Pipes

Lecture 9





Today's Agenda

- Pipes: The Basics
- I/O Streams
- Piping Data

Today's Agenda

- **Pipes: The Basics**
- I/O Streams
- Piping Data

Pipes [1/2]

- Pipes are a common way for programs to communicate on Unix systems
- Most useful for sharing unstructured data (such a text) between processes
- They work like how they sound: if you want to send data to another process, send it through the pipe

Pipes [2/2]

- Pipes are one of the fundamental forms of Unix IPC
- With pipes, we can “glue” several utilities together:
  - `grep neato file.txt | sort`
  - This will search for “neato” in file.txt and print each match
    - Next, these matches get sent over to the ‘sort’ utility
- We **redirect** the standard output stream into another program!
  - Does not get displayed in the terminal

Pipe History

- Have you used pipes in your shell before?
- How can we find out?
- Let’s see what pipelines you’ve run… or can run.
- If you need more of a challenge: look for pipes in `generate-traces.sh` and understand what they do

In the Shell

- As we've seen, pipes are used frequently in the shell
- We can mix and match different utilities, and they all work well together
  - Awesome!
- Some genius must have designed all these programs to work this way, right?
  - Well, no. They all just read from `stdin` and then write to `stdout` (and `stderr`)
  - No coordination required between developers

To be clear…

- When you enter ‘ls’ in your shell, you’re running a program
- This functionality is **NOT** built into your shell. Bash simply finds and runs the ‘ls’ program. That’s it!
- So the most basic shell is:
  - fork()
  - exec()
  - Wait for the child process to finish

Built-Ins

- There are some shell “commands” that actually aren’t programs, called **built-ins**
- For example, `history` : printing previous commands
  - The shell knows what commands you’ve entered, so having an external program do this would be… somewhat questionable
- And, obviously, `exit`
  - If you create a new process, it’s not like it can quit for the parent process, right?
    - Well, I guess there **IS** a way to do that…

