

CS140 Project 2

Parallelized grep Runner

Justin Jose R. Ruaya

Submitted to:
WILSON M. TAN
JUAN FELIPE CORONEL
ANGELA A. ZABALA

Department of Computer Science
University of the Philippines - Diliman

Contents

1	References	1
2	Multithreaded	2
3	Multiprocess	8

List of Figures

2.1	Runthrough of multithreaded	2
-----	---------------------------------------	---

List of Codes

2.1 Struct node_t and queue_t	3
2.2 Queue logic	3
2.3 main() code and struct work_args	4
2.4 threaded Work()	5
2.5 accessDir() function	5
2.6 Global variables necessary for (c)	7

**In partial fulfillment
of the requirements
in
CS 140: Operating Systems**

Part 1

References

1. Here are the references. Take note that the links are clickable:

- <https://stackoverflow.com/questions/6417158/c-how-to-free-nodes-in-the-linked-list>
This reference is for freeing a linked list.
- <https://stackoverflow.com/questions/4553012/checking-if-a-file-is-a-directory-or-just>
This one is for checking if the output of `readdir()` is a file (true) or a directory (false).
- Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). Operating Systems: Three Easy Pieces (1.00). Arpaci-Dusseau Books.
This code for a thread-safe queue came from here. This was also cited in CS 140 2223A - Lecture 16.

For the test case, we will use this directory (with the `rootpath` being `testdir_docu`:

```
cs140@DESKTOP-5163LQN:~/cs140221project2-j-ruaya/testdir_docu$ tree
```

```
.
| DIR1
|   DIR6
|   DIR2
|   absent3.txt
|   DIR3
|   DIR4
|   |   absent4.txt
|   |   present.txt
|   DIR5
|   absent5.txt
|   absent.txt
|   absent2.txt
```

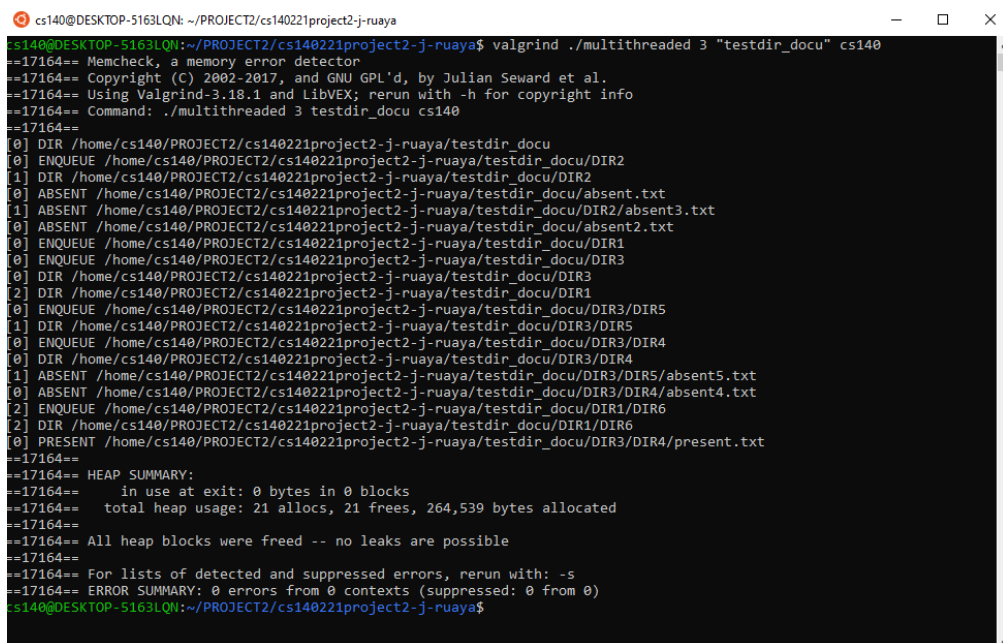
```
6 directories, 6 files
```

Part 2

Multithreaded

- 2 (a) Walkthrough of code execution with sample run having at least $N=2$ on a multicore machine, one *PRESENT*, five *ABSENT*s and six *DIR*s.

