NASA Lecture 2 (with lab)

2018/03/05

~~Michael Tsai~~ Kai-Ling Lo (@coderalo)

Outline

- A Brief Introduction to UNIX/Linux
- The Beginning of Your Linux Tour (SSH, man, editor)
- Shell in Operating System
- File System in Linux/UNIX
- File Attributes and Permission
- The Hello World of Shell Scripting
- Pipes and Redirection
- Variables and Flow Control
- Test in Shell Scripting
- && and || Operators
- Usually Used Commands
- Regular Expression
- Exercise

A Brief Introduction to UNIX/Linux

- **CTSS** (1961 - 1973, MIT): one of the first time-sharing operating system
- **Multics Project** (1960s, Bell Labs): a heavy “second-system” of CTSS
- **UNIX** (1970s, Bell Labs): The great successor of Multics
- The father of UNIX: **Ken Thompson and Dennis Ritchie** (*also the father of C language!*)

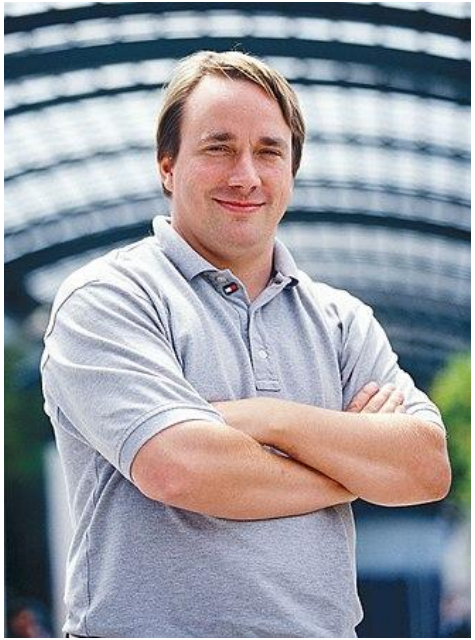
A Brief Introduction to UNIX/Linux

- The beginning of UNIX: Space Travel, PDP-7
- *“What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form.”*
- ARPANET (1969)

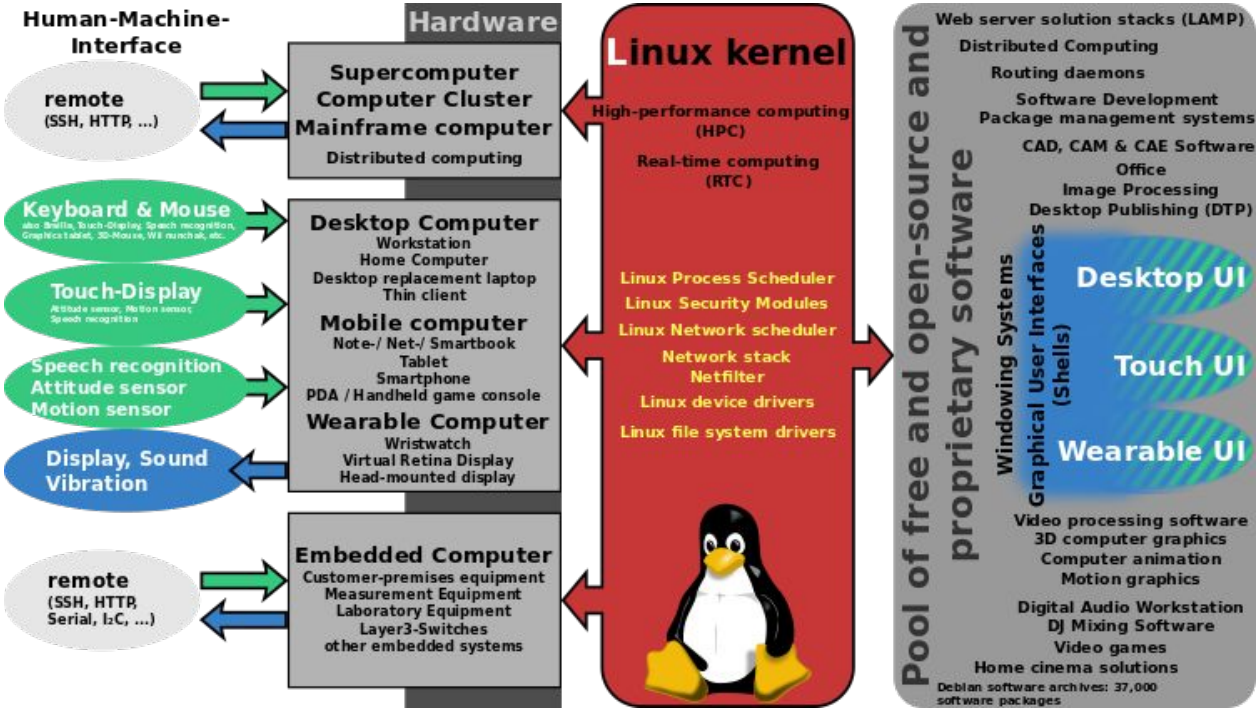
Reading: http://www.faqs.org/docs/artu/ch02s01.html

