CPS 110 Problem Set #1

Spring 1999

This problem set is optional. However, it will be used as a source of questions for exams.

1. In class we discussed synchronization using semaphores and mutex/condition variable pairs. Are these facilities equivalent in power, i.e., is there any synchronization problem that can be solved with either facility that cannot be solved with the other? Prove your answer.
2. The kernel memory protection boundary prevents dangerous access to kernel data by user-mode threads. Similarly, mutexes prevent dangerous accesses to shared data by concurrent threads. However, kernel memory protection is mandatory, whereas the protection afforded by mutexes is voluntary. Explain this statement.
3. Nachos implements threads and synchronization in a library that is linked with user applications. In contrast, many systems provide threads and synchronization through the kernel system call interface, e.g., some Unix systems and Taos/Topaz. Your implementations of the thread and synchronization primitives in Nachos use ASSERT to check for inconsistent usage of the primitives (e.g., waiting on a condition variable without holding the associated mutex). Explain why this is appropriate for a library-based implementation and why it would not be appropriate for a kernel-based implementation.
4. Traditional uniprocessor Unix kernels use the following scheme to synchronize processes executing system call code. 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. The process continues until it returns from the system call or blocks inside the kernel *sleep* primitive. Since kernel code is careful to restore all invariants on shared kernel data before blocking or exiting the kernel, the kernel is safe from races among processes.

   Old-timers in the biz call this the *monolithic monitor* approach to kernel synchronization. Explain this name, with reference to the monitor concept discussed in class and in Nutt section 9.2. Chase says: "Despite their historical importance, monitors are merely a special case of the mutex/condition variable pairs discussed in class and typically used in modern systems." Explain this statement.
5. The original *Hoare semantics* for condition variables has been widely abandoned in favor of the *Mesa semantics* used in Nachos and discussed in the Birrell paper. What effect does this have on programs that use condition variables? What effect does it have on the implementation of condition variables?
6. Suppose *Condition::Signal()* is not guaranteed to wake up only a single thread, and/or is not guaranteed to wake them up in FIFO order. Could this affect programs that use condition variables? Explain your answer, e.g., give an example of a program that would fail using the weaker semantics.
7. The Nachos code release comes with a thread test program that forks a single thread and then "ping-pongs" between the main thread and the forked thread. Each thread calls a procedure *SimpleThread()*, which loops yielding repeatedly. Draw the state of the ready list and each thread's stack and thread descriptor after each thread has yielded once, i.e., immediately after the second thread yields for the first time.

8. Tweedledum and Tweedledee are separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation. The *Sleep()* and *Wakeup()* routines operate as discussed in class: *Sleep* blocks the calling thread, and ***Wakeup*** unblocks a specific thread if that thread is blocked, otherwise its behavior is unpredictable (like Nachos *Scheduler::ReadyToRun*).

```
void
Tweedledum()
{
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledee thread);
    }
}
```

```
void
Tweedledee()
{
    while(1) {
        x = Quarrel(x);
        Wakeup(Tweedledum thread);
        Sleep();
    }
}
```

a) The code shown above exhibits a well-known synchronization flaw. Briefly outline a scenario in which this code would fail, and the outcome of that scenario.

b) Show how to fix the problem by replacing the *Sleep* and *Wakeup* calls with semaphore *P* (down) and *V* (up) operations. No, you may not disable interrupts or use *Yield*.

c) Implement Tweedledum and Tweedledee correctly using a mutex and condition variable.

9. Microsoft NT has kernel-supported threads, and it provides an *EventPair* object to help support fast request/response message passing between client and server processes (this is sometimes called *cross-domain procedure call* or *remote procedure call*, RPC). *EventPair* synchronizes a pair of threads (a *server* and a *client*) as follows. The server thread waits for a request by calling *EventPair::Wait()*. The client issues a request by placing a request message on a queue in shared memory, then calling *EventPair::Handoff()*. *Handoff* wakes up the server thread and simultaneously blocks the client to wait for the reply. The server thread eventually responds by placing the reply message in another shared queue and calling *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 a mutex and condition variable.