For our test, we will use $N=3$. The output is given below. Take note that Valgrind may take a while to execute this (whereas it is almost instant without it), due to the threads working as a **spinlock** (its specifics on the next sections) that checks if the counter (again, see the next section) is zero.



```
cs140@DESKTOP-5163LQN: ~/PROJECT2/cs140221project2-j-ruaya
cs140@DESKTOP-5163LQN:~/PROJECT2/cs140221project2-j-ruaya$ valgrind ./multithreaded 3 "testdir_docu" cs140
==17164== Memcheck, a memory error detector
==17164== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==17164== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==17164== Command: ./multithreaded 3 testdir_docu cs140
==17164==
[0] DIR /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu
[0] ENQUEUE /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR2
[1] DIR /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR2
[0] ABSENT /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/absent.txt
[1] ABSENT /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR2/absent3.txt
[0] ABSENT /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/absent2.txt
[0] ENQUEUE /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR1
[0] ENQUEUE /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3
[0] DIR /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3
[2] DIR /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR1
[0] ENQUEUE /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3/DIR5
[1] DIR /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3/DIR5
[0] ENQUEUE /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3/DIR4
[0] DIR /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3/DIR4
[1] ABSENT /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3/DIR5/absent5.txt
[0] ABSENT /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3/DIR4/absent4.txt
[2] ENQUEUE /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR1/DIR6
[2] DIR /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR1/DIR6
[0] PRESENT /home/cs140/PROJECT2/cs140221project2-j-ruaya/testdir_docu/DIR3/DIR4/present.txt
==17164==
==17164== HEAP SUMMARY:
==17164==   in use at exit: 0 bytes in 0 blocks
==17164==   total heap usage: 21 allocs, 21 frees, 264,539 bytes allocated
==17164==
==17164== All heap blocks were freed -- no leaks are possible
==17164==
==17164== For lists of detected and suppressed errors, rerun with: -s
==17164== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cs140@DESKTOP-5163LQN:~/PROJECT2/cs140221project2-j-ruaya$
```

Figure 2.1: Runthroug of multithreaded

In here, it starts accessing the rootpath `testdir_docu`. Then, it enqueues the folders inside of it (see tree on last page). If the object is a file, then it performs `grep "<search_word>" "<path>" > /dev/null`. If the return value is 0, then it prints *PRESENT*. Otherwise, it prints *ABSENT*. In our case, `present.txt` contains the string `cs140`, so it prints *PRESENT* `<abs path>/present.txt`. Likewise for the `absentX.txt`. Take note that each worker can work on a single directory at the same time. In our case, `[0]` means that that is the directory worker 0 is working on, where it also enqueues subdirectories it can find for the other threads to get. This means that it has to exhaust every file in its directory before moving on to a different *DIR*.

We know we have traversed through the entire thing if we see 7 *DIR* (1 from rootpath and 6

subdirectories), 6 ENQUEUEs, 5 ABSENT (absent1-5) and one present (present.txt)

The explanation of how all of this magic is on the next section.

- (b) *Explanation of how the task queue was implemented using your synchronization/IPC construct of choice; include how race conditions were handled*

Let's start with constructs. The task queue uses a single **mutex**. The mutex is for the head and tail lock (from the lectures), as well as for the counter (to be discussed later).

Code Block 2.1: Struct `node_t` and `queue_t`

```
1 // Logic on Queue [CS 140 2223A - Lecture 16]
2 typedef struct __node_t {
3     char file[PATH_MAX];
4     struct __node_t *next;
5 } node_t;
6 typedef struct __queue_t {
7     node_t *head;
8     node_t *tail;
9     pthread_mutex_t queue_lock; // Merged lock into one
10 } queue_t;
```

This is just similar to the one from OSTEP (or lec 16), but instead each node contains `file[PATH_MAX]` rather than `int`. `PATH_MAX` is the maximum number of bytes for a linux path (`limits.h`), which is 4096 (assuming that this program was ran on linux). Eitherway, it's longer than the required 250 characters. Not only that, we merged the head and tail lock into a single lock because head-tail collisions cause a data race despite getting the correct output.

Now we define the queue operations.

