NWI-IBC019: Operating systems / Assignment: shell / 2017

In this assignment, you will program a shell in C/C++, as these are the traditional programming languages for operating systems. By creating a shell by yourself, you will gain insights in how processes communicate with the operating system, and how processes are created and controlled by each other.

START TIME *After you read the specified parts of the first three chapters you can start with this assignment.*

DURATION *Keep in mind the time you spend on this assignment: on average it will take **15 hours**.*

OBJECTIVES

- *being able to make system calls from your program;*
- *start processes and let them communicate with each other;*
- *being able to describe the structure of command prompts;*
- *gain insight into the workings of operating systems and processes within;*
- *being able to read the documentation of system calls (using the `man` command);*
- *getting hands on experience with an open-source operating system from the UNIX family.*

DEADLINE *You should hand it in before Sunday October 1, 23:59. Send it to us using Blackboard.*

Background

A shell is a text based interface to the operating system. Shells formed one of the first interactive interfaces for modern computers, with at first a line printer instead of a screen for output. Even today, they have a large part in daily operations, as shells allow users to script applications, combine commands, and quickly pass along extra arguments. That is not possible with the same ease and speed in graphical interfaces. You can see examples of such modern command prompts in recent IDE's (such as Visual Studio Code) and recent shells (such as the fish shell).

The primary function of a shell is to let a user enter commands, execute these commands, and return the results to the user. Examples of UNIX commands are `ls` (directory listing), `pwd` (prints the current directory), and `date` (prints the current date and time). The prompt is also used to pass extra arguments to applications, such as `ls -l` (prints extra information of the entries in the current directory) and `ls [dirname]` (lists the specified directory). Chained execution of commands is also possible so that the output of one application is used as the input to another application. To properly handle all these tasks, a shell needs tight integration with the operating system. System calls such as `fork()`, `execvp()`, `pipe()`, `dup()`, and `waitpid()` facilitate this integration.

Nowadays, shells support higher level programming constructs (they are Turing complete). A simple shell (such as the first shells and the shell you are writing in this assignment) can be constructed with relative ease. Such a simple shell has the following control flow: (1) print a prompt, (2) read user input, (3) parse the user input, (4) execute the commands while directly writing the output of these commands to the screen. For such a simple shell, there is little need for a full blown parser (as in an LL or LR capable parser). Alternatively, the user input can be parsed using the following approach. First, divide the input into the distinct commands (as they can be chained, separated by `|`). Next, these commands are processed separately. The first one command is checked for the `<` operator, and the last command for the `>` operator.

```
⟨input⟩        ::= ⟨expr⟩ ['&']

⟨expr⟩         ::= ⟨command⟩ ['<' ⟨file⟩] ['>' ⟨file⟩]
                 | ⟨command⟩ ['<' ⟨file⟩] '|' ⟨expr-tail⟩

⟨expr-tail⟩    ::= ⟨command⟩ ['>' ⟨file⟩]
                 | ⟨command⟩ '|' ⟨expr-tail⟩

⟨command⟩      ::= ⟨whitespace⟩ ⟨part⟩ ⟨whitespace⟩ ⟨parts⟩

⟨parts⟩        ::= ϵ
                 | ⟨part⟩ ⟨whitespace⟩ ⟨parts⟩

⟨whitespace⟩   ::= ' '

⟨part⟩         ::= 'a' .. 'z' | 'A' .. 'Z' | '0' .. '9' | '_' | '-' | '/'
```

listing 1 Syntax of the requested shell in EBNF notation. A syntax in EBNF describes rules: between ' and ' are literals, between < and > are invocation of (other) rules, and between [are] the optional parts of a rule.

Specification

The syntax (this is not the grammar) that your shell needs to support is described in listing 1. You can use this syntax to construct your parser of the user input. The behaviour of the shell, i.e. the semantics of the input, is specified below.

Your shell will first display the current directory, followed by the prompt '$', and then wait for a line of user input (that is terminated with an enter). Your shell must process the following input and execute it in the way specified below:

- If one of the commands, ⟨*command*⟩ in the syntax, is equal to `exit`:
  The shell will exit.
- If one of the commands start with cd and has one argument:
  The shell changes the current working directory to the specified directory.
- If the input contains a command:
  The shell will execute the command, and displays its output and error messages on the screen. The command can process direct input from the user. The shell will wait with processing new input until the command completes.
- If the command consists of multiple (chained) commands, separated by '|':
  The output of an earlier command is used as input for the next command. Only the output of the last command will be displayed on the screen. All the error messages of all the commands will be directly printed to the screen. The first command can process input from the user.

