CS:3620 — Fall 2022 — Homework 3

This homework is due on November 8th, 2022 at 11:59 PM CDT.

General Requirements Submit your homework as a *single* `tar` file on ICON. When unpacked, the `tar` file must have the following directory structure:

⟨`your_HawkID`⟩
- `compile.sh`
- `task1.c`
- `task2.c`
- `task3.c`
- `task4.pdf`

You must write your code using the language C in .c files.

Task1 (10 Points)

Write a shell script named `compile.sh`. The purpose of this shell script is to generate an executable file of all the C programs that you will write to complete tasks 1,2, and 3. This `compile.sh` script must compile 1) the code in task1.c in a program called `task1, 2)` the code in `task2.c` in a program called `task2`, and 3) the code in `task3.c` in a program called `task3`

Task2 (15 Points)

Write a C program printing to stdout all 1) its arguments (including `argv[0]`), one per line, 2) its environment variables (again, one per line), 3) the parent process ID, and 4) the current process ID. 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 `task1.c`. If you run the code of this C program in the following way: `./task1 argument1 argument2`, the output should be similar to:

```
argument1
argument2
XDG VTNR=7
ORBIT SOCKETDIR=/tmp/shehroze
```

```
XDG SESSION ID=c1
VIRTUALENVWRAPPER SCRIPT=/usr/share/virtualenvwrapper/virtualenvwrapper.sh
. . . XAUTHORITY=/home/shehroze/.Xauthority
COLORTERM=gnome-terminal=./printargsandenv
OLDPWD=/home/shehroze/git
Parent Process ID: 5189
Child Process ID: 5190
```

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

Task3 (15 Points)

Write a C program that creates two child processes. In the first child process, stdout all its arguments (including `argv[0]`), one per line. In the second child process, stdout all its environment variables. On creation, also print that the child process has been created and print the process id of each child process. For example, If you run the code of this C program in the following way: `./task2 argument1 argument2`, the output should be similar to:

```
Child process created to print arguments:  The process id is:
3891.
argument 1
argument 2
Child process created to print Environement variables:  The
process id is:  3891.
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/shehroze
XDG_SESSION_ID=c1
VIRTUALENVWRAPPER_SCRIPT=/usr/share/virtualenvwrapper/virtualenvwrapper.sh
. . .
```

(I omitted many lines. Also, the values of your environment variables and process id will likely be different.) Save the source code of this program in `task2.c`.

Task4 - Your Custom Shell (30 Points)

This task requires you to write a C program similar to a Linux shell. 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. 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:
1) ⟨binary path⟩
2) ⟨arguments⟩
3) ⟨extra environment⟩
4) ⟨copy environment⟩
5) ⟨use path⟩
6) ⟨niceness⟩
7) ⟨wait⟩
8) ⟨timeout⟩
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.

Details of the fields:
`⟨binary path⟩`: This is the path (absolute or relative) of the program you need to execute.
`⟨arguments⟩`: Specify the arguments of the program you have to run.

⟨extra environment⟩: Specify some environment variables you should set for the executed program. The extra environment variables are separated by delimiter.
⟨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.
⟨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.
⟨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

Task5 (30 Points)

For each program, you need to add comments after every line. For printf function, you need to give the output of it; for pointers, you need to explain how they move and how the value of it changes.
Example:

```c
#include<stdio.h>
int main ()
{
    int* pointer; // create an integer pointer
    int var1 = 1; // create an integer pointer and initialize it
    pointer = &var1; // the pointer points to the address of var1
    *pointer = 11; // change the value o f var1 to 11
    printf(var); // the output is 11
    return 0;
}
```

1.

```c
#include<stdio.h>
int main ()
{
    char* ptr = "helloworld";
    printf(ptr + 4);
    return 0;
}
```

2.

```c
#include<stdio.h>
int main(){
    int* ptr;
    printf("%d", sizeof(ptr));
    return 0;
}
```

3.

```c
#include<stdio.h>
int main (){
    int* ptr = 2;
    printf("%d", sizeof(ptr));
    return 0;
```

```
}
```

4.

```
#include<stdio.h>
int main(){
    char* ptr;
    char string[ ] = "learn C from class";
    ptr = string;
    ptr += 6;
    printf ("%s", ptr);
    return 0 ;
}
```

5.

```
#include<stdio.h>
void function(char **);
int main ()
{
    char* arr [ ] = {"ant", "bat", "cat", "dog", "egg",
        "fly"} ;
    function(arr);
    return 0;
}
void function(char ** ptr)
{
    char* ptr1;
    ptr1 = (ptr += sizeof(int))[-2];
    printf("%s \n", ptr1);
}
```

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

Examples for Task3

You can find the files mentioned in this section in the tar file of the homework. Obviously, some output, such as the values of the different pids or file timestamps, may be different.

By running:
./task3 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:
./task3 ls2.txt

Your output should be:
New child process started 31340
ls: cannot access 'nonexistingfile': No such file or directory
Child process 31340 terminated with exit code 2

By running:
./task3 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:
./task3 printargsenv.txt
Your output should be:
New child process started 29317
./printargsandenv
100
var1Custom=CustomVal
var2Custom=CustomVal2
Child process 29317 terminated with exit code 0

By running:
./task3 sleeptimeout.txt ls2.txt
Your output should be:
New child process started 29593
New child process started 29596
ls: cannot access 'nonexistingfile': No such file or directory
Child process 29596 terminated with exit code 2
Child process 29593 terminated with exit code 143
The last line should be printed approximately 10 seconds after the others.