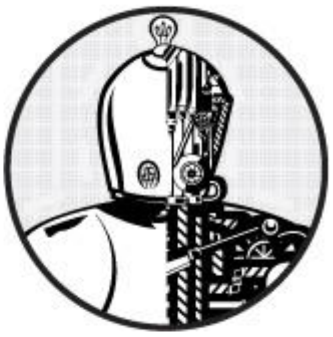
Local storage is much faster for tasks involving many small files, such as compiling software packages and starting desktop environments. The picture becomes more complicated when you have a larger network with many users accessing many different machines, because there are tradeoffs between convenience, performance, and ease of administration.

Chapter 13. User Environments



This book's primary focus is on the Linux system that normally lies underneath server processes and interactive user sessions. But eventually, the system and the user have to meet somewhere. Startup files play an important role at this point, because they set defaults for the shell and other interactive programs. They determine how the system behaves when a user logs in.

Most users don't pay close attention to their startup files, only touching them when they want to add something for convenience, such as an alias. Over time, the files become cluttered with unnecessary environment variables and tests that can lead to annoying (or quite serious) problems.

If you've had your Linux machine for a while, you may notice that your home directory accumulates a bafflingly large array of startup files over time. These are sometimes called *dot files* because they nearly always start with a dot (.). Many of these are automatically created when you first run a program, and you'll never need to change them. This chapter primarily covers shell startup files, which are the ones you're most likely to modify or rewrite from scratch. Let's first look at how much care you need to take when working on these files.

13.1 Guidelines for Creating Startup Files

When designing startup files, keep the user in mind. If you're the only user on a machine, you don't have much to worry about because errors only affect you and they're easy enough to fix. However, if you're creating startup files meant to be the defaults for all new users on a machine or network, or if you think that someone might copy your files for use on a different machine, your task becomes considerably more difficult. If you make an error in a startup file for 10 users, you might end up fixing this error 10 times.

Keep two essential goals in mind when creating startup files for other users:

- **Simplicity**. Keep the number of startup files small, and keep the files as small and simple as possible so that they are easy to modify but hard to break. Each item in a startup file is just one more thing that can break.
- **Readability**. Use extensive comments in files so that the users get a good picture of what each part of a file does.

13.2 When to Alter Startup Files

Before making a change to a startup file, ask yourself whether you really should be making that change. Here are some good reasons for changing startup files:

- You want to change the default prompt.
- You need to accommodate some critical locally installed software. (Consider using wrapper scripts first, though.)
- Your existing startup files are broken.

If everything in your Linux distribution works, be careful. Sometimes the default startup files interact with other files in */etc*.

That said, you probably wouldn't be reading this chapter if you weren't interested in changing the defaults, so let's examine what's important.

13.3 Shell Startup File Elements

What goes into a shell startup file? Some things might seem obvious, such as the path and a prompt setting. But what exactly *should* be in the path, and what does a reasonable prompt look like? And how much is too much to put in a startup file?

The next few sections discuss the essentials of a shell startup file—from the command path, prompt, and aliases through the permissions mask.

13.3.1 The Command Path

The most important part of any shell startup file is the command path. The path should cover the directories that contain every application of interest to a regular user. At the very least, the path should contain these components, in order:

```
/usr/local/bin

/usr/bin

/bin
```

This order ensures that you can override standard default programs with site-specific variants located in */usr/local*.

Most Linux distributions install executables for nearly all packaged software in */usr/bin*. There are occasional differences, such as putting games in */usr/games* and graphical applications in a separate location, so check your system's defaults first. And make sure that every general-use program on the system is available through one of the directories listed above. If not, your system is probably getting out of control. Don't change the default path in your user environment to accommodate a new software installation directory. A cheap way to accommodate separate installation directories is to use symbolic links in */usr/local/bin*.

Many users use a *bin* directory of their own to store shell scripts and programs, so you may want to add this to the front of the path:

```
$HOME/bin
```

NOTE

A newer convention is to place binaries in `$HOME/.local/bin`.

