Question 1. *(15 points) You are implementing a multilevel feedback queue (MLFQ) scheduler and are deciding what values to use for the time slice at each priority level. Your friend suggests simplifying the design by just using the same time slice value for each level. Is this a good or bad suggestion? Explain why. You may find it useful to consider what would happen using a uniform time slice value t if you assume that any given process always spends exactly k time doing computation before releasing the CPU (but k may be different for each process).*

Question 2. *(10 points) Explain whether user-level or kernel-level threads would be better suited to each of the following applications and why. Assume that you are running on a uniprocessor machine.*

1. *A weather modeling system in which large numbers of threads perform continuous computation.*
2. *A file sharing application in which each user is allocated a thread that reads and writes files stored on disk.*
3. *Suppose that you are running on a multiprocessor machine instead of a uniprocessor machine. Would your answer to parts 1 and/or 2 change? Why or why not?*

Question 3. *(10 points) In the context of the Too Much Milk synchronization problem, explain how we decided whether possible solutions worked (i.e., what were the general requirements of any solution to the problem)?*

Question 4. *(10 points) From our final implementation of a test&set-based lock, the final step of a thread attempting to acquire a lock that's already held consists of setting the guard variable to zero and putting the thread to sleep. Explain why it is necessary to have these two operations occur atomically – i.e., what problem could occur if a context switch occurred after setting the guard to zero but before putting the thread to sleep? Be specific.*

Question 5. *(10 points) In the producer/consumer problem, explain why the call to* `wait` *in the* `remove` *method must be wrapped in a* `while` *loop instead of a regular conditional.*

Question 6. *(15 points) Suppose you are implementing a multi-threaded application where Thread A needs to execute a function* `foo` **before** *a concurrent Thread B executes a function* `bar`*. In class, we discussed how to do this using a semaphore: set the initial* `sem = 0`*, then in Thread A, do* `foo(); sem.signal();` *while in Thread B, do* `sem.wait(); bar();`*. Implement this same scheduling constraint using a lock and a condition variable instead of a semaphore. Define your shared variables, which should include at least* `lock` *and* `cv` *(you may want more than just these), and then write code snippets for Threads A and B that interact with the shared variables and call* `foo` *and* `bar`*, respectively, ensuring that* `foo` *is called before* `bar`*.*