Operating Systems lab 1: Developing a simple shell

Dennis Dubrefjord, Viktor Vestlund

September 26, 2021

1 Implementation of our solution

We have implemented a shell according to the specifications outlined in the lab instructions. The requirements on the shell were as follows:

- It must allow users to execute simple commands such as ls, date or who.
- It must be able to execute commands in the background so that many programs can execute at the same time.
- It must support the use of one or more pipes.
- It must allow redirection of the standard input and output to file.
- It must provide cd and exit as built-in functions.
- Pressing Ctrl-C should terminate the execution of a program running on your shell, but not the execution of the shell itself.
- Ctrl-C should not terminate any background jobs.

Below we will outline how we implemented the requirements, if any problems became apparent during the implementation and if so, how we solved them.

1.1 Simple commands

When a simple command is received, the process forks itself to create a child process. The child process is then instructed to execute the command using `execvp()`, and the parent process waits, using `wait(NULL)` until the child finishes (and removes the child's entry in the process table) before it returns to the main loop where it can accept new commands.

There were no big problems in this part.

1.2 Run in background

For this requirement we took the implementation of the previous requirement and added that if a command included the background character, the parent does not wait. It instead immediately returns to accepting new commands. If the background character is not included, the parent waits for the child like before. However, we had an issue where the parent would wait for all of its child processes before returning, instead of just the one in the foreground. We solved this by replacing the `wait(NULL)` instruction that we previously used with a `waitpid(pid, &status, 0)` instruction to make sure we only wait for the foreground process.

Finally, since the parent does not wait on the children that are run in the background, we needed to make sure the parent still removed them from the process table upon termination, lest we get zombie processes in the system. We solved this by adding a signal handler. If the parent receives the signal `SIGCHLD`, which it does when one of its child processes terminate, it runs `while(waitpid((pid_t)(-1), WHNOHANG) > 0){}`. The `WNOHANG` makes sure that the parent process does not actually wait for all of its children to terminate. Instead it simply checks whether any of its children has terminated and if so, removes them from the process table.

1.3 Pipes

To add support for pipes, we first had to figure out how many pipes were present in the command. We then create all the pipes using the `pipe()` instruction, and save the file descriptors of the pipe ends in an array. Finally we loop through the instructions (that are separated by pipes) in the command. For each iteration we fork the process and direct the output and input from and to each command to the appropriate pipe ends. The first and the last command are handled a bit differently that the others since they do not redirect their input and output, respectively. Finally the child process is instructed to execute the instruction that it has been assigned from the command thus sending its output either to `stdout` or to the appropriate pipe end. This is done using the `execvp()` instruction in combination with `dup2()` to assign the correct pipe end to the commands `stdin` and/or `stdout`.

1.4 Redirection to file

To satisfy this demand, the child process checks whether there are any redirects in the command, right after it is created in the fork. If so, it opens the files it has been instructed to read from and write to and sets them as input and output, respectively. It then sets the file descriptor to the file to `stdin` when reading the file or `stdout` when writing, it does this with the `dup2()` command. When this is done it is important to remember to close the file to stop the file from waiting for input or trying to write everything to a file, if the closing is not done then the shell will stop and wait endlessly (unless Ctrl-C is pressed). When the reading/writing is done and the file is closed we can execute the command as usual to get the input/output and simply use `stdin` or `stdout`.

1.5 Built-in functions

The exit instruction was solved by simply checking whether the received command was "exit". If so, the program exits using `exit(1)`.

The cd instruction was also solved by placing a check just after receiving the command, where we check whether the command equaled "cd". If so, we extract the path provided in the command and run `chdir(path)`. If this instruction returns something other than zero, it means that an error occurred. If so, we print that there was an error in the path.

The extraction of the path was a bit tricky to get right. We drew a lot of inspiration from the PrintPgm-function that was included from the beginning and finally managed to solve it. The extraction is showed below.

```
char **pl = cmd->pgm->pgmlist;
char* path = *(pl+1);
```

1.6 Ctrl-C does not terminate the shell, but it does terminate programs running in the shell

In order to make sure the shell does not terminate upon receiving a SIGINT signal, we needed to write a signal handler for the parent that overrides the default handler for SIGINT. This requirement was developed together with the next one, since they both handle SIGINT. The discussion will thus continue in the next section.

