CS 446/646 – Principles of Operating Systems

Homework 1

Due date: Tuesday, 9/19/2023, 11:59 pm

Objectives: You will implement the general system process call API in C using **`fork`**, **`wait`**, and **`execvp`**. You will be able to describe how Unix implements general process execution of a child from a parent process. You will write a shell program in C so that you can see how child processes are created and destroyed & how they interact with parent processes.

General Instructions & Hints:

- All work should be your own.
- The code for this assignment must be in C. Global variables are not allowed and you must use function declarations (prototypes). Name files exactly as described in the documentation below. All functions should match the assignment descriptions. If instructions about parameters or return values are not given, you can use whatever parameters or return value you would like.
- There is a significant amount of documentation in this assignment that you must read; you are better off starting it sooner rather than later.
- All work must compile on **Ubuntu 18.04**. Use a **`Makefile`** to control the compilation of your code. The **`Makefile`** should have at least a default target that builds your application.
- To turn in, create a folder named **`PA1_Lastname_Firstname`** and store your **`Makefile`** and your **`.c`** source code file in it. Then compress the folder into a **`.zip`** or **`.tar.gz`** file with the same filename as the folder, and submit that on WebCampus.

Note: Notation such as **`<netid>`** indicates that you should replace the angled brackets and word with your version, e.g. John Doe would do: **`<netid>_code.c`** → **`jdoe_code.c`**

Part 1, the Process API Background:

Normally, when you log in, the OS will create a *User Process* that binds to the login port; this causes the *User Process* at that port to execute a *Shell*. A *Shell* (command line interpreter) allows the user to interact with the OS and the OS to respond to the user. The shell is a character-oriented interface that allows users to type characters terminated by Enter (the **`\n`** char). The OS will respond by echoing characters back to the screen. If the OS is using a GUI, the window manager software will take over any shell tasks, and the user can access them by using a screen display with a fixed number of characters and lines. We commonly call this display your "*Terminal*", "*Shell*", or "*Console*", and in Linux, it will output a *Prompt*.

The *Prompt* is usually **`userName@machineName`** followed by the terminal's ***Current Working Directory*** in the file system, and then a **`$`**.

Common commands in Linux use Bash, and usually take on the form of:
`command argument1 … argumentN`

For example, in: **`chmod u+x <filename>`**

- **`chmod`** is the command,
- **`u+x`** is an argument,
- **`<filename>`** is also argument.

Not all commands require arguments- for example, you can run **`ls`** with and without any arguments.

After entering a command at the prompt, the shell creates a *Child Process* to execute whatever command you provided. The following are the steps that the shell must take to be functional:

1. Print a *Prompt* and wait for input.
2. Get the command line input.
3. Parse the command line input.
4. Find any associated files.
5. Pass any arguments from the shell to the OS system call (man 2) / C-library function (man 3).
6. Execute the command (with applicable arguments).

General Directions:

The program will represent a very stripped down shell that uses a very stripped down process API. Name your program **`simpleshell.c`** . You will turn in C code for this. You will need to write a minimum of 4 functions: **`main`**, **`parseInput`**, **`changeDirectories`**, and **`executeCommand`**. For each function, the name, edge cases and exceptions that you must address, and description of functionality are provided. You may implement additional functions as you see fit.
Overall, you will work with / implement:

- An **`executeCommand`** function which will use the [**`fork`**](https://linux.die.net/man/3/fork) system call and [**`execvp`**](https://linux.die.net/man/3/execvp) C-library function to launch a new *Child Processes* & replace the forked *Child Process* image with a one that corresponds to "running" a specific system command (executable) / program.
- Calling **`cd`** (to change directory on your simpleshell) which will be handled via a direct OS system call at the *Parent Process* (as there is no **`cd`** system command (executable) to **`fork`** & **`execvp`**, and generally a forked *Child Process* cannot –unless special arrangements are made– alter the variables –including the *Current Working Directory*– of the *Parent Process* (http://www.faqs.org/faqs/unix-faq/faq/part2/section-8.html))
- Calling **`exit`** (to cause normal process termination of your simpleshell) which is also handled via a direct C-library function call at the *Parent Process* (when we **`execvp`** we are –for all intents and purposes– causing the current process to *die* and be replaced by the new one, so it would be redundant to have an **`exit`** program just for that, while when making such a call we would also be throwing away the original *Process*' variables which may be storing information relevant to the exit status).

Generally speaking, the program will execute a simplified shell ("simpleshell") and loop until the user chooses to. This simpleshell will be interactive- that is, you will run the program (**`./<exename>`**) to launch a new shell (just like if you had opened a terminal). Then you should be able to type into the shell either: **`exit`**, **`cd`**, or various **`execvp`**-launchable commands (such as **`ls -la`** or **`clear`**), and have it perform the associated behavior.

You may only use the following libraries, and not all of them are necessary:

```
<stdio.h>
<string.h>
<stdlib.h>
<sys/wait.h>
<sys/types.h>
<unistd.h>
<fcntl.h>
<errno.h>
<sys/stat.h>
```

Any necessary display can be done using the [**write()**](https://linux.die.net/man/2/write) system call or the [**printf()**](https://linux.die.net/man/3/printf) library function. Any reading can be done using [**fgets()**](https://linux.die.net/man/3/fgets) library function and/or [**scanf()**](https://linux.die.net/man/3/scanf) library function. You may use any appropriate read/write function so long as you don't import any additional libraries. The man pages for any appropriate system calls / library functions are provided online (linked above). Note that if you are asked to use a specific system call / library function (like **execvp**), then you must use that one, and not any alternatives.

