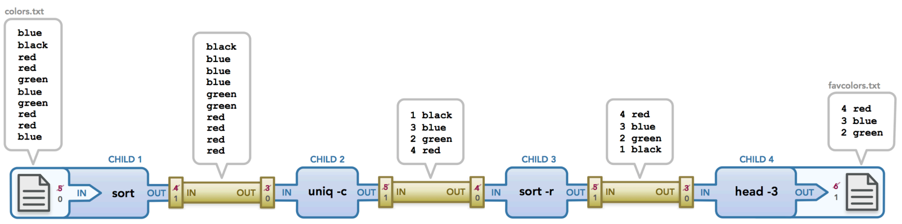
rozmichelle (https://www.rozmichelle.com/)

THE PERSONAL BLOG OF CREATIVE SOUL ROSLYN MICHELLE CYRUS MCCONNELL


colors.txt
blue black red red green blue green red red blue
black blue blue blue green green red red red red
1 black 3 blue 2 green 4 red
4 red 3 blue 2 green 1 black
favcolors.txt
4 red 3 blue 2 green
CHILD 1
CHILD 2
CHILD 3
CHILD 4
sort
uniq -c
sort -r
head -3
IN
OUT
5
0
4
1
3
6

(https://www.rozmichelle.com/pipes-forks-dups/)

PIPES, FORKS, & DUPS: UNDERSTANDING COMMAND EXECUTION AND INPUT/OUTPUT DATA FLOW (HTTPS://WWW.ROZMICHELLE.COM/PIPES-FORKS-DUPS/)

BY ROSLYN MCCONNELL (HTTPS://WWW.ROZMICHELLE.COM/AUTHOR/ROZROZ/) // NOVEMBER 8, 2017 (HTTPS://WWW.ROZMICHELLE.COM/PIPES-FORKS-DUPS/) // 💬 30 COMMENTS (HTTPS://WWW.ROZMICHELLE.COM/PIPES-FORKS-DUPS/#COMMENTS)

Note: a basic familiarity with Unix commands and C/C++ is necessary to understand this post. My goal is to explain the flow of data between processes when running commands. If you want to jump directly to the pipeline stuff, click here.

I'm currently enrolled in a systems programming class at Stanford (CS110: Principles of Computer Systems). It is the second systems class I've taken (the first was CS107 which teaches C and focuses on understanding pointers and memory management). This class focuses mainly on the inner workings of the operating system, using C and C++ to teach us concepts like process management, program execution, and handling data. While I enjoyed and quickly grasped the concepts taught in CS107, I've

SEARCH MY *site*

Search

resume

Download My F
(https://www.rc
content/uploac

MY *poems*

Spacetime (https://www.rozmichelle.com/spacetime/)

Wait (https://www.rozmichelle.com/wait/)

Sankofa Love

had a harder time understanding the material in CS110. The class itself is extremely interesting and well-taught, but my main pain point has been understanding the way processes share data and how input and output work across commands entered in the terminal. In the last few days, however, I finally found clarity when I started creating diagrams to model process behavior and the path that data takes as it travels from one command to another. I'd like to share what I've learned with you. In this post, we'll go over how Unix commands pass data to each other via pipes and input/output redirection and I'll illustrate what actually happens to the flow of data when a command is executed.

File Descriptors

Let's start out with a basic "type something into the keyboard, press enter, and get a result" model of running a single command in the terminal with no input/output redirection. Unix associates input with the terminal keyboard and output with the terminal display by default. Unix is famous for modeling pretty much everything in the computer as a file, including the keyboard and monitor. Thus writing to the "display" is really just writing to the file that manages the display of data on the screen. Similarly, reading data from the keyboard means reading data from the file that represents the keyboard. In the context of this discussion, we'll refer to input and output as text data that goes into and out of a process.

Data flows via streams that transfer bytes from one area to another. There are three default input/output (I/O) streams: **standard input (stdin)**, **standard output (stdout)**, and **standard error (stderr)**. By

(https://www.rozmichelle.com/sankofa-love/)

Hey Bright Soul (https://www.rozmichelle.com/hey-bright-soul/)

Lasso (https://www.rozmichelle.com/lasso/)

Soul Swim (https://www.rozmichelle.com/soul-swim/)

The Color Of Your Eyes (https://www.rozmichelle.com/the-color-of-your-eyes/)

Kingdom Come (https://www.rozmichelle.com/kingdom-come/)

Prism

default, these streams each have a specific file descriptor. A file descriptor is an integer that is associated with an open file (the workings of which are beyond the scope of this discussion), and processes use file descriptors to handle data. The three default streams have the following file descriptor numbers: stdin = 0, stdout = 1, and stderr = 2. File descriptors are stored in a file descriptor table, and every process has its own file descriptor table (with 0, 1, and 2 created and mapped to their appropriate streams by default when the process is created). Each stream has no idea where the data sent to or read from its descriptor comes from or goes to; the streams simply deal with the file descriptors, not the data sources themselves. The process only needs to handle the file descriptor, not the file itself; the kernel safely manages the file.

