CPSC 213 – Assignment 10
Synchronization

Due: Part 1: Wednesday, August 7, 2019 at 11:59pm
You may use one of your two late days on this part to make it due Thursday.

Part 2: Friday, August 9, 2019 at 11:59pm
You may hand in this assignment up to **72 hours late** (*i.e.*, until **Monday, August 12, 11:59pm**) without any penalty and without using up a late day.

Goal

The goal of this assignment is to give you experience writing concurrent programs. Writing concurrent code that works correctly is hard. Debugging concurrent code that doesn't work correctly is much harder. These skills are becoming increasingly important. This assignment introduces you to this set of challenges by having you first solve a number of small, isolated concurrency problems, and then scaling up to larger concurrent programming problems.

What to Do

In Part 1, you will solve some simple, isolated concurrency problems and one larger problem, the classic bounded-buffer, producer-consumer problem.

In Part 2, you will solve two additional, well-known concurrency problems using *threads*, *mutexes* and *condition variables* and then solve the bounded-buffer problem from Part 1 (and the others from this part for Bonus) using semaphores. The first is a well known problem and the second is our variation of a somewhat less well-known problem from a book called *The Little Book of Semaphores* by Allen B. Downey. You can download the book for free 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 in real concurrent systems such as operating systems, video games, servers, etc.

For each problem, your program will consist of a solution to the concurrency puzzle and a test harness that creates a set of threads 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 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, condition variables and semaphores, as well as a `Makefile` that you will use to build the various parts of this assignment. For each problem on this assignment, you will find skeleton files with some parts implemented for you and some parts, listed as `TODO`, left for you.

NOTE: Synchronization bugs introduce non-determinism into programs, which means that they may produce different results each time they are run, even on the exact same input. This also means that a program with a synchronization bug might execute correctly sometimes, so to be sure that your program is bug free it is necessary to execute it many times. You must take this step when testing your code for this assignment. The Unix command-line shell provides ways to automate this task; you might find it useful to investigate how to do this.

For example, in the default Linux *bash* shell, you can use a C-style *for* loop to repeat a command (note the use of double parentheses):

```
for ((i=0; i<100; i++)); do ./q1; done
```

Hint: Using Valgrind with uthreads

As usual, you should ensure that your code is free of memory errors and memory leaks. However, the uthread library introduces a bit of a wrinkle, as it does not free all of its memory upon exit (as threads might still be running). Consequently, to properly use valgrind to catch leaks, you'll need to tell Valgrind to ignore leaks coming from the uthread library.

In the code package, you'll find a file called `uthread.supp`. This file directs Valgrind to ignore allocations made by the uthread library. To use it, run Valgrind as follows:

```
valgrind --suppressions=uthread.supp --leak-check=full --show-leak-kinds=all ./prog <arguments>
```

Here, we use `--suppressions=uthread.supp` to tell Valgrind to use our suppression file and `--leak-check=full --show-leak-kinds=all` to check for all leaks (including leaks caused by global variables). You are strongly encouraged to check all your programs with Valgrind - it could save you a lot of time debugging (however, note that Valgrind won't catch deadlocks - that's still up to you!)

Part 1

Question 1.1: Simple Thread Ordering with Join [5%]

Modify the program `part1/q1.c` by adding calls to `uthread_join` (and making ***no*** other changes) so that it always prints the lines "`zero`" to "`three`" in order; *i.e.*, its output must ***always*** be the following.

```
zero
one
two
three
------
```

Question 1.2: Simple Mutual Exclusion [5%]

Modify the program `part1/q2.c` to use a mutex so that the program is free of *race conditions* (*i.e.*, so that you have guaranteed *mutual exclusion* for all *critical sections* in the code) and thus that it ***always*** prints "`counter = 0 OK!`" when it terminates.

Question 1.3: Simple Thread Rendezvous at a Barrier [5%]

Modify the program `part1/q3.c` to use a *mutex* and zero or more *condition variables* so that every thread prints "`a`" before any thread prints "`b`". This type of synchronization is called a *barrier* and we say that threads all *rendezvous* at the *barrier* before any are allowed to continue past it. Your solution must work for any value of `num_threads` (specified as an argument to the program). For example, for 3 threads your program must ***always*** output the following.

```
a
a
a
b
b
b
------
```

Question 1.4: Thread Ordering with Condition Variables [5%]

Modify the program `part1/q4.c` to use a *mutex* and one or more *condition variables* to order the threads to produce the same output as your solution to Question 1.1. You solution must use condition variables to order the threads and it must never deadlock. A correct solution must consider the issue illustrated by Question 1.3 and also the constraints on the ordering of calls to *signal* and *wait.*

Question 1.5: Producer-Consumer [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.

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 it's slower.

There are two synchronization issues. First, the resource pool is a shared resource accessed 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

Examine the program `part1/pc.c`. To keep things simple, the shared resource pool in this program 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`.

Modify the program this program to use a mutex and condition variable(s) so that the program is race free and that it satisfies the constraint that *0 <= items <= max_items.*

Your code must ensure that `items` is never less than `0` nor more than `max_items`. *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 using *condition variables*.

Test your solution with two consumer and two producer threads and the number of processors set to 4; this is already done for you in `pc.c`.

Testing

To test your solution, run a large number of iterations of each thread. There are `assert` statement(s) in the code that ensure the constraint `0 <= items <= max_items` is never violated; if it is the program will crash. You should test with a range of different numbers of max_items, producers and consumers using different command-line arguments. For some possible values, even a correctly-implemented version of the code will deadlock - can you figure out why?

