Evolution of shells in Linux

From Bourne to Bash and beyond

M. Tim Jones
Platform Architect
Intel

09 December 2011
(First published 06 December 2011)

Pointing and clicking is fine for most day-to-day computing tasks, but to really take advantage of the strengths of Linux over other environments, you eventually need to crack the shell and enter the command line. Lots of command shells are available, from Bash and Korn to C shell and various exotic and strange shells. Learn which shell is right for you. *[Note: Minor corrections were made to Listings 2 and 3.]*

Connect with Tim

Tim is one of our most popular and prolific authors. Browse all of Tim's articles on developerWorks. Check out Tim's profile and connect with him, other authors, and fellow developers in the developerWorks community.

Shells are like editors: Everyone has a favorite and vehemently defends that choice (and tells you why you should switch). True, shells can offer different capabilities, but they all implement core ideas that were developed decades ago.

My first experience with a modern shell came in the 1980s, when I was developing software on SunOS. Once I learned the capability to apply output from one program as input to another (even doing this multiple times in a chain), I had a simple and efficient way to create filters and transformations. The core idea provided a way to build simple tools that were flexible enough to be applied with other tools in useful combinations. In this way, shells provided not only a way to interact with the kernel and devices but also integrated services (such as pipes and filters) that are now common design patterns in software development.

Let's begin with a short history of modern shells, and then explore some of the useful and exotic shells available for Linux today.

A history of shells

Shells as little languages

Shells are specialized, domain-specific languages (*little languages*) that implement a specific use model—in this case, providing an interface to an operating system. In addition to text-

based operating system shells, you can find graphical user interface shells as well as shells for languages (such as the Python shell or Ruby's irb). The shell idea has also been applied to Web searching through a web front end called *goosh.* This shell over Google permits command-line searching with Google using commands such as `search`, `more`, and `go`.

Shells—or *command-line interpreters*—have a long history, but this discussion begins with the first UNIX® shell. Ken Thompson (of Bell Labs) developed the first shell for UNIX called the *V6 shell* in 1971. Similar to its predecessor in Multics, this shell (/bin/sh) was an independent user program that executed outside of the kernel. Concepts like globbing (pattern matching for parameter expansion, such as *.txt) were implemented in a separate utility called *glob,* as was the `if` command to evaluate conditional expressions. This separation kept the shell small, at under 900 lines of `c` source (see Resources for a link to the original source).

The shell introduced a compact syntax for redirection (`< >` and `>>`) and piping (`|` or `^`) that has survived into modern shells. You can also find support for invoking sequential commands (with `;`) and asynchronous commands (with `&`).

What the Thompson shell lacked was the ability to script. Its sole purpose was as an interactive shell (command interpreter) to invoke commands and view results.

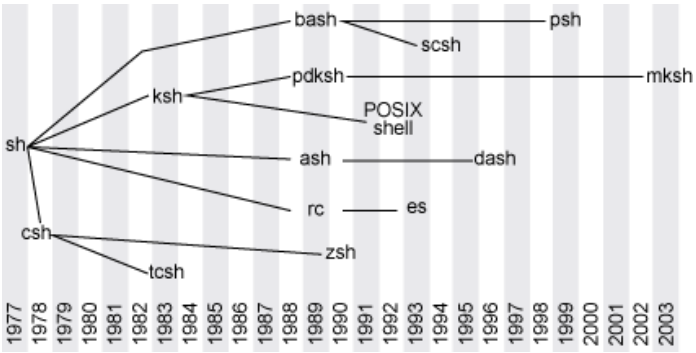
UNIX shells since 1977

Beyond the Thompson shell, we begin our look at modern shells in 1977, when the Bourne shell was introduced. The Bourne shell, created by Stephen Bourne at AT&T Bell Labs for V7 UNIX, remains a useful shell today (in some cases, as the default root shell). The author developed the Bourne shell after working on an ALGOL68 compiler, so you'll find its grammar more similar to the Algorithmic Language (ALGOL) than other shells. The source code itself, although developed in `c`, even made use of macros to give it an ALGOL68 flavor.

