Intermediate's Linux Tutorial

Damien Levac

September 25, 2017

Outline

1 UNIX Permissions

Nobody can hurt me without my permission.

(*Mahatma Gandhi*)

1.1 stat

Display file or file system status.

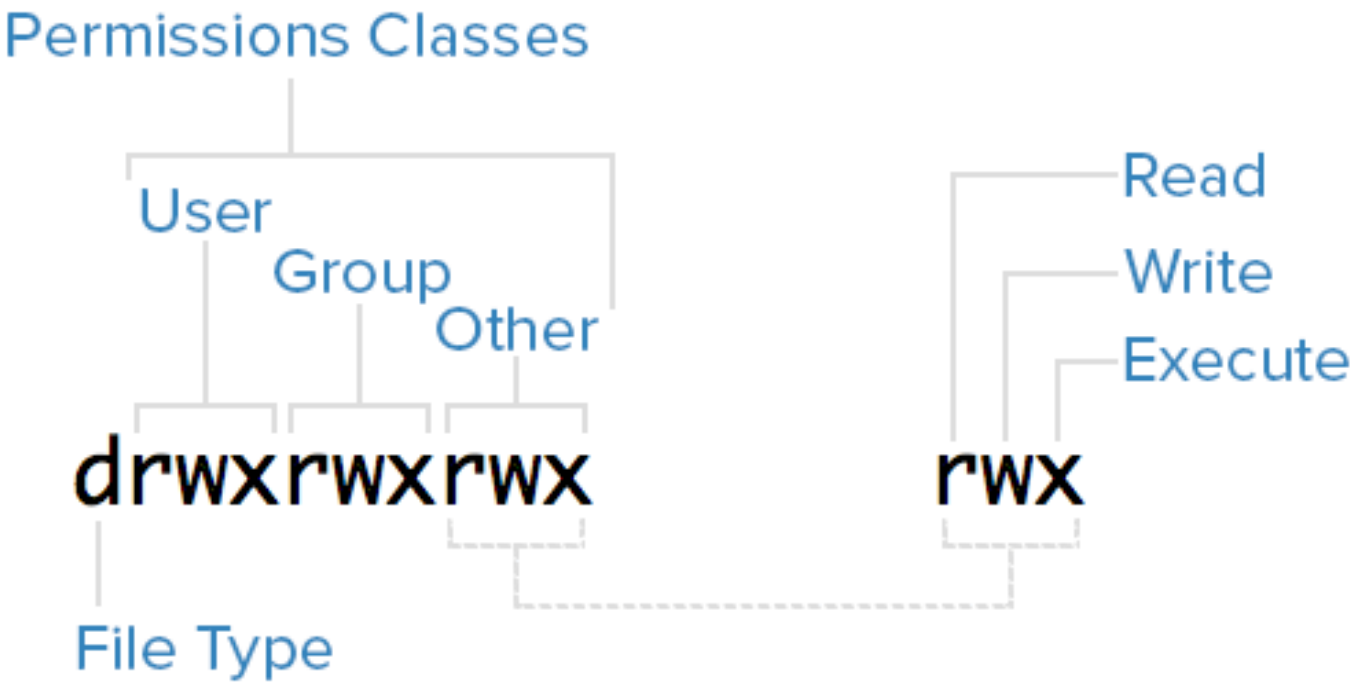
```
$ stat ~
>   File: '/home/2011/dlevac'
>   Size: 72  Blocks: 129  IO Block: 32768  directory
> Device: 5eh/94d Inode: 271823  Links: 42
> Access: (0700/drwx------)  Uid: (21413/ dlevac)
>                            Gid: (65534/ nogroup)
> Access: 2016-04-04 15:48:19.000000000 -0400
> Modify: 2017-09-12 10:32:55.034780000 -0400
> Change: 2017-09-12 10:32:55.034780000 -0400
>  Birth: -
```

