CS6
Practical System Skills
Fall 2019 edition
Leonhard Spiegelberg
lspiegel@cs.brown.edu
Errata
Welcome to different standards...

*NIX is not *NIX...

Mac OS X:  `ls -G` displays colors

GNU/Linux:  `ls --color`
Lecture 02: (Slide 20) GNU/Linux vs. BSD

**cp -R folder/**.

**Mac OS X/BSD:** ⇒ the trailing / in cp is accounted for

**GNU/Linux:** ⇒ the trailing / in cp is not accounted for, however to get BSD behavior use `cp -R folder/*`. 
Recap
05.07 Recap - File permissions

 Unix has file permission to restrict access

 Permissions can be changed using chmod
 ⇒ symbolic mode
 ⇒ numeric mode

<table>
<thead>
<tr>
<th>Octal</th>
<th>Binary</th>
<th>String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>---</td>
<td>no permissions</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>--x</td>
<td>execute only</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>-w-</td>
<td>write only</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>-wx</td>
<td>write and execute</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>r--</td>
<td>read only</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>r-x</td>
<td>read and execute</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>rw-</td>
<td>read and write</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>rwx</td>
<td>read, write and execute</td>
</tr>
</tbody>
</table>

`chmod u=rw,g=rx,o= file.txt` ⇒ `chmod 650 file.txt`
05.07 Recap - Streams & Pipes

standard streams: 0 = stdin, 1 = stdout, 2 = stderr

⇒ can connect streams of commands via pipe operator |
⇒ >, <, >>, << to redirect streams to/from files
More on stream redirection
05.08 Redirecting streams

When the shell is started, it sets up the 3 standard file descriptors (0=stdin, 1=stdout, 2=stderr) and redirects them to the terminal.
05.08 Redirecting streams

1> (or >) to redirect stdout, 2> to redirect stderr

```
cmd > stdout.txt 2> stderr.txt
```
05.08 Redirecting streams

>& out.txt to redirect both stdout and stderr to out.txt

cmd &> out.txt
Redirecting streams to file descriptors

0. Terminal
1. `out.txt`
2. `cmd > out.txt 2>&1`

Note the order here! First, redirect stdout to the file `out.txt`. Then redirect stderr to stdout. If `2>&1 > out.txt` was used, stderr would print to the terminal!

&n references file descriptor n.
⇒ can use this to redirect stderr to stdout!
Can we redirect streams to both the terminal and a file?

⇒ tee file *reads from stdin and writes to stdout and* file

⇒ use tee -a file *to append to* file

Why is this useful?
⇒ You can log a command and see its output while it's running.

```bash
cmd 2>&1 | tee out.txt
```
06.01 Shells
06.01 Shells

On *NIX systems, there are multiple shells available

shell = CLI to the operating system

⇒ pick your favourite shell
⇒ each has a different syntax and unique features
⇒ In CS6 we'll learn bash

<table>
<thead>
<tr>
<th>Shell</th>
<th>Version</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh</td>
<td>Bourne shell</td>
<td>1977</td>
</tr>
<tr>
<td>ksh</td>
<td>Korn shell</td>
<td>1983</td>
</tr>
<tr>
<td>csh</td>
<td>C shell</td>
<td>1978</td>
</tr>
<tr>
<td>tcsh</td>
<td>Tenex C shell</td>
<td>1983</td>
</tr>
<tr>
<td>bash</td>
<td>Bourne again shell</td>
<td>1989</td>
</tr>
<tr>
<td>zsh</td>
<td>Z shell</td>
<td>1990</td>
</tr>
<tr>
<td>fish</td>
<td>friendly interactive shell</td>
<td>2005</td>
</tr>
</tbody>
</table>

 ⇒ widely deployed, de facto standard to write scripts

⇒ documentation under `man bash`

⇒ typically stored under `/bin/bash` or `/usr/bin/bash`
Shell scripts allow to create new commands & save us a lot of time

⇒ automate daily tasks

⇒ system administration can be also automated (e.g., installation of dependencies, technical users, configuration)

⇒ often they are required to deploy services (wrapper scripts, startup scripts)
06.03 Writing scripts - the basics

⇒ scripts are text files, simply create and edit them using e.g. vim

⇒ typical extension for shell scripts: .sh

⇒ to execute a script script.sh, set read&execute permissions (i.e. >= 500) and run it via an interpreter (i.e. a shell), e.g. bash script.sh

⇒ Alternative: you can add a shebang (or bang) line to script.sh, and execute it then like an executable via ./script.sh

```
#!/bin/bash
```

If the first line of script.sh is formatted as

```
#!/<interpreter>
```

`.script.sh` will be the same as

```
<interpreter> script.sh
```
06.03 Writing scripts - the basics

⇒ everything after # is treated as a comment

hw.sh

#!/bin/bash

# a first shell script
clear # reset screen
echo "Hello world"

chmod 500 hw.sh

./hw.sh

Hello world

clears terminal screen

prints Hello world to stdout
06.03 Multiple commands in one line

⇒ multiple statements/commands can be written in one line by separating them using ;

Example:

cd /usr/bin;ls;pwd

is the same as

cd /usr/bin
ls
pwd
06.03 source

⇒ with the `source` command a script may be executed within the current shell.

