Student Name:

CSCI U511 – Operating Systems

Homework-3Key, Weight: 40 points

Due on Tuesday, Oct 1, 2019 at the beginning of the lecture (Hard Copy)

Note: You need to include your calculation and details to receive full credit!

Please remember to write your name on your answer sheet.

Q1. For the following program, when thread_join returns for thread i, what is thread i's thread state? What is the minimum and maximum number of times that the main thread enters the WAITING state?

```
#define NTHREADS 10

thread_t threads[NTHREADS];

main() {

    for (i = 0; i < NTHREADS; i++)  thread_create(&threads[i],
&go, i);

    for (i = 0; i < NTHREADS; i++) {

        exitValue = thread_join(threads[i]);

        printf("Thread %d returned with %ld\n", i, exitValue);

    }

    printf("Main thread done.\n");

    return 0;

}

void go (int n) {

    printf("Hello from thread %d\n", n);

    thread_exit(100 + n);

    // REACHED?

   }
```

ANS: When join returns, thread i has finished running and excited. The runtime system saved the exit value in the TCB and moved the TCB to the finished list. The thread is thus in the FINISHED state.

Minimum is 0 and maximum is 10.

Student Name:

Q2. Show that solution 3 (discussed in lecture 8) to the Too Much Milk problem is safe—that it guarantees that at most one roommate buys milk.

Solution: This proof looks a lot like the proof given in the text for our second "solution" to the Too Much Milk problem. The only "trick" is to observe that thread 2's code looks a lot like the code for "solution" 2, so if we focus on thread 2's code here, the new proof will not need to change much at all. Of course, other solutions are possible.

Q3. Suppose that you mistakenly create an automatic (local) variable *v* in one thread *t1* and pass a pointer to *v* to another thread *t2*. Is it possible that a write by *t1* to some variable other than *v* will change the state of *v* as observed by *t2*? If so, explain how this can happen and give an example. If not, explain why not.

Solution: Yes. This can happen if t1 returns from the procedure that allocated v and then calls another procedure that allocates something else in the same place on the stack.

```
foo () {
        bar ();
        baz ();
}
bar () {
        Object v;
        sthread_t t2;
        sthread_init(\&t2, function, \&v);
}
baz() {
        int buffer[1000];
        int i;
        for (i=0; i<1000; i++) {
                buffer[i] = 42;
        }
}
```

Q4. You have been hired by a company to do climate modelling of oceans. The inner loop of the program matches atoms of different types as they form molecules. In an excessive reliance on threads, each atom is represented by a thread.

Your task is to write code to form water out of two hydrogen threads and one oxygen thread (H2O). You are to write the two procedures: **HArrives()** and **OArrives()**. A water molecule forms when two H threads are present and one O thread; otherwise, the atoms must wait. Once all three are present, one of the threads calls **MakeWater()**, and only then, all three depart.

You must use locks and Mesa-style condition variables to implement your solutions. Obviously, an atom that arrives after the molecule is made must wait for a different group of atoms to be present. There should be no busy-waiting and you should correctly handle spurious wakeups. There must also be no useless waiting: atoms should not wait if there is a sufficient number of each type.

Solution:
Since OArrives and HArrives are only trivially different, I provide pseudocode for just HArrives.

First, a lock should be used to protect all shared variables; students sometimes try to use a lock for each type of atom, and that rarely works.

Student Name:
Second, Mesa semantics with spurious wakeups means that all condition variables need to be in loop; they are waking up the thread they want, and so it does not need to wait. But it does!

Third, it is tricky with Mesa semantics to prevent bypassing (a thread arriving after MakeWater is called, that "grabs" the condition variable wakeup). Instead, create a condition variable per waiting thread to allow exactly the chosen thread to be woken up. (This is described in more detail in the implementation of FIFO Bounded Buffer and also semaphores given in Section 5.8.)

Fourth, if you don't use the approach given below, the program logic can easily be quite complex and will need to be very carefully thought out. It is fairly easy to devise a solution that works for some interleavings, yet causes useless waiting (e.g., two H's and one O waiting, no one makes water) given certain thread interleavings.

```
Status {
  Status::Status() { woken = FALSE; }
  public:
    bool woken; // initially false
    CV cv;      // initially empty
};

List<Status *> HQueue, OQueue;  // a list of waiting threads

int numH = 0; // number of waiting H's
int numO = 0; // number of waiting O's
Lock lock;

H::Arrives() {
    Status *h, *o, me;

    lock.Acquire();
    numH++;
    if (numH == 2 && numO >= 1) {
        // wakeup one of each
        h = HQueue.Remove();
        o = HQueue.Remove();
        h->cv.Signal();
        o->cv.Signal();
        h->woken = TRUE;
        o->woken = TRUE;
        numH -= 2;
        numO--;
        MakeWater();
```

Student Name:

```
        } else {
            // need to wait
            me = new Status;
            hQueue.Append(me);
            while (!me->woken) { // check for spurious wakeups
                me->cv.Wait(&lock);
            }
            delete me;
        }
        lock.Release();
    }
```