Code Block 2.2: Queue logic

```
1 // CS 140 2223A - Lecture 16
2 void Queue_Init(queue_t *q) {
3     node_t *tmp = malloc(sizeof(node_t));
4     tmp->next = NULL;
5     q->head = q->tail = tmp;
6     pthread_mutex_init(&q->queue_lock, NULL);
7 }
8
9 // CS 140 2223A - Lecture 16
10 void Queue_Enqueue(queue_t *q, char file[]) {
11     node_t *tmp = malloc(sizeof(node_t));
12     assert(tmp != NULL);
13     strcpy(tmp->file, file);
14     tmp->next = NULL;
15     pthread_mutex_lock(&q->queue_lock); // One lock instead of tail lock
16     q->tail->next = tmp;
17     q->tail = tmp;
18     pthread_mutex_unlock(&q->queue_lock); // One lock instead of tail lock
19     pthread_mutex_lock(&m_counter); // Counter incrementation
20     counter++;
21     pthread_mutex_unlock(&m_counter);
22 }
23
24 // CS 140 2223A - Lecture 16
25 int Queue_Dequeue(queue_t *q, char file[]) {
26     pthread_mutex_lock(&q->queue_lock); // One lock instead of head lock
27     node_t *tmp = q->head;
```



```
28     node_t *new_head = tmp->next;
29     if(new_head == NULL) {
30         pthread_mutex_unlock(&q->queue_lock); // One lock instead of head lock
31         return -1;
32     }
33     strcpy(file, new_head->file);
34     q->head = new_head;
35     pthread_mutex_unlock(&q->queue_lock); // One lock instead of head lock
36     free(tmp);
37     return 0;
38 }
39
40 // https://stackoverflow.com/questions/6417158/c-how-to-free-nodes-in-the-linked-list
41 // To be executed ONLY on the main tread, after the threads are waiting.
42 void Queue_Free(queue_t *q) {
43     node_t *tmp = q->head;
44     while(q->head != q->tail) {
45         tmp = q->head;
46         q->head = q->head->next;
47         free(tmp);
48     }
49     free(q->tail);
50     free(q);
51 }
```

Init, Enqueue and dequeue is **the same** as the one from the lectures. The race conditions are handled by locking **the head/tail** using the same **mutex**. This is different than in OSTEP because collisions of head and tail causes a **data race**! This was picked up by a thread sanitizer despite outputting the correct output. Another difference was the use of **char file[]** rather than the integer, and the usage of the counter. Enqueueing increments the counter (protected by lock to avoid race conditions). For queue_free, we use the logic on how to free a linked list from memory. Since this will only be executed in the main thread, **once** all threads have exited, we no longer need to lock anything.

The queue works by Enqueueing the argument path, as shown on the driver code below.

Code Block 2.3: main() code and struct work_args

```
1  struct work_args {
2      queue_t *q;
3      char word[64];
4      int worker_id;
5  } work_args;
6
7  struct work_args args[8];
8
9  int main(int argc, char *argv[]) {
10     N = atoi(argv[1]);
11     queue_t *QUEUE = malloc(sizeof(queue_t)); //Allocates queue to memory
12     Queue_Init(QUEUE); // Initializes queue
13
14     char *path = realpath(argv[2], NULL); // Converts relative rootpath (if ever) to absolute path
15
16     Queue_Enqueue(QUEUE, path); // Enqueues the rootpath
17     //Create threads and execute code
18     for(int x = 0; x < N; x++) {
19         args[x].q = QUEUE; // copy arguments
20         strcpy(args[x].word, argv[3]);
```

```
21     args[x].worker_id = x;
22     pthread_create(&tid[x], NULL, work, (void *)&args[x]); //create thread
23 }
24 for(int x = 0; x < N; x++) {
25     pthread_join(tid[x], NULL); //waits for threads to exit
26 }
27 free(path); // frees
28 Queue_Free(Queue);
29 return 0;
30 }
```

It initializes the QUEUE and the other components. `argv[2]` is the path. Since we're working on threads where we put the function pointer as argument and only one args, we have to create a `struct work_args` that contains the queue, the string to be searched and the worker id. Then, it creates the threads `tid[x]` where `x` is from `0` to `N-1`, with the necessary pointers as arguments. Lastly (for this part), it frees the QUEUE before returning `0`.

Below is the work function.

