# PA2: Implementing a Linux Shell

Due: Sunday 9/20/20 at 11:59pm

In this programming assignment, you are going to implement your very own linux shell. Your shell should have the ability to function almost as much as the linux/ubuntu shell in your OS, which lets a user navigate through the file system and perform a wide variety of tasks using a series of easy to remember and simple commands. Please take a look at this website for some interesting and commonly used linux commands.

### Features of Shell

Environment

The shell maintains many variables which allow the user to maintain some settings and information visible throughout the system. This is analogous to a collection of global variables in a C program. The difference is that all programs running from the shell can use environment variables. For instance, the current working directory and the PATH are two of many important variables. As its name implies, the current working directory variable keeps track of the user's current directory. The PATH variable on the other hand consists of a colon separated directory list that is searched wherever you type a command in the terminal. If an executable file by the same name does not exist in any of these directories or the current directory, the shell says “command not found”, which is something we experience when trying to run a program before installing it. One may modify the PATH at any time to add and remove directories to search for executables. The following shows the commands for printing the current directory and the content of the PATH variable:

```
shell> pwd
/home/tanzir
shell> echo $PATH
/usr/local/bin:/usr/bin:/usr/local/games:..
```

Command Pipelining

While the individual linux commands are useful for doing specific tasks (e.g., **`grep`** for searching, **`ls`** for listing files, **`echo`** for printing), sometimes the problems at hand are more complicated as they require multiple commands together and feeding the output of one to the input of the next. The linux shell lets you combine commands by putting the character | between then. A pipe in between two commands causes the standard output of one to be redirected into the standard input of another. An example of this is provided below, using the pipe operation to search for all processes with the name "bash".

```
shell> ps -elf | grep bash | awk '{print $10}' | sort
3701
4197
```

Input/Output Redirection

Many times, the output of a program is not intended for immediate human consumption (if at all). Even if someone isn't intending to look at the output of your program, it is still immensely helpful to have it print out status/logging messages during execution. If something goes wrong, those messages can be reviewed to help pinpoint bugs. Since it is impractical to have all messages from all system programs print out to a screen to be reviewed at a later date, sending that data to a file as it is printed is desired.

Other times, a program might require an extensive list of input commands. It would be an unnecessary waste of programmer time to have to sit and type them out individually. Instead, pre-written text in a file can be redirected to serve as the input of the program as if it were entered in the terminal window.

In short, the shell implements input redirection by redirecting the standard input of a program to an file opened for reading. Similarly, output redirection is implemented by changing the standard output (and sometimes also standard error) to point to a file opened for writing.

```
shell> echo "This text will go to a file" > temp.txt
shell> cat temp.text
This text will go to a file
shell> cat < temp1.txt
This text came from a file
```

Background Processes

When you run a command in the shell, it suspends until the command finishes. We often do not notice this effect because many commonly used commands finish soon after they start. However, if the command takes a while to finish, the shell stays inactive for that duration and you cannot use it. For instance, typing **`sleep 5`** in the shell causes the shell to suspend for 5 seconds. Only after that the prompt comes back and you can type the next command. You can change this behavior by sending the program to the background and continue using the shell. If you type **`sleep 5 &`** in the shell for example, it will return the shell immediately because the corresponding process for the sleep runs in the background instead of in the foreground where regular programs run.

Implementation Hint: First of all, remove the '&' symbol from the command before passing it to exec(). But use the symbol to set a boolean that tells you that it is supposed to run in the background. After that, from the parent process do not call **`waitpid()`**, which you have been doing for the regular processes. Rather put the pid into a list/vector of background processes that are currently running and periodically check on the list to make sure that they do not become Zombies or do not stay in that state for too long. A good frequency of check is before scanning the next input from the user inside the main loop. Keep in mind that waitpid() function suspends when called on a running process. Therefore, calling it as it is on background processes may cause your whole program to get suspended. However, there is an option in waitpid() that makes it non-blocking, which is the desired way of calling it on background processes. You can find this option from the man pages.

Use of Single/Double Quotes

White-spaces are usually treated as argument separators except when they are used inside quotes. For example, notice the difference between the following two commands:

```
shell> echo -e "cat\ta.txt"
cat     a.txt
shell> echo "-e cat\ta.txt"
-e cat\ta.txt
```

Note that the "-e" option for the **`echo`** command prints the string with interpretation of some symbols. Now, in the first command, the string is put inside quotes to make sure that it is interpreted as a single string. As a result of using the -e option, the string is printed after interpreting the "\t" as a tab. In the second example, "-e" is part of the string which masks '\t' interpretation and prints a multi-word sentence which is impossible without putting quotes around. Also note the following example:

```
shell> echo -e `<<<<< This message contains a |||line feed >>>>>\n'
<<<<< This message contains a |||line feed >>>>>

shell>
```

Which does not consider the above command to have redirections or pipes because the corresponding symbols are inside quotation marks.

