HP World/Interex 2002

Linux BASH ShellProgramming

Chris Cooper
(734) 805-2172
chris_cooper@hp.com

George Vish II
(404) 648-6403
george_vish@hp.com

hp education services
education.hp.com

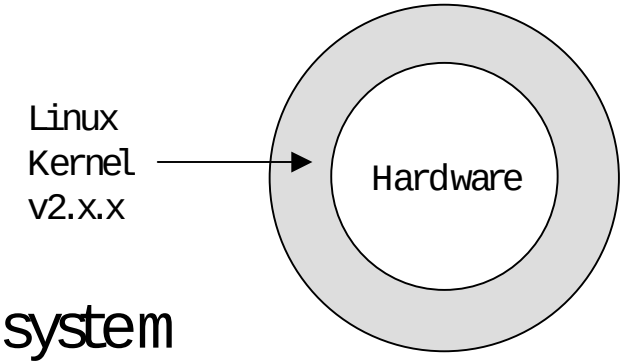
BASH
the Bourne Again Shell
A brief overview

Version A.00

U2794S Module 8 Slides

The Linux Kernel

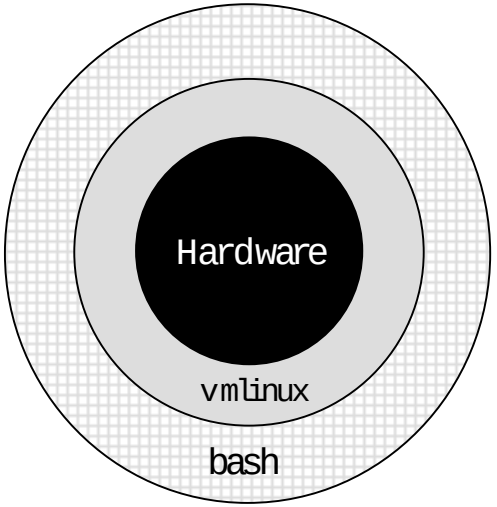
- The kernel interacts with low-level system features.
- The kernel is responsible for
  - device drivers
  - memory management
  - CPU scheduling
  - implementation of the file system



- The kernel *is* Linux. This is the only part of the system controlled by Linus Torvalds.

The Shell

- The shell acts as an intermediary between the user and the kernel.
- The shell interprets user commands and passes them to the kernel.
- Many shells are available for Linux; the default is the Bourne-Again SHell, **bash**.



Available Shells

There are five major shell environments, although others are available:

sh	The Bourne shell is the the original UNIX shell, written by Steven Bourne.
csh	The C (*California*) shell is a shell with syntax similar to the C programming language.
ksh	The Korn shell is the most widely used commercial Unix shell, derived largely from the Bourne shell.
POSIX sh	A standardized version of the Korn shell (and the default shell for the HP-UX operating system release since 10.x)
bash	The Bourne-again shell is the default for Linux, GNU alternative to the Korn shell, with extended features.

Using the **bash** Shell

- The shell is a non-graphical environment providing a command line interpreter as well as a script execution environment.
- Shell commands are entered at the command prompt, which can be accessed through a non-graphical login, or a graphical terminal emulator, such as an X-term or the Gnome terminal.
- The shell has a set of built-in commands with terse syntax.
- Extra commands may be written and utilized like built-in shell commands.

Variables

- Values can be stored by the shell in variables.
- Variables can be set by
  - assigning a value directly

    ```
    # myvar=hello
    ```

  - assigning the output of a command to a variable

    (command expansion)

    ```
    # myvar='ls'   or   myvar=$(ls -a)
    ```

- The value stored in a variable can be accessed with the dollar symbol ($).

  ```
  $myvar
  ```

- The **echo** command can be used to display text, including the contents of variables.

  ```
  # echo $myvar
  ```

Using Variables

- Variables in the shell are not strongly typed. For example, a variable containing the value 23 may be either be treated as the number 23 or as the string "23".

- To perform numeric addition:

```
variable3=$((variable1 + variable2))
```

- To perform string concatenation:

```
variable1=$variable2$variable3
```

Environment Variables

- The shell maintains a set of variables that can be accessed by the shell and by programs that run in it. These variables are part of the shell environment and are called environment variables.

- Currently set environment variables can be viewed with the **env** command.

