Project #1: MyShell

Today: March 28, 2023
Submission Due: April 18, 2023

1 Introduction

In this project, the students will learn and become familiar with the concepts of system-level process control, process signaling, interprocess communication, and running processes and jobs in the background in Linux Shell. Students will learn this by programming a simple yet customized Linux shell that supports all the aforementioned functionalities in their own programmed shell.

Linux Shell. A shell is an interactive command-line terminal program whose primary purpose is to execute user-provided commands and run other programs. A shell repeatedly prints a prompt, waits for a command line on the terminal via `stdin`, and then performs action as directed by the contents of the command line.

Project Prerequisites. Familiarity with C/C++ programming and Linux shell commands.

This project consists of three incremental phases, where each phase must be done for the next phase, i.e., You will be extending the functionality of your shell in every project phase.

2 Project Phase I: Building and Testing Your Shell (Points: 30)

In this phase, Your first task is to write a simple shell and starting processes is the main function of linux shells. So, writing a shell means that you need to know exactly what is going on with processes and how they start.

Your shell should be able to execute the basic internal shell commands such as,

- `cd`: navigate the directories in your shell
- `ls`: list the directory contents
- `mkdir, rmdir`: create and remove directory using your shell
- `touch, cat, echo`: creating, reading and printing the contents of a file
- `history`: tracks shell commands executed since your shell started (see specification below)
- `exit`: terminate all the child processes and quit the shell

As illustrated in Figure 1, commands should be executed by the child process created via forking by the parent process except `cd` and `history`.

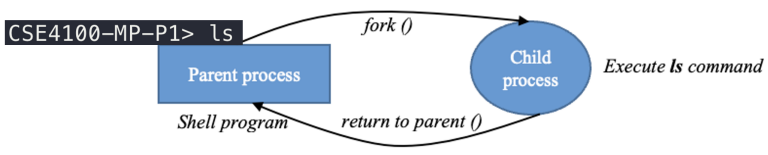

Figure 1: Main Structure of Shell Program

Here is an example of how your shell works.

```
~ $ ./myShell
CSE4100-MP-P1> ls
Makefile  README  myShell  myShell.c
CSE4100-MP-P1> mkdir myshell-dir
CSE4100-MP-P1> touch myshell-dir/cse4100
CSE4100-MP-P1> ls
Makefile  README  myShell  myShell.c  myshell-dir
CSE4100-MP-P1> cd myshell-dir
CSE4100-MP-P1> ls
cse4100
CSE4100-MP-P1> cd ..
CSE4100-MP-P1> ls
Makefile  README  myShell  myShell.c  myshell-dir
CSE4100-MP-P1> exit
~ $
```

Hints: The shell mainly relies on `fork( )` and `exec( )` system calls. These two system calls are actually the building blocks for how most programs are executed on Linux. First, an existing process forks itself into two separate ones. Then, the child uses `exec( )` to replace itself with a new program.

Your shell is constantly running loop with three functionalities inside;

do{
    // *Shell Prompt: print your prompt*
    print "CSE4100-MP-P1>"

    // ***Reading:*** *Read the command from standard input.*
    input = myshell_readinput ();

    // ***Parsing:*** *transform the input string into command line arguments.*
    args = myshell_parseinput (input);

    // ***Executing:*** *Execute the command by forking a child process and return to parent process.*
    myshell_execute(args);
} **while** (*true*);

Note: Please refer to the man pages for the fork(), exec(), wait(), and other related system calls.

history: The history command is executed as a built-in command. The history command should keep track of commands executed since your shell was executed. The history command of the default shell provides many functions, but in this project, only two functions need to be implemented.

