GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

CSE344 Systems Programming
Spring 2025
Homework 4 Report

Osmancan Bozali
220104004011

1. Introduction:

This report details the implementation of a Multithreaded Log File Analyzer in C using POSIX threads (pthreads). The program is designed to efficiently search large log files for a user-specified keyword by distributing the workload across multiple worker threads. It employs a producer-consumer pattern where a single manager thread (the main thread) reads log file lines and places them into a shared, bounded buffer. Multiple worker threads concurrently consume lines from this buffer, search for the keyword, and count occurrences. Synchronization between threads is achieved using POSIX mutexes, condition variables, and a barrier to ensure data integrity and coordinated reporting. The program accepts command-line arguments for buffer size, number of worker threads, log file path, and the search term. It also includes error handling, graceful shutdown upon receiving a SIGINT signal (Ctrl+C), and ensures proper memory management.

2. Code Explanation:

The implementation is divided into 220104004011_main.c (core logic, manager thread, worker thread function), buffer.c (shared buffer implementation), and buffer.h (buffer interface).

2.1 buffer.h

Defines the Buffer structure, which encapsulates all necessary components for the shared, bounded, thread-safe circular buffer. (circular buffer is used for this implementation.)

- **data:** A dynamically allocated array of character pointers (char **) to store the log lines.
- **size:** The maximum capacity of the buffer.
- **head & tail:** Indices for managing the circular queue logic.
- **count:** The current number of items in the buffer.
- **mutex:** A *pthread_mutex_t* to protect access to the buffer's shared data (data, head, tail, count, terminating).
- **space_available:** A *pthread_cond_t* signaled by consumers when space becomes available.
- **items_available:** A *pthread_cond_t* signaled by the producer when items are added.
- **terminating:** An integer flag used to signal workers that no more items will be produced, especially during SIGINT handling.
- Declares the public functions for buffer operations**: buffer_init**, **buffer_destroy**, **buffer_push**, and **buffer_pop**.

2.2 buffer.c

- **fatal(const char *msg)**: A helper function to print error messages using perror and exit gracefully upon critical errors (e.g., memory allocation failure, mutex/condition variable initialization failure).
- **buffer_init(Buffer *buf, size_t size)**:
  - Allocates memory for the buffer's data array using *calloc*.
  - Initializes buffer metadata (size, head, tail, count, terminating).
  - Initializes the mutex and both condition variables using *pthread_mutex_init* and *pthread_cond_init*, checking for errors.
- **buffer_destroy(Buffer *buf)**:
  - Frees the memory allocated for the data array.
  - Destroys the mutex and condition variables using *pthread_mutex_destroy* and *pthread_cond_destroy* to release system resources.
- **buffer_push(Buffer *buf, char *item)** (Producer Logic):
  - Acquires the buffer mutex (*pthread_mutex_lock*).
  - Enters a while loop condition which checks if the buffer is full and if termination hasn't been signaled. If true, it waits on the *space_available* condition variable (*pthread_cond_wait*), atomically releasing the mutex while waiting and re-acquiring it upon wakeup.
  - After potentially waiting, it checks the terminating flag again. If set it unlocks the mutex and returns immediately without adding the item.
  - If there's space and not terminating, it adds the item to the buffer at the tail index.
  - Updates tail and increments count.
  - Signals the *items_available* condition variable (*pthread_cond_signal*) to wake up one potentially waiting worker thread.
  - Releases the buffer mutex (*pthread_mutex_unlock*).
- **buffer_pop(Buffer *buf)**:
  - Acquires the buffer mutex.
  - Enters a while loop condition that checks if the buffer is empty and termination hasn't been signaled. If true, it waits on the *items_available* condition variable, releasing the mutex while waiting.
  - After potentially waiting, it checks if the buffer is still empty and termination has been signaled If both are true, it means the producer finished (or was interrupted by a signal) and there are no more items. It unlocks the mutex and returns NULL to signal the worker to exit. This correctly handles the case where SIGINT occurs while workers are waiting on an empty buffer.
  - If there's an item available, it retrieves the item from the head index.
  - Updates head and decrements count.
  - Signals the *space_available* condition variable to wake up one potentially waiting manager(main) thread.
  - Releases the buffer mutex.
  - Returns the retrieved item.

