CS6 Practical System Skills

Fall 2019 edition

Leonhard Spiegelberg
lspiegel@cs.brown.edu

Recap

Last lecture:

- More on streams
- Bash scripting
  - Variables and their environments
  - source script vs. ./script,sh
  - Quoting: "..." vs. '...' vs. `...`
  - Arithmetic expressions via ((…)) and $((…)

Today: More scripting!

Recap

What's the difference between

```
message="hello world"
```

and

```
message = "hello world"
```

?

Recap

What's the difference between

```
message="hello world"
```

variable message is declared

and

```
message = "hello world"
```

command message with 1st parameter = and 2nd parameter "hello world" is executed

?

07 Control flow

CS6 Practical System Skills

Fall 2019

Leonhard Spiegelberg *lspiegel@cs.brown.edu*

07.01 Return codes

⇒ each command, script or program exits with an integer return code (also called exit status) in the range `0-255` (incl., i.e. 1 byte)

⇒ to explicitly exit a script, use the shell builtin `exit code`

⇒ 0 means success, a non-zero indicates an error.

⇒ there are some reserved exit codes frequently encountered, e.g.

1	general errors (e.g. div by zero)
2	misuse of shell builtins

⇒ more extensive list under http://www.tldp.org/LDP/abs/html/exitcodes.html

07.01 Return codes

⇒ You can access the return code of the last executed command via `$?`

Example:

```
tux@server:~$ echo 'Hello world'
Hello world
tux@server:~$  echo $?
0
tux@server:~$  cat filethatdoesnotexist.txt
cat: filethatdoesnotexist.txt: No such file or directory
tux@server:~$  echo $?
1
```

echo returns success

cat failed,
thus non-zero exit status/code

07.01 Executing commands conditional on others

⇒ What is happening when we run

`echo "hello";cp;chown /root` ?

commands are executed after each other, `cp` and `chown` fail and print to stderr

```
sealion@server:~$ echo "hello";cp;chown /root
hello
cp: missing file operand
Try 'cp --help' for more information.
chown: missing operand after ‘/root’
Try 'chown --help' for more information.
```

07.01 Executing commands conditional on others

⇒ `&&` and `II` allow to execute commands depending on each others exit status

⇒ `cmd1 && cmd2` cmd2 is executed iff cmd1 returned 0

⇒ `cmd1 || cmd2` cmd2 is executed iff cmd1 returned non-zero

Example:

```
echo "hello" || echo "world" # <= prints hello

echo "hello" && echo "world" # <= prints hello and world
```

07.01 More on && and ||

⇒ execution occurs from left to right (left associative),
with || and && have same precedence, i.e. read from left to right
Examples:

```
 true && echo 'true always returns $?=0' >&2 || echo 'not printed'
# stderr will receive 'true always return $?=0'

echo "A " && echo "B " && false || echo "C"
# output will be A NL B NL C (NL = new line)
```

=> cmd may be a pipe!

e.g. `cat file.txt | head -n 5 && echo "pipeline done"`

07.01 A longer example

```
touch /file.txt && echo "succeeded at /" || \

touch /usr/file.txt && echo "succeeded at /usr" || \

touch /usr/local/file.txt && echo "succeeded at /usr/local/" || \

touch $HOME/file.txt && echo "succeeded to store at home" || \

echo "failed to store the file in /, /usr, /usr/local or /"
```

you can use \ to break up a command over multiple lines
⇒ that's why \ needs to be escaped as \\

⇒ tries to create a file at /, /usr, /usr/local. However, user has (typically) no rights to do so. Finally, file can be stored at $HOME

⇒ Note: you can silence warnings using e.g. 2> /dev/null on each command!

07.01 Practical example for && and ||

```
apt-get update &&
apt-get install -y openjdk-8-jdk &&
apt-get install -y openssh-server &&
wget http://apache.cs.utah.edu/spark/spark-2.4.0/spark-2.4.0-bin-hadoop2.7.tgz &&
tar xf spark-2.4.0-bin-hadoop2.7.tgz &&
mkdir -p /usr/local/spark &&
chown -R ubuntu /usr/local/spark &&
mv spark-2.4.0-bin-hadoop2.7/* /usr/local/spark &&
rm -rf spark-2.4.0-bin-hadoop2.7* &&
echo "export SCALA_HOME=/usr/local/scala" >> $HOME/.bashrc ||
echo "failed to install spark" && exit 1
```

part of a setup script to install Apache Spark

this starts execution of the following command in case any of the preceding commands failed

display message and exits script with error return code

Compound commands

commands involving commands!

