CS3.306 Algorithms and Operating Systems **Due: October 25, 2021**

Assignment #03 - Sprint #02

Basics of Shell and Signal Handling — Programming Assignment

Instructions: Read all the instructions below carefully before you start working on the assignment, and before you make a submission.

- Start your work early and do not wait till the deadline. The assignment will take a fair amount of time to read and understand. Assignments are due at **23:55 hours Indian Standard Time** on the due date.
- The submission will be through GitHub Classroom and the link to the assignment will be available on *courses.iiit.ac.in* shortly. The supplementary files and instructions are available in your repository.
- You are **encouraged to discuss the assignment with classmates**. You must understand the solution well enough in order to reconstruct it by yourself. Your final submission should be your own work.
- The institute's Academic Code of Conduct will be strictly enforced. **There will be strict plagiarism checking and any breach of the rule could result in serious actions**.
- *You must use C language. You may use C++ for the string-handling parts of the assignment.*
- Please reach out to any of the teaching assistants of the course if you require any help. You are highly encouraged to attend the TA Office Hours and get your doubts clarified.

Exercise One: Introduction to UNIX Signals Programming

What's is the right way to terminate the program? What are the ways to receive notifications from operating system about events that occur. Traditional Unix systems have the answers ready. The answer to these questions is signals. Signal Handling in Linux blog tries to explain what signals are, their nature and talks about the right ways to handle signals, what signals to handle and the pitfalls of signal handling in Linux in particular.

Problems

First and foremost, read and understand the Signal Handling in Linux blog referenced above.

1. The program `program01.c` is an example of handling `SIGINT` and `SIGTERM` signals. Starter code and example output is provided. The process should print a custom message asking a question about whether the program should really exit, and exit only on confirmation.
2. Let's learn more about signals. `example01.c` is a program that spawns number of child processes using the following command `./example01 n` where `n` is the number of child processes. Each child sleeps for a random duration and exits. We override the `SIGCHLD` handler in the parent process and print the sequence of exits of the child processes. Understand the code thoroughly, and write a shell script to run the program for any four different `n` of your choice. Redirect the output of the program and append it to `output01.txt` for the different runs. You have to submit the shell script and the output file.
3. *What would happen if a child process forked another (grand) child process, which eventually terminated.* Execute `example02.c`, check the output and reason about it. Two situations to test—when the grandchild process terminates before the child and when the child terminates before the grandchild. Recreate these situations, and submit one output file of each case - `output_a.txt` and `output_b.txt`. Write your observations and reasoning as part of the submission in `answer02.txt`.

4. Program sources by themselves don't make an application. The way you put them together and package them for distribution matters, too. If you're developing in C or C++, an important part of the recipe for building your application will be the collection of compilation and linkage commands needed to get from your sources to working binaries. Entering these commands is a lot of tedious detail work, and most modern development environments include a way to put them in command files or databases that can automatically be re-executed to build your application. Unix's `make` program, the original of all these facilities, was designed specifically to help C programmers manage these recipes. It lets you write down the dependencies between files in a project in one or more 'makefiles'. Each `makefile` consists of a series of productions; each one tells make that some given target file depends on some set of source files, and says what to do if any of the sources are newer than the target. You don't actually have to write down all dependencies, as the make program can deduce a lot of the obvious ones from filenames and extensions.

   Watch this short video on How to Create a Simple Makefile - Introduction to Makefiles and create one for working with the three C programs given to you. You may refer to this tutorial as well.

Exercise Two: Fundamentals of Operating System Shells

The outermost layer of the operating system is called the shell. It's just another computer program. In UNIX based systems, the shell is generally a command line interface. Most Linux distributions ship with `bash` as the default. There are several others: `csh`, `ksh`, `sh`, `tcsh`, `zsh`. Shell is the interface to the operating system. Shells incorporate a programming language to control processes and files, as well as to start and control other programs. The shell manages the interaction between you and the operating system by prompting you for input, interpreting that input for the operating system, and then handling any resulting output from the operating system.