In addition to 0, 1, and 2, processes use other file descriptors as necessary. The lowest unused (unopen) file descriptor is always used when a new file descriptor is assigned. Thus file descriptor 3 is usually first in line to be used after 0, 1, and 2 are set up by default.

Data Flow

Now we are ready to talk about data flow in depth. When we run commands in the terminal, any input and output need to be handled appropriately. The process that gets created for each command needs to know what data, if any, to take in as input and possibly what data to output. Each command's process also needs to know where to send and receive such data. To represent the flow of data in (via stdin from the keyboard by default) and out (via stdout to

(https://www.rozmichelle.com/prism/)

Origami (https://www.rozmichelle.com/origami/)

Sugar On The Rim (https://www.rozmichelle.com/sugar-on-the-rim/)

Cliffhanger (https://www.rozmichelle.com/cliffhanger/)

Imagine (https://www.rozmichelle.com/imagine/)

Mechanical Heart (https://www.rozmichelle.com/mechanical-heart/)

Astronomy Lessons (https://www.rozmichelle.c

the terminal by default, and also via stderr if something goes wrong), I'll use diagrams like the one below:

Conceptual data flow for the standard input (0) and output (1) streams

The above figure represents the default setup of the input and output streams. The keyboard passes data to the program that runs the command (from the command's perspective, it receives input via stdin), and that program sends output to the terminal via stdout. I represent the flow of data from left to right. I also use the words "in" and "out" to represent data that goes into one area and out of another area, respectively. Although in this case "in" and "out" are associated with stdin and stdout respectively, this will not necessarily be the case in future diagrams. Thus I put the file descriptor numbers next to the associated "file" that corresponds to the relevant "in"/"out" action to make it clear which file descriptor is being used for which purpose. Generally, data flowing "into" something is considered input (and is being **read** in from a source via a file descriptor) and data flowing "out" of something is considered output (and is being **written** out to a source via a file descriptor). Put another way: **input is read from somewhere; output is written somewhere**. This mental model will prove helpful in future diagrams that are more complex.

om/astronomy-lessons/)

Purple (https://www.rozmichelle.com/purple/)

Circles (https://www.rozmichelle.com/circles/)

Alchemy Dream (https://www.rozmichelle.com/alchemy-dream/)

T.h.a.w. (https://www.rozmichelle.com/t-h-a-w/)

Helium (https://www.rozmichelle.com/helium/)

The First Draft (https://www.rozmichelle.com/the-first-draft/)

Love Letters

One thing to note is that there are actually two streams that can write output to the terminal by default: stdout and stderr. The stderr stream is used when something goes wrong when trying to execute a command. For example, the following **`ls dir_x`** command in my terminal tries to list the contents of the nonexistent directory dir_x:

```
$ ls dir_x
ls: cannot access dir_x: No such file or
directory
```

In this example, the stream that is used to display the second line is actually stderr, not stdout. Since stderr also goes to the terminal by default, we see the error message in the terminal. If the directory existed, then stdout would output the directory's contents to the screen.

Here is an updated diagram that shows the output ends of the stream for both stdout and stderr:

Conceptual data flow for the standard input (0), output (1), and error (2) streams

Remember that the word "out" simply means output, and the file descriptor associated with each output is shown next to it. Now that you understand that stderr exists and can be used, I'll actually leave it out of future data stream diagrams unless an example specifically uses stderr. Just remember that it exists!

(https://www.rozmichelle.com/love-letters/)

Black Sand (https://www.rozmichelle.com/black-sand/)

Organ Donor (https://www.rozmichelle.com/organ-donor/)

Quantum Physics (https://www.rozmichelle.com/quantum-physics/)

Godiva (https://www.rozmichelle.com/godiva/)

Friday The 13th (https://www.rozmichelle.com/friday-the-13th/)

Silk (https://www.

Now we can explore data flow using commands. Some commands both read input and write output, but others only do one or neither. We'll explore different cases, but first, let's discuss what input really means here. Technically, from the shell's perspective (the shell processes the command line that is given to the terminal), anything typed into the keyboard (including the command itself) is "input" in the general sense, but we are dealing specifically with the input and output that commands need in order for the processes that run the commands to transfer data to and from files (including the keyboard and display). Command arguments that are options are really read in from the command line (as an argument array); actual input is read in from an open file that is associated with a file descriptor. Thus I define the input to a command as data that is specifically passed in using stdin (or another repurposed file descriptor that can be read from), whether it is typed in via the keyboard, redirected via I/O redirection (explained later), or possibly passed to the command as a file argument (versus an option argument). If a file is passed as an argument, then I consider it input if the process will actually read or manipulate the **contents** of that file (e.g. to sort the contents), versus simply referring to the file itself (e.g. to move or rename it).

