CSE 344 - Homework #4: Multithreaded Log File Analyzer

Student Name: Recep Furkan Akın
Student ID: 210104004042
Course: CSE 344
Homework #: 4

1. Introduction

This program, `LogAnalyzer`, is a multithreaded log file analyzer implemented in C using POSIX threads (pthreads). It demonstrates the producer-consumer pattern to process large log files efficiently. The program takes a log file, a search term, the size of a shared buffer, and the number of worker threads as command-line arguments.

The main thread acts as a **manager** (producer). It reads the specified log file line by line and places these lines into a shared, bounded buffer. Multiple **worker** threads (consumers) concurrently retrieve lines from this buffer, search for the user-defined keyword within each line, and count the occurrences.

Synchronization between the manager and worker threads for accessing the shared buffer is achieved using `pthread_mutex_t` for mutual exclusion and `pthread_cond_t` for signaling (i.e., manager waits if the buffer is full, workers wait if it's empty). Additionally, a `pthread_barrier_t` is used to ensure all worker threads complete their search and counting tasks before a final summary report of total matches is printed by one designated worker thread. The program also includes error handling, graceful shutdown on SIGINT (Ctrl+C), and aims for proper memory management.

2. Code Explanation

The program is structured into several C files: `210104004042_main.c` (containing the main logic, manager and worker threads), `buffer.c` (implementing the shared buffer), and `buffer.h` (the header file for the buffer).

2.1. Buffer Implementation (`buffer.h` and `buffer.c`)

The shared buffer is a critical component for communication between the manager and worker threads. It is implemented as a thread-safe circular queue.

2.1.1. `Buffer` Structure (`buffer.h`)

```
typedef struct
{
```

```
    char **data;              // Array of strings (lines from the log
file)
    int size;                 // Maximum capacity of the buffer
    int count;                // Current number of items in the buffer
    int head;                 // Index where the next item will be added
    int tail;                 // Index where the next item will be
removed
    pthread_mutex_t mutex;    // Mutex for thread-safe access
    pthread_cond_t not_full;  // Condition variable to signal when
buffer is not full
    pthread_cond_t not_empty; // Condition variable to signal when
buffer is not empty
} Buffer;
```

- `data`: A dynamically allocated array of character pointers. Each pointer will store a line read from the log file.
- `size`: The maximum number of lines the buffer can hold (defined by a command-line argument).
- `count`: The current number of lines present in the buffer.
- `head`: The index in the `data` array where the next line produced by the manager will be inserted.
- `tail`: The index in the `data` array from which the next line will be consumed by a worker.
- `mutex`: A `pthread_mutex_t` used to protect the buffer's shared data (`data`, `count`, `head`, `tail`) from concurrent access, ensuring atomicity of buffer operations.
- `not_full`: A `pthread_cond_t` used by the manager. If the buffer is full (`count == size`), the manager waits on this condition variable. Workers signal `not_full` after consuming an item.
- `not_empty`: A `pthread_cond_t` used by worker threads. If the buffer is empty (`count == 0`), workers wait on this condition variable. The manager signals `not_empty` after producing an item.

2.1.2. `init_buffer()` (`buffer.c`)

```
int init_buffer(Buffer *buffer, int size);
```

- **Purpose:** Initializes the `Buffer` structure.
- **Functionality:**
  1. Validates input arguments (`buffer` pointer and `size`).
  2. Dynamically allocates memory for the `buffer->data` array (an array of `char*`) based on the given `size`.
  3. Initializes `buffer->size` to the provided size, and `buffer->count`, `buffer->head`, and `buffer->tail` to 0.

4. Initializes the `pthread_mutex_t` (`buffer->mutex`) using `pthread_mutex_init()`.
5. Initializes the two `pthread_cond_t` variables (`buffer->not_full` and `buffer->not_empty`) using `pthread_cond_init()`.
6. Performs error checking for memory allocation and initialization of synchronization primitives, returning -1 on failure and 0 on success.