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

10. The Nutt text introduces semaphores using spin-waiting rather than *sleep*, i.e., a thread trying to *P* on a zero semaphore waits by spinning rather than blocking. The assumption is that the scheduler will eventually force the waiting thread to relinquish the processor via a timer interrupt, eventually allowing another thread to run and break the spin by calling *V* on the semaphore. To avoid wasting CPU cycles in pointless spinning, Nutt proposes (in the box on p. 213) that a waiting thread call *yield* each time it checks the semaphore and finds that it is still zero. Let us call this alternative to blocking *spin-yield*.

    Compare and contrast spin-yield with blocking using *sleep*. Is spin-yield subject to the same concurrency race difficulties as *sleep*, e.g., is it necessary to (say) disable interrupts before calling *yield*? Why or why not? What is the effect of spin-yield on performance? How will spin-yield work in a thread system that supports priorities?

11. Synchronization primitives are often used for coordination rather than mutual exclusion. For example, it is often necessary for threads to wait for a particular event to occur. Use the semaphore primitives to

implement an *Event* class with the following semantics, similar to the fundamental coordination primitives used throughout Windows and NT. Initially, an event object is in the *unsignaled* state. *Event::Wait()* blocks the calling thread if the event is in the unsignaled state. *Event::Signal()* transitions the event to the *signaled* state, waking up all threads waiting on the event. If *Wait* is called on an event in the signaled state, it returns without blocking the caller. *Event::Reset()* resets an event object to the unsignaled state.

Your implementation will use a single semaphore with no additional synchronization. Once you're done, spend some more time ruminating about *Event* in order to fully appreciate the power of semaphores here. Note that *Event* can be viewed as a sort of "safe" condition variable (or a safe *sleep/wakeup*), in that the primitives are similar, but it can be used without fear of sleep/wakeup races, even without an accompanying mutex. What property of the *Event* class (and semaphores) accounts for this "safety"?

How will your *Event* class behave if *Signal* is called more than once without an intervening *Reset*? Is this behavior "safe"? How should it behave? Can you modify your implementation to detect this case without introducing new synchronization? What weakness of semaphores does this example expose?

Now reimplement *Event* with a mutex and condition variable. Was it easier or harder than your solution using semaphores? Is it easier or harder to convince yourself that your solution is correct? How easy is it to extend your implementation to detect the "double signal" case above?

12. The espresso franchise in the strip mall near my house serves customers FIFO in the following way. Each customer entering the shop takes a single "ticket" with a number from a "sequencer" on the counter. The ticket numbers dispensed by the sequencer are guaranteed to be unique and sequentially increasing. When a barrista is ready to serve the next customer, it calls out the "eventcount", the next highest unserved number previously dispensed by the sequencer. Each customer waits until the eventcount reaches the number on its ticket. Each barrista waits until the customer with the ticket it called places an order.

    Show how to implement sequencers, tickets, and eventcounts using mutexes and condition variables. Your solution should also include code for the barrista and customer threads.

13. Trapeze is a locally grown high-speed messaging system for gigabit-per-second Myrinet networks. When a message arrives, the network interface card (NIC) places it in a FIFO queue and interrupts the host; if the host does not service the interrupts fast enough then incoming messages build up in the queue. Most incoming messages are handled by a special thread (the server thread). However, many Trapeze messages are replies to synchronous requests issued earlier from the local node; the reply handler code must execute in the context of the thread that issued the request, and is sleeping to await the reply. As an optimization, the receiver interrupt handler services reply messages by simply waking up the requester thread, saving an extra context switch to the server thread if it is not already running.

    Show how to synchronize the receiver interrupt handler and the server thread to implement the following behavior. On an interrupt, the handler removes messages from the queue, processing reply messages by waking up waiting threads. If it encounters a message that is not a reply message then it wakes up the server thread to process all remaining pending messages, and returns from the interrupt. The server thread simply loops processing messages in order, sleeping whenever the queue is empty.

14. The Klingons are attacking. The Federation vessels can escape through the wormhole, but sensors indicate that the wormhole is unstable. The ships' captains plan to create a subspace distortion to prevent

the wormhole from closing on them while they are in it. To do this, they will enter the wormhole three at a time while emitting a tachyon pulse through their main deflector dishes. Each ship must lower its shields before initiating its tachyon pulse, and once a ship starts emitting tachyons it can have no contact with the other ships.