⇒ helpful, if you want to "save" multiple commands in a file and execute them.
06.04 Variables

Define variables using

VARIABLE=value

⇒ variable names must consist of alphanumeric character or underscores (_) only

⇒ variable names are case sensitive

⇒ you can define a NULL variable (i.e., no value), using VARIABLE=

⇒ many people use a capital letter naming convention for bash variables
To print or access the value of a variable, use $

Examples:

DEST=/home/tux

cd $DEST

MESSAGE="hello world"

echo $MESSAGE

quotes allow for whitespace here!
06.04 Variables and environments

⇒ when variables are defined using `VARIABLE=value`, they are added to the local environment of the executing process.

⇒ E.g., if we type `VARIABLE=value` directly in the shell, then `VARIABLE` is added to the local environment of the shell.

⇒ If we write `VARIABLE=value` in a script, it is added to the local environment of the script during execution.

06.04 Shell variables and environment variables

⇒ when a script is invoked, bash will export its global environment to the script.

⇒ to add a variable to the global environment, use
  `export VARIABLE`
  or `export VARIABLE=value`

⇒ bash defines a set of predefined variables, called shell variables which are always exported.

⇒ to list the global environment, run `printenv`
Some useful shell variables (many more are available):

- **HOME**: the path of the home directory
- **USER**: name of the user
- **SHELL**: path to the shell
- **PWD**: current working directory

⇒ e.g. `cd $HOME` will go to the home directory
06.04 Exporting variables

```javascript
export var
```
source script

 Allows to override any variables (incl. the shell ones)! Don't blindly source a script!
06.04 In a nutshell: source vs. export

⇒ with `export VARIABLE=value` you can pass a variable to a script.

⇒ with `source script.sh` you can add variables to the shell's environment.
Operations on variables
basic arithmetic operations may be executed using

- `let expression`
- `((expression))`
- `$(expression)`

 equivalently

- `let expression and ((expression))`

or

`let expression and ((expression))` evaluate the provided expression using bash's rules regarding arithmetic evaluation.
06.05 Integer operations

⇒ `$(expression)` evaluates the expression and performs then substitution of the result, i.e. returns the result.

⇒ use arithmetic evaluation **for integers only**!

(no floating point support in bash)
## 06.05 Arithmetic evaluation - operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Unary plus/minus</td>
</tr>
<tr>
<td>+</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>**</td>
<td>Multiplication, division, remainder (modulo)</td>
</tr>
<tr>
<td>* / %</td>
<td>(Binary) addition, subtraction</td>
</tr>
<tr>
<td>~ &amp; ^</td>
<td>Bitwise negation, AND, exclusive OR and OR</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>Left/right bitwise shifts</td>
</tr>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>&lt;= =&gt; &lt; &gt;</td>
<td>Comparison operators</td>
</tr>
<tr>
<td>== !=</td>
<td>Equality / inequality</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
</tbody>
</table>
### 06.05 Arithmetic evaluation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var++</code></td>
<td>variable post-increment or post-decrement</td>
</tr>
<tr>
<td><code>var--</code></td>
<td></td>
</tr>
<tr>
<td><code>++var</code></td>
<td>variable pre-increment and pre-decrement</td>
</tr>
<tr>
<td><code>--var</code></td>
<td></td>
</tr>
<tr>
<td>`= *= /= %= += -= &lt;&lt;= &gt;&gt;= &amp;= ^=</td>
<td>=`</td>
</tr>
<tr>
<td><code>exprA?exprB:exprC</code></td>
<td>conditional operator (i.e. if exprA then return exprB else return exprC)</td>
</tr>
<tr>
<td><code>expr1, expr2</code></td>
<td>list operator (more next lecture)</td>
</tr>
</tbody>
</table>

⇒ can use parentheses, precedence like in C
06.05 Arithmetic evaluation - example

```bash
x=42
echo $x #=> 42

let x=x+42
echo $x #=> 84

# use " to allow for whitespace
let "x = x - 4"
echo $x #=> 80

((x--))
echo $x #=> 79

# can use whitespace within (( )) here
(( x *= 7 )) #=> 553
echo $x

let "a=3"
let "b = 4"

let "c = a**2 + b **2"
echo $c

# clamp to [10, ...) 
# use $(( )) to get the result
echo $(( c > 10 ? c : 10 )) #=> 25
```

Note: within (( )) or let or $(( ))), the variables are referenced using their name var, not by $var.

- let expression ⇒ executes expression, but returns no result
- (( expression )) ⇒ executes expression, but returns no result
- $(( expression )) ⇒ executes expression and returns result
we can use variables as part of strings, e.g.

```
cd $HOME/.local/bin will change the directory to /home/tux/.local/bin if HOME=/home/tux
```

Problems:

What is `$avariableinasentence`?

How can we define a variable with content `$HOME`?

