Scripting
Tools, languages and the Shell

intERLab at AIT

Network Management Workshop

March 11-15 – Bangkok, Thailand

Hervey Allen

Languages

Interpreted

bash/sh, Perl, PHP, Python, Ruby

Compiled

C, C++, Java

Tools

- sed: Stream EDitor
- awk: Pattern scanning & processing
- dc: Arbitrary precision calculator
- bc: Arbitrary precision calculator language.
- tr: Translate or delete characters
- grep: Print lines matching a pattern

The Shell (bash)

- export
- printenv
- ~/.bashrc
- ~/.profile
- /etc/profile
- /etc/bash.bashrc
- /etc/skel (in Ubuntu)

The Shell cont.

By default, on Ubuntu, we use the Bourne Again SHell, or BASH.

The shell is your interface to the kernel.

Programs use the shell to determine their environment.

Your shell environment can be customized. We'll go over how it's configured in Ubuntu.

The Shell cont.

Flow of bash execution at startup:
Interactive login

1. /etc/profile
2. ~/.profile
3. ~/.bashrc

This is true for Ubuntu. See "`man bash`" for standard startup sequence.

We'll go through this now...

The Shell cont.

Flow of bash execution at startup:

Interactive shell, no login

1. /etc/bash.bashrc
2. ~/.bashrc

Non-interactive startup, run a script

1. Script looks for variable BASH_ENV
2. If it is set and is a file bash does this:

```
if [ -n "$BASH_ENV" ]; then . "$BASH_ENV"; fi
```

Using the Shell
Command Line Interpreter or CLI

To understand scripts let's practice a bit with the CLI. At the shell prompt try:

```
# cd; echo "Hello, World" > test.txt;
  cp test.txt test.txt.bak; vi test.txt
```

The above is all on one line.

What happened?

Now try this:

```
# cp test.txt $( date +%F )test.txt
```

Using the Shell
In a file

Now create a new file and place the some of our previous command in it:

```
# cd
# vi newscript
   echo “Hello world” > hello.txt
   cp hello.txt hello.txt.bak
   cat hello.txt+hello.txt.bak > new.txt
   cat new.txt
:wq
```

Using the Shell
In a file

Now we can execute those commands in the order in which they appear in the file by doing this:

```
# bash newscript
# sh newscript
# . newscript
```

Using the Shell
As a shell script

Now we can take the last step and start to create self-contained scripts that run on their own.

We'll need to do two things:

1. Specify the CLI to use, and
2. Make our file executable

The “Shebang”

To specify that a file is to become a shell script you specify the interpreter like this at the very start of the file:

#!/bin/sh

or

#!/bin/bash

etc...

This is known as the “Shebang” (#!).

What's Next?

Now let's create a very simple shell script. This will simply echo back what you enter on the command line:

```
#!/bin/sh
echo $1
```

Enter this in a file new.sh, then do:

```
# chmod 755 new.sh
```

Run your script

To run the script do:

```
# ./new.sh text
# ./new.sh “text with spaces...”
```

Try updating the script to echo two, or more separated items on the command line.

When Not to Use

A long'ish list...

Resource-intensive tasks, especially where speed is a factor (sorting, hashing, etc.)

Procedures involving heavy-duty math operations, especially floating point arithmetic arbitrary precision calculations, or complex numbers

Cross-platform portability required

Complex applications, where structured programming is a necessity (need type-checking of variables, function prototypes, etc.)

Project consists of subcomponents with interlocking dependencies

Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion)

Need native support for multi-dimensional arrays or data structures, such as linked lists or trees

Need to generate or manipulate graphics or GUIs

Need direct access to system hardware or port or socket I/O

Resources

The books you received in class:

Classic Shell Scripting

The Bash Cookbook

The on-line *Advanced Bash Scripting Guide*, available at:

http://www.tldp.org/LDP/abs/html/