2.1.3. `add_to_buffer()` (`buffer.c`)

```
void add_to_buffer(Buffer *buffer, char *line);
```

- **Purpose:** Adds a line (produced by the manager) to the shared buffer. This is the "produce" operation.
- **Functionality:**
  1. Locks the `buffer->mutex` to gain exclusive access.
  2. **Waits if full:** Enters a `while` loop that checks if `buffer->count == buffer->size` (buffer is full) AND the `running` flag is true.
     - If full and `running` is true, it calls `pthread_cond_wait(&buffer->not_full, &buffer->mutex)`. This atomically releases the mutex and puts the manager thread to sleep until `buffer->not_full` is signaled by a worker. Upon waking, it re-acquires the mutex.
     - If `running` becomes false while waiting or before adding, it unlocks the mutex, frees the `line` (as it won't be added), and returns to allow the manager to terminate.
  3. **Adds item:** If there's space (or space becomes available), it places the `line` (a `char*`) into `buffer->data[buffer->head]`.
  4. Updates `buffer->head = (buffer->head + 1) % buffer->size` to implement circular behavior.
  5. Increments `buffer->count`.
  6. **Signals consumer:** Calls `pthread_cond_signal(&buffer->not_empty)` to wake up one waiting worker thread (if any), as there is now an item in the buffer.
  7. Unlocks the `buffer->mutex`.

2.1.4. `remove_from_buffer()` (`buffer.c`)

```
char *remove_from_buffer(Buffer *buffer);
```

- **Purpose:** Removes and returns a line from the shared buffer for a worker thread to process. This is the "consume" operation.
- **Functionality:**
  1. Locks the `buffer->mutex`.

2. **Waits if empty:** Enters a `while` loop that checks if `buffer->count == 0` (buffer is empty) AND the `running` flag is true.
    - If empty and `running` is true, it calls `pthread_cond_wait(&buffer->not_empty, &buffer->mutex)`. This atomically releases the mutex and puts the worker thread to sleep until `buffer->not_empty` is signaled by the manager. Upon waking, it re-acquires the mutex.
    - If `running` becomes false while waiting or before removing, it unlocks the mutex and returns `NULL` to allow the worker to terminate.
3. **Removes item:** If there's an item (or an item becomes available), it retrieves the `char*` line from `buffer->data[buffer->tail]`.
4. Sets `buffer->data[buffer->tail] = NULL` for safety (optional, but good practice).
5. Updates `buffer->tail = (buffer->tail + 1) % buffer->size`.
6. Decrements `buffer->count`.
7. **Signals producer:** Calls `pthread_cond_signal(&buffer->not_full)` to wake up the manager thread (if it's waiting because the buffer was full), as there is now space.
8. Unlocks the `buffer->mutex`.
9. Returns the retrieved `line`. The caller (worker thread) is responsible for `free`ing this line.

2.1.5. `free_buffer()` (`buffer.c`)

```
void free_buffer(Buffer *buffer);
```

- **Purpose:** Cleans up all resources associated with the buffer.
- **Functionality:**
    1. Checks if `buffer->data` is already `NULL` to prevent double freeing.
    2. Iterates through any remaining items in the buffer (from `buffer->tail` for `buffer->count` items) and `free`s each `char*` line. This is important if the program terminates while items are still in the buffer (e.g., due to an error or SIGINT).
    3. `free`s the `buffer->data` array itself.
    4. Destroys the mutex using `pthread_mutex_destroy(&buffer->mutex)`.
    5. Destroys the condition variables using `pthread_cond_destroy(&buffer->not_full)` and `pthread_cond_destroy(&buffer->not_empty)`.

2.2. Main Program Logic (`210104004042_main.c`)

This file contains the `main` function, the manager thread function, the worker thread function, and the signal handler.

Global Variables:

- `volatile int running = 1;` : A flag used to control the main loops of the manager and worker threads. It's `volatile` because it's modified by a signal handler and accessed by multiple threads. Set to `0` to signal threads to terminate.
- `int num_workers;` : Stores the number of worker threads to be created, parsed from command-line arguments.
- `int *match_counts;` : A dynamically allocated array where each worker `i` stores its count of found matches at `match_counts[i]` .
- `char *search_term;` : Stores the keyword to search for in log lines, parsed from command-line arguments.
- `Buffer buffer;` : The shared buffer instance.
- `pthread_barrier_t barrier;` : The barrier used to synchronize worker threads before the final summary report.

