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 consists of multiple parts. **All Parts of this lab are July 20th 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 portions of this assignment. If your `Makefile` does not build a portion, then it is a zero for that portion of the assignment.** Do yourself a favor, write your `Makefile` first and keep it up to date.

Part 1 – `reclookup` (75 Points)

Your program for part 1, written in C, will allow you to lookup records in the binary data file. Have you been thinking about `lseek()` lately? You will be.

Your lookup program must accept the following command line options:

Option	Description
`-i <file>`	The name of the file to be used as binary data input to the program. **This is a required option**. **If this option is not given on the command line, the program should exit with an error; the exit value should be 3.**
`-o <file>`	The name of the output file. If the `-o` option is not given on the command, the program should send all data output to `stdout`.
`-h`	Show the help text **and exit**. Have something reasonable for this, like a listing of all the command line options. The exit value should be 0.
	If an invalid command line option is given, print an error message to `stdout` and `exit` with a value 7.

All items on the command line that follow the command line options are record numbers that should be looked up in the input binary data file. Do not try to fit the entire input data file in memory. Do not open and begin a scan of the input file for each record number. Record numbers start at zero and go up from there. **You must use `lseek()` to find the records in the file.** The initial record in the binary input file is record 0. Record numbers will not be given in ascending order. They can be given in any order.

You should allow for any record numbers to be given on the command line. In my implementation, if there are no record numbers are given on the command line, I read them from `stdin`, one record number per line. **Consider this an opportunity for even extra 10% credit.**

The output from your program must look exactly like this for each record number located in the data file (**your output must EXACTLY match mine**):

```
# ./reclookup -i SampleData.bin 8

id: 13-7914896
 first_name: Jimmy
 middle_name: Tammy
 last_name: Roberts
 street: 69091 Ridgeview Circle
 city: Halmstad
 zip: 302 55
 country: SE
 email: troberts0@marketwatch.com
 phone: 46-(824)628-4077
```

My code for the `reclookup` program is about 210 lines long, but I have a bunch extra fluff in there. The function calls from my code that may be new: `lseek()`, `memset()`, `isblank()`, `strtol()`. You might not need all of those.

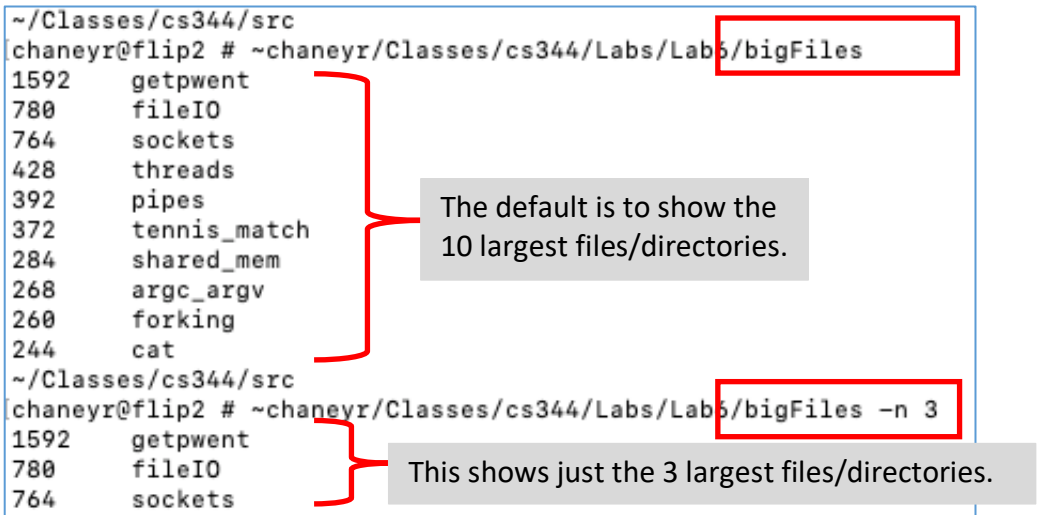
You must use the `read()` function to read from the binary input file. Neglect to do this will have a significant effect on the grade for your assignment, in a bad way (something like a zero). You can (and should) use the `fopen()`/`fprintf()` functions for output of the text data.