07.02 If statement

man bash:

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

`list` ⇒ a list of words (e.g. a command with parameters)

```
if TEST
then
    COMMAND
fi
```

execute COMMAND if exit code of TEST is 1

```
if TEST
then
    COMMAND1
else
    COMMAND2
fi
```

execute COMMAND1 if exit code of TEST is 1, if exit code is non-zero execute COMMAND2

07.02 If statement - example

```
#!/bin/bash

if chown sealion:sealion /home/tux; then echo "took over Tux's igloo"
else
    echo "attempt to take over Tux's igloo failed :("
fi
```

You can use tabs to format input, or ; to write parts of the command on a line

⇒ Sealion has no root privileges, thus owning Tux's home dir fails.

How to work with variables, files?
How to check permissions?

07.02 If statements - test and [

⇒ test or [are commands which allow to test for a condition and return 0 or non-zero exit status

```
test EXPRESSION

[ EXPRESSION ]
```

last argument should be]

Both test and [are programs stored typically in /usr/bin/test and /usr/bin/[

⇒ status is determined by `EXPRESSION`

⇒ note the whitespace after `test` and `[`!

07.02 if statements - basic tests

`EXPRESSION`	Description
`! EXPRESSION`	The EXPRESSION is false.
`-n STRING`	The length of STRING is greater than zero.
`-z STRING`	The length of STRING is zero (i.e. it is empty)
`STRING1 = STRING2`	STRING1 is equal to STRING2
`STRING1 != STRING2`	STRING1 is not equal to STRING2
`INTEGER1 -eq INTEGER2`	INTEGER1 is numerically equal to INTEGER2
`INTEGER1 -gt INTEGER2`	INTEGER1 is numerically greater than INTEGER2
`INTEGER1 -lt INTEGER2`	INTEGER1 is numerically less than INTEGER2

07.02 if statements - examples

example.sh

```
#!/bin/bash

true ; echo $?                     # => 0
false ; echo $?                    # => 1

[ ! true ] ; echo $?               # => 1

[ -n "hello world" ] ; echo $?     # => 0

EMPTYVAR=
[ -z $EMPTYVAR ] ; echo $?         # => 0

[ "abc" = "ABC" ];echo $?          # => 1

[ 20 -gt 10 ]; echo $?             # => 0
```

!!! Note that 0 is success !!!

-n checks for non-zero string
-z checks for empty/zero string

07.02 if tests - files & permissions

EXPRESSION	Description
`-e FILE`	FILE exists.
`-d FILE`	FILE exists and is a directory.
`-f FILE`	FILE exists and is a regular file
`-L FILE`	FILE exists and is a symbolic link
`-r FILE`	FILE exists and the read permission is granted.
`-w FILE`	FILE exists and the write permission is granted.
`-x FILE`	FILE exists and the execute permission is granted.

Note: permission checks for the user who executes the script.

07.02 if tests - file test examples

test_files.sh

```
#!/bin/bash


[ -e /tux ] && echo "/tux exists" || echo "/tux does not exist"

if [ -w /etc/profile ]; then
    echo "$USER has write permissions to /etc/profile"
else
    echo "$USER has no write permissions to /etc/profile"
fi
```

```
sealion@server:~$ ./test_files.sh
/tux does not exist
sealion has no write permissions to /etc/profile
```

07.02 using ((…)) for tests

⇒ Last lecture: `(( … ))` and `$(( … ))`

⇒ `(( … ))` equivalent to let

⇒ `(( expression ))` evaluates expression, `$(( expression ))` evaluated expression and returned its result

⇒ *man bash:* If the value of `expression` is non-zero,

exit status of `(( expression ))` is `0`, otherwise `1`.

07.02 Example for ((…)) and tests

```
sealion@server:~$ (( 10 > 20 )); echo $?
1
sealion@server:~$ (( 42 == 42 )); echo $?
0
sealion@server:~$ (( 7 * 3 > 20 )); echo $?
0
sealion@server:~$ x=30
sealion@server:~$ (( x * x > 500 )); echo $?
0
sealion@server:~$ if (( 10 < y && y < 30 ));then echo "y in (10, 30)";fi
y in (10, 30)
```

if exit status is 0, then execute command after then keyword

07.02 Difference to $((…))

⇒ What is happening when we execute

```
$((3+4)); echo $?
```

```
7: command not found

127
```

special exit status if command was not found

07.03 Combining tests with && and ||

⇒ can use && and || to combine multiple tests

⇒ what about a logical expression like $a \wedge (b \vee c)$?

