (options)

in any order. You can assume that when an option is specified, all its arguments are present and in correct form.

When simulation is started, you will get and record the wall clock time (i.e., time of day) from the system by calling the gettimeofday function. Then, whenever you need the current **simulation time**, you will get and record the current wall clock time again. The difference (in ms) between the current wall clock time and start wall clock time (i.e., the elapsed time from start) will be considered as the current simulation time, i.e., the **timestamp** (in ms) at that instant. All time units in this document are in ms.

Each **processor** will be simulated by a separate **thread**. We will call such a thread as a **processor thread** or simply a thread. To simulate N processors (N CPUs), your program will create N threads (using `pthread_create`). A **ready queue** will be implemented as a linked list (if you wish you can use another dynamic structure). Each queue will be protected by a mutex lock. If multi-queue approach is used, there will be one (ready) queue per processor, and one lock per queue.

After creating processor threads, **your program** (main thread) will **read** and process **the input file** sequentially. The input file contains information about bursts (processes) to simulate and interarrival times between them. An example **input file** content is shown below. You must follow the same format.

```
PL 80
IAT 100
PL 200
IAT 10
PL 50
IAT 70
PL 120
IAT 250
PL 100
IAT 20
PL 40
IAT 30
PL 80
```

Each line has information about another burst (PL) or another interarrival time (IAT). After reading a burst information (a PL line) from the file, the program (main thread) will create and prepare a **burst item** (a C structure). A burst item represents a cpu burst (i.e., a process) to be simulated (like a PCB). A burst item may include the following information about a burst (you may put more information if you wish): pid, burst length, arrival time, remaining time, finish time, turnaround time, and the id of the processor in which the burst has executed. Simulated processors (i.e., processor threads) have ids 1 through N. Remaining time is the remaining required cpu time of the burst. It is initialized to the length of the burst and is used only by RR algorithm. We assume each process has **a single cpu burst** to execute and does not do any I/O operation. Hence, for a process, the burst length will be equal to the process length. Pid

is burst (process) id. Pids start at 1 and increase sequentially. The first burst is assigned 1 as pid, second burst 2, and so on. Before adding the burst item to a queue, you will obtain a timestamp, and that timestamp will be the arrival time of the burst. Then you will add the burst to the tail of a queue. Access to the queue should happen after acquiring the related lock (always use locks to access shared structures). Then the program will read the next line, an **IAT line**. Let X be the IAT value that is read. The program will then sleep for X ms (by using usleep) to simulate this interarrival time. Then it will loop again and read another PL line. This behavior will continue until we reach the end of file.

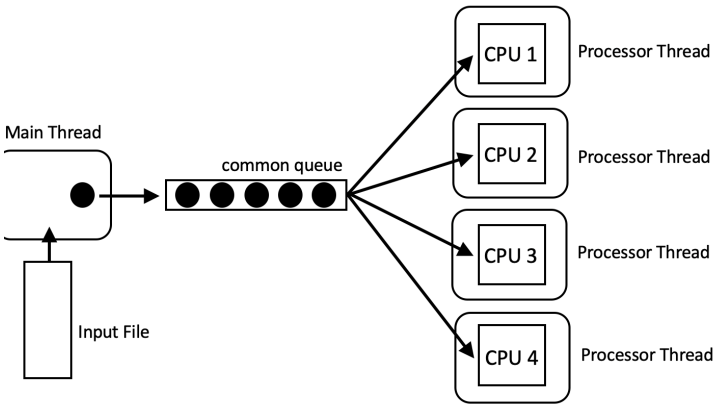

Figure 1. Single-queue approach.

In **single-queue approach**, a burst will be added to the tail of a single common queue. All processors will pick bursts from that queue. In **multi-queue approach**, bursts will be **added to queues** by using one of the following methods:

1. **Round robin Method** (`RM`). In this case, burst 1 (first burst) will be added to queue 1, next burst to queue 2, and so on. After queue N, we will add to queue 1 again.
2. **Load-balancing Method** (`LM`). A burst item will be added to the queue that has the least load at that time (total remaining length of the bursts in the queue is smallest). In case of tie, the queue with smaller id is selected.

