Please **read this entire assignment**, **every word**, before you start working on the code. There might be some things in here that make it easier to complete.

This is an individual assignment.

This lab is April 25th by 11:59pm.

Submit a single gzipped tar file to **Canvas**. If you don't know how to create a gzipped `tar` file, you need to learn before you submit this assignment. Do not just `tar` up all the junk in your development directory and submit that into Canvas. Look to see if there is a helpful target in the Makefile. **It will be a 20% deduction if I see a bunch of junk like `.o` files, executable files, revision control files, or editor droppings in the tar file you submit**. If you really need to know how to create the tar.gz file, read the `Makefile`.

Part 1: `mystat` (60 points)

For this part of this assignment, you will write a C program that will **display the `inode` meta data for each file given on the command line**. You must call you source file `mystat.c` and your program will be called `mystat`. **The output of your code must exactly match that of my code.**

An example of how my program displays the `inode` data is shown the Figure *1*. You might also want to look at the output from the `stat` command (the command not a system function, `man 1 stat`). Though the `stat` command not as pretty (or in some cases as complete as the replacement you will write), it is the standard for showing `inode` information.

Requirements for your program are:

1. Display the file name
2. Display the file type (regular, directory, symbolic link, …) as a human readable string. **If the file is a symbolic link, look 1 step further to find the name of the file to which the symbolic link points**. **See** Figure *1***.**
3. Display the device id, both as its hex value and its decimal value (see the `stat` command).
4. Display the `inode` value.
5. Display the mode as both its **octal** values and its symbolic representation. The symbolic representation will be the `rwx` string for user, group, and other. **See** Figure *1* or '`ls -l`' for how this should look.
6. Show the hard link count.
7. Show both the `uid` and `gid` for the file, as both the symbolic values (names) and numeric values. This will be pretty darn easy if you read through the list of suggested function calls. **See** Figure *1* for how this should look.
8. Preferred I/O block size, in bytes.
9. File size, in bytes.
10. Blocks allocated.
11. Show the 3 time values in local time/date. This will be pretty darn easy if you read through the list of suggested function calls. **See** Figure *1* for how these look.

```
rchaney # d -d FUNNY*
lrwxrwxrwx 1 rchaney them 14 Apr 17 05:52 FUNNYbroken -> DOES_NOT_EXIST
drwxr--r-- 2 rchaney them  2 Jun  7  1990 FUNNYdir/
-rw-r--r-- 2 rchaney them 32 Mar  3  1970 FUNNYhlink
---------- 1 rchaney them  5 Apr 17 06:01 FUNNYnoaccess
prw-rw-r-- 1 rchaney them  0 Apr  5  1980 FUNNYpipe|
-rw-r--r-- 2 rchaney them 32 Mar  3  1970 FUNNYregfile
lrwxrwxrwx 1 rchaney them 12 Apr 17 05:51 FUNNYslink -> FUNNYregfile
srwxr--rw- 1 rchaney them  0 Mar 14  2001 FUNNYsocket=
```

Figure 1: The FUNNY* files.

System and function calls that I believe you will find interesting include: `stat()` and `lstat()` (you really want to do "`man 2 stat`" and read that `man` entry closely, all of it [**yes really, all of it**]), `readlink()`, `memset()`, `getpwuid()`, `getgrgid()`, `strcat()`, `localtime()`, and `strftime()`. Notice that `ctime()` is NOT in that list and you don't want to use it. Since you must be able to show the file type if a file is a symbolic link, I encourage you use `lstat()` over `stat()`.

My implementation is about 280 lines long, but is some dead code in my file. I have code commented out to support features not required for your assignment. There is no complex logic for this application, just a lot of long code showing values from the `struct stat` structure from `sys/stat.h`. Honestly, the longest portion of your code will likely be devoted to displaying the symbolic representation of the mode. Formatting these strings is a little *awkweird*. I suggest you create a function. Don't worry about sticky bits or set uid/gid bits.

You need to be able to show the following file types: regular file, directory, character device, block device, FIFO/pipe, socket, and symbolic link. When formatting the human readable time for the local time, I'd suggest you consider this "`%Y-%m-%d %H:%M:%S %z (%Z) %a`", but read through the format options on `strftime()`.

```
rchaney # d /dev/vda /dev/mem
crw-r----- 1 root kmem   1, 1 Mar 30 19:21 /dev/mem
brw-rw---- 1 root disk 252, 0 Mar 30 19:21 /dev/vda
```

Figure 2: Block and character files.

I have some examples in my `Lab1` directory for you to use in testing (the `FUNNY*` files, Figure 1). You can find a block device as `/dev/mem` and a character device as `/dev/vda`. See Figure 2.

I use spaces for formatting the output to look nice.

