Shell

Peter J. Denning

5/20/19

In the commercial operating systems of the 1950s and 1960s, programmers had to work with two languages: their programming language (such as FORTRAN or ALGOL) for their applications and the operating system's command language. The command language was a set of statements to the operating system instructing it on how to set up for the execution of a program – for example, how much memory, which tapes needed to be mounted on tape drives, and which external files needed to be opened. These command languages were extremely hard to use. Many users spent as much time debugging their commands as they did debugging their programs. It was a mess.

The designers of Multics in the mid 1960s set out to put an end to this for time-sharing systems. Those systems accommodated a user community centered around a shared file system. In those systems, users cycled between two states: *thinking* about what command to issue next, and *waiting* for a current command to complete. The file system contained command libraries – programs that perform simple commands, such as "list all the files in a directory". The designers saw no difference between a system-programmed command and a user-programmed command: both were executable files that could be called into action by typing their names. They called the command interpreter program the shell because it was like a eggshell separating the innards of the system from the outside world. The shell contained no commands of its own. It simply interpreted what the user typed as a command name, and invoked an executable program of that name from the library.

The first Unix system simplified the shell program to just a few pages of C code. (You can find it with a Web search for "sh.c".) The user interface was not at all complex. It was a dramatic change from the command language interpreters of early operating systems.

The simplicity of the shell rested on the simplicity of the execution model. The simplest possible shell only allowed you to type a single command name followed by a list of arguments:

```
cmd args
```

It is easy to build a parser that picks out the command name and calls a library program of that name with the given arguments. In our terminology, you type the name of a command and the shell asks the operating system to construct a virtual machine (process) running that program. The shell cycled between two states: listening for a command and executing the command. These two shell states directly mirror the user's think state (issuing a new command) and waiting state (waiting for a response). Thus when the process was running, its parent shell was

waiting, and when the process was finished, its parent shell was awake and listening to the user.

Shell designers soon saw that a simple extension to the above idea would make it more powerful. The extension was the pipeline. A series of processes could be executed simultaneously with the output of one feeding into the input of the next:

```
cmd args | cmd args | ... | cmd args
```

The vertical bar is a "pipe" symbol and represents a channel that conveys the output of one process to the input of the next. Pipelines allow programmers to rapidly construct more complex functions from building-blocks in the library.

To accommodate this, the virtual machines implementing processes use a standard input and output that could be connected to any file or pipe. Internally, the virtual machine reads from its standard input and writes into its standard output, leaving the decision of what to plug into standard input or output to run time. Designers noticed that files and pipes are instances of bit-streams; they said that any bit-stream object can be connected to a virtual machine's input or output. With these generalizations, a pipeline consists of a sequence of processes connected by pipes; the initial process may have either its parent's input or a file connected; the final process may have either its parent's output or a file connected.

```
cmd [args][< file] [| cmd [args]]* [> file]
```

As an example, let's consider the Unix cat command. This command takes a series of files as arguments and generates an output stream consisting of the listed files run together one after the next. When supplied with a standard input, this command ignores its arguments and instead passes the input stream directly to the output stream. Thus

```
cat A B
```

generates the concatenation of files `A` and `B`,

```
cat A B > C
```

places that stream into file `C` and

```
cat A B < D > C
```

copies the contents of `D` into `C`. The command

```
cat A B | sort -d | print
```

concatenates the files `A` and `B`, sorts the result by lines in descending order of the first words, and streams that result to the printer.

Even with all these embellishments the original shell program (sh.c) was small. Modern shells are bigger because they incorporate a few commands as built-in commands to speed them up (instead of loading them from the library) and because they include a scripting language to build multi-line shell commands.