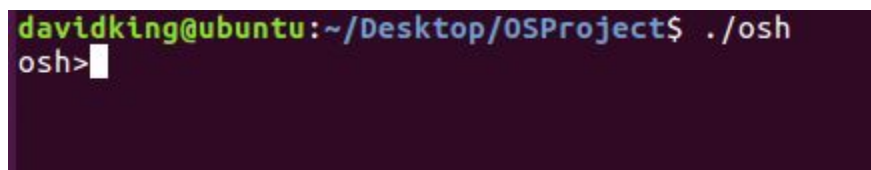


In open source environment like Unix allows an operator to dedicate that environment for a given task or project. The ability to track or log the input commands, allows the user to access the most recently entered commands to verify inputs and outputs. This can be beneficial for troubleshooting purposes. For this project we were to design a shell interface that accepts user commands and execute the input command in a separate process. This project is an example of process concepts and scheduling. A parent process first examines the input command and then create a separate child process to perform the command. If more than one command is entered to the shell, another child process would be created to execute the command. For certain commands, like piping and redirection, the parent process creates two child processes that pipes information from one to the other.

To execute the shell, first the Makefile and UnixShell.c programs must be within the same folder. Using the command prompt or terminal, browse to selected folder. To run Unix Shell History Feature, enter “make all”, without the quotations. To then enter the Unix shell, next enter “./osh”. The user has now entered into the shell, and will track commands entered. Details below show some of the features accompanied by the Unix shell. Figure 1 shows how to enter the shell, and what it looks like once entered.



```
davidking@ubuntu:~/Desktop/OSProject$ ./osh
osh>
```

Figure 1: User Entering Unix shell by command “./osh”

Unix Shell with History Feature Program:

After initializing library calls and defining functions and global variables, the program enters the main program to prime the shell. As the operator enters a command, a series of checks are parsing the argument looking for key indicators within the command. Some of these indicators include help, history, and exit. For all commands that are not identified, a separate child process is created for the command to execute. Some of the History shell limitations are predefined within the program and can be adjusted to modify specific for the shell a user is interested in building.

```
#define MAX_LINE 80 /* the maximum length of command */  
#define HISTORY_SIZE 10 /* history */
```

Figure 2: Shell Limitations pre-defined in Program

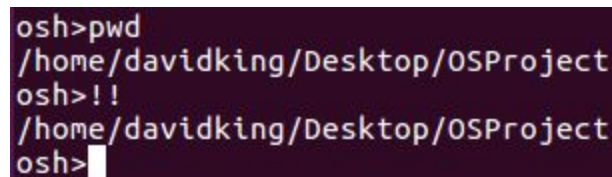
The maximum length of the command is limited to eighty characters, and the number of commands that will be displayed from the “history” command is ten. As new commands are entered, the tenth command is popped off the history stack, and new commands are appended to the stack.

```
osh>history  
10 ls -l  
9 pwd  
8 ps  
7 cal  
6 who  
5 date  
4 dir  
3 clear  
2 ls -la  
1 history  
osh>
```

Figure 2: “History” command showing last ten entered commands

The program works by parsing through input commands and identifying different arguments in the command. Depending on the arguments defined, separate functions are called to execute the command, and add the command to the history stack. After initial parsing, a child process is created to execute the command. If multiple commands exist within the input, separate child processes are created to run the commands. From the executing child process, the result is redirected back to the parent process and displayed on the terminal. At the same time, the history stack can be called using history to display the given shell input commands.

If a user enters “!!” the terminal will respond with the previously entered command.



```
osh>pwd
/home/davidking/Desktop/OSProject
osh>!!
/home/davidking/Desktop/OSProject
osh>
```

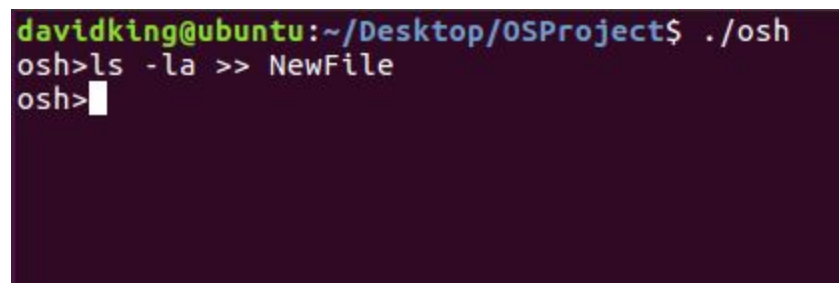
Figure 3: Example of Shell Command “!!”

