Quick Note

PLEASE NOTE: The professor said that using Markdown was an acceptable format. I had directly confirmed with him.

FURTHERMORE: To have the best reading experience, we HIGHLY recommend you to use a MarkDown editor as that will provide the best reading experience. While you can read the PDF, we recommend you to use some kind of MarkDown editor.

Some Markdown options include (but are not limited to) VSCode, GitHub, JetBrains, Atom, and most other common IDEs / editors. You can also use `grip` by installing `pip3 install --upgrade pip grip` and then `grip <path-to-README.md>`. This will create a local server that you can access to view the Markdown file.

While you can use the PDF also included here, we highly recommend you to use a Markdown editor as the PDF may have formatting imperfections. While we did our best to validate it, for the easiest experience, we highly recommend using one of the options above.

Authors

- Carolette Saguil (cas699)
- Hasnain Ali (ha430)

Introduction - Multi-threaded Word Wrapper

This is a Word Wrapper that applies the "Greedy Wrapping" algorithm. This means, given a column size by the user, we will wrap a file based on that column size. For example, suppose `file1.txt` is written like so:

```
This is very good text that has been put into a file for the purposes



of being
            an
example.


That                    is good.
```

Then calling `./ww 15 file1.txt` will produce in `stdout`:

```
This is very
good text that
has been put
into a file for
the purposes

of being an
example.

That is good.
```

As you can see in the example above, the original text from `file1.txt` is now wrapped to a column width of 15. If the next word exceeds column width, it is printed on the next line. This is consisted with `pa2.pdf` and the write-up given to us by Professor Menendez.

READ THIS PART... PA3 Information Starts Here
Additionally, now, in `pa3` (this project), this same program now has the option for recursive directory traversal and multithreading. For example, if the `-r` option is passed followed by a column size and a directory (respectively), this program will loop through each directory and every subdirectory, wrapping the files as it goes. It will create new wrapped files with wrapped text in the following format: `wrap.<file_name>`. Inside, there will be wrapped text from the wrapped files passed in. This will be true for the current directory and all subdirectories.

Also, along with passing in the `-r` option, we also allow for specifying multiple file threads and directory threads. For example, an option like this `-r5` will create 5 files threads. Whereas, an option like `-r5,6` will create 5 directory threads and 6 files threads. Please note the spacing as this program is space sensitive.

PLEASE NOTE: WE DID COMPLETE THE EXTRA CREDIT. THIS PROGRAM DOES ALLOW FOR MULTIPLE ARGUMENTS TO BE PASSED IN AND HAVE ALL THOSE ARGUMENTS BE COMPLETED.

Pre-Requisites

This program was only intended to be executed on `x86` and `x86_64` architectures. This program assumes that the hardware will be consistent with that of `x86` and `x86_64` architecture. This requires the hardware to use 2s complement and be consistent with Little Endian endianness. Additionally, this program requires you to be using some form of Unix as only Unix based operating systems have the proper file system structure that will enable all the features of this code to properly execute.

This program access `/dev/stdout` and `/dev/stdin`. Please make sure you have the appropriate permissions to access these files and any other files you pass into `ww`. Furthermore, please make sure you have at least read, write, and execute permissions for any files or directories you pass in.

Test Plan

For our test plan, all the same functionality specified below (and in PA2) must still work. That means that all the original functionality must still be the same. For our code to be correct, we need to allow the `-r` option indicating recursive directory traversal. That is, every time we find a new directory, we also wrap all the files in that directory. Furthermore, if the user specifies `N` file threads, then our program will be multi-threaded. That is, `N` threads will be spawned to do the actual file wrapping. So, if the argument is passed in like this: `-rN` (with all the other proper arguments, that is, with a column size and a directory name), then `N` threads will be spawned and the files will be wrapped according to the proper format (with the `wrap.`) prefix. Likewise, if the option `-rM,N` where `M` is the directory threads and `N` is the file threads, then `M` directory threads will be spawned to do the directory traversal and `N` file threads will be spawned to do the wrapping.

