CMPUT 379 — Operating System Concepts **Fall 2025**

Assignment 1 — Simple Shell for Linux

The assignment is due on **October 1, 2025 at 5:59 PM** Mountain Time.

Objective

In this assignment, you will gain hands-on experience with system calls for process management and interprocess communication. You will use these system calls to build a full-fledged shell for Linux.

Interactive Linux Shell

Shell programs, such as the standard shell (sh), Bourne-again shell (bash) and C shell (csh), provide a powerful programming environment allowing users to utilize many of the services provided by the operating system. In this assignment, you will implement *dragonshell*, an interactive command-line interpreter or shell for Linux. An interactive shell displays a prompt, reads and executes the user command, prints out the result, and displays the prompt again.

Your shell will execute a small set of built-in commands in addition to external programs that can be found in the user-specified path. More specifically, the shell program contains the code to execute the built-in commands, but it does not contain the code to execute other commands or programs. Hence, it searches for the program name in the specified path or in the current working directory to identify the executable. Once it is found, it spawns a child process to execute that program with the provided arguments.

In this assignment you will use system calls, more specifically the wrapper functions listed in Section 2 of the Linux manual page). Thus, the first step is to familiarize yourself with these system calls, including chdir(2), getcwd(2), fork(2), execve(2), _exit(2), wait(2), waitpid(2), open(2), close(2), dup(2), pipe(2), kill(2), and sigaction(2).

Feature Highlights

1. Support the built-in commands listed below. These commands must be implemented in the shell.
   (a) `pwd` for printing the shell's current working directory;
   (b) `cd` for changing the shell's current working directory;
   (c) `jobs` for displaying the status of all running and suspended processes spawned by the shell;
   (d) `exit` for gracefully terminating the shell and all spawned processes that have not terminated.
2. If none of the above commands is input, then run an external program with the provided arguments when its (absolute or relative) path is provided or when it can be found in the current directory. The syntax is `cmd arg*` where `cmd` is the name of a program or its pathname and `*` indicates zero or more arguments.
3. Support the background execution of a program when `&` is put at the end of the command line: `cmd &`
4. Support redirecting the standard output of a program to a file using syntax `cmd > fname`
5. Support redirecting the contents of a file to the standard input of a program using syntax `cmd < fname`
6. Support piping the standard output from the command on the left side of `|` into the standard input of the command on the right of `|`. The syntax is `cmd1 | cmd2`
7. Handle signals generated by keyboard shortcuts for stopping or pausing a process spawned by the shell. Specifically, you should handle Ctrl - C which sends SIGINT, and Ctrl - Z which sends SIGTSTP. Note that 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.

Built-in Commands

The following built-in commands must be implemented by `dragonshell`. In other words, they are part of the shell and must be executed directly **without forking a child process**.

Each built-in command takes zero or one argument. Your shell should ignore any extra arguments that are passed to a built-in command without printing an error message.

A. The `pwd` Command

The `pwd` command prints the current working directory of the shell and takes no arguments.

Example

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

B. The `cd` Command

The `cd` command changes the current working directory of your shell. It takes only one argument: the absolute or relative path of the target directory.

Example

```
dragonshell > pwd
/home/<ccid>/CMPUT-379/Assignment-1
dragonshell > cd ..
dragonshell > pwd
/home/<ccid>/CMPUT-379
dragonshell > cd Assignment-12
dragonshell: No such file or directory
dragonshell > cd
dragonshell: Expected argument to "cd"
dragonshell >
```

The shell must prompt user if it cannot find the directory or is missing an argument as shown in the above example. In that case, the error message **must be identical** to the one shown in the above example.

C. The `jobs` Command

The `jobs` command prints out the contents of the process table, a data structure maintained by the shell. This data structure, which can be implemented as a linked list, contains information about all running and suspended processes that were spawned by the shell. Specifically, for every process, you need to store its PID and state (either running or suspended), and the command that created this process in a structure.

Running an external program, in foreground or background, adds an entry to the table. Once a process finishes execution, it must be removed from the table. Terminating a process via Ctrl - C removes it from the table. Suspending a process via Ctrl - Z changes the execution state of a process in the table.