2.3 220104004011_main.c

- **Includes and Globals:** Includes necessary headers. Declares global variables:
  - **worker_data:** Pointer to an array holding *WorkerData* structs for each thread.
  - **num_workers:** Stores the number of worker threads specified by the user.
  - **finish_barrier:** A *pthread_barrier_t* used to synchronize workers before the final summary report.
  - **terminate_flag:** A *volatile sig_atomic_t* flag set by the SIGINT handler.
- **WorkerData Struct:** Holds thread-specific data passed to each worker: a pointer to the shared Buffer, the search term, a counter for matches found by that thread (*match_count),* and the thread's ID (id) for reporting thread specific result.
- **sigint_handler(int sig)**: Signal handler for SIGINT. It sets the *terminate_flag* to 1.
- **worker(void *arg)** (Worker Thread Function):
  - Enters a loop that continues as long as *buffer_pop* returns a non-NULL line.
  - Calls *buffer_pop* to retrieve a line from the shared buffer.
  - If *buffer_pop* returns NULL, it signifies the end of input (either normal EOF or signaled termination), so the loop breaks.
  - Uses *strstr* to check if the retrieved line contains the search term. Increments a local counter (local) if a match is found.
  - After the loop, stores the local count into the thread's *WorkerData* struct (*wd->match_count*).
  - Prints a message indicating the thread ID and the number of matches it found.
  - Calls *pthread_barrier_wait(&finish_barrier).* All worker threads will block here until num_workers threads have reached this point.
  - pthread_barrier_wait returns *PTHREAD_BARRIER_SERIAL_THREAD* for exactly one arbitrarily chosen thread. This thread proceeds to calculate the total match count by summing the match_count from all worker_data entries and prints the final summary report.
- **usage(const char *prog)**: Prints the correct command-line usage instructions to stderr.
- **xmalloc(size_t size)**: A wrapper for malloc that checks the return value and calls perror and exit on failure.
- **main(int argc, char *argv[]):**
  - **Setup:** Parses and validates command-line arguments. Installs a signal handler for SIGINT. Initializes the shared Buffer, allocates memory for worker data and thread IDs, and initializes the *pthread_barrier*.
  - **Thread Creation:** Creates *num_workers* worker threads, passing each a pointer to its *WorkerData* struct containing the buffer reference and search term.
  - **File Processing:** Opens the log file. Reads the file line-by-line using getline. In a loop:
    - Checks the *terminate_flag,* breaks loop if set.
    - Duplicates the read line.
    - Pushes the duplicated line onto the shared buffer using *buffer_push*.
  - **Termination Signaling:** After the loop (due to EOF or SIGINT):
    - Frees the getline buffer and closes the file.
    - **If SIGINT:** Sets the terminating flag within the buffer structure and broadcasts on both buffer condition variables (*items_available* and *space_available*) to wake up all waiting threads promptly.

        - **If Normal EOF:** Pushes *num_workers* NULL pointers onto the buffer as sentinels, signaling each worker to exit.
    - **Join & Cleanup:** Waits for all worker threads to complete using *pthread_join*. Destroys the barrier and the buffer, and frees all dynamically allocated memory.

