Rawan Alzowaid
CSCI-3753
Problem set 2

Q1:

```
lock mutexLock;
semaphore eastSem, westSem;
int eastCars = 0, westCars = 0;

go_east() {
    P(eastSem);
    P(mutexLock);
    eastCars++;
    if (eastCars == 1) {
        P(westSem); // lock the west side
    }
    V(mutexLock);
    V(eastSem);
    // code to cross the bridge
    P(mutexLock);
    eastCars--;
    if (!eastCars) {
        V(westSem); // release west side
    }
    V(mutexLock);
}

go_west() {
    P(westSem);
    P(mutexLock);
    westCars++;
    if (westCars == 1) {
        P(eastCars); // lock the east side
    }
    V(mutexLock);
    V(westSem);
    // code to cross the bridge
    P(mutexLock);
    westCars--;
    if (!westCars) {
        V(eastSem); // release east side
    }
    V(mutexLock);
}
```

Rawan Alzowaid
CSCI-3753
Problem set 2

Q2:

- ➔ Using pseudocode

```
lock mutex_lock;
condition c;
int t1_complete = 0;

// Task T1
Acquire(mutex_lock);
// execute code c1
t1_complete = 1;
Release(mutex_lock);
c.signal();
// End Task T1

// Task T2
Acquire(mutex_lock);
while (!t1_complete) {
    Release(mutex_lock);
    c.wait();
    Acquire(mutex_lock);
}
// execute code c2
Release(mutex_lock);

// End Task T2
```

Q3:

The 3rd solution to the Readers/Writers problem is starvation free because when the writing thread wants to enter into the critical section for writing data, future arrivals of reading threads will have to wait until the writing thread finishes writing while currently reading threads can continue to read. This prevents writer threads from being starved. On the other hand, reading threads are not starved since multiple readers are allowed to access the critical section as long as there are no writer threads.

Q4:

a) No, the swap() function is not thread-safe . Suppose there were two threads (A and B) that are calling the swap() function. If A enters swap() and halts execution after "*y = *z" either by interruption or a context switch, the consequence is now that half of a swap has occurred and the reference to y is now lost. If a context switch occurred in the middle of this swap, the value of temp would be overwritten and the value of y in thread A would be lost, if A were to resume execution (context switched back) it would essentially use the wrong value of temp.

b) Yes, swap() function is reentrant because there is no use of a semaphore, monitor, or lock whatsoever in the swap function anymore. Therefore calling this ISR function would pose no risk of causing deadlock and it would execute normally.

Rawan Alzowaid
CSCI-3753
Problem set 2

Q5:
Memory Sharing IPC would be ideal for this situation since we have a large file (10 MB) that we want to pass from one process P1 to another process P2. Process P1 could first read the file into main memory, then wake process P2 while passing the pointer to memory where the file is stored and wait asynchronously for P2 to complete. Once process P2 wakes, it will synchronize access to that memory and begin to compress the data in that memory location. Once the compression finishes, P2 will release the lock to memory, send an interrupt to P1 to alert that the file has finished compression, and resume sleeping.

Rawan Alzowaid
CSCI-3753
Problem set 2

Q6:

```
monitor SomeMonitor {
   condition c1, c2, c3;
   private int v1, v2, v3;

   public function increment() {
      c1.wait();
      v1++;
      c1.signal();
   }

   public function decrement() {
      c1.wait();
      v1--;
      c1.signal();
   }

   public function square() {
      c2.wait();
      v2 *= v2;
      c2.signal();
   }

   public function squareRoot() {
      c2.wait();
      v2 = sqrt(v2);
      c2.signal();
   }

   public function sin() {
      c3.wait();
      v3 = sin(v3);
      c3.signal();
   }

   public function cos() {
      c3.wait();
      v3 = cos(v3);
      c3.signal();
   }
}
```