

CSE 344

Homework #4

200104004074

Bedirhan Ömer Aksoy

1) Introduction

In my implementation, main function first checks argument count and returns exit failure if argc count is not sufficient to ensure the program called correctly. Then parses CLI arguments to set buffer size, number of workers, filename and search term. Then opens the log file descriptor that will be read. Then signal handler initialized before creating the manager and worker threads to catch Ctrl + C signal. After that, shared buffer initialized which will be shared by all the threads. Then the barrier has initialized with `pthread_barrier_init()` function for all worker threads to ensure they are all complete before the final reporting.

Then function creates the manager thread which will read the log file line by line and insert them into the shared buffer. When the end of file is readed, it push NUL once for each worker to signal termination. Then each worker thread is created in a for loop with assigning their id's, search term and match count variable of the `worker_arg_t` struct's index.

After that, manager thread and worker threads are collected with join function. Then total match count is calculated after all workers are collected and printed to the console. Then shared buffer and thread barrier is destroyed and program returns with success.

2) Code Explanation

2.0) global variables:

I have defined a struct called `worker_arg_t` which will represent the arguments of each worker thread. It has `id`, `search_term` and `match_count` variables. I have defined another struct called `buffer_t`. It has `data`, `capacity`, `count`, `head`, `tail`, `mutex`, `not_empty` and `not_full` variables. I have a `buffer_t` variable in `main` for shared buffer, a barrier variable and a `sig_atomic_t` `stop` variable to stop manager with signal handler.

```
10
11 typedef struct {
12     int id;
13     char *search_term;
14     int match_count;
15 } worker_arg_t;
16
17 buffer_t *shared_buffer;
18 int num_workers;
19 pthread_barrier_t barrier;
20 volatile sig_atomic_t stop = 0;
21
```

2.1) void sigint_handler(int sig): This function handles SIGINT (ctrl + c) signal and stops manager thread with setting stop variable 1 and wakes all waiting worker threads to unblock if any thread is waiting so they finish their jobs and exits gracefully.

```
21
22 void sigint_handler(int sig) {
23     (void)sig;
24     stop = 1;
25     buffer_signal_all(shared_buffer);
26 }
27
```

2.2) void *manager_thread(void *arg): This function reads the log file line by line and puts them into the buffer until it reaches the MAX_LINE or the stop variable becomes one (the SIGINT scenario). After putting each line into the buffer and reaching the EOF, it puts NULL into buffer number of workers time so that all workers gets the NULL element and end their jobs.

```
28 void *manager_thread(void *arg) {
29     FILE *fp = (FILE *)arg;
30     char line[MAX_LINE];
31
32     while (!stop && fgets(line, MAX_LINE, fp)) {
33         buffer_put(shared_buffer, strdup(line));
34     }
35     for (int i = 0; i < num_workers; i++)
36         buffer_put(shared_buffer, NULL); // EOF marker for each worker
37     fclose(fp);
38     return NULL;
39 }
40
```

2.3) void *worker_thread(void *arg): Worker thread function has argument input which contains id, match_count and search_term. It gets a line from buffer with buffer_get() function until it gets NULL from buffer and checks for the search term. It updates the match count if it encounters the expected term. After getting the NULL from the buffer (the EOF), it waits by a barrier and after that it prints its own finding results with a print statement and returns NULL.

```
41 void *worker_thread(void *arg) {
42     worker_arg_t *warg = (worker_arg_t *)arg;
43     char *line;
44
45     while ((line = buffer_get(shared_buffer)) != NULL) {
46         if (strstr(line, warg->search_term))
47             warg->match_count++;
48         free(line);
49     }
50
51     pthread_barrier_wait(&barrier);
52     printf("Worker %d found %d matches\n", warg->id, warg->match_count);
53     return NULL;
54 }
55
```

