Onurhan TALAN

210104004065

CSE 344

System Programming

Homework #4

A. Introduction

This program is a multi-threaded text search utility that searches for a specific term within a log file.

The program takes four command-line arguments:

1. Buffer size
2. Number of worker thread
3. Log file path
4. Search term to look for

The “**main**” thread opens the **log file**, reads one line at a time, and **inserts** the line that is just read to the buffer. If the buffer is **full** it **waits** for the buffer to have an empty slot to insert the line.

The “**worker**” threads are created by the “**main**” thread. A “**worker”** thread removes a line from the buffer and looks for if the **target search term** is **present** in the line or not. It **keeps track of** the **lines** that includes the target search term and **number of the lines** the target search term is found. If the search term is present in the line, it stores the line and increments the counter. If the buffer is **empty**, it waits until a valid slot is present in the buffer.

A **mutex** is used to prevent threads to access the buffer at the same time. This prevents **race conditions**.

Conditional variables “**not_empty**” and “**not_full**” are used to block the threads until the buffer has a valid slot to remove or insert. Conditional variables are used instead of **busy waiting** since busy waiting would cause **CPU hogging**.

B. Main Program Flow

The program is divided into 11 parts. Below, the piece of code corresponding to a part and its explanation is given:

1. Initial Setup and Signal Handling

```
setup_signal_handler();
```

- Sets up signal handling for graceful shutdown (Ctrl+C)
- Registers the SIGINT handler to set should_exit flag

2. Command Line Argument Processing

```
Params params = parse_args(argc, argv);

print_params(params);
```

- Validates and parses command-line arguments
- Checks for correct number of arguments
- Validates buffer size and worker count

- Verifies log file existence
- Prints program parameters for user confirmation

3. File Opening

```
int file_fd = open(params.log_file, O_RDONLY);
```

- Opens the log file in read-only mode
- Exits if file cannot be opened

4. Buffer Initialization

```
Buffer buffer;
init_buffer(&buffer, params.buffer_size);
```

- Creates circular buffer with specified capacity
- Initializes synchronization primitives (mutex, condition variables)
- Sets up buffer structure for thread-safe operations.

5. Barrier Initialization

```
pthread_barrier_t barrier;
pthread_barrier_init(&barrier, NULL, params.num_workers + 1);
```

- Creates barrier for thread synchronization
- Number of threads = workers + main thread
- Ensures all threads complete their work before proceeding

6. Worker Thread Creation

```
pthread_t* workers = NULL;
WorkerParams* worker_params = NULL;
create_workers(&workers, &worker_params, params.num_workers, &buffer,
params.search_term, &barrier);
```

- Allocates memory for worker threads and parameters
- Initializes worker parameters with:
- Unique worker IDs
- Buffer pointer

- Search term
- Barrier pointer
- Match tracking structures
- Creates worker threads that begin processing

7. Main Thread Processing Loop

```
while(!should_exit && (line = read_line(file_fd)) != NULL)
        insert_line(&buffer, line);
```

- Reads log file line by line
- Inserts lines into buffer
- Continues until:
- End of file is reached
- SIGINT is received (Ctrl+C)
- Error occurs

8. Signal End of Processing

```
insert_line(&buffer, NULL);
```

- Inserts NULL to signal workers to stop
- Workers will exit when they receive NULL

9. Thread Synchronization and Cleanup

```
pthread_barrier_wait(&barrier);
for(unsigned int i = 0; i < params.num_workers; i++)
        pthread_join(workers[i], NULL);
```

- Waits for all workers to complete
- Joins worker threads
- Ensures all processing is finished

10. Results Reporting

```
print_report(worker_params, params.num_workers);
```

- Displays search results

- Shows matches per worker
- Lists matching lines

11. Resource Cleanup

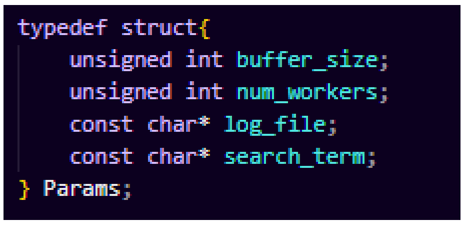
cleanup(file_fd, &buffer, workers, worker_params, params.num_workers, &barrier);

- Closes file descriptor
- Destroys buffer and its resources
- Frees worker threads and parameters
- Destroys barrier
- Ensures no memory leaks