- To modify an environment variable or add a new variable to the environment, it must be exported.

```
# PATH=$PATH:/usr/newbin
# export PATH
```

Streams (I/O redirection)

- Input from and output to the terminal is handled through an I/O stream.
  Streams are treated like files by the shell.
- The default input and output streams are

stdin	The standard input stream	(file descriptor 0)
stdout	The standard output stream	(file descriptor 1)
stderr	Error output from commands	(file descriptor 2)

- The output from or input to commands can be redirected to other commands or files.
  - **>** *filename* Redirect to a file.
  - **>>** *filename* Append to a file.
  - **<** *filename* Redirect file contents to a command.
  - **|** Pipe command output to the input of another command.
- The pipe (}) is a very powerful tool, as it allows the creation of sophisticated commands from the basic building blocks. For example:

```
# head -1000 myfile | tail -150 | more
```

- The **2>** redirection operator can be used to direct command errors to a different location, such as an error log file, or to discard error messages completely.

```
# some-command 2> errors
# some-other-command 2> /dev/null
```

Aliases

- Aliases can be used as shortcuts for commonly used complex commands. The shortened alias is replaced with its full text when it appears at the start of a command line.
  - Current aliases may be displayed by entering: **`# alias`**
  - To create an alias: **`# alias more=more`**
  - An alias may be cancelled with: **`# unalias <alias-name>`**
- RedHat Linux aliases **`rm`**, **`cp`**, and **`mv`** to their interactive modes for root, always to prompting when a file is to be destroyed.
- Local user aliases may be defined in the **`$HOME/.bashrc file`** (Globally used aliases may be defined in **`/etc/bashrc`**)

Process Control

- When a command is run, it may start several processes. All the currently running processes can be viewed with the **ps** (or the **bash** shell built in **jobs** command).

- Processes can run in the foreground or background:
  - **&** Run the command in the background.
  - **^C** Terminate the current foreground process.
  - **^Z** Suspend the current foreground process.
  - **bg** Move suspended foreground processes to the background.
  - **fg** Move the background jobs into the foreground.

Resource Files

- Every user has a set of resource scripts in their home directory that allows per-user configuration of the Linux environment.
- User resource files are located in the home directory. The names of these files usually begins with a period (.) character.

.bash_profile	Executes once at login.
.bashrc	Executes every time a shell is started.
.bash_logout	Executes when a shell is closed.
.bash_history	A list of recently entered commands.

Shell Scripts

- Shell scripts are widely used in the UNIX environment to:
  - Maintain and monitor the system
  - Simplify complex processes
  - Grant restricted permissions to ordinary users
  - Configure the user login procedure and environment

- Shell scripts use the same syntax as the command line and can call any executable program, including shell commands, scripts, and applications.

Writing Shell Scripts

- Shell scripts should begin with a line that defines which interpreter to use for the script.

  `#!/bin/bash` (this must be on the very first line of the file)
- Comments are added with the hash (#) character. Everything to the right of the # is ignored.
- Commands are run in order from the top to the bottom of the file.
- Shell scripts include commands for flow control allowing conditional execution and looping.
- The shell is not a high performance programming language, it is intended as a glue for the built in shell commands and other user or third-party created applications.

Conditional

- Commands contained within an **if** statement will only be executed if the condition is met.

```
if [ condition ]
then
   ..
elif [ condition ]
then
   ..
fi
```

- Case statements are a convenient way of expressing actions to handle a list of possible variable values.

Comparing Variables

- Two sets of comparison operators exist for comparing strings or numbers.

String Test	Numerical Test	
=	-eq	Equals
!=	-ne	Not equal to
>	-gt	Greater than
<	-lt	Less than
>=	-ge	Greater than or equal to
<=	-le	Less than or equal to

File Testing

- Several conditions exist to test the status of files.

-e *<filename>*	File exists
-d *<filename>*	File is a directory
-f *<filename>*	File is an ordinary file
-w *<filename>*	File is writable
-x *<filename>*	File is executable

Loops

- **`while`** loops repeat as long as the condition is true.

```
while [ condition ]
do
   ..
done
```

- **`for`** loops repeat for every value in a list.

```
for instance in $list
do
   ..
done
```

Reading User Input

- User input is collected one line at a time using the **read** command.

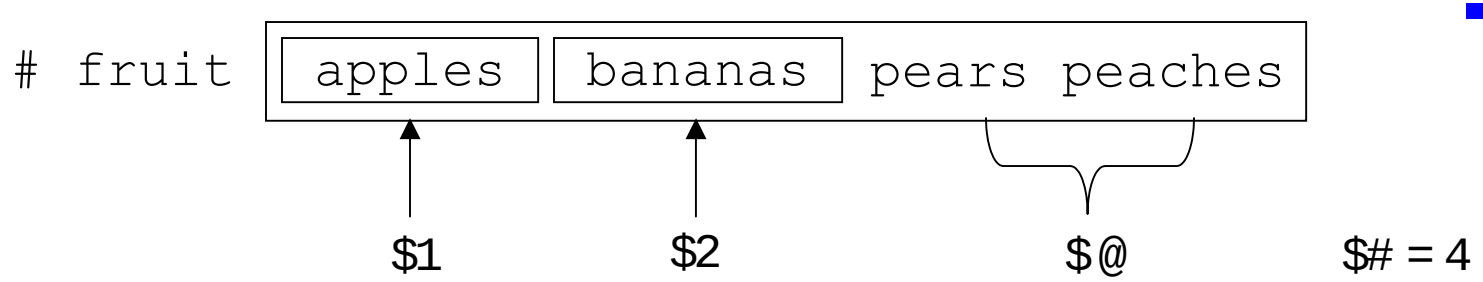
```
read line
```

- Files can also be read a line at a time using a combination of the **while** and **read** commands.

```
while read line
do
   ..
Done < $file
```

- When the end of file marker is reached, **read** will return false.
  If user input spans multiple lines, it can be terminated with the Ctrl+ d character.

