Angelica Julianne A. Raborar
2020-10057

CS 140 Project 2: Parallelized `grep` Runner (Multithreaded version)

Contents

Documentation video link:
https://drive.google.com/file/d/1EM24j3yfuz00LZSEIYxEUreMARSMjQHR/view?usp=sharing

Repository link: https://github.com/UPD-CS140/cs140221project2-a-raborar

I. References

- *Linked list implementation of a queue*, from the book *Data Structures* by Evangel Quiwa.

  Used as a guide for the task queue implementation

- *List files in a directory (recursively too!)*. CodeVault. (2021, January 4). Retrieved from https://code-vault.net/lesson/18ec1942c2da46840693efe9b526daa4

  Consulted as a reference on the usage of `opendir()` and `readdir()`

- Linux manual pages
  - realpath(3) - Consulted as a reference on the usage of `realpath()`
  - system(3) - Consulted as a reference on the usage of `system()`
  - readdir(3) - Consulted as reference on the usage of `readdir().` Specifically, how it detects end of filestream
  - grep(1) - Consulted as reference on return values/exit status of `grep`

- Operating Systems: Three Easy Pieces (OSTEP), chapters 27 and 29

  Used as a guide for locks and threadsafe queue implementations

- https://randomwordgenerator.com/ to create files in `testdir`
- https://tree.nathanfriend.io/ to generate folder structure diagram for illustration purposes

II. Implementation

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

Test 1. Given the following directory tree:

```
TEST1
├── folder1
│   ├── note.txt
│   └── note2.txt
├── folder2
│   ├── folder 2.1
│   │   └── alphabet.txt
│   └── f2note.txt
├── folder3
│   └── folder 3.1
├── a.txt
└── b.txt
```

Compiled with `gcc -pthread -o multithreaded multithreaded.c` and executed using `./multithreaded 8 TEST1 jelly`

Main

```c
int main(int argc, char *argv[]) {
    assert(argc == 4);

    int n_workers = atoi(argv[1]);
    char *rootpath = argv[2];
    char *search_string = argv[3];
    global_search_string = search_string;
```

`main()` takes three command line inputs `n_workers, rootpath, and search_string`, with `argv[0]` containing the executable name by default. The assertion `'argc == 4'` aborts the program if the incorrect amount of input was received.

`search_string` is assigned to a global variable `global_search_string` since the thread functions require this variable for execution.

Globals

```c
// Globals
char *global_search_string;
int open_dirs = 1;

struct queue *taskqueue;
```

Aside from `global_search_string`, other global variables include `taskqueue` of type `struct queue` and `open_dirs` which is used for the terminating condition of the threads.

Thread creation

```c
    // Main thread enqueues rootpath
    taskqueue = initqueue();
    enqueue(taskqueue, rootpath);

    // Thread creation
    pthread_t workers[n_workers];
    for (uintptr_t i = 0; i < n_workers; i++) {
        pthread_create(&workers[i], NULL, search, (void *)i);
    }

    for (uintptr_t i = 0; i < n_workers; i++) {
        pthread_join(workers[i], NULL);
    }
```

The main thread first initializes the task queue and then enqueues the given `rootpath`. Then, proceed to creation of `n_workers` threads (8 in this case).

We require the threads to pass their thread number (later referred to as `worker_ID`) to the search function. To ensure safety in type conversion between pointer and `long int`, we use the type `uintptr_t` from `stdint.h`.

Search function – Deqeueing the next task

```c
39 void *search(void *id) {
40     uintptr_t worker_ID = (uintptr_t) id;
41     char dir_path[250];
42 
43     while (open_dirs > 0) {
44         int empty = 0;
45         pthread_mutex_lock(&lock);
46             empty = isEmpty(taskqueue);
47         pthread_mutex_unlock(&lock);
48 
49         if(!empty){
50 
51             // Take a new job
52             pthread_mutex_lock(&lock);  // Only one thread can take a single job at a time
53             if(isEmpty(taskqueue)) {
54                 pthread_mutex_unlock(&lock);
55                 continue;    // Continue loop if task queue has become empty.
56             }
57             strcpy(dir_path, taskqueue->front->path);
58             dequeue(taskqueue);
59             pthread_mutex_unlock(&lock);
```

Convert the passed pointer argument back to a `long int worker_ID`. Initialize the string `dir_path` which is a maximum of 250 characters.

