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 lab consists of multiple parts. **All Parts of this lab are July 13th by midnight.** Submit a single gzipped tar file to **TEACH**. The single gzipped tar file should contain the all source files (C source [`*.c` and `*.h`] and the `Makefile`). **You must have a single `Makefile` to build all the C programs of this assignment.**

Part 1 – `mystat` – Working with `inodes` (75 Points)

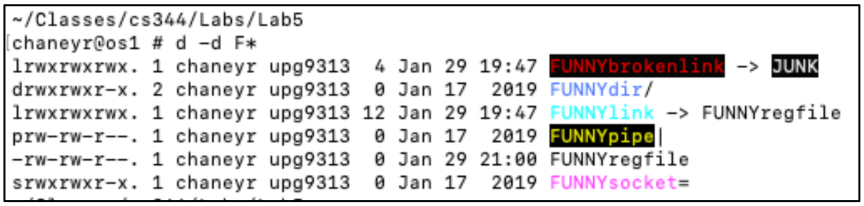
For this part of this assignment, you will write a C program that will **display the `inode` meta data for each file given on the command line**. You must call you source file `mystat.c` and your program will be called `mystat`.

An example of how my program displays the `inode` data is shown the **Error! Reference source not found.** you might also want to look at the output from the `stat` command (the command not a system function, `man` section 1). Though 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. **Show the `inode` data for ALL (valid) files on the command line**. If a file given on the command line is invalid (cannot be stat-ed, ignore it).
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** Error! Reference source not found.**.**
3. Display the device id.
4. Display the `inode` value.
5. Display the mode (aka permissions) as both its **octal value and its symbolic representation**. The symbolic representation will be the `rwx` string for user, group, and other. **See** Error! Reference source not found. 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** Error! Reference source not found. for how this should look.
8. File size, in bytes.
9. Blocks allocated.
10. Show the three time values in local time/date. This will be pretty darn easy if you read through the list of suggested function calls. See **Error! Reference source not found.** for how these need to look.

Nearly all of the above data just values in the `struct stat` structure (from the `sys/stat.h` include file). The data not directly from that structure are derived from data in that structure.



```
~/Classes/cs344/Labs/Lab5
[chaneyr@os1 # d -d F*
lrwxrwxrwx. 1 chaneyr upg9313  4 Jan 29 19:47 FUNNYbrokenlink -> JUNK
drwxrwxr-x. 2 chaneyr upg9313  0 Jan 17  2019 FUNNYdir/
lrwxrwxrwx. 1 chaneyr upg9313 12 Jan 29 19:47 FUNNYlink -> FUNNYregfile
prw-rw-r--. 1 chaneyr upg9313  0 Jan 17  2019 FUNNYpipe|
-rw-rw-r--. 1 chaneyr upg9313  0 Jan 29 21:00 FUNNYregfile
srwxrwxr-x. 1 chaneyr upg9313  0 Jan 17  2019 FUNNYsocket=
```

Figure 1: Some sample files to test your `mystat`.

Some system and function calls that I believe you will find useful include: `stat()` and `lstat()` (you really want to do “`man 2 stat`” and read that entire `man` page entry closely (`man 2 stat`), 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. You will make use of both `lstat()` and `stat()` in your code. I strongly suspect that you’ll use `lstat()` first and later in your code, call `stat()`. You don’t have any need to open the file at all, just `lstat/stat` it and peruse the meta data from the `inode` in the `struct stat` structure.

My implementation is about 280 lines long, but I have a quite a bit of dead code in my file. I have code commented out code 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 hardest portion of your code will likely be devoted to displaying the symbolic representation of the mode (aka permissions). Formatting these strings is a little *awkweird*. I suggest you create a function. Don’t worry about sticky bits or set uid/gid bits. It possible that I can be persuaded to provide quite a bit of guidance on this.

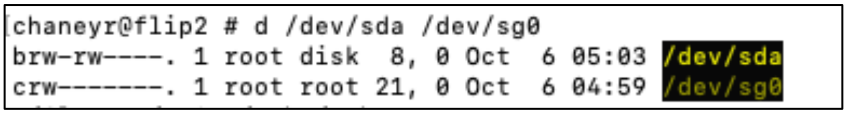
You must be able to show the following file types:

- regular file,
- directory,
- character device,
- block device,
- FIFO/pipe,
- socket, and
- symbolic link (both a good one and a broken one, see).