The program also maintains a histogram of the values that `items` takes on that prints when the program terminates. The program prints the histogram when it terminates to give you a view into what happened during the execution. A correct implementation should have a sort of random distribution across the values of items, weighted a bit to lower values, and it will be different each time you run your program. The sum of the counts in the histogram must equal the total number of changes made to `items` (i.e., the total number of producer and consumer steps); the final assert statement checks this constraint.

As with every question in this assignment, run your solution many times to ensure that it completes correctly each time.

Part 2

Question 2.1: 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 health-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, `part2/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. 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.

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 with 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 remove 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 a 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 (the program now prints all of the debugging statements)):

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

To turn them off (the program now prints no debugging statements):

```
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 2.2: The Lilliputian Endianness Well Problem [30%]

As has been documented, a great schism befell the land of Lilliput some generations ago when, following an egg-eating tragedy involving the emperor's son, it was decreed that all eggs be eaten small-end first from that time forth — the big end being a far too dangerous place to commence egg ingestion. But, sadly, as often happens with imperial decrees, some in the kingdom remained steadfastly committed to the traditional big-end-first manner of eating and as time passed the endianness schism grew deeper and the risk of conflict more profound.

In the present day, a peace-keeping arrangement has been established that separates Lilliput into two endianness halves so that the people never risk meeting someone with a different point of view on the whole egg thing. There is but one problem. Lilliput has only a single well from which all souls must get their drinking water. A protocol has therefore been established to control access to the well. It being important to understand the protocol to complete this question, I will tell it to you now.

There exists a gatekeeper who controls access to the area surrounding the well. The gatekeeper grants access in such a way as to ensure two things. First, it must never be the case that people of opposing endianness are at the well at the same time, to avoid the untold calamity that would surely unfold should either side be confronted with views they don't share (there is a separate protocol for social media, but that is of no importance to the gatekeeper). Second, no more than three people can be in the well area at the same time (a fire-code thing).

The gatekeeper, being noble and fair, ensures that people waiting for water eventually get to use the well and that their waiting times are roughly uniform, provided that the people at the well leave on a regular basis. The gatekeeper is also bit of a technocrat and so wants the well to operate efficiently (i.e., at high capacity), even if this means there's a bit of queue jumping (i.e., *sometimes* people can enter the well ahead of someone who has been waiting longer than them). Finally the gatekeeper is wise and has thus devised an exquisite technique to balance the tradeoff between fairness and efficiency.

Now, it has fallen to you to simulate the Lilliputian well using threads, mutexes, and condition variables. Every person is a thread. The well is a critical section protected by a mutex. The gatekeeper is a procedure that each thread calls when attempting to enter the well and when leaving. When a thread is unable to enter the well it waits on a condition variable. When a thread leaves the well it delivers whatever signals are necessary to wakeup the thread or threads that are allowed to enter when it leaves.

Requirements

Implement a solution to the Endianness Well problem in a C program called `well.c`; start from the provided file, `part2/well.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 endianness. Threads should loop attempting to enter the well a large, fixed number of times. When a thread is in the well, it should call `uthread_yield()` a total of `N` times and then exit the well. It should then call `uthread_yield()` at least another `N` times before attempting to enter the well again. The program terminates when every thread has entered the well the specified number of times. Experiment with different values of `N`, starting with small numbers while 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 well-occupancy constraints are never violated using an `assert` statement. Count the number of times that each of the following occupancy conditions occur: one big endian, two big, three big, one little, two little, and three little. Print these numbers when the program terminates.

Implement a counter that is incremented each time a thread enters the well. For each thread entering the well, record the value of the counter when it starts waiting and the value when it enters the well. 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 these 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 well 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, let's assume there is a long queue of people waiting on a condition variable. When `signal` is called indicating that a well position is available, the thread that has been waiting the longest is awoken. This is fair as 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 well, 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 well 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 well 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 well. You might experiment with calling `uthread_yield()` more times after leaving the well (or less) and see how this affects fairness. Resolving this unfairness is tricky and not necessary for this assignment.

Question 2.3: Producer Consumer with Semaphores [10%]

Re-implement Question 1.5 using *semaphores* by modifying `part1/pc.c` to use semaphores for all synchronization. Place your solution in a file named `pc_sem.c`. Your solution must not use anything from the `uthread_mutex` / `uthread_cond` API.

Bonus 2.4: Endianness Well with Semaphores [+20%]

Re-implement Question 2.2 using semaphores as the only synchronization primitive and place your solution in the file `well_sem.c`; start from the provided file (or your solution to Question 4).

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 well. 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.

You will handin parts 1 and two separately.

The assignment directory for Part 1 is `~/cs-213/a10part1`. It should contain the following ***plain-text*** files.

1. (optional) `PARTNER.txt` containing your partner's CWL login id and nothing else. Your partner should not submit anything.

2. The files: `q1.c`, `q2.c`, `q3.c`, `q4.c`, and `pc.c`. These should stored directly in the a10part1 directory and not in a subdirectory.

The assignment directory for Part 2 is `~/cs-213/a10part2`. It should contain the following ***plain-text*** files.

1. (optional) `PARTNER.txt` containing your partner's CWL login id and nothing else. Your partner should not submit anything.
2. The files `smoke.c`, `well.c`, `pc_sem.c` and, if you did the bonus question, `well_sem.c` and `BONUS.txt`.