CPS 110 Problem Set #1 with Solutions

Spring 2000

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.

   To prove that the implementations are equivalent, you must implement each in terms of the other. You implemented mutexes and condition variables using semaphores for lab 2. An implementation of semaphores using mutexes and condition variables was given in class. To properly answer this question, you must write pseudocode for both implementations.

2. *Late-Night Pizza*. A group of students are studying for a CPS 110 exam. The students can study only while eating pizza. Each student executes the following loop: *while (true) { pick up a piece of pizza; study while eating the pizza}*. If a student finds that the pizza is gone, the student goes to sleep until another pizza arrives. The first student to discover that the group is out of pizza phones Satisfactions at Brightleaf to order another pizza before going to sleep. Each pizza has *S* slices.

   Write code to synchronize the student threads and the pizza delivery thread. Your solution should avoid deadlock and phone Satisfactions (i.e., wake up the delivery thread) exactly once each time a pizza is

exhausted. No piece of pizza may be consumed by more than one student. [Andrews91]

```
int slices=0;
Condition orderPizza, deliverPizza;
Lock mutex;
bool first = true, havePizza = false;
```

```
Students() {
   while(TRUE) {
     mutex.Acquire();
     while( !havePizza ) {
        if( slices > 0 ) {
          slices--;
          havePizza = true;
        }
        else {
           if( first ) {
             orderPizza.Signal(mutex);
             first = false;
           }
           deliver.Wait(mutex);
                      }
     }  // end while !havePizza
     mutex.Release();
     Study();
     havePizza = false;
   }
}
```

```
Satisfactions() {
  while(TRUE) {
      mutex.Acquire();
      orderPizza.Wait(mutex);
      makePizza();
      slices = S;
      first=true;
      deliver.Broadcast(mutex);
      mutex.Release();
  }
}
```

3. *Flipping Philosophers*. The Dining Philosophers have worked up a solution to avoid deadlock, with a little help from a consultant with a recent doctorate in algorithms. Before eating, each philosopher will flip a coin to decide whether to pick up the left fork or the right fork first. If the second fork is taken, the philosopher will put the first fork down, then flip the coin again. Is this solution deadlock-free? Does it guarantee that philosophers will not starve? Modify the solution as necessary to guarantee freedom from deadlock and starvation. [Andrews91]

   This solution is deadlock-free, however it does not guarantee freedom from starvation. The philosophers are never stuck waiting and unable to do "useful work". However, a philosopher might not ever get a chance to eat because at least one of his/her forks is busy.

   In class, we went over several solutions to this problem, but one solution fits this problem best..

4. Expensive Candy. Three engineering professors have gone to the faculty club to eat licorice. Each piece of licorice costs 36 cents. To buy a piece of licorice, a professor needs a quarter, a dime, and a penny (they do not give change, and they don't take American Express). The first professor has a pocket full of pennies, the second a supply of quarters, and the third a supply of dimes. A wealthy alum walks up to the bar and lays down at random two of the three coins needed for a piece of licorice. The professor with the third coin takes the money and buys the licorice. The cycle then repeats. Show how to synchronize the professors and the alum. [Patil71, Parnas75, Andrews91]

5. 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,

```
coin coinsOnTable[3];
Lock mutex;
Condition placedCoins, needMoreCoins;
```

```
coin myCoin;
Professors() {
  mutex.Acquire();
  placedCoins.Wait();
  mutex.Release();

  while(TRUE) {
    mutex.Acquire();
    if( myCoin != coinsOnTable[0] &&
        myCoin != coinsOnTable[1] ){
      coinsOnTable[2] = myCoin;
      BuyLicorice();
      needMoreCoins.Signal( mutex );
    } else {
      placedCoins.Wait( mutex );
    }
    mutex.Release();
  }
}
```

```
Alumnus() {
   while(TRUE) {
     mutex.Acquire();
     twoRandomCoins( coinsOnTable );
     placedCoins.Broadcast( mutex );
     needMoreCoins.Wait( mutex );
     mutex.Release();
   }
}
```

kernel memory protection is mandatory, whereas the protection afforded by mutexes is voluntary. Explain this statement.

Kernel memory is mandatory in that there is no way for an untrusted program to modify the data structures in a protected kernel (unless there's a bug in the protection mechanism.) User-mode code will trigger a machine exception if it attempts to store to kernel memory (if the addresses are even meaningful.) In contrast, mutex protection is voluntary in that there is no mechanism to prevent a thread or process from accessing critical data in shared memory without acquiring the mutex. If the thread is executing a correctly written program, then this won't happen. If the program is written incorrectly and this does happen, then the program can only damage itself. To put it another way, the kernel protection boundary is like a bank safe preventing you from stealing something that belongs to others, while the mutex is like a stair railing that you may optionally hold onto to prevent yourself from falling and breaking your own leg.

