Multithreaded Log File Analyzer

CSE 344 - System Programming

Homework #4

Student Name: Ziya Kadir TOKLUOGLU

Student ID: 210104004228

May 12, 2025

Contents

1 Introduction

1.1 Problem Statement

This report documents the development of a Multithreaded Log File Analyzer implemented in C using POSIX threads. The application is designed to efficiently search through large log files for specific keywords by dividing the work among multiple worker threads, coordinated by a manager thread.

1.2 Overview of the Solution

The program follows a producer-consumer pattern where:

- A **manager thread** (producer) reads the log file and adds lines to a shared buffer
- Multiple **worker threads** (consumers) take lines from the buffer and search for matches
- The threads communicate through a thread-safe bounded buffer with proper synchronization

This multithreaded approach allows the program to process large log files more efficiently by leveraging the parallel processing capabilities of modern CPUs.

1.3 Key Features

The implementation includes:

- A thread-safe bounded linked list buffer
- POSIX synchronization mechanisms (mutexes, condition variables, barriers)
- Signal handling for graceful termination
- Boyer-Moore string search algorithm
- Chunk-based file reading for optimal I/O performance
- Per-thread statistics tracking and reporting
- Comprehensive final summary reporting

2 Code Explanation

2.1 Program Architecture

The program is structured into three main components:

- `main.c`: Contains the main function, thread creation logic, manager function, and worker function
- `buffer.c/h`: Implements the thread-safe bounded buffer data structure
- `utils.c/h`: Provides utility functions for command-line parsing and string searching

2.2 Data Structure Design

2.2.1 Thread-Safe Bounded Linked List

The core data structure is a bounded thread-safe linked list that serves as the shared buffer between threads. This structure was chosen for several important reasons:

- **Dynamic memory allocation**: Log lines vary greatly in length, making fixed-size arrays inefficient
- **O(1) operations**: Both enqueue and dequeue operations are constant time regardless of buffer size
- **No pre-allocation required**: Memory is allocated only when needed, unlike fixed arrays
- **No wraparound complexity**: Unlike circular buffers, linked lists don't require complex index management

The implementation uses a head and tail pointer approach for efficient operations at both ends. The `count` field tracks current size for bound checking, while the `capacity` field enforces the maximum buffer size specified by the user.

The synchronization mechanism uses three key components:

- `mutex`: Ensures atomic updates to the linked list structure
- `not_full`: Condition variable for producers waiting when buffer is at capacity

- `not_empty`: Condition variable for consumers waiting when buffer is empty

The `eof` field serves as the termination signal, indicating to worker threads that no more data will be added to the buffer and they should finish processing.

```
typedef struct Buffer_node {
    char* data;
    struct Buffer_node* next;
} Buffer_node;

typedef struct {
    Buffer_node* head;
    Buffer_node* tail;
    int count;
    int capacity;

    pthread_mutex_t mutex;
    pthread_cond_t not_full;
    pthread_cond_t not_empty;

    bool eof;
} Bounded_TS_Linked_List;
```

Listing 1: Buffer Data Structure

2.2.2 Thread Arguments Structure

To pass data to worker threads and collect results, a custom structure is used:

```
typedef struct {
    Bounded_TS_Linked_List* buffer;
    Program_config* config;
    pthread_barrier_t* barrier;
    int thread_id;
    int match_count;
    struct timespec start_time;
    struct timespec end_time;
} Thread_Arg;
```

Listing 2: Thread Arguments Structure

This structure enables:

- Passing shared resources to threads
- Keeping track of thread-specific statistics
- Storing timing information for performance reporting

2.3 Thread Management

2.3.1 Manager Thread

The manager thread employs several optimization techniques to efficiently process large log files:

- **Chunk-based reading**: Instead of reading line-by-line, the manager reads data in 4KB chunks using `read()`, which minimizes system calls and significantly improves I/O performance.
- **Adaptive line buffering**: A dynamic buffer grows as needed to handle arbitrarily long lines, allocating memory only when required.
- **Efficient line splitting**: Uses `strchr()` to locate newlines, which is more efficient than character-by-character scanning.
- **Partial line handling**: Carefully manages lines that span multiple chunks by maintaining state between reads, ensuring data integrity regardless of line size or chunk boundaries.
- **Memory preallocation**: Pre-reserves buffer space for efficiency using the formula: `line_buffer_size = line_length + piece_length + READ_BUFFER_SIZE`, reducing the number of reallocation operations needed.
- **Progress reporting**: Reports progress periodically (every 500 lines) without flooding the console, using carriage returns for cleaner display.