When formatting the human readable time for the local time, I’d suggest you consider this as a format string "`%Y-%m-%d %H:%M:%S %z (%Z) %a`", but read through the format options on `strftime()`, they are fun.

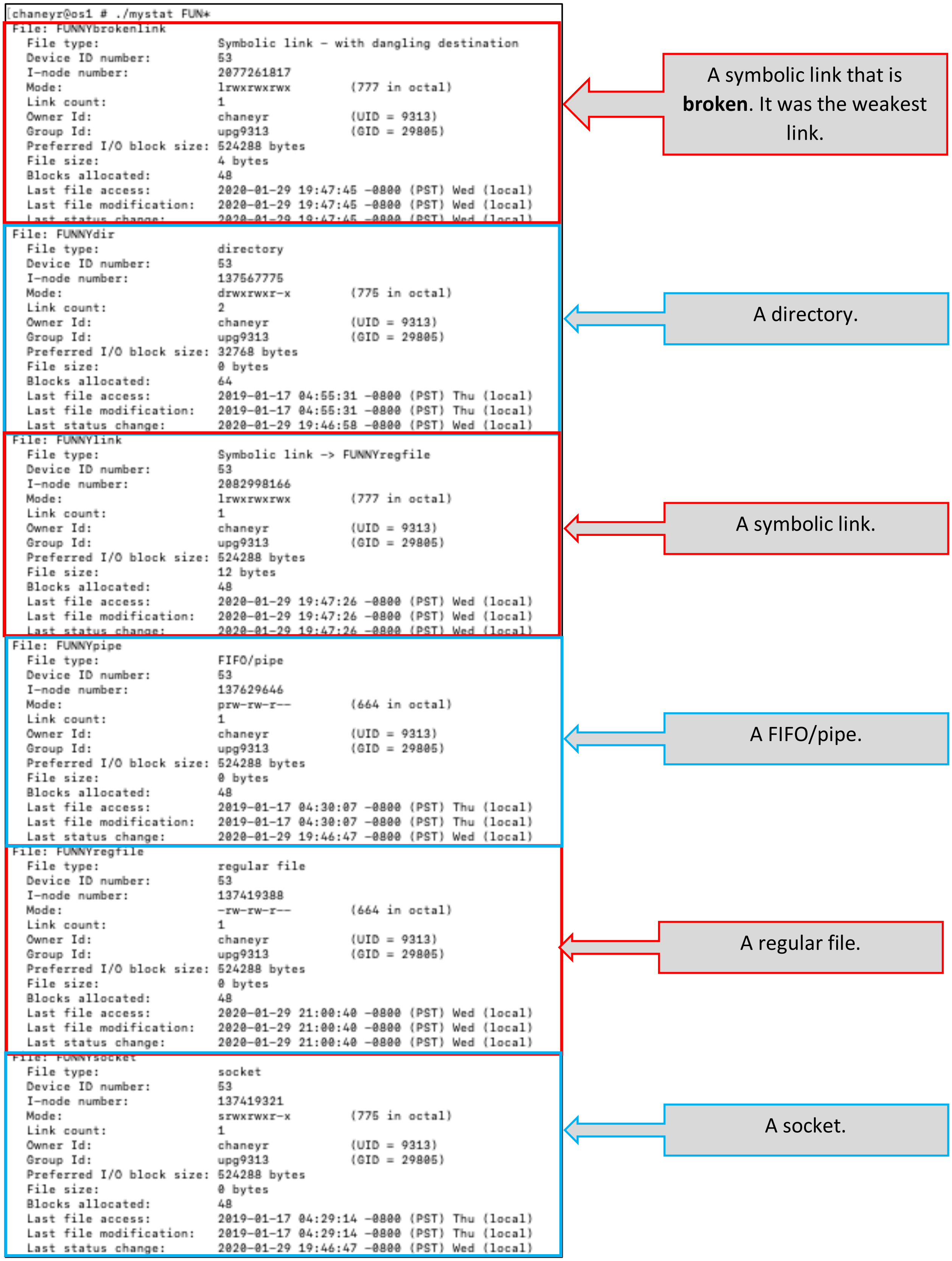
I have some examples of both a FIFO/pipe, a socket, symbolic link in my `Lab4` directory for you to use in testing (the `FUNNY*` files, **Error! Reference source not found.**). You can find a block device as `/dev/sda` and a character device as `/dev/sg0`. See Figure 2.

