Assignment 1 - Report

Philipp Küppers Sofie Vos

September 2022

1 Implementation

We choose to traverse the expression data structure imperatively because we found it easier to make the case distinctions in this case.
At the beginning we first check if the user entered `exit` or `cd` at one point during the expression. Initially we wanted to place the `exit` system call inside the `execute_command` function, however exit only terminates the current process which is running. This means that if we would call `exit` inside `execute_command`, we only terminate the child process we created beforehand to avoid `execvp` terminating our shell. Unfortunately we didn't find a smart way to still place `exit` inside `execute_command` and did not want to keep calling exit until all processes are terminated. Therefore we just check inside `execute_expression`, before calling `execvp`, whether the expression includes exit and then terminate the shell. A similar reason is why we decided to first execute the internal command `cd` before executing external commands. With our implementation we also did not want to allow strange commands like `cmd | cmd | cd Documents | cmd`.

Next we check if there was no command because in these two cases we do not need to `fork`, since we do not call `execvp` which would replace our shell program.

At first we had an extra if statement before our loop to check whether we just have one command. However we decided to not make an extra case for that in order to save some redundant code, because in both cases you can have input and output from and to a file. There we now say, that if the expression has one or more commands we go through the vector of commands with a for-loop use the following strategy: For n commands we create $n-1$ pipes, since we can not re-use the pipes for other commands.

The reason is that we need to run chained commands simultaneously. Hence we use a pipe just for a connection between two commands, which equals $n-1$ connections for n commands. This means that with less pipes we would either need a buffer or run into the infinite buffer problem where two processes read from the same pipe.

Then we follow the specifications from the assignment to decide for each case what the input and output of the command are and when to close the inputs. This was the most efficient case distinction we could come up with: For our commands we always have an input or output. However there are different situations and settings for when to set what output. Therefore our strategy is to define variables for the input and output and overwrite them in case something extra happens. This reduced the number of lines of code significantly. Our regular case is that a command has a command beforehand and one afterwards and needs to get the output from the previous one as input and set his output as the input for the next command. As mentioned, there are a few special cases. For the first command executed, the user can decide to give input from a file. For this we open the file with only read access and copy the content of the file, to input.

For the last command, the use can decide to redirect the output to a file. Here we set the flags in the open system call accordingly to give the behaviour specified in the assignment.

In case the last command of the expression ends with a &, the command is supposed to run in the background and the first command can not get direct input from a user. We implement this by closing `STDIN_FILENO` for the first command when it should run in the background and after executing the expression, we do not wait for all children to finish and instead directly return, so the user gets prompted with a shell again.

After setting the right input and output, we execute the current command. For this we pass the command to the function `execute_command`, which executes the command using `execvp`. Inside this function we also return the error message of the executed command or give an error message to the user in case the command is invalid.

In the assignment it was specified that in order to run chained commands such that the infinite buffer problem is avoided, we need to run the commands simultaneously. Therefore we keep executing commands and only at the end of the expression, we wait for all child processes to finish.

Here is a complete list of system calls we used with explanation on why we used them:

- `execvp()` - we use this system call to execute each of our external commands.
- `fork()` - we use it to create child processes. This is useful to create a child process whenever we want to execute a command using `execvp`, since it replaces the currently running process (which would be our shell program at first).
- `pipe()` - this system call makes it a lot easier to executed chained com-

mands, since we can create a pipe to pass the output from one command to the input of the other command.

- `waitpid()` - with this system call we want to make sure that the process we created terminates before continuing with the code. Especially useful when we have multiple chained commands and want to first fully execute the first command before continuing with the second command.
- `open()` - this allows us to open and create files to interact with them. It opens a file with specified parameters, like which permissions it has and whether it should create a new file if no file exists. Using the pointer to the file it opens, we can copy from and to this file.
- `close()` - we make use of this system call to close file descriptors to standard in / out and files.
- `dup2()` - we frequently use this system call to copy and redirect an existing file descriptor by letting it point to a new description.
- `chdir` - this system call only gets used in our implementation when changing the directory we are in.
- `exit` - this special system call is needed to exit the current process and execute the command `exit` of the user.

2 Tests

We execute all tests in the test-dir containing the files 1-4. Additionally we also ran the `shell.test.cpp` file, where our implementation passed all tests.

command	expected result	actual result
echo a b c d	echo a b c d	
pwd	<path>/test-dir	<path>/test-dir
ls	1 2 3 4	
date | tail -c 5	2022	
cat 1	content of 1	content of 1
cat 1 | tail -n 3		
ls > output-file | cat < inputfile	invalid command	
ls > output	create output file with current directories	
date 6	date: invalid date '6'	date: invalid date '6'
date &	<currentdate>	<currentdate>
exit	shell terminates	shell terminates
grep Hide 2	Hide	Hide
cat 2 | grep -v a | sort -r	Hide First Dog Apple	Hide First Dog Apple

Table 1: Tests

Our tests try to cover a large amount of edge cases that could lead to errors. However they are not complete and we only assume that our shell does not contain any errors.

Since the formal EBNF syntax description allows for an infinite number of possible inputs (there is no maximum number of chained commands you can use), we can not test all possible inputs to confirm that no input will lead to an error. Therefore we can just try to design very good tests which try to cover as many edge cases as possible.

3 Infinite buffer problem

The infinite buffer problem occurs if two processes read from the same pipe(), then a byte written on that pipe will be given to only one of the processes and never to both. There must be a process in-between that reads the input and writes it to two file descriptors. The UNIX program tee does this for output, while the command cat can be used to mix two inputs.

Our implementation does not suffer from the infinite buffer problem. For every command we do create a new process, however no more than 2 commands use the same pipe, since we use $n - 1$ pipes for n commands. Additionally we use a variable to redirect the output from one pipe to the input for the next pipe.