CPSC 213: Assignment 9

Due: Wednesday, November 26, 2014 at 11:59pm

After an eight-hour grace period, no late assignments accepted.

Goal

The goal of this assignment is to give you some experience writing concurrent programs. Writing concurrent code that works correctly is hard. Debugging concurrent code that doesn't work correctly is hard too. These skills are becoming increasing important. This assignment introduces you to this set of challenges by having you solve three simple concurrent programming problems.

Groups

You may do this assignment in groups of two if you like. If you do, be sure that you both contribute equally to the assignment and that you each work on every part of the assignment. Do ***not*** split up the assignment in such a way that one of you does one part and the other does another part. Keep in mind that the core benefit to you of doing the assignment is the learning that happens while you do it. Each assignment is worth only around 1.8% of your grade, but what you learn while doing the assignment goes a long way to determining the other 85%.

If you choose to do the entire assignment with a partner, submit one solution via **`handin`** and list both of your names, student numbers, and computer-science login ids in the **`README`** file and include an addition file called **`PARTNER.txt`** that contains the id of your partner (i.e., the student that did not submit the assignment).

Alternatively, you can also choose to collaborate with another student for a portion of the assignment and have your work marked individually. Do do this, simply submit your own version via **`handin`** and list the other student as a collaborator in your **`README`** file. Just don't do this if you and your partner are turning in identical solutions, as we would like to realize marking efficiency where possible. You may have at most one collaborator for the assignment; i.e., you can not collaborate with one student for one part of the assignment and another student for another part.

Writing Concurrent Code

You will solve two additional *toy* concurrent problems using *uthreads* and its implementation of mutexes and condition variables and one problem using its implementation of semaphores. The first of the problem is well known, canonical problem and the second is a bit less well know and comes from a wonderful free book called *The Little Book of Semaphores* by Allen B Downey.

You can download the book if you like, but it is not necessary (and probably not that helpful) for this assignment. The third problem is the canonical producer-consumer problem from Assignment 8. While these problems are toys, they were designed to model specific types of real-world synchronization problems that show up real concurrent systems such as operating systems.

In each case your program will consist of a solution to the concurrency puzzle and a test harness that creates a set of threads to exercise your code and instruments your code to collect information that you can use to convince yourself (and us) that you have implemented the problem correctly. Bugs will either be in the form of incorrect results (i.e., violating the stated constraints) or deadlock (i.e., your program hangs). The problems are kept as simple as possible.

Problem 1: *The Cigarette Smokers Problem*

The cigarette smokers problem is a classic synchronization problem, posed by Suhas Patil in 1971. In this problem there are four actors, each represented by a thread, and three resources required to construct and smoke a cigarette: tobacco, paper, and matches. One of the actors is the *agent* and the other three are *smokers*. The agent has an infinite supply of all of the resources. Each smoker has an infinite supply of one resource and nothing else; each smoker possesses a different resource.

The three smoker threads loop attempting to smoke, which requires that they obtain one unit of both of the resources they do not possess. The agent loops repeatedly, randomly choosing two ingredients to make available to smokers. Each time the agent does this, one of the three smokers should be able to achieve its heath-destroying goal. For example, if the agent chose paper and matches, then the tobacco-possessing smoker can consume these two items, combined with its own supply of tobacco, to smoke.

This is a simple model of a general resource-management problem that operating systems deal with in many forms. To ensure that it captures that real problem correctly, the agent has a few additional constraints placed on it.

The agent is only allowed to communicate by signalling the availability of a resource. It is not permitted to disclose resource availability in any other way; i.e., smokers can not ask the agent what is available. In addition, the agent is not permitted to know anything about the resource needs of smokers; i.e., the agent can not wakeup a smoker directly. Finally, each time the agent makes two resources available, it must wait for one smoker to smoke before it can make any additional resources available.

The problem is tricky because when the agent makes two items available, every smoker thread can use one of them, but only one can use both. If you aren't careful, you might create a solution that results in deadlock. For example, if the agent makes paper and matches available, both the *paper* and the *matches* smokers want one of these, but neither will be able to smoke because neither has tobacco. But, if either of them does wake up and consume a resource, that will prevent the tobacco thread from begin able to smoke and thus also prevent the agent from

waking up to deliver additional resources. If this happens, the system is deadlocked; no thread will be able to make further progress.

Requirements

Implement a deadlock-free solution to the cigarette smokers problem in a C program called **`smoke.c`**. Use *uthreads* initialized to use a single processor (or more if you like).

Create four threads: one for the agent and one for each type of smoker. The agent thread should loop through a set of iterations. In each iteration it chooses two resources randomly, signals their condition variables, and then waits on a condition variable that smokers signal when they are able to smoke. When smoker threads are unable to run they must be waiting on a condition variable. When a smoker wakes up to find both of the resources it needs, it signals the agent and goes back to waiting for the next chance to smoke.

The agent must use exactly four condition variables: one for each resource and one to wait for smokers. The agent must indicate that a resource is available by calling **`signal`** on that resource's condition variables exactly once. There is no other way for any other part of the system to know which resources are currently available.