C. Function explanations:

Structs

1.Params

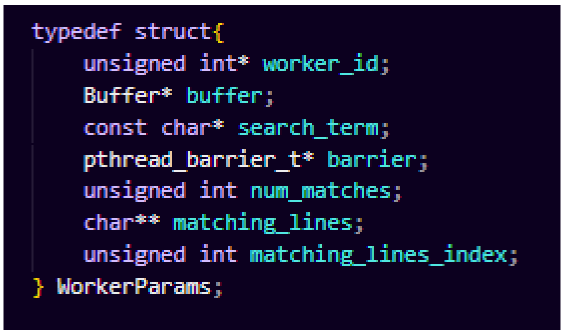


```
typedef struct{
    unsigned int buffer_size;
    unsigned int num_workers;
    const char* log_file;
    const char* search_term;
} Params;
```

This struct holds the program's configuration parameters:

- **buffer_size**: Determines how many lines can be stored in the buffer at once
- **num_workers**: Specifies how many parallel worker threads to create
- **log_file**: Path to the file that will be searched
- **search_term**: The text pattern to search for in the log file

2. WorkerParams Struct

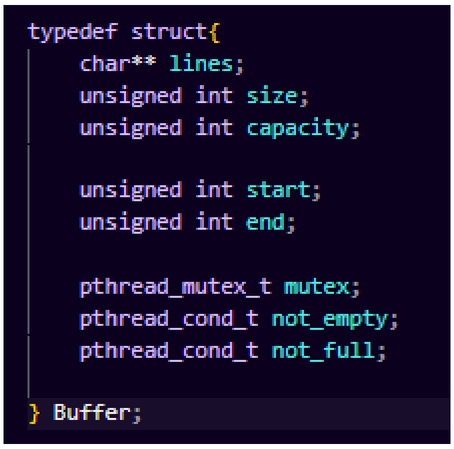


```c
typedef struct{
    unsigned int* worker_id;
    Buffer* buffer;
    const char* search_term;
    pthread_barrier_t* barrier;
    unsigned int num_matches;
    char** matching_lines;
    unsigned int matching_lines_index;
} WorkerParams;
```

This struct contains parameters for each worker thread:

- **worker_id**: Unique identifier for the worker thread
- **buffer**: Pointer to the shared circular buffer
- **search_term**: The term to search for in lines
- **barrier**: Used for thread synchronization
- **num_matches**: Tracks how many matches this worker found
- **matching_lines**: Stores the actual matching lines
- **matching_lines_index**: Tracks position in matching_lines array

3. Buffer Struct



```c
typedef struct{
    char** lines;
    unsigned int size;
    unsigned int capacity;

    unsigned int start;
    unsigned int end;

    pthread_mutex_t mutex;
    pthread_cond_t not_empty;
    pthread_cond_t not_full;

} Buffer;
```

This struct implements a thread-safe circular buffer:

- **lines**: Array of pointers to stored lines
- **size**: Current number of lines in the buffer
- **capacity**: Maximum number of lines the buffer can hold
- **start**: Index of the first line in the buffer

- **end**: Index where the next line will be inserted
- **mutex**: Protects buffer access from multiple threads
- **not_empty**: Signals when buffer is no longer empty
- **not_full**: Signals when buffer is no longer full

The circular buffer implementation allows:

- Efficient line storage and retrieval
- Thread-safe operations
- Producer-consumer pattern between main thread and workers
- Proper synchronization using mutex and condition variables

These structs work together to:

1. Configure the program (Params)
2. Manage worker threads (WorkerParams)
3. Handle thread-safe line storage and retrieval (Buffer)

1.Signal Handling Functions

void sigint_handler(int signum)

- Handles Ctrl+C (SIGINT) signal
- Sets global should_exit flag to 1
- Prints cleanup message

void setup_signal_handler(void)

- Configures SIGINT signal handling
- Sets up sigaction structure
- Registers sigint_handler for SIGINT signals

2.Command Line Processing

Params parse_args(int argc, char *argv[])

- Validates command-line arguments
- Converts buffer size and worker count to integers
- Checks if log file exists
- Returns structured parameters

void print_params(Params params)

- Displays program parameters

- Shows buffer size, worker count, log file, and search term

unsigned int string_to_int(const char *str)

- Converts string to unsigned integer
- Validates input is a valid number
- Exits on invalid input