Once a history stack has been established, a given number of command can be recalled from an index within the limitations of the history stack. Command “!n” will re-enter the command of given index n. Figure 5 below shows an example of the “!n” command by re executing the fifth command entered. This figure also show more in depth how the history of commands is added onto the stack by showing previous commands entered, as well as the entire history entered since the start of the shell.

```
davidking@ubuntu:~/Desktop/OSProject$ ./osh
osh>pwd
/home/davidking/Desktop/OSProject
osh>ps
  PID TTY          TIME CMD
  6280 pts/4    00:00:00 bash
  6510 pts/4    00:00:00 osh
  6512 pts/4    00:00:00 ps
osh>ls -l
total 52
-rw-rw-r-- 1 davidking davidking  420 Nov 28 16:15 File
-rwxrw-rw- 1 davidking davidking  144 Nov 28 15:58 Makefile
-rwxrwxr-x 1 davidking davidking 18296 Nov 28 16:00 osh
-rwxrw-rw- 1 davidking davidking 10408 Nov 28 15:58 UnixShell.c
-rw-rw-r-- 1 davidking davidking 11304 Nov 28 16:00 UnixShell.o
osh>dir
File Makefile  osh  UnixShell.c  UnixShell.o
osh>history
5 pwd
4 ps
3 ls -l
2 dir
1 history
osh>!5
/home/davidking/Desktop/OSProject
osh>
```

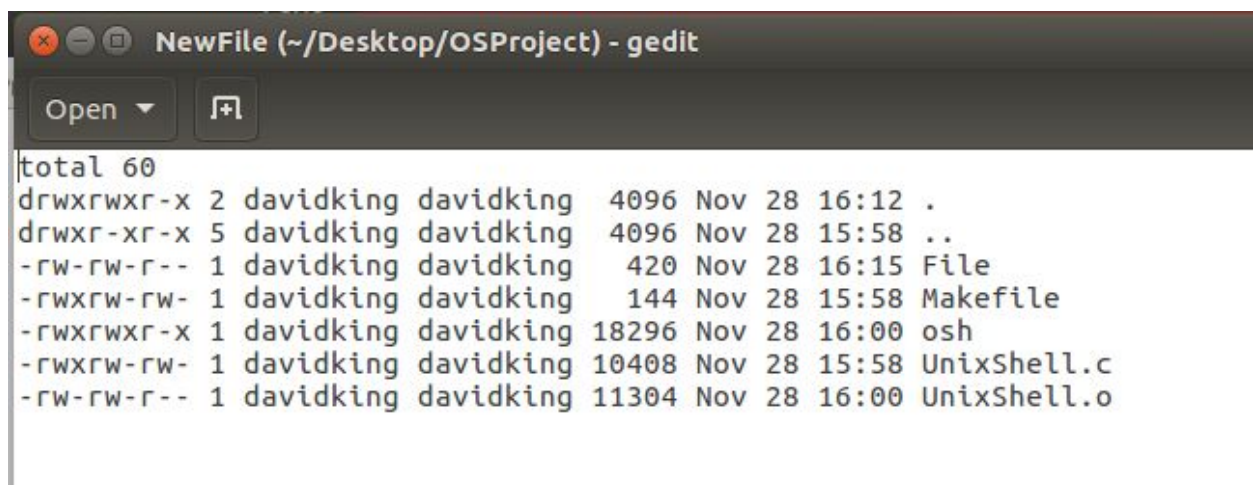
Figure 4: Shell Command “!n” example

The Unix Shell also works on creating a separate file with input commands as shown in Figures 5 and 6. When another command is entered for the same file, the file is overwritten with the input command. In this way, the shell supports a form of redirection. Figure 5 demonstrates the use of redirection on the command “ls”. Notice how arguments for the command, “-la”, also continue to work despite having a second command after it. Figure 6 shows the file that has the redirected output. Notice that the contents of the file are the same if the user had only entered the “ls -la” command.