The manager implements a throttling mechanism through condition variables, pausing when the buffer reaches capacity and resuming when space becomes available. This prevents memory exhaustion while maintaining high throughput.

Time tracking is implemented using `CLOCK_MONOTONIC` for accurate performance measurement that's immune to system time changes.

2.3.2 Worker Threads

The implementation uses a thread pool design pattern where multiple worker threads process data in parallel. Each worker thread:

- **Operates independently**: Minimizes synchronization overhead by only communicating through the buffer

- **Self-monitors performance**: Tracks its own execution time and match statistics
- **Reports matches in real-time**: Shows matches as they are found rather than buffering them
- **Reports its own match count**: Each worker prints its match count before reaching the barrier
- **Has consistent termination logic**: Handles three termination conditions: EOF signal, empty buffer with EOF, and SIGINT

The thread pool design enables near-linear scaling with the number of CPU cores, minimizing thread context switching overhead by having each thread perform substantial work between buffer operations.

Thread parameters are passed through a single `Thread_Arg` structure, which also stores performance data. This approach simplifies thread creation while enabling detailed per-thread reporting.

Thread 0 is designated as the "leader thread" and has special responsibilities after the barrier:

- Collecting statistics from all worker threads using pointer arithmetic (`all_args = args - args->thread_id`)
- Generating a comprehensive performance report showing processing time for each thread
- Calculating the total matches found across all threads
- Reporting buffer statistics to help analyze buffer size efficiency

This leader thread approach minimizes the need for additional shared memory or global variables.

2.4 Synchronization Mechanisms

2.4.1 Producer-Consumer Pattern

The implementation follows a classic producer-consumer pattern with several advanced synchronization techniques:

- **Double-condition synchronization**: Uses two separate condition variables (`not_full` and `not_empty`) rather than a single condition, enabling more precise thread wakeups.

- **Predicate-based waiting**: All waits are in while loops checking complex conditions (`while (buf->count >= buf->capacity && !terminate_requested)`), preventing spurious wakeups and making the code resilient to race conditions.
- **Signal vs. Broadcast**: Uses targeted `pthread_cond_signal()` for normal operations, waking only a single thread, and `pthread_cond_broadcast()` for termination, ensuring all threads are notified.
- **Multiple termination paths**: The system can terminate through normal EOF, SIGINT signal, or error conditions, all converging to the same cleanup procedure.
- **Lock ordering**: Consistent lock acquisition order prevents deadlocks.

A key optimization is the minimization of critical sections. The lock is only held while modifying the buffer structure itself, not during search operations, maximizing parallel execution.

```
// In buffer_push (producer)
while (buf->count >= buf->capacity && !terminate_requested) {
    pthread_cond_wait(&buf->not_full, &buf->mutex);
}

pthread_cond_signal(&buf->not_empty);

// In buffer_pop (consumer)
while (buf->count == 0 && !buf->eof && !terminate_requested)
   {
    pthread_cond_wait(&buf->not_empty, &buf->mutex);
}

pthread_cond_signal(&buf->not_full);
```

Listing 3: Condition Variables for Wait/Signal

2.4.2 Barrier-Based Final Synchronization

The barrier synchronization is used specifically for the final reporting phase:

- **Centralized initialization**: The barrier is initialized in the main thread with exactly the number of worker threads, preventing misalignment issues.
- **Thread-local data collection**: Each thread stores its match count and timing data locally, eliminating the need for atomic operations or locks during the search phase.