A Brief Introduction to UNIX/Linux

- So, that is the beginning of UNIX, then how about Linux we use today?
- **BSD** (Berkeley Software Distribution): based on 6th edition of UNIX (from AT&T)
  -> be trapped in the lawsuit (early 1990s), which results in the limited development
- Incomplete **GNU** project (1983, Richard Stallman, MIT)
- Intel's 32-bit instruction set (1985), **MINIX** (1987)
- **Linus Torvalds** announced his Linux project in 1991
- A "cheap UNIX system for everyone"

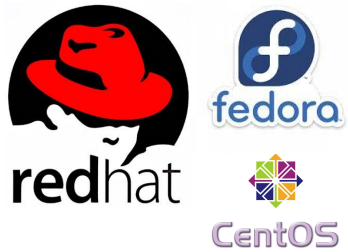


A Brief Introduction to UNIX/Linux



A Brief Introduction to UNIX/Linux

- The Linux family



The Beginning of Your Linux Tour

- **SSH** is always your good friend
- SSH to workstation: https://wslab.csie.ntu.edu.tw/ssh/
- Important notices:
  - ALWAYS set a **strong password** (maybe a random password)
  - Use **SSH key** is better (the password would become plain text on remote machine, even though it's hard to be stolen in a "safe" system, it's still possible)
  - It's even more better to set a **keyphrase** on your SSH key (you still can login without any password with **SSH agent** (you'll try it in HW1))
  - Do **NOT** copy your private key over the network (**ONLY** on your own computer)
- Keyword: ssh-keygen
- Try it now!

The Beginning of Your Linux Tour

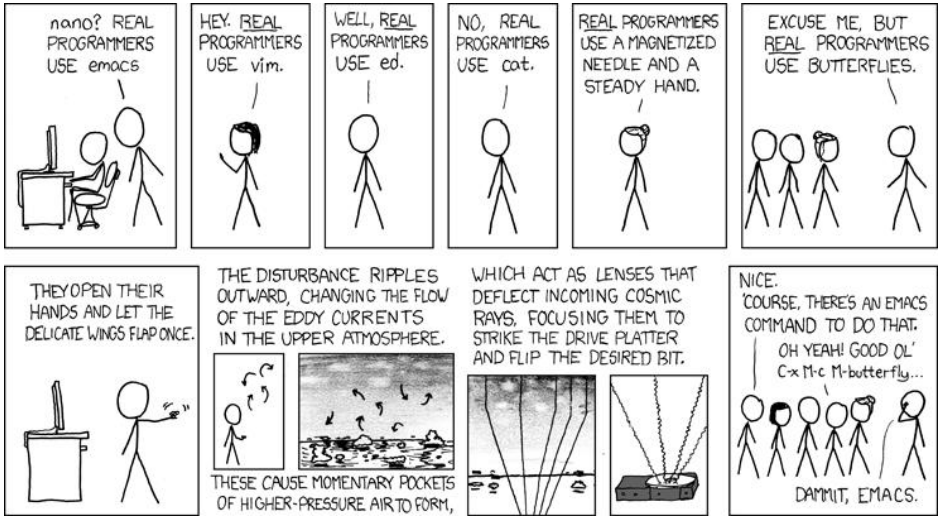
- You should make sure your terminal handles Chinese well (change the **encoding**)
- Select a good font and the respective font size is important, too
- You'll playing with it more in HW1 ! **(can't wait, right?)**

