Name: Yenzy Urson S. Hebron
Date: January 14, 2023
Lab Instructor: Sir Juan Felipe Coronel
Subject: CS 140 LAB - 4

Project 2: Parallelized **grep** Runner

Google Drive Link for the Video Documentation
https://drive.google.com/file/d/1TQsPHLUj58J7ZiazSH22b_iuG5ZbK7XZ/view?usp=share_link

Multithreaded Version

1. References

- OSTEP (Ch. 29): The chapter's threadsafe implementation of a dynamic queue served as a guide for my own implementation which utilizes a heap-based unbounded buffer for the task queue.
- OSTEP (Ch. 30): The chapter's discussion on the producer-consumer bounded buffer problem served as guide for my use of locks and condition variables for thread synchronization.
- OSTEP (Ch. 39): This chapter helped me understand directory traversal and the handling of absolute and relative paths.
- https://stackoverflow.com/questions/3736320/executing-shell-script-with-system-returns-256-what-does-that-mean : For why we should divide the return value of system() by 256.
- https://unix.stackexchange.com/questions/119648/redirecting-to-dev-null : For how to redirect to /dev/null.
- https://stackoverflow.com/questions/4553012/checking-if-a-file-is-a-directory-or-just-a-file : For how to check if a "file" is a regular file or a directory. This is where I got the idea of the isDir function from.

2. Discussion

a. Walkthrough of Code Execution

Please read Item b and Item c before proceeding with this item.

As always, we begin with main:

We start by preprocessing the arguments by either copying them to a local variable or having a local pointer to them.

```
int main(int argc, char *argv[]) {
    // argv preprocessing
    int N = atoi(argv[1]);
    // strncpy is sane because len(relpaths) < len(abspaths) <= MAXPATH (\0 
inclusive)
    char rootpath[MAXPATH]; strncpy(rootpath, argv[2], MAXPATH);
    char *search_string = argv[3];
    ...
}
```

Afterwards we initialize the locks and the condition variables that we will be using and also allocate memory for the threads and their `data`.

```
int main(int argc, char *argv[]) {
```

```
    ...
    // Initializations
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);
    pthread_cond_init(&fill, NULL);

    pthread_t thread[8];
    struct thread_data data[8];
    ...
}
```

Then we also initialize the task queue (further explained in Item b):

```
int main(int argc, char *argv[]) {
    ...
    // Init task queue and store first task (rootpath) into it
    Queue_Init(&task_queue);
    // IDEA: Only work with abs path, convert supplied rel path to abs path
    // If first char of rootpath is '/', it is an abs path, else it is rel path
    count++;
    if (rootpath[0] == '/') {
        Queue_Enqueue(&task_queue, rootpath);
    } else {
        char origcwd[MAXPATH];
        getcwd(origcwd, MAXPATH);
        chdir(rootpath);
        getcwd(rootpath, MAXPATH);
        chdir(origcwd);
        Queue_Enqueue(&task_queue, rootpath);
    }
    ...
}
```

Note that if the rootpath is an absolute path (starts with a '/'), we immediately enqueue it to the task queue, and if it is a relative path, we first convert it to an absolute path before enqueuing it to the task queue.

Converting a relative path to an absolute path (without the use of realpath) entails switching to that relative path using chdir then obtaining the absolute path of that directory using getcwd (which always returns a null-terminated string corresponding to the absolute path of the current working directory).

Now that the first task is enqueued to the task queue, we can then launch the threads. Before each thread is launched, we must first supply the thread with its tid (for later printing) and the search string it will use (common among all threads). This is also passed to the `routine` they will execute as an argument as a pointer to their parent's thread local storage (where `thread_data` can be found).

```
int main(int argc, char *argv[]) {
    ...
    // Launch Threads
    busyThreads = N;    // determines when thread exits
    for (int i = 0; i < N; i++) {
        data[i].tid = i;
        data[i].search_string = search_string;
        pthread_create(&thread[i], NULL, (void *) routine, &data[i]);
    }

    // Wait for Threads
    for (int i = 0; i < N; i++) {
        pthread_join(thread[i], NULL);
    }

    // Memory housekeeping
    // 1. Confirm task queue is empty
    assert(Queue_Is_Empty(&task_queue));
    // 2. Then, free up dummy node remaining inside task queue
    free(task_queue.head);
    pthread_mutex_destroy(&task_queue.headLock);
    pthread_mutex_destroy(&task_queue.tailLock);

    pthread_mutex_destroy(&lock1); pthread_mutex_destroy(&lock2);
    pthread_cond_destroy(&fill);

    return 0;
}
```

After launching its child threads, the main thread then waits for them complete using pthread_join(). And once they are all complete, the main thread then performs some memory housekeeping before fully terminating the program. This means freeing up the dummy node and destroying the headLock and the tatilLock of the task queue, and also destroying the other locks and condition variable.

