CS:3620 — Spring 2020 — Homework 2

In this homework, you will build something similar to a Linux shell. Your shell, however, will not get commands iteratively from the user, but it will parse some files to determine which commands to run.

Please, ask general questions about the homework on ICON, so that everyone can benefit from the answers.

General Requirements Submit your homework as a *single* `tar` file. When unpacked, the `tar` file must have the following directory structure (substitute ⟨your_HawkID⟩ with your actual HawkID):

⟨`your_HawkID`⟩
```
    compile.sh
    printargs.c
    printargsandenv.c
    shell.c
```

The `.sh` file must have the `executable bit` set. No other files should be present in the archive.

You must write your code using the language C.

Preliminary Tasks

1. Write a C program printing to sdout all its arguments (including `argv[0]`), one per line. This is similar to what you had to write for **Homework 1 – task8**. However, this time you do not have to hex-encode the printed arguments. Save the source code of this program in `printargs.c`.

2. Write a C program printing to sdout all its arguments (including `argv[0]`), one per line, followed by all its environment variables (again, one per line). You can access environment variables using the following variant for your `main` function:
`int main(int argc, char *argv[], char* env[])`
The environment variables are stored as an array of strings in the variable `env`. This array is terminated by an element whose value is `NULL`. Save the source code of this program in `printargsandenv.c`.

If you run the code of this C program in the following way:
`./printargsandenv argument1 argument2`
the output should be similar to:

```
./printargsandenv
argument1
argument2
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-antoniob
XDG_SESSION_ID=c1
VIRTUALENVWRAPPER_SCRIPT=/usr/share/virtualenvwrapper/virtualenvwrapper.sh
.  .  .
XAUTHORITY=/home/antoniob/.Xauthority
COLORTERM=gnome-terminal
_=./printargsandenv
OLDPWD=/home/antoniob/git
```

(I omitted many lines. Also, the values of your environment variables will likely be different.)

3. Copy the file `shell.c` I provided in the homework `tar` file. This file contains some of the code you will need to use to write your shell.

4. Write a shell script named `compile.sh`. This script must compile the code in `printargs.c` in an program called `printargs`, the code in `printargsandenv.c` in an program called `printargsandenv`, and the code in `shell.c` in an program called `shell`.

The `printargs` and `printargsandenv` programs will be useful to test the functionality of your shell. The code I provided in `shell.c` constitutes a starting point you should use to write your shell.

In the following sections, I will also refer to some `.txt` files. You can find them in the `tar` file of the homework. When testing your shell, I suggest to place them in the same folder where your `shell` compiled program is. This folder should also contain `printargs` and `printargsandenv`.

You can assume that your shell will be run using ⟨your_HawkID⟩ as the current working directory.

Main Task

You need to write a C program (save its source code in the file called `shell.c`) similar to a Linux shell. Your shell, however, will not get commands iteratively from the user. Conversely, it will parse some files to determine which commands to run.

In particular, your shell must accept as arguments an arbitrary amount of paths to `command_files`. A `command_file` is a text file containing, one per line, the following fields:
⟨`binary_path`⟩
⟨`stdin`⟩
⟨`stdout`⟩
⟨`stderr`⟩
⟨`arguments`⟩
⟨`extra_environment`⟩
⟨`use_path`⟩
⟨`copy_environment`⟩
⟨`niceness`⟩
⟨`wait`⟩
⟨`timeout`⟩
The meaning of each field will be explained below.

In general, every `command_file` specifies a subcommand your shell should execute and some properties regarding how it should be executed. The commands described in the `command_files` specified as arguments of your shell should be executed in order.

The file `shell.c`, which you can find in the homework `tar` file, provides a scaffolding for the code of your shell. In particular, it provides code to parse a `command_file` and generate a variable of type `command`. You can assume that if a `command_file` is correctly parsed by the function `parse_command` (i.e., the function `parse_command` returns 1), all the parsed values respect the specification given below.

All the commands must be executed in different child processes of the main parent process. Every time a new process is created, your shell must print (in a separate line):
"New child process started ⟨`new_process_pid`⟩"
Every time a child process terminates, your shell must print (in a separate line):
"Child process ⟨`child_process_pid`⟩ terminated with exit code

⟨child_process_exit_code⟩"
Your shell must not terminate before all the executed commands are terminated as well.

The aforementioned fields of a `command_file` have the following meaning:

⟨`binary_path`⟩: This is the path (absolute or relative) of the program you need to execute.

⟨`stdin`⟩: If this field is not an empty string, the command's stdin should be read from the specified file, instead than from the terminal.

⟨`stdout`⟩: If this field is not an empty string, the command's stdout should be redirected to the specified file.

⟨`stderr`⟩: If this field is not an empty string, the command's stderr should be redirected to the specified file.

⟨`arguments`⟩: Specify the arguments of the program you have to run.

The arguments are hex-encoded, you have to write your own code to hex-decode them.

When hex-encoded, the different arguments are separated by the string "00", therefore, once decoded, the different arguments will be separated by a `NULL` byte.

⟨`extra_environment`⟩: Specify some environment variables you should set for the executed program. This field is hex-encoded. As for ⟨`arguments`⟩, when hex-encoded, the different extra environment variables are separated by the string "00", therefore, once decoded, the different arguments will be separated by a `NULL` byte.

⟨`use_path`⟩: This value can be either 0 or 1. If 1, your shell should search for ⟨`binary_path`⟩ in all the directories specified by the `PATH` environment variable. You can obtain this behavior by using the `execvpe` function call.

