CS 451 – Operating Systems

Project 1

**Due: 11:59 PM on Monday, 3 August 2020**

# Objectives

There are five objectives to this assignment:

- To familiarize yourself with the Linux programming environment.
- To develop your defensive programming skills in C.
- To gain exposure to the necessary functionality in shells.
- To learn how processes are handled (i.e., starting and waiting for their termination).

# Overview

In this assignment, you will implement a command line interpreter or shell (**`my-shell.c`**). The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, **but much simpler than**, the one you run every day in Unix. You can find out which shell you are running by typing **`echo $SHELL`** at a prompt. You may then wish to look at the man pages for **sh** or the shell you are running (more likely `tcsh` or `bash`) to learn more about all of the functionality that can be present. For this project, you do not need to implement much functionality; but you will need to be able to handle running multiple commands simultaneously.

To exit the shell, the user can type quit. This should just exit the shell and be done with it (the exit() system call will be useful here). Note that quit is a **built-in** shell command; it is not to be executed like other programs the user types in. If the "**quit**" command is on the same line with other commands, you should ensure that the other commands execute (and finish) before you exit your shell.

# Program Specifications

## Basic Shell

Your basic shell is basically an interactive loop: it repeatedly prints a prompt "**`myshell>`** " (note the space after the sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types "**`quit`**". The name of your final executable should be **myshell**:

```
$ ./myshell
myshell>
```

To compile this program, you will do the following:

```
$ gcc -o myshell myshell.c -Wall -Werror
myshell>
```

You should structure your shell such that it creates a new process for each new command (there are a few exceptions to this, which we will discuss below). There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows for concurrency; that is, multiple commands can be started and allowed to execute simultaneously. The maximum length of a line of input to the shell is 512 bytes.

## Unix-Utilities to be implemented

In this assignment, you will build a set of Linux utilities but much simpler versions of commonly used commands like ls, cat etc. We say simpler because to be mildly put the original are quite complicated. For ex - you can find the source of cat command which concatenates files and writes out to standard output is around 700 lines. We will call each of these utilities slightly different to avoid confusion - **`my-cat`**, **`my-uniq`**, **`my-wc.`**

### `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:

```
$ ./my-cat foo.txt
This is a foo test file ...
```

As shown, **my-cat** reads the file **foo.txt** 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, you'll 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:

```
$ gcc -o my-cat my-cat.c -Wall -Werror
```

**Details**

- **my-cat** should accept file name as input and print the output to the console.
- If **my-cat** encounters a file that it can't open it should print. "my-cat: cannot open filename" (followed by a newline) and exit with status code 1 immediately.
- The input file can be really large as well as each line can be arbitrarily long [**max 1024**].
- The input file should not be modified.

## my-uniq

The second 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, removes the duplicates, and prints the output.

For example, consider the following command:

```
$ ./my-uniq foo.txt
```

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

**Details**

- If **my-uniq** encounters a file that it can't open it should print. "my-uniq: cannot open filename" (followed by a newline) and exit with status code 1 immediately.
- **my-uniq** should accept file name as input and print the output to the console. It should delete duplicates within the file.
- The input file can be really large as well as each line can be arbitrarily long [**max 1024**].
- The input file should not be modified.

## my-wc

The third utility you will build is called **my-wc**, a version of the unix utility wc (it should be pretty obvious by now). The original unix wc command counts the words, newlines, or bytes of each input file, and outputs the result. But **my-wc** will only be used to count the newlines and words of each input file.

**Details**

- If **my-wc** encounters a file that it can't open it should print. "my-wc: cannot open file" (followed by a newline) and exit with status code 1 immediately.
- The input file can be really large as well as each line can be arbitrarily long [**max 1024**].
- The input file should not be modified.

# pipes

The last (**phew!**) you will implement is handling pipes. A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character **'|'**.

Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act as input to the next command and so on. It can also be visualized as a temporary connection between two or more commands/ programs/ processes. The command line programs that do the further processing are referred to as filters.

In this assignment you will implement pipes for the 3 unix utilities you developed.

For example, the following lines are all valid and have reasonable commands specified:

```
myshell> my-cat file1.txt | my-uniq
No duplicates in this line
myshell> my-cat file2.txt | my-wc
45 (lines) 60 (words)
myshell> my-uniq file3.txt | my-wc
10 (lines) 25 (words)
myshell> my-cat file3.txt | my-uniq | my-wc
10 (lines) 25 (words)
```

# Grading

For this project, you must upload a **zip archive** to the course Blackboard assignment. This zip archive must contain the following items:

- Bunch of .c files: **myshell.c** and one each for a utility: **my-cat.c, my-wc.c, my-uniq.c** (no object files or executables, please!)
- Each should compile successfully when compiled with the **`-Wall`** and **`-Werror`** flags.
- A **Makefile** for compiling your source code. Sample make file → **Click here**
- A README file with some basic documentation about your code

# Notes

1. ***Start early***!
2. To ensure that we compile your C correctly for the demo, you will need to create a simple **makefile;** this way our scripts can just run make to compile your code with the right libraries and flags. If you don't know how to write a makefile, you might want to look at the man pages for make, or better yet, read this little **tutorial.**
3. Be sure that you thoroughly exercise your program's capabilities on a wide range of test suites, so that you will not be unpleasantly surprised when we run our tests.
4. Due to the simplicity of this project, the documentation for this project is fairly minimal. It may be more extensive for future projects. The majority of your grade for this assignment will depend upon **how well your implementation works** and a smaller portion will be given for documentation and style.
5. Even though you will not be heavily graded on style for this assignment, you should still follow all the principles of software engineering you learned in CS 101, CS 102, and elsewhere, such as top-down design, good indentation, meaningful variable names, modularity, and helpful comments.
6. **Don't be sloppy!** You should be following these principles for your own benefit.

# Original Work

This assignment must be the original work of you. Unless you have explicit permission, you may not include code from any other source or have anyone else write code for you. Use of unattributed code is considered plagiarism and will result in academic misconduct proceedings as described in the Syllabus.

**Remember: No late projects will be accepted!**