- **Memory layout optimization**: Thread argument structures are allocated contiguously, allowing thread 0 to access all thread results using pointer arithmetic after the barrier.
- **Role delegation**: Only thread 0 generates the report, avoiding redundant work and potential output races.

This approach ensures all worker threads have completed processing and printed their individual match counts before the summary report is generated, while minimizing the use of shared memory.

```
// In main function
pthread_barrier_t barrier;
pthread_barrier_init(&barrier, NULL, config.num_workers);

// In worker thread
printf("Worker %d found %d matches\n", args->thread_id,
    match_count);
pthread_barrier_wait(args->barrier);

// Thread 0 generates the report after the barrier
if (args->thread_id == 0) {
    // Generate summary
}
```

Listing 4: Barrier Synchronization

2.5 Signal Handling

Signal handling is implemented with a focus on graceful termination in interrupt scenarios. Key aspects include:

- **Volatile global flag**: Uses a `volatile sig_atomic_t terminate_requested` flag for thread-safe, atomic signal notification.
- **Graceful shutdown**: The handler doesn't immediately exit but signals all threads to terminate at safe points.
- **Safe signal handling**: Uses `sigaction()` rather than `signal()` for more reliable, portable behavior with proper signal mask handling.
- **Minimal work in handler**: Performs only the essential operations in the signal handler itself (setting flag and waking threads) to avoid unsafe functions.

- **Global buffer reference**: Maintains a `g_buffer` global pointer to ensure the signal handler can access the buffer even from outside the main thread context.
- **Resource cleanup**: Ensures all dynamically allocated memory is freed even during abrupt termination.

The most critical aspect is waking all potentially blocked threads by broadcasting to both condition variables. Without this step, threads waiting on condition variables would never be notified of termination.

```
void handle_sigint(int sig) {
    (void)sig;
    terminate_requested = 1;

    printf("\nSIGINT received (Signal %d), shutting down...\n
    ", sig);

    if (g_buffer) {
        pthread_mutex_lock(&g_buffer->mutex);
        g_buffer->eof = true;
        pthread_cond_broadcast(&g_buffer->not_empty);
        pthread_cond_broadcast(&g_buffer->not_full);
        pthread_mutex_unlock(&g_buffer->mutex);
    }
}
```

Listing 5: Signal Handling Implementation

2.6 Search Algorithm

The program implements the Boyer-Moore string search algorithm, which offers significant performance advantages:

- **Sublinear time complexity**: Achieves O(n/m) in the best case (where n is text length and m is pattern length), significantly faster than the O(n*m) of naive approaches.
- **Skipping capability**: Can skip portions of the text that cannot possibly match, becoming more efficient for longer search patterns.
- **Right-to-left scanning**: Processes the pattern from right to left, which increases the chance of finding a mismatch early.
- **Bad character rule**: Uses a precomputed lookup table to determine how far to skip ahead when a mismatch occurs.

The algorithm consists of two key components:

1. **Preprocessing phase**: Builds a 256-entry lookup table indexed by ASCII character values, storing the rightmost occurrence of each character in the pattern.
2. **Search phase**: Uses the table to determine optimal skip distances when a character mismatch is found.

For log analysis, this algorithm is particularly well-suited because:

- Search terms are typically words or phrases (multiple characters)
- The algorithm's performance advantage grows with pattern length
- Log files often contain many non-matching lines that can be quickly skipped

The implementation includes optimizations like using the `inline` directive for the `max()` function and type-casting characters to `unsigned char` to handle the full ASCII range correctly.

