Jeremy Holloway
CPSC-3220-001
5/24/2019

Pre-class Assignment #7

1. Fill in the last four rows of values in the middle three columns for the following trace table that reflects the lost update example in the slides. Each thread has its own copy of register r1 and is able to access the shared global variable x using load and store instructions. ('?' means unknown value.)

thread A action	**thread A's r1**	**global variable x**	**thread B's r1**	**thread B action**
(initially)	?	0	?	*(initially)*
load r1,x	0	0	?	
add r1,r1,#1	1	0	?	
	1	0	0	load r1,x
	1	0	1	add r1,r1,#1
store r1,x	1	1	1	
	1	1	1	store r1,x

2. How may different ways can you interleave the six machine instructions from thread A and thread B in the lost update example above? (Assume within a thread that the instructions are in sequential order. That is, assume that you will always observe the sequential ordering of load, add, and store within a given thread.)

There are 20 different ways to interleave the instructions while keeping the commands in sequential order.

N!/((n-k)! * k!) = 6!/(3! * 3!) = 20

3. Define these terms:

- race condition: When the behaviors of a program relies on the interleaving of operations of different threads.
- atomic operation: Indivisible operations that cannot be interleaved with or split by other operations.
- mutual exclusion: Only one thread does a particular thing at a time.
- critical section: Piece of code that only one thread can execute at once.
- condition variable: A synchronization object that lets a thread efficiently wait for a change to shared state that is protected by a lock.

4. Suppose we add a method to ask if a lock is free. Suppose it returns true. Is the lock now free, busy, or do you no longer know?

You now longer know because another thread could have used the lock immediately after checking its status.

5. What is the difference between a condition variable signal and broadcast?

The difference between condition variable signal and broadcast is that signal will only mark one thread as eligible to run, while broadcast marks all.

6. What do we mean when we say a condition variable is memoryless?

A condition variable is memoryless if it has no internal state other than a queue of waiting threads and/or if there are no threads currently on the waiting list.

7. Under Mesa semantics, when a thread is woken up from a wait on a condition variable, will it run immediately?

No, it is still on the ready list so it would still have to wait to enter a running state.

8. Under Hoare semantics, when a thread is woken up from a wait on a condition variable, will it run immediately?

Yes, when the thread is woken up it is guaranteed to run next.

9. List the five steps given in section 5.5.1 for synchronizing accesses to a shared object in a multithreaded environment. (A single sentence for each step is sufficient.)

1. Add a lock
2. Add code to acquire and release the lock
3. Identify and add condition variables
4. Add loops to wait using to wait using the condition variables
5. Add signal and broadcast calls

10. List the six best practices given in section 5.5.2 for implementing synchronization for a shared object. (A single sentence for each practice is sufficient.)

1. Consistent Structure: This is a meta rule that underlies the other five rules, to follow a consistent structure.

2. Always synchronize with locks and condition variables: Code using locks and condition variables is clearer than the equivalent code using semaphores because it is more self-documenting.
3. Always acquire the lock at the beginning of a method and release it right before the return: This allows for synchronization to be structured on a method by method basis.
4. Always hold the lock when operating on a condition: Condition variables are useless without a shared state, and the shared state should only be accessed while holding a lock.
5. Always wait in a while() loop: Using while holds the principle of consistent structure and gives you freedom without breaking modularity or portability.
6. (Almost) never use thread_sleep: There should always be another thread ready to execute after one is complete.