We now look inside the worker thread `routine`:

A thread immediately enters an infinite while loop that will only be broken once all the threads are non-busy (as explained in Item c).

```
void routine(struct thread_data *data) {
    while (1) {
        ...
    }
}
```

The critical sections inside this routine are explained in Item b and Item c. We only touch on them here abstractly. The next section we discuss is the critical section corresponding to the case when a thread acts as a consumer (as explained in Item b):

```
void routine(struct thread_data *data) {
    while (1) {
        pthread_mutex_lock(&lock1);
        while (count == 0) {
            busyThreads--;
            if (busyThreads == 0) {       // wait for "signal 2"
                pthread_cond_signal(&fill);
                pthread_mutex_unlock(&lock1);
                return;
            }
            pthread_cond_wait(&fill, &lock1);        // wait for signal 1
            busyThreads++;
        }

        // Task can be successfully dequeued (preempt count--)
        count--;
        pthread_mutex_unlock(&lock1);
        ...
    }
}
```

After entering the infinite loop, the thread then enters a critical section defined by `lock1` where it mutually-exclusively checks if there are tasks to be dequeued. If none, it enters the while (count == 0) loop and marks itself non-busy, and if it further finds out that all the other threads are already non-busy, the thread will then know to terminate but not before waking up another sleeping non-busy thread. If there are still busy threads, then the thread will spin, waiting for the busy threads to produce (enqueue) something for the spin-waiting thread to consume (dequeue). Note here that every time a thread is about to go to sleep or possibly exit, it marks itself as non-busy, and every time a thread wakes up, it marks itself as busy (please see Item c for why).

Then if the thread finds out that count != 0, it can exit the while loop, pre-emptively decrementing count before exiting the critical section to tell other threads that are about to enter or are inside the critical section that there is one less task in the task queue.

The thread then dequeues a task from the task queue and stores it inside taskpath. This is also the time when a thread prints out a DIR corresponding to said dequeue.

```
void routine(struct thread_data *data) {
    while (1) {
        ...
        char taskpath[MAXPATH];
```

```
        Queue_Dequeue(&task_queue, taskpath);   // act as a consumer
        // No need to wake up thread sleeping on fill, because thread atm is a consumer (dequeues).
        // Do wake-up on fill once thread is a consumer (enqueues).

        printf("[%d] DIR %s\n", data->tid, taskpath);
        ...
    }
}
```

Afterwards, the thread then enters another critical section defined by lock2. Here, the critical section is more concerned with the sensitivity of certain operations to changes in the current working directory, which is a construct that is shared between threads of the same process, and hence has to be protected by locks.

```
void routine(struct thread_data *data) {
    while (1) {
        ...
        // IDEA: Current Working Directory is also shared between threads
        // Fix race condition here when it comes to changing directories
        pthread_mutex_lock(&lock2);
        chdir(taskpath);
        DIR *dirp = opendir(".");
        // If realpath() is not available
        char pathbuff[MAXPATH];   // use with constructing abspath corr to next lower dir or a regular file in curr dir
        char origcwd[MAXPATH];
        getcwd(origcwd, MAXPATH);  // store orig
        pthread_mutex_unlock(&lock2);
        ...
    }
}
```

What happens here is that the thread switches its current working directory to that denoted by taskpath using chdir, and then stores a DIR pointer dirp to that taskpath using opendir. Note that dirp is now a local variable. Also note that MAXPATH = 250 as provided by the project guide. We may also let it to be greater than 250 as explained in Item b.

Now, pathbuff will be used later on for constructing the next absolute path to be enqueued in the task queue and printed with ENQUEUE for child directories or printed with PRESENT or ABSENT for child files. And origcwd (storing a clean copy of the absolute path of the current working directory) will be used as the base to which the name of a child directory or child file of the current working directory will be appended to in order to construct their absolute path.

Afterwards, we dynamically allocate grepbuff which will be potentially used for executing grep on a file using the system() function. BUFFSIZE works with grepbuff, wherein it anticipates the max length of abspath + variable length of search_string + the 25 aux chars in grepbuff.

And then we traverse the current working directory using a loop as provided by OSTEP (Ch. 39) with the help of readdir(dirp) storing it in a struct dirent pointer d (d will mainly be used for its d->d_name).

These are shown below:

```
void routine(struct thread_data *data) {
    while (1) {
        ...
        int BUFFSIZE = MAXPATH + strlen(data->search_string) + 25;
        char *grepbuff = (char *) malloc(sizeof(char) * (BUFFSIZE));

        struct dirent *d;
        // readdir here operates on clean dirp which was locked previously
        // This makes the lines below threadsafe
        while ((d = readdir(dirp)) != NULL) {   // loop until end of dir stream
is reached
            ...
            }
        }
        free(grepbuff); // don't forget to free grepbuff
        closedir(dirp); // don't forget to close directory
    }
}
```

Note that we also do not forget to free up the heap memory allocated for grepbuff after it’s use is done.

