CSE 344 - Homework #4 Report

Multithreaded Log File Analyzer

Student Name: Anhelina Bondarenko

Student ID: 220104004928

Course: CSE 344 - System Programming

Homework #: 4

Date: May 12, 2025

1. Introduction

This report documents the implementation of a Multithreaded Log File Analyzer program developed in C using POSIX threads. The program is designed to read log files in parallel, search for user-specified keywords, and report matching lines. The implementation uses a producer-consumer pattern with a shared buffer, where the manager thread (producer) reads the log file line by line and places them into the buffer, while multiple worker threads (consumers) process these lines and search for the specified keyword.

The program demonstrates several important multithreading concepts:

- Thread synchronization using mutexes and condition variables
- Producer-consumer problem solution with a bounded buffer
- Thread coordination using barriers
- Signal handling for graceful termination
- Memory management in a multithreaded environment

2. Code Explanation

The implementation consists of three main files:

- main.c: Contains the main program logic, thread functions, and signal handlers
- buffer.c: Implements the thread-safe bounded buffer
- buffer.h: Defines the buffer structure and interface

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

2.1.1 Buffer Structure

```
typedef struct {
    char **items;           // Array of string pointers
    int capacity;           // Maximum buffer size
    int size;               // Current number of items
    int front;              // Index of the front item
    int rear;               // Index of the last item
    pthread_mutex_t mutex;  // Mutex for thread safety
    pthread_cond_t not_full;    // Condition variable for producers
    pthread_cond_t not_empty;   // Condition variable for consumers
} buffer_t;
```

The buffer is implemented as a circular queue for efficient memory usage. It maintains pointers to strings, along with synchronization primitives.

2.1.2 Buffer Creation and Destruction

The buffer_create function initializes the buffer with a specified capacity, allocating memory for the items array and initializing synchronization primitives:

```
buffer_t* buffer_create(int capacity) {
    buffer_t *buffer = malloc(sizeof(buffer_t));
    if (buffer == NULL) {
        return NULL;
    }
    buffer->items = malloc(capacity * sizeof(char*));
    if (buffer->items == NULL) {
        free(buffer);
        return NULL;
    }
    buffer->capacity = capacity;
    buffer->size = 0;
    buffer->front = 0;
    buffer->rear = -1;

    // Initialize synchronization primitives
    if (pthread_mutex_init(&buffer->mutex, NULL) != 0) {
```

```
        free(buffer->items);
        free(buffer);
        return NULL;
    }
    // ... initialize condition variables ...
    return buffer;
}
```

The buffer_destroy function cleans up all resources, including freeing any remaining items in the buffer and destroying synchronization primitives.

2.1.3 Buffer Operations

The buffer_add function adds an item to the buffer, blocking if the buffer is full:

```
void buffer_add(buffer_t *buffer, char *item) {
    pthread_mutex_lock(&buffer->mutex);

    // Wait until there's space in the buffer or termination signal
    while (buffer->size == buffer->capacity && !terminate) {
        pthread_cond_wait(&buffer->not_full, &buffer->mutex);
    }

    // ... add item to buffer ...

    // Signal that buffer is not empty
    pthread_cond_signal(&buffer->not_empty);
    pthread_mutex_unlock(&buffer->mutex);
}
```

The buffer_remove function removes and returns an item from the buffer, blocking if the buffer is empty:

```
char* buffer_remove(buffer_t *buffer) {
    pthread_mutex_lock(&buffer->mutex);

    // Wait until there's an item in the buffer or termination signal
    while (buffer->size == 0 && !terminate) {
        pthread_cond_wait(&buffer->not_empty, &buffer->mutex);
    }
    // ... remove item from buffer ..
    // Signal that buffer is not full
    pthread_cond_signal(&buffer->not_full);
    pthread_mutex_unlock(&buffer->mutex);
    return item;
}
```

2.2 Main Program (main.c)

2.2.1 Global Variables and Data Structures

```
buffer_t *buffer = NULL;
pthread_t *workers = NULL;
int num_workers = 0;
pthread_barrier_t barrier;
volatile sig_atomic_t terminate = 0;
```

The program uses several global variables for coordination and cleanup:

- buffer: Shared buffer between threads
- workers: Array of worker thread IDs
- num_workers: Number of worker threads
- barrier: Barrier for synchronizing worker threads
- terminate: Flag for signaling program termination

2.2.2 Thread Argument Structures

```
typedef struct {
    buffer_t *buffer;
    char *search_term;
    int worker_id;
    int matches_found;
} worker_args_t;

typedef struct {
    buffer_t *buffer;
    char *filename;
} manager_args_t;
```

These structures are used to pass arguments to the manager and worker threads.

