Lab 2: the *xssh* shell

It can be argued that the main purpose of an OS is to simplify the execution of computer programs. A central part of this is the program that allows a user to execute other programs: on Unix-based OSes this is the *terminal* or *shell*. The shell is not part of the operating system kernel, but it is a natural extension of it and in many ways represents the operating system itself!

We never want to take programs like `bash` or `csh` for granted, so we'll go through the dirty work of building our own shell from scratch. Through the implementation, we will practice and learn system calls that are related to process, which is an important abstraction in any OS.

In this lab assignment, you will:

1. Spawn new processes with the `fork()` and `execvp()` system calls
2. Wait for child processes with the `waitpid()` system call
3. Perform basic signal handling
4. Change the working directory of a process with `chdir()` system call
5. Perform complex input parsing in C – Jing: I have helped you to implement most of it.
6. In optional challenge exercises: pipe data between processes and stdin/stdout redirection

---

You will need a Linux machine (or a MacOS) to complete the lab. You may see warnings with the provided code, depending on which OS and compiler you are using (e.g. LLVM in MacOS).

This lab has 6 points in total.

Submission:

When finished, submit only your modified `xssh.c` in Moodle.
Note that you must not include any binary executable files, meaning any compiled programs or intermediate build objects (.o files, for example).

Preparation:

Download the `xssh.c` and `Makefile`. You will implement everything in `xssh.c`, in which many hints and detailed instructions are provided in addition to basic comments. The `Makefile` can help you to compile your code. To compile, type:

```
make
```

To clean up the executable files, type:

```
make clean
```

To execute the xssh shell after make, type:

```
./xssh
```

To complete the exercises below, follow the instructions and hints provided in xssh.c indicated by “FIXME” and “hint”. After you finish the implementation, do try different commands, including the examples provided below, to see if your implementation is correct or not.

During your implementation, please write good comments explaining your code. Additional credits may be deducted if your xssh shell does not pass all the test commands and it is not clear which exercises are causing this problem by looking at your comments.

Required Exercises:

The xssh should be able to run the following instructions:

0. exit I:
   Exit the shell and return status I
   If I is valid, return status I; if exit is the only command, return 0; else return -1.
   Currently, this may be the only way to exit properly.
   e.g. `exit 1`
1. show W:
   Display (print to screen) whatever W is.
   e.g. `show some random words`
2. team:
   Print the name of your team members in the format "team members: xxx; yyy; zzz" in one line; Make sure that the name of each person who worked on these exercises is listed.
   If your team has finished both optional exercises, add “ finished optional (a) and (b)” in the same line; if (a) is finished, add “ finished optional (a)” in the same line; similarly for (b)
   e.g. `team`
3. chdir D:
   Change the current working directory to ./D
   If ./D does not exist, print an error message "-xssh: chdir: Directory 'D' does not exist"
   If ./D exists, print a message "-xssh: change to dir 'D'\n"
   e.g. `chdir ./`
4. wait P:
   Wait the child process with pid P, and print some message specified in xssh.c.
   If P is -1, wait all the child processes, and print some message specified in xssh.c.
   e.g. `wait -1`
5. External executable instruction to be execute by the shell:
   e.g. `sleep 1`
   You need to support the use of '&' at end of a line to indicate that the program will be executed in the background. The code to parse '&' is provided, but you need to make sure it is at the correct place and write the code to distinguish whether '&' appears.
   e.g. `sleep 10&`
6. "CTRL-C" handler:
   Terminate the foreground process but not xssh, and print "-xssh: Exit pid childpid", where childpid is the pid of the foreground process
   e.g. type “`sleep 100`”, then type control key and c key

The xssh also supports the following functionalities; the corresponding code has been provided, but you may need to put them in the correct place in order to run correctly:

7. Support comment '#', blank lines are ignored, support multiple whitespaces.
   e.g. `team #some random comments`
8. Support PID variable $$, and last background pid $!
   e.g. `show $$` <-- this will print the pid of the current xssh process
   e.g. `show $!` <-- this will print the pid of the last process that was executed by xssh in the background

The xssh currently does not support many functionalities that are supported in other shell, such as pipe using '|', multiple commands in the same line ';', stdin/stdout redirection '<' and '>', etc.

Optional challenge exercises:

a. Support UNIX-style pipe C1|C2|C3 (*worth 1 extra credit if corret*):
   See "The UNIX Timesharing System" paper Section 6.2 for more explanation.
   In xssh.c, implement (replace the printf inside) the function 'pipeprog'.
   You will need to add the code to parse '|' correctly by yourselves.
   e.g. `pwd | ls | head -1`

b. Support stdin/stdout redirection (*worth 1 extra credit if corret*):
   The redirection syntax is "C < F1 > F2", where it can only have either '<' or '>', but if it has both then the order should be '<' before '>'.
   Also, the file should already exist in order to open it. If the file does not exist, print an error message.
   You will need to implement this in function program() in xssh.c.
   You will need to add the code to parse '<' and '>' correctly.
   e.g. `vi log1.txt; vi log2.txt; cat < log1.txt > log2.txt`

Documentation

The following man pages may be useful:

- execvp (2)
- fork (2)
- wait (2)
- strtok (3)
- strcmp (3)
- chdir (2)
- close (2)
- dup2 (2)
- pipe (2)
- signal (2)
- sigaction (2)