The Bourne shell had two primary goals: serve as a command interpreter to interactively execute commands for the operating system and for scripting (writing reusable scripts that could be invoked through the shell). In addition to replacing the Thompson shell, the Bourne shell offered several advantages over its predecessors. Bourne introduced control flows, loops, and variables into scripts, providing a more functional language to interact with the operating system (both interactively and noninteractively). The shell also permitted you to use shell scripts as filters, providing integrated support for handling signals, but lacked the ability to define functions. Finally, it incorporated a number of features we use today, including command substitution (using back quotes) and HERE documents to embed preserved string literals within a script.

The Bourne shell was not only an important step forward but also the anchor for numerous derivative shells, many of which are used today in typical Linux systems. Figure 1 illustrates the lineage of important shells. The Bourne shell led to the development of the Korn shell (ksh), Almquist shell (ash), and the popular Bourne Again Shell (or Bash). The `c` shell (csh) was under development at the time the Bourne shell was being released. Figure 1 shows the primary lineage but not all influences; there was significant contribution across shells that isn't depicted.

Figure 1. Linux shells since 1977

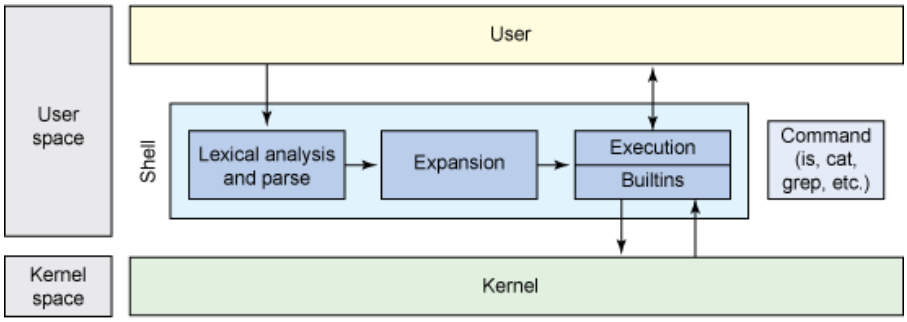


We'll explore some of these shells later and see examples of the language and features that contributed to their advancement.

Basic shell architecture

The fundamental architecture of a hypothetical shell is simple (as evidenced by Bourne's shell). As you can see in Figure 2, the basic architecture looks similar to a pipeline, where input is analyzed and parsed, symbols are expanded (using a variety of methods such as brace, tilde, variable and parameter expansion and substitution, and file name generation), and finally commands are executed (using shell built-in commands, or external commands).

Figure 2. Simple architecture of a hypothetical shell



In the Resources section, you can find links to learn about the architecture of the open source Bash shell.

Exploring Linux shells

Let's now explore a few of these shells to review their contribution and examine an example script in each. This review includes the c shell, the Korn shell, and Bash.

The Tenex C shell

The c shell was developed for Berkeley Software Distribution (BSD) UNIX systems by Bill Joy while he was a graduate student at the University of California, Berkeley, in 1978. Five years later, the shell introduced functionality from the Tenex system (popular on DEC PDP systems). Tenex introduced file name and command completion in addition to command-line editing features. The Tenex c shell (tcsh) remains backward-compatible with csh but improved its overall interactive features. The tcsh was developed by Ken Greer at Carnegie Mellon University.

One of the key design objectives for the c shell was to create a scripting language that looked similar to the c language. This was a useful goal, given that c was the primary language in use (in addition to the operating system being developed predominantly in c).

A useful feature introduced by Bill Joy in the c shell was command history. This feature maintained a history of the previously executed commands and allowed the user to review and easily select previous commands to execute. For example, typing the command `history` would show the previously executed commands. The up and down arrow keys could be used to select a command, or the previous command could be executed using `!!`. It's also possible to refer to arguments of the prior command; for example, `!*` refers to all arguments of the prior command, where `!$` refers to the last argument of the prior command.

Take a look at a short example of a tcsh script (Listing 1). This script takes a single argument (a directory name) and emits all executable files in that directory along with the number of files found. I reuse this script design in each example to illustrate differences.

The tcsh script is divided into three basic sections. First, note that I use the shebang, or hashbang symbol, to declare this file as interpretable by the defined shell executable (in this case, the tcsh binary). This allows me to execute the file as a regular executable rather than precede it with the interpreter binary. It maintains a count of the executable files found, so I initialize this count with zero.