1.2 File Permissions

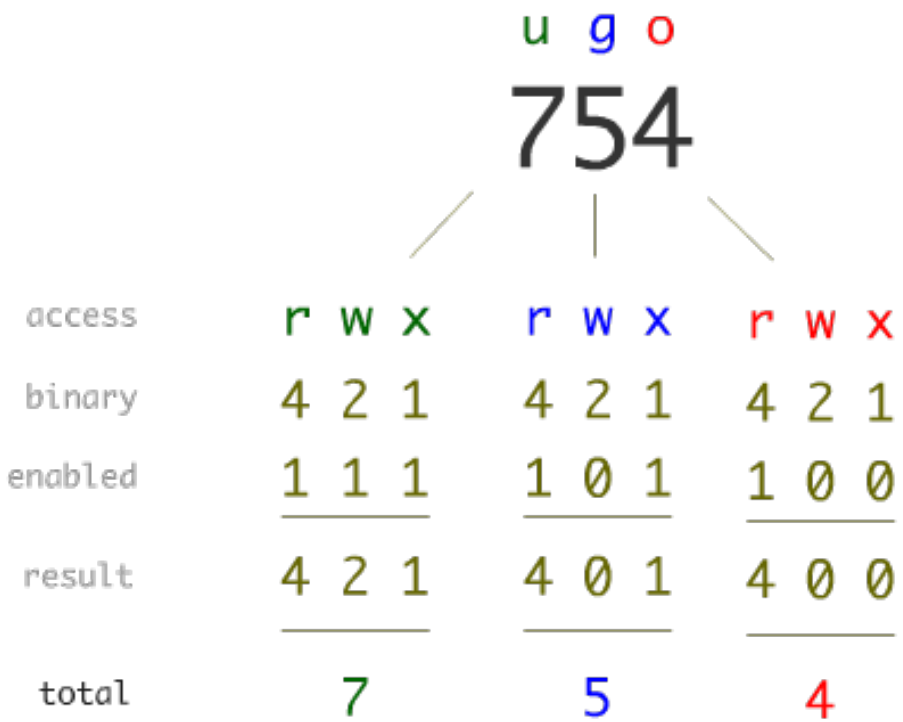
In a distributed system such as the one you are using on campus, it is important keep a close look on the permissions you yield to other users.

Unless a more elaborate security system is used (e.g. selinux), every file can be *readable*, *writable* or *executable* by its *owner*, *group* or *everyone else*.

For legacy reasons and, to some extend, because it is convenient too; permissions are displayed in 2 different format: with abbreviations and as an octal code.



Source: `https://assets.digitalocean.com/articles/linux_basics/mode.png`



Source: `https://danielmiessler.com/images/permissions.png`

1.3 `chmod` (CHange MODe)

Change file mode bits.

```
$ chmod 711 myfile
$ stat -c %A myfile
> drwx--x--x
$ chmod a+r myfile
$ stat -c %A myfile
> drwxr-xr-x
```

The `chmod` command can set the octal mode directly or use the following syntax: `chmod [augo][+-][rwx] <FILENAME>` where we basically specify which of *All*, *User (owner)*, *Group* or *Other* should *add (+)* or *remove (-) Read*, *Write* or *Execute* permission.

1.4 File Ownership

Every file belongs to a user and to a group. Only *root* or administrators can change the user ownership of a file. Group ownership and permissions can be changed by either *root* or its *owner*.

While the concept of ownership and permissions might seem useful only from the point of view of sharing files between users, it is of paramount important when running processes which typically runs with the permission of the user who executed it. While this is slightly outside the scope of this tutorial you might want to fine-tune the permissions of a program you do not trust to avoid it causing damage to your system.

1.5 `chgrp` (CHange GRouP)

Change group ownership.

```
$ chgrp mygroup myfile
$ stat -c %G myfile
> mygroup
```

Note that the group needs to exists for a user to change group ownership to that group. Creating groups unfortunately requires admin privilege.

1.6 `chown` (CHange OWNer)

Change file owner and group.

```
$ chown dlevac myfile
$ stat -c %U myfile
> dlevac
$ chown root:mygroup myfile
$ stat -c '%U %G'
> root mygroup
```

Note that only an administrator change ownership for a file.

