NWI-IBC019: Operating systems / Assignment: forking / 2021 v1

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 other processes, and how processes are created and controlled.

NOTE *This is a group assignment.*

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 **18 hours** per group member. Are you stuck? Please seek assistance.*

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 Thursday 30 September, 23:59. Send it to us using Brightspace.*

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. For example, `cat file.txt` prints the contents of `file.txt` while `cat file.txt | tail -n 3` only prints the last 3 lines of `file.txt`. 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. 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.

Specification

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` or the standard input of the shell is closed (closing the input stream can be simulated by pressing Control-D):
  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. This is a so called internal command, as the state of the shell process is changed (in step 1, see the man pages of `chdir` and `execvp`).
- 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. Only 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. Upon request, we have a formal EBNF syntax description (default way of specifying the input that is accepted by a program or function) available of this specification.

Method

The **shell** you are required to make, is a process on your system just like all other programs. It executes multiple commands after each other. For each command, new (operating system) processes are started and the shell waits until all of them are finished. The started processes can communicate with each other, and the kernel of the operating system facilitates this communication. To make the communication easier, many operating systems have a general concept of communication channel, called **file descriptors**.

File descriptors are an important concept in operating systems, especially during programming a shell. We are using the UNIX variant of this concept, in which file descriptors are the general interface to files, other processes, network resources, hardware, and the user. In the case of our shell, we will also 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`), and `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. A communication channel between processes, such as those made by `pipe()`, has write-ends and read-ends. These are also described by file descriptors. The write-ends and read-ends of a communication channel can be duplicated (with `dup2()` or just like all file descriptors with `fork()`), and the command `close()` closes only one of the write-ends or read-ends. If on one of the write-ends a byte is written, only one of the read-ends receives this byte. Once **all** write-ends are closed, all read-ends get a `end-of-file` signal (the next `read()` system call on that pipe will produce a return code of 0).

There are technical reasons why not all commands are accepted (see the specification). For example, we have the **invalid** command `cmd > output-file | cmd2 < inputfile`. This command is invalid **by design**. The write-ends and the read-ends of pipes can not be used referenced/connected multiple times to duplicated the output/input. This is because the operating system does not support this functionality due to the infinite buffer problem: 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.

The obvious solution for executing chained commands would be the following: wait until the first one is finished, keep its output, and then you start the second command and write the data to this process. This is not a solution that will work for large outputs, with potentially infinite data streams. Instead both commands have to be started simultaneously and the input/output must be passed immediately to the other command (without buffer inside the shell process).

Problem

Your goal is to build the shell as specified above. There are project files handed out that deal with the parsing. Both C, and 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). The goal of the assignment is to understand how to communicate efficiently with the operating systems, not learn C(++). If you are stuck, please seek help immediately.

HINTS

- *A code template is available; please use this a starting point (see step 0).*
- *Strings in C have the type* `char*`. *You can not compare C-strings using* `a == b` *(as you are compare memory addresses instead of contents), you should use* `strcmp(a, b) == 0` (`strcmp()` *calculates the difference between the strings). To retrieve a C string from a C++ string named* `str`, *call* `str.c_str()`. *Or use the C++ string compare (*`a == b` *if the type of* `a` *and* `b` *is* `std::string`*).*
- *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.*
- *When executing chained commands, all commands should be run concurrently. It may seem logical to run the commands in one at a time, but this will result in buffering problems when much data is redirected and processed.*

STEP 0 Read the instructions how to set up a C++ development environment (using the Visual Studio Code editor), and download the project files at `https://gitlab.science.ru.nl/operatingsystems/assignment1`. Try to get the project working. Run it. It should give you a prompt, and you can enter a single command which it will execute. Afterwards your shell will terminate (and probably return to the default shell).

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()`. These are the only system calls you will need (hint: do not use `read()` or `write()`). Also take a look at the 'related' section in the manual pages, as these could be relevant for your understanding.

STEP 2 Look at the template given. Zoom in on the data structure given below (defined in `shell.h`). The commands are executed from this data structure. Take a look at the given implementation skeleton in the project to understand the basic structure.

```
struct Command {
  vector<string> parts = {};
};

struct Expression {
  vector<Command> commands;
  string inputFromFile; // if empty input is from keyboard
  string outputToFile; // if empty output goes to screen
  bool background = false;
}
```

STEP 3 Think about how to execute the data structure. Which system calls do you need? How is the data structure traversed to execute it correctly? Write this down **briefly**. Hint: both the recursive and iterative version can work, most find the iterative version easier. Focus on readability and correctness.

STEP 4 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 5 Extend the given tests, and describe weaknesses in the testing. Given your extended tests, are you sure your shell does not contain any errors? Make clear your shell does not leave file descriptors open unnecessary, and does not suffer from an infinite buffer problem (see above).

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 3).*
- *implementation (as described in step 4).*
- *the used tests (as described in step 5).*
- *reflection on the testing (as described in step 5).*