- `!!` : Print the latest executed command. then, Executes the command. (!! command doesn't update history log.)
- `!#` : Print the command on the # line. then, Executes the command. (e.g. !12)

Here is an example of history command.

```
~ $ ./myShell
CSE4100-MP-P1> ls
Makefile  README  myShell  myShell.c
CSE4100-MP-P1> mkdir myshell-dir
CSE4100-MP-P1> touch myshell-dir/cse4100
CSE4100-MP-P1> cd myshell-dir
CSE4100-MP-P1> ls
cse4100
CSE4100-MP-P1> history
1   ls
2   mkdir myshell-dir
3   touch myshell-dir/cse4100
```

```
4   cd myshell-dir
5   ls
6   history
CSE4100-MP-P1> !5
ls
cse4100
CSE4100-MP-P1> !!
ls
cse4100
CSE4100-MP-P1> history
1   ls
2   mkdir myshell-dir
3   touch myshell-dir/cse4100
4   cd myshell-dir
5   ls
6   history
7   ls                // this log is for !5 command. !! related commnad(ls) is not updated
8   history
CSE4100-MP-P1> exit
~ $
```

Your history log should be maintained even after the shell exits. In other words, if you run your shell again and execute the history command, the same history log as before should be output. also, Your shell history log can grow infinitely.

Evaluation: Your shell should perform all the functionalities explained in task specifications above.

3 Project Phase II: Redirection and Pipe (Points: 30)

In this phase, you will be extending the functionality of the simple shell example that you programmed in project phase I. Start by creating a new process for each command in the pipeline and making the parent wait for the last command. This will allow running simple commands such as "`ls -al | grep filename`". The key idea is; passing the output of one process as input to another. Note that you can have multiple chains of pipes as your command line argument.

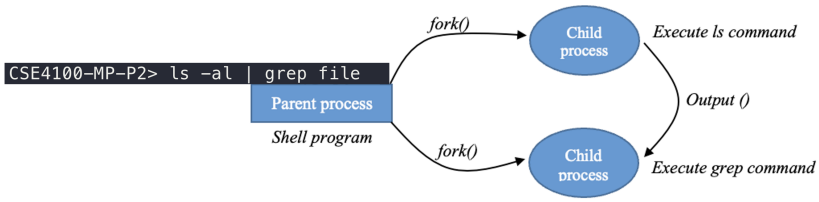

Figure 2: Main Structure of Shell Program with pipe and redirection

Hints: The Pipe is a command in Linux that lets you use two or more commands such that the output of one command serves as input to the next. In short, the output of each process acts as input to the next one, like a pipeline. The simplest way to solve multiple pipes in your command line arguments; is to use a recursive function called until there are no more piped commands during parsing.

Note: Please consult the man pages for the dup(), dup2() system calls.

Note: In this phase, You don't need to implement redirection. However, redirection works in a similar way to pipe. The difference is that redirection handles file-related commands.

Following shell commands with piping can be evaluated, e.g.,

- `ls | grep` *filename*
- `cat` *filename* `| less`
- `cat` *filename* `| grep -i "abc" | sort -r`

4 Project Phase III: Run Processes in Background (Points: 40)

It is the last phase of your MyShell project, where you enable your shell to run processes in the background. Linux shells support the notion of job control, which allows users to move jobs back and forth between background and foreground, and to change the process state (running, stopped, or terminated) of the processes in a job.

Your shell must start a command in the background if an '&' is given in the command line arguments. Besides, your shell must also provide various built-in commands that support job control.

Following shell commands with piping can be evaluated, e.g.,

- `jobs`: List the running and stopped background jobs.
- `bg` ⟨*job*⟩: Change a stopped background job to a running background job.
- `fg` ⟨*job*⟩: Change a stopped or running a background job to a running in the foreground.
- `kill` ⟨*job*⟩: Terminate a job

Note that one should not be required to separate the '&' from the command by a space. For example, the commands '`sort foo.txt` &', and '`sort foo.txt`&' and '`sort foo.txt` &' (blanks after the ampersand) are all valid.

