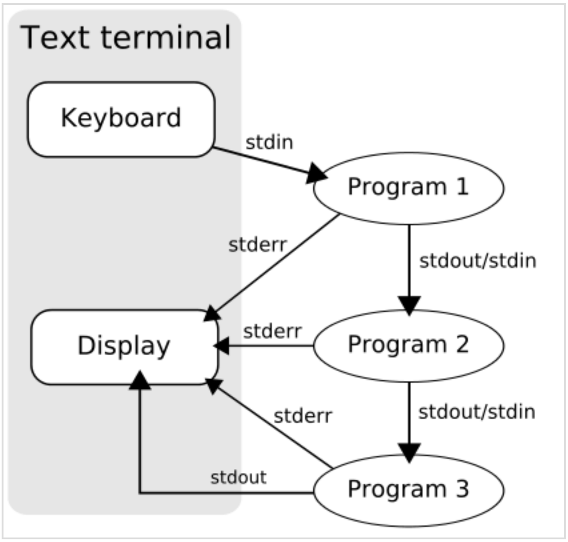
WIKIPEDIA
The Free Encyclopedia

Pipeline (Unix)

In Unix-like computer operating systems, a **pipeline** is a mechanism for inter-process communication using message passing. A pipeline is a set of processes chained together by their standard streams, so that the output text of each process (*stdout*) is passed directly as input (*stdin*) to the next one. The second process is started as the first process is still executing, and they are executed concurrently. The concept of pipelines was championed by Douglas McIlroy at Unix's ancestral home of Bell Labs, during the development of Unix, shaping its toolbox philosophy.[1][2] It is named by analogy to a physical pipeline. A key feature of these pipelines is their "hiding of internals" (Ritchie & Thompson, 1974). This in turn allows for more clarity and simplicity in the system.



A pipeline of three program processes run on a text terminal

This article is about anonymous pipes, where data written by one process is buffered by the operating system until it is read by the next process, and this uni-directional channel disappears when the processes are completed. This differs from named pipes, where messages are passed to or from a pipe that is named by making it a file, and remains after the processes are completed. The standard shell syntax for anonymous pipes is to list multiple commands, separated by vertical bars ("pipes" in common Unix verbiage):

```
command1 | command2 | command3
```

For example, to list files in the current directory (`ls`), retain only the lines of `ls` output containing the string `"key"` (`grep`), and view the result in a scrolling page (`less`), a user types the following into the command line of a terminal:

```
ls -l | grep key | less
```

The command `ls -l` is executed as a process, the output (stdout) of which is piped to the input (stdin) of the process for `grep key`; and likewise for the process for `less`. Each process takes input from the previous process and produces output for the next process via *standard streams*. Each **|** tells the shell to connect the standard output of the command on the left to the standard input of the command on the right by an inter-process communication mechanism called an (anonymous) pipe, implemented in the operating system. Pipes are unidirectional; data flows through the pipeline from left to right.

Example

Below is an example of a pipeline that implements a kind of spell checker for the web resource indicated by a URL. An explanation of what it does follows.

```
curl "https://en.wikipedia.org/wiki/Pipeline_(Unix)" |
sed 's/[^a-zA-Z ]/ /g' |
tr 'A-Z ' 'a-z\n' |
grep '[a-z]' |
sort -u |
comm -23 - <(sort /usr/share/dict/words) |
less
```

1. **`curl`** obtains the HTML contents of a web page (could use `wget` on some systems).

2. **`sed`** replaces all characters (from the web page's content) that are not spaces or letters, with spaces. (Newlines are preserved.)
3. **`tr`** changes all of the uppercase letters into lowercase and converts the spaces in the lines of text to newlines (each 'word' is now on a separate line).
4. **`grep`** includes only lines that contain at least one lowercase alphabetical character (removing any blank lines).
5. **`sort`** sorts the list of 'words' into alphabetical order, and the `-u` switch removes duplicates.
6. **`comm`** finds lines in common between two files, `-23` suppresses lines unique to the second file, and those that are common to both, leaving only those that are found only in the first file named. The `-` in place of a filename causes `comm` to use its standard input (from the pipe line in this case). `sort /usr/share/dict/words` sorts the contents of the `words` file alphabetically, as `comm` expects, and `<( ... )` outputs the results to a temporary file (via process substitution), which `comm` reads. The result is a list of words (lines) that are not found in /usr/share/dict/words.
7. **`less`** allows the user to page through the results.

Pipelines in command line interfaces

All widely used Unix shells have a special syntax construct for the creation of pipelines. In all usage one writes the commands in sequence, separated by the ASCII vertical bar character `|` (which, for this reason, is often called "pipe character"). The shell starts the processes and arranges for the necessary connections between their standard streams (including some amount of buffer storage).

