unix_shell

November 2, 2015

1 The unix shell

1.1 What and why?

The shell is a program which shields off the (gory details of) the operating system from you. Examples: DOS shell, Unix shells: sh, csh, tcsh, bash

We shall be focusing on bash (Bourne Again SHell)

Why would we care about the shell? 1. It is behind sooooooooo many things 2. You can use it from within a python or C program 3. It facilitates complex and repetitive tasks

2 Files and directories

They form a tree-structure - A *directory* is a container for folders/subdirectories - A *file* is a container for information (bytes) which is handled as a single unit - The *path* is the location of a file - The *absolute path* gives you the location with respect to the root '/' of the directory tree - The *relative path* gives you the location with respect to the *current* collection > Examples:
`onderwijs/colleges/CasimirComputerCourse`
(This is a relative path; I am in `/home/jost` This is my *home directory*)
`/home/jost/onderwijs/colleges/CasimirComputerCourse` (This is an absolute path)
`~/onderwijs/colleges/CasimirComputerCourse` (This is an absolute path; `~/` is the - user dependent - home dir)

Special paths: `~/`
Home dir `./`
This is the current dir (where you are now)
Parent dir `../`
(the dir the present dir is a subdirectory of).

Creating and deleting files and directories 1. Remove files: `rm myfile` 2. Create files (i) from editor (*save as*); (ii) touch myfile 3. Create directory: `mkdir ~/my_new_dir`; `mkdir /home/jost/my_new_dir`; `mkdir my_new_dir` 4. Remove directory: `rmdir ~/my_new_dir` (`my_new_dir` should be empty!) 5. Copy files: `cp file1 file2` 6. Move files/directories: `mv src dest` 7. Files have *permissions*: read (r), write (w) and execute (x). These permissions can be different for different users: the *owner* (o), the members of the same user group as the owner (g) or everybody (w)

General structure shell commands: > cmd flags args
flags: start with - args: names of (files) objects, variables

Example: `ls -lartF ~/onderwijs`

cmd: `ls`; flags `-l, -a, -r -t, -F`; args: `~/onderwijs` To see the meaning of these flags: explainshell.com - In this case: sort with latest file as the last (`-rt`), show all files (including hidden ones, `-a`) and show whether it is a directory or a file (`-F`, puts `/` after dir-names)

Wildcards: 1. `ls *` (anything), 2. `?` (a single arbitrary character), 3. `[0-9]` (a digit)

For example,

```
ls *[0-9][0-9]*
```

lists all files/directories with two sequential digits in the name, eg. `fragment34_lagrange.mkv`
Interesting, but *DO NOT TRY THIS AT HOME*:

```
rm -rf *
```

`-f` means 'force'; `-r` recursively scans subdirectories.
The file system may encompass different physical devices, such as mem keys, remote file systems, drive partitions. You can visualize these using mount.
A mount couples the root of a file system to a node on *your* directory tree.

```
mount -a
```

realizes all mounts. The information about the mounting structure in your system can be found in `/etc/fstab`
Locating files:

```
find . -name jos*.tex
```

(takes wildcards) or:

```
locate jos
```

(uses a database and is therefore fast; updatedb to refresh the database) Does not work on all systems.

3 Your environment

When you are in the shell, you can have an environment. You can modify this environment.

```
alias l='ls -lartF'
```

After this, `l` is equivalent to `ls -lartF`.
Variables:

```
myname='Jos Thijssen'
```

is a shell variable. Usually

```
export myname='Jos Thijssen'
```

is preferable (wider scope of the variable).
Print things:

```
echo 'hello jos'
```

Print variables:

```
echo myname
```

just prints 'myname'.

```
echo $myname
```

prints ‘Jos Thijssen’
Quotes:

```
echo '$myname'
```

prints `$myname`

```
echo "$myname" prints 'Jos Thijssen'
```

Print unix commands or their output?

```
echo 'ls -l'
```

or

```
echo "ls -l"
```

prints `ls -l`

```
echo `ls -l`
```

prints list of files/directories
Some users are more equal than others: super users (admins)
`sudo` *cmd*
means that you execute the command as an administrator (if you have permission for this).
Command prompts: variables PS1, PS2, PS3..

```
echo $PS1
```

Try:

```
PS2='yes dear? '
```

visible after incomplete quotes.

3.0.1 What are your peers doing?

```
who
```

`who` is on the system

```
w
```

What is the load on the system?

```
top
```

Which processes are slowing you down?
List all runing processes:

```
ps ax
```

Various convenient commands

```
date
```

prints the date and

```
time
```

prints the time.

4 Manipulating files and read what's in them

```
cat myfile
```

prints file content.

```
sort myfile
```

print (alphabetically) sorted file.

```
sort -n myfile
```

print numerically sorted file

```
wc myfile
```

Prints number of lines, words and chars
If you want to search content in a file:

```
grep 'jos' myfile
```

searches for the text 'jos'

```
grep -i 'jos' myfile
```

Searches for 'jos' case insensitive, i.e. it also finds 'Jos'

```
grep -ir 'jos' *
```

searches recursively down the dir tree.
Changes text in (multiple) files:

```
sed -e 's/jos /Jos /g' myfile
```

outputs myfile, with all occurrences of 'jos' changed into 'Jos'

5 I/O, pipes

Unix commands typically take input and generate output. Eg:

```
cat myfile
```

takes the content of myfile as input and generates the same output.

```
sort myfile
```

reads input and ouputs the sorted content.
We can write the ouput to a file using >:

```
cat file1 > file2
```

This is equivalent to `cp file1 file2`
Append:

```
cat file1 >> file2
```

appends the contents of file1 to that of file2 and file2 contains the output.
Reading input from a file can be done using < (rather rare)

```
sort myfile
```

is equivalent to

```
sort < myfile
```

The output of a command can be fed into another command. For this we use pipes: |

```
ls -l | sort
```

takes the output of the `ls -l` command and sorts it.
A third way to output the sorted content of `myfile`:

```
cat myfile | sort
```

6 Doing things over and over

Loop over files:

```
for file in *.txt
  do
    ls $file
  done
```

This lists all the files ending with `.txt`
Suppose you want to run a program many times with input variables in some file `in_params`.
Contents of `in_params`

```
outfile10_1.0 10 1.0
outfile10_2.0 10 2.0
outfile20_1.0 20 1.0
```

You have a simulation which takes as input the outfilename, a size (an integer) and a temperature (a float). You want to run the simulation for N=10, T=1.0 etcetera. Then you fill the file `in_params` as above and type

```
while read -r line; do simul; done < in_params
```

7 Shell scripts

Collect several unix commands into a script (file). Contents of such a file. E.g.: the file `listfile` may contain:

```
#!/bin/bash
ls -l
```

If the file `listfile` was created in some editor, it does not have execute permissions:

```
./listfile
```

leads to

```
listfile: command not found
```

(note the `./` which tells you that the shell should search in the current dir)
Solution 1:

```
bash listfile
```

works!
Solution 2:

```
chmod +x listfile
```

followed by

```
./listfile
```

works!
Interesting things you can do with scripts: give arguments.
In the script, they are recognized as $1 $2 etc.
The script `test_args` reads:

```
#!/bin/bash
echo $0 $1 $2
```

Make this executable and call using

```
./test_args anton michael
```

yields:

```
test_args anton michael
```

`$0` is the name of the script!