CPSC 213 – Assignment 10
Synchronization

Due: Friday, April 7, 2016 at 11:59PM
After a 72-hour grace period (i.e., Monday at 11:59PM) 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.

What to Do

You will solve three well-know concurrency problems using uthreads, mutexes and condition variables and then solve one (and another for Bonus) using semaphores.

The first problem you'll tackle is the classic Producer Consumer problem; you'll solve this problem two ways: first using spinlocks and polling and then using blocking locks. The second is a well known, problem and the third 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. 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.

The code you need for this assignment is in *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a10/code.zip*. There you will find the complete uthread package, including the implementation of monitors and condition variables and semaphores. You will also find a `Makefile` that you will use to build the various parts of this assignment and skeleton files for each of the questions with some parts implemented for you and some parts, listed as `TODO`, left for you.

Question 1: The Producer-Consumer Problem (Spinlocks) [10%]

General Description of the Problem

The producer-consumer problem is a classic. This problem uses a set of threads that add and remove things from a shared, bounded-size resource pool. Some threads are producers that add items to the pool and some are consumers that remove items from the pool.

Video streaming applications, for example, typically consist of two processes connected by a shared buffer. The producer fetches video frames from a file or the network, decodes them and adds them to the buffer. The consumer fetches the decoded frames from the buffer at a designated rate (e.g., 60 frames per second) and delivers them to the graphics system to be displayed. The buffer is needed because these two processes do not necessarily run at the same rate. The producer will sometimes be fast and sometimes slow (depending on network performance or video-scene complexity). On average it is faster than the consumer, but sometimes its slower.

There are two synchronization issues. First, the resource pool is a shared resource access by multiple threads and thus the producer and consumer code that accesses it are critical sections. Synchronization is needed to ensure mutual exclusion for these critical sections.

The second type of synchronization is between producers and consumers. The resource pool has finite size and so producers must sometimes wait for a consumer to free space in the pool, before adding new items. Similarly, consumers may sometimes find the pool empty and thus have to wait for producers to replenish the pool.

What to do

The provided file **`pc_spinlock.c`** contains a very simple outline of the problem. Modify this file to add threads and synchronization according to the following requirements.

Your solution must use at least four threads (two producers and two consumers). Use *uthreads* initialized to four processors (i.e., `uthread_init(4)`).

To keep things simple, the shared resource pool is just a single integer called `items`. Set the initial value of `items` to `0`. To add an item to the pool, increment `items` by `1`. To remove an item, decrement `items` by `1`.

Producer threads should loop, repeatedly attempting to add items to the pool one at a time and consumer threads should loop removing them, one at a time. Ensure that each of these add-one or remove-one operations can interleave arbitrarily when the program executes.

Use *spinlocks* to guarantee mutual exclusion. To use a spinlock, you must first allocate and initialize (i.e., create) one:

```
    spinlock_t lock;
```

```
spinlock_create (&lock);
```

Then you lock and unlock like this:

```
spinlock_lock (&lock);
…
spinlock_unlock (&lock);
```

Your code must ensure that `items` is never less than `0` nor more than `MAX_ITEMS` (which you can set to `10`). Consumers may have to wait until there is an item to consume and producers may have to wait until there is room for a new item. In both cases, implement this waiting by ***spinning*** on a read of the `items` variable; consumers waiting for it to be non-zero and producers waiting for it to be less than `MAX_ITEMS`. Be sure not to spin while holding the spinlock, because doing so will cause a deadlock (i.e., your program will hang). And be sure to double-check the value of `items` once you do hold the spinlock to handle a possible race condition between two consumers or two producers. This code will look similar to the final spinlock implementation shown in class, though in C:

```
      ld   $lock, r1   # r1 = &lock
loop: ld   (r1), r0    # r0 = lock
      beq  r0, try     # goto try if lock==0 (available)
      br   loop        # goto loop if lock!=0 (held)
try:  ld   $1, r0      # r0 = 1
      xchg (r1), r0    # atomically swap r0 and lock
      beq  r0, held    # goto held lock was 0 before swap
      br   loop        # try again if another thread holds lock
held:                  # we now hold the lock
```

First spin on the condition without hold the lock, then acquire the lock and re-check the condition. If the condition no longer holds, release the lock and go back to the first spinning step.