1.7 Ctrl-C does not terminate background jobs

The reason Ctrl-C terminates background jobs is that all processes in the same group as the process that received the Ctrl-C will receive the SIGINT. We can handle this in two ways:

- Make sure the children have signal handlers that instructs them to ignore the SIGINT when they receive it.
- Make sure the children are not part of the same process group as their parent (which they are by default after forking).

If one goes for the first strategy, one must make sure that the signal handler survives when the child runs `execvp()` to execute the command. When reading documentation, it seems that signals that are ignored before `execvp` will keep on being ignored afterwards. Thus this might be a viable strategy.

However, we decided to go with the latter option. To make sure the children were not part of the process group, we added the instruction `setsid()` before the execvp in the child. This changes the child's gid and solves the issue. But now we need to add a way for the parent process to kill the child-process in the foreground, since that process will no longer get interrupted by the `SIGINT` to the parent. We added a global variable `child_pid` that is set to -1 by default. Upon forking, if the parent should wait for the child (meaning it is run in the foreground), the parent sets the variable `child_pid` to be the process ID of the child. When the child has finished executing, the parent will stop waiting and once again set the `child_pid` to equal -1.

The result is that we have a variable that has the pid of the child that is executing in the foreground, if one exists. In the signal handler we check that `child_pid != -1` and if so sends a kill-signal to the process with `pid == child_pid` and once again set `child_pid` to equal -1.

2 Self test cases

In this section we will show the result of running a set of commands in our shell. If warranted, we will justify why it behaves the way it does.

2.1 Simple commands

Running `date` and `hello` returns what is expected: The child processes that are created are terminated

```
vikves@ed-3582-15:~/Documents/OperativeSystems$ ./lsh
> date
Sun 26 Sep 2021 08:17:50 PM CEST
> hello
Could not execute the command
>
```

when they are done with their execution.

2.2 Commands with parameters

Running `ls -al -p` returns what is expected:

```
> ls -al -p
total 142
drwxr-x--- 3 vikves vikves    12 Sep 26 20:16 ./
drwxr-x--- 3 vikves vikves     3 Sep 26 20:16 ../
drwxr-x--- 8 vikves vikves    13 Sep 26 20:16 .git/
-rw-r----- 1 vikves vikves   441 Sep 26 20:16 .gitignore
-rwxr-x--- 1 vikves vikves 38208 Sep 26 20:16 lsh
-rw-r----- 1 vikves vikves  9077 Sep 26 20:16 lsh.c
-rw-r----- 1 vikves vikves 29752 Sep 26 20:16 lsh.o
-rw-r----- 1 vikves vikves   383 Sep 26 20:16 Makefile
-rw-r----- 1 vikves vikves  3152 Sep 26 20:16 parse.c
-rw-r----- 1 vikves vikves   343 Sep 26 20:16 parse.h
-rw-r----- 1 vikves vikves 12408 Sep 26 20:16 parse.o
-rw-r----- 1 vikves vikves  1313 Sep 26 20:16 prepare-submission.sh
>
```

2.3 Redirection with in and out files

Running the commands does not return anything. This is expected since we copy the contents of tmp.1 into tmp.2 just before we compare them.

```
> ls -al > tmp.1
> cat < tmp.1 > tmp.2
> diff tmp.1 tmp.2
>
```

2.4 Background process

Running the sequence of sleep instructions, followed by a Ctrl-C, behaves as expected. The fore-

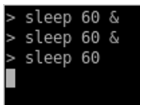

Figure 1: Commands that were executed.

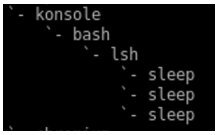

Figure 2: Ttop forest view before hitting Ctrl-C.

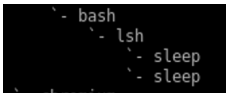

Figure 3: Top forest view after hitting Ctrl-C.

Figure 4: Top forest view after 60 seconds, when the backgrounded sleep commands have finished executing.

ground process terminates while the other two keep executing. After 60 seconds, they also terminate. No zombies remain.

