Ryan Margono
Homework 2

1.
Unlike semaphores, conditional variables require some sort of condition like a flag, and a mutex that protects the condition.
A semphore doesn't require a condition, but rather can be locked and unlocked in order to allow mutual exclusion.
Semaphores also have memory while conditional variables do not.

2.
A mutex protects a piece of memory from race conditions by only allowing one thread to alter it at a time. They need to provide mutual exclusion, don't cause deadlocks, and doesn't starve threads. These must be true in all other sync primitives since they utilize mutices.

3.
```
#define MAX_THREADS 10

static mutex mut;
static int total_sum = 0;
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;

int aggregateStats(double stat_a, double stat_b, double stat_c)
{
     mut.lock():
     sum_stat_a += stat_a;
     sum_stat_b -= stat_b;
     sum_stat_c -= stat_c;
     mut.unlock();
}
void init(void) {
    pthread threads[MAX_THREADS];
    // args (stats) come in someway
    for (int i=0; i<MAX_THREADS; i++) {
```

```
        pthread_create(&threads[i], NULL, aggregateSum, (void
*)&args);

    }
    for (int i=0; i<MAX_THREADS; i++) {
          pthread_join(threads[i], NULL);
    }
    total_sum = sum_stat_a + sum_stat_b + sum_stat_c;
}
```

4.

```
#define MAX_THREADS 10
using namespace std;

static mutex mut_a;
static mutex mut_b;
static mutex mut_c;
static int total_sum = 0;
static int sums[MAX_THREADS] = {0};
static int current_thread = 0;
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;

void aggregateSum(double stat_a, double stat_b, double stat_c) {
    struct arg_struct *args = arguments;
    aggregate_a(arguments -> a)
    aggregate_b(arguments -> b)
    aggregate_c(arguments -> c)
}

int aggregate_a(double stat) {
    unique_lock lock(mut_a);
    sum_stat_a += stat_a;
}
int aggregate_b(double stat) {
```

```
    unique_lock lock(mut_b);
    sum_stat_b += statb;
}
int aggregate_c(double stat) {
    unique_lock lock(mut_c);
    sum_stat_c += stat_c;
}
void init(void) {
    pthread threads[MAX_THREADS];
    // args (stats) come in someway
    for (int i=0; i<MAX_THREADS; i++) {
        pthread_create(&threads[i], NULL, aggregateSum, (void
*)&args);
    }
    for (int i=0; i<MAX_THREADS; i++) {
    pthread_join(threads[i], NULL);
    }
    total_sum = sum_stat_a + sum_stat_b + sum_stat_c;
}
```

5.

The Lost Wakeup problem is when a thread fails to go to sleep, therefore missing its wakeup call, therefore the signal is lost. It happens when a lock is not held while testing the condition of a condition variable.

One scenerio is as follows: a thread sends a conditional variable signal, another thread is in the process of testing that condition, and the signal is lost.

It can be fixed by having a mutex lock the condition.

6.

a)

```
template <class T> class ThreadSafeListenerQueue {
private:
    mutex mut;
    condition_variable data_cond;
```

```
public:
    list<T> data;
    ThreadSafeListenerQueue() {
        data = list<T>();
    }
    bool push(const T element);
    bool pop(T& element);
    bool listen(T& element);
    bool empty() const;
};

template <class T> bool ThreadSafeListenerQueue<T>::push(const T
element) {
    lock_guard<mutex> lk(mut);
    data.push_back(element);
    data_cond.notify_one();
    return true;
}

template <class T> bool ThreadSafeListenerQueue<T>::pop(T
&element) {
    lock_guard<mutex> lk(mut);
    if (data.empty()) {
        return false;
    }
    element = data.front();
    data.pop_front();
    return true;
}

template <class T> bool ThreadSafeListenerQueue<T>::listen(T
&element) {
    unique_lock<mutex> lk(mut);
    data_cond.wait(lk, [this]{return !data.empty();});
    element = data.front();
    data.pop_front();
```

```
    return 0;
}
```

b)

```
template <class T> bool ThreadSafeListenerQueue<T>::listen(T 
&element) {
    unique_lock<mutex> lk(mut);
    while(data.isEmpty()) sleep(1)2w2w
    element = data.front();
    data.pop_front();
```