CPSC 213: Assignment 8

Due: Monday, November 17, 2014 at 11:59pm

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

Goal

This is the first of two assignments that deal with threads and asynchrony. This assignment has three parts. In the first part you continue where you left off in Assignment 7 to write a threaded version the disk read application and to compare its performance to the other two version. Then, you implement a classic synchronization problem — Producer Consumer — using *spinlocks*. Finally, you re-implement the problem using *monitors (aka mutexes) and condition variables* (i.e., *blocking* locks).

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.

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.

Provided Code

Most of the code you need for this assignment was provided with Assignment 7.

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

> **`www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a8/code.zip`**

It contains the following new files:

- **tRead.c**
- **uthread_mutex_cond.[ch]**
- **pc_nosync.c**
- **uthread.[ch], uthread_util.h** (also included with Assignment 7)

Part 1: Threads and Synchronous Reads

Starting from where you left off in Assignment 7, open the new file **tRead.c** in an editor and examine it carefully. Compare this version to both of the other versions on worked on in Assignment 7 (i.e., **sRead.c** and **aRead.c**). The goal in this new version is to read and handle disk blocks sequentially in a manner similar to **sRead** but to do so using threads to get performance similar to **aRead**.

Thus, the reading and handling of a single disk block must be done in the procedure called **readAndHandleBlock**. That is, this procedures must schedule the disk read, stop its thread to wait for the disk, then when awoken, process the read by calling **handleRead**, which of course in this case doesn't do anything interesting, but in a real program would be where you used the data you just got from the disk (e.g., file).

Compile and test your program.

To compile, you can use the Makefile from Assignment 7 by modifying it to add this line:

```
tRead: tRead.o disk.o uthread.o queue.o
```

Or you can compile from the command line (assuming all .c files are in your current working directory) by typing:

```
gcc -o tRead tRead.c disk.c uthread.c queue.c -lpthread
```

Evaluate this version as you did the other two. Compare their performance and record your data and observations in **README.txt**. One additional thing. Compare both the elapsed time (as you have previous) and the system time of **aRead** and **tRead**. If there is a significant difference note it. You are not expected to be able to explain this difference, but try if you like (no marks off for not trying or for getting it wrong). All of this is **QUESTION 1**.

Here are some additional notes to help you with the implementation.

Hints: Creating and Joining with Threads in Run

You should create a separate thread for each call to **readAndHandleBlock** using:

```
uthread_create (void* (*start_proc)(void*), void* start_arg)
```

Note that the start procedure takes only one argument of type **void*** but read takes three arguments. So, you will probably need to create a struct that stores these arguments. One hard thing about using threads (or even-driven programming) is keeping in mind that some things aren't sequential anymore. Every thread can run in parallel. The call to create a thread returns immediately, before the thread it creates starts. And so, if you create an object to store the

arguments for one thread, be sure to create a separate object for every thread you create. A common mistake is to create a single object and to change its value for each call, something like this.

```
int i;
for (i=0; i<4; i++)
    uthread_create (foo, &i);
```

Now, this does call **foo** four times, but you probably wanted the parallel equivalent of this:

```
foo (0);
foo (1);
foo (2);
foo (3);
```

But you got this:

```
foo (4);
foo (4);
foo (4);
foo (4);
```

The explanation of this behaviour is that the argument to **foo** is a pointer to the variable **i** and **i** will have the value of **4** when the loop terminates. Now, I'm also assuming in this example that the procedure that creates the threads also joins with them. If it doesn't and instead returns before the threads have finished, passing a pointer to a local variable in this way creates dangling pointers to the variable in each of the threads, because they now have a pointer to a variable in the stack frame of returned procedure.

The same is true for the buffer (**buf**) into which each disk read copies its data. Be sure to use a separate **buf** for every thread that might run concurrently.

If you dynamically allocate objects with **malloc** you must ensure that you free them explicitly at the right time by calling **free**.

Another thing to note about **run** is that it joins with all of the threads it creates, thus returning only after all of them have finished. This step is necessary to keep the program from terminating before all of its threads have; returning from **main** terminates the program. For **run** to be able to join with these threads, it must have a pointer to each of them. This pointer is the value returned by **uthread_create**.

Hints: Blocking and Unblocking Threads

A thread can block itself at any time by calling **uthread_block**. Another thread can wakeup a blocked thread (**t**) by calling **uthread_unblock(t)**. Recall that you will need to block threads after they call **disk_scheduleRead** and before they call **handleRead**. And that this blocked thread should be awoken when the disk read on which it is waiting has completed.

Part 2: The Producer-Consumer Problem (Spinlocks)

Overview

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 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.

Requirements

The provided file **`pc_nosync.c`** contains a very simple outline of the problem. Create a new file called **`pc_spinlock.c`** that starts from this version and adds 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 at least 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 in this way while holding the spinlock, because doing so will cause a deadlock. 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 very much like the *test-and-set* code shown in class. 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.

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
```

To compile your program you might want to create a **Makefile**. Use the Assignment 7 as a guide. Or if you compile on the command line directly, be sure to include **uthread.c** (or **.o**). For example:

```
gcc pc_spinlock.c uthread.c -pthread
```

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.

Briefly describe your testing procedure in **README.txt** and describe how you used the values of the wait counters and items histogram to understand the execution of your program.

Part 3: The Producer-Consumer Problem (Mutexes and Conditions)

Create a new file called **`pc_mutex_cond.c`**. Copy your spinlock implementation into this file and then modify it to replace all spin waiting with blocking waiting. Use **`uthread_mutex`** for mutual exclusion and **`uthread_cond`** for synchronizing producers and consumers.

When compiling this program you much include both **`uthread`** and **`uthread_mutex_cond`**, sometime like this:

```
gcc pc_mutex_cond.c uthread.c uthread_mutex_cond.c -pthread
```

Test this implementation as you did Part 2.

Briefly describe your testing procedure in **`README.txt`** and describe how you used the values of the wait counters and items histogram to understand the execution of your program.

Requirements

Here are the requirements for this week's assignment.

1. Complete the implementation of **`tRead.c`** and test it.
2. Answer **`QUESTION 1`** described above.
3. Implement and test **`pc_spinlock.c`**.
4. Briefly describe your testing procedure, including how you used values of the wait counters and items histogram to understand the execution of your program.
5. Implement and test **`pc_mutex_cond.c`**.
6. Briefly describe your testing procedure, including how you used values of the wait counters and items histogram to understand the execution of your program.

What to Hand In

Use the **`handin`** program. The assignment directory is **a8**.

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`**".
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.tx`**t file, include the text "**`COLLABORATOR – MARK SEPARATELY`**", and turn in copies separately via **`handin`**.

4. Your modified versions of **`tRead.c`**.
5. Your answer to **`QUESTION 1`** in the **`README.txt`** file.
6. Your file **`pc_spinlock.c`**.
7. Your file **`pc_mutex_cond.c.`**
8. A description in **`README.txt`** of how you tested these two programs, including how you used values of the wait counters and items histogram.