UNIVERSITY OF
GOTHENBURG

CHALMERS

Lab 1: Developing a simple shell

Noah Márquez
Madhav Goel

September 22, 2022

Contents

1 Description of our solution

Our shell starts out by typing the **prompt**, a ">" character in this case, which tells the user that the shell is waiting to accept a command. If the user now types *date* for example, the shell creates a child process and runs the *date* program as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next input line.

Our shell is able to find the location of commands (e.g., if a command is in /usr/bin, /usr/local/bin or any other folder) by using the **execvp()** command.

execvp() takes in the name of the UNIX command to run as the first argument, given by the first element of the array of *chars pgmlist* (from the struct *Pgm*). Here, we refer to a "command" as any binary executable file that is part of the **PATH** environment variable.

The second argument (*argv*) represents the list of arguments to the command. This is an array of *char** strings. It contains the complete command, along with it's arguments (that's why we pass the whole array *pgmlist*).

The **execvp()** gives control of the current process to the command, so anything that comes after **execvp()** will NOT execute, since our program is taken over completely. That's why we use **fork()**, to first spawn a new process and then use **execvp()** on that new process. This is what we call the "fork-exec" model.

Regarding the background processes, if a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. We did this by associating the background processes to a different process group ID and also by telling the parent (the shell) to only wait for its child processes if they are not background processes.

The output of one program can be used as the input for another program by connecting them with a pipe. We have to have in mind that the last command should output to the original process' file descriptor 1 and the first should read from original process file descriptor 0. We just spawn the processes in order, carrying along the input side of the previous *pipe* call.

In order to implement the pipes we developed a function to recursively execute the corresponding pipes in which the commands take their inputs from last-to-first into the pipe.

We will not enter into much detail in this explanation because we were asked to briefly summarize our implementation and since we commented our code explaining everything, we will keep it simple here. To execute the pipes, we do it for 2 commands at a time and then follow this steps (simplified):

1. We declare an integer array of size 2 for storing the file descriptors (file descriptor 0 is for reading and 1 is for writing).
2. We then open a pipe using the **pipe()** function with the previous array as an argument.
3. We fork a child process.
4. In the parent process we close the unused end of the pipe (read end), write to the pipe and then close the write end of the pipe.
5. In the child process we close the write end of the pipe, read from the pipe and then close the read end of the pipe.

We have to mention that if we're with the first two commands of the pipe before forking we call to the recursive function to execute the rest of the pipe (explained and commented in the code).

Our shell also allows redirection of the standard input and output to file. With the skeleton code that was given, our struct *Command* has two attributes of type char which tells us if we have standard input, output or both. First we get the file descriptor of the input/output file, then we close standard input or output with **close()** and after that we use **dup2()** to adjust the file descriptor of the file so it corresponds to the standard input/output. Finally, we close the file descriptor of the file.

We provide **cd** and **exit** as built-in functions, because in the main *while* loop before we start executing anything we check if the command given is either **cd** or **exit**. For doing such distinction we just use the **strcmp()** function to compare the command given with an string, in this case *exit* or *cd*.

If we receive **cd** we check a couple of things first. If the **cd** command is given with some pipes, we simply skip the **cd** command because it neither receive an input nor produces an output, so we skip to the next command of the pipe. We also check if **cd** is given with no arguments or with an '~', meaning that we have to go back to the home environment. For doing that we used the **chdir()** function and inside of it **getenv("HOME")** which reads the environment variable **HOME**. Lastly, we try to change the directory using the function **chdir**, if no error is returned, we are done and in the new directory specified. Otherwise, we print an error message saying: "No such file or directory".

If we receive an **exit**, we simply use **exit(0)** (exit success is indicated by exit(0) statement which means successful termination of the program) to quit the shell.

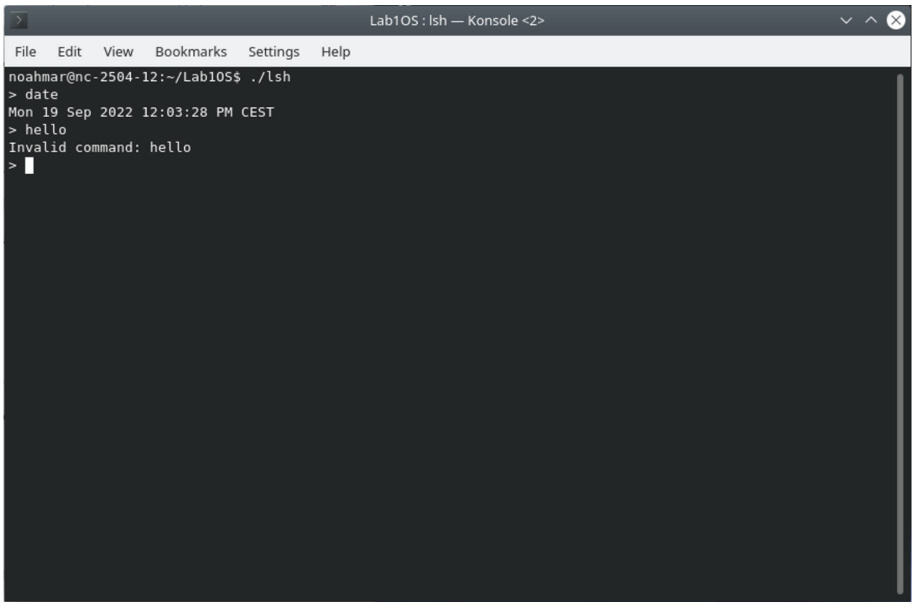
Pressing *Ctrl-C* will terminate the execution of any program running on the foreground in our shell, but not the execution of the shell itself. For doing that we just had to use the **signal()** function to tell the parent code (the shell) to just print again the prompt if it detects a *SIGINT* (*Ctrl-C)*. But any other process that we created through a **fork()** to execute a command will stop its execution if the *SIGINT* signal arrives.

To prevent *Ctrl-C* from terminating any background jobs we associate them to another process groupd ID using **setpgid()** so we treat them different than if we were not executing a command in the background.

2 SELF-TEST EXAMPLES

In the present section we provide examples that will help verify that our solution correctly implements the required specifications. We will provide the output of each command that we have been asked to run.

