Systems, Tools, and Terminal Science

Shell from vi

Posted on

A good sign of a philosophically sound interactive Unix tool is the facilities it offers for interacting with the filesystem and the shell: specifically, how easily can you run file operations and/or shell commands with reference to data within the tool? The more straightforward this is, the more likely the tool will fit neatly into a terminal-driven Unix workflow.

If all else fails, you could always suspend the task with Ctrl+Z to drop to a shell, but it's helpful if the tool shows more deference to the shell than that; it means you can use and (even more importantly) *write* tools to manipulate the data in the program in whatever languages you choose, rather than being forced to use any kind of heretical internal scripting language, or worse, an over-engineered API.

`vi` is a good example of a tool that interacts openly and easily with the Unix shell, allowing you to pass open buffers as streams of text transparently to classic filter and text processing tools. In the case of Vim, it's particularly useful to get to know these, because in many cases they allow you to avoid painful Vimscript, and to do things your way, without having to learn an ad-hoc language or to rely on plugins. This was touched on briefly in the of the series.

By default, `vi` will use the value of your `SHELL` environment variable as the shell in which your commands will be run. In most cases, this is probably what you want, but it might pay to check before you start:

```
:set shell?
```

If you're using Bash, and this prints `/bin/bash`, you're good to go, and you'll be able to use Bash-specific features or builtins such as `[[` comfortably in your command lines if you wish.

You can run a shell command from `vi` with the `!` `ex` command. This is inherited from the same behaviour in `ed`. A good example would be to read a manual page in the same terminal window without exiting or suspending `vi`:

```
:!man grep
```

Or to build your project:

```
:!make
```

You'll find that exclamation point prefix `!` shows up in the context of running external commands pretty consistently in `vi`.

You will probably need to press Enter afterwards to return to `vi`. This is to allow you to read any output remaining on your screen.

Of course, that's not the only way to do it; you may prefer to drop to a forked shell with `:sh`, or suspend `vi` with `^Z` to get back to the original shell, resuming it later with `fg`.

You can refer to the current buffer's filename in the command with `%`, but be aware that this may cause escaping problems for files with special characters in their names:

```
:!gcc % -o foo
```

If you want a literal `%`, you will need to escape it with a backslash:

```
:!grep \% .vimrc
```

The same applies for the `#` character, for the *alternate buffer*.

```
:!gcc # -o bar
:!grep \# .vimrc
```

And for the `!` character, which expands to the previous command:

```
:!echo !
:!echo \!
```

You can try to work around special characters for these expansions by single-quoting them:

```
:!gcc '%' -o foo
:!gcc '#' -o bar
```

But that's still imperfect for files with apostrophes in their names. In Vim (but not `vi`) you can do this:

```
:exe "!gcc " . shellescape(expand("%")) . " -o foo"
```

The Vim help for this is at `:help :!`.

Also inherited from `ed` is reading the output of commands into a buffer, which is done by giving a command starting with `!` as the argument to `:r`:

```
:r !grep vim .vimrc
```

This will insert the output of the command *after* the current line position in the buffer; it works in the same way as reading in a file directly.

You can add a line number prefix to `:r` to place the output after that line number:

```
:5r !grep vim .vimrc
```

To put the output at the very start of the file, a line number of `0` works:

```
:0r !grep vim .vimrc
```

And for the very *end* of the file, you'd use `$`:

```
:$r !grep vim .vimrc
```

Note that redirections work fine, too, if you want to prevent `stderr` from being written to your buffer in the case of errors:

```
:$r !grep vim .vimrc 2>>vim_errorlog
```

To run a command with standard input coming from text in your buffer, but *without* deleting it or writing the output back into your buffer, you can provide a `!` command as an argument to `:w`. Again, this behaviour is inherited from `ed`.

By default, the whole buffer is written to the command; you might initially expect that only the current line would be written, but this makes sense if you consider the usual behaviour of `w` when writing directly to a file.

Given a file with a first column full of numbers:

```
304 Donald Trump
227 Hillary Clinton
3   Colin Powell
1   Spotted Eagle
1   Ron Paul
1   John Kasich
1   Bernie Sanders
```

We could calculate and view (but not save) the sum of the first column with `awk(1)`, to see the expected value of 538 printed to the terminal:

```
:w !awk '{sum+=$1}END{print sum}'
```

We could limit the operation to the faithless electoral votes by specifying a line range:

```
:3,$w !awk '{sum+=$1}END{print sum}'
```

You can also give a range of just `.`, if you only want to write out the current line.

In Vim, if you're using visual mode, pressing `:` while you have some text selected will automatically add the `'<,'>` range marks for you, and you can write out the rest of the command:

```
:'<,'>w !grep Bernie
```

Note that this writes every *line* of your selection to the command, not merely the characters you have selected. It's more intuitive to use visual line mode (Shift+V) if you take this approach.

