Introduction aux Systèmes d'Exploitation

Unit 3: Scripting

François Taïani

http://britishlibrary.typepad.co.uk/digitisedmanuscripts/2011/08/index.html

Unit Outline

- **Shell Variables & Environment Variables**
- **Command line substitutions**
  - ➔ Globbing
  - ➔ Quotes
- **Shell Programming**
  - ➔ Scripts, loops, tests
- **A note on $PATH**
- Note:
  - ➔ (ba)sh syntax and workings
  - ➔ many other kind of shells (csh, zhk, ksh, tclsh, zsh, ...)
  - ➔ principles similar, but syntax & details may vary

Shell Variables

```
$ MA_VARIABLE="quelle belle journee"
$ echo MA_VARIABLE
MA_VARIABLE
$ echo $MA_VARIABLE
quelle belle journee
$
```

- Defined using '=' (no space around the equal sign)
- Value retrieved by prefixing a $ sign
- "Normal" shell variables only exist within a shell instance
  - ➔ Very different from *Environment Variables*

Environment Variables

- Special type of variables **managed by the OS**

- One set of environment variables **per process**
  - ➔ Java: `System.getenv(..)`
  - ➔ C: `getenv(..)` (+ `putenv(..)`, `setenv(..)`, `unsetenv(..)`)

- **Inherited** by child processes
  - ➔ E.g. by `Runtime.exec()` [Java] or `fork()` + `exec()` [C]
  - ➔ For C: man -s3 <function> to learn more about it

Querying Environment Var

- In Java

```
while(true) {
   out.print("Entrez le nom d'une variable: ");
   String variable = in.readLine();
   if (variable==null) break;
   out.println("'"+variable+"'"     +
               " a pour valeur : " +
               System.getenv(variable));
} // EndWhile
```

- 'null' → no environment variable with this name

Shell & Environment Vars

- Like any process each shell has its environment
  - ➔ Made available as shell variables
  - ➔ Inherited by commands launched from the shell

- Special commands
  - ➔ **printenv** : print all env variables
  - ➔ **env** : same with no args (but does more)
  - ➔ **export** : same with no args (but does more, see after)

Shell & Environment Vars

```
$ MA_VARIABLE="quelle belle journee"
$ echo $MA_VARIABLE
quelle belle journee
$ java EnvVarObserver
Entrez le nom d'une variable: HOME
'HOME' a pour valeur : /home/ftaiani
Entrez le nom d'une variable: MA_VARIABLE
'MA_VARIABLE' a pour valeur : null
Entrez le nom d'une variable:
```

- MA_VARIABLE is not an environment variable
  - ➔ It does not get inherited by the JVM process

Shell & Environment Vars

```
$ export MA_VARIABLE
$ java EnvVarObserver
Entrez le nom d'une variable: MA_VARIABLE
'MA_VARIABLE' a pour valeur : quelle belle journee
Entrez le nom d'une variable:
$
```

- 'export' transforms a local variable into an env variable
  - ➔ Alternative syntax: export MA_VARIABLE="new value"
  - ➔ MA_VARIABLE is now inherited by child processes

A note on $PATH

- An environment variable
  - ➔ accessed as a normal shell variable
  - ➔ but also exists outside the shell

- Set to a default value by system

```
$ echo $PATH
/home/ftaiani/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/
usr/games:.
```

- Used to find executables, or with 'which'

Command Line Substitutions

- Reminder (Unit 1): Shell's fundamental cycle
  - 1- Print prompt, e.g. '`MacFrancois ftaiani [ftaiani] $`'
  - 2- Read line from user (terminated by newline)
  - 3- **Interpret line**, and execute command(s) found in line
  - 4- Wait for execution of command(s) to terminate
  - 5- Go back to #1
- A lot happens during this interpretation steps
  - Wildcards (*,?) are expanded
  - Shell variables are substituted
  - Nested commands are executed
  - Redirections are enforced
  - (...)

Wildcards

- Three CLI wildcards: **?** , **[..]**, and *
- Any args with these → expanded as **list of files** (globbing)
  - ? : any single character
  - [..] : any of the characters in the brackets
  - * : any sequence of characters
- File names are looked for in the **working directory**
  - If no matching file name: error message
- Examples
  - *.java : file names ending with "java"
  - [ab]*.jpg : file names starting with 'a' or 'b' + ending w/ "jpg"

Globbing: Example

- ls *
  - ➔ list all files not starting with a dot
  - ➔ note the expansion is done by bash, not by 'ls'
- gzip [ac]*.log
  - ➔ compress any file starting by a or c and ending with .log
- rm exp?.txt
  - ➔ remove all text files starting with exp followed by 1 char
- ls "*"
  - ➔ list the file whose name is *

Reminder (Unit 1)

```c
#include <stdio.h>

int main (int argc, char* argv[]) {
   printf("Hi you! I am your first CLI command. ");
   printf("Nice to meet you.\n");
   printf("You have entered ");
   printf("%i arguments on the command line.\n", argc);
   printf("Here they are:\n");
   int i;
   for(i=0;i<argc;i++)
      printf("%s\n", argv[i]);
}
```

- Can you remember what it does?