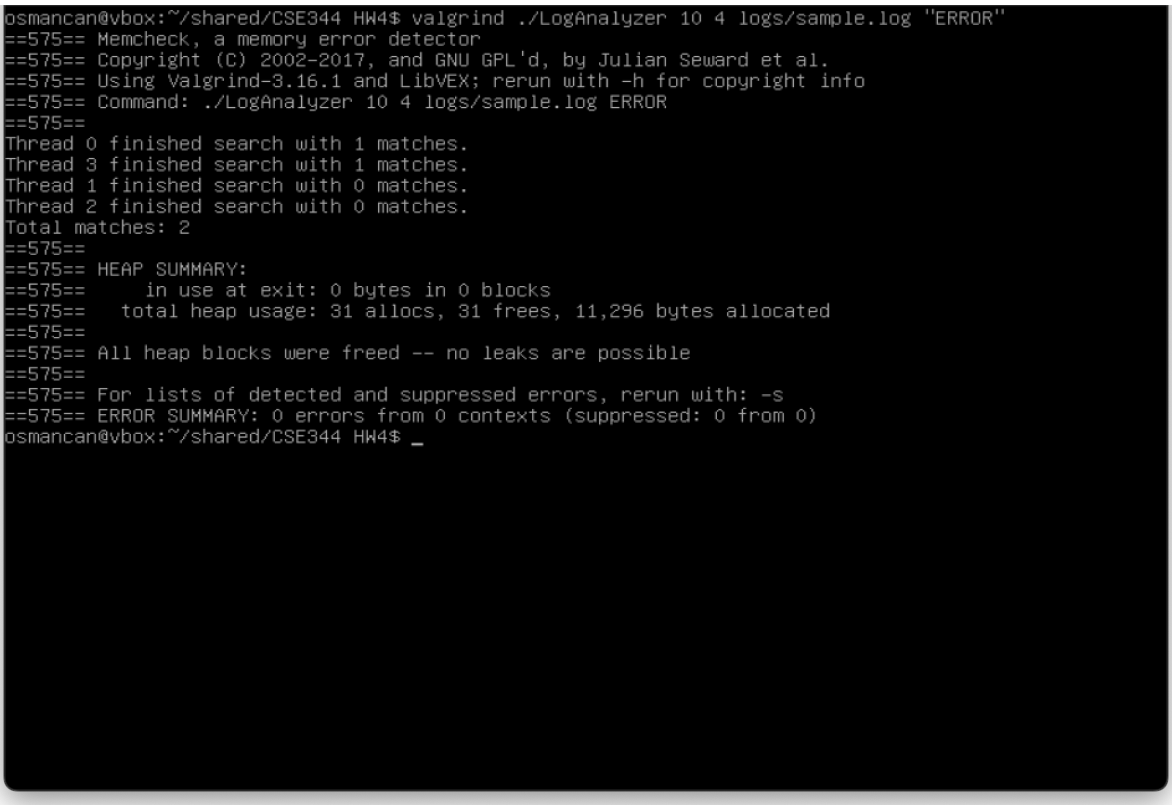
3. Screenshots:

Scenario 1: valgrind ./LogAnalyzer 10 4 logs/sample.log "ERROR"

Expected Results:

- 4 worker threads should be created.
- Total matches should be 2. (see logs/sample.log in source code for double checking.)
- Valgrind should show no memory leaks.

Actual Results: Matching with the expected result!

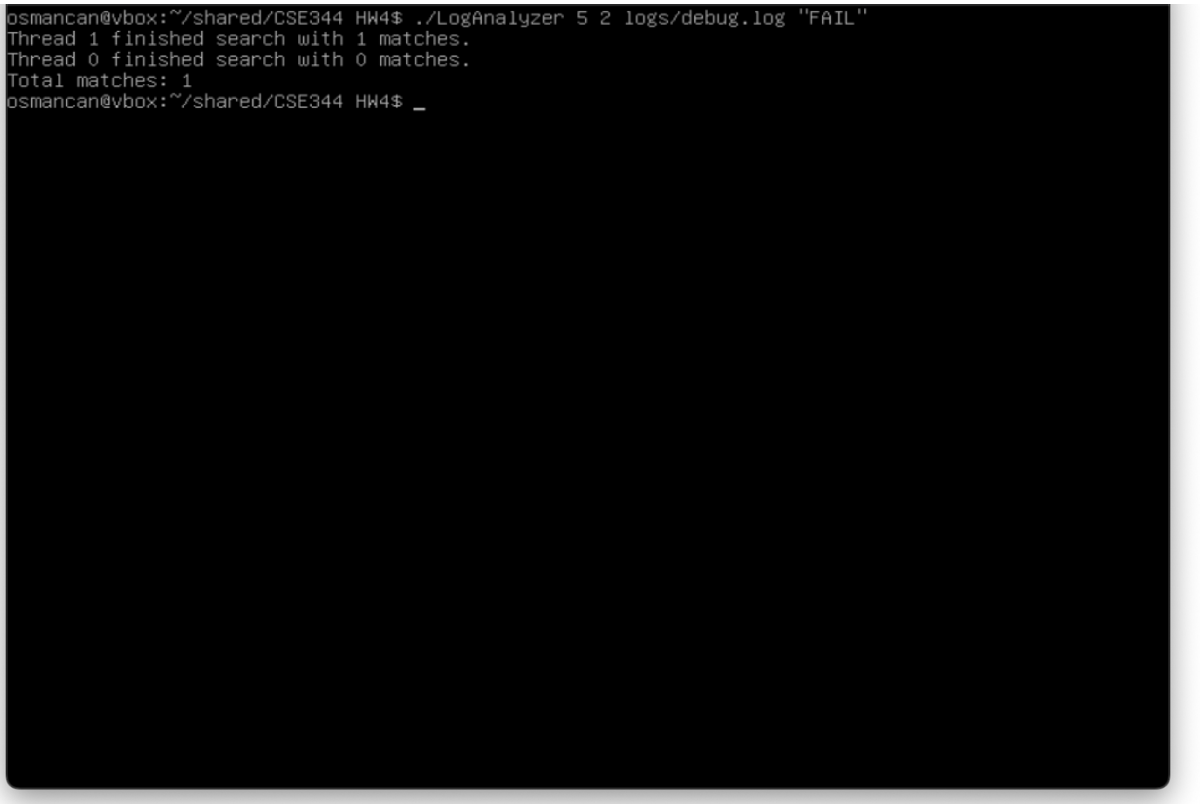


Scenario 2: ./LogAnalyzer 5 2 logs/debug.log "FAIL"