What about whitespace/tokenization?
06.06 Quoting

double quotes " . . . " ⇒ perform string interpolation

single quotes ' . . . ' ⇒ treat characters within literally

backticks ` . . . ` ⇒ treat ... as command and return its stdout

⇒ all details available under man bash
06.06 Quoting - single quotes: ' .... '

⇒ single quotes treat each character within them as literal value.
⇒ However, ' can't be contained within ' '

Examples:

echo '$variables are not substituted'

echo 'All sorts of things are ignored in single quotes, like $ & * ; |.'

MESSAGE='hello world!'

echo $MESSAGE
06.06 Quoting - single quotes '...' 

⇒ i.e. single quotes preserve ALL chars except ' 

⇒ can use this for multiline strings, e.g. 

```
sealion@server:~$ echo 'hello
> world'
hello
world
```
06.07 Quoting - double quotes

⇒ double quotes " " preserve literal value (incl. newline!) of characters within them, except for $, `", \ and !. They can be escaped using \, i.e. \\

⇒ $ performs parameter/variable expansion

⇒ ` performs command substitution

⇒ \ is the escape character

⇒ ! performs history expansion
Examples:

MESSAGE="hello world"

echo $MESSAGE

echo "message is: $MESSAGE"

echo "message is: ${MESSAGE}\!"

cache_dir=./cache/

echo "images will be saved to ${cache_dir}images"

echo "images will be saved to $cache_dirimages"
06.08 The $ character

⇒ $ performs parameter expansion, command substitution or arithmetic expansion

⇒ ${parameter} is substituted by the value of parameter
   (if parameter exists, else the empty string)

⇒ $(command) executes command in a subshell
   and returns its stdout

⇒ $ can do a lot more, cf. man bash
```
cmd` is a shortcut for $(cmd)
```

Examples:

```
echo "ls returns `ls`"
echo "ls returns $(ls)"
echo "the current user is $(whoami) (should be ${USER})"
```
06.08 Combining the different quote types

⇒ we can combine the different quote types

Examples:

```bash
echo 'To escape ""\""\" simply surround it with "'
```
```
echo 'result of ls without newlines is: `ls`
```
quoting just allows us to write special chars, but the values are still passed as words

Example:

PARAMS="file.txt dest"

```bash
cp $PARAMS  # <= expands to cp dest src!
cp "${PARAMS}"  # <= expands to cp dest\ src!
cp "${PARAMS}"  # <= expands to cp dest\ src!
cp '$PARAMS'  # <= expands to cp \$PARAMS
```

these commands will raise an error to stderr: `cp: missing destination file operand after ...`
Another example:

PARAMS="word1 word2"

echo PARAMS  # <= output will be PARAMS

echo $PARAMS  # <= output will be the word1 word2
Passing input
06.09 Passing input to scripts

We can pass data in different ways to a script:

1.) as parameters
2.) via stdin
3.) via (exported) environment variables
4.) via an interactive prompt
06.09 Passing parameters

./script.sh param1 param2 param3 ... param20

⇒ access the nth parameter via ${n} in a script.

⇒ $0 (short for ${0}) holds the command name (here ./script.sh)

=> $1 is param1

⇒ $\{20\} is param20

⇒ $\{100\} is NULL/empty string (not set)
within scripts it may be sometimes useful to access stdout, stderr, stdin as files

bash creates 3 special files for the 3 streams to which a command may write to or read from:

stdout /dev/stdout
stderr /dev/stderr
stdin /dev/stdin

Example: `echo 'Hello world' > /dev/stdout`
can use either cat for this and access stdin indirectly

```bash
STDIN=$(cat)
```

or use the special file

```
/dev/stdin
```

execute this script via

```
./stdin.sh < file.txt
```
you can access variables that have been exported in the parent shell, via $VARIABLE

Example:

```
#!/bin/bash

info.sh

echo "\$USER started this script via \$SHELL"
```
06.09 Interactive prompt

⇒ use `read -p PROMPT VARIABLE` to display PROMPT, wait for user to type input and save it to VARIABLE.

Example:

```
prompt.sh

#!/bin/bash

echo "what is your favourite animal?"
read -p '> ' ANSWER
echo "It's a ${ANSWER}, so cool!"
```

There are multiple ways to customize the prompt, e.g. for passwords (-s) etc. ⇒ check man bash
Interactive prompt - multiple variables

read -p PROMPT VAR1 VAR2 VAR3 will issue a prompt, perform word splitting on the received input and fill in the variables.

Example:

```bash
#!/bin/bash

echo "Please write a sentence"
read -p '> ' WORD1 WORD2 WORD3
echo 'First word: '"$WORD1"'
Second word: '"$WORD2"'
Third word: '"$WORD3"
```
So long, and thanks for all the fish.
Next Lecture:

⇒ more advanced variable/parameter expansions
⇒ control structures
  - conditional statements (if)
  - loops (while/for)
⇒ arrays & dictionaries
Homework 2 out today!

⇒ get started early!
⇒ the first scripting homework
⇒ if you're stuck, get help
⇒ man bash is your friend.
End of lecture.

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