⇒ we can use parentheses to group tests!

Example:

```
x=25
((( x > 20 )) || (( x < -20))) && (( x % 5 == 0))
echo $?          # <= will yield 0!
```

07.04 Grouping commands

How does it work under the hood?

⇒ we already had in the last lecture `$(cmd)` (equivalent to `` `cmd` ``) to execute cmd and return its stdout

⇒ in fact `( list )` with `list` being a *list of commands* (separated by `;`), opens up a new shell and returns (as exit status) the exit status of the last command

Example:

```
(true;false); echo $?    # => prints 1
(true;true); echo $?     # => prints 0
```

07.04 example

```
true && (true || false)
```

What is happening?

1. true has exit status 0
2. && checks $?, last exit status is 0 so execution is continued
3. (...) opens up a new subshell
   a. true yields exit status 0
   b. || checks the status, it was 0 so false is not executed
4. exit status of subshell is 0 (status of true)
5. $? will have 0 (the status taken from the subshell)

07.03 Problems with test/[

```
[ ! ! false ]
```

no nesting

⇒ complains, too many arguments

```
[ $x > 0 ] && [ $x < 100]
```

should have used -gt and -lt instead of > and <!

⇒ complains: 100 no such file or directory

```
[ true && (true && false) ]
```

can be fixed by using `true && …`

⇒ complains: syntax error near unexpected token]

The issue: command syntax of [/ test feels rather unintuitive

07.03 Introducing bash's [[expression]]

```
[[ expression ]]
```

is an extension of bash, allowing to write expressions similar to ((expression)).

⇒ i.e. can use parentheses, !, &&, || and all of the switches of [/test

Example:

```
[[ ($PREFIX==/usr/local && -w $PREFIX) || $PREFIX=$HOME/.local ]]
```

can use = or == to compare strings

07.03 Example

```
[[ ($PREFIX == /usr/local && -w $PREFIX) || \
  $PREFIX == $HOME/.local ]]
```

can read expression like in many other programming languages

vs.

```
([ $PREFIX = /usr/local ] && [ -w $PREFIX ]) || \
 [ $PREFIX = $HOME/.local ]
```

always think of exit statuses rather than conditional expressions

07.03 [vs [[...]]

⇒ when using test/[**whitespace is important!**

⇒ when using [[…]], you may delete whitespace

```
[[ ($PREFIX == /usr/local && -w $PREFIX) || \
  $PREFIX == $HOME/.local ]]
```

is the same as

```
[[($PREFIX==/usr/local&&-w $PREFIX) || $PREFIX==$HOME/.local ]]
```

whitespace required here to separate tokens

whitespace required here to end [[

07.04 Comparison [vs. [[

`[`	⇒ use `eq, ne, lt, gt` for comparison
`[[`	⇒ use `==, !=, <, >` for comparison, can use `&&, ||, (...)` for combining

07.05 When to use what?

string and file checks ⇒ use `[` or `[[ ... ]]`

numbers ⇒ use `(( ... ))`

Arrays

07.06 Arrays

⇒ bash supports one-dimensional arrays

⇒ **No support** for nested, multi-dimensional arrays.

=> declare an array via

```
ARRAY=(100 200 300 400 500)

EMPTYARRAY=()
```

whitespace to separate elements

array with zero elements declared via ()

07.06 Arrays - read/write element access

⇒ access n-th element of ARRAY via `${ARRAY[n]}`

⇒ arrays in bash are 0-indexed

⇒ to set the n-th element of ARRAY to value, use

```
ARRAY[n]=value
```

07.06 Arrays - retrieving all elements/indices

What is happening when we run the following code?

```
A=(1 2 3 4 5)
```

```
A[10]=42
```

⇒ many programming languages would throw an out-of-bounds error. bash allows this, because arrays are per default indexed with numbers as keys.

