The Narrative

This article lives in the "narrative" section, which has 480 articles.

Using /usr/local

Posted on November 30, 2005 by Dan Benjamin
Tagged as

In ancient times, the dinosaurs ruled the earth. Amazon.com ~~– with its then white background, default serif font, and unadorned submit buttons –~~ sold a few programming-related books. Servers ran operating systems like SunOS 4.1.2 and AIX on expensive hardware. FreeBSD was still 386BSD, and the Linux kernel was just a gleam in some Finnish guy's eye. You were working as a Sr. Systems and Network Administrator for an aerospace company's corporate office.

At that time, computer hardware was expensive. Hard drives were *very* expensive, and their storage capacity ~~– considered ample in those days –~~ would be Lilliputian by today's standards.

Enter Partitions

To get around the limitation of small and expensive hard drives, UNIX systems allowed for the connection of many hard drives (using an external SCSI interface, remember link-lights, terminators, and beloved bus-errors, kids?).

Administrators could partition hard drives into separate logical divisions for these different parts of the filesystem, and they could also use *individual hard drives* for this purpose. These hard drives would then *become* paths in the filesystem using *mount points* (directories that are actually references to a hard drive).

Today's UNIX and UNIX-like systems (such as Mac OS X, Linux, FreeBSD, Solaris, etc.) still work this way. If one opens a Terminal window and runs a command like `ls /`, one would see a listing of the

main folders on the system. On many UNIX systems, mounting drives has become automatic, and the *mount points* for the drives live in a folder called `/mnt`. In Mac OS X, they live in a folder called `/Volumes`. If one were to do an `ls` in these folders, one might see a list of the external (or remote!) drives connected to the machine.

Hard Drives as Mount Points

In the old days (and today too, if needed), setting mount points for drives was usually a manual process. One would edit a file like `/etc/fstab` (still around on most UNIX systems, but managed automatically) and hard-code the mount points for each drive.

On any UNIX system, typing `mount` at a prompt will show the currently-mounted filesystems and their mount points. Here's what shows up for an old FreeBSD system you use (some extraneous information removed):

```
/dev/wd0s1a on /
/dev/wd0s1e on /tmp
/dev/wd0s1f on /usr
/dev/wd0s1d on /var
```

Without getting into too much detail here, looking at that list, we can tell that there's one hard drive in use here (the `wd0` tells us that), but there are many partitions on that drive (the `s1a`, `s1e` bits stand for drive *slices*, a.k.a. partitions).

An administrator of this machine could, for example, see that the `/var` partition was filling up … maybe users aren't pulling down their mail to their local machines but are leaving it on the server (a common problem in those days).

In response to this ~~– and after weeks of pleading with their boss for the thousands of dollars it would cost for a 400MB SCSI 2 drive –~~ the admin could drive in to work at 5am on Sunday morning, shut the server down, connect the new drive, boot up into single-user mode, partition and format the drive, copy the contents of the old `/var` folder over to it, rename the old `/var` folder to `/var.old`, set the mount point for the new drive to be `/var` in `/etc/fstab`, and then boot into multi-user mode.

Sounds fun, right?

The benefit of this would be that instead of a paltry 20MB `/var` partition on the primary system drive to hold everybody's mail, the admin would now have a ginormous 400MB partition.

Certainly that much space would last a lifetime.

Introducing `/usr/local` (and FHS)

Today, especially for "normal" users, understanding drives and partitions this way is much less important. In fact, in most cases, knowing about the UNIX underpinnings of an operating system like Mac OS X doesn't get you a whole lot … until you want to roll your own Ruby, Ruby on Rails and Lighttpd installation (the advantages of which are many, one of the main ones is detailed below).

Readers may be asking themselves: "Why do I need to know about partitions and drives in order to roll my own Ruby, Rails, and Lighttpd installation?"

To understand the importance of this, we need to understand one more thing about UNIX filesystems,

namely the Filesystem Hierarchy Standard, or FHS.

The FHS sets up basic rules for use by UNIX and UNIX-like systems when mapping out a filesystem. Mac OS X breaks with this default in some places (as do many UNIX variants) but overall, most systems respect this organization system.

Running a Customized yet Still "Stock" System

Running with a heavily customized installation of your favorite operating system can make using your computer a more pleasant, and often more productive task, and installing third-party applications is almost always a necessity.

Mac OS X, for example, is smart enough to realize that you'll want to install your own apps, and makes special accommodations for you to do this **without doing any harm to the rest of the system**. On OS X, just create a folder in your home directory named `Applications`, and OS X will "magically" give it the Applications icon, and treat it the same way it does with the "normal" Applications folder at the root of your hard drive.

In the same way, UNIX and UNIX-like systems make the same provision for you, using a method suggested by the FHS … the `/usr/local` folder.

The FHS defines the `/usr/local` as the "tertiary hierarchy for local data installed by the system administrator." Translated into English, this means **put apps you build yourself here**.

Think of `/usr/local` as a "safe haven" for command-line, open source, and similar utilities and programs you download or build yourself. Just like Santiago in A Few Good Men ... it is *not to be touched*.

Using the `/usr/local` folder as the destination when compiling and installing command-line software (such as Ruby, Rails, Lighttpd, wget, Apache, etc.) is critical for many reasons but there's one big one we'll discuss here: System Updates and their system-wide impact.

Solving the System Update "Problem"

Mac OS X, Windows, commercial UNIX systems, and Linux distributions all make use of software updates to deliver newer (and theoretically improved) versions of their software to their users.

This process involves an automatic or manual process where the operating system checks for an update from the mothership, and then downloads and installs the update, which usually consists of newer versions of applications, bugfixes, files, etc.

These new components are often moved into place with brute force … regardless of what was there before. It's possible (and probable) that any customizations one may have made to the system's files, binaries, executables, or Applications can **and will** be overwritten at will by the software update … but only if these files were placed outside of their safe haven, `/usr/local`.

Imagine the scenario: a developer has just compiled and installed an updated copy of the most cutting-edge version of Ruby, which her software relies upon to function. But, not knowing, she accepted the default paths at compile time, and the binary now lives in `/usr/bin`.

Later, after the project launches and users are logging in, she notices that there's a software update and

lets it run. Unfortunately, she doesn't see the note about "system binaries" being updated.

Suddenly, upon reboot, her software stops working and she can't figure out why. Everything else seems fine, there are no errors.

It turns out that the update has actually overwritten the Ruby binary she's compiled with a different version, and her software no longer works.

This could have been avoided if she'd installed the Ruby binary in a safer place, and set her shell and her software to use *that* binary.

In general, regardless of platform, software updates will respect the FHS's designation of `/usr/local` as untouchable.

Using `/usr/local`

Using `/usr/local` is easy. There are only three things one needs to do:

Compile Using `--prefix`

When compiling command line utilities, programs, and system tools, tell the system to install things into `/usr/local`. Just append `--prefix=/usr/local` to the `./configure` command in the build process.

Set The Path

By default, tell the system to look in `/usr/local` for files *first* by editing your path. On Mac OS X, either create or edit a file called `.bash_login` in your home folder (note the "."), it's a hidden file) and add the following line to it:

```
export PATH="/usr/local/bin:/usr/local/sbin:$PATH"
```

Specify the `/usr/local` Prefix in Scripts

When writing scripts, specify the full path to the executable you've installed in the first line of the script. So to specify a Ruby binary in `/usr/local/bin`, one would use a line like this:

```
#!/usr/local/bin/ruby
```

That's all there is to it.

Sponsored By

Photostream