3.File Operations

char* read_line(int file_fd)

- Reads one line from file descriptor
- Allocates buffer for line
- Reads character by character until newline
- Returns NULL on error or EOF

4.Thread Management

int create_workers(pthread_t workers, WorkerParams** worker_params, unsigned in t num_workers, Buffer* buffer, const char* search_term, pthread_barrier_t* barrier)**

- Creates worker threads
- Initializes worker parameters
- Allocates memory for thread structures
- Returns 0 on success, -1 on failure

void* worker_thread(void* arg)

- Main worker thread function
- Processes lines from buffer
- Searches for term in each line
- Tracks matches and matching lines
- Handles graceful shutdown

5.Cleanup and Reporting

void cleanup(int file_fd, Buffer* buffer, pthread_t* workers, WorkerParams* worker_pa rams, unsigned int num_workers, pthread_barrier_t* barrier)

- Closes file descriptor
- Destroys buffer
- Frees worker threads

- Frees worker parameters
- Destroys barrier

void print_report(WorkerParams* worker_params, unsigned int num_workers)

- Displays search results
- Shows matches per worker
- Lists matching lines for each worker

6.Buffer Operations

int init_buffer(Buffer* buffer, unsigned int cap)

- Initializes circular buffer
- Allocates memory for lines
- Sets up synchronization primitives

void destroy_buffer(Buffer* buffer)

- Frees buffer memory
- Destroys mutex and condition variables

void insert_line(Buffer* buffer, char* line)

- Thread-safe line insertion
- Waits if buffer is full
- Signals when line is inserted

char* remove_line(Buffer* buffer)

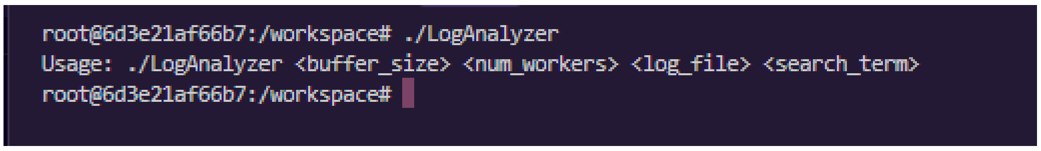
- Thread-safe line removal
- Waits if buffer is empty
- Returns next line from buffer

These functions work together to create a multi-threaded text search program that:

1. Reads a log file line by line
2. Processes lines in parallel using worker threads
3. Searches for a specified term
4. Reports matches found by each worker
5. Handles graceful shutdown and cleanup