The `while` loop depends on the number of open directories `open_dirs`, which was initialized at `1` since the root directory is enqueued at the start. Lines 45 to 47 access the front of `taskqueue` to check if it is empty. If not, (`empty == 0`) proceed to line 51 up to 59. There is a second check on the front of `taskqueue` in case two consecutive threads attempt to dequeue when there is only one task left.

Copy (using `strcpy`) the path contained at the front of `taskqueue` to `dir_path`, and finally dequeue and take the next task.

Search function – opening a directory

```
61                  char abs_path[250] = {0};
62                  realpath(dir_path, abs_path);
63                  printf("[%ld] DIR %s\n", worker_ID, abs_path);
64
65                  DIR *dir = opendir(dir_path);
66                  if (dir == NULL) continue;
67                  struct dirent *content;
```

The instructions require the absolute path to be printed; we can accomplish this with the `realpath()` function and save the return value to `abs_path`. Then print a line indicating the `worker_ID` that will open this directory.

Lines 65 to 67 handle the opening of the directories; line 66 ensures that an error in opening a directory causes the thread to restart the while loop.

Search function – child is a directory

```
69              while ((content = readdir(dir)) != NULL) {
70                  if (!strcmp(content->d_name, ".") || !strcmp(content->d_name, "..")) continue;
71
72                  int type = content->d_type;
73                  if (type == DT_DIR) {           // Child is a directory
74                      pthread_mutex_lock(&lock);  // Only one thread can modify task queue at a time
75                      char new_path[250] = {0};
76                      strcat(new_path, dir_path);
77                      strcat(new_path, "/");
78                      strcat(new_path, content->d_name);
79                      enqueue(taskqueue, new_path);
80
81                      realpath(new_path, abs_path);
82                      printf("[%ld] ENQUEUE %s\n", worker_ID, abs_path);
83                      pthread_mutex_lock(&increment_done_lock);
84                      open_dirs++;
85                      pthread_mutex_unlock(&increment_done_lock);
86                      pthread_mutex_unlock(&lock);
87                  }
```

Enter another `while` loop that reads through all subdirectories of the rootpath.

Line 70 makes the program ignore all `.` and `..` file objects.

Line 73 checks the type of file that was read using the `d_type` member of the `dirent` struct. `DT_DIR` indicates that this is a directory. Declare and initialize a string `new_path`. We generate the path to this directory by appending its name to `dir_path` using a series of `strcat` calls and store it in `new_path`. Then, enqueue `new_path` and print a line indicating the thread's `worker_ID` that enqueued this directory. Finally, increment `open_dirs` to keep track of how many directories the threads must search through.

Search function – child is a file

```
89                  else if (type == DT_REG) {        // Child is a file
90                      char grep_command[600];
91                      char file_path[250] = {0};
92                      strcat(file_path, dir_path);
93                      strcat(file_path, "/");
94                      strcat(file_path, content->d_name);
95                      sprintf(grep_command, "grep \"%s\" \"%s\" 1> /dev/null", global_search_string, file_path);
96
97                      int grep_retval = system(grep_command);
98                          realpath(file_path, abs_path);
99                          if (grep_retval == 0)
100                             printf("[%ld] PRESENT %s\n", worker_ID, abs_path);
101                         else
102                             printf("[%ld] ABSENT %s\n", worker_ID, abs_path);
103                 }
104             }
```

Line 89 checks the type of file that was read using the `d_type` member of the `dirent` struct. `DT_REG` indicates that this is a file. Declare a string `grep_command` that is 600 characters long, and a string `file_path` to hold the path towards the file that was found.

