Working in the Bash shell

An introduction

Skill Level: Intermediate

Robert Brunner (rb@ncsa.uiuc.edu)
NCSA Research Scientist, Assistant Professor of Astronomy
University of Illinois, Urbana-Champaign

30 May 2006

Get an introduction to the Bash shell, which you can use on nearly any UNIX®-based operating system. Bash is a mature, powerful, yet easy-to-use shell that is freely available. This tutorial provides a brief history of Bash, which indicates how the Bash shell is different than some of the other popular UNIX shells, and also provides an overview of the major features available within Bash. Next, you'll learn more about the UNIX file system, how to work with both directories and files, and several methods for customizing the appearance and behavior of Bash. Finally, the tutorial concludes with a discussion of the job control functionality of Bash.

Section 1. Before you start

Learn what to expect from this tutorial and how to get the most out of it.

About this tutorial

This tutorial provides a basic overview of Bash. After reading this tutorial, you should be able to perform basic operations within a UNIX® terminal that is running Bash. This includes moving around the file system, working with files and directories, and running basic commands.
Objectives

This tutorial is written for anyone who wants to learn how to use Bash to interact with their computer directly from the command line. If you have a computer running Linux®, Mac OS® X, or another UNIX system and want to learn how to get started working in Bash, this tutorial is for you.

Prerequisites

This tutorial has no prerequisites.

System requirements

There are no system requirements for this tutorial -- you can simply read along and learn about Bash. To maximize your benefits from this tutorial, however, you need to be able to try the techniques the tutorial presents. This requires an operational Bash shell, preferably version 2.05, or higher. If you do not have a current version of Bash installed on your computer, visit the Bash home page for information about how to obtain your own copy (see the Resources section).

Section 2. Overview of Bash

Most computer users are safely insulated from the technology inside their computers that perform all of the actual work. For example, if you surf the Web, send and receive e-mails, or write a document, you are interacting with your computer at a fairly high level. The software that isolates the user from the low-level details is known as an operating system. Operating systems have a specific component, called the kernel, that interacts directly with the hardware.

Over the years, graphical interfaces like Microsoft® Windows®, K Desktop Environment (KDE), GNU Network Object Model Environment (GNOME), Apple's Aqua, and the X Consortium’s X11 have been layered over the base operating system to simplify the common tasks that users often perform with their computers. However, this wasn't always the case. Prior to the popularization of windowing systems, computer users operated at a command-line prompt, directly interacting with the kernel to perform their work. As this was a difficult task and prone to errors, a new software layer, called a shell, was developed that simplified the task of communicating with the kernel.
Brief history of the UNIX shell

The first commonly-used shell was developed by Stephen R. Bourne at AT&T Bell Labs in 1974 and is called the Bourne shell. This shell was written to allow a user to interact more easily with the Seventh Edition of the Bell Labs Research version of UNIX. The Bourne shell provides a complete programming language that enables a user to control program input and output and contains a powerful expression-matching capability.

An alternative UNIX implementation was developed around the same time at the University of California, Berkeley, which came to be known as BSD UNIX (for Berkeley Software Distribution). In 1978, Bill Joy, also at Berkeley, developed a new UNIX shell, called C Shell (/bin/csh), that added new functionality, including job control, aliasing, and improved interactive capabilities. In addition, the programming features of C Shell were changed to more closely match the C programming language. However, some of these changes complicated the development and maintenance of shell scripts.

Other shells were subsequently developed, including the Korn shell (/bin/ksh) and the TC shell (/bin/tcsh) that extended the two original shells, the Bourne shell and C Shell, respectively. While these shells offered several improvements, they both suffered from drawbacks. The Korn shell was initially closed source, and the TC shell suffered from many of the same scripting difficulties as the original C Shell. Shortly afterwards, an international specification was developed, known as POSIX (Portable Operating System Interface) 1003.2, that indicated how a shell should interact with a user.

Bash (/bin/bash), which is an acronym for the Bourne-Again SHell, was originally written by Brian Fox of the Free Software Foundation and was developed to overcome the limitations of previous shells. Building on the Bourne shell tradition, Bash has the following benefits:

- It provides a powerful, easy-to-use scripting language.
- It incorporates the benefits of interactive use from the C Shell family of shells.
- It is free and completely open source.
- It provides a compliant implementation of the POSIX 1003.2 specification.