2.4) int main(int argc, char *argv[]): Main function checks argument counts, initializes the buffer size and number of workers, filename and search term. Then opens the log file descriptor in read mode, then initializes the sigint handler, shared buffer and barrier. After that manager thread and worker threads are created. Then it joins manager thread and joins worker threads. Total match count is calculated and printed to the console. Then buffer and barrier are destroyed and returns EXIT_SUCCESS.

```
56 int main(int argc, char *argv[]) {
57     if (argc != 5) {
58         fprintf(stderr, "Usage: ./LogAnalyzer <buffer_size> <num_workers> <log_file> <search_term>\n");
59         return EXIT_FAILURE;
60     }
61
62     int buffer_size = atoi(argv[1]);
63     num_workers = atoi(argv[2]);
64     char *filename = argv[3];
65     char *search_term = argv[4];
66
67     FILE *fp = fopen(filename, "r");
68     if (!fp) {
69         perror("Failed to open file");
70         return EXIT_FAILURE;
71     }
72
73     signal(SIGINT, sigint_handler);
74     shared_buffer = buffer_init(buffer_size);
75     pthread_barrier_init(&barrier, NULL, num_workers);
76
77     pthread_t manager;
78     pthread_create(&manager, NULL, manager_thread, fp);
79
80     pthread_t workers[num_workers];
81     worker_arg_t args[num_workers];
82     for (int i = 0; i < num_workers; i++) {
83         args[i].id = i;
84         args[i].search_term = search_term;
85         args[i].match_count = 0;
86         pthread_create(&workers[i], NULL, worker_thread, &args[i]);
87     }
88
89     pthread_join(manager, NULL);
90     for (int i = 0; i < num_workers; i++)
91         pthread_join(workers[i], NULL);
92
93     int total = 0;
94     for (int i = 0; i < num_workers; i++)
95         total += args[i].match_count;
96
97     printf("Total matches: %d\n", total);
98
99     buffer_destroy(shared_buffer);
100    pthread_barrier_destroy(&barrier);
101    return EXIT_SUCCESS;
102 }
```

2.5) `buffer_t *buffer_init(int size)`: This function initializes the buffer variable with allocating the amount of size entered by user in arguments, initializes capacity, head, count and tail variables. Then it initializes the mutex and condition variables.

```
16
17 buffer_t *buffer_init(int size) {
18     buffer_t *buf = malloc(sizeof(buffer_t));
19     buf->data = malloc(sizeof(char *) * size);
20     buf->capacity = size;
21     buf->count = 0;
22     buf->head = 0;
23     buf->tail = 0;
24     pthread_mutex_init(&buf->mutex, NULL);
25     pthread_cond_init(&buf->not_empty, NULL);
26     pthread_cond_init(&buf->not_full, NULL);
27     return buf;
28 }
29
```

2.6) `buffer_t *buffer_put(buffer_t *buf, char *line)`: This function gets a line input and writes this line into the buffer with locking the mutex before writing to buffer. If buffer reached its capacity, it waits for the not_full condition variable with `pthread_cond_wait()` function, so we do not encounter any deadlocks since we temporarily unlock the mutex variable with this function before sleeping on the not_full condition.

If buffer is not full, we write the next file line to the end of the buffer (tail), increase the count variable of the buffer and signals the not empty condition variable. Then unlocks the mutex at the end.

```
29
30 void buffer_put(buffer_t *buf, char *line) {
31     pthread_mutex_lock(&buf->mutex);
32     while (buf->count == buf->capacity)
33         pthread_cond_wait(&buf->not_full, &buf->mutex);
34
35     buf->data[buf->tail] = line;
36     buf->tail = (buf->tail + 1) % buf->capacity;
37     buf->count++;
38
39     pthread_cond_signal(&buf->not_empty);
40     pthread_mutex_unlock(&buf->mutex);
41 }
42
```

2.7) `buffer_t *buffer_get(buffer_t *buf)`: Function first locks the mutex of the buffer. If buffer is empty, function waits for the `not_empty` cond variable. If it's not empty, then it gets the buffer data from the head position (which is written to buffer first), then increases the head pointer with 1 and decreases the count variable with 1. Then signals the `not_full` condition variable and unlocks the mutex and returns the line.

```
43 char *buffer_get(buffer_t *buf) {  
44     pthread_mutex_lock(&buf->mutex);  
45     while (buf->count == 0)  
46         pthread_cond_wait(&buf->not_empty, &buf->mutex);  
47  
48     char *line = buf->data[buf->head];  
49     buf->head = (buf->head + 1) % buf->capacity;  
50     buf->count--;  
51  
52     pthread_cond_signal(&buf->not_full);  
53     pthread_mutex_unlock(&buf->mutex);  
54     return line;  
55 }
```

2.8) `buffer_t *buffer_signal_all(buffer_t *buf)`: This function is used for terminating program gracefully if SIGINT signal occurs and being sure that any threads are not blocked. First function locks the mutex variable first, then signals `not_empty` and `not_full` condition variables with `pthread_cond_broadcast()` function to unblock all threads which could be blocked. Then unlocks the mutex.