2.7 Chunk-Based File Reading

The file reading implementation uses a hybrid chunk-based approach that balances I/O efficiency with memory usage:

- **System call reduction**: Using 4KB chunks drastically reduces the number of system calls compared to line-by-line reading, especially beneficial for large files with thousands or millions of lines.
- **Zero-copy optimization**: The implementation preserves pointers into the read buffer when possible, avoiding unnecessary data copying within chunks.
- **Chunked processing algorithm**: The algorithm processes each chunk in three phases:
    1. Process all complete lines within the current chunk
    2. Store any partial line at the end of the chunk for continuation
    3. Read the next chunk and combine with any stored partial line

- **Boundary handling**: Special care is taken for partial lines at chunk boundaries, ensuring that lines are processed correctly regardless of where chunk boundaries fall.
- **Memory management strategy**: The implementation follows a clear allocation policy:
    1. Allocate a read buffer of fixed size (4KB) on the stack
    2. Dynamically allocate the line buffer only when needed
    3. Grow the line buffer in larger increments to reduce reallocation frequency
    4. Make individual copies of completed lines for thread safety
- **Error resilience**: Comprehensive error checking at each I/O and memory allocation operation ensures the system can recover or exit gracefully from resource constraints.

This approach provides near-optimal I/O performance while maintaining manageable memory usage even for very large files with long lines. The specific chunk size (4KB) was chosen to align with typical filesystem block sizes, further improving read efficiency.

```
void manager_function(Bounded_TS_Linked_List* buffer,
    Program_config* config) {
    // Time tracking setup
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // Open file
    int fd = open(config->log_file, O_RDONLY);

    // Read in 4KB chunks
    char read_buffer[READ_BUFFER_SIZE + 1];
    ssize_t bytes_read;
    char* line_buffer = NULL;
    size_t line_buffer_size = 0;
    size_t line_length = 0;
    int lines_processed = 0;

    while (!terminate_requested &&
           (bytes_read = read(fd, read_buffer,
    READ_BUFFER_SIZE)) > 0) {
        // Process chunk and find complete lines
        // ...
    }

```

```
    // Signal EOF to worker threads
    pthread_mutex_lock(&buffer->mutex);
    buffer->eof = true;
    pthread_cond_broadcast(&buffer->not_empty);
    pthread_mutex_unlock(&buffer->mutex);
}
```