If you're interested in systems utilities (such as `traceroute`, `ping`, and `lsmod`), add the *sbin* directories to your path:

```
/usr/local/sbin

/usr/sbin
```

```
/sbin
```

Adding Dot (.) to the Path

There is one small but controversial command path component to discuss: the dot. Placing a dot (`.`) in your path allows you to run programs in the current directory without using `./` in front of the program name. This may seem convenient when writing scripts or compiling programs, but it's a bad idea for two reasons:

- It can be a security problem. You should *never* put a dot at the *front* of the path. Here's an example of what can happen: An attacker could put a Trojan horse named `ls` in an archive distributed on the Internet. Even if a dot were at the end of the path, you'd still be vulnerable to typos such as `sl` or `ks`.
- It is inconsistent and can be confusing. A dot in a path can mean that a command's behavior will change according to the current directory.

13.3.2 The Manual Page Path

The traditional manual page path was determined by the `MANPATH` environment variable, but you shouldn't set it because doing so overrides the system defaults in */etc/manpath.config*.

13.3.3 The Prompt

Experienced users tend to avoid long, complicated, useless prompts. In comparison, many administrators and distributions drag everything into a default prompt. Your choice should reflect your users' needs; place the current working directory, hostname, and username in the prompt if it really helps.

Above all, avoid characters that mean something significant to the shell, such as these:

```
{ } = & < >
```

NOTE

Take extra care to avoid the > character, which can cause erratic, empty files to appear in your current directory if you accidentally copy and paste a section of your shell window (recall that > redirects output to a file).

Even a shell's default prompt isn't ideal. For example, the default `bash` prompt contains the shell name and version number.

This simple prompt setting for `bash` ends with the customary `$` (the traditional `csh` prompt ends with `%`):

```
PS1='\u\$ '
```

The `\u` is a substitution for the current username (see the PROMPTING section of the bash(1) manual page). Other popular substitutions include the following:

- **\h** The hostname (the short form, without domain names)
- **\!** The history number
- **\w** The current directory. Because this can become long, you can limit the display to just the final component with `\W`.
- **\$** `$` if running as a user account, `#` if root

13.3.4 Aliases

Among the stickier points of modern user environments is the role of *aliases*, a shell feature that substitutes one string for another before executing a command. These can be efficient shortcuts that save some typing. However, aliases also have these drawbacks:

- It can be tricky to manipulate arguments.

- They are confusing; a shell's built-in `which` command can tell you if something is an alias, but it won't tell you where it was defined.
- They are frowned upon in subshells and noninteractive shells; they do not work in other shells.

Given these disadvantages, you should probably avoid aliases whenever possible because it's easier to write a shell function or an entirely new shell script. Modern computers can start and execute shells so quickly that the difference between an alias and an entirely new command should mean nothing to you.

That said, aliases do come in handy when you wish to alter a part of the shell's environment. You can't change an environment variable with a shell script, because scripts run as subshells. (You can also define shell functions to perform this task.)

13.3.5 The Permissions Mask

As described in Chapter 2, a shell's built-in `umask` (permissions mask) facility sets your default permissions. You should run `umask` in one of your startup files to make certain that any program you run creates files with your desired permissions. The two reasonable choices are these:

- **`077`** This mask is the most restrictive permissions mask because it doesn't give any other users access to new files and directories. This is often appropriate on a multi-user system where you don't want other users to look at any of your files. However, when set as the default, it can sometimes lead to problems when your users want to share files but don't understand how to set permissions correctly. (Inexperienced users have a tendency to set files to a world-writable mode.)
- **`022`** This mask gives other users read access to new files and directories. This can be important on a single-user system because many daemons that run as pseudo-users are not be able to see files and directories created with the more restrictive `077` umask.

NOTE

Certain applications (especially mail programs) override the umask, changing it to `077` *because they feel that their files are the business of no one but the file owner.*

13.4 Startup File Order and Examples

Now that you know what to put into shell startup files, it's time to see some specific examples. Surprisingly, one of the most difficult and confusing parts of creating startup files is determining which of several startup files to use. The next sections cover the two most popular Unix shells: `bash` and `tcsh`.

13.4.1 The bash Shell

In `bash`, you can choose from the startup filenames *.bash_profile*, *.profile*, *.bash_login*, and *.bashrc*. Which one is appropriate for your command path, manual page path, prompt, aliases, and permissions mask? The answer is that you should have a *.bashrc* file accompanied by a *.bash_profile* symbolic link pointing to *.bashrc* because there are a few different kinds of `bash` shell instance types.