Furthermore, this program also can do the extra credit assignment, that is, it can take in multiple arguments and wrap the file or directory accordingly. Please see Canvas for more details about this. This program is capable of doing that.

What We Are Trying To Prove (PA3)

- We are trying to prove that this program is capable of recursively traversing through a specified directory and wrapping the files as it goes.
- If `N` file threads are specified like so: `-rN`, then it will spawn `N` threads to do the file wrapping.
- If `M` directory threads are specified and `N` directory threads, like this: `-rM,N`, then `M` threads are spawned for the directory wrapping and `N` threads are spawned for the file wrapping.
- Lastly, if multiple sets of arguments are passed in, then as per the extra credit, all those arguments will be processed and wrapped accordingly.

How We Determined Our Program Was Correct (PA3):

To determine our program was correct, we ran a series of stress tests.

- We first put print statements in basically everywhere the treading and enqueuing occurred. This is how we knew that the threading was occurring.
- We also tested to make sure that the threads were ending at the right time. We did this by keeping track of everything the threads would need to know to terminate.
- We also made sure that our return value was correct when returning at the end.
- We made sure that the program can take in multiple arguments.
- We made sure that each of the threads were starting, once again with print statements.
- We made sure that the wrapped files were being generated as specified above.
- We made sure that the text was actually wrapping.
- We made sure all previous functionality worked.
- We made sure the threads ended in the way they were supposed to
- We made sure that everything is working as intended according to the project write up.
- We also wrote a quick script that generated random bytes and wrote them into directories and subdirectories. You can use this script in your own testing
    - In the `src` directory, run the following command:
        - `./recursivebytes` This will create a randomly generated directory with subdirectories that we also did testing on. This allowed us to create very big files easily to see the effects of multithreading and to ensure that our code was working. Feel free to use it as you wish.

- We also have functions in `unbounded_queue.c` that will check if memory allocation or if any pthread function fails.

Everything Below Must Be True As These Are The Same Parameters For PA2. Everything below still applies to our test plan.

In order to stress test our word wrapper, we used various different stress tests we will detail below. From strange while space to multiple paragraphs to everything being in one line, we have tested even the most unlikely scenarios to make sure everything was correct. Below, we detail this further.

What We Are Trying To Prove

We are trying to prove the following properties of our code, and it's behavior. For our us to consider our code correct it needs to consist of the following properties.

- We are trying to prove every single wrapped text will consist of a newline at the end.
- We are trying to prove that each line will be as long as possible without exceeding the `column size` width specified by the user (excluding the `\n` character).
- We are trying to prove no amount of white space at any point in the file will cause errors or unexpected behavior; that is, regardless of whitespace, the file will always be wrapped in the way we intend; that is, every line is of a specified length or less and that every line is as long as it can be.
- We are trying to prove that no word is ever cut off, that every last character of every line will always be the `\n`.
- We are trying to prove that every regular file in a specified directory will be wrapped the way we intend for it to be wrapped. Additionally, all new files created in the directory will have the prefix `wrapped.`.
- We are trying to prove that any invalid input will throw an error with `exit code 2`.
- We are trying to prove no matter how long or how much whitespace present in a file, the word wrapping algorithm will wrap the file given the `column size` and output it to the desired location
- We are trying to prove that if a word is longer than column size, the wrapper continue printing the word and subsequent words; upon termination, we `exit code 1`.

Below and throughout this document, we will proceed to prove each of those properties by showing what test cases and design properties we used. Through extensive testing and robust code, we have been able to prove each property listed above. We will now get into explaining how we were able to prove each property.

How We Determined Our Program Was Correct

We determined that in order for our code to be correct we needed to be able to:

- Read from `stdin` and print to `stdout`, if only `columnSize` is given with no other file name.
- Wrap a regular file of given `columnSize` and print to `stdout`, if the second argument is a regular file (here, we are assuming that `argv[0]` is just the file path to the executable. When we say "second argument," we are excluding `argv[0]` from our count)
- Wrap all the regular files in a directory (ignoring files that begin with `.` and `wrap.`) and write to a new file in the same directory, if the second argument is a directory.
    - We check to make sure the directory passed in `argv[2]` is actually a directory that exists.

