

Shell TUTORIAL

- 2025-03
- ICS@XJTU
- Yunguang Li, Tang Tang, Yuxuan Li

***A brief tutorial for beginners**, so feel free to absent if you are familiar with shell 🙊🙈*

INTERACTIVE WITH COMPUTER SYSTEM

As we all know, there are many ways to interact with a computer system: GUI, CLI, AR, VR, etc.



1970



2020

WHY SHOULD I USE CLI?

1. Some times, **GUI is not available** (e.g. server, embedded system).
And many powerful tools are CLI only (e.g. `git`, `ssh`, `vim`)
2. CLI is more **efficient** (e.g. `mv` v.s. drag and drop)
3. CLI is more **flexible** and **programmable** (e.g. `>`, `|`, `&&`)
4. **ICS** hopes you to use CLI 😂

OVERVIEW

1. **Brief Intro**: all you need to know about starting using shell.
2. **Recommend**: basic but useful command line tools.
3. **Automation**: write a bash scripts.
 - **RTFM**: use *man* and *tl;dr*.

1. BASIC SETUP

1. Terminal (emulator): emulate a (texted-based) terminal inside the GUI environment.

- Linux: `kitty`, `gnome-terminal`, `konsole`, `xterm`, `terminator`, etc.
- Windows: Windows Terminal
- *Open vscode and `Ctrl + ~`*

2. SSH to server:

- `ssh <your stuid>-ics@igw.dfshan.net -p2291`

3. ***Try the tty: `Ctrl + Alt + F1` (F1-F6, in some Linux distros)***

The TTY demystified

2. SHELL: THE "SHELL" OF THE KERNEL

We focus on `bash shell`

- `echo $0`

```
command-name arg1 arg2 arg3 ... # Basic format
```

DO -	In GUI	In CLI (bash shell)
Create a file	Right click, New file	<code>touch filename</code>
Move a file	Drag and drop	<code>mv file1 file2</code>
Launch an app	Click icon	<code>./app</code>
Quit an app	Click close button	<code>Ctrl + C</code>
Suspend an app	Minimize	<code>Ctrl + Z</code>
Show background process	Task Manager	<code>jobs</code>

BASIC TOOLS (COMMANDS)

- Directories: `pwd`, `cd`, `mkdir`
- File: `touch`, `cp`, `mv`, `rm`, `cat`, `less`
- Simple functions: `sort`, `wc`, `echo`
- Others: `grep`, `chmod`
- Code Editor: `vim`
- monitor: `top`, `htop`
- Network: `ping`, `ssh`, `scp`

Tar

Usage Scenario: archive files in 1 bundle

- -c: create a tarball
- -x: open a tarball
- -v: verbose mode [displays progress]
- -t: list files in a tarball
- -f: specify file name

-f is always the last option

```
0 tar -cf name-of-archive.tar /path/to/dir/ # compress directory
1 tar -cf name-of-archive.tar /path/to/filename # compress file
2 tar -cf name-of-archive.tar dir1 dir2 dir3 # compress multiple dirs
3 tar -xf name-of-archive.tar # open a tar file in current directory
```

Tmux

Usage Scenario: manage multiple terminal sessions

The screenshot shows a tmux terminal window with two panes. The left pane contains a C program with a loop and assertions. The right pane contains a C program with a loop and assertions. Annotations with arrows point to the panes and the window title bar.

```
12 // ...
13 srandtime(0);
14 uint32_t sample[32];
15 int i;
16 for(i = 0; i < 32; i++) {
17     sample[i] = rand();
18     reg[i] = sample[i];
19     assert(reg_val == (sample[i] & 0xffff));
20 }
21
22 assert(reg_val == (sample[0] & 0xffff));
23 assert(reg_val == (sample[1] & 0xffff));
24 assert(reg_val == (sample[2] & 0xffff));
25 assert(reg_val == (sample[3] & 0xffff));
26 assert(reg_val == (sample[4] & 0xffff));
27 assert(reg_val == (sample[5] & 0xffff));
28 assert(reg_val == (sample[6] & 0xffff));
29 assert(reg_val == (sample[7] & 0xffff));
30
31 assert(sample[0] == cpu.eax);
32 assert(sample[1] == cpu.eax);
33 assert(sample[2] == cpu.edi);
34 assert(sample[3] == cpu.edi);
35 assert(sample[4] == cpu.ebx);
36 assert(sample[5] == cpu.ebx);
37 assert(sample[6] == cpu.ebx);
38 assert(sample[7] == cpu.ebx);
39
40 // ...
41 void cpu_exec(uint32_t n) {
42     volatile uint32_t n_temp = n;
43     for(i = 0; i < n; i++) {
44         setjmp(jbuf);
45         swaddf.i.eip_temp = cpu.eip;
46         int instr_len = exec(cpu.eip);
47         cpu.eip += instr_len;
48         if(n_temp == 1) { (enable debug && !quiet) {
49             print_bin_instr(eip_temp, instr_len);
50             puts(assembly);
51         }
52         if(n_temp == 1) {
53             printf("user interrupt\n");
54             return;
55         }
56         else if(n_temp == 0) { return; }
57     }
58 }
59
60 // ...
61 // project/nemu/src/cpu/req.c [c] unix utf-8 in 39, col 179
```

Annotations in the image:

- Two blue arrows pointing to the two panes, labeled "different tabs in vim".
- A blue arrow pointing to the window title bar, labeled "the same tab in vim".
- A red arrow pointing to the right pane, labeled "different panes".