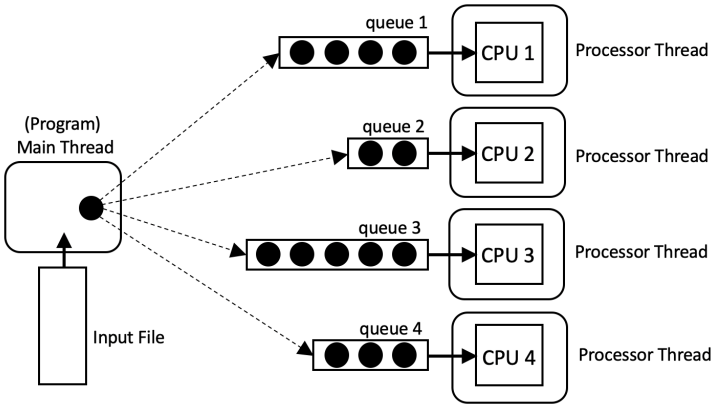

Figure 2. Multi-queue approach.

When the program (main thread) adds the last burst item into a queue, it will additionally add a **dummy item** (end of simulation marker) to each queue. Then the program (main thread) will wait until all threads terminate.

A **processor thread** will do the following in a loop till the end of the simulation. It will first **pick** (select) a process from the queue according to the scheduling **algorithm** simulated. For SJF, if there is a tie, the item closer to head is picked. If there is no process to pick, the thread will sleep for 1 ms (using `usleep`) (this is better than busy looping which will cause too much load on your processor), and will try again, until a burst item is retrieved or a dummy item is read. In Part C, you will use a condition variable to enforce a processor thread to sleep as long as the respective queue is empty. In that case you will not need to sleep for 1 ms repeatedly.

After a burst item is **picked**, its running will be simulated by sleeping (via usleep) for a while. Let X be the length of the burst. For FCFS and SJF algorithms, the thread will sleep for X ms. For RR scheduling, let R be the remaining time of the burst (R <= X). Then, if R is less than or equal to time quantum Q (R <= Q), the thread will sleep for R ms, and after that, burst is considered as finished. Otherwise, if R is greater than Q (R > Q), the processor thread will sleep for Q ms (to simulate one round of execution for the burst). After Q ms of sleeping, the processor thread will update the remaining time for the burst item (RT = RT – Q) and will put it back to the ready queue (to the tail of the queue). Then another burst item will be retrieved from the queue.

When a **burst finishes**, the processor thread will get a timestamp and consider the timestamp as the finish time of the burst. The difference between finish time and arrival time will be the **turnaround** time of the burst. The waiting time of the burst can be calculated by subtracting the burst length (that was taken from the input file) from the turnaround time. These information will be recorded in the respective burst item. We can collect all **finished burst items** into a **list**, to be sorted and processed by the main thread at the end of the simulation (do not forget to use a lock for this list as well).

This behavior will be repeated until the processor thread reads a **dummy item** from the queue, which indicates the end of simulation. If N > 1 and single-queue approach is used, the processor thread will not delete the dummy item from the queue (if you delete, you should put it back immediately), so that each processor thread can read the dummy item and detect the end of simulation. After reading a dummy item, a processor thread can terminate.

With `-m OUTMODE option`, we can specify what should be printed out to the screen while program is running. `OUTMODE` can be `1`, `2`, or `3`. If `OUTMODE` is `1`, nothing will be printed out. If it is `2`, a processor thread will print the following information whenever it picks a burst from the queue (before usleep): timestamp (i.e., current simulated time), the cpu id, the pid of the burst that is picked, burst length, and the remaining time of the burst.