In order for the wrapped file to be correct it would need to:

- Have lines that are less than the given column width without cutting off words, but placing new words that can't fit, onto a new line by themselves.
- Make paragraphs when the inputted file has a sequence of at least two newlines characters (`'\n'`).
    - **Please Note: At the end of a while, if there are is more are two consecutive new line characters in a row, i.e: there are two new lines at the end of the file rather than one, new line (with no words on it), this program will print two newline characters at the end, rather than one because it is expecting a new paragraph to start. the directions were not clear about this, but we thought it was the best design.**
        - So, for example:

```
here is some test i guess


```

        - When wrapped, this program is expecting a new paragraph to begin. This was done by design and by choice. We were unsure of what do to in this case and thought this was the best design. So, wrapped with a `colSize` of 100, the above example will look like this:

```
here is some test i guess


```

        - Whereas, if there is only one new line character at the end, only one newline character will be printed at the end like so:
            - Original Text:

```
here is some test i guess

```

        - Wrapped to `colSize` 100:

```
here is some test i guess

```

        - Note how above, there is only one newline. If there are multiple newline characters at the end of the file (more than two), at most, only two will be printed because the program is expecting to start a new paragraph. This was done by design for robustness as the

directions in the writeup were not clear about this scenario.

- Print a word on a single line when it exceeds the width and finish wrapping, then return `exit code 1`.

Please Note: If the user does not give `argv[1]` to be `columnSize`, the program will fail and set `errorno`!

Types of Files and Test Scenarios

We tested scenarios that would cause errors or return exit failure such as:

- Giving no arguments.
- Giving a first argument that is not a positive integer.
- Giving a first argument that is smaller than the size of a word in a given file (in this case, we continue printing, but return 1 on termination to indicate an error).
- Giving a second argument that is not a regular file that exists.
- Giving a second argument that is a directory, in which case we open the directory and wrap every regular file to the specified column width.

Please Note: We have a function called `checkArguments()` which checks all these arguments and reacts accordingly based on the arguments.

We tested files that included:

- Whitespace sequences containing at least two newlines characters (and more).
- We tested files with multiple space characters between words; in the very beginning of the file, in the middle, and at the end.
- Extremely long words
- Extremely short words
- Lines where words had one or more spaces in between.
- Lines that started or ended with any kind of white space.
- An empty file.
- Very long files with various types of formatting.
- Files that don't exist.
- Column width of 1
- Column width of 999
- Column width of 64
- Word at the very beginning longer than colSize
- All words longer than `column size` will be printed on their own line. This will result in `exit code 1` being returned.

We tested directories that included:

- Files that started with `.` or `wrap.`
- Directories with regular files inside them.
- Directories that don't exist.

Design Properties / Implementation Details

This program is designed in two parts. The first part actually reads from the input file (which may be `stdin`), tokenizes the words, and wraps each string according to rules stated here as well as in the write-up `pa2.pdf`. The second part writes the wrapped string denoted as a `struct wrappedString` into either the file specified by the user, or `stdout` depending on `argv`.

(PA3) Part 1

- If the `-r` option is passed, the following function will be called:
  - `int recursiveThreading(char **args)`
    - This function will check if the `-r` option is passed properly (with a column size and a directory that exists). If that's the case, then this function will recursively iterate through the directories and wrap each file in the directory and all of its subdirectories with new wrapped files created, all with the `wrapped.` prefix.

(PA3) Part 2

- If the `-r` option is passed with `N` threads, like `-rN` then the program will be multithreaded and iterate through all the files in the directory and its subdirectories.
  - just like it did in part 1. The only difference here is that the program will use `N` threads to do the wrapping.

(PA3) Part 3

- If the `-r` option is passed with `M` directory threads and `N` file threads, then the program once again will be multi-threaded and will also be recursive. It will create `M` directory threads and `N` file threads. Similar to part 2, all files in the directory and its subdirectories will be wrapped in the format specified above. Only this time, there will be `M` directory threads doing the traversal and `N` file threads doing wrapping.

