

CS 140 Project 2: Parallelized grep Runner

Documentation

Ragunton, Carl David B.

2020-04243

LAB-2

Documentation Video Link:

https://drive.google.com/file/d/1PB0XSrMvH_xHuQvpAHkkfPYYJbsBK_DY/view?usp=sharing

I was able to do single thread and multithreaded. I wasn't able to do the multiprocessor one.

Multithreaded Documentation

1. References

<https://www.digitalocean.com/community/tutorials/queue-in-c>

-to know how queue works

<https://man7.org/linux/man-pages/man3/opendir.3.html>

-to know how opendir works

<https://pubs.opengroup.org/onlinepubs/7908799/xsh/readdir.html>

-to know how readdir works

<https://stackoverflow.com/questions/29559414/how-to-get-the-output-of-grep-in-c>

-to know how to use grep in C

<https://stackoverflow.com/questions/298510/how-to-get-the-current-directory-in-a-c-program>

-to know how to get current directory

<https://www.cyberciti.biz/faq/howto-use-grep-command-in-linux-unix/>

-to know what grep to utilize in the code

https://www.gnu.org/software/libc/manual/html_node/Directory-Entries.html

-to know if the file is a regular file or a directory

The ideas from these references are included in the codes. They are more or less explained in the video.

2.a.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <limits.h>
#include <semaphore.h>
```

First, we include these into the code. Note that not all of them are used since the code was continuously edited.

```
pthread_t tid[8]; //create id for threads
int letin[8]; //at start and after every enqueue, let threads enter the while loop
int sleeping[8]; //to see if the thread is sleeping
int currsleep=0; //to know the number of threads sleeping
```

These are global variables/arrays and their job are as stated in the comments.

```
pthread_mutex_t lock; //for locking critical sections
sem_t queuelock; //for forcing a worker sleep when queue is empty
int fin = 0; //if 1, every worker should wake up
```

The purpose of the mutex and queuelock are as shown in the picture. Having fin = 0 means that the queue will still be having more values coming in.

```
//for using popen - to use grep command
FILE *popen(const char *command, const char *mode);
int pclose(FILE *stream);
```

These are for using the grep command from C.

```
//declaring the queue
char *queue[100000];
int Rear = - 1;
int Front = - 1;
```

We just declare the queue and its start and end index through Front and Rear.

```
//for the multithreaded version
struct args {
    char *exe;
    int ID;
    char *root;
    char *search;
    int N;
};
```

This is what the parameter of the thread function will get.

```
//enqueueing in the queue
void enqueue(char *insert_item)
{
    if (Rear == 100000 - 1)
        printf("Overflow \n");
    else
    {
        if (Front == - 1)

            Front = 0;
        Rear = Rear + 1;
        strcpy(queue[Rear],insert_item);
    }
}
```

This is the function for enqueueing in the queue.

```
//dequeueing in the queue
void dequeue()
{
    if (Front == - 1 || Front > Rear)
    {
        printf("Underflow \n");
        return ;
    }
    else
    {
        Front = Front + 1;
    }
}
```

This is the function for dequeueing in the queue.

```
//just for checking the queue
void show()
{
    if (Front == - 1)
        printf("Empty Queue \n");
    else
    {
        printf("Queue: \n");
        for (int i = Front; i <= Rear; i++)
            printf("%s ", queue[i]);
        printf("\n");
    }
}
```

This is something that I just used for checking the contents of the queue. This function will never be called in the actual code.

```

void main(int argc, char* argv){
    //allocating space for the queue
    for (int j=0; j<100000; j++){
        queue[j]=malloc(300);
    }

    for (int j=0; j<8; j++){
        letin[j]=1;
        sleeping[j]=0;
    }
}

```

For the main, we use int argc and char* argv[] as the parameters to be able to get the inputs included in running the code. Then, we allocate space for the whole queue. We also initialize letin and sleeping.

```

sem_init(&queuelock, 0, 1); //initializing the semaphore

```

We initialize the semaphore to 1. Meaning after a process passed through it, no other process can as long as sem_wait is not called.

```

char *executable=argv[0];
char *Number=argv[1];
char *rootpath=argv[2];
char *searchfor=argv[3];

```

We access the inputs included in running the code.

```

int N = atoi(Number);
int slash = argv[2][0]; //get the first character of argv[2]

char cwd[300]; //declare cwd or current directory

if (slash == 47){ //if the first character of the rootpath is "/"
    strcpy(cwd, rootpath); //use it as the rootpath
}
else{
    getcwd(cwd, sizeof(cwd)); //get the current directory
    strcat(cwd, "/");
    strcat(cwd, argv[2]); //add the given rootpath to it
}

```

We make N an int from the Number. We get the first character of the rootpath as ASCII in decimal. If it is a “/”, we use it as it is. If not, we get the current directory and connect the given rootpath to it. This is the first directory that we need to access.

```
enqueue(cwd);
```

This enqueues the directory we got earlier to the queue.

```

struct args arg[N];
for (int i=0;i<N;i++){
    arg[i].exe=executable;
    arg[i].ID=i;
    arg[i].root=rootpath;
    arg[i].search=searchfor;
    arg[i].N=N;
    pthread_create(&tid[i], NULL, (void *) work, &arg[i]);
}

```

Here, we declare a struct which we will use to store the values passed to the thread. Note that some of these are not used in the threads. Based on the given N or argv[2], we will create threads which will have arg[i] as its parameter.

There will be N threads which will go to the same function.

```

void *work(struct args *arg){
    DIR *dir;
    struct dirent *dp;

    while (Front<=Rear || letin[arg->ID] == 1){ //stop when there are nothing to do left; after every enqueue, threads can come again
        pthread_mutex_lock(&lock);
        sleeping[arg->ID] = 1; //the current thread will sleep
        currsleep++;
        pthread_mutex_unlock(&lock);

        if (currsleep==(struct args*)arg->N)sem_post(&queueunlock); //force a thread to wake up
        else if (currsleep<((struct args*)arg->N) sem_wait(&queueunlock); //makes the worker sleep if queue is empty
    }
}

```

All the threads will execute this function.

We declare *dir and *dp. They will be used later for the opendir and readdir. While front<=rear or the queue is still not empty, they will go in here. Also, there is the array letin which tells if the threads can enter here unconditionally. At first, all the threads have their letin = 1, so, they can enter here for the first time.

When the thread goes in, it is almost bound to go to sleep aside from the first thread that goes here. Thus, we will set the sleeping of the thread to be 1 as it passes here. Of course, currsleep which is the number of sleeping threads should also go up. Note that we locked this part with mutex because these are global variables and arrays.

If the number of supposed to be sleeping threads is the number of workers, we will not use sem_wait, instead we are going to use sem_post to wake up a sleeping thread. If we are to call sem_wait in this scenario, all the threads will sleep and nothing will work. If the number of sleeping threads is still less than N, we call sem_wait. The main purpose of this sem_wait is to only let one thread go in at first because there is only one queue entry in that time. This is to let the other threads wait for the first thread to enqueue more directories in the queue.