Append the name of this file to `dir_path` using a series of `strcat` calls and store it in `file_path`. Then, create a formatted string using `sprintf`, with the format "grep "{search_string}" "{file_path}" 1> /dev/null". The quotation marks around `search_string` and `file_path` handle cases with whitespaces in the search string or directory and file names. Output (`stdout`) is redirected to `/dev/null`. This formatted string is stored in `grep_command`, which is passed to the `system()` function to execute as a shell command.

Based on the exit status of `grep`, we can determine if the `search_string` was found within the file. If `grep_retval == 0`, print a line indicating the thread `worker_ID` that completed the successful search with "`PRESENT`". Otherwise, print the same but with "`ABSENT`" instead.

Search function – closing a directory

```
105                 closedir(dir);
106                 pthread_mutex_lock(&increment_done_lock);
107                 open_dirs--;
108                 pthread_mutex_unlock(&increment_done_lock);
109             }
110             else {
111                 continue;
112             }
113         }
114         return NULL;
115     }
```

Deallocate directories that have been searched through using `closedir()`, then decrement `open_dirs`. The `else` at line 110 is the counterpart to the `if` statement that checks if the task queue is empty. Simply continue the `while` loop when the queue is empty.

The while loop breaks when `open_dirs` has reached `0`. Finally, return `NULL` and go back to `main()`, and free the memory allocated for the queue.

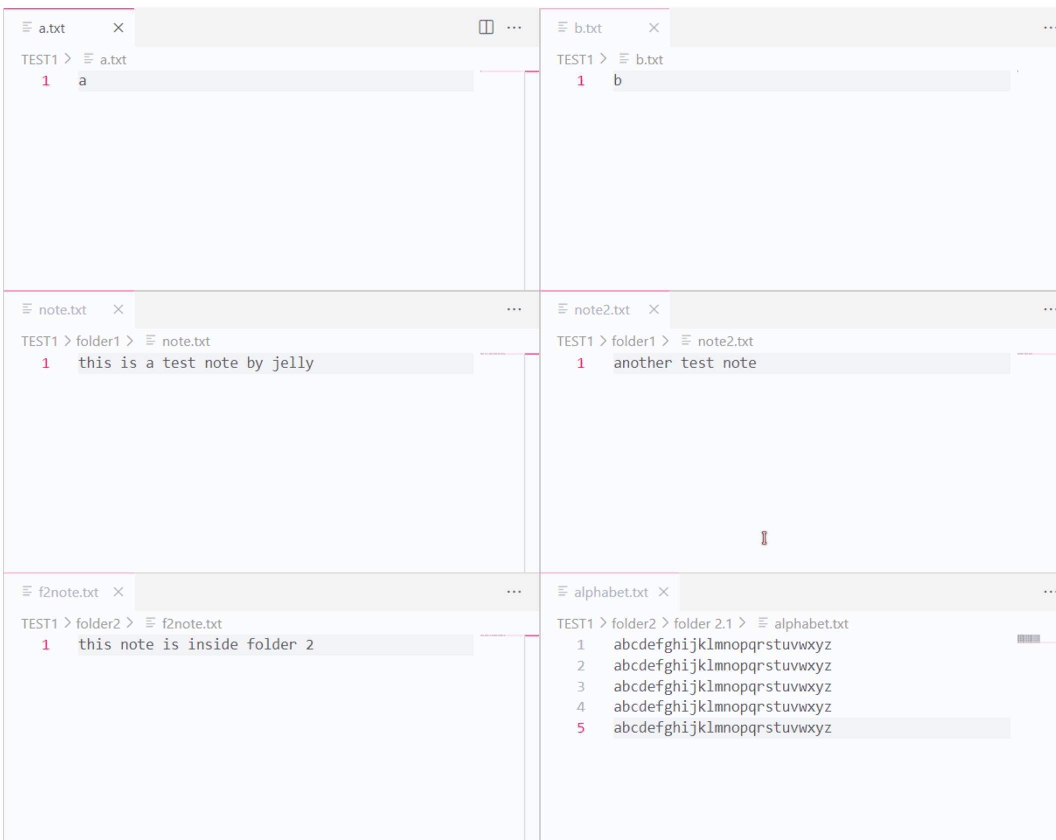
```
    free(taskqueue);

    return 0;
```