If you want to *replace* text in your buffer by filtering it through a command, you can do this by providing a range to the `!` command:

```
:1,2!tr '[:lower:]' '[:upper:]'
```

This example would capitalise the letters in the first two lines of the buffer, passing them as input to the command and replacing them with the command's output.

```
304 DONALD TRUMP
227 HILLARY CLINTON
3 Colin Powell
1 Spotted Eagle
1 Ron Paul
1 John Kasich
1 Bernie Sanders
```

Note that the number of lines passed as input need not match the number of lines of output. The length of the buffer can change. Note also that by default any `stderr` is included; you may want to redirect that away.

You can specify the entire file for such a filter with `%`:

```
:%!tr '[:lower:]' '[:upper:]'
```

As before, the current line must be explicitly specified with `.` if you want to use only that as input, otherwise you'll just be running the command with no buffer interaction at all, per the first heading of this article:

```
:.!tr '[:lower:]' '[:upper:]'
```

You can also use `!` as a *motion* rather than an `ex` command on a range of lines, by pressing `!` in normal mode and then a motion (`w`, `3w`, `}`, etc) to select all the lines you want to pass through the filter. Doubling it (`!!`) filters the current line, in a similar way to the `yy` and `dd` shortcuts, and you can provide a numeric prefix (e.g. `3!!`) to specify a number of lines from the current line.

This is an example of a general approach that will work with any POSIX-compliant version of `vi`. In Vim, you have the `gU` command available to coerce text to uppercase, but this is not available in vanilla `vi`; the best you have is the tilde command `~` to *toggle* the case of the character under the cursor. `tr(1)`, however, is specified by POSIX–including the locale-aware transformation–so you are much more likely to find it works on any modern Unix system.

If you end up needing such a command during editing a lot, you could make a generic command for your , say named `upp` for uppercase, that forces all of its standard input to uppercase:

```
#!/bin/sh
tr '[:lower:]' '[:upper:]'
```

Once saved somewhere in `$PATH` and made executable, this would allow you simply to write the following to apply the filter to the entire buffer:

```
:%!upp
```

The main takeaway from this is that the scripts you use with your editor don't have to be in shell. You might prefer :

```
#!/usr/bin/awk -f
{ print toupper($0) }
```

Or :

```
#!/usr/bin/env perl
print uc while <>;
```

, or …

Incidentally, this "filtering" feature is where `vi`'s heritage from `ed` ends as far as external commands are concerned. In POSIX `ed`, there isn't a way to filter buffer text through a command in one hit. It's not too hard to emulate it with a temporary file, though, using all

the syntax learned above:

```
*1,2w !upp > tmp
*1,2d
*0r tmp
*!rm tmp
```

Posted in | Tagged , , , , , , ,

Bash hostname completion

Posted on

As part of its programmable completion suite, Bash includes **hostname completion**. This completion mode reads hostnames from a file in `hosts(5)` format to find possible completions matching the current word. On Unix-like operating systems, it defaults to reading the file in its usual path at `/etc/hosts`.

For example, given the following `hosts(5)` file in place at `/etc/hosts`:

```
127.0.0.1      localhost
192.0.2.1      web.example.com www
198.51.100.10  mail.example.com mx
203.0.113.52   radius.example.com rad
```

An appropriate call to `compgen` would yield this output:

```
$ compgen -A hostname
localhost
web.example.com
www
```

```
mail.example.com
mx
radius.example.com
rad
```

We could then use this to complete hostnames for network diagnostic tools like `ping(8)`:

```
$ complete -A hostname ping
```

Typing `ping we` and then pressing Tab would then complete to `ping web.example.com`. If the `shopt` option `hostcomplete` is on, which it is by default, Bash will also attempt host completion if completing any word with an `@` character in it. This can be useful for email address completion or for SSH `username@hostname` completion.

We could also trigger hostname completion in any other Bash command line (regardless of `complete` settings) with the Readline shortcut Alt+@ (i.e. Alt+Shift+2). This works even if `hostcomplete` is turned off.

However, with DNS so widely deployed, and with system `/etc/hosts` files normally so brief on internet-connected systems, this may not seem terribly useful; you'd just end up completing `localhost`, and (somewhat erroneously) a few IPv6 addresses that don't begin with a digit. It may seem even less useful if you have your own set of hosts in which you're interested, since they may not correspond to the hosts in the system's `/etc/hosts` file, and you probably really do want them looked up via DNS each time, rather than maintaining static addresses for them.

There's a simple way to make host completion much more useful by defining the `HOSTFILE` variable in `~/.bashrc` to point to any other file containing a list of hostnames. You could, for example, create a simple file `~/.hosts` in your home directory, and then include this in your `~/.bashrc`:

```
# Use a private mock hosts(5) file for completion
HOSTFILE=$HOME/.hosts
```

You could then populate the `~/.hosts` file with a list of hostnames in which you're interested, which will allow you to influence hostname completion usefully without messing with your system's DNS resolution process at all. Because of

`HOSTFILE` , you don't even have to fake an IP address as the first field; it simply scans the file for any word that doesn't start with a digit:

```
# Comments with leading hashes will be excluded
external.example.com
router.example.com router
github.com
google.com
...
```

You can even include other files from it with an `$include` directive!

```
$include /home/tom/.hosts.home
$include /home/tom/.hosts.work
```

Author's note: This really surprised me when reading the source, because I don't think `/etc/hosts` *files generally support that for their usual name resolution function. I would love to know if any systems out there actually do support this.*

The behaviour of the `HOSTFILE` variable is a bit weird; all of the hosts from the `HOSTFILE` are *appended* to the in-memory list of completion hosts each time the `HOSTFILE` variable is set (not even just changed), *and* host completion is attempted, even if the hostnames were already in the list. It's probably sufficient just to set the file once in `~/.bashrc`.