Use the included *makefile* to compile your program.

Testing

To test your solution, run a large number of iterations of each thread. Add `assert` statement(s) to ensure that the constraint `0 <= items <= MAX_ITEMS` is never violated. Count the number of times that producer or consumer threads have to wait by using two global variables called `producer_wait_count` and `consumer_wait_count`. In addition, maintain a histogram of the values that `items` takes on and print it out at the end of the program. Use the histogram to ensure that the total number of changes to `items` is correct (i.e., equal to the total number of iterations of consumers and producers). Print the values of the counters and the histogram when the program terminates. The histogram would look something like this.

```
int histogram [MAX_ITEMS + 1];
…
histogram [items] ++;  // do this when items changes value
```

Ensure that your program prints these values when it terminates exactly as is done in the provided code.

Since concurrency bugs are non-deterministic — they only show up some of the time — be sure to run your program several times. This repeated execution is particularly important to ensure that your program is deadlock-free.

Question 2: Blocking Producer Consumer [10%]

Make a copy of `pc_spinlock.c` and call it `pc_mutex_cond.c`. Modify this file to replace spinlocks with blocking mutexes and spinning with condition variables so that all waiting is now blocking waiting instead of busy waiting.

Test your program the same way you tested `pc_spinlock.c` in Question 1.

Question 3: The Cigarette Smokers Problem [30%]

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 using a condition variable. 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 on a condition variable for a 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`; start from the provided file. 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. The code for the agent is provided for you. You do not need to change this code, but you can. Just be sure you follow the rules we have just outlined.

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.

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.

Be sure that the strings you print with `printf` end with a new line character (i.e., “`\n`”), because `printf` does not actually print until it sees this character or the program terminates. If you print without the newline and then your program deadlocks, you will not see the string printed and you will be confused about where the program deadlocked.

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 -std=gnu11 -o smoke smoke.c uthread.c uthread_mutex_cond.c -pthread
```

To turn them off:

```
gcc -std=gnu11 -o smoke smoke.c uthread.c uthread_mutex_cond.c -pthread
```

Testing

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.

Question 4: The Unisex Washroom Problem [30%]

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 it is just installing one to be shared by all 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 have the same gender identity (we will simplify to two genders: male and female, while acknowledging that the real world is a bit more diverse).

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`; start from the provided file. 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 gender identity. 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. This is the same problem we've seen in class: a race between the awoken thread re-entering the critical section when returning from wait and a new thread calling lock to enter.

To see what is happening in this case, 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 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 turned around and tried 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 with 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.

Question 5: Producer Consumer with Semaphores [20%]

Re-implement Question 2 using semaphores. Put your solution in the file called `pc_sem.c`; start from the provided file.

The only synchronization primitives are you permitted to use are `uthread_sem_wait` and `uthread_sem_signal`. Use semaphores to replace both the mutexes and condition variables.

Bonus: Unisex Washroom with Semaphores [20%]

Re-implement Question 4 using semaphores as the only synchronization primitive and place your solution in the file `washroom_sem.c`; start from the provided file.

Notice what happens to the unfairness problem we saw with `wait` in Question 4 that caused threads to occasionally lose their place in line and end up waiting a very long time to get into the washroom. Explain the difference you see and say why semaphores are different, placing your answer in the file `BONUS.txt`.

What to Hand In

Use the `handin` program.

The assignment directory is `~/cs213/a10`, it should contain the following ***plain-text*** files.

1. `README.txt` that contains the name and student number of you and your partner
2. `PARTNER.txt` containing your partner's CS login id and nothing else (i.e., the 4- or 5-digit id in the form a0z1). Your partner should not submit anything.
3. For Question 1: `pc_spinlock.c`.
4. For Question 2: `pc_mutex_cond.c`.
5. For Question 3: `smoke.c`.
6. For Question 4: `washroom.c`.
7. For Question 5: `pc_sem.c`.
8. For the Bonus Question, if you did it: `washroom_sem.c` and `BONUS.txt`.