Implement a synchronization scheme to allow the Federation to retreat through the wormhole in an orderly fashion. Your scheme should have the property that no ship lowers its shields until just before it enters the wormhole, no ship enters the wormhole until two others are ready to go in with it, and all ships in each group of three enter the wormhole before any of the ships in the next group. (You may find the last condition difficult to guarantee in the presence of preemptive timeslicing.) Note: some have complained that the solutions to this problem, while instructive, are inelegant. This problem will not appear on an exam.

15. Nutt chapter 8, problems 3 and 4.

16. The Alpha and MIPS 4000 processor architectures have no atomic read-modify-write instructions, i.e., no test-and-set-lock instruction (TS). Atomic update is supported by pairs of *load_locked* (LDL) and *store-conditional* (STC) instructions.

    The semantics of the Alpha architecture's LDL and STC instructions are as follows. Executing an *LDL Rx, y* instruction loads the memory at the specified address (*y*) into the specified general register (*Rx*), and holds *y* in a special per-processor *lock register*. *STC Rx, y* stores the contents of the specified general register (*Rx*) to memory at the specified address (*y*), but only if *y* matches the address in the CPU's lock register. If STC succeeds, it places a one in *Rx*; if it fails, it places a zero in *Rx*. Several kinds of events can cause the machine to clear the CPU lock register, including traps and interrupts. Moreover, if any CPU in a multiprocessor system successfully completes a STC to address *y*, then every other processor's lock register is atomically cleared if it contains the value *y*.

    Show how to use LDL and STC to implement safe busy-waiting, i.e., spinlock *Acquire* and *Release* primitives. Explain why your solution is correct.

17. The kernel synchronization scheme described in problem #4 must be supplemented with explicit operations to temporarily inhibit interrupts at specific points in the kernel code. When is it necessary to disable interrupts? Why is it not necessary to disable interrupts in user mode? Why is it impossible to disable interrupts in user mode? What will happen if user code attempts to disable interrupts?

    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 described above are (broadly) still necessary but are not sufficient. Why are they not sufficient? Explain what the problem is in each case, and how to augment the uniprocessor scheme to fix it using the technique from problem #17. Why is it still important to disable interrupts and protect kernel-mode code from preemption, even with the extra synchronization? Note: this is a thought question intended to show a bit more of an iceberg whose tip was exposed in class. This problem will not appear on an exam in this form.

18. Highway 110 is a two-lane north-south road that passes through a one-lane tunnel. A car can safely enter the tunnel if and only if there are no oncoming cars in the tunnel. To prevent accidents, sensors installed at each end of the tunnel notify a controller computer when cars arrive or depart the tunnel in either direction The controller uses the sensor input to control signal lights at either end of the tunnel.

Show how to implement the controller program correctly using mutexes and condition variables. You may assume that each car is represented by a thread that calls your ***Arrive()*** and ***Depart()*** functions, passing an argument indicating the direction of travel. You may also assume that ***Arrive()*** can safely stop the arriving car by changing the correct signal light to red and blocking the calling thread. Your solution should correctly handle rush hour, during which most cars approach the tunnel from the same direction. (Translation: your solution should be free from starvation.)

19. Is the Nachos semaphore implementation fair, e.g., is it free from starvation? How about the semaphore implementation (using mutexes and condition variables) presented in class? If not, show how to implement fair semaphores in each case.

20. Round-robin schedulers (e.g., the Nachos scheduler) maintain a *ready list* or *run queue* of all runnable threads (or processes), with each thread listed at most once in the list. What can happen if a thread is listed twice in the list? Explain how this could cause programs to break on a uniprocessor. For extra credit: what additional failure cases could occur on a multiprocessor?

21. [Tanenbaum] A student majoring in anthropology and minoring in computer science has embarked on a research project to see if African baboons can be taught about deadlocks. He locates a deep canyon and fastens a rope across it, so the baboons can cross hand-over-hand. Several baboons can cross at the same time, provided that they are all going in the same direction. If eastward moving and westward moving baboons ever get onto the rope at the same time, a deadlock will result (the baboons will get stuck in the middle) because it is impossible for one baboon to climb over another one while suspended over the canyon. If a baboon wants to cross the canyon, he must check to see that no other baboon is currently crossing in the opposite direction. Write a program using semaphores that avoids deadlock. Do not worry about a series of eastward moving baboons holding up the westward moving baboons indefinitely.