2.2.1. `manager()` (Producer Thread)

```
void *manager(void *arg); // arg is char *file_name
```

- **Purpose:** Reads lines from the log file and adds them to the shared buffer.
- **Functionality:**
  1. Takes the log file name as an argument.
  2. Opens the log file using `open()` in read-only mode. Handles file open errors.
  3. Uses a `read_buf` to read chunks from the file and a `current_line` buffer to assemble lines.
  4. Reads data from the file in chunks using `read()` .
  5. Iterates through the `read_buf` character by character:
     - If a newline (`\n`) is encountered, the `current_line` is complete.
     - It `strdup()` s the `current_line` to create a dynamically allocated copy (since `current_line` buffer will be reused). Handles `strdup` errors.
     - Calls `add_to_buffer()` to put the copied line into the shared buffer.
     - Resets `current_line_idx` .
     - If a character is not a newline, it's appended to `current_line` . Handles potential line overflow by truncating and processing.
  6. The loop continues as long as `running` is true and `read()` returns positive bytes.
  7. After the loop, it handles any remaining characters in `current_line` (if the file doesn't end with a newline).
  8. Handles `read()` errors.
  9. Closes the file descriptor using `close()` .
  10. **EOF Signaling:** After processing the entire file (or if `running` becomes false), it adds `num_workers` special `EOF_MARKER` strings (defined as `"END"`) to the buffer. Each worker consuming an `EOF_MARKER` will know that there's no more data. These markers are also `strdup()` d.

11. If critical errors occur (file open, `strdup`), it sets `running = 0` and broadcasts on `buffer.not_empty` to wake up any waiting workers so they can terminate.

2.2.2. `worker()` (Consumer Thread)

```
void *worker(void *arg); // arg is int *id (worker's ID)
```

- **Purpose:** Consumes lines from the shared buffer, searches for the `search_term`, and counts matches.
- **Functionality:**
  1. Takes its worker ID as an argument. Initializes a local `count` for matches to 0.
  2. Enters a loop that continues as long as `running` is true.
  3. Calls `remove_from_buffer()` to get a line.
     - If `remove_from_buffer()` returns `NULL` (e.g., `running` became false while waiting), the worker breaks the loop.
  4. **EOF Check:** Compares the retrieved line with `EOF_MARKER` using `strcmp()`. If it's the EOF marker, it `free()`s the line and breaks the loop, signaling the end of its work.
  5. **Search:** If not EOF, it uses `strstr(line, search_term)` to check if the `search_term` exists in the `line`. If found, increments its local `count`.
  6. `free()`s the `line` (which was `strdup()`d by the manager or `remove_from_buffer` if it was an EOF marker from the manager).
  7. After the loop terminates, it stores its local `count` into the shared `match_counts[id]` array.
  8. Prints the number of matches it found (e.g., "Worker X found Y matches").
  9. **Barrier Synchronization:** Calls `pthread_barrier_wait(&barrier)`. This blocks the worker until all `num_workers` threads have reached this barrier.
  10. **Summary Report:** The `pthread_barrier_wait()` returns `PTHREAD_BARRIER_SERIAL_THREAD` for one arbitrary thread after all threads arrive. The code specifically checks if `id == 0` (and `running` is true) after the barrier. If so, this worker (Worker 0) calculates the total matches by summing all values in `match_counts` and prints the "Total matches found".

2.2.3. `handle_signal()`

```
void handle_signal();
```

- **Purpose:** Gracefully handles `SIGINT` (Ctrl+C).
- **Functionality:**
  1. Sets the global `volatile int running` flag to `0`. This signals all threads (manager and workers) to terminate their main loops.

2. Calls `pthread_cond_broadcast(&buffer.not_full)` and `pthread_cond_broadcast(&buffer.not_empty)`. This is crucial: if any threads are currently blocked in `pthread_cond_wait()` on these condition variables (manager waiting because buffer is full, or workers waiting because buffer is empty), they will be woken up. Upon waking, they will re-check the `running` flag (which is now `0`) and proceed to terminate.