```

pthread_mutex_lock(&lock);
sleeping[arg->ID]=0; //the current thread wakes up
currsleep--;
pthread_mutex_unlock(&lock);

if (fin == 1) break; //there is nothing to do left

```

Assuming a thread woke up or passes through the sem_wait, it should not be sleeping. Thus we set the sleeping of that thread back to 0 and decrement currsleep.

For the if statement, we would discuss it later.

```

char current_file[300];
pthread_mutex_lock(&lock);
letin[arg->ID]=0;
strcpy(current_file,queue[Front]); //get the first item/directory in the queue
dequeue();
printf("[%d] DIR %s\n",arg->ID,current_file);
pthread_mutex_unlock(&lock);

dir=opendir(current_file);

```

We declare `current_file` which we let to be the first entry of the queue or the current directory that we want to access. We also make the `letin` of the current thread to be 0. If this is 0, it can't just enter the while loop unconditionally.

Then, we dequeue the first directory in the queue. We print the correct format. We get the `dir` by using `opendir`.

```

while ((dp = readdir(dir)) != NULL) {
    if ((strcmp(dp->d_name, ".")==0 || (strcmp(dp->d_name, "..")==0)){ //ignore these files
    }
    else{
        char checkfile[300];
        strcpy(checkfile, current_file);
        strcat(checkfile, "/");
        strcat(checkfile, dp->d_name); //the true current directory that needs to be checked
    }
}

```

Then, to gain access to all of the contents of the directory, we use a while loop and `readdir`. The `dp` will gain the value of the `readdir` call. If the name of the file/directory is "." or "..", we just ignore them. Else, we declare `checkfile` to be the `current_file` plus the name of the next file/directory.


```

if (dp->d_type == DT_DIR){ //if it is a directory
    char deepcheckfile[300];
    strcpy(deepcheckfile,checkfile); //copy will be enqueued to the queue

    pthread_mutex_lock(&lock);
    enqueue(deepcheckfile);

    for (int j=0; j<8; j++){ //after an enqueue, refresh letin to let the threads come in again
        letin[j]=1;
    }

    printf("[%d] ENQUEUE %s\n",arg->ID,checkfile);
    sem_post(&queueunlock);

    pthread_mutex_unlock(&lock);
}

```

We need to check if the checkfile is a directory or a regular file. This is the part for when it is a directory. If it is a directory, we make another variable deepcheckfile which is the copy of checkfile. Then, we enqueue it to the queue. Because we have just enqueued a new directory, we should give the threads a chance to handle it. So, we set all the letin to 1 again.

Then, we print the correct format of the output as seen. Since the queue has a new entry, we let one of the sleeping threads wake up with sem_post and handle that queue entry.

```

else if(dp->d_type == DT_REG){ //if it is a regular file
    char command[100000];

    strcpy(command, "grep -c \"");
    strcat(command, ((struct args*)arg)->search);
    strcat(command, "\" \"");
    strcat(command, checkfile); //make the grep command
    strcat(command, "\"");

    FILE *grep;
    grep= popen(command, "r"); //execute the grep command
    if (grep == NULL) {
        perror("popen");
        exit(EXIT_FAILURE);
    }

    char grep_value[1024];
    fgets(grep_value, sizeof(grep_value), grep); //get the output of grep
    int Grep_value = atoi(grep_value);
    pclose(grep);

    if (Grep_value!=0){ //something has been found
        printf("[%d] PRESENT %s\n",arg->ID,checkfile);
    }
    else if(Grep_value==0){ //nothing has been found
        printf("[%d] ABSENT %s\n",arg->ID,checkfile);
    }
}

```

This is the case wherein the file we access is a regular file. If it is a regular file, we are to perform grep in it. Thus, we prepare the grep command by concatenating different strings to one another. In building this string, we put a " " on the searched for text and on the directory. This is to let the program handle directories and searches with spaces on them. We perform the grep operation to get its int value using grep_value. If the grep_value has a value of not equal to 0, it means that it has found a match. If it is zero it has no match. We just print the correct output.

```

        else if(Grep_value==0){ //nothing has been found
            printf("[%d] ABSENT %s\n",arg->ID,checkfile);
        }
    }

    }

    closedir(dir); //close the dir after using it
}

pthread_mutex_lock(&lock); //there is nothing to do left
fin = 1;
pthread_mutex_unlock(&lock);
sem_post(&queueunlock); //wakes up every sleeping process and force them to end
}

```

After using the entire dir, we just close it.

If a thread ever comes out of the while loop, it means that there will be no more directories to enqueue. If this happens, we set fin to 1 and wake up a sleeping thread.

```

while (Front<=Rear || letin[arg->ID] == 1){ //stop when there are nothing to do left; after every enqueue, threads can come again
    pthread_mutex_lock(&lock);
    sleeping[arg->ID] = 1; //the current thread will sleep
    currsleep++;
    pthread_mutex_unlock(&lock);

    if (currsleep==((struct args*)arg)->N)sem_post(&queueunlock); //force a thread to wake up

    else if (currsleep<((struct args*)arg)->N) sem_wait(&queueunlock); //makes the worker sleep if queue is empty

    pthread_mutex_lock(&lock);
    sleeping[arg->ID]==0; //the current thread wakes up
    currsleep--;
    pthread_mutex_unlock(&lock);