- Required functions:

parseInput :
This function should take the command entered by the user and stored in **main**. It should split the command on any spaces so that the first word (the command) can be gathered separately from the **char* array** containing the user's entry. This can be accomplished using [**strtok()**](https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm).

main :
The program **main** function should get the Command Line Input (CLI), and then it should display the *Prompt* **<netid>:<Current_Working_Directory>$** (where you replace **<netid>** with your actual netid, and the **<Current_Working_Directory>** with a C-string filled out by the [**getcwd**](https://man7.org/linux/man-pages/man2/getcwd.2.html) C-library function call.

The **main** function should gather the user's CLI string, and pass it to **parseInput** so that you can split it into the command and any flags and/or arguments as separate C-strings (i.e. for an **ls -al** CLI string you want **ls** to be a separate **char*** and the **-la** to be a separate **char*** as well). You may assume that command & flags/arguments will always come separated by whitespaces.

You will need to implement an array of C-strings (**char****) to hold the above information (a statically-allocated 2D **char** array is acceptable):
a) The first C-string element of that array should be the command to execute.
b) Every subsequent C-string element of that array will be the corresponding whitespace delimited substrings that follow the command, and represent flags and/or arguments.
c) The final C-string element after the command & flags/arguments should be a pure **NULL** C-string (to indicate the end of any function call arguments).
Note: The above correspond to the call interface of [**execvp**](https://linux.die.net/man/3/execvp).
The return value of **execvp** will be -1 if it wasn't successful (and it will also set the [**errno**](https://linux.die.net/man/3/errno) to indicate the type of the last error that occurred).
You can (/should) read up more on **execvp** on its (online) manpages that are linked above.

Your **main** should check the first element of the filled C-string array. You can use the [**strcmp**](https://www.cplusplus.com/reference/cstring/strcmp/) function to check the CLI substring elements entered by the user.
If the element is anything other than **cd** (change directory) or **exit** (normally terminate current process), then **executeCommand** should be called, passing it also the array of C-strings containing the command-&-flags/arguments. The return value of the **executeCommand** function should be used to determine if the call was successful (non-0 return values indicate some error status in Unix
If instead it is **cd**, then **changeDirectories** should be called with appropriate arguments.
If finally it is **exit**, then it should stop looping and **return** from main with a 0 return value.

`executeCommand` :
This function should [**`fork`**](https://linux.die.net/man/3/fork) a *Child Process* for the CLI-provided command & any arguments.
It should subsequently use the [**`execvp`**](https://linux.die.net/man/3/execvp) C-library function to execute the CLI-provided system command (executable) as a *Process*. It should also pass to the command (e.g. **`ls`**) the user-provided arguments (e.g. **`ls -al`**) as described by **`execvp`**'s manpage.
The **`fork`** call should be checked to see if the *Child Process* was successfully created, through its return value which will be set to -1 (returned to *Parent*) on failure (and it will as well also set the **`errno`** to indicate the type of the last error that occurred).
The return value of **`execvp`** should be stored to check for errors.
Finally, after checking for errors, [**`wait`**](https://linux.die.net/man/3/wait) should be used to wait on the *Child Process* which was taken over by **`execvp`** to finish, before the *Parent Process* can move on.
This function should finally indicate success or failure by using its return value.

Edge Cases: If a process is not successfully forked, your function should print: **`fork Failed:`** and then append the result of the [**`strerror`**](https://linux.die.net/man/3/strerror) C-library function to explain the error based on the **`errno`** (e.g. **`strerror(errno)`**).

`changeDirectories` :
As explained above, there is no **`cd`** system command / program (executable), it is instead a shell built-in.
This function (**`changeDirectories`**) will use [**`chdir`**](https://linux.die.net/man/2/chdir), which should be supplied by the desired path input on the CLI (after **`cd`** in command string, so the path to change into a directory named **`dir1`** from your relative location would be: **`./here/`** and the full CLI entry would be: **`cd ./dir1/`**). The return value will be -1 on error (and it will as well also set the **`errno`** to indicate the type of the last error that occurred).
If successful, the *Current Working Directory* location should change.
Otherwise, you have to output **`chdir Failed:`** and then append the result of the **`strerror`** C-library function to explain the error based on the **`errno`** (e.g. **`strerror(errno)`**).

Edge Cases: Normally, if you enter **`cd`** without any path in a Linux environment, the shell will change directory to the home directory. For this exercise, you do not need to have the shell do this. Instead, you should check the number of provided arguments, and if you have **`cd`** but no path, or too many trailing arguments, you should display **`Path Not Formatted Correctly!`**

Submission Directions:
When you are done, create a directory named **`PA1_Lastname_Firstname`**, place your **`Makefile`** (which has to have at least one default target that builds your **`simpleshell`** application executable) and your **`simpleshell.c`** source code file into it, and then compress it into a **`.zip`** or **`.tar.gz`** with the same filename as the folder.
Upload the archive (compressed) file on Webcampus.

Early/Late Submission: You can submit as many times as you would like between now and the due date.
A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects that are up-to 24 hours late will receive a 15% penalty, ones that are up-to 48 hours late will receive a 35% penalty, ones that are up-to 72 hours late will receive a 60% penalty, and anything turned in 72 hrs after the due date will receive a 0.

Verify your Work:
You will be using **`gcc`** to compile your code. Use the **`-Wall`** switch to ensure that all warnings are displayed. Strive to minimize / completely remove any compiler warnings; even if you don't warnings will be a valuable indicator to start looking for sources of observed (or hidden/possible) runtime errors, which might also occur during grading.

After you upload your archive file, re-download it from WebCampus. Extract it, build it (e.g. run **`make`**) and verify that it compiles and runs on the ECC / WPEB systems.

- Code that does not compile will receive an automatic 0.