CS342 Operating Systems – Spring 2022
Project #2 –Synchronization and Scheduling

Assigned: Mar 4, 2022.
Due date: Mar 26, 2022, 23:59. Document version: 1.0

*This project can be done in groups of two students. You can do it individually as well. You will use the C programming language. You will develop your programs in Linux. As always, the project is **for learning** and **acquiring skills**.*

Objectives: To learn and practice with thread synchronization, process scheduling, mutex and condition variables, locking concept, multi-threaded programming, generating random values according to a distribution, collecting statistics, doing experiments, process concept and representation of processes, sleeping until a condition happens, measuring time, avoiding race conditions, considering deadlock situations and preventing them.

Part A (80 pts):

In this project you will implement a multi-threaded program that will simulate processes and their scheduling in a system. The program will be called **systemsim**.

The simulated system will have one **CPU** and two I/O devices: **I/O device1** and **I/O device2**. Processes will be simulated with threads. We will call these threads as **process threads** or simply **threads** or **processes**. Hence, if we want to simulate the existence of N processes in the system, then there will be N threads (created with `pthread_create` function). There will be a ready queue (representing the list of READY processes) for the CPU. Ready queue will have an associated condition variable. Each I/O device will also have an associated condition variable. Threads simulating processes will sleep on these condition variables when necessary.

Each process thread will have a **C structure** associated with it, keeping information about the process, like its pid, state, corresponding thread id, next CPU burst length, total time spent in the CPU, etc. We can call that structure as **PCB**. One of the threads will be in RUNNING state, the others will be in READY or WAITING state. A thread in READY or WAITING state will be sleeping on the related condition variable. A thread in RUNNING state will have the CPU. A thread in WAITING state will be using the related device when its turn comes. You can represent the CPU and I/O devices with proper C structures as well.

The **ready queue** will have a condition variable on which process threads in READY state will sleep, waiting to be scheduled. A condition variable has an *internal queue* where sleeping threads are represented, but we do not have control over the condition variable internal queue. Therefore, you have to implement a ready **queue explicitly**. We need explicit queue implementation,

because we want to do CPU scheduling with a particular scheduling algorithm (FCFS, SJF, or RR). For SJF algorithm, serving from the queue will not be in FIFO order. You can implement your **queue** as a singly or doubly **linked list** of **PCBs**. It is also possible to implement your queue as a **linked list** of **nodes,** where nodes are keeping some identifying information (or *pointers* to PCBs) about threads sleeping on the condition variable of the ready queue.

Besides the process threads simulating the processes, we will have two **control threads**: 1) a *process generator thread*: it will create new process threads and these new threads will add themselves to the ready queue; 2) a *CPU scheduler thread*: it will select a process thread to run next.

Process generator thread: initially, process generator thread will create 10 process threads immediately (representing processes). Then, the process generator thread will execute a loop. In the loop, it will first sleep for 5 ms (using `usleep`). After that, if the number of process threads in the system is not MAXP, it will create another thread with probability pg. Then it will loop again. If the total number process threads created has reached ALLP, then no more process threads will be generated. The simulation will end when the last process thread terminates and leaves the system.

CPU scheduler thread: while not running, the CPU scheduler thread will sleep on its condition variable. We can call this condition variable as scheduler condition variable. When waken up, the scheduler thread will check if scheduling is necessary or not. If it is not necessary, it will seep again. If it is necessary, then the scheduler thread will select a process thread from the ready queue. The selection will be done according to the scheduling algorithm specified at command line. The selected node's (or PCB's) thread will use the CPU next. To use the CPU, the selected thread should be waken up. For that the scheduler thread can use the **broadcast** operation on the condition variable of the ready queue, instead of signal operation. This is because we cannot wake up a specific thread by just doing a signal operation on the condition variable (when we signal we do not know who will be waken up). A broadcast operation will wake up all the process threads sleeping on the condition variable of the ready queue. Therefore, each waken up process thread must check in a **while loop** if it is selected for CPU or not. If not, the process thread will sleep again on the ready queue condition variable, otherwise it will start running in the CPU (**simulated running**).

After scheduling a process, the CPU scheduler thread will sleep again on its condition variable. The CPU scheduler thread is waken up in the following cases:

1. when the running thread terminates and leaves the CPU,
2. when the running thread starts I/O and leaves the CPU,
3. when time quantum for the running thread expires (for RR scheduling only) and it gets added back to the ready queue.
4. when a process returns from I/O and gets added to the ready queue,

5. when a new process thread is generated and added to the ready queue.