2 Environment Variables

> Computing is kind of a mess. Your computer doesn't know where you are. It doesn't know what you're doing. It doesn't know what you know.
>
> (*Larry Page*)

2.1 env

Print the environment or run a program in a modified environment.

```
$ env
> XDG_VTNR=7
> SSH_AGENT_PID=3637
> XDG_SESSION_ID=c4
> XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/dlevac
> CLUTTER_IM_MODULE=xim
> SHELL=/bin/bash
> [...]
```

2.2 Processes

Each processes, including your shell, has its own separate set of *environment variables*. By default, a process inherit the environment of its parent. Thus, if you invoke the command `echo`, it will run with the environment of your shell.

You can access the content of an *environment variable* by prefixing its name by a dollar sign (`$`).

```
$ echo $USER $HOME $SHELL
> dlevac /home/2011/dlevac /bin/bash
$ echo $PWD
> /home/2011/dlevac
```

2.3 A Note on Security

A process is unable to modify the *environment variables* of its parent. This is why the command `cd` is actually a built in shell utility and not an actual independant program. That being said, you could always bypass `cd` completely by setting the `PWD` *environment variable* yourself.

This avoid a malicious program trying to inject harmful *environment variables* into your environment. For example, setting `HOME` to `/tmp` could trick the user into saving confidential files in a place everyone can read freely.

2.4 Setting Environment Variables

The syntax to set an *environment variable* in your current shell is pretty straightforward:

```
$ A=1
$ echo $A
> 1
```

To set the *environment variable* for a single command, just write the assignment on the same line as the command as follow:

```
$ DEBUG=1 ./my_program
```

Be wary of a common pitfall however:

```
$ A=1 echo $A
>
```

The issue here is that `$A` is expanded before the assignment `A=1` occurs. You could get around this limitation by setting temporarily `A` for the whole shell the time of the command and unsetting it afterward:

```
$ A=1; echo $A; unset A
> 1
$ echo $A
>
```

Note that ; delimits commands the same way whitespaces does.

2.5 Exporting Environment Variables

Setting *environment variable* as we did above do not actually modify the output of the `env` command: it only made the variable visible in the current shell.

To modify the output of the `env` command and make your *environment variable* visible by every child processes, you need to export it as follow:

```
$ export A=1
$ env
> A=1
> [...]
```

2.6 PATH

PATH is a very important *environment variable* that looks like this:

```
$ echo $PATH
> /bin:/var/bin:/usr/local/bin:/usr/bin
```

This is a colon separated list of directories to look for commands. Thus, when trying to run command `eclipse`, it will first look in `/bin`, then `/var/bin`; since `eclipse` exists in `/var/bin` the search stops and that version of eclipse is executed.

2.7 Modifying PATH

Lets say you write your own scripts and programs and you want to be able to simply type the name of program for it to be run by the shell; without specifying the full path each time.

One popular way to do it would be to place your scripts in, for example, `~/bin` directory and the modifying your `PATH` as follow:

```
$ export PATH=$HOME/bin:$PATH # or export
                              # PATH=$PATH:$HOME/bin
                              # to be searched last
                              # instead
```

Note that # mark the start of a comment extending until end of line.

2.8 `~/.bashrc` & `~/.profile`

While scripting itself is outside the scope of this tutorial, it might be tedious to set your environment everytime you login. Fortunately, there are some files whose commands written into get executed everytime you login:

.bashrc executed everytime `/bin/bash` is started

.profile (or .bash_profile) executed when you login (by the login shell)

So you could write export `export PATH=$HOME/bin:$PATH` at the end of `~/.bashrc` to have your `PATH` set everytime you open a shell.

3 Command Chaining

> Before software can be reusable it first has to be usable.
>
> (*Ralph Johnson*)

3.1 I/O Streams

In UNIX, the abstraction for any concept is a *file*. 3 of these files allow easy interaction between user and processes:

- `/dev/stdout`: default output stream for program who wish to print information for users.
- `/dev/stderr`: default output stream for program who with to report errors for users.
- `/dev/stdin`: default input stream for program who listen for user's input.