Example

```
dragonshell > /usr/bin/touch readme.md
dragonshell > jobs
dragonshell > long-running-job < input.txt &
dragonshell > jobs
56188 R long-running-job < input.txt &
dragonshell > long-running-job < input.txt
^Z
dragonshell > /usr/bin/sleep 10 &
dragonshell > /usr/bin/sleep 15 &
dragonshell > jobs
56188 R long-running-job < input.txt &
56190 T long-running-job < input.txt
56200 R /usr/bin/sleep 10 &
56201 R /usr/bin/sleep 15 &
```

As shown above, the `jobs` command lists all processes spawned by the shell that are either running or suspended. Terminated processes are removed from the shell's process table and are not shown. Specifically, the output must be printed in a space-separated format, with three fields representing:

- The first field is the process ID (PID) in the operating system.
- The second field is the process state which can be either `R` for running or `T` for suspended.
- The last field is the command that started the job.

Note that you cannot read process states from the kernel, e.g. from `/proc/[pid]/status`. The shell itself is responsible for tracking process states whenever it receives SIGCHLD signals using the waitpid system call.

C. The `exit` Command

When user enters `exit`, the shell will gracefully terminate any process running in the background or suspended (if any). Then, it will terminate itself. Note, for graceful termination, SIGTERM must be sent to a process.

Example

```
dragonshell > exit
# back to default Linux shell
```

Create a New Process for Each External Program

When user enters a command that is not a recognized built-in command, `dragonshell` will spawn a new process for executing this external program with the specified arguments and print out its output similar to what happens when you run this program in bash. If the program is not found in the current working directory or in the (absolute or relative) path specified by user, then the shell will print the following message: `dragonshell: Command not found`. You will not check the `PATH` environment variable in this assignment.

The shell must return control to user, i.e., display the prompt again, only after the new process terminates. The exception to this rule is background execution which is described later. Note that a program can take zero, one, or several space separated arguments. These arguments will be provided directly after the program's name/pathname.

Example

```
dragonshell > pwd
/home/<ccid>/CMPUT-379/Assignment-1
dragonshell > /usr/bin/ls readme.md
```

```
/usr/bin/ls: cannot access 'readme.md': No such file or directory
dragonshell > /usr/bin/touch readme.md
dragonshell > /usr/bin/ls readme.md
readme.md
dragonshell > rand
329
dragonshell > ../rand
dragonshell: Command not found
dragonshell >
```

In the above example, it is assumed that `/usr/bin/touch` is the path to the `touch` program, and `/usr/bin/ls` is the path to the `ls` program. It is also assumed that a user program that is called `rand` and prints out a random number is in the current directory but not in the parent directory, hence the error.

Input & Output Redirection

It is sometimes useful to direct the standard output of a command to a file (i.e. output redirection) or direct a file's contents to the standard input of a command (i.e. input redirection). In bash, this is done by entering `cmd > fname` or `cmd < fname` , respectively. Your shell program must support input and output redirection for external programs only. Additionally, it must support chaining input and output redirection within a single line using this syntax `cmd < fname > fname2` where `cmd` is the program's name or pathname, `fname` is the name or pathname of the input file, and `fname2` is the name or pathname of the output file. We will not test your code with the same file being used for input and output redirection, i.e. `fname` and `fname2` are different names (pathnames) in the above example.

Example

```
dragonshell > /usr/bin/echo "test"  > ../a.out
```

This should redirect the output from `/usr/bin/echo` to `a.out` which is one level up from the current directory. If this file already exists, the shell will write over any existing data. Otherwise, it will create a new file and write `test` in it.

Note that supporting standard error redirection is not required in this assignment.

Pipe

Your shell must support 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 in a directory (recursively) that have a .cpp file extension. In bash, you can do this by entering:

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

For this assignment, you will support one level of piping, i.e., it is not required to support multiple (chained) pipes in a single line as in the above example. Specifically, `cmd1 | cmd2` tells `dragonshell` to transfer the standard output from `cmd1` to the standard input of `cmd2`.

Example

```
dragonshell > /usr/bin/find ./ | /usr/bin/sort
./
./dragonshell
./dragonshell.c
./dragonshell.h
```

```
./Makefile
./README.md
dragonshell >
```

Explanation: This returns a sorted list of all files and folders listed in the current working directory.

