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.

```c
#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.

```c
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.

```c
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 prupose 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.

```c
//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.

```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.

```c
//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.

```c
//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.

```c
//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.

```c
//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(&queuelock); //force a thread to wake up

        else if (currsleep<((struct args*)arg)->N) sem_wait(&queuelock); //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, the 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(&queuelock);

    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.

```c
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(&queuelock); //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(&queuelock); //force a thread to wake up

    else if (currsleep<((struct args*)arg)->N) sem_wait(&queuelock); //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
```

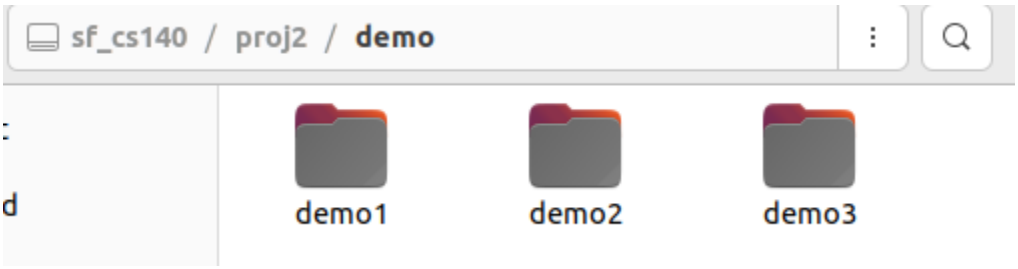
Going back to the part earlier, a thread will wake up and will see that fin==1. Thus, the thread that woke up will go out of the while loop where another sem_wait awaits. It will wake up another process and the loop will continue until all the threads have woken and stopped.

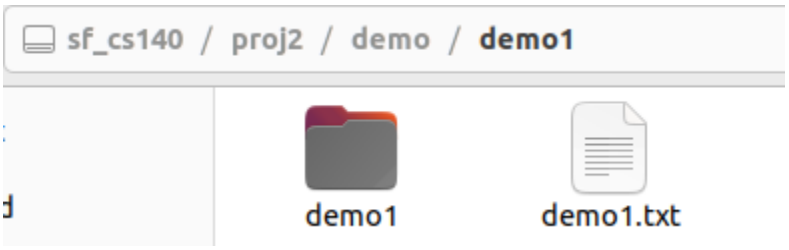
```
    for (int i=0;i<N;i++){
        pthread_join(tid[i], NULL);
    }

    for (int k = 0; k<100000; k++){
        free(queue[k]);
    }

    pthread_mutex_destroy(&lock);

