## Resumo

os programas em linux e unix tem 3 arquivos padrões (de IO)que são abordados logo nas primeiras linhas do texto. os arquivos padrões são os que mostram as mensagens de entrada e saída dos programas, exemplo, o input vem do teclado e mouse e o output vai pra tela do pc. uma pipe em bash faz com que, ao invez do output vá para a tela, ele seja encaminhado para outro programa como uma variável de início. por exemplo vc tem um programa que no final da execução te retorna 5 como resultado. uma bash pipe pega esse 5 e transforma ele em variável do programa.

n Linux (and Unix in general) each process has three default file descriptors:

1. fd #0 Represents the standard input of the process
2. fd #1 Represents the standard output of the process
3. fd #2 Represents the standard error output of the process

Normally, when you run a simple program these file descriptors by default are configured as following:

1. default input is read from the keyboard
2. Standard output is configured to be the monitor
3. Standard error is configured to be the monitor also

Bash provides several operators to change this behavior (take a look to the >, >> and < operators for example). Thus, you can redirect the output to something other than the standard output or read your input from other stream different than the keyboard. Specially interesting the case when two programs are *collaborating* in such way that one uses the output of the other as its input. To make this collaboration easy Bash provides the pipe operator |. Please note the usage of collaboration instead of *chaining*. I avoided the usage of this term since in fact a pipe **is not sequential**. A normal command line with pipes has the following aspect:

> program\_1 | program\_2 | ... | program\_n

The above command line is a little bit misleading: user could think that program\_2 gets its input once the program\_1 has finished its execution, which is not correct. In fact, what bash does is to launch **ALL** the programs in parallel and it configures the inputs outputs accordingly so every program gets its input from the previous one and delivers its output to the next one (in the command line established order).

Following is a simple example from [Creating pipe in C](http://www.tldp.org/LDP/lpg/node11.html) of creating a pipe between a parent and child process. The important part is the call to the pipe() and how the parent closes fd[1](http://www.tldp.org/LDP/lpg/node11.html) (writing side) and how the child closes fd[1](http://www.tldp.org/LDP/lpg/node11.html) (writing side). Please, note that the pipe is a **unidirectional** communication channel. Thus, data can only flow in one direction: fd[1](http://www.tldp.org/LDP/lpg/node11.html) towards fd[0]. For more information take a look to the manual page of pipe().

#include <stdio.h> #include <unistd.h> #include <sys/types.h> int main(void) { int fd[2], nbytes; pid\_t childpid; char string[] = "Hello, world!\n"; char readbuffer[80]; pipe(fd); if((childpid = fork()) == -1) { perror("fork"); exit(1); } if(childpid == 0) { /\* Child process closes up input side of pipe \*/ close(fd[0]); /\* Send "string" through the output side of pipe \*/ write(fd[1], string, (strlen(string)+1)); exit(0); } else { /\* Parent process closes up output side of pipe \*/ close(fd[1]); /\* Read in a string from the pipe \*/ nbytes = read(fd[0], readbuffer, sizeof(readbuffer)); printf("Received string: %s", readbuffer); } return(0); }

Last but not least, when you have a command line in the form:

> program\_1 | program\_2 | program\_3

The return code of the whole line is set to the **last** command. In this case program\_3. If you would like to get an intermediate return code you have to set the **pipefail** or get it from the **PIPESTATUS**.

<https://stackoverflow.com/questions/9834086/what-is-a-simple-explanation-for-how-pipes-work-in-bash>

# Introduction to Named Pipes

[SysAdmin](https://www.linuxjournal.com/tag/sysadmin)

by Andy Vaught on September 1, 1997

One of the fundamental features that makes Linux and other Unices useful is the “pipe”. Pipes allow separate processes to communicate without having been designed explicitly to work together. This allows tools quite narrow in their function to be combined in complex ways.

A simple example of using a pipe is the command:

ls | grep x

When bash examines the command line, it finds the vertical bar character **|** that separates the two commands. Bash and other shells run both commands, connecting the output of the first to the input of the second. The **ls** program produces a list of files in the current directory, while the **grep** program reads the output of **ls** and prints only those lines containing the letter **x**.

The above, familiar to most Unix users, is an example of an “unnamed pipe”. The pipe exists only inside the kernel and cannot be accessed by processes that created it, in this case, the bash shell. For those who don't already know, a parent process is the first process started by a program that in turn creates separate child processes that execute the program.