Now, as we traverse the directory stream (wherein we ignore the self . and parent .. directories), looping until the end of the stream is reached, we construct the absolute path of the next child directory (for enqueue) or regular file (for grep execution) using string manipulation with the help of strncpy, strcat, and strncat. After these sequence of operations, we now have the absolute path of the current object in the directory stream stored in pathbuff.

This is made up of the concatenated pathbuff = origcwd + / + d->d_name. These are shown below:

```
void routine(struct thread_data *data) {
    while (1) {
        ...
        while ((d = readdir(dirp)) != NULL) {
            // Ignore self . and parent .. directories
            if (strncmp(d->d_name, ".", MAXPATH) == 0 || strncmp(d->d_name,
            "..", MAXPATH) == 0) {
```

```
                continue;
            }

            // Prepare abspath of next lower dir or regular file
             strncpy(pathbuff, origcwd, MAXPATH);  // revert pathbuff to clean
             copy of cwd
            strcat(pathbuff, "/");
            strncat(pathbuff, d->d_name, MAXPATH);
            ...
        }
        ...
    }
}
```

Finally, with pathbuff containing the absolute path of the current object being worked on inside the current working directory, we first check if the object is a directory or a regular file using the following utility function

```
int isDir(const char *path) {
    struct stat path_stat;
    stat(path, &path_stat);
    return S_ISDIR(path_stat.st_mode);
}
```

And then with that information at hand, we either:

- Work on the object as a directory, wherein we enqueue pathbuff into the task queue (the thread is now a producer), print ENQUEUE, and then enter the same critical section as with the consumer threads (defined by lock1) to increment the number of tasks inside the task queue and wake-up a sleeping consumer on fill.
- Or work on the object as a file, wherein with the help of snprintf and grepbuff, we construct the command that we will execute as we would have done on the shell, and then check the return value of that execution (return or exit code of system() which has to be divided by 256 according to one of our references) if the search string is present or absent on the file.

These are shown below:

```
void routine(struct thread_data *data) {
    while (1) {
        ...
        while ((d = readdir(dirp)) != NULL) {   // loop until end of dir stream
is reached
            ...
            // Do work on task.
```

```
            // Use isDir with abspath pathbuff, not relpath d->d_name to avoid 
chdir()-related race conditions!
            if (isDir(pathbuff)) {
                // We now have true concurrency of Enqueue and Dequeue operations
                Queue_Enqueue(&task_queue, pathbuff);   // act as a producer
                printf("[%d] ENQUEUE %s\n", data->tid, pathbuff);
                pthread_mutex_lock(&lock1);
                count++;
                pthread_cond_signal(&fill);
                pthread_mutex_unlock(&lock1);

            } else {
                // Search using absolute path!
                // Don't use relative path d->d_name as it's vulnerable to change 
in cwd.
                // NOTE: grep "search_string" "file_name"
                snprintf(grepbuff, BUFFSIZE, "grep \"%s\" \"%s\" > /dev/null", 
data->search_string, pathbuff);
                if (DEBUG) printf("[%d] grepbuff after snprintf: \"%s\"\n", data-
>tid, grepbuff);
                int check = system(grepbuff)/256;   // execute grep (issue with 
system() ret so divide by 256)
                if (check == 0) {
                    printf("[%d] PRESENT %s\n", data->tid, pathbuff);
                } else if (check == 1) {
                    printf("[%d] ABSENT %s\n", data->tid, pathbuff);
                } else {
                    printf("[%d] ERROR %s\n", data->tid, pathbuff); // should NOT 
appear
                }
            }
        }
        ...
    }
}
```

And after the directory stream dirp is exhausted, the thread begins the routine (the loop) all over again.

A sample run is now presented below:

First, we note that the absolute path of the working directory from which we compile and execute `multithreaded.c` is /media/sf_project2/cs140221project2-y-hebron.

The virtual machine we will use is multicore (4 cores).

Our test case is the following directory given by `tree` which it says has 7 subdirectories and 11 files:

```
cs140@cs140:/media/sf_project2/cs140221project2-y-hebron$ tree testdir
testdir
├── aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
│   └── a
├── aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaz
├── aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.txt
├── aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab.txt
├── tc1
│   ├── nested1
│   │   ├── hello.txt
│   │   └── nested11
│   │       ├── absent1.txt
│   │       └── hello.txt
│   └── tc1.1.txt
└── tc2
    ├── absent2.txt
    ├── nested2
    │   └── hello.txt
    ├── tc2.1.txt
    └── tc2.2.txt

7 directories, 11 files
```

Note that the absolute path of testdir is /media/sf_project2/cs140221project2-y-hebron/testdir. This serves as our `rootpath`. The inclusion of the files and directories with very long absolute paths (maxing out the 250 character limit provided by the project guide, null-terminator inclusive) serves as edge cases on string handling.