Error stream

By default, the standard error streams ("stderr") of the processes in a pipeline are not passed on through the pipe; instead, they are merged and directed to the console. However, many shells have additional syntax for changing this behavior. In the csh shell, for instance, using `|&` instead of `|` signifies that the standard error stream should also be merged with the standard output and fed to the next process. The Bash shell can also merge standard error with `|&` since version 4.0[3] or using `2>&1`, as well as redirect it to a different file.

Pipemill

In the most commonly used simple pipelines the shell connects a series of sub-processes via pipes, and executes external commands within each sub-process. Thus the shell itself is doing no direct processing of the data flowing through the pipeline.

However, it's possible for the shell to perform processing directly, using a so-called **mill** or **pipemill** (since a `while` command is used to "mill" over the results from the initial command). This construct generally looks something like:

```
command | while read -r var1 var2 ...; do
    # process each line, using variables as parsed into var1, var2, etc
    # (note that this may be a subshell: var1, var2 etc will not be available
    # after the while loop terminates; some shells, such as zsh and newer
    # versions of Korn shell, process the commands to the left of the pipe
    # operator in a subshell)
    done
```

Such pipemill may not perform as intended if the body of the loop includes commands, such as `cat` and `ssh`, that read from `stdin`:[4] on the loop's first iteration, such a program (let's call it *the drain*) will read the remaining output from `command`, and the loop will then terminate (with results depending on the specifics of the drain). There are a couple of possible ways to avoid this behavior. First, some drains support an option to disable reading from `stdin` (e.g. `ssh -n`). Alternatively, if the drain does not *need* to read any input from `stdin` to do something useful, it can be given `< /dev/null` as input.

As all components of a pipe are run in parallel, a shell typically forks a subprocess (a subshell) to handle its contents, making it impossible to propagate variable changes to the outside shell environment. To remedy this issue, the "pipemill" can instead be fed from a here document containing a command substitution, which waits for the pipeline to finish running before milling through the contents. Alternatively, a named pipe or a process substitution can be used for parallel execution. GNU bash also has a `lastpipe` option to disable forking for the last pipe component.[5]

Creating pipelines programmatically

Pipelines can be created under program control. The Unix `pipe()` system call asks the operating system to construct a new anonymous pipe object. This results in two new, opened file descriptors in the process: the read-only end of the pipe, and the write-only end. The pipe ends appear to be normal, anonymous file descriptors, except that they have no ability to seek.

To avoid deadlock and exploit parallelism, the Unix process with one or more new pipes will then, generally, call `fork()` to create new processes. Each process will then close the end(s) of the pipe that it will not be using before producing or consuming any data. Alternatively, a process might create new threads and use the pipe to communicate between them.

Named pipes may also be created using `mkfifo()` or `mknod()` and then presented as the input or output file to programs as they are invoked. They allow multi-path pipes to be created, and are especially effective when combined with standard error redirection, or with `tee`.

Implementation

In most Unix-like systems, all processes of a pipeline are started at the same time, with their streams appropriately connected, and managed by the scheduler together with all other processes running on the machine. An important aspect of this, setting Unix pipes apart from other pipe implementations, is the concept of buffering: for example a sending program may produce 5000 bytes per second, and a receiving program may only be able to accept 100 bytes per second, but no data is lost. Instead, the output of the sending program is held in the buffer. When the receiving program is ready to read data, the next program in the pipeline reads from the buffer. If the buffer is filled, the sending program is stopped (blocked) until at least some data is removed from the buffer by the receiver. In Linux, the size of the buffer is 65,536 bytes (64KiB). An open source third-party filter called bfr (https://linux.die.net/man/1/bfr) is available to provide larger buffers if required.

Network pipes

Tools like netcat and socat can connect pipes to TCP/IP sockets.

History

The pipeline concept was invented by Douglas McIlroy[6] and first described in the man pages of Version 3 Unix.[7][8] McIlroy noticed that much of the time command shells passed the output file from one program as input to another.

His ideas were implemented in 1973 when ("in one feverish night", wrote McIlroy) Ken Thompson added the `pipe()` system call and pipes to the shell and several utilities in Version 3 Unix. "The next day", McIlroy continued, "saw an unforgettable orgy of one-liners as everybody joined in the excitement of plumbing." McIlroy also credits Thompson with the `|` notation, which greatly simplified the description of pipe syntax in Version 4.[9][7]

Although developed independently, Unix pipes are related to, and were preceded by, the 'communication files' developed by Ken Lochner [10] in the 1960s for the Dartmouth Time Sharing System.[11]

In Tony Hoare's communicating sequential processes (CSP) McIlroy's pipes are further developed.[12]

The robot in the icon for Apple's Automator, which also uses a pipeline concept to chain repetitive commands together, holds a pipe in homage to the original Unix concept.

Other operating systems

This feature of Unix was borrowed by other operating systems, such as MS-DOS and the CMS Pipelines package on VM/CMS and MVS, and eventually came to be designated the pipes and filters design pattern of software engineering.