1 Simple Commands


Lab1OS : lsh — Konsole <2>
File Edit View Bookmarks Settings Help

```
noahmar@nc-2504-12:~/Lab1OS$ ./lsh
> date
Mon 19 Sep 2022 12:03:28 PM CEST
> hello
Invalid command: hello
> 
```

Figure 2.1: Simple Commands' output

Question: 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:
If any of the commands given by the user is not valid, we print the following message: "Invalid command: ...", as shown in Figure 2.1.

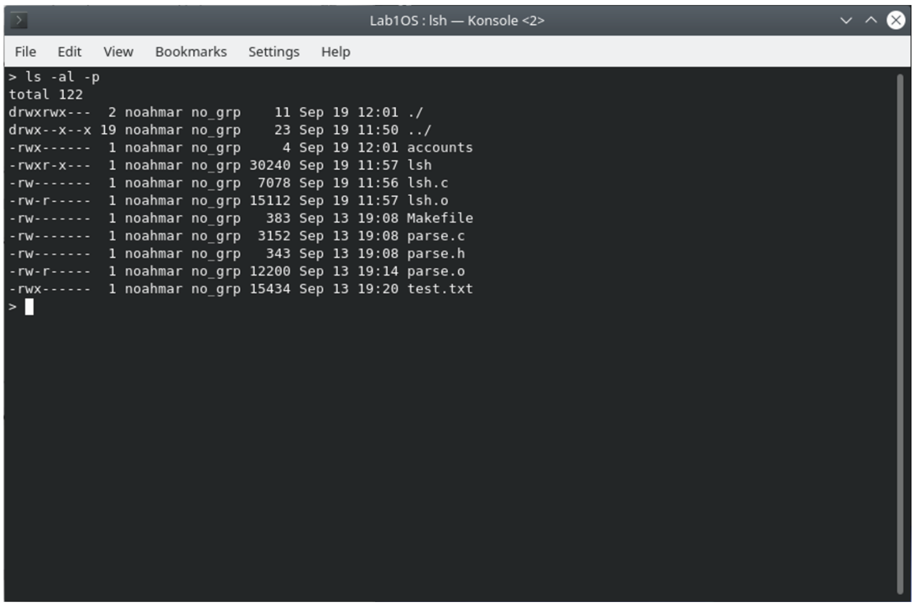
As of the part of observing the system calls that are executed, we gave much more detail in section 1 about how our shell works and which system calls take place.

Programs can also fail when forking (we check if fork failed as soon as we have called it and we check for the value returned), when setting a file as standard input/output (we have separated functions for both input/output in which we check for the respective errors) or even because the user did not enter a valid file or directory on the commands.

When we want to run a program, the shell creates a child process and runs the program as the child. While the child process is running, the shell waits for it to terminate. When the child finished, the shell types the prompt again and tries to read the next input line.

As soon as a program fails, if there is any child process that has been created that relies on the program whose execution was stopped because of an error, it handles the error and finishes its execution.

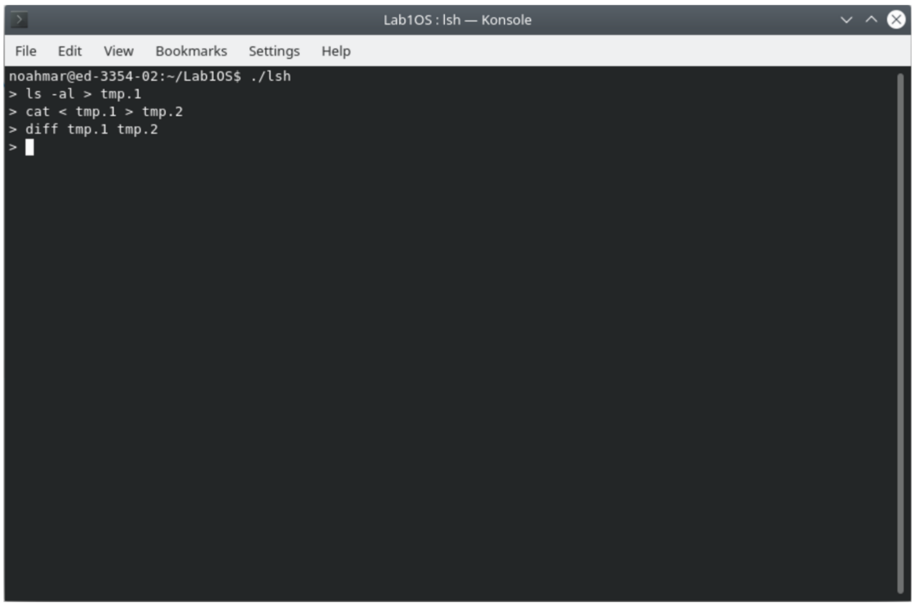
2 Commands with parameters


Lab1OS : lsh — Konsole <2>
File Edit View Bookmarks Settings Help

```
> ls -al -p
total 122
drwxrwx---  2 noahmar no_grp    11 Sep 19 12:01 ./
drwx--x--x 19 noahmar no_grp    23 Sep 19 11:50 ../
-rwx------  1 noahmar no_grp     4 Sep 19 12:01 accounts
-rwxr-x---  1 noahmar no_grp 30240 Sep 19 11:57 lsh
-rw-------  1 noahmar no_grp  7078 Sep 19 11:56 lsh.c
-rw-r-----  1 noahmar no_grp 15112 Sep 19 11:57 lsh.o
-rw-------  1 noahmar no_grp   383 Sep 13 19:08 Makefile
-rw-------  1 noahmar no_grp  3152 Sep 13 19:08 parse.c
-rw-------  1 noahmar no_grp   343 Sep 13 19:08 parse.h
-rw-r-----  1 noahmar no_grp 12200 Sep 13 19:14 parse.o
-rwx------  1 noahmar no_grp 15434 Sep 13 19:20 test.txt
> 
```

Figure 2.2: Commands with parameters' output

3 Redirection with in and out files



```
noahmar@ed-3354-02:~/Lab1OS$ ./lsh
> ls -al > tmp.1
> cat < tmp.1 > tmp.2
> diff tmp.1 tmp.2
> 
```

