VERMONT TECH

# Command Line

CIS 2230 Linux System Administration

Lecture 5

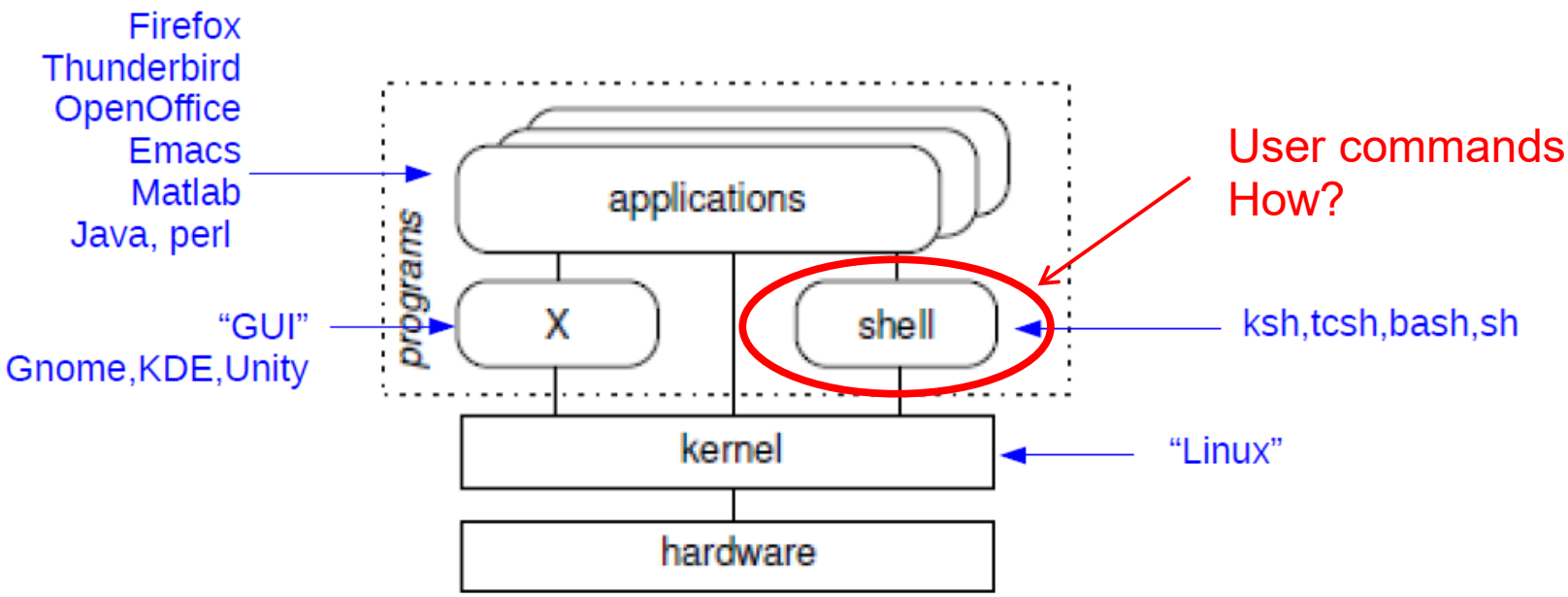
Steve Ruegsegger

## Review

- What is the primary command for help on a command?
- What do these do? `apropos, info, whereis, whatis, which`

## Shells (review)

- A shell provides an interface between the user and the operating system kernel
- A shell accepts your instruction or commands in “English” (mostly) and passes it to kernel.
- It's a program which aids the user in communicating with the kernel – the kernel talks in binary/assembly and you don't.
- Elements of a shell:
  - programming language, env. variables, history, cursor editing, path display, etc.



## Shells

- There are many different user shells
- The original shell is `/bin/sh`
- Ubuntu and Fedora uses `bash` as the default. AIX uses `ksh`.
- Each has a programming/scripting language. You need to know it!
- Each has it's own +'s and -'s and unique idiosyncrasies

| Shell Name | Developed by | Where | Remark |
|---|---|---|---|
| BASH ( Bourne-Again SHell ) | Brian Fox and Chet Ramey | Free Software Foundation | Most common shell in Linux. It's Freeware shell. |
| CSH (C SHell) | Bill Joy | University of California (For BSD) | The C shell's syntax and usage are very similar to the C programming language. |
| KSH (Korn SHell) | David Korn | AT & T Bell Labs | -- |
| TCSH | See the man page. Type $ man tcsh | -- | TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH). |

`sh` – 'bare-bones', very basic, used for simple scripting

## How do I communicate with the shell?

ORDER is important

- Shell commands have 3 "parts":
  - 1) a command, 2) options, 3) arguments
  - Separated by spaces
  - Format: `$ command <options> <arguments>`
  - Examples?