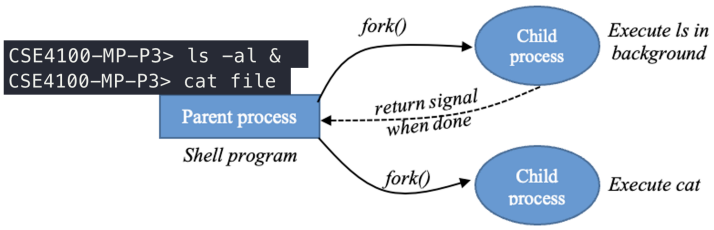

Figure 3: Main Structure of Shell Program with all features

Hints: When pressing ctrl-c causes a SIGINT signal to be delivered to each process in the foreground job. The default action for SIGINT is to terminate the process. Similarly, pressing ctrl-z causes a SIGTSTP signal to be delivered to each process in the foreground job. The default action for SIGTSTP is to place a process in the stopped state, where it remains until it is awakened by receiving a SIGCONT signal.

Note: Please consult the registering signal handlers in Chapter 8 to complete this project phase. SIGINT, SIGSTP, and SIGCONT are must-read items.

Following shell commands can be evaluated for this phase, e.g.,

- Any command from project phase II can be given with '&' at the end of command line
- `ls -al` *filename* | `grep` *filename* &
- `cat` *filename* | `grep -i "abc"` &

5 Submission Guideline

Hand-In Specifications: The submission should contain only source code file(s), including file(s), a makefile (all lowercase please), and the readme file. No executable program should be included. TA, in charge of grading your project, will automatically rebuild your shell program from the provided source code. **Please strictly follow the submission instruction below. If you not, you may get a penalty on your overall project score.**

1. You have to submit one archive file, and the archive file (`.tar.gz`) to be submitted should be named as `prj1_`*your_student_id*`.tar.gz`.
2. If you unarchive the file, the name of the root directory of the tar file should be your student id.
3. You should create directories for each phase with the name of "phase1/2/3."
4. Each directory for each phase should include a Makefile, README.md, and source codes. You also have to submit one document for 3 phases.

Attachment File:

- Phase1 folder(source code(myshell.c, myshell.h), Makefile, readme) (30 point)
- Phase2 folder(source code(myshell.c, myshell.h), Makefile, readme) (30 point)
- Phase3 folder(source code(myshell.c, myshell.h), Makefile, readme) (30 point)
- Documnet(about phase 1, 2, 3) (20 point)

Following is an example of a submission.

```
$~> ls
prj1_20231234.tar.gz
$~> tar -zxvf prj1_20231234.tar.gz
$~> ls
20201234  prj1_20231234.tar.gz
$~> tree
20231234
├── document.docx
├── phase1
│   ├── Makefile
│   ├── README.md
│   ├── myshell.c
│   └── myshell.h
├── phase2
│   ├── Makefile
│   ├── README.md
│   ├── myshell.c
│   └── myshell.h
└── phase3
    ├── Makefile
    ├── README.md
    ├── myshell.c
    └── myshell.h
```

All students are requested to upload the archived source files on eclass (cyber campus).

Note:Please make sure to compile your source code on CSPRO server, as the TA will build and compile your submitted project on that server. If the submitted code does not compile it cannot be scored!!!

6 Miscellaneous

6.1 Best practices the system programmers must comply to

WHAT YOU MUST DO:

1. You must read the entire Chapter 8 (Processes, Process Controls, and Signals) of the book to fully understand, learn and complete this project.
2. Read the complete project specifications before programming your shell project.
3. You may find it useful to check the Linux man pages on `fork()`, `exec()`, `getenv()`, `access()`, `waitpid()`, `opendir()`, and other related features mentioned in those man pages.
4. Source code commenting is a professional programmer's practice, and a must-do in this project as well.
5. Make sure to include a README file in your project explaining the basics of each project phase in your own way, so we can know your understanding of each phase concept.

WHAT YOU MUST NOT DO:

1. Do not hand-in any binary or object code files in your source code directory.
2. Do not include any hardcoded paths in your makefile.
3. Makefile should include all dependencies (libraries etc) required to build your program.
4. Do not copy or share your source code, it is strictly prohibited otherwise you will be given penalty for such an action.