As a side note, command line option arguments are the result of another Unix design choice that allows behavior modification of an executed command to be passed in separately from the received input. Keeping the arguments and input separate makes life easier when pipes are involved.

rozmichelle.com/silk/)

Speak Now (https://www.rozmichelle.com/speak-now/)

It's Not Like (https://www.rozmichelle.com/its-not-like/)

Glass Jar (https://www.rozmichelle.com/glass-jar/)

Love In The City (https://www.rozmichelle.com/love-in-the-city/)

R&B (https://www.rozmichelle.com/rnb/)

Inside (https://www.rozmichelle.com/inside/)

Write On

Now let's look at some examples. To illustrate a command that can have no input but has output, consider `ls` , which lists all the files in the current directory:

```
$ ls
dir1
file1
file2
```

This can be visualized as follows:

The ls command.

If a command doesn't accept input from stdin, then the data passed to such a command will simply be ignored by the program that runs the command since it was not written to handle input data. For example, `< words.txt ls` will list the files and directories in the current directory and ignore the input that was redirected into stdin (this uses I/O redirection, which I'll explain later).

Let's look at a command that, if all goes well, takes no input and gives no output: `mv` , which can be used to move or rename files. If I give it the name of a file or directory that can be moved or renamed successfully, then no data is output via stdout or stderr. Remember that since this file's contents

(https://www.rozmichelle.com/write-on/)

Just Like Jamaica (https://www.rozmichelle.com/just-like-jamaica/)

Death (https://www.rozmichelle.com/death/)

Sweetness (https://www.rozmichelle.com/sweetness/)

Lynch (https://www.rozmichelle.com/lynch/)

Rapture (https://www.rozmichelle.com/rapture/)

New Roads (https://www.rozmichelle.com/new-roads/)

aren't being read or used in any way, the file that is passed in is not considered input. In a successful call to this command, I'd have this very simple diagram:

No input or output

If, however, I use **`mv`** incorrectly such that an error occurs, then I will have output to stderr:

```
$ mv
mv: missing file operand
Try 'mv --help' for more information.
```

Calling mv with no arguments

Let's make things more interesting. One of my favorite examples of a command that both reads input and writes output is **`sort`** . When used with no file arguments and no input redirection, the terminal waits for the user to enter the strings to sort (one string per line). Once the user types Ctrl-D (which closes the write end of the communication channel that connects the keyboard to the stdin of the **`sort`** process), the process running **`sort`** will know that all desired strings have been entered. Thus these strings are passed via stdin into the process that runs

[Celestial Woman (https://www.rozmichelle.com/celestial-woman/)](https://www.rozmichelle.com/celestial-woman/)

[Timeless (https://www.rozmichelle.com/timeless/)](https://www.rozmichelle.com/timeless/)

[Bittersweet (https://www.rozmichelle.com/bittersweet/)](https://www.rozmichelle.com/bittersweet/)

[Lens (https://www.rozmichelle.com/lens/)](https://www.rozmichelle.com/lens/)

RECENT
posts

[React Game Design: Recreating Wordle (https://www.rozmichelle.com/react-game-design-recreating-](https://www.rozmichelle.com/react-game-design-recreating-)

the command, sorted by said process, and then written to the terminal via stdout. Pretty nifty! Here's sample input/output:

```
$ sort
cherry
banana
apple
apple
banana
cherry
```

The bolded strings are user input and the strings that follow represent the sorted output. Here's the data flow for this example:

The sort command. Input is typed into the keyboard, then the output is displayed in sorted order.

Note that `sort` can also take a filename argument to get input from the specified file instead of waiting for data to be entered by the user (for example, `sort words.txt`), which follows our definition of input since it is a file and not an option argument like in `sort -r`. In addition, `sort` can take input via input redirection, which I'll explain later.

wordle/)

Spacetime (https://www.rozmichelle.com/spacetime/)

Improving Insight At NETGEAR (https://www.rozmichelle.com/improving-insight-at-netgear/)

Interactive Storytelling with Twine (https://www.rozmichelle.com/interactive-storytelling-with-twine/)

Wait (https://www.rozmichelle.com/wait/)

categories

Computer Science

Now that we understand the general idea of data flow from stdin to stdout or stderr, we can discuss how to control the flow of input and output. I'll cover two ways to do this: using pipes, which allow the output of one process to be passed as input into another process, and using I/O redirection, which allows files to be the source and destination of data instead of the default keyboard and terminal. Fun stuff! Let's dive right in.

