Documentation for Multi-Threaded Job Queue Program

# 1. Overview

This program is a multi-threaded job queue implementation that processes commands   
read from an input file. It supports multiple worker threads, synchronization using mutexes   
and condition variables, and optional logging. The program executes commands like 'increment',   
'decrement', and 'msleep', tracks statistics, and generates output files for analysis.

# 2. Purpose

The purpose of this program is to simulate job scheduling and execution   
using worker threads. It demonstrates concepts of concurrency, thread synchronization,   
and resource management in C programming.

# 3. Input and Usage

To compile and run the program, use the following commands:

1. 1. Compile:  
    gcc -o hw2 hw2.c -lpthread
2. 2. Run:  
    ./hw2 <cmdfile> <num\_threads> <num\_counters> <log\_enabled>

Where:

- `<cmdfile>`: Input file containing job commands.  
- `<num\_threads>`: Number of worker threads.  
- `<num\_counters>`: Number of counters/files to initialize.  
- `<log\_enabled>`: Set to 1 to enable logging or 0 to disable logging.

# 4. Key Features

* Multi-threading: Worker threads process jobs concurrently.
* Job queue: A thread-safe circular queue to hold jobs.
* Synchronization: Uses mutexes and condition variables for thread safety.
* Logging: Generates logs for job execution and thread activity.
* Statistics: Calculates total runtime, turnaround times, and more.

# 5. Code Structure and Functions

## Main Function

The main function initializes resources, creates worker threads, reads job commands,   
and ensures graceful shutdown of the program. Key tasks include:  
  
- Parsing command-line arguments.  
  
- Initializing job queues and synchronization primitives.  
  
- Reading job commands from the input file.  
  
- Waiting for all jobs to complete and cleaning up resources.

## worker\_thread Function

This function runs on each worker thread and processes jobs dequeued from the job queue.   
It handles special commands like 'repeat' to execute commands multiple times. The function updates   
statistics for turnaround times and ensures safe access to shared resources using mutexes.

## init\_queue Function

Initializes the job queue with front, rear, count, and shutdown status. It also sets up   
mutexes and condition variables for synchronization between producer (main thread) and consumer (worker threads).

## enqueue and dequeue Functions

- \*\*enqueue\*\*: Adds a job to the circular job queue. If the queue is full, it waits using a condition variable.  
  
- \*\*dequeue\*\*: Removes a job from the queue. If the queue is empty, it waits until new jobs are available or shutdown is signaled.

## execute\_command Function

Parses and executes job commands:  
  
- `increment <counter>`: Increments the value in a counter file.  
  
- `decrement <counter>`: Decrements the value in a counter file.  
  
- `msleep <time>`: Sleeps for the specified duration in milliseconds.  
  
It uses mutexes to ensure thread-safe access to counter files.

## print\_to\_log\_file Function

Logs job execution details to a log file for each worker thread.

## create\_stats\_file Function

Generates a 'stats.txt' file summarizing:  
  
- Total runtime.  
  
- Sum, minimum, average, and maximum job turnaround times.

# 6. Error Handling

The program includes error checks for:  
  
- Invalid arguments.  
  
- File I/O errors.  
  
- Memory allocation failures.  
  
- Thread creation errors.  
  
Appropriate error messages are printed, and the program terminates gracefully.

# 7. Example Input File (cmdfile.txt)

Sample commands:  
  
worker increment 1;  
  
worker msleep 100;  
  
worker decrement 1;  
  
dispatcher\_msleep 200;  
  
dispatcher\_wait

# 8. Output Files

* countXX.txt: Counter files storing values for increment and decrement operations.
* threadXXXX.txt: Log files for each worker thread (if logging is enabled).
* dispatcher.txt: Log file for dispatcher activity.
* stats.txt: Summary of runtime and job turnaround statistics.

# 9. Conclusion

This program demonstrates efficient multi-threaded job scheduling using a thread-safe   
queue, synchronization primitives, and file-based job execution. It provides clear logging and   
statistics for analyzing job performance.