See also

- Everything is a file – describes one of the defining features of Unix; pipelines act on "files" in the Unix sense
- Anonymous pipe – a FIFO structure used for interprocess communication
- GStreamer – a pipeline-based multimedia framework
- CMS Pipelines
- Iteratee
- Named pipe – persistent pipes used for interprocess communication
- Process substitution — shell syntax for connecting multiple pipes to a process
- GNU parallel
- Pipeline (computing) – other computer-related pipelines
- Redirection (computing)
- Tee (command) – a general command for tapping data from a pipeline
- XML pipeline – for processing of XML files
- xargs

References

1. Mahoney, Michael S. "The Unix Oral History Project: Release.0, The Beginning" (http://www.princeton.edu/~hos/Mahoney/expotape.htm). "McIlroy: It was one of the only places where I very nearly exerted managerial control over Unix, was pushing for those things, yes."
2. "Prophetic Petroglyphs" (http://www.bell-labs.com/usr/dmr/www/mdmpipe.html). *www.bell-labs.com*. Archived (https://web.archive.org/web/19990508221104/http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.html) from the original on 8 May 1999. Retrieved 22 May 2022.
3. "Bash release notes" (https://tiswww.case.edu/php/chet/bash/NEWS). *tiswww.case.edu*. Retrieved 2017-06-14.
4. "Shell Loop Interaction with SSH" (https://web.archive.org/web/20120306135439/http://72.14.189.113/howto/shell/while-ssh/). 6 March 2012. Archived from the original (http://72.14.189.113/howto/shell/while-ssh/) on 6 March 2012.
5. John1024. "How can I store the "find" command results as an array in Bash" (https://stackoverflow.com/a/23357277). *Stack Overflow*.
6. "The Creation of the UNIX Operating System" (https://web.archive.org/web/20040914025332/http://csdev.cas.upm.edu.ph/~pfalcone/compsci/unix/unix-history1.html). Bell Labs. Archived from the original (http://csdev.cas.upm.edu.ph/~pfalcone/compsci/unix/unix-history1.html) on September 14, 2004.
7. McIlroy, M. D. (1987). *A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971–1986* (http://www.cs.dartmouth.edu/~doug/reader.pdf) (PDF) (Technical report). CSTR. Bell Labs. 139.
8. Thompson K, Ritchie DM (February 1973). *UNIX Programmer’s Manual Third Edition* (https://dspinellis.github.io/unix-v3man/v3man.pdf#page=178) (PDF) (Technical report) (3rd ed.). Bell Labs. p. 178.
9. "Pipes: A Brief Introduction" (http://www.linfo.org/pipe.html). The Linux Information Project. August 23, 2006 [Created April 29, 2004]. Retrieved January 7, 2024.

10. "Dartmouth Timesharing" (http://www.cs.rit.edu/~swm/history/DTSS.doc) (DOC). *Rochester Institute of Technology*. Retrieved January 7, 2024.
11. "Data" (https://www.bell-labs.com/usr/dmr/www/hist.html). *www.bell-labs.com*. Archived (https://web.archive.org/web/19990220165130/http://cm.bell-labs.com/who/dmr/hist.html) from the original on 20 February 1999. Retrieved 22 May 2022.
12. Cox, Russ. "Bell Labs and CSP Threads" (https://swtch.com/~rsc/thread/). *Swtchboard*. Retrieved January 7, 2024.

- Sal Soghoian on MacBreak Episode 3 "Enter the Automatrix"

External links

- History of Unix pipe notation (https://www.bell-labs.com/usr/dmr/www/hist.html#pipes) Archived (https://web.archive.org/web/20150408054606/http://cm.bell-labs.com/cm/cs/who/dmr/hist.html#pipes) 2015-04-08 at the Wayback Machine
  - Doug McIlroy's original 1964 memo (http://doc.cat-v.org/unix/pipes/), proposing the concept of a pipe for the first time
- `pipe (https://www.opengroup.org/onlinepubs/9699919799/functions/pipe.html)`: create an interprocess channel – System Interfaces Reference, The Single UNIX Specification, Version 4 from The Open Group
- Pipes: A Brief Introduction (http://www.linfo.org/pipe.html) by The Linux Information Project (LINFO)
- Unix Pipes – powerful and elegant programming paradigm (Softpanorama) (http://www.softpanorama.org/Scripting/pipes.shtml)
- *Ad Hoc Data Analysis From The Unix Command Line* at Wikibooks (https://en.wikibooks.org/w/index.php?title=Ad_Hoc_Data_Analysis_From_The_Unix_Command_Line) – Shows how to use pipelines composed of simple filters to do complex data analysis.
- Use And Abuse Of Pipes With Audio Data (https://web.archive.org/web/20170911203245/https://debian-administration.org/article/145/use_and_abuse_of_pipes_with_audio_data) – Gives an introduction to using and abusing pipes with netcat, nettee and fifos to play audio across a network.
- stackoverflow.com (https://stackoverflow.com/questions/19122/bash-pipe-handling) – A Q&A about bash pipeline handling.