2.2.3 Signal Handler

```
void signal_handler(int sig) {
    if (sig == SIGINT) {
        printf("\nReceived SIGINT, cleaning up and exiting...\n");
        terminate = 1;
        if (buffer != NULL) {
            buffer_signal_all(buffer); // Wake up any waiting threads
        }
    }
}
```

The signal handler sets the termination flag and wakes up any waiting threads, allowing for graceful shutdown.

2.2.4 Manager Thread

```
void *manager_thread(void *arg) {
    manager_args_t *args = (manager_args_t *)arg;
```

```
    // ... open file and read lines ...
    while (!terminate && (read = getline(&line, &len, file)) != -1) {
        // ... process line and add to buffer ...
        buffer_add(buffer, line_copy);
    }
    // Add EOF marker to signal end of file
    if (!terminate) {
        for (int i = 0; i < num_workers; i++) {
            buffer_add(buffer, NULL);
        }
    }
    // ... cleanup ...
    return NULL;
}
```

The manager thread reads the input file line by line, adds each line to the shared buffer, and finally adds EOF markers (NULL pointers) to signal that no more data is available.

2.2.5 Worker Thread

```
void *worker_thread(void *arg) {
    worker_args_t *args = (worker_args_t *)arg;
    // ... process lines from buffer ...
    while (!terminate) {
        char *line = buffer_remove(buffer);
        // Check if it's the EOF marker
        if (line == NULL) {
            break;
        }
        // Search for the keyword
        if (strstr(line, search_term) != NULL) {
            printf("Worker %d found match: %s\n", worker_id, line);
            matches++;
        }
        free(line);
    }
    args->matches_found = matches;
    pthread_barrier_wait(&barrier); // Wait for all workers to finish
    // ... print summary (only one thread) ...
    return NULL;
}
```

Worker threads consume lines from the buffer, search for the keyword, and wait at a barrier before printing the final report.

2.2.6 Main Function

```
int main(int argc, char *argv[]) {
    // ... parse and validate arguments ...
```

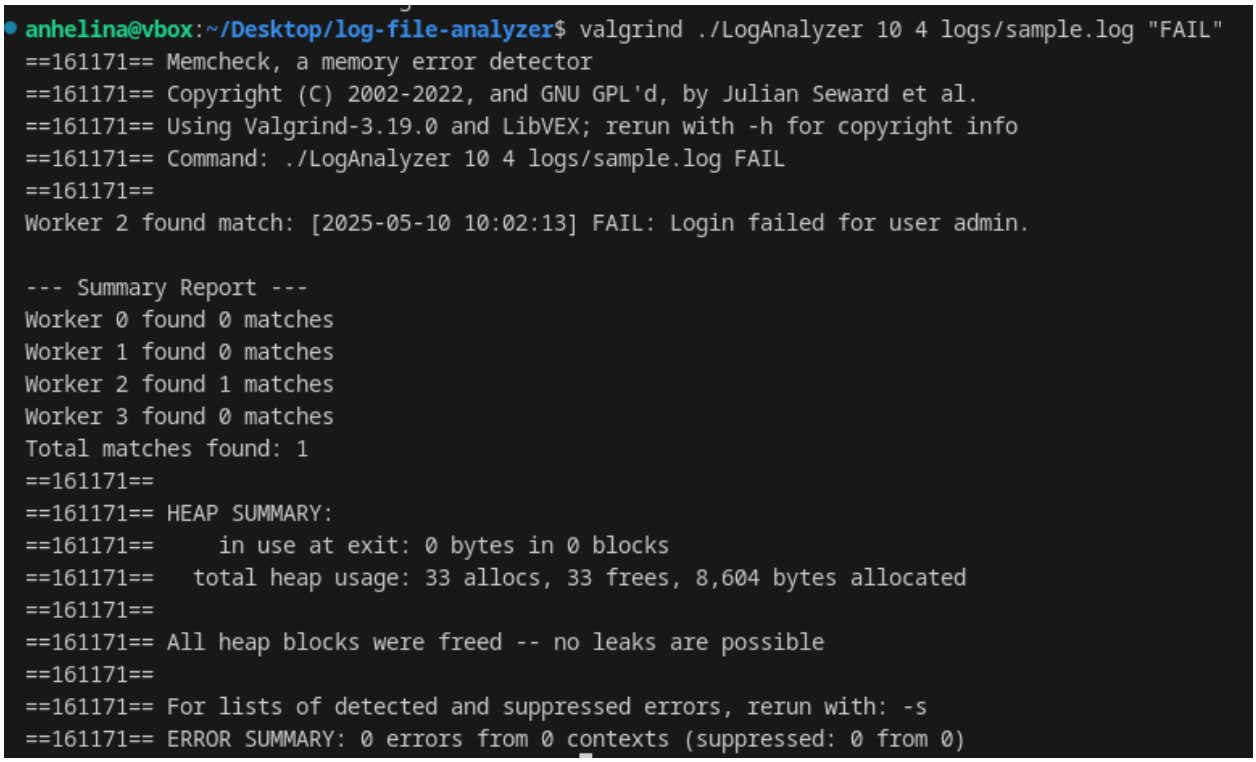
```
    // Register signal handler
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = signal_handler;
    sigaction(SIGINT, &sa, NULL);

    // Initialize buffer and barrier
    buffer = buffer_create(buffer_size);
    pthread_barrier_init(&barrier, NULL, num_workers);
    // ... create worker threads ...

    // Start manager thread
    pthread_t manager;
    pthread_create(&manager, NULL, manager_thread, &manager_args);
    // ... wait for threads to finish ..
    // Clean up resources
    free(worker_args);
    cleanup_resources();

    return 0;
}
```