The Beginning of Your Linux Tour

- **MAN** is also your friend, too. (especially when you don't have a web browser to use!)
- Your first man: "man man"
- 9 sections of man pages (you can see them with "man man")
- []: optional argument
- |: choose-one argument (cannot be used together)
- **...**: repeatable argument
- If you don't know what exactly to man: man **-k** "..."
- A funny Easter egg of man

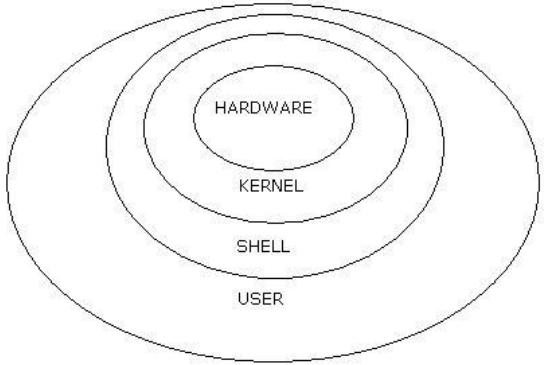
The Beginning of Your Linux Tour

- Editor War (vim v.s. emacs)
- **Vim is the BEST editor in the universe**
- For vim, you can start from the built-in tutorial - **vimtutor**.
- If you think that vim is too hard for you to start, then you can just start from **nano** or **joe**.



Shell in Operating System

- Kernel - the heart of operating system
- Shell - the user interface to kernel
- The shell here is usually the CLI shell (black background with only words lol)
- There are a bunch of different shells, like
  - sh
  - bash
  - tcsh
  - ksh
  - zsh
  - ...



Shell in Operating System

- We'll focus on the scripting in "the Bourne-again shell" (**bash**), but of course, you can choose your favorite shell for daily use. e.g. I'm use zsh (+ oh-my-zsh) on my laptop and workstations.
- We choose bash mainly because it is the **default login shell** on most systems. (e.g. it's your default shell on workstations), and you can use bash *almost* everywhere.
- However, it's important that *bash is an **extension** to POSIX shell (sh)*, and uh... it means that we can't expect that the script written in bash can be run in all systems, i.e. **the script is not reusable (bashism)** (Well, in most of the occasions, it's not a big deal, though).

