Homework 1: Critical Sections. Locks, Barriers, and Condition Variables

Sana Monhaseri
Group 11

Spring 2026

1. Compute the Sum, Min, and Max of Matrix Elements

a) Max and Min with Barrier

I extended `matrixSum.c` to find max/min values and their positions. Each worker stores partial results in global arrays (`max_val[myid]`, `min_val[myid]`, etc.). Since each worker writes to a different index, no mutex is needed.

Local variables are initialized to the first element of the worker's strip (`matrix[first][0]`) to handle all-negative matrices correctly. After the barrier, Worker 0 compares all partial results and prints the final output.

b) Mutex Locks, No Barrier

I removed the barrier and arrays, using shared global variables protected by three separate mutex locks (for sum, max, min). Three locks allow more parallelism than one - a thread updating sum doesn't block another updating max.

I initialized `max` to `INT_MIN` and `min` to `INT_MAX` so any matrix value triggers an update. Each worker finds local results in its strip, then acquires locks to update globals directly. No Worker 0 combining phase needed.

For the main thread, I replaced `pthread_exit(NULL)` with `pthread_join()` to wait for all workers to finish before printing results. Using `pthread_exit()` would have caused main to exit immediately without printing, while the workers continued running.

c) Bag of Tasks

Instead of static strips, workers dynamically grab rows from a shared counter `nextRow` protected by `nextRow_lock`. I removed `stripSize`, `first`, and `last`.

Each worker loops: acquire lock, check if `nextRow >= size` (if yes, break), take row number, increment `nextRow`, unlock, process row. After the loop, workers update global variables with locks (same as part b).

The advantage is load balancing - if some rows take longer, fast workers grab more rows instead of sitting idle.

4. The Linux tee command

The Linux tee command reads from standard input, then writes the output to both the standard output (screen) and to a file at the same time.

I used the producer-consumer approach with three threads: one producer to read from stdin, and two consumers - one writes to stdout, one writes to a file. They share a buffer where the producer puts data and consumers take from.

Initially I had two separate consumer functions, but they were doing mostly the same thing. So I combined them into one `Consumer` function that takes an ID as argument and uses `readersDone[myId]` to track its state.

Synchronization

I used condition variables instead of busy waiting (which wastes CPU cycles):

- `dataAvailable`: Producer broadcasts to wake both consumers
- `readDone`: Consumers signal producer when done reading

I use `while` loops (not `if`) with condition variables because after waking up, the condition must be re-checked - another thread might have taken the data, or there could be spurious wakeups.

Local Buffers

Both consumers and producer use local buffers. Writing to file/stdout is slow - if a consumer holds the lock while writing, others are blocked. Instead, we do a fast copy, release the lock, then write outside the critical section.

For the producer, reading directly into the shared buffer caused a race condition since `fgets` happens outside the lock. The solution is to read into a temporary (reader) buffer first, wait for consumers, then copy to shared buffer.

When Consumers Are Done

Using a single counter had problems - a fast consumer could increment it twice before the slow one read once. My solution uses `readersDone[2]` with one flag per consumer. Each consumer sets only its own flag. The producer waits until both are 1, then resets both to 0. This ensures each consumer reads each piece of data exactly once.