    if (fin == 1) break;//there is nothing to do left
}

```

Going back to the part earlier, a thread will wake up and will see that fin==1. Thus, the thread that woke up will go out of the while loop where another sem_wait awaits. It will wake up another process and the loop will continue until all the threads have woken and stopped.

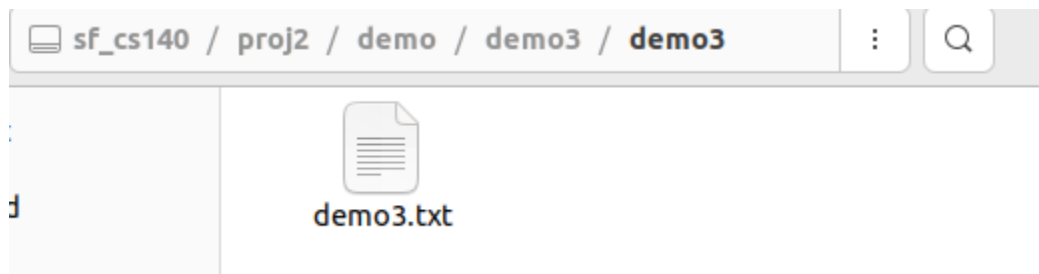
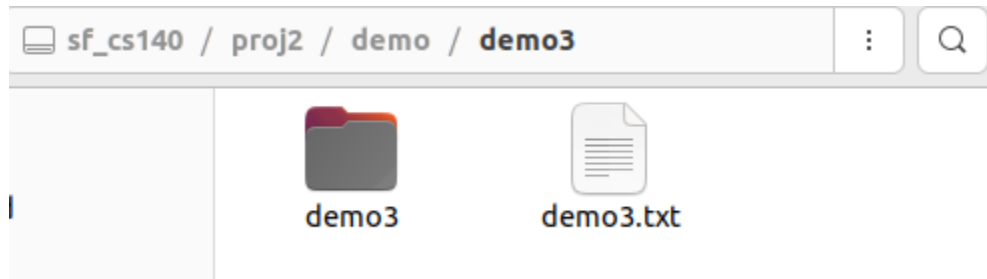
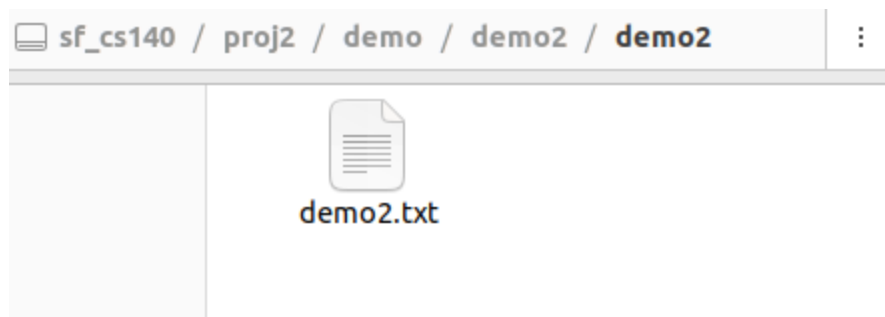
```
    for (int i=0;i<N;i++){  
        pthread_join(tid[i], NULL);  
    }  
  
    for (int k = 0; k<100000; k++){  
        free(queue[k]);  
    }  
  
    pthread_mutex_destroy(&lock);  
  
    return;  
}
```

This is back on the main function. The threads will be caught using `pthread_join`. Then we free all the spaces taken up by the queue. Then, destroy the mutex lock and return.

Now that we know how the code works, we can proceed to the demo with N=2, 1PRESENT, 5ABSENT, and 6DIRs.

The rootpath will be demo inside /media/sf_cs140/proj2.





We are going to make the searched for text be "CS140". Out of all the .txt files shown in the pictures above, only one contains "CS140". That is the demo1.txt inside /media/sf_cs140/proj2/demo/demo1.

If we compile with “gcc -pthread multithreaded.c -o multithreaded”.

Then, run “./multithreaded 2 demo CS140”.

The result will be:

```
cs140@cs140:/media/sf_cs140/proj2$ gcc -pthread multithreaded.c -o multithreaded
cs140@cs140:/media/sf_cs140/proj2$ ./multithreaded 2 demo CS140
[1] DIR /media/sf_cs140/proj2/demo
[1] ENQUEUE /media/sf_cs140/proj2/demo/demo1
[1] ENQUEUE /media/sf_cs140/proj2/demo/demo2
[1] ENQUEUE /media/sf_cs140/proj2/demo/demo3
[0] DIR /media/sf_cs140/proj2/demo/demo1
[1] DIR /media/sf_cs140/proj2/demo/demo2
[1] ENQUEUE /media/sf_cs140/proj2/demo/demo2/demo2
[0] ENQUEUE /media/sf_cs140/proj2/demo/demo1/demo1
[1] ABSENT /media/sf_cs140/proj2/demo/demo2/demo2.txt
[0] PRESENT /media/sf_cs140/proj2/demo/demo1/demo1.txt
[1] DIR /media/sf_cs140/proj2/demo/demo3
[0] DIR /media/sf_cs140/proj2/demo/demo2/demo2
[1] ENQUEUE /media/sf_cs140/proj2/demo/demo3/demo3
[1] ABSENT /media/sf_cs140/proj2/demo/demo3/demo3.txt
[1] DIR /media/sf_cs140/proj2/demo/demo1/demo1
[0] ABSENT /media/sf_cs140/proj2/demo/demo2/demo2/demo2.txt
[0] DIR /media/sf_cs140/proj2/demo/demo3/demo3
[1] ABSENT /media/sf_cs140/proj2/demo/demo1/demo1/demo1.txt
[0] ABSENT /media/sf_cs140/proj2/demo/demo3/demo3/demo3.txt
cs140@cs140:/media/sf_cs140/proj2$
```

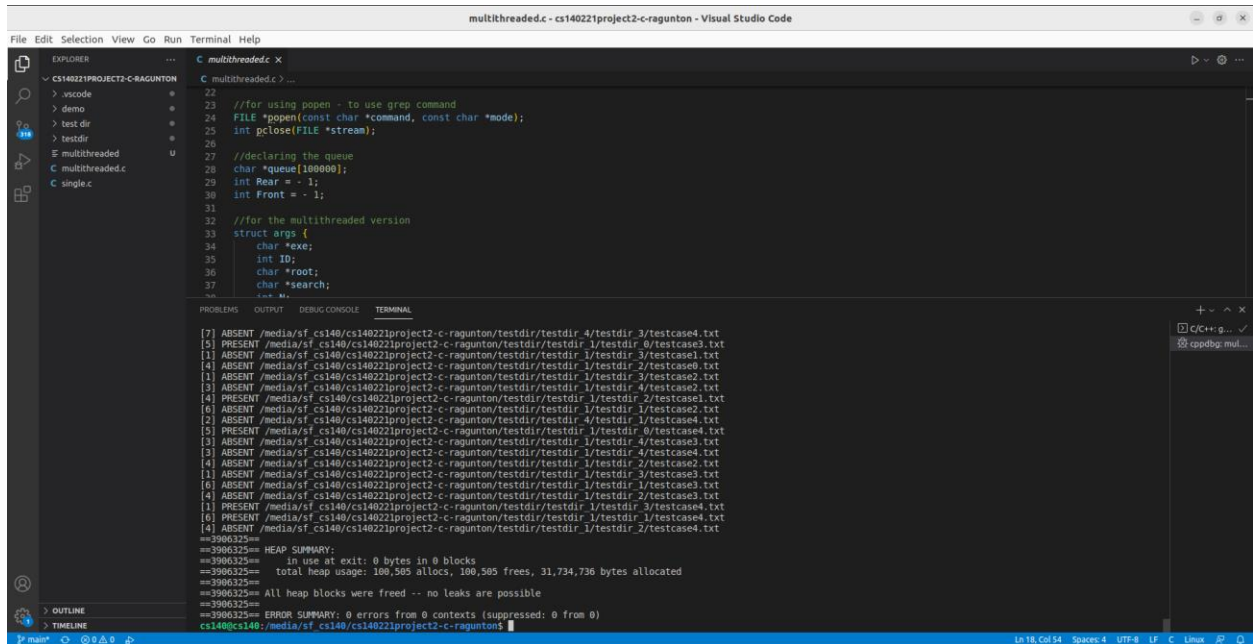
If N=3:

```
[0] ABSENT /media/sf_cs140/proj2/demo/demo3/demo3/demo3.txt
cs140@cs140:/media/sf_cs140/proj2$ gcc -pthread multithreaded.c -o multithreaded
./multithreaded 3 demo CS140
[0] DIR /media/sf_cs140/proj2/demo
[0] ENQUEUE /media/sf_cs140/proj2/demo/demo1
[0] ENQUEUE /media/sf_cs140/proj2/demo/demo2
[0] ENQUEUE /media/sf_cs140/proj2/demo/demo3
[1] DIR /media/sf_cs140/proj2/demo/demo1
[0] DIR /media/sf_cs140/proj2/demo/demo2
[1] ENQUEUE /media/sf_cs140/proj2/demo/demo1/demo1
[0] ENQUEUE /media/sf_cs140/proj2/demo/demo2/demo2
[2] DIR /media/sf_cs140/proj2/demo/demo3
[1] PRESENT /media/sf_cs140/proj2/demo/demo1/demo1.txt
[1] DIR /media/sf_cs140/proj2/demo/demo1/demo1
[0] ABSENT /media/sf_cs140/proj2/demo/demo2/demo2.txt
[0] DIR /media/sf_cs140/proj2/demo/demo2/demo2
[1] ABSENT /media/sf_cs140/proj2/demo/demo1/demo1/demo1.txt
[0] ABSENT /media/sf_cs140/proj2/demo/demo2/demo2/demo2.txt
[2] ENQUEUE /media/sf_cs140/proj2/demo/demo3/demo3
[2] ABSENT /media/sf_cs140/proj2/demo/demo3/demo3.txt
cs140@cs140:/media/sf_cs140/proj2$
```

If N=8:

```
● cs140@cs140:/media/sf_cs140/proj2$ gcc -pthread multithreaded.c -o multithreaded
./multithreaded 8 demo CS140
[0] DIR /media/sf_cs140/proj2/demo
[0] ENQUEUE /media/sf_cs140/proj2/demo/demo1
[0] ENQUEUE /media/sf_cs140/proj2/demo/demo2
[0] ENQUEUE /media/sf_cs140/proj2/demo/demo3
[2] DIR /media/sf_cs140/proj2/demo/demo1
[4] DIR /media/sf_cs140/proj2/demo/demo2
[1] DIR /media/sf_cs140/proj2/demo/demo3
[2] ENQUEUE /media/sf_cs140/proj2/demo/demo1/demo1
[4] ENQUEUE /media/sf_cs140/proj2/demo/demo2/demo2
[5] DIR /media/sf_cs140/proj2/demo/demo1/demo1
[3] DIR /media/sf_cs140/proj2/demo/demo2/demo2
[1] ENQUEUE /media/sf_cs140/proj2/demo/demo3/demo3
[6] DIR /media/sf_cs140/proj2/demo/demo3/demo3
[1] ABSENT /media/sf_cs140/proj2/demo/demo3/demo3.txt
[2] PRESENT /media/sf_cs140/proj2/demo/demo1/demo1.txt
[4] ABSENT /media/sf_cs140/proj2/demo/demo2/demo2.txt
[5] ABSENT /media/sf_cs140/proj2/demo/demo1/demo1/demo1.txt
[3] ABSENT /media/sf_cs140/proj2/demo/demo2/demo2/demo2.txt
[6] ABSENT /media/sf_cs140/proj2/demo/demo3/demo3/demo3.txt
● cs140@cs140:/media/sf_cs140/proj2$
```

Here is a screenshot of using valgrind showing that there are no memory leaks when using the code:



```
multithreaded.c - cs140221project2-c-ragunton - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
  CS140221PROJECT2-C-RAGUNTON
    .vscode
    demo
    test_dir
    testdir
    multithreaded
    multithreaded.c
    single.c
  ...
  C multithreaded.c x
    22
    23 //for using popen - to use grep command
    24 FILE *popen(const char *command, const char *mode);
    25 int pclose(FILE *stream);
    26
    27 //declaring the queue
    28 char *queue[100000];
    29 int Rear = - 1;
    30 int Front = - 1;
    31
    32 //for the multithreaded version
    33 struct args {
    34     char *exe;
    35     int ID;
    36     char *root;
    37     char *search;
    38 } args;
    39
    40 void *thread(void *arg)
    41 {
    42     struct args *a = (struct args *)arg;
    43     char *root = a->root;
    44     char *exe = a->exe;
    45     int ID = a->ID;
    46     char *search = a->search;
    47     DIR *dir = opendir(root);
    48     if (dir == NULL)
    49         return 0;
    50     struct dirent *ent;
    51     while ((ent = readdir(dir)) != NULL)
    52     {
    53         if (ent->d_type == DT_DIR)
    54         {
    55             char *new_root = (char *)malloc(strlen(root) + strlen(ent->d_name) + 1);
    56             strcpy(new_root, root);
    57             strcat(new_root, ent->d_name);
    58             if (strcmp(ent->d_name, ".") != 0 && strcmp(ent->d_name, "..") != 0)
    59             {
    60                 if (strcmp(ent->d_name, search) == 0)
    61                 {
    62                     char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    63                     strcpy(new_exe, exe);
    64                     strcat(new_exe, ent->d_name);
    65                     FILE *fp = popen(new_exe, "r");
    66                     if (fp != NULL)
    67                     {
    68                         char *line;
    69                         while ((line = fgets(line, sizeof(line), fp)) != NULL)
    70                         {
    71                             if (strcmp(line, search) == 0)
    72                             {
    73                                 char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    74                                 strcpy(new_queue, queue);
    75                                 strcat(new_queue, line);
    76                                 free(line);
    77                             }
    78                         }
    79                     }
    80                     free(new_exe);
    81                     free(new_root);
    82                 }
    83                 if (strcmp(ent->d_name, search) != 0)
    84                 {
    85                     char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    86                     strcpy(new_queue, queue);
    87                     strcat(new_queue, ent->d_name);
    88                     free(new_exe);
    89                     free(new_root);
    90                 }
    91             }
    92             if (strcmp(ent->d_name, search) == 0)
    93             {
    94                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    95                 strcpy(new_exe, exe);
    96                 strcat(new_exe, ent->d_name);
    97                 FILE *fp = popen(new_exe, "r");
    98                 if (fp != NULL)
    99                 {
    100                     char *line;
    101                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    102                     {
    103                         if (strcmp(line, search) == 0)
    104                         {
    105                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    106                             strcpy(new_queue, queue);
    107                             strcat(new_queue, line);
    108                             free(line);
    109                         }
    110                     }
    111                 }
    112                 free(new_exe);
    113                 free(new_root);
    114             }
    115             if (strcmp(ent->d_name, search) != 0)
    116             {
    117                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    118                 strcpy(new_queue, queue);
    119                 strcat(new_queue, ent->d_name);
    120                 free(new_exe);
    121                 free(new_root);
    122             }
    123             if (strcmp(ent->d_name, search) == 0)
    124             {
    125                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    126                 strcpy(new_exe, exe);
    127                 strcat(new_exe, ent->d_name);
    128                 FILE *fp = popen(new_exe, "r");
    129                 if (fp != NULL)
    130                 {
    131                     char *line;
    132                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    133                     {
    134                         if (strcmp(line, search) == 0)
    135                         {
    136                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    137                             strcpy(new_queue, queue);
    138                             strcat(new_queue, line);
    139                             free(line);
    140                         }
    141                     }
    142                 }
    143                 free(new_exe);
    144                 free(new_root);
    145             }
    146             if (strcmp(ent->d_name, search) != 0)
    147             {
    148                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    149                 strcpy(new_queue, queue);
    150                 strcat(new_queue, ent->d_name);
    151                 free(new_exe);
    152                 free(new_root);
    153             }
    154             if (strcmp(ent->d_name, search) == 0)
    155             {
    156                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    157                 strcpy(new_exe, exe);
    158                 strcat(new_exe, ent->d_name);
    159                 FILE *fp = popen(new_exe, "r");
    160                 if (fp != NULL)
    161                 {
    162                     char *line;
    163                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    164                     {
    165                         if (strcmp(line, search) == 0)
    166                         {
    167                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    168                             strcpy(new_queue, queue);
    169                             strcat(new_queue, line);
    170                             free(line);
    171                         }
    172                     }
    173                 }
    174                 free(new_exe);
    175                 free(new_root);
    176             }
    177             if (strcmp(ent->d_name, search) != 0)
    178             {
    179                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    180                 strcpy(new_queue, queue);
    181                 strcat(new_queue, ent->d_name);
    182                 free(new_exe);
    183                 free(new_root);
    184             }
    185             if (strcmp(ent->d_name, search) == 0)
    186             {
    187                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    188                 strcpy(new_exe, exe);
    189                 strcat(new_exe, ent->d_name);
    190                 FILE *fp = popen(new_exe, "r");
    191                 if (fp != NULL)
    192                 {
    193                     char *line;
    194                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    195                     {
    196                         if (strcmp(line, search) == 0)
    197                         {
    198                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    199                             strcpy(new_queue, queue);
    200                             strcat(new_queue, line);
    201                             free(line);
    202                         }
    203                     }
    204                 }
    205                 free(new_exe);
    206                 free(new_root);
    207             }
    208             if (strcmp(ent->d_name, search) != 0)
    209             {
    210                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    211                 strcpy(new_queue, queue);
    212                 strcat(new_queue, ent->d_name);
    213                 free(new_exe);
    214                 free(new_root);
    215             }
    216             if (strcmp(ent->d_name, search) == 0)
    217             {
    218                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    219                 strcpy(new_exe, exe);
    220                 strcat(new_exe, ent->d_name);
    221                 FILE *fp = popen(new_exe, "r");
    222                 if (fp != NULL)
    223                 {
    224                     char *line;
    225                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    226                     {
    227                         if (strcmp(line, search) == 0)
    228                         {
    229                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    230                             strcpy(new_queue, queue);
    231                             strcat(new_queue, line);
    232                             free(line);
    233                         }
    234                     }
    235                 }
    236                 free(new_exe);
    237                 free(new_root);
    238             }
    239             if (strcmp(ent->d_name, search) != 0)
    240             {
    241                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    242                 strcpy(new_queue, queue);
    243                 strcat(new_queue, ent->d_name);
    244                 free(new_exe);
    245                 free(new_root);
    246             }
    247             if (strcmp(ent->d_name, search) == 0)
    248             {
    249                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    250                 strcpy(new_exe, exe);
    251                 strcat(new_exe, ent->d_name);
    252                 FILE *fp = popen(new_exe, "r");
    253                 if (fp != NULL)
    254                 {
    255                     char *line;
    256                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    257                     {
    258                         if (strcmp(line, search) == 0)
    259                         {
    260                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    261                             strcpy(new_queue, queue);
    262                             strcat(new_queue, line);
    263                             free(line);
    264                         }
    265                     }
    266                 }
    267                 free(new_exe);
    268                 free(new_root);
    269             }
    270             if (strcmp(ent->d_name, search) != 0)
    271             {
    272                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    273                 strcpy(new_queue, queue);
    274                 strcat(new_queue, ent->d_name);
    275                 free(new_exe);
    276                 free(new_root);
    277             }
    278             if (strcmp(ent->d_name, search) == 0)
    279             {
    280                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    281                 strcpy(new_exe, exe);
    282                 strcat(new_exe, ent->d_name);
    283                 FILE *fp = popen(new_exe, "r");
    284                 if (fp != NULL)
    285                 {
    286                     char *line;
    287                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    288                     {
    289                         if (strcmp(line, search) == 0)
    290                         {
    291                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    292                             strcpy(new_queue, queue);
    293                             strcat(new_queue, line);
    294                             free(line);
    295                         }
    296                     }
    297                 }
    298                 free(new_exe);
    299                 free(new_root);
    300             }