Features of Bash

Given its rich history, Bash provides several useful features that improve the interactive use of the shell and also simplify the development of shell scripts, which
are simple programs that invoke shell features to automate tasks. Bash is easily customizable, both through special startup initialization files and also through specific Bash options. Table 1 lists some of the basic features that improve the interactive use of Bash.

Table 1. Basic features of the Bash shell

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command line editing</td>
<td>This feature allows you to easily move the cursor around or to edit text at the command line.</td>
</tr>
<tr>
<td>Command history</td>
<td>This feature allows you to <em>replay</em>, or optionally edit and redo, a command that you have already entered at the command line.</td>
</tr>
<tr>
<td>IO redirection</td>
<td>This feature allows you to easily change where a program acquires its input or send its output.</td>
</tr>
<tr>
<td>Aliases</td>
<td>This feature allows you to create a shorthand replacement for a single-line command.</td>
</tr>
<tr>
<td>Functions</td>
<td>This feature allows you to create a shorthand replacement for multi-line commands.</td>
</tr>
</tbody>
</table>

While Bash provides many useful features for simplifying the development and maintenance of shell scripts, they are beyond the scope of this tutorial.

Section 3. Working at a command prompt in Bash

Bash is a program that runs in a UNIX terminal. Therefore, you must start a new terminal application to use it. Your system might be configured to run a different shell by default. To see what shell you are running, enter `echo $SHELL` at the command prompt in your terminal window, and then press Enter (or Return). If `/bin/bash` is the response, you know you are running Bash. If you get a different response, you are probably running a different shell.

To see if Bash is installed on your system, enter `which bash` at the command line. This command locates Bash and provides the full path to the program. If Bash is not installed on your system, you can freely download and install the latest version (see the Resources section). If Bash is installed on your system, you can change your default shell to Bash by using the command `chsh bash`. Note that on some systems, like MAC OS X, the command is slightly different. You also can start a new Bash shell (as long as Bash is installed on your system, even if you are running another shell by default), by entering `bash` at the command prompt and then pressing Enter (or Return).
Working at the command line

When you have Bash running, you can easily customize it by changing different options. To see the list of options, enter the `set -o` command at the Bash prompt, as shown in Listing 1. Note that in this code listing the Bash prompt is the character string `rb$`. The prompt might be different on your system. I address customizing your prompt later in this tutorial.

Listing 1. Setting options in Bash

```bash
rb$ echo $SHELL
/bin/bash
rb$ whereis bash
/bin/bash
rb$ set -o
alllexport off
braceexpand on
emacs on
errexit off
hashall on
histexpand on
history on
ignoreeof off
interactive-comments on
keyword off
monitor on
noclobber off
noexec off
noglob off
nolog off
notify off
nounset off
onecmd off
physical off
posix off
privileged off
verbose off
vi off
xtrace off
```

As shown in this example, there are numerous options that you can change to control how Bash interacts with the user. Some of the more useful options to enable include the `emacs` and `history` options. The former allows you to use Emacs key bindings to move around at the command line, while the latter indicates whether the shell should maintain a list of commands that have been executed so you can easily replay them.

The Emacs key bindings might not be familiar to you if you have never used the popular Emacs (or XEmacs) editor. However, they are easy to learn and provide a powerful way to move through a long command. Some of the more popular key bindings include the following:

- CTRL+A moves to the start of the current line.
- CTRL+E moves to the end of the current line.
CTRL+K deletes all characters past the cursor in the current line.

Mastering these (and related) key bindings can make life at the command prompt considerably easier. Likewise, enabling the `history` option allows you to more easily replay, or edit and then replay, previous commands. The default for most Bash installations is a history record of the last 500 commands, which should be sufficient most of the time. But, if not, you can easily change this value, as shown later in this tutorial. To move around through the history file, you can use the following key bindings:

- CTRL+P (for previous), or the Up Arrow key, moves to the previous command in the history buffer.
- CTRL+N (for next), or the Down Arrow key, moves to the next command in the history buffer.
- CTRL+R (for reverse search) searches for a previous command.

