1. Warm-up: Provide the minimal code for implementing the lock() and unlock() methods of a mutex with a counting semaphore, wherein you assume the program that calls this implementation always calls unlock() only when the mutex is locked and only via the thread that locked it.

```
void lock() {
        s.down();
}

void unlock() {
        s.up();
}
```

This is enough because we assume the lock is only unlocked when the mutex is locked and only the thread that locked it will unlock the thread. We don't need to have another semaphore to create a critical section where we use a boolean and a reference to keep track of if the lock is being locked and who locked it.

2. 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.

The most commonly talked scenario where the lost wakeup problem might occur is when two consumers and a producer are using a concurrent queue. The queue supports pop which blocks the caller if it is empty by a conditional variable and push which pushes data into the queue and notify a consumer if and only if the size gets from 0 to 1. The conditional variable is associated with a boolean isEmpty. Assume two consumers start to listen on the queue and are all blocked since the queue is initially empty. And the producer wants to push two pieces of data to the queue. Ideally, these two consumers should happily receive one piece of data and return from the pop function, but it might not work all the time.
Consider the case where the producer pushes one piece of data into the queue, and since the queue now has size of one, it notifies a consumer that's waiting with the conditional variable. But immediately after it sends out the signal, the producer becomes lucky and pushes the second piece of data into the queue before the first consumer pops the first piece of data out. This would become a problem as when the second piece of data gets pushed in, the size changes from one to two, and no signal would be fired, thus the other consumer will not be able to wake up and pop the second data out.

One easy way to solve this problem is to not only send signal when the queue turns from empty to size one but send it every time a new piece of data is pushed in. Thus, even if the producer is quicker than the first consumer, it will then still wake up the second consumer when

the second piece of data is pushed in.

3. If a machine has 4 cores, how many threads should your program spawn? What characteristics of the hardware, your program's threads, or the other software running on this machine would change the answer?

We should spawn 4 threads if we have 4 cores normally. Because having more threads than the number of cores will not result in true simultaneous execution and context switching, scheduling, spawning and destroying threads, would cost some performance. However, when there is a particular interest that can be enhanced by having more threads, it is important to do so. For example, when a thread is used to deal with IO, then two threads, one running the main server logic and the other one trying to read from or
write to a disk, would be more ideal than having the main server suspend while waiting for the read or write to finish. When there are other programs running on the same computer, then the number of threads might need to diminish. When the need of having more than four jobs execute at the same time becomes necessary, we can spawn more than four threads but only at most four threads will be running truly simultaneously but all spawned threads will run alternately, making all of them progress somehow in accordance. Besides, depending on whether the hardware supports hyper-threading, if so, the program can utilize more threads more efficiently.

4. Consider the following code:

```
1 static double sum_stat_a = 0;
2 static double sum_stat_b = 0;
3 static double sum_stat_c = 1000;
4 int aggregateStats(double stat_a, double stat_b, double
stat_c) {
5       sum_stat_a += stat_a;
6       sum_stat_b -= stat_b;
7       sum_stat_c -= stat_c;
8       return sum_stat_a + sum_stat_b + sum_stat_c;
9 }
10 void init(void) { }
```

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

we can add a global mutex called lock:

```
pthread_mutex_t lock;
```

and initialize it in the init method:

```
pthread_mutex_init(&lock, NULL);
```

before line 5 and after line 4, we can require the lock:

```
pthread_mutex_lock(&lock);
```

we can change line 8 to as follows:

```
double total = sum_stat_a + sum_stat_b + sum_stat_c;
pthread_mutex_unlock(&lock);
```

```
        return total;
```

5. 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.

```
        static double sum_stat_a = 0;
        static double sum_stat_b = 0;
        static double sum_stat_c = 1000;

        pthread_mutex_t alock;
        pthread_mutex_t block;
        pthread_mutex_t clock;

        int aggregateStats(double stat_a, double stat_b, double stat_c) {
                double total = 0;

                pthread_mutex_lock(&alock);
                sum_stat_a += stat_a;
                total += sum_stat_a;
                pthread_mutex_unlock(&alock);

                pthread_mutex_lock(&block);
                sum_stat_b -= stat_b;
                total += sum_stat_b;
                pthread_mutex_unlock(&block);

                pthread_mutex_lock(&clock);
                sum_stat_c -= stat_c;
                total += sum_stat_c;
                pthread_mutex_unlock(&clock);

                return total;
        }

        void init(void) {
                pthread_mutex_init(&alock, NULL);
                pthread_mutex_init(&block, NULL);
                pthread_mutex_init(&clock, NULL);
        }
```

6. What criterion or criteria must a correct mutex fulfill? Which of these do you think must be true for
a general thread synchronization primitive (e.g., barriers, mutices,

other types of locks, condition
variables, etc.)?

a. Only one thread can hold it at a time.
b. Only the one who requires it can release it.
c. There should be no deadlocks.
d. The fairness should be guaranteed so that no thread gets starved.

7. What’s busy waiting? What’s good and bad about it?

Busy waiting is the process of continuously checking if a condition has been achieved. For example, checking if a boolean variable has been set to true from false.

Busy waiting is bad because it keeps running and doesn't do useful things except checking for validity of a condition, which takes CPU resources and can be potentially wasteful since it blocks other useful works to be processed. However, busy waiting can sometimes be beneficial. Since context-switching is somehow expensive, if the waiting time is expected to be relatively short, simply making it do several loops would be better than making it sleep and waking it up later. And due to its simplicity, it is not as error prone as other more advanced tricks and it could be really helpful during early stage of development or the experiment of certain algorithms or protocols where the time it saves and correctness it guarantees is more important than what it might cost us.