Chapter 5e: Multithreading Final Review

Gene Cooperman

Contents

THIS IS A DRAFT OF A WORK IN PROGRESS.

1 Multithreading Final Exam Review

This review pulls together the key ideas from our multithreading unit: memory layout, mutexes, semaphores, condition variables, the Acquire–Release pattern, common optimizations, and deadlock. Use it alongside the original handouts:

- `5a-multithreaded-intro.pdf` — memory layout
- `5b-multithreaded-mutex.pdf` — mutexes
- `5c-multithreaded-semaphore.pdf` — semaphores (three patterns)
- `5d-multithreaded-condvar.pdf` — condition variables
- `acquire-release.pdf` — the Acquire–Release diagram

1.1 Memory Layout: Why It Matters for Threads

Recall the memory layout from `5a-multithreaded-intro.pdf`:

- **Global variables** live in the **data segment**. They are **shared** by all threads of the process.
- **Local variables** live in a **call frame on the stack**. Each thread has its **own private stack**, so local variables are **private to the thread** that declared them.
- The **heap** starts with size 0 at compile time and grows via `malloc()` in C (or `new` in C++/Java).

1.1.1 Side note: variables are never "declared on the heap"

Every variable is declared as either **global** (static data segment) or **local** (a call frame on a stack). A global variable can be a **pointer** whose value is an address on the heap, but the *variable itself* still lives in the static data segment. The pointer value is stored there; the thing it points to lives on the heap.

1.1.2 Consequence for synchronization

Because threads share the data segment but not each other's stacks, every synchronization primitive — `pthread_mutex_t`, `sem_t`, `pthread_cond_t` — **must be a global variable**. If it were local, each thread would have its own private copy and no synchronization would happen.

That is also why thread operations pass the **address** of the primitive using `&`. For example, `pthread_mutex_lock(&mymutex)`.

1.2 Mutexes and Critical Sections

A **mutex** protects one or more **shared, global variables** (or shared resources such as a file descriptor or socket). The code between `pthread_mutex_lock(&m)` and `pthread_mutex_unlock(&m)` is a **critical section** for mutex `m`.

1.2.1 Canonical example (from `5b-multithreaded-mutex.pdf`)

```
void *do_task(void *arg_notused) {
  pthread_mutex_lock(&mymutex);
  int mytask = task_num++;
  pthread_mutex_unlock(&mymutex);
  do_work(mytask);
  return NULL;
}
```

Things to notice:

- `task_num` is a **global shared** variable. It *must* be protected by a mutex.
- `mytask` is a **local** variable on this thread's private stack. It does **not** need protection.
- We deliberately copy `task_num` into the local `mytask` so we can **exit the critical section quickly**, then call `do_work(mytask)` outside of it.

1.2.2 Rules to remember

1. Every shared global variable accessed from multiple threads must be protected by a mutex.
2. If the same shared variable is accessed in several places, **every** access must use `lock`/`unlock` with the **same mutex**.
3. Each critical section is associated with exactly **one unique mutex**.
4. **Critical sections should be short.** Other threads may be blocked waiting to enter. (The one unavoidable exception is when the critical section must do slow I/O — reading/writing a file, a socket, etc.)

1.2.3 Self-Check: What Goes Wrong Here?

Before reading on, try to spot the bug:

```
void foo(pthread_mutex_t mymutex);

void *do_task(void *arg_notused) {
  foo(mymutex);
  return NULL;
}

void foo(pthread_mutex_t mymutex) {
  pthread_mutex_lock(&mymutex);
  int mytask = task_num++;
  pthread_mutex_unlock(&mymutex);
  do_work(mytask);
}
```

1.2.4 The bug

`foo` takes `mymutex` **by value**. C makes a **copy** into `foo`'s call frame on the thread's private stack. The global `mymutex` is still sitting in the data segment, but `foo` is now locking and unlocking a **local copy**.

Every thread that calls `foo` ends up locking its own private mutex, so no thread ever actually waits for any other thread. The mutual exclusion is silently gone, and `task_num++` is a race condition again.

Fix: pass the mutex by pointer — `void foo(pthread_mutex_t *mymutex)` — and call `pthread_mutex_lock(mymutex)` inside. Or just reference the global directly without passing it.

1.3 Semaphores

Semaphores are easier to learn than condition variables. The file `5c-multithreaded-semaphore.pdf` shows **three patterns**, and most real semaphore programs are instances of one of them.

Study strategy: do not try to memorize semaphore code case by case. Learn the three patterns, recognize which one a problem needs, and fill in the details.

1.4 Condition Variables and the Acquire–Release Pattern

Condition variables are the most subtle of the three tools. Use the Acquire–Release diagram from `5d-multithreaded-condvar.pdf` as your mental model.

1.4.1 Skeleton

```
// ---- Acquire ----
pthread_mutex_lock(&mutex);
while ( !CONDITION ) {
```

```
    pthread_cond_wait(&cond_var, &mutex);   // Waiting Room
}
// ... modify shared global variables (DO_TASK) ...

// ---- Release ----
// ... possibly modify shared globals again ...
pthread_cond_broadcast(&cond_var);
pthread_mutex_unlock(&mutex);
```

1.4.2 Things the diagram is telling you

- The **while loop** (not `if`) is inside the critical section of `mutex`.
- The **Waiting Room** is *outside* the critical section — `pthread_cond_wait` atomically releases `mutex` while the thread sleeps, and re-acquires it on wake-up.
- A thread that wakes from `pthread_cond_wait` re-tests `CONDITION` on the current values of the shared globals. That is why a spurious wake-up or a broadcast that wakes "too many" threads is never a correctness problem.
- The critical section of `mutex` is **not contiguous** — it has two regions: the Acquire section and the Release section.

