Project 1: Mini Unix Utilities

Introduction

This project should be programmed entirely in C and must compile and run on standard Linux environments, e.g., Ubuntu. (The labs and UX use Ubuntu 22.04 LTS.) Copying & pasting code from other teams or from the web is considered cheating and results in an automatic F.

In this assignment, you will build a set of simplified Linux utilities like **cat**, **sed**, and **uniq**. We say simplified because the original version are quite complicated. For example, the source of the **cat** command – which concatenates files and writes out to standard output – is around 700 lines. We will use slightly different names for the utilities in this assignment to avoid confusion - **my-cat**, **my-sed**, and **my-uniq**.

You MAY NOT use the original Linux commands (i.e., *cat*, *sed*, and *uniq*) for the implementation of *my-cat*, *my-sed*, and *my-uniq*. Doing so will result in an automatic F.

Objectives

- Re-familiarize yourself with the C programming language
- Re-familiarize yourself with a shell/terminal/command-line of UNIX
- Learn (as a side effect) how to use a proper code editor not necessarily emacs/vim
- Learn a little about how UNIX utilities are implemented

While the project focuses upon writing simple C programs, you can see from the above that even that requires a bunch of other previous knowledge, including a basic idea of what a shell is and how to use the command line on some UNIX-based systems (e.g., Linux or macOS), how to use an editor, and of course a basic understanding of C programming. If you do not have these skills already, this is not the right place to start.

Summary of what gets turned in:

- Bunch of .c files, one each for a utility: **my-cat.c**, **my-sed.c**, **my-uniq.c**.
  - Each should compile successfully with the **-Wall** and **-Werror** flags.
  - Each should (hopefully) pass tests we supply.
- You must supply a **Makefile** to compile your source.
- Include a single README.md for all the files describing your implementation.
- Assume that input files for each utility can be larger than RAM available on system.

1. my-cat

The program **my-cat** is a simple program. Generally, it reads a file as specified by the user and prints its contents. A typical usage is as follows, in which the user wants to see the contents of *main.c*, and thus types:

```
prompt> ./my-cat main.c
#include <stdio.h> ...
```

As shown, **my-cat** reads the file *main.c* and prints out its contents. The "./" before the **my-cat** above is a UNIX thing; it just tells the system which directory to find **my-cat** in (in this case, in the "." (dot) directory, which means the current working directory).

To create the **my-cat** binary, youll be creating a single source file, **my-cat.c**, and writing a little C code to implement this simplified version of **cat**. To compile this program, you will do the following:

```
prompt> gcc -o my-cat my-cat.c -Wall -Werror
```

This will make a single executable binary called **my-cat** which you can then run as above.

You'll need to learn how to use a few library routines from the C standard library (often called **libc**) to implement the source code for this program, which we'll assume is in a file called **my-cat.c**. All C code is automatically linked with the C library, which is full of useful functions you can call to implement your program.

For this project, we recommend using the following routines to do file input and output: **fopen()**, **fgets()**, and **fclose()**. Whenever you use a new function like this, the first thing you should do is read about it – how else will you learn to use it properly?

On UNIX systems, the best way to read about such functions is to use what are called the **man** pages (short for **manual**). In our HTML/web-driven world, the man pages feel a bit antiquated, but they are useful and informative and generally quite easy to use.

To access the man page for **fopen()**, for example, just type the following at your UNIX shell prompt:

```
prompt> man fopen
```

Then, read! Reading man pages effectively takes practice; why not start learning now?

We will also give a simple overview here. The fopen() function "opens" a file, which is a common way in UNIX systems to begin the process of file access. In this case, opening a file just gives you back a pointer to a structure of type **FILE**, which can then be passed to other routines to read, write, etc.

Here is a typical usage of **fopen()**:

```
FILE *fp = fopen("main.c", "r");
if (fp == NULL) {
    printf(``cannot open file \n'');
    exit(1);
}
```

A couple of points here. First, note that **fopen()** takes two arguments: the *name* of the file and the *mode*. The latter just indicates what we plan to do with the file. In this case, because we wish to read the file, we pass "r" as the second argument. Read the man pages to see what other options are available.

Second, note the critical checking of whether the **fopen()** actually succeeded. This is not Java where an exception will be thrown when things goes wrong; rather, it is C, and it is expected (in good programs, i.e., the only kind you'd want to write) that you always will check if the call succeeded. Reading the man page tells you the details of what is returned when an error is encountered; in this case, the macOS man page says:

Upon successful completion fopen(), fdopen(), freopen() and fmemopen() return a FILE pointer. Otherwise, NULL is returned and the global variable errno is set to indicate the error.

Thus, as the code above does, please check that **fopen()** does not return NULL before trying to use the FILE pointer it returns.

Third, note that when the error case occurs, the program prints a message and then exits with error status of 1. In UNIX systems, it is traditional to return 0 upon success, and non-zero upon failure. Here, we will use 1 to indicate failure.

Side note: if **fopen()** does fail, there are many reasons possible as to why. You can use the functions **perror()** or **strerror()** to print out more about why the error occurred; learn about those on your own (using ... you guessed it ... the man pages!).

Once a file is open, there are many different ways to read from it. We suggest you use **fgets()** for this part of the project. **fgets()** is used to get input from files, one line at a time.

