CS214 Spring 2022
Project III

David Menendez

Due: April 19, 11:59 PM

This is a group project. You may work alone, or with a partner. Your partner may be in a different section.

To submit your assignment to Canvas, you *must* join a group in the Project III group set. These groups will be named Project III Group X, where X is some number. You must join a group, even if you are the only person in the group.

1 Summary

We will extend the `ww` program from Project II to support recursive directory traversal and multi-threading. You are encouraged to re-use or improve your word-wrapping code from Project II. (If you are working with a different partner, you may combine ideas from both partner's implementations.)

The word-wrapping requirements for this project are unchanged from Project II. The new features, to be added in this project, are:

Part I (40 points) Recursive directory traversal. When the first argument to `ww` is `-r`, `ww` will wrap all files in the specified directory and its descendent directories, recursively.

Part II (35 points) Concurrent wrapping. When the first argument to `ww` is `-rN`, where N is a positive integer, `ww` will wrap all files in the specified directory and descendent directories, using N threads for wrapping.

Part III (25 points) Concurrent wrapping with concurrent directory traversal. When the first argument to `ww` is `-rM,N`, where M and N are positive integers, `ww` will wrap all files in the specified directory and its descendent directories using M threads to read directory files and N threads to wrap regular files.

While it is not required, it would be reasonable to use a common implementation for all three parts, such that `-r` is treated as `-r1,1` and `-rN` is treated as `-r1,N`.

Example A directory foo contains a file bar and a subdirectory baz, which contains a file quux. This command:

```
$ ./ww -r1,2 20 foo
```

will create files `foo/wrap.bar` and `foo/baz/wrap.quux`, each wrapped to 20 columns. The argument `-r1,2` tells `ww` to use 1 thread for reading directories and 2 threads for wrapping files.

2 Recursive directory traversal

Using `opendir()` and `readdir()`, we can easily obtain a list of all files in a particular directory. One could imagine making this recursive by making your code recursive: when reading some directory A, you encounter a subdirectory B, handle it and its descendents in a recursive call, and then continue reading directory A. Unfortunately, there is a limit to how many files and directories a processes can have open at once, which would limit the depth of the directory subtree our program could handle.

Instead, `ww` will use a *work queue*. Each time a subdirectory is found, its path will be added to the directory work queue. Once the directory is finished (and closed), the directory traversal function will obtain the next path from the work queue, and begin reading its entries. Thus, the number of directories open at a time is fixed, regardless of the depth of the directory tree.

Note As with Project II, our directory traversal will ignore all files whose names begin with a period or the string "`wrap.`". (This neatly avoids any infinite recursion caused by the special `.` and `..` directories.)

For the record, the easiest way to determine if a string `foo` begins with a period is the test `foo[0] == '.'`. There are several easy ways to check the prefix of a string, including `strncmp()` (note the 'n').

3 Multithreaded word wrap

The main thread will start one or more work threads that will perform the actual word-wrapping tasks. Again, we will use a work queue to distribute the load. Each time the directory traversal system encounters a regular file, it adds that file's path to the file work queue. Each wrapping thread will repeatedly obtain a file path from the file work queue and wrap that file's output to the corresponding output file.

If the work file queue is empty, and the directory traversal is complete, the file thread should terminate.

Note Each file thread is a loop that will handle multiple files. We do not start a new thread for each file.

The use of a single work queue tends to balance the amount of work done by each thread. If a thread is working on a particularly long file, other threads will automatically obtain new files and redistribute the work load.

4 Multithreaded directory traversal

Extending directory traversal to multiple threads is conceptually simple. Each threads reads a directory path from the directory work queue and procedes to open and read the directory, adding the new files it finds to the file or directory work queues, as appropriate. If there are N directory threads, there will be at most N directories open concurrently in the process.

The directory threads should not stop until (1) the directory queue is empty and (2) no directory thread is currently reading a directory. When both conditions are met, all directory threads should halt.

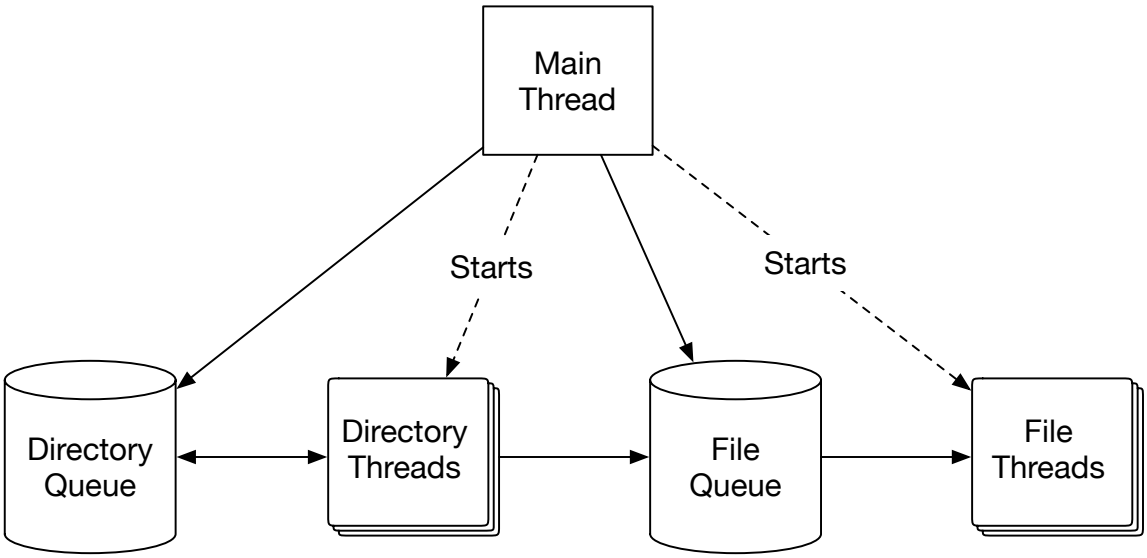

Figure 1: Organization of `ww`

5 Synchronized data structures

For the *file work queue*, you may use a bounded or unbounded synchronized queue, although a bounded queue is preferable. (A bounded queue will automatically slow down the directory threads if they are moving too quickly for the wrapping threads to handle.