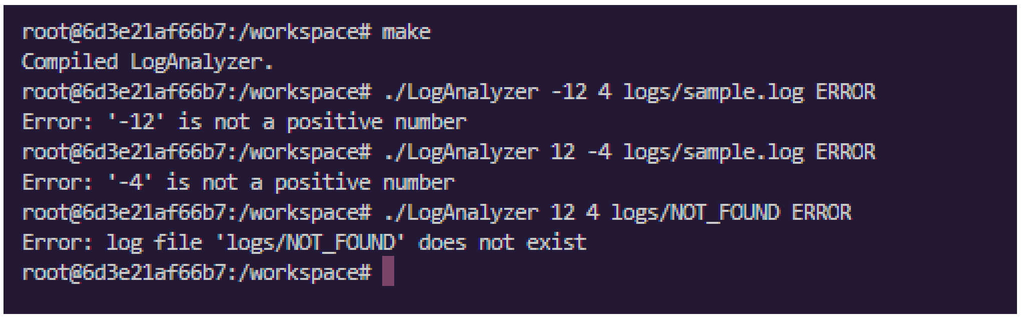
D. Screenshots

1. Usage Warning



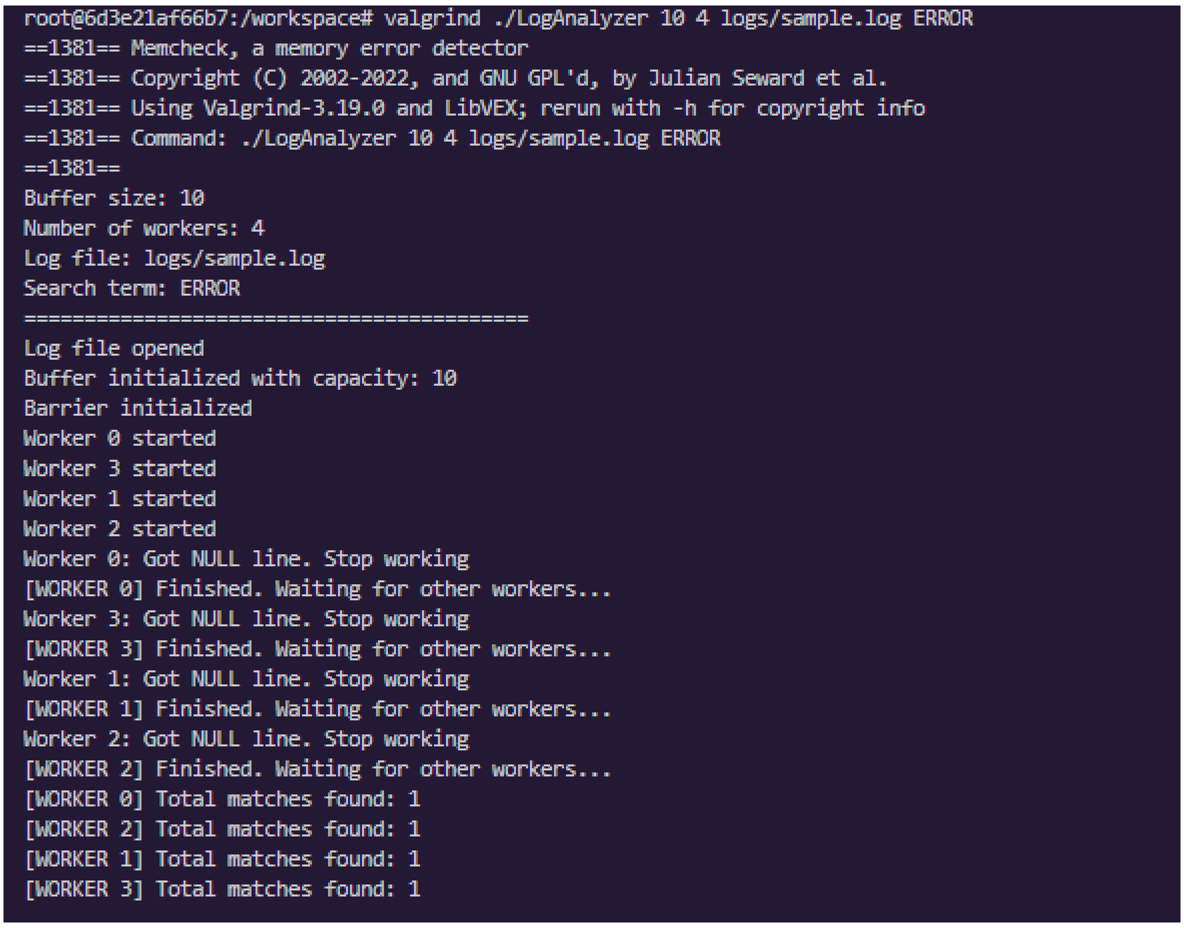
```
root@6d3e21af66b7:/workspace# ./LogAnalyzer
Usage: ./LogAnalyzer <buffer_size> <num_workers> <log_file> <search_term>
root@6d3e21af66b7:/workspace#
```