This setup allows you to set hostname completion as the default method for all sorts of network-poking tools, falling back on the usual filename completion if nothing matches with `-o default`:

```
$ complete -A hostname -o default curl dig host netcat ping telnet
```

You could also use hostname completions for `ssh(1)`, but to account for hostname aliases and `ssh_config(5)` , I prefer to read `Host` directives values from `~/.ssh/config` for that.

If you have machine-readable access to the complete zone data for your home or work domain, it may even be worth periodically enumerating all of the hostnames into that file, perhaps using `rndc dumpdb -zones` for a BIND9 setup, or using an `AXFR` request. If you have a locally caching recursive nameserver, you could even periodically examine the contents of its cache for new and interesting hosts to add to the file.

Posted in | Tagged , , ,

Custom commands

Posted on

As users grow more familiar with the feature set available to them on UNIX-like operating systems, and grow more comfortable using the command line, they will find more often that they develop their own routines for solving problems using their preferred tools, often repeatedly solving the same problem in the same way. You can usually tell if you've entered this stage if one or more of the below applies:

- You repeatedly search the web for the same long commands to copy-paste.
- You type a particular long command so often it's gone into muscle memory, and you type it without thinking.
- You have a text file somewhere with a list of useful commands to solve some frequently recurring problem or task, and you copy-paste from it a lot.
- You're keeping large amounts of history so you can search back through commands you ran weeks or months ago with `^R`, to find the last time an instance of a problem came up, and getting angry when you realize it's fallen away off the end of your history file.
- You've found that you prefer to run a tool like `ls(1)` more often with a non-default flag than without it; `-l` is a common example.

You can definitely accomplish a lot of work quickly with shoving the output of some monolithic program through a terse one-liner to get the information you want, or by developing muscle memory for your chosen toolbox and oft-repeated commands, but if you want to apply more discipline and automation to managing these sorts of tasks, it may be useful for you to explore more rigorously defining your own commands for use during your shell sessions, or for automation purposes.

This is consistent with the original idea of the Unix shell as a ; the tools provided by the base system are intentionally very general, not prescribing how they're used, an approach which allows the user to build and customize their own

command set as appropriate for their system's needs, even on a per-user basis.

What this all means is that you need not treat the tools available to you as holy writ. To leverage the Unix philosophy's real power, you should consider customizing and extending the command set in ways that are useful to you, refining them as you go, and sharing those extensions and tweaks if they may be useful to others. We'll discuss here a few methods for implementing custom commands, and where and how to apply them.

The first step users take toward customizing the behaviour of their shell tools is often to define shell aliases in their shell's startup file, usually specifically for interactive sessions; for Bash, this is usually `~/.bashrc`.

Some aliases are so common that they're included as commented-out suggestions in the default `~/.bashrc` file for new users. For example, on Debian systems, the following alias is defined by default if the `dircolors(1)` tool is available for coloring `ls(1)` output by filetype:

```
alias ls='ls --color=auto'
```

With this defined at startup, invoking `ls`, with or without other arguments, will expand to run `ls --color=auto`, including any given arguments on the end as well.

In the same block of that file, but commented out, are suggestions for other aliases to enable coloured output for GNU versions of the `dir` and `grep` tools:

```
#alias dir='dir --color=auto'
#alias vdir='vdir --color=auto'

#alias grep='grep --color=auto'
#alias fgrep='fgrep --color=auto'
#alias egrep='egrep --color=auto'
```

Further down still, there are some suggestions for different methods of invoking `ls`:

```
#alias ll='ls -l'
#alias la='ls -A'
#alias l='ls -CF'
```

Commenting these out would make `ll`, `la`, and `l` work as commands during an interactive session, with the appropriate options added to the call.

You can check the aliases defined in your current shell session by typing `alias` with no arguments:

```
$ alias
alias ls='ls --color=auto'
```

Aliases are convenient ways to add options to commands, and are very common features of `~/.bashrc` files shared on the web. They also work in POSIX-conforming shells besides Bash. However, for general use, . For one thing, you can't process arguments with them:

```
# An attempt to write an alias that searches for a given pattern in a fixed
# file; doesn't work because aliases don't expand parameters
alias grepvim='grep "$1" ~/.vimrc'
```

They also don't work for defining new commands within scripts:

```
#!/bin/bash
alias ll='ls -l'
ll
```

When saved in a file as `test`, made executable, and run, this script fails:

```
./test: line 3: ll: command not found
```

So, once you understand how aliases work so you can read them when others define them in startup files, my suggestion is there's no point writing any. Aside from some , they have no functional advantages over shell functions and scripts.