Test 1 Output

```
jellynux@LAPTOP-M0RVCM01:~/cs140221project2-a-raborar$ ./multithreaded 8 TEST1 jelly
[0] DIR /home/jellyunix/cs140221project2-a-raborar/TEST1
[0] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/TEST1/folder1
[3] DIR /home/jellyunix/cs140221project2-a-raborar/TEST1/folder1
[0] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/TEST1/folder2
[4] DIR /home/jellyunix/cs140221project2-a-raborar/TEST1/folder2
[0] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/TEST1/folder3
[1] DIR /home/jellyunix/cs140221project2-a-raborar/TEST1/folder3
[1] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/TEST1/folder3/folder 3.1
[5] DIR /home/jellyunix/cs140221project2-a-raborar/TEST1/folder3/folder 3.1
[0] ABSENT /home/jellyunix/cs140221project2-a-raborar/TEST1/a.txt
[4] ABSENT /home/jellyunix/cs140221project2-a-raborar/TEST1/folder2/f2note.txt
[4] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/TEST1/folder2/folder 2.1
[6] DIR /home/jellyunix/cs140221project2-a-raborar/TEST1/folder2/folder 2.1
[3] PRESENT /home/jellyunix/cs140221project2-a-raborar/TEST1/folder1/note.txt
[0] ABSENT /home/jellyunix/cs140221project2-a-raborar/TEST1/b.txt
[6] ABSENT /home/jellyunix/cs140221project2-a-raborar/TEST1/folder2/folder 2.1/alphabet.txt
[3] ABSENT /home/jellyunix/cs140221project2-a-raborar/TEST1/folder1/note2.txt
```

The substring `jelly` was found in 1 file (note.txt) out of 6 files. The contents of each text file are shown below:



Test 2. Given the following directory tree:

```
testdir
├── testcase1
│   ├── text1.1.txt
│   ├── text1.2.txt
│   └── text1.3.txt
├── testcase2
│   ├── dir2.1
│   │   ├── dir2.2
│   │   │   └── text 2.2.1.txt
│   │   └── text2.1.1.txt
│   ├── text2.1.txt
│   └── text2.2.txt
└── testcase3
    └── dir 3.1
        └── dir 3.2
```

Each text file contains a randomized list of 25 words. Executed using `./multithreaded 5 /home/jellyunix/cs140221project2-a-raborar/testdir love`

Test 2 Output

```
jellynux@LAPTOP-M0RVCMO1:~/cs140221project2-a-raborar$ ./multithreaded 5 /home/jellyunix/cs140221project2-a-raborar/testdir love
[0] DIR /home/jellyunix/cs140221project2-a-raborar/testdir
[0] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/testdir/testcase3
[4] DIR /home/jellyunix/cs140221project2-a-raborar/testdir/testcase3
[4] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/testdir/testcase3/dir 3.1
[4] DIR /home/jellyunix/cs140221project2-a-raborar/testdir/testcase3/dir 3.1
[0] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2
[0] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/testdir/testcase1
[0] DIR /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2
[3] DIR /home/jellyunix/cs140221project2-a-raborar/testdir/testcase1
[4] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/testdir/testcase3/dir 3.1/dir 3.2
[2] DIR /home/jellyunix/cs140221project2-a-raborar/testdir/testcase3/dir 3.1/dir 3.2
[0] PRESENT /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2/text2.1.txt
[0] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2/dir 2.1
[3] ABSENT /home/jellyunix/cs140221project2-a-raborar/testdir/testcase1/text1.3.txt
[1] DIR /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2/dir 2.1
[3] ABSENT /home/jellyunix/cs140221project2-a-raborar/testdir/testcase1/text1.1.txt
[1] ABSENT /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2/dir 2.1/text2.1.1.txt
[1] ENQUEUE /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2/dir 2.1/dir 2.2
[1] DIR /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2/dir 2.1/dir 2.2
[0] ABSENT /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2/text2.2.txt
[3] ABSENT /home/jellyunix/cs140221project2-a-raborar/testdir/testcase1/text1.2.txt
[1] PRESENT /home/jellyunix/cs140221project2-a-raborar/testdir/testcase2/dir 2.1/dir 2.2/text 2.2.1.txt
```