Another easy way to see what the output from your `mystat` should look like is to run the example I have in the `Lab4` directory. You can’t see the source code, but you certainly can look at the output by running it.



```
[chaneyr@flip2 # d /dev/sda /dev/sg0
brw-rw----. 1 root disk  8, 0 Oct  6 05:03 /dev/sda
crw-------. 1 root root 21, 0 Oct  6 04:59 /dev/sg0
```

Figure 2: Block and character files.



```
[chaneyr@os1 # ./mystat FUN*
File: FUNNYbrokenlink
  File type:                  Symbolic link - with dangling destination
  Device ID number:           53
  I-node number:              2077261817
  Mode:                       lrwxrwxrwx        (777 in octal)
  Link count:                 1
  Owner Id:                   chaneyr           (UID = 9313)
  Group Id:                   upg9313           (GID = 29805)
  Preferred I/O block size: 524288 bytes
  File size:                  4 bytes
  Blocks allocated:           48
  Last file access:           2020-01-29 19:47:45 -0800 (PST) Wed (local)
  Last file modification:     2020-01-29 19:47:45 -0800 (PST) Wed (local)
  Last status change:         2020-01-29 19:47:45 -0800 (PST) Wed (local)
File: FUNNYdir
  File type:                  directory
  Device ID number:           53
  I-node number:              137567775
  Mode:                       drwxrwxr-x        (775 in octal)
  Link count:                 2
  Owner Id:                   chaneyr           (UID = 9313)
  Group Id:                   upg9313           (GID = 29805)
  Preferred I/O block size: 32768 bytes
  File size:                  0 bytes
  Blocks allocated:           64
  Last file access:           2019-01-17 04:55:31 -0800 (PST) Thu (local)
  Last file modification:     2019-01-17 04:55:31 -0800 (PST) Thu (local)
  Last status change:         2020-01-29 19:46:58 -0800 (PST) Wed (local)
File: FUNNYlink
  File type:                  Symbolic link -> FUNNYregfile
  Device ID number:           53
  I-node number:              2082998166
  Mode:                       lrwxrwxrwx        (777 in octal)
  Link count:                 1
  Owner Id:                   chaneyr           (UID = 9313)
  Group Id:                   upg9313           (GID = 29805)
  Preferred I/O block size: 524288 bytes
  File size:                  12 bytes
  Blocks allocated:           48
  Last file access:           2020-01-29 19:47:26 -0800 (PST) Wed (local)
  Last file modification:     2020-01-29 19:47:26 -0800 (PST) Wed (local)
  Last status change:         2020-01-29 19:47:26 -0800 (PST) Wed (local)
File: FUNNYpipe
  File type:                  FIFO/pipe
  Device ID number:           53
  I-node number:              137629646
  Mode:                       prw-rw-r--        (664 in octal)
  Link count:                 1
  Owner Id:                   chaneyr           (UID = 9313)
  Group Id:                   upg9313           (GID = 29805)
  Preferred I/O block size: 524288 bytes
  File size:                  0 bytes
  Blocks allocated:           48
  Last file access:           2019-01-17 04:30:07 -0800 (PST) Thu (local)
  Last file modification:     2019-01-17 04:30:07 -0800 (PST) Thu (local)
  Last status change:         2020-01-29 19:46:47 -0800 (PST) Wed (local)
File: FUNNYregfile
  File type:                  regular file
  Device ID number:           53
  I-node number:              137419388
  Mode:                       -rw-rw-r--        (664 in octal)
  Link count:                 1
  Owner Id:                   chaneyr           (UID = 9313)
  Group Id:                   upg9313           (GID = 29805)
  Preferred I/O block size: 524288 bytes
  File size:                  0 bytes
  Blocks allocated:           48
  Last file access:           2020-01-29 21:00:40 -0800 (PST) Wed (local)
  Last file modification:     2020-01-29 21:00:40 -0800 (PST) Wed (local)
  Last status change:         2020-01-29 21:00:40 -0800 (PST) Wed (local)
File: FUNNYsocket
  File type:                  socket
  Device ID number:           53
  I-node number:              137419321
  Mode:                       srwxrwxr-x        (775 in octal)
  Link count:                 1
  Owner Id:                   chaneyr           (UID = 9313)
  Group Id:                   upg9313           (GID = 29805)
  Preferred I/O block size: 524288 bytes
  File size:                  0 bytes
  Blocks allocated:           48
  Last file access:           2019-01-17 04:29:14 -0800 (PST) Thu (local)
  Last file modification:     2019-01-17 04:29:14 -0800 (PST) Thu (local)
  Last status change:         2020-01-29 19:46:47 -0800 (PST) Wed (local)
```

