Project Overview

In this project, the students will learn and become familiar with the concepts of system-level process control, process signalling, 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 Pre-requisites
Familiarity with c/c++ programming and Linux shell commands.

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

Project Phase I: Building and Testing Your Shell — Points: 30

Task Specifications:

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.

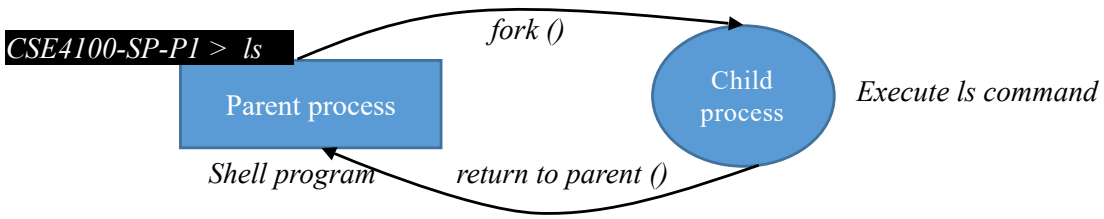
Your shell should be able to execute the basic internal shell commands such as,

- **cd, cd..**: to navigate the directories in your shell
- **ls**: listing the directory contents
- **mkdir, rmdir**: create and remove directory using your shell
- **touch, cat, echo**: creating, reading and printing the contents of a file
- **exit**: exit all the child processes and shell quits

A command is always executed by the child process created via forking by the parent process except cd.

```
gr120210194@cspro:~/CSE4100-myShell-P1$ ./myShell
CSE4100-SP-P#1> ls
Makefile  README.md  myShell  myShell.c
CSE4100-SP-P#1> mkdir myshell-dir
CSE4100-SP-P#1> touch myshell-dir/cse4100
CSE4100-SP-P#1> ls
Makefile  README.md  myShell  myShell.c  myshell-dir
CSE4100-SP-P#1> cd myshell-dir
CSE4100-SP-P#1> ls
cse4100
CSE4100-SP-P#1> cd ..
CSE4100-SP-P#1> ls
Makefile  README.md  myShell  myShell.c  myshell-dir
CSE4100-SP-P#1> exit
gr120210194@cspro:~/CSE4100-myShell-P1$
```

```
gr1202101940@cspro:~/CSE4100-myShell-P1$ ./myShell
CSE4100-SP-P#1> ls
Makefile  README.md  myShell  myShell.c  myshell-dir
CSE4100-SP-P#1>
```



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
printf ("CSE4100:P4-myshell >");

***Reading**: Read the command from standard input.*
input = myshell_readinput ();
***Parsing**: transform the input string into command line arguments*
args = myshell_parseinput (input);
***Execute:** Execute the command by forking a child process and return to parent process*
myshell_execute(args);
} while{true};

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

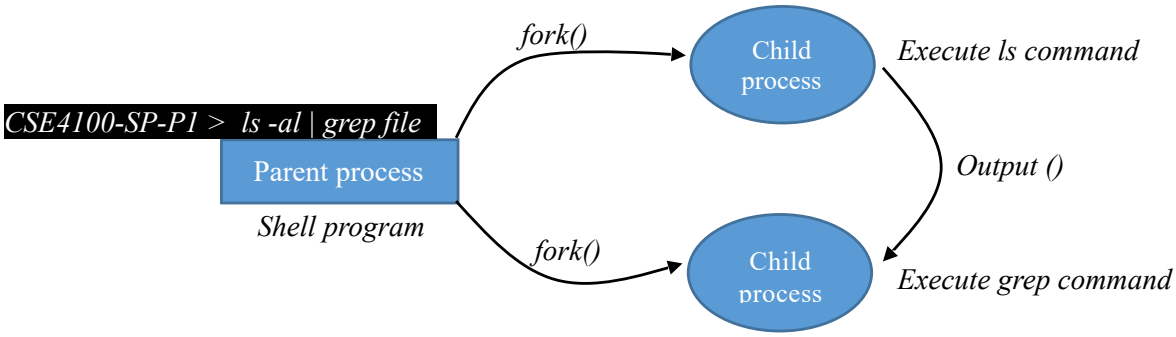
Evaluation:

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

Project Phase II: Redirection and Piping in Your Shell — Points: 30

Task Specifications:

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 output of one process as input to another. Note that, you can have multiple chains of pipes as your command line argument.