Part 2 – `bigFiles` (100 Points)

You are going to write a program called `bigFiles`. The program `bigFiles` will behave as the following `bash` command line behaves:

```
ls | xargs du -s | sort -nr | head [-n #]
```

the output of the `bigFiles` program will show (by default) the 10 largest files or directories in the current directory. When checking a directory, it shows the sum of all files in that directory. This will be a lot like the `zhead`/`ztail` program that we did in class. One difference you'll see right away is that `bigFiles` is a **4 stage pipeline** while `zhead` is a 2 stage pipeline. **You will need to manage the pipes you create carefully**. Remember what happens to your program when you neglect to close unused ends on a pipe? **Block Party!!!** Examples of running `bigFiles` are below.



The `bigFiles` program has a single command line option, `-n #`. If given, it represents the number of file/directories to show. If the `-n #` command line is not given, it uses the default for `head`, which is 10 (which you can find in the man page for head).

Don't blow a gasket fretting about how to do this assignment. You understand how to do `zhead` program, you are 73.62% done. This is just a little more complex. Start by understanding what the pipelined command does. I can think of 2 basic ways to do this: the easy way and the cowboy way. The cowboy way will help you with a future lab (i.e. `Lab6`), the easy way gets you done sooner. I did it the easy way, in about 45 minutes. A rough outline of what I did (the easy way):

1. Process `argv` (using my new bff, `getopt`).
2. Open 3 sets of pipes (a set of pipes will be used between each of the 4 stages in the pipeline). This is easily done with the `pipe()` function.

3. `fork()` a new process. For me, this is the `ls` process. Setting of the pipe for this one is pretty easy, just one pipe on which to call `dup2()`. The pipe is on the output side. Close ALL the pipes, after the call to `dup2()` and before the call to the exec function. Call an exec function on `ls`.
4. `fork()` a new process. For me, this is the `xargs` process. Setting up the pipe is a bit more involved, because this is an interior process; so, it has a pipe on each side. There are 2 `dup2()` calls in this. Close ALL the pipes, after the calls to `dup2()` and before the call to the `exec` function. Call an exec function on `xargs`.
5. `fork()` a new process. For me, this is the `sort` process. The pipes in this stage are like the pipes in step 4. Close ALL the pipes, after the calls to `dup2()` and before the call to the `exec` function. Call an `exec` function on the `sort` command.
6. We're done forking around. This is my `head` process. Though it is the parent process of the other processes (`ls`, `xargs`, and `sort`), it is last in the pipeline. Since this process is on the end, it makes use of just one pipe (as the `ls` process does, step 3). The pipe is on the input side. C Close ALL the pipes, after the call to `dup2()` and before the call to the `exec` function. all an `exec` function on head.
7. Since all processes call an exec function, I don't have a parent process that needs to call `wait()` on the child processes.
8. I make sure I call `perror()` and `_exit()` on statements that follow each call to an `exec` function. If the `exec` function fails, I want the process to print an error message and exit immediately.
9. My code does not have any loops (except for the `getopt()` loop), just a BUNCH of straight line code that does forking and exec-ing. I made 3 calls with `execlp()` and 1 with `execvp()`.

Expect somewhere around 150 lines of C code, if you do it the easy way.

You will need to be happy with the following functions: `pipe()`, `fork()`, `dup2()`, and the `exec()` friends.

Part 3 – BennySh – baby BennySh (175 Points)

This is Stage 1 of a **2-week assignment**. It will be completed in 2 stages: Stage 1 and Stage 2.

- Stage 1 is due on **July 20th**. Stage 1 should be reusable as the beginning of Stage 2.
- Stage 2 is due on **July 27th**. Stage 2 is the more challenging portion of the assignment. There will be more information about Stage 2 next week.

In this portion assignment (Stage 1), you will be creating your own UNIX shell. You've heard about `bash`, `csh`, `sh`, `tcsh`, `zsh`, and others. Now you will write `BennySh`. It will not be as capable as the other shells, but it will be yours to improve.

Your shell, `BennySh` (and it must be called that), will process 2 different classes of commands: built-in commands and external commands. The built-in commands are shown in the table below. The built-in commands will always appear on the command line without any other commands and without any form of pipe or redirection.

Built-in Commands	Action
`exit`	The exit built-in command will exit `BennySh`. If the user types a `Control-D` at the command line (as the first character on the line), the result will be the same as the `exit` build-in command.
`cd` `cd <dir>`	The `cd` built-in has 2 forms, one identifies the target directory, and the other does not. If a target directory is given, change the current working directory of the shell to the specified target directory. If no target directory is given, change to the **user's home directory** (just as it does in the other shells).
`pwd`	Print the present working directory of the shell. You can look at the `man` page for `pwd` to see how this work. The version you develop for your `BennySh` does not take any command line options.
`echo`	Display back to the user the value that follows the `echo` command. Works like the `man` page for `echo` describes, but does not expand any variables in the output, simply echo back the text.