Now, we will `grep` for the search string "hello" using 4 threads. That is, the command we will execute is,

./multithreaded 4 testdir hello

We expect hello to be present in the files named hello.txt, as we've made the test directory so that those are the only files that contain hello.

Observe that we are giving "testdir" as a relative root path.

Executing the command, we get:

```
cs140@cs140:/media/sf_project2/cs140221project2-y-hebron$ gcc multithreaded.c -pthread -o multithreaded
cs140@cs140:/media/sf_project2/cs140221project2-y-hebron$ ./multithreaded 4 testdir hello
[0] DIR /media/sf_project2/cs140221project2-y-hebron/testdir
[0] ABSENT /media/sf_project2/cs140221project2-y-hebron/testdir/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.txt
[0] ENQUEUE /media/sf_project2/cs140221project2-y-hebron/testdir/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
[1] DIR /media/sf_project2/cs140221project2-y-hebron/testdir/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
[0] ENQUEUE /media/sf_project2/cs140221project2-y-hebron/testdir/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaz
[2] DIR /media/sf_project2/cs140221project2-y-hebron/testdir/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaz
[0] ABSENT /media/sf_project2/cs140221project2-y-hebron/testdir/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab.txt
[0] ENQUEUE /media/sf_project2/cs140221project2-y-hebron/testdir/tc1
[1] ABSENT /media/sf_project2/cs140221project2-y-hebron/testdir/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa/a
[1] DIR /media/sf_project2/cs140221project2-y-hebron/testdir/tc1
[0] ENQUEUE /media/sf_project2/cs140221project2-y-hebron/testdir/tc2
[1] ENQUEUE /media/sf_project2/cs140221project2-y-hebron/testdir/tc1/nested1
[2] DIR /media/sf_project2/cs140221project2-y-hebron/testdir/tc2
[0] DIR /media/sf_project2/cs140221project2-y-hebron/testdir/tc1/nested1
[1] ABSENT /media/sf_project2/cs140221project2-y-hebron/testdir/tc1/tc1.1.txt
[2] ABSENT /media/sf_project2/cs140221project2-y-hebron/testdir/tc2/absent2.txt
[0] PRESENT /media/sf_project2/cs140221project2-y-hebron/testdir/tc1/nested1/hello.txt
[2] ENQUEUE /media/sf_project2/cs140221project2-y-hebron/testdir/tc2/nested2
[0] ENQUEUE /media/sf_project2/cs140221project2-y-hebron/testdir/tc1/nested1/nested11
[3] DIR /media/sf_project2/cs140221project2-y-hebron/testdir/tc2/nested2
[1] DIR /media/sf_project2/cs140221project2-y-hebron/testdir/tc1/nested1/nested11
[1] ABSENT /media/sf_project2/cs140221project2-y-hebron/testdir/tc1/nested1/nested11/absent1.txt
[3] PRESENT /media/sf_project2/cs140221project2-y-hebron/testdir/tc2/nested2/hello.txt
[2] ABSENT /media/sf_project2/cs140221project2-y-hebron/testdir/tc2/tc2.1.txt
[1] PRESENT /media/sf_project2/cs140221project2-y-hebron/testdir/tc1/nested1/nested11/hello.txt
[2] ABSENT /media/sf_project2/cs140221project2-y-hebron/testdir/tc2/tc2.2.txt
cs140@cs140:/media/sf_project2/cs140221project2-y-hebron$ |
```

And as we could see, if we count the number of PRESENT and ABSENT, there are 11 in total, which is correct. And if we count the number of DIR and ENQUEUE, there are 15 in total, which is correct (14/2 subdirectories, + 1 DIR for the rootpath directory).

We can also see that the stress test has been passed on the large file names, and that there are no duplicated tasks.

Hence we can say that this is a correct execution.

Also, as required, this is run with at least N = 2 on a multicore machine, with the output yielding at least one PRESENT, five ABSENTs, and six DIRs.

