

CS162, Spring 2004
Section 5 Quiz 1
Steve Martin

1. (Easy) Consider the following code used to control bank transactions:

```
int withdraw(account, amount) {  
    balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    return balance;  
}  
  
int deposit(account, amount) {  
    balance = get_balance(account);  
    balance += amount;  
    put_balance(account, balance);  
    return balance;  
}
```

Every time someone uses an ATM, a thread is spawned at the central bank computer to handle the request. The accounts and balances are in a database that is shared among threads.

a) Is this a safe way to handle this? Why or why not?

No. The outcome is not deterministic when multiple threads access the same account simultaneously.

b) What are the possible balances returned if two threads try and withdraw x dollars from the same account with balance b at the same time?

$b-x$, $b-2x$, more if other threads are depositing to the same account, since that's not controlled either.

c) If you think there might be a problem with this code, how would you solve it and synchronize these methods (**without** using built in language features)?

Lock operations that touch the balance! (A couple ways to do this)

2. (Medium) The following code is intended to be placed in a global library, where multiple threads can potentially execute it concurrently.

```
int total;  
int grandTotal = -1;  
  
int myRoutine(int a, int b) {
```

```

int product, left, right;
product = a * b;
left = a;
right = b;
total = 0;

while (left > 0) {
    total += right;
    left--;
}

grandTotal = grandTotal + (total - product);

return grandTotal;
}

```

- a) Is this code safe? If so, argue informally but convincingly that it is. If not, modify it to make it thread safe. (You may use synchronization primitives, rearrange code, etc)

No, its not. The global variables pretty much assure this.

Two possible answers:

- i) Create a new local unassigned int retVal and replace the return statement with:*

```

retVal = grandTotal;
return retVal;

```

In addition, create a global mutex, lock it just before "total = 0;" and unlock it just before the (new) return statement.

- ii) Move the declaration of total to inside MyLibRoutine (i.e. make it local), then do the same as in 1, except put the lock/unlock only around the two statements affecting grandTotal.*

3. (hard) The goal of this exercise is for you to create a monitor with methods Hydrogen() and Oxygen(), which wait until a water molecule can be formed and then return. For example, if two threads call Hydrogen() and then a third calls Oxygen(), the third thread should wake up the first two threads and then they should all return.

- a) Specify the correctness constraints. Be succinct and explicit.

- *Each hydrogen waits to be grouped with an oxygen and one other hydrogen before returning.*
- *Each oxygen waits to be grouped with two hydrogens before returning.*

- *Only one thread accesses shared state at a time.*

Observe that there is only one condition any thread will wait for (that a water molecule can be formed). However, it will be necessary to signal hydrogen and oxygen threads independently. Consequently, we use two condition variables, waitingH and waitingO.

Define wH and wO to be the number of hydrogen and oxygen threads waiting in the monitor, and define aH and aO as the number of assigned threads waiting in the monitor. All variables are initialized to 0. Start with the following code:

```

Hydrogen() {
    wH++;
    lock.acquire();
    while (aH == 0) {
        if (wH >= 2 AND wO >= 1) {
            wH -= 2; aH += 2; wO -= 1; aO += 1;
            waitingH.broadcast();
            waitingO.signal();
        }
        else {
            lock.release();
            waitingH.wait();
            lock.acquire();
        }
    }
    lock.release();
    aH--;
}

Oxygen() {
    wO++;
    while (aO == 0) {
        if (wH >= 2 AND wO >= 1) {
            wH -= 2; aH += 2; wO -= 1; aO += 1;
            waitingH.signal();
            waitingH.signal();
        }
        else {
            waitingO.broadcast();
        }
    }
    aO--;
}

```

- b) Does Hydrogen() work, not work, or is it dangerous (works intermittently)? If it is broken, explain how to fix it.

This method is dangerous for several reasons:

- *Starvation and deadlock are possible. The lock should not be acquire()d and release()d around the wait() call-- the condition variable does that for you.*
- *State variables are modified outside of the critical sections.*

Changing broadcast() call to a signal() call would improve the efficiency of the solution.

c) How about Oxygen? If it is broken, explain how to fix it.

This method does not work.

- *The broadcast() call should be a wait().*
- *State variables are modified outside of the critical section. In fact, there is no critical section because the lock is not used.*

4. (Medium) A simple implementation of a lock is given below. Why must the interrupts be disabled in the **Release** procedure?

```
class Lock {
    int value = FREE;

    Acquire() {
        Disable interrupts
        if (value == BUSY) {
            put on wait Q
            sleep
            disable interrupts
        } else {
            value = BUSY;
        }
        Enable interrupts;
    }

    Release() {
        Disable interrupts;
        if (wait queue not empty) {
            take thread off wait Q
            add to ready Q
        } else {
            value = FREE;
        }
        Enable interrupts;
    }
}
```

If the release method did not disable interrupts, a thread could check to see if a thread needs to be taken off the wait queue and then be context switched out of the processor before releasing the lock. Another thread attempts to acquire the lock, finds that it is not available, adds itself to the waiting queue, and sleeps. When the initial thread wakes and releases the lock, there is a thread sleeping on the waiting queue with no way to wake it up

5. (Easy) Suppose we had the following situation:

- There is a box of donuts on the counter.
- Jim goes to the kitchen to eat a donut.
- Fred also goes to the kitchen to eat a donut.
- Bob the health nut doesn't eat donuts, but instead puts a single donut into the donut box every so often for his roomies to eat.

We (well, you) are going to write the code the threads that perform these tasks.

a) What are the correctness constraints?

*Needs to be a donut there before it can be taken out and eaten.
Needs to be enough room in the donut box to put donuts in.*

b) How would you write the pseudo-code for Jim and Fred?

(note that this doesn't take the size of the box into account)

Condition d;

Lock on donut box;

Acquire lock on donut box

Wait on d

Take a donut out of the box

Release the lock

c) How would you write the pseudo-code for Bob?

Acquire lock on donut box

Put a donut in the box

Signal on d

Release donut box lock.