Listing 1. File all executable files script in tcsh

```
#!/bin/tcsh
# find all executables

set count=0

# Test arguments
if ($#argv != 1) then
  echo "Usage is $0 <dir>"
  exit 1
endif

# Ensure argument is a directory
if (! -d  $1) then
  echo "$1 is not a directory."
  exit 1
endif

# Iterate the directory, emit executable files
foreach filename ($1/*)
  if (-x $filename) then
    echo $filename
    @ count = $count + 1
  endif
end

echo
echo "$count executable files found."

exit 0
```

The first section tests the arguments passed by the user. The `#argv` variable represents the number of arguments passed in (excluding the command name itself). You can access these

arguments by specifying their index: For example, `#1` refers to the first argument (which is shorthand for `argv[1]`). The script is expecting one argument; if it doesn't find it, it emits an error message, using `$0` to indicate the command name that was typed at the console (`argv[0]`).

The second section ensures that the argument passed in was a directory. The `-d` operator returns True if the argument is a directory. But note that I specify a `!` symbol first, which means *negate.* This way, the expression says that if the argument is not a directory, emit an error message.

The final section iterates the files in the directory to test whether they're executable. I use the convenient `foreach` iterator, which loops through each entry in the parentheses (in this case, the directory), and then tests each as part of the loop. This step uses the `-x` operator to test whether the file is an executable; if it is, the file is emitted and the count increased. I end the script by emitting the count of executables.

Korn shell

The Korn shell (ksh), designed by David Korn, was introduced around the same time as the Tenex `c` shell. One of the most interesting features of the Korn shell was its use as a scripting language in addition to being backward-compatible with the original Bourne shell.

The Korn shell was proprietary software until the year 2000, when it was released as open source (under the Common Public License). In addition to providing strong backward-compatibility with the Bourne shell, the Korn shell includes features from other shells (such as history from csh). The shell also provides several more advanced features found in modern scripting languages like Ruby and Python—for example, associative arrays and floating point arithmetic. The Korn shell is available in a number of operating systems, including IBM® AIX® and HP-UX, and strives to support the Portable Operating System Interface for UNIX (POSIX) shell language standard.

The Korn shell is a derivative of the Bourne shell and looks more similar to it and Bash than to the `c` shell. Let's look at an example of the Korn shell for finding executables (Listing 2).

Listing 2. Find all executable files script in ksh

```
#!/usr/bin/ksh
# find all executables

count=0

# Test arguments
if [ $# -ne 1 ] ; then
  echo "Usage is $0 <dir>"
  exit 1
fi

# Ensure argument is a directory
if [ ! -d  "$1" ] ; then
  echo "$1 is not a directory."
  exit 1
fi

# Iterate the directory, emit executable files
for filename in "$1"/*
do
  if [ -x "$filename" ] ; then
    echo $filename
```

```
    count=$((count+1))
  fi
done

echo
echo "$count executable files found."

exit 0
```

The first thing you'll notice in Listing 2 is its similarity to Listing 1. Structurally, the script is almost identical, but key differences are evident in the way conditionals, expressions, and iteration are performed. Instead of adopting `c`-like test operators, ksh adopts the typical Bourne-style operators (`-eq`, `-ne`, `-lt`, and so on).

The Korn shell also has some differences related to iteration. In the Korn shell, the `for in` structure is used, with command substitution to represent the list of files created from standard output for the command `ls '$1/*` representing the contents of the named subdirectory.

In addition to the other features defined above, Korn supports the alias feature (to replace a word with a user-defined string). Korn has many other features that are disabled by default (such as file name completion) but can be enabled by the user.

The Bourne-Again Shell

The Bourne-Again Shell, or Bash, is an open source GNU project intended to replace the Bourne shell. Bash was developed by Brian Fox and has become one of the most ubiquitous shells available (appearing in Linux, Darwin, Windows®, Cygwin, Novell, Haiku, and more). As its name implies, Bash is a superset of the Bourne shell, and most Bourne scripts can be executed unchanged.

In addition to supporting backward-compatibility for scripting, Bash has incorporated features from the Korn and `c` shells. You'll find command history, command-line editing, a directory stack (`pushd` and `popd`), many useful environment variables, command completion, and more.