    301             if (strcmp(ent->d_name, search) != 0)
    302             {
    303                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    304                 strcpy(new_queue, queue);
    305                 strcat(new_queue, ent->d_name);
    306                 free(new_exe);
    307                 free(new_root);
    308             }
    309             if (strcmp(ent->d_name, search) == 0)
    310             {
    311                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    312                 strcpy(new_exe, exe);
    313                 strcat(new_exe, ent->d_name);
    314                 FILE *fp = popen(new_exe, "r");
    315                 if (fp != NULL)
    316                 {
    317                     char *line;
    318                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    319                     {
    320                         if (strcmp(line, search) == 0)
    321                         {
    322                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    323                             strcpy(new_queue, queue);
    324                             strcat(new_queue, line);
    325                             free(line);
    326                         }
    327                     }
    328                 }
    329                 free(new_exe);
    330                 free(new_root);
    331             }
    332             if (strcmp(ent->d_name, search) != 0)
    333             {
    334                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    335                 strcpy(new_queue, queue);
    336                 strcat(new_queue, ent->d_name);
    337                 free(new_exe);
    338                 free(new_root);
    339             }
    340             if (strcmp(ent->d_name, search) == 0)
    341             {
    342                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    343                 strcpy(new_exe, exe);
    344                 strcat(new_exe, ent->d_name);
    345                 FILE *fp = popen(new_exe, "r");
    346                 if (fp != NULL)
    347                 {
    348                     char *line;
    349                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    350                     {
    351                         if (strcmp(line, search) == 0)
    352                         {
    353                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    354                             strcpy(new_queue, queue);
    355                             strcat(new_queue, line);
    356                             free(line);
    357                         }
    358                     }
    359                 }
    360                 free(new_exe);
    361                 free(new_root);
    362             }
    363             if (strcmp(ent->d_name, search) != 0)
    364             {
    365                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    366                 strcpy(new_queue, queue);
    367                 strcat(new_queue, ent->d_name);
    368                 free(new_exe);
    369                 free(new_root);
    370             }
    371             if (strcmp(ent->d_name, search) == 0)
    372             {
    373                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    374                 strcpy(new_exe, exe);
    375                 strcat(new_exe, ent->d_name);
    376                 FILE *fp = popen(new_exe, "r");
    377                 if (fp != NULL)
    378                 {
    379                     char *line;
    380                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    381                     {
    382                         if (strcmp(line, search) == 0)
    383                         {
    384                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    385                             strcpy(new_queue, queue);
    386                             strcat(new_queue, line);
    387                             free(line);
    388                         }
    389                     }
    390                 }
    391                 free(new_exe);
    392                 free(new_root);
    393             }
    394             if (strcmp(ent->d_name, search) != 0)
    395             {
    396                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    397                 strcpy(new_queue, queue);
    398                 strcat(new_queue, ent->d_name);
    399                 free(new_exe);
    400                 free(new_root);
    401             }
    402             if (strcmp(ent->d_name, search) == 0)
    403             {
    404                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    405                 strcpy(new_exe, exe);
    406                 strcat(new_exe, ent->d_name);
    407                 FILE *fp = popen(new_exe, "r");
    408                 if (fp != NULL)
    409                 {
    410                     char *line;
    411                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    412                     {
    413                         if (strcmp(line, search) == 0)
    414                         {
    415                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    416                             strcpy(new_queue, queue);
    417                             strcat(new_queue, line);
    418                             free(line);
    419                         }
    420                     }
    421                 }
    422                 free(new_exe);
    423                 free(new_root);
    424             }
    425             if (strcmp(ent->d_name, search) != 0)
    426             {
    427                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    428                 strcpy(new_queue, queue);
    429                 strcat(new_queue, ent->d_name);
    430                 free(new_exe);
    431                 free(new_root);
    432             }
    433             if (strcmp(ent->d_name, search) == 0)
    434             {
    435                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    436                 strcpy(new_exe, exe);
    437                 strcat(new_exe, ent->d_name);
    438                 FILE *fp = popen(new_exe, "r");
    439                 if (fp != NULL)
    440                 {
    441                     char *line;
    442                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    443                     {
    444                         if (strcmp(line, search) == 0)
    445                         {
    446                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    447                             strcpy(new_queue, queue);
    448                             strcat(new_queue, line);
    449                             free(line);
    450                         }
    451                     }
    452                 }
    453                 free(new_exe);
    454                 free(new_root);
    455             }
    456             if (strcmp(ent->d_name, search) != 0)
    457             {
    458                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    459                 strcpy(new_queue, queue);
    460                 strcat(new_queue, ent->d_name);
    461                 free(new_exe);
    462                 free(new_root);
    463             }
    464             if (strcmp(ent->d_name, search) == 0)
    465             {
    466                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    467                 strcpy(new_exe, exe);
    468                 strcat(new_exe, ent->d_name);
    469                 FILE *fp = popen(new_exe, "r");
    470                 if (fp != NULL)
    471                 {
    472                     char *line;
    473                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    474                     {
    475                         if (strcmp(line, search) == 0)
    476                         {
    477                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    478                             strcpy(new_queue, queue);
    479                             strcat(new_queue, line);
    480                             free(line);
    481                         }
    482                     }
    483                 }
    484                 free(new_exe);
    485                 free(new_root);
    486             }
    487             if (strcmp(ent->d_name, search) != 0)
    488             {
    489                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    490                 strcpy(new_queue, queue);
    491                 strcat(new_queue, ent->d_name);
    492                 free(new_exe);
    493                 free(new_root);
    494             }
    495             if (strcmp(ent->d_name, search) == 0)
    496             {
    497                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    498                 strcpy(new_exe, exe);
    499                 strcat(new_exe, ent->d_name);
    500                 FILE *fp = popen(new_exe, "r");
    501                 if (fp != NULL)
    502                 {
    503                     char *line;
    504                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    505                     {
    506                         if (strcmp(line, search) == 0)
    507                         {
    508                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    509                             strcpy(new_queue, queue);
    510                             strcat(new_queue, line);
    511                             free(line);
    512                         }
    513                     }
    514                 }
    515                 free(new_exe);
    516                 free(new_root);
    517             }
    518             if (strcmp(ent->d_name, search) != 0)
    519             {
    520                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    521                 strcpy(new_queue, queue);
    522                 strcat(new_queue, ent->d_name);
    523                 free(new_exe);
    524                 free(new_root);
    525             }
    526             if (strcmp(ent->d_name, search) == 0)
    527             {
    528                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    529                 strcpy(new_exe, exe);
    530                 strcat(new_exe, ent->d_name);
    531                 FILE *fp = popen(new_exe, "r");
    532                 if (fp != NULL)
    533                 {
    534                     char *line;
    535                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    536                     {
    537                         if (strcmp(line, search) == 0)
    538                         {
    539                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    540                             strcpy(new_queue, queue);
    541                             strcat(new_queue, line);
    542                             free(line);
    543                         }
    544                     }
    545                 }
    546                 free(new_exe);
    547                 free(new_root);
    548             }
    549             if (strcmp(ent->d_name, search) != 0)
    550             {
    551                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    552                 strcpy(new_queue, queue);
    553                 strcat(new_queue, ent->d_name);
    554                 free(new_exe);
    555                 free(new_root);
    556             }
    557             if (strcmp(ent->d_name, search) == 0)
    558             {
    559                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    560                 strcpy(new_exe, exe);
    561                 strcat(new_exe, ent->d_name);
    562                 FILE *fp = popen(new_exe, "r");
    563                 if (fp != NULL)
    564                 {
    565                     char *line;
    566                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    567                     {
    568                         if (strcmp(line, search) == 0)
    569                         {
    570                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    571                             strcpy(new_queue, queue);
    572                             strcat(new_queue, line);
    573                             free(line);
    574                         }
