CSC 246 Fall 2019 Homework 2

Due: February 4 2019, 11:55PM

ChangeLog

01/31: clarification on MLFQ that within each queue, we make it FIFO. See below for details.

01/29: homework 2 online

Instructions

(a) You can work on any current Linux installation. We will use the EOS Linux machines (one of the Linux desktop machines in EB 3 or Daniels, or via remote-linux.eos.ncsu.edu) to do the grading. If you are using Linux, type "ssh [unity_id]@remote-linux.eos.ncsu.edu" to access the machines remotely. For Windows, use "putty" to ssh into remote-linux.eos.ncsu.edu. Use your campus password to get through.

(b) Makefile and README: Use Makefile to compile all source files of your code. Use gcc as your compiler. Also write a README explaining what your code does and how to compile and execute it. The README is in addition to the comments in your source files at critical points in your code explaining what does that piece of code do. You can refer to the lab tutorial chapter of the OSTEP book for some help on Makefile.

(c) File names: For programming problems, name your source files as prob_x.c and header files as prob_x.h, where x is the question number here. Try to just use one .c and one .h file for each problem. For non-programming problems, put the anwser to one single ASCII format text file and name it problems.txt. Please submit ASCII format text file ONLY for non-programming problems. You may be penalized for incorrect filenames or formats.

(d) The first line of each file: You MUST put your name and unity id on the top of each file you submit, formatted as follows:
/* unity-id FirstName MiddleInitial LastName */
Don't leave any field blank.

(e) Before you submit: *** IMPORTANT*** Please make sure your programs compile and execute normally on EOS Linux machines, you will receive 0 for programs that do not compile or do not execute. You will receive partial credit for programs that do not produce correct output.

Problem 1

In this problem, we will examine how different scheduling policies affect performance. The table below shows the time when each process arrives in the ready queue and the time each process takes.

Process	Arrival Time	Execution Time
P1	0	12
P2	1	8
P3	3	6
P4	6	4

P5	**10**	**2**
P6	**15**	**1**

We will examine three different polices:

1. FIFO
2. Round Robin with time quantum = 4
3. MLFQ where 1) the i-th queue has a time quantum q=2^(i-1) (2 to the power i-1), 2) all processes start in the first queue with i equals to 1 and move to the next queue once the time quantum in the current queue is used up, 3) the rule to boost processes to the highest-priority queue from the lowest-priority queue will not be applied, as the duration of the processes is short in this problem, 4) if a process in the i-th queue is running and a process in the j-th queue (where j is less than i) becomes ready, the process in the j-th queue will run immediately, 5) within each queue, we use FIFO, and preempted processes will not change its location in the queue.

With round robin, it could happen that one process just arrives and enters the ready queue and another process is preempted and returns to the ready queue. If this happens, put the preempted process behind newly arrived process in the queue (so the newly arrived process will get to run earlier).

For polices with a time quantum, if a process finishes, regardless of whether the current quantum is used up, the next process will be scheduled immediately.

(a) Draw a Gantt chart for each of the three different polices under the given workload. Make sure you label the process ID and timestamp correctly.

(b) Also, for each schedule, compute average response time, average turnaround time, and average waiting time, where the waiting time is the amount of time that is taken by a process in ready queue (being in the ready queue means the process is not running).

Put your answers in problems.txt (ASCII file).

Problem 2

In this problem, we're going to learn how to use Valgrind, a powerful debugging tool. Although Valgrind can detect different types of bugs, we only focus on Helgrind, which is a part of Valgrind for data race detection.

The manual page for Helgrind contains a simple data race example and instructions on how to interpreting the race error messages. We will try that example and check the results. The example under Section 7.4.1 is provided here as race.c.

Compile it with "gcc -g -o race race.c -lpthread"
and then invoke Helgrind with "valgrind --tool=helgrind ./race".

1. What is the output you get? Just copy and paste.
2. How does your output compare with the output listed on the manual page? Please also describe your interpretation of the difference.

Put your answers in problems.txt (ASCII file).

Problem 3

Our goal here is to write multithreaded programs that can calculate various statistical values for a list of numbers stored in a file, where a number takes one line in the file. The programs will be passed a file name on the command line and will then use two different strategies.

In the first strategy, we create three worker threads. One thread will determine the sum of the numbers, the second will determine the maximum value, and the third will determine the minimum value. Please name the source code as prob_3_1.c and compile it to prob_3_1

In the second strategy, we again create three worker threads, but each thread now determine the sum, maximum, and minimum values of roughly one third of numbers stored in the file. After all worker threads finish, the main thread can determine the sum, maximum, and minimum values of all numbers stored in the file based on the results from the worker threads. In this strategy, we want to give the three worker threads a similar amount of workload to keep them balanced. You need to decide how you would like to divide the workload. Please name the source code as prob_3_2.c and compile it to prob_3_2

For both strategies, the main thread should read the file and parse it, create worker threads, wait them to finish, and finally print out the sum, maximum, and minimum values before finishing. The programs take only one parameter, i.e., the name of a file storing numbers.

Since they are multithreaded programs, you need to be careful while testing and debugging. Make sure you reason about all possible interleavings and use necessary synchronization operations in your code.

As always, if your code catch certain error conditions, print meaningful error messages in your own format. After you finish, turn in your source files prob_3_1.c and prob_3_2.c, Makefile, and README.