Hints:

The Pipe is a command in Linux that lets you use two or more commands such that 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 that is called until there are no more piped commands during parsing.

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

Evaluation:

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

- *ls -al | grep filename*
- *cat filename | less*
- *cat filename | grep -v "abc" | sort – r*

Project Phase III: Run Processes in Background in Your Shell Points: 40

Task Specifications:

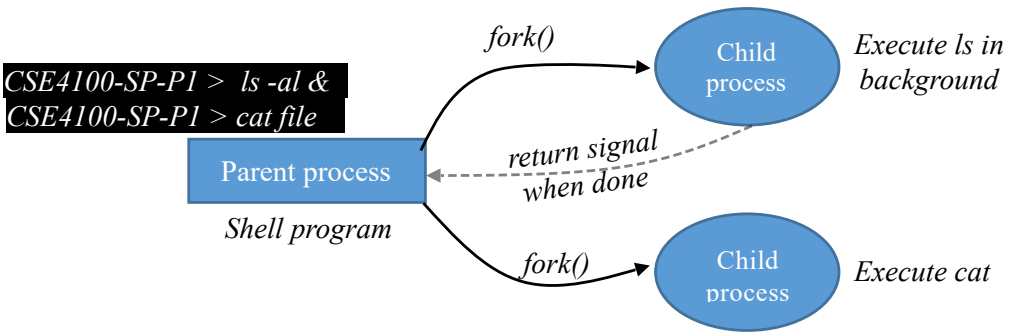
It is the last phase of your MyShell project, where you enable your shell to run processes in background. Linux shells support the notion of job control, which allows users to move jobs back and forth between background and foreground, and to change the process state (running, stopped, or terminated)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

For example:

- `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 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.



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 the receipt of 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.

Evaluation:

- *Any command from project phase II can be given with '&' at the end of commandline*
- *ls -al | grep filename &*
- *cat sample | grep -v a &*

Project Submission: How to Submit Your Shell Project

Submission Due: ~4/14(Thu) 23:39pm (late : ~4/17(Sun) 23:59pm -10% deduction per day**)**

Hand-In Specifications:
The submission should contain only source code file(s), include file(s), a makefile (all lower case please), and the readme file. ***No executable program should be included.*** The person marking your project will be automatically rebuild your shell program from the provided source code. You also submit one document for 3 phases.

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!!!

Source Code Management:

As described above, the files in the submitted directory would be:

- *makefile (mandatory)*
- *myshell.c (mandatory)*
- *myshell.h (mandatory)*
- *Readme (Optional but not a Mandatory requirement)*

How to Submit:

After completing each of the project phase, you need to submit your compressed/zipped source files. Please make sure that your source files contain appropriate Makefile, so that it can be compiled, build and executed for testing purpose.

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

Please use the following mentioned ***format*** for the ***attachment***.

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) (40 point)
- Document (about phase 1, 2, 3) (20 point)

Create a student ID folder, save the reports and folders in it, and compress the folder as follows in the location where the student ID folder is located.

- **tar –cvzf [HW1]20201234.tar.gz ./20201234**

Best Practices the system programmers must comply to:

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 about the each phase concepts.

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.

Programmers Pro-Tips:

1. Add your details on the header of your source files. (project, name, copyrights etc)
2. Your source file should be well structured and written in multiple functions to allow others to understand your source easily. (***Please do not make us do mental workout***)
3. Each function needs to be relatively short (less than one screen at a good font size is a good rule of thumb! by professional programmers.)