Operating Systems

Complutense University of Madrid

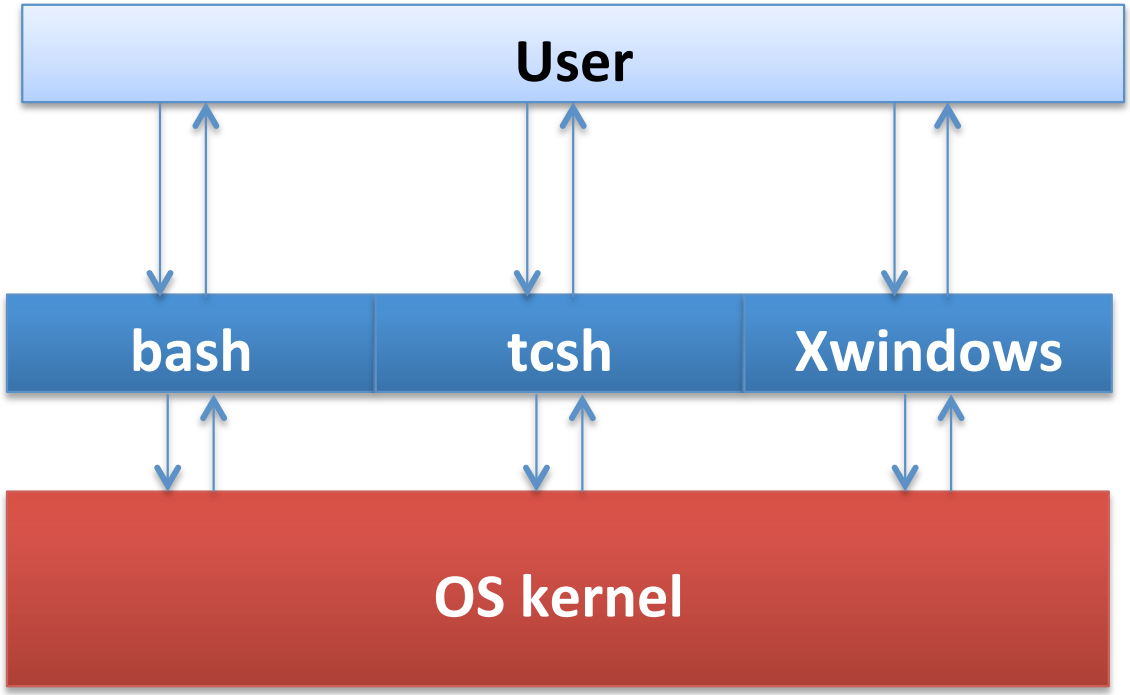
2020-2021

Introduction to the BASH shell

Juan Carlos Sáez

Shell

- Program enabling the user to interact with the operating system



The BASH shell

- Bash (Bourne-again shell) is a Unix shell written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell (sh)
- BASH is the default shell in many UNIX and UNIX-like OSes: GNU/Linux, Mac OS X, Solaris
- It has been also ported to Microsoft Windows (Cygwin)
- Other shells:
  - **sh** or C shell: the syntax of this shell resembles that of the C programming language
  - **tcsh** or TENEX C shell: a superset of the common C shell, enhancing user-friendliness and speed
  - **ksh** or the Korn shell: sometimes appreciated by people with a UNIX background

BASH invocation

Interactive shell behavior:

- Reads startup files (`~/.bashrc`)
- Prompts are set (`PS1`,`PS2`)
- Command history and history expansion are enabled (`HISTFILE`)
  - It is possible to search the command history (`Ctrl+R`)
- Alias expansion is enabled
- Modify signal handlers (`Ctrl+C`).
- Commands are read line by line (`readline`) and executed upon read
- If the EOF is read (`Ctrl+D`) or the `exit` command is typed the shell terminates

Running commands

Command types

- **Internal or Built-in commands**: these are implemented within the shell program (internal functions)
- **External commands**: standalone programs or utilities installed in the system (binary executable files or scripts)

BASH scripts

- A sequence of shell commands can be included in a text file, referred to as **a BASH script**, to be executed at any time
  - When a BASH script is launched, the OS creates a new BASH process

Built-in commands

Bourne Shell built-ins:

`:`, `.`, `break`, `cd`, `continue`, `eval`, `exec`, `exit`, `export`, `getopts`, `hash`, `pwd`, `readonly`, `return`, `set`, `shift`, `test`, `[`, `times`, `trap`, `umask`, `unset`.

