

Using the Command Line Interface for fun and profit

Linuxdays 2008



Eric Dondelinger // eric.dondelinger@pt.lu
Marc Seil // marc.seil@uni.lu

This course will show

- a short introduction into very basic shell use
- a brief introduction to a number of useful commands
- a more advanced usage of the different commands and the command line
- practical exercises using many techniques

- unix/linux provides many small utilities, each performing a single task, but doing this exceedingly well
- the shell allows launching and combining these commands, allowing automation of complex tasks

- **man, info, apropos (man -k)**
- **<command> -help (--help | -h)**
- **/usr/share/doc**
- **“Advanced Bash-Scripting Guide”**
by Mendel Cooper : tldp.org/LDP/abs/html/
- <http://www.google.com/>
- <http://groups.google.com/>
- help@linux.lu

- filesystem is hierarchical, tree structure
- **pwd** – print working directory
- **ls** – list directory contents
- **cd** – change directory
- “.” is the current directory, “..” is it's parent

create, delete, move files/directories

- **touch file** – create file or update time
- **mv source destination** – move file / dir
- **rm file** – remove file
- **mkdir dir** – make directory
- **rmdir dir** – remove directory

- get file details with
`ls -l`
- get the file type
 - `file filename`
- change file owner with
`chown user:group file`
- change permissions with
`chmod [ugoa][+ -][rwx] file`

- files which contain user / group information
 - `/etc/passwd`, `/etc/groups`, `/etc/shadow`
- Commands for user and group creation
 - `useradd`, `groupadd`
- Command to retrieve account informations
 - `id`

- The shell is your gate to a huge set of powerful commands
- The shell is configured through a set of files (depending on the linux distro)

`/etc/profile, /etc/bash.bashrc,`

`/etc/environment`

`~/.bashrc, ~/.profile ...`

- The invoked commands are stored in a history file
 - ~/.bash_history (after logout)
- with the **up** and **down** cursor keys the history can be browsed
- with [**CTRL**] – [**R**] a reverse history search is possible

- [CTRL] – [A] allows to jump to the beginning of the line
- [CTRL] – [E] allows to jump to the end of the line
- [CTRL] – [C] interrupts the current command or line
- [CTRL] – [L] redraw/clears the terminal

- [CTRL] – [T] swaps two characters
- [ESC] – [T] swaps two words
- [CTRL] – [W]ipe deletes one word backwards
- [CTRL] – [Y]ank paste the last word deleted with [CTRL] – [W]

- [CTRL] – [K] cleans the rest behind the cursor
- [CTRL] – [U] cleans everything before the cursor
- [CTRL] – [D] exits the current shell
- [TAB] autocompletes filenames or commands

- The environment variables are used to customize the environment in which the applications are executed
- The major commands to customize and create these variables/attributes are:

`export`, `unset` and `printenv`

- A variable can be assigned with “=”

```
VAR1="hello world"
```

```
echo $VAR1 # will print the contents to stdout
```

- If an application should run in an environment where the variable is available the command export can be used

```
export VAR2="hello world"
```

```
xterm -e "echo $VAR2; sleep 2"
```

- The command **unset** can be used to remove environment variables

```
unset VAR1
```

```
unset VAR2
```

```
echo $VAR2
```

- The command **printenv** can be used to print the environment variables to stdout

```
printenv
```

- The command **env** can be used to run an application in an appropriate customized environment

```
env foo=bar /bin/bash --noprofile --norc
```

```
$ printenv # [CTRL-D] to logout
```

```
env -i foo=bar /bin/bash --noprofile --norc
```

```
$ printenv # [CTRL-D] to logout
```

Some major environment variables

- **PATH** - indicates the directories where the shell tries to find the commands
- **PWD** – the current working directory (can also be retrieved with the command `pwd`)
- **USERNAME** – the user which invoked the shell
- **HOME** – the user's home directory

- **LD_LIBRARY_PATH** – shared library directories (beside the default values)
- **\$?** - the return error of the previous command [**0=ok,true** ; **0<>error,false**]
- **\$0** – contains the string of the current running application
- **\$#** - the number of arguments passed
./cmdArguments.sh -d second third

