# Operating System Lab 1 - Group - Souptik Paul, Venkat Sai Dinesh Uddagiri

The approach we used for the implementation is that we started with once functionality, tested it and then proceeded on to the subsequent functionalities. Prior to implementing the functionalities and program specifications we have gone through some materials for understanding the usage of some of the basic functions such as ((fork(),exec(), wait(), pipe(),…)). After writing and compiling the code, we then executed the test/example commands, in the lab manual. This helped us understand the execution of the code.

We have started adding our part of the code, starting from the function RunCommands() which is the core of our implementation. We call this function from the previously defined(it was already written in the skeleton code) function RunCommands(). This is due to the fact that we were using the DebugPrintCommand function for debugging purpose, throughout our implementation(we have removed it as was mentioned!).

# User defined functions:

RunCommands() - This is the core of our functionality. We use this to defined key variables and perform checks for calling the subsequent functions accordingly, for executing various functionalities.

Commands\_Execution() – This function is used to execute Single word commands, Background commands and commands with one or multiple pipes.

Commands\_Execution\_Defined\_cmd() – This function is used to execute defined commands such as “cd” and “exit” using built-in functions of c programming.

Commands\_ExecutionIO() - This function is used to execute the commands with stdin and stdout.

parent\_handler() and child\_handler() - These function are used process signals from the parent and child processed, such as not termination process with ” ctrl+c”.

**Specifications**

In order to pass the lab, your shell, called lsh, needs to implement the following functionality correctly:

1. It must allow users to execute simple commands such as ls, date or who. It should be able to find

the location of commands (e.g., if a command is in /usr/bin, /usr/local/bin or any other folder),

so it needs to be aware of the path where it should search for commands. The standard convention

on UNIX systems is that the environment variable PATH contains the list of directories that should

be searched.

**Implementation** – To execute simple commands we are starting with a fork() system call in our RunCommands() function, then passing the commands to Commands\_Execution(), in which only the last else block will be executed as the preliminary if condition will not be satisfied as p->next is NULL. Then command is passed for to execvp() for execution.

2. It must be able to execute commands in the background so that many programs can execute at

the same time. For example,

$ sleep 30 &

runs the command sleep 30 in the background.

**Implementation** – To execute background commands we are using the “”background” variable of Command structure. In the parent block of RunCommands(), we are checking if it is not a background process, then we are waiting for the child process to execute the command. If it is a background process we are using a signal handler. In our code parent process is not waiting for child to complete its execution, if it is running in the background. So once sleep complete its execution in background process is still alive with sleep dfunct nothing but created zombie. To handle it we have used signal handler as follows “signal(SIGCHLD,SIG\_IGN);” . When the child is terminated, a corresponding SIGCHLD signal is delivered to the parent, if we call the ‘signal(SIGCHLD,SIG\_IGN)’, then the SIGCHLD signal is ignored by the system, and the child process entry is deleted from the process table. Thus, no zombie is created. However, in this case, the parent cannot know about the exit status of the child.

3. It must support the use of one or more pipes e.g.

$ ls | grep out | wc -w

outputs the number of files with filenames that contain the word “out” in the directory.

**Implementation –** The commands having one or more pipes is handled in Commands\_Execution() function. To handle pipes we need two file descriptors which needs to be passed as argument for pipe() system call. Pipe() function will help us to connect one command output to next command input. So first we are using the fork() system call to create a child, then in the child block we are copy the STOUT\_FILENO is the write file descriptor. Then we are using a recursive function call to traverse to the next command of the pipe. The recursive function call will, run the same set of commands until, the base condition which is p->next!=NULL is true(it reaches the end of the linklist). Then in the parent block we read the input STIN\_FILENO using the read file descriptor, then for each child the command gets executed in the parent block using the execvp() system call. Then for the last element of the linkedlist, the base condition is false and the subsequent else block will be executed, in which the last command in the linked list will get executed using the execvp() system call.

4. It must allow redirection of the standard input and output to file. For example,

$ wc -l < /etc/passwd > accounts

creates a new file “accounts” containing the number of accounts on the machine.

**Implementation** – In the RunCommands() function we are checking for using the rstdin and rstdout variables of the Command structure for checking if there are input and output redirections. If there are we are calling the function Command\_ExectionIO() for handling the same. Similar to the process for handling pipes we are using fork() system call for creating the child process. Inside the child process we are checking, if there is an input redirection or output redirection and handling the same accordingly. For input redirection we are opening the file in read-only mode using the appropriate arguments and assigning it to the file descriptor. Subsequently we are duplicating the file descriptor with STDIN\_FILENO. Similar process is followed for output redirection, where are opening the file in write or create mode(depending on whether the file was present or not). The subsequent processes are similar to input redirection. In the event there is a pipe, we are following we are using recursive function call to parse the linked list and execute the next command in a similar manner.

