Programming Assignment 2

CPS 210, Spring 1999

Due Date: Midnight 3/22

In this assignment you will exercise your concurrent programming skills using a C++ user-level thread package (SimpleThreads or **sthreads**) for a shared-memory multiprocessor. You can debug your code on any of the research Alphas (e.g., *flood*, *gale*, *torrent*), but once you have it working you should test it on *twister*, a dual-processor Alpha.

Sthreads and related code and test programs can be found in **/usr/project/courses/cps210/src** on the CS department machines. You can copy this code into your directory to build it. Please build your code using **genmake** in **/usr/project/courses/cps210/genmake**. Follow all instructions on the genmake web page carefully.

Part 1 Threads

When you look at the **sthreads** code, you will find that several critical blocks of code have been deleted. Specifically, the following routines have been gutted:

- *Thread::Sleep* and *Thread::Wakeup*,
- *Thread::Terminate* and *Thread::Join*,
- the *Wait*, *Signal*, and *Broadcast* methods of the *Condition* class.

The omitted code blocks have been replaced with lines that say:

```
Panic("ADD YOUR CODE HERE");
```

It should be clear from class discussions what the gutted routines are supposed to do, and what the code that replaces them actually does. The test programs can be built with the gutted **sthreads** package and executed, but they will crash instead of executing correctly.

Your mission is to replace the calls to *Panic* in those routines with correctly functioning code. When you are finished, your restored version of **sthreads** should execute the test programs provided, and your multithreaded programs from part 2 of this assignment.

You will need to copy the **sthreads** source directory into your source pool and modify it to test your code. However, you should hand in only the files you actually modified. You should not need to modify any files other than **Locks.c** and **Thread.c**.

Some tips:

- Every **sthreads** program must call *exit* when it completes, instead of returning from *main*. This is a "feature" of **sthreads**.

- Thread stacks in **sthreads** are only 32K. If you overflow your stack the value of *thisthread* (which should return a pointer to the current thread) will be corrupted. This can lead to extremely painful debugging.
- If you find yourself spending more than seven or eight hours debugging, step back. This assignment is not difficult. You should not need to write more than a page of code; the trick will be to figure out what page of code to write. Write your code very carefully, and if it doesn't work, go back and read it before spending hours. Learn to "think like a thread".
- Get an early start. In particular, spend some time browsing the **sthreads** code as soon as possible.

Part 2 Multithreaded Programs

1. Mutexes and condition variables are powerful enough to implement a full range of synchronization mechanisms. One popular mechanism for parallel programs is the *barrier*, which you will implement for this exercise.

   A barrier synchronizes a group of threads at a particular point in their execution. Each barrier object is created for a specified number of slave threads and one master thread. Threads block as they arrive at the barrier. When all threads have arrived, the master thread is awakened. The master then releases the slave threads, waking them up so that they may continue past the barrier.

   Create (**int** n) -- *Create barrier for **n** slaves.*
   Arrive () -- *Slaves call Arrive when they reach the barrier.*
   Wait () -- *Block the master thread until all slaves have arrived.*
   Release () -- *Release blocked slaves (all slaves must have arrived).*

   Implement barriers in C++ using the interface in the header file in *proj2*. Your implementation should have *assert* checks to verify that the barriers are used correctly. To test your implementation, write a program *barriertest* with *k* threads that synchronize at the same barrier *t* times; *k* and *t* are arguments to the program.

   Question. Do threads synchronizing at your barrier wake up in the same order in which they arrived? Why or why not?

2. This problem is due to Tom Anderson. You have been hired by the CS department to write code to help synchronize a professor and his/her students during office hours. The professor, of course, wants to take a nap if no students are around to ask questions; if there are students who want to ask questions, they must synchronize with each other and with the professor so that (i) only one person is speaking at any time, (ii) each student question is answered by the professor, and (iii) no student asks another question before the professor is done answering the previous one. You are to write four procedures: *AnswerStart(), AnswerDone(), QuestionStart(), QuestionDone()*. The professor loops running the code: *AnswerStart()*; give answer; *AnswerDone()*. *AnswerStart* doesn't return until a question has been asked. Each student loops running the code: *QuestionStart()*; ask question; *QuestionDone()*. *QuestionStart* does not return until it is the student's turn to ask a question. Since professors consider it rude for a student not to wait for an answer, *QuestionEnd()* should not return until the professor has finished answering the question.

   Write the four procedures for the interface in *proj2/StudProf.h*, together with a test program *studproftest* to simulate the students and professor to verify that your solution is correct.

3. Write a circular producer-consumer queue for the interface in *proj2/PCQueue.h.* If the queue is full the

producers will sleep until more data is consumed. If the queue is empty the consumers will sleep until more data is produced. Your queue will use a fixed amount of buffer memory in a single *malloc* block. The queue buffer will be subdivided by the *Put* routine into “entries”, each containing some requested amount of buffer space. The producers are responsible for filling each allocated queue entry (or part of it) and then notifying the queue when the entry is ready for consumption. Similarly, consumers retrieve pointers to queue entries via *Get*, and are responsible for processing the data in each entry and then notifying the queue when the entry is ready for reuse. Entries should be delivered to the consumers in the order in which they are allocated by the producers. *Warning*: be careful about the alignment of queue entries in the buffer.