**CMPUT 379 — Operating System Concepts** **Fall 2019**

**Assignment 1 — A simple UNIX Shell in C/C++**

---

This assignment will be due **October 4, 2019 at 11:59pm** Mountain Time.

## Objective

This programming assignment is intended to give you experience in using UNIX system calls for process management (e.g., fork(), execv(), wait(), open(), and close()) and establishing pipes for interprocess communication.

## An Interactive Shell

Shell programs, such as the standard shell (sh), Bourne-again shell (bash) and C shell (csh), provide a powerful programming environment that allows users to utilize many of the services provided by the operating system. In this assignment, you will implement a simple, *interactive* command-line interpreter or shell, which we call `dragonshell`, in C or C++ (whichever you prefer). An interactive shell displays a prompt, reads and executes commands from user, prints out the result, and displays the prompt again. Unlike shells that support batch processing only, the user can interact with such a shell.

Your shell program must be able to execute a small set of built-in commands besides any external command that can be located in the path. In the former case, the shell contains the code to execute these commands. In the latter case, the shell does not understand the command, so it merely searches for the command name in its path or the working directory to identify the corresponding program and executes it with the provided arguments.

To implement your shell program, you will have to use system calls directly in your code. Thus, the first step is to familiarize yourself with UNIX system calls including the ones mentioned in this table:

| **system call** | **man page** |
|---|---|
| `chdir()` | man |
| `fork()` | man |
| `execve()` | man |
| `exit()` | man |
| `wait()` | man |
| `open()` | man |
| `close()` | man |
| `dup2()` | man |
| `pipe()` | man |
| `kill()` | man |

### Feature Highlights

Your task is to implement an interactive shell with the following required features:

1. Support the basic commands listed below. These commands must be implemented in your shell.
   (a) `cd` for changing the present working directory;
   (b) `pwd` for printing the present working directory;
   (c) `a2path` for overwriting or appending an address to the `path` variable;
   (d) `exit` for gracefully terminating the shell and all forked processes.

2. Run an external program with the provided arguments when its full (absolute) path is provided, or when the program can be located in the path or in the present working directory.
3. Run multiple commands written in a single line, separated by ";".
4. Support background execution when an "&" is put at the end of the command line.
5. Support redirecting the output from one program to a file.
6. Support piping the output of one program to another program.
7. Handle signals, generated by specific keyboard shortcuts, for stopping or pausing processes created by the shell. In particular, it should capture and handle Ctrl-C which sends SIGINT, and Ctrl-Z which sends SIGTSTP. Note that pressing Ctrl-C should not kill the shell, nor should Ctrl-Z interrupt the shell and redirect you to bash.

Below we describe these requirements in detail.

## Create a New Process for Each Command

When user enters a new command, `dragonshell` should either create a new process for executing this command with the specified arguments or print out an error message (`dragonshell: Command not found`) if the command is not a recognized built-in command, and it is not found in the present working directory and in the shell's path. The shell must return control to user (i.e., displays the prompt) only after the new process terminates. Note that a program can take zero, one, or several space separated arguments.

**Example**

```
dragonshell > pwd
/home/<csid>/CMPUT-379/Assignment-1
dragonshell > /usr/bin/which which
/usr/bin/which
dragonshell > rand
dragonshell: Command not found
```

In the above example, we assumed that `/usr/bin/which` is the full path to the `which` program, and the `rand` program was not found in the shell's path or in the current working directory.

**Note**: You must use the shell's path variable defined using a built-in command (described below) rather than the `PATH` environment variable.

## Add Support for Built-in Commands

The following built-in commands must be implemented by `dragonshell` rather than invoking external programs which perform the same operations.

### A. `cd` Command

Like bash and a lot of other shells, `cd` is used to change the current working directory. `cd` should take only one argument which is the full or relative path of a directory that user wants to change to.

**Example**

```
dragonshell > pwd
/home/<csid>/CMPUT-379/Assignment-1
```

```
dragonshell > cd ..
dragonshell > pwd
/home/<csid>/CMPUT-379
dragonshell > cd Assignment-12
dragonshell: No such file or directory
dragonshell > cd
dragonshell: expected argument to "cd"
dragonshell >
```

You need to handle possible error conditions. Specifically, the shell must prompt the user if it cannot find the directory or is missing an argument. In the former case, it should output `dragonshell: No such file or directory` and in the latter case it should output `dragonshell: expected argument to "cd"`.

