Moving beyond basic Bash

SHELL GAMES

Even beginners can benefit from a greater understanding of the Bash shell's many built-in commands. **BY BRUCE BYFIELD**

Most GNU/Linux users know that the command line uses Bash (Bourne Again Shell) [1], but what fewer know is that, by default, they use the plainest version of Bash. The truth is that Bash is a command much like any other. It might be the command that runs other commands – a command interpreter – but, just like any other command, its behavior can be radically altered by the options you add when starting it up or by running some of its built-in commands (builtins). Moreover, you can even choose an entirely different shell through which to interact with your system.

Admittedly, many of Bash's options and builtins [2] are of interest only if you are writing scripts. Desktop users are unlikely, for example, to find uses for a login shell or builtins like *continue* or *declare*. Still, even beginners at the command prompt can benefit from learning more about the options available from the command line.

Modifying Bash is easy. From any computer or terminal, you can start Bash or another shell just like any other command. In Gnome, you can add options by running Gnome Terminal and creating a custom profile. When the profile is created, select *Edit* | *Profiles* | *Edit* | *Title & Command*. Then check the *Run a custom command instead of my shell* checkbox (Figure 1). In the *Custom command* field, enter the form of the Bash command you want to run, then, directly below it, select what you want to happen when you exit from the command.

Similarly, in KDE, open Konsole and create a new profile by selecting *Settings* | *Manage Profile*. Highlight the new profile and click *Edit Profile* | *General* to modify the Bash command (Figure 1). If you add the Konsole widget to your panel or folder view, it presents a list of profiles from which you can choose to suit your needs.

Options Effects

The basic structure of the Bash command is no different from any other command: *bash [options] [arguments]*. And, just like any other command, *bash* has a mixture of single-letter options beginning with a hyphen (-) that it inherited from earlier incarnations, along with longer options that begin with a double hyphen (--) that were added by the GNU Project.

However, as a command, *bash* does behave in some unusual ways. The first argument after the options can be a file full of commands that Bash runs instead of waiting for your input at the keyboard. Even more significantly, as Bash starts, it refers to */etc/profile*, the generic Bash profile, and *~/.profile* or *~./.bash_profile*, the customized profile in the home directory of the current account, as well as */etc/bash.bashrc* and *~/.bashrc* for non-login shells. You can prevent the reading of profile files by adding the *--noprofile option* and of the **bashrc* files by adding the *--norc* option. Instead, you can force Bash to use a replacement for all *bashrc* files with the option *--rcfile [file]*.

Many of Bash's options are of interest mainly to advanced users, although you might try running *--verbose* for a while to see which environment variables are invoked by each command (Figure 2). Many, like those that change the startup files, modify resources that Bash uses.

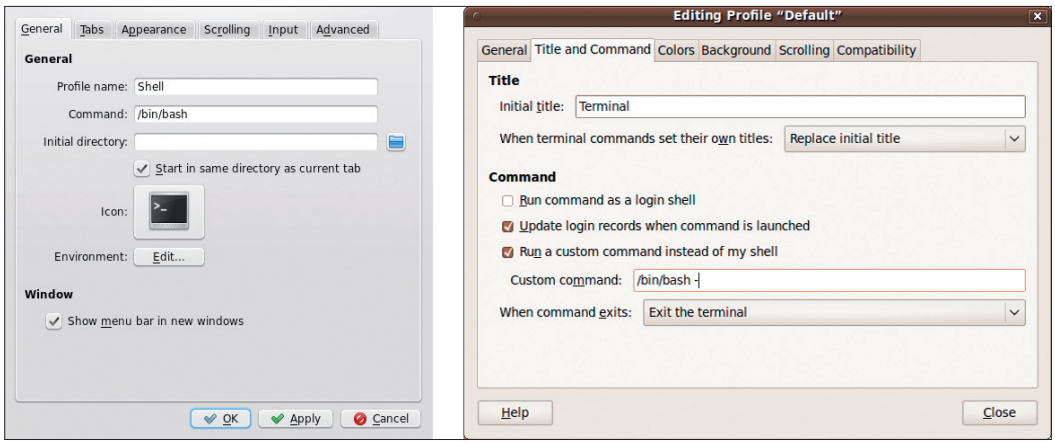

Figure 1: Whether you run KDE (left) or Gnome (right), you can run a modified version of Bash, or another shell altogether, by making a simple change in your desktop.