A symbolic link that is **broken**. It was the weakest link.

A directory.

A symbolic link.

A FIFO/pipe.

A regular file.

A socket.

Figure 3: Examples of running `mystat` on the `FUNNY*` files

Part 2 – `csv2bin` – Create a binary file (75 Points)

For the part 2 of the assignment, you will write a C program and will read a text file (a csv file) and then, from the csv data, create a binary file that conforms to the data structure given in `~chaneyr/Classes/cs344/Labs/Lab4` directory, `file_struct.h`. **Do not modify the `.h` include in any way**. You can create a symbolic link to it or make a copy of it, but do not modify it. When I grade your program, I will first delete any copy of `file_struct.h` you may have included with your submission and then link to the correct one.

You need to open a csv data file as input (or read csv data from `stdin`), use the example `csv` files, and parse the contents into the relevant fields (are we `strtok`-in yet?). The first row of the input file contains header information. You do not put that into the resultant binary data file. As you parse a line from the input file, you will populate an example of the structure from the `file_struct.h` file write it to output. **You will write the entire structure (all of it at once) into the output data file.** You will do that for each and every data record from the `csv` file. You will be creating a binary file that holds the same data as the `csv` file. As you copy data from the `csv` file into the `bin_file_t` structure, make sure you don't leave any chaff from earlier iterations (thoughts of `memset()` should dance through your mind). The output from your `csv2bin` program must match exactly the output from my `csv2bin` program (for a given csv file input).

```
typedef struct bin_file_s {
  char id[..];
  char fname[..];
  char mname[..];
  char lname[..];
  char street[..];
  char city[..];
  char zip[..];
  char country_code[..];
  char email[..];
  char phone[..];
} bin_file_t;
```

My implementation is about 150 lines long. I have to admit that I wrote a pretty kick-butte macro that shortened my source file. The extra diagnostic messages in mine (available from the `-v` option) makes my code a good deal longer. Like the `mystat` program, there is no complex logic for this application.

Additional requirements common to parts 2 and 3 are listed following Part 3.

Part 3 – `bin2csv` – Create a `csv` file (75 Points)

For the third part of the assignment, you need to write a program that reads the binary output from the Part 2 program and creates a `csv` file, like you used as input for the second part. When you do it right, you should be able to test your code by comparing the output from the Part 3 program with the original input to Part 2. They should be the same (no differences). You will need to match the header strings from the original file. The output from your `bin2csv` program must match exactly the input from the `csvsbin` program.

Additional requirements common to parts 2 and 3 are listed after part 3.

My implementation is about 140 lines long. My kick-butte macro did not save me any lines, but it does save me a few characters. Diagnostic messages in mine make it longer. Like the `mystat` and `csv2bin` programs, there is no complex logic for this application.

Requirements Common for Parts 2 and 3

Make sure your C code does not have any memory leaks. Remember `valgrind`? This should be really easy, as I see no need to allocate memory from the heap in these programs.

Both parts 2 & 3 must make use of the unaltered `file_struct.h` file.