File System in Linux/UNIX

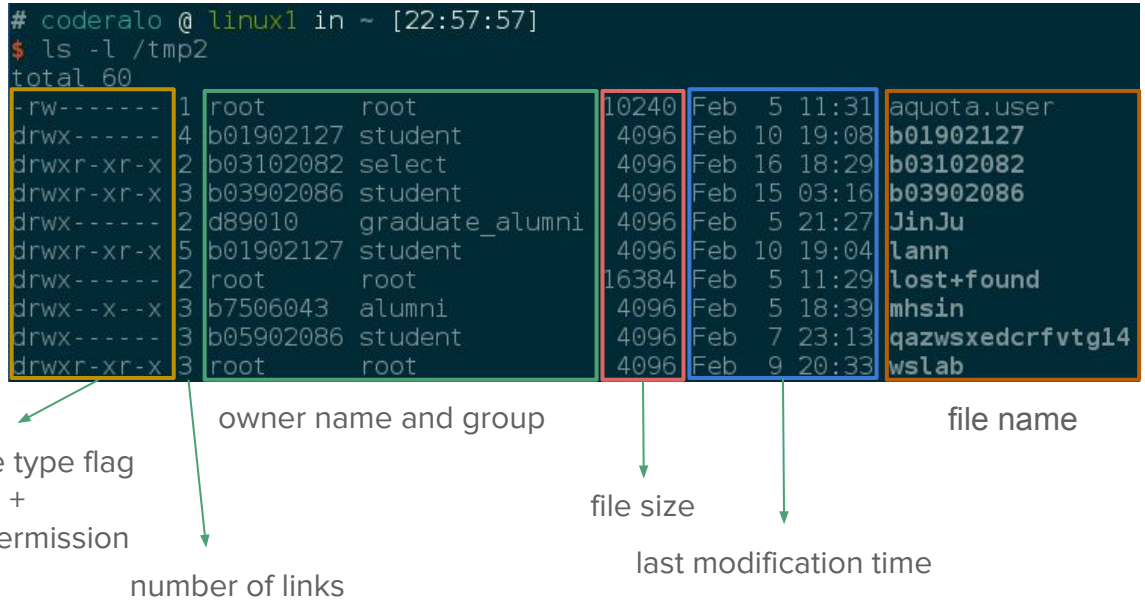
- With your CSIE instinct, you might realize that things in file system are not stored on disk by name, but using a numbered data structure called **inode**. The data itself is stored on disks, and inodes contain pointers to the disk blocks.
- What a directory stores is actually **a mapping from name to inode**, not the data itself. If we try to move the file from one directory to another, what we'll do is actually erase the file from list of the original directory, and write the file into the list of the new one. (That's also why mv is much more faster than cp if it happens in the same file system)
- It's important for us to understand how permission works.

File System in Linux/UNIX

- When you're in a session, or writing a shell script, you can infer the directories as below.
- Directories:
  - *It's a tree*
  - absolute path (full path): start from the root directory (/), and grouped by '/' in a hierarchical manner
    e.g. /home/coderalo/NASA/
  - relative path: start from the working directory, using '.' and '..' to represent current working directory and parent directory.
    e.g. ../../NASA/hw1/
    ** You can use "pwd" command to get your working directory*
  - home path:
    you can use the symbol '~' to represent your home directory (dependent on user);
    you can also use "~[USER]" to represent the home directory of the user, e.g. "~coderalo"

File Attributes and Permission

- There are two main ways for us to get file attributes: **ls** and **stat**
- Examples:



File Attributes and Permission

```
# coderalo @ linux1 in ~ [23:20:58]
$ stat /tmp2
  File: /tmp2
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: fe04h/65028d    Inode: 2           Links: 12
Access: (1777/drwxrwxrwt)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2018-02-05 11:29:29.000000000 +0800
Modify: 2018-02-16 18:29:41.189188716 +0800
Change: 2018-02-16 18:29:41.189188716 +0800
 Birth: -
```

- Blocks: the total number of physical blocks actually allocated on disk
- Device: the device which the file is on
- Modify: the last time the file was modified (content has been modified)
- Change: the last time the metadata of the file was changed (e.g. permission)
- Birth: the time the file was created on the file system (it’s empty due to ext4)

File Attributes and Permission

- File flag: “-” for normal file, “d” for directory, and “l” for symbolic link
- Permission: 9 bits, **first** 3 bits for **owner**, **middle** 3 bits for **group**, and **last** 3 bits for **others**
- Every group of 3 bits have the same meaning for each bit: the **first** bit for **read** permission, the **second** bit for **write** permission, and the **third** bit for **execute** permission.
- “-” is off, and “[r|w|x]” is on.
- Permission for files is easy to understand: read permission for read, write permission for modification, and execute permission for execute (if the file is executable).
- Directory permission is more complex: briefly, with execute permission, we can **change directory (cd)** into it; with read permission, we can **list the properties of files (ls)** in it; and with write permission, we can **create / delete / rename (touch, rm, mv ...)** the files in it.
- *You’ll know more about permission in HW1.*

File Attributes and Permission

- In general, we can use “**chmod**”, “**chown**”, and “**chgrp**” to modify the permission.
- We have several ways using the command chmod to modify the permission bits:
  - chmod u+x test.sh
  - chmod 700 ~
  - chmod ug=rw, o=r list.txt
- We can use chown to change file ownership:
  - chown b04902010 hw1.pdf
  - chown coderalo:ta NASA_secret.txt
- The command chgrp is used to change group ownership of file:
  - chgrp student /tmp2/NASA_reference/

