FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
DEPARTAMENTO DE INFORMÁTICA

Implementation of a simple Shell
(Lab-03 – September 2019)

1. Introduction and Lab objective

A shell, as any other command line interpreter, has the ability to execute a number of commands, both internal, i.e., implemented as code (usually structured around distinct functions) and external, i.e., other programs that are spawn by the shell. Although the most common usage of a shell is interactive, i.e., the user writes a command in the keyboard that gets executed and prints its output in the terminal (while the shell itself is blocked if the command is external) the input and output can be retrieved (and/or printed) from a text file.

In this lab students will develop a simple shell that executes both internal and external commands, support concurrent (a.k.a. background) execution and input/output redirection. External commands require **`fork()`**/**`exec()`** calls to be programmed; synchronous (i.e., non-concurrent) execution requires the use of **`wait()`**; and, finally, redirection requires a knowledge of how the file descriptor table is changed by **`open()`** and **`close()`** calls.

2. The simple shell

The way the Simple Shell (**`sish`**) 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`**; **`sish`** analyses the string and invokes the appropriate function; the output goes to the terminal.

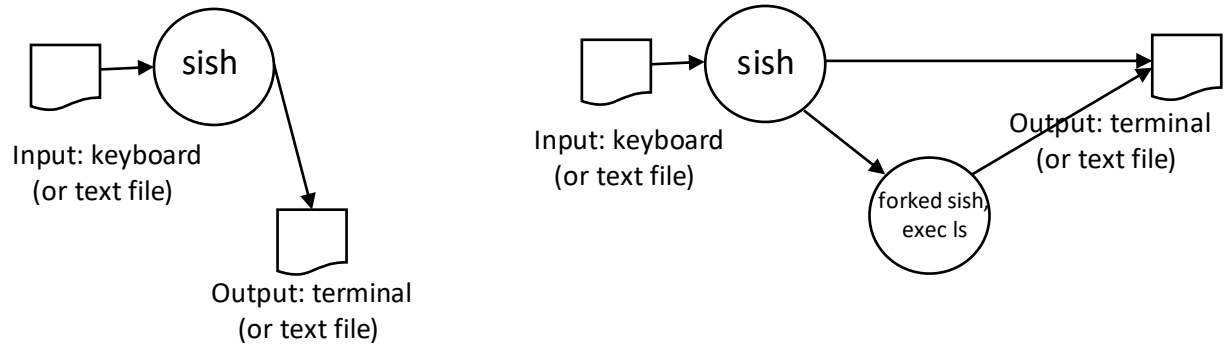

Figure 1. The sish simple shell command execution: left, internal; right external

The execution of an external command (right side) is as follows: the user types the command, e.g., **ls**; **sish** analyses the string and invokes a **fork()**, followed by an **exec()**, while the original **sish** process is blocked on a **wait()**; the output of the **ls** goes to the terminal; finally, the **ls** terminates and that also terminates the **wait()**, and the original **sish** process resumes execution.

Shells advertise they are ready to accept commands via a prompt; our **sish** prompt will be **sish>**, so the previous examples can be shown as (user input in blue):

```
sish> echo blabla                sish> ls
blabla                           sish    sish.c    sish.o
sish>                            sish>
```

The simplified pseudocode for the **sish** program is:

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

2.1. sish commands

The two internal commands are "echo" and "end"; naturally, "end" terminates **sish**, while "echo" prints every character that follows the echo word, as seen in the above examples.

Any word that is not recognized as internal, will be assumed as an external command available on the /bin directory, and its execution will be attempted. If the "command" execution is not successful, a friendly error message must be outputted. Notice that command switches (e.g., **ls -la**) must be accepted.

2.2. Background execution and Redirection

If the input string ends with the ampersand character "&" and the command is external, it will be executed concurrently with **sish**; usage of "&" with internal commands must be flagged as an error.

If the input string includes the characters "<" and/or ">", followed by a valid filename, the command will be executed with the input taken from the filename that follows "<" and/or the output will be printed in the file whose filename is specified after the ">". For the output redirection, if *filename* already exists, it should be truncated.

3. Available working code

The redirection and exec demos that were performed in the classes ("aulas teóricas") are available in a companion zip, stored in CLIP. You should try, modify and experiment different situations before taking on the development of your shell.

4. Simplifying the exec code

The exec demo that was performed in the classes ("aulas teóricas") used the **`execl()`** system call. This is fine for executing programs with a predefined number of arguments. However, in our shell, the number of switches/arguments is variable, and **`execl()`** is not at all suitable (try it!). So, when refining your code to deal with a variable number of arguments, you should use the **`execv*()`** family of functions, which use an array of pointers terminated by NULL to pass the various arguments to the system call.

The function bellow is very helpful to create that array:

```
int makeargv(char *s, char *argv[]) {
  // in: s points a text string with words
  // pre: argv is predefined as char *argv[ARGVMAX]
  // out: argv[] points to all words in the string s (*s is modified!)
  // return: number of words pointed to by the elements in argv (or -1 in case of error)

  int ntokens;

  if ( s==NULL || argv==NULL || ARGVMAX==0 )
      return -1;

  ntokens = 0;
  argv[ntokens]=strtok(s, " \t\n");
  while ( (argv[ntokens]!= NULL) && (ntokens<ARGVMAX) ) {
      ntokens++;
      argv[ntokens]=strtok(NULL, " \t\n");  // breaks 's' inline at separators
  }
  argv[ntokens] = NULL;  // terminate with NULL reference
  return ntokens;
}
```

4. Must Read list

Now, you:

- Must read, from the FSO recommended book, the lab tutorial chapter available on http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf
- On-line manual pages for LibC and system call functions: fork, wait, exit, exec and strcmp.

Finally: if you find any error, mistake, typo, etc., please report it to poral@fct.unl.pt.