A more flexible method for defining custom commands for an interactive shell (or within a script) is to use a shell function. We could declare our `ll` function in a Bash startup file as a function instead of an alias like so:

```
# Shortcut to call ls(1) with the -l flag
ll() {
    command ls -l "$@"
}
```

Note the use of the `command` builtin here to specify that the `ll` function should invoke the *program* named `ls`, and not any *function* named `ls`. This is particularly important when writing a function wrapper around a command, to stop an infinite loop where the function calls itself indefinitely:

```
# Always add -q to invocations of gdb(1)
gdb() {
    command gdb -q "$@"
}
```

In both examples, note also the use of the `"$@"` expansion, to add to the final command line any arguments given to the function. We wrap it in double quotes to stop spaces and other shell metacharacters in the arguments causing problems. This means that the `ll` command will work correctly if you were to pass it further options and/or one or more directories as arguments:

```
$ ll -a
$ ll ~/.config
```

Shell functions declared in this way are specified by POSIX for Bourne-style shells, so they should work in your shell of choice, including Bash, `dash`, Korn shell, and Zsh. They can also be used within scripts, allowing you to abstract away multiple instances of

similar commands to improve the clarity of your script, in much the same way the basics of functions work in general-purpose programming languages.

Functions are a good and portable way to approach adding features to your interactive shell; written carefully, they even allow you to port features you might like from other shells into your shell of choice. I'm fond of taking commands I like from Korn shell or Zsh and implementing them in Bash or POSIX shell functions, such as Zsh's `vared` or its `cd` features.

If you end up writing a lot of shell functions, you should consider putting them into to keep your shell's primary startup file from becoming unmanageably large.

You can take a look at some of the shell functions I have defined here that are useful to me in general shell usage; a lot of these amount to implementing convenience features that I wish my shell had, especially for quick directory navigation, or adding options to commands:

- `sh`
-

-

You can manipulate variables within shell functions, too:

```
# Print the filename of a path, stripping off its leading path and
# extension
fn() {
    name=$1
    name=${name##*/}
    name=${name%.*}
    printf '%s\n' "$name"
}
```

This works fine, but the catch is that after the function is done, the value for `name` will still be defined in the shell, and will overwrite whatever was in there previously:

```
$ printf '%s\n' "$name"
foobar
$ fn /home/you/Task_List.doc
Task_List
$ printf '%s\n' "$name"
Task_List
```

This may be desirable if you actually want the function to change some aspect of your current shell session, such as managing variables or changing the working directory. If you *don't* want that, you will probably want to find some means of avoiding name collisions in your variables.

If your function is only for use with a shell that provides the `local` (Bash) or `typeset` (Ksh) features, you can declare the variable as local to the function to remove its global scope, to prevent this happening:

```
# Bash-like
fn() {
    local name
    name=$1
    name=${name##*/}
    name=${name%.*}
    printf '%s\n' "$name"
}

# Ksh-like
# Note different syntax for first line
function fn {
    typeset name
    name=$1
    name=${name##*/}
    name=${name%.*}
```

```
    printf '%s\n' "$name"
}
```

If you're using a shell that lacks these features, or you want to aim for POSIX compatibility, things are a little trickier, since local function variables aren't specified by the standard. One option is to use a , so that the variables are only defined for the duration of the function:

```
# POSIX; note we're using plain parentheses rather than curly brackets, for
# a subshell
fn() (
    name=$1
    name=${name##*/}
    name=${name%.*}
    printf '%s\n' "$name"
)

# POSIX; alternative approach using command substitution:
fn() {
    printf '%s\n' "$(
        name=$1
        name=${name##*/}
        name=${name%.*}
        printf %s "$name"
    )"
}
```

This subshell method also allows you to change directory with `cd` within a function without changing the working directory of the user's interactive shell, or to change shell options with `set` or Bash options with `shopt` only temporarily for the purposes of the function.

Another method to deal with variables is to manipulate the directly (`$1`, `$2` …) with `set`, since they are local to the function call too:

```
# POSIX; using positional parameters
fn() {
    set -- "${1##*/}"
    set -- "${1%.*}"
    printf '%s\n' "$1"
}
```

These methods work well, and can sometimes even be combined, but they're awkward to write, and harder to read than the modern shell versions. If you only need your functions to work with your modern shell, I recommend just using `local` or `typeset`. The Bash Guide on Greg's Wiki has a of functions in Bash, if you want to read about this and other aspects of functions in more detail.

As you get comfortable with defining and using functions during an interactive session, you might define them in ad-hoc ways on the command line for calling in a loop or some other similar circumstance, just to solve a task in that moment.