⟨`copy_environment`⟩: This value can be either 0 or 1. If 0, the environment variables of the shell should not be propagated to the executed command. If 1, the environment variables of the shell should be propagated to the executed command. In both cases, the environment variables potentially specified in ⟨`extra_environment`⟩, should be set in the executed command.

⟨`niceness`⟩: This value is a number between -20 and 19. You need to set the niceness of the executed command to this value (use the `setpriority` function in the child process, after `fork`, but before `execv`).

⟨`wait`⟩: This value can be either 0 or 1. If 0, the shell will not wait for the termination of this command, before the execution of the next command. Otherwise, the shell will wait for the termination of the execution of this command, before executing the next one (if present). Please notice that,

regardless of the value of ⟨`wait`⟩, before exiting, the shell has to wait for the termination of **all** the launched commands. As soon as **any** child process is terminated, the shell has to print out the pid of the terminated process together with its exit code, as specified before.

⟨`timeout`⟩: This value is an integer number greater or equal than 0. If different than 0, you have to terminate the executed command after ⟨`timeout`⟩ seconds. To implement this functionality, use the utility called `timeout`. In particular, suppose that the command you need to execute is `ls -la` and ⟨`timeout`⟩ is set to 15. Then, instead of just executing the command `ls -la`, you should execute the following command:
`/usr/bin/timeout --preserve-status -k 1 15 ls -la`

Other Details and Hints

When calling `execv` and any of its variants, the valued specified as `path` and the value specified as `argv[0]` must always be the same. This must be true even when running a command with a timeout. The different arguments eventually specified by the field ⟨`arguments`⟩ in the `command_file`, indicates what you should pass as `argv[1], argv[2], ...`

To use the function `execvpe`, you need to add the following line at the beginning of your source file:
`#define _GNU_SOURCE`
In addition, `execvpe` does not work if the `PATH` environmental variable is not set. This is a problem when the command does not inherit this environment variable from the parent process (i.e., ⟨`copy_environment`⟩ is set to 0). You can ignore this issue.

Setting a `niceness` value lower than 0 does not work unless the shell is executed as the `root` user. You can ignore this issue.

You should be carefully in how you open the files used for `stdin`, `stdout`, and `stderr` redirection. First of all, you can assume that, if specified, the `stdin` file exists. Likewise, you can assume that the directories containing `stdout` and `stderr` already exist and they are writable by your shell's user. If already existing, you should delete the content of the `stdout` and `stderr` files, before writing into them. You can get the aforementioned behaviors by using:
`open(path, O_WRONLY|O_CREAT|O_TRUNC, 0664)`
to open the files used as `stdout` and `stderr`, and using:

`open(path, O_RDONLY, 0664)`
for the file used as `stdin`.

Examples

You can find the files mentioned in this section in the `tar` file of the homework. Also, these examples assume that you have the program `printargs` and `printargsandenv` already compiled in the same directory where you compiled your shell. Obviously, some output, such as the values of the different pids or file timestamps, may be different.

By running:
`./shell ls.txt`
Your output should be:

```
New child process started 31007
total 180
drwxr-xr-x 12 root root 4096 Aug 30 16:39 .
drwxr-xr-x 26 root root 4096 Aug 30 16:39 ..
drwxr-xr-x 2 root root 102400 Sep 4 16:26 bin
drwxr-xr-x 2 root root 4096 Aug 30 16:42 games
drwxr-xr-x 76 root root 4096 Aug 30 16:44 include
drwxr-xr-x 174 root root 16384 Sep 4 16:24 lib
drwxr-xr-x 3 root root 4096 Aug 30 16:40 lib32
drwxr-xr-x 3 root root 4096 Aug 30 16:40 libx32
drwxr-xr-x 10 root root 4096 Jul 24 22:03 local
drwxr-xr-x 2 root root 12288 Aug 30 16:59 sbin
drwxr-xr-x 428 root root 16384 Sep 4 15:07 share
drwxr-xr-x 7 root root 4096 Aug 30 14:04 src
Child process 31007 terminated with exit code 0
```

By running:
`./shell ls2.txt`
Your output should be:

```
New child process started 31340
ls:  cannot access 'ZZZnonexistingfile':  No such file or directory
Child process 31340 terminated with exit code 2
```

By running:
`./shell sleeptimeout.txt`
Your output should be:

```
New child process started 29172
Child process 29172 terminated with exit code 143
```

The last line should be printed approximately 10 seconds after the first one.

By running:
`./shell printargsenv.txt`
Your output should be:

```
New child process started 29317
./printargsandenv
100
ZZZ=z
ZZZ=z
YZZ=z
Child process 29317 terminated with exit code 0
```

By running:
`./shell sleeptimeout.txt bash.txt stracegrep.txt`
Your output should be:

```
New child process started 29593
New child process started 29594
Child process 29594 terminated with exit code 0
New child process started 29596
Child process 29596 terminated with exit code 0
Child process 29593 terminated with exit code 143
```

The last line should be printed approximately 10 seconds after the others. Your shell should also create a file called `stdout`, containing the word `hello`, and a file called `stderr` containing the list of all the system calls executed by the command `grep` (obtained by using `strace`). Both files should be saved in the shell's current working directory.

By running:

```
./shell sleep.txt sleeplonger.txt
```

Your output should be:

```
New child process started 4610
New child process started 4611
Child process 4610 terminated with exit code 0
Child process 4611 terminated with exit code 0
```

The third line of your shell's output should be printed approximately 5 seconds after the start of your shell. The fourth line should be printed approximately 10 seconds after the start of your shell. The third line should contain the pid of the first started child process and the fourth line should contain the pid of the second started child process.