University of Calgary
Faculty of Science
Department of Computer Science

CPSC 457- Principle of Operating Systems
Assignment 1
A Simple Shell Using UNIX System calls
Winter 2022

1 Objectives

The shell is the term often used for the command line interpreter of an operating system (particularly for UNIX systems). That is, the shell is the program that reads and parses the user's input command line and then starts up the processes needed to carry out these command(s). Your assignment is to write a simple shell for Linux using standard Linux system calls.

2 Background

In order to write a shell you will need to understand the Linux mechanisms for process creation/termination, inter- process communication, and basic file management, as well as the C system calls that are used to implement these operating system constructs. Such a short program can be deceptively difficult, particularly because you will be dealing with the creation and synchronization of multiple concurrent processes. We would recommend that you think carefully and make sure you understand the above mechanisms well before attempting to write any code.

Before you begin this assignment, make sure you understand the basic functionality of the standard UNIX shell, in particular the following features:

1. Redirection of a command's input/output from/to a file, i.e., the use of $>$ and $<$ in the Linux shell. For example, the command ls $>$ file.lst will run the ls command and put the output of the command (a listing of all the file in the directory) into the file *file.lst*.

2. The use of pipes to connect the output of one command to the input of the following command on the command line, i.e., the use of `|` in the Linux shell. For example, the command `ls | wc` will run the ls command and use the output of the ls command as the input to the word count program, wc.

3. The execution of commands "in the background," i.e., the use of & as a command line terminator in the UNIX shell. When a command line is terminated with &, the shell will begin execution of the command(s), immediately reissue the prompt, and then accept and execute additional commands without waiting for the first command line commands to terminate. These commands in the first command lines are said to be running "in the background." Before attempting this assignment, you should also try out these features under the UNIX shell, as you'll have to try them out in your own shell later on.

3 Implementation Details

You may need to use the following system calls in your shell: fork(), wait(), exit(), execvp(), close(), open(), pipe(), dup().

The Linux documentation for these system calls can be obtained by using man commands. For example, to obtain a description of pipe() system call, use the command man pipe; to obtain a hard copy, use

```
man pipe | lpr - Pprinter_name
```

When a process executes the `execvp(file, argv)` system call, the image of the calling process is overlayed with the image of the executable file file. That is, the image of the calling process becomes the image file file. Thus for example, if file is the string **ls**, a process executing the *execvp()* will execute the ls command (and then terminate). Note that there is no *return* from the *execvp()* system call to the calling program, unless the *execvp()* fails.

You will need a very simple command line parser in order to get the name of the Unix command and its parameter. (This will give you some needed UNIX c/c++ programming experience that will be helpful in later assignments.) You may consider a single blank (space) character as a separator among command and parameters. The parser itself will parse a command line containing an arbitrary number of commands (separated by pipes), redirection of I/O, and background flag. It will also perform limited checking for the correct use of `<, > and |` in the same command line.

Using the above system calls and your own parser you should write a shell which will:

1. Execute a single command line which may include up to one argument

2. Execute a command line containing an arbitrary number of commands each can include up to one argument (separated by pipes) or input/output redirection.

3. Execute commands separated (linked) by a different pipe sign $. For example, consider four commands cmd1, cmd2, cmd3, and cmd4:

```
%cmd1 $ cmd2 cmd3
%cmd1 cmd2 $ cmd3
%cmd1 cmd2 $ cmd3 cmd4
```

   In the first example, the output of cmd1 is input to both cmd2 and to cmd3, the second, the input to cmd3 comes from cmd1 and from cmd2, and in the third, the input comes from both cmd1, cmd2 and goes to both cmd3 and to cmd4

   Note: in this case you may consider commands without parameters.

4. Execute background commands using the & as command terminator.

5. Check for correct use of `<` , `>` and `|` in the command line. Note that `<` , `>` and `|` can not be used together in a completely arbitrary manner.

6. Execute a command with multiple pipes (extra credit)

4 How to do it

The following helps you achieve your work on time:

1. Start with implementing and testing your parser. Follow this by programming your shell so it can handle a single command.

2. Expand your shell to handle two commands on a command line (separated by pipes). Depending on how you structure your shell, handling arbitrary commands in a command line maybe a trivial extension of the above solution.

3. Make sure you check the return code for every system call that you make and print out an error message if an error or unexpected return code is encountered. (This will greatly aid debugging.)

4. Once the parent process (your shell) has created a pipe and forked the two child processes that will read and write from the pipe, make sure the parent explicitly closes the file descriptors for its read and write access to the pipe. If you fail to do this, the child process reading from the pipe will never terminate. This is because the

child reading from the pipe will never get the END-OF-FILE (end-of-pipe) condition (and hence never terminate) as long as at least one process (in this case your shell program, by mistake) has an open write file descriptor for the pipe. The overall result will be a deadlock - your shell waiting for a child to terminate and the child waiting for the shell to close the pipe, which was inadvertently left open.

5. The ps command is useful for listing all processes associated with your session. If you see a lot of processes lying around in the zombie state, you've probably got a bug in your shell.

6. Make sure you consider what happens when the execvp() for the child fails.

5 Test Cases

Here is a sample set of test cases (they are other tests): Suppose `mysh%` is the prompt of your shell (it is not a file from which you make a redirection !!!)

```
mysh% date
mysh% gcc prog1.c
mysh% date > file.txt
mysh% ls -l > file.lst
mysh% wc -l < file.txt
mysh% ls -l | sort -r
mysh% cmd1 $ cmd2 cmd3
mysh% cmd1 cmd2 $ cmd3
mysh% cmd1 cmd2 $ cmd3 cmd4
mysh% cat < poem | grep are | wc -l > numberof.are
```

Your shell should recognize errors such as:

```
mysh% file.txt > more
mysh% ls | more < file.txt
mysh% ls |
```

6 Grading policy

Students could be randomly selected to demonstrate their programs and to show that they understand their codes. Work should be done individually. Any source of help MUST be indicated and cited.

Grading will be done as follows:

1. Total: 100 points + 10 points extra credit.

2. Parser: 10 points.
3. Single command: 10 points.
4. Commands with arguments: 10 points.
5. Input/Output redirection: 10 points.
6. Pipe: 10 points.
7. cmd1 cmd2 $ cmd3 : 10 points
8. cmd1 $ cmd2 cmd3 : 10 points
9. cmd1 cmd2 $ cmd3 cmd4: 10 points.
10. Background: 10 points.
11. Checking for errors: 10 points.
12. You will receive less than 20% if your program compiles, but it doesn't work at all.
13. You will receive 0 points if your program will not compile.
14. For identical code, in part or full, the UofC rules and regulations for plagiarism will apply.
15. After the due time, a %20 deduction will apply on every late day or part of the day.

It's better to have something working, even with little functionality, than a big program that crashes (or doesn't compile). We expect to see your own program, otherwise the above will apply.

7 To Submit

You need to submit source files, executables, along with a *Makefile* that will be used to compile and link your shell. These files will be combined into a tar file that will unpack the files into a directory whose name is your user-id. The tar file will be submitted on the D2L page of the course.

8 Deadline

February 11, 2022 before 11:55 P.M.