6. 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.

   Using ASSERTS in a library-based implementation is appropriate because an ASSERT occurs when a process performs an illegal action, so to protect other processes, the violating threaad is forced to exit. On the otherhand, in the kernel-based implementation, if a violation occurs, an ASSERT would cause the kernel to exit, which is not an acceptable way to handle an error.

7. 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 retains control of the CPU 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. Explain why you agree or disagree with the following statement: "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."

   One monolithic monitor protects all of the kernel's private data and private operations by allowing only one process in the kernel at a time.

8. The kernel synchronization scheme described in the previous problem 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 not permissible to disable interrupts in user mode? What will happen if user code attempts to disable interrupts?

9. 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.

```
.globl Acquire(X)
    again:
        LDL R1, X        # load the spinlock from memory
        BNEZ R1, again # loop if held (non-zero)
        LDI R2, 1        # load 1 into a register
        STC R2, X        # store 1 into the spinlock
        BEZ R2, again    # STC clears R2 if the store failed with respect to LDL, so try again if zero

.globl Release(X)
    ST #0, X             # clear the spinlock
```

10. Modern operating systems for shared memory multiprocessors are "symmetric", which means that all processors run the same kernel and any processor may service traps or interrupts. On these systems, the uniprocessor kernel synchronization schemes described in earlier problems 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 the previous problem. 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.

11. 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?

Overall, both semantics work the same. With Hoare, the assumption is that when a process signals a process waiting on a condition, the waiting process must begin executing right away because at that moment the condition is true. After the previously waiting process is done executing, the signaling process resumes what it was doing. There were two context switches in that example..

If Mesa semantics are used instead, when a waiting process is signaled, it doesn't start until the signaling process has completed execution. However, there is no guarantee that the condition is still true after the signaling process is done; therefore, the formerly waiting process now must check on the condition it was waiting on. If the condition is true, it can continue running. If the condition is false, the process has to wait again. In this situation, there was only one context switch.

Programmers writing code that uses condition variables with Mesa semantics must make sure that, when a thread wakes up, it rechecks the condition it was waiting on before continuing execution. Mesa semantics require fewer context switches, improving overall performance.

The implementation of Mesa semantics changes an `if` *to a* `while`*--the condition must be rechecked after a process is woken up.*

12. 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.

A program that assumes that Signal() wakes up only one thread and then assumes that after a thread waits in Wait(), it now has mutual exclusion of any shared data will fail given these weaker semantics.

Suppose the program starts two threads and both immediately call Condition::Wait(). The program then calls Condition::Signal() to wake one of them, which in turn will call another Signal() when it is safe for the second to proceed. This program would break if Signal() woke up more than one thread.

13. 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.

14. 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.

Missed wakeup--one thread calls Wakeup before the other one calls Sleep. A missed wakeup can occur during any iteration of the loop if the scheduler forces a Yield between the Wakeup *and the* Sleep.

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*.

(For parts b and c) Once you recognize that this is the same as the ping-pong problem presented in class, refer to the extensive coverage of the ping-pong problem in the lecture notes.

```
TweedleDum() {
      while (TRUE) {
            dum->P();
            x = Quarrel(x);
            dee->V();
      }
}

TweedleDee() {
      while (TRUE) {
             dee->P();
             x = Quarrel(x);
             dum->V();
      }
}
```

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

```
Tweedle*()
{
      mx->Acquire();
      while (TRUE) {
            x = Quarrel(x);
           cv->Signal(&mx);
            cv->Wait(&mx);
      }
      mx->Release();
}
```

15. 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 *local/remote procedure call*, LPC or 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.

This is a standard "ping-pong" problem exactly like Tweedledum and Tweedledee. For the mutex/CV solution, a single condition variable is sufficient. The ping-pong loop code (e.g., Handoff) is as follows:

```
mx->Acquire();
cv->Signal(mx);
cv->Wait(mx);
mx->Release();
```

This is one case in which you do not need to "loop before you leap": it is the exception that proves the rule. In this case, there is only one thread waiting and one thread issuing the signal, and no condition that must be true (other than that the signal occurred).

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

Two semaphores are needed to prevent one party from consuming its own signal (V). No mutex is needed. The semaphore code looks like:

```
othersem->V();
mysem->P();
```

For EventPair, the Wait() primitive exists just to prime the "ping" so there's a thread blocked to "pong". This code is easy (just P() or cv->Wait()). Also, for EventPair there needs to be some way to keep track of which semaphore is the "current" semaphore in handoff, and switch to the other semaphore after each handoff. For TweedleDum and TweedleDee, put the code fragments above in a loop, one loop for Dee and one loop for Dum (make sure each loop Ps on its own semaphore and Vs on the other guy's. In the Mx/CV solution, the Mx acquire may be outside the loop.