    return;
}
```

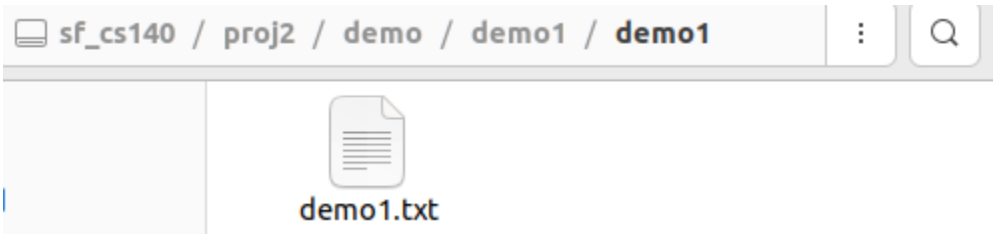
This is back on the main function. The threads will be caught using pthread_join. Then we free all the spaces taken up by the queue. Then, destrop the mutex lock and return.

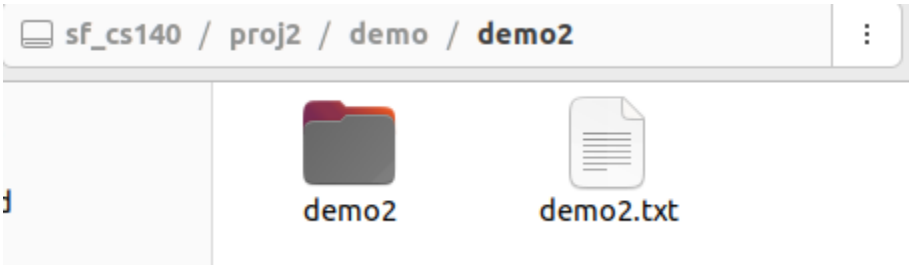
Now that we know how the code works, we can proceed to the demo with N=2, 1PRESENT, 5ABSENT, and 6DIRs.

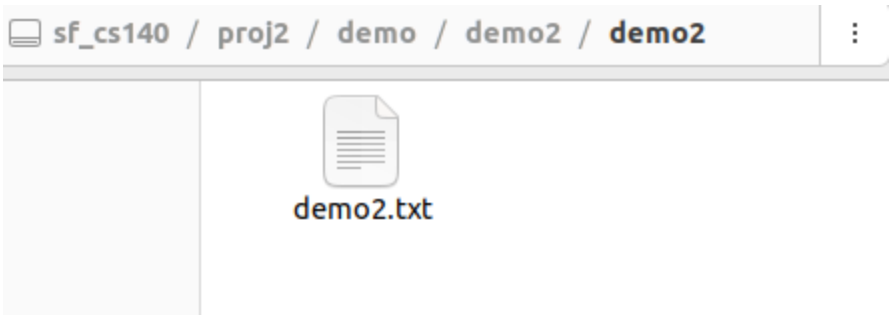
The rootpath will be demo inside /media/sf_cs140/proj2.

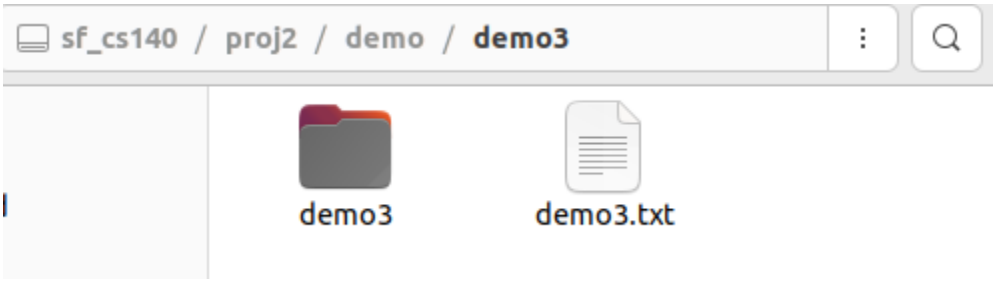


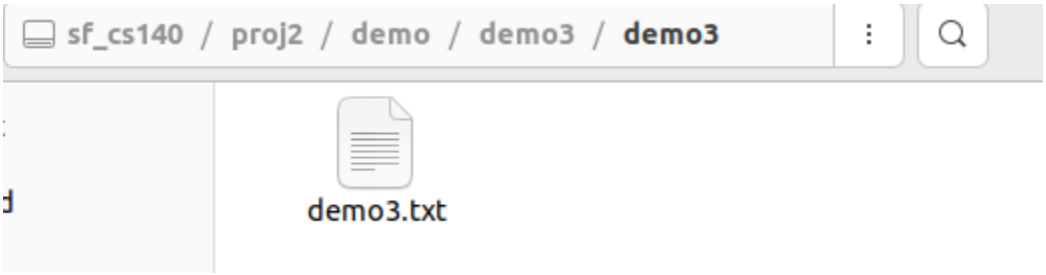










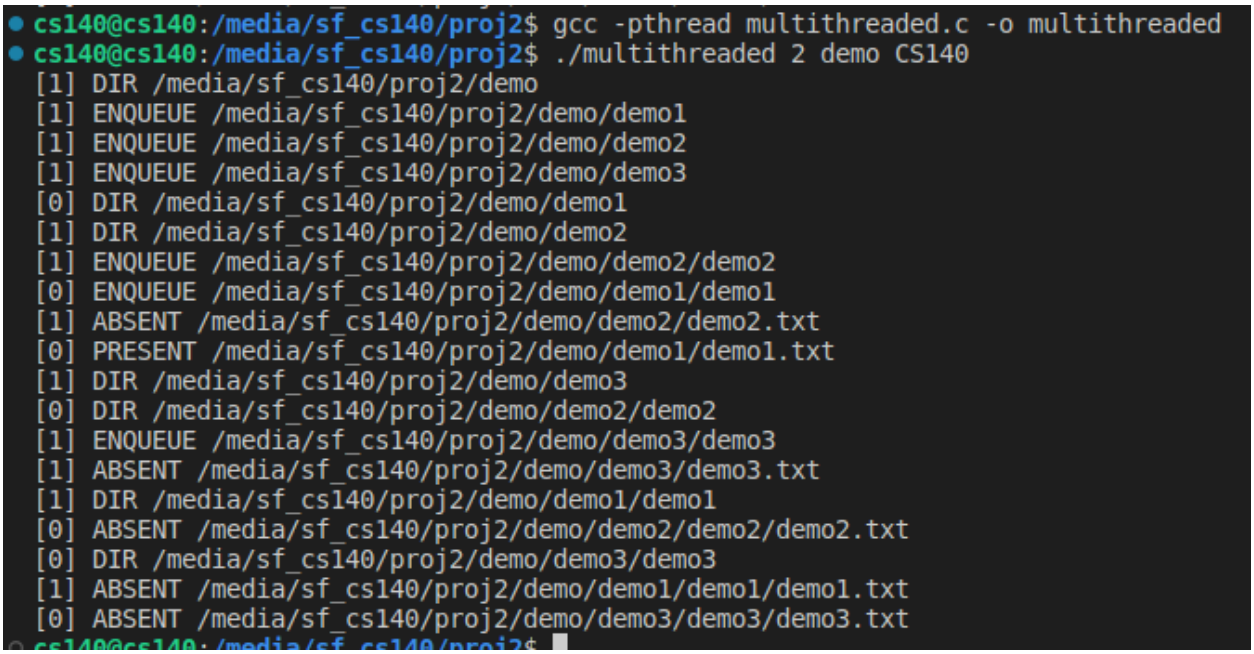