Table 1: Table of built-in commands for your `BennySh`.

The external commands are all of the other commands available in Linux (`cat`, `emacs`, `ls`, `wc`, …). The external commands should be able to use their normal set of command line options and arguments. You need to `fork()` and `exec()` in your shell for the external commands to work.

The (happy) prompt presented to the user of your shell must look like the following:

```
BennySh user-name :-)
```

where "`user-name`" is the Linux account name of the person running your `BennySh`. For example, when I am running your `BennySh`, the prompt I should see is:

```
BennySh chaneyr :-)
```

When you run your `BennySh`, it will show your login name, not mine. Consider looking in the environment variables for something that contains that log name of the current user.

Because your `BennySh` will run for many commands over its lifetime, you need to make sure it has zero memory leaks. Run and test your shell thoroughly using `valgrind` to identify and eliminate all memory leaks before submitting the assignment.

Your Stage 1 requirements are:

1. Properly handle all the built-in commands, shown in Table 1.
2. External commands will consist of one single command with any number of (correct) command line options and arguments, **with optional redirection of either `stdin` or `stdout`.** For example:
   - `ls -l -aFg -r`
   - `df -hm`
   - `who -H`
3. External commands must allow for redirection of either or both of `stdin` and/or `stdout`. The characters < and > will always appear **with one or more spaces around them**. For example:

- `ls -l > file1.txt`
- `who -H > file2.txt`
- `wc < file1.txt`
- `wc < file1.txt > file2.txt`

I will not give commands such as these:

- `ls -l >file1.txt`
- `who -H>file2.txt`
- `wc<file1.txt`
- `wc<file1.txt>file2.txt`

4. No memory leaks.
5. No zombies. No orphans.
6. Good programming style.
7. You must have a `Makefile` (see Part 4) which will build your `BennySh` (and the other portions of the lab).
8. Do not use the `system()` function in your code. Use of the `system()` function will result in a zero for this assignment.

NOTE: Special note: I know there is a web site out there called "Write a Shell in C." It is a crappy web site. You can do better. I do not like how it implements commands. I discourage you from leaning on that code to complete this project. I've seen students use that web site and **have it add weeks** to their development time and not be able to reuse it in the second half of the shell assignment.

Part 4 – The `Makefile` (50 points)

You must have a single `Makefile` that compiles all the programs. If you do not have a `Makefile` that builds all programs, it will put a major dent in your grade for the assignment (think zero for each part not compiled). **Be sure your code does not generate any errors or warnings when compiled**. Hunt down and fix all warnings in your code. I will deduct 20% for each warning your code produces when compiled. 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).

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

Your `Makefile` must include the following targets:

- `all` – should build all out-of-date programs and prerequisites.
- `clean` – clean up the compiled files and editor chaff.
- `reclookup` – linking the part 1 program, rebuilding the `reclookup.o` file, if necessary

- `bigFiles` – linking the part 2 program, rebuilding the `bigFiles.o` file, if necessary
- `BennySh` – linking the part 3 program, rebuilding the `BennySh.o` file, if necessary
- `reclookup.o` – compile the `reclookup.c` file, if necessary
- `bigFiles.o` – build the `bigFiles.c` file, if necessary
- `BennySh.o` – build the `BennySh.c` file, if necessary
- if your `BennySh` is built from multiple C files, you'll need to have additional targets to support the additional C files.

Note

In my `Lab5` directory, there are some source files that you might find helpful: `cmd_parse.c`, `cmd_parse.h`, and `parse_example.c`. They will chop up the command you issue within your program into a nice convenient linked list. It has some other handy things built into it the code as well. One thing the code does not do is deallocate the memory for the linked lists. That is a task left to the user. You are welcome to use the code as you may find helpful. Spending some time getting to understand the code will probably help, a LOT.

I really urge you to try running the `parse_example` with the -v option and then enter a few commands. It will give you some really good ideas about how it works.

There is also a file called `bennysh_TestCommands.txt` than contains several examples of valid command lines to test your `BennySh` program.

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.**