Documentation Project

Jeevan Shah

Rutgers

November 12, 2025

Changing the Default Shell (on MacOS)

Lets say you just started working on a new machine and you want to ensure that the current shell is bash. We can check the current shell and switch it if it isn't our preferred shell in a few simple steps:

1. Run `ps -p $$` to check the current shell - `ps` stands for 'process supply' and is a command we can use to check which processes are running in our terminal. The `-p` flag allows us to specify a specific process, so we pass the argument `$$` which is a variable that holds the process id (PID) of the current shell.
2. If the current shell is our desired shell then we are done! If it isn't we can change the shell using the `chsh` command.
    1. Start by running `cat /etc/shells` to view the available shells on the machine.
    2. Next run `chsh -s $1` where `$1` is the file path of your choosen shell from the previous command (the `-s` flag simply allows us to change the shell as the `chsh` command can update a variety of information about our user).

Changing the Default Shell (Example)

```
❯ ps -p $$
  PID TTY           TIME CMD
 8019 ttys000    0:03.05 -zsh
❯ cat /etc/shells
# List of acceptable shells for chpass(1).
# Ftpd will not allow users to connect who are not using
# one of these shells.

/bin/bash
/bin/csh
/bin/dash
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh
❯ chsh -s /bin/bash
Changing shell for jeevanshah.
Password for jeevanshah:
```

As you can see this process may need administrator abilites!

An Overview of `.bashrc` and `.bash_profile`

What are they?

`.bashrc` and `.bash_profile` are files found in the home directory which allow you to control the behavior of your shell.

Whats the difference?

- `.bashrc` is called every time the shell is *initalized* (for example, each time you open up a new terminal window). This is the best place to put things like aliases and functions to guarentee that they are loaded each time you launch a new shell.
- `.bash_profile` is called every time you *login* to the shell (for example, using SSH to remotely access the shell). This is the best place to store environment variables to ensure they are available to all programs, not just the interactive shell.

Editing .bashrc and .bash_profile

These days, almost all systems come installed with `vi` which is a text editor that works soley in the terminal, so we can use it to edit our files.

Using `vi`

Vi is intended to be completely accessable from the keyboard. The default state of vi is **normal mode** which allows us to move with h (left), j (up), k (down), and l (right). When you want to start editing, hit 'i' to enter **insert mode**. You can exit insert mode and return to normal mode with the escape key.

In order to edit `.bashrc`, enter '`vi ~/.bashrc`'. We can then enter insert mode with 'i' and add the line '`alias ll = ls -la`', hit escape to return to normal mode, and save and quit from vi with '`:wqa`'. Now, we can reload our shell (either by opening a new terminal or running '`source ~/.bashrc`') and check that everything is working correctly by running '`which ll`'. If you did everything right you should see '`ll:  aliased to ls -la`'

The PATH variable

When working in a shell environment certain pieces of information are stored in environment variables which hold information pertaining to the current environment, and if you store them in `.bash_profile` (or `.bashrc`) then they will persist from session to session. When working in the shell you can access your environment variables using a dollar sign followed by the name of your variable. One of, if not the, most important enviornment variables is the `PATH` variable which contains a set of directories that tells your shell where to look for executables. To view your `PATH`, run '`echo $PATH`'.

How to Edit the `PATH` variable

There often come times in which you need to edit your `PATH` variable, usually to add an additional directory to it. Say you want to add the directory '`~/.program/bin`' to your path. We can easily do this by opening up our `.bash_profile` or `.bashrc` and adding the following line to the bottom: '`export PATH = "$HOME/.program/bin:$PATH"`'.

The tar Command

The tar command is used to manipulate (compressed) archive files. You can use tar to both extract and archive/compress files.
Lets say you have some files in the form 'files.tar.gz' and you want to use the tar command to analyze/unpack them.

- `tar -tvf files.tar.gz`: prints file names, permissions, owners etc of the contents of the archive.
- `tar -xzvf files.tar.gz`: unpacks the **entire** archive into the *current directory* (add the '-p' flag to ensure permissions and preserved)
- `tar -xzvf files.tar.gz folder1/file1`: unpacks **only** `folder1/file1` from `files.zip` into the *current directory*
- `tar -xzvf files.tar.gz -C /myFiles/extracted`: unpacks the **entire** archive into the folder `/myFiles/extracted` (the provided directory must exist beforehand)
- `tar --exclude='folder1/file1.txt' -czvf files.tar.gz -C /tmp .`: Recreates `files.tar.gz` **without** `folder1/file1.txt`

The tar Command (cont.)