You may find it useful to create other threads and add additional condition variables. It is perfectly fine to do so as long as you follow the constraints imposed on the agent thread. For example, notice that we have not said *how* the smokers wait other than to say that they wait on some condition variable. This is a hint.

To generate a random number in C you can use the procedure **`random()`** that is declared in **`<stdlib.h>`**. It gives you a random integer. You if want a random number between **`0`** and **`N`**, one way to do that is to use the modulus operator; i.e., **`random() % N`**. This procedure returns random numbers starting of a seed value. Every time you run your program it will by default use the same seed and so calls to **`random()`** will produce the same sequence of random numbers. That is fine.

But, if you want a completely different set of random numbers every time you run the program, you need to use set a random seed before calling **`random()`**. To do so, add the following to the top of your program.

```
#include <fcntl.h>
#include <unistd.h>
```

Declare this procedure:

```
void mysrandomdev() {
  unsigned long seed;
  int f = open ("/dev/random", O_RDONLY);
  read    (f, &seed, sizeof (seed));
  close   (f);
  srandom (seed);
}
```

And call **mysrandomdev()** from **main()** when your program starts, before calling **random()**.

Testing

The most common problem with attempts to solve this problem is deadlock. The simplest way to diagnose this problem initially is to use **printf** statements in the agent and smokers that tell you what each is doing. A **printf** just before and just after every statement that could block (e.g., every **wait**) is probably a good idea. If the printing stops before the program does, you have a deadlock and the last few strings printed should tell you where. Start with one iteration of the agent. Get that to work, then try more than one.

Once you think you've got this working, you'll want to turn off the **printf**'s so that you can drive the problem through a large number of iterations without being bombarded with output. One way to do this is to use the *C Preprocessor* to surround each of your **printf** statements with a **#ifdef** directive like this:

```
#ifdef VERBOSE
    printf ("Tobacco smoker is smoking.\n");
#endif
```

A better way — though the more you do with macros the trickier it can get — is to define a macro called **VERBOSE_PRINT** that is printf if **VERBOSE** is defined and the empty statement otherwise. To do this, include the following macro definition at the beginning of your program.

```
#ifdef VERBOSE
#define VERBOSE_PRINT(S, ...) printf (S, ##__VA_ARGS__);
#else
#define VERBOSE_PRINT(S, ...) ;
#endif
```

And then use the macro instead of **printf** for debugging statements, like this:

```
VERBOSE_PRINT ("Tobacco smoker is smoking.\n");
```

In either case you can now selectively define the **VERBOSE** macro when you compile your program to turn diagnostic **printf**'s on or off.

To turn them on:

```
gcc -D VERBOSE -o smoke smoke.c uthread.c -pthread
```

To turn them off:

```
gcc -o smoke some.c uthread.c -pthread
```

Test your program by driving the agent through a large set of iterations. Instrument the agent to count the expected times each smokers should smoke and instrument each smoker to count the number of times that each does smoke. Compare these to ensure they match and print them when the program terminates.

Problem 2: *The Unisex Washroom Problem*

Overview

This is an interesting, but less classical problem. It is a bit of a generalization of the reader-writer problem discussed in class. In this problem a particularly cheap company is responding to employee complaints by installing a new washroom. But the company is cheap and so is just installing one to be shared by both male and female employees. And its even cheaper in that it does not want any more than three of its employees to be able to use the washroom at once, in the belief that its employees are up to no good and that allowing more than three in the washroom at once will lower productivity or lead to general unrest.

You are to implement the washroom gatekeeper that decides who is allowed into the washroom. The gate keeper maintains the following two constraints: (1) no more than three people are allowed in the washroom at once and (2) all of the people in the washroom must be of the same sex. The gatekeeper is otherwise fair and ensures that people waiting to use the washroom eventually get to do so, provided that people actually leave the washroom on regular basis. It is also efficient, ensuring that the washroom is at maximum capacity as often as possible when people are waiting, but trading off in a reasonable way with the fairness and eventual-entry constraints. It does not, however, ensure that people always enter the washroom in the order they start waiting.

Each person is represented by a thread. The washroom is a critical section protected by a mutex. The gatekeeper is a procedure that each thread runs when attempting to enter the washroom and when leaving it. When a thread is unable to enter the washroom it waits on a condition variable. When a thread leaves the washroom it delivers whatever signals are necessary to wakeup the thread or threads that can enter the washroom when it leaves.

Requirements

Implement a solution to the unisex washroom problem in a C program called **`washroom.c`**. Use *uthreads* initialized to use a single processor (or more if you like). Use mutexes for mutual exclusion and condition variable for thread signalling.

Create **`N`** threads and assign each a randomly chosen sex. Threads should loop attempting to enter the washroom a large, fixed number of times. When a thread is in the washroom, it should call **`uthread_yield()`** a total of **`N`** times and then exit the washroom. It should then call **`uthread_yield()`** at least another **`N`** times before attempting to enter the washroom again. The program terminates when every thread has entered the washroom the specified number of times. Experiment with different values of **`N`**, starting with small numbers while you debugging and ending with a number that is at least twenty.

