CSCI 3323 (Principles of Operating Systems), Fall 2017

Homework 2

Credit: 55 points.

1 Reading

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

2 Honor Code Statement

Please include with each part of the assignment the Honor Code pledge or just the word "pledged", plus one or more 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 written assignments, it should go right after your name and the assignment number; for programming assignments, it should go in comments at the start of your program(s).

- 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.)*
- 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.)*

3 Problems

Answer the following questions. You may write out your answers by hand or using a word processor or other program, but please submit hard copy, either in class or in one of my mailboxes (outside my office or in the ASO).

1. (10 points) Consider two systems:

[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.

- System A supports processes only (no threads).
- System B supports both processes and threads, with processes containing threads, such that each process has at least one thread, and threads cannot exist outside an "enclosing" process.

If you were designing data structures for a process table for System A and process and thread tables for System B, for each of the following say whether you would include it in the process table for System A, the process table for System B, the threads table for System B, or some combination of those (e.g., both process tables but not the threads table for B). Also briefly explain way.

- A place to save CPU registers.
- A place to save information about what memory is owned by the process or thread.

2. (5 points) When a computer is being designed, it is common to first simulate it using a program that runs one (simulated) instruction at a time. Even computers with more than one processor are simulated strictly sequentially like this. Is it possible for a race condition to occur when, as in this situation, there are no truly simultaneous events? Why or why not?

3. (10 points) In class we discussed a proposed solution to the mutual-exclusion problem based on disabling interrupts, and rejected it because it doesn't work for systems with more than one CPU. For a system with a single CPU, however, this could be an acceptable solution, especially if the critical region is short. Write pseudocode for an implementation of semaphores for a single-CPU system that might not have a TSL instruction but does have library functions `enable_int()` and `disable_int()` to enable and disable interrupts respectively. (I.e., say what variables you would need for each semaphore, and give pseudocode for `up()` and `down()`.)

4. (10 points) Restrooms are usually designated as men-only or women-only, but this requires having two restrooms if everyone is to be accommodated. A less expensive approach consistent with cultural norms in the U.S. would be to have one restroom with a sign on the door that indicates its current state — empty, in use by at least one woman, or in use by at least one man. If it is empty, either a man or a women may enter; if it is occupied, a person of the same sex may enter, but a person of the opposite sex must wait until it is empty. Write pseudocode for four functions to implement this approach: `woman_enter`, `man_enter`, `woman_leave`, and `man_leave`, to be used by the following pseudocode:

```
/* woman process */
while (TRUE) {
    woman_enter();
    use_restroom();
    woman_leave();
    do_other_stuff();
}
/* man process */
while (TRUE) {
    man_enter();
    use_restroom();
    man_leave();
    do_other_stuff();
}
```

You can use any of the synchronization mechanisms we have talked about (shared variables, semaphores, monitors, or even message passing). *Hint:* The key issue in solving this problem is making processes interact in the desired way. Shared variables are helpful, but unlikely to work as desired unless you combine them with one of the synchronization mechanisms we discussed.

4 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 `bmassing@cs.trinity.edu` with each file as an attachment. Please use a subject line that mentions the course and the assignment (e.g., "csci 3323 hw 2" or "O/S hw 2"). 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.

(For this assignment, "system that provides the needed functionality" means something UNIX-like, and really it's probably best just to use the department's machines since the recommended approach to provide "memory fences" doesn't even try to be portable.)

1. (20 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[2]. 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 frequently happens that all the threads are in their 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[3] 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, notice that the program prints a meant-to-be-helpful usage message if run with no arguments.)

   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.)

[2] `http://www.cs.trinity.edu/~bmassing/Classes/CS3323_2017fall/Homeworks/HW02/Problems/m-e-problem.c`

[3] `http://www.cs.trinity.edu/~bmassing/Classes/CS3323_2017fall/Homeworks/HW02/Problems/Makefile`

  - 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, as do most processors. 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. For the latter, what is needed is a "memory fence", i.e., a way of specifying that at a particular point in the program all memory reads and writes have completed. As far as I know 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. (*Disclaimer:* At one time the version of this function present on our classroom/lab machines apparently did nothing! This may be a bug in `gcc`, and whether is has been fixed I do not know. My sample solutions seem to work correctly anyway. If your code seems correct to you but does not work, please ask for help.) Note that some library functions for synchronization (e.g., the ones included with POSIX threads) incorporate this functionality.

2. (Optional — up to 10 extra-credit points) Write a program to test your solution to problem 4.

   If you want to do this using C and POSIX threads, you could start with the code for the programming problem above. Note that the POSIX threads library also contains functions to define and work with condition variables. `man -k pthread_cond` will give you a list of relevant `man` pages. (I haven't tried these functions myself, but they look to me like together with locks they allow you to implement the "monitor" abstraction.)

   You could also write in Java and use its `Thread` class, synchronized methods/blocks, and methods `wait`, `notify`, and `notifyAll`. You could also use `Thread` together with `Lock` and `Condition` from `java.util.concurrent`. (`java.util.concurrent` provides a pretty rich collection of higher-level constructs, but I think for this class you might learn more by not using them. But `Lock` and `Condition` allow you to implement the full "monitor" abstraction, as base Java doesn't really.) Examples on request.

   Or you could write in Scala and use these same Java classes and methods. There will shortly be some quick crude examples linked from the "useful links and information" page.