However, you might try running the command *bash --debugger -O extdebug* to run Bash in debugger mode. Another moderately advanced option is to start a restricted shell with *-r*, *--restricted*, or *-O restricted_shell*. A restricted shell is exactly what it sounds like: one in which some basic actions are not permitted. These include changing directories, turning off restrictions with *set*, and another nine or ten actions [3].

Some system administrators use a restricted shell to prevent users from moving about in the shell, but this technique is a weak security tactic because you do not need to be much of an expert to start a shell that is interactive. Instead, a restricted shell is most valuable when you want to test programs that might be buggy or untrusted (Figure 3). By running such programs in a restricted shell, you can hope to minimize any damage that they cause.

Perhaps the most interesting option is *-O [modification]*, which affects Bash in a number of simple yet productive ways. You can start the same modifications with Bash's builtin *shopt -s [option]*.

Besides the *extdebug* and *restricted_shell* options already mentioned, some of the most immediately useful modifications (mods) affect navigation. For instance, if you run *bash -O autocd*, Bash interprets the name of a subdirectory that is entered as an alias for using the *cd* command to move to the subdirectory (Figure 4). Another handy mod is *cdspell*, which tries to correct the misspellings of directory names automatically – although it usually requires the full path and fails when you use abbreviations like *./* for the current directory. Similarly, *dirspell* corrects misspellings of directory names in file completion.

Another mod for file completion is *nocaseglob*, which ignores the usual distinction between upper- and lowercase letters. Also, *checkjobs* directs Bash to display the status of running or stopped jobs when you exit Bash. The *mailwarn* mod has Bash tell you when a file containing mail has been read since the last time Bash accessed it.

```
bruce@nanday:~$ cd /home/bruce
cd /home/bruce
echo -ne "\033]0;${USER}@${HOSTNAME}: ${PWD}\007"
bruce@nanday:~$
```

Figure 2: If you've ever wondered what happens when you run a command, run Bash with the verbose option.

Builtin Commands

Many of Bash's builtins are for scripting, which is a topic for another day, but in addition to *shopt*, with its modifications of Bash's behavior, the builtins also include functions that are useful in everyday desktop computing.

In fact, Bash's builtins are unavoidable. If you have ever typed *pwd* to check which directory you are in or created an alias so that entering *ls* is equivalent to typing *ls --color=auto*, then you have used Bash's builtins. For example, you can't even navigate at the prompt without using the *cd* builtin.

```
bruce@nanday:~$ bash -r
bruce@nanday:~$ cd /etc
bash: cd: restricted
bruce@nanday:~$
```

Figure 3: A restricted shell is just what it sounds like: a shell in which some functionality, like changing directories, is not allowed.

Several builtins give you basic information about your system. For instance, the *type* command either tells you whether a command is an alias or builtin or it gives the path to the executable (Figure 5). You cannot see system commands like *apt-get* unless you are logged in as root, but *type* can satisfy your curiosity about the commands you are using and how each is regarded. For example, *cp* (copy) is a essential command and is placed in */bin*, whereas *sudo*, which allows you to perform root functions from another account, is less essential to your system and is therefore placed in */usr/bin*. Also, you can use *type* to check to see whether a command is an alias, although you might prefer to use the *alias* command instead.

Another builtin that gives you information is *jobs*, which lists processes and whether they are running or stopped – something that can be hard to tell if you are running commands in the background, especially if they follow the Unix tradition of not providing completion messages.

Entering the *bare* command will give you a list of the processes owned by the current user account and the status of each. If you want to end a process because it is misbehaving, you can enter *job -l* to find its process ID, then enter *kill [processID]* to shut it down.

Other builtins both give information and edit system behavior. For example, the unadorned command *set* lists all environmental variables, whereas other options allow you to turn off or on standard system behavior such as brace expansion (*-B*), which is Bash's ability to replace a variable typed in curly braces with its value.