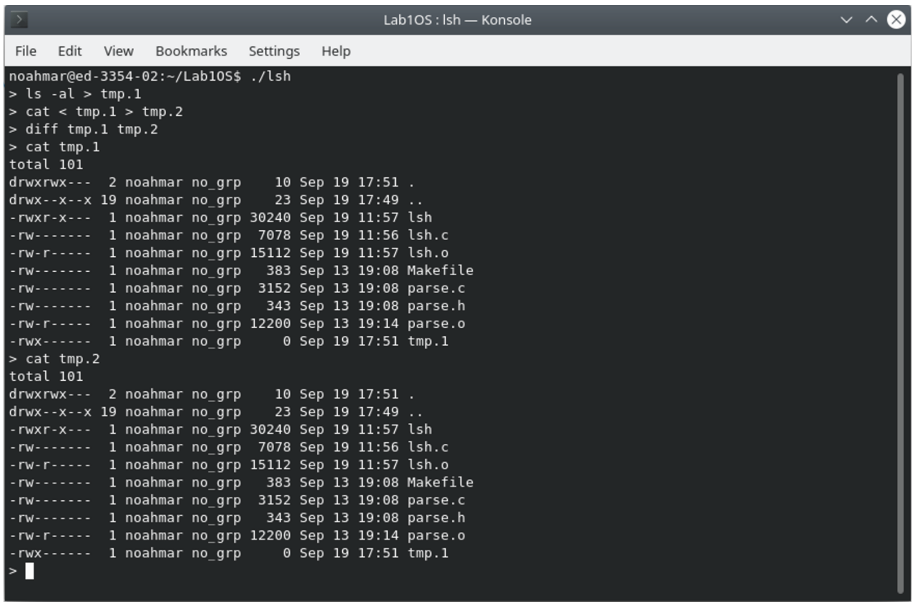
Figure 2.3: Redirection with in and out files output

Question: Is the output of *diff* what you expected?

$\boxed{\textbf{Answer:}}$
It is what we expected since both the files are the same. This is because the first command is used to list files or directories in a table format with extra information including hidden files or directories and we redirect its output to a file named 'tmp.1'. After that we read the content of

'tmp.1' with the **cat** command and we redirect the output to a file named 'tmp.2'. It is another way of copying the content of a file to another, that's why **diff** does not return anything since both files are the same.

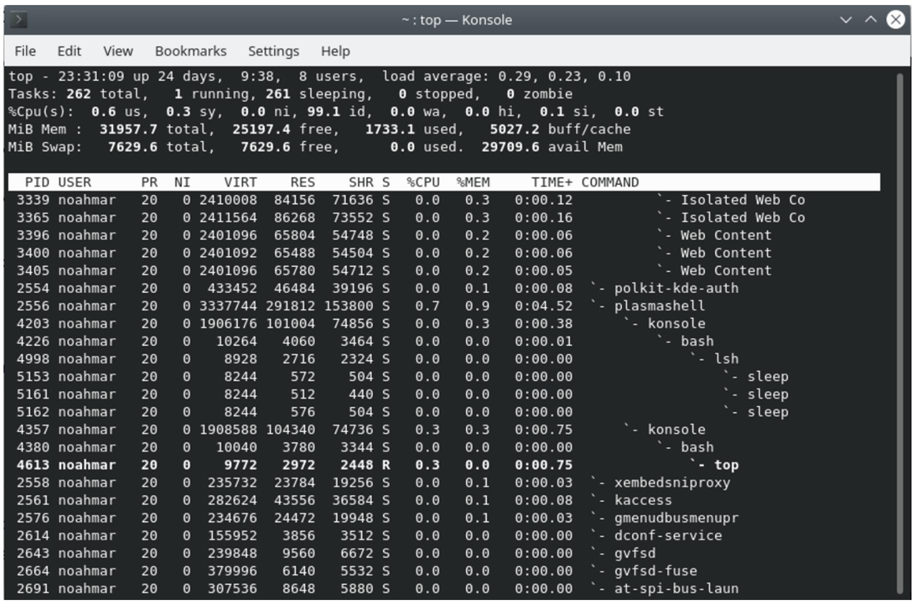
We attach an image showing both the contents of 'tmp.1' and 'tmp.2' using the **cat** command, in which we can see that both files have the same content.



```
noahmar@ed-3354-02:~/Lab1OS$ ./lsh
> ls -al > tmp.1
> cat < tmp.1 > tmp.2
> diff tmp.1 tmp.2
> cat tmp.1
total 101
drwxrwx---  2 noahmar no_grp    10 Sep 19 17:51 .
drwx--x--x 19 noahmar no_grp    23 Sep 19 17:49 ..
-rwxr-x---  1 noahmar no_grp 30240 Sep 19 11:57 lsh
-rw-------  1 noahmar no_grp  7078 Sep 19 11:56 lsh.c
-rw-r-----  1 noahmar no_grp 15112 Sep 19 11:57 lsh.o
-rw-------  1 noahmar no_grp   383 Sep 13 19:08 Makefile
-rw-------  1 noahmar no_grp  3152 Sep 13 19:08 parse.c
-rw-------  1 noahmar no_grp   343 Sep 13 19:08 parse.h
-rw-r-----  1 noahmar no_grp 12200 Sep 13 19:14 parse.o
-rwx------  1 noahmar no_grp     0 Sep 19 17:51 tmp.1
> cat tmp.2
total 101
drwxrwx---  2 noahmar no_grp    10 Sep 19 17:51 .
drwx--x--x 19 noahmar no_grp    23 Sep 19 17:49 ..
-rwxr-x---  1 noahmar no_grp 30240 Sep 19 11:57 lsh
-rw-------  1 noahmar no_grp  7078 Sep 19 11:56 lsh.c
-rw-r-----  1 noahmar no_grp 15112 Sep 19 11:57 lsh.o
-rw-------  1 noahmar no_grp   383 Sep 13 19:08 Makefile
-rw-------  1 noahmar no_grp  3152 Sep 13 19:08 parse.c
-rw-------  1 noahmar no_grp   343 Sep 13 19:08 parse.h
-rw-r-----  1 noahmar no_grp 12200 Sep 13 19:14 parse.o
-rwx------  1 noahmar no_grp     0 Sep 19 17:51 tmp.1
> 
```