*Implementation Hint:* In your command parser that looks for certain tokens, have a counter of how many single or double quotes you have seen so far. If the number is odd, it means you are inside a quote and you should just ignore the token. On the other hand, if you are outside (indicated by an even count), you should consider the token by its face value.

## Your Task

In this assignment you will design a simple shell which implements a subset of the functionality of the Bourne Again Shell (bash). The requirements are below:

1. Continually prompt the user for the next input
2. Parse the user input to extract the command(s) in it and execute it (them). The rules for parsing are described below
3. For executing a command from the shell, you must use the **`fork()+exec()`** function pair. You cannot use the **`system()`** function to do it because that creates a child process internally without giving us explicit control. In addition, your shell must wait for the executed command to finish which is achieved by using the **`wait()/waitpid()`** functions
4. Support input redirection from a file and output redirection to a file. Note that a single command can have both
5. Allow piping multiple commands together connected by "|" symbols in between them. Every process preceding the symbol must redirect its standard output to the standard input of the following process. This is done using an Interprocess Communication mechanism called pipe that is initiated by calling the **`pipe()`** system call
6. Run the user command in the background if the command contains a "**`&`**" symbol. Note that you must avoid creating Zombie processes in this case

7. Allow directory handling commands (e.g., pwd, cd). Note that some of these commands are not recognized by the **`exec()`** functions because there are no executables by the same name. These are some additional shell features that must be implemented using system calls instead of forwarding to **`exec()`**.
8. Print a custom prompt to be shown before taking each command. This should include your user name and current date-time
9. Bonus: Allow $-sign expansion. See the last command in the grading instructions command list in the following
10. Write a report describing your unique (you can skip things that are common to the entire class) design choices, algorithms (e.g., how you implemented single/double quotes, $-sign expansions) and any implemented bonus features with implementation technique
11. Make a youtube video of the demonstration following the grading instructions and include a link to the video in the report

**Grading Instructions for Shell**

1. Checking single and double quotes: 5 points (2.5 points each)
   A. `echo “Hello world | Life is Good > Great $”`
   B. `echo ‘Hello world | Life is Good > Great $’`

2. Simple commands with arguments: 20 points
If commands only w/o arguments work deduct 15 points.
   A. `ls`
   B. `ls -l /sbin # should list from the /sbin directory that has many files,`
   C. `ls -l -a`
   D. `ls -la #this should be same as c`
   E. `ps -aux`
   F. `ls -l .. # lists the parent directory`
   G. `To check if they wait for the child process, run the following:`
   H. `sleep 5`
This should block the shell for 5 seconds. If the shell comes back immediately, deduct 5 points because they are not using wait()/waitpid().

3. Input/Output redirection (20 points – 10 points each):
   A. `ps aux > a`
   B. `grep /init < a`
   C. `grep /init < a > b # grep output should go to file b`
If they do not have the last part (i.e., input output redirection in the same line), deduct 5 points

4. Single pipe (10 pts):
   Any single pipe command

5. Two or more pipes (20 points):

```
A. ps aux | awk '/init/{print $1}' | sort -r
B. ps aux | awk '/init/{print $1}' | sort -r | awk '/ro/' | grep ro
C. ps aux | awk '{print $1$11}' | sort -r | grep root
```

6. Two or more pipes with input and output redirection (5 points):

```
A. ps aux > test.txt
B. awk '{print $1$11}'<test.txt | head -10 | head -8 | head -7 |
   sort > output.txt
C. cat output.txt
```

7. Background processes: 10 points

```
A. sleep 1 &
B. sleep 2 &
C. sleep 20 &
```

The prompt should come back immediately, otherwise deduct 5 points. Also, look for the code where they handled background processes. They should get the pid of the child process, put that in a vector or list, and before getting every command they should cycle through each pid in the list with waitpid (pid, 0, WNOHANG). This should not take time at all and they cannot get stuck in a long running process because WNOHANG makes the function non-blocking.

Give some partial points for effort. Otherwise, deduct full 10 points if they do not have anything significant about bg processes.

8. Directory processing (5):

```
A. cd ../../
B. cd .
C. cd /home/
D. cd - # goes back to the last directory you were in before this
   one. It is similar to the back button in the browser
```

9. Own prompt 5 points and misc: Deduct 5 points if they do not print any prompt

10. Bonus (10 pts): $ sign expansion

```
A. cat /proc/$(ps|grep bash|head -1|awk '{print $1}')/status
B. mkdir a
C. cd $(ls -l | grep '^d'|head -1|awk '{print $9}'   # goes inside
   the first directory in the current directory
```

11. Other open-ended bonus options (anything outside the below list will be credited based on effort):

A. [3 pts] Command history (pressing Up/Down button goes to previous/next command)
B. [3 pts] Autocomplete by pressing tab