```
### Dispatcher/Worker Model Explanation

This program implements a Dispatcher/Worker model using Linux `pthreads` to process tasks efficiently from an input command file. It manages multiple worker threads, synchronizes tasks, and logs activities to analyze performance. Below is a detailed step-by-step explanation of the key components and flow of the program.

---

### **Main Components**

1. **Main Function (`main()`)**:
    - **Purpose**: Serves as the entry point of the program, initializing resources, creating threads, and processing commands.
    - **Initialization**:
      - Reads command-line arguments: `cmdfile.txt`, number of threads, number of counters, and whether logging is enabled.
      - Initializes a shared work queue.
      - Creates counter files (e.g., `count00.txt`) to store persistent counter values.
      - Spawns worker threads, passing unique data to each thread.
    - **Processing Commands**:
      - Reads and categorizes commands from the input file as either dispatcher commands (e.g., `dispatcher msleep x`) or worker jobs (e.g., `worker increment x`).
      - Uses helper functions (`parse_dispatcher_command` and `parse_worker_job`) to handle each type of command.
      - Logs command reads if logging is enabled.
    - **Shutdown**:
      - Signals all worker threads that no further jobs will be added.
      - Waits for worker threads to complete and calculates overall statistics.

---

### **Dispatcher**
    - **Purpose**: Manages commands sequentially and delegates jobs to worker threads.
    - **Key Features**:
      - Processes `msleep` commands to delay execution for a specified duration.
      - Handles the `wait` command to ensure all queued jobs are completed before continuing.
      - Delegates worker-specific jobs by enqueuing them into the work queue for parallel processing.

---

### **Worker Threads**
    - **Purpose**: Executes jobs offloaded by the dispatcher.
    - **Execution Flow**:
      1. Waits for jobs in the work queue.
      2. Processes commands (e.g., incrementing/decrementing counters or sleeping for a duration).
      3. Updates global statistics (e.g., job turnaround times) and logs activity if enabled.
    - **Thread Safety**:
      - Synchronizes access to shared resources like counters and the work queue using `pthread_mutex`.

---

### **Work Queue**
    - **Purpose**: Acts as a shared buffer between the dispatcher and workers.
    - **Structure**:
      - Circular queue to manage fixed capacity efficiently.
      - Tracks queue size, front, and rear indices for enqueue/dequeue operations.
    - **Synchronization**:
      - Uses condition variables (`cond_empty`, `cond_full`, `cond_wait`) to ensure proper coordination:
        - Workers sleep when the queue is empty.
```

```
54.        - Dispatcher waits if the queue is full.
55.
56. ---
57.
58. ### **Commands**
59.    - **Dispatcher Commands**:
60.      - `msleep x`: Pauses the dispatcher for `x` milliseconds.
61.      - `wait`: Ensures all worker jobs are complete before proceeding.
62.    - **Worker Commands**:
63.      - `msleep x`: Puts a worker thread to sleep for `x` milliseconds.
64.      - `increment x`: Increments the value in counter file `countxx.txt`.
65.      - `decrement x`: Decrements the value in counter file `countxx.txt`.
66.      - `repeat x`: Repeats a sequence of commands `x` times.
67.
68. ---
69.
70. ### **Logging**
71.    - **Purpose**: Provides detailed traceability of execution flow and performance.
72.    - **Dispatcher Log**:
73.      - Records commands read from the input file with timestamps.
74.    - **Worker Logs**:
75.      - Logs the start and end of each job with timestamps in individual files (`threadxx.txt`).
76.
77. ---
78.
79. ### **Statistics**
80.    - **Metrics Tracked**:
81.      - Total program runtime.
82.      - Sum, minimum, maximum, and average job turnaround times.
83.    - **Calculation**:
84.      - Turnaround time = Job end time - Job start time.
85.      - Results are written to `stats.txt` for analysis.
86.
87. ---
88.
89. ### **Utility Functions**
90.    - **Time Management**:
91.      - `get_current_time()`: Retrieves the current time in milliseconds for accurate logging
and statistics.
92.      - `msleep(milliseconds)`: Pauses execution for a specified number of milliseconds.
93.    - **File Operations**:
94.      - `create_counter_files(num_counters)`: Initializes counter files with a starting value of
`0`.
95.      - `increment_counter(counter_id)` / `decrement_counter(counter_id)`: Modifies the value in
the corresponding counter file.
96.    - **Queue Management**:
97.      - `enqueue_work(command)`: Adds a new job to the work queue.
98.      - `dequeue_work()`: Retrieves the next job from the queue for processing.
99.
100. ---
101.
102. ### **Summary**
103. This program efficiently implements a Dispatcher/Worker model, ensuring parallel job execution
and robust synchronization. It uses logging for detailed analysis and maintains statistics for
performance evaluation. Key features include:
104. - Flexible task handling through worker threads.
105. - Accurate time tracking for job performance metrics.
106. - Persistent counter management using files.
107.
108. This structure ensures scalability and reliability, making it suitable for various task
distribution scenarios.
109.
```