NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTAMENTO DE INFORMÁTICA

Implementation of a simple Shell

LAB 2

Objectives

System calls *fork( )*, *exec()* and *wait( )*. Implementation of a simple shell.

The simple shell

A shell is a command line interpreter (CLI) with the ability to execute commands, usually written by the user at the terminal. There can be internal commands, implemented as code in the shell; and external commands, that are other programs spawned by the shell. The shells can execute external commands in two modes: synchronous and asynchronous (or background). The former is the default and ensures that a new command is only accepted after the conclusion of the previous one. If the background command execution is selected, the shell does not wait for the conclusion of such command, allowing for others to be issued and executed concurrently.

The way a Shell operates is depicted in Fig. 1 below. The execution of an internal command (left side) is as follows: the user types the command, e.g., **`echo hello`**; the shell analyses the string and invokes the appropriate function; the output goes to the terminal.

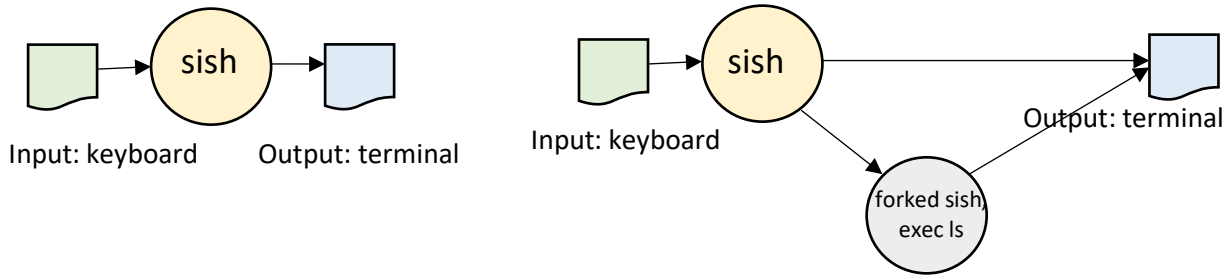

Figure 1. Command execution: internal (left) and external (right) commands

The execution of an external command (right side) is as follows: the user types the command, e.g., **`ls`**; the shell analyses the string and invokes a **`fork()`**. The child process executes **`ls`** program using **`execve()`** or a similar call, while the original process (the parent) is blocked on a **`wait()`**; the output of the **`ls`** goes to the terminal; finally, the **`ls`** terminates and that also terminates the **`wait()`**, unblocking the parent process and resuming the execution of the original shell process.

Shells advertise that they are ready to accept commands via a prompt. Our Simple Shell, **`sish`**, prompt will be **`sish>`**, so the previous examples can be shown as:

```
sish> echo blabla
blabla
sish>
```

```
sish> ls
sish    sish.c    sish.o
sish>
```

Requesting the execution of a command in background is a matter of terminating the command line with "**`&`**". The simplified pseudocode for the **`sish`** program is:

```
While !end-of-program:
      Print "sish> "
      Read User input
      Analyze string  (internal-command? background?)
      if internal-command, execute function
      else fork() and exec(… external-command …)
           if !background, wait()
end-while
```

sish commands

Just one internal command must be implemented: "exit". This will, naturally, terminate **`sish`**.

Any word that is not recognized as internal, will be assumed as an external command, available on a directory in the PATH[1], and its execution will be attempted. If the "command" execution is not successful, a friendly error message must be outputted. Start by assuming that a command is just one word, without arguments.

Background execution

If the input string ends with the ampersand character "&" and the command is external, e.g., "**`ls&`**", it will be executed in the background, concurrently with **`sish`**.

Accepting command arguments

Notice that command options are commonly used (e.g., **`ls -la /bin`**). So, improve your code to deal with a variable number of arguments, by parsing the command line and building an array of pointers terminated by **`NULL`**, as a main's argv. The provided function `makeargv`, is very helpful to create that array from a string:

```
int makeargv(char *s, char *argv[])
```

Bibliography

- Section "Laboratory: Tutorial" of recommended book: http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf
- Must read, from the FSO recommended book, the chapter about process creation available on https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf
- On-line manual pages (use man command) for LibC and system call functions: **`fork`**, **`wait`**, **`exit`**, **`execve`**, **`execvp`**, **`strcmp`**, **`isspace`**, **`etc`**.

[1] If one uses a function like `execvp`, the `PATH` will be used.