Similarly, *ulimit -a* gives you information about system resource limitations, such as the maximum number of threads allowed or the maximum number of pro-

```
bruce@nanday:~$ bash -O autocd
bruce@nanday:~$ download
cd download
bruce@nanday:~/download$ bash -O cdspell
bruce@nanday:~/download$ cd /ect
/etc
```

Figure 4: Here, the -O option tells Bash to treat a directory name that is entered as the equivalent of changing to that directory, and it will try to correct directory names that are misspelled.

cesses that a single user can own (Figure 6). But add a number, and you can reset the limit. For example, *ulimit -x 10* sets the number of file locks to *10*. You can also choose whether a limit is hard or soft – that is, whether only the root user can raise it, or anyone can.

Security-minded users might also be interested in the umask utility. Just running the *umask* command without options or a mode tells you the default permissions when a new file is created on

```
bruce@nanday:/etc$ type mv ls cd
mv is /bin/mv
ls is aliased to `ls --color=auto'
cd is a shell builtin
```

Figure 5: You can use the type command to find out whether a command is an alias, builtin, or external command.

the system. If I run *umask* on the system I am using currently, I find that the default is set to *0077* in octal notation, whereas if I run *umask -S*, I find that, in symbolic notation, the default permissions are *u=rwx,g=, o=* (Figure 7). Both of these notations mean the same thing: The owner of a file can read, write, or execute it, but neither their group nor anyone else except root can do any of these things. If I wanted to allow others to read new files, I could enter *umask 4477* to change the default.

Yet another builtin worth knowing about is *history*, which manages the command history stored for each user in the *.bash-history* file in their home directory. The *bare* command lists all the commands entered at the command line, with the oldest listed first. For privacy, you can type *history -d[position]* to remove a specific command, or *history -c* to clear the entire history.

Alternative Shells

The more advanced your computing skills, the more benefits you can get from Bash, both in terms of customization and functionality. The examples given here are no more than an introduction to some of the diversity you can find when you start moving away from the defaults for Bash.

Once you have thoroughly explored Bash, you might want to consider exploring other shells. Most major distributions package several shells, installing their executables into the */bin* directory. To use them, all you need to do is edit the profiles of the Gnome Terminal or Konsole and the *$BASH* environment variable to refer to them.

```
bruce@nanday:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 38912
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 38912
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Figure 6: The ulimit -a command shows system resource limitations for your computer. The table lists a resource, its measurement, the option to list it separately, and whatever limits are currently placed on it. Also, you can use the ulimit command to change limits.

Relatively few users bother with the original Bourne shell or with the original C shell (csh) or Korn shell (ksh), although they still have enough enthusiasts that large distributions like Debian include them. However, most users exploring alternative shells today are more likely to want the added functionality of their successors, such as *tcsh* [4] or *zsh* [5].

I have heard of shells like tcsh and zsh described as being to Bash what Bash is to the Windows command line – in other words, vastly superior. That is an exaggeration and, in tcsh's case, is probably helped by the fact that tcsh is the default shell of FreeBSD. But, without a doubt, these shells possess numerous features that Bash either lacks or has in less sophisticated form. For instance, tcsh not only has a scripting syntax similar to that of the C programming language, but it also has programmable word completion and spelling correction. Similarly, zsh boasts spell-checking, programmable completion, and multiple redirection.

New users might also want to explore recently developed shells like *fish* [6], whose goal is to make working with the command line easier. Fish includes highlighted syntax and enhanced history and tab completion.

```
bruce@nanday:~$ umask
0077
bruce@nanday:~$ umask -S
u=rwx,g=,o=
```

Figure 7: Use umask to list or set the default permissions, either in octal (no option) or symbolic notation (-S).

Each of these alternatives takes some adjustment, but often commands will be similar enough that you should have only minimal trouble adjusting to each shell. Like Bash, with its options and builtins, though, these alternative shells emphasize one of the most important points about the command line: Just as on the desktop, you don't have to settle for what you're given at the GNU/Linux prompt. If you explore, you will soon discover that usually you can do things your own way – and the more you learn, the truer that will be. ■

INFO

[1] Bash: *http://www.gnu.org/software/bash/*

[2] Bash builtins: *http://www.faqs.org/docs/bashman/bashref_55.html#SEC55*

[3] Restricted shell: *http://www.faqs.org/docs/bashman/bashref_75.html*

[4] Tcsh: *http://www.tcsh.org/Welcome*

[5] Zsh: *http://www.zsh.org/*

[6] Fish: *http://fishshell.org/index.php*