20875 Software Engineering

Useful shell commands

1 Terminals and shells

- A **terminal** (or **console**) is an interface for processing keyboard input and displaying text. The name comes from the fact that terminals used to be physical devices. Nowadays, the device is usually simulated in a graphical application (a "*terminal emulator*").
- A **shell** is an application that reads commands from its input and executes them. In the case of an **interactive shell**, the input is the keyboard input of a terminal. However, shells can also read their commands from a file (which is then called a "shell script").

Most interactive shells will understand the following keys:

Control+C	Interrupt the currently-running command
Control+D	Indicate end-of-file (see redirections below)
Up/Down	Browse the history of previously-entered commands
TAB	Complete the partially-entered command as far as unambiguously possible
TAB (again)	In case of ambiguity, list possible completions

For all the commands we will use, their documentation is accessible using the command `man`. For example, `man ls` gives the manual page for the command `ls`.

Anything after a `#` symbol is considered a comment and is ignored (unless the `#` symbol is itself inside a string delimited by `'` or `"`).

2 Filesystem

The filesystem is a (directed) graph whose nodes are *directories* (also known as *folders*) and *files*. Files are the leaf nodes. Nodes are labeled by strings: the file or directory *name*.

An *absolute path* for a file (or directory) describes where that file (or directory) is located in the filesystem: It starts with a slash, and is followed by the labels of the nodes on a directed path from the root to the file (or directory)'s node, separated by slashes.

Example 1. *Consider the filesystem tree in Figure 1. An absolute path for* `file_2` *is* "`/directory_A/subdirectory_D/file_2`".

Every directory d contains two specially-named subdirectories: "`.`" (a single dot) refers to d itself, and "`..`" (two dots) refers to directory containing d (its parent directory).

Example 2. *Another absolute path of* `file_2` *is* "`/directory_A/subdirectory_C/../subdirectory_D/file_2`".

Note. *The special directories "`.`" and "`..`" are aliases (technically: "hard links"), which we typically ignore when drawing the filesystem (as we did in Figure 1), allowing us to avoid cycles and draw it as a tree.*

```
+---+ directory_A
|   |
|   +---+ subdirectory_C
|   |   |
|   |   +--- file_1
|   |
|   +---+ subdirectory_D
|   |   |
|   |   +--- file_2
|   |
|   +--- file_3
|
+-- directory_B
```

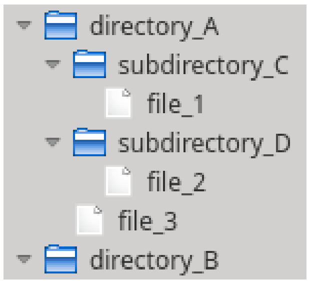

Figure 1: Filesystem tree for Example 1.

Any process (including the shell) has a *working* directory (also known as *current* directory, or *current working* directory), which may change over time. File paths can be expressed *relative* to this current directory. A path that does not start with a slash is considered a relative path, and consists in the labels on a path from the current directory to the targeted file or directory, separated by slashes.

Example 3. *If the current working directory is* `/directory_A/subdirectory_C`*, then a relative path for* `file_2` *is “*`../subdirectory_D/file_2`*”.*

3 Commands dealing with files and the filesystem

`pwd`	Print the working directory to standard output
`ls`	List files – by default, prints the contents of the current directory to standard output
`cd`	Change the current directory
`cat`	Print the contents of files to standard output
`less`	Display the contents of files (allows browsing them with Up/Down) – Type `q` to exit
`hexdump`	Print the contents of files in hexadecimal
`strings`	Print the parts of files that are printable (ASCII) characters
`cp`	Copy a file
`mv`	Move (rename) a file
`rm`	Remove (delete) a file
`mkdir`	Create (make) a directory
`rmdir`	Remove an empty directory

4 Other useful commands

`echo`	Print command-line arguments to standard output
`wget`	Download files from the internet
`curl`	Download files from the internet
`zip`	Compress and decompress files in the `zip` format
`tar`	Create and extract archives in the `tar` (and `tgz`) format
`top`	Display currently-running processes – Type `q` to exit
`time`	Run commands passed on the command line, measure the time they take to run
`touch`	Update the last-modified time of a file, create it if it does not exist
`chmod`	Change the access rights (read, write, execute) of a file
`chown`	Change the ownership (user and group) of a file
`sudo`	Run commands passed on the command line as `root` (superuser / administrator)
`head`	Print the beginning of a file
`tail`	Print the end of a file

5 Running executables

Many of the above commands actually correspond to executable files. We do not need to specify their complete (either absolute or relative) paths, because they are located in specially-configured directories where the shell searches for them.

Instead, if we want to tell the shell to run an executable file designated by its path, our command must contain a slash. For example, we could type an absolute path (since it always starts with a slash). To run an executable that is located in the current working directory, we can prepend its name with `./` (since the `.` relative path refers to the current working directory).

Example 4. *With the following commands, we make* `file_3` *executable and run it:*

```
cd /
cd directory_A
chmod +x file_3
./file_3
```

6 Standard input and output, redirections

By default, every process starts with 3 files already open: standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`). Unless otherwise specified, reading from standard input yields the operator's keyboard input, and writing to standard output or error prints on the terminal.

However, we can *redirect* `stdin` and `stdout` to actual files, or *pipe* them to other commands.

`command > path`	Redirect the standard output of `command` to the file designated by `path`
`command < path`	Take the standard input of `command` from the file designated by `path`
`command1 | command2`	Pipe the standard output of `command1` to the standard input of `command2`

The point of having `stderr` in addition to `stdout` is to give commands an opportunity to report errors to the user even when `stdout` is redirected. For this reason, it is less frequently useful (albeit possible) to redirect `stderr`.

Example 5. *Various redirections:*

```
# Write the current directory's file list to a file called "list.txt":
ls > list.txt

# Print the content of "list.txt" in hexadecimal, pipe to the "less" pager:
hexdump list.txt | less

# Print the content of "list.txt"
cat < list.txt
```