1. Explain (in your own words!) how Peterson's Algorithm works. Why does it need to be re-written for more than 2 threads? For an extra point or two, feel free to explore exactly *what* changes you would need to make.

Peterson's algorithm provides guaranteed mutual exclusion by using only the shared memory. The idea is that first a thread(T1) expresses its desire to acquire lock and sets flag[T1] = true and then gives the other thread(T2) a chance to acquire the lock. If T2 desires to acquire the lock, then, it gets the lock and then passes the chance to T1. If it does not desire to get the lock then the while loop breaks and the T1 gets the chance. It needs to be rewritten for more than 2 threads for handling the flags. The condition in lock() will instead check if at least one flag of other threads is true && victim == me then block.

```
int  victim;
bool flag[2] = { false, false };

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    victim = me
    while (flag[other] && victim == me) {
     no-op;
    }
}
void unlock(void) {
     int me = my_tid % 2
     flag[me] = false
}
```

2. Running on some specific machine *X* with set hardware and software, how many threads should your program spawn, assuming that it has enough parallel tasks to run to occupy many threads? Think about how the hardware (eg, memory and number of cores) and software (ie, programs already occupying some resources) affect the number of threads you should spawn. Don't overthink this one!

The optimal number of threads depends on the number of processors available and the nature of the tasks on the work queue.

For tasks that may wait for I/O to complete, we would want to increase the thread number beyond the number of available processors, because not all the threads will be working at all times. We can estimate the ratio of waiting time (WT) to service time (ST) for a typical request. If we call this ratio WT/ST, for an N-processor system, we want to have around N*(1+WT/ST) threads to keep the processors fully utilized.

3. If multiple on-die caches (eg, L2 and L3 cache) are higher-latency than an L1 cache, why do we have them anyway? (Hint: how big and slow is main memory? How big and slow is the L1 cache?)

L1, L2, L3 cache are some specialized memory which work together to improve computer performance. When a request is made to the system, CPU has some set of instructions to execute, which it fetches from the RAM. Thus to cut down delay, CPU maintains a cache with some data which it anticipates it will be needed.

L1 Cache(2KB - 64KB) - Instructions are first searched in this cache. L1 cache very small in comparison to others, thus making it faster than the rest.

L2 Cache(256KB - 512KB) - If the instructions are not present in the L1 cache then it looks in the L2 cache, which is a slightly larger pool of cache, thus accompanied by some latency.

L3 Cache (1MB -8MB) - With each cache miss, it proceeds to the next level cache. This is the largest among the all the cache, even though it is slower, it is still faster than the RAM.

4. To the level of detail you feel is necessarily, explain (a) what the lost wakeup condition is in the context of a thread-safe condition variable implementation (or use) is, (b) and how to avoid it. Use C++ or pseudocode in your explanation iff you find it necessary.

In multi-threaded programming, the thread that misses a wake-up would sleep forever and do nothing. Take an implementation of thread-safe queue for example, if the pop() method always block until the queue is not empty, and the push() method only signal one thread that it is no longer empty when the size of the queue goes from 0 to 1, it will cause the lost wakeup condition problem.

Say that we have two threads(T1, T2) calling pop() in each of them, and one thread(T3) calling push(1), push(2). At the beginning of the program, T1 and T2 are waiting because the queue is empty. And then T3 push(1) and signal T1 that the queue is not empty, and continuously push(2) makes the queue size become 2 without sending another not empty signal to T2. As a result, T1 will perform pop() but T2 will block forever.

The main problem is that push() only signal when the size of queue changes from 0 to 1. To avoid this problem, we can change the condition in push() to send signal of “not empty” to all the threads every time when there is an element pushed into the queue.
In addition, an alternative way is to have a “timed wait” in pop() method that would force a thread to wait up after a period of time to check for the condition no matter it has been signaled or not.

5. Consider the following code:

```
1     static double sum_stat_a = 0;
2     static double sum_stat_b = 0;
3     static double sum_stat_c = 10000;
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.

```
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 10000;
static std::mutex mu;

int aggregateStats(double stat_a, double stat_b, double stat_c) {
      mu.lock();
      sum_stat_a += stat_a;
      sum_stat_b += stat_b;
      sum_stat_c -= stat_c;
      mu.unlock();
      return sum_stat_a + sum_stat_b + sum_stat_c;
}
void init(void) { }
```

6. 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 = 10000;
static std::mutex mu_a;
static std::mutex mu_b;
static std::mutex mu_c;

int aggregateStats(double stat_a, double stat_b, double stat_c) {
     mu_a.lock();
     sum_stat_a += stat_a;
     mu_a.unlock();

     mu_b.lock();
     sum_stat_b += stat_b;
     mu_b.unlock();

     mu_c.lock();
     sum_stat_c -= stat_c;
     mu_c.unlock();
     return sum_stat_a + sum_stat_b + sum_stat_c;
}
void init(void) { }
```