** You can use “groups” command to get the groups to which a user belongs*

File Attributes and Permission

- Well, the basic permission sometimes can’t fit our need. e.g. how do we set different permission for distinct users, or distinct groups?
- One possible solution is **access control list (ACL)**, which is now supported by most of the Linux distributions on several kinds of file system (ext*, xfs, etc.)
- You can check whether the system supports ACL by using the command dmesg: *dmesg | grep -I acl*
- Identify user / group, and then apply the permission
- Use **getfacl** to get the ACL of the file, and use **setfacl** to set the ACL of it.

File Attributes and Permission

```
# coderalo @ linux1 in /tmp2 [1:56:02]
$ getfacl test.sh
# file: test.sh
# owner: coderalo
# group: ta
user::rw-
user:yunchih:r-x
group::r--
group:student:r--
mask::r-x
other::r--
```

user::rw-: the permission of owner
group::r--: the permission of the group “ta”
other::r--: the permission of other users which are not specified in this list

user:yunchih:r-x: the permission of user “yunchih”
group:student:r--: the permission of group “student”
mask::r-x: the “max” permission of other users and groups

File Attributes and Permission

- Use getfacl to get the file permission: getfacl file ...
- Some examples of setfacl:
  - setfacl -m user:b04902010:rwx test.sh
  - setfacl -x group:student test.sh
  - setfacl -R user:lyt:--- /tmp2/coderalo/ (recursively set the permission)

The Hello World of Shell Scripting

- Make sure your shell is bash: *echo $SHELL*
- If not, you need to change it; you can use **chsh** to change your default shell, then login again. (FYI, the chsh on workstations isn't normal chsh lol)
- Another solution is using "bash -l" to enter bash as if it's login shell (DON'T only enter bash, which might cause some issues about environment variables)
- **Every commands you'll use are either shell built-in or a script/executable (you can use **which** command to check)*

The Hello World of Shell Scripting

- Now you can start building your hello-world script. Open an editor (e.g. vim, ~~emacs~~), and write down the lines below:

```
-----------------------------------------
#! /usr/bin/env bash

# your code starts here!
echo "Hello, world"
-----------------------------------------
```

- Save the code to a file (like hello.sh), and now you have two ways to run it.
  1. Run it with “bash hello.sh”
  2. Give it execute permission and run it as “./hello.sh”

Pipes and Redirection

- Every program we run on the command line has three data streams connect to it: **STDIN** (standard input), **STDOUT** (standard output), and **STDERR** (standard error). We can connect these streams between programs and files by **pipes**, which is **redirection**.
- To / From files:
  - > : redirect STDOUT to file (cover)
  - >>: redirect STDOUT to file (append)
  - >&: redirect STDOUT and STDERR to file
  - 2>: redirect STDERR to file
  - <: redirect file to STDIN
- To / From programs: |
- Examples:
  - echo "Hello, world" > /tmp2/test.txt
  - ls -l /etc/ | grep "system"

Variables and Flow Control

- As other programming languages you've used before, **variables** and **flow control** are two essential parts in shell script. Here are some examples:
  - Assign value: **[VAR]=[VALUE]**, e.g. coderalo_dir="/home/coderalo"
  - Take value: **$[VAR]**, e.g. echo $coderalo_dir
  - You can use **{}** to precisely specify the variable name, e.g. echo ${coderalo_dir}/NASA/
  - When we're specify a string, there are several possible usage:
  - Use **''** to print the same thing as you specify; use **""** to substitute value for variable name, and use **$()** (*or* ``) to specify the code being executed. E.g.
  - $ echo '${coderalo_dir} is the same as `pwd`'
    ${coderalo_dir} is the same as `pwd`↵
  - $ echo "${coderalo_dir} is the same as `pwd`"
    /home/coderalo is the same as /home/coderalo↵

