CMPU-224 Lab12 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) Which statement best describes what `pthread_join` does?

   A. Creates a new thread that runs the given start function.

   B. Blocks the calling thread until the target thread terminates, optionally collecting its return value.

   C. Acquires a mutex so the calling thread can safely enter a critical section.

   D. Detaches a thread so its resources are reclaimed automatically when it exits.

   Solution: B. `pthread_join(tid, &retval)` blocks until thread `tid` finishes and (if `retval` is non-NULL) collects its return value. It is the way the parent thread synchronizes with its children.

2. (2 points) Two threads each execute `g_counter++` in a loop, where `g_counter` is an unprotected global `int`. After both threads finish, the final value is sometimes lower than expected. Which statement best explains why?

   A. The compiler emits incorrect code for the `++` operator on shared variables.

   B. The variable overflows because `int` is too small to hold the result.

   C. `g_counter++` compiles to a load / modify / store sequence; if two threads interleave those steps, one thread's update can overwrite the other's and an increment is silently lost.

   D. `pthread_create` occasionally fails to launch one of the threads, so its iterations never run.

   Solution: C. The increment is not atomic. It expands to roughly `load r,[g_counter]; add r,1; store [g_counter],r`. If two threads each load the same value before either stores, both stores write `old+1` and one increment is lost. This is a classic data race.

3. (2 points) Consider two implementations a parallel computation. In implementation A, every thread acquires a shared mutex *inside* its inner loop and updates a shared variable on each iteration. In implementation B, every thread accumulates into a local variable inside its inner loop and acquires the

mutex *once at the end* to merge its private result into the shared variable. Why is B typically much faster than A?

A. B uses fewer threads than A.

B. B is actually incorrect; B only appears faster because it skips work.

C. B shrinks the critical section and removes nearly all lock contention — threads no longer fight for the mutex on every iteration.

D. B avoids using `malloc`, while A must allocate memory on every iteration.

Solution: C. In A the mutex is acquired and released on every iteration, so threads continually contend for the same lock and serialize through a tiny critical section. In B each thread does its inner-loop work on a local variable, where no synchronization is required, and the lock is acquired exactly once per thread (at merge time). The total number of lock acquisitions drops from $O(N)$ per thread to 1 per thread, which essentially eliminates contention.

4. (2 points) Consider the following thread function:

```
typedef struct { int x; int y; } pair_t;

void *worker(void *arg) {
    pair_t result;       /* declared here */
    result.x = 1;
    result.y = 2;
    return (void *) &result;
}
```

The main thread joins this worker and dereferences the returned pointer. What is wrong with this code?

A. Nothing — the runtime preserves a thread's stack until every other thread has read from it.

B. The cast `(void *)&result` is illegal in C.

C. `result` lives on the worker's stack, which is reclaimed when `worker` returns. The pointer the parent receives is dangling, so dereferencing it is undefined behavior.

D. `pthread_join` cannot pass a pointer back to the parent thread; it can only return an integer status.

Solution: C. Each thread has its own stack. As soon as `worker` returns, its stack frame — including `result` — is gone. The address handed back through `pthread_join` points to memory that may already be reused. The fix is to put the return value somewhere with a longer lifetime: `malloc` it on the heap (and have the caller `free` it), or write it into storage owned by the caller (e.g., a field of the argument struct).

5. (2 points) A program calls `pthread_create` several times to launch worker threads, but *never* calls `pthread_join` on any of them. `main` then prints `"done"` and returns. Which statement is most accurate?

A. `pthread_join` only matters when you need a worker's return value; if you don't, omitting it is harmless and any `printf` side effects are still guaranteed to appear.

B. The parent has no way to wait for the workers, so the process may exit before some of them finish; their output can be missing, truncated, or interleaved unpredictably.

C. The program fails to compile because every `pthread_create` requires a matching `pthread_join`.

D. The output is identical to a version that does call `pthread_join`, just slightly faster.

Solution: B. `pthread_join` is what synchronizes the parent with its children. Without it, the parent has no way to know when (or whether) the workers have finished. If `main` then returns, the whole process exits and any worker still running — still in the middle of `printf`, for example — is killed mid-execution.