3.2 Output Redirection

We have seen in the previous tutorial the command `echo` which we said "print back its arguments on screen". A better description would be "Write its arguments to *stdout*".

We will be able to convince ourselves of this by introducing *output redirection*. It allows us to redirect the outputs of a program to any file we want. So, to redirect the output of our `echo` command to a file at `\~/foo`, we could write:

```
$ echo Hello, World! >> ~/foo # redirect stdout to ~/foo
$ echo Hello, World! 2>> ~/foo # redirect stderr to ~/foo
```

Using a single > means *overwrite* the file, instead of *appending*.

```
$ cat ~/foo
> Hello, World!
```

Note that `1` and `2` refer to the *file descriptors* for `stdout` and `stderr` respectively. Note that you can use the following syntax: `2>&1` to redirect `stderr` to `stdout` for example.

This shouldn't surprise you, but:

```
$ echo Hello, World! >> /dev/stdout
> Hello, World!
```

You might also want to silence the output of a command:

```
$ echo Hello, World! 2>&1 >> /dev/null
```

3.3 Exit Codes

Apart from the output a program write to files, there is another bit of information that every program return to the calling shell once it finishes: an *exit code*. They range from `0` to `255` where `0` means no error and any other value typically is an error code.

The exit code of a process is stored in the special *environment variable* `?`. For example

```
$ which true; which false
> /bin/true
> /bin/false
$ true; echo -n "$? "; false; echo $?
> 0 1
```

3.4 Logical Operators

In any UNIX shell, an exit code of `0` is analogous to `true` and any other exit code to `false`. We can use these facility to conditionally run commands:

```
$ true && echo test
> test
$ false && echo test
$ true || echo test
$ false || echo test
> test
```

Where `&&` is to be understood as *and* and `||` is *or*.

3.5 Pipes

Creating a *pipe* between 2 processes means redirecting what the first process writes to `stdout` to the `stdin` of the second process. A *pipe* is represented by a `|` symbol.

Let us revisit the `cat` program with this updated definition: "`cat` concatenates the content of every file given as argument, using `stdin` if no argument is provided, and writes the resulting data to `stdout`.

```
$ echo Hello, World! | cat
> Hello, World!
```

4 Intermediate Commands & Examples

> If you have any trouble sounding condescending, find a UNIX user to show you how it's done.
>
> (*Scott Adams*)

4.1 `seq`

Print a sequence of numbers using the syntax `seq <FIRST> [INCREMENT] <LAST>`.

```
$ seq 1 3
> 1
> 2
> 3
```

Useful options:

- `-s (--separator)` specify delimitor to use between numbers (newline by default)

4.2 `grep`

Print lines matching a pattern.

```
$ seq 1 10 > foo && grep 3 foo
> 3
$ seq 1 10 | grep 3
> 3
```

Useful options:

- `-r (--recursive)` search all directory provided recursively
- `-i (--ignore-case)` case insensitive line matching

Most commands that operates on files will default to `stdin`.

4.3 `sort`

Sort lines of text files and write sorted concatenation of all files to standard output.

```
$ seq 1 2 > foo && seq 1 2 >> foo && sort foo
> 1
> 1
> 2
> 2
```

Useful options:

- `-n (--numeric-sort)` compare according to string numerical value

4.4 `uniq`

Report or omit adjacent repeated lines.

```
$ seq 1 3 > foo && seq 1 3 >> foo && sort foo | uniq
> 1
> 2
> 3
```

Useful options:

- `-c (--count)` prefix lines by the number of occurrences

4.5 `awk`

Pattern scanning and processing language.

We will only cover the use case of selecting a column in a file using the syntax: `awk '{ print $<N> }' <FILES>` where `<N>` is the column number.

```
$ seq -s ' ' 1 10 | awk '{ print $3 }'
> 3
```

`awk` is actually a full-blown language that would take an entire tutorial just to cover its basics.