When working with the binary data files (output from part 2 and input for part 3), you must use the `read()` and `write()` functions. Along with `read()` and `write()`, you will use `open()` and `close()` (not their `f*()` cousins). **If you do not use the `read()` and `write()` functions for those sections of your code, I will simply give you a zero for parts 2 and 3**. Use of `fread()` or `fwrite()` for binary files will also result in a zero. You can (and should) use `printf()`, `fprintf()`, and `fgets()` when working with text data (the `csv` file input to part 2 or the `csv` output from part 3) or diagnostic messages (stuff to `stderr`).

Both parts 2 and 3 (`csv2bin` and `bin2csv`) must accept the following command line options (check the header file for these):

Option	Description
`-i <file>`	• The name of the file to be used as input to the program. • If the `-i` option (with its file name argument) is not given on the command line, then the program should read all data from `stdin`. • It a file name is given (using the `-f` option with an argument), and the program cannot open the file, it should print an error message and exit with a value 2.
`-o <file>`	• The name of the output file. • If the `-o` option (with its file name argument) is not given on the command, the program should send all data output to `stdout`. • It a file name is given, and the program cannot open the file, it should print an error message and exit with a value 3.
`-h`	Show the help text and exit (with a value 0). Have something reasonable for this, like a listing of all the command line options. You could run my program and see what it does (mine are kind of lame). Your program should not perform any actions on input, simply show the help text and exit.
`-v`	Verbose processing. This is really to help you follow what your code is doing. You need to accept this switch, have your code emit some diagnostics with this set. Look at what my code produces. Run my program to get an idea for how this can look. **All the messages from the verbose output must be sent to `stderr`, NOT `stdout`.**

If an illegal command line option is passed to the program (such as `-J`), it should print an error message to `stderr` and exit with an exit value of `1`. If the program runs successfully, its exit/return value should be `0` (which is `EXIT_SUCCESS`).

Of course, the Part 1, and Part 2 programs must be written in C. The source from Part 2 must be named `csv2bin.c` and the program `csv2bin`. The source from Part 3 must be named `bin2csv.c` and the program bin2csv. Your single unified Makefile must build all parts of the assignment. See Part 4 for more information about the Makefile.

When done, you should be able to set up your programs as filters.

```
      cat SampleDataSmall.csv | ./csv2bin | ./bin2csv > SampleData_test.csv
```

or

```
      ./csv2bin < SampleDataSmall.csv | ./bin2csv > SampleData_test.csv
```

or

```
      ./csv2bin -iSampleDataSmall.csv | ./bin2csv -o SampleData_test.csv
```

Then you can simply run diff on SampleDataSmall.csv with SampleData_test.csv.

If a field from the csv file is too long to fit into its respective field in the structure, you must truncate it to length.

System/function calls that you will find especially useful include: open(), close(), read(), write(), memset(), strtok().

I have put 3 sample csv files in the Lab4 directory. I recommend you test with each of them. I have also placed the 3 bin files generated from my csv2bin program. Your bin files must exactly match my bin files.

Part 4 - The Makefile (50 points)

You must have a single Makefile that compiles all the C programs (Part 1, Part 2, and Part 3). If you do not have a Makefile that builds all programs, it will put a major dent in your grade for the assignment. Your code must compile without any errors or warnings from gcc. Do not adorn your calls to gcc with any -std=… options.

You must use the following gcc command line options in your Makefile when compiling your code (make your life easier, use variables).

```
      -Wall
      -Wshadow
      -Wunreachable-code
      -Wredundant-decls
      -Wmissing-declarations
      -Wold-style-definition
      -Wmissing-prototypes
      -Wdeclaration-after-statement
      -Wno-return-local-addr
      -Wuninitialized
      -Wunused
      -Wextra
```

Your Makefile must include the following targets:

- all – should build all out-of-date programs (parts 1, 2, and 3) and prerequisites.
- clean – clean up the compiled files **and editor droppings**.
- mystat – linking the Part 1 program, rebuilding the mystat.o file if necessary.

- mystat.o – build the mystat.o file from mystat.c file
- csv2bin – linking the Part 2 program, rebuilding the csv2bin.o file if necessary
- csv2bin.o – build the csv2bin.o file from the csv2bin.c file
- bin2csv – linking the Part 3 program, rebuilding the csv2bin.o file if necessary
- bin2csv.o – build the csv2bin.o file from the bin2csv.c file

Final note

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.**