Introducing Pipes

Unix has a simple yet valuable design philosophy, as explained by Doug McIlroy, the inventor of the Unix pipe:

> *"Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."*

The concept of a pipe is extremely powerful. Pipes allow data from one process to be passed to another (via unidirectional data flow) so that commands can be chained together by their streams. This allows commands to work together to achieve a larger goal. This chaining of processes can be represented by a pipeline: commands in a pipeline are connected via pipes, where data is shared between processes by flowing from one end of the pipe to the other. Since each command in the pipeline is run in a separate process, each with a separate memory space, we need

(https://www.rozmichelle.com/category/computer-science/) (9)

Creative Writing (https://www.rozmichelle.com/category/creativewriting/) (48)

Poetry (https://www.rozmichelle.com/category/creativewriting/poetry/) (45)

Prose (https://www.rozmichelle.com/category/creativewriting/prose/) (3)

Design (https://www.rozmichelle.c

a way to allow those processes to communicate with each other. This is is exactly the behavior that the **`pipe()`** system call provides.

Implementation-wise, pipes are actually just buffered streams that are associated with two file descriptors that are set up so that the first one can read in data that is written to the second one. Specifically, in the code written to handle the execution of commands in a pipeline, an array of two integers is created and a **`pipe()`** call populates the array with two available file descriptors (usually the lowest two values available) such that the first file descriptor in the array can read in data written to the second.

Physical pipes are naturally a great analogy for this abstraction. We can think of the data stream that starts in one process as water in an isolated environment, and the only way to allow the water to flow to the environment of the next process is to connect the environments with a pipe. In this way, the water (data) flows from the first environment (process) into the pipe, filling up the pipe with all its water and then draining its water into the other environment. This data flow is exactly what I try to capture in the diagram for the following pipeline example, **`sort | grep ea`**:

sort | grep ea

Let's break this down piece by piece. The **`sort`** command, like the previous example, waits for input from the user (who enters three strings to sort) via

om/category/design/) (22)

Artwork (https://www.rozmichelle.com/category/design/artwork/) (3)

Digital Art (https://www.rozmichelle.com/category/design/digital-art/) (5)

Games (https://www.rozmichelle.com/category/design/games/) (5)

Logos & Icons (https://www.rozmichelle.co

stdin (file descriptor 0). Next, the strings are sorted and sent as output via stdout which is fed into the pipe. It does this by allowing stdout to feed data to the left end of the pipe (file descriptor 4) instead of to the terminal. I'm leaving out a detail here but this process is explained in more depth later in this post.

Before we continue, here's an important detail: remember when I mentioned that each process gets its own file descriptor table? Well, since each command in the pipeline is run in a separate process, each command has its own version of the file descriptors, including its own stdin, stdout, and stderr. This means that the 0 shown on the left side of the diagram belongs to the process running `sort` and is thus in a different file descriptor table than the 1 shown on the right, which belongs to the process running `grep`. However, since the streams are set up to send data beyond process boundaries, the end result is that data ends up where it belongs as long as it was properly passed down the pipeline.

Moving on: now that the `sort` command has a sorted list of strings as output, it must pass it through the created pipe to communicate the data to the next process, `grep`. Ignoring file descriptors 3 and 4 for a moment, look at the "in" and "out" words: we see that data flows **out of** the `sort` process and **into** the pipe, where it then is passed **out of** the pipe and **into** the grep process. "In" and "out" are phrased based on the context they are used in: either inside the pipe or outside of it.

With that in mind, we can now discuss the file descriptors handed out by the `pipe()` call. Assume that in the code that executes the commands in the pipeline, the `pipe()` call populates a file descriptor

m/category/design/logos-and-icons/) (1)

Portfolio (https://www.rozmichelle.com/category/design/portfolio/) (13)

Journal (https://www.rozmichelle.com/category/journal/) (45)

Alter Ego (https://www.rozmichelle.com/category/journal/alter-ego/) (2)

Music (https://www.rozmichelle.com/category/journal/music/)

array {3, 4} such that data written to 4 can be read from 3. It really doesn't matter what these numbers are or even that they are in increasing order. Imagine that the array is {pickle, mickeymouse} if that helps; the given values only matter to the process, but the purpose of the file descriptors matters to the data! The purpose of each file descriptor depends on which index each is in the array.

