Report

0. Names & and student numbers:

Matthijs Muis, s1066918

Disclaimer: I thought I could format the chain operator `'|'` with `\|` in markdown, (because it normally creates a table column delimiter). However, it still shows `\|` in some commands. These *should* be read as `'|'`.

1. Implementation

- **Brief discussion of your implementation. Mention how the code traverses the Expression data structure, and motivate your choice.**

First, we have gathered how Expressions are created from parsing of the command line in the function `parse_command_line()`.

- commands of the expression are simply the command line split on the `'|'`-operator (chain).
- arguments/parts of the commands are all the substrings split on spaces. We split on one whitespace character, so if we delimit args with more than one than one whitespace these will end up in an argument.

The for-loop in `parse_command_line()` will check for the following conditions and modify Expression:

1. if the last command ends with a `&`, Expression.background is set to true. This is to denote that the execution of the expression may happen in the background while a new prompt is presented. Then, `&` is popped from the arguments of the last command.
2. if the last command then ends with (">"), these two args are popped and Execution.outputFile is set to .
3. if the

The first thing we need to do is check whether any of the command matches ("cd") or ("exit" *), we need to handle these internal commands using the `chdir()` system call for `cd`:

```
char path[expression.commands[0].parts[1].size()];
strcpy(path, expression.commands[0].parts[1].c_str());
chdir(path);
```

and the `exit()` call for `exit`.

We only exit if the expression consists of exactly one command (so if it is not chained) and this command has as first part `exit`. In bash, chained commands containing `exit` will execute these commands in forked processes, in which case the shell's process is not terminated. a command `exit | exit` will not close the terminal, but just show you a new prompt

However, I think such an expression lik `exit | exit` containing chained `exit`s has no real meaning and should raise an error, because it is probably unintended.

So we check first if:

- exit occurs
- if so, whether the expression has more than 1 command (yes -> raise semanic error)
- if so, whether the command has no extra arguments to exit (yes -> raise syntax error)
- no -> whether there is an input or output file specified (yes -> raise semantic error) (we want to throw an error if something like `exit > hi` or `exit < hi` is given),
- no -> whether the expression runs in background (yes -> raise semantic error)

we also check for expressions that have `cd` in them but in a way that their semantics make no sense. That is:

- `cd` is in the middle of a chain
- `cd` has more than one argument

So this is how `execute_expression()` traverses an `Expression` object:

1. Check for Empty Expression:
   - The code starts by checking if the expression contains any commands. If it's empty, it returns an error code EINVAL. This is a quick check to ensure there's something to execute.
2. Check for Empty Commands:
   - It then goes on to iterate through each command in the expression.

  - For each command, it checks if the parts vector (representing the command and its arguments) is empty. If any command is empty, it prints a syntax error message and returns 0.

3. Handle Internal Commands ('cd' and 'exit'):

    - The code checks for internal commands like 'cd' and 'exit'.
    - If an 'exit' command is found, it performs additional checks: - If 'exit' is part of a chain of commands, it prints a semantic error. - If there are input/output redirections along with 'exit,' it prints a semantic error. - If 'exit' has more than one argument, it prints a syntax error. - If none of the above conditions are met, it exits the program with a status code of 0.
    - If 'cd' is the first command, it checks if it has exactly one argument. If not, it prints a syntax error.
    - If 'cd' has one argument, it changes the working directory using the chdir function.
    - After 'cd' is executed, it removes the 'cd' command from the expression.

4. Create Child Processes and Pipes:

    - The code creates child processes and pipes for external commands.
    - For each command in the expression, it creates a new pipe to connect the output and input of forked processes.
    - It forks a child process for each command and stores their process IDs (PIDs) in the children vector.
    - In the child processes, it sets up input and output redirection using the dup2 function.

5. Close Unused Pipes:

    - After setting up pipes and redirections, it closes the unused ends of the pipes to free up resources.
    - It does this for all pipes except the ones currently being used.