Wildcards and commands

```
$ ls
a.out           my_first_command.c
example.sh      MyFirstCommand.java
ma_command      MyOwnShell2.java
mon_script      MyOwnShell.java
mon_script_ruby   script.rb
my_first_command
$ ./my_first_command *.java
```

- Result of `./my_first_command *.java` ? Why?

Quotes

- Bash recognizes 3 types of quotes
  - ➔ `..` → executes .. and replaces `..` by output (same as **$(..)**)
  - ➔ **".."** → one arg, no globbing (but variables are substituted)
  - ➔ **'..'** → one arg, no globbing, no variable substitution, no `..`
- Examples

```
$ echo $HOME *.c
/Users/ftaiani my_first_command.c
$ echo "$HOME *.c"
/Users/ftaiani *.c
$ echo '$HOME *.c'
$HOME *.c
$ VAR=`users` ; echo $VAR
ftaiani
```

Shell Scripting

- Often useful to store shell commands in a **shell script**
  - ➔ A special text file with list of shell commands

```
$ cat > my_first_scrit
echo Hello $USER
echo Today is `date`
$ source my_first_script
Hello ftaiani
Today is Tue Jan 27 23:43:17 CET 2015
$ sh my_first_script
Hello ftaiani
Today is Tue Jan 27 23:43:21 CET 2015
```

- '**source**' and '**sh**' are not exactly equivalent
  - ➔ Can you guess the difference?

Shell Scripting

- Avoiding sh and source → use a **shebang** !

```
#!/bin/sh
echo Hello $USER
echo Today is `date`
echo $$
```

➔ Need to make file executable to use it: chmod u+x ...

- Not limited to shell scripts! (works w/ python, perl, ...)

```
#!/usr/bin/ruby
5.times do
  puts "I love ruby"
end
```

Shell Scripting (cont.)

- Shell scripts are programs
  - ➔ They received arguments, read from sdtin, write to stdout
- Writing and reading to stdout / from stdin
  - ➔ **echo** Some Message
  - ➔ **read** SOME_VARIABLE
- Arguments: obtained through special variables
  - ➔ **$0, $1, $2, ...**: command line arguments ($0: name of script)
  - ➔ **$#** : number of args (not counting $0)
  - ➔ **$@** : all args starting from $1 (respecting spaces with **"$@"**)
  - ➔ **$*** : all args starting from $1 (causes problems with spaces)

Shell Programming

- Bash has its own full blown programming language
- We have already seen variables
  - ➔ greeting=hello; echo $greeting
- But Bash can also do **loops**

```
for i in joe mary francois
do
  echo $greeting $i
done
```

- **If** statements

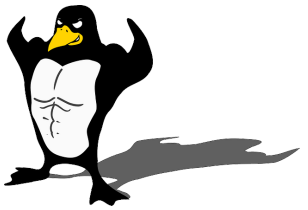
```
if [ $greetings = helloo ]
then
  echo hey
fi
```

- and a lot more…
- Used in Linux itself, so good to know about

Example: looping through args

```
#!/bin/sh
for i in "$@"
do
    echo $i
done
```

- To remember 
  - ➔ "$@" works as intended all the time, protect spaces
  - ➔ $* almost identical, but not as safe (see bonuses)



Tests

- Many tests possible
  - ➔ If files exists, are directory
  - ➔ Equality, inequalities
  - ➔ For a full list see http://www.tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

Example

```
#!/bin/sh

if [ $# -ge 2 ]
then
    echo You have at least two arguments
fi

if [ -e $1 ]
then
    echo $1 exists
else
    echo $1 does not exist
fi
```

