CS 140 Project 2
Audrey Vianca Maegan Sy
2020-11654
LAB-4

Documentation Video: https://drive.google.com/file/d/1LO38tzgRFEJqrJ5l3rrrFkdSo0iXRIiH/view?usp=sharing

1. All references used with purpose specified (*if any; otherwise, explicitly state that you did not use any external resource*)
    a. Queue of strings in C – Used as a basis for the task queue's pointer array implementation
    b. Passing an integer as last argument of pthread_create – Used as a basis in implementing the TID of the threads
    c. string.h in C – Used as a reference in using strcpy(), strcat() and strcmp(), which were functions frequently used in assessing and manipulating the paths
    d. readdir - Linux manual page – Used as a reference for the correct usage of readdir(), which was used in traversing through the directories
    e. C program to list all files in a directory recursively – Used as a basis for the directory traversal implementation, also referenced for the correct usage of opendir()
    f. realpath - Linux manual page – Used as a reference for the correct usage of realpath(), which was used in obtaining the absolute paths of the found directories and files to be traversed or checked
    g. stdlib.h in C – Used as a reference for the correct usage of system()

2. For each version submitted (excluding single-process, single-threaded version):
    a. Walkthrough of code execution with sample run having at least N = 2 on a multicore machine, one PRESENT, five ABSENTs, and six DIRs
    For example, the following commands were run:

```
cs140@cs140:/media/sf_CS140_P2$ gcc multithreaded.c -pthread -o proj2
cs140@cs140:/media/sf_CS140_P2$ ./proj2 8 testdir test
```

    Where multithreaded.c was compiled as proj2, and executed with three arguments—8 corresponding to the number of threads, testdir being the root directory to traverse, and "test" being the search string.

    Upon starting the program, it will first extract the information from the arguments and allocate them accordingly, int NTHREADS being set to the number of threads, the inputted root directory is set up as rootpath, and the search string is copied to the global variable search_string. The initial values of the pointer array implementation of the task queue are then initialized, and rootpath''s absolute path is obtained using realpath() before being enqueued.

```c
int main(int argc, char *argv[]) {
    // Initialize values from input arguments
    NTHREADS = strtol(argv[1], NULL, 10);
    char rootpath[250];
    strcpy(rootpath, argv[2]);
    strcpy(search_string, argv[3]);

    // Initialize queue pointer array
    q = malloc(sizeof(struct queue));
    q->head = NULL;
    q->tail = NULL;

    // Set first task in queue as rootpath and enqueue it
    char first[250];
    realpath(rootpath, first);
    enqueue(first);
```

The specified 8 threads are then initialized with a for loop, where they are set to do the function find_grep() with an argument arg that takes note of the order they were initialized at (int i), which serves as its TID.

```
    pthread_t thread[NTHREADS];

    // Initialize threads
    for (int i = 0; i < NTHREADS; i++) {
        int *arg = malloc(sizeof(*arg));
        *arg = i; // to take note of i as the TID (helps keep track of the order of initialization)
        pthread_create(&thread[i], NULL, (void *) find_grep, arg);
    }

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

Upon entering find_grep(), the thread's TID will be set as it is needed for printing information. Next, an infinite while loop is used to keep the threads looping in the function until there is no more work for them to do. A mutex lock is then acquired by a thread, and first goes through a conditional while loop that will see if there are tasks in the queue (represented by the value check, which is set to 1 at the beginning of the program since rootpath is the first task in queue), if there are no available tasks in the queue it will increment the integer waiting, which takes note of the number of threads waiting, and if there are still other active threads then it will be made to wait and it will give up the lock.