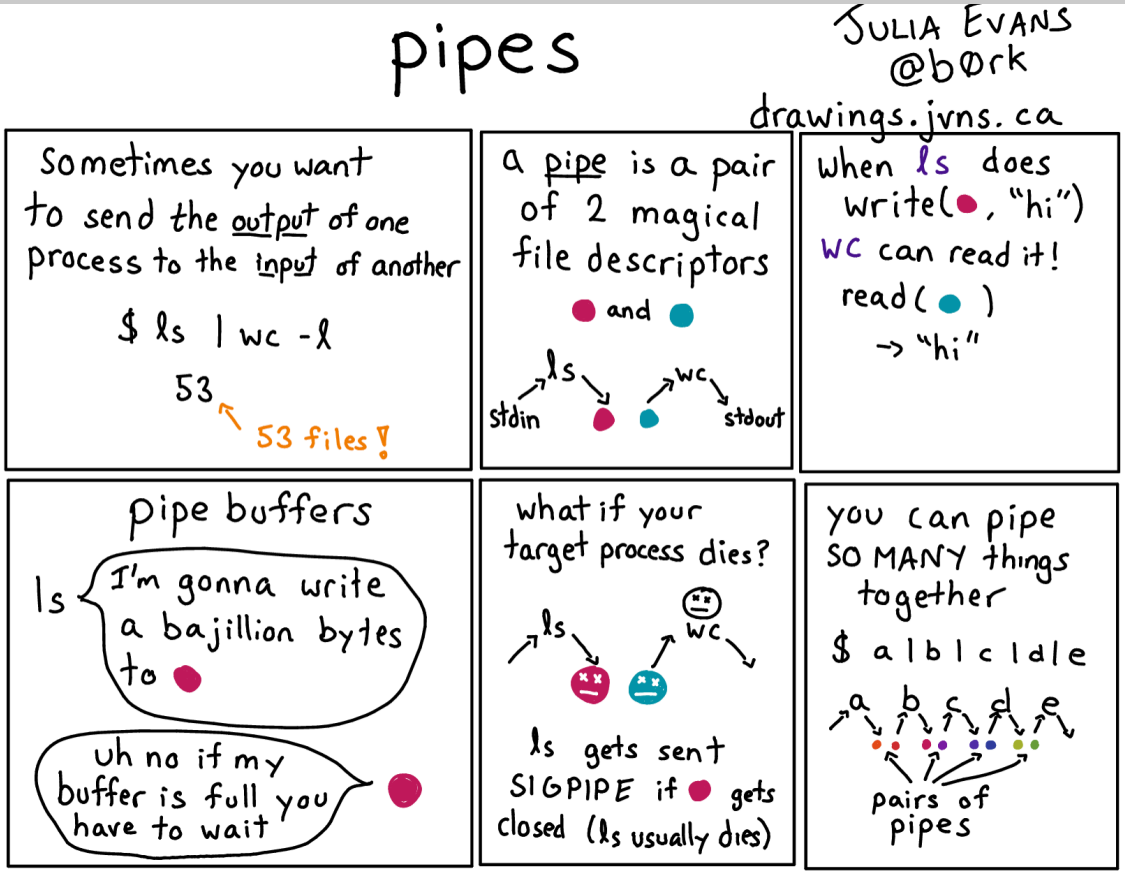
Another Built-In: cd

- Can you think of why `cd` needs to be a built-in command?
  - It's similar to the problem with `exit`
- We can change the current directory of **ourselves**, but not our parent
- This is true for general Unix-based systems, **but** if we think back to our discussion on `clone()` …
  - …Linux allows us to actually share the CWD with our parent (man pages of `clone` talk about this `cd` case)

Going to the Source

- I have posted a video from Bell Labs on the schedule that discusses several design aspects of Unix
- It's actually pretty interesting… well, at least as far as OS content goes!
- Discussion on pipes starts right around the 5 minute mark

Another Take (Julia Evans)



Today's Agenda

- Pipes: The Basics
- **I/O Streams**
- Piping Data



Input/Output Streams

- Each program gets allocated three streams by default:
  - `stdout` (standard output)
  - `stderr` (standard error)
  - `stdin` (standard input)
- As with most things in Unix, these are represented as files (via *file descriptors*)
- These streams have different functions…

stdout (fd 1)

- When you call `printf`, you are writing to `stdout`
- This stream is designed for general program output; for example, if you type `ls` then the list of files should display on stdout
- You can pipe stdout to other programs:
  - `ls -l / | grep 'bin'`
- …or redirect to a file:
  - `ls -l / > list_of_files.txt`

stderr (fd 2)

- The standard error stream is used for diagnostic information
- This way, program output can still be passed to other programs/files but we'll still see diagnostics printed to the terminal
  - Helps us know when something went wrong
- Unlike `stdout`, `stderr` is **not** buffered

stdin (fd 0)

- The final stream, stdin, one way to provide program input (via `scanf`, for example)
- This can be entered by the user, or we can pipe input directly into a program:
  - `ls -l / | grep 'bin'`
    - `ls -l /` – writes file list to `stdout`
    - `grep 'bin'` – reads file list from `stdin`

fprintf

- We can control where printed output goes with `fprintf`
  - File printf
- stderr is actually a file – in fact, on Unix systems most devices are represented as files
  - Try `ls /dev` to view the devices on your machine (includes macOS)
- So if we want to write data to a file, just pass it in:
  - `fprintf(file, "My name is: %s", "Bob");`

Today's Agenda

- Pipes: The Basics
- I/O Streams
- **Piping Data**

The pipe system call

- Now back to pipes: we can create them with the pipe() system call
  - Returns a set of file descriptors: the **input** and **output** sides of the pipe
- Pipes aren't very useful if they aren't connected to anything, though
  - We can do this by `fork()`ing another process

Piping to Another Process

- After calling `fork()`, both processes have a copy of the pipe file descriptors
- Pipes only operate in one direction, though, so we need to close the appropriate ends of the pipe
  - You can think of a forked() pipe as one with four ends: two input and output ends each
  - We eliminate the ends we don't need to control the direction of data flow
  - Amazing ASCII art drawing: `>---<`

Controlling Flow

- To control data flow through the pipe, we close the ends we won't use
- For example:
  - Child process closes **FD 0** and reads from **FD 1**
  - Parent process closes **FD 1** and writes to **FD 0**

Async Process Creation

- You may be wondering: what good are pipes when we have to start all the cooperating processes?
- There's actually another option: **FIFO**s, aka **named pipes**
- Create with the `mkfifo` command, then open as you would a regular file descriptor

Redirecting Streams: dup

- `dup` allows us to redirect streams by **duplicating** them
  - `int dup(int fildes);`
- Let's say we want to make our standard output stream go through the pipe we just created
- We'll do:
  - `close(fd[0]); // close read end of pipe`
  - `close(1); // close stdout (fd 1)`
  - `dup(fd[1]); // stdout to pipe`

What??

- In the previous example, we redirected `stdout` to the pipe by passing in the **write** end of the pipe's fd
- `dup` duplicates its arg fd into the first unused file descriptor
- We made sure the first unused file descriptor is `stdout` by closing it before the `dup`

dup2

- Many operating systems also support `dup2` to make this a bit clearer
- Here's an example of using `dup2` to redirect to a file:
  - ```
    int output = open("output.txt",
    O_CREAT | O_WRONLY | O_TRUNC, 0666);
    ```
  - ```
    dup2(output, STDOUT_FILENO);
    ```
- Cool, right? Ok, let's try this out…

Demos

- pipe.c
- io-redir.c
- Shell commands