(PA3) EXTRA CREDIT

- If multiple arguments are passed in, that is, if more than the optional `-r` option, a column size, and a file/directory are passed in, then each of those arguments will be processed. For example, if the user passes in `-r <column size> <directoryName> <fileName> <directoryName>`, then all the directories, files, and second directories will be wrapped. The second directory will also be recursively wrapped since the `-r` option was specified.

(PA2) Part 1

- `int wrapFile(int fd, size_t colSize, int wfd)`
  - This function is the main function that will take a file descriptor input `fd`, read the files contents word by word (tokenizing each word), and wrap the files contents according to the rules described above in here as well as in the write-up `pa2.pdf`. It is here where the greedy word wrap algorithm is executed.
  - For this function to be correct, we account for different white space and newline patterns, in that with any form of white space or new line given, the word wrap algorithm will still execute.

We account for different white space patterns, long words, short words, all wrapped based on the user's requirements denoted by `colSize`.
    - Lastly, to be clear, for maximum performance and efficiency, we wrote our own method of tokenizing that meets our specific needs to maximize performance. Since we need to read the buffer array character by character anyway to detect and deal with whitespace, in the file, while we are traversing the buffer array, we tokenize each word and store it (in case we reach the end of the buffer and need to store the word for the next buffer), and print it on the appropriate line so that the line does not exceed `colSize`.
    - In our extensive testing, we tested many scenarios that stressed tested these design properties.
    - Returns either 1 or 0 based on the exit status; were we able to wrap to the specified `colSize`? Did every word fit within the specified `colSize`?

(PA2) Part 2

- `char* readPathName(char* dir, char* de)`:
    - This function takes the arguments of the directory name and the file name.
    - Uses `strcat` to add a `/` in between the directory name and the file name to get the path for the file we need to read (e.g. `dirName/fileName.txt`).
    - Returns the path name created.
- `char* writePathName(char* dir, char* de)`:
    - This function takes the arguments of the directory name and the file name.
    - Uses `strcat` to add a prefix `wrap.` before the name of each regular file we read from, then add a `/` in between to get the path for the file we need to write to (e.g. `dirName/wrap.fileName.txt`).
    - Returns the path name created for the specified wrapped file.
- `int wrapDirectory(DIR *dir, char* dirName, int colSize)`:
    - This function takes the arguments of the `directory`, `directory name`, and `colSize`.
    - First checks if the directory was opened correctly and successfully then traverse through directory entries searching for regular files.
    - It skips entries that begin with `.` and `wrap.`.
    - Use `char* readPathName` to get the path for the file and use `stat()` to get file info of the specified path that was returned.
    - If and only if the entry is a regular file, then we use `char* writePathName` to get the path for the `wrap.file` we will write to, and use `int wrapFile` to wrap the file and print it out to the `wrap.file` file where the wrapped input file is stored.
    - Also has a status and if there is a case in one of the files where the column size is smaller than word size then it will return 1 when it has finished wrapping the directory. See part 1 for more information.
    - Closes directory once it has finished wrapping.

Other Design Notes

- `char checkArgs(int argc, char **argv)`:
    - Checks if the arguments are valid by checking if there is:
        - The correct number of arguments inputted by the user.
        - If first argument is a positive integer OR the `-r` recursive argument.