You can also use the `history` command to display previous commands, along with each command’s number in the history buffer. You can then use this number, preceded with an exclamation mark (!), to execute a specific command in the buffer, as in `!382`. You can also perform relative indexing into the history buffer by using the exclamation mark followed by a negative number. For example, `!-2` executes the second-to-last command in the history buffer. Because the history buffer can extend across login sessions, it provides a powerful technique for replaying previous commands.

You probably noticed some other options were shown in Listing 1, some of which might be slightly different (or have different default values) depending on the exact version of your Bash shell. You can easily set these options, by entering `set -o option-name`, or unset these options, by entering `set +o option-name`. For example, `set -o emacs` turns on the Emacs key binding option.

Reading the manual

To learn more about Bash options or any command on a UNIX system, you can use the online UNIX manual. Use the `man` command to access it. Listing 2 shows the manual page for Bash.

**Listing 2. Reading the UNIX man page for Bash**

```
rbs$ man bash
BASH(1) BASH(1)
NAME
  bash - GNU Bourne-Again SHell
```
SYNOPSIS
bash [options] [file]

COPYRIGHT
Bash is Copyright (C) 1989-2004 by the Free Software Foundation, Inc.

DESCRIPTION
Bash is an sh-compatible command language interpreter that executes commands read from the standard input or from a file. Bash also incorporates useful features from the Korn and C shells (ksh and csh).

Bash is intended to be a conformant implementation of the IEEE POSIX Shell and Tools specification (IEEE Working Group 1003.2).

OPTIONS
In addition to the single-character shell options documented in the description of the set builtin command, bash interprets the following options when it is invoked:

Section 4. Working with files and directories in Bash

Most of the work you might do in Bash will somehow involve working with the UNIX filesystem. If you have ever used a graphical file browser, you are probably aware of the tree analogy for a file system. The same is true at the command line. All files and directories are rooted at /, which is the root node of the file system. Table 2 lists several standard directories that most UNIX systems share.

Table 2. Common UNIX system directories

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>This directory is the root directory of the UNIX system, which contains all other files.</td>
</tr>
<tr>
<td>/bin</td>
<td>This directory contains binary versions of system application files, such as the Bash program itself.</td>
</tr>
<tr>
<td>/dev</td>
<td>This directory contains pseudofiles that represent physical devices like disk drives.</td>
</tr>
<tr>
<td>/etc</td>
<td>This directory contains the majority of the system configuration files.</td>
</tr>
<tr>
<td>/lib</td>
<td>This directory contains library files needed for system applications.</td>
</tr>
<tr>
<td>/opt</td>
<td>This directory contains optional system components or applications.</td>
</tr>
<tr>
<td>/tmp</td>
<td>This directory contains temporary files used by system or user applications.</td>
</tr>
<tr>
<td>/usr</td>
<td>This directory contains user and non-critical system applications and related components.</td>
</tr>
<tr>
<td>/var</td>
<td>This directory contains various files required by</td>
</tr>
</tbody>
</table>
system applications, such as log or spool files.

Working with directories in Bash

There are several UNIX commands that allow you to easily create, list, delete directories, and change the directory in which you are currently working:

- `cd` changes the current working directory to your home directory.
- `cd dirname` changes the current working directory to the `dirname` directory.
- `ls dirname` lists the contents of the `dirname` directory.
- `mkdir dirname` makes a new directory called `dirname`.
- `pwd` prints the full path of the current working directory.
- `rmdir dirname` removes the directory called `dirname`.

To use any of these programs, simply enter the command at the Bash prompt (including any necessary directory name) and press **Enter** (or **Return**).

There are also several useful shortcuts for directory names:

- A single period (.) denotes the current working directory.
- Two periods (..) denotes the parent directory of the current working directory.
- The tilde character (~) denotes your home directory (generally the directory in which a new Bash shell starts).

Listing 3 demonstrates how easy it is to use these commands.