2.2.4. `main()`

```
int main(int argc, char *argv[]);
```

- **Purpose:** Parses arguments, initializes resources, creates and manages threads, and cleans up.
- **Functionality:**
  1. **Argument Parsing:**
     - Checks if `argc` is 5 (program name + 4 arguments). If not, prints usage message and exits.
     - Parses `<buffer_size>`, `<num_workers>`, `<log_file>`, and `<search_term>` from `argv`.
     - Validates that `buffer_size` and `num_workers` are positive.
  2. **Initialization:**
     - Calls `init_buffer(&buffer, buffer_size)` to initialize the shared buffer.
     - Allocates memory for `match_counts` array using `calloc` (to initialize counts to 0).
     - Initializes the `pthread_barrier_t` using `pthread_barrier_init(&barrier, NULL, num_workers)`. `num_workers` is the count of threads that must reach the barrier.
     - Handles errors from all initialization steps.
  3. **Signal Handling Setup:**
     - Sets up a signal handler for `SIGINT` using `sigaction()`. `handle_signal` is registered as the handler function.
  4. **Thread Creation:**
     - Creates the single manager thread using `pthread_create()`, passing `manager` as the thread function and `log_file` as its argument.
     - Allocates memory for an array of `pthread_t` handles (`workers`) and an array of `int` for worker IDs (`ids`).
     - Creates `num_workers` worker threads in a loop using `pthread_create()`. Each worker thread executes the `worker` function and is passed a pointer to its unique ID from the `ids` array.
     - Handles errors from `pthread_create()`. If a worker thread fails to create, it sets `running = 0`, wakes up any waiting threads, joins already created

threads, and cleans up before exiting.

5. **Thread Joining:**
   - Waits for the manager thread to complete using `pthread_join(manager_thread, NULL)`.
   - Waits for all worker threads to complete using `pthread_join()` in a loop.
6. **Cleanup:**
   - Calls `free_buffer(&buffer)` to release buffer resources.
   - `free()`s `match_counts`, `workers`, and `ids` arrays.
   - Destroys the barrier using `pthread_barrier_destroy(&barrier)`.
7. Returns `0` on successful completion.

3. Output and Final Report

- Each worker thread prints the number of matches it found individually. For example:
  `Worker 0 found 15 matches`
  `Worker 1 found 12 matches`
  ...
- After all worker threads have finished their processing and synchronized at the barrier, one designated worker thread (Worker 0 in this implementation) prints a summary report:
  `--------------------`
  `Total matches found: XX`
  (where XX is the sum of matches found by all workers).

4. Error Handling

- **Command-line Arguments:** If the wrong number of arguments is provided, or if `buffer_size` or `num_workers` are not positive integers, a usage message is printed to `stderr`, and the program exits with status 1.
  `Usage: ./LogAnalyzer <buffer_size> <num_workers> <log_file> <search_term>`
- **File Operations:** The `manager` thread handles errors during file opening (`open()`) and reading (`read()`) by printing an error message using `perror()` and attempting a graceful shutdown by setting `running = 0` and signaling workers.
- **Memory Allocation:** `malloc()`, `calloc()`, and `strdup()` failures are checked. If allocation fails, `perror()` is called, and the program attempts a graceful shutdown or exits.
- **Thread Creation/Synchronization Primitive Initialization:** Errors during `pthread_create()`, `pthread_mutex_init()`, `pthread_cond_init()`, and `pthread_barrier_init()` are checked. Error messages are printed (often using `strerror()` for pthread functions), and the program attempts to clean up and exit.
- **Signal Handling (SIGINT):** `SIGINT` (Ctrl+C) is handled by the `handle_signal` function. It sets `running = 0` and broadcasts on condition variables to wake up any

waiting threads, allowing them to exit their loops, free resources, and terminate gracefully.
- **Memory Management:** All dynamically allocated memory is intended to be freed. This includes lines in the buffer, the buffer's data array, `match_counts`, `workers` array, `ids` array. The requirement to test with `valgrind` emphasizes this.

