

CS 241: Systems Programming

Lecture 6. Shell Scripting 1

Spring 2024

Prof. Stephen Checkoway

Permissions

Every user has an id (uid), a group id (gid) and belongs to a set of groups

Every file has an **owner**, a **group**, and a set of **permissions**

```
steve@clyde:~$ id
uid=1425750506(steve) gid=1425750506(steve) groups=1425750506(steve),1425700508(faculty)
steve@clyde:~$ ls -ld /home
drwxr-xr-x 4 root root 4096 Aug 13 2013 /home
steve@clyde:~$ ls -ld ~
drwxr-x--x 30 steve faculty 50 Sep 2 11:31 /usr/users/noquota/faculty/steve
steve@clyde:~$ ls -l hello.py
-rwx----- 1 steve steve 100 Aug 31 14:31 hello.py
```

First letter of permissions says what type of file it is: - is file, d is directory

Permissions

Permissions

The next 9 letters `rwXrwXrwX` control who has what type of access

- ▶ `owner`
- ▶ `group`
- ▶ `other (everyone else)`

Each group of 3 determines what access the corresponding users have

Permissions

The next 9 letters `rwXrwXrwX` control who has what type of access

- ▶ `owner`
- ▶ `group`
- ▶ `other (everyone else)`

Each group of 3 determines what access the corresponding users have

- ▶ `Files`
 - ▶ `r` — the owner/group/other can read the file
 - ▶ `w` — the owner/group/other can write the file
 - ▶ `x` — the owner/group/other can execute the file (run it as a program)

Permissions

The next 9 letters `rwXrwXrwX` control who has what type of access

- ▶ owner
- ▶ group
- ▶ other (everyone else)

Each group of 3 determines what access the corresponding users have

- ▶ Files
 - ▶ r — the owner/group/other can read the file
 - ▶ w — the owner/group/other can write the file
 - ▶ x — the owner/group/other can execute the file (run it as a program)
- ▶ Directories
 - ▶ r — the owner/group/other can see which files are in the directory
 - ▶ w — the owner/group/other can add/delete files in the directory
 - ▶ x — the owner/group/other can access files in the directory

Permissions example

Permissions example

```
-rw-r--r-- 1 steve steve 0 Sep  3 14:25 foo
```

The owner (steve) can read and write foo, everyone else can read it

Permissions example

```
-rw-r--r-- 1 steve steve 0 Sep  3 14:25 foo
```

The owner (steve) can read and write foo, everyone else can read it

```
-rwx----- 1 steve steve 100 Aug 31 14:31 hello.py
```

The owner can read, write, or execute, everyone else can do nothing

Permissions example

```
-rw-r--r-- 1 steve steve 0 Sep  3 14:25 foo
```

The owner (steve) can read and write foo, everyone else can read it

```
-rwx----- 1 steve steve 100 Aug 31 14:31 hello.py
```

The owner can read, write, or execute, everyone else can do nothing

```
drwxr-x--x 33 steve faculty 54 Sep  3 14:25 .
```

```
drwxrwxr-x 2 steve faculty 4 Sep  2 11:45 books/
```

steve and all faculty have full access to ./books, everyone else can see the directory contents

Changing owner/group/perms

Handy shell commands

- ▶ `chown` — Change owner (and group) of files/directories
- ▶ `chgrp` — Change group of files/directories
- ▶ `chmod` — Change permissions for files/directories

Permissions are often specified numerically in octal (base 8)

- ▶ 0 = --- 4 = r--
- ▶ 1 = --x 5 = r-x
- ▶ 2 = -w- 6 = rw-
- ▶ 3 = -wx 7 = rwx

Common values 755 (rwxr-xr-x) and 644 (rw-r--r--)

After running `ls -l` we see the line

```
drwxr-x--- 6 steve faculty 14 Dec 18 15:59 hw6-solutions
```

What of the following statements is **false**?

- A. `hw6-solutions` is a directory
- B. User `steve` is the only one who can read files in `hw6-solutions`
- C. User `steve` is the only one who can create/delete files in `hw6-solutions`
- D. Users (other than `steve`) who are not in the `faculty` group cannot see a directory listing for `hw6-solutions`

Shell script basics

The shell executes lines one after another

Here's a file named space (helpfully colored by vim)

```
echo "Hello ${USER}."  
disk_usage="$(du --summarize --human-readable "${HOME}" | cut -f 1)"  
echo "Your home directory uses ${disk_usage}."
```

I can run this on mcnulty

```
steve@mcnulty:~$ bash space
```

```
Hello steve.
```

```
Your home directory uses 353M.
```

