1.Warm-up: As we discussed in class, explain at least two important differences between semaphores and condition variables.

1.Semaphores have an internal state stored in memory while condition variables don't. For example, even when sem_wait() is not called, sem_post() can increment the semaphore, and therefore this signal is stored in that internal state. On the other hand, if signal() is called when there's not a waiting thread, this signal() will be lost. So if after this wait() is called, it will wait without knowing that signal() was called before that, until another signal() is called.

2.Condition variables can signal to wake up one thread using signal() or all waiting threads using broadcast() while semaphores only wake up 1 waiting thread

2. What criterion or criteria must a correct mutex fulfill? Without looking it up, reason out which of these criteria (and/or possibly others) must be true for other thread synchronization primitive (namely condition variables and semaphores), and explain why.

Mutual Exclusion
Deadlock free
Starvation free

3. Use a single pthread mutex or std::mutex to make this function thread-safe. Add global variables, and content to the init() function, iff necessary.

```
#include "pthread.h"
pthread_mutex_t lock;
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) {
        pthread_mutex_lock(&lock);
        sum_stat_a += stat_a;
        sum_stat_b -= stat_b;
        sum_stat_c -= stat_c;
        pthread_mutex_unlock(&lock);
        return sum_stat_a + sum_stat_b + sum_stat_c;
}
void init(void) {
        pthread_mutex_init(&lock, NULL);
}
```

4. Let's make this more parallelizable. We always want to reduce critical sections as much as possible to minimize the time threads need to wait for a resource protected by a lock. Modify the original code from question 3 to make it thread-safe, but use three mutices this time, one each for sum_stat_a, sum_stat_b, and sum_stat_c. Hint: explain what guarantee(s) you provide about the return value of aggregateStats().

```
#include "pthread.h"
```

```
pthread_mutex_t lock_a;
pthread_mutex_t lock_b;
pthread_mutex_t lock_c;
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) {
        pthread_mutex_lock(&lock_a);
        sum_stat_a += stat_a;
        pthread_mutex_unlock(&lock_a);

        pthread_mutex_lock(&lock_b);
        sum_stat_b -= stat_b;
        pthread_mutex_unlock(&lock_b);

        pthread_mutex_lock(&lock_c);
        sum_stat_c -= stat_c;
        pthread_mutex_unlock(&lock_c);

        return sum_stat_a + sum_stat_b + sum_stat_c;
}
void init(void) {
        pthread_mutex_init(&lock_a, NULL);
        pthread_mutex_init(&lock_b, NULL);
        pthread_mutex_init(&lock_c, NULL);
}
```

5. In your own words, describe one of the possible causes of the Lost Wakeup Problem, including a scenario that triggers this cause, and how to fix it. Use C++ or pseudocode in your explanation iff you find it necessary. If you do research and find other causes beyond what we discussed in class, please note your source(s).

Lost Wakeup Problem is when a thread misses the signal() call to wake up and thus continues to wait. One of the possible causes of this problem is that we went to sleep without being enqueued into the queue of threads waiting. Therefore, we would never be woken up. A scenario that triggers this if you call signal_one() in enqueue and dequeue to go from an empty queue to a non-empty queue. Here you have a bunch of threads already waiting for the queue to be non-empty. Let's assume our queue gets filled to 2 elements, since signal_one() only sends one signal to wake up one thread, the second thread will not be signaled and will stay asleep even though it's supposed to be woken up. The solution is to call broadcast() to wake up all threads instead of just one and/or call signal() every time you enqueue an element. This way, there won't be any lost thread when the queue goes from empty to non-empty.

6. Using any pseudocode style you prefer, outline the functionality of a thread-safe queue twice, with at least the methods:

- push(item): Add item to the end of the queue
- pop(): Remove item from the beginning of the queue and return it
- listen(): wait for and then return an item when available

these two ways:

(a) Built using mutices, semaphores, and/or condition variables, and then
(b) Built only using semaphores.

a)
```
pthread_cond_t empty;
pthread_cond_init(&full, NULL);

push(item) {
    Pthread_lock(queue_lock);
    If (empty) {
        head = item {
        tail = head;
    Else {
        tail.next = item;
    empty = false;
    pthread_cond_signal(empty);
    pthread_unlock(queue_lock);

pop() {
    pthread_lock(queue_lock);
    while(empty) {
        pthread_cond_wait(empty, queue_lock);
    }
    new_item pointer = head;
    head = head.next;
    return new_item;
    pthread_unlock(queue_lock);

listen() {
    while(empty) {
        pthread_cond_wait(empty, queue_lock);
    }
    new_item object = tail;
    return new_item;
```

b)

```
    sem_t num_elem;
    sem_t mutex
    sem_init( &num_elem, 0, 0 );
    sem_init(&mutex, 0, 1);

    push(item) {
        sem_wait(mutex);
        If (!num_elem) {
            head = item {
            tail = head;
        Else {
            tail.next = item;
        sem_post(num_elem);


    pop() {
        sem_wait(mutex);
        new_item pointer = head;
        head = head.next;
        return new_item;

        sem_wait(num_elem);
```