5. Buffer Synchronization (Producer-Consumer Logic)

- **Mutual Exclusion:** The `buffer.mutex` is locked before any access or modification to the shared buffer's data (`data` array, `head`, `tail`, `count`). This prevents race conditions.
- **Manager Waits When Buffer Full:**
  - In `add_to_buffer()`, if `buffer.count == buffer.size` (buffer is full), the manager thread calls `pthread_cond_wait(&buffer.not_full, &buffer.mutex)`.
  - This atomically releases the `mutex` and puts the manager to sleep.
  - It waits until a worker thread consumes an item and signals `buffer.not_full`.
- **Workers Wait When Buffer Empty:**
  - In `remove_from_buffer()`, if `buffer.count == 0` (buffer is empty), the worker thread calls `pthread_cond_wait(&buffer.not_empty, &buffer.mutex)`.
  - This atomically releases the `mutex` and puts the worker to sleep.
  - It waits until the manager thread adds an item and signals `buffer.not_empty`.
- **No Busy-Waiting:** The use of condition variables (`pthread_cond_wait`, `pthread_cond_signal`) ensures that threads sleep when they cannot proceed, avoiding wasteful CPU cycles associated with busy-waiting.
- **Signaling:**
  - When the manager adds an item, it calls `pthread_cond_signal(&buffer.not_empty)` to wake up *one* potentially waiting worker.
  - When a worker removes an item, it calls `pthread_cond_signal(&buffer.not_full)` to wake up the *manager* if it's waiting due to a full buffer.
- **Spurious Wakeups:** The conditions (`buffer.count == buffer.size` and `buffer.count == 0`) are checked in `while` loops (e.g., `while (buffer->count == 0)`) rather than `if` statements. This is standard practice with condition variables to handle spurious wakeups correctly and to re-verify the condition after waking up.
- **Graceful Shutdown Integration:** The `running` flag is checked within the wait loops and before operations. If `running` becomes `0`, threads exit their wait loops and proceed to terminate, and allocated items that are not added to the buffer (in `add_to_buffer`) are freed.

6. Barrier Use

- A `pthread_barrier_t barrier` is initialized in `main()` with a count equal to `num_workers`.
  `pthread_barrier_init(&barrier, NULL, num_workers);`
- Each worker thread, after finishing its line processing loop and recording its local match count in the `match_counts` array, calls:
  `pthread_barrier_wait(&barrier);`
- This call blocks the worker thread until all `num_workers` threads have called `pthread_barrier_wait()`.
- Once all workers have reached the barrier, they are all unblocked simultaneously.
- The `pthread_barrier_wait()` function returns `PTHREAD_BARRIER_SERIAL_THREAD` to exactly one of the synchronized threads (chosen arbitrarily by the implementation) and `0` to the others.
- In this program, after the barrier, the worker with `id == 0` is designated to perform the final task: calculating the sum of all entries in `match_counts` and printing the "Total matches found" summary. This ensures that the summary is printed only after all workers have completed their individual counts and contributed to the `match_counts` array.
- The barrier is destroyed in `main()` using `pthread_barrier_destroy(&barrier)` during cleanup.

7. Testing Scenario Screenshots

Test Scenario 1: Valgrind Memory Check

Command: `./LogAnalyzer 10 4 logs/sample.log "ERROR"`
(Buffer size: 10, Workers: 4)

```
recepfurkanakin@system:~/SystemHW25/hw4$ valgrind ./LogAnalyzer 10 4 logs/sample.log "ERROR"
==463822== Memcheck, a memory error detector
==463822== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==463822== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==463822== Command: ./LogAnalyzer 10 4 logs/sample.log ERROR
==463822==
Worker 1 found 1 matches
Worker 0 found 0 matches
Worker 2 found 1 matches
Worker 3 found 0 matches
--------------------
Total matches found: 2
==463822==
==463822== HEAP SUMMARY:
==463822==     in use at exit: 0 bytes in 0 blocks
==463822==   total heap usage: 34 allocs, 34 frees, 3,708 bytes allocated
==463822==
==463822== All heap blocks were freed -- no leaks are possible
==463822==
==463822== For lists of detected and suppressed errors, rerun with: -s
==463822== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
recepfurkanakin@system:~/SystemHW25/hw4$
```

Note: sample.log is the one grader provided.