- If the input ends with '`&`':
  The shell executes the input as specified in the other rules, but does not wait for the completion of the command. The commands will be executed in the background (and the shell continues with displaying a prompt and processing input). The output and error messages will continue to be displayed on the screen. The first command can not get direct input from a user (if the input is not redirected from elsewhere, this input channel should be closed).

- If the last command ends with '`>`' ⟨*file*⟩:
  The shell executes the input as specified in the other rules, but writes the output of the last command to a file in the current directory, which is named ⟨*file*⟩. If this file exists, it will be truncated and overwritten. Of this last command, only the error messages will be displayed on the screen (not the regular output).

- If the first command ends with '`<`' ⟨*file*⟩:
  The shell executes the input as specified in the other rules, but reads the input of the first command from a file named ⟨*file*⟩ in the current directory.

- In all cases, clear error messages must be given if certain commands can not be executed (not found, parse error, etc.).

Afterwards, the shell will again display the prompt, and the above process will repeat.

Method

File descriptors are an important concept in operating systems, especially during programming a shell. File descriptors are the general interface to files, other processes, network resources, hardware, and the user. In the case of our shell, we will use file descriptor as the interface to the screen. With each process, three file descriptors are associated: `stdin` (input, accessible in code as `STDIN_FILENO`), `stdout` (output, accessible in code as `STDOUT_FILENO`), en `stderr` (error messages, accessible in code as `STDERR_FILENO`). You can read from and write to these file descriptors with the `read()` and `write()` system calls.

When a process duplicates (with the `fork()` system call), both resulting processes will obtain the same file descriptors. The standard input `stdin` is thus shared, likewise for the standard output `stdout`, and the standard error stream `stderr`. By replacing these file descriptors, the process can read/write to a different source/destination. This replacing enables chaining commands and redirecting input and output to files.

Problem

To get you up and running, a **VirtualBox environment is made available** to you together with this assignment. Contained in this VirtualBox environment is both a C(++) compiler, the proper settings as well as the Qt Creator IDE. Both C, as well as C++, are allowed. C is the low-level programming language with good interfacing with almost every library, operating system and hardware. You must manage memory on your own in C. The other option is C++, which is the object-oriented version of C. C++ supports you in memory management, as it will take care already of more things (like calling destructors of stack objects).

HINTS

- *The VirtualBox environment has a user 'os' with no password. If you are asked for a password, just hit enter.*
- *A code template is available; please use this a starting point.*
- *Strings in C have the type* `char*`*. You can not compare C-strings using* `a == b`*, you should use* `strcmp(a, b) == 0` *(*`strcmp()` *calculates the difference between the strings).*
- *Useful test commands are:* `echo a b c d`*,* `pwd`*,* `ls -l`*, and* `date`*. To check the expected behaviour of these commands, please use the regular shell.*
- *Examples of chained commands are* `du -kd1 | sort -n` *(sorting all subdirectories based on size) and* `find /etc | head -n 10` *(displays first ten files in* `/etc`*). These chained commands are central to the UNIX way: lots of smaller programs that are very efficient in their specific jobs, and which can be easily combined to achieve powerful effects.*
- *Use the system resources sparsely; the system can run out of resources quite easily.*

STEP 1 To gain insight into the working of the operating system, shells, and processes you can query the manual pages of system calls on every UNIX machine by executing the command `man [system call]`. Look up at least the following commands: `fork()`, `execvp()`, `pipe()`, `waitpid()`, `open()`, `close()`, `dup()`, `chdir()`, and `exit()`. Also take a look at the 'related' section in the manual pages, as these could be relevant for your understanding. Also look at the given template for your solution.

STEP 2 Think about how you are going to implement your shell. Design the approach you are going to use to interpret and execute the command line, especially for the chaining of commands. Take care that you have all the needed information when you start processes. Document your design and your data structured in a short fashion (one paragraph).

STEP 3 Implement your solution, one step at a time (in the same order as the specification). A testing framework is included in the template project (look for `shell.test.cpp`) if you want to use unit tests and functional tests during development, which is highly recommended. Document your code.

STEP 4 Extend the given tests, and describe weaknesses in the testing. Given your extended tests, are you sure your shell does not contain any errors?

Discussion

The code that you have written in this assignment does not have that many lines of code. If you think about the functionality that your code contains, and what you can control with it, the logical conclusion is that the interface to the system that is offered is very dense and powerful. This is especially true for the UNIX paradigm (with lots of smaller commands that you can chain). To extend your shell to a real one, lots of useful features are needed such as tab completion and running processes in the background. However, the core of the shell stays compact.

HAND IN

- *a short description of the design of your solution, covering the relevant data structures and system calls (as described in step 2).*
- *implementation (as described in step 3).*
- *the used tests (as described in step 4).*
- *reflection on the testing (as described in step 4).*