We are going to make the searched for text be “CS140”. Out of all the .txt files shown in the pictures above, only one contains “CS140”. That is the demo1.txt inside /media/sf_cs140/proj2/demo/demo1.

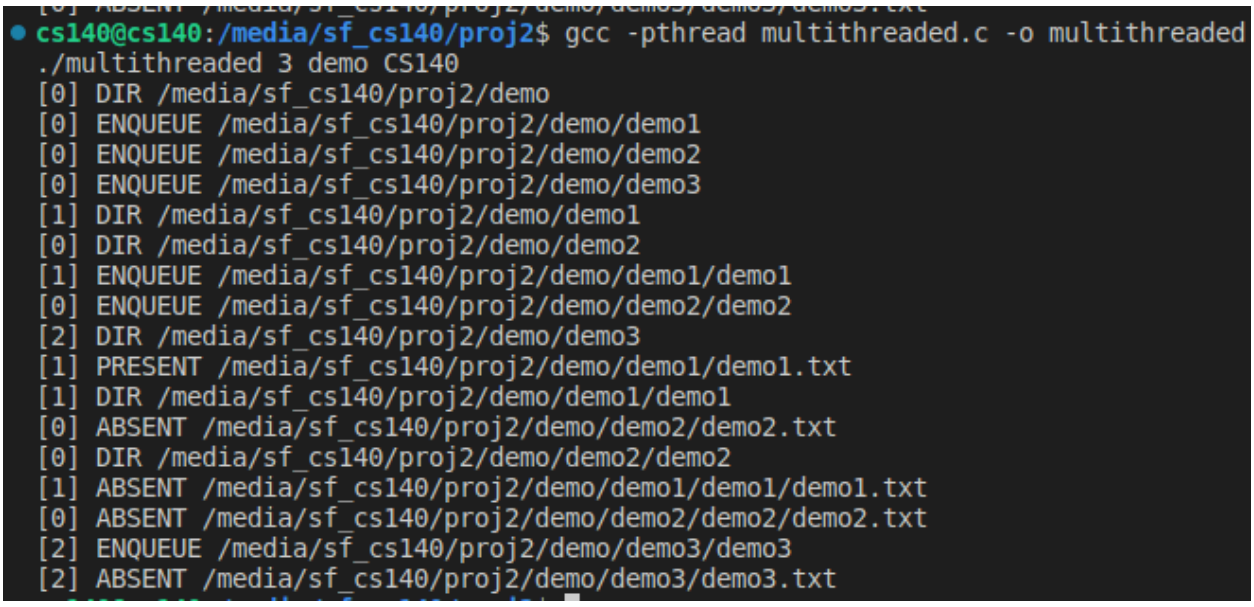
If we compile with “gcc -pthread multithreaded.c -o multithreaded”.

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

The result will be:



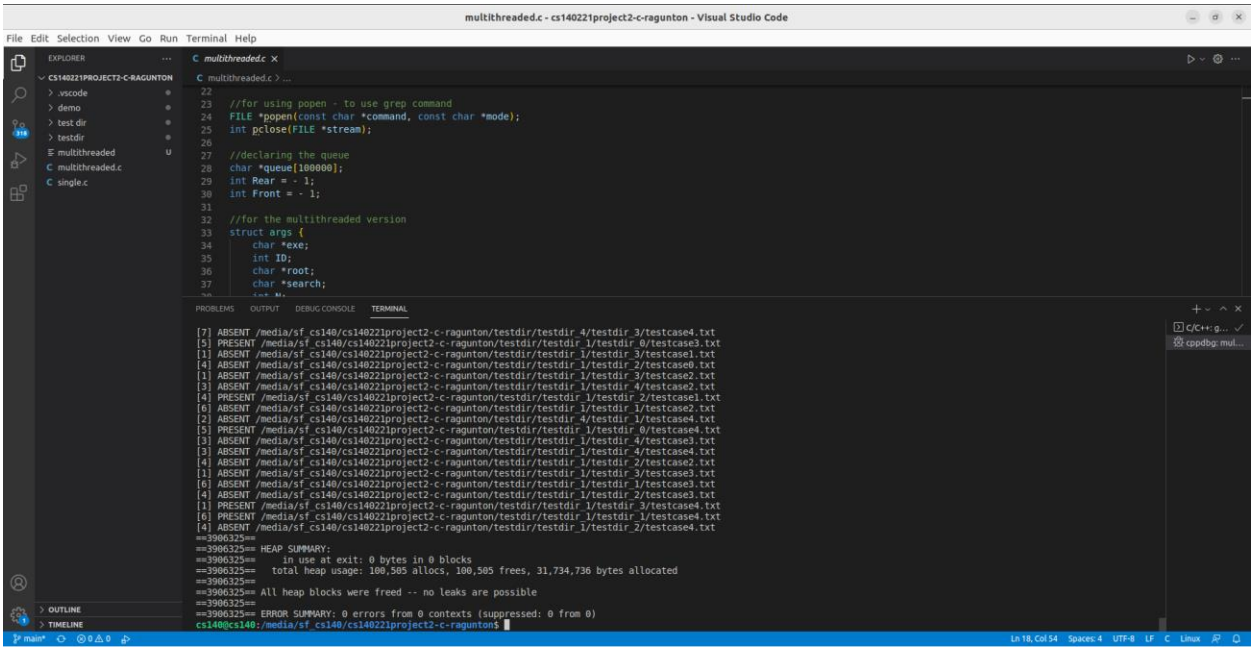
If N=3:



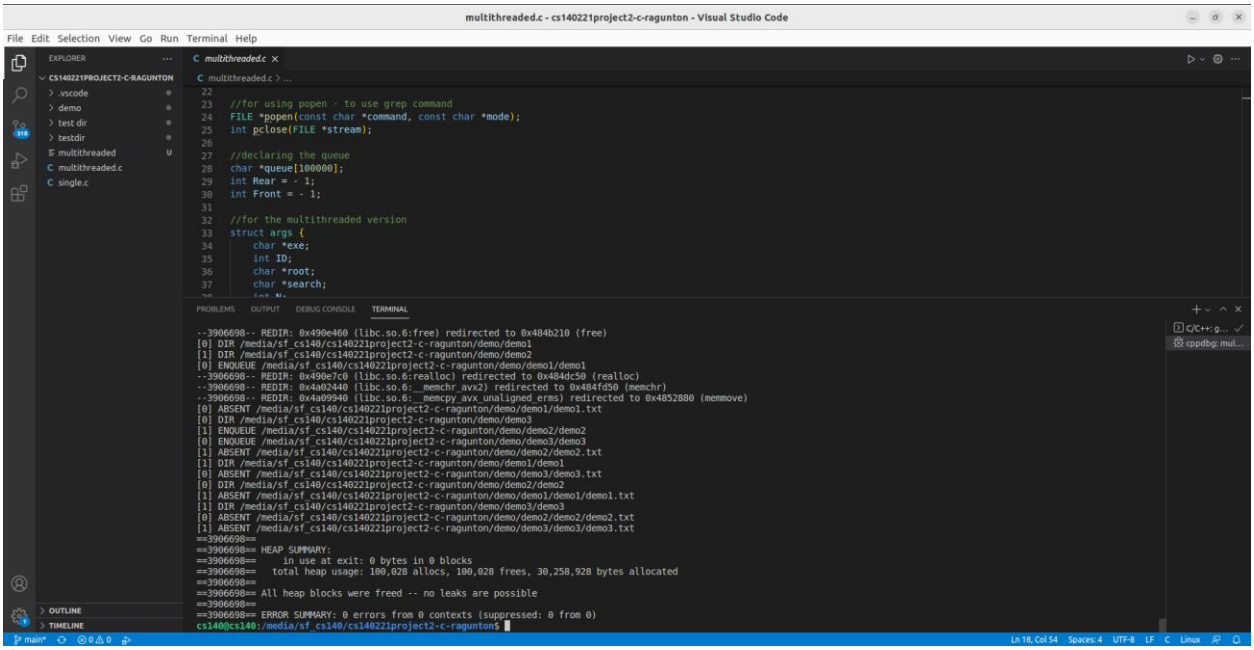
If N=8:

```
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
```

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



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



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(&queuelock);

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(&queuelock); //force a thread to wake up

        else if (currsleep<((struct args*)arg)->N) sem_wait(&queuelock); //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(&queuelock); //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(&queuelock); //force a thread to wake up

    else if (currsleep<((struct args*)arg)->N) sem_wait(&queuelock); //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(&queuelock); //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.