As an example, I recently made an ad-hoc function called `monit` to run a set of commands for its hostname argument that together established different types of monitoring system checks, using an existing script called `nmfs`:

```
$ monit() { nmfs "$1" Ping Y ; nmfs "$1" HTTP Y ; nmfs "$1" SNMP Y ; }
$ for host in webhost{1..10} ; do
> monit "$host"
> done
```

After that task was done, I realized I was likely to use the `monit` command interactively again, so I decided to keep it. Shell functions only last as long as the current shell, so if you want to make them permanent, you need to store their definitions somewhere in your startup files. If you're using Bash, and you're content to just add things to the end of your `~/.bashrc` file, you could just do something like this:

```
$ declare -f monit >> ~/.bashrc
```

That would append the existing definition of `monit` in parseable form to your `~/.bashrc` file, and the `monit` function would then be loaded and available to you for future interactive sessions. Later on, I ended up converting `monit` into a shell script, as its use wasn't limited to just an interactive shell.

If you want a more robust approach to keeping functions like this for Bash permanently, I wrote , which allows you to quickly store functions and variables defined in your current shell into separate and appropriately-named files, including viewing and managing the list of names conveniently:

```
$ keep monit
$ keep
monit
$ ls ~/.bashkeep.d
monit.bash
$ keep -d monit
```

Shell functions are a great way to portably customize behaviour you want for your interactive shell, but if a task isn't specific only to an interactive shell context, you should instead consider putting it into its own script whether written in shell or not, to be invoked somewhere from your `PATH`. This makes the script useable in contexts besides an interactive shell with your personal configuration loaded, for example from within another script, by another user, or by an X11 session called by something like `dmenu`.

Even if your set of commands is only a few lines long, if you need to call it often–especially with reference to other scripts and in varying contexts– making it into a generally-available shell script has many advantages.

`/usr/local/bin`

Users making their own scripts often start by putting them in `/usr/local/bin` and making them executable with `sudo chmod +x`, since many Unix systems include this directory in the system `PATH`. If you want a script to be generally available to all users on a system, this is a reasonable approach. However, if the script is just something for your own personal use, or if you don't have the permissions necessary to write to this system path, it may be preferable to have your own directory for logical binaries, including scripts.

Unix-like users who do this seem to vary in where they choose to put their private logical binaries directory. I've seen each of the below used or recommended:

- `~/bin`
- `~/.bin`
- `~/.local/bin`
- `~/Scripts`

I personally favour `~/.local/bin`, but you can put your scripts wherever they best fit into your `HOME` directory layout. You may want to choose something that fits in well with the , or whatever existing standard or system your distribution chooses for filesystem layout in `$HOME`.

In order to make this work, you will want to customize your login shell startup to include the directory in your `PATH` environment variable. It's better to put this into `~/.profile` or , so that it's only run once. That should be all that's necessary, as `PATH` is typically exported as an environment variable for all the shell's child processes. A line like this at the end of one of those scripts works well to extend the system `PATH` for our login shell:

```
PATH=$HOME/.local/bin:$PATH
```

Note that we specifically put our new path at the *front* of the `PATH` variable's value, so that it's the first directory searched for programs. This allows you to implement or install your own versions of programs with the same name as those in the system; this is useful, for example, if you like to experiment with `$HOME`.

If you're using a systemd-based GNU/Linux, and particularly if you're using a display manager like GDM rather than a TTY login and `startx` for your X11 environment, you may find it more robust to instead set this variable with the . Another option you may prefer on systems using PAM is to set it with .

After logging in, we first verify the directory is in place in the `PATH` variable:

```
$ printf '%s\n' "$PATH"
/home/tom/.local/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

We can test this is working correctly by placing a test script into the directory, including the `#!/bin/sh` , and making it executable by the current user with `chmod(1)`:

```
$ cat >~/.local/bin/test-private-bindir
#!/bin/sh
printf 'Working!\n'
^D
$ chmod u+x ~./local/bin/test-private-bindir
$ test-private-bindir
Working!
```

I publish the more `~/.local/bin`, which I keep up-to-date on my personal systems in version control using Git, along with my configuration files. Many of the scripts are very short, and are intended mostly as building blocks for other scripts in the same directory. A few examples:

- `gscr(1df)`: Run a set of commands on a Git repository to minimize its size.
- `fgscr(1df)`: Find all Git repositories in a directory tree and run `gscr(1df)` over them.
- `hurl(1df)`: Extract URLs from links in an HTML document.
- `maybe(1df)`: Exit with success or failure with a given probability.
- `rfcr(1df)`: Download and read a given document.
- `tot(1df)`: Add up a list of numbers.

For such scripts, I try to write them as much as possible to use tools specified by POSIX, so that there's a decent chance of them working on whatever Unix-like system I need them to.

On systems I use or manage, I might specify commands to do things relevant specifically to that system, such as:

- Filter out uninteresting lines in an Apache HTTPD logfile with awk.
- Check whether mail has been delivered to system users in `/var/mail`.
- Upgrade the Adobe Flash player in a private Firefox instance.

The tasks you need to solve both generally and specifically will almost certainly be different; this is where you can get creative with your automation and abstraction.

An additional advantage worth mentioning of using scripts rather than shell functions where possible is that they can be called from environments besides shells, such as in X11 or by other scripts. You can combine this method with X11-based utilities such as `dmenu(1)`, libnotify's `notify-send(1)`, or ImageMagick's `import(1)` to implement custom interactive behaviour for your X windows session, without having to write your own X11-interfacing code.

Of course, you're not limited to just shell scripts with this system; it might suit you to write a script completely in a language like , or even . If portability isn't a concern for the particular script, you should use your favourite scripting language. Notably, don't fall into the trap of implementing a script in shell for no reason …