Note that we will not test your implementation with a command line that includes both pipe and input/output redirection. Also, a pipe will not be used with built-in commands.

Handling Signals

Most shells let user terminate or pause the execution of a command with special keystrokes. These special keystrokes, such as Ctrl - C and Ctrl - Z , are handled in a way that the respective signal is sent to the shell's subprocess that runs the command rather than the shell process itself to avoid terminating or suspending the shell. This is the expected behavior in `dragonshell` too.

Example

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

Notice that `dragonshell` is not terminated (or suspended) in the above example. However, `myprogram` would be terminated (or suspended) if SIGINT (or SIGTSTP) was sent when it was still running.

Sending Jobs to Background

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

Your shell will support the execution of an external program in the background when "&" appears at the end of the command line. You can assume that at most one program will run in the background. Built-in commands cannot be executed in the background. Moreover, you do not need to handle the case where both pipe and "&" are used in a single command line.

Example

```
dragonshell > /usr/bin/ls &
PID 1741 is sent to background
dragonshell >
```

When a process is sent to background, `dragonshell` prints out `PID X is sent to background`, with `X` being the PID of the new process that is sent to background.

Note that you do not need to implement a command to bring a background process to the foreground. Furthermore, the process running in the background does not need to print a message in the terminal to notify the user upon completion.

Running dragonshell

We will run your shell by entering the following commands in bash:

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

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

```
Welcome to Dragon Shell!

dragonshell >
```

Submission

Submit all the required files via GitHub Classroom. Do not use compressed files or archives.

When you accept the assignment and clone the repository, there should be a file called `dragonshell.c` that contains two functions: `tokenize` and `main`. You may implement other functions in this file. You may also add other .c and .h files as needed.

The following files must be included in your repository.

1. A custom `Makefile` (use this exact name) with at least these three targets: (a) the main target `dragonshell` that links all object files to produce an executable called `dragonshell`, (b) the target `compile` that compiles your code and produces the object file(s), (c) the target `clean` that removes the object file(s) and the executable file(s). You can add more targets to your Makefile.
2. A plain text document, called `README.md` (use this exact name), that contains a header (see below), explains your design choices, lists the system calls that you used to implement each of the required features, and elaborates on how you tested your implementation. Make sure to cite all sources that you used in this file. Feel free to use a markup language, such as Markdown, to format this plain text file.

   At the very top of the readme file before any other lines, include **a header** with the following information: **your name, student ID (SID), and CCID**. For example:

   ```
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   # Name : Jane Doe
   # SID : 1234567
   # CCID : jdoe
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   ```

3. All files required to build your project, including `dragonshell.c`.

Misc. Notes

- This assignment must be completed individually without consultation with anyone. Questions can be asked from TAs in the labs.

- The shell program must be written in C (**not C++**) and compiled using `gcc`. No warnings should be printed when your code is compiled with `-Wall` flag. Also make sure that your code does not print extra lines for debugging. Marks will be deducted for not complying with these requirements.
- Check your code for memory leaks. You may use the Valgrind tool suite:

  ```
  valgrind --tool=memcheck --leak-check=yes ./dragonshell
  ```

  Marks may be deducted for memory leaks and other memory related issues.
- You must use system call wrapper functions from Section 2 of the Linux manual page in your code. Marks may be deducted if you use C library functions from Section 3 of the Linux manual page or commands and utilities from Section 1 of the manual to implement the key features of the shell. Library functions might be used for memory allocation, string processing, and basic input/output operations only.
- You are not allowed to use `system( )` or `popen( )` to implement the required features.
- To make things simple and minimize the programming effort, you may use the following constants in your code:

  ```
  LINE_LENGTH    100   // Max # of characters in an input line
  MAX_ARGS         5   // Max # of args to a command (not incl. command name)
  MAX_LENGTH      20   // Max # of characters in an argument
  ```
- You can assume that all system calls will be successful.
- You may use your personal computer for programming, but you must make sure that your code compiles and runs correctly on a Linux lab machine. The list of these machines can be found on Canvas.
- During your development of a solution to this assignment, an incorrect implementation may leave processes running in the background after you log out. It is your responsibility to make sure that you clean up all processes. Students who leave processes running on machines may be penalized.