Assignment 2: Building your own shell

by Ananya, 12040180

In this assignment, we made a shell which emulates a linux shell. We made C programs for this in which rather than built-in functions system calls were used.

I used the following system calls:

#	System Calls	Definition
1	fork( )	**fork**( ) creates a new process by duplicating the calling process. The new process is referred to as the *child* process.
2	dup2( )	The **dup**( ) system call allocates a new file descriptor that refers to the same open file description as the descriptor *oldfd*. The **dup2**( ) system call performs the same task as **dup**( ), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in *newfd*.
3	execvp( )	The **exec**( ) family of functions replaces the current process image with a new process image.
4	close( )	**close**( ) closes a file descriptor, so that it no longer refers to any file and may be reused.
5	perror( )	The **perror**( ) function produces a message on standard error describing the last error encountered during a call to a system or library function.
6	exit( )	The **exit**( ) function causes normal process termination and the least significant byte of *status* (i.e., *status & 0xFF*) is returned to the parent.
7	waitpid( )	The **waitpid**( ) system call suspends execution of the calling thread until a child specified by *pid* argument has changed state.
8	open( )	The open( ) system call opens the file specified by pathname. If the specified file does not exist, it may optionally (if O_CREAT is specified in flags) be created by open().
9	pipe( )	**pipe**() creates a pipe, a unidirectional data channel that can be used for interprocess communication.

Every *shell* is structured as the following loop:

1. Print out a prompt
2. Read a line of input from the user
3. Parse the line into the program name, and an array of parameters
4. Use the **fork( )** system call to spawn a new child process
   - The child process then uses the **exec( )** system call to launch the specified program
   - The parent process (the shell) uses the **wait( )** system call to wait for the child to terminate
5. When the child (i.e. the launched program) finishes, the shell repeats the loop by jumping to 1.

Whenever the *shell* executes a new command, it spawns a child *shell* and lets the child execute the command. This behavior is implemented with the **fork( )** and **execvp( )** system calls. If the *shell* receives a sequence of commands, each concatenated with "|", it must recursively create children whose number is the same as that of commands. Which child executes which command is a kind of tricky. The farthest offspring will execute the first command, the grand child will execute the 2nd last, and the child will execute the last command. Their standard input and output must be indirected accordingly using the **pipe** and **dup2** system calls.

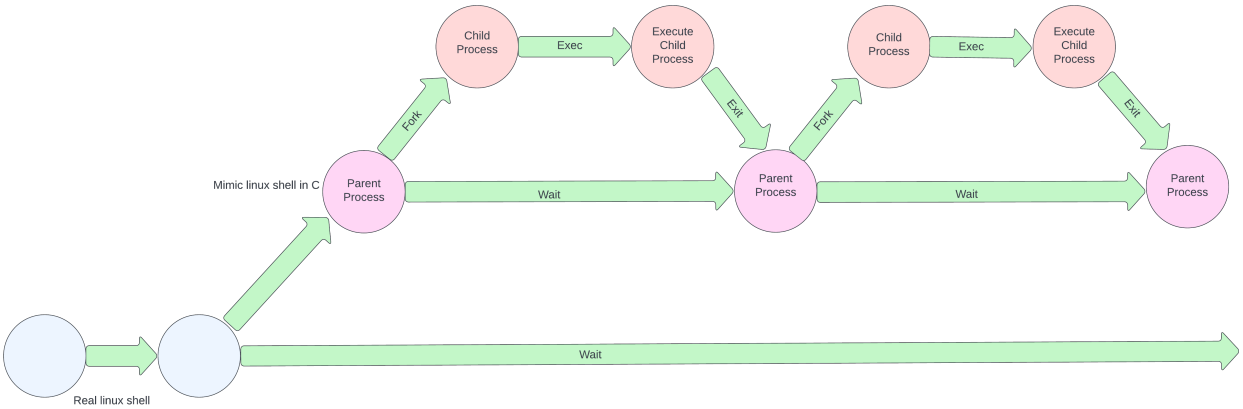


Diagram when a shell is invoked

PS: To run the my codes, instructions are available in Readme.md file. Code is explained in the comments.