An example output can be as follows (use this format):

```
time=3450, cpu=3, pid=5, burstlen=90, remainingtime=40
```

If `OUTMODE` is `3`, then a lot of information will be printed out while the simulation is running. The format is up to you. But please print related information (in a separate line) for the following events at least: a burst is to be added to a queue, a burst is picked for CPU, the time slice expired for a burst, a burst has finished.

At the end of the simulation, when all processor threads have terminated, the program (main thread) will **print** the following information (by using the burst item list collected) to the screen, no matter what `OUTMODE` is. (in ascending sorted order with respect to pid) before it also terminates. For each burst (process), it will write the pid, cpu-id (where burst executed), burst length (same value that is read from input file), time burst arrived, time burst finished, total waiting time in the ready queue, and turnaround time. After all these are printed, the *average turnaround time* (considering all processes) will be printed out.

An example output can be as follows (use this format strictly):

```
pid   cpu  burstlen   arv   finish  waitingtime turnaround
1      1    1000        0    1000       0        1000
2      1     500      200    1500     800        1300
average turnaround time: 1150 ms
```

The numbers above are just to give the format. They do not have to make sense.

If `-r T T1 T2 L L1 L2 PC` option is specified, then interarrival times and burst times will be generated *randomly* inside the program (default values for the parameters are `200, 10, 1000, 100, 10, 500, 10,` respectively). `T` is the mean interarrival time (approximately), `T1` is the minimum acceptable interarrival time and `T2` is the maximum acceptable interarrival time. Similarly, `L` is the mean burst length (approximately), `L1` is the minimum acceptable burst length and `L2` is the maximum acceptable burst length. `PC` is the number of bursts to simulate. The unit of all time related parameters is ms.

A random interarrival time (an integer) will be generated as follows. First, an exponentially distributed random value x in range $[0, \infty)$ will be generated. To do this, we take the rate parameter (λ) of the exponential distribution as $1/$`T`. For example, if `T` is 100, then $\lambda = 0.01$. Then we select a random number u between 0 and 1 (*uniformly* distributed) using the `rand()` function, and let $x = ((-1) \cdot ln(1-u))/\lambda$. If x is in range [`T1, T2`], we round it to an integer and we are done. Otherwise, we regenerate x until it is in range [`T1,T2`]. The same method can be used to generate a random burst length. The random interarrival times or burst lengths selected with this method will not be exactly exponentially distributed, but this is fine. They will be random enough.

If `-i` option is specified but `-r` option is not specified, then burst information will be taken from the specified input file (not generated randomly). If `-i` option is not specified but `-r` option is specified, then burst information will be generated randomly. If neither `-i` nor `-r` option is specified, then burst information will be generated randomly using default distribution parameter values. If both `-i` and `-r` are specified, then again burst information will be generated randomly (`-i` option will be ignored).

Example **invocations** of the program are shown below.

```
./mps -n 4 -a S NA -s RR 20 -i infile.txt -m 1
./mps -n 4 -a M LM -s SJF 0 -i infile.txt -m 2 -o out.txt
./mps -n 1 -a S NA -s RR 30 -i infile.txt
./mps -n 8 -a M RM -s RR 20 -i in.txt -m 3 -o out.txt
./mps -n 8 -a M RM -s FCFS 0 -i in.txt
./mps -n 2 -a M LM -s FCFS 0 -r 200 50 600 100 10 400 100
./mps -n 4 -a M LM -s RR 50  -r 150 10 1000 80 10 250 50
```

You will use POSIX **pthreads mutex** variables (locks). You need to get the related lock (by calling `pthread_mutex_lock` function) before accessing a queue, and you need to release the lock (by calling `pthread_mutex_unlock`) after you are done with the access. Be careful about deadlocks. Write your code in such a manner that there will be no deadlocks and race conditions.