Figure 2.4: Proof of the answer

4 Background Processes



```
top - 23:31:09 up 24 days,  9:38,  8 users,  load average: 0.29, 0.23, 0.10
Tasks: 262 total,   1 running, 261 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.6 us,  0.3 sy,  0.0 ni, 99.1 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
MiB Mem :  31957.7 total,  25197.4 free,   1733.1 used,   5027.2 buff/cache
MiB Swap:   7629.6 total,   7629.6 free,      0.0 used.  29709.6 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 3339 noahmar   20   0 2410008  84156  71636 S   0.0   0.3   0:00.12         `- Isolated Web Co
 3365 noahmar   20   0 2411564  86268  73552 S   0.0   0.3   0:00.16         `- Isolated Web Co
 3396 noahmar   20   0 2401096  65804  54748 S   0.0   0.2   0:00.06         `- Web Content
 3400 noahmar   20   0 2401092  65488  54504 S   0.0   0.2   0:00.06         `- Web Content
 3405 noahmar   20   0 2401096  65780  54712 S   0.0   0.2   0:00.05         `- Web Content
 2554 noahmar   20   0  433452  46484  39196 S   0.0   0.1   0:00.08 `- polkit-kde-auth
 2556 noahmar   20   0 3337744 291812 153800 S   0.7   0.9   0:04.52 `- plasmashell
 4203 noahmar   20   0 1906176 101004  74856 S   0.0   0.3   0:00.38     `- konsole
 4226 noahmar   20   0   10264   4060   3464 S   0.0   0.0   0:00.01         `- bash
 4998 noahmar   20   0    8928   2716   2324 S   0.0   0.0   0:00.00             `- lsh
 5153 noahmar   20   0    8244    572    504 S   0.0   0.0   0:00.00                 `- sleep
 5161 noahmar   20   0    8244    512    440 S   0.0   0.0   0:00.00                 `- sleep
 5162 noahmar   20   0    8244    576    504 S   0.0   0.0   0:00.00                 `- sleep
 4357 noahmar   20   0 1908588 104340  74736 S   0.3   0.3   0:00.75     `- konsole
 4380 noahmar   20   0   10040   3780   3344 S   0.0   0.0   0:00.00         `- bash
 4613 noahmar   20   0    9772   2972   2448 R   0.3   0.0   0:00.75             `- top
 2558 noahmar   20   0  235732  23784  19256 S   0.0   0.1   0:00.03 `- xembedsniproxy
 2561 noahmar   20   0  282624  43556  36584 S   0.0   0.1   0:00.08 `- kaccess
 2576 noahmar   20   0  234676  24472  19948 S   0.0   0.1   0:00.03 `- gmenudbusmenupr
 2614 noahmar   20   0  155952   3856   3512 S   0.0   0.0   0:00.00 `- dconf-service
 2643 noahmar   20   0  239848   9560   6672 S   0.0   0.0   0:00.00 `- gvfsd
 2664 noahmar   20   0  379996   6140   5532 S   0.0   0.0   0:00.00 `- gvfsd-fuse
 2691 noahmar   20   0  307536   8648   5880 S   0.0   0.0   0:00.00 `- at-spi-bus-laun
```

Figure 2.5: Background Processes

Question: 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.

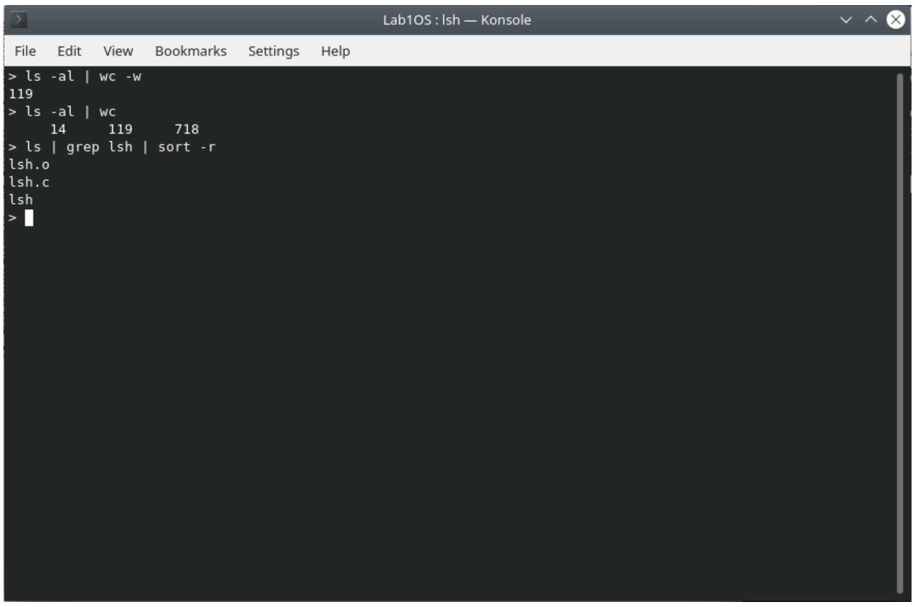
Try pressing Ctrl-C in the **lsh** after the last sleep. Does the foreground process stop? Do the background processes also stop? What is the expected behavior? Wait 60 seconds. Are there any zombie processes left?

Answer:
Running top on another terminal we can see that the three processes' parent is our shell, *lsh*. The first two commands run in the background for 60 seconds, and the third command rounds in the foreground also for 60 seconds. If we only run the first two, we get the prompt immediately, but if we run the last command, we will have to wait until it's finished or we will have to kill it (either by *Ctrl+C* or using **kill** on another terminal to kill the process). Meaning that we will have to kill the process that's on the foreground to get back the prompt.

Preessing *Ctrl-C* in the *lsh* after the last **sleep** stops the foreground process, but the background processes remain intact. The expected behavior is that the background processes will eventually finish without ending up as zombies.

After running the commands several times we can confirm that there are not any zombies left.