The scheduling algorithm can be FCFS, SJF, or RR(q). For FCFS and RR scheduling algorithms, the thread at the head of the ready queue is selected and the corresponding thread is waken up to use the CPU. If the scheduling algorithm is SJF, however, the thread that has the smallest next CPU burst length is selected from the queue and the corresponding thread is waken up to use the CPU.

A process thread: a process thread starts its life when it is created by the process generator thread. A process thread simulates a process. Here, we will also call a process thread simply a **thread**. Each thread will have a **PCB** allocated. A thread can do the allocation by itself or the process generator thread can allocate the PCB for the thread it has created. A created thread will initialize and fill its PCB structure with necessary information. It will also determine its **next CPU burst length** and put that information into its PCB as well. Then, the thread will add a new node (for itself), or its PCB, to the end of the ready queue and will sleep on the condition variable of the ready queue by calling the **wait** operation.

When selected for CPU, a thread is waken up. The waken up thread will remove its node (or its PCB) from the ready queue and will start using the CPU. It will run in the CPU for x ms where x is its next CPU burst length (ms). If the scheduling algorithm is RR, however, it will run at most q ms. The **running** of the thread will be **simulated** by the thread by sleeping x ms (via **`usleep`**), or q ms at most (in RR algorithm). For example, if the scheduling algorithm is FCFS and the CPU burst length of the process thread is 30 ms, then the thread will sleep for 30 ms. Then we consider the thread has used the CPU for 30 ms and finished its burst. If scheduling algorithm is RR, the thread may expire its **time slice** before finishing its burst. In this case, it will add itself back to the ready queue and sleep there.

After **finishing the burst**, the running thread will leave the CPU, and will either terminate, or will go for I/O with one of the I/O devices. The thread will terminate with probability p0, or will go for I/O with device1 with probability p1, or will go for I/O with device2 with probability p2 (p0 + p1 + p2 = 1). When the running thread **decides to go for I/O** with a device, it will do the following. If the device is unused and there is no thread sleeping on the condition variable of the device, then the thread will start using the device immediately (you can use an **integer variable to count** the threads sleeping on a condition variable). Otherwise, the thread will sleep on the condition variable associated with the device.

A thread using an I/O device will use the device for T1 ms (if it is device1) or for T2 ms (if it is device 2). T1 and T2 are fixed time intervals (ms). This usage can be **simulated** again by sleeping for Tx ms (using `usleep`). After that, the thread will leave the device. The leaving thread will wake up (via

signal operation) another thread to use the device if there is any such thread sleeping on the device condition variable.

The thread that has used and left the device, i.e., completed I/O with it, will go to the ready queue. Before that it will determine its **next CPU burst length** value and put it into its PCB. A process thread will be able to access its own PCB structure (it is ok, this is simulation). Then the process thread will add itself to the end of the ready queue and will sleep on the condition variable of the ready queue.

You **do not need** to implement a queue for a device explicitly (for *simplicity*). Use of the condition variable's **internal queue implicitly** will be enough (a thread is added to that queue when it does a wait operation on the device condition variable). We do not care about how a condition variable selects a thread to wake up when the condition variable is signaled. That means we do not care about **how I/O is scheduled**. Probably, a condition variable is using a FIFO queue internally.

In this way, a process thread will **travel** between different queues and components of the system during its lifetime. The **state** of the thread can be READY, RUNNING, or WAITING. You can use additional states if you wish. Finally, the thread will terminate and will be deleted (it will call `pthread_exit`).

The program will have the following command line **parameters**.

systemsim <ALG> <Q> <T1> <T2> <burst-dist> <burstlen> <min-burst> <max-burst> <p0> <p1> <p2> <pg> <MAXP> <ALLP> <OUTMODE>

<ALG> can be one of "FCFS", "SJF", or "RR". <Q> is the time quantum (time slice). For FCFS and SJF, it is "INF" (infinite). <T1> is the service time of device1 (i.e., how long it will take to do one I/O operation with the device). Max <T1> is 100ms, min <T1> is 30ms. <T2> is the service time of device2. Max <T2> is 300ms, min <T1> is 100ms. <burst-dist> can be one of "fixed", "exponential", or "uniform". <burstlen>, <min-burst>, and <max-burst> are parameters related with the burst distribution. <p0> is the probability that a running thread will terminate. <p1> is the probability that a running thread will go for I/O with device 1. <p2> is the probability that a running thread will go for I/O with device 2. <pg> is the probability that a new thread is generated. <MAXP> is the maximum number of process threads that can exist simultaneously in the system. Maximum value of <MAXP> is 50ms, minimum value is 1. <ALLP> is the number of process threads that will be created before simulation ends. Max value of <ALLP> is 1000. Min value is 1.