```
// Function entered by threads, continually checks for queued paths to traverse
// and files to perform 'grep' on
void find_grep(void *id) {
    // Set thread's TID based on their order of initialization
    int TID = *((int *) id);
    free(id);

    while (1){
        // acquire lock to dequeue a task
        pthread_mutex_lock(&lock);

        // Check for available tasks, if none, enter while loop
        while (check < 1) {
            waiting++; // increment to note that a thread is waiting

            // If all the threads are waiting, there will be nothing left to queue
            // so we exit the loop
            if (waiting >= NTHREADS){
                pthread_mutex_unlock(&lock); // give up lock before leaving
                goto end;
            }

            // Otherwise, make thread wait and give up lock
            pthread_cond_wait(&cond, &lock);
        }
```

CS 140 Project 2
Audrey Vianca Maegan Sy
2020-11654
LAB-4

Once it passes this loop, that means there was an available task and it can now dequeue it. Since this current thread had the lock, it can freely dequeue it without worrying about another thread possibly dequeuing it and accidentally working on the same directory. Dequeuing this task will have the thread obtain a path and store it in p. check is also decremented at this time to indicate that there is one less task available, before printing the “[n] DIR path” statement. The lock can now be given up to a different thread who can look for a task, as the current thread can work on its locally dequeued task.

```
        // Dequeue task and place new path in p
        char p[250];
        strcpy(p, dequeue());
        check--; // Decrement check to note that there is one less task
        printf("[%d] DIR %s\n", TID, p);

        // Give up lock to let other threads check for tasks
        pthread_mutex_unlock(&lock);
```

The thread can now proceed with traversing through its found directory, by first preparing the directory by using opendir() on the path p from earlier, and the absolute path of this found task is set up using realpath(), before ‘/’ is appended to it in preparation for the absolute paths of directories/files to be found. readdir() is then called repeatedly until no more files are in the current directory, and it checks for directories based on the d_type outputted by readdir() (ignoring . and ..). If a directory is found, it can now append the directory’s name to the path set up earlier to serve as its absolute path. The lock then needs to be acquired as we will be manipulating the queue and we do not want to enqueue something at the same time as a different thread. We then enqueue the found absolute path, increment check to indicate that a new task is available, and print the needed “[n] ENQUEUE path” statement. It then checks if there are any threads waiting, and if there is, waiting will be decremented as pthread_cond_signal() is used to signal a thread to wake one up. The lock is then given up for any other threads to need to access the queue.

CS 140 Project 2
Audrey Vianca Maegan Sy
2020-11654
LAB-4

```c
    // Directory traversal
    struct dirent *dp;
    DIR *dir = opendir(p);

    // Set up absolute path of dequeued path, add '/' at the end in
    // preparation of concatenating the newly found files/directories
    char first[250], slash[2];
    realpath(p, first);
    strcpy(slash, "/");
    strcat(first, slash);

    while ((dp = readdir(dir)) != NULL){
        char buffer[250];
        // If directory is found...
        if (dp->d_type == DT_DIR && strcmp(dp->d_name, ".") != 0 && strcmp(dp->d_name, "..") != 0){
            strcpy(buffer, first);
            strcat(buffer, dp->d_name); // Add the newly found directory's name to the path

            pthread_mutex_lock(&lock);

            enqueue(buffer); // Enqueue the new absolute path
            check++; // Increment to note that one new task is available
            printf("[%d] ENQUEUE %s\n", TID, buffer);

            // If there is a thread waiting, decrement the number of
            // waiting threads and wake one up
            if (waiting > 0) {
                waiting--;
                pthread_cond_signal(&cond);
            }
            pthread_mutex_unlock(&lock);
```

If a file is found instead, its name can then be appended to the path set up earlier, to obtain the file's absolute path. The command to be sent through system() is then set up through a series of strcpy and strcat's, which would ultimately result in a command of the form "grep -c [search_string] [absolute path aka. buffer] > /dev/null" so that we may execute grep without printing its output. The output of the system() call is then sent to the integer ret. We now acquire the lock to ensure that the printing of the statement goes smoothly. Depending on the output of system(), if ret == 0 (grep succeeded) then "[n] PRESENT path" will be printed. Otherwise, "[n] ABSENT path" gets printed. The lock can now be given up for other uses. When the directory traversal ends, the directory can now be closed.

