CPSC 3500 COMPUTING SYSTEMS

PROJECT 1: BUILDING SIMPLE UNIX UTILITIES

1. Assignment Requirement

In this project, you'll build a few different UNIX utilities, simple versions of commonly used commands like **cat**, **ls**, etc. We'll call each of them a slightly different name to avoid confusion; for example, instead of **cat**, you'll be implementing **wcat** (i.e., "washington" cat).

Objectives:

- Re-familiarize yourself with the C/C++ programming language
- Re-familiarize yourself with a shell / terminal / command-line of UNIX
- Learn a little about how UNIX utilities are implemented
- If you want to Re- familiarize yourself with command line arguments example, then you can refer this link geeks for geeks .

While the project focuses upon writing simple C/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 such as emacs, and of course a basic understanding of C/C++ programming. If you **do not** have these skills already, this is not the right place to start.

Summary of what gets turned in:

- A bunch of single .c (or .cpp) files for each of the utilities below: **wcat.c**, **wgrep.c**, **wzip.c**, and **wunzip.c**.
- Each should compile successfully when compiled with the **-Wall** and **-Werror** flags.
- A Makefile that compiles all the source files and produces the corresponding executables.

If you want to re-familiarize yourself with cat, grep, zip, unzip commands:

- https://man7.org/linux/man-pages/man1/cat.1.html
- https://man7.org/linux/man-pages/man1/grep.1.html
- https://www.unix.com/man-page/v7/1/zip/
- https://linux.die.net/man/1/unzip

1.1 wcat

The program **wcat** 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> ./wcat main.c
#include <stdio.h> ...
```

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

To create the **wcat** binary, you'll be creating a single source file, **wcat.c**, and writing a little C code (or C++ code) to implement this simplified version of **cat**. Table 1 shows the recommended system calls to do file I/O. 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
```

Specifically, **wcat** should meet the following requirements:

- Your program **wcat** should display (e.g., **printf()**) the exact contents of the file(s).
- Your program **wcat** can be invoked with one or more files on the command line; it should just print out each file in turn.
- If **wcat** is passed no command-line arguments, it should print "wcat: file [file ...]" (followed by a newline) and exit with status 1.
- In all non-error cases, **wcat** should exit with status code 0, usually by returning a 0 from **main()** (or by calling **exit(0)**).
- If the program tries to **fopen()** a file and fails, it should print the exact message "wcat: 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 **wcat** exits).

1.2 wgrep

The second utility you will build is called **wgrep**, a variant of the UNIX tool **grep**. This tool looks through a

file, line by line, trying to find a user-specified search term in the line. If a line has the word within it, the line is printed out, otherwise it is not.

Here is how a user would look for the term **foo** in the file **bar.txt**:

```
prompt> ./wgrep foo bar.txt this line ha
foo in it so does this foolish line; do yo
see where
even this line, which has barfood in it, will be printed.
```

Specifically, **wgrep** should meet the following requirements:

- Your program **wgrep** is always passed a search term and zero or more files to grep through (thus, more than one is possible). It should go through each line and see if the search term is in it; if so, the line should be printed, and if not, the line should be skipped.
- The matching is case sensitive. Thus, if searching for **foo**, lines with **Foo** will *not* match.
- Lines can be arbitrarily long (that is, you may see many many characters before you encounter a newline character, \n). **wgrep** 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 **wgrep** is passed no command-line arguments, it should print "wgrep: searchterm [file ...]" (followed by a newline) and exit with status 1.
- If **wgrep** encounters a file that it cannot open, it should print "wgrep: cannot open file" (followed by a newline) and exit with status 1.
- In all other cases, **wgrep** should exit with return code 0.
- If a search term, but no file, is specified, **wgrep** should work, but instead of reading from a file, **wgrep** 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.
- For simplicity, if passed the empty string as a search string, **wgrep** can either match NO lines or match ALL lines, both are acceptable.
- If a search term consists of multiple words, the wgrep should work as follows:

```
prompt> ./wgrep “foolish line” bar.txt so
does this foolish line; do you see where?
```

1.3 wzip and wunzip

The next tools you will build come in a pair, because one (**wzip**) is a file compression tool, and the other (**wunzip**) is a file decompression tool.

The type of compression used here is a simple form of compression called *run-length encoding* (*RLE*). RLE is quite simple: when you encounter **n** characters of the same type in a row, the compression tool (**wzip**) will turn that into the number **n** and a single instance of the character.

Thus, if we had a file with the following contents:

```
aaaaaaaaaabbbb
```

the tool would turn it (logically) into:

```
10a4b
```

However, the exact format of the compressed file is quite important; here, you will write out a **4-byte integer in binary format** followed by the single character in ASCII. Thus, a compressed file will consist of

some number of 5-byte entries, each of which is comprised of a 4-byte integer (the run length) and the single character.

To write out an integer in binary format (not ASCII), you should use **fwrite()**. Read the man page for more details. For **wzip**, all output should be written to standard output (the **stdout** file stream, which, as with **stdin**, is already open when the program starts running).

Note that typical usage of the **wzip** tool would thus use shell redirection in order to write the compressed output to a file. For example, to compress the file **file.txt** into a (hopefully smaller) **file.z**, you would type:

```
prompt> ./wzip file.txt > file.z
```

The "greater than" sign is a UNIX shell redirection; in this case, it ensures that the output from **wzip** is written to the file **file.z** (instead of being printed to the screen).

The **wunzip** tool simply does the reverse of the **wzip** tool, taking in a compressed file and writing (to standard output again) the uncompressed results. For example, to see the contents of **file.txt**, you would type: `prompt> ./wunzip file.z`

wunzip should read in the compressed file (likely using **fread()**) and print out the uncompressed output to standard output using **printf()**.

Specifically, **wzip and wunzip** should meet the following requirements:

- Correct invocation should pass one or more files via the command line to the program; if no files are specified, the program should exit with return code 1 and print "wzip: file1 [file2 ...]" (followed by a newline) or "wunzip: file1 [file2 ...]" (followed by a newline) for **wzip** and **wunzip** respectively.
- The format of the compressed file must match the description above exactly (a 4-byte integer followed by a character for each run).
- If **wzip** encounters a file that it cannot open, it should print "wzip: cannot open file" (followed by a newline) and exit with status 1.
- If **wunzip** encounters a file that it cannot open, it should print "wunzip: cannot open file" (followed by a newline) and exit with status 1.
- Do note that if multiple files are passed to **wzip*, they are compressed into a single compressed output, and when unzipped, will turn into a single uncompressed stream of text (thus, the information that multiple files were originally input into **wzip** is lost). The same thing holds for **wunzip**.

1.4 Summary

In this project, you will build 4 utilities: **wcat**, **wgrep**, **wzip** and **wunzip**, which requires 4 corresponding C/C++ source files. You will also need to create a **Makefile** to compile your source code and build the four executables named wcat, wgrep, wzip and wunzip, respectively.

Each utility program should behave as specified including command-line parameters, inputs & outputs, error messages and return codes.