2.5 Pipes

The tests in the pipe section behave as expected.

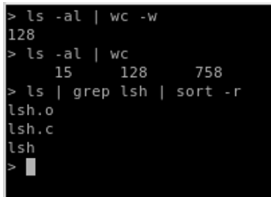

Figure 5: The commands run in this test, along with their output. The prompt does appear after the output of the commands.

Figure 6: The commands run in this test, along with their output. The output is printed on the line where the prompt is, so we have pressed enter once afterwards to get a new prompt on a new line.

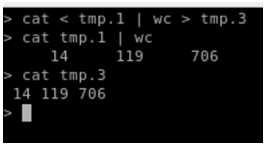

Figure 7: The commands run in this test, along with their output. The output of the two last commands are the same, which makes sense because they both equal tmp.1 — wc.

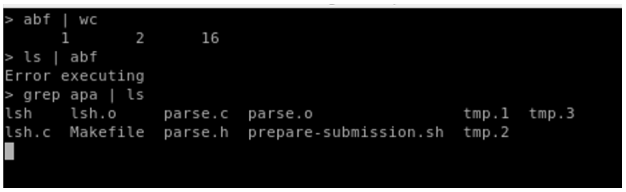

Figure 8: The commands run in this test, along with their output. To get the prompt back after the last command, we need to press Ctrl-D (shown in Figure 9) since the grep-command is waiting for input. In the first command, the error message printed by abf gets sent to wc. In the second command, we receive an error message from abf again.

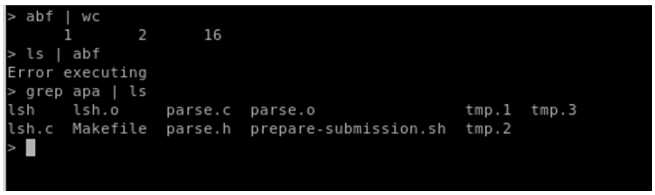

Figure 9: The same test as shown in Figure 8. Here, Ctrl-D has been pressed and the prompt has been returned.

2.6 Built in commands

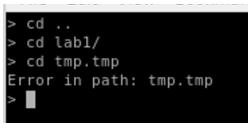

Figure 10: The commands run in this test, along with their output. The cd-command works as intended. Upon executing it, we change the current directory to the specified path, if it exists. In the last command it does not exist. Thus, an error is generated.

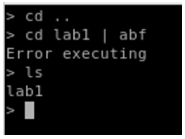

Figure 11: The commands run in this test, along with their output. The error stops the cd from executing. The ls works fine as the prompt has been returned.

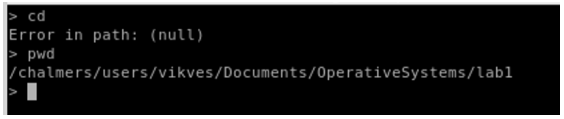

Figure 12: The commands run in this test, along with their output. There was an error because the provided path (null) does not exist.

Figure 13: The commands run in this test, along with their output. The shell does not quit, since the "exit" is not interpreted as a command, but instead as a text string that grep searches for in the file.

Figure 14: The commands run in this test, along with their output. During execution of exit with two spaces in front, the shell actually exits. This is because we remove leading or trailing white spaces when parsing the commands, thus it works as expected.

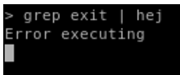

Figure 15: The commands run in this test, along with their output. Yes there was an error, but the error is the word *hej* which is not a command and not the other part.

Figure 16: The commands run in this test, along with their output. When running this command the shell just keeps running and no output appears. Due to *cd* waiting for an argument that does not appear and thus it never executes.

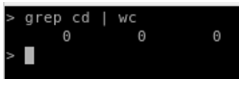

Figure 17: The commands run in this test, along with their output. After pressing Ctrl-D the output appears, just executing and exiting the *wc* command without any input thus returning what it returns.

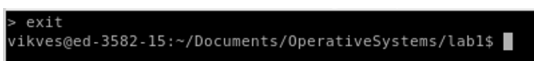

Figure 18: The commands run in this test, along with their output. Running the exit command exits the entire shell as expected without leaving any zombies behind.