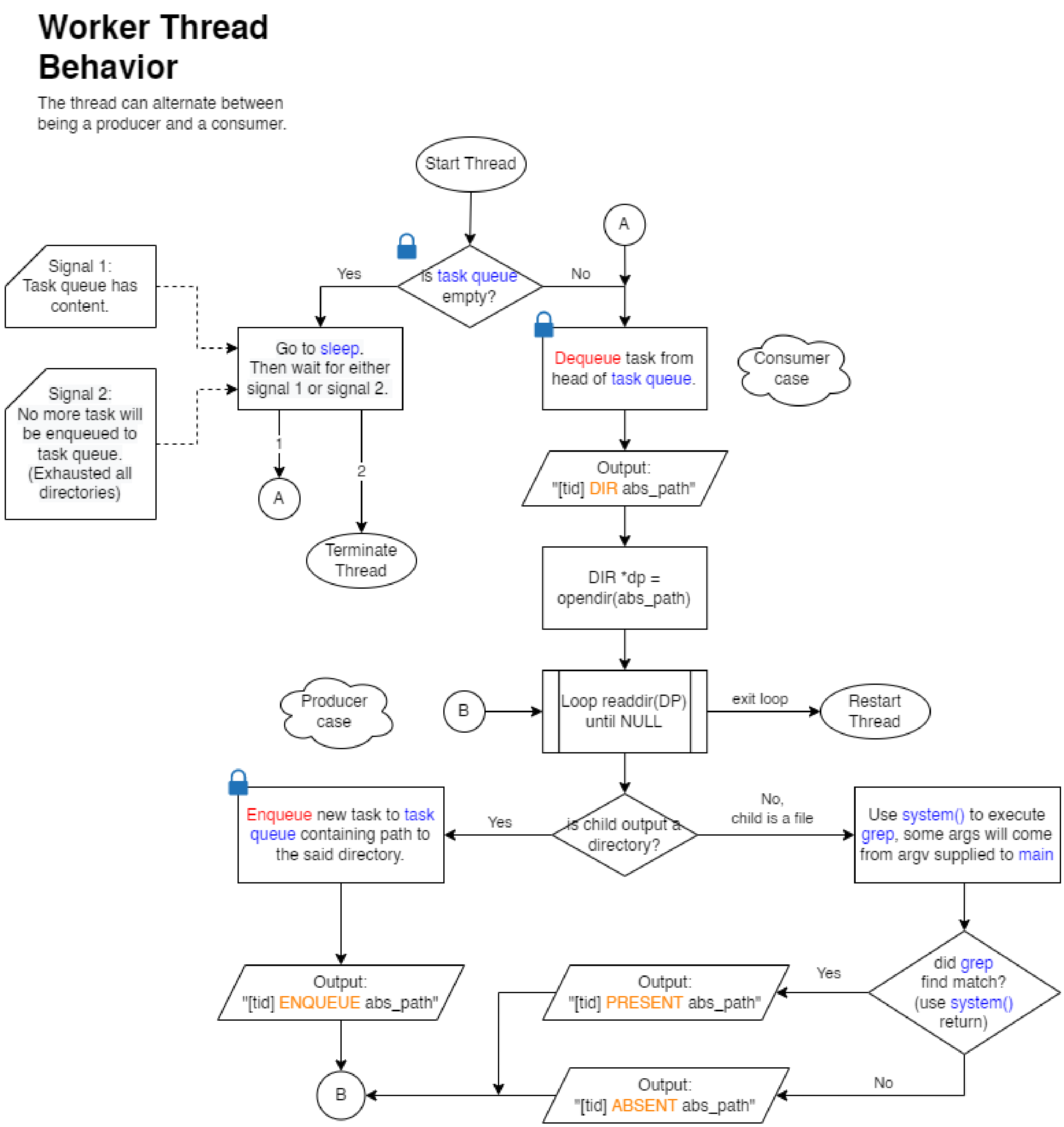
b. Explanation of Task Queue Implementation

For the **task queue**, we chose to use a **heap-based unbounded buffer approach**, i.e. a **dynamic approach**, in order to solve the following issue:

- On a bounded buffer, there exists a chance (100% if N = 1) that the worker threads acting as producers will end up getting deadlocked after they fully fill the task queue such that when they try to enqueue more tasks, they will have to call `wait`. This happens when all threads simultaneously act as producers because no other threads (acting as consumers) will be there to free up some space from the buffer and wake up the producers.
- On a bounded buffer, once the buffer is fully filled, any producer that attempts to enqueue to it goes to sleep, and hence adds context switching overhead. Although this may be desired by others to avoid thread starvation, since we are just multithreading grep and the threads are running the same routine, thread starvation isn't really an issue (the threads are all working towards the same goal), so we prefer to have less context switching overhead.

The aforementioned issues are non-problems when an unbounded buffer is used for the task queue because the "producer" threads will practically always have room to enqueue another task and thus doesn't have to go to wait. Interrupting the "producer" threads now relies entirely on the threads exhausting their current directory's jobs and on the thread scheduler.

Note here that the worker threads can act as either a **consumer** or a **producer** based on which part of the `routine` they are currently in. That is, there's an imaginary line separating the routine wherein a worker can be a consumer or a producer. We demonstrate this with the help of the following diagram:



As we can see, a thread acts as a consumer when it is trying to obtain a task from the task queue, but before it can do so, it must check if the task queue is empty. If the task queue is empty, it must wait until it has content, and if it is non-empty, it can proceed to dequeue. A thread then acts as a producer when it is exploring the (absolute path of the) directory given to it by the task it just dequeued during its previous iteration as a consumer. In this part, it is going to enqueue the absolute paths of the child directories (new tasks) of the current directory (current task) and wake up any thread who might be waiting for the task queue to have content. Once it is done doing so, it restarts the routine as a consumer. These threads only exit once they have told one another that the task queue is empty and none of them are busy (the task queue won't have another content) which will be elaborated in Item c.

We say *acts* here because the acting and producing parts is located within the same routine.