The substring `love` was found in 2 files (testcase2/text2.1.txt and testcase2/dir 2.1/dir 2.2/text 2.2.1.txt) out of 7 files. The contents of the two text files are shown below:

text2.1.txt

testdir > testcase2 > text2.1.txt

```
competition
interaction
measurement
development
love
bath
explanation
solution
paper
conversation
method
satisfaction
coffee
hospital
hair
birthday
election
menu
description
supermarket
emphasis
database
two
internet
thanks
```

text 2.2.1.txt

testdir > testcase2 > dir 2.1 > dir 2.2 > text 2.2.1.txt

```
weakness
employment
alcohol
psychology
chemistry
way
hospital
context
diamond
sample
safety
impression
community
medicine
steak
negotiation
love
effort
error
owner
combination
student
opinion
flight
bath
```

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

The task queue was implemented as a linked list queue. This was made concurrent and threadsafe using a `mutex` (simple lock).

```c
struct queue *taskqueue;
struct node {
    char path[250];
    struct node *next;
};

struct queue {
    struct node *front;
    struct node *rear;
    pthread_mutex_t queue_lock;
};
```

The `node` struct has two fields. First is `path`, occupying a maximum of 250 characters and containing the path of the directory to search. Second is `next`, a pointer to the next node in the linked list.

The `queue` struct fields include pointers to the nodes of the front and rear of the queue, and a lock `queue_lock`.

The task queue has five associated functions: `initqueue`, `enqueue`, `dequeue`, `isEmpty`, and `printqueue`.

`initqueue()`

```
struct queue* initqueue() {
    struct queue *tasks = (struct queue*)malloc(sizeof(struct queue));
    tasks->front = NULL;
    tasks->rear = NULL;
    pthread_mutex_init(&tasks->queue_lock, NULL);
    return tasks;
}
```

First uses `malloc` to allocate memory for the queue named `tasks`. Set the `front` and `rear` pointers of `tasks` to NULL, and initialize the queue lock.

`enqueue()`

```
154 void enqueue(struct queue *tasks, char *path) {
155     struct node *newnode = (struct node*)malloc(sizeof(struct node));
156     assert(newnode != NULL);
157 
158     strcpy(newnode->path, path);
159     newnode->next = NULL;
160 
161     pthread_mutex_lock(&tasks->queue_lock);
162     if (tasks->front == NULL) {
163         tasks->front = newnode;
164         tasks->rear = newnode;
165     }
166     else {
167         tasks->rear->next = newnode;
168         tasks->rear = newnode;
169     }
170     pthread_mutex_unlock(&tasks->queue_lock);
171 }
```

First uses `malloc` to allocate memory for a new node named `newnode`, which cannot be `NULL`. Copy the argument `path` to be enqueued into the `path` field of `newnode`, and set its `next` field to `NULL`.

The next section (lines 161 to 170) must be protected by a lock because we must ensure that only one thread is modifying the queue at any time.

If the queue is empty, set the `front` and `rear` pointers to point to `newnode`. Else, attach `newnode` to the rear of the queue.

`dequeue()`

```c
void dequeue(struct queue *tasks) {
    pthread_mutex_lock(&tasks->queue_lock);
    struct node *temp = tasks->front;
    if (temp == NULL) {
        pthread_mutex_unlock(&tasks->queue_lock);
        return;
    }
    else {
        tasks->front = temp->next;
        pthread_mutex_unlock(&tasks->queue_lock);
        free(temp);
        return;
    }
}
```

This function must be protected by a lock because we must ensure that only one thread is modifying the queue at any time.

Create a node `temp` to store the `front` node pointer. If this is `NULL`, it means the queue is empty. Do nothing and release the lock, then return.

Else if the `front` node is not `NULL`, set the node after `front` as the new `front` node. Release the lock, free the `temp` node, then return.