5 Process Communication (Pipes)



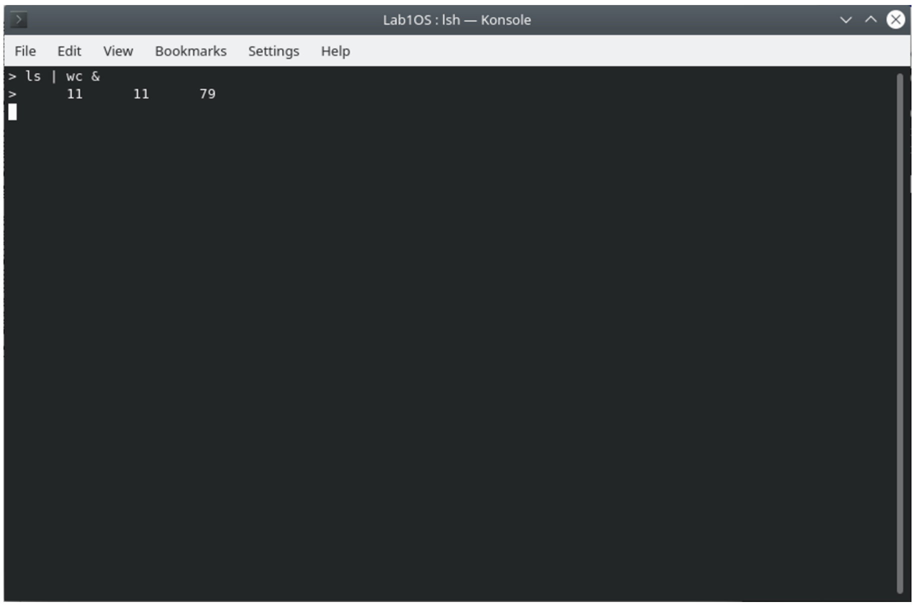
```
> ls -al | wc -w
119
> ls -al | wc
     14     119     718
> ls | grep lsh | sort -r
lsh.o
lsh.c
lsh
> 
```

Figure 2.6: Pipe's output (1)

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

Answer:
Yes, it appears. The last command consists of two pipes: the first one redirects the output of **ls** (command used to list files or directories) to the command **grep lsh** (**grep** command searches the given output for lines/files containing a match to the given strings or words, in this case 'lsh'), the second one redirects the results of **grep** to the command **sort -r** (which sorts the contents in reverse order).

After returning the output, the prompt appears without any error/problem.

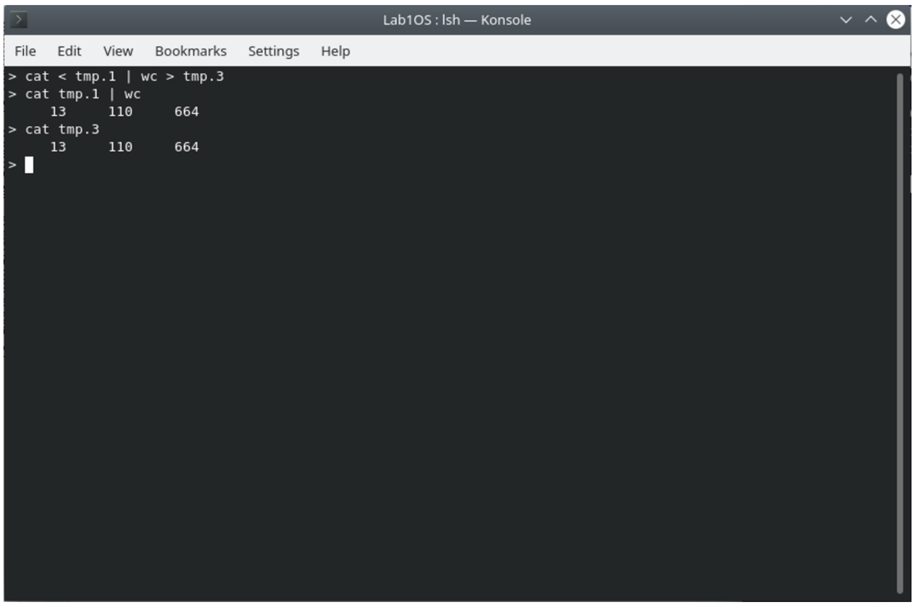


```
> ls | wc &
>      11      11      79
```

Figure 2.7: Pipe's output (2)

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

Answer:
As we can see in the image, the prompt appears before **wc** returned its output. That's because we asked the shell to run the command in the background (**&** flag).



```
> cat < tmp.1 | wc > tmp.3
> cat tmp.1 | wc
     13     110     664
> cat tmp.3
     13     110     664
>
```

Figure 2.8: Pipe's output (3)

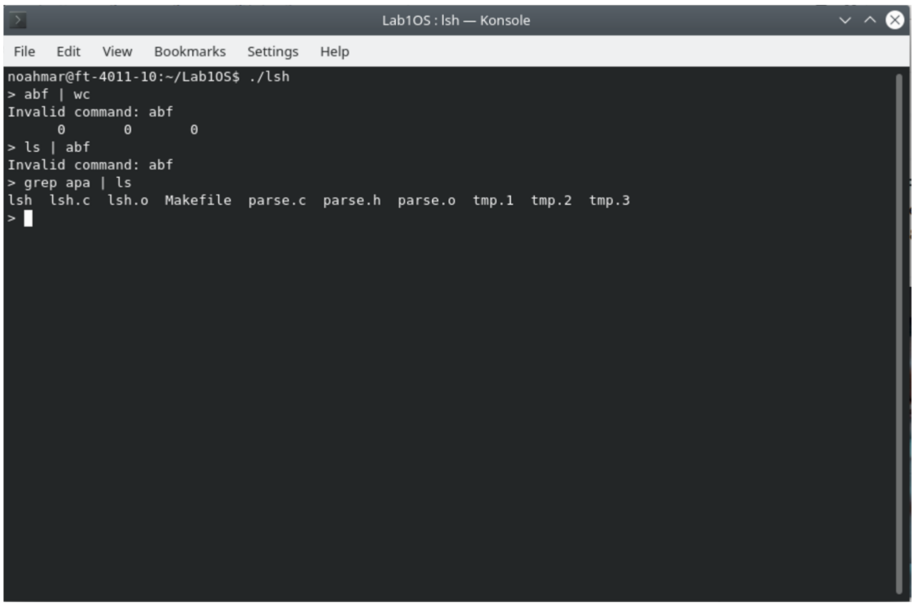
Question: Compare the output of the last two commands above. Are they the same? Why/why not?

Answer:
The output of the last two commands, as seen in the picture above, is the same.

With the first command we read through **cat** the content of 'tmp.1' and using a pipe we pass that content to **wc** (used to find out number of lines, word count, byte and characters count) and redirect it to a file named 'tmp.3'

With the second command we read the content of 'tmp.1' and through a pipe we use again the **wc** command. And with the last command we read the content of the 'tmp.3' file.