6. Input Redirection Handling:

    - If the current command has input redirection (inputFromFile is not empty), it opens the specified file for reading and redirects it to STDIN_FILENO using dup2. This is only done for the first command in the chain (`i == 0`)

7. Background Execution Handling:

    - If the first command in the expression is executed in the background (expression.background is true) and there's no input redirection, it closes STDIN_FILENO to prevent the process from reading from the terminal.

8. Output Redirection Handling:

    - If the current command has output redirection (outputToFile is not empty), it opens the specified file for writing and redirects it to STDOUT_FILENO using dup2. This is only done for the last command in the chain (and `i == expression.commands.size() - 1`).

9. Execute External Commands:

    - Finally, it executes the external command using the execute_command function.
    - If the executable for the command is not found, it prints a warning and aborts the process.

10. Cleanup and Wait for Child Processes:

    - The parent process closes all remaining pipe ends.
    - If the expression is not executed in the background, it waits for the termination of all child processes stored in the children vector using waitpid.

11. Return Result:

    - The function returns 0 to indicate successful execution.

- **A list of system calls used by your solution and why you use them.**

  1. pipe():

      - Used for creating pipes to establish communication between parent and child processes.
      - Allows data to be transferred between processes.

  2. fork():

      - Creates a new child process.
      - Used to execute commands in separate processes, which is a fundamental concept in shell scripting.

  3. chdir():

      - Changes the current working directory.
      - Used to implement the 'cd' (change directory) command by changing the working directory as specified in the command.

  4. dup2():

  - Duplicates file descriptors, allowing redirection of input and output.
  - Used to set up input and output redirection for child processes when executing external commands.

5. open():

   - Opens files for reading or writing.
   - Used to open files specified for input or output redirection.
   - We use flag `O_RDONLY` for input files and for output we call `open()` with the extra argument `open(path, O_WRONLY | O_CREAT, 0666)`, since if we only read from the file (O_WRONLY), if the file does not exist we want to create a new file (O_CREAT). And with 0666 we set the access bits on linux such that the file is `-rw-rw-r--` in long-listing format. set READ right (S_IRUSR) for user and WRITE right (S_IWUSR) for user.

6. close(): - Closes file descriptors and pipes. - Used to close unused ends of pipes and of STDIN if the expression is in background mode.

7. waitpid():

   - Waits for a specific child process to terminate.
   - Used to ensure that the parent process waits for the termination of child processes, unless the commands are executed in the background.

2. Tests

A table of tested commands.

If you run them in this order, it will not create files in strange places on your computer but just do some random things in `test-dir`

Command	Expected Result	Actual Result
`cd ../test-dir`	Changes directory to `test-dir`	Matches expected.
`ls -l`	List files and directories in long format.	Matches expected
`echo "Hello, World!"`	Print "Hello, World!" to the console.	Matches expected
echo "Hello, World!"	Print "Hello, World!" to the console.	Prints "Hello, World!" to the console. Due to parsing, extra whitespaces are lost (in fact, "Hello, and World!" are parsed as two separate arguments and then combined into a string with only one whitespace which is executed in `execvp()`).
`pwd`	Prints the current working directory.	As expected: Prints `/home/muis/Documents/BSc_Computing_Science/23-24/osc/os-concepts/assignment1/test-dir`
`ls -l > files.txt`	List files and directories in long format and redirect the output to `files.txt`.	created `files.txt` which contains `total 4 -rw-rw-r-- 1 muis muis 27 sep 17 11:22 1 -rw-rw-r-- 1 muis muis 0 sep 17 11:22 2 -rw-rw-r-- 1 muis muis 0 sep 29 12:31 3 -rw-rw-r-- 1 muis muis 0 sep 17 11:22 4 -rw-rw-r-- 1 muis muis 0 sep 29 14:57 files.txt`` (with appropriate newlines, but markdown tables don't seem to want to display that) .
`cat files.txt`	Display the contents of `files.txt`.	As expected.
`grep "Hello" files.txt`	Search for lines containing "Hello" in `files.txt`.	No results, no output. As expected
`ls -l | grep "file"`	List files and directories in long format and filter lines containing "file".	No results.
`ls -l | grep "file"`	List files and directories in long format and filter lines containing file	One result: `-rw-rw-r-- 1 muis muis 221 sep 29 14:57 files.txt`
`mkdir test_directory`	Create a directory named "test_directory".	Created directory `test_directory`
`cd test_directory`	Change the current directory to "test_directory".	As expected