The other sort of pipe is a “named” pipe, which is sometimes called a FIFO. FIFO stands for “First In, First Out” and refers to the property that the order of bytes going in is the same coming out. The “name” of a named pipe is actually a file name within the file system. Pipes are shown by **ls** as any other file with a couple of differences:

% ls -l fifo1

prw-r--r-- 1 andy users 0 Jan 22 23:11 fifo1|

The **p** in the leftmost column indicates that fifo1 is a pipe. The rest of the permission bits control who can read or write to the pipe just like a regular file. On systems with a modern **ls**, the **|** character at the end of the file name is another clue, and on Linux systems with the color option enabled, **fifo|** is printed in red by default.

On older Linux systems, named pipes are created by the **mknod** program, usually located in the /etc directory. On more modern systems, **mkfifo** is a standard utility. The **mkfifo** program takes one or more file names as arguments for this task and creates pipes with those names. For example, to create a named pipe with the name **pipe1** give the command:

mkfifo pipe

The simplest way to show how named pipes work is with an example. Suppose we've created **pipe** as shown above. In one virtual console1, type:

ls -l > pipe1

and in another type:

cat < pipe

Voila! The output of the command run on the first console shows up on the second console. Note that the order in which you run the commands doesn't matter.

If you haven't used virtual consoles before, see the article “Keyboards, Consoles and VT Cruising” by John M. Fisk in the November 1996 *Linux Journal*.

If you watch closely, you'll notice that the first command you run appears to hang. This happens because the other end of the pipe is not yet connected, and so the kernel suspends the first process until the second process opens the pipe. In Unix jargon, the process is said to be “blocked”, since it is waiting for something to happen.

One very useful application of named pipes is to allow totally unrelated programs to communicate with each other. For example, a program that services requests of some sort (print files, access a database) could open the pipe for reading. Then, another process could make a request by opening the pipe and writing a command. That is, the “server” can perform a task on behalf of the “client”. Blocking can also happen if the client isn't writing, or the server isn't reading.

Pipe Madness

Create two named pipes, pipe1 and pipe2. Run the commands:

echo -n x | cat - pipe1 > pipe2 &

cat <pipe2 > pipe1

On screen, it will not appear that anything is happening, but if you run **top** (a command similar to **ps** for showing process status), you'll see that both **cat** programs are running like crazy copying the letter **x** back and forth in an endless loop.

After you press ctrl-C to get out of the loop, you may receive the message “**broken pipe**”. This error occurs when a process writing to a pipe when the process reading the pipe closes its end. Since the reader is gone, the data has no place to go. Normally, the writer will finish writing its data and close the pipe. At this point, the reader sees the **EOF** (end of file) and executes the request.

Whether or not the “broken pipe” message is issued depends on events at the exact instant the ctrl-C is pressed. If the second **cat** has just read the **x**, pressing ctrl-C stops the second **cat**, **pipe1** is closed and the first **cat** stops quietly, i.e., without a message. On the other hand, if the second **cat** is waiting for the first to write the **x**, ctrl-C causes **pipe2** to close before the first **cat** can write to it, and the error message is issued. This sort of random behavior is known as a “race condition”.

Command Substitution

Bash uses named pipes in a really neat way. Recall that when you enclose a command in parenthesis, the command is actually run in a “subshell”; that is, the shell clones itself and the clone interprets the command(s) within the parenthesis. Since the outer shell is running only a single “command”, the output of a complete set of commands can be redirected as a unit. For example, the command:

(ls -l; ls -l) >ls.out

writes two copies of the current directory listing to the file ls.out.

Command substitution occurs when you put a **<** or **>** in front of the left parenthesis. For instance, typing the command:

cat <(ls -l)

results in the command **ls -l** executing in a subshell as usual, but redirects the output to a temporary named pipe, which bash creates, names and later deletes. Therefore, **cat** has a valid file name to read from, and we see the output of **ls -l**, taking one more step than usual to do so. Similarly, giving **>(*commands*)** results in Bash naming a temporary pipe, which the commands inside the parenthesis read for input.

If you want to see whether two directories contain the same file names, run the single command:

cmp <(ls /dir1) <(ls /dir2)

The compare program **cmp** will see the names of two files which it will read and compare.

Command substitution also makes the **tee** command (used to view and save the output of a command) much more useful in that you can cause a single stream of input to be read by multiple readers without resorting to temporary files—bash does all the work for you. The command:

ls | tee >(grep foo | wc >foo.count) \

>(grep bar | wc >bar.count) \

| grep baz | wc >baz.count

counts the number of occurrences of **foo**, **bar** and **baz** in the output of **ls** and writes this information to three separate files. Command substitutions can even be nested:

cat <(cat <(cat <(ls -l))))

works as a very roundabout way to list the current directory.

As you can see, while the unnamed pipes allow simple commands to be strung together, named pipes, with a little help from bash, allow whole trees of pipes to be created. The possibilities are limited only by your imagination.