Testing

Test your program with **`N=20`** and each thread performing a least **`100`** iterations to ensure that the two washroom-occupancy constraints are never violated using an **`assert`** statement. Count the

number of times that each of the following occupancy condition variables occur: one male, two males, three males, one female, two females, and three females. Print these numbers when the program terminates.

Implement a counter that is incremented each time a thread enters the washroom. For each thread entering the washroom, record the value of the counter when it starts waiting and the value when it enters the washroom. Subtract these two numbers to determine the thread's waiting time and record this information in a histogram like this.

```
if (waitingTime < WAITING_HISTOGRAM_SIZE)
   waitingHistogram [waitingTime] ++;
else
   waitingHistogramOverflow ++;
```

Declare a large histogram array of size **`WAITING_HISTOGRAM_SIZE`**. Print the histogram and the overflow bucket when the program terminates. If you access the histogram or other test data from multiple threads be sure to guarantee mutual exclusion for critical sections.

You will notice that no matter how hard you try to make this fair, if you have enough people trying to get into the washroom at the same time, you can't make it fair for everyone. You will see that people occasionally end up waiting much longer than it seems they should. The problem is that there is an inherent unfairness with **`wait`**. To see why, lets assume there is a long queue of people waiting on a condition variable. When **`signal`** is called indicating that a washroom position is available, the thread that has been waiting the longest is awoken. This is fair and is ensured by the fact that the condition-variable waiter queue is a fifo. **However**, if there some other thread that has not been waiting at all is, at this very moment, trying to get into the critical section and it beats the awoken thread into the critical section, then it may get that thread's position in the washroom, bypassing the awoken thread and every thread on the waiter queue. When this happens the awoken thread must **`wait`** again; and it does this by moving all the way to the back of the waiter queue. In our case the budger is the thread that just left the washroom and that just turns around and tries to get back in again, sometimes succeeding to budge to the front of the line, grabbing the space it just vacated and forcing that poor sucker it just woke up to go to the back of the line. The purpose of the **`uthread_yield()`** loop after exiting the bathroom is to minimize how often this situation occurs. You won't see it happen often. But, it will happen often enough that a few threads occasionally end up waiting a very long time to get into the washroom. You might experiment will calling **`uthread_yield()`** more times after leaving the washroom (or less) and see how this affects fairness. Resolving this unfairness is tricky and not necessary for this assignment.

Problem 3: *Producer Consumer with Semaphores*

Implement a new version of the producer-consumer problem from Assignment 8, but this time using semaphores for synchronization instead of mutexes and condition variables or spinlocks. The only synchronization primitives are you permitted to use are **`uthread_sem_wait`** and **`uthread_sem_signal`**. Place your implementation in a file called **`pc_sem.c`**.

Bonus

For bonus credit, re-implement one or more of the two concurrency problems using your semaphores instead of mutexes and condition variables; i.e, the only synchronization primitives you can use are **`uthread_sem_wait`** and **`uthread_sem_signal`**. If you take this on, you might want to consult the *Little Book of Semaphores* for help. Place these implementation(s) in files called **`smoke_sem.c`** and/or **`washroom_sem.c`**.

Provided Code

The code provided for this assignment is available in a zip file at the url

`www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a9/code.zip`

It contains the following files. The only change from Assignment 8 is the addition of **`uthread_sem`**.

- **`uthread.[ch]`**
- **`uthread_mutex_cond.[ch]`**
- **`uthread_sem.[ch]`**

Requirements

1. Implement and test **`smoke.c`**.
2. Implement and test **`washroom.c`**.
3. Implement and test **`pc_sem.c`**.
4. For bonus marks reimplement one or more of **`smoke_sem.c`** or **`washroom_sem.c`**.

What to Hand In

Use the **`handin`** program. The assignment directory is **a9**

1. A single file called "**`README.txt`**" that includes your name, student number, four- or five-digit cs-department undergraduate id (e.g., the one that's something like **`a0b1`**), and all written material required by the assignment as listed below.
2. If you had a partner for the entire assignment, turn in only one copy of your joint solution under one of your ids and list both student's complete information in the **`README.txt`** file and include the text "**`PARTNER – MARK JOINTLY`**". Also include a file named **`PARTNER.txt`** that contains the cs-department id of your partner and nothing else.
3. If, on the other hand, you collaborated with another student for a portion of the assignment, but your solutions are not identical and you want them marked individually, each of you should include the other student's complete information in your **`README.txt`** file, include the text "**`COLLABORATOR – MARK SEPARATELY`**", and turn in copies separately via **`handin`**.
4. Your implementations of **`smoke.c`**, **`washroom.c`**, and **`pc_sem.c`**

5. Sample output from test runs of each program including the specified instrumentation values. Include this information in **`README.txt`**.
6. And, if you did the bonus your semaphore versions of the concurrent programs in files named **`smoke_sem.c`** and **`washroom_sem.c`**.