<OUTMODE> specifies what should be printed out to the screen while *simulator is running*. It can be 1, 2, or 3. If <OUTMODE> is 1, nothing will be printed out. If it is 2, the running thread will print out the current time (in ms from the start of the simulation), its pid and its state to the screen. It should

do this before going to sleep (using `usleep`). An example output can be as follows (use this format):

```
3450 7 RUNNING
```

A thread that is using a device (doing I/O with device1 or device2) will print out the current time (ms), its pid, and the device that it is using. This printing out will happen before the thread calls the `usleep` function to sleep for T1 or T2 ms. An example output can be as follows (use this format):

```
4210 9 USING DEVICE1
```

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: new process created, process will be added to the ready queue, process is selected for CPU, process is running in CPU, process will be added to the device queue (i.e., the related condition variable's internal queue), process doing I/O with the device, process expired time quantum, process finished, etc.

Example **invocations** are shown below.

```
./systemsim FCFS INF 50 100 fixed       20 10 100 0.1 0.6 0.7 0.5 30 200 1
./systemsim SJF  INF 50 100 fixed       20 10 100 0.1 0.6 0.7 0.3 30 200 1
./systemsim RR   10  50 100 fixed       20 10 100 0.1 0.6 0.7 0.7 30 200 2
./systemsim FCFS INF 50 100 exponential 10 10 100 0.1 0.6 0.7 1.0 25 200 3
./systemsim FCFS INF 50 100 uniform      0 10 100 0.1 0.6 0.7 0.5 40 200 2
```

For a process thread, when the **next CPU burst length** is to be determined, the value will be selected between <min-burst> and <max-burst>. For example, the selected value can be 40ms. If "fixed" burst length will be used, the selected value will be equal to the <burstlen> parameter. If selection will be according to **uniform** distribution, a uniformly distributed random value between <min-burst> and <max-burst> will be selected. In this case (uniform), the <burstlen> parameter will have value 0 and therefore will not be used. If selection will be according to **exponential** distribution, an exponentially distributed random value will be selected. The λ parameter of the exponential distribution will depend on the <burstlen> parameter value (in ms) entered at command line. No matter which way is used, the selected next CPU burst length value should be in range [<min-burst>, <max-burst>]. The minimum value of <min-burst> is 10ms. The maximum value of <max-burst> is 10000ms. Typical values will be 10ms for <min-burst> and 100ms for <max-burst>.

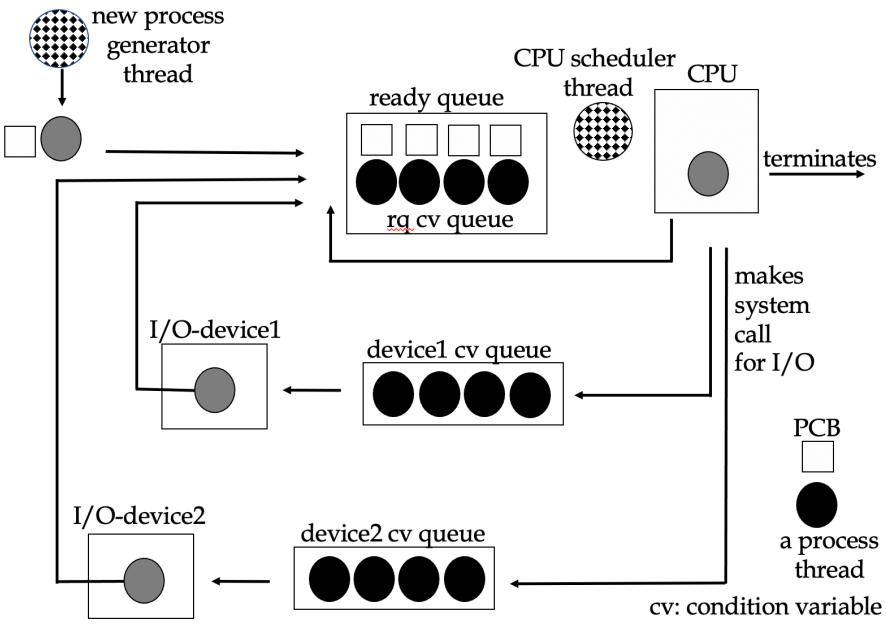
A **PCB structure** may contain the following information about a process thread:

- pid (a positive integer)
- thread id (of type `pthread_t`),
- state,
- next CPU burst length (ms),
- remaining CPU burst length to execute (ms) (for RR scheduling),

- number of CPU bursts executed so far,
- time spent in ready list so far (ms),
- number of times I/O is done with device1,
- number of times I/O is done with device2,
- start time (ms), i.e., arrival time,
- finish time (ms),
- total execution time in CPU so far,
- ….

You can put more information into a PCB if you wish. Note that, when the simulator is **started**, **time is 0** (ms). You can record the real time when the simulator has started, and then, whenever you need, you can measure the elapsed time (in ms) between the current real time and the recorded start time. To obtain the real time you can use the **gettimeofday** function. The elapsed time calculated as **t** (ms) at a moment will be the **current time** in ms at that moment.

You will use **POSIX pthreads mutex** and **condition variables**. You will use **locks** (mutex variables) while accessing shared structures or variables. You need to define several locks for that purpose. For example, a **lock** for the ready queue is needed. Be careful about **deadlocks**. Write your code in such a manner that no deadlocks will occur. Avoid **race conditions** by using locks. The program should not have race conditions while running.



The figure above shows a **snapshot** of the system at a time t. There are 4 process threads sleeping on the ready queue condition variable. Their PCBs (or nodes) are in the ready queue. That means there are 4 processes in READY state. There is one process running in CPU. The CPU scheduler thread is sleeping on its condition variable. There are 4 process threads sleeping on

device1 condition variable. There is a thread using the device1. A similar situation exists for the second device: device2. The process generator thread has created another new thread. That new thread has its PCB as well. The created thread will add itself to the ready queue. The process generator thread will not wait on any condition variable. It will just periodically sleep using the synchronous `usleep` function in a loop.

At the **end of the simulation**, when the last process (thread) has left the system (terminated), the program will write the following **information** to the **screen** before terminating. For each process, it will write the pid, time process arrived (ms – from the start of the simulation), time process finished (ms), total time process has run in the CPU (ms), total time process waited in the ready queue (ms), turnaround time (ms), number of CPU bursts executed, number of times device1 is used, and the number of times device2 is used. An example output can be as follows (use this **format** strictly):

```
pid  arv dept  cpu    waitr  turna   n-bursts  n-d1  n-d2
1    10  5010  1000   2000   5000    20        5     3
2    20  7020  1500   1500   7000    50        4     6
3    40  8000  2000   2500   7960    25        6     5
```

The numbers above are just to give the format. They may not make sense always.

Part B – Experiments (20 pts):

Measure the arrival time, finish time, total waiting time in the ready queue, and turnaround time of the first 30 processes. Do this for each scheduling algorithm: FCFS, SJF, RR(q). It is up to you to set the parameters to proper values. Do a lot of such experiments. Plot graphs or create tables. Try to interpret the results. Write a report and convert the report to PDF. Have it in PDF form in the folder that you tar and gzip and upload.

Submission:

Put all your files into a directory named with your Student ID (if done as a group, the ID of one of the students will be used). In a README.txt file, write your name, ID, etc. (if done as a group, all names and IDs will be included). Do not include executable file in your tar package. Include a Makefile so that we can just type make and compile your program. Then tar and gzip the directory. For example, a student with ID 21404312 will create a directory named "21404312" and will put the files there. Then he will tar the directory (package the directory) as follows:

```
tar cvf 21404312.tar 21404312
```

Then he will gzip the tar file as follows:

```
gzip 21404312.tar
```

In this way he will obtain a file called 21404312.tar.gz. Then he will upload this file into Moodle.

Tips and Clarifications:

- Start early, work incrementally.
- Each process will have a unique pid >=1. Pids will be recycled. When a thread is started, it will be assigned the smallest unused pid (integer). Hence you should keep track of the used pid values (maybe with a list or array).
- The process generator thread and the CPU scheduler thread will be created when the simulator program is started. You may want to do some initialization before you create these two threads. These control threads will be terminated before the program terminates.
- In this project, all processes have equal priority.