Expected Results:

- 2 worker threads should be created.
- Total matches should be 1. (see logs/debug.log in source code for double checking.)

Actual Results: Matching with the expected result!



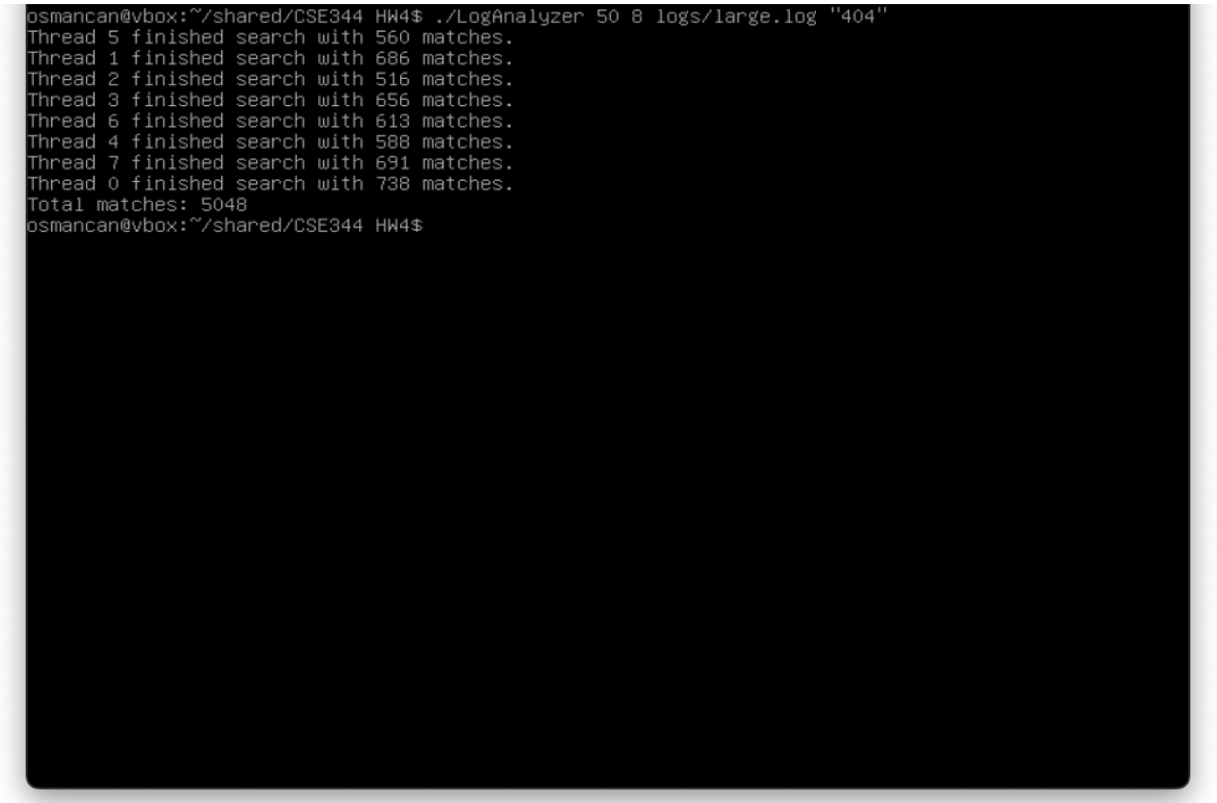
Scenario 3: ./LogAnalyzer 50 8 logs/large.log "404"

This log file have more than 50000 lines (see the source code for double checking.)

