Directory Organization for the Homework Assignment

1. **Base Directory Structure**

c

Copy code

```
hw2_id1_id2/
├── src/
│   ├── main.c
│   ├── dispatcher.c
│   ├── dispatcher.h
│   ├── worker.c
│   ├── worker.h
│   ├── queue.c
│   ├── queue.h
│   ├── utils.c
│   ├── utils.h
├── include/
│   ├── common.h
├── tests/
│   ├── test.txt
│   └── other_tests.txt
├── output/
│   ├── counter files/
│   ├── log files/
│   ├── stats.txt
├── Makefile
├── hw2_id1_id2.pdf
└── README.md
```

2. **Files and Their Contents**
    - **src/main.c**
        - Contains the main function.
        - Parses command-line arguments.
        - Initializes the dispatcher and worker threads.
        - Manages cleanup and final statistics.
        - Keeps track of program start time.
    - **src/dispatcher.c**
        - Contains the dispatcher logic.
        - Reads and parses the input command file.
        - Handles dispatcher commands (msleep, wait).
        - Queues jobs for workers.
        - Writes to the dispatcher log file.
    - **src/dispatcher.h**
        - Declares functions for the dispatcher logic.
    - **src/worker.c**
        - Implements worker thread functionality.
        - Executes jobs from the shared queue.
        - Handles basic commands (msleep, increment, decrement, repeat).
        - Manages worker-specific log files.
    - **src/worker.h**
        - Declares functions for worker logic.
    - **src/queue.c**
        - Implements a thread-safe queue for the shared work queue.
        - Handles synchronization between dispatcher and worker threads.
    - **src/queue.h**
        - Declares queue-related data structures and functions.

- **src/utils.c**
  - Contains utility functions (e.g., time calculation, file operations).
  - Manages counter file updates (increment, decrement).
  - Provides common synchronization functions.
- **src/utils.h**
  - Declares utility functions and shared constants.
- **include/common.h**
  - Contains global constants and shared macros (e.g., max line width, max threads, max counters).
  - Defines shared data structures, such as the job structure.
- **tests/test.txt**
  - A sample command file for testing.
  - Includes commands to test all dispatcher and worker functionality.
- **output/**
  - Subdirectories for counter files (countxx.txt), log files (threadxx.txt, dispatcher.txt), and stats.txt.
- **Makefile**
  - Builds the executable (hw2).
  - Includes targets: all, clean, test.
- **hw2_id1_id2.pdf**
  - External documentation describing the implementation, challenges, and design decisions.
- **README.md**
  - Provides an overview of the project, instructions for building and running the program, and examples.

Work Plan for Implementation

1. **Setup and Initialization**
    - **Goal:** Create the project structure and basic stubs for all files.
    - **Steps:**
        1. Set up the directory structure.
        2. Write the Makefile to compile all source files.
        3. Write initial versions of main.c, dispatcher.c, and worker.c with placeholder functions.
2. **Core Dispatcher Logic**
    - **Goal:** Implement the dispatcher logic for sequential commands and job queuing.
    - **Steps:**
        1. Parse command-line arguments in main.c.
        2. Implement the dispatcher command handling (msleep, wait) in dispatcher.c.
        3. Add logic to read the command file and queue jobs.
3. **Thread-Safe Work Queue**
    - **Goal:** Implement a thread-safe queue to manage jobs.
    - **Steps:**
        1. Define the queue data structure in queue.h.
        2. Implement enqueue, dequeue, and synchronization in queue.c.
        3. Test the queue with a simple producer-consumer model.
4. **Worker Thread Implementation**
    - **Goal:** Implement worker threads to execute queued jobs.
    - **Steps:**
        1. Create worker threads in main.c during initialization.
        2. Implement job handling in worker.c, including parsing and executing basic commands.
        3. Use pthread_mutex and pthread_cond for synchronization.

5. **File Operations**
    - **Goal:** Manage counter files and ensure atomic updates.
    - **Steps:**
        1. Create counter files during initialization.
        2. Implement atomic increment and decrement in utils.c.
        3. Ensure thread safety using file locks or synchronization primitives.
6. **Logging and Timing**
    - **Goal:** Add logging for workers and the dispatcher.
    - **Steps:**
        1. Record start and end times for jobs.
        2. Write log messages to the appropriate files.
        3. Use gettimeofday or clock_gettime to calculate timestamps.
7. **Statistics Collection**
    - **Goal:** Collect and output job statistics.
    - **Steps:**
        1. Track job turnaround times.
        2. Compute total, min, max, and average turnaround times.
        3. Write the statistics to stats.txt.
8. **Testing and Debugging**
    - **Goal:** Ensure correctness and handle edge cases.
    - **Steps:**
        1. Write comprehensive test cases in tests/test.txt.
        2. Validate against edge cases (e.g., invalid commands, zero counters, max threads).
        3. Debug and optimize synchronization to avoid deadlocks.
9. **Documentation and Submission**
    - **Goal:** Complete external documentation and prepare the submission package.
    - **Steps:**

1. Write the PDF documentation (hw2_id1_id2.pdf).
2. Test the program and verify submission requirements.
3. Package the solution into a .tgz file for submission.