07.06 Arrays - retrieving all elements/indices

⇒ You can use `${!ARRAY[*]}` or `${!ARRAY[@]}` to retrieve the indices/keys of `ARRAY`

Example:

```
tux@server:~$ ARRAY=("abc" 10 "3.141" 42)
tux@server:~$ ARRAY[19]=19
tux@server:~$ echo ${ARRAY[@]}
abc 10 3.141 42 19
tux@server:~$ echo ${!ARRAY[@]}
0 1 2 3 19
```

07.06 Arrays - difference between @ and *

⇒ there is a small but subtle difference between `@` and `*` for arrays:

⇒ Let `ARRAY=(abc 42)`

`"${ARRAY[@]}"` gets expanded to `"abc" "42"` ⇐ two words!

`"${ARRAY[*]}"` gets expanded to `"abc 42"` ⇐ one word!

07.07 Arrays - number of elements

⇒ number of elements in ARRAY (i.e. its size) can be computed using `${#ARRAY[@]}` or `${#ARRAY[@]}`

Example:

```
tux@server:~$ a=(1 3 4 5 61 0 9 2)
tux@server:~$ echo ${#a[@]}
8
```

07.07 Arrays - appending elements

⇒ you can append another array to an array using `+=(...)`

Example:

```
tux@server:~$ a=(1 2 3 4)
tux@server:~$ b=(6 7)
tux@server:~$ a+=(5)
tux@server:~$ echo ${a[@]}
1 2 3 4 5
tux@server:~$ a+=(${b[@]})
tux@server:~$ echo ${a[@]}
1 2 3 4 5 6 7
```

note that there is no whitespace before +=

07.07 Arrays - slicing

⇒ you can get a subarray via

```
${ARRAY[@]:2:3}
```

first number is the starting index (incl.), second number the number of elements of the slice

Example:

```
tux@server:~$ a=(1 2 3 4 5 6 7 8 9)
tux@server:~$ echo ${a[@]:2:3}
3 4 5
```

07.07 Reading in arrays via read

⇒ you can use `read -a` to read words into an array!

⇒ for more options, take a look at http://linuxcommand.org/lc3_man_pages/readh.html

loops

07.08 for loops

```
for name [ [ in [ word ... ] ] ; ] do list ; done
```

⇒ iterates over a list of words, defining in each run a variable `name`

Example:

```
tux@server:~$ for x in 1 2 3 4; do echo $x; done
1
2
3
4
```

07.08 for loops over arrays

```
tux@server:~$ for x in ${a[*]}; do echo $x; done
abc
42
X
tux@server:~$ for x in ${a[@]}; do echo $x; done
abc
42
X
tux@server:~$ for x in "${a[@]}"; do echo $x; done
abc
42
X
tux@server:~$ for x in "${a[*]}"; do echo $x; done
abc 42 X
```

@ splits into words, whereas * doesn't

07.08 for loops - more details

⇒ seq is a command to quickly create a range of numbers

⇒ *man seq:*

```
seq [OPTION]... LAST
seq [OPTION]... FIRST LAST
seq [OPTION]... FIRST INCREMENT LAST
```

Example:

```
tux@server:~$ echo `seq -2 4`
-2 -1 0 1 2 3 4
tux@server:~$ a=(`seq 3 3 30`)
tux@server:~$ echo ${a[@]}
3 6 9 12 15 18 21 24 27 30
```

07.08 for loops using arithmetic expressions

⇒ there is a second version of `for` using arithmetic expressions, similar to many other C-like programming languages

⇒ Details from *man bash*:

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done

First,  the  arithmetic  expression expr1 is evaluated according to the rules described
below under ARITHMETIC EVALUATION.  The arithmetic expression expr2 is  then  evaluated
repeatedly  until it evaluates to zero.  Each time expr2 evaluates to a non-zero value, list
is executed and the arithmetic expression expr3 is evaluated.  If any expression is omitted,
it behaves as if it evaluates to 1.  The return value is the exit status of the last command
in  list
that is executed, or false if any of the expressions is invalid.
```

07.09 while and until loops

⇒ bash also provides while and until loops, from man bash:

```
while list-1; do list-2; done
until list-1; do list-2; done
```

based on exit codes again!