2. Invalid Command-Line Arguments

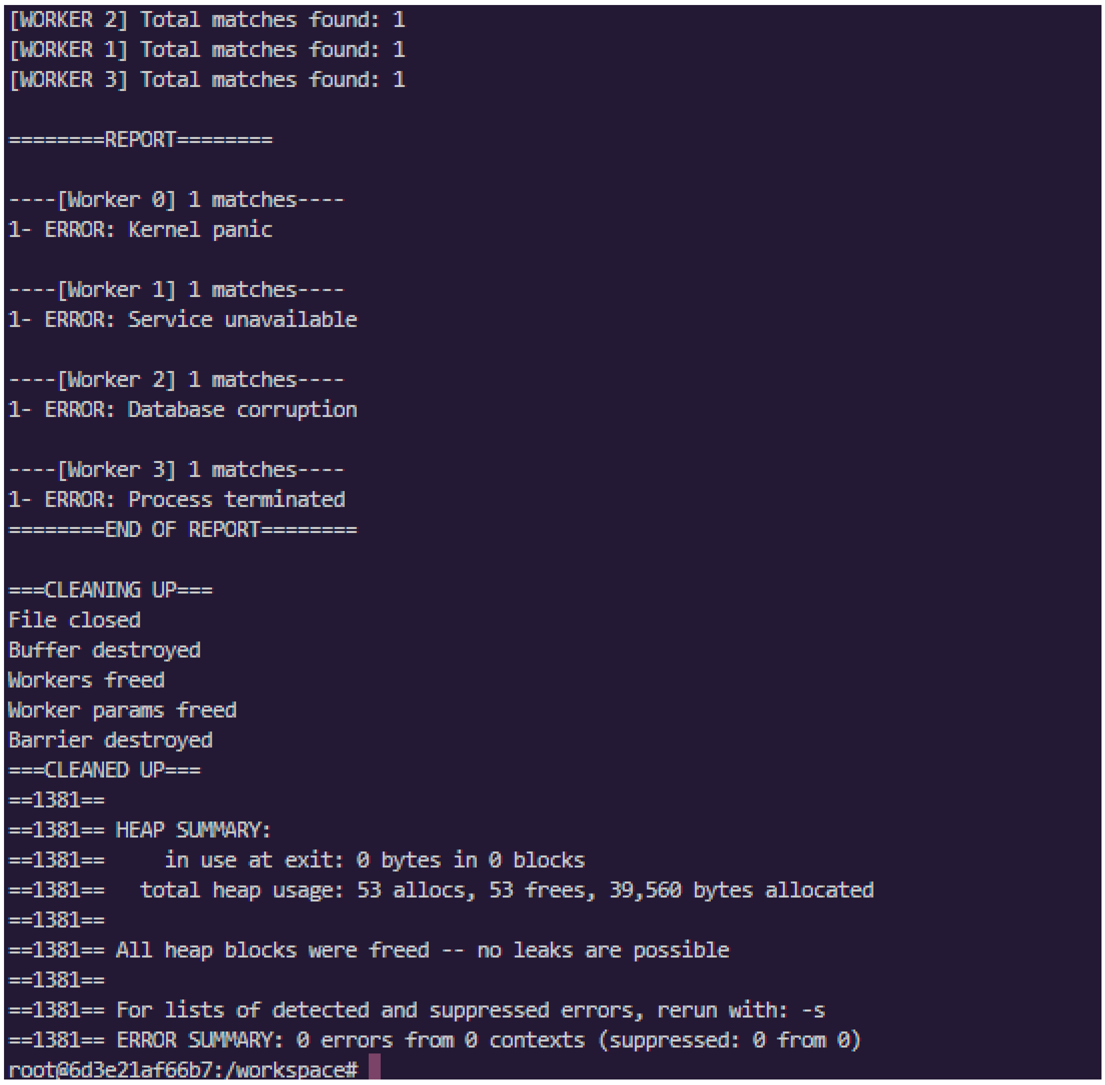


```
root@6d3e21af66b7:/workspace# make
Compiled LogAnalyzer.
root@6d3e21af66b7:/workspace# ./LogAnalyzer -12 4 logs/sample.log ERROR
Error: '-12' is not a positive number
root@6d3e21af66b7:/workspace# ./LogAnalyzer 12 -4 logs/sample.log ERROR
Error: '-4' is not a positive number
root@6d3e21af66b7:/workspace# ./LogAnalyzer 12 4 logs/NOT_FOUND ERROR
Error: log file 'logs/NOT_FOUND' does not exist
root@6d3e21af66b7:/workspace#
```