Part B – Experiments (20 pts):

Do experiments and try to interpret the results. `OUTMODE` should be `1` while doing the experiments (nothing will be printed out while simulation is running).

a) Run experiments to compare the performance of single-queue approach with multi-queue approach. Also compare RM method with LM method. You can use average turnaround time as the metric. For RR, one quantum value will be enough.
b) Run experiments to compare the performance of the three scheduling algorithms (FCFS, SJF, RR) in terms of average turnaround time.

Repeat your experiments for various values of program arguments (N, Q).

Try to interpret your results. Plot graphs or create tables. Write a report and have it in PDF form (report.pdf) in the folder that you tar and gzip and upload.

PART C - Use of condition variables (20 pts):

You will implement the same program in part A, but this time you will use a **condition variable** to cause a processor thread to **sleep** (by using `pthread_cond_wait`) as long as the respective queue is empty. When an item is put to the queue, then the processor thread will be **waken up** (by using

`pthread_cond_signal`). For each queue, you will use another condition variable. If we have just one queue, then one condition variable is enough. With condition variables you no longer need to sleep for 1 ms repeatedly when the respective queue is empty. We call this program as **mps_cv**. Hence you will have two programs in your submission folder: `mps.c` and `mps_cv.c`.

Submission:

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '-'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source files. We should be able to compile your **programs** by just typing `make`. No binary files (executable files) will be included in your submission. Then tar and gzip the directory, and submit it to Moodle.

For example, a project group with student IDs 21404312 214052104 will create a directory named "21404312-214052104" and will put their files there. Then, they will tar the directory (package the directory) as follows:

```
tar cvf 21404312-214052104.tar 21404312-214052104
```

Then they will gzip the tar file as follows:

```
gzip 21404312-214052104.tar
```

In this way they will obtain a file called 21404312-214052104.tar.gz. Then they will upload this file into Moodle (one upload per group). For a project done individually, just the ID of the student will be used as file or directory name.

Tips and Clarifications:

- You need to initialize any mutex or condition variables you are using. And you need to destroy them as soon as you are finished with them.
- Minimum burst length can be 10 ms.
- Minimum interarrival time can be 10 ms.
- N (number of processors) can be at most 10.
- Minimum time quantum (Q) can be 10 ms. Maximum time quantum (Q) can be 100 ms.
- In RR scheduling, a burst stays always in the same queue, no matter which algorithm (RM or LM) is used in selecting a queue.
- Normally, we do not develop a simulator in the way described here, but this is just an exercise project for scheduling, multithreading and synchronization.
- "`man pthreads`" will you give information about POSIX threads API. You can also find a lot of information on the Web about POSIX pthreads.
- A thread that is simulating a processor (CPU) can understand that the last burst arrived to the ready queue via the dummy item (after reading the

dummy item from the queue). The thread should continue serving the bursts (if any in the queue) (for example with round robin) until the queue becomes completely empty.

- Use of locks is quite easy. You will define and initialize a mutex variable (i.e., a lock variable). Then you will use `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions to acquire and release a lock. You can read the related man pages.
- While a processor thread is executing a long burst (say 60 ms) with RR (e.g., q=20), another processor thread can not execute the same burst concurrently. That means a burst can be executed by one processor at a time (in other words, a burst must be executed sequentially).
- While accessing a queue (to insert a burst, or to remove a burst, or to check the size), a thread needs to get a lock first, so that we don't have race conditions.
- For single-queue approach, if RR is used, it is possible that the cpu a burst runs may change (it may start in a cpu in a round, and later it can be scheduled to another cpu in a later round). In that case, in the output cpu column, you ned to write the id of the last cpu on which the burst executed before termination (before finishing).
- In multi-queue approach, load balancing decision is given when a new burst is to be added to one of the queues. If RR is applied, it is possible that a burst execition will take several rounds. You will not do load balancing while re-inserting such a partially executed burst to the queue again. It will be added to the queue it was earlier. Hence, a burst sticks to the queue to which it is initially added (it remains in the same queue).