The output is the same because doing *cat < tmp.1* or *cat tmp.1* returns the same output, and since we are doing essentially the same in the first two lines just with the difference that in the first one we redirect the output to 'tmp.3' and in the second line we print it to screen, so both outputs are equivalent.



```
noahmar@ft-4011-10:~/Lab1OS$ ./lsh
> abf | wc
Invalid command: abf
      0       0       0
> ls | abf
Invalid command: abf
> grep apa | ls
lsh  lsh.c  lsh.o  Makefile  parse.c  parse.h  parse.o  tmp.1  tmp.2  tmp.3
> 
```

Figure 2.9: Pipe's output (4)

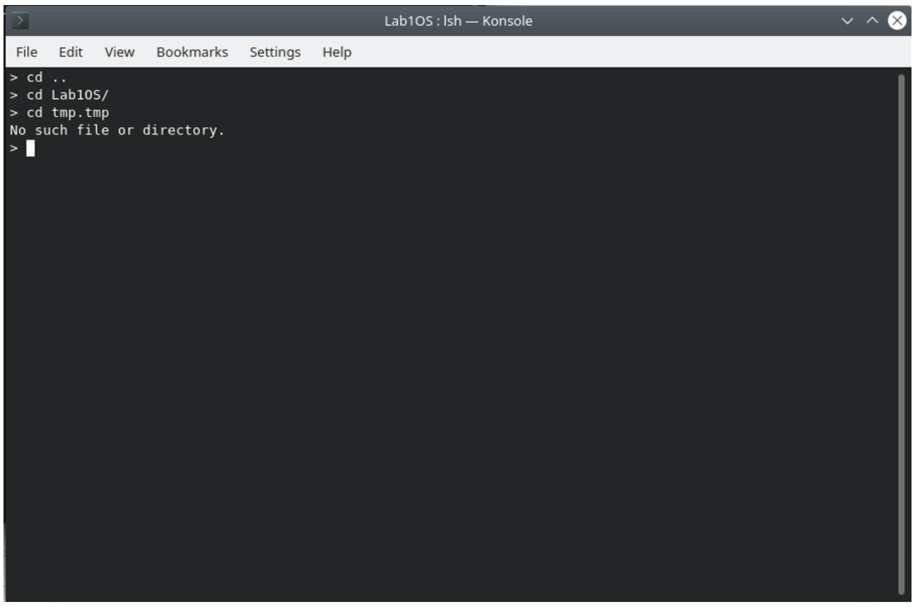
Question: What are the outputs? When does the prompt appear? Use *Ctrl-D*, if necessary, to let the grep finish and to let the shell process take over. Does the **grep** command terminate eventually (use **top** to check). Why/why not?

Answer:
The outputs are shown in the image above.

In the three commands the prompt appears after the output.

After the last command we had to use *Ctrl-D* to let the shell process take over. The **grep** command does not terminate and that's because *grep apa* is expecting its input from standard in. If you're not providing any input (via the keyboard, a pipe, etc), then it will block indefinitely. Before pressing *Ctrl-D* **ls** showed its output as we can see in the image.

6 Built-in Commands

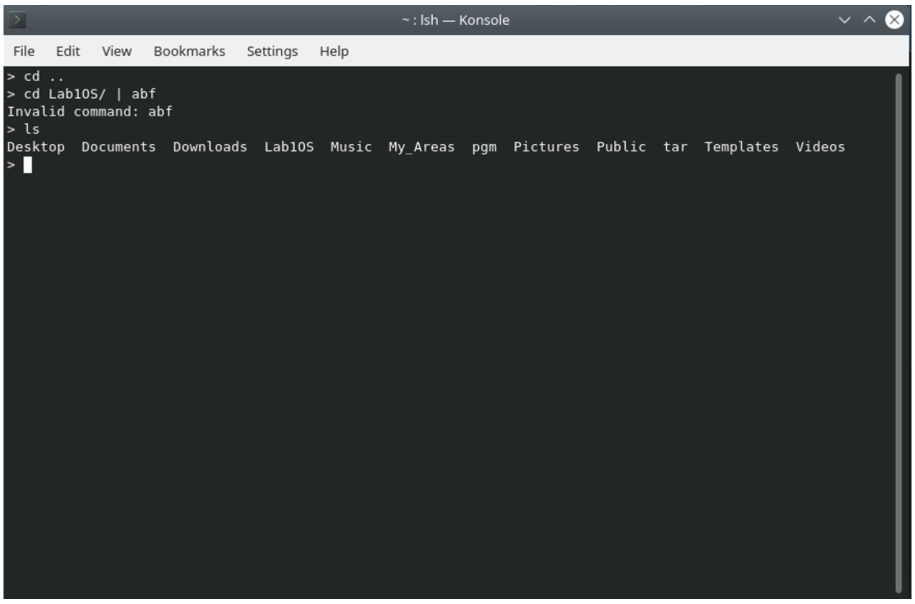


```
> cd ..
> cd Lab1OS/
> cd tmp.tmp
No such file or directory.
>
```

Figure 2.10: Built-in Commands' output (1)

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

Answer:
The first two commands worked without any problems and the third one returned a message saying 'No such file or directory' (shown in the figure), as expected because we don't have any file or directory with that name in the *lab1* folder.

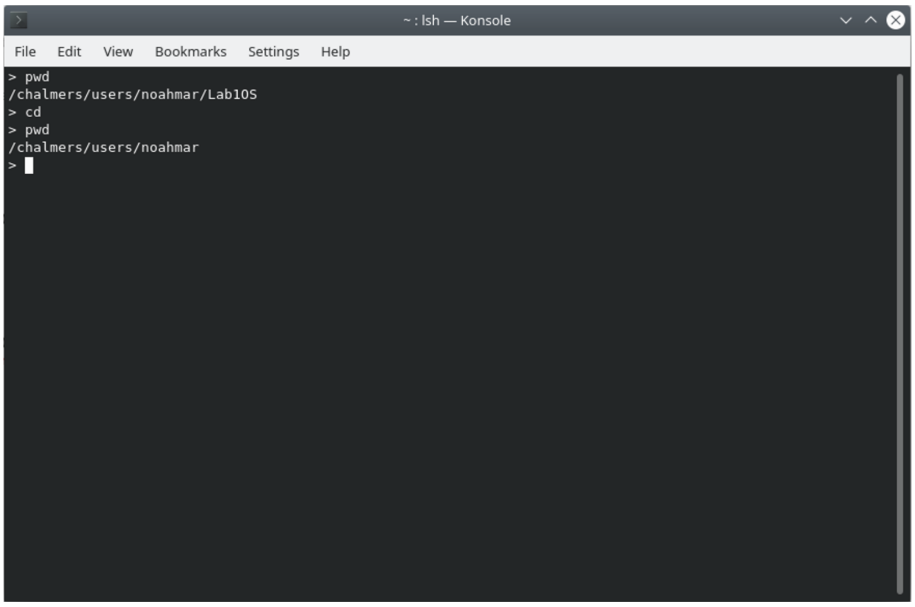


```
> cd ..
> cd Lab1OS/ | abf
Invalid command: abf
> ls
Desktop  Documents  Downloads  Lab1OS  Music  My_Areas  pgm  Pictures  Public  tar  Templates  Videos
>
```

Figure 2.11: Built-in Commands' output (2)

Question: Did the command ls work?