Making the script executable

Provide a "shebang" line

- ▶ For bash: `#!/bin/bash`
- ▶ This will cause the OS to run `/bin/bash` with the script path as its argument

```
#!/bin/bash

echo "Hello ${USER}."
disk_usage="$(du --summarize --human-readable "${HOME}" | cut -f 1)"
echo "Your home directory uses ${disk_usage}."
```

Make the script executable and run it

```
steve@mcnulty:~$ chmod +x space
```

```
steve@mcnulty:~$ ./space
```

```
Hello steve.
```

```
Your home directory uses 353M.
```

For loops

```
for var in word...; do
  commands
done
```

The words undergo expansion

```
for file in *.*; do
  # Expand file and replace everything up to and including the first
  # period with a single period.
  echo "${file/#*./}"
done
```

Prints out the file extension of each file in the current directory

For loop example

```
for num in {1..10}; do  
    echo "${num}"  
done
```

Brace expansion makes this identical to

```
for num in 1 2 3 4 5 6 7 8 9 10; do  
    echo "${num}"  
done
```


C-style for loop

```
for ( ( num = 1; num <= 10; ++num ) ); do  
    echo "${num}"  
done
```

Which for loop should we use to loop over all files with extension .txt?

A. `for file *.txt; do`
 `cmds`
`done`

D. `for ((file; *.txt; ++file)); do`
 `cmds`
`done`

B. `for file in *.txt; do`
 `cmds`
`done`

E. `for ((file; ++file; *.txt)); do`
 `cmds`
`done`

C. `for file in "*.txt"; do`
 `cmds`
`done`

Exit values

Every command returns an integer in the range {0, 1, ..., 255}

- ▶ 0 means success
- ▶ Everything else means failure

After each command, bash sets the variable `$?` to the exit value of the command

```
$ echo hi; echo "$?"
```

```
hi
```

```
0
```

```
$ ls nonexistent; echo "$?"
```

```
ls: cannot access 'nonexistent': No such file or directory
```

```
2
```

Conditionals

```
if cmd; then  
    more_cmds  
fi
```

If `cmd` returns 0 (success), then run `more_cmds`

Conditionals

```
if cmd; then
  more_cmds
fi
```

If cmd returns 0 (success), then run more_cmds

```
if cmd1; then
  then_cmds
elif cmd2; then
  then_cmds2
else
  else_cmds
fi
```

```
if true; then
  echo 'Our intuition works!'
fi
```

When run, this code will print out "Our intuition works!"

Given that, what value must `true` return?

A. 0

B. 1

C. true

D. false

E. Some other integer

Other loops

while loop

- ▶ execute cmds as long as cmd returns 0

until loop

- ▶ execute cmds until cmd returns 0

```
while cmd; do  
    cmds  
done
```

```
until cmd; do  
    cmds  
done
```

Conditional expressions

`[[expr]]`

- ▶ Evaluates `expr` and returns 0 if it is true and 1 if it is false

String comparisons

- ▶ `str1 OP str2` — `OP` is one of `=`, `!=`, `<`, or `>`
- ▶ `-z str` — true if `str` is an empty string (**z**ero length)
- ▶ `-n str` — true if `str` is not an empty string (**n**onzero length)

Integer comparisons

- ▶ `arg1 OP arg2` — `OP` is one of `-eq`, `-ne`, `-lt`, `-le`, `-gt`, or `-ge`

Examples

```
if [[ "${mode}" = debug ]]; then  
    echo 'the mode variable has the value debug'  
fi
```

```
if [[ -z "${var}" ]]; then  
    echo 'the var variable is the empty string'  
fi
```

```
if [[ $# -gt 0 ]]; then  
    echo "the first script/function argument is $1"  
fi
```

Complete example

```
#!/bin/bash

# Play a guessing game.

num=$(( RANDOM % 10 + 1 ))

IFS= read -p 'Guess a number between 1 and 10: ' -e -r guess
if [[ "${num}" -eq "${guess}" ]]; then
    echo 'Good guess!'
else
    echo "Sorry. You guessed ${guess} but the number was ${num}."
fi
```

```
$ ./guess
```

```
Guess a number between 1 and 10: 3
```

```
Sorry. You guessed 3 but the number was 6.
```

Conditional expressions

File tests

- ▶ `-e file` — true if `file` exists
- ▶ `-f file` — true if `file` exists and is a regular file
- ▶ `-d file` — true if `file` exists and is a directory
- ▶ There are a whole bunch more, read `bash(1)` under `CONDITIONAL EXPRESSIONS`

Other operators

- ▶ `(expr)` — grouping
- ▶ `! expr` — true if `expr` is false
- ▶ `expr1 && expr2` — logical AND
- ▶ `expr1 || expr2` — logical OR