BASH built-in commands:

`alias`, `bind`, `builtin`, `command`, `declare`, `echo`, `enable`, `help`, `let`, `local`, `logout`, `printf`, `read`, `shopt`, `type`, `typeset`, `ulimit`, `unalias`.

Essential commands

Command	Description
`ls`	displays a list of files in the current working directory
`pwd`	displays the path of the current working directory
`cd directory`	change the working directory
`man command`	reads man pages on command
`apropos string`	searches the whatis database for strings
`file filename`	displays file type of file with name filename
`cat textfile`	dumps the content of textfile on the screen
`exit or logout`	terminates the shell session
`grep`	searches in files for lines containing a match to a pattern
`echo`	prints a line of text
`env`	prints the set of environment variables
`export`	set the export attribute for variables

Variables and operators

Variables

```
a=5                 #assignment
echo $a             #expansion
b=$(( $a+3 ))       #example of integer arithmetic operation
b=$(( $a<<1 ))      #bitwise operation
```

Arithmetic and bitwise operators:

```
+ - / * % & | ^ << >>
```

BASH redirections

- Default file descriptors:
  - `stdin`(0), `stdout`(1) and `stderr`(2)
- Default behaviour when launching a program (binary or script) from a text console:
  - `stderr` and `stdout` are mapped to the console
  - `stdin` is associated with the keyboard
- Standard output redirection:
  - `command > filename`
- Standard error output redirection:
  - `command 2> filename`
- Standard input redirection:
  - `command < filename`

BASH redirections

Examples

```
ls -l > listing
ls -l /etc >> listing
ls /bin/basha 2> error
find / -name 'lib*' -print > libraries 2>&1
```

BASH pipes

- A command's standard output becomes the standard input of another command

```
ls -l | more
```

- Pipes and redirections can be combined

```
ps aux | grep -v root > ps.out
```

Lists of commands

- **`$?`** builtin variable
  - Exit status (code) of the last command
  - Convention: 0 means success; non-zero means failure
- `cmd1 ; cmd2`
  - `cmd2` is executed upon termination of `cmd1`.
    - The final value of `$?` is the status code of `cmd2`
- `cmd1 && cmd2`
  - `cmd2` is executed in the event that `cmd1` succeeded (status = 0)
- `cmd1 || cmd2`
  - `cmd2` is executed in the event that `cmd1` failed (status != 0)

Foreground vs. background execution modes

- Non-graphical interactive programs are typically invoked in **foreground**
  - This is the default execution mode
  - When running a command in foreground from the shell, the shell's prompt is not displayed until the command completes
- Applications with a user interface (GUI) are typically launched in **background**
- Users can launch commands in background by appending '&' to the command:

```
$ xeyes &
[2] 7584
```

Job_ID

PID

Bash wildcards

- Wildcards make it possible to build patterns to refer to a set of files in a simple way
  - `*` matches every string/substring in a file name
  - `?` matches a single character in a file name
  - `[set]` matches every character in `set`
    - Example of set: `[abc]`
- Example of pattern with wildcards: `?[a-c]*.h`
  - Matches those files whose name ends with ".h" and begins with any caracter followed by 'a','b' or 'c'
- Patterns with wildcards can be used almost everywhere in BASH scripts and interactive sessions
  - Example: `rm ?[a-c]*.h`
  - What does the above command do?