Variables and Flow Control

- We can also see command-line arguments as variables:
  - **$#**: The number of arguments
  - **$0**: The command itself
  - **$1, $2, ... $n**: The n-th argument
- You can use the variables as number and calculate by using **$((VAR))**.
  E.g.:
  a=1
  b=$((2))
  c=$a+$b -> The value of c is “1+2”
  d=$(($a+$b)) -> The value of d is 3
- *You’ll learn more about variables in HW1 (how to parse the arguments?)*

Variables and Flow Control

- As the C language you're familiar with *(uh.. you're familiar with it, right?)*, you can use the statements (**if, case**) and loops (**for, while**) in shell scripts.
- If statement:

```
if condition; then
    commands
elif condition; then
    commands
else
    commands
fi
```

- Case statement:

```
case VARIABLE in
    pattern1)
        commands
        ;;
    pattern2)
        commands
        ;;
esac
```

Variables and Flow Control

- For loop:
  - for VARIABLE in 1 2 3 4 5 .. N;
    do
        commands
    done
  - for VARIABLE in $(COMMAND);
    do
        commands
    done
- While loop:
  - while condition; do
        commands
    done

Variables and Flow Control

- Break and continue:

```
while condition; do
    if condition; then
        command
    elif condition; then
        continue
    else
        break
    fi
```

- break: leave the loop
- continue: go to the next iteration

Test in Shell Scripting

- You can use **test** command as the conditions in statements and loops; you won't see the word "test" in your command, since the command is not called directly but as "[]" (it's executable!).
  E.g. if [$message_level -le $LOG_LEVEL]; then ...
- There are a bunch of flags can be used in test, and you can just look for them in the manual. (They won't be totally listed here because they're too many!)
- Next page is some frequently used flags.

Test in Shell Scripting

String	Numeric	True if
x = y	x -eq y	x is equal to y
x != y	x -ne y	x isn’t equal to y
x < y	x -lt y	x is less than y
x <=- y	x -le y	x is less or equal to y
x > y	x -gt y	x is greater than y
x >= y	x -ge y	x is greater or equal to y
-n x		x is not null
-z x		x is null

-d file	File exists and is a directory
-e file	File exists
-f file	File exists and is a regular file
-s file	File exists and is not empty
-r file	Have read permission of the file
-w file	Have write permission of the file
file1 -nt file2	File1 is newer than file2
file1 -ot file2	File1 is older than file2

&& and || Operators

- **&&** and || are two important operators in shell scripting. You can see them as kind of exception handling.
- When we use “&&” to combine two pieces of code, the right side of && will only be evaluated if the exit status of the left side is zero (i.e. the left side is successfully executed).
- On the other hand, when we use “||”, the right side of || will only be evaluated if the exit status of the left side is nonzero.
- *You can also use them in conditions of statements and loops, too.*

Frequently-Used Commands

- Wow! Now you know a lot about shell scripting!
- Next step is to learn some frequently-used commands, like
  - **ls, stat** - get file attributes
  - **cd, pwd** - working directory management
  - **mkdir, touch** - create new files / directories
  - **rm, mv, cp, rsync** - move or delete files / directories
  - **find, which** - search for files
  - **cat, less, head, tail, echo, printf** - print file or message
  - **ps, pgrep, top, kill, pkill** - process management
  - **awk, grep, sed, cut, tr, sort, truncate, wc** - file processing
  - **tee** - data stream redirection
- We'll only cover some of them in class, but it's great to get familiar with **all of them** :)

Frequently-Used Commands

- **echo** is a command which helps you display messages on terminal (standard output).
  Usage: echo [OPTION] ... [STRING] ...
- There are some useful options:
  - -n, print without trailing newline
  - -e, enable interpretation of backslash escapes
- Examples:
  - $ echo “Hello, world”
    Hello, world↵
  - $ echo -n “Hello, world”
    Hello, world
  - $ echo -e “Hello, world\n\n\n”
    Hello, world↵↵↵
  - $ echo -ne “Hello, world\n”
    Hello, world↵
- You can also use **printf** to do formatting printing.

