CS 162 Section 3

True/False

1. A thread needs to own a semaphore, meaning the thread has called `semaphore.P()`, before it can call `semaphore.V()`.
   False. Any thread with a reference to the semaphore can call V().

2. A thread needs to own the monitor lock before it can `signal()` a condition variable.
   True. A thread must acquire the monitor lock before it can access any of the monitor's state.

3. Changing the order of semaphores' operations in a program does not matter.
   False. This is a bit of a tricky one. We are trying to hint at the producer-consumer example from lecture. However, one can argue that it is true because semaphores are commutative (ie you can swap the order in which you call P() and V() between two threads).

Short Answer

1. Give two reasons why the following implementation of a condition variable is incorrect (assume that `MySemi` is a semaphore initialized to 0):

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

   Reason 1: Semaphores are commutative, while condition variables are not. Practically speaking, if someone executes MySemi.V() followed by MySemi.P(), the latter will not wait. In contrast, execution of Signal() before Wait() should have no impact on Wait().

   Reason 2: The above implementation of Wait() will deadlock the system if it goes to sleep on MySemi.P() (since it will go to sleep while holding the monitor lock).

2. Explain why semaphores can be used to do anything monitors are used for. If this is not true, explain why not.
   In Condition.java, a monitor is implemented with semaphores. We can simply remove that abstraction barrier and use semaphores directly.

Longer Answer

Semaphores

Implement the `P()` and `V()` methods of a `Semaphore` class backed by monitors (i.e. the `Lock` and `CondVar` classes). Neither of the methods should require more than five lines. Assume that monitors are Mesa-scheduled.

```
public class Semaphore {
    Lock lock; // every monitor has a Lock and CondVar
    CondVar c;
    Int value; // semaphores have an integer value
```

```
    public Semaphore(int initialValue) {
        value = initialValue;
        lock = new Lock();
        c = new CondVar(lock);
    }

    public P() {
        lock.Acquire();
        while (value == 0)
            c.Wait();
         value--;
        lock.Release();
    }

    public V() {
        lock.Acquire();
        value++;
        c.Signal();
        lock.Release();
    }
```

Producer and Consumer

Consider the following two functions implementing a producer and consumer by using monitors:

```
void send(item) {
    lock.acquire();
    enqueue(item);
    printf("before signal()\n");
    dataready.signal(&lock);
    printf("after signal()\n");
    lock.release();
}

item = get() {
    lock.acquire();
    while (queue.isEmpty()) {
        printf("before wait()\n");
        dataready.wait(&lock);
       printf("after wait()\n");
    }
    item = dequeue();
    lock.release();
}
```

a. Assume two threads T1 and T2, as follows:

```
T1              T2
send(item);     item = get();
```

What are the possible outputs if the monitor uses the Hoare implementation?

[T1, T2]
before signal
after_signal

[T2, T1]
before wait
before signal
after wait
after signal

b. Repeat question (a) for a Mesa implementation of the monitor.

[T1, T2]
before signal
after_signal

[T2, T1]
before wait
before signal
after signal
after wait

c. Now assume a third thread T3, i.e.,

```
T1                T2                 T3
send(item);       item = get();      send(item);
```

What are the possible outputs if the monitor uses the Hoare implementation? Please specify from which thread does an output come by specifying the thread id in front of the output line, e.g., [T1] before signal or [T2] after wait.

[T1, T2/T3]
[T1] before signal
[T1] after signal
[T3] before signal
[T3] after signal

[T3, T1/T2]
[T3] before signal
[T3] after signal
[T1] before signal
[T1] after signal

[T2, T1, T3]
[T2] before wait
[T1] before signal
[T2] after wait
[T1] after signal
[T3] before signal
[T3] after signal

[T2, T3, T1]
[T2] before wait
[T3] before signal
[T2] after wait
[T3] after signal
[T1] before signal
[T1] after signal