PART A

In part a, we were asked to implement some simple commands and customize the prompt. With the screenshot attached below you can see the customization and commands which I ran on my mimic shell. I got some important observations. Since we had to mimic a very basic shell, most commands were having the same behaviour as the commands being run on a real terminal.

```
→  12040180_os_assignment2 ls
Readme.md cmdline.c cmdline.h parta.c   partb.c   partc.c   shell.c   test
→  12040180_os_assignment2 gcc cmdline.c parta.c -o a
→  12040180_os_assignment2 ./a
Ananya:12040180>> ls
Readme.md       cmdline.c       parta.c         partc.c         test
a               cmdline.h       partb.c         shell.c
Ananya:12040180>> pwd
/Users/ananyahooda/Desktop/12040180_os_assignment2
Ananya:12040180>> mkdir hello
Ananya:12040180>> cd hello
Ananya:12040180>> pwd
/Users/ananyahooda/Desktop/12040180_os_assignment2/hello
Ananya:12040180>> touch file1 file2
Ananya:12040180>> ls
file1   file2
Ananya:12040180>> cd ..
Ananya:12040180>> pwd
/Users/ananyahooda/Desktop/12040180_os_assignment2
Ananya:12040180>> ls -la
total 176
drwxr-xr-x@ 12 ananyahooda  staff    384 Mar 13 22:23 .
drwx------@ 35 ananyahooda  staff   1120 Mar 13 21:30 ..
-rw-r--r--   1 ananyahooda  staff    887 Mar 13 22:21 Readme.md
-rwxr-xr-x   1 ananyahooda  staff  51086 Mar 13 22:22 a
-rw-r--r--@  1 ananyahooda  staff   1169 Mar 13 22:01 cmdline.c
-rw-r--r--@  1 ananyahooda  staff    395 Mar 13 08:05 cmdline.h
drwxr-xr-x   4 ananyahooda  staff    128 Mar 13 22:23 hello
-rw-r--r--@  1 ananyahooda  staff   1355 Mar 13 08:05 parta.c
-rw-r--r--@  1 ananyahooda  staff   3317 Mar 13 08:05 partb.c
-rw-r--r--@  1 ananyahooda  staff   5111 Mar 13 08:05 partc.c
-rw-r--r--@  1 ananyahooda  staff   5401 Mar 13 08:05 shell.c
drwxr-xr-x   2 ananyahooda  staff     64 Mar 13 21:15 test
Ananya:12040180>>
```

Commands with same behaviour as that of a real terminal

In the screen shot attached below you can see that in echo command environment variables are not being echoed rather their as it is value, i.e. being treated a string is being displayed. Moreover here the double quotes are also being printed along with the string.

```
Ananya:12040180>> ps
  PID TTY           TIME CMD
84239 ttys000    0:00.16 -zsh
84300 ttys000    0:00.01 ./a
83687 ttys004    0:00.25 -zsh
83830 ttys004    0:00.01 ./a.out
81287 ttys005    0:00.27 -zsh
81414 ttys005    0:00.00 ./a.out
81435 ttys005    0:00.01 ./a.out
81553 ttys005    0:00.00 ./a.out
Ananya:12040180>> ps -a
  PID TTY           TIME CMD
84237 ttys000    0:00.02 login -pfl ananyahooda /bin/bash -c exec -la zsh /bin/zsh
84239 ttys000    0:00.16 -zsh
84300 ttys000    0:00.01 ./a
84366 ttys000    0:00.01 ps -a
83685 ttys004    0:00.03 login -pfl ananyahooda /bin/bash -c exec -la zsh /bin/zsh
83687 ttys004    0:00.25 -zsh
83830 ttys004    0:00.01 ./a.out
81286 ttys005    0:00.03 login -pfl ananyahooda /bin/bash -c exec -la zsh /bin/zsh
81287 ttys005    0:00.27 -zsh
81414 ttys005    0:00.00 ./a.out
81435 ttys005    0:00.01 ./a.out
81553 ttys005    0:00.00 ./a.out
Ananya:12040180>> echo $pid
$pid
Ananya:12040180>> echo state
state
Ananya:12040180>> echo $state
$state
Ananya:12040180>> echo $
$
Ananya:12040180>> echo $$
$$
Ananya:12040180>> echo $PS1
$PS1
Ananya:12040180>> echo ananya
ananya
Ananya:12040180>> echo "ananya"
"ananya"
```

Commands with somewhat different behaviour as that of a real terminal

Here if in terminal a wrong command is entered, perror( ) is used and error is displayed. Here shell is quitted through both Ctrl+C and exit command.

```
Ananya:12040180>> man ps
Ananya:12040180>> vim hello.py
Ananya:12040180>> python3
Python 3.10.1 (v3.10.1:2cd268a3a9, Dec  6 2021, 14:28:59) [Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> exit()
Ananya:12040180>> ls
Readme.md       cmdline.c       hello           parta.c         partc.c         test
a               cmdline.h       hello.py        partb.c         shell.c
Ananya:12040180>> python3 hello.py
hello
Ananya:12040180>> cd
Ananya:12040180: Bad address
Ananya:12040180>> hi/
Ananya:12040180: No such file or directory
Ananya:12040180>> pwd
/Users/ananyahooda/Desktop/12040180_os_assignment2
Ananya:12040180>> exit
➜  12040180_os_assignment2 ./a
Ananya:12040180>> ^C
➜  12040180_os_assignment2 ./a
Ananya:12040180>> ls
Readme.md       cmdline.c       hello           parta.c         partc.c         test
a               cmdline.h       hello.py        partb.c         shell.c
Ananya:12040180>> clear
```

Wrong commands being entered and then giving error messages and exit commands

Commands executed in above screenshots in the mimic shell: **ls, pwd, cd <dir_name>, cd.. , touch, ls -la, ps, ps -a, echo, man, vim, python3, exit, clear, Ctrl+C**. Other commands can be run as well.