- prefix key: `Ctrl + b`
- Client-Server model: `tmux (server) + tmux attach (client)`

grep

Usage Scenario: search for a specific string in a file

grep + regex

- **-i**: case insensitive
- **-r**: recursive search
- **-n**: show line number
- **-v**: invert match

```
0 grep -i "hello" file.txt # search for "hello" in file.txt
1 grep -r "hello" . # search for "hello" in all files in current directory
2 grep -r "hello" . -n # search for "hello" in all files in current directory and show
3 grep -r "hello" . -v # search for files that do not contain "hello" in current direct
```

INTERLUDE: SO MANY COMMAND

- `-h, --help`
- `man`: `man` is the system's manual pager (Ask the man XD)
 - `man -k ipc`
 - `man man`
 - Some of the following command can be found their manpage, but how about `cd`?
- `tldr`: <https://github.com/tldr-pages/tldr>
 - There is room for simpler help pages focused on practical examples.
 - `man tar` v.s. `tldr tar`

Find

Usage Scenario: search files in a directory

- -name: search by name
- -type: search by type
- -exec: execute command on each file found

```
0 find . -type f -name "*.txt" # find all txt files in current directory
1 find . -type f -name "*.txt" -exec cat {} \; # cat all txt files in current directory
```

MORE TOOLS

- `awk`: a powerful pattern scanning and processing language
- `sed`: a stream editor for filtering and transforming text
- `curl`: transfer data from or to a server
- `ag`: a code-searching tool similar to `grep`
- `tree`: list contents of directories in a tree-like format
- `htop`: an interactive process viewer for Unix
- `cmatrix`: a program that simulates the display from "The Matrix"
- `sl`: a steam locomotive runs across your terminal

INSTALL SOFTWARE IN CLI

1. Package manager: apt (ubuntu, Debian), brew(macOS), dnf(fedora), pacman(arch)

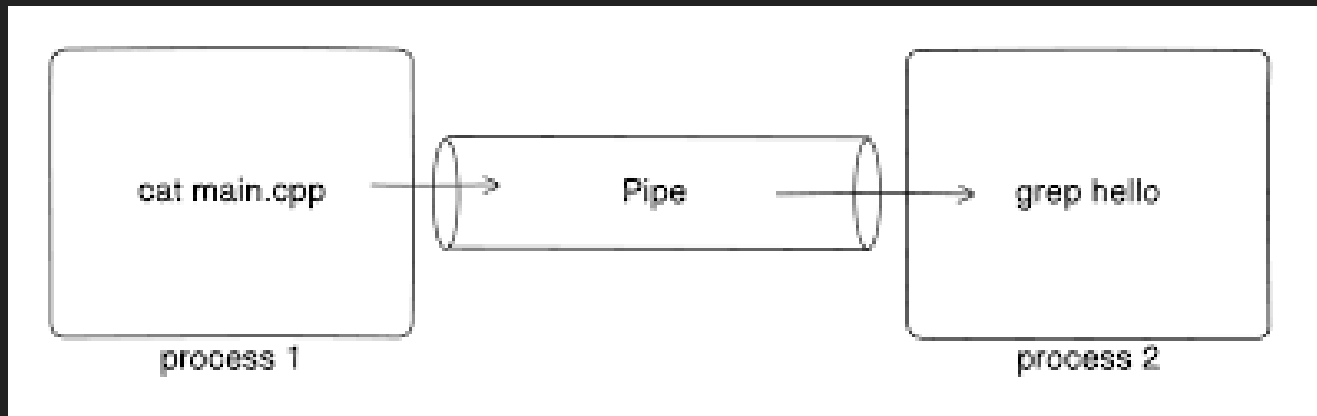
- Search (e.g. `apt search`)
- <https://command-not-found.com/>