If you want to see how my solution displays all the data for the FUNNY* files, run the following command:

```
~rchaney/Classes/cs333/Labs/Lab1/mystat ~rchaney/Classes/cs333/Labs/Lab1/FUNNY*
```

Part 2: `mywc` (60 points)

Write a C program called `mywc` from a source file called `mywc.c`. You must use `getopt()` to process the command line options. The required command line options are shown in **Table 1.** You must be able to give the command line options in any order, but `getopt()` will take care of that for you. **The output of your code must exactly match that of my code. See Figure 3.**

Command line Option	Description
`-h`	Call a function that displays a message showing the command line options and exit with an exit value `0` (zero). I use tabs when formatting the help output.
`-c`	Count and display the number of characters in the input.
`-w`	Count and display the number of words in the input. Words are delimited by newline or whitespace. Strongly consider use of `strtok()`.
`-l`	Count and display the number of lines in the input. Lines in the file are newline delimited. Note that this is an lowercase L.
`-f file_name`	The name of the file to be processed. The name of the file will not exceed 99 characters in length. • If no file name is given on the command line (which means there is not a `-f` command line argument), your program must read from `stdin`. Such as `./mywc < fileName` • If the file cannot be opened, the program should print an error message to `stderr` and exit with an **exit value of 2**.

Table 1: Required command line options for `mywc`.

If an illegal command line option is passed to the program (such as `-Q`), it should print an error message to `stderr` and **exit with an exit value of 1** (one).

If none of `-c`, `-w`, `-l` are specified on the command line, assume the user intended to enable all of them to be enabled (like the regular UNIX command `wc`). In fact, the way you can validate your code is to compare the values displaced from your code to those displayed from `wc`. **The values from your code should be in the same order as those produced by `wc`**. Check the man page for `wc` to find what that is.

No line from the input file will exceed 1,024 characters in length**. If the program runs successfully, its exit/return value should be 0**. For comparison, my implementation has less than 140 lines. The easy thing to do is to always keep all the counters, but only output the ones you need at the end.

Other than `getopt()`, the only new function I'd encourage you to investigate is `strtok()`. It is really cool, but a little awkward to initially understand. Unfortunately, the man page only gives an example of `strtok_r()`, which accomplishes the same thing (and is reentrant), but more difficult to understand and use.

See the examples of running `mywc` in the image below. You can find the sample input files in `~rchaney/Classes/cs333/Labs/Lab1/*.txt`. You can also find the working examples of my implementation of `mywc` in that directory. Sadly, the source code is not viewable to you.

```
rchaney # ./mywc -h
./mywc
        options: clwf:hv
        -c      : display the number of characters in the input
        -l      : display the number of lines in the input
        -w      : display the number of words in the input
        -f file : use file as input, defaults to stdin
        -h      : display a command options and exit
        -v      : give LOTS of gross verbose trace output to stderr.
 babbage ~/Classes/cs333/Labs/Lab1
rchaney # ./mywc < ingredients1.txt
10 11 62
 babbage ~/Classes/cs333/Labs/Lab1
rchaney # ./mywc -f ingredients1.txt
10 11 62 ingredients1.txt
 babbage ~/Classes/cs333/Labs/Lab1
rchaney # ./mywc -l -f ingredients1.txt
10 ingredients1.txt
 babbage ~/Classes/cs333/Labs/Lab1
rchaney # ./mywc -l -f ingredients1.txt -c
10 62 ingredients1.txt
 babbage ~/Classes/cs333/Labs/Lab1
rchaney # ./mywc -l -f ingredients1.txt -wc
10 11 62 ingredients1.txt
 babbage ~/Classes/cs333/Labs/Lab1
rchaney # ./mywc -l  -wc < ingredients2.txt
66 94 594
 babbage ~/Classes/cs333/Labs/Lab1
rchaney # ./mywc -lwc < ingredients2.txt
66 94 594
 babbage ~/Classes/cs333/Labs/Lab1
rchaney # ./mywc -f ingredients2.txt -l -w -c
66 94 594 ingredients2.txt
```

Figure 3: The output of mywc.

Supplement: How to Create a gzipped tar File

If you do not submit `gzip` compressed tar file, it is very likely that your assignments will not be graded (therefore give you a zero for all your hard work). So, you should make sure you know how to:

1. Create a `gzip` compressed tar file.
2. Extract files from a `gzip` compressed `tar` file.
3. List the contents of a `gzip` compressed `tar` file.

Read the `man` page on `tar` and make sure you understand the following command line options for tar: `x`, `c`, `t`, `f`, `v`, and `a`. **Ponder the deeper meaning of the following lines**:

- `tar cvfa Lab1.tar.gz *.[ch] [mM]akefile`
- `tar xvfa Lab1.tar.gz`
- `tar tvfa Lab1.tar.gz`

Final notes

I do not, yet, have testing scripts for these programs, I'm sure I will get those done soon. I'll let you know when they are ready.

The labs in this course are intended to give you basic skills. In later labs, we will ***assume*** that you have mastered the skills introduced in earlier labs. **If you don't understand, ask questions.**