PART B

1. The 1st pat of PART B included piping.
   Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process

- It is possible to have a series of processes arranged in a a pipeline, with a pipe between each pair of processes in the series.
- Implementation: A pipe can be implemented as a 10k buffer in main memory with 2 pointers, one for the FROM process and one for TO process
- One process cannot read from the buffer until another has written to it
- The UNIX command-line interpreter provide a pipe facility. e.g. ls | wc -l. This command runs the prog1 program and send its output to the more program.

```
➜  12040180_os_assignment2 gcc partb.c cmdline.c -o b
➜  12040180_os_assignment2 ./b
Ananya:12040180>> pwd
/Users/ananyahooda/Desktop/12040180_os_assignment2
Ananya:12040180>> ls
Readme.md       b               cmdline.h       hello.py        partb.c         shell.c
a               cmdline.c       hello           parta.c         partc.c         test
Ananya:12040180>> ls | grep cmd
cmdline.c
cmdline.h
Ananya:12040180>> ls | grep cmd | wc -l
       2
Ananya:12040180>> ls | grep cmd | grep line.h
cmdline.h
```

Piping

Here you can see in the screen shot, piping is working and giving correct outputs.

```
➜  12040180_os_assignment2 ls | hoo/
zsh: no such file or directory: hoo/
➜  12040180_os_assignment2 hoo/ | ls
zsh: no such file or directory: hoo/
Readme.md    Report.pages a           b           c           cmdline.c    cmdline.h    parta.c      partb.c      partc.c      shell.c
➜  12040180_os_assignment2 ./b
Ananya:12040180>> ls | hoo/
Ananya:12040180: No such file or directory
Ananya:12040180>> hoo/ | ls
Ananya:12040180: No such file or directory
Readme.md       a               c               cmdline.h       partb.c         shell.c
Report.pages    b               cmdline.c       parta.c         partc.c
```

Piping with some erroneous commands

In the above screenshot, you can see that the shell I made mimics the real shell even when a erroneous command is there. Based on the basic concept of piping.