A terminal window with a dark purple background. The prompt is 'davidking@ubuntu:~/Desktop/OSProject\$./osh'. The user enters 'osh>ls -la >> NewFile'. The prompt changes to 'osh>' with a white cursor.

```
davidking@ubuntu:~/Desktop/OSProject$ ./osh
osh>ls -la >> NewFile
osh>
```

Figure 5: Shell Input Command with New File Creation

A screenshot of a gedit window titled 'NewFile (~/Desktop/OSProject) - gedit'. The window shows the contents of the 'NewFile' file, which is a list of directory entries from the 'ls -la' command. The text is as follows:

```
total 60
drwxrwxr-x 2 davidking davidking 4096 Nov 28 16:12 .
drwxr-xr-x 5 davidking davidking 4096 Nov 28 15:58 ..
-rw-rw-r-- 1 davidking davidking 420 Nov 28 16:15 File
-rwxrw-rw- 1 davidking davidking 144 Nov 28 15:58 Makefile
-rwxrwxr-x 1 davidking davidking 18296 Nov 28 16:00 osh
-rwxrw-rw- 1 davidking davidking 10408 Nov 28 15:58 UnixShell.c
-rw-rw-r-- 1 davidking davidking 11304 Nov 28 16:00 UnixShell.o
```

Figure 6: Created File with Input Command

To run multiple commands within Ubuntu, commands can be separated by “&&”. The second command will execute if the first one exited with a non-zero status. Command B displays before Command A to the terminal. Figure 7 shows the usage of “&&” within the shell. Notice that arguments to commands still work for the specific command it belongs to.

```
osh>ls -l && pwd
/home/davidking/Desktop/OSProject
\total 56
-rw-rw-r-- 1 davidking davidking 420 Nov 28 16:15 File
-rwxrw-rw- 1 davidking davidking 144 Nov 28 15:58 Makefile
-rw-rw-r-- 1 davidking davidking 480 Nov 28 16:27 NewFile
-rwxrwxr-x 1 davidking davidking 18296 Nov 28 16:00 osh
-rwxrw-rw- 1 davidking davidking 10408 Nov 28 15:58 UnixShell.c
-rw-rw-r-- 1 davidking davidking 11304 Nov 28 16:00 UnixShell.o
osh>
```

Figure 7: Shell Command(s) Example “&&”

In the Unix Shell, it's possible to run multiple commands that are pipelined using arguments.

```
osh>ls -l >> File
osh>
osh>dir
A File Makefile NewFile osh UnixShell.c UnixShell.o
osh>
osh>
osh>cat File | more
total 56
-rw-rw-r-- 1 davidking davidking 368 Nov 30 06:06 A
-rw-rw-r-- 1 davidking davidking 0 Nov 30 06:13 File
-rw----- 1 davidking davidking 144 Nov 29 16:50 Makefile
-rw-rw-r-- 1 davidking davidking 480 Nov 28 16:27 NewFile
-rwxrwxr-x 1 davidking davidking 18296 Nov 29 16:51 osh
-rw----- 1 davidking davidking 11193 Nov 29 16:50 UnixShell.c
-rw-rw-r-- 1 davidking davidking 11328 Nov 29 16:51 UnixShell.o
osh>
osh>
```

Figure 8: Shell Command Example “|”

Within the shell, running a single command concurrently using an ampersand “&”. When a program is called without an ampersand the shell, the parent process will wait until that program is finished, before the user can enter another command. With an ampersand, the user can continue using the shell normally.