- **set** can be used to customize a broad range of options which allow to customize the behaviour of the shell and the command line

set -o vi # default **set -o** emacs

set -u # an unset variable will be reported as an error

- **man set** will bring up a lot of different customizations

The **alias** command can be used to create shortcuts to customized applications calls or complete command sequences

alias # list all defined aliases of the running environment

alias ll="ls -l"

ll # executes the application **ls** with the argument **-l**

- **which** <command> - look where a command is located
- **locate** / **updatedb** – database of filenames
- **find** – find files and do stuff with them
- **grep** – search within files

- **<command> &** to run it in background
- **jobs** – list processes launched in this session
- **ctrl-z** – stop the current job
- **bg** – send job to background, **fg** – foreground
- **ps** – process status

- **top** - “live” show of running processes
- **nice** – set job / command priority (-20 – high priority; 19 – low priority)
- **kill, killall** – send signals to processes

- streams are the foundation for input/output
- shell provides 3 standard streams
 - 0: standard input
 - 1: standard output
 - 2: error messages
- keyboard - is standard input (0)
- stdout - is your screen (1)

- redirection = file descriptor manipulation

program < input_file > output_file

- Example (reading from stdin)

- `./redirection01.sh < echoLine.txt`

- `cat echoLine.txt | ./redirection01.sh`

➤ `>>` appends to a file

```
echo "hello world" > test.file
```

```
date >> test.file
```

```
echo "end of test" >> test.file
```

➤ Example

```
./redirection02.sh
```

```
cat redirection02.out
```

➤ Example

Redirecting **stderr** to **stdout**

```
ls zzz
```

```
ls zzz 2>/dev/null
```

```
ls zzz 1>/dev/null
```

```
ls zzz 2>&1 > /dev/null
```

```
ls zzz 1>/dev/null 2>&1
```

Example

Read the input from **stdin**

- `read VAR01; echo "contents: $VAR01"`

- two programs can be hooked together
stdout from the first is sent in as the stdin of the second

- `program1 | tee log_file | program2`

- Example

```
cat /etc/passwd | tee stdin.txt | grep root
```

```
cat stdin.txt # should contain a copy of stdin
```

- **head, tail** – display start/end of file
- **wc** – word count
- **grep** – selects lines/matches based on regexp
- **cut** – can separate fields
- **diff** – tells differences between files (even binary)
- **strings** – outputs strings from binaries

- **sort** – sort lines
- **uniq** – select individual lines
- **sed** – edit streams of data
- **awk** – pattern scanning and processing

- print `n` first lines of a file:

```
head -n filename
```

- print `n` last lines of a file:

```
tail -n filename
```

- get line `x`

```
head -x /var/log/messages | tail -1
```

- observe a logfile as it is written to:

```
tail -f /var/log/messages # kill this with ctrl-c
```

- globally look for regular expression and print
- looks for character strings in files and writes all matching lines to stdout
- select lines containing a certain **word**
 - **grep pop3 /etc/services**
- use **-l** (little L) option to get filenames only

```
grep -l pop3 /etc/*
```

- use **-s** to remove warnings / errors
- use **-i** to ignore case (uppercase or lowercase are both matching)
- use **-o** to return only the part of the line which is matched by the pattern
- use **-v** to invert the sense of matching (**--invert-match**)

- can count the number of characters, words, lines in a file
- the three common options are:
 - l return the number of lines
 - w return the number of words
 - c return the number of characters

Example

```
cat /var/log/message | grep restart | wc -l
```

- **cut** can cut files into peaces that can be pasted back together for some other use
- **cut** can operate on character- or field-basis or a combination of both
 - **-f** option indicates the fields (start at 1)
 - **-d** defines the delimiter

Example

Retrieve a table from the file `/etc/passwd` which contains the user logins and their respective shell

➤ `cat /etc/passwd | cut -f 1,7 -d :`

sort can be used to sort lines, fields

Example

Sorting an input by numerical value

- **sort -n** numbers01.txt | **less**

Example

Reverse sort order

- **sort -r** letters01.txt | **less**

Example

sort the password file by user id, extract users having home directories in /home

```
sort -t: -n -k3 /etc/passwd | grep home | cut -d: -f1,3
```

[HINT] In order to increase the performance of this command sequence, the `grep` command should precede the `sort`

