HW2 – Dispatcher–Worker Model with POSIX Threads

Dan Shamia – 208004119 Daniel Halperin – 207826314

Overview:

In this assignment we implemented a multi-threaded dispatcher–worker system using pthreads. The program reads commands from an input file, processes dispatcher-level commands sequentially, and dispatches worker-level commands to a pool of worker threads via a shared job queue.

The system supports:

- concurrent execution of worker jobs
- synchronization using mutexes and condition variables
- logging, timing, and statistics collection
- graceful shutdown of worker threads

Architecture:

The program is divided into two logical components:

Dispatcher (Main Thread)

- Parses command-line arguments
- Reads commands from the input file
- Executes dispatcher commands (dispatcher_msleep, dispatcher_wait)
- Enqueues worker jobs into a shared queue
- Manages job lifecycle and shutdown

Worker Threads

- Continuously dequeue jobs from the shared queue
- Parse and execute job commands
- Update counters stored on disk
- Record execution statistics
- Log job start and completion

3. Threading Model

- One **dispatcher thread** (main thread)
- N **worker threads**, created at startup

- Workers run an infinite loop and block on a condition variable when the queue is empty
- Shutdown is coordinated via a shared shutdown flag in the queue

Job Queue Design:

A FIFO queue is implemented using a linked list.

Synchronization

- pthread_mutex_t mutex protects queue state
- pthread_cond_t not_empty is used to wake workers when jobs are enqueued

Blocking Behavior:

- Workers block in queue_dequeue() while the queue is empty and shutdown is not requested
- When shutdown is set and the queue is empty, queue_dequeue() returns NULL, signaling workers to exit

This design encapsulates all queue synchronization logic inside the queue module, keeping worker code clean and simple.

Shutdown Mechanism:

The shutdown process follows these steps:

1. Dispatcher finishes reading the input file
2. Dispatcher waits until all outstanding jobs are completed
3. Dispatcher sets queue.shutdown = 1 under the queue mutex
4. Dispatcher broadcasts on queue.not_empty to wake all workers
5. Workers detect shutdown and exit gracefully
6. Dispatcher joins all worker threads

This ensures:

- No jobs are lost
- No worker remains blocked indefinitely
- Clean program termination

Dispatcher Commands:

dispatcher_msleep <ms>

Suspends the dispatcher thread for the given number of milliseconds using usleep().

dispatcher_wait

Blocks the dispatcher until all previously dispatched worker jobs have completed.
This is implemented using:

- a global g_outstanding_jobs counter
- a mutex and condition variable
- workers signal the dispatcher when the counter reaches zero

7. Worker Job Execution

Worker jobs consist of one or more commands separated by semicolons.
Supported commands include:

- msleep x – sleep for x milliseconds
- increment i – increment counter i
- decrement i – decrement counter i
- repeat x – repeat the job x times

Workers parse each job line, execute commands sequentially, and support nested repetition as defined by the assignment.

8. Counters Implementation

Counters are stored as files (countXX.txt) on disk.

- One mutex per counter ensures atomic updates
- Each increment/decrement:
    - locks the counter mutex
    - reads the current value from file
    - updates the value
    - writes it back

This design prevents race conditions between workers accessing the same counter.

9. Timing and Statistics

Timing

- All timing is based on clock_gettime(CLOCK_MONOTONIC)
- Program start time is recorded once at startup
- Each job records its turnaround time

Statistics Collected:

- Total program runtime
- Sum of job turnaround times
- Minimum job turnaround time
- Maximum job turnaround time
- Average job turnaround time

At the end of execution, statistics are written to stats.txt in the required format.

Logging:

Dispatcher Logging

- Written to dispatcher.txt
- Logs each command read from the input file
- Format:
- TIME <ms>: read cmd line: <line>

Worker Logging

- Each worker writes to threadXX.txt
- Logs job start and completion times

Logging is enabled or disabled via a command-line argument.

Synchronization Summary:

The program uses the following synchronization primitives:

- Mutexes:
    - Queue mutex
    - Outstanding jobs mutex
    - Statistics mutex
    - One mutex per counter
- Condition variables:
    - Queue not_empty
    - Outstanding jobs completion condition

All condition waits are performed while holding the relevant mutex, following POSIX best practices and preventing lost wakeups.

Memory Management:

- Job structures are dynamically allocated by the dispatcher
- Job ownership is transferred to workers via the queue
- Workers free job memory after execution
- Queue nodes are freed upon dequeue
- All resources are released before program termination

Conclusion:

This implementation satisfies all assignment requirements and demonstrates:

- Correct use of POSIX threads
- Proper synchronization using mutexes and condition variables
- Safe producer–consumer queue design
- Clean shutdown semantics
- Modular and maintainable code structure

The division of responsibility between dispatcher and workers ensures clarity, correctness, and scalability.