Shells provide a way for you to communicate with the operating system. This communication is carried out either interactively (input from the keyboard is acted upon immediately) or as a shell script. A shell script is a sequence of shell and operating system commands that is stored in a file. When you log in to the system, the system locates the name of a shell program to execute. After it is executed, the shell displays a command prompt. This prompt is usually a $ (dollar sign). When you type a command at the prompt and press the Enter key, the shell evaluates the command and attempts to carry it out. Depending on your command instructions, the shell writes the command output to the screen or redirects the output. It then returns the command prompt and waits for you to type another command.

A command line is the line on which you type. It contains the shell prompt. The shell considers the first word of a command line (up to the first blank space) as the command and all subsequent words as arguments.

The shell lives in user space, along with most other programs, as opposed to in kernel space which is where the kernel lives. Software living in kernel space can execute privileged instructions, such as dealing directly with hardware. We don't want any software to be able to do this, as it could overwrite the operating system itself. The way the shell talks to the kernel is by system calls. These system calls allows the user to do things like open files and create processes. Since software in user space always have to go through the kernel to perform such operations, the kernel can make sure the shell does not do anything it does not want to allow. Note that this is different from a super-user or running as root, which is about user privilege that software in user space have.

How does a shell execute a simple command?

If you want to see a list of what's inside the directory in your present working directory you can use the program `ls`. Typing `ls` in a shell and pressing enter runs it and returns the contents of that folder. When you press enter the shell first parses what you wrote into some internal representation. What does it mean to run `ls`? To answer that we must first understand what the fork system call does and what a child process is.

Recall that an operating systems allows several programs to run on one computer, and processes are a big part of how it does that. A process in Unix-like systems is a program that runs and has access to its own piece of memory which contains the program's instructions, data and stack. The operating system then makes sure each process gets to do what it wants to do in some reasonable manner. Fork is a system call that allows a process to create another process. The process calling fork is called the parent process, and the process it creates is called the child process. When a child process starts, its memory is initally almost an exact, but separate, copy of its parent process's memory.

In the case of our `ls` command, the shell runs the parsed command in a forked child process. In C code it would look something like this:

```
int pid = fork();
if(pid == 0)
  runcmd(parsecmd(buf));
wait();
```

When we call fork we create a child process, and we get back an integer that we call pid for process id. We now have two different processes running independently, and the variable pid is different in the two. In the parent process, the pid is some number that is used to uniquely identify the process, whereas in the child process it's simply equal to 0.

A good way to think about the above piece of code is to see it as being two different programs. In the parent process, the if-statement returns false and it gets stuck at wait, which is another system call that just waits until a child process is finished. The kernel makes sure the parent is notified when that happens. In the child process, the if-statement returns true and it runs the command. This executes `ls` and gives over all control to `ls` for that process. Once the child process has finished running - and when that happens is completely up to `ls` and how it is implemented - the parent process, i.e. the shell, will resume running, and we can type another command.

How I/O redirection works?

Let's say we want to save the output from running `ls` above. We can do so using something called I/O redirection (I/O stands for input/output): `ls > foo`

There are three parts to this command. When the shell parses the command, it figures out that it's a redirection from `ls` to `foo`. After the shell has forked to a child process it runs `ls` and saves the output in a file called `foo`. An ordinary file contains either symbolic or binary data, is written in some format, and has some metadata associated with it (such as who is allowed to read and write to it), and we can access it by using its pathname. One of the most common type of file is a text file, which contains a string of characters. There are special files too, and in fact even devices and directories are represented as files.

How does the output of `ls` end up in foo? To understand that we have to know a bit more about files work in Unix. The system call open is used to see the contents of a file. When we open a file to read or write to it, we get a file descriptor back. This file descriptor is just an integer that represents a specific file that a process can read or write to. There are three special integers, 0, 1, and 2. These are called, in order, STDIN (standard input), STDOUT (standard output), and STDERR (standard error). Another word for these is standard streams. By default, when the shell reads something, such as a command you typed in, it does this from STDIN. Likewise, when the shell prints something, such as the result of running some command, it does this to STDOUT.