As mentioned, the task queue we are using is dynamic which requires the linking of together of task queue nodes. Our queue nodes have the following structure:

```
struct task {
    char abspath[MAXPATH];
    struct task *next;
};
```

Wherein abspath stores the absolute path corresponding to this task and next links to another task in the task queue.

We define a **task** as a "directory that is yet to be traversed by a worker".

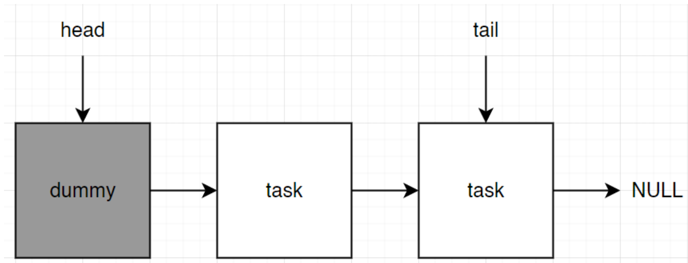
The size of the abspath string is bounded by MAXPATH >= 250 in accordance to the project guide. Note that we can let MAXPATH to be greater than 250 characters to avoid issues with string manipulation especially when it comes to null termination. While I haven't encountered problems with absolute path length during my testing with edge cases (i.e. file or directory absolute paths with lengths of 250), using a larger buffer for it as a preventive measure still seems like a good idea.

Now, the structure of the queue itself is given by:

```
struct queue{
    struct task *head;
    struct task *tail;
    pthread_mutex_t headLock;
    pthread_mutex_t tailLock;
} task_queue;
```

Wherein head points to the **dummy node** of the queue, which is the node always immediately before the first proper task node if the queue is non-empty; and wherein tail points to the last proper task node and is always linked to NULL.

As we will see later, the dummy node allows us to better separate the node at the actual head of the queue from the node at the tail of the queue such that when there's only one task enqueued, the head and the tail pointers still point at different nodes. This degree of separation enables both enqueue and dequeue operations to be performed simultaneously. This use of a dummy node is directly inspired by OSTEP. We can see a sample abstraction of the queue we're using below:



The queue structure also have separate locks for the head (headLock) and the tail (tailLock), and these are used for the dequeue and enqueue operations respectively. Having separate locks for the head and tail is also a prerequisite for **simultaneous dequeues and enqueues** because doing so separates the critical sections of the operations.

And as you may have also noticed, we initialize the task_queue as a global variable, allowing it to be easily shared among the worker threads. We also note that since our task queue nodes are allocated in the heap segment which is also shared among threads of the same process, they work well with the global task_queue structure.

We now discuss the operations that we can perform on the queue. We coded these operations into separate functions, as shown below.

- **Queue_Init**

```
void Queue_Init(struct queue *q) {
    struct task *dummy = malloc(sizeof(struct task));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

This function is called once to initialize the queue structure. The dummy node is prominent here. After this function, the head and the tail of the queue both point at the dummy node which is linked to NULL. The head lock and the tail lock are also initialized.

- **Queue_Enqueue**

```
void Queue_Enqueue(struct queue *q, char *abspath) {

    struct task *new = malloc(sizeof(struct task));
    // insert abs path to new node
    strncpy(new->abspath, abspath, MAXPATH);
```

```
    new->next = NULL;   // ensure tail->next after this routine is always NULL

    // link to tail of queue
    pthread_mutex_lock(&q->tailLock);
    q->tail->next = new;
    q->tail = new;
    pthread_mutex_unlock(&q->tailLock);
}
```

The enqueue operation attaches a new task to the tail of the task queue and makes it the new tail by linking it to `NULL`. We use `strncpy` here to copy the absolute path of the directory referenced by `abspath` to the `abspath` member of the new task. The critical section here (surrounded by `q->tailLock`) is the part where the task queue `q` is actually accessed and modified.

- **Queue_Dequeue**

```
int Queue_Dequeue(struct queue *q, char *taskpath) {
    pthread_mutex_lock(&q->headLock);
    struct task *tmp = q->head;
    struct task *newHead = tmp->next;

    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1;  // queue was empty
    }

    strncpy(taskpath, newHead->abspath, MAXPATH);
    q->head = newHead;  // newHead becomes new dummy, rem conts invalidated
    pthread_mutex_unlock(&q->headLock);

    free(tmp);
    return 0;
}
```

The dequeue operation takes in a pointer `taskpath` to a string buffer to which the absolute path contained by the dequeued task node will be written to. Observe here how dequeuing makes good use of the dummy node by only actually accessing the values contained by the node immediately after the dummy node (the actual head) and after copying those values (specifically the `abspath` to `taskpath`), that node is made the new dummy node, invalidating its contents.

The critical section here (surrounded by `q->headlock`) is almost the entire function aside from the operation that frees the heap memory of the previous dummy node referenced by `tmp`. We can `free(tmp)` without a lock because `tmp` is essentially already orphaned and since this is thread-local, it cannot be reached by any other process.

- **Queue_Is_Empty** (Deprecated)