1. Command
   - The first word is the command to run
   - Most commands are a small binary program in a specific directory
   - *Few* commands are built into the shell itself – called "built-ins"
2. Options
   - Start with dash (-) or double dash (--)
3. Arguments
   - Limited or unlimited; optional or required
   - Extra information from the user for the program to use

## Examples of Command-Line Options

- Default – list files in current directory

```
$ ls
```

- With option – list the files in the 'long format':

```
$ ls -l
```

- Option and args – List full information about some specific files:

```
$ ls -l notes.txt report.txt
$ command <option> [list of arguments]
```

- Use a wildcard – List full information about all the .txt files:

```
$ ls -l *.txt
```

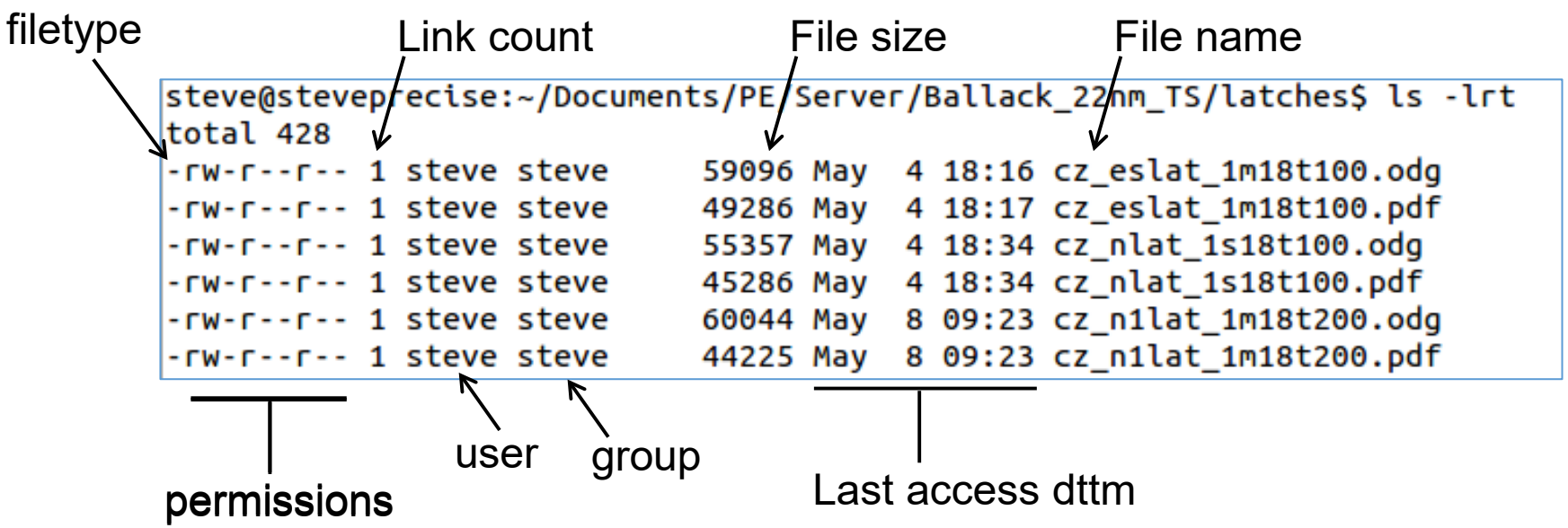
- Multiple options – List all files in long format, even the hidden ones:

```
$ ls -l -a
$ ls -la
```

## Key ls options

There are many options listed in `$ man ls`. Here are some key ones you should know.

```
$ ls
$ ls -a
$ ls -l
$ ls -la
$ ls -lrt
$ ls -lrS
$ ls -d <dir>
$ ls -F
```



## Calling a command

- Explicitly – giving the *exact path* (absolute or relative)
  - What do these do?

```
$ /bin/ls
$ /home/steve/cleanup.pl
$ ./backup_files.sh
$ ../../bob/scripts/runme.sh
```

- Implicitly – *no path*, letting the shell "find" the command

```
$ firefox
$ calculator
```

  - Sounds 'scary'
  - *How could this be 'bad'?*

## Which command to run implicitly?

- If not *explicit* (i.e. *implicit* or no path), the shell must "looks for" the *proper* program
  - *"firefox, you say; I wonder which one?"*
- The environment variable `$PATH` lists the directories in which to search
- Directory names are separated by colon, for example:

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

- *e.g.* Running `$ whoami`
  - will look for `/bin/whoami` or `/usr/bin/whoami` or `/usr/local/bin/whoami`
  - In the order the directories exist in the path
  - `$ which whoami` tells you "which one"

## Security thoughts about $PATH

- How important is $PATH?
- Why is it not good to have "." (cwd) in the path?
  - BAD --> `$PATH = .:/bin:/usr/bin: ...`
  - e.g. 'bad guy' `~/bin/ls`

- *Note*: Ubuntu changed default `$PATH` around 14.04.
- I *don't like* this change.

```
steve@xerus:~/bin$ echo $PATH
/home/steve/bin:/home/steve/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
steve@xerus:~/bin$
```

Ack! A bit of a security risk.