The two main shell instance types are interactive and noninteractive, but of those, only interactive shells are of interest because noninteractive shells (such as those that run shell scripts) usually don't read any startup files. Interactive shells are the ones that you use to run commands from a terminal, such as the ones you've seen in this book, and they can be classified as *login* or *non-login*.

Login Shells

Traditionally, a login shell is what you get when you first log in to a system with the terminal using a program such as */bin/login*. Logging in remotely with SSH also gives you a login shell. The basic idea is that the login shell is an initial shell. You can tell if a shell is a login shell by running `echo $0`; if the first character is a `-`, the shell's a login shell.

When `bash` runs as a login shell, it runs */etc/profile*. Then it looks for a user's *.bash_profile*, *.bash_login*, and *.profile* files, running only the first one that it sees.

As strange as it sounds, it's possible to run a noninteractive shell as a login shell to force it to run startup files. To do so, start the shell with the `-l` or `--login` option.

Non-Login Shells

A non-login shell is an additional shell that you run after you log in. It's simply any interactive shell that's not a login shell. Windowing system terminal programs (`xterm`, GNOME Terminal, and so on) start non-login shells unless you specifically ask for a login shell.

Upon starting up as a non-login shell, `bash` runs */etc/bash.bashrc* and then runs the user's *.bashrc*.

The Consequences of Two Kinds of Shells

The reasoning behind the two different startup filesystems is that in the old days, users logged in through a traditional terminal with a login shell, then started non-login subshells with windowing systems or the `screen` program. For the non-login subshells, it was deemed a waste to repeatedly set the user environment and run a bunch of programs that had already been run. With login shells, you could run fancy startup commands in a file such as *.bash_profile*, leaving only aliases and other "lightweight" things to your *.bashrc*.

Nowadays, most desktop users log in through a graphical display manager (you'll learn more about these in the next chapter). Most of these start with one noninteractive login shell in order to preserve the login versus non-login model described above. When they do not, you need to set up your entire environment (path, manual path, and so on) in your *.bashrc*, or you'll never see any of your environment in your terminal window shells. However, you *also* need a *.bash_profile* if you ever want to log in on the console or remotely, because those login shells don't ever bother with *.bashrc*.

Example .bashrc

In order to satisfy both non-login and login shells, how would you create a *.bashrc* that can also be used as your *.bash_profile*? Here's one very elementary (yet perfectly sufficient) example:

```
# Command path.
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games
PATH=$HOME/bin:$PATH

# PS1 is the regular prompt.
# Substitutions include:
# \u username \h hostname \w current directory
# \! history number \s shell name \$ $ if regular user
PS1='\u\$ '

# EDITOR and VISUAL determine the editor that programs such as less
# and mail clients invoke when asked to edit a file.
EDITOR=vi
```

```
VISUAL=vi

# PAGER is the default text file viewer for programs such as man.
PAGER=less

# These are some handy options for less.
# A different style is LESS=FRX
# (F=quit at end, R=show raw characters, X=don't use alt screen)
LESS=meiX

# You must export environment variables.
export PATH EDITOR VISUAL PAGER LESS

# By default, give other users read-only access to most new files.
umask 022
```

As described earlier, you can share this *.bashrc* file with *.bash_profile* via a symbolic link, or you can make the relationship even clearer by creating *.bash_profile* as this one-liner:

```
. $HOME/.bashrc
```

Checking for Login and Interactive Shells

With a *.bashrc* matching your *.bash_profile*, you don't normally run extra commands for login shells. However, if you want to define different actions for login and non-login shells, you can add the following test to your *.bashrc*, which checks the shell's `$-` variable for an `i` character:

```
case $- in
 *i*) # interactive commands go here
    command
    --snip--
    ;;
 *)   # non-interactive commands go here
    command
    --snip-
    ;;
esac
```

13.4.2 The tcsh Shell

The standard `csh` on virtually all Linux systems is `tcsh`, an enhanced C shell that popularized features such as command-line editing and multi-mode filename and command completion. Even if you don’t use `tcsh` as the default new user shell (we suggest using `bash`), you should still provide `tcsh` startup files in case your users happen to come across `tcsh`.

