**PEX 3: A Multi-Threaded CPU Scheduling Simulator - 100 Points**

Due: 2300 hrs after Lesson 25, Thursday, 19 Mar / Friday 20 Mar

# Help Policy

|  |
| --- |
| **AUTHORIZED RESOURCES:** Any, except another cadet’s assignment or published solutions to the assigned problem.  **NOTE:**   * Never copy another person’s work and submit it as your own. Here are a few blatant examples of copying:   + Making an electronic copy of another cadet’s solution and then modifying it slightly to make it appear as your own work.   + Reading a printout or other source of another cadet’s work as you implement your solution.   + Completing your entire solution by following explicit instructions from another cadet, while he/she refers to his/her own solution * Do not jointly implement a solution. * Helping your classmates learn and understand the homework concepts is encouraged, but extensive assistance should generally be provided by DFCS instructors. Only provide assistance up to your depth of understanding, beyond which assistance by more qualified individuals is more appropriate and will result in greater learning. If you have to look at your solution while giving help, you are most likely beyond your depth of understanding. * Help your classmates maintain their integrity by never placing them in a compromising position. Do not give your solution to another cadet in any form (hard copy, soft copy, or verbal). * **DFCS will recommend a grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.** **Allowing another cadet to see your assignment to help them will result in a zero on this assignment.** |

# Documentation Policy

|  |
| --- |
| 1. You must document all help received from sources other than your instructor or instructor-provided course materials (including your textbook). 2. The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance. 3. If no help was received on this assignment, the documentation statement must state “NONE.” 4. If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why. 5. Vague documentation statements must be corrected before the assignment will be graded and will result in a grade deduction equal to 5% (ceiling) of the total possible points. |

**OBJECTIVES**

* Be able to program multi-threaded programs.
* Be able to use semaphores and mutexes to synchronize programs, ensure mutual exclusion, prevent race conditions, and prevent deadlock.
* Understand different CPU-scheduling algorithms and use them to schedule a set of processes.

**SUBMISSION INSTRUCTIONS**

Submit your PEX via Moodle. Submit all your c source (.c), header (.h), and object (.o) files and your makefile.

Provide your documentation as a separate documentation.txt file. You may include in-line documentation, but please consolidate your documentation into one file.

Do NOT structure your project into subfolders or compressed archives. Your instructor will download your submission, extract it into one folder and should be able to type make clean and then make and then run your program.

NOTE:

* Moodle documentation has been disabled for this assignment. All your documentation must be in your documentation file. As you will not have Moodle to enforce the quality of documentation, be sure you are thorough in your documentation of WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance.
* There is a 2MB file size limit on Moodle. If you exceed this limit for this assignment there is probably something drastically wrong with your code, but you may submit your files via email if that becomes necessary.

**OVERVIEW**

For this PEX, you must implement several scheduling algorithms which will be used in a multithreaded simulation of a multiple processor computer. A [framework](https://piazza.com/class_profile/get_resource/i408kvi6dkf6kd/i6wsqwv9qf9ic) has been provided that includes a makefile, two object files, two header files, and a **template** CPUs.c file. Your task is to modify the CPUs.c file to implement the scheduling algorithms.

You may include any additional files as you see fit, but **DO NOT change any of the header files**. This PEX will involve working with semaphores to synchronize the threads involved in the simulation, and mutexes to protect access to shared data structures (specifically, the ready and finished queues where processes are stored).

**You must write your PEX in C on your Ubuntu virtual machine. It is also required that you use gcc to build and make to manage your project. Do NOT program in Windows.**

**CPU THREADS AND SYNCHRONIZATION**

The threads in the simulation consist of the main thread and one for each CPU. The main thread, which has been implemented for you and is explained in the next section, is responsible for signaling the CPU threads when it is time to begin the next timestep in the simulation. The CPU threads signal the main thread when each timestep is complete. Therefore, each thread has a semaphore associated with it.

When a given scheduling function is invoked, it is passed a pointer to a cpuParams struct. As specified in CPUs.h, this struct contains 1) an integer indicating the current CPU’s ID, or thread number; and 2) a sharedVars struct. You can also find the sharedVars definition in CPUs.h, but it includes the mutexes protecting access to the ready and finished queues, an array storing the semaphore for each CPU called cpuSems, the semaphore for the main thread, the ready and finished queues, and the quantum (which only matters for certain scheduling algorithms). By typecasting the void\* param as a cpuParams\*, you can extract the thread number and the sharedVars\*.