Answer:
Yes, it worked without any problem. With the first command we went back one directory, and with the second one we received an ’Invalid command: abf’ output. We have to understand that **cd** does not make sense with pipes, since it does not produce any output and it does not receive any input. With **ls** we listed directory contents of files and directories.

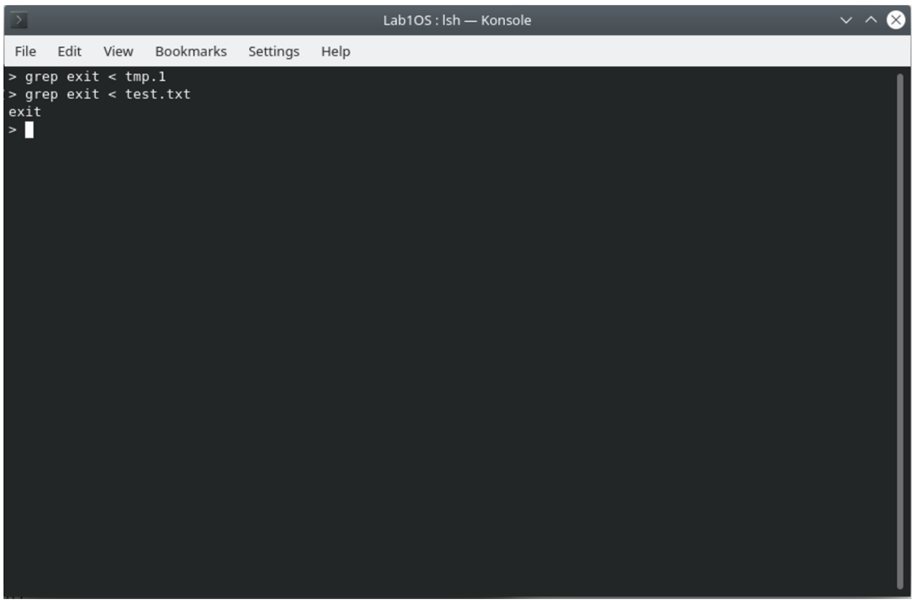


```
> pwd
/chalmers/users/noahmar/Lab1OS
> cd
> pwd
/chalmers/users/noahmar
> 
```

Figure 2.12: Built-in Commands’ output (3)

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

Answer:
There was no error and everything worked as expected as we can see in the image above. When we execute the **cd** command with no arguments the user is returned to the home directory. For doing so we used the function **char *getenv(const char *name)** that searches for the environment string pointed to by name and returns the associated value to the string; we used it with the "HOME" environment string.

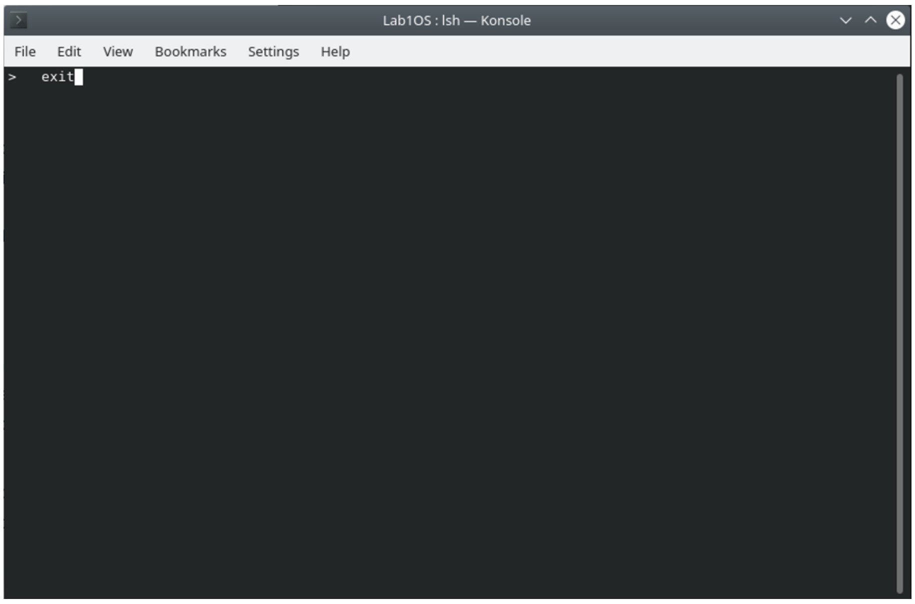


```
> grep exit < tmp.1
> grep exit < test.txt
exit
> 
```

Figure 2.13: Built-in Commands' output (4)

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

Answer:
As we can see in the picture above, the shell didn't quit because it considered *exit* as a string to find in a file. In this case we didn't get any output because the file didn't have that string, but just to show how it would work if it found the string *exit* in the text, we created a test file with the word, and as we can see the output is what we expected.



