PA2: Aggie Shell

In this programming assignment, you are going to implement your very own Linux shell that we will choose to call "Aggie Shell". After all, the earliest OS interfaces were indeed the terminal shell or shell-like interfaces! The Linux/Ubuntu shell in your OS lets a user navigate through the file system and performs a wide variety of tasks using a series of simple commands. It also offers capabilities of inter-process communication and file input/out redirection. Your shell should have the ability to function almost as well as BASH. Each Linux command (e.g., cat, echo, cd, etc.) is to be run as child processes in this assignment. Most of these commands sit as executables in your system and are recognized by a call to an **exec()**-family function. Refer to this underline for some interesting and commonly used Linux commands.

Features of the Aggie Shell:

Note: For some of the features below, we also provide implementation hints in *italics*.

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. We may want to run a series of commands that require the output of one command to be fed as the input of the next. The Linux shell lets you run a series of commands by putting the pipe character ("|") between each command. It causes the standard output of one command to be redirected into the standard input of the next. In the example below, we use the pipe operation to search for all processes with the name "bash". The resultant output (3701 and 4197) is all the process ids or pids of these processes.

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

3701
4197
```

> *Implementation Hint: Feed in an array of 2 integers to the **pipe(...)** system call which populates the index 0 of the array with the read end and index 1 of the array with the write end file descriptors (as covered in Lab3A). You are provided a vector of all commands present in the command pipeline (in order) courtesy of the* Tokenizer *class member **commands**. Use **fork() + execvp(...)** to execute each command from start to end. Make sure to appropriately use the **dup2(...)** system calls to redirect stdout and stdin as necessary.*

Input/Output Redirection

Sometimes, the output of a program is not intended for immediate human consumption. 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 which can be reviewed to help pinpoint bugs. Since it is impractical to have all messages from all system programs printed out to a screen to be reviewed at a later time, sending that data to a file is convenient and desirable. Sometimes it is also done out of necessity (where the result file is packaged for consumption by another entity). Output redirection is implemented by changing the standard output (and sometimes also standard error) to point to a file opened for writing.

An example of output redirection is as follows:

```
shell> echo "This text will go to a file" > temp.txt
```

Executing this command will result in temp.txt holding the contents "This text will go to a file". Note that ">" is the character used for output redirection. If we execute this command without output redirection, the string "This text will go to a file" will be printed out to stdout instead.

We can execute the `cat` command to verify the contents of temp.txt as a result of the previous command; in this case `cat` prints out the contents of the file to stdout:

```
shell> cat temp.text

This text will go to a file
```

Other times, a program might require an extensive list of input commands. It would be a waste of time to 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 a file opened for reading.

An example of input redirection is as follows:

```
shell> cat < temp2.text

This text came from a file
```

In this particular example, the contents of the file temp2.txt is "This text came from a file". We redirected the contents of temp2.txt to serve as the input, which results in the shown output.

*Implementation Hint: For input redirection, you should use the **dup2()** system call to redirect the file descriptor that corresponds to stdin to the one referred to by the file descriptor opened for reading. Similarly, for output redirection, you should use the **dup2()** system call to redirect the file descriptor that corresponds to stdout to the one referred to by the file descriptor opened for writing.*

Background Processes

When you run a command in the shell, the shell suspends until the command completes its execution. 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 and you cannot use it for that duration. 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 continuing use of 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: The "&" symbol is removed from the tokenized command and a boolean is set in the Command object that tells you that it runs in the background. From the parent process do not call `waitpid()`, which you have been doing for the regular processes. You should instead put the pid into a list/vector of background processes that are currently running. 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. 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.

cd commands

You must use the **chdir()** system call to execute the cd command functionality. For the particular command, **cd -**, you must keep track of the previous directory; the system call **getcwd()** may be of use to you here.

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" flag 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 means the character "\t" is not interpreted as tab, rather literally so.

Also note the following example:

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

<<<<< This message contains a |||line feed >>>>>
```

The example does not consider the above command to have redirections or pipes because the corresponding symbols are inside quotation marks. It also interprets the '\n' character due to the -e option given outside the quotation marks.

Your Task:

In this assignment you will design a simple shell which implements a subset of the functionality of the Bourne Again Shell (BASH). You will then write a report as well. The requirements are detailed below and are followed by the feature list and associated rubrics:

(1) Feature Implementation (explicit feature list is defined in the *Shell Features and Rubrics* section below):

- Continually prompt the user for the next command input. Print a custom prompt to be shown before taking each command. This should include your user name, current date-time, and the absolute path to the current working directory. The system calls `getenv("USER")`, `time()+ctime()`, and `getcwd()` will help you with this.
  E.g.: <mark>Sep 23 18:31:46 user:/home/user$</mark>

- Execute commands passed in by the user, parsed by the provided classes:

  - For executing a command from the shell, you must use the `fork()+execvp(...)` 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 `waitpid(...)` function.

  - The provided classes *Tokenizer* and *Command* parse the user input into argument lists stored in a vector. Further documentation is provided in the respective header files.

- Support input redirection from a file (e.g. command < filename) and output redirection to a file (e.g. command > filename). Note that a single command can also have both input and output redirection.

- Allow piping multiple commands together connected by "|" symbols in between them (e.g. command1 | command2). 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 (IPC) mechanism called pipe that is initiated by calling the `pipe()` system call

● Run the user command in the background if the command contains a "`&`" symbol at the end (e.g. **command &** or **command arglist &**). Note that you must avoid creating zombie processes in this case.