Bash has continued to evolve, with new features, support for regular expressions (similar to Perl), and associative arrays. Although some of these features may not be present in other scripting languages, it's possible to write scripts that are compatible with other languages. To this point, the sample script shown in Listing 3 is identical to the Korn shell script (from Listing 2) except for the shebang difference (/bin/bash).

Listing 3. Find all executable files script in Bash

```
#!/bin/bash
# find all executables

count=0

# Test arguments
if [ $# -ne 1 ] ; then
  echo "Usage is $0 <dir>"
  exit 1
fi

# Ensure argument is a directory
```

```
if [ ! -d  "$1" ] ; then
  echo "$1 is not a directory."
  exit 1
fi

# Iterate the directory, emit executable files
for filename in "$1"/*
do
  if [ -x "$filename" ] ; then
    echo $filename
    count=$((count+1))
  fi
done

echo
echo "$count executable files found."

exit 0
```

One key difference among these shells is the licenses under which they are released. Bash, as you would expect, having been developed by the GNU project, is released under the GPL, but csh, tcsh, zsh, ash, and scsh are all released under the BSD or a BSD-like license. The Korn shell is available under the Common Public License.

Exotic shells

For the adventurous, alternative shells can be used based on your needs or taste. The Scheme shell (scsh) offers a scripting environment using Scheme (a derivative of the Lisp language). The Pyshell is an attempt to create a similar script that uses the Python language. Finally, for embedded systems, there's BusyBox, which incorporates a shell and all commands into a single binary to simplify its distribution and management.

Listing 4 provides a look at the find-all-executables script within the Scheme shell (scsh). This script may appear foreign, but it implements similar functionality to the scripts provided thus far. This script includes three functions and directly executable code (at the end) to test the argument count. The real meat of the script is within the `showfiles` function, which iterates a list (constructed after `with-cwd`), calling `write-ln` after each element of the list. This list is generated by iterating the named directory and filtering it for files that are executable.

Listing 4. File all executable files script in scsh

```
#!/usr/bin/scsh -s
!#

(define argc
        (length command-line-arguments))

(define (write-ln x)
        (display x) (newline))

(define (showfiles dir)
        (for-each write-ln
                (with-cwd dir
                        (filter file-executable? (directory-files "." #t)))))

(if (not (= argc 1))
        (write-ln "Usage is fae.scsh dir")
        (showfiles (argv 1)))
```

Conclusion

Many of the ideas and much of the interface of the early shells remain the same almost 35 years later—a tremendous testament to the original authors of the early shells. In an industry that continually reinvents itself, the shell has been improved upon but not substantially changed. Although there have been attempts to create specialized shells, the Bourne shell derivatives continue to be the primary shells in use.

Resources

Learn

- The V6 Thompson Shell Port (osh), developed and maintained by J.A. Neitzel, is a great resource for the osh source as well as the external shell utilities that it relies on (such as `if` and `goto`). You can also find an archive of utilities written in the Thompson shell in addition to the original source code itself.
- Goosh is the unofficial Google shell, which implements a shell interface over the commonly used Google search interface. Goosh is an interesting example of how shells can be applied to nontraditional interfaces.
- The Bourne shell is the anchor from which our current shells were derived. The source files have a certain ALGOL68 flavor that was accomplished through the use of `C` macros.
- The Bourne-Again Shell is the most commonly used shell in Linux, combining features of the Bourne shell, Korn shell, and `C` shell. For a great read, learn about the structure and internals of Bash in the third chapter of "The Architecture of Open Source Applications."
- Check out additional developerWorks articles on shell scripting, such as Daniel Robbins' "Bash by example" Part 1 (March 2000), Part 2 (April 2000), and Part 3 (May 2000). You can also learn about Korn shell scripting (June 2008) and Tcsh shell variables (August 2008).
- At the kornshell site, get the latest news on the Korn shell, including documentation and other resources.
- Wikipedia includes a great comparison of shells, including general characteristics, interactive features, programming features, syntax, data types, and IPC mechanisms.
- Tim's article "BusyBox simplifies embedded Linux systems" (developerWorks, August 2006) explores the BusyBox application and how to add new commands to this static shell architecture.
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools, as well as IT industry trends.
- Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.
- Follow Tim on Twitter. You can also follow developerWorks on Twitter, or subscribe to a feed of Linux tweets on developerWorks.

Get products and technologies

- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach*, *GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim works at Intel and resides in Longmont, Colorado.