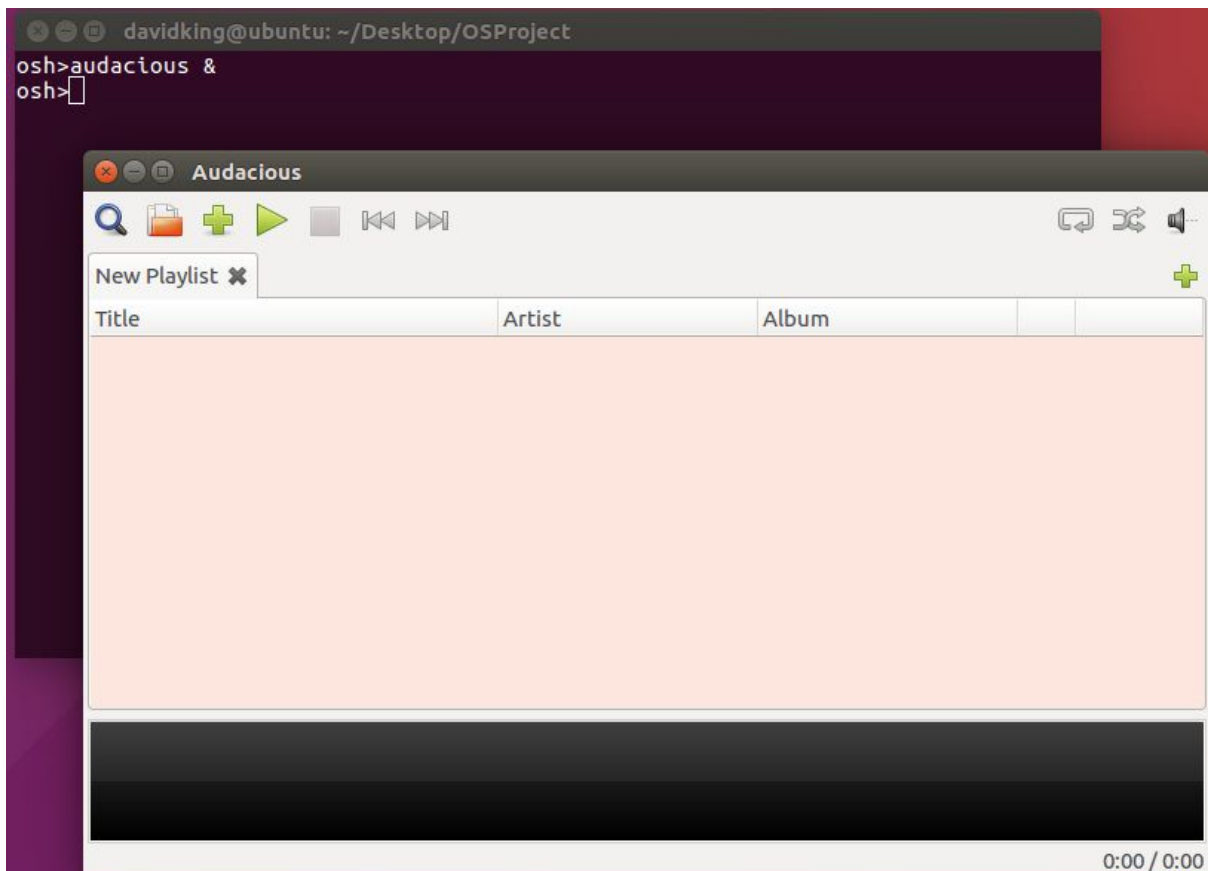
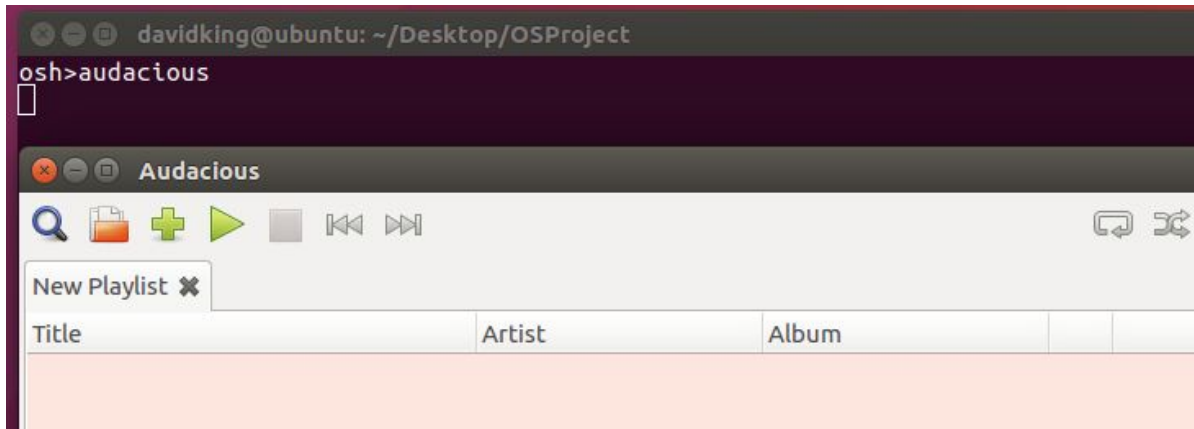