The main function parses command-line arguments, sets up resources, creates threads, waits for them to complete, and cleans up resources.

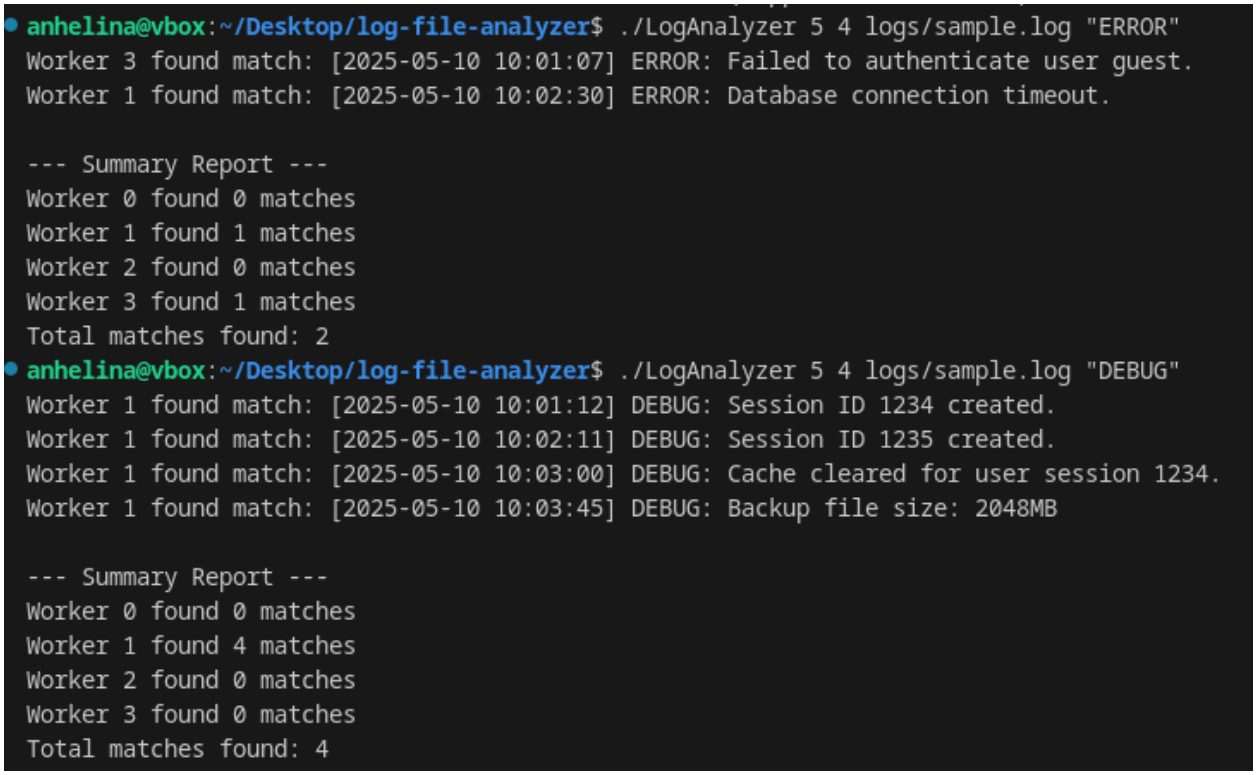
3. Screenshots and Test Results

Test Case 1: Memory Leak Check with Valgrind

$ valgrind ./LogAnalyzer 10 4 logs/sample.log "FAIL"