2. Build from source

- README/INSTALL doc
- configure and make install

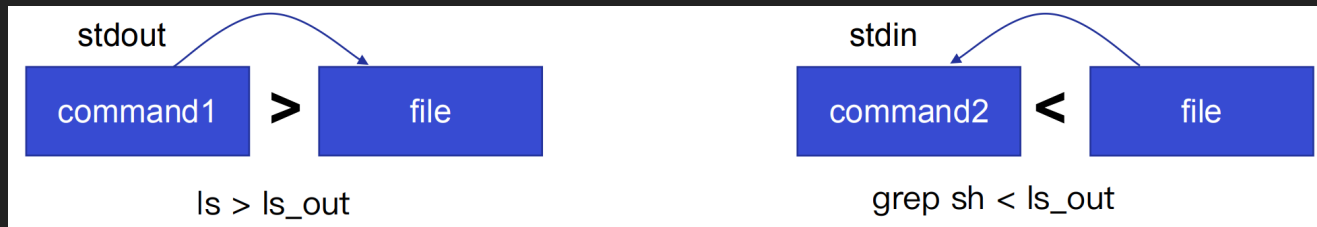
COMMUNICATION: PIPE

- A lot of CLI tools, communication is required to do complex jobs.
- Pipe: | use the `stdout` of previous command as the `stdin` of the next.



COMMUNICATION: REDIRECT 1

- A lot of CLI tools, communication is required to do complex jobs.
- Redirect: > & <, stdout to file or file to stdin (normally).



COMMUNICATION: REDIRECT 2

- A lot of CLI tools, communication is required to do complex jobs.
 - 0 - `stdin`, the standard input stream.
 - 1 - `stdout`, the standard output stream.
 - 2 - `stderr`, the standard error stream.

```

test.c  buffers
1 #include <stdio.h>
2
3 int main(){
4     fprintf(stdout, "Here is STDOUT!\n");
5     fprintf(stderr, "Here is STDERR!\n");
6     return 0;
7
~/Code
> gcc test.c -o test
~/Code
> ./test
Here is STDOUT!
Here is STDERR!
~/Code
> ./test > test.out
Here is STDERR!
~/Code
> ./test 2> test.out2
Here is STDOUT!
~/Code
> ./test > test.out 2> test.out2
~/Code
> cat test.out
Here is STDOUT!
~/Code
> cat test.out2
Here is STDERR!
~/Code
>

```


COMBINING COMMANDS (FURTHER MORE)

1. Count students number in server
2. fetch all include file
3. diff between two directories
4. check the disk usage of all files in /usr/bin

- *xargs*
- *<() : temporary file*
- *\$() : command substitution*

- Build a temporary tools combination
- We a programming
 - a "Natural programming language"
 - Cooprate tools together

3. SHELL SCRIPTS

Shell is also a programming language, which allows you to combine a series of commands and execute

VARIABLES

In bash, the syntax for assigning a value to a variable is `foo=bar`, and to access the value stored in a variable, the syntax is `$foo`.

Notes:

1. `foo = bar` (with spaces around `=`) will not work: the interpreter will try to run a program `foo` with `=` and `bar` as arguments.
2. **In shell scripts, spaces are used to separate arguments.**

STRINGS

In Bash, strings can be defined using ' and ", but they have different meanings:

- Strings defined with ' are literal strings, where variables are not replaced.
- Strings defined with " are strings where variables are replaced with their values.
- read more in Official [Bash Manual](#)

```
foo=bar
echo "$foo" # print bar
echo '$foo' # print $foo
```

CONTROL STRUCTURES

```
if [ expression ]; then
    # do something
elif [ expression ]; then
    # do something
else
    # do something
fi
```

```
for i in 1 2 3 4 5; do
    echo $i
done
```

```
while [ expression ]; do
    # do something
done
```

TEST

```
test expression  
[ expression ]  
[[ expression ]]
```

- [-e file]: if file exists, then true.
- [string]: if string is not empty (length > 0), then true.
- [string1 != string2]: if string1 and string2 are different, then true.
- [integer1 -eq integer2]: if integer1 equals to integer2, then true.
 - **do not confuse with numeric and string comparison**

FUNCTIONS

```
mcd () {  
    mkdir -p "$1"  
    cd "$1"  
}
```

SPECIAL VARIABLES

Different from other scripting languages, bash uses many special variables to represent parameters, error codes, and related variables

- `$0` : script name
- `$1 ~ $9` : script parameters. `$1` is the first parameter, and so on.
- `$@` : all parameters
- `$#` : number of parameters
- `$?` : return value of the previous command
- `$$` : process ID of the current script
- `!!` : the last command, including parameters. Common usage: when you fail to execute a command due to insufficient permissions, you can use `sudo !!` to try again.
- `$_` : the last parameter of the last command

SHABANG

- shabang (#!) is a special comment that tells the system which interpreter to use to execute the script
- **#! + <Path of interpreter>**

```
#!/bin/bash  
echo "Hello, World!"
```

```
#!/usr/bin/env python3  
# use env to find python3 in PATH  
print("Hello, World!")
```

builtin

- `source` or `.`: run commands in the current shell
- `cd`: change directory
- ..., read more in `man bash-builtins`

\$. THE BEST WAY TO LEARN IT , IS TO USE IT.

"Unix is user-friendly; it's just choosy about who its friends are."

- MIT - [The Missing Semester](#)
- USTC - [Linux101](#)
- [The Art of Command Line](#)