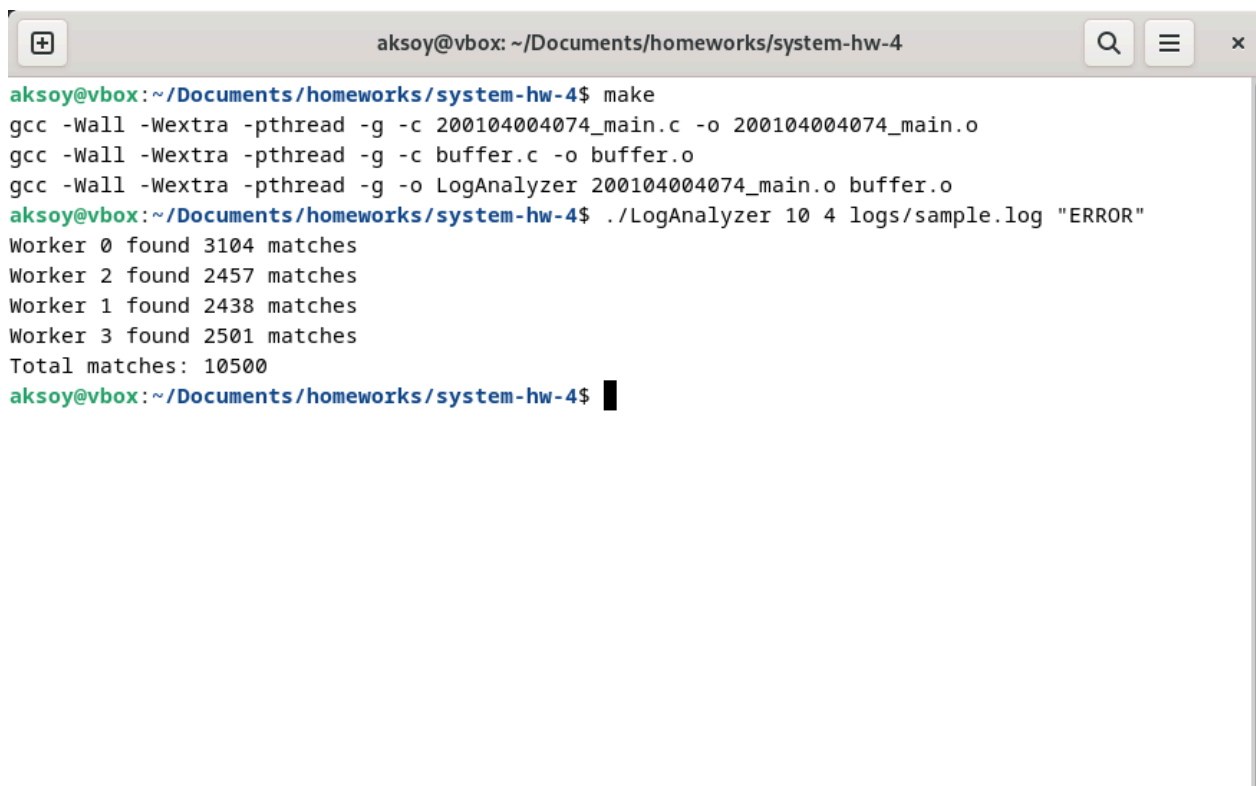
```
56  
57 void buffer_signal_all(buffer_t *buf) {  
58     pthread_mutex_lock(&buf->mutex);  
59     pthread_cond_broadcast(&buf->not_empty);  
60     pthread_cond_broadcast(&buf->not_full);  
61     pthread_mutex_unlock(&buf->mutex);  
62 }  
63
```