Note: for TweedleDum/TweedleDee, the problem with the sleep solution is "the missed wakeup prob-

lem", where one thread issues its wakeup before the other is asleep. Understand how the Mx/CV and semaphore solutions avoid this problem.

16. *Event*. This problem asks you to implement an *Event* class similar to the fundamental coordination primitives used throughout Windows and NT. Initially, an *Event* object is in an *unsignaled* state. *Event::Wait()* blocks the calling thread if the event object 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. Once an event is signaled, it remains in the signaled state until it is destroyed.

    (a) Implement *Event* using a single semaphore with no additional synchronization.

```
Semaphore sem = 0;

Signal() {                    Wait() {
   sem.V();                      sem.P();
}                                sem.V();
                              }
```

    (b) Implement *Event* using a mutex and condition variable.

```
Lock mutex;
ConditionVariable waitForEvent;
bool signal=false;

Signal() {                                  Wait() {
   mutex.Acquire();                            mutex.Acquire();
   signal=true;                                if( !signal ) {
   waitForEvent.Broadcast(mutex);                 waitForEvent.Wait(mutex);
   mutex.Release();                            }
}                                              mutex.Release();
                                            }
```

    (c) Show how to use *Event* to implement the synchronization for a *Join* primitive for processes or threads. Your solution should show how *Event* could be used by the code for thread/process *Exit* as well as by *Join*. For this problem you need not be concerned with deleting the objects involved, or with passing a status result from the *Join*.

17. The Nutt text introduces semaphores using spin-waiting rather than *sleep*, i.e., a thread executing a *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 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?

    In spin-yield, the Yield indicates that the thread cannot do anything useful at the moment, so some other thread can utilize the CPU instead. This technique wastes less CPU cycles, which seems to imply increased performance. However, spin-yield also adds context switching overhead. If the overhead of switching contexts is less than the utilized CPU cycles, then performance is still better. But, if over-

head is more expensive than the utilized cycles, then performance declines.

Spin-yield does not have the same race condition difficulties. When a thread puts itself to sleep, it must be sure that the other thread (or just another thread) has been notified about what has happened. In spin-yield, the thread will periodically be scheduled to run, so it doesn't matter when another thread completes its execution or causes some signal. It is not necessary to disable interrupts before calling Yield because, assuming a good scheduler, the thread will resume execution at some later point in time.

Another problem with spin-yield is that it assumes that the scheduler will choose to run a different *thread. Yield is just a suggestion, and the scheduler can ignore it. Using Yield assumes that the scheduler will fairly choose which thread to run. While that usually is a fair assumption, your synchronization should not rely upon it.*

18. *Cold coffee*. 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 lowest 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.

```
Lock sharedLock;
ConditionVariable sequencerCV, baristaCV;
int eventcount=0, nextTicket=0;

customer() {
   sharedLock->Acquire();
   int myTicket = nextTicket++;
// atomically acquire ticket under
// sharedLock
   while (myTicket != eventcount)
      sequencerCV->Wait(&sharedLock);
   // place order
   baristaCV->Signal(&sharedLock);
   sharedLock->Release();
}

barrista() {
 sharedLock->Acquire();
 while(TRUE) {
  eventcount++;
  sequencerCV->Broadcast(&sharedLock);
  // announce a new eventcount
  baristaCV->Wait(&sharedLock);
  // wait for order or customer to
  // arrive
 }
 sharedLock->Release();
 // outside while but still
 // good practice.
}
```

19. 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 thethe 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 in the host (the server thread). However, many Trapeze messages are replies to synchronous requests (RPCs) issued earlier from the local node; the reply handler code must execute in the context of the thread that issued the request, which 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 follow-

ing 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 (including reply messages). The interrupt handler completes if it awakens the server thread, or if the FIFO queue is empty. The server thread simply loops processing messages in order, sleeping whenever the queue is empty.

20. *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 ***Arrive()*** and ***Depart()*** functions in the controller, 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.)

```
Lock tunnelMx;
Condition tunnelCv;
int waiters[2];
int count = 0;
int turn = free;
```

```
Arrive (int dir) {
   tunnelMx->Acquire();  /* lock */

   while(turn == other ||
        (turn == dir &&
         waiters[other] > N...) ) {
       waiters[dir]++;
       tunnelCv[dir]->Wait(...);
       waiters[dir]---;
   }
   turn = dir;
   count++;

   tunnelMx->Release();
}
```

```
Depart (int dir) {
   tunnelMx->Acquire();

   count--;
   if (count == 0)  {
      turn = free;
      tunnelCv[other]->Broadcast(...);
   }

   tunnelMx->Release();
}
```

21. 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.