There is some confusion about the difference between files and streams, and people can mean different things when they talk about them. For our present purposes, we can treat them as equivalent - as long we let go of our preconceptions of what a file is. As alluded to before, many things are seen as as files from the kernel's point of view in UNIX-like systems.

When we run `ls`, it returns the result by printing it to STDOUT, and STDOUT is what we see in the shell. File descriptors are handed out by the kernel starting from the lowest available file descriptor, and when the shell starts it opens the three standard streams. We can use this fact to get I/O redirection with something like this:

```
close(fd);
open(file, mode);
runcmd(cmd);
```

This is a simplified version of the actual code and it has no error handling. The second argument to open is mode, which is where we say if we want to read or write to the file. Assuming we want to redirect using >, we close STDOUT. When we then open the file, foo in this case, to write to it, it will pick the lowest available file descriptor, which is 1. When we then run the command `ls`, it will print to STDOUT - which is bound to our file `foo`.

How pipes allow for inter-process communication?

Let's look at another example. There's a program called `ps` that shows the status of processes. If we want to have a list of all my processes sorted by their process id we can pipe the result of running `ps` to the program sort: `ps | sort`

When the shell parses this command it sees the symbol "`|`" and knows it's a pipe command. A pipe command has two sides: a left and a right side. When the program on the left side writes to STDOUT it can be read from the program on the right side through STDIN. When the shell parses this command it sees the symbol "`|`" and knows it's a pipe command. A pipe command has two sides: a left and a right side. When the program on the left side writes to STDOUT it can be read from the program on the right side through STDIN. that lives in kernel space and allows processes to talk to each other, which is called inter-process communication. This communication happens continuously as new data is written to the pipe. It's also a queue, so even if new data comes in faster than you can process it in the right process the data doesn't disappear, and it doesn't block either. How does this work? Let's look at the code.

```
int p[2];
pipe(p);

if(fork() == 0) {
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
  runcmd(left);
}
if(fork() == 0) {
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(right);
}
close(p[0]);
close(p[1]);
wait();
wait();
```

We create an array of two integers, which is where we will keep track of our file descriptors. We then use the system call `pipe`, which creates a pipe between two file descriptors and and puts these in `p`, where `p[0]` is for reading and `p[1]` is for writing. After that, we create two child processes - one for the left process and one for the right one. These are the two `if`-blocks that check if `fork` returns 0, which it does in the child processes.

In the left process we close STDOUT. Then we use another system call `dup` that duplicates a file descriptor. What this means is that we can refer to the same file or stream but using a different file descriptor. Since STDOUT is closed and it's the lowest available file descriptor, the write-end of our pipe gets connected to STDOUT in this process.

Recall that a child process has almost exactly the same memory as a parent process. This includes file descriptors, so after we have connected the left process to STDOUT we want to close the file descriptors in `p`. If we don't, we might get a deadlock where we don't ever see anything printed. For example, if we forget to close the write-end of the pipe (`p[1]`) in the right child process, the read-end of the pipe (`p[0]`) will keep waiting for data. This means that the left child process won't ever finish, and the the parent process will wait forever. It's only when the write-end of a pipe is closed that the read-end stops waiting. This is similar to how ordinary files have a specific end-of-file character that tells us when a file is finished.

Depending on the exact command, things might still work out fine even if you forget to close a file descriptor. However, in order to avoid subtle bugs, consider it good hygiene to close a file descriptor when you are done with it. After all that, we run the left process and it prints to STDOUT. Similarly, the right process does almost the exact same thing but for STDIN. And finally, the parent closes the file descriptors for the pipe and waits for both child processes to finish (which, if it's a long-running process, may never happen).

Problems

Write a program `turtle.c` that implements a simple command shell for Linux. Sample code `make-tokens.c` is provided to tokenize strings (to deal with user commands); you may reuse this as part of your shell. Use any creative message as a prompt of the shell waiting for user commands. Below are the the commands you need to implement in the shell, and the expected behavior of the shell for that command.