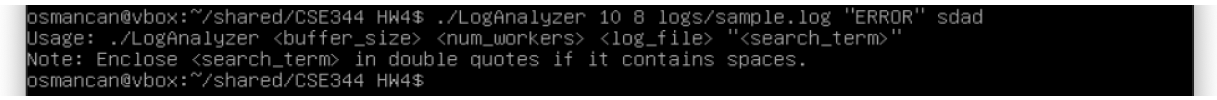
Expected Results:

- 8 worker threads should be created.
- Total matches should be 5048. (again see logs/large.log in source code for double checking.)

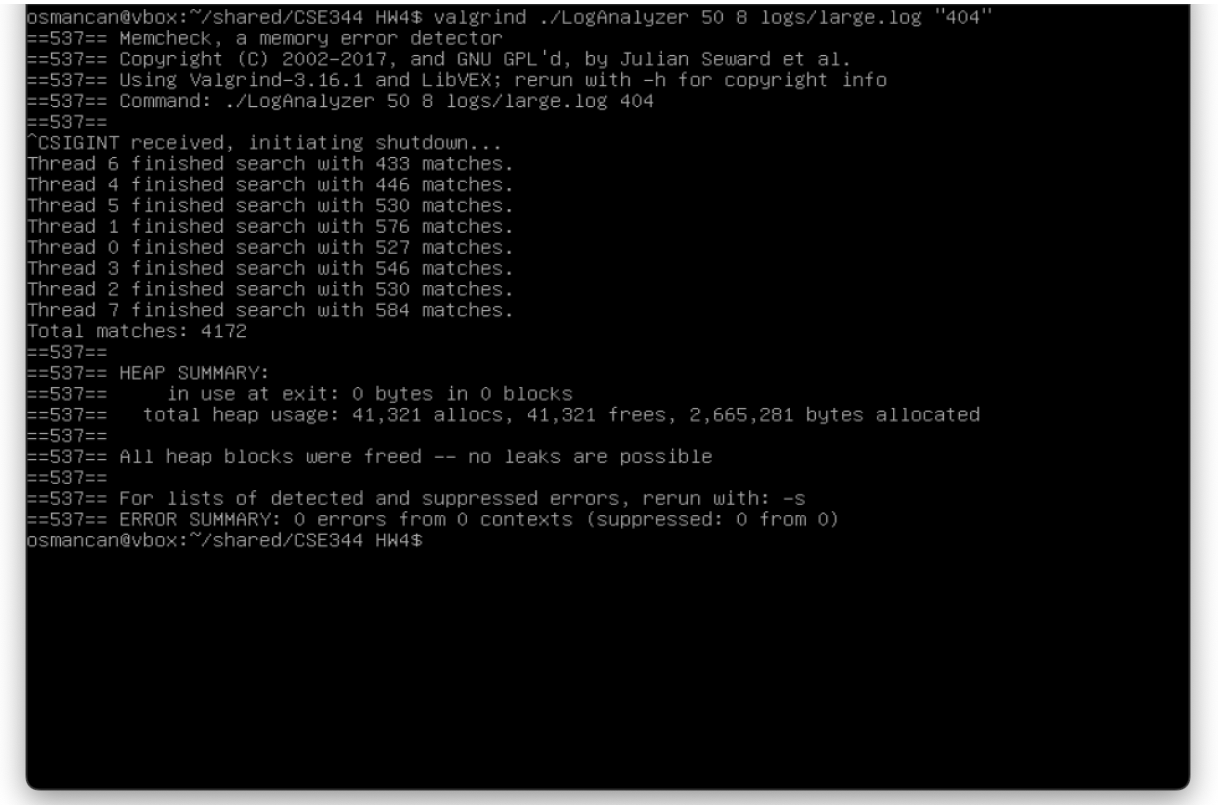
Actual Results: Matching with the expected result



Scenario 4: Additional Usage Show



Scenario 5: SIGINT Termination with Memory Leak Check



Conclusion:

This project successfully implements a program requested as in the homework. The main thread reads lines from the file and worker threads check these lines for a specific word. To connect the producer and consumers, I implemented a shared buffer (like a temporary waiting area for the lines) making sure it was thread-safe by doing it circular. This buffer design was important and required careful use of special tools: mutexes (locks to prevent threads from using the buffer at the same time) and condition variables (to make the producer wait if the buffer was full, and consumers wait if it was empty). A key challenge was ensuring the program stopped correctly, both at the end of the file (solved with NULL signals as stated in PDF) and with Ctrl+C (solved using a flag and waking up waiting threads). Handling memory correctly was another focus, checked using Valgrind. In conclusion, the project successfully used threads for log analysis, with help of a well-designed buffer and synchronization methods using mutexes and conditional variables.