Programming assignment 2

Aggie Shell

Due date

Oct 2, 2023 11:59 PM

Table of content

Academic Integrity

The following actions are strictly prohibited and violate the honor code. **The minimum penalty for plagiarism is a grade of zero and a report to the Aggie honor system office.**

- Uploading assignments to external websites or tutoring websites such as Chegg, Coursehero, Stackoverflow, Bartleby, etc.
- Copying code from external websites or tutoring websites such as Chegg, Coursehero, Stackoverflow, Bartleby, etc.
- Copying code from a classmate or unauthorized sources and submitting it as your work.
- Sharing your code with a classmate.

Keywords

Shell, processes, pipes, redirection

Introduction

In this programming assignment, you will implement your Linux shell, the Aggie Shell. The Linux shell in your OS lets a user navigate through the file system and performs multiple tasks using simple commands. It also offers capabilities of interprocess communication and file input/out redirection. Your shell should function almost as well as Bash. Each Linux command (e.g., cat, echo, cd, etc.) should run as a child process in this assignment. These commands are executables in your system and are recognized by a call to an exec()-family function. Refer to this website for some interesting and commonly used Linux commands.

Features of the Aggie Shell

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 operator 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.

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 use. Even if someone doesn't intend to look at the output of your program, it is still 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 later, 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 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.txt
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.txt
This text came from a file
```

In this example, the content 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 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. After that, the prompt returns, and you can type the next command. You can change this behavior by sending the program to the background and continuing to use 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 ensure 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. Remember that the 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 on 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 useful here.

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 the 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 a tab but 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 Tasks

Design a simple shell that implements a subset of the functionality of the Bourne Again Shell (BASH). The requirements are detailed below and are followed by the feature list and associated rubrics:

1. Feature Implementation: the list of features 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. Example:

```
Sep 23 18:31:46 user:/home/user$
```

- 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.

Getting started

1. Go to the assignment’s GitHub classroom: https://classroom.github.com/a/-ut10TVK
2. Create a repository for your project.
3. Watch the 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.

Please note that this assignment does not use an autograder. However, you can use the pa2-tests.sh script to test your code locally.

Shell Features and Rubrics

1. Checking echo (will not have to handle escaped quotation marks): 5 points
echo “Hello world | Life is Good > Great $”

2. Simple commands with arguments: **10 points**
   a. ls
   b. ls -l /usr/bin
   c. ls -l -a
   d. ps aux

3. Input/Output redirection: **15 points**
   a. ps aux > a

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

4. Single pipe: **8 pts**
   ls -l | grep shell

5. Two or more pipes: **6 points**
   ps aux | awk '/usr/{print $1}' | sort -r

6. Two or more pipes with input and output redirection: **15 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 script
   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 script
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. Example:

```
Sep 23 18:31:46 cartman:/home/cartman$
```

10. Secret Tests: **25 points**

11. 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

12. Bonus: extra credit options: 10 points
   a. Command history (pressing the Up/Down arrow button goes to previous/next command).
   b. Autocomplete by pressing tab.

Q&A

1. Where do I look to get some guidance on getting started?

A ‘where to get started’ video has been created for this purpose. Link to the Video (also posted on PA2 assignment): https://www.youtube.com/watch?v=YToA6gxWkVs

2. My program rapidly prints out the prompt a bunch of times. What could be the reason for this?

This likely means your stdin is broken, and so reading input fails immediately and loops back around. The scenario this happens the most is when you're working on pipes -- you overwrite the 0 fd of the parent process and don't copy stdin back afterward. You can fix this with dup and dup2.

3. What should be the user name we use for the user prompt? Is it the user who is running the shell or the GitHub username?

It is the environment username - the user who is running the shell. You would resolve $USER using the appropriate utilities in C++ related to env. The function call, getenv(“USER”) will help you here.

4. Does grep output anything in "grep /init"? For me, it doesn't, whether I try running in the actual terminal or in my terminal.

Try something other than init; for some reason, in some machines init process does not show.

5. Where am I supposed to connect "cd /home/" to? I am running on MAC OS, so I don't necessarily have a home directory.

/home is an absolute path to the home directory on Linux; you should only be running it in a Linux environment; you cannot run your code on MAC OS.

6. How can I verify that I'm handling the background process correctly?

In the case of background processes that sleep (for example, sleep 20 &), use the command ps to check if the process shows up and then run ps again after the allotted sleep time is over to check if it is finished.

7. Is it possible for a child process to change the working directory of the parent process? If not, how should I approach the cd command? I can change the working directory, but only in the child process, so the parent's working directory never changes.

Don't fork to change directory. Call the appropriate function from the parent, then continue to the next iteration of the while loop. cd doesn't require an exec call.

8. (Bonus - Sign Expansion) Does anyone have any advice on performing the $() expansion? I am currently parsing out the inside and using my fork() loop to take care of

the inside, but I don't know how to get the last output of what's inside the $() into the outer command.

One potential way to do it is to redirect the stdout fd to the write end of a pipe; your pipe implementation code will output to the write end of the pipe, which can be read from the other end (read end) into a string using the read(...) system call. Once you receive this string, you can execute the whole command (by replacing the original sign expansion in the input with the output returned by it).

9. It seems that after forking, my pipe has to be closed on both processes. Is there a mistake? Also, I don't see how to close the pipe after transferring control to the child process executing the command.

The pipe does need to be closed on both sides. So if you are redirecting stdout into the write end of the pipe, you should close the read side and vice versa for stdin. The side that you redirect stdin/stdout onto will be closed once the process whose I/O you redirected ends execution since it'll treat the pipe like it would its stdin or stdout and perform the appropriate cleanup.

10. Why are we constrained to execvp(...) to run the command as a child process? Why can't we use execlp(...) as well?

execlp is a variadic function, whereas the args are in an array of unknown length (at compile time) - there's no (good) way to flatten an array into a variadic function's argument slots.

11. I seem to be printing out my shell prompt just fine, but it seems to be failing the test cases for some reason. Any ideas as to why this may be?

If you do not print out an empty space (" ") after your command prompt, you will fail the test cases.

12. My program works as expected when piping, but I seem to be failing the piping test cases with "/usr" in them. Why would this be?

This is likely because there are no /usr processes in your system; for the purposes of local testing, you can replace the test files containing "/usr" to "/init". But, you must make sure to change them back to "/usr" when pushing to GH.

Code Organization

Differentiate the types of commands in the shell (This discussion does not involve sign expansion):

- change directory (cd)
- Single command (can be a background process or may contain I/O redirection)

- Piped commands (the first command may require input and the last command may require output redirection)

We can direct our control structure as follows:

Perform a check on what the first command of our Tokenizer is.

- If it starts off with a “cd” then you must implement functionality surrounding getcwd(...) and chdir(...). You must keep track of the previous directory in case of a “cd -”.
- Else, if not “cd”, then you know that it is either a command vector of size 1 or a variable length command vector. It is optimal code design to execute the single command as part of the piping structure, but for the sake of simplicity, let’s assume that we’re performing a check on the size of our command vector and directing flow control between a single command and piped commands
    - If single command ->
        - fork to create new child process
        - Perform checks on requiring I/O file redirection using hasInput() and hasOutput() (and redirect using dup2 appropriately)
        - Perform checks if it is a background process using isBackground(); if so store the pid of the child process and periodically wait on it in a non-blocking manner
        - Use execvp to execute command in the child process

    - Else If piped command ->
        - Run a for loop to process each command in order; within this for loop:
            - Create a pipe
            - Fork to create a new child process
            - The child process inherits the pipe as well
            - Within child process:
            - Perform checks for I/O redirection if first or last command
            - If not last command, redirect stdout to write end of pipe
            - Use execvp to execute command in the child process
            - Within parent process:
            - Redirect stdin to read the end of the pipe and close the write end of the pipe within the parent (remember both the child and the parent have access to the write end)
            - If the last command, wait on it to finish executing in the child; you must keep track of the pids of intermediate commands to reap them later as well.

Note: A lot of the functionality is shared between several portions of the program. Consider using functions to shorten your code and make it readable. You may consider implementing things like File I/O redirection and making args array from vector, implemented as function routines.

Piping is hard to wrap your mind around, and it does take a while to completely understand. A further video of piping explaining it diagrammatically is linked here.

Test-plan Development

When testing "cd" functionality, an important scenario to be considered is running multiple consecutive "cd -" together. Running "cd -" takes you back to the previous directory, so you will be switching back and forth between the same two directories upon consecutive "cd -". The current directory becomes the previous, and the previous directory becomes the current.

When testing out file I/O redirection, remember that there is the possibility that some commands can take in both input and output redirection at the same time. You are recommended to initially test whether you have implemented each redirection operator in isolation before testing both together.

For piping, you can generally expect there to be no "edge" cases, per se. Ideally, if your code works on any one variable-length piped command string, you can expect it to work on all variable piped commands. Start off with a simple 2-length piped command and work your way up to larger, variable-length piped commands.

When testing background processes, you may find it useful to use the "ps" command to verify whether you are correctly reaping them. For example, if you are running the background process, "sleep 10 &", first check that the command prompt returns immediately. If it does, wait for 10 seconds and run "ps". If you see the sleep process showing up as <defunct>, it means you have not reaped it correctly (i.e., it is a zombie process). If it does not show up, you have correctly implemented the background process reaping functionality.

Coding Dos and Don'ts

Do's

- When implementing pipes, you must close the write-end file descriptor in the parent. This is because the read end of the pipe only closes when it receives the EOF when there are no writer processes having the pipe open. If you do not close the write end in the parent, the pipe will remain open as the read end expects the writer process to write something to the pipe.
- You must create a pipe for every command in the loop.

- Note that the parent redirects stdin every iteration of the piping command loop to receive the data from the command executing as a child process. When the parent process tries to take in input for the next command from the terminal, it cannot do so because file descriptor 0 (stdin) is still pointing elsewhere. You must redirect 0 back to default so that the parent can take in input from cin. You can do this by storing default stdin in another file descriptor through dup and redirecting 0 to point back to this stored file descriptor after you are done with piping.
- You must use getenv("USER") and not getlogin() when implementing your shell prompt.

Don'ts

- You must be careful not to redirect stdout to the write end of the pipe in the child for the last command; this is because you want the final output (the result of executing the last command) to be printed out the default stdout (i.e., terminal) or to a file if an output redirection operator exists.
- An important thing to remember when waiting on background processes in a non-blocking manner is where you choose to implement this functionality. Remember, if you implement this above your cin, you will run the risk of not having reaped it when you check for it being reaped the first time.