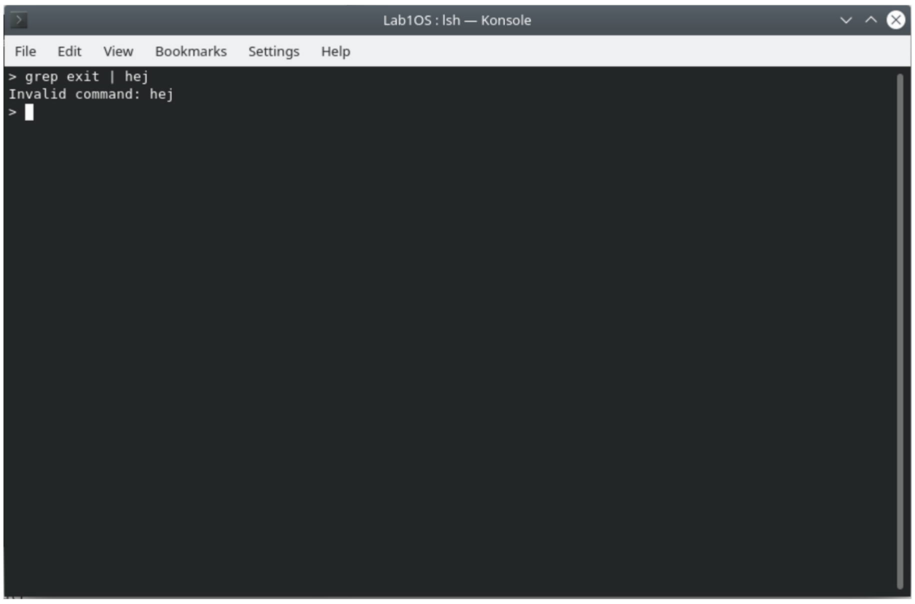
```
>   exit
```

Figure 2.14: Built-in Commands' output (5)

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

Answer:
Here the shell quits, as expected. That's because in the skeleton code we were given, we had

a function implemented called **stripwhite(char *)** which removes leading and trailing white space from the line given by the user.



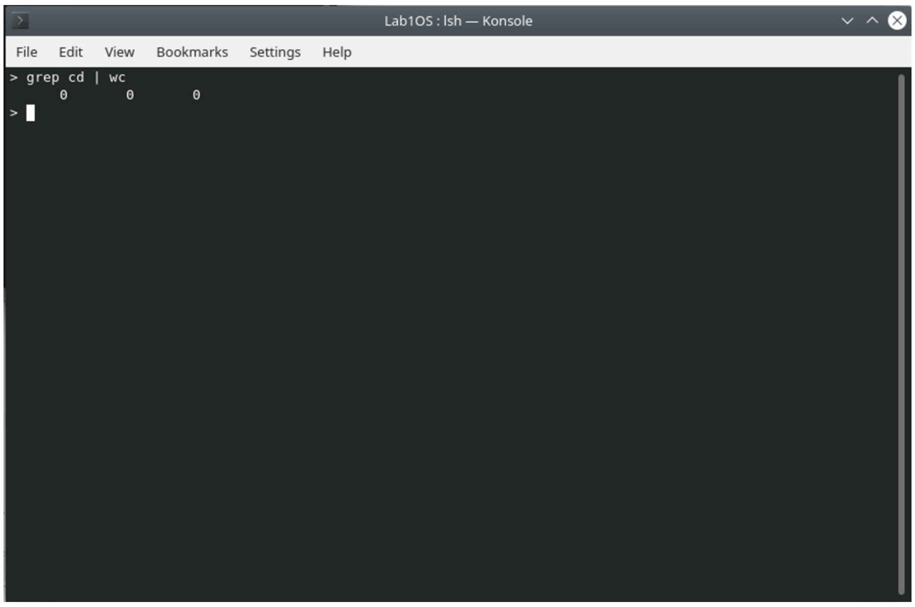
```
> grep exit | hej
Invalid command: hej
> 
```

Figure 2.15: Built-in Commands' output (6)

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

Answer:
The only error as we can see in the image is that the shell detects *hej* as an invalid command. After that we had to use *Ctrl-D* to let the shell process take over and then the prompt reappeared.

The **grep** command does not terminate until we press *Ctrl-D* and that's because *grep exit* is expecting its input from standard in. If you're not providing any input (via the keyboard, a pipe, etc), then it will block indefinitely.



```
> grep cd | wc
      0       0       0
> 
```

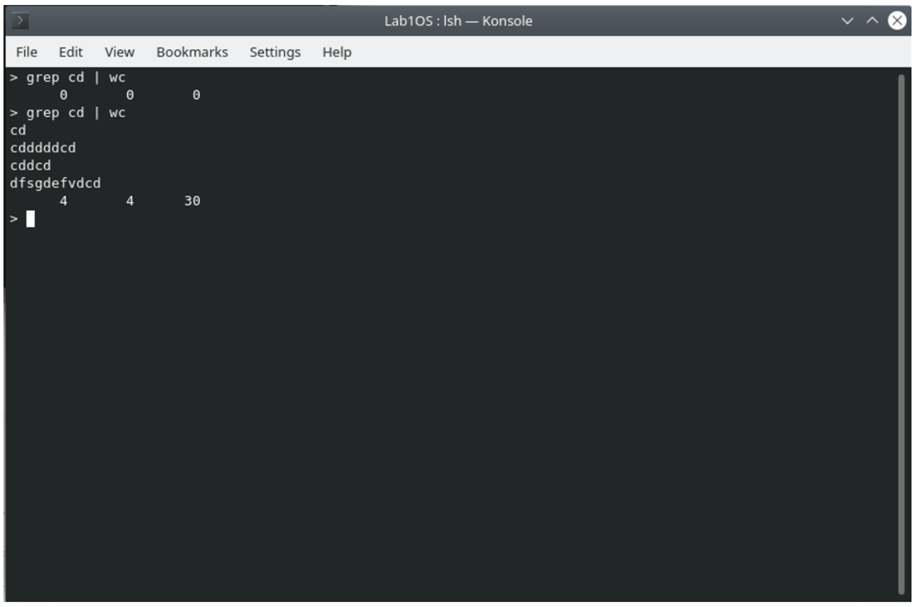
Figure 2.16: Built-in Commands' output (7)

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

Answer:

No output appears because the **grep** command is expecting its input from standard in. If you're not providing any input (via the keyboard, a pipe, etc), then it will block indefinitely. Nevertheless, if we press Ctrl-D the output appears as we can see in the image above.

We also provide the following image in which we can see that if we write some random words via the keyboard and then press Ctrl-D we also get a correct and expected output from **wc**.



```
> grep cd | wc
      0       0       0
> grep cd | wc
cd
cdddddcd
cddcd
dfsgdefvdcd
      4       4      30
> 
```

Figure 2.17: Another example for the command

Question: Are there any zombies after exiting **lsh**?

$\boxed{\textbf{Answer:}}$
With one of our first implementations we found that some zombies where left behind because our code was creating multiple **lsh** shells. After solving that problem now as soon as we use **exit**, our shell does not leave any zombies.

We don't include an image for this part because it would not prove anything, it is better to actually try it.