There's a **very** important concept to grasp here, one that took me a while to finally understand (and that didn't happen until I created these diagrams because of the way my visual brain works). Recall that in my mental model, data flows from left to right in the diagrams. The file descriptors in the array are set up so that what is written to 4 can be read from 3, so you might be wondering why 4 is shown on the left side of the pipe in the image above and 3 is on the right, versus the other way around. The key thing to understand is that **the read and write actions defined by a `pipe()` call are from the perspective of the two processes using the pipe, not the pipe itself!** Thus when the **`pipe()`** call defines 4 to be the writable end of the pipe, it means that it is the end of the pipe that the **first command's process** writes output to so that the pipe itself receives that data as input. It does **not** mean the opposite: that 4 is the side of the pipe where the pipe writes out data, which would tempt you to label the right side of the pipe as 4. Similarly, 3 being the readable end of the pipe means that it is the end of the pipe that the **second command's process** reads data from. It's all about context! :)

Thus the data is passed to the pipe, where it sits until all of the data is received so it can drain itself out to the **`grep`** process. As the last and hopefully

(2)

Natural Hair Journey (https://www.rozmichelle.com/category/journal/natural-hair-journey/) (2)

Roz Daily (https://www.rozmichelle.com/category/journal/roz-daily/) (9)

Travel (https://www.rozmichelle.com/category/travel/) (10)

QUICK *facts*

From

Fort

relatively easy step, the process running **grep** searches the input it received from the pipe's output for lines that contain "ea". It then uses its stdout stream to output the matched strings to the terminal. All done! Not bad for our first pipeline walkthrough! Next, we'll dive even deeper into understanding how code can execute these processes. Our understanding of how **pipe()** works is, in my opinion, half the battle. Understanding **fork()** and **dup2()** is the other half. Let's see how these functions work!

Running Commands in a Pipeline

In the diagrams we've seen so far, pipes were used when passing data from one command's process to another, but we haven't discussed the hierarchy of processes that run such commands. In class, we learned to write programs such that every command is executed in a child process versus in the parent process (the calling process). Typically, the parent does any required setup and then creates a child process via a **fork()** call, which creates a clone of the parent's memory state and file descriptors. Thus the child ends up with an independent copy of the variables and file descriptors that existed in the parent at the time of the **fork()** call. After the **fork()** call, changes to the parent process will not be visible to the child process and vice versa.

This children-execute-commands pattern seems unnecessary for executing a single command since we could simply run the command in the parent without creating a child, but when you think about what it takes to make the code generic enough to work for both a single command and multiple commands in a pipeline, then it makes sense to

Lauderdale, Florida

Residence

Sunrise, Florida

Ancestry

Jamaica, Ivory Coast

Education

Stanford University c/o 2019: M.S. in Computer Science

Cornell University c/o 2007: B.A. in English and Africana Studies

Occupation

Product Design Manager I at Datadog

always have a different child process execute each command. There are exceptions to this rule, such as running a built-in command which can simply run in the parent, but for this discussion, we'll assume that all commands are executed in child processes.

Let's look at a pedantic example of some C code that runs the `sort` command. In this example, input is printed directly to a file descriptor using `dprintf()` to show you a case where a pipe is used to send data from the parent to the child. This is a simplified version of sample code provided to the class by our instructor, Jerry:

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int fds[2];                          // an array
that will hold two file descriptors
  pipe(fds);                           // populates
fds with two file descriptors
  pid_t pid = fork();                  // create c
hild process that is a clone of the parent

  if (pid == 0) {                      // if pid ==
0, then this is the child process
    dup2(fds[0], STDIN_FILENO);        // fds[0] (t
he read end of pipe) donates its data to file d
escriptor 0
    close(fds[0]);                     // file desc
riptor no longer needed in child since stdin is
a copy
    close(fds[1]);                     // file desc
riptor unused in child
    char *argv[] = {(char *)"sort", NULL};   /
/ create argument vector
    if (execvp(argv[0], argv) < 0) exit(0);  /
/ run sort command (exit if something went wron
g)
  }

  // if we reach here, we are in parent process
  close(fds[0]);                  // file descri
ptor unused in parent
  const char *words[] = {"pear", "peach", "app
le"};
  // write input to the writable file descripto
r so it can be read in from child:
  size_t numwords = sizeof(words)/sizeof(word
s[0]);
  for (size_t i = 0; i < numwords; i++) {
    dprintf(fds[1], "%s\n", words[i]);
  }

  // send EOF so child can continue (child bloc
ks until all input has been processed):
  close(fds[1]);