### B. `pwd` Command

Like bash and a lot of other shells `pwd` is used to show the current directory. `pwd` should take no arguments to show the present working directory.

**Example**

```
dragonshell > pwd
/home/<csid>/CMPUT-379/Assignment-1
dragonshell >
```

### C. Showing the path and `a2path` Command

Typing in `$PATH` should tell the user the shell's path. The `a2path` command should allow the user to update the path either by overwriting it or appending to it by using `$PATH:<new-path>`. Like bash, you should use colon (:) as the delimiter to separate different paths.

**Example**

```
dragonshell > $PATH
Current PATH: /bin/:/usr/bin/
dragonshell > a2path $PATH:/usr/local/bin
dragonshell > $PATH
Current PATH: /bin/:/usr/bin/:/usr/local/bin
dragonshell >
```

You can assume that `/bin/` and `/usr/bin/` are initially included in the path.

### D. `exit` Command

When the user types `exit` or presses Ctrl - D, the shell should gracefully terminate all processes running in the background (if any) before terminating itself.

**Example**

```
dragonshell > exit
Exiting
# back to normal shell
```

The same behaviour is expected if Ctrl - D is pressed.

## Run External Programs

Your shell must be able to run any program that is found within its `PATH`; in the present working directory; or in the full path specified by user, and print out its output (similar to what happens when you run this program in bash). If the program is not found in the shell's `PATH` or the specified path, `dragonshell` should prompt the user that an error had occurred that the program is not found.

**Example**

```
dragonshell > /usr/bin/touch readme.md
dragonshell > ls readme.md
readme.md
```

In the above example, the `ls` command was found in the shell's `PATH`, and therefore could be launched by the shell.

## Output Redirection

When running programs, it's sometimes useful to direct the output to a file. The syntax `[process] > [file]` tells your shell to redirect the process's standard output to a file. Bash allows you to chain input and output redirection within a single line, but for simplicity we will only support output redirection.

**Example**

```
dragonshell > ls > lsout.txt
```

This should redirect the output from `ls` to the file `lsout.txt`. It should not display anything to the shell. If the file already exist then it will overwrite the previous content of the file. Otherwise, it will create a new file.

## Pipe

A useful feature of `dragonshell` is that allows for processing the output from one command using another command. This is where pipes are useful. For example, you want to list the first ten files within a directory (recursively) that has a .cpp file extension. You can do in bash by running the line below.

```
$ find ./ | grep ".cpp" | head
```

`dragonshell` needs to support piping as well. The syntax "`[process_1] | [process_2]`" tells your shell to redirect the output from process_1 to the input of process_2.

**Note**: For pipes you only need to support one level of piping. i.e., you will not need to support multiple, or chained pipes.

**Example**

```
dragonshell > find ./ | sort
./
```

```
./dragonshell
./dragonshell.cpp
./dragonshell.hpp
./lsout.txt
./Makefile
./README.md
```

This should return a sorted list of all files and folders listed in the current working directory.

## Running Multiple Commands

Most shells can run multiple commands written in a single command line. For example, in bash you can run multiple commands in a single line using ";" as the delimiter.
Similarly, in `dragonshell` the syntax "`[command_1] ; [command_2]` will first run `command_1` and then `command_2`. Many commands can be executed sequentially using this syntax as long as a ";" is put between every two commands. The shell will give back control to the user after all the commands have been processed.

**Example**

```
dragonshell > ls -al > lsout.txt ; find ./ | sort ; whereis gcc
./
./dragonshell
./dragonshell.cpp
./dragonshell.hpp
./lsout.txt
./Makefile
gcc: /usr/bin/gcc /usr/lib/gcc /usr/share/man/man1/gcc.1.gz
dragonshell >
```

This will write the output from `ls -al` to a file called `lsout.txt`, then it will return a sorted list of all files and folders listed in the current working directory, finally it will return the location of where the `gcc` binary is stored.

You program should be able to run an "infinite" amount of commands in sequence.

## Handling Signals

Most shells let you stop or pause processes with special keystrokes. These special keystrokes, such as `Ctrl-C` and `Ctrl-Z`, work by sending signals to the shell's subprocesses. If we send these signals to your shell either by keystrokes or by terminal then it will terminate or stop the shell itself; we do not want that to happen in `dragonshell`, so we have to send signals to the shell's subprocesses only.