Command	Expected Result	Actual Result
`pwd`	Print the current working directory (should be inside "test_directory").	Yes!
`ls -a`	List all files and directories, including hidden ones.	ouputs `. ..`
`touch new_file.txt`	Create a new empty file named "new_file.txt".	Expected.
`mv new_file.txt renamed_file.txt`	Rename "new_file.txt" to "renamed_file.txt".	Expected
`cp renamed_file.txt copied_file.txt`	Copy "renamed_file.txt" to "copied_file.txt".	Created new file copied_file.txt and copied into it.
`rm copied_file.txt`	Remove "copied_file.txt".	It did.
`cd .. | rmdir test_directory`	Remove the "test_directory" (should be empty).	changed working directory to test-dir, shows `rmdir: failed to remove 'test-directory': Directory not empty`
`echo $HOME`	Display the user's home directory path.	output: `$HOME`
`ls -l > /dev/null`	Redirect the output to `/dev/null` (should not display anything).	It doesn't
`ls non_existent_directory`	List the contents of a non-existent directory (should show an error).	`ls: cannot access 'non_existend_directory': No such file or directory`
`touch nes.sh`	Make non-executable script	It did
`./nes.sh`	Attempt to execute a non-executable script (should show an error).	`Could not find: ./nes.sh`
`chmod +x nes.sh`	Make a script executable.	
`./nes.sh`	Execute an executable script.	It did not throw an error. `nes.sh` is empty so it does not have any effect on output.
`ls -l | grep file > filtered_files.txt`	List files and directories in long format, filter lines containing "file", and redirect the output to `filtered_files.txt`.	As expected.
`cat`	Execute the `cat` command without arguments (should wait for input).	It does! And when I close stdin using `<CTRL><D>` I get to see the prompt again.
`cat files.txt | tee | tee | tee | grep muis | tee > filtered_files_with_extra_steps.txt`	This should put all lines with `muis` in files.txt into filtered_files_with_extra_steps.txt and print nothing as stdout is redirected	It does!

- **Given your tests, are you sure your shell does not contain any errors?**

No, because these are unit tests. They can only recognize errors if there are any, but they cannot guarantee the nonexistence of errors.

- **Given *any* tests, can you be sure the shell does not contain any errors?**

Not in the case of unit tests. Advanced techniques such as static code analysis working from a rigorous formal specification can be used to formally prove that the code follows the formal logical statements in such a specification. Even then we cannot be certain. But since the set of possible inputs and expected behaviours is so big, unit tests cannot exhaust them. We can only hope that we understand the semantics of our code in simple cases and hope to have covered most edge cases with unit tests.

3. Infinite buffer problem

- **Does your implementation suffer from the infinite buffer problem? Explain why.**

The infinite buffer problem is the following problem in interprocess communication that occurs when:

- We have two communicating processes, one producer and one consumer.
- We have an implementation of a communication channel as a buffer/queue. The producer produces first, and only then the consumer consumes from the communication channel. The problem is that the producer is not interrupted with the consequence that the buffer can in theory grow infinitely/too large (to fit in its allocated space), because the producer is never interrupted to let the consumer skim some elements from the buffer.

We don't suffer from this problem because:

- We let the original process create all children in a for loop.
- These children can then run their commands asynchronously, and also asynchronously read/write to their connecting pipes (produce/consume asynchronously to their buffers). The CPUs hardware will ensure that the consumer gets to consume some part of the buffer when the time slice of the producer has expired.

An ultimate proof of concept is given by the following test command:

```
cat /dev/full | head -c 100 > take100.txt
```

If we have an infinite buffer problem, we never terminate (you can almost compare it to eager evaluation) If we don't, this will write 100 `NULL`-characters to `take100.txt`.

I tried it, and it works without any problem. So we can say that producing and consuming is done in parallel, which means that the buffer (unless the producer produces much, much faster than the consumer and there are no safety checks on this in the OS) cannot grow infinitely large.