Latest: Productivity tips for academics
Next: Relational shell programming
Prev: 12 resolutions for programmers
Rand: An implementation of RSA in Scheme

Settling into Unix

[article index] [email me] [@mattmight] [+mattmight] [rss]

Unix plays a supporting role in all of the classes I teach, but it is *essential* in my Scripting Language Design and Implementation course.

Unix is an operating *eco*system for executing and composing scripts.

It ranks high on my list of what every CS major should know.

Unfortunately, students no longer arrive familiar with Unix-style computing, *and* most are not willing to learn it on their own.

Some even see it as a baroque *regression*.

I wrote a brief survival guide to Unix to help students, and this post is a follow-up on taking the next steps: choosing, configuring and customizing your shell, your text editor and your environment.

Please feel free to forward this to anyone that needs a summary of the basic options and pointers to advanced ones (like programmable tab completion).

Future posts in this series will cover software development under Unix; shell scripting; choosing and customizing a window manager; and recipes and strategies for Unix system administration.

Interactive shells

Your interactive shell determines the command-line dialect you will use to interact with your machine.

There are two dominant families of interactive shells: Bourne-shell-compatible and C-shell-compatible.

By default, you've probably been provided with bash, the most popular of the Bourne-shell compatibles.

You can use `echo $SHELL` to figure out which shell you're using:

```
$ echo $SHELL
/bin/bash

$ _
```

Tip: The default Bourne-shell-compatible prompts end with `$` while the default C-shell-compatible prompts end with `%` or `>`. (zsh is an exception to this.) Root (super-user) prompts often end with `#` by default. Of course, this behavior is configurable.

It's a near certainty that a bash interpreter will be available on a Unix system; bash has become the JVM of the Unix world.

For interactive purposes, Bourne-shell-compatible and C-shell-compatible are roughly the same, but they differ substantially for scripting.

Many users (including myself) prefer the Bourne-shell-compatible zsh for interactive purposes.

The modern version of the C shell is tcsh.

You can change your default shell with the command `chsh`:

```
$ chsh -s /bin/zsh
Password for user name:

$ _
```

You can also switch to another shell by typing *`name-of-shell`*:

```
$ tcsh
```

```
> exit

$ _
```

Tip: Hitting *control-d* (which sends "end-of-file") in a shell will often terminate it as well.

Glob matching

Shells support glob matching on filenames.

A glob pattern uses the multicharacter wildcard `*` to match any number of characters (including no characters). For example, `foo*` would match `foo`, `foobar` or `foobaz`. but not `fo`

A glob pattern uses the one-character wildcard `?` to match exactly one unknown character. For example, `?at` would match `cat` and `hat`, but not `ccat`.

A glob pattern uses set pattern `[`*`characters`*`]` to match any of the specificied characters. For example, `[cfh]at` would match `cat`, `fat` and `hat`, but not `mat`.

Shells expand glob patterns into lists of filenames.

For example, to list all of the files in the current directory starting with `f` or `a` and ending with z, you can use `[fa]*z`:

```
$ ls
axb axyz fsz baz

$ echo [fa]*z
axyz fsz

$ _
```

On most systems, the shells live in `/bin`. You can use `ls` with a glob match to see which shells are available:

```
$ ls /bin/*sh
/bin/bash /bin/csh /bin/ksh /bin/sh /bin/tcsh
/bin/zsh

$ _
```

There are many shell-specific extensions to glob-matching.

For instance, in zsh you can qualify a glob match to restrict it. For example, the qualifier `(@)` would list only the symbol links in a directory:

```
$ echo *
file1 file2 link-to-file1

$ echo *(@)
link-to-file1

$ _
```

Tab completion

Most shells support tab completion.

When you have partially typed a command, filename or shell variable, press *tab* to have it autocomplete the remainder.

Where there is more than one completion, the shell will list the possibilities.

In some interactive shells, like bash and zsh, tab completion behavior is programmable.

Shell scripting

Shell languages are actually full programming languages.

Shell scripts are programs written in the syntax of an interactive shell.

Shell scripts are useful for automating repetitive or complex tasks.

Common applications of shell scripts include:

- data backup and restoration;
- cleaning up a directory;
- invoking a program with complex options;
- running commands on remote systems; and
- analyzing files.

You might also use shell scripts for one-time but complex tasks like finding and condensing all the duplicate mp3s spread across your hard drives.

For writing shell scripts, one can decide the shell on a per-script basis.