To print out file contents, just use printf(). For example, after reading in a line with **fgets()** into a variable buffer, you can just print out the buffer as follows:

```
printf("\%s", buffer);
```

Note that you should not add a newline (\n) character to the printf(), because that would be changing the output of the file to have extra newlines. Just print the exact contents of the read-in buffer (which, of course, many include a newline).

Finally, when you are done reading and printing, use **fclose()** to close the file (thus indicating you no longer need to read from it).

More Details:

- Your program **my-cat** can be invoked with one or more files on the command line; it should just print out each file in turn.
- In all non-error cases, **my-cat** should exit with status code 0, usually by returning a 0 from *main()* (or by calling **exit(0)**).
- If no files are specified on the command line, **my-cat** should just exit and return 0. Note that this is slightly different than the behavior of normal UNIX cat (if you'd like to, figure out the difference).
- If the program tries to **fopen()** a file and fails, it should print the exact message "my-cat: cannot open file" (followed by a newline) and exit with status code 1. If multiple files are specified on the command line, the files should be printed out in order until the end of the file list is reached or an error opening a file is reached (at which point the error message is printed and **my-cat** exits).

2. my-sed

The second utility you will build is called **my-sed**, a variant of the GNU/Linux utility called **sed** or streaming editor. For all purposes **sed** is the command line version of find and replace, the power of sed comes from matching regular expressions. Using regexs you don't just match strings but any possible pattern. But thankfully **my-sed** will only be used to find and replace it with the exact given string. It will find the *first* instance of a string in a line and substitute it with another. It will print the output to standard output. Instances following the first instance remain as is. My-sed does not change input files.

In case there is no candidate to substitute the line will be printed as is. Here is how a user will substitute word the *foo* with *bar* from *baz.txt* and *qux.txt*.

```
prompt>./my-sed foo bar baz.txt qux.txt
```

More details:

- Your program is passed a find term, a replace term, and arbitrary number of input files.
- If an empty string (`""`) is specified as the replace term, then instead of replacing, the search term should be deleted (don't substitute a space).
- The matching is case sensitive. So **foo** will not match with **Foo**.
- The find and replace terms will not contain any escape characters like \n or \0.

- The lines can be arbitrarily long (that is, you may see many many characters before you encounter a newline character, \n). **my-sed** should work as expected even with very long lines. For this, you might want to look into the **getline()** library call (instead of **fgets()**), or roll your own.
- If the **my-sed** is passed with no command line arguments or if any of the mandatory arguments are missing it should print "my-sed: find_term replace_term [file ...]" (followed by a newline) and exit with status code 1. Example of such inputs –

  ```
  prompt> ./my-sed
  prompt> ./my-sed foo
  ```

- If **my-sed** encounters a file that it can't open it should print "my-sed: cannot open file" (followed by a newline) and exit with status code 1 immediately.
- If no input file is specified, **my-sed** should work but instead of reading from a file, **my-sed** should read from standard input. Doing so is easy because the file stream stdin is already open; you can use **fgets()** (or similar routines) to read from it.

Following are different ways **my-sed** can be executed:

- In the following case term **foo** will be removed from **bar.txt**.

  ```
  prompt> ./my-sed foo "" bar.txt
  ```

- In the following case word **foo** will be replaced with **bar.txt** taking the input from stdin.

  ```
  prompt> ./my-sed foo bar.txt
  ```

- In the following case replace **foo.txt** with **bar.txt** in **baz.txt**.

  ```
  prompt>./my-sed foo.txt bar.txt baz.txt
  ```

- In the following case replace **foo.txt** with **bar.txt** taking input from stdin.

  ```
  prompt> ./my-sed foo.txt bar.txt
  ```

- On the other hand if the input to **my-sed** is the following

  ```
  prompt> ./my-sed foo.txt
  ```

  The output should be–

  ```
  my-sed: find_term replace_term [file ...]
  ```

- The above examples should have given you an idea exactly when your program should throw an error.

3. my-uniq

The last (phew!) utility you will build is called **my-uniq**, a version of the Unix utility **uniq** (it should be pretty obvious by now). The original Unix **uniq** utility detects adjacent duplicate lines from the input file and removes them and writes out to an output file. **my-uniq** will do exactly this, it finds out adjacent duplicate lines in files, and prints them.

For example, consider the following command:

```
prompt>./my-uniq foo.txt bar.txt
```

This will read contents from file **foo.txt** and **bar.txt**, delete adjacent duplicate lines from each of them and print the output to shell.

More details:

- If **my-uniq** encounters a file that it can't open it should print. "my-uniq: cannot open file" (followed by a newline) and exit with status code 1 immediately.
- **my-uniq** should accept an arbitrary number of files as input and print the output to standard output. However, it should only detect duplicates within each file. For example, if the first line in the second file is the same as the last line in the first file, it should not be removed.
- The input file can be really large as well as each line can be arbitrarily long.
- The input file should not be modified.
- If no files are specified, **my-uniq** should read from standard input.

Deliverables

- Please submit your deliverables using Canvas before 11:59pm on the due date.
- Please include all files (including Makefiles and source code) necessary for a successful build as a zip file.
- The name of your submission should be *cs460-project1.zip*.

Grading

Your submission will be graded by compiling and running it.

- Please make sure your source code can compile. Absolutely no credit if it does not compile.
- Please make sure your programs run ocan standard Linux servers.
- Please don't include the binary files. Do a make clean before submission.
- Please don't leave out any files!
- Please don't modify any files you don't need to!
- Please don't send us the meta-information from your revision control system!