3.Valgrind Memory Leak Check

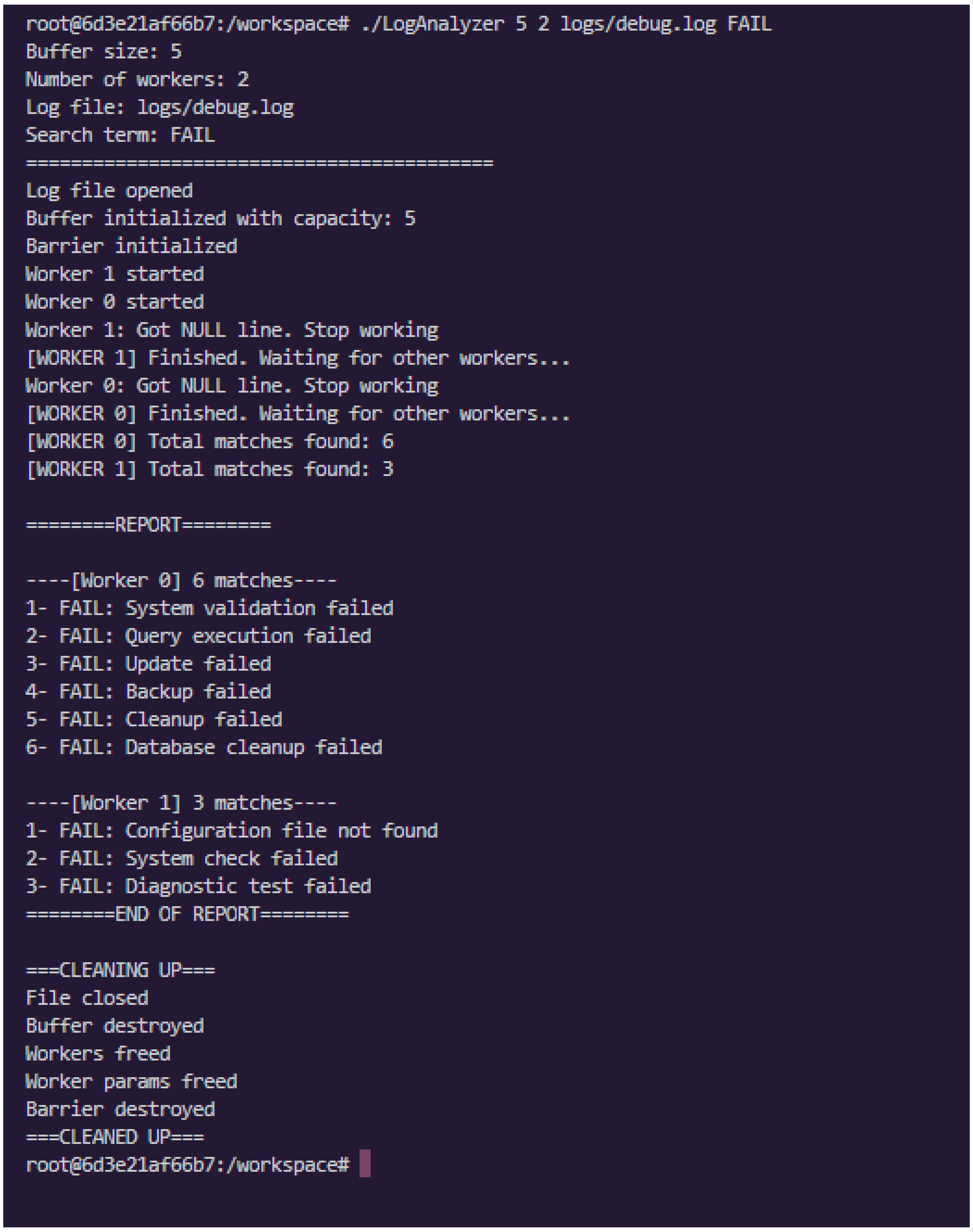


```
root@6d3e21af66b7:/workspace# valgrind ./LogAnalyzer 10 4 logs/sample.log ERROR
==1381== Memcheck, a memory error detector
==1381== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==1381== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==1381== Command: ./LogAnalyzer 10 4 logs/sample.log ERROR
==1381==
Buffer size: 10
Number of workers: 4
Log file: logs/sample.log
Search term: ERROR
==========================================
Log file opened
Buffer initialized with capacity: 10
Barrier initialized
Worker 0 started
Worker 3 started
Worker 1 started
Worker 2 started
Worker 0: Got NULL line. Stop working
[WORKER 0] Finished. Waiting for other workers...
Worker 3: Got NULL line. Stop working
[WORKER 3] Finished. Waiting for other workers...
Worker 1: Got NULL line. Stop working
[WORKER 1] Finished. Waiting for other workers...
Worker 2: Got NULL line. Stop working
[WORKER 2] Finished. Waiting for other workers...
[WORKER 0] Total matches found: 1
[WORKER 2] Total matches found: 1
[WORKER 1] Total matches found: 1
[WORKER 3] Total matches found: 1
```