The top line of the file will contain the path to the shell, preceded by `#!`.

That is, you could use zsh as your interactive shell, but have shell scripts written in a mixture of tcsh and bash.

In fact, some shells, like Scheme shell are meant to be used solely as

scripting shells rather than interactive shells.

To create and run a shell script:

1. create a file with `#!/path/to/shell` at the top;
2. `chmod` it executable, i.e, `chmod u+x filename`;
3. and then run it with `./filename`.

For example, the following creates and executes a simple bash shell script:

```
$ cat > mycommand
#!/bin/bash

echo Hello world.
control-d

$ chmod u+x mycommand

$ ./mycommand
Hello world.

$ _
```

Text editors

In the Unix tradition, everything is stored as text.

Manipulating text efficiently is what text editors in the Unix tradition do.

Most Unix users use either emacs (or xemacs) or vim.

Some (like myself) use both.

For new users, the editor `pico` is popular, since it displays its short-cut keys (like *control-x* for exit) at the bottom.

Unix users should master emacs, vim or both.

It will take about thirty minutes to complete the tutorial for either one.

Try each one out.

Emacs

Emacs is the kitchen-sink approach to text editors.

Emacs is really a Lisp interpreter inside of which a text editor and many other applications live.

For almost any task, there exists an emacs plug-in to do it.

Complex editing tasks in emacs are invoked through key combinations.

New users find some of these combinations complex at first. (Even exiting the program is *control-x control-c*.)

Emacs is heavily customizable through scripts written in emacs lisp, a dynamically scoped variant of the Lisp programming language.

The per-user configuration file for emacs is `~/.emacs`.

To take the emacs tutorial, run `emacs` and press *control-h* and then hit *t*.

Vim

Vim (an improved version of the original vi editor) takes a different approach to editing text than most editors: typical user behavior is to toggle between an insertion mode and a command ("normal") mode.

There is also a "visual" mode, a "command-line" mode and an "ex" mode.

In insertion mode, you are directly editing text.

In command mode, every key press might perform an editing operation on the file. Complex changes take only a few keystrokes.

To switch from command mode to insertion mode, press *i*.

To switch from insertion mode to command mode, press *escape* (or *control-c*).

To quit without saving, hit `:q!` and *enter*.

Vim is customizable through vimscript.

The per-user configuration file for vim is `~/.vimrc`.

To take the vim tutorial, run `vim` and then press `:help tutor`, or run `vimtutor` directly.

Dot files

Configuration files for individual users live in their home directory, and by convention, have a `.` in front of them.

By default, these files are not shown with commands like `ls`.

To see the hidden dot files in the home directory, use `ls -a ~`:

```
$ ls -a ~
.
..
.bash_history
.bash_profile
.htaccess
.emacs
.ssh
.vimrc
.zprofile
docs
repos

$ _
```

Modifying these files customizes the behavior of programs and the environment.

The site dotfiles.org is a collection of user-submitted configuration files. It's a great way to see how other people customize their Unix environment.

Customizing the shell

To customize bash, add commands to run every time you login to a bash shell in `.bash_profile`.

To customize zsh, the equivalent file is `.zprofile`.

For tcsh, it's `.tcshrc`.

It's common to place alias directives in these files.

An alias is an abbrevation for a command.

For instance, I have an alias set in my `.zprofile` for each account to which I frequently ssh:

```
alias utah='ssh might@shell.cs.utah.edu'
alias might='ssh matt@might.net'
```

The alias syntax is slightly different for C-shell-compatible shells:

```
alias utah ssh might@shell.cs.utah.edu
alias might ssh matt@might.net
```

For example, if I want to check the amount of free disk space on `might.net`, I can do it quickly with the command `might df`.

Environment variables

Much of the Unix environment is customized through environment

variables. Many standard environment variables are set in the shell initialization files (`.bash_profile`, `.zprofile`, `.tcshrc`).

When writing shell scripts, environment variables also play the same roll as regular variables in ordinary programming languages.

To view the current values of environment variables, run the command `env`.

You can also access the contents of an environment variable in a shell by prefixing it with `$`:

```
$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin

$ _
```

In Bourne-compatible shells, the assignment operator `=` sets environment variables:

```
$ foo=3

$ echo $foo
3

$ _
```

In C-shell-compatible shells, the `set` command sets environment variables:

```
> set foo=3

> echo $foo
3

> _
```

Exported variables

When a one program invokes another program, the child program receives all of the *exported* environment variables of the parent.

