Session 3 - Unix shell

Cygwin

> "Get that Linux feeling - on Windows"

HowTo: Installation steps Mirrors: https://www.cygwin.com/mirrors.html

Cygwin - What is it?

- a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows.
- a DLL (cygwin1.dll) which provides substantial POSIX API functionality.

Cygwin - What it is *not*?

- a way to run native Linux apps on Windows.
- a way to magically make native Windows apps aware of UNIX® functionality like signals, ptys, etc.

You must rebuild your application from source if you want it to run on Windows or for Windows applications to take advantage of Cygwin functionality.

This is copied from the Cygwin website.

Why Cygwin?

Because it provides us a proper Unix shell environment in a Windows world.

Hä? Unix?

But it's a Linux/MacOS shell...

- The term "shell" may refer to any CLI on different OSs.
- The generic term is "Unix shell", since this is the common ancestor.
- Common behavior/language elements on "unixoid/*nix" OSs.

BASH?

There are different shell options on "*nix-systems":

- sh: Bourne Shell - 1979
- bash: Bourne-again Shell - 1989
- zsh: Z shell - 1990
- ...

Among GNU/Linux distros, BASH is currently often the default.

More shell fun!

- ~ = /home/$USER
- **cat** = output file contents
- **cd -** = Return to previous folder
- **&&, ||** = and/or: Combine commands
- **man CMD** = Show manual page of CMD

Make ourselves at home

```
$ mkdir ~/videos
$ cd ~
$ ls
$ cd vid[TAB]
```

Cygwin & FFmpeg

- On unix-like systems, just call "ffmpeg".
- Cygwin: `cmd /c c:/ffmpeg/bin/ffmpeg.exe`
- Or (better): Add FFmpeg path to %PATH% environment variable.

For convenience in scripts in Cygwin that call FFmpeg, simply do:

```
FFMPEG="cmd /c c:/ffmpeg/bin/ffmpeg.exe"
$FFMPEG ...
```

Or in order to be able to start FFmpeg from anywhere, add it's "bin" folder to the %PATH% environment variable:

- Control Panel > System > Advanced System Settings
- Advanced (Tab) > Environment Variables (Button)
- System variables: Edit "PATH"
- Add absolute path where FFmpeg/bin is (separated by ";"). So it looks like this (example): `%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;%SYSTEMROOT%\System32\WindowsPowerSh`

Streams and redirection

- <, >, | = Output redirection ("piping")
- >> = append.
- tee = Redirect *and* output.

More: BASH Shell Redirect (nixCraft)

Streams and redirection

```
$ ls -la ~/*.* | less
$ ls -la ~/*.* > ~/videos/dirlist.txt
```

grep: A new best friend.

(for filtering)

```
$ cat /etc/services
$ cat /etc/services | grep http
```

More: Guru99 tutorial - Grep

Other good friends:

- sort:
  Sort contents of a text file line by line.
- uniq:
  Filters out repeated lines.
- cut:
  Removes sections of each line of a text.
- md5sum:
  Calculate (and display) MD5 hash.

Fun with pipes and filters

```
$ for i in a b c; do echo "$i" > $i.txt; done
$ cp a.txt d.txt
$ md5sum *.txt | tee MD5SUMS.md5
$ cat MD5SUMS.md5 | sort
$ cat MD5SUMS.md5 | cut -d ' ' -f 1
$ cat MD5SUMS.md5 | cut -d ' ' -f 1 | sort | uniq
```

FFprobe: Feature extraction

```
$ ffprobe -show_streams MYVIDEO.MOV > ~/video/ffprobe.txt
$ cat ~/video/ffprobe.txt | grep "TAG:timecode" | uniq | cut -d '=' -f 2
```

Back to coding :)

Set Execute Permission

On unix-like systems, there are 3 basic access rights:

> rwx = read, write, execute

In order for a shell script to become executable, set the “x” bit:

> $ chmod +x myscript.sh

More: File permissions and attributes (ArchLinux Wiki)

Shebang!

Every text script must declare its “program loader”:

```
#!/bin/bash
```

See: Starting Off With a Sha-Bang

Windows needs the file suffix to determine if a file is considered executable and what to use to run it.

Unix-like systems don’t depend on file suffixes like that. They usually look inside the file to determine its type. Since a shell script is nothing but a text file, it therefore needs some means of identifying its interpreter.