**Listing 3. Working with directories in Bash**

```
rb$ pwd
/home/rb
rb$ mkdir temp
rb$ cd temp
rb$ ls
rb$ cd ..
rwb$ pwd
/home/rb
rb$ rmdir temp
rb$ ls ..
lost+found   rb    root
rb$ ls /
bin  etc  lib  misc  proc usr
```
The commands executed in the previous listing are fairly straightforward. First, you print out the path for your current working directory. Next you create a new directory named temp, change to this new directory, and list the contents of this new directory (nothing is displayed since the new directory is empty). You then change to the parent directory of your current working directory, verify that you are back to where you started, and delete the temporary directory you just created. Finally, you list the contents of both the parent directory of your home directory (which, in this case, is /home) and the root directory of your file system.

Working with files in Bash

When working in Bash, there are many commands that deal directly with files. The full list is rather extensive, but the following are some of the more useful commands:

- `cp file1 file2` copies `file1` to `file2`.
- `mv file1 file2` renames `file1` to `file2`.
- `mv filename dirname` moves the `filename` file into the `dirname` directory.
- `less filename` displays the contents of a file in a format that is easily viewed in a terminal window.
- `file filename` displays a best guess for the type of a file.
- `cat filename` displays the entire contents of a file.
- `head filename` displays the first ten lines (exact number can be controlled) of a file.
- `tail filename` displays the last ten lines (exact number can be controlled) of a file.
- `diff file1 file2` displays the differences, if any, between `file1` and `file2`.
- `grep string filename` searches the target file for the indicated string.
- `rm filename` removes the file.

Listing 4 shows how simple it is to use these commands. Feel free to open a terminal and try them out for yourself. And, remember, if you ever want to know...
more, just view the command's manual page. For example, enter `man cp` to read
the manual page for the copy command.

**Listing 4. Working with files in Bash**

```
rb$ cp .bashrc old.bashrc
rb$ diff old.bashrc .bashrc
rb$ cat ~/.bash_profile
source ~/.bashrc
rb$ cat ~/.bashrc
export PS1="\u\$ \\
export DERBY_INSTALL=/opt/Apache/db-derby-10.1.2.1-bin'
export CLASSPATH=$DERBY_INSTALL/lib/derby.jar:$DERBY_INSTALL/lib/derbytools.jar:
rb$ head -4 /usr/share/dict/web2
A
a
aa
aal
rb$ tail -4 /usr/share/dict/web2
Zythia
zythum
Zyzomys
Zyzzogeton
rb$ grep exuberance /usr/share/dict/web2
exuberance
rb$ file bash-tutorial.zip
bash-tutorial.zip: Zip archive data, at least v2.0 to extract
rb$ file bash-tutorial.xml
bash-tutorial.xml: XML document text
```

The file commands demonstrated in the previous code listing are all fairly simplistic
and provide examples of copying files, viewing the contents of files, searching for
string occurrences in a file, or learning the types of different files. First, you use the
copy command to duplicate the .bashrc file, which, as I'll discuss shortly, you can
use to customize the behavior and appearance of your Bash shell. You use the
diff command to check if the two files are identical, which they are; thus, no
differences are printed. Next, you use the cat command to display the contents of
both the .bash_profile and .bashrc files. I'll discuss the contents of these two files in
greater detail in the next section.

The next two examples use the head and tail commands to display the first or last
few lines of a file, respectively. These examples use a flag, `-4`, to indicate that only
four lines should be displayed, as opposed to the default value of ten lines. The next
example uses the grep command to find any occurrence of the string `exuberance`
in a dictionary installed on my computer. Finally, the last two examples use the file
command to display the file type for two different files.

---

**Section 5. Customizing Bash**
Earlier in this tutorial, I discussed the options that you could set to change the behavior of your current Bash shell. Another method for changing your Bash shell is to modify environment variables. You can use an environment variable to modify the behavior of Bash and, also, to pass information to commands that you might run in Bash. Table 3 lists some of the more common environment variables that you might modify in Bash.