The main thread’s semaphore can vary between 0 and the number of CPUs. This semaphore tracks how many CPUs are still executing the current timestep. Each CPU’s semaphore should be restricted to only 0 or 1[[1]](#footnote-1) and should be used to pause and resume execution. Here’s how that should work, and what you are responsible for in each scheduling function:

* At the beginning of an infinite while loop, wait on the CPU’s semaphore, as indexed in the cpuSems array by the CPU’s ID (thread number). The main thread will eventually signal the CPU via the semaphore to simulate a timestep.
* Decide either to continue executing the process the CPU had during the previous timestep, or select a process from the ready queue to execute, if one is available.
* When a process is removed from the ready queue, print a statement that it has been scheduled and list its process ID. Treat this as part of the queue manipulation (i.e. inside the lock).
* To simulate executing a process, decrement the remaining burst time.
* If the process finishes, place that process in the finished queue.
* Once complete, signal the main thread (via the main thread’s semaphore), indicating that this thread is ready to proceed to the next timestep.

Access to both the ready and finished queues should be protected with mutex locks. Your threads should hold the lock for only the least amount of time that’s needed.

You will be responsible for implementing the following scheduling algorithms:

* First Come First Served (FCFS) (non-preemptive)
* Shortest Job First (SJF) (non-preemptive)
* Non-Preemptive Priority (NPP)
* Round Robin (RR) (preemptive)
* Shortest Remaining Time First (SRTF) (preemptive)
* Preemptive Priority (PP)

**MAIN THREAD**

The main thread of the program has been implemented for you. It reads in and checks the command line arguments to determine how long to run the simulation, how many processors to create, what scheduling algorithm to use (shared for all CPUs), and how long the quantum should be (relevant for certain scheduling algorithms). It then spawns off threads to simulate the CPUs and begins the simulation. During the simulation, the main thread stochastically generates an arrival, and then signals the CPUs to simulate one timestep. Once it receives notification that all CPUs have finished their simulations for the round, it updates the wait times of all processes in the ready queue and decreases the time left for the simulation before beginning the next timestep. Upon completion of the simulation, the main thread prints the full states of both queues before exiting, which cancels all CPU threads in the process.

**Notes**

* Running processes are held in CPUs and should be removed from the ready queue
* FCFS, SJF, and NPP only need to make a scheduling decision when the process they have been simulating is complete
* In PP and SRTF, you will need to check every timestep to see if there is a higher priority/sorter process that should preempt the process they are running.
* RR will requeue its running process every quantum timesteps and pull the first process off of the ready queue.
  + Be sure to set the requeued flag when placing a process back onto the ready queue.
* Lower numbers represent higher priority process (i.e. schedule priority 0 first)
* Ties are handled in the following way:
  + Keep the process in the CPU on the CPU
  + First come first served (as determined by location in the queue)
* Structs you will work with directly include:
  + processQueue.h
    - process
    - node
    - queue
  + CPUs.h
    - sharedVars
    - cpuParams
* Functions you will work with directly include:
  + processQueue.h
    - Qinsert
    - Qremove

**PLAN OF ATTACK/HINTS**

1. Start early. Compile and test often. Backup your code. Schedule and come in for EI early.
2. Read the entire assignment, and think about your design first. In particular, think about what structures have been provided and how your threads interact with the provided code.
3. Think about how you will test your code.
   1. To get a usage statement to see the parameter list, call your program as  
      $ ./PEX3 –h  
      You can call your program with parameters or with user interaction.
   2. Be aware that the first parameter controls debug output, meaning you can print the states of the queues at each time step or not.
   3. Also, the prompts and beginning/finished simulation lines are output to stderr. This means that you can redirect the output of the program to a file. For example, call your program as $ ./PEX3 1 25 1 1 > out.csv
4. Start with First Come First Served. Once this is implemented and working correctly, you can use it as a template for your other scheduling algorithms, which should only require minor modifications.

**EXPECTATIONS**

|  |  |
| --- | --- |
| **Requirements**   1. Design/Implementation/Coding Standards | **+4/4** |
| 1. Functionality    1. General CPU functionality       1. Properly manipulation of ready & finished queues       2. Properly synchronizes CPUs and main thread       3. Correctly holds and manipulates a process          1. Each CPU exclusively holds its running process          2. Correctly decrements remaining CPU burst    2. FCFS functions properly       1. Selects correct process to run       2. Process runs to completion    3. SJF functions properly       1. Selects correct process to run       2. Process runs to completion    4. NPP functions properly       1. Selects correct process to run       2. Process runs to completion    5. RR functions properly       1. Selects correct process to run       2. Process runs to completion or preemption       3. Proper replacement into ready queue    6. SRTF functions properly       1. Selects correct process to run       2. Process runs to completion or preemption       3. Proper replacement into ready queue    7. PP functions properly       1. Selects correct process to run       2. Process runs to completion or preemption       3. Proper replacement into ready queue | **+10/10**  **+8/8**  **+12/12**  **+12/12**  **+18/18**  **+18/18**  **+18/18** |
| **Penalties**   1. Vague/Missing Documentation (5%) 2. Late Submission (25% cap/day) | **-0/5** |
| **Total** | **100** |

1. Ordinarily a semaphore restricted to values of 0 and 1 would be a mutex - but we are not using a pthread\_mutex \_t because the signal and wait calls are in different threads. [↑](#footnote-ref-1)