Listing 6: Manager Function with Chunk-Based Reading

3 Program Outputs

3.1 Test Case 1: Memory Leak Check

`valgrind ./LogAnalyzer 10 4 logs/sample.log "ERROR"`

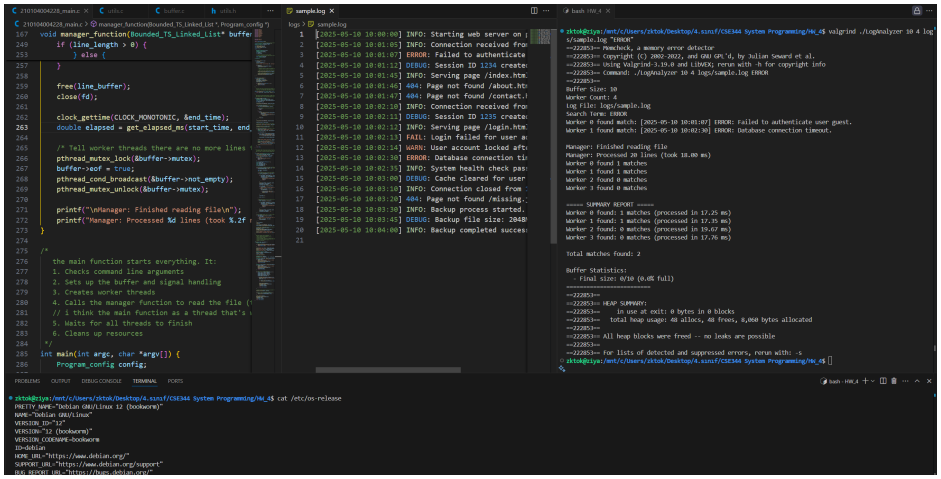

Figure 1: Valgrind output showing no memory leaks

3.2 Test Case 2: Small Buffer with Few Workers

`./LogAnalyzer 5 2 logs/debug.log "FAIL"`

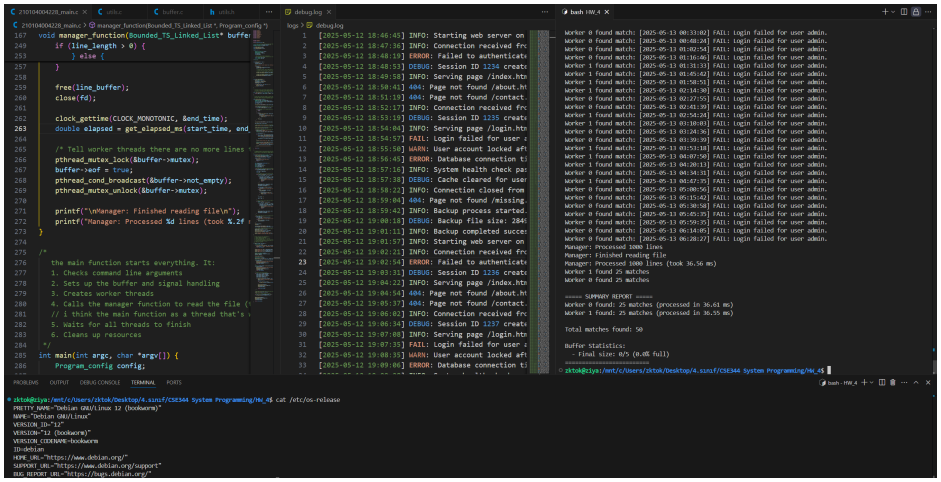

Figure 2: Program output with small buffer and few workers

3.3 Test Case 3: Large Buffer with Many Workers

`./LogAnalyzer 50 8 logs/large.log "404"`

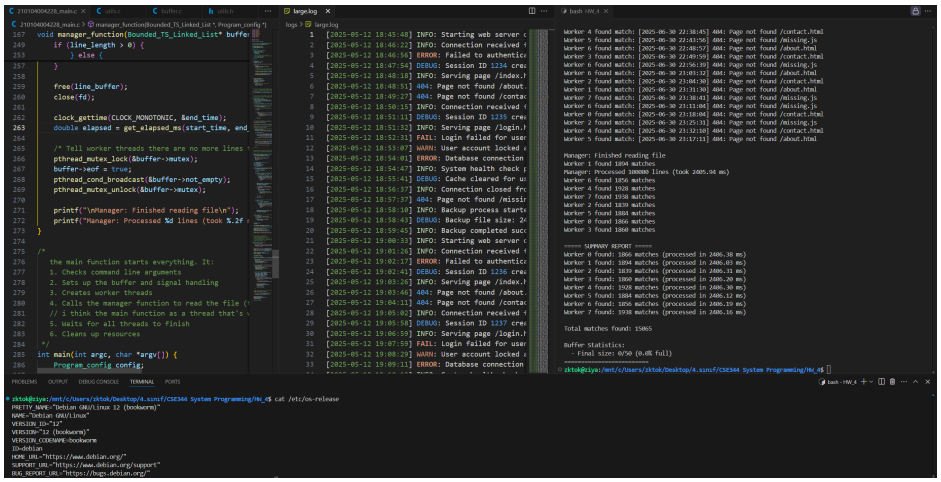

Figure 3: Program output with large buffer and many workers

4 Conclusion

The implemented Multithreaded Log File Analyzer successfully meets all the requirements of the assignment. The program demonstrates effective use of POSIX threading mechanisms, proper synchronization techniques, and efficient algorithms for both file I/O and string searching.

The design follows a clear producer-consumer pattern with a manager thread feeding lines to worker threads through a bounded buffer. This approach effectively parallelizes the processing of large log files, leveraging multiple CPU cores for improved performance.

The implementation shows robust error handling, proper resource management, and graceful termination capabilities. The use of advanced features like the Boyer-Moore search algorithm and chunk-based file reading demonstrates an understanding of performance optimization principles beyond the basic requirements.

Overall, this project provided valuable hands-on experience with system programming concepts, particularly in the areas of multithreading, synchronization, and efficient I/O handling.