22. [Tanenbaum] A distributed system using mailboxes has two IPC primitives, **send** and **receive**. The latter primitive specifies a process to receive from, and blocks if no message from that process is available, even though messages may be waiting from other processes. There are no shared resources, but processes need to communicate frequently about other matters. Is deadlock possible? Discuss.

23. Cinderella and Prince are getting a divorce. To divide their property, they have agreed on the following algorithm. Every morning, each one may send a letter to the other's lawyer requesting one item of property. Since it takes a day for letters to be delivered, they have agreed that if both discover that they have requested the same item on the same day, the next day they will send a letter canceling the request. Among their property is their dog, Woofer, Woofer's doghouse, their canary, Tweeter, and Tweeter's cage. The animals lover their houses, so it has been agreed that any division of property separating an animal from its house is invalid, requiring the whole division to start over from scratch. Both Cinderella and Prince desperately want Woofer. So they can go on (separate) vacations, each spouse has programmed a personal computer to handle the negotiation. When they come back from vacation, the computers are still negotiating. Why? Is deadlock possible? Is starvation possible?

24. 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.

25. *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*.

26. *The Sleeping Professor Problem*. Once class is over, professors like to sleep — except when students bother them to answer questions. You are to write procedures to synchronize threads representing one professor and an arbitrary number of students.

    A professor with nothing to do calls *IdleProf()*, which checks to see if a student is waiting outside the office to ask a question. *IdleProf* sleeps if there are no students waiting, otherwise it signals one student to enter the office, and returns. A student with a question to ask calls *ArrivingStudent()*, which joins the queue of students waiting outside the office for a signal from the professor; if no students are waiting, then the student wakes up the sleeping professor. The idea is that the professor and exactly one student will return from their respective functions "at the same time": after returning they discuss a topic of mutual interest, then the student goes back to studying and the professor calls *IdleProf* again.

    a) Implement *IdleProf* and *ArrivingStudent* using mutexes and condition variables. You may assume that mutexes and condition variables are fair (e.g., FIFO).

    b) Implement *IdleProf* and *ArrivingStudent* using semaphores. You may assume that semaphores are fair (e.g., FIFO).

27. Implement reader/writer locks (as described in Birrell and in class as *SharedLock*) using semaphores.

28. Most implementations of Sun's Network File Service (NFS) use the following Work Crew scheme on the server side. The server node's incoming network packet handler places incoming requests on a shared work queue serviced by a pool of server threads. When a server thread is idle (ready to handle a new request), it examines the shared queue. If the queue is empty, the server thread goes to sleep. The incoming packet handler is responsible for waking up the sleeping server threads as needed when new requests are available.

    a) Show how to implement the NFS server-side synchronization using a mutex and condition variable.

For this problem you may assume that the incoming packet handler is a thread rather than an interrupt handler. (Note: this is more-or-less identical to the SynchList class in Nachos

b) This problem is similar to but slightly different from the Trapeze example in an earlier problem. Summarize the differences, and make up an explanation for the differences in terms of the underlying assumptions of each system. It should be a good explanation.

c) What will happen to the queue if the server machine is not powerful enough to process requests at their arrival rate on the network? What should the incoming packet handler do in this case? How is this different from the standard producer-consumer *BoundedBuffer* problem assigned in Lab #3 and used to implement pipes in Lab #5?

d) Early NFS server implementations used *Broadcast* as a wakeup primitive for the server threads, because no *Signal* primitive was available in Unix kernels at that time (around 1985). This was a common performance problem for early NFS servers. Why is *Signal* more efficient than *Broadcast* here?

e) This and other examples led Carl Hauser and co-authors from Xerox PARC to claim (in a case study of thread usage in SOSP93) that "*Signal* is just a performance hint". What they meant was that *Signal* is not necessary to implement correct programs using condition variables, and that (more generally) any use of *Signal* can be replaced by *Broadcast* without affecting the program's correctness, although the *Signal* might be more efficient. Do you believe this statement? What assumptions does it make about condition variables?