That’s what the “shebang” is for.

hello.sh

```
#!/bin/sh
NAME="$1"
echo "Hello $NAME. :)"
```

Batch vs BASH

Some similar, some different:

bat	sh
set “NAME=value”	NAME=“value”
echo Hello %NAME%!	echo “Hello $NAME!”
if “%NAME%”==“” GOTO Label	if [-z “$NAME”]; then ... fi
for %%I in (1 2 3) DO	for $I in 1 2 3; do ... done
%1 %2 %3 %*	$1 $2 $3 $*

More: MS-DOS vs Linux/Unix (computerhope.com)

FOR: Files in folder

Batch script:

```
@echo off
FOR /R "C:\Videos\" %%F IN (*.avi) DO (
    echo %%~nF
)
pause
```

Bash script:

```
for FILE in ~/videos/*.avi; do
  ...
done
```

The “;” before the “do” is important.

FOR: Iterate through parameters

```
for ARG in $*; do
  echo "Param: $ARG"
done
```

New! - Functions

```
function MyFunction {
  echo "Do stuff here."
  echo "My 1st parameter: $1"
  local VAR1="whoopie!"
  return 1
}
```

Exit status

The variable "$?" contains the numeric exit value of the previous execution (of command or function).

```
# Outputs "1", because source file not found:
$ ffmpeg -i x
$ echo $?

# Outputs "0", because all was okay:
$ ffmpeg -version
$ echo $?
```

Example: Rsync exit values

Applications (should) return 0 when they executed successfully. Non-zero means "something happened that you might want to know/check"

What non-zero codes mean depends on the application and is not standardized. Good practice is to provide a list of the exit values they use. Unfortunately, that's not always the case.

Tests / Conditionals

```
if [ $EXITSTATUS -ne 0 ]; then
  echo "ERROR: Unsuccessful execution ($EXITSTATUS)"
  exit 2
fi
```

Tests / Conditionals

```
# Check if variable matches string:
if [ $NAME == "Dragon" ]; then ...; fi

# Check if variable is empty:
if [ -z $NAME ]; then ...; fi

# Check if parameter is a folder:
if [ -d $SOURCE ]; then ...; fi

# Check if value is greater-than 1:
if [ $VALUE -gt 1 ]; then ...; fi
```

The space around the "[]" brackets is important. Without the spaces the interpreter will throw a syntax error.

Comparison operators (numeric)

- -eq : equal to
- -ne : not equal to
- -gt : greater than
- -ge : greater or equal
- -lt : less than
- ...

Comparison operators (string)

- =/== : string equal to
- != : string not equal to

- -z : string is NULL
- -n : string is not NULL
- ...

More: Other comparison operators (tldp.org)

File test operators

- -e: file exists
- -f: is a file (not a folder or device)
- -d: is a directory
- -s: is *not* zero size
- ...

More: File test operators (tldp.org)

Bash "Tests"

```
if [ -z "$NAME ]; then
  echo "ERROR: Name is empty"
  exit 1
fi
echo "Hello $NAME"
```

Bash "Tests"

```
function CheckDir {
  local DIR=$1
  if [ ! -d "$DIR" ]; then
    echo "ERROR: Invalid folder '$DIR'."
    return 1
  fi
}
```

- local: This declares the variable "$DIR", but makes it only available inside the CheckDir function. So it doesn't collide with global variables.

String operations (BASH)

```
${#string}                  Length of $string

${string:position}          Extract substring from $string at $position
${string:position:length}   Extract $length characters substring from $string at $position

${string#substring}         Strip shortest match of $substring from front of $string
${string##substring}        Strip longest match of $substring from front of $string
${string%substring}         Strip shortest match of $substring from back of $string
${string%%substring}        Strip longest match of $substring from back of $string

${string/substring/replacement}     Replace first match of $substring with $replacement
${string//substring/replacement}    Replace all matches of $substring with $replacement
${string/#substring/replacement}    If $substring matches front end of $string, substitute $replacement
${string/%substring/replacement}    If $substring matches back end of $string, substitute $replacement
```

See "Advanced Bash-Scripting Guide (tldp.org)" for additional information.

Links (.sh)

- Bash Reference Manual (gnu.org)
- Advanced Bash-Scripting Guide (tldp.org)
- Bash Scripting Tutorial for Beginners (linuxconfig.org)

Homework

- Translate the “ffmpeg_helper.bat” from Batch to Bash.
- Think each step in the script through and consider where error checks may be necessary/possible - and implement error handling routines.
- Popular cases are:
  - Does source file/folder exist?
  - Is file a file? Is folder a folder?
  - Did ffmpeg run successfully?