COMP3300 *Homework # 4* **Total Points: 100**

Question 1:

```
class Lock {

    private int turn = 0;

    public void acquire(int tid) {
        while (turn == (1 - tid));
    }
    public void release(int tid) {
        turn = (1 - tid);
    }
}
```

1. **Yes, guarantees mutual exclusion; only the thread with tid matching turn is able to acquire the lock.**
2. **No, does not guarantee progress; if thread 0 never tries to acquire the lock (it is executing other code), thread 1 will not be able to acquire the lock.**
3. **Limitations (Not required): Only works with two processes, uses busy waiting.**

```
class Lock {

    private int turn = 0;
    private boolean lock[2] = {false, false};

    public void acquire(int tid) {
        lock[tid] = true;
        turn = 1 - tid;
        while (lock[1-tid] && turn == (1 - tid));
    }

    public void release(int tid) {
        lock[tid] = false;
    }
}
```

1. **Yes, guarantees mutual exclusion.**
2. **Yes, guarantees progress.**
3. **Limitations: Only works for two threads; involves busy-waiting.**

```
class Lock {

    public void acquire() {
       disableInterrupts();
    }
    public void release() {
       enableInterrupts();
    }
}
```

1. **Guarantees mutual exclusion on a uniprocessor, but not on a multiprocessor. On a uniprocessor, once the timer interrupt is disabled, the scheduler won't be able to switch to another process (assuming the scheduled process doesn't voluntarily relinquish the CPU). However, on a multiprocessor, it is possible for the other CPU to be running a process that also acquires the lock.**
2. **Yes, guarantees progress; once a process is scheduled, it is able to acquire the lock without incident.**
3. **Limitations: Only works on uniprocessors; allows user processes to disable interrupts for an arbitrary long period; cannot service other important interrupts during critical section (e.g., I/O); cannot schedule any other processes when lock is held, even those not contending for the lock.**

```
class Lock {

    private boolean lock = true;

    public void acquire() {
        while (TestAndSet(lock, true);
    }
    public void release() {
        lock = false;
    }
}
```

Note that the lock was initialized to the wrong value.

1. **Yes, guarantees mutual exclusion. It is not possible for more than one process to acquire the lock.**
2. **No, does not guarantee progress. No process is able to acquire the lock, since it is initialized to true, designating that the lock is already acquired.**
3. **Limitations: Not needed.**

Question 2:

```
class PizzaShop {
   public:
      void Enter();
      void Finish();
   private:
      int ticket;
      Semaphore mutex;              // control access to shared variables
      Semaphore servers;            // Sales people
 }
PizzaShop::PizzaShop(int n) {
   ticket = 0;
   mutex.value = 1;
   servers.value = n;               // Number of Sales people
}

PizzaShop::Enter() {
   mutex.Wait();
   ticket++;
   mutex.Signal();
   servers.Wait();
}

PizzaShop:: Finish() {
   servers.Signal();
}
```

Question 3:

Answer: The `signal()` operation associated with monitors is not persistent in the following sense: if a signal is performed and if there are no waiting threads, then the signal is simply ignored and the system does not remember that the signal took place. If a subsequent wait operation is performed, then the corresponding thread simply blocks. In semaphores, on the other hand, every signal results in a corresponding increment of the semaphore value even if there are no waiting threads. A future wait operation would immediately succeed because of the earlier increment.

Question 4:

The following solution uses three semaphores, *customers*, which counts waiting customers (excluding the customer in the shoe shiner chair, who is not waiting), *shoe_shiners*, the number of shoe shiners (0 or 1) who are idle, waiting for customers, and *mutex*, which is used for mutual exclusion. We also need a variable, *waiting*, which also counts the waiting customers. The reason for having *waiting* is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he/she stays; otherwise, he/she leaves.

When the shoe shiner shows up for work in the morning, he/she executes the procedure *shoe_shiner*, causing him/her to block on the semaphore *customers* because it is initially 0. The shoe shiner then goes to sleep. He/she stays asleep until the first customer shows up. When a customer arrives, he/she executes *customer*, starting by acquiring *mutex* to enter a critical region. If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released *mutex*. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he/she releases *mutex* and leaves without a shoe shine.

