CS 446/646 – Principles of Operating Systems

Homework 1

Due date: Thursday, 9/19/2024, 11:59 pm

Objectives: You will implement the general system process call API in C using **`fork`**, **`wait`**, **`execvp`**, and **`dup2`**. You will be able to describe how Unix implements general process execution of a Child Process 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, as well as how shell redirection is implemented.

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 its actual value, e.g. for John Doe: **`cd /home/<netid>`** → **`cd /home/jdoe`**

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 into an array of C-strings.
4. Interpret the parsed command line input array elements into:
    a. a command C-string
    b. if applicable its additional arguments C-strings

   and pass them all to the appropriate OS system call (exec) that will execute this command.

It is also possible that shell redirection is desired for a given command. Shell redirection may involve replacing the standard **`stdin`** or **`stdout`** (or **`stderr`**) functionality for the calling *Process* with a *File Descriptor*, so that we can e.g. a) accomplish *Output Redirection* (with the **>** character) of a command's "normal" **`stdout`** (i.e. what you would normally see printed to the screen) to a file, or b) *Input Redirection* (with the **<** character) from a file to the command's "normal" **`stdin`** (i.e. input arguments that you would normally be providing from the keyboard.

For example we can do: **`ls -al >`** <**`outfilename`**>
Note: The standalone **>** character represents output redirection, while the **<**...**>** enclosed name is the desired filename that we wish to redirect the *Process*' **`stdout`** to.
So in this case:

- **`ls`** is the command,
- **`-al`** is its argument (modifier for printing additional information),
- **>** means that for the command: **`ls -al`** , we want to redirect its "normal" **`stdout`** output
- <**`outfilename`**> is where we want the redirected output to go.

Similarly we can do: **`cat -e <`** <**`infilename`**>
Note: The standalone **<** character represents input redirection, while the **<**...**>** enclosed name is the desired filename from which we wish to redirect to the *Process*' **`stdin`**.
So in this case:

- **`cat`** is the command,
- **`-e`** is its argument (modifier for printing special characters),
- **<** means that for the command: **`cat -e`** , we want to redirect to its "normal" **`stdin`** input
- <**`infilename`**> is where we want the redirected input to come from.

So in the case where we also have shell redirection, the following are the steps that the shell must take to be functional (effectively we need one additional step, #4):

1. Print a *Prompt* and wait for input.
2. Get the command line input.
3. Parse the command line input into an array of C-strings.
4. Inspect the resulting parsed command line input array to see whether input/output redirection is desired, and set it up by using the appropriate OS system call (dup2).
5. Interpret the remaining parsed command line input array elements into:
    a. a command C-string
    b. if applicable its additional arguments C-strings

   and pass them all to the appropriate OS system call (exec) that will execute this command.

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 first launch a new *Child Processes*, and then to replace the forked *Child Process* image with a one that corresponds to "running" a specific system command (executable) / program.
  For the *Child Process*, it will also take care of setting up *Input/Output Redirection* if desired by using [**`dup2`**](https://linux.die.net/man/3/dup2) before invoking **`execvp`**, so that the *Child Process*' "normal" **`stdin`**/**`stdout`** are appropriately redirected once the new program / command (executable) takes over this *Process*.
- 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 be *completely replaced* by the execution of new program, 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, your code will implement 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`**), with optional input/output redirection with **`<`** or **`>`**, 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:

```
int parseInput(char * input,
               char splitWords[][500],
               int maxWords);
```

This function should take the Command Line Input (CLI) entered by the user as the **input** C-string, parse and split it into words (assuming they are delimited by whitespace characters), and store the resulting words into the **splitWords** array of C-strings. This can be accomplished using [**strtok()**](https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm). The extra parameter **maxWords** specifies the size of the provided array of C-strings (**splitWords**).

Returns: The number of valid CLI elements (split words) that have been copied (using [**strcpy**](https://www.cplusplus.com/reference/cstring/strcpy/) to the **splitWords** C-string array elements.

main :
The program **main** function should get the 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 C-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 be separated by whitespace characters.

You will need to implement a local variable that is an array of C-strings, to hold the above information (working with statically-allocated 2D **char** array is acceptable). This will be provided to **parseInput** to store the resulting parsed CLI elements.

a. 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 first element is **cd**, then **changeDirectories** should be called with appropriate arguments.
b. If the first element is **exit**, then it should stop looping and **return** from main with a 0 return value.
c. If the first element is anything else, then this means it is a command / (executable) program that should be executed in a *Child Process* (your shell program will be the *Parent Process*). In such a case, your C-string array should first be checked whether it contains the input redirection (**<**) or output redirection (**>**) special character.
   i. If it doesn't, then this means that all the valid elements of the C-string array are a CLI command (the first element) and its arguments (all the rest of the valid elements), and these should be provided in a properly formatted structure to **executeCommand**.
      To implement the necessary structure you will need to implement a local variable that is a *Dynamically Allocated* C-string array (**char * ***). Its size should be **one element more** than the number of valid CLI elements, and:
      I. The first C-string element will represent the CLI command / (executable) program

to execute. This actually the first element of your parsed/split words C-string array.