**Table 3. Common Bash environment variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASH</td>
<td>BASH provides a full path for the currently running Bash shell.</td>
</tr>
<tr>
<td>BASH_VERSION</td>
<td>BASH_VERSION provides a version of the currently running Bash shell.</td>
</tr>
<tr>
<td>CLASSPATH</td>
<td>CLASSPATH provides a list of directories to search for Java class files or Java archive files.</td>
</tr>
<tr>
<td>HOME</td>
<td>HOME is the home directory of the current user.</td>
</tr>
<tr>
<td>HOSTNAME</td>
<td>HOSTNAME is the name of the current machine.</td>
</tr>
<tr>
<td>LANG</td>
<td>LANG is the current locale setting, which allows the shell to be customized for different countries and languages.</td>
</tr>
<tr>
<td>OSTYPE</td>
<td>OSTYPE provides a description of the current operating system.</td>
</tr>
<tr>
<td>PATH</td>
<td>PATH is a list of directories to search for a command or application when the full location of the file is not specified</td>
</tr>
<tr>
<td>PS1</td>
<td>PS1 is used as the primary prompt string.</td>
</tr>
<tr>
<td>PS2</td>
<td>PS2 is used as the secondary prompt string.</td>
</tr>
<tr>
<td>PROMPT_COMMAND</td>
<td>This command should be executed prior to displaying the Bash prompt.</td>
</tr>
<tr>
<td>PWD</td>
<td>PWD is the full path for the current working directory.</td>
</tr>
<tr>
<td>SHELL</td>
<td>SHELL is the full path name of the current shell.</td>
</tr>
<tr>
<td>USER</td>
<td>USER is the login name of the current user.</td>
</tr>
</tbody>
</table>

To display the current value of an environment variable, use the `echo` command, as demonstrated earlier in this tutorial when you displayed the value of the SHELL variable. To set a variable, you assign a value to the variable by using 

```bash
$varname=new-value
```

and then export the variable to Bash by using

```bash
export $varname
```

Be careful when you are modifying environment variables. If you make a mistake, you might have an unresponsive shell, or worse. In general, you only change the PATH, PS1, or related environment variables because they are used to locate programs, or program components, or to change the appearance of the Bash
prompts.

Customizing the Bash prompt

Changing your Bash prompt actually provides more power than you might imagine. Not only can you change the text that appears as a prompt, but you can display the current time, username, machine name, and even the current working directory. You can also change the foreground and background colors and the text that appears at the top of your terminal window.

To change the value of your prompt, you merely need to change the value of the $PS1$ environment variable. So, to set your prompt to display “I am the Greatest!$”, you enter $PS1='I am the Greatest!$ ' and press Enter (or Return). Several things are worth commenting on, even in this simple example. First, you enclose the full string in quotes. Second, you end the string with a space character so that any text you type is separated from the actual prompt. This is important when you need to scroll through previous commands (and their outputs) in a terminal window. Finally, you use the escape sequence \$, rather than just a dollar sign. This makes Bash display a dollar sign ($) when the user is a regular user, but the shell changes the dollar sign to a hash sign (#) when the user is a superuser (that is, root). If you ever need to administer a computer, this sort of subtle change can be very important, because it helps remind you to be extra careful when you are root.

Bash supports several special escape sequences, which make it easy to customize the appearance of your prompt. For a complete discussion of these sequences, look in the online manual page for Bash or see the Resources section. The following are some of the more popular sequences:

- \u expands to the current user’s username.
- \h expands to the hostname (up to the first period in the DNS name).
- \w expands to the current working directory (\~ indicates the user’s home directory).
- \t expands to the time in a 24-hour clock format (HH:MM:SS).
- \A expands to the time in a 24-hour clock format (HH:MM).
- \e (the ASCII escape character) passes more advanced options to the terminal window.
- \[ indicates the start of a non-printing character sequence.
- \] indicates the end of a non-printing character sequence.
With these sequences, you can now start to understand prompt strings. For example, the prompt string set in my .bashrc file is `PS1='\u\$ '`, which expands to `rb\$`, because I am logged into my computer as an ordinary user, with username `rb`. A more complex example is `PS1='[\A][\u@\h:\w]\$ '`, which expands (on my computer) to `[14:14][rb@home:~]\$`, because the time is currently 2:14 p.m. -- I am logged into the machine called `home` as user `rb` -- and I am working in my home directory. This sort of information can be very useful if you are logged into multiple machines using different terminal windows. With your prompt set appropriately in each Bash shell, you can quickly see what window corresponds to each machine.

