Shell Scripting

Amir Masoumzadeh

CSI 402 – Systems Programming

February 6, 2018

Administrivia

- create issues on GitHub from now on to ask your homework questions
  - in `hwXX` repo for general questions about homework (not specific to your solution)
    - everyone else may benefit from your question and answer
  - in your own repo `hwXX-githubuser` for questions about your own solution
- homework 3 will be out tomorrow
- Exam #1 next week
  - cheatsheet

Running bash scripts

- set file permission (+x)
- running a script
    - `scrit-name` if script directory included in PATH
    - `./script-name` if in the current directory and not in PATH
    - `bash script-name`
    - `source script-name`
        - alternative: `. script-name`
        - runs script line by line
        - script doesn't need execute permission

debugging

- `bash -x script-name`
    - print every command after expansion and before execution
- selective debugging

```
set -x      # activate debugging from here
command1
...
set +x      # stop debugging from here
```

Script format

- start with the shell on the first line
  - `#!/bin/bash`
  - note: starts a child process, inherits env. vars, not aliases/functions
- comments
  - `# comment`

Script writing: best practices

- use long option names
  - `ls -all` instead of `ls -l`
- break long commands into multiple lines using line-continuation (\ followed by linefeed)

Functions

- syntax 1:
```
function name {
    commands
}
```
- syntax 2:
```
name () {
    commands
}
```

Variables

- all vars are internally stored as strings!
    - `number=12`
    - `name=john`
    - `name='john doe'`
- no spaces on either side of assignment sign =
- always a good practice to use double quotes when intending to retrieve values of variables
    - `user1="$name"`
- local variables: visible only within block of code in which it appears
    - `local name=jane`
- export a variable: make it accessible to sub-processes
    - `export name`

Flow control: if

- syntax:

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

Exit status

- each command will return an exit status, a value between 0 and 255
    - 0 means success
- $? provides exit status of last command executed
- related shell builtin commands:
    - `true`: returns 0
    - `false`: returns 1
    - `exit [n]`: causes shell to exit, optionally setting exit status to n
    - `return [n]`: return from function, optionally setting exit status to n

Testing conditions

- syntax 1:

  test expression
- syntax 2:

  [expression]

- rich expression syntax to test files, strings, or integers
- exit status
    - 0: if expression is true
    - 1: if expression is false

Testing conditions (cont.)

- more modern version can test for regular expressions and more:

  `[[ expression ]]`

- regular expression example:

  `if [[ "$INT" =~ ^-?[0-9]+$ ]]; then`

- path expansion example:

  `if [[ $FILE == foo.* ]]; then`

Arithmetic test

- syntax:

  `(( expression ))`

- allows for simpler format of integer expressions

  `if (( ((INT % 2)) == 0)); then`

- exit status
  - true if result of arithmetic evaluation is non-zero

Logical operators

- AND
  - && in [[ ]] or (( ))
  - -a in test
- OR
  - || in [[ ]] or (( ))
  - -o in test
- NOT
  - ! in both cases
- Examples

  ```
  if [ "$RESULT" -a "$INT" -le "$MAX_VAL" ]; then

  if [[ ! ("$RESULT" && "$INT" -le "$MAX_VAL") ]]; th
  ```

Control operators

- command1 && command2
  - command1 executed first;
  - command2 executed iff command1 is successful
- command1 || command2
  - command1 executed first;
  - command2 executed iff command1 is unsuccessful

Branching with `case`

- syntax:
  ```
  case word in
  pattern [| pattern]...)
      commands ;;
  ...
  esac
  ```
- patterns are similar to those used by pathname expansion
- using `;;&` instead of `;;` after commands allows matching multiple cases

Loops: `while`

- syntax:

  ```
  while CONTROL-COMMAND; do
      CONSEQUENT-COMMANDS;
  done
  ```

- manual control of flow inside loops
  - `break`: terminate a loop
  - `continue`: skip remainder of loop (and resume next iteration)

Loops: `until`

- syntax:

```
until TEST-COMMAND; do
    CONSEQUENT-COMMANDS;
done
```

Loops: for

- syntax:
  ```
  for variable [in words]; do
      commands
  done
  ```

- example:
  ```
  for i in {A..D}; do
      echo $i
  done
  ```

- C-like syntax:
  ```
  for (( expression1; expression2; expression3 )); do
      commands
  done
  ```

Positional parameters (command line arguments)

- $0: basename of executed program
- $1 .. $9: its arguments
- $#: number of arguments
- command shift
    - shifts arguments down by one ($2 value moves to $1, $3 to $2, ...)
    - you can loop through them by using shift in every iteration
- functions will have their own arguments when called
- all arguments can be referred to at once via $*, "$*", $@, or "$@"