● 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 (i.e., `chdir()`) instead of forwarding to `exec()`.

● **For Extra Credit:** Allow $-sign expansion. **See the last command in the Shell grading instructions command list in the following.**

(2) Report Documentation:

Write a report describing your unique design choices, algorithms (e.g., implemented file input/output redirection, piping, etc.) and any implemented bonus features along with its implementation technique. The report must be a minimum of a page. We are not looking for generic items that are common to the entire class as part of this PA instructions, but we are looking for your insights and unique contributions.

Location of the Starter Code:

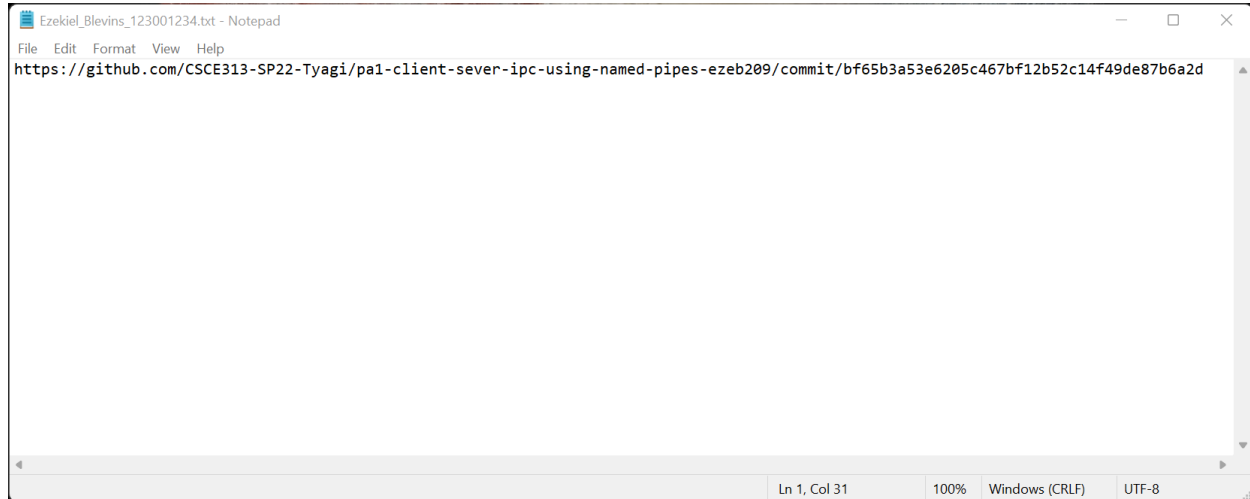This assignment will be hosted using the **GitHub Classroom** platform. You must visit **https://classroom.github.com/a/M2KY2dBp** to create a repository for your project.

How to Begin

PA2 Getting Started video: **https://youtu.be/YToA6gxWkVs**

After you compile and run the starter code, you will see that the program can already handle one word commands (e.g., *ls*) and *exit*. You must further the implementation of the given starter code so that the program can handle commands as mentioned in the features above and in the *Shell Features and Rubrics* section below.

PA 2 Submission Instructions:

Once you have completed the assignment and submitted your final code to GitHub Classroom, on Canvas, turn in a text document named *FirstName_LastName_UIN.txt* containing the link to the commit hash that you are submitting (PA1 example below) and your report named *PA2_FirstName_LastName_UIN.pdf*.

Shell Features and Rubrics:

1. Checking echo (will not have to handle escaped quotation marks): 2 points

```
A. echo "Hello world | Life is Good > Great $"
```

2. Simple commands with arguments: 8 points

```
A. ls

B. ls -l /usr/bin

C. ls -l -a

D. ps aux
```

3. Input/Output redirection: 12 points

```
A. ps aux > a

B. grep /init < a

C. grep /init < a > b # grep output should go to file b
```

4. Single pipe: 6 pts

```
A. ls -l | grep shell
```

5. Two or more pipes: 4 points:

```
A. ps aux | awk '/usr/{print $1}' | sort -r
```

6. Two or more pipes with input and output redirection: 12 points

```
A. ps aux > test.txt

B. awk '{print $1$11}' < test.txt | head -10 | tr a-z A-Z | sort
> output.txt

C. cat output.txt
```

7. Background processes: 5 points # not tested by autograder

```
A. sleep 3 &

B. sleep 2
```

8. Directory processing: 6 points (done with chdir() - do not use fork/exec)

```
A. cd ../../

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

9. User Prompt: 5 points # not tested by autograder

- Print a custom prompt to be shown before taking each command. This should include your user name, current date-time, and the absolute path to the current working directory. E.g.: Sep 23 18:31:46 cartman:/home/cartman$

10. Secret Tests: 25 points

11. Report: 15 points

12. BONUS: $ sign expansion and misc: 15 points

```
A. cat /proc/$(ps|grep bash|head -1|awk '{print $1}')/status

B. mkdir _dir

C. cd $(ls -l | grep '^d'|head -1|awk '{print $9}') # Goes
inside the first directory in the current directory
```

13. BONUS: Open-ended extra credit options: 10 points

A. Command history (pressing Up/Down arrow button goes to previous/next command)

B. Autocomplete by pressing tab