CPS 110 Problem Set #1

Fall 1997

This problem set is optional. However, it will be used as a source of questions for the midterm exam on *Thursday, October 9*.

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

2. Write a modified version of the semaphore *Up()* and *Down()* primitives that take counts of the number of resources to be atomically acquired or released. Your solution should be free of starvation and deadlock, assuming that no thread ever calls *Down()* when it already holds resources. Can you make your solution safe from deadlock if this assumption is relaxed? What additional assumptions are needed to apply the banker's algorithm?

```
// value, reserved are private members of the FairSemaphore class.  reserved is initially zero.
FairMultiSemaphore::P(n) {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    while (value < n) {
        reserved += value; // move value contents to reserved so no one else can steal it.
        value = 0;
        queue->Append((void *)currentThread);
        currentThread->Sleep();
        value+= reserved;
        reserved = 0;
    }
    value -= n;
    if (!queue->IsEmpty())
        scheduler->ReadyToRun((Thread *)queue->Remove()); // value may be enough to satisfy another
    interrupt->SetLevel(oldLevel);
}
FairMultiSemaphore::V(n)
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    Assert(value<=1);
    if (!queue->IsEmpty()) {
        reserved += n;
        scheduler->ReadyToRun((Thread *)queue->Remove());
    } else
        value += n;
    interrupt->SetLevel(oldLevel);
}
```

3. 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.

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

customer() {
    sharedLock->Acquire();
    int myTicket = nextTicket++; // atomically acquire ticket under sharedLock
    while (myTicket != eventcount)
        sequencerCV->Wait(&sharedLock);
    // place order
    barristaCV->Signal(&SharedLock);
}
barrista() {
    sharedLock->Acquire();
    while(1) {
        sequencerCV->Broadcast(&SharedLock); // announce a new eventcount
        barristaCV->Wait(&sharedLock); // wait for order or customer to arrive
    }
    sharedLock->Release(); // outside while(1) but still good practice.
}
```

4. 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.

   Kernel memory protection 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 it 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.

5. 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.

```
// value, reserved are private members of the FairSemaphore class.  reserved is initially zero.
FairSemaphore::P() {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    while (value == 0) {
        queue->Append((void *)currentThread);
        currentThread->Sleep();
        value++;
    }
    value--;
    interrupt->SetLevel(oldLevel);
}
FairSemaphore::V()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    if (!queue->IsEmpty())
        scheduler->ReadyToRun((Thread *)queue->Remove());
    else
        value++;
    interrupt->SetLevel(oldLevel);
}
```

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? How?

   Suppose the program starts two threads and both immediately 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.

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

   The semantics of LDL and STC are as follows. Executing an *LDL Rx, y* instruction loads the memory at the specified address (*y*) into the specified register (*Rx*), and holds *y* in a special per-processor lock register. *STC Rx, y* stores the contents of the specified 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
```

8. 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.

9. 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.

10. Tanenbaum ch 2, problem 35

```
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();
}
```

11. Tanenbaum ch 3, problem 16

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.

12. Tanenbaum ch 3, problem 19

If both parties always request the same item (eg Woofer), livelock occurs because the transactions are always cancelled. Deadlock is not possible because illegal settlements restart the negotiations (same for starvation).