The journal of Apple technology.

Volume Number: 18 (2002)
Issue Number: 07
Column Tag: Section 7

by Rich Morin

More Basic Commands

and a first look at “shell scripts”

Previous “Section 7” columns have looked at several BSD commands, but they haven’t said much about the ways in which commands get performed. Let’s do that now, starting with the Terminal application. Each time you start up a new Terminal window, Mac OS X has to start up a program to talk to it. The program it starts is a “command interpreter”, most often referred to as a “shell”.

Whenever a Terminal window is active, your keyboard is tied to the “standard input” of the program that is currently running “in” the window. Similarly, the window’s text area displays the program ‘s “standard output”. If the shell starts up another program (such as ls), the shell must “loan” these I/O connections to the other program. In short, a Terminal window is always connected to some program; most often, this will be the shell.

Each shell runs as a separate process, so the “state” that gets set in one “shell session” is not shared with other shell sessions. Thus, if you use cd to change the “current directory” in one Teminal window, the shell sessions in your other windows will not be affected.

Because cd affects information that the shell must maintain (and hand off to any programs it starts up), cd is actually “built in” to the shell itself. Most commands, however, are not built into the shell. Instead, they are implemented as:

- aliases or shell functions – These are sequences of commands, kept in the shell’s memory and interpreted upon demand.
- shell scripts – These are sequences of commands, kept in separate files and interpreted upon demand. Sometimes the shell itself will interpret the script; other times, as discussed below, it will start up a separate program to do so.
- executable binaries – These are separate programs, kept in separate files and executed upon demand. To run them, the shell must fork(2), then exec(3) the relevant file.

In the case of built-ins, aliases, and shell functions, the shell “knows” what needs to be done. In the case of shell scripts and executable binaries, however, it must find the requested file, determine its nature, and proceed accordingly.

If a command is not a built-in, an alias, or a shell function, the shell looks for an executable file of that name in one of several directories. Because the shell may cache the contents of these directories in an internal data structure, some shells require the user to run a special command (“rehash”) when a new command is added. The list of these directories, called the “search path”, is contained in the $path variable:

```
[localhost:~] rdm% echo $path
/Users/rdm/bin/powerpc-apple-darwin /Users/rdm/bin ...
```

The which command can be used to predict the shells' behavior for a given command:

```
[localhost:~] rdm% which cwd l
cwd:     aliased to echo $cwd
l:       aliased to ls -lg
[localhost:~] rdm% which cd echo setenv
cd: shell built-in command.
echo: shell built-in command.
setenv: shell built-in command.
[localhost:~] rdm% which apropos keytool
/usr/bin/apropos
/usr/bin/keytool
```

Finding and executing files (e.g., /usr/bin/apropos) takes time, but it allows us to add new commands quite trivially. Imagine how awkward and constraining it would be if each new command had to be linked into the shell!

Now that we have the full path names of the commands, we can examine them a bit:

```
[localhost:~] rdm% cd /usr/bin
[localhost:/usr/bin] rdm% file apropos keytool
apropos: Mach-O executable ppc
keytool: Bourne shell script text
```

As file informs us, /usr/bin/apropos is an executable binary file for the PowerPC, encoded in "Mach-O" format. /usr/bin/keytool, in contrast, is a text file which must be run interpretively by sh (the "Bourne shell") as a "shell script". But wait a second; how do file (and the shell) know that? Maybe it's time to take a look at /usr/bin/keytool:

```
[localhost:/usr/bin] rdm% cat keytool
#!/bin/sh
#
# Shell wrapper to launch keytool.
#
...
```

The first two characters of the file ("#!") tell the user's shell that this is a script. The rest of the line gives the full path name ("/bin/sh") of the program which must be used to interpret the script. Because each script can declare its own language (e.g., Bourne shell, C shell, Perl), new scripting languages can be added at any time.

If a script starts with a sharp sign ("#"), but not the "#!" sequence, it is treated as a C shell script. If a script begins with any other character, it is treated as a Bourne shell script. Finally, you may encounter scripts that start like "#!/usr/bin/env perl". This tells the system to run /usr/bin/env, which will then go off and find the "appropriate" copy of perl.

The same header information is used by file to determine the type of a given file. To see file's list of "magic numbers" (and interpretations), let's look at /etc/magic:

```
[localhost:/usr/bin] rdm% cd /etc
[localhost:/etc] rdm% wc -l magic
    3837 /etc/magic
[localhost:/etc] rdm% grep 'shell script' magic
0  string  #!/bin/sh     Bourne shell script text
...
```

“wc -l” (word count, in line mode) tells us that there are about 4K lines in /etc/magic, so we don’t try to list it. Instead, we use grep (global regular expression print) to display lines that contain the string “shell script”.

As discussed last month, most of BSD’s “system metadata” (e.g., control files, log files) is kept in ASCII format. This works well with BSD’s wide range of text processing tools, allowing us to extract and process information in a very flexible manner.

If /etc/magic had been encoded in some proprietary, binary format (e.g., Microsoft Word), the exercise above would have required us to start up a particular application. And, if the application wasn’t able to do what we wanted, we’d have been stymied.

Scripting languages

BSD provides a variety of scripting languages. You are free to use any, all, or none of them, as you prefer. Here are some (biased :-) notes that may help you in making your selections:

- The Bourne shell (sh) is not particularly convenient as an interactive command interpreter, but it works well as a programming language. The Korn and Z shells (ksh, zsh) are upwardly-compatible versions of the Bourne shell. The former is not supplied as part of Mac OS X, but the latter is and may deserve a look.
- The Bourne-Again shell (bash) is upwardly compatible with the Bourne shell, but it also borrows a variety of features from other shells. It is the default shell for most Linux distributions, but it is not supplied as part of Mac OS X.
- The C shell (csh) is very convenient as an interactive command interpreter, but it loses badly as a programming language due to its lack of syntax recognition. For example, multi-line commands always require backslashes at the end of non-terminal lines, even when the shell should be able to recognize that the command isn’t finished. Most BSD systems, including Mac OS X, actually provide tcsh, rather than csh. As the tcsh man page says:
  “tcsh is an enhanced, but completely compatible, version of csh(1). It is a command language interpreter, usable both as an interactive login shell and as a shell-script command processor. It includes a command-line editor, programmable word completion, spelling correction, a history mechanism, job control, and a C-like syntax.”
- Awk (awk) is a convenient and simple scripting language, suitable for simple formatting and calculation tasks.
- If things start to get too complex for Awk or the shell to handle, you should probably turn to Perl (perl). Perl is an extremely powerful language, with a large and active user community.
- Python and Ruby are object-oriented scripting languages with small but vocal followings. Neither language is supplied as part of Mac OS X, but (like the other languages mentioned above), they are readily available for you to download and install.
- Finally, if you’re a Tcl fan, you might want to try out tclsh (“a shell-like application that reads Tcl commands“).

Rich Morin has been using computers since 1970, Unix since 1983, and Mac-based Unix since 1986 (when he helped Apple create A/UX 1.0). When he isn’t writing this column, Rich runs Prime Time Freeware (www.ptf.com), a publisher of books and CD-ROMs for the Free and Open Source software community. Feel free to write to Rich at rdm@ptf.com.