Fundamentos de Sistemas de Operação

MIEI 2018/2019

Laboratory session 3

Objectives

Command interpreter (*shell*) with external commands (*execve* system call), IO redirection, background execution and signal handling.

Execution of External Commands

Complete the function `runcommand(char *argv[])` that receives as argument a vector of strings defining **one command and its arguments**, to execute this command in a new child process. For this, you must use the *execve* system call, or another related function like ***execvp***, to execute the new program in the child process. When using *execvp,* the command specified in *argv[0]* can be any executable file accessible from a directory in the environment variable *PATH* (note: to show the current value of this environment variable in a terminal, run the command "`echo $PATH`"). For now, the implementation of *runcommand* continues to wait for its child process to terminate, by using the *wait* system call, before returning to the Shell's main execution cycle.

Executing commands on the background

The Unix shell allows users to run commands on the background, that is, run commands without having the shell waiting for their conclusion before presenting the prompt. To trigger such behavior the user simply has to terminate the command line with character '**&**', as in the following example:

```
> gedit &
   Process 12345 is executing in the background.
>
```

In such case, `gedit` executes in background, while the shell becomes immediately ready to execute a new command. This is achieved by having the shell process, after using fork to launch the new process *p*, not wait for *p*'s conclusion but instead proceed to printing the prompt and read the user's next command. Implement this functionality in your shell and, in order to inform the user of this behavior, print a message with format "`Process pid_of_child_process is executing in the background`" as depicted in the previous example.

Please note that, if no extra action is performed by the shell (the father process) when the child process terminates, the latter is kept in a *zombie* status by the system. To avoid this, i.e. to clear any trace of a child process, its father process (in this case the shell) has to invoke the ***wait*** or ***waitpid*** system calls later so that it receives its child's exit status. At this point you should print a message acknowledging the conclusion of the command and its termination status. An example of such a message is:

```
Process 12345 has concluded its execution with status 0
```

The command `ps` can be used to inspect the current processes in the system, including the "*zombie*" processes, e.g. by doing `ps aux`.

Sequence of Commands

Extend your shell so that it supports a sequence of two commands in the same command line, which is specified by the character **';'** acting as a command separator/sequencer. This means the shell has to wait for the execution of the first command to terminate before executing the second command. Only then, the shell returns to its main execution cycle. For instance, considering the following command line,

```
> gcc -Wall -O -c mycalc.c ; gcc -o mycalc mycalc.o -lm
```

the shell first compiles the C file "mycalc.c" generating an object file. Only when this compilation ends, the shell launches the linkage of the resulting object file with the math library.

From the code above, the array of strings *av* now defines **two commands and its arguments** separated by a string containing the separator character, i.e. "**;**". Your implementation has to parse the array looking for the string ";" in order to first identify the words composing the first command, execute it in a process child, and wait for its termination. Subsequently, the code launches the execution of the second command and waits for its termination, before returning to the Shell's main execution cycle.

Bringing Commands Back to the Foreground

Implement the `fg` internal command that brings to the foreground that last command issued for background execution. This means that the shell now waits for the conclusion of such process.

Extend this functionality to bring back any command issued for background execution. To that end, the *fg* command may now receive an optional *pid* argument. If no *pid* is given, the shell waits for the last command issued for background execution as before, otherwise it waits for the conclusion of the process with the given *pid*. For instance, in the following example

```
> fg 12345
```

the shell waits for the conclusion of process 12345, if such process exists. If not, it emits an error message.

Bibliography

Look at examples from classes 8 and 9.
Check section 5.4 from OSTEP book (ostep.org)
Read reference manuals (*man*) for `execvp, wait, fg, bash`.