5. It must provide cd and exit as built-in functions.

**Implementation** – In RunCommands, we have declared the list of cd and exit as built-in commands. Then we are using a loop and if statement for checking whether a build in command was passed. If the built in command was passed we are calling a function Commands\_Execution\_Defined\_cmd, using a flag CmdChecker. In the function, Commands\_Execution\_Defined\_cmd we are using switch case to check the which built in command was passed and we are exit() and chdir() functions for executing them.

6. Pressing Ctrl-C should terminate the execution of a program running on your shell, but not the

execution of the shell itself.

**Implementation** – To ensure that Ctrl-C does not terminate the execution of the shell, we are using the signal() system call along with the standard signal SIGNIT.

7. Ctrl-C should not terminate any background jobs.

**Implementation** –

**4.1 Self-Test Examples**

In this section and below we have copied the test cases as mentioned in the Lab Manual and provided the explanation on the process we have used for executing the various commands, along with the answers to the given questions and the output that we have received after running these commands.

**4.1.1 Simple Commands**

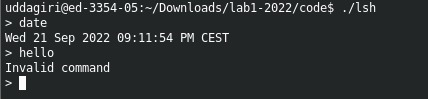
The brief of instruction we written is as follows To execute the commands we used execvp () system call function. But exec functions don’t execute any following instructions after the system call. But to handle this situation we have used fork() system call function, to create the child process and execute execvp() system call inside child process. Then process which call the fork () and child process will execute next instructions concurrently.

**Test**

$ date

$ hello

**Output**



The first command exists and the second one does not. Observe the system calls that are executed. If

any of the programs fail, what is printed? Where? What happens to any child processes that your shell

has created?

**Answer-** For running the first commands the fork(), wait() and execvp() are the system calls that are executed.

On running the second command the output will be “Invalid command”, as per the implementation in the code.

Child process created by the shell will get terminated after execvp() system call.

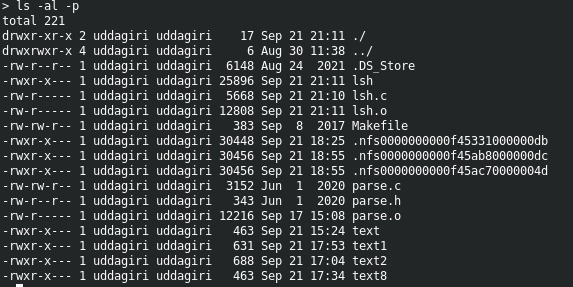
**4.1.2 Commands with parameters**

**Implementation** - The command with parameter and no pipe will work with same instructions written for simple commands. execvp() functions needs two arguments. The first argument is a character string that contains the name of a file to be executed. The second argument is a pointer to an array of character strings. So the command entered by user is stored in pointer to an array of character strings, pass the first element of pointer to an array of character strings as first argument and pointer to an array of character strings as second.

**Test**

$ ls -al –p

**Output**

****

**4.1.3 Redirection with in and out files**

**Implementation** – The commands with in and out files redirections is done using Commands\_ExecutionIO() function. We can create/read/ write in for, In and out files using open() function. We have used file descriptors to hold indexesof input files for reading, and the index of output files for writing. I have duplicated the file descriptors to set read end of file to standard input and write end of file descriptor to standard output. Then using execvp() commands are executed.

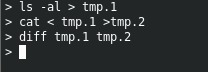
**Test**

$ ls -al > tmp.1

$ cat < tmp.1 > tmp.2

$ diff tmp.1 tmp.2

**Output**

****

Is the output of diff what you expected?

**Answer -** Yes, the diff command runs as expected and provides the differences between contents of the two files.

**4.1.4 Background Processes**

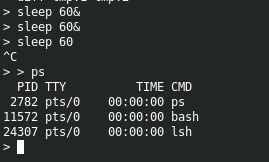
**Test**

$ sleep 60 &

$ sleep 60 &

$ sleep 60

**Output**

****

Try to look at the parent process that is waiting for the child process using top, as described in the

debugging section. Run the list of commands several times and use kill to see after which command it is

possible to generate a prompt.

Try pressing Ctrl-C in the lsh after the last sleep. Does the foreground process stop? Do the background

processes also stop? What is the expected behavior? Wait 60 seconds. Are there any zombie processes

left?

**Answer:-** First we running the sleep statements in the background, which is executing perfectly, however on running the sleep in foreground and pressing Ctrl-C in the lsh after the last sleep, the foreground process stops. The background processed also stops. However this is not the expected behaviour, as the background processes should continue to run.

There are not zombies left as we have handled the creation of zombies, by using signal handler as follows “signal(SIGCHLD,SIG\_IGN);” . When the child is terminated, a corresponding SIGCHLD signal is delivered to the parent, if we call the ‘signal(SIGCHLD,SIG\_IGN)’, then the SIGCHLD signal is ignored by the system, and the child process entry is deleted from the process table. Thus, no zombie is created.

**4.1.5 Process Communication (Pipes)**

**Test**

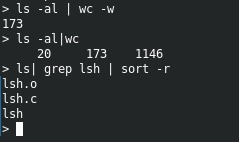
Verify that your shell supports one or more pipes.