```
The while command continuously executes the list list-2 as long as  the  last
command in the list list-1 returns an exit status of zero. The until command
is identical to the while command, except that the test is negated: list-2 is
executed  as  long as the last command in list-1 returns a non-zero exit
status. The exit status of the while and until commands is the exit  status
of the last command executed in list-2, or zero if none was executed.
```

07.10 Exiting loops

⇒ as part of the body of the loop, you can use

`break [n]`	⇒ leave loop, optional parameter [n] specifies how many loops shall be exited, n must be larger than 1
`continue [n]`	⇒ skip to loop condition, again with optional parameter n

⇒ to quit the script, you may also use `exit [status_code]`

functions

07.11 Functions

⇒ you can define functions in bash, with 2 options:

```
name () compound-command [redirection]

function name [()] compound-command [redirection]
```

⇒ function is called/invoked like any other command, e.g. `mul 3 4` for a function mul

07.11 Functions - example

functions.sh

```bash
#!/bin/bash

# you can declare a function using () syntax
mul () {
  # use echo to print to stdout,
  # and then command substitution to get a return value
  echo $(( $1 * $2 ))
}

a=3
b=4
res=$(mul $a $b)
echo "$a * $b = $res" # should be 12

# other option is to use syntax involving function keyword
function hw() {
   echo "$0: Hello world"
}

hw # prints functions.sh: Hello world
```

parameters for functions are passed like to a script in special variables ${n} for the n-th parameter. $0 is the file name!

```
tux@server:~$ ./functions.sh
3 * 4 = 12
./functions.sh: Hello world!
```

07.11 Grouping commands via {}

⇒ in the previous example, we've seen `{ }` to group several commands. This in fact works generally too:

⇒ `{ list; }` allows to execute several commands (list) to be executed in the current shell context

Example:

`{` `echo "hello" 1>&2` `echo "world"` `} > out.txt 2>&1`	`{echo "hello" 1>&2; echo "world"; } > out.txt 2>&1`
prints hello to stderr, world to stdout. The grouped commands stdout is redirected to out.txt, stderr to stdout.	the same, just in one line

07.12 a note on (list) vs. {list; }

⇒ (...) opens a subshell, i.e. won't override variables in the environment of the parent

⇒ {...} executes within the current context, i.e. may override variables

```
tux@server:~$ a=1; (a=2; echo "inside: a=$a"); echo "outside: a=$a"
inside: a=2
outside: a=1
```

note the whitespace, else syntax errors will happen!

```
tux@server:~$ a=1; { a=2; echo "inside:  a=$a"; }; echo "outside: a=$a"
inside:  a=2
outside: a=2
```

dictionaries / associative arrays

07.13 Dictionaries / associate arrays

⇒ indexed bash arrays allow for integer keys only, e.g.

```
tux@server:~$ a=(1 2 3)
tux@server:~$ a[hello]=90
tux@server:~$ echo ${!a[@]}
0 1 2
tux@server:~$ echo ${a[@]}
90 2 3
```

first element gets weirdly overwritten

⇒ bash has support for non-integer keys as well

⇒ in fact, if keys/indices are not specified explicitly, bash assumes integers

07.13 Dictionaries

⇒ similar to arrays, there is also an inline syntax to declare a dict

```
animals=([dog]=woof [cow]=moo)
```

specify through [key] the key! If none is given, bash uses integers as default.

⇒ element read access: `${animals[dog]}`

⇒ element write access: `animals[dog]="woof woof"`

⇒ `${animals[*]}`, `${animals[@]}`, `${!animals[@]}`, `${!animals[*]}` work as well.

07.13 alternative syntax: declare

⇒ builtin `declare` allows to define variables with attributes

```
declare [-aAfFgilnrtux] [-p] [name[=value] ...]
```

`declare VAR`	declares an empty VAR (same as VAR=)
`declare -a ARRAY`	declares an empty Array(same as ARRAY=())
`declare -A ARRAY`	declares an empty associative array
`declare -r VAR`	makes VAR read-only or creates new read-only VAR

07.13 Dictionaries - example

```
sealion@server:~$ declare -A animals
sealion@server:~$ animals[cow]=moo
sealion@server:~$ animals[dog]=woof
sealion@server:~$ echo ${animals[*]}
woof moo
sealion@server:~$ echo ${!animals[*]}
dog cow
sealion@server:~$ echo ${animals[dog]}
woof
sealion@server:~$ echo ${animals[cow]}
moo
```

07.13 Checking whether a key exists:

```
declare -A dict

dict[USDINEUR]=1.08

[ ${dict[USDINEUR]} ]; echo $?
```
⇒ if key exists, returns 0!

```
[ ${dict[USDINCAD]} ]; echo $?
```
⇒ returns 0

End of lecture.

Next class: Tue, 4pm-5:20pm @ CIT 477