CPS 110 Midterm

March 10, 1998

There are seven questions equally weighted at 30 points each. *Choose six of them.* You get 20 points for signing your name and stapling all your answers together, for a total of 200 points. Allocate your time carefully. Your answers will be graded on content, not style. For code answers, any kind of pseudocode is fine as long as its meaning is clear. For written answers to "essay" questions, please do not waste any words. You have 75 minutes.

1. An *EventPair* object synchronizes a pair of threads (a *server* and a *client*) for a stream of request/response interactions. The server thread waits for a request by calling *EventPair::Wait()*. The client issues a request by placing data in shared memory and calling *EventPair::Handoff()*. *Handoff* wakes up the server thread and simultaneously blocks the client to wait for the reply. The server thread eventually places the reply in shared memory and calls *Handoff* again; this wakes up the client to accept the response, and simultaneously blocks the server to wait for the next request.

   a) Show how to implement *EventPair* using mutexes and condition variables.

   b) Show how to implement *EventPair* using semaphores.

2. In Nachos Lab #3 you implemented a classical producer/consumer bounded buffer object in the *BoundedBuffer* class. Let's assume that your implementation functions correctly.

   a) Suppose a *BoundedBuffer* is used by *N* readers and *N* writers (no thread is both a reader and a writer). Is deadlock possible? Explain.

   b) Suppose the threads are exchanging information through a collection of *BoundedBuffer* objects, and each thread is both a reader and a writer. Is deadlock possible? Explain.

3. Suppose threads ***A*** and ***B*** execute on two separate CPUs in a device attached to a network. Within the device, data is passed through *BoundedBuffer* objects in shared memory. A sequence of one-kilobyte request messages arrive from the network at an average rate of ten per second, and are placed by the network into a *BoundedBuffer* read by ***A***. ***A*** uses an average of 60 milliseconds of CPU time to process each message, then writes the modified message (still one kilobyte) into a second *BoundedBuffer* read by ***B***. ***B*** also uses an average of 60 milliseconds of CPU time to process each message before posting a response message to the network. Assume that Little's Law holds in this example.

   a) What is the throughput of this system?

   b) What is the average utilization of each CPU? What is the average delay from the time an input message arrives to the time the resulting output message appears?

   c) How much buffer space must each *BoundedBuffer* provide to avoid overflow? [*Note*: you may assume that the worst-case queue length is at most twice the average.]

d) How will the system behave if its designers attempt to reduce cost by running ***A*** and ***B*** on the same CPU? What is the throughput of the low-cost version? What is its response time?

4. What is priority inversion, and why is it bad? Illustrate with an example. Use your example to illustrate one technique that avoids priority inversion.

5. Compare and contrast Round Robin scheduling with Shortest Job First (SJF) scheduling. Briefly discuss the strengths and weaknesses of each scheme with respect to the usual goals of a CPU scheduler. Why do most modern CPU schedulers combine Round Robin and SJF by favoring I/O bound jobs that have short CPU service demands?

6. *Barriers* are useful for synchronizing threads, typically between iterations of a parallel program. Each barrier object is created for a specified number of "slave" threads and one "master" thread. Barrier objects have the following methods:

   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 () -- *Master calls Release to wake up blocked slaves (all slaves must have arrived).*

   Initially, the master thread creates the barrier, starts the slaves, and calls *Barrier::Wait*, which blocks it until all threads have arrived at the barrier. The slave threads do some work and then call *Barrier::Arrive*, which puts them to sleep. When all threads have arrived, the master thread is awakened. The master then calls *Barrier::Release* to awaken the slave threads so that they may continue past the barrier. *Release* implicitly resets the barrier so that released slave threads can block on the barrier again in *Arrive*.

   Show how to implement *Barrier* using mutexes and condition variables.

7. In traditional uniprocessor Unix kernels, synchronization of processes executing system call code hinges on the following property: when a process enters the kernel via a system call trap, it becomes non-preemptible, i.e., the scheduler will not force it to relinquish the CPU involuntarily. A process in kernel mode continues executing until it returns from the system call or blocks in *sleep*.

   a) How can the kernel use this property to prevent race conditions among processes executing system call code? What other constraints must the kernel observe for kernel-mode processes to be safe from these races?

   b) Is this scheme sufficient to prevent kernel race conditions in the presence of interrupts? If not, explain the difficulties and show how to extend the scheme to handle these cases.

   c) **(extra credit)** Modern operating systems for shared memory multiprocessors are "symmetric", which means that any processor may service traps or interrupts. On these systems, the uniprocessor synchronization schemes in (a) and (b) are generally still necessary but are not sufficient. Why are they not sufficient? Explain the difficulties and show how to extend the scheme to handle the new cases.