- Removes identical lines from a file. For this to work, the file must be sorted.
- If no filenames are given, `uniq` uses `stdin` and `stdout`
- The `-d` option reports the duplicate lines (just gives the line once)
- To get a count of repetitions, use `-c` option

➤ Example

sort by count

```
cat sort01.txt | sort | uniq -c | sort -n
```

only report duplicate lines

```
sort sort01.txt | uniq -d
```

Usage of regular expressions

- Pattern matching

Some commands / applications which allow using regular expressions

- grep, awk, sed, vi, perl, python and many others including most programming languages

- `[a-z]` matches single char, of type listed
 - ^ can be used at start of pattern to negate it (eg. `[^A]` any character but not an uppercase A)
- `.` matches any single char except newline
- `^` and `$` - `^` matches start of line, `$` end

- **A?** zero or one matches
- **A*** null or multiple matches
- **A+** one or multiple times
- **A{n,m}** from n to m repetitions
- **** escapes special characters
- **A|B** logical or
- **(..)** allows to group patterns

- Where to find more information
 - www.regular-expressions.info
 - en.wikipedia.org/wiki/Regular_expressions
 - www.refcards.com
 - and many different books

Example

Retrieve from a spam archive all the unique email addresses mentioned in the **From:** block and sort them.

- `grep "^From:.*$" spam.mbox > tmp1`
- `grep --only-matching "[^ <]*@[^ >]" tmp1 > tmp2`
- `sort tmp2 | uniq`

Examples

Retrieve a list of subdirectories till a recursive level of 2

➤ `find -maxdepth 2 -type d`

Now check the disk usage of this directories and sort the outcome

```
find -maxdepth 2 -type d  
-exec du --summarize {} \; | sort
```

Examples

If we want to remove the hidden directories a further **egrep** is necessary

```
➤ find -maxdepth 2 -type d  
    -exec du --summarize {} \; | sort -n |  
    egrep "^[09]*.*[.]([/][^./][^/]*)*$"
```

[HINT] a hidden file is identified through the char sequence **“/.”** but a file can have a suffix like **“.txt”**

Examples

Find all the files which where the state changed in the last 2 minutes in your home directory

- `find -cmin -2`
- `find -cmin -2 -printf "%A+\t%f\n"`
- `find -cmin -10 -cmin +4
-printf "%A+\t%f\n"`

The **printf** command allows to produce a formatted output (similar to fprintf in “C”)

```
printf FORMAT [argument]
```

Examples

```
printf “an integer value %d\n” 10
```

```
printf “an integer value %3d\n” 11
```

```
printf “an integer value %03d\n” 11
```

Examples

```
printf "a string value: -> %s <-\n" "hello"
```

```
printf "the hex value of %d is %x\n" 15 15
```

- The **test** command can be used for a broad range of compare operations. It includes string, integer, file and logic tests

string tests (\$? - the return value)

- **-n** string at least one character [**true=0**]
- **-z** zero length [**true=0**]
- **=** strings are equal [**true=0**]
- **!=** strings differ [**true=0**]

Number tests

- › **-gt** greater then [**true=0**]
- › **-ge** greater or equal [**true=0**]
- › **-lt** less then [**true=0**]
- › **-le** less or equal [**true=0**]
- › **-eq** equal [**true=0**]
- › **-ne** not equal [**true=0**]

File tests

- › **-e** file exists [true=0]
- › **-d** is a directory [true=0]
- › **-x** file is an executable [true=0]
- › and many more ...

Logical tests

- › **-a** logical AND operation [true=0]
- › **-o** logical OR operation [true=0]

Examples [true=0]

- › `test -n "hello world"; echo $? # string test`
- › `test 12 -gt 24; echo $? # integer test`
- › `test -d $HOME; echo $? # file test`
- › `test -w $HOME/..; echo $? # file test`
- › `test 12 -eq 12 -a 12 -gt 1; echo $? # logical test`
test in scripts use the `[[]]` and `&&` and `||`

- The bash provides some keywords to evaluate enclosed tasks.

