Operating Systems Lab 1: Shell

Rules

This is the first lab session of the course *Operating Systems*. There are 5 labs in total. Your final lab grade is computed as the average of all lab grades. New labs are typically published every week, with about 1-2 weeks time to submit. For these labs you will work in pairs. Both students will receive the same grade for each lab if they submitted together. If you cannot find a lab partner, please send an email to the TA helpdesk and we will try to help you out.

Each lab consists of one or multiple programming exercises. Some labs might require you to write a short and compact report, in the form of a `README` file accompanying the submitted code. Even when a lab does not require you to submit a `README`, this does not mean you can deliver poorly-documented code: code comments and style continue to be important.

The exercises will all be submitted through Themis, available on `https://themis.housing.rug.nl`. Before submitting your code, **make sure to enroll in a group with your partner first!** You absolutely need to do so before submitting any code, old submissions will not be transferred to the group and this will clutter the view when grading. Your code will automatically be tested for functionality and correctness, but not for style - that will of course be graded by the TAs.

In general, the following grading guidelines apply to all assignments:

- The grade is composed of 70% correctness and 30% quality and style.

  Correctness refers to whether or not you passed all tests in Themis. If you did not pass them all, you might still get partial points. A manual code check might override the results from Themis, for example in the case you hardcoded answers or circumvented explicit assignment instructions. This includes having no memory leaks in a 'valgrind' testcase, if one is present.

  Quality and style include, but are not limited to: sufficient documentation, clear naming of variables, proper separation of concerns, proper formatting, clear code structure, etc. *This is not an exhaustive list!* If you passed very few or no test cases on Themis, we might not award all style points.

- The assessment will be based on the *most recent* submission. Even though you might have submitted a fully-functional submission before, we will only look at the most recent one, also if it does not pass all tests.

- Grades will be uploaded to Brightspace, accompanied with a quick note of feedback on your code. If you have any questions regarding your grade, please contact the TA helpdesk.

Shell - Command execution and composition

For three of the five labs in this course, you will work on developing your own implementation of a shell. A shell is the program that runs inside your terminal: the terminal is the black square on your screen, while the contents of this square are usually dictated by the shell. At the end of the course, your shell will support the following features:

- Starting a program that can be found anywhere in the user's search path (`$PATH`).

- String parsing, for example `./a.out "some string <> with 'special' characters"`.
- Command composition, for example: `./a.out && ./b.out || ./c.out; ./d.out`.
- I/O redirection, for example: `./a.out < in > out`.
- Pipes, for example: `./a.out | ./b.out | ./c.out`.
- Background processes, for example: `./a.out &`.
- Signals, so that you can send Ctrl+C to terminate the child application instead of the shell itself.
- A `kill` command that will terminate a specific (or all) child process(es).

For this first lab, we will focus on the first three aspects: simple command execution and composition. After starting the shell, the user should be able to enter commands on `stdin` after which each command is run. The behaviour is interactive like a normal shell: pressing enter after a command will run it. Of course, it should be possible to run multiple command successively! The shell should exit upon the command `exit`, or when reaching `EOF`.

Your shell implementation must search for the binary to be executed in the user's standard search path available in the environment variable `PATH`. You can access these by having a third argument `char **envp` of `int main()`, or by using the standard library function `getenv()`. It is explicitly not allowed to use system facilities to search the `PATH`.

Any command that is executed should have its `stdin` and `stdout` connected to those of the shell; when using `fork` this will work automatically and no special care has to be taken. Any input consumed by the executed program will automatically be ignored by your shell, unless you purposefully write code to do otherwise. Only in the next iteration we will make this behaviour more complicated.

For this first lab, the focus is on properly parsing commands (including strings, which may contain newlines) together with the composition operations `&&`, `||`, `;`, and `\n`. If you want to prepare your implementation already for features to come in the next labs, you can immediately implement the full grammar, which is formally specified below. This should help you get the right behaviour, since there will be a few small differences compared to traditional shells like `bash`, the most notable being that the I/O redirection acts on a sequence of commands as a whole in a 'pipeline' fashion.

You are allowed to use flex as a lexer generator and/or bison as a parser generator for your shell implementation; you might be familiar with those if you took the course Compiler Construction. Of course, you are **not** required to use those. Any other lexer/parser generators are not installed on Themis, so you won't be able to use them.

Your implementation should also check for command validity. For this first lab, you should check whether the syntax is valid (e.g. strings are properly terminated), or print the error ! `Error: invalid syntax!`. Additionally, you should check that the command exists, otherwise print the error ! `Error: command not found!`. You should recover from such an error and you should not terminate the shell; command interpretation should continue on the next input line.

