Operating Systems Lab 1: Shell

Rules

This is the first lab session of the course *Operating Systems*. There are 4 labs in total. Your final lab grade is computed as a weighted average of all lab grades, as indicated on Brightspace. The duration of each lab can be also found on Brightspace - they range from 11 days for the earlier ones to 18 days for the last. 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 use the discussion forum on Brightspace or send an email to the TA helpdesk and we will try to help you out.

Each lab consists of one or multiple programming exercises. With the exception of lab 2, each lab will also offer the opportunity to earn some bonus points (for a grade >10) that can be used to compensate lower grades in other labs (however, the average lab grade will be capped at 10). 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. Only in exceptional cases you can change groups, but only after informing the TAs beforehand and only in between lab assignments! 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. Correctness includes having no memory leaks - a 'runtime error' with exit code 111 on Themis indicates a memory leak. Use the 'Error output' to find more details.

    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, and a short note of feedback will be made available on Themis. If you have any questions regarding your grade, please contact the TA helpdesk.

Shell - Command execution and composition

For three of the four 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: it provides an interactive environment to operate your system and start various other programs. 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 execute it. Of course, it should be possible to run multiple commands successively! The shell should exit upon the command ▶ `exit` 〉, or when reaching `EOF`.

Parsing

To get the basic features of the shell working, you need to properly parse 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. Of course, for the first lab it is **not** required to do so. A simple parser accepting just commands, strings, and composition operations is sufficient.

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.

If you have taken courses like *Algorithms and Data Structures in C* or *Functional Programming*, you should already possess the knowledge required to build a working parser for this assignment without these tools. In any case, on Brightspace we will provide two template parser implementations that should get you going quickly.

Command execution

To execute a command, your shell implementation should 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()`. You can also use any appropriate variant of `exec()` that might assist in this.

When a command is executed, the `stdin` and `stdout` of the shell should be passed on to the executed command. When using the `fork()` system call followed with an `exec()` this should work

automatically: any input consumed by the executed command 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 granular.

Shell operation

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), and if that is not the case, 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, just like in a normal shell.

Each composition operator used in this lab (`&&`, `||`, `;`, and `\n`) 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`, as this is interpreted as `▶ (true || echo "a") && echo "b"` (but of course, we do not support parentheses). Note that when a command is not found, its 'exit code' should be `127`.

Since proper command composition requires you to keep track of the exit code of the most recent process, we might just as well add a command to expose the exit code. Implement a command `▶ status` that will print the exit code of the most recent process. For example, after a successful command you should print `The most recent exit code is: 0`. Syntax errors and the built-in command `▶ status` itself should not change the most recent exit code. You can easily simulate various exit codes by calling `▶ bash -c "exit <code>"` in your shell.

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 function `system()` or any other method that will automatically handle forking and process management for you.

- Note that `getenv()` will not return a *copy* of the environment variables, it will in fact return a pointer to the original environment variables which will be passed to your spawned processes as well! Make sure your child processes obtain the proper set of environment variables, so don't modify the contents returned by `getenv()`.

- Currently, there are only two built-in commands: `▶ exit` and `▶ status`. In your program design, keep in mind that future labs will ask you to implement other built-in commands, including ones that will have their own exit code. Make sure that your code is prepared for this, and 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 ordering and to prevent creating orphan processes (ones that are still running but no longer have a parent) or zombie processes (processes that have finished but are not `wait`ed for). Themis will mercilessly check for this!

- You should **not** use `setpgid()` or similar methods, as this might break Themis when you do not terminate child processes correctly! Such an issue 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 `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**. Don't worry about memory leaks between a `fork` and `exec`: `valgrind` won't report those as the entire process will be replaced by the `exec` call. Also exiting your child process after a failed `exec` won't be reported to Themis.

Grammar

The full grammar describing the syntax to accept in your shell is defined below. It already includes features to be implemented in future labs, for completeness. In general, 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 - the objective of this lab is not to build a perfect parser, but a usable shell.

In the grammar, 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 (possibly empty!) set of parameters/strings that should be put into `argv` of the program or command.

From the grammar, 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 (`""`). Strings are only parsed using double quotes, single quotes can be treated as normal characters.

⟨*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*⟩

Bonuses

For each shell-related assignment, there is the possibility to implement some bonuses to obtain up to 2 bonus points (yielding a maximal grade of 12). For this first lab session, these are some of our suggestions:

- 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 creative as you want and do more, we will award points proportional to the features you implemented with a cap of 2 points. If you decide to implement some of these bonuses, make sure to make them togglable using a compilation flag (`#if EXT_PROMPT`, compile with `-DEXT_PROMPT`) and submit to Themis in the separate 'bonuses' entry. Make sure to include a `README` documenting your work - we can only award points for bonuses discussed in your `README`.

As a last note, each bonus item can be awarded points only once in the course - you won't get extra points for the same bonus implementations in the next labs!