(We have already seen `[[]]`)

- Evaluate a command `()`

(replaced the backticks)

```
VAR01=$( ls /etc/* | grep passwd )
```

```
echo $VAR01
```

➤ Making some basic calculations (())

(only integer values -> else use **bc**)

```
((a=10))
```

```
echo $a;
```

```
((a--))
```

```
echo $a;
```

```
((a+3)); echo $a
```

```
a=$((a+3)); echo $a
```

➤ Manipulating Variables `${VARNAME..}`

only some examples

```
VAR01="hello world"
```

```
echo ${#VAR01}
```

```
echo ${VAR01//l/+} # replaces the first l
```

```
echo ${VAR01///l/+} # replaces all the l
```

```
echo ${VAR01:6} # from the 7 char
```

```
echo ${VAR01:0:3} # the first 3 characters
```

....

- A bash script is a text file which contains the commands, variables and functions which should be executed by the bash.
- A bash script is (can) be identified by the first line
- The basic structure (sha-bang - **#!**)

#!/bin/bash

your stuff

- Sometimes the bash scripts start with the following line

```
#!/bin/sh
```

```
some stuff
```

`/bin/sh` indicates the default shell (Ubuntu – the dash shell). In the case that the bash shell is the default shell, `/bin/sh` indicates that no `rc` files are read (same as `/bin/bash --norc`).

- The **if** statement follows the **test** constructs
- Following the “Advanced Bash-Scripting Guide” the test expression should be enclosed in `[[]]`

```
if [[ 12 -lt 24 ]]; then
    echo "12 is less than 24";
fi
```

Example

Create a bash script which compares two numbers and returns if both are equal or which of the number is biggest

Solution: `exampleIf02.sh`

Example

Create a script which checks if the first argument is a “-n”. Further it should report the rest of the arguments.

[HINT] **man shift, \$***

Solution: `exampleIf02.sh`

- Loops can be used to execute a set of commands multiple times
- The **for** loop can be used in a “C”-like notation

Examples

```
for ((i=0;i<11;i++)); do  
    echo $i  
done
```

- The for loop can also be used on strings or shell globbing (*.txt) substitutions

Examples

```
SAFEIFS=$IFS; IFS="" " # try to change the IFS
```

```
VAR01="hello world 1 2 3"
```

```
for ENTRY in $VAR01; do
```

```
    echo $ENTRY
```

```
done
```

```
IFS=$SAFEIFS
```

Examples

```
for ENTRY in *.sh; do
    echo $ENTRY;
done
```

- The **while** loop is loop which is executed till a condition is reached (till **test** returns 1)

Examples

```
((i=0))
```

```
while [[ $i -lt 11 ]]; do
```

```
    printf "counter %2d\n" $i; # or test %02d
```

```
    ((i++))
```

```
done
```

- It is also possible to use the **while** loop in a “C” - like notation

Examples

```
((i=0))  
while (( i < 10)); do  
    printf "counter %02d\n" $i  
    ((i++))  
done
```

Solution: exampleWhile01.sh

- The command **read** can be used to *read* something from **stdin**
- **read** will pass the input to the variable after [enter]

```
echo -n 'enter a value: '; read VAR01  
echo $VAR01
```

Solution: exampleRead01.sh

- **read** can be used to read a stream passed to a bash script (with < or |)

```
while read LINE; do
    echo $LINE
done
```

Solution: exampleRead02.sh

functions allows to group commands in the bash shell

```
function printHello {  
    echo "hello";  
}
```

printHello

it is also possible to pass arguments to a
function

```
function printHello {  
    echo "hello" $1;  
}
```

printHello girls

Exercise string repeat

write a script which reads a string and repeat value from stdin and creates the respective output.

[HINT] use the command **read**

solution: exercise01.sh

Exercise batch rename

Create a script which simulates a rename of the files which have a suffix and which are part of a target directory passed as argument. The counting should start with 0000 and the **rootName** is passed as second argument.

sample output

file.txt -> **rootName**0000.txt

zdein.txt -> **rootName**0001.txt

Solution: exercise02.sh

- write script to sort fotos by date, into subdirectories, based on their EXIF data, and set the copyright EXIF tag
- generate properly rotated thumbnails
- generate simple HTML indexes
- read source / destination directories from the command-line arguments
- allow for storing constants in a config file (thumbnail size, author)

?