  int status;
  pid_t wpid = waitpid(pid, &status, 0); // wa
it for child to finish before exiting
  return wpid == pid && WIFEXITED(status) ?
WEXITSTATUS(status) : -1;
}
```

The program is written to run one specific command: `sort`. Here's how the code works: in the parent process, an array is created to store two file descriptors. After the `pipe()` call, the array is populated with the hooked up file descriptors, where the first one will be read from the child process and the second one will be written to by the parent process. Then **`fork()`** is called to create the child process, which has a copy of the parent's file descriptors and memory. From that point, we check if we are running in the child process. If we are, the child calls **`dup2()`** to cause its stdin to associate itself with the readable end of the pipe, which corresponds to fds[0]. An important detail about the way **`dup2()`** works is that it will first close its second parameter, which is a file descriptor, if necessary. Thus in this example, stdin (which is open by default) is first closed, which will remove its reference to the default keyboard file. Then the stdin of the child process will be able to receive data via fds[0] instead of from the keyboard. That's the magic of **`dup2()`**!

Now that the child's stdin is ready to read in data, the child closes the file descriptors created by the **`pipe()`** call since they are no longer needed in the child. The child then executes the **`sort`** command, waiting for all the parent's data to be written into the appropriate end of the pipe before sorting the data.

It is possible that when **`fork()`** is called, the child gets going before the parent continues, in which case the child hangs until it receives all input. Once the sort command finishes, the child process finishes after the **`execvp()`** call (which executes the given command) and closes its default file descriptors 0, 1, and 2 automatically. After the **`fork()`** call that creates the child, the parent closes fds[0] since it is not needed in the parent (the parent only needs to write data, not read it). The parent then writes each word in the given array to the writable end of the pipe (fds[1]), adding a new line character at the end to allow the sort command to properly receive each word on a new line. When all words have been written, the parent closes fds[1] since it is done writing data, which sends an EOF to the child to allow it to execute the sort command. The parent responsibly waits for the child to finish (via the **`waitpid()`** call) before exiting. The last line is just a tidy way to return a value that depends on whether things went as expected.

When all is said and done, this is the data flow diagram for the entire sequence of events:

The final data flow of the code example. In the child, file descriptor 3 was copied to the child's stdin, then 3 was closed and only the child's stdin is used to get data, as represented by the numbers on the right side of the pipe.

Keep in mind that this is example shows how a pipe is used, but a pipe isn't necessary in this case. For example, the child could simply access data from the default stdin without any interference from the parent, which wouldn't require using a pipe. This code simply shows how pipes set up the communication from one process to another, and this pattern is crucial when managing pipelines that have more than one command.

To get an idea of what happens in this code, I've designed the following diagrams. In the images below, the lines show the association between a file descriptor and the open file it has a pointer to. The line arrow directions represent the flow of data. These diagrams should hopefully make it clear which file descriptors are needed by the parent and child, which in turn should help explain when it is appropriate to close a file descriptor to avoid a leak. Also keep in mind that because the instructions in the parent aren't guaranteed to run before the instructions in the child, some of the steps below could happen at different times. The images are just to give you an idea of what could happen during execution, even though a few steps could be swapped along the way.

When the program starts, the parent process is created with t
set up in its file descriptor table. The arrows show the flow of
input from the keyboard and stdout and stderr send output to

The pipe() call finds the next two available file descriptors and the appropriate end of the created pipe. In this case, a process write via 4.

The fork() call creates the child process, which is a copy of th and file descriptor table at that point in time. Whatever file descriptors are associated with are the same files that the chi are associated with.

The parent closes the file descriptor it doesn't need. The child
its stdin be a copy of fds[0], closing file descriptor

The parent writes data to the writable end of the pipe. The c
descriptors it doesn't need.

After writing all the data, the parent closes fds[1] to let the chi

has been sent.

The child process executes the sort command on tl

The sorted output is sent to the terminal and the child send
terminates, which will then allow the parent process

The processes clean up after their default file descriptors
additional file descriptors used during program execution h
closed.

In particular, look at the steps that have blue lines, which represent the flow of data at that point in time. If you chain those steps together, you get the following: the parent writes data to the writable end of the pipe, which then is read from the readable end of the pipe by the child via the child's stdin file descriptor, and lastly, that data is sent from the child as output to the terminal. In a nutshell, all the other lines in the diagram end up either handing off data to others or not even being used. Also, when the program is over, the file descriptors that needed to be closed are gone. This is a small program example, but you can see how messy it can get when pipes are involved. This, however, is a beautiful process: keep track of the data and clean up after yourself, and the chaos will sort itself out in the end!

One thing you'll notice here is that the file descriptors that a pipe starts with may get redirected to another stream as necessary. The pipe is a convenience that gives you two file descriptors that are set up to work together, but their purposes can be redirected as necessary to ensure that data flows to and from the right places.

Hopefully, these diagrams clarify what happens when creating processes to run commands. It's important to understand not only how the file descriptors are used but also when they are not used so they can be closed appropriately. They are powerful and very easy to get wrong or leave behind.

I/O Redirection

There's one last topic I'd like to go over. In our discussion thus far, we've explored the default behavior that comes with using the three default file descriptors, where we use the keyboard and terminal for all initial input and final output, respectively. We've also looked at how we can pass data between processes in a pipeline. What if we want to use an existing file as input to the first command in a pipeline instead of using the keyboard for input, or what if we want to send the output of the last pipeline command to a file? This can be done with I/O redirection. On the command line, the "<" character is used for input redirection and ">" is used for output redirection, where the output file is created if it doesn't exist or overwritten if it already exists. To append data to an output file instead of overwriting the contents, you can use ">>". Let's look at an example that uses both input and output redirection. Say we have the file **words.txt** with the following content:

```
$ cat words.txt
pear
peach
apple
```

We can use this file as input to the `sort` command and then output the contents to another file (or even the same file if desired) as shown below:

```
$ < words.txt sort > words2.txt
```

Note that there is no output to the screen because the output is stored in **words2.txt**. We could also write this command as `sort < words.txt > words2.txt`. If we use `cat` to print out the contents of the output file, we get the following:

```
$ cat words2.txt
apple
peach
pear
```

It turns out that implementing I/O redirection is relatively simple to do. We can simply use the **`dup2()`** magic that we saw earlier:

```
// if first command in pipeline has input redir
ection
if (hasInputFile && is1stCommand) {
  int fdin = open(inputFile, O_RDONLY, 0644);
  dup2(fdin, STDIN_FILENO);
  close(fdin);
}

// if last command in pipeline has output redir
ection
if (hasOutputFile && isLastCommand) {
  int fdout = open(outputFile, O_WRONLY |
O_CREAT | O_TRUNC, 0644);
  dup2(fdout, STDOUT_FILENO);
  close(fdout);
}
```

This code is hopefully pretty straightforward. If there is input to redirect (the detection of which is handled in another function not shown here), then call **`open()`** on the file and assign that data stream to the file descriptor that **`open()`** uses. Then use **`dup2()`** magic to allow stdin to read that file's contents as input. Similarly, if output redirection is at the end of the pipeline, then redirect stdout to write the contents of the last command to the specified file.

Here's a diagram that represents the previous example:

Input and output redirection

You may be wondering why the file descriptors both start off as 3. I ran this command in a mini-shell I wrote for a homework assignment, so I am able to print out the file descriptors that are assigned when I run any command. My shell uses the code shown above. Note that first I check if there is input redirection. If there is, I call the **`open()`** command to read the data, which assigns this stream to file descriptor 3. Once I redirect stdin to handle the data that 3 handles, I then close 3, which makes 3 available for the output redirection check. Thus both files start out using 3 but then are appropriately redirected to the stream that needs the data, as shown by the red struckthrough text.

You can see the power of Unix at work here, where we can not only chain together small programs to make a larger program, but we can also load data into a pipeline and output data into a file for future use. I find it all so magical. :)

Summary

To wrap up what we've discussed, let's look at a long pipeline with a trivial but potentially useful example. Let's imagine you want to find out what the most loved color is among a group of people. Someone types up a list of colors where each line represents one person's favorite color. It is your job to take that file and run some commands on it, ideally in one impressive pipeline, such that the three most popular colors are saved in a new file along with a count of how many votes each color received. Here's how you could do it:

```
$ < colors.txt sort | uniq -c | sort -r | head
-3 > favcolors.txt
```

And here's the diagram:

[illegible]
Finding the most popular color

The commands work as follows: **colors.txt** contains a list of colors that were entered in random order. The **`uniq`** command removes any line that is the same as the line preceding it, effectively removing all consecutive duplicate lines. In order for this to work as desired, we need to first sort the list of colors, which is why we call **`sort`** first. Then we call **`uniq -c`**, where the **`-c`** option will remove the duplicate colors and also show a count of how many times each color appeared. Next, we sort this data in descending order (which is what the **`-r`** option does when passed to **`sort`**). Lastly, we call **`head -3`** to get the top three lines in the result, and we store that output in **favcolors.txt**. Sweet! In the end, **favcolors.txt** has the following desired data:

```
$ cat favcolors.txt
      4 red
      3 blue
      2 green
```

This is more complex than other examples we've studied, and the file descriptors shown in the diagram make that clear. Because my shell program calls pipe() before it checks for I/O redirection, the first pipe gets file descriptors 3 and 4 and then the file descriptors for the input and output files are assigned 5 and 6, respectively. Once 5 and 6 are redirected to stdin and stdout, respectively, they are closed (as shown on the far left and far right sides of the diagram). By the time the second pipe is created, it can use some recycled file descriptors, and the same goes for the last pipe. Don't worry too much about how the file descriptors are recycled because that is specific to my code, but know that everything gets cleaned up in the end and the data is passed off correctly.

We've covered quite a bit of material, and although I love my audience, I really wrote this more for my own sanity than anything else! But as always, I hope this was helpful. Writing this all out and creating the diagrams definitely solidified my own understanding of everything, although maybe a week too late for the midterm and homework nightmares I experienced last week. :)

A special "thank you!" goes out to Hemanth, the course assistant who answered my countless questions about these topics! I sent him many diagram mockups to make sure they were both accurate and clear, and his help really got me to a solid point of understanding. The course staff for this class has been amazing, but Hemanth really has gone above and beyond to help me. The same goes for the professor, Jerry, who has tolerated my many questions and panic attacks about my course performance. :) Shoutout to him as well for reviewing this post for accuracy. A support system like this is why I love Stanford so much.

Thanks for reading!

THIS ENTRY WAS POSTED IN **COMPUTER SCIENCE (HTTPS://WWW.ROZMICHELLE.COM/CATEGORY/COMPUTER-SCIENCE/)** AND TAGGED **UNIX (HTTPS://WWW.ROZMICHELLE.COM/TAG/UNIX/)**.

NAVIGATION

← Fun with Fractals (https://www.rozmichelle.com/fractals/)

A New Christmas Toy: Digital Sketching with the Slate 2 → (https://www.rozmichelle.com/slate2/)

30 COMMENTS

← Older Comments (https://www.rozmichelle.com/pipes-forks-dups/comment-page-2/#comments)

Roslyn McConnell (https://www.rozmichelle.com) says:

September 3, 2023 at 9:19 am (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-639440)

Hello! Thank you! You'd be correct if I sorted a list of numbers without using uniq, but since the `uniq -c` command renders each line as a string (because it prints the count followed by a space and the counted value), each line is interpreted as a string and sorted as such. :)

AKS says:

August 22, 2023 at 8:48 pm (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-639389)

Nice explanation. I am pretty sure in that last pipeline, you need to use sort -nr, -n for numerical sort to get the required result. But well that's besides the point.

Roslyn McConnell (https://www.rozmichelle.com) says:

July 27, 2023 at 9:41 am (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-639305)

Hi! Thanks for the kudos! Can you explain a bit more about what you mean by typing outfile? Are you referring to redirecting data to a file?

meroV says:

July 25, 2023 at 12:28 pm (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-639295)

Thanks for the article! This is by far the best illustration I have seen so far. I just have a follow up question. What is the exact mechanism when I type " outfile" in unix Bash? how come the command ends? given that /dev/random file itself never ends?

Leo González Pérez says:

April 14, 2023 at 2:41 am (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-638974)

Roslyn,
Just thank you very much. Your article is clear, complete and it was a very pleasant find for me. Kind regards!

Leon says:

March 27, 2023 at 7:57 am (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-638910)

Hey Roslyn,
thanks for the great article! It really helped me getting into this topic. I am curious how you implemented your multiple pipes example. Do you have the source code somewhere and are ok with sharing it? Thank you in any case! Have a nice day!

Marcel says:

March 17, 2023 at 3:27 am (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-638847)

Thank you so much for this explanation. Finally I was able to do my assignment at school.
There's one paragraph and a pictogram that could be added about pipe(fds):
– That dup2(fds[0], stdin) hands over a copy of fds[0]. An fds[0] can be closed, because the redirection of the stdin is still valid.
– Pictogram: fds = { READ, WRITE } and on the pipe a write and read.
In my assignment I have to take care of the stderror etc. as well, which further complicates the task. Perhaps with your skills to draw diagrams you can help to shed some light into that jungle as well?
Really great article and thank you again for all the time and effort you put into it!

Arisha says:
February 13, 2023 at 6:43 pm (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-638657)

Amazing Explanation!!! Very clear and organized.

Thanks a lot for the information.

Roslyn McConnell (https://www.rozmichelle.com) says:
February 6, 2023 at 6:06 pm (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-638620)

Hi Lisa! Thank you so much for your kind words! They mean a lot to me and it is feedback like yours that inspires me to keep making this type of content! :)

Lisa says:

January 25, 2023 at 10:45 am (https://www.rozmichelle.com/pipes-forks-dups/comment-page-3/#comment-638537)

Wow, the diagrams are so neat. This article helped me a lot to understand this topic better. Thank you so much for sharing this with us. Your experiences at Stanford University sound amazing, I wish I could study in such a nice, intelligent and creative environment. I found your website because I was looking for an article that explains handling pipes and I'm glad I found it, because your blog is inspiring, you are really a person that I look up to. Lots of appreciation, Lisa

← Older Comments (https://www.rozmichelle.com/pipes-forks-dups/comment-page-2/#comments)

LEAVE A REPLY

Comment *

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.

Post Comment

ABOUT *me*

Product Design Manager by day. Artist, developer, writer, home cook, and gamer by night! I love creating and experiencing beautiful things.

Read more (https://www.rozmichelle

resume

Download My Resume (https://www.rozmichelle content/uploads/rozresu

LATEST *pics*

Follow on Instag (https://www.instagram.com/

FOLLOW *me*

(https://www.facebook.com/rozmichelle) (https://www.twitter.com/rozmichelle) (https://www.pinterest.com/rozmichelle/) (https://www.instagram.com/rozmichelle)

2023 *goals*

To be an amazing mommy!