CSC-641
Operating Systems
Programming Project 1
FALL 2024

Background

Simple Shell
The Linux shell is a command interpreter. It allows users to execute commands without knowledge of the internals of the system. A shell can execute commands directly from the terminal input window or it can read shell commands from a file. These files are known as shell script files.

There are many shells available to the user. Some of the most common ones are: `sh`, `bash` and `csh`. They very much perform the same functions and differ basically in the way the can be programmed through script files. This project concentrates on command line input to the shell.

Commands from the terminal window are read by the shell and executed on behalf of the user. To execute a command the shell creates a process to execute the code of the particular command. For example to execute the `ls` command, the shell will create a process to execute the code for `ls`. The shell will wait for the command to terminate and then will prompt the user for another command. Output for the command goest to the standard output file. The shell uses three predefined files for input, output and error. The `stdin`, `stdout` and `stderr`, for input, output end error messages respectively. By default, all files are associated with the terminal window.

Description
To execute a command the shell requires three systems calls, `fork`, `exec` and `wait`. The `fork()` system call is used to create a process. `exec()` is used to overwrite the code of the child process with the code of the command to be executed, and the `wait()` is used to wait for the child process to terminate. This way the child process does not have to execute the same code as the parent process. The `exec()` has many variations, `execl`, `execv` ,`execle`, `execve`, `execlp` and `execvp`. The difference is in the way the arguments are sent.

When `exec()` is executed, the program file given by the first argument will be loaded into the caller's address space and over-write the program there. Then, the arguments will be provided to the program and execution starts. Once the specified program file starts its execution, the original program in the caller's address space is gone and is replaced by the new program. `exec()` returns a negative value if the execution fails. The most common failure is that the file could not be found .

Program Guidelines

The purpose of this programming project is to write a simple shell program. Your shell program should use the same style as the `Bourne` shell for running programs. In particular, when the user types a line such as:

```
command  [identifier[identifier]]
```

Your shell should parse the command line to build the `argv` array, with the following structure shown in Figure 1.

If you type the following

```
%./a.out Hello World
```

as input to your shell, a data structure with the following structure must be created, since this will be the argument to the program what the shell calls `exec()` to execute your request.

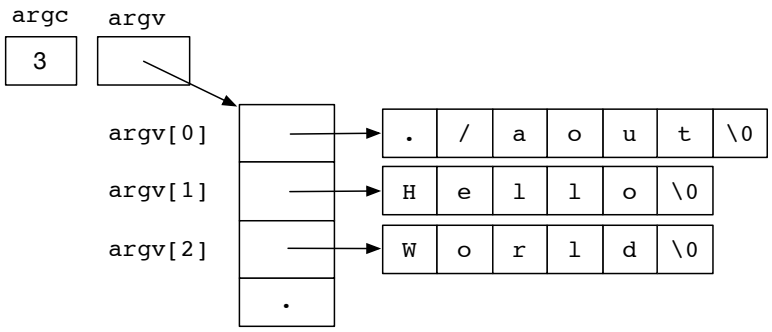

Figure 1

Notice that each argument is terminated by the `\0` character. All C strings terminate with a `\0`.

Your shell will `fork` a process. The child process will execute the `execvp()` system call to execute the file that contains the command. This way the `exec()` will search for the file in the paths specified by the PATH shell variable.

The shell must also interpret the "&" operator as a command terminator. A command terminated with "&" should be executed concurrently with the shell rather than the shell waiting for the command to terminate before it prompts the user for another command (background execution).

If a command is given that `execvp()` cannot execute (such as an erroneous command), an appropriate error message must be displayed.

The shell is in a loop, prompting the user for input and executing the command. The `exit` command is used to exit the shell. It should be the only token in the command line.

Your program should **not** terminate with a ^C signal. That signal should be ignored by the shell.