CST8177

Scripting 1: Why?

Why scripting?

Why would anyone want to know how to write a script?

Why is is particularly important for a sysadmin?

Here are 7 reasons to consider:

1. avoid complex typing, preventing possible errors
2. automate repetitive tasks
3. use when an alias gets too complex or not possible
4. make new, specialized commands
5. automate long and/or complex tasks
6. handle rare but complex activities
7. create a "wrapper" for a program

These are all valid reasons, especially for a sysadmin managing a Linux/Unix server on behalf of an enterprise. The reasonable use of scripting will make you more productive, more accurate, and more efficient, increasing your value to your employer.

1. avoid complex typing, preventing possible errors

Imagine that you wish to perform an essentially simple action, but you first have to establish a certain environment or set variables in the environment, such as the pid number of a service or something similar. Imagine also that there's a choice of method depending on what you're about to do, each one essentially simple in itself.

Write a script, then, to make the choice and establish the desired environment. You will make fewer mistakes, especially if this is a task not often done.

1. avoid complex typing, preventing possible errors

Example:

In order to work with a raw hard drive, on the partition table for example, you must first lock the device in one of a few ways in order to prevent **automount** from mounting it as part of the file system.

The GNU partition editor **/usr/sbin/gparted** is a script which uses **devkit-disks** and possibly **hal-lock** for mount prevention if the binaries exist and their daemons are running. If neither is available, then it just runs **/usr/bin/gpartedbin** (the program) and hopes.

Setting both lock types requires this command:

```
devkit-disks --inhibit -- \
  hal-lock --interface org.free[too long]... \
   --exclusive --run "$BASE_CMD"
```

2. automate repetitive tasks

There's something you have to run daily, weekly, and monthly, or maybe at different event-based times. It's the same thing over and over, and you find it tedious. Boring. Not worth your time. You'd rather play Solitaire.

Write a script so you only have to start it. Maybe you can even get **`cron`** or **`anacron`** to run it on schedule, send errors to the system log, and only have to check up on it every now and again.

Sounds good to me!

2. automate repetitive tasks

Example:

The **anacron** program itself needs invoke each valid program in a directory. It needs to skip any program (executable or script) related to the RPM package manager's backup files or any **vim** swap files, or any program specifically denied or allowed. It also needs to log the process for each job run, with its results.

The **anacron** program **/usr/sbin/run-parts** is a script, and does the task as outlined above, logging messages to the **cron.notice** facility/priority via **logger** at the start and end of each job run. It's given the daily, weekly, and monthly directories (**/etc/cron.daily**, **/etc/cron.weekly**, and **/etc/cron/monthly**) from **/etc/anacrontab** at the appropriate time, and runs everything it needs to. The sysadmin need only ensure the right script or program is in the correct directory.

3. use when an `alias` gets too complex or not possible

The man page for **`bash`** says this about the **`alias`** built-in:

> Aliases allow a string to be substituted for a word when it is used as the first word of a simple command. ... The first word of each simple command, if unquoted, is checked to see if it has an alias. If so, that word is replaced by the text of the alias.
>
> [snip]
>
> There is no mechanism for using arguments in the replacement text. If arguments are needed, a shell function should be used (see FUNCTIONS below).

An alias is a simple substitution with no possible arguments, and anything complex more requires a script or a function (a type of script).

3. use when an alias gets too complex or not possible

Example:

If I want a short command to list all the soft links in a directory, I will discover that there is no way to use **alias** to do it. This is just not possible, when I want to put a directory reference where the **???** is.

```
alias lsl="ls -l ??? | grep '^l'"
```

However, this will work just fine as the script file **lsl**:

```
#! /bin/bash
ls -l $@ | grep '^l'
```

```
System Prompt$ lsl /etc/rc.*
lrwxrwxrwx. 1 root root 13 date /etc/rc.local ->
                                rc.d/rc.local
lrwxrwxrwx. 1 root root 15 date /etc/rc.sysinit ->
                                rc.d/rc.sysinit
```

4. make new, specialized commands

The **lsl** script above is a new, specialized, but trivial command. There are often opportunities to create much more useful commands (I can easily type in an **lsl**-equivalent when I need it).

Consider a more complicated and more useful command, such as the **service** command we've just addressed for controlling daemon processes. It is a script.