Test Case 2: Small Buffer and Few Workers



```
anhelina@vbox:~/Desktop/log-file-analyzer$ ./LogAnalyzer 5 4 logs/sample.log "404"
Worker 3 found match: [2025-05-10 10:01:46] 404: Page not found /about.html
Worker 1 found match: [2025-05-10 10:01:47] 404: Page not found /contact.html
Worker 3 found match: [2025-05-10 10:03:20] 404: Page not found /missing.js

--- Summary Report ---
Worker 0 found 0 matches
Worker 1 found 1 matches
Worker 2 found 0 matches
Worker 3 found 2 matches
Total matches found: 3
```

```
anhelina@vbox:~/Desktop/log-file-analyzer$ ./LogAnalyzer 5 4 logs/sample.log "INFO"
Worker 3 found match: [2025-05-10 10:00:00] INFO: Starting web server on port 80.
Worker 3 found match: [2025-05-10 10:02:10] INFO: Connection received from 10.0.0.3.
Worker 3 found match: [2025-05-10 10:02:12] INFO: Serving page /login.html
Worker 3 found match: [2025-05-10 10:02:35] INFO: System health check passed.
Worker 3 found match: [2025-05-10 10:03:10] INFO: Connection closed from 10.0.0.2.
Worker 3 found match: [2025-05-10 10:03:30] INFO: Backup process started.
Worker 3 found match: [2025-05-10 10:04:00] INFO: Backup completed successfully.
Worker 2 found match: [2025-05-10 10:01:05] INFO: Connection received from 10.0.0.2.
Worker 1 found match: [2025-05-10 10:01:45] INFO: Serving page /index.html

--- Summary Report ---
Worker 0 found 0 matches
Worker 1 found 1 matches
Worker 2 found 1 matches
Worker 3 found 7 matches
Total matches found: 9
```

```
anhelina@vbox:~/Desktop/log-file-analyzer$ ./LogAnalyzer 5 2 logs/sample.log "WARN"
Worker 1 found match: [2025-05-10 10:02:14] WARN: User account locked after 3 failed attempts.

--- Summary Report ---
Worker 0 found 0 matches
Worker 1 found 1 matches
Total matches found: 1
```

Test Case 3: Error handling

```
anhelina@vbox:~/Desktop/log-file-analyzer$ ./LogAnalyzer 5 2 logs/sample.log "COMPUTER"

--- Summary Report ---
Worker 0 found 0 matches
Worker 1 found 0 matches
Total matches found: 0
anhelina@vbox:~/Desktop/log-file-analyzer$ ./LogAnalyzer 5  logs/sample.log "COMPUTER"
Usage: ./LogAnalyzer <buffer_size> <num_workers> <log_file> <search_term>
```

4. Conclusion

Challenges Faced

1. **Thread Synchronization:** Implementing correct synchronization for the shared buffer was challenging. I had to ensure that the manager thread would wait when the buffer was full and worker threads would wait when the buffer was empty. Using condition variables helped solve this issue.
2. **Signal Handling:** Ensuring graceful termination on SIGINT (Ctrl+C) required careful handling of the terminate flag and waking up any waiting threads.
3. **Memory Management:** Preventing memory leaks in a multithreaded environment required careful tracking of allocated memory and ensuring proper cleanup, especially during abnormal termination.
4. **Resource Cleanup:** Ensuring all resources (threads, mutex, condition variables, barrier) were properly cleaned up required implementing a comprehensive cleanup function.

Solutions

1. **Buffer Implementation:** I implemented the buffer as a circular queue for efficiency and used mutex and condition variables for thread safety.
2. **Signal Handling:** I used a volatile sig_atomic_t flag to signal termination and implemented a signal handler to set this flag and wake up waiting threads.
3. **Memory Management:** I implemented proper memory cleanup in all paths, including normal termination, signal-based termination, and error paths.
4. **Thread Coordination:** I used a barrier to synchronize worker threads before printing the final summary, ensuring that all threads had completed their work.

Final Thoughts

This project provided valuable experience in multithreaded programming concepts, including thread synchronization, producer-consumer patterns, and resource management. The use of POSIX threads, mutexes, condition variables,

and barriers demonstrated how to implement complex multithreaded applications in C.

The program successfully meets all the requirements specified in the assignment, providing a flexible and efficient solution for analyzing log files using parallel processing. With multiple worker threads searching for keywords simultaneously, the program can process large log files efficiently, making it a practical tool for real-world applications.