CMPU-224 Lab13 Quiz Solutions

Spring 2026

Name: _____

This is a closed book, closed notes quiz. No electronic devices are allowed. You have until 3:30pm to complete the quiz. There are a total of 5 questions and 10 points.

There should be enough space on the quiz for your answers. If you need more space to work out a problem, blank paper will be available, just ask.

If you finish with time remaining, raise you hand and I will come and collect your quiz. You may then work on the lab assignment.

Good Luck!

1. (2 points) A thread calls `pthread_cond_wait(&c, &m)` while holding mutex `m`. Which statement most accurately describes what happens to `m`?

   A. The mutex stays locked the whole time the thread sleeps.

   B. The mutex parameter is only used when debugging the threads in `gdb`.

   C. The call atomically releases `m` and sleeps, then re-acquires `m` before returning.

   D. The thread spins on `m` in a busy loop until the condition is signaled.

   Solution: C. `pthread_cond_wait` releases the mutex and sleeps as a single atomic step, which is exactly why a separate `unlock` then `wait` cannot be used: the signal could arrive in the gap and be lost. On wakeup the thread re-acquires the mutex before returning, so the caller is back in the critical section when control resumes.

2. (2 points) The canonical pattern around `pthread_cond_wait` is

```
pthread_mutex_lock(&m);
while (!predicate)
    pthread_cond_wait(&c, &m);
/* predicate is true here */
pthread_mutex_unlock(&m);
```

   Why must the check be a `while` loop rather than a single `if`?

   A. `pthread_cond_wait` is non-blocking and must be invoked repeatedly until the predicate becomes true.

   B. A thread returning from `pthread_cond_wait` is not guaranteed that the predicate still holds, so it must be re-checked before proceeding.

   C. `pthread_cond_signal` is unreliable on Linux and must be retried until it succeeds.

   D. The loop ensures fairness: without it, one thread could monopolize the condition variable and starve the other waiters.

Solution: B. A signal is only a hint that the predicate *might* be true. By the time the woken thread re-acquires the mutex, the state of the world may have changed: another thread may have raced in and consumed the resource, or the wakeup may have been spurious. The discipline is always to re-check the predicate in a loop after `cond_wait` returns.

3. (2 points) Suppose you are tyring to implement a solution to the producer consumer problem with one condition variable and one lock. You run a four-producer / four-consumer stress test and the test eventually deadlocks. Which explanation is correct?

   A. POSIX requires at least one condition variable per participating thread, so a single condition variable can support at most one producer and one consumer.

   B. `pthread_cond_signal` can only wake a thread on the same core, so producers and consumers miss each other's wakeups on a multi-core system.

   C. Calling `pthread_cond_signal` while holding the mutex prevents other threads from aquiring the lock.

   D. A `pthread_cond_signal` can wake a same-side waiter (e.g., consumer wakes consumer), leaving the waiter on the other side unsignaled.

Solution: D. This is the second broken version walked through in lecture. The `while` loop fixes the spurious-wakeup bug, but not the wrong-side-wakeup bug. With separate `not_empty` and `not_full`, a producer that has just `put` an item signals `not_empty` (waking a consumer), and a consumer that has just `get`-ed an item signals `not_full` (waking a producer), so signals can never be absorbed by the wrong side.

4. (2 points) In the multi-threaded allocator discussed in lecture and implemented in lab, multiple threads can be blocked at the same time requesting *different* sizes, and a single release of X bytes may satisfy some waiters but not others. The lab asks you to use one mutex and one condition variable. To avoid stranding a waiter that *could* proceed when another waiter (woken first) cannot, the `alloc_release` function must call

   pthread_cond_broadcast instead of `pthread_cond_signal`.

Solution: `pthread_cond_broadcast`. This is the covering conditions pattern. Because `pthread_cond_signal` wakes *one* arbitrary waiter, the runtime may pick a waiter whose request is still too large; that waiter will re-check the predicate, find it false, and go back to sleep, leaving smaller waiters that could have proceeded permanently asleep. `pthread_cond_broadcast` wakes *every* waiter, so each one re-checks the predicate and any that can now proceed will. The cost is some unnecessary wakeups for threads whose requests are still too large, but no waiter is stranded.

5. (2 points) The cyclic barrier implemented in lab stores both an arrival counter *and* a monotonically increasing `phase` field that is incremented once per round. A waiter snapshots `phase` on entry and sleeps until it changes, rather than sleeping until the arrival counter takes a particular value. Why is the `phase` field necessary?

   A. The POSIX standard requires every barrier implementation to expose a `phase` counter as part of its API, so it is added to match `pthread_barrier_t`.

   B. Without `phase`, the mutex would have to be held across the entire wait, preventing any thread from making forward progress between rounds.

C. Without it, a fast thread can re-enter `barrier_wait` and increment the counter off of zero before slow waiters re-check the predicate, leaving them asleep forever.

D. The `phase` field guarantees that threads are released from `barrier_wait` in the same order they arrived, preventing slower threads from being starved across many rounds.

Solution: C. The arrival counter alone is unsafe under reuse because its "zero" state is transient: a fast thread that has just been released can re-enter `barrier_wait` for the next round and increment the counter off of zero before all of the previous round's waiters have re-checked the predicate. Those slow waiters then see a non-zero count, treat their wakeup as spurious, and sleep indefinitely. Predicating the wait on `phase` (which only moves forward) guarantees that a slow waiter can always tell whether its round has ended, even if a faster thread has already started the next one.