2.9) void buffer_destroy(buffer_t *buf): This function is used for destroying the buffer struct object with destroying the buffer's mutex, condition variables and free the allocated data of it and finally the buffer's itself.

```
63
64 void buffer_destroy(buffer_t *buf) {
65     pthread_mutex_destroy(&buf->mutex);
66     pthread_cond_destroy(&buf->not_empty);
67     pthread_cond_destroy(&buf->not_full);
68     free(buf->data);
69     free(buf);
70 }
```

3) Screenshots

Testing run in my Debian 12 virtual machine:



```
aksoy@vbox: ~/Documents/homeworks/system-hw-4
aksoy@vbox:~/Documents/homeworks/system-hw-4$ make
gcc -Wall -Wextra -pthread -g -c 200104004074_main.c -o 200104004074_main.o
gcc -Wall -Wextra -pthread -g -c buffer.c -o buffer.o
gcc -Wall -Wextra -pthread -g -o LogAnalyzer 200104004074_main.o buffer.o
aksoy@vbox:~/Documents/homeworks/system-hw-4$ ./LogAnalyzer 10 4 logs/sample.log "ERROR"
Worker 0 found 3104 matches
Worker 2 found 2457 matches
Worker 1 found 2438 matches
Worker 3 found 2501 matches
Total matches: 10500
aksoy@vbox:~/Documents/homeworks/system-hw-4$
```


Testing the program's memory checking with using Valgrind:

```
aksoy@vbox: ~/Documents/homeworks/system-hw-4
aksoy@vbox:~/Documents/homeworks/system-hw-4$ make
gcc -Wall -Wextra -pthread -g -c 200104004074_main.c -o 200104004074_main.o
gcc -Wall -Wextra -pthread -g -c buffer.c -o buffer.o
gcc -Wall -Wextra -pthread -g -o LogAnalyzer 200104004074_main.o buffer.o
aksoy@vbox:~/Documents/homeworks/system-hw-4$ valgrind --leak-check=full ./LogAnalyzer 10 4 logs/sample.log
==3529== Memcheck, a memory error detector
==3529== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3529== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==3529== Command: ./LogAnalyzer 10 4 logs/sample.log
==3529==
Usage: ./LogAnalyzer <buffer_size> <num_workers> <log_file> <search_term>
==3529==
==3529== HEAP SUMMARY:
==3529==   in use at exit: 0 bytes in 0 blocks
==3529==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3529==
==3529== All heap blocks were freed -- no leaks are possible
==3529==
==3529== For lists of detected and suppressed errors, rerun with: -s
==3529== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
aksoy@vbox:~/Documents/homeworks/system-hw-4$
```

Testing the SIGINT signal handler:

```
aksoy@vbox: ~/Documents/homeworks/system-hw-4
aksoy@vbox:~/Documents/homeworks/system-hw-4$ make
gcc -Wall -Wextra -pthread -g -c 200104004074_main.c -o 200104004074_main.o
gcc -Wall -Wextra -pthread -g -c buffer.c -o buffer.o
gcc -Wall -Wextra -pthread -g -o LogAnalyzer 200104004074_main.o buffer.o
aksoy@vbox:~/Documents/homeworks/system-hw-4$ ./LogAnalyzer 10 2 logs/sample.log "ERROR"
Worker 0 found 2993 matches
Worker 1 found 7507 matches
Total matches: 10500
aksoy@vbox:~/Documents/homeworks/system-hw-4$ ./LogAnalyzer 10 2 logs/sample.log "ERROR"
^CWorker 0 found 2376 matches
Worker 1 found 2989 matches
Total matches: 5365
aksoy@vbox:~/Documents/homeworks/system-hw-4$
```

4) Conclusion

Key challenges in the project were synchronizing access to the shared circular buffer with using `pthread_mutex_t` and the condition variables to prevent race conditions and ensure blocking behavior. I have solved this problems with proper wait/signal logic in `buffer_put()` and `buffer_get()` functions. Handling clean shutdown on SIGINT was solved with implementing `buffer_signal_all()` to wake all blocked threads. Coordinating worker thread completion with using `pthread_barrier_t` required careful use of EOF markers and barrier waits. Finally, I ensured memory safety by freeing dynamically allocated lines in workers and validating with `valgrind` which I showed in the screenshots part.

Bedirhan Ömer Aksoy

200104004074