1. Why does adding a sentinel simplify a concurrent queue? How does it make it possible to construct a lock-free queue?

Because adding a sentinel would eliminate the case where there is no node in the queue and both head and tail point to null pointer. In this case, we can write header and tail to, in one Enqueue or Dequeue, distinct parts of the node.

In x86 architecture, reading and writing to a pointer (data with size smaller or equal to a cache line) would be atomic. Adding a sentinel would decouple head and tail pointers so that Enqueue and Dequeue would check and update either head or tail at a time. Also since we can use CAS, we can make sure that the reference we're holding is the most recent one, thus making our operating atomic.

2. Consider the performance of a program responsible for taking a set of 10,000 large numbers (e.g., between 1e20 and 1e40), and determining the prime factors of each number. A user has a four-core machine, and wants to run this program in the background while they play a fast-paced game (that spawns two threads, one for the game logic and one for rendering) at the same time.

a. What kind of bounds (minimum and maximum) can you give on the number of threads the prime factor program should spawn? Why not fewer or more?

Minimum is trivially one because you need one thread to run the program even though it might take a long time to terminate.
Maximum is two because there is another program that uses two threads to play a game which is really important and should be granted with two cores most of the time. If we spawn more than two threads, then the two threads used by this game might not always be executing, and some context switches might let these two threads to rest and thus make this game run less fluently.

b. What performance metric(s) would you as a programmer use? What would you optimize for in this situation, not just in your own program, but system-wide?

I would choose throughput metrics. For the application program, if the algorithm is randomized, we can try to use deterministic algorithms with minimal number of clock cycles. If the algorithm is randomized, we can try to make the difference caused by randomness as small as possible. Increasing the clock rate and adding more processors is also a good way to improve the performance. Recompile this program and run on a different ISA where the average CPI is smaller.

3. Consider the following two threads accessing shared global variable global_thread. (a) What kind(s) of concurrency bug(s) are present in this code? (b) Describe (ie, don’t just list) two possible ways to fix

this. If it helps, use code to demonstrate why these are good fixes.

a) It will be an atomicity violation bug. Because it is possible an interleaving that right after the if check in thread 1 is satisfied, thread 2 nullifies the pointer, and then fputs will dereference the pointer which raises a segmentation fault.
b) The simplest way of solving this is to use a mutex to wrap the section both in thread 1 and 2. So that the if check and fputs become one step, and the nullification in thread 2 will not be able to happen in between the check and fputs method. Another way to change this is to declare a local variable that stores the value of thr->proc_info and in this case, even though nullification might happen in between, the local variable still holds the address of the actual data regardless of fact that pro_info points to NULL right now.

In thread 1:

```
auto local_ref = thr->proc_info;  /* use auto without loss of
generality */
if (local_ref) {
        fputs(local_ref);
}
```

In thread 2:

```
thr->proc_info = nullptr;
```

4.Regarding Compare And Swap (CAS):

a. Why do we want to try to make lock-free algorithms with CAS instead of mutices or similar synchronization primitives?

Because first, CAS is faster than mutices or other synchronization primitives as it is implemented using low-level hardware instructions. And then, CAS needs no assumptions other than the fact that the target architecture has an implementation of it somehow. But other primitives might have some assumptions or at least check those assumptions when using them. For examples, only those who required a mutex can release it. Third, other primitives have to consider fairness rules explicitly whereas CAS's way of dealing with contention is solely based on which thread executes first (total ordering determined by hardwares). Fourth, problems would occur if the thread who holds the lock somehow failed, and the lock would never be released, causing a problem that takes extra consideration to solve, especially when the system needs certain feature like fault-tolerance.

b. Review our CAS-based queue enqueue() method, and to the best of your ability, explain why we need to allow the tail pointer to point to either the last or second-to-last entry in the queue instead of only the last one.

The problem is that we need two steps to successfully enqueue a new node: 1. we want to create it and set the tail's next pointer to it. 2. we want to set the tail pointer to the new node. But one CAS can only guarantee atomicity of one step (one check and one assignment operation), hence unable to make two steps atomic. That means, even though the first step will be atomic, it is not guaranteed that these two steps altogether will be atomic, which would result in certain interleaving that causes unexpected results. For example, thread A might finish step 1, and now tail's next is pointing to node A (new node created by thread A). Right after that, thread B comes in and finishes step 1, at which point the tail's next is pointing to node B rather than node A. At this moment, A is a floating node that no other nodes point to but the tail is still the old tail (doesn't change), which makes A's second step's requirement fulfilled. And after this, tail will point to A which is no longer the real "tail". The problem occurs because we assumed two CASs will execute atomically but actually not. Thus, loosing our invariance from pointing only to the last node to pointing to either the last or the second-to-last is the way to solve this problem. And this won't make our algorithm incorrect because if the tail is "unfortunately" pointing to the second-to-last, the next operation on the queue will try to fix this, making the tail point to the last node before any further operation is executed.

Think about the properties of CAS, what we talked about in class, and the slides. Show how you think: if you still don’t quite understand it from class, you may want to explain your thought process as you try to work through examples of how two threads might interleave their enqueue() operations (not just the one explained in the slides). You should come up with some fundamental deductions about this and other CAS-based algorithms and the global states visible to other threads in a CAS-based algorithm that wouldn’t be visible with mutex-based critical sections. Again: when in doubt, show your work!

5. What are the advantages and disadvantages of binary instrumentation versus dynamic binary translation?

Binary instrumentation can inject the instrumentation code into the binary executable before it runs on certain platform whereas dynamic binary translation uses an emulator to do the job at runtime. Dynamic binary translation can take a "block" of code, translate it, and caches the result as it executes. And therefore, programmers can pause the execution of the program and start to inspect on aspects of the state at various points at runtime. Static binary instrumentation is harder to use correctly because it is hard to predict the behavior of the program before it actually runs, but since it is injected into the binary before it runs, it will run really fast. Dynamic binary translation can do more flexible things as it can actually see what is running such as examining the memory, the register values, and etc.

But the downside of it is that it needs to do extra work as the emulator is running the binary, it might be less efficient and runs slower than static binary instrumentation. The difference and pros and cons between binary instrumentation and dynamic binary translation is like that between interpreted language and compiled language.