II. Every subsequent C-string element will be the individual CLI arguments and/or flags for that command / (executable) program. These are actually all the remaining valid elements of your parsed/split words C-string array.

III. The final C-string element should be a pure & simple **`NULL`** pointer (this is used to indicate the end of any CLI command arguments).

Note: The above structure actually corresponds to the call interface of [**`execvp`**](https://linux.die.net/man/3/execvp) which will be used in **`executeCommand`**.

ii. If it does, then it should first copy the element that follows the redirection character (i.e. is "after" it in the parsed/split words C-string array) into a different local C-string variable **`infile`** or **`outfile`** (depending an input or output redirection character was encountered).

Then it should treat every element of the parsed/split words C-string array **up to (and excluding) the redirection character** as a CLI command and its arguments. This means that the exact same process for creating the structure to hold that information (that will be provided to **`executeCommand`**) as described in **c.-i**. above can be followed. The only thing that has changed is how many of the parsed/split words C-string array valid elements we are keeping.

For example: **`ls -al > myfile.txt`** which has 4 whitespace-delimited elements will result in 4 valid elements in the parsed/split words C-string array, but because an output redirection character (**`>`**) exists, we will copy the C-string following it (**`myfile.txt`**) into our **`outfile`** local variable and keep the first 2 valid elements (**`ls`** and **`-al`**) to create a *Dynamically Allocated* C-string array of size 3 (=2+1), where the 1st element will be the command C-string (**`ls`**), the 2nd element will an arguments/flags C-string (**`-al`**), and the 3rd element will be a **`NULL`** pointer.

```
int executeCommand(char * const* enteredCommand,
                   const char* infile,
                   const char* outfile);
```

This function accepts the *Dynamically Allocated* C-string array of the CLI command / (executable) program and its arguments/flags that was created in **`main`**, as well as an **`infile`** and an **`outfile`** C-string that represent the input / output redirection filenames (if desired) for the *Child Process* that will be created.

It should first [**`fork`**](https://linux.die.net/man/3/fork) a *Child Process* for the CLI-provided command & any arguments. The call to **`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).

If it succeeds, then at this point you will have 2 *Processes*, the *Parent Process* and the *Child Process*, and you can disambiguate between them based on the return value of **`fork`**):

a. If you are within the *Child Process*, the function you should:

i. Check whether input or output redirection is desired by inspecting the **`infile`** and **`outfile`** C-strings. If one of them (for simplicity we can assume only input or output redirection each time) is valid (non-**`NULL`** pointer and not empty strings i.e. **`""`**), then it should used the [**`dup2`**](https://linux.die.net/man/3/dup2) C-library function to it up. **`dup2`** is used to close an open file descriptor (its second argument) and create a copy of it into another open file descriptor (its first argument).

So we can perform redirection of the "normal" **`stdout`** by doing:

```
int fd = open(outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
dup2(fd, STDOUT_FILENO); //STDOUT_FILENO is 1
```

or redirection of the "normal" **stdin** by doing:

```
int fd = open(infile, O_RDONLY, 0666);
dup2(fd, STDIN_FILENO); //STDIN_FILENO is 0
```

ii. 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.
Also, if **execvp** returns, it means that an error has occurred. In this case, and since this is performed by the *Child Process*, the *Child Process* should be terminated "immediately" (after printing out the error message described below) using the appropriate exit call variant, i.e. [**_exit**](https://linux.die.net/man/2/_exit). It is noted that the return value of **execvp** (i.e. if it wasn't successful) will be **-1**, and it will also set the [**errno**](https://linux.die.net/man/3/errno) to indicate the type of the last error that occurred). Again, **execvp** will only return if an error has occurred. You can (/should) read up more on **execvp** on its (online) manpages that are linked above.

b. If you are within the *Parent Process*, the function should (after checking for errors) use [**wait**](https://linux.die.net/man/3/wait), to wait on the *Child Process* which was taken over by **execvp** to finish, before the *Parent Process* can move on.
The *Child Process'* exit status retrieved by **wait** should be used inform the user whether it failed via a message: **Child finished with error status: <status>**

Returns: An integer value to indicate success (**0**) or failure (non-**0**).

Edge Cases: If a *Process* is not successfully forked or exec fails, your function should print: **fork Failed:** or **exec Failed:** respectively, 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)**).

```
void changeDirectories(const char * path);
```

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)**).

Returns: Nothing.

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.