```c
int Queue_Is_Empty(struct queue *q) {
    // Queue is empty when head and tail both points to dummy node
    if (q->head == q->tail) {
        return 1;
    } else {
        return 0;
    }
}
```

Queue_Is_Empty essentially returns 1 once both the head and the tail of the queue points to the dummy node. The functionality of this function is superseded by the use of the global **count** status variable that is tied to the **fill** condition variable. count tracks the current number of tasks inside the queue. We favor it over Queue_Is_Empty for explicitness and to maximize concurrency as this allows us to separate the queue structure from the actual checking of how many tasks are inside the queue (i.e. count is not immediately tied to Queue_Dequeue and Queue_Enqueue). Hence, Queue_Dequeue and Queue_Enqueue doesn't have to be surrounded by the same lock (lock1) that synchronizes when a thread will wait and or signal. Instead, only count has to be locked with the aforementioned lock.

This approach also avoids the race condition wherein Queue_Is_Empty reads dirty pointer values for q->head and q->tail especially when another thread is in the process of enqueing or dequeing.

We do not explain the Queue_Log function as it's only used for debugging.

Other global or shared variables to note are

```c
pthread_mutex_t lock1, lock2;
pthread_cond_t fill;    // status variable count linked to this
int count = 0;
int busyThreads = 0;
```

Only the variables in **bold** are the ones relevant for this item. As mentioned before, lock1, fill, and count work together to synchronize when a thread acting as a consumer must wait, wake-up, and spin or proceed to dequeue, or a thread acting as producer must wake-up a consumer.

Again, note that producer threads in our unbounded buffer implementation have no need to wait on a condition variable and spin on a status variable because they can enqueue all the time, hence we don't have the empty variable that can be found in the OSTEP (Ch. 30) implementation that was used as reference for this. These are only done for the consumer threads.

Specifically, inside the infinite while loop that's only broken when a thread finds out that all the other threads are already done with all their possible jobs (to be explained in the next item), we have

```c
pthread_mutex_lock(&lock1);
while (count == 0) {
    ...
    pthread_cond_wait(&fill, &lock1);        // wait for signal 1
    ...
}
```

```
...
// Task can be successfully dequeued (preempt count--)
count--;
pthread_mutex_unlock(&lock1);

char taskpath[MAXPATH];
Queue_Dequeue(&task_queue, taskpath);

printf("[%d] DIR %s\n", data->tid, taskpath);
...
while ((d = readdir(dirp)) != NULL) {
    ...
    if (isDir(pathbuff)) {
        // We now have true concurrency of Enqueue and Dequeue operations
        Queue_Enqueue(&task_queue, pathbuff);   // act as a producer
        printf("[%d] ENQUEUE %s\n", data->tid, pathbuff);
        pthread_mutex_lock(&lock1);
        count++;
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&lock1);

    } else {...}
    ...
}
```

Observe that there's a critical section here shared by the producer and the consumer threads denoted by the lines surrounded by `lock1`. The consumer acts on the top part of the routine's infinite loop, while the producer acts on the bottom part of the same infinite loop.

As we can see, when there is no task in the queue (`count == 0`), the consumer thread waits on the explicit queue provided by the condition variable `fill`. When there is task in the queue, the consumer thread can break the loop (possibly after being woken up by a producer who calls signal on `fill` after every enqueue). And before fully exiting the critical section, the consumer must **pre-emptively decrement** the shared variable `count`.

The race condition we are trying to avoid by pre-emptively decrementing `count` (as opposed to decrementing `count` after exiting the critical section) is the case when a thread incorrectly breaks the while loop even when another thread is about to empty it because the misbehaving thread thought there's still something to dequeue from the task queue. This race condition happens when count is not immediately updated before a thread exits the critical section and gets in line to dequeue, causing count to buffer and possibly be read dirtily.

Once again, due to the buffer being unbounded, the call to `Queue_Enqueue` by a producer is very simple (no `wait`), and only the `count++` and `signal` lines of the producer are surrounded by the same lock used to surround the spin routine of the consumer.

Some other important things we need to note is that:

- It is implicit in the function `pthread_cond_wait(&cv, &mutex)` to acquire the lock referenced by `mutex` before the thread calling it goes to sleep (to let others into the critical section while it waits), and it is also implicit that the lock is released just before a woken up thread returns. This is also why we use the same lock `lock1` for surrounding the `count++` and `signal` lines of the producer because we want a shared critical section so that `count` wouldn't be updated when there are consumer threads still in the process of reading from it nor `signal` would be sent when the thread that should've been sleeping on wait is not yet asleep.
- According to its man page, `pthread_cond_signal(&cv)` wakes up any thread blocked on cv **if any**. Hence, it is alright for the producer to call `signal` after every enqueue even when there's no thread blocked on it.

Obviously, without the locks, there will be multiple opportunities for race conditions because we are multithreading.

The other important components of the code is discussed in Item a.

c. Explanation of Worker Thread Termination Synchronization

As we've mentioned earlier, the threads will know to terminate once:

- **The task queue is empty**, and
- **There are no more busy threads** (acting as producers), hence the task queue will never have another task enqueued to it ever again.

We fulfill these conditions with the help of the following shared variables (in bold):