Another good technique for quickly acquiring information is to use color as a marker. If you ever work at a terminal window for a long time and need to scroll back to see the output of a particular command, you realize how hard it can be to pick out a prompt in a sea of uniformly-colored characters. Fortunately, you can change the color of your prompt, which makes it stand out from the ordinary text. To indicate that a color code is being provided, you use the ASCII escape sequence `\e`. In fact, you use this sequence any time you want to pass a non-printing escape sequence to Bash. For colors, you use a string of the form `\e[##m` or `\e[#;##m`, where the `##` characters denote a specific number. Use the first form when you want to set either the text or the background color. Use the second form when you want to set the text color and an extra code that indicates a special feature should be used, such as bold colors versus light colors. Putting this all together, you can colorize your previous prompt by using `PS1='\e[1;31m[\A]\e[1;34m[\u@\h:\w]\e[0m\$ '`. This gives you a prompt that displays the time in bold red and the username, hostname, and current working directory in bold blue. Note that you change the color back to zero (which corresponds to nothing, or the original default value), at the end of your prompt. This allows you to easily distinguish any commands you type, or the output of any commands, from the prompt.

One complication that arises when using these ASCII escape sequences, however, is that any time you change your prompt, Bash calculates how many characters it believes are in your prompt and uses that value when wrapping text within your terminal window. Because the color codes aren't actually printed this biases the shell's length calculation. To prevent non-printing characters in your prompt string from being counted, you use the last two escape sequences shown, `\[ and `\]`, around any text that doesn't actually appear in your prompt. While you can sometimes wrap multiple escape sequences within one non-printing sequence, it is generally best (and only put one escape sequence within a single non-printing sequence. With this new knowledge, you can put it all together and properly build a complex Bash prompt

```
PS1='[\e[1;31m[\A]\e[1;34m[\u@\h:\w]\e[0m\$ '.
```

A full listing of color codes is beyond the scope of this tutorial and is also dependent on the type of terminal you are using. In addition, there are other escape sequences you can use to further customize the behavior and appearance of your terminal window and the Bash prompt. For more information, browse the online manual page...
for Bash or explore the Resources section.

Using Bash initialization files

Clearly, you are unlikely to want to reset your options every time you start a new Bash shell. Fortunately, there are several easy ways to record your customizations. Whenever you start a new Bash login shell (generally this happens when you open the first terminal window), two files are read in which specific customizations can be made. The first file, /etc/profile, enables an administrator to make system-wide customizations that are applied uniformly to every user’s Bash login shell. The second file, ~/.bash_profile or ~/.profile, is a per-user customization file that allows you to control the setup of your Bash shell. Remember that the tilde character denotes your home directory. Also, whenever a file name starts with a period, it is hidden, which makes it harder to accidentally move or delete these files. Thus, many system configuration files often start with a period.

Every time you start a new Bash shell (after you have started your login shell), a new file, ~/.bashrc, is read instead of the ~/.bash_profile file. To minimize confusion, the recommended practice is to put all customization settings (such as setting options and environment variables) in your ~/.bashrc file, and source this file from within your login initialization file. You can easily do this by adding source ~/.bashrc to the end of your ~/.bash_profile file, as shown in Listing 4. There are other customizations you can make, including changing the Bash prompt (by modifying the value of the PS1 environment variable), changing the colors displayed in your terminal, and even indicating commands that should be performed when you log out of Bash (using the ~/.bash_logout file).

Section 6. Job control in Bash

One of the most powerful features of Bash is its ability to assist the user in controlling the execution of different commands. Figure 1 shows the basic UNIX command execution model. In this model, every command has a standard mechanism for handling Input/Output (IO):

- **STDIN** (standard input) allows a program to accept input from the shell.
- **STDOUT** (standard output) allows a program to pass its output to the shell.
- **STDERR** (standard error) allows a program to pass error information to the shell.
In general, **STDIN** is the keyboard, where you enter information to the command (like you do at the Bash prompt), and **STDOUT** and **STDERR** are the screen, where results are printed. However, you can reassign these IO mechanisms, so that, for example, a command reads from and writes to a file.

**Figure 1. Basic command execution**

To change the standard IO mechanisms, use the Bash redirection sequences listed in **Table 4**.