(hint: $ hash –r )

## Running a script not in $PATH

- Therefore, if "." is not in `$PATH`, then you *must* run commands "explicitly" with "./"

```
$ ./myscript.pl
```

## Bash has a speedier builtin: hash

- Not a file, a bash shell *builtin*

> `hash` is a bash built-in command. The hash table is a feature of `bash` that prevents it from having to search `$PATH` every time you type a command by caching the results in memory. The table gets cleared on events that obviously invalidate the results (such as modifying `$PATH`)

- Notes:
  - Builds the hash in 'real-time'; i.e. as commands are used
  - The "first time" you use a command, $PATH is searched, but the "next times", the hash lookup is used first! (*fast*)
  - $ hash –r to reset the hash (i.e. $PATH has changed)

```
steve@xerus:~$ hash
hits    command
   2    /usr/bin/whoami
   3    /bin/ls
steve@xerus:~$ hash -l
builtin hash -p /usr/bin/whoami whoami
builtin hash -p /bin/ls ls
steve@xerus:~$
```

## Command `<options>`

- Typical/conventional syntax:
  - Single letter options start with a single hyphen: `-B`
  - Less cryptic options are whole words or phrases, and start with two hyphens: `--ignore-backups`
- Which form?
  - Depends on the command (*e.g.* `adduser`)
  - Some commands support both (*e.g.* `ls, mv`)
- Some options take *arguments*
  - argument is the next word after the option
  - e.g.: `$ sort -o output_file input_file`
- A few programs use different styles of command-line options
  - For example, long options (not single letters) sometimes start with a single - rather than --
  - *e.g.* `firefox`
- Summary: use the man, man

## Differentiating Options and Arguments

- The options *usually* come first, but not always
- Options <u>always</u> start with a dash, single or double
- Arguments are usually filenames, directories, *etc*., on which to operate and can be a long, long list
- *Example*: file commands have list of files for arguments
  - There is a limit.
  - There are ways to get around these limits when they are encountered. (we'll investigate later)
- What if an argument starts with a dash?
  - e.g. try: `$ touch -foo.bar` ← *Strange, but <u>could</u> happen!*
  - *Hint*: special option '--' → what does it do?

## required vs optional arguments

- Some arguments are *required*
  - If you don't include the argument(s), the command gives an error
  - e.g. run `rm, mv` or `cp` without arguments
- Some arguments are optional
  - They do something special or have a default value
  - *e.g.* `cd <dir>`
    - <dir> is optional, without it, the default is `$HOME`
  - *e.g.* `ls <paths-or-files>`
    - without an argument, the default is . (pwd)
  - *e.g.* sort -- defaults are stdin and stdout
- How do you know if an argument is required or optional?
  - It can be tricky
  - The man pages might specify
  - Often documentation uses <> or [] to mean *optional*
    - Study `$ man cp` vs `$ man ls`

## Option vs Argument Order

- Does *order* matter?
  - Is `$ ls -al` different than `$ls -la` ?
  - Is `$ mv file1 file2` different than `$ mv file2 file1` ?
- For options, order *doesn't* matter
- For arguments, order often *does* matter
- This is what differentiates options from arguments
- Why? Think about these examples from perl:

```
use Getopt::Std;
&getopts('vm');
if($opt_v) { ... }
```

```
$dir = shift @argv;
$file = shift @argv;
```

## Command reuse, reduce, recycle

- As expected
  - Up/down arrow keys will scroll through the command history.
  - Left/right arrows move cursor
  - Typing inserts, backspace deletes
- This can be used to repeat or correct or modify commands
- The emacs keys work: ^p, ^n, ^a, ^e, ^t, ^d, ^k

## History

- `$ history`
- Very useful “bang” shortcuts:

```
$ !n
$ !-n
$ !!
$ !string
```

## Combining Commands on One Line

- A couple methods....

1. Separate with ";"
   - Don't start the 2nd until the 1st finishes
   - This is 2 commands. It's like typing in the 2nd command after the 1st one finishes.

```
$ time-consuming-program ; log_result
```

2. Separate with "&&"
   - This is seen as one command where...
   - 2nd command runs only if the 1st one succeeds

```
$ potentially-failing-program && cp results.txt ..
```

3. Separate with "||"
   - 2nd command runs only if the 1st one fails

```
$ rm file1 || echo file1 not found
```

## Background – getting the prompt back

- Unix is a multitasking OS and runs many programs at once.
- You can put your command in the background with "&" at the end of the command string

```
$ firefox -display 0 &
$ emacs userlog.log &
```

- If you forget the &, you can suspend a job with ^Z, then put in background with "bg"
- What do these commands do?

```
$ bg
$ fg
$ jobs
$ ps
```

## Background and don't die

- If the terminal ends, all background jobs end.
- **Unless**, you start with nohup (no hangup)

```
$ nohup really-long-command &
```

- Text output goes to file nohup.out