Pipe and Filter Architecture

Subjects: Computer Science, Software Engineering
Contributor: HandWiki Huang

In software engineering, a pipeline consists of a chain of processing elements (processes, threads, coroutines, functions, etc.), arranged so that the output of each element is the input of the next; the name is by analogy to a physical pipeline. Usually some amount of buffering is provided between consecutive elements. The information that flows in these pipelines is often a stream of records, bytes, or bits, and the elements of a pipeline may be called filters; this is also called the pipes and filters design pattern. Connecting elements into a pipeline is analogous to function composition. Narrowly speaking, a pipeline is linear and one-directional, though sometimes the term is applied to more general flows. For example, a primarily one-directional pipeline may have some communication in the other direction, known as a return channel or backchannel, as in the lexer hack, or a pipeline may be fully bi-directional. Flows with one-directional tree and directed acyclic graph topologies behave similarly to (linear) pipelines – the lack of cycles makes them simple – and thus may be loosely referred to as "pipelines".

Keywords: directed acyclic graph ; filters design ; return channel

1. Implementation

Pipelines are often implemented in a multitasking OS, by launching all elements at the same time as processes, and automatically servicing the data read requests by each process with the data written by the upstream process – this can be called a *multiprocessed pipeline.* In this way, the CPU will be naturally switched among the processes by the scheduler so as to minimize its idle time. In other common models, elements are implemented as lightweight threads or as coroutines to reduce the OS overhead often involved with processes. Depending upon the OS, threads may be scheduled directly by the OS or by a thread manager. Coroutines are always scheduled by a coroutine manager of some form.

Usually, read and write requests are blocking operations, which means that the execution of the source process, upon writing, is suspended until all data could be written to the destination process, and, likewise, the execution of the destination process, upon reading, is suspended until at least some of the requested data could be obtained from the source process. This cannot lead to a deadlock, where both processes would wait indefinitely for each other to respond, since at least one of the two processes will soon thereafter have its request serviced by the operating system, and continue to run.

For performance, most operating systems implementing pipes use pipe buffers, which allow the source process to provide more data than the destination process is currently able or willing to receive. Under most Unices and Unix-like operating systems, a special command is also available which implements a pipe buffer of potentially much larger and configurable size, typically called "buffer". This command can be useful if the destination process is significantly slower than the source process, but it is anyway desired that the source process can complete its task as soon as possible. E.g., if the source process consists of a command which reads an audio track from a CD and the destination process consists of a command which compresses the waveform audio data to a format like MP3. In this case, buffering the entire track in a pipe buffer would allow the CD drive to spin down more quickly, and enable the user to remove the CD from the drive before the encoding process has finished.

Such a buffer command can be implemented using system calls for reading and writing data. Wasteful busy waiting can be avoided by using facilities such as poll or select or multithreading.

Some notable examples of pipeline software systems include:

- RaftLib – C/C++ Apache 2.0 License

2. VM/CMS and Z/OS

CMS Pipelines is a port of the pipeline idea to VM/CMS and z/OS systems. It supports much more complex pipeline structures than Unix shells, with steps taking multiple input streams and producing multiple output streams. (Such functionality is supported by the Unix kernel, but few programs use it as it makes for complicated syntax and blocking modes, although some shells do support it via arbitrary file descriptor assignment).

Traditional application programs on IBM mainframe operating systems have no standard input and output streams to allow redirection or piping. Instead of spawning processes with external programs, CMS Pipelines features a lightweight dispatcher to concurrently execute instances of built-in programs to run the pipeline. More than 200 built-in programs that implement typical UNIX utilities and interface to devices and operating system services. In addition to the built-in programs, CMS Pipelines defines a framework to allow user-written REXX programs with input and output streams that can be used in the pipeline.

Data on IBM mainframes typically resides in a Record-oriented filesystem and connected I/O devices operate in record mode rather than stream mode. As a consequence, data in CMS Pipelines is handled in record mode. For text files, a record holds one line of text. In general, CMS Pipelines does not buffer the data but passes records of data in a lock-step fashion from one program to the next. This ensures a deterministic flow of data through a network of interconnected pipelines.

3. Object Pipelines

Beside byte stream-based pipelines, there are also object pipelines. In an object pipeline, processing elements output objects instead of text. Windows PowerShell includes an internal object pipeline that transfers .NET objects between functions within the PowerShell runtime. Channels, found in the Limbo programming language are other examples of this metaphor.

4. Pipelines in GUIs

Graphical environments such as RISC OS and ROX Desktop also make use of pipelines. Rather than providing a save dialog box containing a file manager to let the user specify where a program should write data, RISC OS and ROX provide a save dialog box containing an icon (and a field to specify the name). The destination is specified by dragging and dropping the icon. The user can drop the icon anywhere an already-saved file could be dropped, including onto icons of other programs. If the icon is dropped onto a program's icon, it's loaded and the contents that would otherwise have been saved are passed in on the new program's standard input stream.

For instance, a user browsing the world-wide web might come across a .gz compressed image which they want to edit and re-upload. Using GUI pipelines, they could drag the link to their de-archiving program, drag the icon representing the extracted contents to their image editor, edit it, open the save as dialog, and drag its icon to their uploading software.

Conceptually, this method could be used with a conventional save dialog box, but this would require the user's programs to have an obvious and easily accessible location in the filesystem that can be navigated to. In practice, this is often not the case, so GUI pipelines are rare.

5. Other Considerations

The name "pipeline" comes from a rough analogy with physical plumbing in that a pipeline usually[1] allows information to flow in only one direction, like water often flows in a pipe.

Pipes and filters can be viewed as a form of functional programming, using byte streams as data objects; more specifically, they can be seen as a particular form of monad for I/O.[2]

The concept of pipeline is also central to the Cocoon web development framework or to any XProc (the W3C Standards) implementations, where it allows a source stream to be modified before eventual display.

This pattern encourages the use of text streams as the input and output of programs. This reliance on text has to be accounted when creating graphic shells to text programs.

References

1. There are exceptions, such as "broken pipe" signals.
2. "Monadic I/O and UNIX shell programming". http://okmij.org/ftp/Computation/monadic-shell.html

---