2. In 2nd part of PART B, && is implemented.

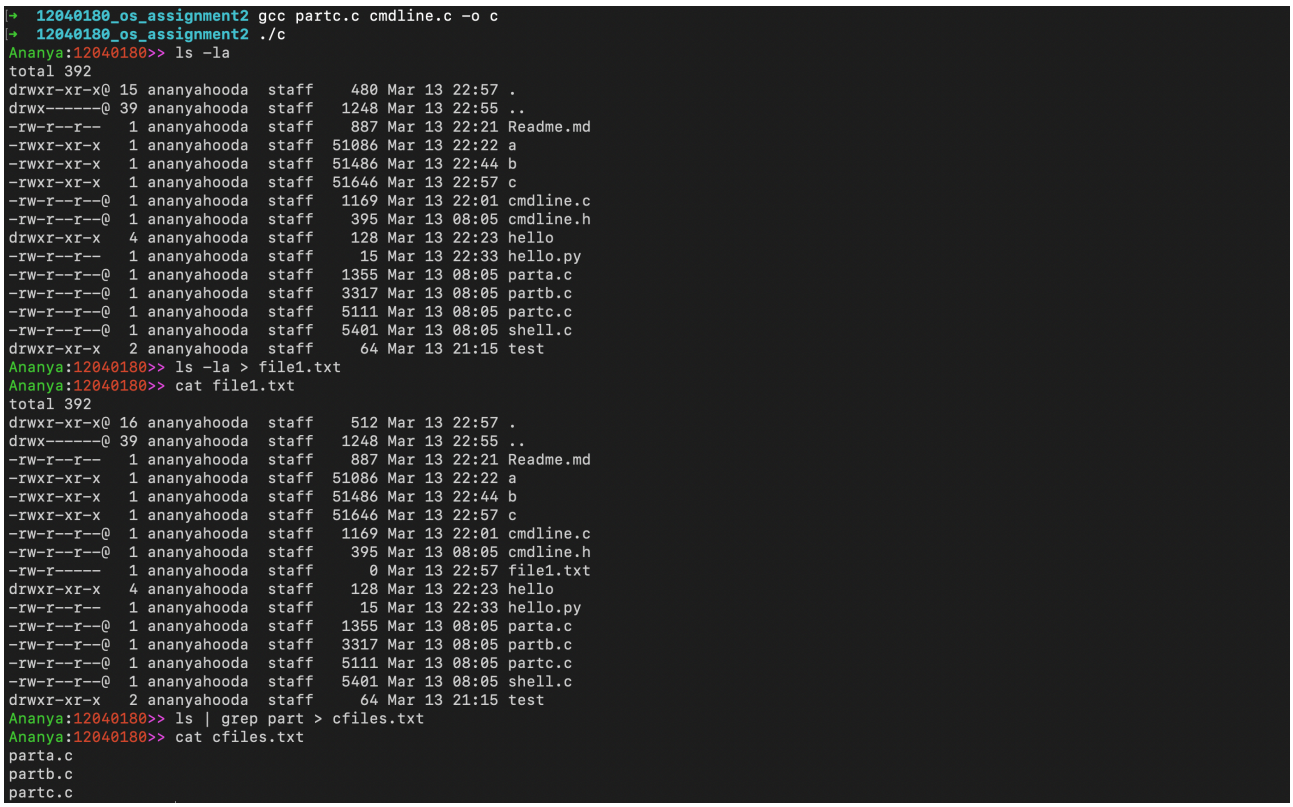
The intent is to execute the command that follows the && only if the first command is successful. It intends to prevent the running of the second process if the first fails.

```
Ananya:12040180>> ls
Readme.md       a.out           cfiles.txt      file1.txt       parta.c         shell.c
Report.pages    b               cmdline.c       hello           partb.c         test
a               c               cmdline.h       hello.py        partc.c
Ananya:12040180>> ps
  PID TTY          TIME CMD
84239 ttys000    0:00.35 -zsh
84430 ttys000    0:00.01 ./a
84467 ttys000    0:00.01 ./b
84660 ttys000    0:00.01 ./c
86278 ttys002    0:00.12 -zsh
86341 ttys002    0:00.01 ./b
81287 ttys005    0:00.27 -zsh
81414 ttys005    0:00.00 ./a.out
81435 ttys005    0:00.01 ./a.out
81553 ttys005    0:00.00 ./a.out
Ananya:12040180>> ls && ps
Readme.md       a.out           cfiles.txt      file1.txt       parta.c         shell.c
Report.pages    b               cmdline.c       hello           partb.c         test
a               c               cmdline.h       hello.py        partc.c
  PID TTY          TIME CMD
84239 ttys000    0:00.35 -zsh
84430 ttys000    0:00.01 ./a
84467 ttys000    0:00.01 ./b
84660 ttys000    0:00.01 ./c
86278 ttys002    0:00.12 -zsh
86341 ttys002    0:00.01 ./b
81287 ttys005    0:00.27 -zsh
81414 ttys005    0:00.00 ./a.out
81435 ttys005    0:00.01 ./a.out
81553 ttys005    0:00.00 ./a.out
Ananya:12040180>> ls && ps && ls | grep part
Readme.md       a.out           cfiles.txt      file1.txt       parta.c         shell.c
Report.pages    b               cmdline.c       hello           partb.c         test
a               c               cmdline.h       hello.py        partc.c
  PID TTY          TIME CMD
84239 ttys000    0:00.35 -zsh
84430 ttys000    0:00.01 ./a
84467 ttys000    0:00.01 ./b
84660 ttys000    0:00.01 ./c
86278 ttys002    0:00.12 -zsh
86341 ttys002    0:00.01 ./b
81287 ttys005    0:00.27 -zsh
81414 ttys005    0:00.00 ./a.out
81435 ttys005    0:00.01 ./a.out
81553 ttys005    0:00.00 ./a.out
parta.c
partb.c
partc.c
```

exection of &&

PART C

In this part we had to implement *output redirection(>).*
Every Unix-based operating system has a concept of “a default place for output to go”. It is called as “standard output” , or “stdout”. Your shell is constantly watching that default output place. When your shell sees new output there, it prints it out on the screen so that you, the human, can see it. Otherwise echo hello would send “hello” to that default place and it would stay there forever.
A file descriptor, or FD, is a positive integer that refers to an input/output source. For example, stdin is 0, stdout is 1, and stderr is 2. Those might seem like arbitrary numbers, because they are: the POSIX standard defines them as such, and many operating systems (like OS X and Linux) implement at least this part of the POSIX standard.



Standard output redirection

This screenshot shows us output redirection into a file.