1.4.3 Rule of thumb: correctness first

> **Always use `pthread_cond_broadcast()` in the Release section first.** Optimize only after the code is correct. Premature optimization is almost always a mistake.

Broadcast wakes every thread in the Waiting Room. That is never wrong — each one rechecks `CONDITION` in its `while` loop, and only one runs in the critical section at a time.

1.5 Optimizing Condition-Variable Code

There are two common optimizations, in roughly this order.

1.5.1 Optimization A: switch `broadcast` to `signal`, or skip it entirely

- If you can prove only **one** waiting thread could possibly make progress, replace `pthread_cond_broadcast(&cv)` with `pthread_cond_signal(&cv)`.
- If you can prove that the state change in this Release section cannot satisfy any waiter's `CONDITION`, wrap the `broadcast`/`signal` in an `if` statement so that you skip it entirely.

1.5.2 Optimization B: multiple Waiting Rooms (multiple condition variables)

Often there are **two (or more) types of thread** — say, *readers* and *writers*. You can tell them apart because `pthread_create()` gives them different start functions.

Just as each critical section has a unique mutex, **each Waiting Room has a unique condition variable**. For readers/writers:

- Create `cond_reader` and `cond_writer`.

- Readers wait on `pthread_cond_wait(&cond_reader, &mutex)` with their own `CONDITION_READER`.
- Writers wait on `pthread_cond_wait(&cond_writer, &mutex)` with their own `CONDITION_WRITER`.
- The Release section now has to decide *which* kind of thread (zero, one, or all) to wake up. This usually means one or more `if` statements, each calling `pthread_cond_broadcast/signal` on the appropriate condition variable.

1.5.3 Other multi-waiting-room patterns

Readers/writers is just one example. Others include:

- **Thread priorities.** A Priority-1 waiter should be woken before any Priority-2 thread is allowed to enter. The same principles allow you to have an arbitrary number of priority levels.
- **Bounded concurrency.** For example, allowing at most 5 reader threads inside `DO_TASK` at once.

1.5.4 Verify with a model checker

You can often audit the simple version (one Waiting Room, `broadcast`) by hand. The optimized versions have too many interleavings to check manually. For the optimized version it is **highly recommended** to verify correctness with a model checker such as **McMini**.

1.6 Converting Semaphores to Condition Variables

It is trivial to convert semaphore code to condition-variable code:

1. Create a shared global `int sem_count`.
2. Create one Waiting Room (one condition variable) for the semaphore.
3. Replace `sem_wait()` with the Acquire pattern (wait while `sem_count == 0`, then decrement).
4. Replace `sem_post()` with the Release pattern (increment, then signal/broadcast).

Because of this, you usually do not mix semaphores and condition variables in the same program. If you find yourself wanting to, convert the semaphore code into condition-variable code and merge.

1.7 Deadlock

Definition. Deadlock occurs when all threads are blocked.

Deadlock Cycles. Deadlock occurs **if and only if** there is a cycle of threads, each holding one or more resources and waiting on an additional resource held by the next thread in the cycle.

A thread is waiting on a resource if it is blocked inside of:

- `pthread_mutex_lock()`
- `sem_wait()`
- `pthread_cond_wait()`

A thread is holding a resource if it has:

- successfully called `pthread_mutex_lock()` and not yet called `pthread_mutex_unlock()`, or
- successfully called `sem_wait()` and not yet called the matching `sem_post()`, or

- returned from `pthread_cond_wait()` and modified shared globals, but not yet restored them inside the Release section.

FOR FURTHER READING: Deadlock requires **all** of: mutual exclusion, hold-and-wait, no preemption, and circular wait. Break any of these and deadlock is impossible.

1.8 Cheat Sheet

Concept	Key rule
Sync primitives (`pthread_mutex_t`, `sem_t`, `pthread_cond_t`)	Must be **global**; always passed by address (`&`).
Passing a mutex by value	**This is a bug.** It is copied to the thread's stack and protects nothing.
Critical section	Short; one unique mutex; same mutex at every access of the shared variable.
`while` vs `if` stmt around `pthread_cond_wait`	Always `while` — re-test the condition on wake-up.
Correctness-first rule for cond vars	Use `pthread_cond_broadcast` in Release.
Optimization A	Downgrade to `signal`, or guard with `if`, when provably safe.
Optimization B	One condition variable per *type* of thread (Waiting Room).
Semaphore → cond var	Trivially convertible; do not mix them.
Deadlock	Cycle of threads each *holding* one resource and *waiting* on another.

1.9 Self-Test Questions

1. Why must `pthread_mutex_t`, `sem_t`, and `pthread_cond_t` be declared as global variables?
2. In the `do_task` example, why does `mytask` not need mutex protection, but `task_num` does?
3. Explain precisely what goes wrong in the buggy `foo(pthread_mutex_t mymutex)` example.
4. Why is the loop around `pthread_cond_wait` a `while` loop rather than an `if` statement?
5. Why is `pthread_cond_broadcast` never *incorrect*, even when only one thread could possibly make progress?
6. When would you split a single Waiting Room into two (or more)? Give one example besides readers/writers.
7. Give the precise definition of deadlock, and name the three operations that can block a thread on a resource.
8. Describe how to convert a program using one semaphore into an equivalent condition-variable program.

Good luck on the exam — and remember: **correctness first, optimization later.**