(Lab 3) Git prompt & Deep Git, Shell

CS2013 Systems Programming

Department of CSE, IIT Palakkad

IIT Palakkad

Aug 14, 2025

Quiz 3 (15 minutes)

1. Your home already contains a folder named `data`. What happens if you try to run the following and why ?

```
$ touch data
```

2. Write down the sequence of commands to create a directory with the following structure. Assume b, d, e are also directories.

```
     a
    / \
   b   c
      / \
     d   e
```

3. Is it possible to create 1000 empty files with just one command in bash ? If yes, what is it ? If not, argue why ?

Plan

- What happens when we type `ls` in shell ?
- (Deep) Shell scripting
- Making Git more friendly
- Deeper understanding of Git

Announce: Creation of regular class repo

- Accessible via
  - `git clone git@gitserver:class_repo`
- **Contains: all the in-class demo files and examples**
- To see latest changes, do
  - `git pull`
- Reminder: complete reading assignments and practise questions
- Make sure to type commands and run !

What happens when a shell commands is run ?

- Environment variable - `PATH`
  - `$ echo $PATH`

What happens when a shell commands is run ?

- Environment variable - `PATH`
  - `$ echo $PATH`
- Bash searches all folders in `PATH` variable for the executable
- Remember: **Commands are case sensitive**
- Find where the command is located ?
  - Command `which`
  - `$ which git`
  - `$ man which`
- Try - `$ which CHECK`

Shell Scripting Basics

- Shell builtins
  - `echo` - inbuilt - available in shell
  - `ls` - not a builtin - creates a process
  - Run `help` in bash to know more builtins
- Help !
  - manpages - `man bash`
  - Searching manpages - `apropos`

Shell Scripting Basics

- Shell builtins
  - `echo` - inbuilt - available in shell
  - `ls` - not a builtin - creates a process
  - Run `help` in bash to know more builtins
- Help !
  - manpages - `man bash`
  - Searching manpages - `apropos`
- Shell = Interactive command line interpreter
- Shells - zsh, ksh, . . . , **bash** ($\leftarrow$ **Focus**)

How commands are run ?

First word is the program name, rest are arguments

Programming in Shell

- Variables, evaluating expressions

```
# Correct way
$ name="value"
$ count=5

# Wrong way
$ name = "value"
$ count = 5
```

- No spaces before or after =
- Accessing contents
  - Place a $ infront of name
  - Example: `echo $name`.
- In-built shell variables - `PATH, HOME`

Special variables starting with $

- `$?` - Return value of previously run command
- Arguments passed to programs
  - `$$` : PID of current shell
  - `$@` : entire arguments / program name
  - `$#` : number of arguments
  - `$1` : first argument
  - `$2` : second argument
  - ...

Performing evaluations

- For arithmetic expressions . . . , use `$((expr))`

```
# Evaluate an expression
$ a=$((2+3))
$ echo $a
5
```

- For commands . . . , use $(command)

```
# Execute a command
$ result=$(ls)
$ echo $result
```

Repetition

`for` statements

- Usage 1:

```
for i in $(ls)
do
   echo $i
done
```

- Usage 2:

```
for ((i=0;i<10;i++))
do
   echo $i
done
```

Conditionals

if statement

```
name="IITPKD"
if [[ $name = "IITPKD"  ]]
then
    echo "Hello $name"
else
    echo "Bye $name"
fi
```

while statement

```
count=10
while [[ $count -gt 0 ]]
do
    echo "Day $count"
    count=$((count-1))
done
```

While using conditionals / logical expressions

- Remember: space after `if` and `while` and before `[`
- Remember: space before `]`
- . . . for usage as shell scripts
- . . . for direct commandline (interactive) use
  - single line mode using `;` (Demo)

While using conditionals / logical expressions

- Remember: space after `if` and `while` and before `[`
- Remember: space before `]`
- ... for usage as shell scripts
- ... for direct commandline (interactive) use
  - single line mode using ; (Demo)
- Logical connectives ? `&&` , `||`, `!`
- Detour - usage for `&` and `|` ?