Now, lets say you have the following files: `file1.txt`, `file2.txt`, and `folder1/file3.txt`. If you want to archive and/or compress them here are some important commands you might use:

- `tar -czvf files.tar.gz file1.txt file2.txt folder1/file3.txt`: Creates a **compressed** archive ('`-z`' indicated gzip compression) with the provided files.
- `tar -czvf files.tar.gz file1.txt file2.txt folder1/file3.txt file4.txt`: Adds `file4.txt` to `files.tar.gz` (by uncompressing and then recompressing - which is why you must specify the files already in `files.tar.gz`)

`.tar` vs `.tar.gz`

You may see that some files end in '`.tar`' rather than '`.tar.gz`'. All this mean is that the '`.tar`' file is **uncompressed** and rather just an archive of files. If you're working with files that end in '`.tar`' simply just omit the '`-z`' flag from your `tar` commands (and use '`files.tar`' instead).

File Permissions

Earlier I mentioned that using the '`-p`' flag with `tar` would preserve permissions. This is important because every file/directory has read (r), write (w), and execute (x) permissions specified for the user, the group, and the system (sometimes referred to as 'other'). Using a command like '`ls -la`' will show the contents of the current directory along with the permissions of each file. When looking at this you can see a string that looks like '`rwxrw-r--`'.

Understanding Permissions

- The next three characters indicate the permissions for the **user** (who owns the file)
- The following three characters indicate the permissions for the **group** (that the owner belongs too)
- The final three characters indicate the permissions for any user **not** apart of the group.

File Permissions (cont.)

Sometimes you need to change the permissions of a file or directory. We do this using the 'chmod' command. The chmod command works in the following way: 'chmod [0-7][0-7][0-7] file'. There are three numbers passed for each set of permissions.

Numerical Permissions

When changing permissions it's important to understand that read=4, write=2, and execute=1. Every combination of adding these three numbers corresponds to a different permissions.

- 0 would be '---'
- 1 would be '--x'
- 2 would be '-w-'
- $3 = 1 + 2$ would be '-wx'
- 4 would be 'r--'
- $5 = 4 + 1$ would be 'r-x'
- $6 = 4 + 2$ would be 'rw-'
- $7 = 4 + 2 + 1$ would be 'rwx'

Thus, `chmod 764 file` would result in `file` having the following permissions: 'rwxrw-r--'

File Permissions (cont.)

Theres also another way to use the `chmod` command. Rather than dealing with numbers we can also change the permissions using letters. In this case the syntax is: `chmod (u,g,o)±(r,w,x)`.

1. First, we choose either 'u', 'g', or 'o' to change the permissions for the user, group, or other.
2. Then we use either '+' or '−' to add or remove permissions.
3. Finally, we determine which permissions to change by selecting from 'r', 'w', or 'x'.

For example '`chmod u+x file`' would add execute permissions to our user for `file`. We can also adjust multiple permissions as the same time: '`chmod o+r-wx,u+rwx`' would add read permissions and remove write and execute permissions from other while adding read, write, and execute permissions to the user.

File Permissions (cont.)

There is one final thing to discuss, the **sticky bit**. The sticky bit a permission that *only directories have* and it tells us who is allowed to rename and delete files in a certain folder. The sticky bit is mainly used in folders that are accessable to all users on the system, such as `/tmp`. We can set the sticky bit using a similar `chmod` syntax:

- `chmod +t /var/share` (or `chmod 1764 /var/share` if the previous permissions were 'rwxrw-r–'): turns on the sticky bit in `/var/share` meaning that only a file's owner (or administrator) can delete or rename it.
- `chmod -t /var/share` (or `chmod 0764` if the previous permissions were '`rwxrw-r--`'): turns off the sticky bit

Hard vs Soft Links

Hard Links

Hard links are links that point directly to memory on the hard drive. This means that if `file1` is hard linked to `file2` and `file2` is deleted, `file1` will survive because they both point to the same memory on the hardd rive. Thus, editing `file1` will change `file2` and vice versa. Hard links are best used when you need multiple references to the same file such as system config files. Hard links **cannot** point to directories. You can create a hard link using the command '`ln file1.txt file2.txt`'.

Soft Links

Soft links, or symbolic links, are links that point to a different pathname, **including directories**. If `file1` is soft linked to `file2` and `file2` is deleted, the soft link becomes 'broken' and doesn't function anymore. Soft links are best used to point to directories or across file systems. You can create a soft link using the command '`ln -s file1.txt file2.txt`'.

UNIX File Structure

The UNIX file strcuture is a (mostly) standarized structure found across all types of UNIX machines. It organizes all data into *files* and all files into *directories*. All data lives in the root directory called `/`. In the root directory there are many important folders such as

- `/usr` - stores executables, libraries, and other shared resources deemed to be not critical to the system (such as your window manager/desktop environment)
- `/etc` - holds all system wide config files and databases (recall that the list of available shells was stored in `/etc/shells`)
- `/home` - holds all the home directories for each user
- `/tmp` - a location to store all temporary files as `/tmp` is usually cleared upon each shutdown
- `/var` - home to files that often change, usually in size (things like logs or caches)
- `/bin` - contains built-in binaries accessable to all users (things like `ls` or `mv`)