PennState

CMPSC 311 - Introduction to Systems Programming

Shell Programming

Professor

Suman Saha

(Slides are mostly by *Professor Patrick McDaniel* and *Professor Abutalib Aghayev*)

Shell programming

PennState

- aka "shell scripting," "Bash scripting"
- What is it?
  - Series of commands
  - Programming with
- What for
  - Automating
  - System administration
  - Prototyping

A sample script: `shello`

PennState

- First line: interpreter
  - The `#!` is important!
- Comment: `#` to end-of-line
- Give the file execut
  ```
  chmod +x shello
  ```
  - Not determined by file extension
  - Typical: `.sh` or none
- Run it
  ```
  ./shello
  ```

```
#! /bin/bash
# Greetings!
ello    world

ariable
lo "$USER"

# Set a variable
greetz=Shellutations
echo "$greetz world"
```

Shell variables

PennState

- Setting/unsetting
  - `export var=value`
  - No spaces!
  - `unset var`
- Using the value
  - `$var`
- Untyped by default
  - Behave like strings
  - (See `help declare` for more)

Special variables

PennState

- Change shell behavior or give you information
  - `PWD`: current directory
  - `USER`: name of the current user
  - `HOME`: the current user's ho
    - Can usually be abbr
  - `PATH`: where to search for executables
  - `PS1`: Bash prompt (will see later)

Exercise

PennState

- Make a directory `~/bin`
- Move `shello` script there
- Prepend the direct
  - `PATH=~/bin:$PAT`
- Change to home dir
- Run by typing `shello`

Shell initialization

PennState

- Set custom variables
- At startup, bash runs shell commands from `~/.bashrc`
  - Just a shell script
- This script can do

Fun with prompts

PennState

- Main prompt: `$PS1`
- Special values
  - \u: username
  - \h: hostname
  - \w: working dir
  - \e: escape (for colors)
  - many others
- This is something you can set in your `.bashrc`

Very detailed treatment: http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/

Special characters

PennState

```
echo Penn State is #1
echo Micro$oft Windows
echo Steins;Gate
```


- What happened?
- Many special characters
  - Whitespace
  - #$*&^?!~'`"\{}[]<>()|;
- What if we want these characters?

Quoting

PennState

- Removes specialness
- Hard quotes: '...'
  - Quote everything except closing '
- Soft quotes: "..."
  - Allow variables (an
  - Good practice: "$var"
- Backslash (escaping)
  - Quotes next character

Arguments

PennState

- In C: `argc` and `argv[]`
- Split at whitespace
  - How can we override this?
- Arguments to a scr
  - `./script foo bar`
  - `$#` is the same as argc
  - `"$@"`: all args
  - `"$1"` to `"$9"`: individual

Debug mode

PennState

- Shows each command
  - Variables expanded
  - Arguments quoted
- Run with `bash -x`
  - Temporary – just f
  - `bash -x shello`
  - Use `-xv` for even more info

Exercises

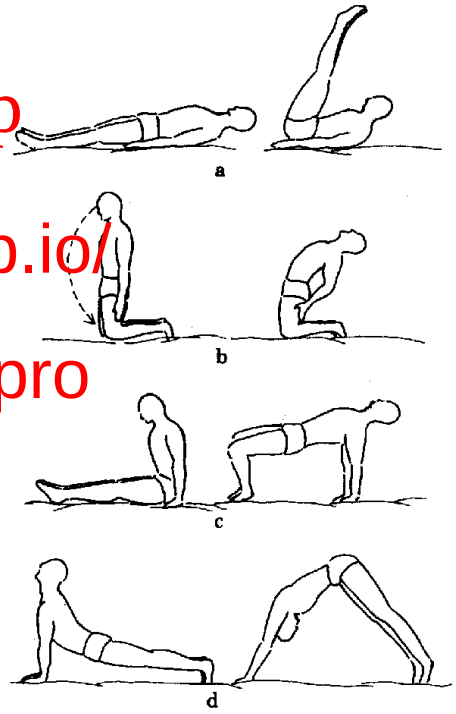
PennState

- Make a script that:
  - Prints its first argument doubled

```
./script1 foo
foofoo
```

  - Prints its first four

```
./script2 "foo bar" baz
[foo bar]
[baz]
[]
[]
```

Redirecting input/output

PennState

- Assigns stdin and/or stdout to a file
- `echo hello > world`
- `echo hello >> world`
- `tr h j < world`

Pipelines

PennState

- Connect stdout of one command to stdin of the next

```
echo hello | tr h j
... | rev
... | hexdump -C
... | grep 06
```

- UNIX philosophy at w

Some references

PennState

- Advanced Bash-Scripting Guide
  - http://tldp.org/LDP/abs/html/
  - A great reference from beginner to advanced
- commandlinefu.co
  - Lots of gems, some
  - Fun to figure out how they work
- Bash man page
  - `man bash`

How to kill a process

PennState

- Today we're learning some loops
- If it starts to run away, Ctrl-C is your friend
  - Sends a signal that ends the process
    - More on signals lat
  - Works on many different progra command line
  - Displayed as ^C

Return from `main`

PennState

- In C, the main function always returns an `int`
  - Used as an error code for the entire process
  - Same convention as any other C function
    - Zero: success
    - Nonzero: failure, e
- Also known as the *exit status* of the

Status sample program

PennState

```
$ ./status 0
$ echo $?

$ ./status 2
$ echo $?

$ ! ./status 2
$ echo $?

$ ./status -1
$ echo $?
```


Custom prompt for today

PennState

- You can include `$?` in your prompt
- Now try:


```
./statu
```

Exit status in scripts

PennState

- `$?`: get exit status of the previous command
- The exit status of a script comes from the last command it runs
  - Or use the `exit` builtin to exit early, e.g. `exit 1`
- `! cmd` reverses the success
  - Exactly like the ! (“

Conditionals

PennState

- Exit status is used as the test for `if` statements:

```
if list; then
  cmds
fi
```


- Runs *list*, and if the *exit* then *cmds* is executed
- There are also `elif` and `else` commands that work the same way.