    575                     }
    576                 }
    577                 free(new_exe);
    578                 free(new_root);
    579             }
    580             if (strcmp(ent->d_name, search) != 0)
    581             {
    582                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    583                 strcpy(new_queue, queue);
    584                 strcat(new_queue, ent->d_name);
    585                 free(new_exe);
    586                 free(new_root);
    587             }
    588             if (strcmp(ent->d_name, search) == 0)
    589             {
    590                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    591                 strcpy(new_exe, exe);
    592                 strcat(new_exe, ent->d_name);
    593                 FILE *fp = popen(new_exe, "r");
    594                 if (fp != NULL)
    595                 {
    596                     char *line;
    597                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    598                     {
    599                         if (strcmp(line, search) == 0)
    600                         {
    601                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    602                             strcpy(new_queue, queue);
    603                             strcat(new_queue, line);
    604                             free(line);
    605                         }
    606                     }
    607                 }
    608                 free(new_exe);
    609                 free(new_root);
    610             }
    611             if (strcmp(ent->d_name, search) != 0)
    612             {
    613                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    614                 strcpy(new_queue, queue);
    615                 strcat(new_queue, ent->d_name);
    616                 free(new_exe);
    617                 free(new_root);
    618             }
    619             if (strcmp(ent->d_name, search) == 0)
    620             {
    621                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    622                 strcpy(new_exe, exe);
    623                 strcat(new_exe, ent->d_name);
    624                 FILE *fp = popen(new_exe, "r");
    625                 if (fp != NULL)
    626                 {
    627                     char *line;
    628                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    629                     {
    630                         if (strcmp(line, search) == 0)
    631                         {
    632                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    633                             strcpy(new_queue, queue);
    634                             strcat(new_queue, line);
    635                             free(line);
    636                         }
    637                     }
    638                 }
    639                 free(new_exe);
    640                 free(new_root);
    641             }
    642             if (strcmp(ent->d_name, search) != 0)
    643             {
    644                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    645                 strcpy(new_queue, queue);
    646                 strcat(new_queue, ent->d_name);
    647                 free(new_exe);
    648                 free(new_root);
    649             }
    650             if (strcmp(ent->d_name, search) == 0)
    651             {
    652                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    653                 strcpy(new_exe, exe);
    654                 strcat(new_exe, ent->d_name);
    655                 FILE *fp = popen(new_exe, "r");
    656                 if (fp != NULL)
    657                 {
    658                     char *line;
    659                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    660                     {
    661                         if (strcmp(line, search) == 0)
    662                         {
    663                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    664                             strcpy(new_queue, queue);
    665                             strcat(new_queue, line);
    666                             free(line);
    667                         }
    668                     }
    669                 }
    670                 free(new_exe);
    671                 free(new_root);
    672             }
    673             if (strcmp(ent->d_name, search) != 0)
    674             {
    675                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    676                 strcpy(new_queue, queue);
    677                 strcat(new_queue, ent->d_name);
    678                 free(new_exe);
    679                 free(new_root);
    680             }
    681             if (strcmp(ent->d_name, search) == 0)
    682             {
    683                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    684                 strcpy(new_exe, exe);
    685                 strcat(new_exe, ent->d_name);
    686                 FILE *fp = popen(new_exe, "r");
    687                 if (fp != NULL)
    688                 {
    689                     char *line;
    690                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    691                     {
    692                         if (strcmp(line, search) == 0)
    693                         {
    694                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    695                             strcpy(new_queue, queue);
    696                             strcat(new_queue, line);
    697                             free(line);
    698                         }
    699                     }
    700                 }
    701                 free(new_exe);
    702                 free(new_root);
    703             }
    704             if (strcmp(ent->d_name, search) != 0)
    705             {
    706                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    707                 strcpy(new_queue, queue);
    708                 strcat(new_queue, ent->d_name);
    709                 free(new_exe);
    710                 free(new_root);
    711             }
    712             if (strcmp(ent->d_name, search) == 0)
    713             {
    714                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    715                 strcpy(new_exe, exe);
    716                 strcat(new_exe, ent->d_name);
    717                 FILE *fp = popen(new_exe, "r");
    718                 if (fp != NULL)
    719                 {
    720                     char *line;
    721                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    722                     {
    723                         if (strcmp(line, search) == 0)
    724                         {
    725                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    726                             strcpy(new_queue, queue);
    727                             strcat(new_queue, line);
    728                             free(line);
    729                         }
    730                     }
    731                 }
    732                 free(new_exe);
    733                 free(new_root);
    734             }
    735             if (strcmp(ent->d_name, search) != 0)
    736             {
    737                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    738                 strcpy(new_queue, queue);
    739                 strcat(new_queue, ent->d_name);
    740                 free(new_exe);
    741                 free(new_root);
    742             }
    743             if (strcmp(ent->d_name, search) == 0)
    744             {
    745                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    746                 strcpy(new_exe, exe);
    747                 strcat(new_exe, ent->d_name);
    748                 FILE *fp = popen(new_exe, "r");
    749                 if (fp != NULL)
    750                 {
    751                     char *line;
    752                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    753                     {
    754                         if (strcmp(line, search) == 0)
    755                         {
    756                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    757                             strcpy(new_queue, queue);
    758                             strcat(new_queue, line);
    759                             free(line);
    760                         }
    761                     }
    762                 }
    763                 free(new_exe);
    764                 free(new_root);
    765             }
    766             if (strcmp(ent->d_name, search) != 0)
    767             {
    768                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    769                 strcpy(new_queue, queue);
    770                 strcat(new_queue, ent->d_name);
    771                 free(new_exe);
    772                 free(new_root);
    773             }
    774             if (strcmp(ent->d_name, search) == 0)
    775             {
    776                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    777                 strcpy(new_exe, exe);
    778                 strcat(new_exe, ent->d_name);
    779                 FILE *fp = popen(new_exe, "r");
    780                 if (fp != NULL)
    781                 {
    782                     char *line;
    783                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    784                     {
    785                         if (strcmp(line, search) == 0)
    786                         {
    787                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    788                             strcpy(new_queue, queue);
    789                             strcat(new_queue, line);
    790                             free(line);
    791                         }
    792                     }
    793                 }
    794                 free(new_exe);
    795                 free(new_root);
    796             }
    797             if (strcmp(ent->d_name, search) != 0)
    798             {
    799                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    800                 strcpy(new_queue, queue);
    801                 strcat(new_queue, ent->d_name);
    802                 free(new_exe);
    803                 free(new_root);
    804             }
    805             if (strcmp(ent->d_name, search) == 0)
    806             {
    807                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    808                 strcpy(new_exe, exe);
    809                 strcat(new_exe, ent->d_name);
    810                 FILE *fp = popen(new_exe, "r");
    811                 if (fp != NULL)
    812                 {
    813                     char *line;
    814                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    815                     {
    816                         if (strcmp(line, search) == 0)
    817                         {
    818                             char *new_queue = (char *)malloc(strlen(queue) + strlen(line) + 1);
    819                             strcpy(new_queue, queue);
    820                             strcat(new_queue, line);
    821                             free(line);
    822                         }
    823                     }
    824                 }
    825                 free(new_exe);
    826                 free(new_root);
    827             }
    828             if (strcmp(ent->d_name, search) != 0)
    829             {
    830                 char *new_queue = (char *)malloc(strlen(queue) + strlen(ent->d_name) + 1);
    831                 strcpy(new_queue, queue);
    832                 strcat(new_queue, ent->d_name);
    833                 free(new_exe);
    834                 free(new_root);
    835             }
    836             if (strcmp(ent->d_name, search) == 0)
    837             {
    838                 char *new_exe = (char *)malloc(strlen(exe) + strlen(ent->d_name) + 1);
    839                 strcpy(new_exe, exe);
    840                 strcat(new_exe, ent->d_name);
    841                 FILE *fp = popen(new_exe, "r");
    842                 if (fp != NULL)
    843                 {
    844                     char *line;
    845                     while ((line = fgets(line, sizeof(line), fp)) != NULL)
    846                     {
    847                         if (strcmp(line, search) == 0)
    848                         {
    