`isEmpty()`

```c
int isEmpty(struct queue *tasks) {
    pthread_mutex_lock(&tasks->queue_lock);
    if (tasks->front == NULL) {
        pthread_mutex_unlock(&tasks->queue_lock);
        return 1;
    }
    else {
        pthread_mutex_unlock(&tasks->queue_lock);
        return 0;
    }
}
```

A function to check if the queue is empty by checking on the `front` node. Similar to `dequeue`, except with a return value of `1` if the `front` node is `NULL` and a return value of `0` if it is not `NULL`.

`printqueue()`

```c
void printqueue(struct queue *tasks) {
    struct node *current = tasks->front;
    if (current == NULL) {
        printf("No tasks queued.\n");
        return;
    }
    printf("\nQUEUE: ");
    while(current != NULL) {
        printf("%s--", current->path);
        current = current->next;
    }
    printf("\n");
    free(current);
}
```

A currently unused function that I mainly only called during the debugging phase. Creates a node `current` from the `front` node that will traverse the whole linked list and print each node's `path`.

Synchronization

Synchronization was achieved by ensuring that, while all the threads are searching, only one thread at a time can access or modify the task queue. This was achieved by protecting critical sections with a lock.

```c
// Initialize locks
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t increment_done_lock = PTHREAD_MUTEX_INITIALIZER;
```

The critical sections are pictured as follows:

```c
int empty = 0;
pthread_mutex_lock(&lock);
    empty = isEmpty(taskqueue);
pthread_mutex_unlock(&lock);

if(!empty){
```

Checking on `taskqueue` protected

```c
// Take a new job
pthread_mutex_lock(&lock);  // Only one thread can take a single job at a time
if(isEmpty(taskqueue)) {
    pthread_mutex_unlock(&lock);
    continue;    // Continue loop if task queue has become empty.
}
strcpy(dir_path, taskqueue->front->path);
dequeue(taskqueue);
pthread_mutex_unlock(&lock);
```

Dequeueing (and second check) protected

```c
int type = content->d_type;
if (type == DT_DIR) {            // Child is a directory
    pthread_mutex_lock(&lock);  // Only one thread can modify task queue at a time
    char new_path[250] = {0};
    strcat(new_path, dir_path);
    strcat(new_path, "/");
    strcat(new_path, content->d_name);
    enqueue(taskqueue, new_path);

    realpath(new_path, abs_path);
    printf("[%ld] ENQUEUE %s\n", worker_ID, abs_path);
    pthread_mutex_lock(&increment_done_lock);
    open_dirs++;
    pthread_mutex_unlock(&increment_done_lock);
    pthread_mutex_unlock(&lock);
}
```

Enqueueing protected

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

Termination of the `while` loop depends on the global variable `open_dirs`. This is a representation of the number of directories to search through; since the `rootpath` was enqueued in `main`, we automatically have at least one directory, thus `open_dirs` was initialized at `1`. For every new directory enqueued, increment `open_dirs`. For every `closedir()`, decrement `open_dirs`. Eventually, `open_dirs` will reach `0` and the `while` loop will terminate.

However, since `open_dirs` is a global variable, it must be protected against data races by protecting the increment and decrement sections with another lock, `increment_done_lock`.

```c
// Initialize locks
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t increment_done_lock = PTHREAD_MUTEX_INITIALIZER;

// Globals
char *global_search_string;
int open_dirs = 1;
```

The lock and global variable `open_dirs`

```c
while (open_dirs > 0) {
    int empty = 0;
```

The condition of the `while` loop

```
printf("[%ld] ENQUEUE %s\n", worker_ID, abs_path);
pthread_mutex_lock(&increment_done_lock);
open_dirs++;
pthread_mutex_unlock(&increment_done_lock);
```

Critical section: increment `open_dirs` after enqueueing a new directory

```
closedir(dir);
pthread_mutex_lock(&increment_done_lock);
open_dirs--;
pthread_mutex_unlock(&increment_done_lock);
```

Critical section: decrement `open_dirs` after closing a directory