        - If the second argument is an existing file or directory (may also be a column number, depending on if the optional `-r` option was passed).
        - **Please note: When we say first argument, we are ignoring `argv[0]`, that is technically the first argument, but for simplicity's sake, we disregard that and assume it to be true.**
    - If the arguments do not pass these check they will print out errors and exit with status 2 since status 1 is reserved for when a word in the file is longer than the column size.
    - If they do pass these tests then:
        - If the second argument is a regular file, it will return char `f` and wrap the file, then print it out to `stdout`. (Considering the file exists. If the file does not exist, it will fail with `exit status 2`.)
        - If the second argument is a directory "file", it will return char `d` and wrap all regular files in the directory. (Considering the directory exists. If the file does not exist, it will fail with `exit status 2`.)
        - If there is no second argument, it will return char `e` and wrap the input to `stdin`.
- `void printDirEntry(DIR *dir)`:
    - Was used for testing purposes.
    - Just traverses through the directory and prints out the directory entries.
    - Is a test function that is not implemented in the overall program. Simply used for debugging purposes.
- `void checkWriteSuccess(ssize_t writeValue)`
    - Checks if all calls to `write` have succeeded. If it has not, it exits with status 2 for reasons stated above. Exit status 1 is reserved by other parts of this program.
    - On success, we return and continue with the program.
- `void checkIfMemoryAllocationFailed(void *ptr)`
    - This function checks if calls to `malloc` or `calloc` or `realloc` were successful. If not, we return with exit code status 2 and end the program.
- `recursiveThreading(char **args)`
    - This function will process the `-r` argument and if there are a specified number of file, or a specific number of file threads and directory threads, this function will spawn them all in and multi-thread.
    - So, if only the `-r` option is passed into the program, the directories and all the subdirectories will be wrapped as expected (with the `wrap.` prefix behind each new file created).
    - If `N` file threads are passed with the `-r` option like this: `-rN`, then multiple file threads will be spawned in and created. These threads will do the file wrapping in the directories as new directories and files are added to the queue.
    - If `M` directory threads are passed and `N` file threads are passed with the `-r` option like this: `-rM,N`, then `M` directory threads will be spawned to traverse the directories and `N` file threads will be spawned to wrap the files in the directory in the format stated previously.

Execution Instructions

To run this program, the directions are fairly simple and obvious. Nonetheless, here are some instructions to follow in case you are having trouble.

Please Note: This program was tested and executed on the Rutgers iLab Machines! Please see Pre-

Requisites for more details on what types of systems are required for this program to run as expected. To any graders: Please only execute this program on iLab machines running `gcc` version 9.4. It is only in this way we can guarantee this program will behave as expected.

To compile this program, please navigate to the the `src` directory. In the terminal:

```
cd <path/to/project-folder>/src
```

Please make sure your path is the path to the project folder. Then navigate to the `src` directory.

In the `src` directory, to compile `Word Wrapper`, enter the following:

```
make
```

To execute the program, make sure that the code compiled successfully. **Please note, the device you use should have at least some version of `gcc` version 9 installed.**

To execute:

```
./ww <colsize> <path-to-input-file (is an optional argument)>
```

or

```
./ww <colsize> <path-to-directory>
```

or

```
./ww <colsize>
```

or you can specify the recursive option with -r

```
./ww -r <colsize> <directoryName>
```

or you can also specify N file threads you want to spawn

```
./ww -rN <colsize> <directoryName>
```

or you can spawn N file threads and M directory threads by doing the following:

```
./ww -rM,N <colsize> <directoryName>
```

To clean all compiled executables and any other binary files left behind, enter:

```
make clean
```

Which will clean the project and only leave behind the source code, `ww.c` and the `Makefile`.

`gcc` Compiler Flags Used:

- `ww` (in the Makefile provided) is compiled with the following flags for the following reasons:
  - `-g`: provides debugging information in case the use for a debugger like `gdb` is needed. It can provide more debugging information.
  - `-std=c99`: This is C standard that we are required to use for this class.
  - `-Wall`: Enable all major warnings.
  - `-Wvla`: Warn if variable length arrays are used. This is enabled to ensure `malloc` is used for the character arrays rather than variable length arrays.
  - `-Werror`: Treat all warnings as errors.
  - `-Wundef`: Warn if any undefined behavior occurs.
  - `-Wpointer-arith`: Warn if any invalid or risky pointer arithmetic occurs.
  - `-fsanitize=address,undefined`: We include this because this program involves the use of dynamic arrays, and we want to make sure that our memory is handled as best as we can (to ensure no overflows or leaks occur). We also compile with undefined sanitizer to ensure no undefined behavior occurs. This is for security and behavioral purposes.
  - `-pthread` Required when using the `pthread.h` library.