```c
        // If a file is ofund...
        } else if (strcmp(dp->d_name, ".") != 0 && strcmp(dp->d_name, "..") != 0){
            strcpy(buffer, first);
            strcat(buffer, dp->d_name); // Add the newly found file's name to the path

            // Set up command to send to system by
            // preparing and concatenating the needed strings
            char space[2], command1[20], command2[20], tempbuffer[250];
            strcpy(space, " ");
            strcpy(command1, "grep -c ");
            strcpy(command2, " > /dev/null");
            strcpy(tempbuffer, buffer);

            strcat(command1, search_string);
            strcat(command1, space);
            strcat(tempbuffer, command2);
            strcat(command1, tempbuffer);

            // Execute command and check if it succeeded
            int ret = system(command1);
            pthread_mutex_lock(&lock);

            // Print "PRESENT" if grep succeeded, "ABSENT" if not
            if (ret == 0) printf("[%d] PRESENT %s\n", TID, buffer);
            else printf("[%d] ABSENT %s\n", TID, buffer);

            pthread_mutex_unlock(&lock);
        }
    }
    closedir(dir);
```

CS 140 Project 2
Audrey Vianca Maegan Sy
2020-11654
LAB-4

This process (Look for task-> dequeue found directory path-> traverse through directory, enqueueing found directories and using grep on files) repeats until all of the directories and files under rootpath are checked, and it ends once waiting is equal to the number of threads (meaning that all threads are waiting and nothing more will get enqueued.

```
        // If all the threads are waiting, there will be nothing left to queue
        // so we exit the loop
        if (waiting >= NTHREADS){
            pthread_mutex_unlock(&lock); // give up lock before leaving
            goto end;
        }
```

In that case, last thread that entered the waiting loop (which wasn't put to sleep since it knows it's the last thread awake) will give up the lock before going to "end" found at the end of the function.

```
    // Goes here when all threads are waiting, wake up others to exit
    end:
    pthread_cond_broadcast(&cond);

}
```

Here, it wakes up all the other threads so that they may see that all the threads are now waiting, and then can also skip to end to exit the infinite while loop, as well as find_grep(). Once all the threads have ended, the process can end and destroy the lock.

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

    pthread_mutex_destroy(&lock);

    return 0;
}
```

CS 140 Project 2
Audrey Vianca Maegan Sy
2020-11654
LAB-4

When the command in the beginning was executed, it looks like the following:

```
cs140@cs140:/media/sf_CS140_P2$ gcc multithreaded.c -pthread -o proj2
cs140@cs140:/media/sf_CS140_P2$ ./proj2 8 testdir test
[3] DIR /media/sf_CS140_P2/testdir
[3] ABSENT /media/sf_CS140_P2/testdir/testabsent1.TXT
[3] ENQUEUE /media/sf_CS140_P2/testdir/testdir2
[3] ENQUEUE /media/sf_CS140_P2/testdir/testdir3
[6] DIR /media/sf_CS140_P2/testdir/testdir2
[2] DIR /media/sf_CS140_P2/testdir/testdir3
[2] ABSENT /media/sf_CS140_P2/testdir/testdir3/testabsent2.TXT
[2] ENQUEUE /media/sf_CS140_P2/testdir/testdir3/testdir5
[2] ENQUEUE /media/sf_CS140_P2/testdir/testdir3/testdir6
[7] DIR /media/sf_CS140_P2/testdir/testdir3/testdir5
[2] DIR /media/sf_CS140_P2/testdir/testdir3/testdir6
[3] PRESENT /media/sf_CS140_P2/testdir/testpresent.TXT
[6] ABSENT /media/sf_CS140_P2/testdir/testdir2/testabsent3.TXT
[2] ABSENT /media/sf_CS140_P2/testdir/testdir3/testdir6/testabsent5.TXT
[6] ABSENT /media/sf_CS140_P2/testdir/testdir2/testabsent4.TXT
[6] ENQUEUE /media/sf_CS140_P2/testdir/testdir2/testdir4
[0] DIR /media/sf_CS140_P2/testdir/testdir2/testdir4
cs140@cs140:/media/sf_CS140_P2$
```