You don’t have to worry about the difference between login shells and non-login shells in `tcsh`. Upon startup, `tcsh` looks for a *.tcshrc* file. Failing this, it looks for the `csh` shell’s *.cshrc* startup file. The reason for this order is that you can use the *.tcshrc* file for `tcsh` extensions that don’t work in `csh`. You should probably stick to using the traditional *.cshrc* instead of *.tcshrc*; it’s highly unlikely that anyone will ever use your startup files with `csh`. And if a user actually does come across `csh` on some other system, your *.cshrc* will work.

Example .cshrc

Here is sample *.cshrc* file:

```
# Command path.
setenv PATH /usr/local/bin:/usr/bin:/bin:$HOME/bin

# EDITOR and VISUAL determine the editor that programs such as less
# and mail clients invoke when asked to edit a file.
setenv EDITOR vi
setenv VISUAL vi

# PAGER is the default text file viewer for programs such as man.
setenv PAGER less

# These are some handy options for less.
setenv LESS meiX

# By default, give other users read-only access to most new files.
umask 022

# Customize the prompt.
# Substitutions include:
# %n username %m hostname %/ current directory
# %h history number %l current terminal %% %
set prompt="%m%% "
```

13.5 Default User Settings

The best way to write startup files and choose defaults for new users is to experiment with a new test user on the system. Create the test user with an empty home directory and refrain from copying your own startup files to the test user's directory. Write the new startup files from scratch.

When you think you have a working setup, log in as the new test user in all possible ways (on the console, remotely, and so on). Make sure that you test as many things as possible, including the windowing system operation and manual pages. When you're happy with the test user, create a second test user, copying the startup files from the first test user. If everything still works, you now have a new set of startup files that you can distribute to new users.

The following sections outline reasonable defaults for new users.

13.5.1 Shell Defaults

The default shell for any new user on a Linux system should be `bash` because:

- Users interact with the same shell that they use to write shell scripts (for example, `csh` is a notoriously bad scripting tool—don't even think about it).
- `bash` is standard on Linux systems.
- `bash` uses GNU readline, and therefore its interface is identical to that of many other tools.
- `bash` gives you fine, easy-to-understand control over I/O redirection and file handles.

However, many seasoned Unix wizards use shells such as `csh` and `tcsh` simply because they can't bear to switch. Of course, you can choose any shell you like, but choose `bash` if you don't have any preference, and use `bash` as the default shell for any new user on the system. (A user can change his or her shell with the `chsh` command to suit individual preferences.)

NOTE

There are plenty of other shells out there (`rc`, `ksh`, `zsh`, `es`, and so on). Some are not appropriate as beginner shells, but `zsh` and `fish` are sometimes popular with new users looking for an alternative shell.

13.5.2 Editor

On a traditional system, the default editor should be `vi` or `emacs`. These are the only editors virtually guaranteed to exist on nearly any Unix system, which means they'll cause the least trouble in the long run for a new user. However, Linux distributions often configure `nano` to be the default editor, because it's easier for beginners to use.

As with shell startup files, avoid large default editor startup files. A little `set showmatch` in the *.exrc* startup file never hurt anyone but steer clear of anything that significantly changes the editor's behavior or appearance, such as the `showmode` feature, auto-indentation, and wrap margins.

13.5.3 Pager

It's perfectly reasonable to set the default `PAGER` environment variable to `less`.

13.6 Startup File Pitfalls

Avoid these in startup files:

- Don't put any kind of graphical command in a shell startup file.
- Don't set the `DISPLAY` environment variable in a shell startup file.
- Don't set the terminal type in a shell startup file.
- Don't skimp on descriptive comments in default startup files.

- Don’t run commands in a startup file that print to the standard output.
- Never set `LD_LIBRARY_PATH` in a shell startup file (see 15.1.4 Shared Libraries).

13.7 Further Startup Topics

Because this book deals only with the underlying Linux system, we won’t cover windowing environment startup files. This is a large issue indeed, because the display manager that logs you in to a modern Linux system has its own set of startup files, such as *.xsession*, *.xinitrc*, and the endless combinations of GNOME- and KDE-related items.

The windowing choices may seem bewildering, and there is no one common way to start a windowing environment in Linux. The next chapter describes some of the many possibilities. However, when you determine what your system does, you may get a little carried away with the files that relate to your graphical environment. That’s fine, but don’t carry it over to new users. The same tenet of keeping things simple in shell startup files works wonders for GUI startup files, too. In fact, you probably don’t need to change your GUI startup files at all.