```
pthread_mutex_t lock1, lock2;
pthread_cond_t fill;    // status variable count linked to this
int count = 0;
int busyThreads = 0;
```

And they mainly work with respect to thread termination within the producer section of the routine because that's also where we can find the status variable `count` which *could* tell when the task queue is empty. Said section is again given below, fully this time compared to Item 2.

```
pthread_mutex_lock(&lock1);
while (count == 0) {
    busyThreads--;
    if (busyThreads == 0) {        // wait for "signal 2"
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&lock1);
        return;
```

```
    }
    pthread_cond_wait(&fill, &lock1);        // wait for signal 1
    busyThreads++;
}
// Task can be successfully dequeued (preempt count--)
count--;
pthread_mutex_unlock(&lock1);
```

As we may recall from the worker behavior diagram, the **signal 1** being referred to here is the `signal` by the producer thread that wakes up a consumer thread blocked on `fill`, and the **"signal 2"** (quote-unquote because no `pthread_cond_signal(&cv)` is actually called for this) is essentially just the shared variable **`busyThreads`** which counts the number of threads that are still awake and hence busy (and may possibly enqueue another task given the right conditions).

We focus on `busyThreads`: Note here how `busyThreads` is decremented by the thread that currently owns `lock1` immediately upon encountering it after entering the loop to denote that said thread is now non-busy (for all it knows, `count == 0`). Said thread may sleep if there are still `busyThreads`, by breaking `if (busyThreads == 0)`, or it may actually terminate once there are no more `busyThreads`.

If the thread is woken up, it immediately increments `busyThreads` again to denote that is now possibly going to do some tasks, and if it finds out that `count != 0`, then it can finally exit the loop and do said tasks as a busy thread. However, if it finds out that `count == 0`, it repeats the entire sequence by first decrementing `busyThreads` again as it waits for the next signal. So on and so forth.

Ideally, all the worker threads will exit at roughly the same time (once all `grep` tasks and directories have been exhausted) and with our implementation where we mark a thread as non-busy immediately before waiting and mark it as busy immediately after waking, we always ensure this ideal outcome by making the consumer threads sleep when there's no more producer thread enqueuing to it, so the final producer thread that transforms into a consumer thread will eventually find out that the task queue is empty and there are no more busy threads, so it exits, but not before waking up the other non-busy threads sleeping on `fill`. This is why we also make an exiting thread call `pthread_cond_signal(&fill)` right before it terminates, and said call starts a chain reaction that makes the non-busy threads finally exit (and wake-up another sleeping thread) one after another.

Now, all of these have to be done in the same critical section defined by the consumer and the producer subroutines to avoid the race condition wherein a thread may think there are no more busy threads and exits when there are still actually some, or the opposite scenario. That is, we don't just plop the busyThreads outside of the critical section, its use also have to be mutually exclusive.

Supplement on lock2

As you may have noticed, we haven't yet explained what `lock2` is for. What it surrounds is the following critical section inside the `routine`:

```
pthread_mutex_lock(&lock2);
chdir(taskpath);
DIR *dirp = opendir(".");
```

```
        char pathbuff[MAXPATH];
        char origcwd[MAXPATH];
        getcwd(origcwd, MAXPATH);
        pthread_mutex_unlock(&lock2);
```

We make this section a critical section because the conditions inside of it are sensitive to changes in the **current working directory**. Since the current working directory (cwd) is a per-process construct, a switch to another cwd by a thread calling `chdir` affects all the other threads of the process, and hence leads to undefined behavior if this is left unprotected.

For example, a producer thread (say `thread0`) is now carrying `taskpath` (a directory absolute path it most recently dequeued from the task queue) and it just called `chdir(taskpath)` in order to start working on it. However, right before it can call `opendir(“.”)`, it gets interrupted by `thread1` who calls `chdir(taskpath)` for its own `taskpath`, and when control returns to `thread0`, when it calls `opendir(“.”)`, it is now doing so from the `taskpath` given by `thread1`. The `taskpath` of `thread0` is now lost and the work on a directory (`taskpath` by `thread1`) is duplicated! Hence, to avoid something like this, we lock this section up.

That is, we have to keep in mind that `opendir(“.”)` (as a side-effect of using a relative path `“.”`) and `getcwd` are sensitive to which cwd they are called from, so we have to protect them by making the section mutually exclusive.

This will be more apparent in our walkthrough of code execution.

Multiprocess Version

I was able to make the parallel processes complete their tasks.

However, I was unable to synchronize how the threads will terminate.