CPSC 213: Assignment 9

Due: Tuesday, November 22, 2011 at 7am.

Late assignments are accepted until Friday, November 25 at 7am with a 20% penalty per day (or fraction of a day) past the due date. This rule is strictly applied and there are no exceptions.

Goal

In this assignment we extend the uthread.c package to include synchronization. The new version of uthread.c includes a complete implementation of spinlocks and a partial implementation of monitors and condition variables. You will complete the implementation and then use these primitives to solve a few problems.

Notes

The uthreads package runs on Intel x86 machines running Linux, MacOS or Cygwin. You can use the department linux machines by connecting to lin0x.ugrad.cs.ubc.ca.

To compile on Linux or Cygqin it is necessary to explicitly include the pthread library by adding "**`-lpthread`**" to the gcc command line.

Requirements

Here are the requirements for this week's assignment.

1. Read and comment the implementations of spinlocks and monitors.
2. Implement a multiple-reader, single-writer monitor. Recall that this monitor can be in one of three states: (a) held exclusively by a writer, (b) being read concurrently by one or more readers, or (c) free. Writers enter the monitor using the normal uthread_monitor_enter and must wait for the monitor to be in the free state before entering. Readers enter the monitor using the new uthread_monitor_enter_read_only and can enter the monitor if it is in either the reader or free states, but must wait if the monitor is currently held by a writer. You will write uthread_monitor_enter_read_only and make small changes to the monitor data structure and the existing monitor procedures. Use enter as a guide; enter_read_only will be almost identical. The main difference is that enter_read_only does not set the monitor holder field (you will indicate readers are in the monitor some other way).
3. Test your new monitor using the provided reader_ writer_test.c. The test consists of four reader threads and one writer thread access two shared integers. It has three modes of execution: (non-synchronized, monitor-synchronized, and reader-writer-monitor-synchronized). For sufficiently large values of the count command-line option, the non-synchronized version will fail with read, write or end errors. Ensure the correctness of

your new monitor by ensuring that you do not get any of these errors when you run it in reader-writer mode. If you can get access to a dual-core processor, you can tell if your implementation is really allowing multiple readers by timing the normal-monitor and reader-writer-monitor modes of execution. The reader-writer should run twice as fast, more or less.

4. Explain why the reader-writer mode of reader_writer_test should run twice as fast as the pure-monitor mode, even if that is not what you saw with your implementation.
5. Implement condition variables. Their state is stored in the `struct uthread_cv` and they have four operations `uthread_cv_create`, `uthread_cv_wait`, `uthread_cv_notify`, and `uthread_cv_notify_all`. Their implementations will be quite similar to that of monitors in that, for example, they will be implemented by a core data structure protected by a spinlock and that they will block and unblock threads. There is a fairly easy way and an excruciatingly hard way to implement these methods. The fairly easy way is to read the monitor code very carefully, identify the similarities between cv and monitors, and then copy the monitor code to the right places in the cv procedures. It is not exactly cut and paste, but it is close.
6. Test your implementation using a single processor first and a simple test program with two threads. One that waits and one that notifies it.
7. Modify the provided bounded-buffer.c to synchronize producers and consumers using monitors and condition variables. The queue will be shared by producer and consumer threads so use a monitor to synchronize. The queue is fixed size and so it is possible that an enqueue operation will find the queue full and thus have no room for a new element. Use a condition variable to block an enqueue on a full queue until a dequeue operation frees space for the new element. Similarly, a dequeue operation might find the queue empty. Use another condition variable to block a dequeue on an empty queue until an enqueue operation provides a new element.
8. Test your implementation using a single processor and four threads: two “producers” that loop enqueueing integers and two “consumers” that loop dequeueing integers and printing them.
9. Test your implementation with 2 and 4 “processors” by changing the argument to uthread_init. If you can run this on a real multi-processor (e.g., a dual-core CPU) that is great. But, you can also run the multi-threaded version on a uniprocessor. In this case, the multiple kernel threads created in uthread_init will be multiplexed across the single processor by the operating system using its scheduling policy (i.e., preemptive, round-robin) which provides a sufficient emulation of a true multi-processor for testing purposes.
10. Explain the differences you see among the two multi-processor executions and the uni-processor execution from question 5.

Material Provided

The files uthread.h, uthread.c, reader_writer_test.c, and bounded_buffer.c are provided in the file code.zip.

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-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. A version of uthread.c with comments for spinlocks and monitors and with implementations of condition variables and multiple-reader, single-writer monitors.
3. A description of the results of your testing in question 3.
4. Your answer to question 4.
5. A description of the results of your testing in question 6, 8 and 9.
6. Your modified bounded_buffer.c.
7. Your answer to question 10.