Each composition operator described above requires the commands to run *in order*. `;` and `\n` act the same: they always execute the next command. In contrast, `&&` only executes the next command when the previous one had exit code `0`, while `||` only executes the next command when the previous one had a non-zero exit code. These operations only 'bind' to the single next command: `true || echo "a" && echo "b"` should print just `b`.

For submission on Themis, **make sure to be enrolled in a group first**. You should submit your code along with a `Makefile` that produces an executable named `shell` which will be run as `./shell` on Themis. Note that we will also test your error handling and memory management!

Hints

- For this first lab, you may only use the system calls `getenv()`, `fork()`, and (variants of) `exec()` to make processes and start executables. It is explicitly **not** allowed to use the functions `system()` and `exec*p*()` (any variant of `exec` with a 'p' in its name), or any other method that will automatically search the `PATH` for you.
- `getenv()` will not return a copy of the environment variables, but the real ones that will be passed to your spawned processes as well! Make sure your child processes obtain the proper set of environment variables.
- Currently, there is only one built-in command: `exit`. In your program design, keep in mind that future labs will ask you to implement other build-in commands, so make sure that will not become a mess down the line.
- You should make sure to properly `wait()` (or `waitpid()`) for each child process to finish, in order to ensure proper command execution order and to prevent creating orphan (processes that are still running but without a parent) or zombie processes (processes that have finished but are not `wait`ed for). Themis will check for this!
- You should **not** use `setpgid()` or similar methods, as this might break Themis when you do not terminate child processes correctly! This will then cause processes to be orphaned, which will keep Themis waiting indefinitely.
- Make sure to disable input and output buffering using `setbuf(stdin, NULL);` and `setbuf( stdout, NULL);`, to prevent out-of-order prints in the Themis output (and consequently failing testcases).
- If you use flex, make sure to set the option `%option always-interactive`.
- Themis will only check the `stdout`, so make sure to print any errors there and **NOT** on `stderr`!
- Themis will check for memory leaks using `valgrind` in all testcases! Memory leaks will be indicated by a 'Runtime error' with exit code `111`.

Grammar

The full grammar describing the syntax to accept in your shell is defined as follows. This includes features to be implemented in future labs. In here, an 'executable' can be the path to a file, or can refer to a file in the user's `$PATH`, while 'builtin' refers to a built-in command. The 'options' part represents any set of parameters/strings that should be put into `argv` of the program or command. From this, it should be clear that a built-in command cannot join any of the I/O redirection and piping that we will implement in future labs. This might simplify your implementation. Additionally, note that any character appearing in the grammar is a 'reserved character' that can only otherwise occur in strings (`""`). Your intuition and experience with shells will be sufficient to know what to allow/forbid in input and note that we will not test 'exotic' inputs in Themis.

⟨*command*⟩ ::= ⟨*executable*⟩ ⟨*options*⟩

⟨*pipeline*⟩ ::= ⟨*command*⟩ `|` ⟨*pipeline*⟩
| ⟨*command*⟩

⟨*redirections*⟩ ::= `<` ⟨*filename*⟩ `>` ⟨*filename*⟩
| `>` ⟨*filename*⟩ `<` ⟨*filename*⟩
| `>` ⟨*filename*⟩
| `<` ⟨*filename*⟩
| ⟨*empty*⟩

```
⟨chain⟩        ::= ⟨pipeline⟩ ⟨redirections⟩
                 |  ⟨builtin⟩ ⟨options⟩

⟨inputline⟩    ::= ⟨chain⟩ & ⟨inputline⟩
                 |  ⟨chain⟩ && ⟨inputline⟩
                 |  ⟨chain⟩ || ⟨inputline⟩
                 |  ⟨chain⟩ ; ⟨inputline⟩
                 |  ⟨chain⟩
                 |  ⟨empty⟩
```

Extensions

For each shell-related assignment, there is the possibility to implement some extensions to obtain up to 2 bonus points (yielding a maximal grade of 12). For this first lab session, you can choose out of the following options:

- Display a prompt before each input line, such as `[folder]>`. Of course, in this case, `;` and `\n` should not be handled identically anymore.
- Implement a built-in `cd` command to change directories.
- Add support for displaying colours.
- Implement a simple command history (when pressing up/down arrows).

Of course, you can get as crazy as you want and do more, these are just a few suggestions. If you decide to implement some of these extensions, make sure to make them togglable using a compilation flag (`#if EXT_PROMPT`, compile with `-DEXT_PROMPT`) and submit to Themis in the separate 'extensions' entry. Make sure to include a `README` documenting your work.