For the *directory work queue*, you must use an unbounded queue.

You will also want to track

- whether any wrapping thread has encountered an error condition (unreadable input file, unable to create output file, word longer than line width) so that the process can return the correct exit code
- how many directory threads are currently active, so that the directory threads can terminate once their work is complete
- whether the directory threads have finished, so that the wrapping threads can terminate once their work is complete

Some of this information can be attached to an existing synchronized data structure.

Figure 1 shows the relationship of the main thread to various queues and worker threads.

6 Path handling

The working directory is process-wide, so you will not be able to use `chdir()` to open files in a different directory. Thus, you will need to construct paths by concatenating file names separated by slashes. You should use `malloc()` to allocate these paths, as the number of paths as well as their length will not be known in advance.

It may be tempting to use `strcat()` to copy one string to the end of another, but that is not necessary when the length of the first string is known—which it must be, in order for you to use `malloc()`. Consider this code:

```
char *path, *file;
...
int plen = strlen(path);
int flen = strlen(file);

char *newpath = malloc(plen + flen + 2);

memcpy(newpath, path, plen);
newpath[plen] = '/';
memcpy(newpath + plen + 1, file, flen + 1);
```

Don't simply copy this code! Read through it and try to determine why it was written this way. Why use `memcpy()` instead of `strcat()` or `strcpy()`? Why save the lengths of the strings instead of calling `strlen()` repeatedly?

7 Development

To aid in development, you will want to break your code into smaller portions that can be tested individually. It is much easier to confirm that a synchronized queue, for example, has been correctly implemented by building a custom test harness for it. If each unit has been written correctly, your team should be able to assemble them into a complete working project.

Figure 2 shows one way to think of the various parts of `ww`. While some units naturally incorporate others (for example, the file worker incorporates the word wrap function and communicates with the file queue), it is still feasible to test each portion in isolation as well as testing the complete program.

You *will* encounter unexpected synchronization problems in your development, so make sure you are able to find and correct them earlier rather than later. Divide responsibilities among your units in a reasonable way and make their interfaces clear and comprehensible.

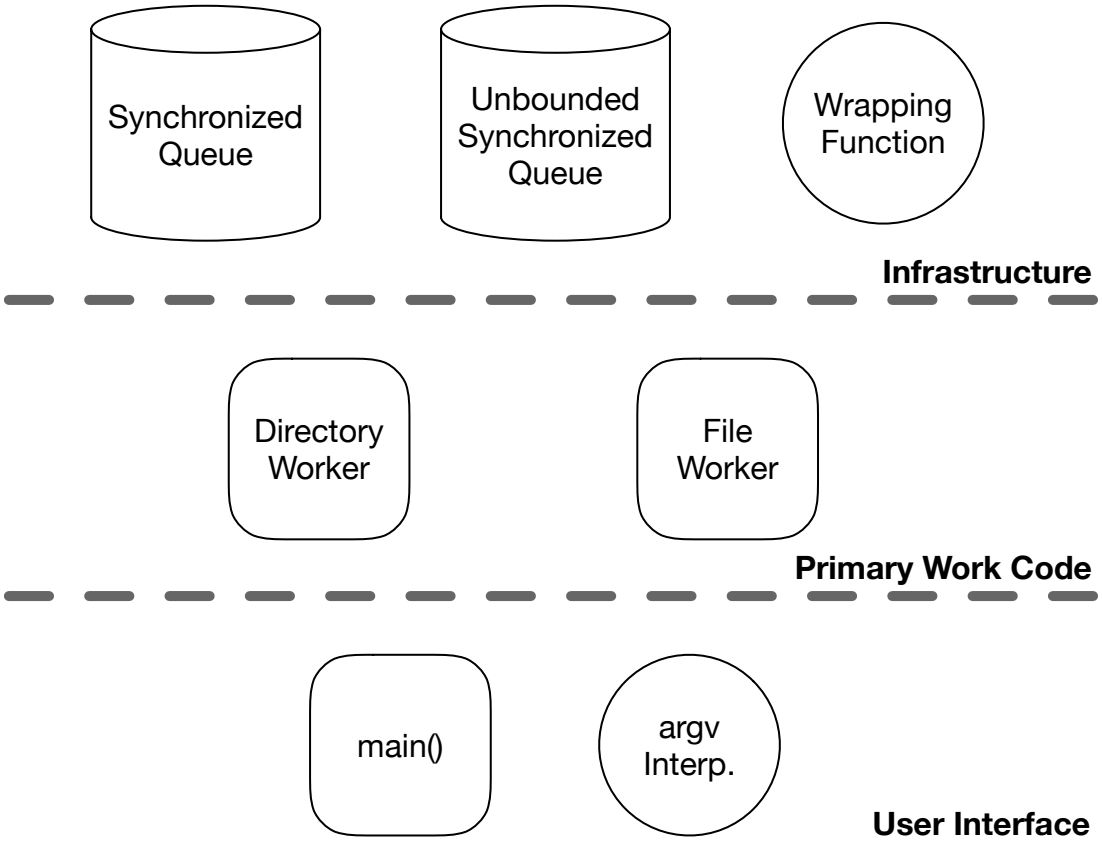

Figure 2: Organization of `ww`