brandonwamboldt.ca

How Linux pipes work under the hood – Brandon Wamboldt

Published by

Piping is one of the core concepts of Linux & Unix based operating systems. Pipes allow you to chain together commands in a very elegant way, passing output from one program to the input of another to get a desired end result.

Here's a simple example of piping:

```
ls -la | sort | less
```

The above command gets a listing of the current directory using `ls`, sorts it alphabetically using the `sort` utility, and then paginates it for easier reading using `less`.

I'm not going to go into depth about pipes as I'll assume you already know what they are (at least in the context of bash pipes). Instead, I'm going to show how pipes are implemented under the hood.

How Pipes Are Implemented

Before I explain how bash does pipes, I'll explain how the kernel implements pipes (at a high level).

- Linux has a VFS (virtual file system) module called pipefs, that gets mounted in kernel space during boot
- pipefs is mounted alongside the root file system (`/`), not in it (pipe's root is `pipe:`)
- pipefs cannot be directly examined by the user unlike most file systems
- The entry point to pipefs is the `pipe(2)` syscall
- The `pipe(2)` syscall is used by shells and other programs to implement piping, and just creates a new file in pipefs, returning two file descriptors (one for the read end, opening using `O_RDONLY`, and one for the write end, opened using `O_WRONLY`)
- pipefs is stored an in-memory file system

Pipe I/O, buffering, and capacity

A pipe has a limited capacity in Linux. When the pipe is full, a `write(2)` will block (or fail if the `O_NONBLOCK` flag is set). Different implementations of pipes have different limits, so applications shouldn't rely on a pipe having a particular size. Applications should be designed to consume data as soon as it is available so the writing process doesn't block. That said, knowing the pipe size is useful. Since Linux 2.6.35, the default pipe capacity

is 65,536 bytes (it used to be the size of the page file, e.g. 4096 bytes in i386 architectures).

When a process attempts to read from an empty pipe, `read(2)` will block until data is available in the pipe. If all file descriptors pointing to the write end of the pipe have been closed, reading from the pipe will return EOF (`read(2)` will return 0).

If a process attempts to write to a full pipe, `write(2)` will block until enough data has been read from the pipe to allow the write call to succeed. If all file descriptors pointing to the read end of the pipe have been closed, writing to the pipe will raise the SIGPIPE signal. If this signal is ignored, `write(2)` fails with the error `EPIPE`.

All of this is important when understanding pipe performance. If a process A is writing data at roughly the same speed as process B is reading it, pipes work very well and are highly performance. An imbalance here can cause performance problems. See the next section for more information/examples.

How Shells Do Piping

Before continuing, you should be aware of how Linux creates new processes.

Shells implement piping in a manner very similar to how they implement redirection. Basically, the parent process calls `pipe(2)` once for each two processes that get piped together. In the example above, bash would need to call `pipe(2)` twice to create two pipes, one for piping ls to sort, and one to pipe sort to

less. Then, bash forks itself once for each process (3 times for our example). Each child will run one command. However, before the children run their commands, they will overwrite one of stdin or stdout (or both). In our above example, it will work like this:

- bash will create two pipes, one to pipe ls to sort, and one to pipe sort to less
- bash will fork itself 3 times (1 parent process and 3 children, for each command)
- child 1 (ls) will set it's stdout file descriptor to the write end of pipe A
- child 2 (sort) will set it's stdin file descriptor to the read end of pipe A (to read input from ls)
- child 2 (sort) will set it's stdout file descriptor to the write end of pipe B
- child 3 (less) will set it's stdin file descriptor to the read end of pipe B (to read input from sort)
- each child will run their commands

The kernel will automatically schedule processes so they roughly run in parallel. If child 1 writes too much to pipe A before child 2 has read it, child 2 will block for a while until child 2 has had time to read from the pipe. This normally allows for very high levels of efficiency as one process doesn't have to wait for the other to

complete to start processing data. Another reason for this is that pipes have a limited size (normally the size of a single page of memory).

Pipe Example Code

Here is a C example of how a program like bash might implement piping. My example is pretty simple, and accepts two arguments: a directory and a string to search for. It will run `ls -la` to get the contents of the directory, and pipe them to grep to search for the string.