Test commands

PennState

- Builtin commands that test handy conditions
- `true`: always succeeds
- `false`: always fails
- Many other conditi
  - Returns 0 if test is true, 1 otherwise
  - Full list: `help test`

What do these do?

PennState

```
$ test -e status.c
$ test -e asdf

$ test -d status.c
$ test -d /etc

$ test 10 -gt 5
$ test 10 -lt 10
$ test 10 -le 10
$ test 12 -ge 15
```

Useful tests

PennState

- `test -e file`
  - True if file exists
- `test -d dir`
  - True if dir exists an
- `test -z "$var"`
  - True if var is empty (zero-length)
- `test -n "$var"`
  - True if var is nonempty
- `test str1 = str2`
- `test num1 -gt num2`
  - or `-lt`, `-ge`, `-le`, `-eq`, `-ne`

Shorthand tests

PennState

- Shorthand test: `[[ … ]]`
  - Workalike for `test`
- For example:

```
age=20
test $age -ge 16
  echo can drive
[[ $age -ge 16 ]] &&
  echo can drive
```

- Now say age=3 and try again

Command lists

PennState

- Simple command list: `;`
  - Runs each command regardless of exit status
  - Example:

```
do_this;
```

- Shortcutting comm
  - `&&` stops after failure
  - `||` stops after success
  - Examples:

```
foo && echo success
bar || echo failed
```

Try it out

PennState

```
true && echo one
true || echo two
false && echo three
false || echo four
test -e Makefile &&
cat dog || echo bird
./status 4 && echo 4
./status 0 && echo 0
cat dog; cat status.c
touch status.c; make
make clean && make
```

Conditional loops

PennState

- You can write a `while` loop using the same idea:

```
while list; do
   cmds
done
```

- Runs *list*, *cmds*, *list*, *cmd* long as *list* succeeds (exit status 0)
- Similarly, an `until` loop will execute as long as *list* fails

Conditional practice

PennState

```
if ! [[ -e foo ]]; then
  echo hello > foo
fi

while [[ "$x" -lt 99999 ]]; do
  echo "$x"
  x="1$x"
done

if cat foo; then
  echo Same to you
fi

if cat dog; then
  echo Woof
fi
```

Command substitution

PennState

- Command substitution allows the output of a command to replace the command itself
  - In recent versions of bash: $(command)
  - More commonly: `com
- file $(which ls)
- file `which ls`

For statement

PennState

- The `for` loop is "for-each" style:

```
for var in words; do
   cmds
done
```


- The *cmds* are execute each argument in *word* set to that argument

- for i in $(seq 1 5); do

  echo $i;

  done

- for i in {1..5}; do echo $i; done

For example...

PennState

```
for a in A B C hello 4; do
  echo "$a$a$a"
done

for ext in h c; do
  cat "hello.$ext"
done
```

Globbing

PennState

- Old name for *filename wildcards*
  - (Comes from "global command")
- * means any number of characters:

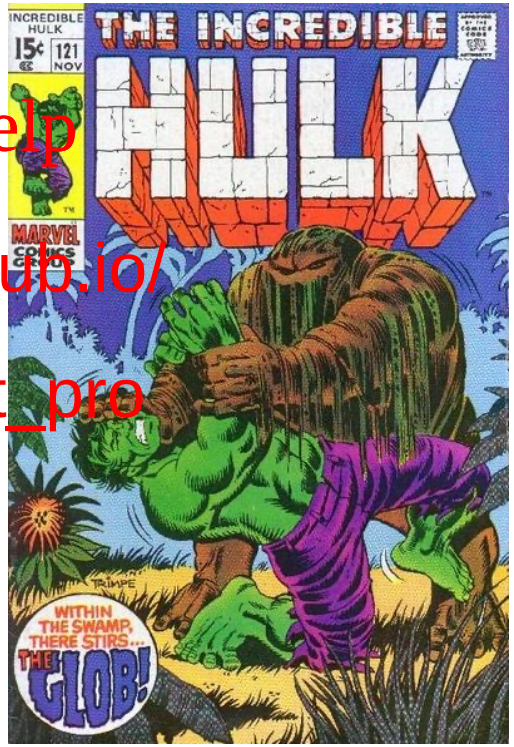
```
echo *
echo *.c
```

- ? means any one character:

```
echo hello.?
```

- Bulk rename:

```
for f in hello.*; do
  mv "$f" "$f.bak"
done
```

Some more useful tools

PennState

- `touch` *foo*: "modify" the file *foo* without really changing it
- `sleep` *t*: wait for *t* seconds
- `grep -F` *string*: *string*
- `find . -name '` directory
  - Many other things you can search for; d
- `file` *foo*: determine what kind of file *foo* is
- `wc`: counts words/characters/lines from stdin
- `bc`: command line calculator

Exercise time

PennState

- Print out "foo" once per second until ^C'd
- Find all the .png files in dir/
- Find all the files in dir which are *actually* PNG graphics
- Use a pipe and bc to calculate the product of 199 and 42