**Problem 5:** (26 pts; 20 minutes) A mini-bus shuttles between three stops around the university circle going in the same direction. Passengers wait at their stop until the bus arrives. The bus has seats for 10 passengers. When the bus reaches a stop, the passengers going down leave the bus, and the new passengers climb on the bus if there is room (assume they leave from different doors). The bus driver waits for all the passengers to climb, before moving to the next station. You may use semaphores, locks and condition locks.

(a) (13 points) Write pseudocode to simulate this bus operation.

(b) (13 points) Assume that the bus has room for 5 standing passengers as well. A passenger sits if a seat is available, otherwise she stands. If a seat becomes available later, she sits down (however, if her station comes up first, she just leaves the bus). Write pseudocode to simulate this situation.

First, write the pseudo-code for each thread without considering synchronization. It should look something like the following:

```plaintext
Bus
-------

while(1) {
    for (i=1; i <=3; i++) {
        wait for bus at from_station;
        drive to station i;
        pick up people from i;
        drop people destined to i;
    }
}

People
-------

int from_station, to_station;
wait for bus at from_station;
get on bus;
wait for bus to get to to_station;
get off bus
```

Now we can worry about adding in the rules via synchronization between the threads. The first rule is that passengers get on the bus only when the bus is at their station. The second rule is that they get off the bus only at their destination. Since multiple people may be waiting for each condition, if we use a semaphore, we dont know how many times to signal it when the bus gets there unless we keep track of the number of people. So, it seems more convenient to use condition variables and broadcast instead. This will come in handy later in adding the 10 seat condition. However, if we want the bus to wait until all people are on/off, then there is no escaping keeping count of people. If we do that, then we can use semaphores and mutexes. This is the approach that I decided to use.

```plaintext
Bus
-------

Semaphore to_station[3]; //one for people at each station; initially 0
Semaphore from_station[3]; //one for people on bus headed to station; initially 0
Semaphore going_down; //initially 0, to wait for people going down
Semaphore going_up; //initially 0, to wait for people going up
int count_from[i]; //count of people queued at a station
int count_to[i]; //count of people on bus headed to station
Lock from_mutex[i]; //mutex for the count_froms
Lock to_mutex[i]; //mutex for the count_tos
```
while(1) {
  for (i=1; i <=3; i++) {
    drive to station i
    to_mutex.Lock();
    for(j=0; j<=count_to_station[i]; j++){
      to_station[i].signal();
    }
    to_mutex.Unlock();

    from_mutex.Lock();
    for(j=0; j<=count_from_station[i]; j++){
      from_station[i].signal();
    }
    from_mutex.Unlock();

    going_down.wait();
    going_up.wait();
  }
}

Notice that the bus code signals all the people waiting on the bus (using the to_count[i] to keep track of that, with mutual exclusion), and then all the people waiting on the station to get on the bus. Later, we can use this to track how many seats are available on the bus and let at most that many people on the bus from the station. Now here is the updated people code.

People
-------

int from_station, to_station;
from_mutex[from_station].Lock();
from_count[from_station]++;
from_mutex[from_station].Unlock();
from_station[i].wait(); //wait on that semaphore get_on_bus

from_mutex[from_station].Lock();
from_count[from_station]--;
if(from_count[from_station]==0) //im last guy on
  going_up.signal(); //signal driver up guys done
from_mutex[from_station].Unlock();

to_mutex[to_station].Lock();
to_count[to_station]++;
to_mutex[to_station].Unlock();
to_station[to_station].wait();
get_down_from_bus;

to_mutex[to_station].Lock();
to_count[to_station]--;
if(to_count[to_station]==0)
    going_down.signal();
to_mutex[to_station].Unlock();
}
}

So, this ended up rather nasty–definitely more than is reasonable for the time budget. This is before including the 10 seat condition. The 10 seat condition requires keeping track of the number of empty seats, and signalling only that many people coming on at the station (the condition in the for loop). Also, it requires that we signal the bus on going_up when the seat count is reached, rather than when no people are waiting at the station.

Also, a nasty problem happens if a person gets on with to_station=from_station (going for a ride around the whole path). See if you can figure out the problem.

Second Problem we discussed—final problem in 99 A monkey community lives on an island that has only one coconut tree. The monkeys take turn going up the tree (which can hold only 3 monkeys at a time). After each monkey grabs a coconut, it climbs down the tree to go and eat it.

(a) Simulate this synchronization problem; each monkey is an independent thread.
(b) Modify your solution if necessary to ensure that monkeys can be climbing up, or down the tree, but not both ways at the same time.
(c) The dominant monkey does not like to wait. When it arrives, it climbs up the tree ahead of any monkeys that got there before it (there still can only be three monkeys up the tree). Implement the synchronization for this problem (show the procedure for the dominant monkey thread and extend the procedure for the regular monkeys). Is starvation possible?

Part (a) is pretty straightforward. Only a single semaphore is needed to keep track of how many monkeys are on the tree. So,

Semaphore get_on_tree; //initialized to 3

get_on_tree.wait();
climb_tree_and_get_coconut; // this can be just a print statement
get_off_tree;
get_on_tree.signal(); //signal another monkey that they can go up

For part (b), we have to make sure that monkeys only move in one direction at a time. This problem is identical to the draw bridge problem. We have to keep a direction variable, and wait until our direction is available. Condition variables seem promising here.

Semaphore get_on_tree; //initialized to 3
Condition waiting_up, waiting_down;
Lock mutex;
int count_up=0, count_down=0;

get_on_tree.wait();
mutex.Lock();
while(count_down!=0)
waiting_up.wait(mutex);
count_up++;
mutex.Unlock();
climb_tree;
mutex.Lock();
count_up--;
if(count_up==0)
    waiting_down.broadcast();
mutex.Unlock();

get_coconut; // this can be just a print statement

mutex.Lock();
while(count_up!=0)
    waiting_down.wait(mutex);
count_down++;
mutex.Unlock();
climb_down_tree;
mutex.Lock();
count_down--;
if(count_down==0)
    waiting_up.broadcast();
mutex.Unlock();
get_on_tree.signal(); //signal another monkey that they can go up

Part (c) is a little tricky. We can create a separate semaphore for the dominant monkies. However, somehow the two semaphores should share the token count. Which means that we have to maintain it manually as a variable with mutual exclusion. Then, we can check the dominant monkey count when getting off the tree and signal them first. In addition, for the non-dominant monkeys, we can wait on a condition based on dominant monkeys present or total monkeys on tree more than 3. The condition can be signalled whenever there are no more dominant monkeys.