Test Scenario 2: Large Log Analysis

Command: `./LogAnalyzer 50 8 logs/large.log "GET"`
(Buffer size: 50, Workers: 8)

```
recepfurkanakin@system:~/SystemHW25/hw4$ valgrind --leak-check=full --show-leak-kinds=all ./LogAnalyzer 50 8 logs/large.log "GET"
==465009== Memcheck, a memory error detector
==465009== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==465009== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==465009== Command: ./LogAnalyzer 50 8 logs/large.log GET
==465009==
Worker 1 found 2262 matches
Worker 5 found 2167 matches
Worker 2 found 2424 matches
Worker 0 found 2208 matches
Worker 7 found 2460 matches
Worker 6 found 2382 matches
Worker 3 found 1914 matches
Worker 4 found 2184 matches
--------------------
Total matches found: 18001
==465009==
==465009== HEAP SUMMARY:
==465009==     in use at exit: 0 bytes in 0 blocks
==465009==   total heap usage: 21,023 allocs, 21,023 frees, 1,774,033 bytes allocated
==465009==
==465009== All heap blocks were freed -- no leaks are possible
==465009==
==465009== For lists of detected and suppressed errors, rerun with: -s
==465009== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
recepfurkanakin@system:~/SystemHW25/hw4$
```

Note: Large log file is an AI-generated dummy file. (provided under logs file)

Test Scenario 3: Large Log Analysis (With Interrupt)

Command: `./LogAnalyzer 50 8 logs/large.log "GET"`
(Buffer size: 50, Workers: 8)

```
recepfurkanakin@system:~/SystemHW25/hw4$ valgrind --leak-check=full --show-leak-kinds=all ./LogAnalyzer 50 8 logs/large.log "GET"
==465009== Memcheck, a memory error detector
==465009== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==465009== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==465009== Command: ./LogAnalyzer 50 8 logs/large.log GET
==465009==
Worker 1 found 2262 matches
Worker 5 found 2167 matches
Worker 2 found 2424 matches
Worker 0 found 2208 matches
Worker 7 found 2460 matches
Worker 6 found 2382 matches
Worker 3 found 1914 matches
Worker 4 found 2184 matches
--------------------
Total matches found: 18001
==465009==
==465009== HEAP SUMMARY:
==465009==     in use at exit: 0 bytes in 0 blocks
==465009==   total heap usage: 21,023 allocs, 21,023 frees, 1,774,033 bytes allocated
==465009==
==465009== All heap blocks were freed -- no leaks are possible
==465009==
==465009== For lists of detected and suppressed errors, rerun with: -s
==465009== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
recepfurkanakin@system:~/SystemHW25/hw4$
```

Note: Same log file interrupted at the middle of execution.

8. Conclusion

This assignment involved developing a multithreaded log file analyzer using key POSIX synchronization primitives: mutexes, condition variables, and barriers. The program successfully implements the producer-consumer pattern, with a manager thread producing log lines and multiple worker threads consuming them to search for a specific term.

Challenges Faced & Solutions:

- **Synchronization Logic:** Ensuring correct use of mutexes and condition variables to prevent race conditions and deadlocks, while also avoiding busy-waiting, was a primary challenge. The solution involved careful locking around shared buffer access, using `pthread_cond_wait` in `while` loops for conditions (buffer full/empty), and appropriate `pthread_cond_signal` calls.
- **Graceful Shutdown (SIGINT):** Implementing a clean exit upon receiving Ctrl+C required a `volatile` global flag (`running`) and broadcasting on condition variables (`pthread_cond_broadcast`) in the signal handler. This ensures that threads blocked on `pthread_cond_wait` are awakened and can check the `running` flag to terminate properly.
- **Memory Management:** Dynamically allocating strings for each log line (`strdup`) and ensuring they are freed by the consumers or during cleanup (`free_buffer`) was crucial. EOF markers also needed careful memory management. Testing with Valgrind helped identify and fix any memory leaks.
- **Line-by-Line File Reading:** Reading a file line by line when `read()` gives chunks of data required careful buffer management to handle lines that might span across multiple `read()` calls and to correctly identify newline characters.
- **Barrier Synchronization:** Using `pthread_barrier_wait` correctly to ensure all workers finish before printing the summary report was straightforward once the concept was understood. Designating one worker (e.g., ID 0) to print the summary after the barrier simplifies the logic.

Final Thoughts:

This project provided valuable hands-on experience with concurrent programming concepts in C. The producer-consumer model is a common and powerful pattern, and understanding how to implement it correctly with pthreads is essential for developing efficient multithreaded applications. The use of barriers for multi-phase synchronization is also a useful technique. The requirement for robust error handling and memory management further reinforces good programming practices. The program demonstrates a functional and reasonably robust solution to the log analysis problem using multithreading.