This structured approach ensures modularity, correctness, and maintainability.

Git Setup

1. **Initialize the Repository**:
    - One developer initializes the Git repository and sets up a remote (e.g., GitHub, GitLab).
    - Create the project structure (hw2_id1_id2/) and add an initial commit with placeholders for the files.
2. **Branching Strategy**:
    - Use a **main** branch for stable, tested code.
    - Create feature branches for different tasks (e.g., feature-dispatcher, feature-worker, feature-queue).
    - Each developer works on separate branches to avoid conflicts.
3. **Commit Guidelines**:
    - Write descriptive commit messages (e.g., Implement job queuing in dispatcher or Fix race condition in worker).
    - Make atomic commits that focus on one feature or bug fix.
4. **Code Review**:
    - Use pull requests (PRs) to merge feature branches into the main branch.
    - One developer reviews the other's code before merging to ensure quality and consistency.

Task Division

Developer 1: Dispatcher and Overall Coordination

- **Primary Responsibilities**:
    - Implement the dispatcher logic.
    - Handle command parsing (msleep, wait, and worker job queuing).
    - Create and manage counter files during initialization.
    - Write dispatcher logs (dispatcher.txt).
- **Tasks**:

1. Parse command-line arguments in main.c.

2. Read and parse the input command file in dispatcher.c.
3. Implement dispatcher-specific commands (msleep, wait).
4. Queue jobs for worker threads using the shared queue.
5. Create counter files (countxx.txt) during initialization.
6. Write statistics to stats.txt at the end.

- **Feature Branches**:
    - feature-dispatcher
    - feature-counter-files
    - feature-logging

Developer 2: Worker Threads and Synchronization

- **Primary Responsibilities**:
    - Implement worker thread functionality.
    - Manage the shared work queue.
    - Handle basic commands (msleep, increment, decrement, repeat).
    - Write worker logs (threadxx.txt).
    - Ensure proper synchronization using pthread_mutex and pthread_cond.
- **Tasks**:

1. Implement the shared work queue (queue.c and queue.h).
2. Create worker threads and handle job execution in worker.c.
3. Implement file operations for counters (increment, decrement).
4. Add logging for workers (threadxx.txt).
5. Ensure thread safety and synchronization between dispatcher and workers.

- **Feature Branches**:
    - feature-worker
    - feature-queue
    - feature-thread-safety

Collaboration Workflow

1. **Daily Updates**:
    - Schedule short sync-up meetings or Git messages to update each other on progress and blockers.
2. **Integration Points**:
    - Developer 1 provides stubs for worker-related functions (worker_enqueue, worker_execute).
    - Developer 2 provides a thread-safe queue implementation early so Developer 1 can integrate it with the dispatcher.
3. **Branch Merging**:
    - Once a feature is complete and tested, open a pull request for review.
    - Use squash-and-merge to keep the main branch history clean.
4. **Testing Together**:
    - Both developers contribute to the test file (tests/test.txt).
    - Run integration tests together once the dispatcher and worker features are complete.
5. **Bug Fixes**:
    - Use bug-specific branches (e.g., bugfix-logging-issue) and assign based on expertise.

Example Timeline

Day	Task Developer 1	Task Developer 2
1	Set up project structure and Git repo.	Implement shared queue structure.
2	Implement command-line parsing.	Implement worker thread creation and basic job handling.
3	Read and parse command file, implement msleep and wait.	Implement queue synchronization with mutexes/condition variables.
4	Handle job queuing and dispatcher logs.	Handle worker-specific commands (msleep, increment, decrement).
5	Integrate dispatcher and worker via the shared queue.	Test worker logging and ensure thread safety.

Day	Task Developer 1	Task Developer 2
6	Collect statistics and write stats.txt.	Debug and optimize worker execution.
7	Final testing, integration, and documentation.	Final testing, integration, and documentation.

Final Submission Workflow

1. **Tag the Final Version**:
    - Tag the final commit as v1.0 before creating the tarball.
    - Ensure the main branch contains all features and passes all tests.
2. **Create .tgz**:
    - Run make clean and compress the directory into hw2_id1_id2.tgz.
3. **Verify Submission**:
    - Double-check the structure, documentation, and Makefile to ensure compliance with the requirements.

By dividing tasks and coordinating effectively with Git, both developers can work efficiently and avoid conflicts.