Processing Arguments



- The **`shift`** command moves all arguments to the left, discarding the first argument.

- The **`getopts`** command is useful for processing UNIX style command flags.

Scripting Tips

- Always attempt to re-use existing programs and scripts whenever possible.
- Do not try to improve on the efficiency of shell commands.
- Try commands on the command line first.
- When first writing scripts, use the **echo** command to display the contents of variables instead of running commands that will modify the system state.
- When tracking errors, use **echo** to track the flow of execution and **#** to deactivate suspect portions of code.
- Solve interesting problems. Practice and have fun!

Managing Input and Output

Version A.02

H4322S Module 13 Slides

Managing Input/Output

This module covers methods of handling input and output in shell script programs.

The shell provides several methods to direct the flow of data into and out of command programs:

- Redirection of input `command < data_file`
- Redirection of output `command > data_file`
- Redirection of input text `command << end_file_marker`
- Piping of output `command | next_command`
- Redirection of errors `command 2> error_file`
- Run in background `command &`

Data-Flow File Descriptors 3-9

Standard file descriptors: ***stdin*** **(0)** ***stdout*** **(1)** ***stderr*** **(2)**

File descriptors **3** to **9** can be used for input or output

Syntax:

- Open file for reading **`exec 3< data_file`**
- Open file for write/overwrite **`exec 3> data_file`**
- Open file for write/append **`exec 3>> data file`**
- Redirection of output **`command >&3`**
- Close input file descriptor **`exec 3<&-`**
- Close output file descriptor **`exec 3>&-`**

Reading/Writing Using File Descriptors

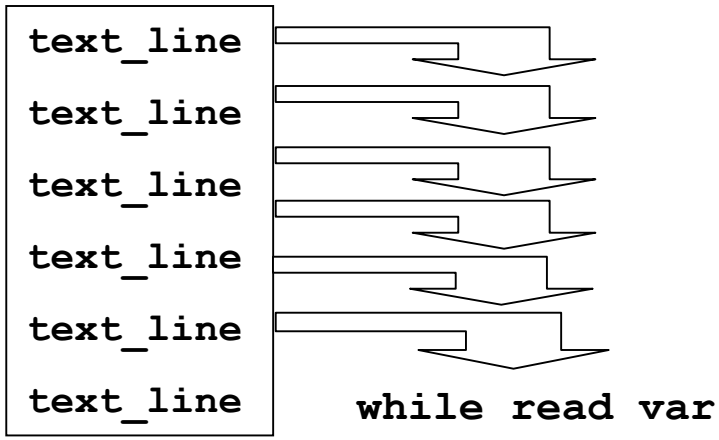
`exec 3> /tmp/output.txt`	Open
`print -u3 "some data"`	Use(write)
`exec 3>&-`	Close

`exec 3< /tmp/output.txt`	Open
`read -u3 var`	Use(read)
`exec 3<&-`	Close

Passing File Contents to a Script

```
command | script
```

```
script < data_file
```



Redirecting to a Loop within a Script

Piping Data to a Loop

```
command | while read var
do
...
command_set
...
done
```

Data is piped into the loop.

The command must create **stdout**.

Redirecting Data to a Loop

```
while read var
do
...
command_set
...
done < data_file
```

Data is redirected into the loop following the word **done**.

Creating Parameter Lists from Input Lines

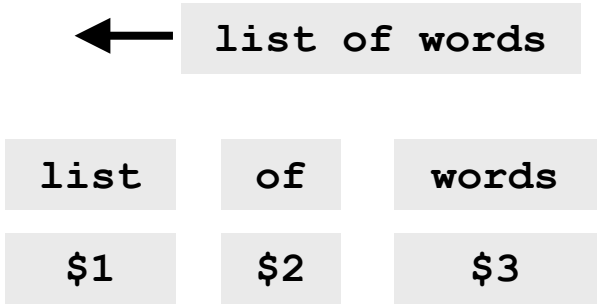
The **set** command is used to generate a parameter list.

Input lines are, conventionally, read into a variable called **line**.

```
Input line: list of words

while read line

set $line
```



Using a **here** Document