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.

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?

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.

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.

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.

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?

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.

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?

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

11. Tanenbaum ch 3, problem 16

12. Tanenbaum ch 3, problem 19