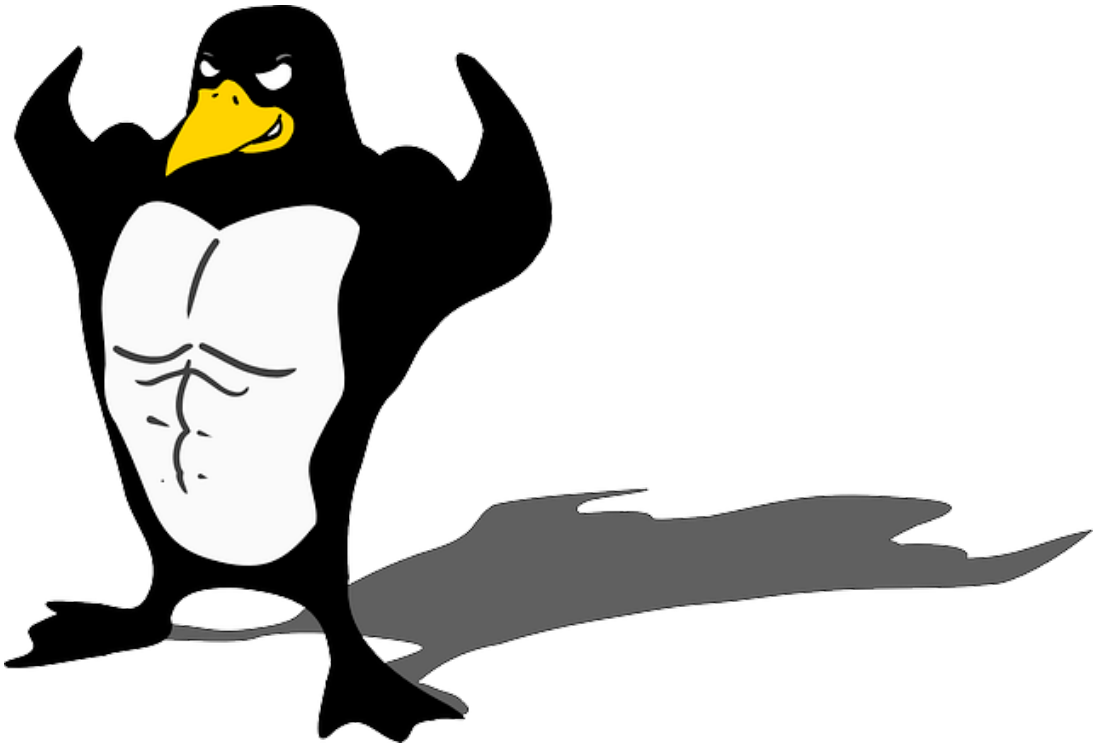
- Beware: **spaces** around [,], -ge, -e are **essential**!

Exercise

Write a script that

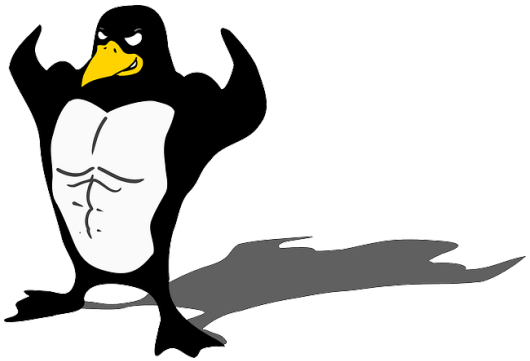
- Checks the number of args
  - ➔ Exits with message if no positional argument
- For each positional argument
  - ➔ Checks if file exists
  - ➔ Prints a corresponding message

Bonuses



Difference between $* and $@

```
#!/bin/sh
for i in   $*   ; do echo $i ; done
for i in "$*" ; do echo $i ; done
for i in "$@" ; do echo $i ; done
```



- To remember: last line
  - ➔ "$@" works as intended all the time
- Difference between "$*" and "$@"
  - ➔ Highlights one weakness of shell programming: spaces!
  - ➔ Very hard to manipulate arguments with internal spaces
  - ➔ Advice: always use "$@" (in quotes!)

Playing with PATH

- Create a bash script call my_ls
  - ➔ don’t forget chmod u+x to make executable
- Compare ./my_ls and my_ls as a command
  - ➔ my_ls does not work
- Create a 'bin' directory in your home folder
  - ➔ move my_ls there. Invoking my_ls still does not work
- add your ~/bin directory to $PATH
  - ➔ be sure to keep old value of $PATH and to use export
  - ➔ export PATH=~/BIN:$PATH
- my_ls now works from anywhere
  - ➔ “which my_ls” locates it

Dangers in the wrong $PATH

- A wrongly set $PATH can make a system unusable
  - ➔ $PATH not only used by shells, but by many scripts and programming environments
- Example
  - ➔ in ~/bin create a symbolic link called “ls” to my_ls
  - ➔ ln -s ./my_ls ls
  - ➔ Try calling ls
  - ➔ Try calling which ls, then which -a ls
- Important if several version of same executables
  - ➔ e.g. interpreters: perl, perl5.12.4, java, ruby