Operating Systems Lab (CS-UH 3010)

Spring 2024

Assignment 3: Extending Shell Functionality

Deadline: February 22, 2024

0. Overview

Pipes and redirects are crucial to inter process communication. A *pipe* is effectively a redirection of the standard output of one command to the standard input of the following command, it is denoted by the use of a vertical slash (‘|’) symbol.

A redirect is used to direct the output of a command to a file or direct input from a file to the standard input of a program.

In a shell, this can be done most effectively by duplicating *streams*. A stream is a generic, catch-all term used to describe the flow of data to its destination. Each data stream is associated with a *file descriptor*. A file descriptor is an integer number used to identify a data stream. The standard data streams we are all familiar with are:

- standard input stream:	file descriptor 0
- standard output stream:	file descriptor 1
- standard error stream:	file descriptor 2

0.1 Running an External Command

A shell is basically a program that runs other programs, simple as that. Typically invoking the *exec()* system call or some variant, the currently running process switches its own code with the code of the invoked program. Essentially what will happen is:

1. Shell’s process calls exec(**path to user program, argument vector**)
2. Shell’s code and virtual memory is replaced with that of the user program
3. User program executes until it exits
4. Since the shell’s code was replaced with the user program’s code, you don’t drop back into the shell, you just exit.

So how does one rectify this?

0.2 Forks and Child Processes

If you are making a shell, you can solve this issue by having your shell process create a ‘copy’ of itself and run it as a separate process, this copy will then execute the command and exit, and the ‘original’ will just wait on the copy to finish executing. Once it finishes, the copy exits and thus terminates, and code execution returns to the original shell’s process.

Creating a ‘copy’ of a running process is known as creating a *fork*, and this copied process would be the *child process* of the shell. Consequently, the original process or the shell process is the *parent process*.

In your typical shell, all commands executed are child processes of the shell’s main process.

0.3 Necessary System Calls

To implement this successfully, you need to know the following system calls:

1. fork(): forks current process and creates an identical copy
2. wait(): if a process is a parent process, wait receives the child processes PID and waits on it to finish. When waiting, the process becomes *Blocked*, more on that later
3. exec(): receives a command or binary, and any arguments needed and executes that binary or command.
4. **pipe()**: takes in an array of two integers, let’s call it FD, and allocates two file descriptors to it; anything written to FD[1] can be read from FD[0]
5. **dup()**: duplicates the file descriptor given as an argument and returns a new file descriptor to the same file, the old file descriptor is then assigned to the lowest unused file descriptor.
6. **dup2()**: takes two arguments: oldFD (old file descriptor) and newFD and switches their slots.[1]

0.4 Redirects

File Descriptors: In C, operating system redirects are typically achieved using file descriptors and the **dup2()** system call. In Unix-like operating systems, file descriptors are small non-negative integers that the operating system uses to identify open files, sockets, pipes, and other I/O resources. Standard input (stdin), standard output (stdout), and standard error (stderr) are typically represented by the file descriptors 0, 1, and 2, respectively.

Redirecting Output: When you want to redirect the output of a program to a file, you typically perform the following steps:

1. **Open or Create the File:** Use the **open()** system call to open or create the file you want to redirect output to. This call returns a file descriptor that represents the opened file.

[1] example given in code examples on Brightspace

2. **Redirect stdout:** Use the **dup2()** system call to duplicate the file descriptor representing the opened file onto the file descriptor representing stdout (file descriptor 1). This effectively redirects stdout to the specified file.
3. **Close the Original File Descriptor:** Once the redirect is set up, you can close the original file descriptor returned by **open()** as it's no longer needed.

Redirecting Input: Similarly, if you want to redirect the input of a program from a file instead of the standard input (keyboard), you can perform the following steps:

Your shell must also be able to handle redirects;

1. From files to standard input of a program
2. From program's standard output to files
3. A combination of both.

This can be done by using dup2 to switch out stdin or stdout for a file descriptor on an open file with the necessary permissions. [2]

0.5 Pipes

A pipe is a form of inter-process communication (IPC) that allows data to be transmitted between two related processes. In Unix-like operating systems, a pipe is a mechanism that allows the output of one process to be directed as the input of another process. Pipes are typically unidirectional, meaning data flows in one direction only.

- Pipes are implemented as a form of special file. They have a read end and a write end, which can be used by processes for communication.
- When a pipe is created, the operating system creates a buffer in memory where data can be stored temporarily.
- Data written to the write end of the pipe is stored in the buffer, and processes can read this data from the read end of the pipe.

Your shell must also be able to handle the output of one program to be piped to the input of another program. This is not limited to one program either; you could feasibly have the following scenario:

```
cmd1 | cmd2 | cmd3
```

For the above simplified example, this can be done by first creating a pipe using the pipe() system call, using dup2 to switch out stdout for FD[1] when running the first command, then using dup2 after execution to switch out stdin for FD[0] effectively creating a pipe between cmd1 and cmd2.[3]

[2] example given in code examples on Brightspace
[3] example given in code examples on Brightspace

0.6 Background Processes

You can push a process to be a background process by appending a **&** to the end of the command. This will allow you to continue using the shell without having to wait on the current command to finish executing. This can be done in case a process needs a lot of computation time but there is no need to get user input.

To implement this you simply have to check whether or not the process is a background process and either call the wait() system call or not.

3. Grading and Submission

22nd February

One or more **commented** C files must be submitted that provide the basic functionality of a shell as well as support for pipes, redirects, and running binaries as background processes. If more than one C file is provided, a MAKEFILE must also be submitted.