Homework 2: Dispatcher/Worker model with Linux pthreads

Submitted by: Ido Josephsberg (ID 322641135) and Noa Keter (ID 318875770)

1. System Overview

This program implements a dispatcher/worker system using POSIX threads on Linux. The dispatcher reads commands from a text-based command file and either executes them directly (dispatcher commands) or packages them as jobs and sends them to worker threads (worker commands). Jobs are stored in a shared job queue protected by a mutex and serviced concurrently by a pool of worker threads. The workers execute simple commands (msleep, increment, decrement) and a compound repeat command that repeats a subsequence of commands. Synchronization is handled using mutexes and condition variables both for the job queue and for individual counter files, ensuring thread-safe access, preventing race conditions, and avoiding busy-waiting. The system logs all activity to text files and finally produces a stats.txt file summarizing total runtime and job turnaround statistics.

2. #define Constants

2.1. **File-related:**

MAX_FILE_NAME – maximum file name length (e.g. "threadXX.txt", "countXX.txt").
MAX_JOB_FILE_LINE – maximum length of one line in the command file.

2.2. **Threads and counters:**

MAX_WORKER_THREADS – upper bound on number of worker threads.
MAX_COUNTER_FILES – maximum number of counter files (countXX.txt).
MAX_DIGITS – maximum digits in a counter value string.

2.3. **Jobs:**

MAX_COMMANDS_IN_JOB – maximum number of commands per job.
MAX_CMD_NAME_LENGTH – maximum length of a command name (e.g. "dispatcher_msleep").

3. Core Data Structures

3.1. **Command:**

Represents a single command within a job (e.g., "msleep" with argument 100, or "increment" with argument 3).

```
typedef struct {
    char cmd_name[MAX_CMD_NAME_LENGTH];
    int cmd_arg;
} Command;
```

3.2. **Job:**

Encapsulates one worker job: the sequence of parsed commands, a copy of the original line for logging, and timing information.

```
typedef struct Job {
    Command* job_cmds;
    struct Job* next;
    char job_line[MAX_JOB_FILE_LINE];
    long long time_after_reading_line_ms;
} Job;
```

3.3. **JobQueue:**

A thread-safe FIFO queue of jobs. In addition to the linked list, it maintains:

- size – current queue length.
- total_jobs_added – total number of jobs ever pushed (used for statistics).
- num_of_working_threads – number of worker threads currently processing jobs.
- log_enabled – toggles generation of log files.
- exit_flag – signals worker threads to terminate once all jobs are done.
- turnaround statistics: total, min, max.
- lock – mutex guarding all fields.
- cond_idle – condition variable signaled when the system becomes idle (no jobs in queue and no working threads).

```
typedef struct JobQueue {
    Job* head;
    Job* tail;
    int size;
    int total_jobs_added;
    int num_of_working_threads;
    int log_enabled;
    int exit_flag;
    long long total_turnaround_time_ms;
    long long min_turnaround_time_ms;
    long long max_turnaround_time_ms;
    pthread_mutex_t lock;
    pthread_cond_t cond_idle;
} JobQueue;
```

4. Code Structure and Concurrency Design

4.1. **Files in the project**:

The program is split into small, focused modules:

- main.c – sets up the global start time and delegates everything to the dispatcher.
- dispatcher.c – parses command-line arguments, initializes the global structures, reads the command file, and either executes dispatcher commands or creates jobs for workers.
- threads.c – creates the worker threads and implements their main routine (fetch job, execute commands, update stats, log).
- job_queue.c (job_queue.h) – job queue implementation (push_job, pop_job) and the Job, Command, JobQueue definitions.
- cmdfile_handler.c (cmdfile_handler.h) – parsing of command lines into Command arrays.
- counter_files.c (counter_files.h) – creation and thread-safe increment/decrement of countXX.txt files using per-file mutexes.
- log_files.c (log_files.h) – creation of threadXX.txt and dispatcher.txt and writing log entries with timestamps.
- stats.c (stats.h) – creation of stats.txt at the end with global timing and job turnaround statistics.
- system_call_error (system_call_error.h) – helper to print system call errors with errno.
- macros.h – global constants.
- global_vars – exposes shared objects like the global job queue and start time so all modules can coordinate using a consistent set of global state.