If there is an available chair, the customer increments the integer variable, *waiting*. Then he/she does a *Signal* on the semaphore *customers*, thus waking up the shoe shiner. At this point, the customer and the shoe shiner are both awake. When the customer releases *mutex*, the shoe shiner grabs it, does some housekeeping, and begins to shine shoe. When the shoe shine is over, the customer exits the procedure and leaves the shop. There is no loop for the customer because each one gets only one shoe shine (obviously for the pair). The shoe shiner loops, however, to try to get the next customer. If one is present, a shoe shine is given. If not, the shoe shiner goes to sleep.

```
#define CHAIRS 5               // # chairs for waiting customers
typedef int semaphore;         // use your imagination
semaphore customers = 0;       // # of customers waiting for service
semaphore shoe_shiners = 0;    // # of shoe shiners waiting for customers
semaphore mutex = 1;           // for mutual exclusion
int waiting = 0;               // customer are waiting (not being cut)
void shoe_shiner(void)
{
   while (TRUE) {
      wait(customers);  // go to sleep if # of customers is 0
      wait(mutex);             // acquire access to "waiting'
      waiting = waiting - 1;   // decrement count of waiting customers
      signal(shoe_shiners);    // one shoe shiner is now ready to cut hair
      signal(mutex);           // release 'waiting'
      cut_hair();              // cut hair (outside critical region
   }
}

void customer(void)
{
   wait(mutex); // enter critical region
   if (waiting < CHAIRS) {    // if there are no free chairs, leave
      waiting = waiting + 1;  // increment count of waiting customers
      signal(customers);      // wake up shoe shiner if necessary
      signal(mutex);          // release access to 'waiting'
      wait(shoe_shiners);     // go to sleep if # of free shoe shiners is 0
      get_shoe_shine();       // be seated and be served
   } else {
         signal(mutex);             // shop is full; do not wait
      }
}
```

Question 5:

```
enum Direction {North, South};
class Tunnel {
   public:
      Arrive(Direction dir);
      Depart(Direction dir);
   private:
      Lock lock;
      CVar goNorth, goSouth;
      int northWait;                        // waiting to go north
      int southWait;                        // waiting to go south
      int northBound;                       // going north in Tunnel
      int southBound;                       // going south in Tunnel
}
Tunnel::Tunnel() {
```

```
    lock.value = 1;                             // lock for shared variables is available
    northWait = 0;                              // no one waiting
    southWait = 0;                              // no one waiting
    northBound = 0;                             // no one in tunnel
    southBound = 0;                             // no one in tunnel
}

Tunnel::Arrive(Direction dir){
    lock.Wait();
    // If no one in the tunnel, car goes
    if ((northBound > 0) || (southBound > 0) {
        if (dir == North) {                     // if no one is waiting to go south,
                                                // car goes north
            if ((southWait > 0) || (southBound > 0)) {
                                                // otherwise, we count north
                                                //waiters, and wait for a signal
                northWait++;
                goNorth.Wait(lock);
                northWait--;
            }
        else {                  // if no one is waiting to go north, car goes south
            if ((northWait > 0) || (northBound > 0)) {
                southWait++;
                goSouth.Wait(lock);
                southWait--;
            }
        }
    }
    // count how many cars are in the tunnel
    if (dir == North)
        northBound++;
    else
        southBound++;
    lock.Signal();
}

// When a northbound car departs, we signal all southbound cars if any are waiting only
// if no northbound cars are in the tunnel. Similarly, when a southbound car departs.
// Note, in the Arrive routine above the car only waits if a car going the other
// direction is in the tunnel.

Tunnel::Depart(Direction dir) {
    lock.Wait();
    if (dir == North) {
        northBound--;
        if (southWait > 0) {
            if (northBound == 0)
                goSouth->Broadcast();
        }
    } else {
        southBound--;
        if (northWait > 0) {
            if (southbound == 0)
                goNorth.Broadcast();
        }
    }
    lock.Signal();
}
```