**Expected behaviour**: this is what we expect to happen

```
dragonshell > ^C
dragonshell > ^C
dragonshell > ^Z
dragonshell >
```

Rather than this

```
dragonshell > ^C
# back to normal bash
```

or this

```
dragonshell > ^Z
[1]+  Stopped                 ./dragonshell
# back to normal bash
```

### Putting Jobs in Background

So far, your shell waits for each process to finish before starting the next one. But most UNIX shells allow the user to run a command in the background by putting an "&" at the end of the command line. In this case, the shell allows the user to run other commands without waiting for the background process to finish.

Your shell program must support the background execution of a command when the "&" character appears at the end of the command line. You will have at most one program running in the background and it will be an external program rather than a built-in command.

**Example**

```
dragonshell > ls &
PID 1741 is running in the background
dragonshell >
```

When you put a process in the background, your shell should output the same message as in the above example, except that 1741 is replaced with the PID of the new process that is put in the background.

You do not need to write a function that will bring a background process to the foreground. Furthermore, the process running in the background does not need to send a message to the terminal upon completion.

## Starter Code

Download `a1-starter-code.zip` file from eclass; it contains the starter code in C and C++ for this assignment (use the one that is appropriate). The starter code includes the main function and a string tokenizer which splits a string into words.

## Deliverables

Submit your assignment as a single compressed archive file (`dragonshell.zip` or `dragonshell.tar.gz`) containing:

1. A custom Makefile with at least these four targets: (a) the main target `dragonshell` which simply links all object files and produces an executable called `dragonshell`, (b) the target `compile` which compiles your code and produces the object file(s), (c) the target `clean` which removes the objects and executables, and (d) the target `compress` which produces the compressed archive (ready for submission). You may add more targets to your Makefile if necessary.
2. A plain text document, called `readme.md`, which explains your design choices, lists the system calls you used to implement each of the required features, and elaborates on how you tested your implementation. Make sure you cite all sources that contributed to your assignment in this file. You may use a markup language, such as Markdown, to format this plain text file.
3. All files required to build your project including the starter code.

Your shell should run by entering the following commands in bash:

```
$ make
$ ./dragonshell
```

A prompt or a greeting should be shown to determine that `dragonshell` has started execution. For example:

```
                                      ______________
                                ,===:'.,            `-._
                                     `:.`---.__         `-._
                                       `:.     `--.         `.
                                         \.        `.         `.
                                 (,,(,    \.         `.   ____,-`.,
                              (,'     `/   \.   ,--.___`.'
                          ,  ,'  ,--.  `,   \.;'         `
                           `{D, {    \  :    \;
                             V,,'    /  /    //
                             j;;    /  ,' ,-//.    ,---.      ,
                             \;'   /  ,' /  _  \  /  _  \   ,'/
                                   \   `'  / \  `'  / \  `.' /
                                    `.___,'   `.__,'   `.__,'
__      __   _                   _         ___                            ___ _        _ _
\ \    / /__| |__ ___ _ __  ___  | |_ ___  |   \ _ _ __ _ __ _ ___ _ _   / __| |_  ___| | |
 \ \/\/ / -_) / _/ _ \ '  \/ -_) |  _/ _ \ | |) | '_/ _` / _` / _ \ ' \  \__ \ ' \/ -_) | |
  \_/\_/\___|_\__\___/_|_|_\___|  \__\___/ |___/|_| \__,_\__, \___/_||_| |___/_||_\___|_|_|
                                                         |___/

dragonshell >
```

It does not have to be fancy like above; your shell can simply output `Welcome to Dragon Shell!` when it is executed.

## Misc. Notes

- This assignment must be completed **individually** without consultation with anyone besides the instructor and TAs.
- You can write the shell program in C or C++. The starter code is provided for both languages. You must compile the program using `gcc` or `g++`. No warnings should be returned when your code is compiled with `-Wall` flag. Also make sure that your code does not print anything extra for debugging.
- You **cannot** use `system( )` or `popen( )` to implement any of the required features.
- You must use system calls directly in your code. Marks will be deducted if you use functions from the standard C library (glibc).
- Check your code for memory leaks. You may use the Valgrind tool suite:

  ```
  valgrind --tool=memcheck --leak-check=yes ./dragonshell
  ```
- You can use your own machine to write the code, but you must make sure that it compiles and runs on the Linux lab machines (e.g., ugXX.cs.ualberta.ca where XX is a number between 00 and 34).
- When developing and testing your program, make sure that you clean up all processes before you logout of a workstation.