Going through what was printed, we can deduce the following flow:

- After the rootpath testdir is turned into its rootpath /media/sf_CS140_P2/testdir, it gets enqueued first and thread 3 is the one that is able to obtain the lock first and dequeue it.
- Thread 3 is then able to go through directory traversal first, since there are no other threads working on a task yet, so it checks the file testabsent1.txt and finds the word “test” to be absent. It also finds two directories testdir2 and testdir3, and enqueues their absolute paths.
- Thread 6 is now able to acquire the lock and dequeue the directory /media/sf_CS140_P2/testdir/testdir2 which was found earlier
- Thread 2 is now able to acquire the lock and dequeue the directory /media/sf_CS140_P2/testdir/testdir3 which was found earlier
- Thread 2 is able to print its traversal findings first, checking the file testabsent2.txt and displaying that grep didn’t find anything, and also enqueuing the two directories found, testdir5 and testdir6
- Thread 7 is now able to acquire the lock and dequeue the directory /media/sf_CS140_P2/testdir/testdir3/testdir5 which was found earlier
- Thread 2, which is the same thread that found this directory, is now able to acquire the lock and dequeue the directory /media/sf_CS140_P2/testdir/testdir3/testdir6 which was found earlier
- Thread 3 is only now able to continue printing its findings from the first directory earlier, finding a file testpresent.txt and displaying that it found a file where grep succeeded.
- Thread 6 continues working on its directory from earlier, checking the file testabsent4, and finding a directory to enqueue, testdir4
- Thread 0 is now able to dequeue the directory /media/sf_CS140_P2/testdir/testdir2/testdir4, but since it is empty and there are no other threads available, the program ends.

CS 140 Project 2
Audrey Vianca Maegan Sy
2020-11654
LAB-4

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

```c
// Serves as a node in the pointer array that queues the tasks
struct task {
    char curpath[250]; // Takes note of the path to be traversed once the task is chosen
    struct task *next;
};
// Facilitates the queueing of the pointer array
struct queue {
    struct task *head; // Takes note of the task in front of the queue
    struct task *tail; // Takes note of the task at the back of the queue
}*q;
// Enqueues new paths as tasks
void enqueue(char p[250]) {
    struct task *t = malloc(sizeof(struct task));
    strcpy(t->curpath, p);
    t->next = NULL;
    if (q->head == NULL && q->tail == NULL) {
        q->head = t;
        q->tail = t;
    } else {
        q->tail->next = t;
        q->tail = t;
    }
}
// Dequeues path to be traversed
char *dequeue() {
    if (q->head != NULL) {
        struct task *t = malloc(sizeof(struct task));
        t = q->head;
        if (q->head == q->tail) q->tail = NULL;
        q->head = q->head->next;
        char *p = malloc(sizeof (char)*250);
        strcpy(p, t->curpath);
        free(t);
        return p;
    } else {
        return NULL;
    }
}
```

The task queue was implemented as a basic form of pointer array, which enqueues and stores the path of the directory to be traversed. It also employs the general scheme of enqueueing and dequeuing of a pointer array (each task being a node that stores the path of a directory, making use of a head and a tail, changing the head's/tail's values when a process is enqueued or dequeued, etc.). With the modification that dequeue also outputs the path p stored in the task's curpath.

This queue is initialized at the beginning of the main function, before any threads are created

```c
    // Initialize queue pointer array
    q = malloc(sizeof(struct queue));
    q->head = NULL;
    q->tail = NULL;
```

Afterwards, the first enqueueing is done to the rootpath.

```c
    // Set first task in queue as rootpath and enqueue it
    char first[250];
    realpath(rootpath, first);
    enqueue(first);
```

Note that this is the only instance of enqueuing that isn't inside a lock, because at this point the threads haven't been created yet and there is no chance of a race condition happening.