Use of `&`

- Process - every command (not builtin ones) creates a process
  - created by OS
  - processes share resources
  - listing processes - `ps`
- Foreground and background process
- `&` - run a program in background
- Bash builtin - `fg` and `bg` (specific to bash)

Standard input, output and error

- Programmer's perspective - only one process is running.
  - Why ? Makes programmers job easier.
  - OS helps achieve this (illusion !)
- A common shared resource - keyboard (standard input) and monitor (standard output)
  - Unix abstraction – Any resource = **File**
  - Unix abstraction – Data (of any kind) = **Streams**
- Standard input - `/dev/stdin`
- Standard output - `/dev/stdout`
- Errors ? Standard error - `/dev/error`

File descriptors and Redirection

- File descriptors
  - 0 = standard input
  - 1 = standard output
  - 2 = standard error
- Redirections >, >> for output

```
$ ls > out.txt
```

  - Redirects **standard output (1)** (aka your display) to file `out.txt`
  - Hence, no output !
  - Clears existing contents. To append, use `$ ls >> out.txt`

Redirection for <

- For input

  ```
  $ tr 'a' 'c' < in.txt
  ```

  - Redirects input to fetch from `in.txt` instead of **standard input (0)** (aka your keyboard)
  - Hence, no wait for input !

Use of Pipes

- Pipes |

```
$ ls | tr 'a' 'c'
```

  - Connects **standard output** of `ls` to **standard input** of `tr`.
  - Allows processing of information as a stream.

Use of Pipes

- Pipes |

  ```
  $ ls | tr 'a' 'c'
  ```

  - Connects **standard output** of `ls` to **standard input** of `tr`.
  - Allows processing of information as a stream.
- Gives a useful way to combine programs (also in combination with >)
- Unix philosophy (due to Peter Salus)
  - Write programs that do one thing and do it well.
  - Write programs to work together.
  - Write programs to handle text streams, because that is a universal interface.

Logical operations

- Logical AND – `&&`
- Logical OR – `||`
- Logical NOT - `!`
- Examples of logical operations (Demo: check `logical.sh`)

Shell Scripting Basics - II

- Basic Coreutils
- System folders, special files types
- Symbolic links and hard links

Coreutils and how to use them

- head, tail, shuf, less
- sort, uniq, wc
- tr, cut
- awk, grep, sed
- du, stat, find
- wget, curl
- date
- bc

Crash course on regular expressions - I

- Wild card - `*`
  - `ls *.txt`
  - Lists all files with `txt` extension.
- Match - `?`
  - `ls name???`
  - Lists file with name followed by any three characters
- Specific - `[ ]`
  - `ls name[123].log`
  - Matches `name1.log`, `name2.log`, `name3.log` (if exists)
- Anchors - ˆ and $ (awk/sed)
  - `grep ^name` – matches lines starting with `name`
  - `grep name$` – matches lines ending with `name`

Crash course on regular expressions - II

- Escape sequence - backslash
  - `\*` – matches asterisk
  - `\\` – matches \
- Characters (sed/awk)
  - `\w` – any words (numeric, characters, underscore)
  - `\W` – all non-words
  - `\d` – any digit
  - `\D` – anything not digit
  - `\s` – any whitespace
  - `\S` – any non-white space

Reduce friction in Git use

- Configure your bash for git usage
- Run `$ which git-prompt.sh`

Reduce friction in Git use

- Configure your bash for git usage
- Run `$ which git-prompt.sh`
- Open the file with nano
  - `$ nano <path_found>`
- Read the comments
- Implement them !

Writing a shell script

- Demo for guessing game

Understanding Git better

- Demo

Lab Exercise

- Question executable available at the address: `http://10.129.4.1/cs2013/lab03/`
- Create `bin` in home. Download the file `INCEPTION`. Copy to the folder `bin`.
- `$ chmod +x bin/INCEPTION`
- `$ INCEPTION`

Class repo (for in-class demo)

- Accessible via
  - `git clone git@gitserver:class_repo`
- To see latest changes, do
  - `git pull`

Class has ended

Humble Request

Please keep the chairs in position before your leave.
(as a token of respect for our CFET staff)