```
#!/bin/sh
awk 'NF>2 && /foobar/ {print $1}' "$@"
```

… when you can instead write the whole script in the main language used, and save a `fork(2)` syscall and a layer of quoting:

```
#!/usr/bin/awk -f
NF>2 && /foobar/ {print $1}
```

Finally, if you end up writing more than a couple of useful shell functions and scripts, you should consider versioning them with Git or a similar version control system. This also eases implementing your shell setup and scripts on other systems, and sharing them with others via publishing on GitHub. You might even go so far as to write a to install them, or for quick reference as documentation … if you're just a little bit crazy …

Posted in | Tagged , , , , , , ,

Cron best practices

Posted on

The time-based job scheduler `cron(8)` has been around since Version 7 Unix, and its `crontab(5)` syntax is familiar even for people who don't do much Unix system administration. It's , reasonably flexible, simple to configure, and works reliably, and so it's trusted by both system packages and users to manage many important tasks.

However, like many older Unix tools, `cron(8)`'s simplicity has a drawback: it relies upon the user to know some detail of how it works, and to correctly implement any other safety checking behaviour around it. Specifically, all it does is try and run the job at an appropriate time, and email the output. For simple and unimportant per-user jobs, that may be just fine, but for more crucial system tasks it's worthwhile to wrap a little extra infrastructure around it and the tasks it calls.

There are a few ways to make the way you use `cron(8)` more robust if you're in a situation where keeping track of the running job is desirable.

The sixth column of a system `crontab(5)` file is the username of the user as which the task should run:

```
0 * * * *  root  cron-task
```

To the extent that is practical, you should run the task as a user with only the privileges it needs to run, and nothing else. This can sometimes make it worthwhile to create a dedicated system user purely for running scheduled tasks relevant to your application.

```
0 * * * *  myappcron  cron-task
```

This is not just for security reasons, although those are good ones; it helps protect you against nasties like scripting errors attempting to .

Similarly, for tasks with database systems such as MySQL, don't use the administrative `root` user if you can avoid it; instead, use or even create a dedicated user with a unique random password stored in a locked-down `~/.my.cnf` file, with only the needed

permissions. For a MySQL backup task, for example, only a few permissions should be required, including `SELECT`, `SHOW VIEW`, and `LOCK TABLES`.

In some cases, of course, you really will need to be `root`. In particularly sensitive contexts you might even consider using `sudo(8)` with appropriate `NOPASSWD` options, to allow the dedicated user to run only the appropriate tasks as `root`, and nothing else.

Before placing a task in a `crontab(5)` file, you should test it on the command line, as the user configured to run the task and with the appropriate environment set. If you're going to run the task as `root`, use something like `su` or `sudo -i` to get a root shell with the user's expected environment first:

```
$ sudo -i -u cronuser
$ cron-task
```

Once the task works on the command line, place it in the `crontab(5)` file with the timing settings modified to run the task a few minutes later, and then watch `/var/log/syslog` with `tail -f` to check that the task actually runs without errors, and that the task itself completes properly:

```
May  7 13:30:01 yourhost CRON[20249]: (you) CMD (cron-task)
```

This may seem pedantic at first, but it becomes routine very quickly, and it saves a lot of hassles down the line as it's very easy to make an assumption about something in your environment that doesn't actually hold in the one that `cron(8)` will use. It's also a necessary acid test to make sure that your `crontab(5)` file is well-formed, as some implementations of `cron(8)` will refuse to load the entire file if one of the lines is malformed.

If necessary, you can set arbitrary environment variables for the tasks at the top of the file:

```
MYVAR=myvalue

0 * * * *  you  cron-task
```

You've probably seen tutorials on the web where in order to keep the `crontab(5)` job from sending standard output and/or standard error emails every five minutes, shell redirection operators are included at the end of the job specification to discard both the standard output and standard error. This kluge is particularly common for running web development tasks by automating a request to a URL with `curl(1)` or `wget(1)`:

```
*/5 * * *  root  curl https://example.com/cron.php >/dev/null 2>&1
```

Ignoring the output completely is generally not a good idea, because unless you have other tasks or monitoring ensuring the job does its work, you won't notice problems (or know what they are), when the job emits output or errors that you actually care about.

In the case of `curl(1)`, there are just way too many things that could go wrong, that you might notice far too late:

- The script could get broken and return 500 errors.
- The URL of the `cron.php` task could change, and someone could forget to add a HTTP 301 redirect.
- Even if a HTTP 301 redirect is added, if you don't use `-L` or `--location` for `curl(1)`, it won't follow it.
- The client could get blacklisted, firewalled, or otherwise impeded by automatic or manual processes that falsely flag the request as spam.
- If using HTTPS, connectivity could break due to cipher or protocol mismatch.

The author has seen all of the above happen, in some cases very frequently.

As a general policy, it's worth taking the time to read the manual page of the task you're calling, and to look for ways to correctly control its output so that it emits only the output you actually want. In the case of `curl(1)`, for example, I've found the following formula works well:

```
curl -fLsS -o /dev/null http://example.com/
```

- `-f`: If the HTTP response code is an error, emit an error message rather than the 404 page.
- `-L`: If there's an HTTP 301 redirect given, try to follow it.
- `-sS`: Don't show progress meter (`-S` stops `-s` from also blocking error messages).
- `-o /dev/null`: Send the standard output (the actual page returned) to `/dev/null`.

This way, the `curl(1)` request should stay silent if everything is well, per the old Unix philosophy .

You may not agree with some of the choices above; you might think it important to e.g. log the complete output of the returned page, or to fail rather than silently accept a 301 redirect, or you might prefer to use `wget(1)`. The point is that you take the time to understand in more depth what the called program will actually emit under what circumstances, and make it match your requirements as closely as possible, rather than blindly discarding all the output and (worse) the errors. Work with ; assume that anything that can go wrong eventually will.

Another common mistake is failing to set a useful `MAILTO` at the top of the `crontab(5)` file, as the specified destination for any output and errors from the tasks. `cron(8)` uses the system mail implementation to send its messages, and typically, default configurations for mail agents will simply send the message to an `mbox` file in `/var/mail/$USER`, that they may not ever read. This defeats much of the point of mailing output and errors.

This is easily dealt with, though; ensure that you can send a message to an address you actually *do* check from the server, perhaps using `mail(1)`:

```
$ printf '%s\n' 'Test message' | mail -s 'Test subject' you@example.com
```

Once you've verified that your mail agent is correctly configured and that the mail arrives in your inbox, set the address in a `MAILTO` variable at the top of your file:

```
MAILTO=you@example.com

0 * * * *    you  cron-task-1
*/5 * * * *  you  cron-task-2
```

If you don't want to use email for routine output, another method that works is sending the output to `syslog` with a tool like `logger(1)`:

```
0 * * * *   you  cron-task | logger -it cron-task
```

Alternatively, you can configure aliases on your system to forward system mail destined for you on to an address you check. For Postfix, you'd use an `aliases(5)` file.

I sometimes use this setup in cases where the task is expected to emit a few lines of output which might be useful for later review, but send `stderr` output via `MAILTO` as normal. If you'd rather not use `syslog`, perhaps because the output is high in volume and/or frequency, you can always set up a log file `/var/log/cron-task.log` … but don't forget to add a `logrotate(8)` rule for it!

Ideally, the commands in your `crontab(5)` definitions should only be a few words, in one or two commands. If the command is running off the screen, it's likely too long to be in the `crontab(5)` file, and you should instead put it into its own script. This is a particularly good idea if you want to reliably use features of `bash` or some other shell besides POSIX/Bourne `/bin/sh` for your commands, or even a scripting language like Awk or Perl; by default, `cron(8)` uses the system's `/bin/sh` implementation for parsing the commands.

Because `crontab(5)` files don't allow multi-line commands, and have other gotchas like the need to escape percent signs `%` with backslashes, keeping as much configuration out of the actual `crontab(5)` file as you can is generally a good idea.

If you're running `cron(8)` tasks as a non-system user, and can't add scripts into a system bindir like `/usr/local/bin`, a tidy method is to start your own, and include a reference to it as part of your `PATH`. I favour `~/.local/bin`, and have seen references to `~/bin` as well. Save the script in `~/.local/bin/cron-task`, make it executable with `chmod +x`, and include the directory in the `PATH` environment definition at the top of the file:

```
PATH=/home/you/.local/bin:/usr/local/bin:/usr/bin:/bin
MAILTO=you@example.com

0 * * * *  you  cron-task
```

Having your own directory with custom scripts for your own purposes has a host of other benefits, but that's another article…

If your implementation of `cron(8)` supports it, rather than having an `/etc/crontab` file a mile long, you can put tasks into separate files in `/etc/cron.d`:

```
$ ls /etc/cron.d
system-a
system-b
raid-maint
```

This approach allows you to group the configuration files meaningfully, so that you and other administrators can find the appropriate tasks more easily; it also allows you to make some files editable by some users and not others, and reduces the chance of edit conflicts. Using `sudoedit(8)` helps here too. Another advantage is that it works better with version control; if I start collecting more than a few of these task files or to update them more often than every few months, I start a Git repository to track them:

```
$ cd /etc/cron.d
$ sudo git init
$ sudo git add --all
$ sudo git commit -m "First commit"
```

If you're editing a `crontab(5)` file for tasks related only to the individual user, use the `crontab(1)` tool; you can edit your own `crontab(5)` by typing `crontab -e`, which will open your `$EDITOR` to edit a temporary file that will be installed on exit. This will save the files into a dedicated directory, which on my system is `/var/spool/cron/crontabs`.

On the systems maintained by the author, it's quite normal for `/etc/crontab` never to change from its packaged template.

`cron(8)` will normally allow a task to run indefinitely, so if this is not desirable, you should consider either using options of the program you're calling to implement a timeout, or including one in the script. If there's no option for the command itself, the `timeout(1)` command wrapper in `coreutils` is one possible way of implementing this:

```
0 * * * *  you  timeout 10s cron-task
```

Greg's wiki has some further suggestions on .

`cron(8)` will start a new process regardless of whether its previous runs have completed, so if you wish to avoid locking for long-running task, on GNU/Linux you could use the `flock(1)` wrapper for the `flock(2)` system call to set an exclusive lockfile, in order to prevent the task from running more than one instance in parallel.

```
0 * * * *  you  flock -nx /var/lock/cron-task cron-task
```

Greg's wiki has some more in-depth discussion of the problem for scripts in a general sense, including important information about the caveats of "rolling your own" when `flock(1)` is not available.

If it's important that your tasks run in a certain order, consider whether it's necessary to have them in separate tasks at all; it may be easier to guarantee they're run sequentially by collecting them in a single shell script.

If your `cron(8)` task or commands within its script exit non-zero, it can be useful to run commands that handle the failure appropriately, including cleanup of appropriate resources, and sending information to monitoring tools about the current status of the job. If you're using Nagios Core or one of its derivatives, you could consider using `send_nsca` to send passive checks reporting the status of jobs to your monitoring server. I've written `nscaw` to do this for me:

```
0 * * * *  you  nscaw CRON_TASK -- cron-task
```

`cron(8)`

If your machine isn't always on and your task doesn't need to run at a specific time, but rather needs to run once daily or weekly, you can install `anacron` and drop scripts into the `cron.hourly`, `cron.daily`, `cron.monthly`, and `cron.weekly` directories in `/etc`, as appropriate. Note that on Debian and Ubuntu GNU/Linux systems, the default `/etc/crontab` contains hooks that run these, but they run only if `anacron(8)` is not installed.

If you're using `cron(8)` to poll a directory for changes and run a script if there are such changes, on GNU/Linux you could consider using a daemon based on `inotifywait(1)` instead.

Finally, if you require more advanced control over when and how your task runs than `cron(8)` can provide, you could perhaps consider writing a daemon to run on the server consistently and fork processes for its task. This would allow running a task more often than once a minute, as an example. Don't get too bogged down into thinking that `cron(8)` is your only option for any kind of asynchronous task management!

Posted in | Tagged , , , , , , , ,

Shell config subfiles

Posted on

Large shell startup scripts (`.bashrc`, `.profile`) over about fifty lines or so with a lot of options, aliases, custom functions, and similar tweaks can get cumbersome to manage over time, and if you keep your dotfiles under version control it's not terribly helpful to see large sets of commits just editing the one file when it could be more instructive if broken up into files by section.

Given that shell configuration is just shell code, we can apply the `source` builtin (or the `.` builtin for POSIX `sh`) to load several files at the end of a `.bashrc`, for example:

```
source ~/.bashrc.options
source ~/.bashrc.aliases
source ~/.bashrc.functions
```

This is a better approach, but it still binds us into using those filenames; we still have to edit the `~/.bashrc` file if we want to rename them, or remove them, or add new ones.

Fortunately, UNIX-like systems have a common convention for this, the `.d` directory suffix, in which sections of configuration can be stored to be read by a main configuration file dynamically. In our case, we can create a new directory `~/.bashrc.d`:

```
$ ls ~/.bashrc.d
options.bash
```

```
aliases.bash
functions.bash
```

With a slightly more advanced snippet at the end of `~/.bashrc`, we can then load every file with the suffix `.bash` in this directory:

```
# Load any supplementary scripts
for config in "$HOME"/.bashrc.d/*.bash ; do
    source "$config"
done
unset -v config
```

Note that we unset the `config` variable after we're done, otherwise it'll be in the namespace of our shell where we don't need it. You may also wish to check for the existence of the `~/.bashrc.d` directory, check there's at least one matching file inside it, or check that the file is readable before attempting to source it, depending on your preference.

The same method can be applied with `.profile` to load all scripts with the suffix `.sh` in `~/.profile.d`, if we want to write in POSIX `sh`, with some slightly different syntax:

```
# Load any supplementary scripts
for config in "$HOME"/.profile.d/*.sh ; do
    . "$config"
done
unset -v config
```

Another advantage of this method is that if you have your dotfiles under version control, you can arrange to add extra snippets on a per-machine basis unversioned, without having to update your `.bashrc` file.

Here's my implementation of the above method, for both `.bashrc` and `.profile`:

- `.bashrc`
- `.bashrc.d`
- `.profile`

- `.profile.d`

Thanks to commenter oylenshpeegul for correcting the syntax of the loops.

Posted in | Tagged , , , , , , ,