    No. A Semaphore::V() places a blocked thread waiting for the semaphore on the runnable queue. If another thread ahead in the runnable queue tries to P(), it will succeed. The 'intended' recipient of the V() will notice count==0 and block again.

22. 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?

If a thread is appears twice in the run queue, it may return from a sleep unexpectedly. For example, ConditionVariable::Wait() could 'return' without a Signal(). On a multiprocessor, both instances of the same thread may run simultaneously, clobering local variables while running and saved registers when switched out.

23.[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, it 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.

```
enum direction { WEST=0, EAST=1 };
Semaphore mutex; // initially 1, protect critical sections
Semaphore blocked[2];
// one for each direction.  Each initially 0, so always sleep on first P
int blockedCnt[2]; // number of baboons waiting on each side
int travellers[2]; // number of baboons on the rope heading each direction
//(at least one is zero at any time)

Baboon(direction dir) {
      int revdir = !dir; // the reverse direction
      mutex->P();
      while (travellers[revdir]) {
         blockedCnt[dir]++; // announce our intention to block
         mutex->V(); // trade mutex for block
         blocked[dir]->P();
         mutex->P();
      }
      travellers[dir]++; // we're free to cross
      mutex->V();
      // cross bridge.
      mutex->P();
      travellers[dir]--;
      if (!travellers[dir]) {
      // if we're the last one heading this way,
      // wakeup baboons waiting for us to finish.
         while(blockedCnt[revdir]--)
            blocked[revdir]->V();
      }
      mutex->V();
}
```

24.[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.

Deadlock is possible. Process A may query process B and await a response from B. B receives A's message but must query process C before it can respond to A. Likewise process C must query process

A in order to answer B's question. A is only listening to B so C will never get an answer and thus cannot respond to B, which in turn cannot respond to A. Deadlock.

25. 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?

If both parties always request the same item (eg Woofer), livelock occurs because the transactions are always cancelled.

Cinderella requests	**Prince requests**	**End Result**
Woofer	Woofer	Cancels after a day because both want the same thing.
Woofer	Woofer's House	Have to start over (after a day) because the dog and his house cannot be separated.
Woofer's House	Woofer	Have to start over (after a day) because the dog and his house cannot be separated.
Woofer's House	Woofer's House	Cancels after a day because both want the same thing.

Deadlock is not possible because illegal settlements restart the negotiations (same for starvation).

26. 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.

No.

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.

Yes.

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

Create (int n) -- *Create barrier for **n** workers.*
Arrive () -- *Workers call Arrive when they reach the barrier.*
Wait () -- *Block the controller thread until all workers have arrived.*
Release () -- *Controller calls Release to wake up blocked workers (all workers must have arrived).*

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

the barrier again in *Arrive*.

```
Condition* slavecv, mastercv;
Mutex* mx;
int count=0;

Create(int n) {
     new slavecv, mastercv, and mx;
     count = n;
}

Arrive() {
     mx->Acquire();
     count--;
     if (count == 0) {
        mastercv->Signal(mx);
        slavecv->Wait(mx);
        mx->Release();
     }
}

Wait() {
     mx->Acquire();
     while (count)
        cv->Wait(mx);
     mx->Release();
}

Release() {
     mx->Acquire();
     slavecv->Broadcast(mx);
     count = n;
     mx->Acquire();
}
```

28. *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).

```
Lock mutex;
Condition prof, student;
bool profBusy = true;
int numStudents = 0;
```

```
while(TRUE) {
   mutex.Acquire();
   profBusy = false;

   IdleProf()  {
      if( numStudents == 0 )
         prof.Wait( mutex );
      else {
         profBusy = true;
         numStudents--;
         student.Signal( mutex );
      }
   }
   mutex.Release();
}
```

```
ArrivingStudent() {
   mutex.Acquire();
   if( !profBusy && numStudents == 0 ) {
      profBusy = true;
      prof.Signal( mutex );
   }
   else {
      numStudents++;
      student.Wait( mutex );
   }
   mutex.Release();
}
```

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

```
Semaphore student = 0;  // student "resources"
Semaphore prof = 0;     // professor "resources"
```

```
while(TRUE) {

   IdleProf() {
      prof.V();   // 1 prof avail.
      student.P(); // 'consume' 1 stud
   }
}
```

```
ArrivingStudent() {
   student.V();  // 1 student avail
   prof.P();     // 'consume' 1 prof
}
```

29. 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. Outline 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 the arrival rate on the network? What should the incoming packet handler do in this case? How is this dif-

ferent 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?