By default, environment variables are *not* exported.

For example, in the following, the child bash process cannot see `$foo` from the parent:

```
$ foo=3

$ echo $foo
```

```
3

$ bash

$ echo $foo

$ exit

$ echo $foo
3

$ _
```

To export a variable under Bourne-compatible shells, use `export`:

```
$ export foo=3

$ echo $foo
3

$ bash

$ echo $foo
3

$ exit

$ echo $foo
3

$ _
```

To export a variable on C-compatible shells, use `setenv`:

```
> setenv foo 3

> echo $foo
3

> tcsh

> echo $foo
3

> exit

> echo $foo
3

> _
```

Command path

The PATH variable is a colon-delimited list of directories to search for

commands.

For instance, if `PATH` is `/bin:/usr/bin:/sbin`, then if the user types the command *`name`*, the shell will check for an executable file named `/bin/`*`name`*, then `/usr/bin/`*`name`*, then `/sbin/`*`name`* when completing the request.

Users often like to add custom scripts to their command path.

One way to accomplish this is to create a user-specific command directory, `~/bin`, store scripts here, and then add it to the `PATH` variable.

Usually, the `PATH` variable is set and exported by the shell initialization file.

Prompts

To customize the prompt, the `PS1` variable controls the look and feel under Bourne-compatible shells, while the `prompt` variables does the same for C-shell-compatible shells.

There are many shell-specific escapes one can add to make the prompt's behavior dynamic:

- bash prompt guide
- zsh and csh customization

Every advanced Unix user has invested in prompt customization well past the point of diminishing returns.

It is a rite of passage.

My fancy zsh prompt looks like a smurf impaled on a Christmas tree.

It displays more information than I've ever needed or used.

But, damn, it looks cool.

Colors

Enabling color is shell- and even program-specific, but it helps a lot.

To control colors in `ls` set the `LS_COLORS` variable. You may need to then alias `ls` to `ls --color` or `ls -G` depending on your Unix flavor.

To enable color in grep, alias `grep` to `grep --color`, and then set `GREP_COLOR`.

To control colors in the prompt, there are shell-specific methods best learned by Google, trial and error.

Other common shell variables

Most shells have the following set:

variable	description
DISPLAY	tells X11 on which display to open windows
EDITOR	default text editor; usually `emacs` or `vim`
HOME	path to user's home directory; same as `~`
PAGER	default page-scroller to use; usually `less`
PWD	current directory; same as output of `pwd`
SHELL	path to the current shell
TERM	current terminal type
USER	account name of current user

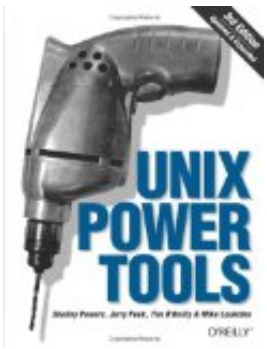
What's next?

After settling into Unix, you ought to investigate software development under Unix; customizing a window manager for X; and the art of Unix systems administration.

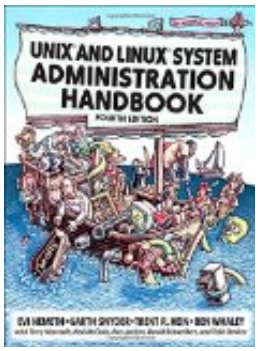
Good books

- The Linux Programming Interface:

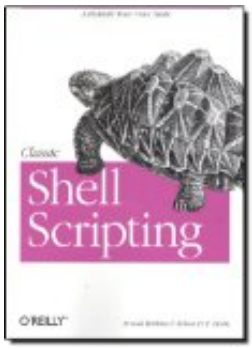
- Unix Power Tools:



- The UNIX and Linux System Administration Handbook:



- Classic Shell Scripting:



Related posts

- Learn Perl by experiment
- A quick overview of programming with bash
- A short introduction to make
- SSH hacks
- Standalone lexers with lex: synopsis, examples, and pitfalls
- Sculpting text with regex, grep, sed and awk
- Relational shell programming
- A survival guide for Unix beginners
- Console productivity hack: Exploiting task frequency
- HOWTO: Word, Excel and PowerPoint without MS Office
- Tips, tricks and tools for Linux and Unix

[article index] [email me] [@mattmight] [+mattmight] [rss]

Latest: Productivity tips for academics
Next: Relational shell programming
Prev: 12 resolutions for programmers
Rand: An implementation of RSA in Scheme