Figure 9: Shell Concurrency Example “&”

Within the Unix Shell the user can still navigate through the host file explorer. Using standard Ubuntu commands “cd” and “pwd”. The command “cd” had to be supported because “execvp” does not run this command correctly.

```
osh>pwd
/home/davidking/Desktop/OSProject
osh>cd ..
osh>dir
HW1Q7_code.zip  notesOnProject  OS_ProgrammingQuestions.odt
nooe            old          OSProject
osh>pwd
/home/davidking/Desktop
osh>cd OSProject
osh>pwd
/home/davidking/Desktop/OSProject
osh>
```

Figure 10: Shell Change Directory Commands

At any time, the user can input the command “Help” which will bring up a help menu within the terminal. This help menu is used to guide the user through the functionality of the Unix Shell.

```
osh>help
osh help: This is a simple shell with history feature.

Currently does not support tab complete or arrow keys.
Incorrectly using "cat" command, especially with "|",
may cause shell to behave unpredictably.
Character limit of 80.
Can use arguments with commands.

1 Ctrl-C: terminates osh
2 exit: ends current session of osh
3 history: displays 10 most recent commands
4 !!: execute most recent command
5 !n: execute nth command up to 10
6 cd ...: change directory to location ...
7 command1 && command2: execute command1 and command2
8 command1 | command2: output of command1 is input to command2
9 command >> filename: overwrites output of command to a file

osh>
```

Figure 11: Shell Command “Help”

If the entered command is “Exit”, the operator is removed from the Unix history shell.

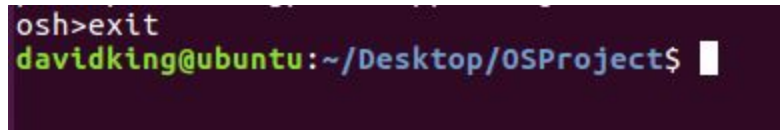
A terminal window with a dark purple background. The first line shows the command 'osh>exit' in white text. The second line shows the prompt 'davidking@ubuntu:~/Desktop/OSProject\$' in green and blue text, followed by a white cursor bar.

Figure 12: Shell Command “Exit”

Difficulties met during the project

Although the project did not call for supporting multiple commands, we decided to support it within our shell. It currently can support at most two commands, using either “&&”, “|”, or “>>”. This proved to be difficult to implement, hence why only two commands are supported and not any number of commands. To solve this issue, we made separate functions that ran these instances separately. A parent process creates two child processes. In the case of “&&”, the child processes simply run one command each. As for the other two, the child process communicate needed information with one another via Unix pipes. Similarly, a separate function had to be created to identify and split the command line arguments appropriately because the “execvp” function does not support certain arguments that would be present in these commands.

Other difficulties that arose during the project was getting the “!!” and “!n” commands to work properly. Certain error handling had to be added to cover some of the edge cases that could arise. For “!!”, a special flag had to be added to determine whether there were actually any commands in the history to run. We also decided to re add the commands to the history file when “!!” and “!n” re executed previous commands. The reason for this is that if we hadn’t, there is a possibility for the history to only contain “!!” and “!n” commands, thus causing either a segmentation fault or enter an infinite loop. By simply just adding the commands back to the history file, this problem is avoided altogether.