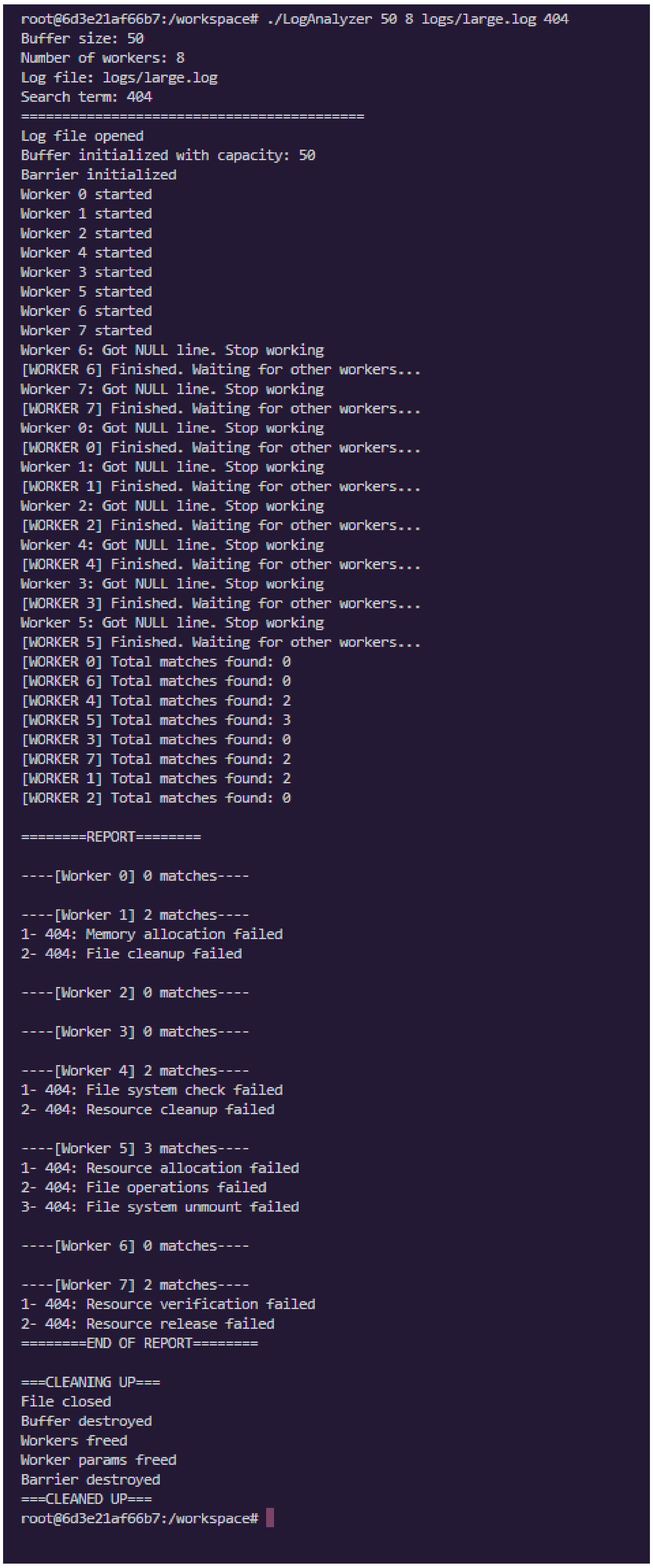
```
[WORKER 2] Total matches found: 1
[WORKER 1] Total matches found: 1
[WORKER 3] Total matches found: 1

========REPORT========

----[Worker 0] 1 matches----
1- ERROR: Kernel panic

----[Worker 1] 1 matches----
1- ERROR: Service unavailable

----[Worker 2] 1 matches----
1- ERROR: Database corruption

----[Worker 3] 1 matches----
1- ERROR: Process terminated
========END OF REPORT========

===CLEANING UP===
File closed
Buffer destroyed
Workers freed
Worker params freed
Barrier destroyed
===CLEANED UP===
==1381==
==1381== HEAP SUMMARY:
==1381==     in use at exit: 0 bytes in 0 blocks
==1381==   total heap usage: 53 allocs, 53 frees, 39,560 bytes allocated
==1381==
==1381== All heap blocks were freed -- no leaks are possible
==1381==
==1381== For lists of detected and suppressed errors, rerun with: -s
==1381== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@6d3e21af66b7:/workspace#
```