```


Here is another one with valgrind but with the test case earlier:

The image shows a Visual Studio Code editor window with a C++ project named 'multithreaded.c'. The code implements a multi-threaded program using pthreads. It defines a queue of 100,000 items and a worker function that processes items from the queue. The program is compiled and run, and the terminal output shows the program running successfully, with memory usage and heap summary information.

```

File Edit Selection View Go Run Terminal Help

EXPLORER
C: \multithreaded\c
└─ CS140221PROJECT2-C-RAGUNTON
    ├── .vscode
    ├── demo
    ├── test_dir
    ├── testdir
    ├── multithreaded
    ├── multithreaded.c
    └── single.c

22
23 //for using popen - to use grep command
24 FILE *popen(const char *command, const char *mode);
25 int pclose(FILE *stream);
26
27 //declaring the queue
28 char *queue[100000];
29 int Rear = - 1;
30 int Front = - 1;
31
32 //for the multithreaded version
33 struct args {
34     char *exe;
35     int ID;
36     char *root;
37     char *search;
38 }
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
```

2. b.

For the race conditions, I used a mutex lock.

```
pthread_mutex_t lock; //for locking critical sections
```

By using a mutex lock, no 2 threads can access sections that must not be used simultaneously.

Here are all the critical sections that I used locks on.

```
pthread_mutex_lock(&lock);  
sleeping[arg->ID] = 1; //the current thread will sleep  
currsleep++;  
pthread_mutex_unlock(&lock);
```

```
pthread_mutex_lock(&lock);  
sleeping[arg->ID]==0; //the current thread wakes up  
currsleep--;  
pthread_mutex_unlock(&lock);
```

```
pthread_mutex_lock(&lock);  
letin[arg->ID]=0;  
strcpy(current_file,queue[Front]); //get the first item/directory in the queue  
dequeue();  
printf("[%d] DIR %s\n",arg->ID,current_file);  
pthread_mutex_unlock(&lock);
```

```
pthread_mutex_lock(&lock);  
enqueue(deepcheckfile);  
  
for (int j=0; j<8; j++){ //after an enqueue, refresh letin to let the threads come in again  
    letin[j]=1;  
}  
  
printf("[%d] ENQUEUE %s\n",arg->ID,checkfile);  
sem_post(&queueunlock);  
  
pthread_mutex_unlock(&lock);
```

```
pthread_mutex_lock(&lock); //there is nothing to do left  
fin = 1;  
pthread_mutex_unlock(&lock);
```

Note that most of the parts protected by these locks are those which are involved with global variables/arrays. Only one thread can access these parts at a time. Thus, there will be no race conditions and the queue and all the global variables/arrays will be synchronized for all the threads.

```
pthread_mutex_destroy(&lock);
```

In the end, we destroy the lock.

2.c.

The idea behind the termination has already been explained but I will explain again with more details.

```
void *work(struct args *arg){
    DIR *dir;
    struct dirent *dp;

    while (Front<=Rear || letin[arg->ID] == 1){ //stop when there are nothing to do left; after every enqueue, threads can come again
        pthread_mutex_lock(&lock);
        sleeping[arg->ID] = 1; //the current thread will sleep
        currsleep++;
        pthread_mutex_unlock(&lock);

        if (currsleep==((struct args*)arg)->N)sem_post(&queueunlock); //force a thread to wake up

        else if (currsleep<((struct args*)arg)->N) sem_wait(&queueunlock); //makes the worker sleep if queue is empty

        pthread_mutex_lock(&lock);
        sleeping[arg->ID]=0; //the current thread wakes up
        currsleep--;
        pthread_mutex_unlock(&lock);
    }
}
```

Note that all the created threads will be lead in this while loop. If there is no letin, the entry of the threads will solely be based on the condition `Front<=Rear`. But that queue is handled by different threads. The queue may seem empty but in reality, they are just being used by other threads and will still enqueue new directories. But without letin, the threads does not know this. A thread may appear here thinking that the queue is already empty.

We prevent that using letin. If a thread's letin in the global array is 1, they can enter the while loop even if `Front>Rear`.

```
for (int j=0; j<8; j++){
    letin[j]=1;
    sleeping[j]=0;
}
```

```
pthread_mutex_lock(&lock);
enqueue(deepcheckfile);

for (int j=0; j<8; j++){ //after an enqueue, refresh letin to let the threads come in again
    letin[j]=1;
}
```

There are two parts wherein we set the thread's letin to 1. At start, we initialize all of them to 1. The first time a thread enters the while loop earlier, they will always go through it.

The second one is when an enqueue happens. If something is enqueued, it means that the queue has a new entry and will need a thread to handle it. Thus, letin of all threads will be set to 1 to let them enter the loop.

Once the while loop does not work for a thread, that thread will go out of the while loop leading into:

```

    }
    pthread_mutex_lock(&lock); //there is nothing to do left
    fin = 1;
    pthread_mutex_unlock(&lock);
    sem_post(&queueunlock); //wakes up every sleeping process and force them to end
}

```

Here, the global variable `fin` will be set into 1. Because it is synchronized, all the threads will see this change. The current thread will wake up a sleeping thread using `sem_post`.

```

while (Front<=Rear || letin[arg->ID] == 1) //stop when there are nothing to do left; after every enqueue, threads can come again
{
    pthread_mutex_lock(&lock);
    sleeping[arg->ID] = 1; //the current thread will sleep
    currsleep++;
    pthread_mutex_unlock(&lock);

    if (currsleep==((struct args*)arg)->N) sem_post(&queueunlock); //force a thread to wake up

    else if (currsleep<((struct args*)arg)->N) sem_wait(&queueunlock); //makes the worker sleep if queue is empty

    pthread_mutex_lock(&lock);
    sleeping[arg->ID]=0; //the current thread wakes up
    currsleep--;
    pthread_mutex_unlock(&lock);

    if (fin == 1) break; //there is nothing to do left
}

```

Note that all of the sleeping threads will be found on the else if statement here. If they wake up, they will eventually go into the last if statement in the screenshot. That if statement will be true and the thread will go out of the while loop.

```

    }
    pthread_mutex_lock(&lock); //there is nothing to do left
    fin = 1;
    pthread_mutex_unlock(&lock);
    sem_post(&queueunlock); //wakes up every sleeping process and force them to end
}

```

Again, that thread will go here repeating the process until there are no sleeping threads.

For threads that are not asleep when this is happening and are still going into the while loop, they will still stop. That is because of the if statement about `fin`. All the threads will be caught by it to make them all exit the while loop to return to the main function. That is how this code terminates the workers. For the race conditions, they are already included in the locks mentioned in 2.b.