$ ls -al | wc -w

$ ls -al | wc

$ ls | grep lsh | sort –r

**Output**

****

**Question**: Does the prompt appear after the output of the above command?

**Answer:** Yes, each of the three commands that were run over pipes produced the desired results, and a prompt asking for the next command to be enter after each successful command execution.

**Test**

$ ls | wc &

**Output**

****

**Question**: After running the above, when does the prompt reappear?

**Answer:** Above command gives the word counts of list (ls), in the order of number of newlines, words and characters. However, the instruction bears the sign "&" at the end. As a result, the command ran in the background and produced its results in the following prompts.

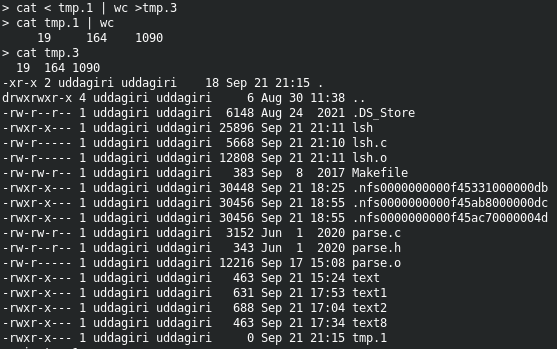
**Test**

$ cat < tmp.1 | wc > tmp.3

$ cat tmp.1 | wc

$ cat tmp.3

**Output**

****

**Question**: Compare the output of the last two commands above. Are they the same? Why/why not?

**Answer**:

**Test**

$ abf | wc

$ ls | abf

$ grep apa | ls

**Output**

What are the outputs? When does the prompt appear? Use Ctrl-D, if necessary, to let the grep finish

and to let the shell process take over. Does the grep command terminate eventually (use top to check).

Why/why not?

**4.1.6 Built-in Commands**

**Implementation –** The build in commands are handled using Commands\_ExecutionIO() function. We used c programming inbuilt function chdir() to change directory and exit(0) to come out the shell.

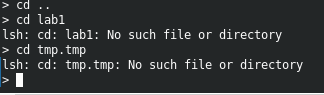
**Test**

$ cd ..

$ cd lab1

$ cd tmp.tmp

**Output**

****

**Question**: Was there an error generated when executing the commands above?

**Answer**: There are errors generated while executing last two commands but the error occurred was expected, because there are no such lab1 and tm.tmp directories present in the folder.

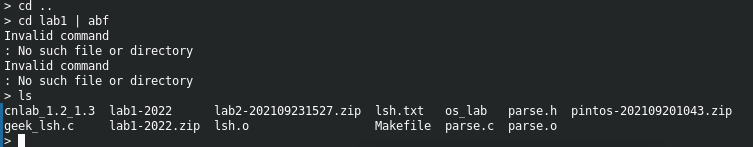
**Test**

$ cd ..

$ cd lab1 | abf

$ ls

**Output**

****

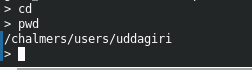
**Question**: Did the command ls work?

**Answer**: Yes the command ls works as excepted and provided appropriate output.

**Test**

$ cd

**Output**

****

**Question**: Was there an error? Use pwd to see which the current working directory is.

**Answer:** There is no error occurred. Just using “cd” command takes me to Home directory. This was observed using “pwd” command. This was expected behavior of shell.

**Test**

$ grep exit < tmp.1

**Output**

****

**Question**: Did the shell quit, or did it consider exit as a text string to find in a file?

**Answer**: The shell was not quit. It consider exit as text to find in a file and prompted next line to enter the command as there are no matching’s in tmp.1 file.

**Test**

$ ..exit

**Output**

****

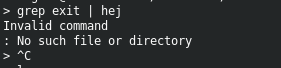
**Question**: And here? (Use spaces instead of dots)

**Answer**: Giving spaces and exit worked same as exit command and taken me out of the shell.

**Test**

$ grep exit | hej

**Output**

****

**Question**: Was there an error here? Does the prompt appear?

**Answer**: As command after pipe is not valid it has given output as invalid command. In our shell it prompts next line but it didn’t allow me to enter command. Once after clicking ctrl+c then it entered in to next line and allows me to enter command(which means grep is still in execution and shell is waiting for output of grep to appear, When are clicked ctrl+c it kills the foreground process and entered new line). But expected behavior is different, It should prompt invalid command and it shouldn’t enter to next line till I press ctrl+c to stop the current running process.

**Test**

$ grep cd | wc

**Output**



**Question:** Did an output appear? Does it appear after pressing Ctrl-D?

**Answer**: After executing command the shell got stuck and didn’t give any output till pressing ctrl+D.

**Test**

$ exit

**Output**



**Question**:

Are there any zombies after exiting lsh?

**Answer**: No zombies are present after exiting the shell. Which was observed using top command.