**Table 4. Redirecting IO in Bash**

<table>
<thead>
<tr>
<th>Command sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command &lt; file</td>
<td>command &lt; file redirects STDIN to read from a file.</td>
</tr>
<tr>
<td>command &gt; file</td>
<td>command &gt; file redirects STDOUT to write to a file.</td>
</tr>
<tr>
<td>command &gt;&gt; file</td>
<td>command &gt;&gt; file redirects STDOUT to append to a file.</td>
</tr>
<tr>
<td>command 2&gt; file</td>
<td>command 2&gt; file redirects STDERR to write to a file.</td>
</tr>
<tr>
<td>command1</td>
<td>command2</td>
</tr>
</tbody>
</table>

You can use the last entry in **Table 4**, known as a pipe, to chain commands together. Connecting commands together using pipes is a key shell programming technique in UNIX, as demonstrated in **Figure 2**. In this model, each command does one simple task and passes the result of its operation on to another command, which does another simple task, and so on. For example, you can use the `cat` program to pipe the contents of a file into the `grep` command that can pipe every line in the file that contains a given string into the `wc` command that can count and print out how many times the string was found.

**Figure 2. Connecting commands using pipes**
Asynchronous execution

The command execution discussed so far is synchronous execution, where one command executes at a time. Sometimes a command or program needs to run for a long time. Rather than tying up the interactive use of the shell, you can execute commands asynchronously. To do this, you append an ampersand character (&) after the full command line. This tells Bash to run the command in the background, thereby allowing you to continue working in the shell in the foreground. Listing 5 demonstrates this and other job control techniques.

Listing 5. Demonstrating job control in Bash

```
rb$ grep paper.pdf /var/log/httpd/access.log | wc -l
5
rb$ python demo.py &
[1] 20451
rb$ jobs
[1]+ Running python demo.py &
rb$ fg 1
python demo.py
```

Listing 5 demonstrates two different job control techniques. First, the `grep` command searches for the string `paper.pdf` in the access log file for the Apache Web server. The output of this command is piped into the `wc -l` command, which counts how many lines are present in the input file. Thus, you can use this compound command to count how many times someone has accessed the paper.pdf file from a Web site.

The second technique invokes a long-running Python program as a background job. Bash starts the job asynchronously in the background and provides the job identifier. By using the `jobs` command, you can list all currently running commands. In this example, only the one Python program is running, which you can promote to the foreground, and thus have it run synchronously, by using the `fg` command. I have only covered the basics of the job control functionality available in Bash. If you truly want to become a shell master, this is one area you'll want to explore in more detail.
Section 7. Conclusions

This tutorial has provided a basic introduction to the Bash shell. After reviewing the history of UNIX shells and introducing the basic features of Bash, I discussed several ways to customize your Bash environment, including setting options and environment variables and customizing your Bash prompt. I also discussed basic techniques for working at the command-line prompt and concluded with an introduction to the basic Bash job control techniques.
Resources

Learn

• **The Creation of the UNIX Operating System**: Read about the creation of UNIX.
• **Bash Reference Manual**: Check out the official manual for definitive answers to all things Bash.
• "**Bash by example**" (developerWorks, March 2000): Read the series for a look at using Bash for advanced system development work.
• "**Tip: Prompt magic**" (developerWorks, September 2000): Check out this tip for a complete discussion on how to change the default prompt and colors of your terminal under Bash.
• Want more? The developerWorks [AIX and UNIX](http://www.ibm.com/systems/aix) zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials.

Get products and technologies

• **Bash**: Download and learn about Bash on the home page hosted by the Free Software Foundation.
• Build your next development project with [IBM trial software](http://www.ibm.com/developerworks/downloads), available for download directly from developerWorks.

Discuss

• **UNIX shells**: Delve into the discussion about the differences between the various UNIX shells hosted by Usenet FAQs.
• Stay current with [developerWorks technical events and webcasts](http://www.ibm.com/developerworks/tutorials).

About the author

Robert Brunner
Robert J. Brunner is a Research Scientist at the National Center for Supercomputing Applications and an Assistant Professor of Astronomy at the University of Illinois, Urbana-Champaign. He has published several books and a number of articles and tutorials on a range of topics. You can reach him at [rb@ncsa.uiuc.edu](mailto:rb@ncsa.uiuc.edu).