For the other instances of enqueueing and dequeuing, they are protected by the mutex lock, as errors will occur in the cases that multiple threads dequeue the same task (repeating the same process unnecessarily), or enqueue different tasks at the same time (overwriting one or more of the tasks, which ends up with us not including some files and directories when checking), or trying to dequeue and enqueue at the same time (ruining the ordering of the queue).

```c
    while (1){
        // acquire lock to dequeue a task
        pthread_mutex_lock(&lock);

        // Check for available tasks, if none, enter while loop
        while (check < 1) {
            waiting++; // increment to note that a thread is waiting

            // If all the threads are waiting, there will be nothing left to queue
            // so we exit the loop
            if (waiting >= NTHREADS){
                pthread_mutex_unlock(&lock); // give up lock before leaving
                goto end;
            }

            // Otherwise, make thread wait and give up lock
            pthread_cond_wait(&cond, &lock);
        }

        // Dequeue task and place new path in p
        char p[250];
        strcpy(p, dequeue());
        check--; // Decrement check to note that there is one less task
        printf("[%d] DIR %s\n", TID, p);

        // Give up lock to let other threads check for tasks
        pthread_mutex_unlock(&lock);
```

CS 140 Project 2
Audrey Vianca Maegan Sy
2020-11654
LAB-4

```
pthread_mutex_lock(&lock);

enqueue(buffer); // Enqueue the new absolute path
check++; // Increment to note that one new task is available
printf("[%d] ENQUEUE %s\n", TID, buffer);

// If there is a thread waiting, decrement the number of
// waiting threads and wake one up
if (waiting > 0) {
    waiting--;
    pthread_cond_signal(&cond);
}
pthread_mutex_unlock(&lock);
```

CS 140 Project 2
Audrey Vianca Maegan Sy
2020-11654
LAB-4

c. Explanation of how each worker knows when to terminate (i.e., mechanism that determines and synchronizes that no more content will be enqueued); include how race conditions were handled

In order to know if there are no more tasks to be enqueued, the while loop at the beginning of the find_grep() function was used to keep track of how many threads are currently waiting. This was done using the global variable “waiting” that was set at the beginning of the program. This integer increments before a thread is set to wait when there are no tasks available to run yet, and decrements when a task gets enqueued. In the case where the last active thread enters this loop, waiting will increment and it will be seen that no more threads will be able to enqueue new tasks. Here, instead of being set to wait like the other threads, it will give up its lock before going to “end” at the last part of the find_grep() function. In end, it will signal the other threads to wake up. When they do, they will each acquire the lock, be placed in the same while loop where they will also be able to give up its lock and go to end. This is so that the threads will be able to each exit the infinite while loop and end their execution.

```c
    while (1){
        // acquire lock to dequeue a task
        pthread_mutex_lock(&lock);

        // Check for available tasks, if none, enter while loop
        while (check < 1) {
            waiting++; // increment to note that a thread is waiting

            // If all the threads are waiting, there will be nothing left to queue
            // so we exit the loop
            if (waiting >= NTHREADS){
                pthread_mutex_unlock(&lock); // give up lock before leaving
                goto end;
            }

            // Otherwise, make thread wait and give up lock
            pthread_cond_wait(&cond, &lock);
        }
```

```c
    // Goes here when all threads are waiting, wake up others to exit
    end:
    pthread_cond_broadcast(&cond);

}
```

Note that this operation is enclosed in a lock, only giving it up after operating on waiting, making sure that only one thread operates on it at a time. This is so that the global value of waiting is consistent with the exact number of inactive threads, and so that we accurately know the program needs to end as there are no more tasks available, and we avoid the possibility of the threads looping infinitely. Also note that the thread needs to give up its lock manually before exiting the loop, as it will not pass through pthread_cond_wait() that will give up the lock on its own, and we avoid a deadlock where the last thread exits on its own without giving up the lock, leaving the other threads unable to wake up.