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>

#define READ_END 0
#define WRITE_END 1

int main(int argc, char *argv[])
{
    int pid, pid_ls, pid_grep;
    int pipefd[2];

    // Syntax: test . filename
    if (argc < 3) {
        fprintf(stderr, "Please specify the
directory to search and the filename to search
```

```
for\n");
        return -1;
    }

    fprintf(stdotestut, "parent: Grepping %s for
%s\n", argv[1], argv[2]);

    // Create an unnamed pipe
    if (pipe(pipefd) == -1) {
        fprintf(stderr, "parent: Failed to
create pipe\n");
        return -1;
    }

    // Fork a process to run grep
    pid_grep = fork();

    if (pid_grep == -1) {
        fprintf(stderr, "parent: Could not fork
process to run grep\n");
        return -1;
    } else if (pid_grep == 0) {
        fprintf(stdout, "child: grep child will
now run\n");

        // Set fd[0] (stdin) to the read end of
the pipe
        if (dup2(pipefd[READ_END], STDIN_FILENO)
== -1) {
```

```
            fprintf(stderr, "child: grep dup2
failed\n");
            return -1;
        }

        // Close the pipe now that we've
duplicated it
        close(pipefd[READ_END]);
        close(pipefd[WRITE_END]);

        // Setup the arguments/environment to
call
        char *new_argv[] = { "/bin/grep",
argv[2], 0 };
        char *envp[] = { "HOME=/", "PATH=/bin:
/usr/bin", "USER=brandon", 0 };

        // Call execve(2) which will replace the
executable image of this
        // process
        execve(new_argv[0], &new_argv[0], envp);

        // Execution will never continue in this
process unless execve returns
        // because of an error
        fprintf(stderr, "child: Oops, grep
failed!\n");
        return -1;
    }
```

```
    // Fork a process to run ls
    pid_ls = fork();

    if (pid_ls == -1) {
        fprintf(stderr, "parent: Could not fork
process to run ls\n");
        return -1;
    } else if (pid_ls == 0) {
        fprintf(stdout, "child: ls child will
now run\n");
        fprintf(stdout,
"--------------------\n");

        // Set fd[1] (stdout) to the write end
of the pipe
        if (dup2(pipefd[WRITE_END],
STDOUT_FILENO) == -1) {
            fprintf(stderr, "ls dup2 failed\n");
            return -1;
        }

        // Close the pipe now that we've
duplicated it
        close(pipefd[READ_END]);
        close(pipefd[WRITE_END]);

        // Setup the arguments/environment to
call
```

```
        char *new_argv[] = { "/bin/ls", "-la",
argv[1], 0 };
        char *envp[] = { "HOME=/", "PATH=/bin:
/usr/bin", "USER=brandon", 0 };

        // Call execve(2) which will replace the
executable image of this
        // process
        execve(new_argv[0], &new_argv[0], envp);

        // Execution will never continue in this
process unless execve returns
        // because of an error
        fprintf(stderr, "child: Oops, ls
failed!\n");
        return -1;
    }

    // Parent doesn't need the pipes
    close(pipefd[READ_END]);
    close(pipefd[WRITE_END]);

    fprintf(stdout, "parent: Parent will now
wait for children to finish execution\n");

    // Wait for all children to finish
    while (wait(NULL) > 0);

    fprintf(stdout, "---------------------\n");
```

```
    fprintf(stdout, "parent: Children has
finished execution, parent is done\n");

    return 0;
}
```

I've commented it thoroughly, so hopefully it makes sense.

Named vs Unnamed Pipes

In the above examples, we've been using unnamed/anonymous pipes. These pipes are temporary, and are discarded once your program finishes or all of their file descriptors are closed. They are the most common type of pipe.

Named pipes, also known as FIFOs (for first in, first out), get created as a named file on your hard disk. They allow multiple unrelated programs to open and use them. You can have multiple writers quite easily, with one reader, for a very simplistic client-server type design. For example, nagios does this, with the master process reading a named pipe, and every child process writing commands to the named pipe.

Named pipes are creating using the mkfifo command or syscall. Example:

```
mkfifo ~/test_pipe
```

Other than their creation, they work pretty much the same as unnamed pipes. Once you create them, you can open them using

`open(2)`. You must open the read end using `O_RDONLY` or the write end using `O_WRONLY`. Most operating systems implement unidirectional pipes, so you can't open them in both read/write mode.

FIFOs are often used as a unidirectional IPC technique, for a system with multiple processes. A multithreaded application may also use named or unnamed pipes, as well as other IPC techniques such as shared memory segments.

FIFOs are created as a single inode, with the property `i_pipe` set as a reference to the actual pipe. While the name may exist on your filesystem, pipes don't cause I/O to the underlying device, as once the inode is read, FIFOs behave like unnamed pipes and operate in-memory.