4.6 `head`

Output the first part of files (10 lines by default).

```
$ seq 1 100 | head -n 3
> 1
> 2
> 3
```

Useful options:

- `-n (--lines)` print only the specified number of lines or omit only the specified number of last lines by prefixing the number with -

4.7 `tail`

Output the last part of files (10 lines by default).

```
$ seq 1 100 | tail -n 3
> 98
> 99
> 100
```

Useful options:

- `-n (--lines)` print only the specified number of lines or omit only the specified number of first lines by prefixing the number with +

5 Process Management

> There is no neat distinction between operating system software and the software that runs on top of it.
>
> (*Jim Allchin*)

5.1 Is GNU/Linux Stable?

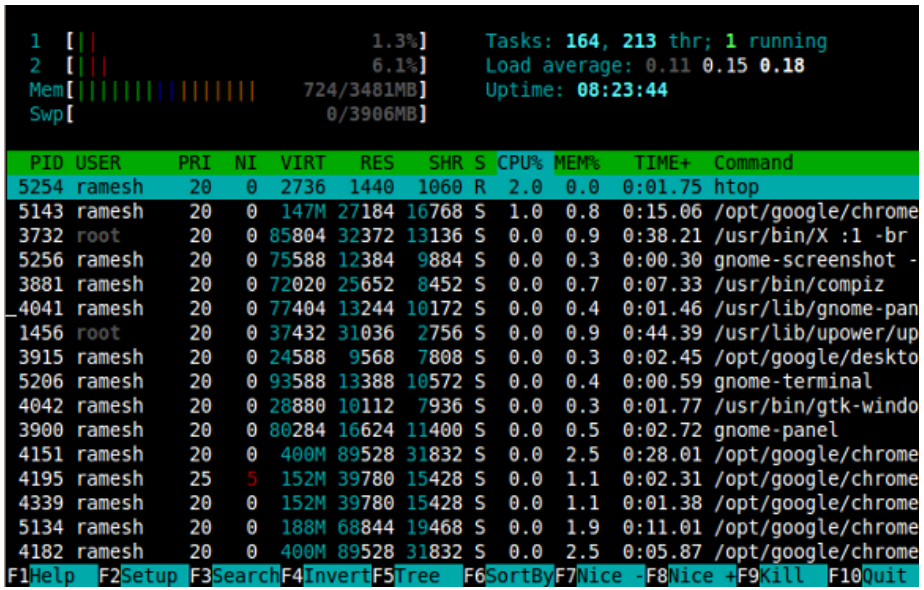
The stability of your UNIX operating system is directly corrolated to the packages and services you run on it. While the Linux kernel and the GNU toolset is known to be rock solid (next time your machine "freeze" login from a console and see if it is truly frozen!)

You can improuve the overall stability of the system by omitting or using simpler and more lightweight solutions for the following:

- Init system
- Desktop environment
- Sound system
- Graphics card driver

5.2 `htop` & `top`

`htop` is an interactive process viewer while `top` is non-interactive.



Source: `http://static.thegeekstuff.com/wp-content/uploads/2011/09/01-htop-output.png`

`htop` is unfortunately not very widespread and not part of any GNU/Linux distribution by default as far as I know. However it is an immensely useful tools that gives an overview of the system usage and allow to easily sort processes and send signals to them.

If you just need an overview of the processes running, `top` is the legacy package which is available on almost all UNIX-like operating systems.

5.3 `ps`

Report a snapshot of the current processes.

```
$ ps
>   PID TTY          TIME CMD
>  9048 pts/4    00:00:00 bash
> 18697 pts/4    00:00:00 ps
```

Useful options:

- `-e` select all processes
- `-f` full-format listing
- `-F` extra full-format listing (implies `-f`)

5.4 `kill` & `pkill`

`kill` send signals to PIDs while `pkill` send signals to process based on name or other attributes.

Report a snapshot of the current processes.

```
$ kill 1234
$ pkill firefox || pkill -s 9 firefox
```

Useful options:

- `-l` list signals
- `-s` specify the signal to send

Questions?