**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"

Зображення, що містить текст, знімок екрана, Шрифт

Вміст, створений ШІ, може бути неправильним.

**Test Case 2: Small Buffer and Few Workers**Зображення, що містить текст, знімок екрана, Шрифт

Вміст, створений ШІ, може бути неправильним.Зображення, що містить текст, знімок екрана, Шрифт

Вміст, створений ШІ, може бути неправильним.Зображення, що містить текст, знімок екрана, Шрифт, дизайн

Вміст, створений ШІ, може бути неправильним.

Зображення, що містить текст, знімок екрана, Шрифт

Вміст, створений ШІ, може бути неправильним.

**Test Case 3: Error handling**

Зображення, що містить текст, знімок екрана, Шрифт

Вміст, створений ШІ, може бути неправильним.

**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.