4.2. **Synchronization & concurrency:**

- The job queue is protected by a mutex inside shared_jobs_queue.
- Workers wait on a condition variable (ava_jobs_cond) when the queue is empty and are woken when new jobs arrive.
- A second condition (cond_idle) plus a num_of_working_threads counter allow the dispatcher to wait until the system is truly idle.
- Each countXX.txt file has its own mutex in file_counters_mutexes, allowing parallel updates to different counters without conflicts.

5. Module Breakdown

5.1. **Initialization Phase** (main.c)
main captures global_start_time using gettimeofday and immediately calls dispatcher(argc, argv). All initialization, threading, and cleanup logic is handled inside the dispatcher, keeping main minimal.

5.2. **The Dispatcher Module** (dispatcher.c)

- The Dispatcher:
  - Parses command-line arguments and validates the number of counters and threads.
  - Initializes counter files, the global job queue, mutexes, and condition variables.
  - Creates worker threads and, if logging is enabled, initializes the dispatcher and thread log files.
  - Reads the command file line by line, logs each line to dispatcher.txt (if enabled), and distinguishes dispatcher commands from worker job lines.
  - Executes dispatcher commands immediately (e.g., dispatcher_msleep, dispatcher_wait).
  - For worker lines, allocates a Command array, parses the line into commands, and calls push_job with a copy of the line and its timestamp.
  - Uses dispatcher_wait to block until all jobs are completed.
  - On termination, sets the exit flag, wakes all workers, joins all threads, and calls create_stats_file() to generate stats.txt.

5.3. **The Worker Module** (threads.c)
Each worker runs the same thread_routine in a loop:

- Wait & Dequeue:

  Acquires the job queue mutex, waits on ava_jobs_cond while the queue is empty and exit is not requested, and calls pop_job to get the next job. It increments num_of_working_threads before releasing the mutex.
- Log Start:

  If logging is enabled, logs the START of the job to its threadXX.txt file, including the original job line and a timestamp.
- Execution:

  Iterates over the job's Command array:
  - For msleep, calls msleep.
  - For increment / decrement, calls the corresponding counter function, which locks the appropriate counter mutex.

- For repeat, executes the next sequence of commands multiple times using nested loops.

- Log End & Stats:

After executing all commands, computes the job's turnaround time from time_after_reading_line_ms. Logs the END of the job to threadXX.txt. Under the queue mutex, it updates aggregate turnaround statistics, decrements num_of_working_threads, and signals cond_idle if the system becomes idle.
Workers free their job structures and command arrays, then loop back to wait for more jobs or for the exit flag.

6. Output Files & Logging

6.1. **dispatcher.txt** – One file logging every line the dispatcher reads from the command file, with timestamps relative to program start.
6.2. **threadXX.txt** – One file per worker thread, containing only START and END records for jobs executed by that thread.
6.3. **countXX.txt** – One file per counter, initially set to zero and updated by worker commands while protected by per-file mutexes.
6.4. **stats.txt** – A final report written at program termination, listing total run time, sum of job turnaround times, and min/avg/max job turnaround times.

7. Compile and Debugging

The program assumes well-formed input files and command-line arguments. It is compiled on Linux using the provided Makefile, which builds all modules with GCC and the flags -pthread -g -wall to enable threading support, debugging symbols, and strict warnings. During development, we used selective printf logging together with GDB and gdbgui to step through execution, inspect shared data structures, and observe thread interactions, which helped detect and fix concurrency and memory issues.

8. Summary

This assignment offered practical practice in building a multi-threaded system around a dispatcher–worker model. Implementing the job queue, worker pool, and logging forced careful use of the pthread API, mutexes, and condition variables. By coordinating shared data and timing information across threads, we gained concrete insight into race conditions, proper locking patterns, and how to design safe and efficient concurrent programs on Linux.