CPSC 213: Assignment 9

Due: Wednesday, March 28, 2012 at 6pm.

Late assignments are accepted until Friday, March 30 at 6pm with a 25% 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` 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, Mac OS, or Cygwin. You can use the department Linux machines by connecting to `linXX.ugrad.cs.ubc.ca`, where `XX` can be `01` through `25`.

To compile on Linux or Cygwin, it is necessary to explicitly include the `pthread` library by adding "`-lpthread`" option 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, i.e., the top and bottom parts of the `uthread.c` file.
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 `uthread_monitor_enter` function 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` function and can enter the monitor if it is in either the reader or free states (i.e., (b) or (c) above), but must wait if the monitor is currently held by a writer. You will implement the `uthread_monitor_enter_read_only` and make small changes to the monitor data structure and the existing monitor procedures. Use `uthread_monitor_enter` as a guide; `uthread_monitor_enter_read_only` will be almost identical. The main difference is that `uthread_monitor_enter_read_only` does not set the monitor holder field (you will need to 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 accessing 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 the reader-writer mode. If you can get access to a multi-core processor (e.g., the department Linux machines have quad-core processors), 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 the `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.
6. Test your implementation using a "single processor" first (by setting the argument 1 to the `uthread_init` function) 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" (by setting the argument 1 to the `uthread_init` function) 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 quad-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 archive file `code.zip`.

What to Hand In

Use the **handin** program. The assignment directory is **a9**. Please hand in exactly the following files with the specified names. Do **NOT** hand in executable or object (.o) files, or a README in formats like `.doc` or `.rtf`.

1. `uthread.c` with comments for spinlocks and monitors, as well as implementations of functions for the condition variable and the single-writer, multiple-reader monitor, as specified in Requirement 2 and 5

2. `bounded_buffer.c` that uses the monitor and condition variable to synchronize the shared queue and the `enqueue` and `dequeue` operations, as specified in Requirement 7

3. `README.txt` that contains:

   - header with your name, student number, four-digit cs-department undergraduate id (e.g., the one that's something like a0b1)

   - statement that "I have read and complied with the collaboration policies" at http://www.ugrad.cs.ubc.ca/~cs213/winter11t2/policies.html

   - description of your testing results of the monitor implementation, as specified in Requirement 3

   - answer (i.e., why the `reader_writer_test` runs faster in the reader-writer monitor mode than in the pure-monitor mode) to Requirement 4

   - description of your testing result of the condition variable implementation, as specified in Requirement 6

   - description of your single-processor testing result, as specified in Requirement 8

   - description of your multi-processor testing results, as specified in Requirement 9

   - explanation for the differences you see between the single-processor execution and the multi-processor executions, as specified in Requirement 10