See service(8) for details on how to use it. It has 3 forms:

```
service SCRIPT COMMAND [OPTIONS]
service --status-all
service --help | -h | --version
```

The second and third forms are processed locally, inside the script. The first form invokes the control script named **SCRIPT** from the **/etc/rc.d/init.d** directory, the same scripts used by **init** at the system start-up.

4. make new, specialized commands

Examples:

```
System Prompt$ service -h
Usage: service < option > | --status-all |
   [ service_name [ command | --full-restart ] ]
System Prompt$ service --version
service ver. 0.91
System Prompt$ service crond status
crond (pid  2779) is running...
System Prompt$ service ups status
upsd is stopped
upsmon is stopped
```

5. automate long and/or complex tasks

Some tasks can require a lot of command typing or force you to put in complicated information (we've already seen some of that). For example, it is sometimes difficult to locate an entry in the man pages, often because of the form of the name or because an entry is hidden behind another of a lower section number.

Yet there is a **whatis** database that contains short descriptions of some system commands by keyword. It would be lovely to be able to search it simply and get man page references.

There are 2 simple ways to do that, neither requiring any knowledge of the database files involved. They are the **-k** option of the **man** command which just invokes the **/usr/bin/apropos** script. The script itself simply uses **grep** and is aware of the **whatis** file location and format.

5. automate long and/or complex tasks

Example:

The description in the man page for **man -k** is:

-k Equivalent to apropos

The **apropos** script is what does all the heavy lifting. Its own man page says:

> **apropos** searches a set of database files containing short descriptions of system commands for keywords and displays the result on the standard output.

It searches the **whatis** database for man pages. Reading whatis(1) gives a little information:

> **whatis** searches a set of database files containing short descriptions of system commands for keywords and displays the result on the standard output. Only complete word matches are displayed.

This is the key line in **apropos:**

```
grep -"$grepopt1" "$grepopt2""$1" $d/whatis
```

6. handle rare but complex activities

There will be activities that will be done only rarely, and naturally become scripts to store the research that has been done on them. There are annual activities for many machines, and indeed even monthly activities can be difficult to recall.

You could document it in a lab book and re-type everything when it's needed, but why not just create a script?

You may also be required to provide an occasional service to users, perhaps to train them in the use of some tool or facility. Scripts can be handy to avoid you going through a setup process on many user systems.

6. handle rare but complex activities

Example:

You can set up **vim** for a tutoring session using the **/usr/bin/vimtutor** script. The facility itself is not complex, but because it's been designed for novice users, it has to cover a wide range of input possibilities.

For example, it supports 11 different names for **vim** plus 10 more for **gvim** (graphical **vim**), which it also supports. It then uses both a standard (**/tmp**) and a non-standard way to create a temporary work file. It starts the correct **vim** from the list of 21 for the session, and even sets up a mechanism for resuming control to clean up for the user after the session is over.

Most of this could, in theory, be handled by some documentation on the process, but the script approach minimizes calls from users for assistance. That's well worth a 74-line script.

7. create a "wrapper" for a program

This may well be the most common reason for a script. We have seen how **vim** tutoring uses a setup script, but it's not exactly a wrapper. Examples include **/usr/bin/firefox** and both **/usr/bin/vmware** and **/usr/bin/vmplayer**.

A simple wrapper is **/bin/gunzip** which invokes the regular **/bin/gzip** program with unzipping arguments. Here is the complete wrapper, less **help** and **usage**:

```
#!/bin/sh
PATH=/bin:$PATH
exec gzip -d "$@"
```

Notice that the hash-bang line invokes **/bin/sh**, the standard shell, and not the bright, shiny new **bash**. It's often called **shell**, but it's spelled **sh**. It's used when wide distribution is needed, since every system must have **sh**.

The script first makes sure the **PATH** includes **/bin** and then invokes **gzip -d** with all the arguments given.

Why write scripts?

The answer to the original question should now be clear in your mind. There's really no alternative to effective scripting for a top-level sysadmin.

We've looked at **bash** and **sh** scripts (all **sh** scripts can run in **bash**; the reverse is not necessarily true) but there are also occasions to use Perl, Python, and many other scripting tools. You will learn to select the right tool for the task at hand through experience.

A script doesn't have to be long (see **gunzip**) or particularly complex (see **vimtutor**). Often much of the script is comments and blank lines for self-documentation and readability, or more importantly, error-checking the input arguments, configuration and other file existence to make the use of the script idiot-proof. The "meat" of a script is often only a few lines.