Code Block 2.4: threaded Work()

```
1 void *work(void *values) {
2     struct work_args *args = values;
3     while(1) {
4         pthread_mutex_lock(&m_counter);
5         if(counter == 0) {
6             pthread_mutex_unlock(&m_counter);
7             break;
8         }
9         pthread_mutex_unlock(&m_counter);
10        char file[PATH_MAX];
11
12        if(Queue_Dequeue(args->q, file) == -1) {
13            continue;
14        }
15        accessDir(file, args->word, args->q, args->worker_id);
16    }
17 }
```

This gets executed by a single worker. This is an infinite loop. For each iteration, it dequeues the next file path (after passing through the counter checker; to be discussed in the next section), then it calls `accessDir(file, args->word, args->q, args->worker_id)` to process it. The race conditions for the termination will be discussed on the next part. Take note that `Queue_Dequeue(args->q, file)` returns `-1` (see code block 2.2 lines 30-33) if there are no elements to be dequeued.

Code Block 2.5: accessDir() function

```
1 void accessDir(char path[], char word[], queue_t *q, int worker_id) {
2     struct dirent *child; //can be a directory or just a file
3     char filename[PATH_MAX];
4     DIR *dir = opendir(path);
5
6     printf("[%d] DIR %s\n", worker_id, path);
7
8     while ((child = readdir(dir)) != NULL) {
9         if(!strcmp(child->d_name, ".") || !strcmp(child->d_name, "..")) continue;
10    }
```

```
10     strcpy(filename, path);
11     strcat(filename, slash);
12     strcat(filename, child->d_name);
13     if(isFile(filename)) {
14         if(getGrepMatch(filename, word) == 0) {
15             printf("[%d] PRESENT %s\n", worker_id, filename);
16         } else {
17             printf("[%d] ABSENT %s\n", worker_id, filename);
18         }
19     } else {
20         printf("[%d] ENQUEUE %s\n", worker_id, filename);
21         Queue_Enqueue(q, filename);
22     }
23 }
24 pthread_mutex_lock(&m_counter);
25 counter--;
26 pthread_mutex_unlock(&m_counter);
27 closedir(dir);
28 }
```

This processes everything in the directory. It starts with printing the necessary DIR which was opened from path. The while loop essentially iterates through every folder/file on the directory accessed, ignoring "." and "..". filename is the path containing the file/folder. If it is a file, then we perform the grep match which prints PRESENT or ABSENT, depending if the return value is 0 or elsewhere. Otherwise, we print ENQUEUE and enqueue the path of the folder. Lastly, the counter gets decremented and closes the directory.

For the **race condition**, it was handled on the queue and dequeue itself! No extra magic for accessdir apart from the counter which is to be explained on the next part.

- (c) *Explanation of how each worker knows when to terminate (i.e., mechanism that determines and synchronizes that no more content will be enqueued); include how race conditions were handled*
In here, we used a single **mutex**! Recall the global variables.

Code Block 2.6: Global variables necessary for (c)

```
1 pthread_t tid[8];
2 // ***** THREADING SHENANIGANS *****
3 int counter = 0;
4 pthread_mutex_t m_counter = PTHREAD_MUTEX_INITIALIZER;
```

To Summarize: tid are just the threads. The concept is that whenever we enqueue a file (starting from `rootpath`), we increment `counter`. If there are more directories to be enqueued, we increment the `counter` by the number of folders that the `accessDir()` found. **As long as there are files to be accessed or not yet traversed on the tree, the counter will never be zero.** Getting the opposite (i.e. `counter == 0`), this means that in **Code block 2.4** (work), that is our **terminating condition** (there won't be race conditions since it waits until `counter == 0`). We unlock the mutex of the counter and break it, gracefully exiting the thread. This follows for the other threads and the program exits after freeing.

Additionally, the increment happens in the `void Queue_Enqueue(queue_t *q, char file[])` call (lines 20-22 of code block 2.2). The decrement happens in the `void accessDir(char path[], char word[], queue_t *q, int worker_id)` call, where it is at the end of the execution signaling the other threads that a folder has been processed. Take note that this also takes into account the new directories enqueued by the loop (line 21 of code block 2.5), which means that the threads will not terminate **until exactly one worker is working, the queue is empty, and there are no new directories to be enqueued.**

Part 3

Multiprocess

- 2 (a) *Walkthrough of code execution with sample run having at least $N=2$ on a multicore machine, one **PRESENT**, five **ABSENTs** and six **DIRs**.*
- (b) *Explanation of how the task queue was implemented using your synchronization/IPC construct of choice; include how race conditions were handled*
- (c) *Explanation of how each worker knows when to terminate (i.e., mechanism that determines and synchronizes that no more content will be enqueued); include how race conditions were handled*