Command substitution

- A command substitution (or expansion) enable to turn the output messages of a command into a string
  - In doing so, linebreaks are removed from the output
  - The resulting string can be used as an argument to another command or stored in a variable

- Example:

```
num=$( ls a* | wc -w )
```

- Alternative syntax:

```
num=`ls a* | wc -w`
```

BASH scripts

- A shell script is a text file that contains a sequence of shell commands
  - Comments start with #
- When a script is launched, a new BASH process is created to run the various commands in the script

my-script.sh

```
#!/bin/bash
mkdir tmp
cd tmp
echo hi > file
cd ..
```

- How to run the script?:
  - `$ ./my-script.sh`
- The file must have execute permissions
  - `$ chmod +x my-script.sh`

BASH scripts

Command-line arguments in shell scripts

- The various arguments can be referenced using the following special variables: $1, $2, $3 ... $9
- The $0 variable stores the path of the script, as stated during invocation
- The $# variable denotes the total number of arguments
- The `shift` removes the first argument and left shifts the remaining arguments
  - $2 becomes $1,$3 becomes $2, and so on.

Conditional statements (I)

if-then-else

```
if condition ; then
    THEN_BLOCK
else
    ELSE_BLOCK
fi
```

Important note

When evaluating the condition, 0 means “true”, and !=0 means “false”

Conditional statements (II)

Example

```
if test -x /bin/bash ; then
    echo "/bin/bash has execute permissions"
else
    echo "/bin/bash does not have execute permissions"
fi
```

Alternative syntax ...

```
if [ -x /bin/bash ]; then
    echo "/bin/bash has execute permissions"
else
    echo "/bin/bash does not have execute permissions"
fi
```

Conditions (I)

Conditions on strings

```
str1 = str2   # True if both are the same
str1 != str2  # True if strings differ
-n str        # True if non-empty string
-z str        # True if str is the empty string
```

Conditions on files

```
-d file    # Is it a directory?
-e file    # File exists?
-f file    # Is it a regular file?
-r file    # Does it have read permissions?
-s file    # Is it a non-empty file?
-w file    # Does it have write permissions?
-x file    # Does it have execute permissions?
```

Conditions (II)

Conditions on integer expressions

```
exp1 -eq exp2     # Both expressions are the same
exp1 -ne exp2     # The expressions differ
exp1 -gt exp2     # exp1 > exp2
exp1 -ge exp2     # exp1 >= exp2
exp1 -lt exp2     # exp1 < exp2
exp1 -le exp2     # exp1 <= exp2
! exp             # True if exp is false
```

For loops

Syntax #1

```
for variable in values
do
    LOOP_BODY
done
```

Syntax #2

```
for (( i=0 ; $i<N; i++ ))
do
    LOOP_BODY
done
```

Example #1

```
for i in `seq 0 1 9`
do
    echo $i
    sleep 1
done
```

Example #2

```
for (( i=0 ; $i<10; i++ ))
do
    echo $i
    sleep 1
done
```

while loops

Syntax

```
while condition; do
    LOOP_BODY
done
```

Example

```
while [ $# -gt 0 ] ; do
    echo $1
    shift
done
```

Regular expressions (I)

- Regular expressions constitute a powerful mechanism to detect text lines that follow a certain pattern
- External commands such `grep`, `sed` or `awk` are equipped with regular-expression support

Basic blocks

- **`char`**: matches a specific character (e.g., 'a')
- **`.`** : matches any character
- **`^`** : specifies the beginning of the line
- **`$`**: specifies the end of the line
- **`[set]`** : matches any character in `set`
- **`[^set]`**: matches any character not belonging to `set`

Regular expressions (II)

Repetition operators (previous item matches)

- `?` : matches 0 or 1 occurrences.
- `*` : matches 0 or more occurrences.
- `{n}` : matches exactly n occurrences.
- `{n,}` : matches at least n occurrences.
- `{,m}` : matches m occurrences at the most.
- `{n,m}` : matches a number of occurrences no lower than n and no greater than m.

Regular expressions (III)

Examples

- `a`: Any string that contains an 'a'.
- `ab`: Any string that contains the "ab" substring.
- `a.b`: Any string that contains the 'a' and 'b' characters and there is just one other character between them.
- `^[abc]`: Any string beginning with an 'a', 'b' or 'c'.
- `[^abc]`: Any string containing characters other than 'a', 'b' and 'c'.