Assignment 5

Georgiy Krylov

October 27, 2024

ASSIGNMENT IS TO BE COMPLETED INDIVIDUALLY BY ALL STUDENTS!

1 Description

This assignment is to learn more about synchronization problems, practice synchronization mechanisms like semaphores and mutexes. **The assignment is Due by 11:59 p.m. on Sunday, 3rd of November 2024 (one minute before Monday).**

2 Task

In this assignment you're to extend your simulation of the **Round Robin** scheduling algorithm with multiple threads. The threads are to simulate multiple CPUs having their own schedules. To assign a job to a CPU, you need to use the processor affinity - a new field in a job's description.

2.1 Requirements

First, your program should read the number of CPUs. After that, your program should read the inputs, very similar to the previous assignment, except there's a new field: Affinity. This field is represented by an integer value and corresponds to the ID of the (simulated) CPU, which should be executing the job. The jobs with affinity 0 can only be executed by the CPU0, the Jobs with affinity 1 can only be executed by the CPU1, etc.

Your program should have three kinds of threads:

- The main thread, which has several responsibilities:
  - Reading the input
  - Creating the CPU threads and a printing thread
  - Printing the summary

- CPU thread(s) - threads that simulate one core of CPU each. You will need to make your entire simulation from the previous assignment solution to run as a separate thread, using `pthread_create()`. The number of the threads should correspond to the number of the CPU cores, which is a user input. Each of the CPUs can have a local running queue, but should compete for access to the "input queue" with another CPUs. Every unit of time within the simulation, the CPUs should send the jobs they were executing to a buffer shared with the printing thread.
- Finally, your program should have a printing thread, also created using `pthread_create`. The thread will be executing a function that is responsible for printing the time and jobs currently performed by each CPU. This function should wait for all CPU threads to finish simulating one unit of execution, and print the contents of the shared buffer, populated by the CPUs. After that, the printing thread should signal the CPU threads that they can resume their simulations. Importantly, this thread should be created before the CPU threads.

Finally, a summary should be printed following the same rules as in the previous assignment.

2.2 Limitations

For this exercise, you are prohibited from allocating a 2d array that would store history of CPU states. The shared buffer (can be a global variable) should be only big enough to hold one character per CPU. You are to use semaphores and mutexes, not pipes. Solutions using busy wait loops will be lightly penalized for each busy wait loop. Solutions using sleep for synchronization will lightly be penalized for each sleep statement in the code.

2.3 Working solution for the previous assignment

If you did not manage to finish the previous assignment or your design choices make the change too difficult, instructor's solution to the previous assignment will be posted on Thursday (29th of October) evening, under Course Content/Misc. useful stuff.

3 Input/Output format

The assignment will contain two lines before the processes table: the number of CPUs, and their corresponding quantums. See the attached sample_input1.txt and sample_output1.txt for clarifications.

4 Submission instructions

Please submit your C file to D2L Assignment box. Make sure your code compiles and runs. Make sure your code follows the specified input/output format. You must use C programming language to solve this assignment. Important to note in this course: if your program produces correct output, it doesn't necessarily mean you have properly managed the resources and penalties are possible. This assignment focuses on synchronization, threads management and communication, and therefore the TA will be asked to make sure the threads are properly created and terminated,semaphores and mutexes are properly allocated and deallocated, critical sections of the code are properly synchronized.

NOTE: THE INPUT AND OUTPUT OF YOUR PROGRAM IS SPECIFIED SUCH THAT THE MARKER CAN AUTO TEST YOUR SUBMISSIONS. PLEASE FOLLOW THE SPECIFICATIONS! INPUT WILL BE READ VIA stdin (E.G. KEYBOARD). OUTPUT SHOULD BE PRINTED TO stdout (E.G. MONITOR).