5. Test Scenario – 2



```
root@6d3e21af66b7:/workspace# ./LogAnalyzer 5 2 logs/debug.log FAIL
Buffer size: 5
Number of workers: 2
Log file: logs/debug.log
Search term: FAIL
==========================================
Log file opened
Buffer initialized with capacity: 5
Barrier initialized
Worker 1 started
Worker 0 started
Worker 1: Got NULL line. Stop working
[WORKER 1] Finished. Waiting for other workers...
Worker 0: Got NULL line. Stop working
[WORKER 0] Finished. Waiting for other workers...
[WORKER 0] Total matches found: 6
[WORKER 1] Total matches found: 3

========REPORT========

----[Worker 0] 6 matches----
1- FAIL: System validation failed
2- FAIL: Query execution failed
3- FAIL: Update failed
4- FAIL: Backup failed
5- FAIL: Cleanup failed
6- FAIL: Database cleanup failed

----[Worker 1] 3 matches----
1- FAIL: Configuration file not found
2- FAIL: System check failed
3- FAIL: Diagnostic test failed
========END OF REPORT========

===CLEANING UP===
File closed
Buffer destroyed
Workers freed
Worker params freed
Barrier destroyed
===CLEANED UP===
root@6d3e21af66b7:/workspace#
```

6- Test Scenario - 3



```
root@6d3e21af66b7:/workspace# ./LogAnalyzer 50 8 logs/large.log 404
Buffer size: 50
Number of workers: 8
Log file: logs/large.log
Search term: 404
==========================================
Log file opened
Buffer initialized with capacity: 50
Barrier initialized
Worker 0 started
Worker 1 started
Worker 2 started
Worker 4 started
Worker 3 started
Worker 5 started
Worker 6 started
Worker 7 started
Worker 6: Got NULL line. Stop working
[WORKER 6] Finished. Waiting for other workers...
Worker 7: Got NULL line. Stop working
[WORKER 7] Finished. Waiting for other workers...
Worker 0: Got NULL line. Stop working
[WORKER 0] Finished. Waiting for other workers...
Worker 1: Got NULL line. Stop working
[WORKER 1] Finished. Waiting for other workers...
Worker 2: Got NULL line. Stop working
[WORKER 2] Finished. Waiting for other workers...
Worker 4: Got NULL line. Stop working
[WORKER 4] Finished. Waiting for other workers...
Worker 3: Got NULL line. Stop working
[WORKER 3] Finished. Waiting for other workers...
Worker 5: Got NULL line. Stop working
[WORKER 5] Finished. Waiting for other workers...
[WORKER 0] Total matches found: 0
[WORKER 6] Total matches found: 0
[WORKER 4] Total matches found: 2
[WORKER 5] Total matches found: 3
[WORKER 3] Total matches found: 0
[WORKER 7] Total matches found: 2
[WORKER 1] Total matches found: 2
[WORKER 2] Total matches found: 0

========REPORT========

----[Worker 0] 0 matches----

----[Worker 1] 2 matches----
1- 404: Memory allocation failed
2- 404: File cleanup failed

----[Worker 2] 0 matches----

----[Worker 3] 0 matches----

----[Worker 4] 2 matches----
1- 404: File system check failed
2- 404: Resource cleanup failed

----[Worker 5] 3 matches----
1- 404: Resource allocation failed
2- 404: File operations failed
3- 404: File system unmount failed

----[Worker 6] 0 matches----

----[Worker 7] 2 matches----
1- 404: Resource verification failed
2- 404: Resource release failed
========END OF REPORT========

===CLEANING UP===
File closed
Buffer destroyed
Workers freed
Worker params freed
Barrier destroyed
===CLEANED UP===
root@6d3e21af66b7:/workspace#
```

E. Conclusion

During the development of this multi-threaded text search program, several significant challenges were encountered and addressed.

The primary challenge was implementing proper thread synchronization, which was solved by creating a thread-safe circular buffer system. This implementation ensured efficient coordination between multiple threads, similar to a well-structured queue system.

Memory management required careful attention to resource allocation and deallocation, ensuring proper cleanup procedures were in place.

Input validation presented its own set of challenges, particularly in handling negative numbers and invalid inputs, which was resolved through a robust validation system.

File handling was implemented with efficiency in mind, processing data line by line while maintaining proper error management.

The shutdown mechanism was carefully designed to ensure graceful program termination.

The search implementation focused on creating an efficient pattern-matching system.

These challenges and their solutions demonstrate the importance of implementing robust error handling and user-friendly interfaces in system programming.