Operating Systems Laboratory (CS39002)
Spring Semester 2018-2019

Assignment 2: Implementation of a rudimentary command-line interpreter running on the Linux operating system

Assignment given on: January 14, 2019

Assignment deadline: January 28, 2019, 1:00 PM

Interim evaluation: January 21, 2019

Implement a shell that will run as an application program on top of the Linux kernel. The shell will accept user commands (one line at a time), and execute the same. The following features must be implemented:

a) **Run an external command:** The external commands refer to executables that are stored as files. They have to be executed by spawning a child process and invoking **`execlp()`** or some similar system calls. Example user commands:

```
./a.out myprog.c
cc –o myprog myprog.c
ls -l
```

b) **Run an external command by redirecting standard input from a file:** The symbol "**<**" is used for input redirection, where the input will be read from the specified file and not from the keyboard. You need to use a system call like "**dup()**" or "**dup2()**" to carry out the redirection. Example user command:

```
./a.out < infile.txt
sort < somefile.txt
```

c) **Run an external command by redirecting standard output to a file:** The symbol "**>**" is used for output redirection, where the output will be written to the specified file and not to the screen. You need to use a system call like "**dup()**" or "**dup2()**" to carry out the redirection. Example user commands:

```
./a.out > outfile.txt
ls > abc
```

d) **Combination of input and output redirection:** Here we use both "<" and ">" to specify both types of redirection. Example user command:

```
./a.out < infile.txt > outfile.txt
```

e) **Run an external command in the background with possible input and output redirections:** We use the symbol "**&**" to specify running in the background. The shell prompt will appear and the next command can be typed while the command is being executed in the background. Example user commands:

```
./a.out &
./myprog < in.txt > out.txt &
```

f) **Run several external commands in the pipe mode:** The symbol "**|**" is used to indicate pipe mode of execution. Here, the standard output of one commands will be redirected to the standard input of the next command, in sequence. You

need to use the "**`pipe()`**" system call to implement this feature. Example user commands:

```
ls | more
cat abc.c | sort | more
```

g) **Run several external commands in the pipe mode and with possible input and output redirection in background:** This is a combination of the earlier modes. Example user commands:

```
ls | cat > outfile.txt &
cat < abc.c | sort | cat > outfile.txt &
```

Implementation Guideline:

For redirecting the standard input or output, you can refer to the book: "*Design of the Unix Operating System*" by Maurice Bach. Actually, the kernel maintains a file descriptor table or FDT (one per process), where the first three entries (index 0, 1 and 2) correspond to standard input (**`stdin`**), standard output (**`stdout`**), and standard error (**`stderr`**). When files are opened, new entries are created in the table. When a file is closed, the corresponding entry is logically deleted. There is a system call **`dup(xyz)`**, which takes a file descriptor **`xyz`** as its input and copies it to the first empty location in FDT. So if we write the two statements: **`close(stdin); dup(xyz);`** the file descriptor **`xyz`** will be copied into the FDT entry corresponding to **`stdin`**. If the program wants to read something from the keyboard, it will actually get read from the file corresponding to **`xyz`**. Similarly, to redirect the standard output, we have to use the two statements: **`close(stdout); dup(xyz);`**

Normally, when the parent forks a child that executes a command using **`execlp()`** or **`execvp()`**, the parent calls the function **`wait()`**, thereby waiting for the child to terminate. Only after that, it will ask the user for the next command. However, if we want to run a program in the background, we do not give the **`wait()`**, and so the parent asks for the next command even while the child is in execution.

A pipe between two processes can be created using the **`pipe()`** system call, followed by input and output redirection. Consider the command: **`ls | more`**. The parent process finds out there is a pipe between two programs, creates a pipe, and forks two child processes (say, X and Y). X will redirect its standard output to the output end of the pipe (using **`dup()`**), and then call **`execlp()`** or **`execvp()`** to execute **`ls`**. Similarly, Y will redirect its standard input to the input end of the pipe (again using **`dup()`**), and then call **`execlp()`** or **`execvp()`** to execute **`more`**. If there is a pipe command involving N commands in general, then you need to create N-1 pipes, create N child processes, and connect each pair of consecutive child processes by a pipe.

Submission Guideline:

- Create a single program for the assignment, and name it Ass2_<groupno>.c or .cpp (replace <groupno> by your group number), and upload it.
- Submit an interim version within January 21, 2019 for intermediate evaluation.
- You must show the running version of the program(s) to your assigned TA during the lab hours.

Evaluation Guidelines:

While entering marks, the partwise break up should also be entered according to the marking guidelines given below. There is a separate component for individual assessment, based on how the student answers questions.

Sl	Items	Marks
(a)	Process creation and running an external command	8
(b)	Redirection of stdin	8
(c)	Redirection of stdout	5
(d)	Redirection of stdin and stdout together	3
(e)	Running an external command in background	8
(f)	Running an external command in background with I/O redirection	8
(g)	Running several external commands in pipe mode	12
(h)	Running several external commands in pipe mode with possible I/O redirection	8
(i)	Avoiding simultaneous redirection to pipe and file and other programming aspects	5
	Total	65