Note: You must use the `fork` and `exec` system calls to implement the shell. The idea is for the main program to act as a parent process that accepts and parses commands and then instantiates child processes to execute the desired commands.

You must **not** use the system function provided by the C-library. Also, you must execute Linux system programs wherever possible, instead of re-implementing functionality. For example: Given the following command `echo "The assignment is due on October 25"` you should use the `echo` binary available rather than implementing `echo`.

The following functionality should be supported:

- `cd directory-name` must cause the shell process to change its working directory. This command should take one and only one argument; an incorrect number of arguments (e.g., commands such as `cd`, `cd dir1 dir2` etc.) should print an error in the shell. `cd` should also return an error if the change directory command cannot be executed (e.g., because the directory does not exist). For this, and all commands below, incorrect number of arguments or incorrect command format should print an error in the shell. After such errors are printed by the shell, the shell should not crash. It must simply move on and prompt the user for the next command.
- All simple standalone built-in commands of Linux e.g., (`ls`, `cat`, `echo`, `sleep`) should be executed, as they would be in a regular shell. All such commands should execute in the foreground, and the shell should wait for the command to finish before prompting the user for the next command. Any errors returned during the execution of these commands must be displayed in the shell.
- Any simple Linux command followed by the output redirector '>' (e.g., `echo hi > hi.txt`) should cause the output of the command to be redirected to the specified output file. The command should execute in the foreground. An incorrect command format must print an error and prompt for the next command. *Note: This can achieved by manipulating the sequence of file open and close actions, and/or using the dup system call.*
- Any list of simple Linux commands separated by "`;;`" (e.g., `sleep 100 ;; echo hi ;; ls -l`) should all be executed in the foreground and sequentially one after the other. The shell should start the execution of the first command, and proceed to the next one only after the previous command completes (successfully or unsuccessfully). The shell should prompt the user for the next input only after all the commands have finished execution. An error in one of the commands should simply cause the shell to move on to the next command in the sequence. An incorrect command format must print an error and prompt for the next command.

Important Guidelines

- When a process completes its execution, all of the memory and resources associated with it are de-allocated so they can be used by other processes. This cleanup has to be done by the parent of the process and is called *reaping the child process*. The shell must also carefully reap all its children that have terminated. For commands that must run in the foreground, the shell must wait for and reap its terminated foreground child processes before it prompts the user for the next input.
- By carefully reaping all children (foreground and background), the shell must ensure that it does not leave behind any zombies or orphans when it exits.
- You must implement all the commands above in your shell. Test your shell using several test cases.

Key Tips

- You are given a sample code `make-tokens.c` that takes a string of input, and "tokenizes" it (i.e., separates it into space-separated commands). You may find it useful to split the user's input string into individual commands.
- You may assume that the input command has no more than 1024 characters, and no more than 64 "tokens". Further, you may assume that each token is no longer than 64 characters.
- You may want to build a simple shell that runs simple Linux built-in commands first, before you go on to tackle the more complicated cases.
- You will find the `dup` system call and its variants useful in implementing I/O redirection and pipes. When you `dup` a file descriptor, be careful to close all unused, extra copies of the file descriptor. Also recall that child processes will inherit copies of the parent file descriptors, so be careful and close extra copies of inherited file descriptors also. Otherwise, your I/O redirection and pipe implementations will not work correctly.
- You must catch the `SIGINT` signal in your shell and handle it correctly, so that your shell does not terminate on a Ctrl+C, but only on receiving the exit command.
- You will find the `chdir` system call useful to implement the `cd` command.
- Carefully handle all error cases listed above for each command. For example, an incorrect command string should always print an error.

Note: We have provided the implementation of two different commands - `dup` and `pipe` - to understand and appreciate the shell functionalities better.

Acknowledgements

The reading material for Exercise Two is borrowed from Oskar Thorén's *What is a shell and how does it work?* blog and IBM's documentation on Operating system shells.