Nachos Assignment #1: Threads
Due Wednesday, February 5

The goal of this assignment is to enhance the threads implementation in the Nachos system. Nachos currently implements semaphores for use in synchronization. You will build on this by providing additional synchronization primitives.

1. Implement locks. A Nachos lock can be *busy* or *free*. There are only two operations allowed on a lock:

   - **Acquire()** – wait until the lock is free, then set it to busy
   - **Release()** – set lock to be free, waking up a thread waiting in Acquire if necessary

   In addition, by convention, only the thread that acquired the lock may release it. As with semaphores, you can't read the lock value (because the value might change immediately after you read it).

   You may either use semaphores as a building block, or you may use more primitive thread routines (such as Thread::Sleep). Nachos provides the public interface to locks and condition variables in synch.h. You need to define the private data and implement the interface. Note that it should not take you very much code to implement locks.

   Test your lock implemenation. Write several test programs and document them to explain how they are testing your locks. Test basic lock operation as well as error conditions. For instance, make sure your implementation prevents users from freeing a lock they don't hold.

2. You have been hired by the state of Ohio to synchronize traffic over a narrow light-duty bridge on a public highway. Traffic may only cross the bridge in one direction at a time and there can only be one car on the bridge at a time. In this system, each car is represented by one thread, which executes the procedure *OneVehicle* when it arrives at the bridge:

   ```
   OneVehicle(int direc)
   {
    ArriveBridge(direc);
    CrossBridge(direc);
    ExitBridge(direc);
   }
   ```

   In the code above, *direc* is either 0 or 1; it gives the direction in which the vehicle will cross the bridge. Write the procedures *ArriveBridge* and *ExitBridge* using locks. The *CrossBridge* procedure should just print out a debug message. *ArriveBridge* must not return until it is safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions). *ExitBridge* is called to indicate that the caller has finished crossing the bridge; *ExitBridge* should take steps to let additional cars cross the bridge. This is a lightly-travelled rural bridge, so you do not need to guarantee fairness or freedom from starvation.

3. Type the code to implement condition variables. The code is in the *synch.cc* and *synch.h* files I gave you in class. Study and understand how condition variables work. A condition variable does not have a value, but threads may be queued, waiting on the variable. These are the only operations on a condition variable:

   - **Wait()** – release the lock, relinquish the CPU until signaled, then re-acquire the lock
   - **Signal()** – wake up a thread, if there are any waiting on the condition
   - **Broadcast()** – wake up all threads waiting on the condition

   All operations on a condition variable must be made while the current thread has acquired a lock. Indeed, all accesses to a given condition variable must be protected by the same lock. In other words, mutual exclusion must be enforced among threads calling the condition variable operations.

4. Implement producer/consumer communication through a bounded buffer, using locks and condition variables. (A solution to the bounded buffer problem is described in Silberschatz and Galvin, using semaphores.)

   The producer places characters from the string "Hello world" into the buffer one character at a time; it must wait if the buffer is full. The consumer pulls characters out of the buffer one at a time and prints them to the screen; it must wait if the buffer is empty. Test your solution with a multi-character buffer and with multiple producers and consumers. Of course, with multiple producers or consumers, the output display will be gobbledygook; the point is to illustrate the simultaneous thread execution.

5. The local laundromat has just entered the computer age. As each customer enters, he or she puts coins into slots at one of two stations and types in the number of washing machines he/she will need. The stations are connected to a central computer that automatically assigns available machines and outputs tokens that identify the machines to be used. The customer puts laundry into the machines and inserts each token into the machine indicated on the token. When a machine finishes its cycle, it informs the computer that it is available again. The computer maintains an array *available[NMACHINES]* whose elements are non-zero if the corresponding machines are available (NMACHINES is a constant indicating how many machines there are in the laundromat), and a semaphore *nfree* that indicates how many machines are available.

   The code to allocate and release machines is as follows:

```
int Allocate()  /* Returns index of available machine.*/
{
  int i;

  P(nfree);     /* Wait until a machine is available */
  for (i=0; i < NMACHINES; i++)
    if (available[i] != 0) {
      available[i] = 0;
      return i;
    }
}

void Release(int machine)    /* Releases machine */
{
  available[machine] = 1;
  V(nfree);
}
```

   The *available* array is initialized to all ones, and *nfree* is initialized to NMACHINES.

   (a) It seems that if two people make requests at the two stations at the same time, they will occasionally be assigned the same machine. This has resulted in several brawls in the laundromat, and you have been called in by the owner to fix the problem. Assume that one thread handles each customer station. Explain how the same washing machine can be assigned to two different customers.

   (b) Modify the code to eliminate the problem.

   (c) Re-write the code to solve the synchronization problem using locks and condition variables instead of semaphores.

When you are finished you will turn in this assignment by placing all of the files you wish to submit in a directory. You submit your assignment by running the program */home/rjc/pub/cps536/nachos-submit "directory name"*. You should see a listing of the files that have been submitted. If you run the submit program more than once, the later submission will replace the earlier one.

Make sure you include everything you need to submit in your directory: README, TESTCASES, and any source files you have changed.