CSCI 3323 (Principles of Operating Systems), Fall 2020

Homework 2b

Credit: 25 points.

1 Reading

Be sure you have read, or at least skimmed, Chapter 2, sections 2.1 through 2.3.

2 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. Submit your program source (and any other needed files) by sending mail to my TMail address with each file as an attachment. Please use a subject line that mentions the course and the assignment (e.g., "csci 3323 hw 2b" or "O/S hw 2b"). You can develop your programs on any system that provides the needed functionality, but I will test them on one of the department's Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (25 points) The starting point for this problem is a simple implementation of the mutual exclusion problem in C with POSIX threads m-e-problem.c. Each thread executes a loop similar to the one presented in class for this problem, except that:

   - Rather than looping forever, each thread makes a finite number of trips through the loop.
   - The critical region is represented by code to print some messages and sleep for a random interval.
   - The non-critical region is represented by code to sleep for a random interval.

   Currently no attempt is made to ensure that only one thread at a time is in its critical region, and if you run it you will see that in fact it can happen that more thaan one thread is in its critical region at the same time. Your mission is to correct this.

   Start by compiling the program, running it, and observing its behavior. To compile with `gcc`, you will need the extra flag `-pthread` and also `-std=c99`, e.g.,

   ```
   gcc -Wall -std=c99 -pthread m-e-problem.c
   ```

   (Or download this Makefile and type `make m-e-problem`.) The program requires several command-line arguments, described in comments at the top of the code. (If you have trouble remembering the order, note that the program prints a meant-to-be-helpful usage message if run with no arguments.) Note that the program may not exhibit the bad behavior for very short delay times but should for longer times.

   You are to produce two corrected versions of this program:

   - The first version should use shared variables only and one of the following algorithms:
     - Strict alternation, extended to work for an arbitrary number of threads. (No, this isn't a perfect solution, but it does enforce the "one at a time" condition.)

  - Peterson's algorithm, for two threads only. For extra credit, research and implement a variation that works for more than two threads. Cite a source for your solution if appropriate — e.g., "I found pseudocode for this solution at the following Web site." Or look up and implement Leslie Lamport's bakery algorithm.
- The second version should use one of the following sets of library functions:
  - The POSIX threads mutex functions. `man pthread_mutex_init` is a good starting point for finding out about these functions.
  - The POSIX threads semaphore functions. `man sem_init` is a good starting point for finding out about these functions.

Places in the program that should change are marked with "TODO" comments. You should not need to add much code. Confirm that your two improved versions behave as expected, i.e., when one thread starts its critical region no other thread can start *its* critical region until the first one finishes. Also be sure to correct the comments at the start of the code — the ones that say the code has no synchronization!

NOTE about shared variables: Optimizing compilers play a lot of tricks to reduce actual accesses to memory, and processors cache values for the same reason. What this means for multithreaded programs is that it is very difficult to guarantee that changes made to a shared variable in one thread are visible to other threads. Declaring shared variables `volatile` avoids at least some compile-time optimizations but does not provide any guarantees about what will happen at runtime, especially if there are multiple processors. What is needed is a "memory fence", which is a way of specifying that at a particular point in the program all memory reads and writes have completed. There is no portable way to achieve this in C99; one must fall back on compiler- or processor-specific code. The starter code includes a function `memory_fence` that invokes a `gcc`-specific function providing a memory fence and recommends its use in the functions to begin and end the critical region. Note that library functions for synchronization (e.g., the ones included with POSIX threads) usually incorporate this functionality.

3 Pledge

Include the Honor Code pledge or just the word "pledged", plus *at least one of the following* about collaboration and help (as many as apply).[1] Text *in italics* is explanatory or something for you to fill in. For programming assignments, this should go in the body of the e-mail or in a plain-text file `pledge.txt` (no word-processor files please).

- This assignment is entirely my own work. *(Here, "entirely my own work" means that it's your own work except for anything you got from the assignment itself — some programming assignments include "starter code", for example — or from the course Web site. In particular, for programming assignments you can copy freely from anything on the "sample programs page".)*
- I worked with *names of other students* on this assignment.
- I got help with this assignment from *source of help — ACM tutoring, another student in the course, the instructor, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*

[1] Credit where credit is due: I based the wording of this list on a posting to a SIGCSE mailing list. SIGCSE is the ACM's Special Interest Group on CS Education.

- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*
- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

4 Essay

Include a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what if anything you think you learned from the assignment, and what if anything you found found interesting, difficult, or otherwise noteworthy. For programming assignments, it should go in the body of the e-mail or in a plain-text file `essay.txt` (no word-processor files please).