Frequently-Used Commands

- When reading a file, standard output or anything coming from a pipe, **head** and **tail** are two useful commands to only read the first or the last part of it.
- Usage:
  - head [OPTION] ... [FILE] ...
  - tail [OPTION] ... [FILE] ...
- Useful options:
  - -[NUM]: print the first NUM lines
  - -v: print the headers (file names)
  - -[NUM]: print the last NUM lines
  - -v: print the headers (file names)
  - -f: output appended data as the file grows
  - -s=[N]: set sleep interval for (appro) N seconds (with -f)

Frequently-Used Commands

- **cut** is a command used to extract selected parts of lines from file or standard output. It's useful when we're only interested in specific part of data, e.g. the timestamp of "ls -l".'
- Usage: cut [OPTION] ... [FILE] ...
- Here are some frequently used options:
  - -c=LIST: select only the characters specified in LIST
  - -d=DELIM: use DELIM instead of **TAB** for field delimiter
  - -f=LIST: select only the fields specified in LIST
  - -s: do not print lines not containing delimiters
- Examples:
  - $ ls -l | cut -s -c1-10
  - $ cut -d , -f 1,3 data.csv

Frequently-Used Commands

- **sort** is a magical command which can sort the data for us!
- Usage: sort [OPTION] ... [FILE] ... *or sort [OPTION] --files0-from=F*
- Useful options:
  - -k=KEYDEF: sort via a key (KEYDEF)
  - -t=SEP: use SEP instead of **non-blank** to blank transition
  - -h: compare human readable numbers (e.g. file size)
  - -n: compare according to string numerical value
  - -r: reverse the result of comparisons
- Examples:
  - $ sort -t \t -k 2,3 result.txt
  - $ df -h | tail -n +2 | sort -k 2 -h

Frequently-Used Commands

- **awk** is a *programming language* (yes, it's a programming language...) for text processing. We usually insert some awk code into our script to process the data by using the command **awk**.
- *Here we'll only go through some basic usage of awk; if you want to learn more about it, you can read the awk manuals, or search for the tutorials on Internet.*
- First, you can specify separator and variables by using arguments (-f [SEP] | -v [VAR] ...)
- It's useful to do something *before and after* processing. We can achieve it by making use of the **BEGIN** and **END** block in awk.
- Example:
  - cat /etc/passwd | awk -F ":" 'BEGIN {print "USER:HOME"} {print $1":"$6}'

Frequently-Used Commands

- **grep** and **sed** are two powerful tools for text processing; **grep** is used to extract lines matching specific pattern, and **sed** is a stream editor for filtering and transforming text.
- Usage:
  - grep [OPTIONS] PATTERN [FILE]
  - grep [OPTIONS] [-e PATTERN]... [-f FILE] ... [FILE]
- Here are some useful options of grep:
  - -v: select non-matching lines
  - -i: Ignore case distinctions in both the PATTERN and the input files
  - -c: Print a count of matching lines for each input file, instead of the normal output
  - -o: Print only the matched (non-empty) parts of a matching line

Frequently-Used Commands

- Usage: sed [OPTION] [SCRIPT] ...
- Examples:
  - ls -l | sed '1**d**' | head -10 # Omit the header of the output
  - cat a.txt | sed '2**a** coderalo' # Print a.txt with ‘coderalo’ inserted next to line 2 (on line 3)
  - cat a.txt | sed '2**i** coderalo' # Insert before line 2
  - cat a.txt | sed '**s**/apple/banana/**g**' # Replace all “apple” to “banana”
- **grep** and **sed** are much more powerful with **regex**!

Regular Expression

- Regular expression (regex) is an expression method describing some form of texts. *(Well, you'll learn more about regex in the course "Formal languages and automata theory")*
- There are several standards of regex: e.g. IEEE POSIX standard has three sets of compliance: **BRE** (Basic Regular Expressions), ERE (Extended Regular Expressions), and SRE (Simple Regular Expressions, deprecated)
- By default, grep use BRE, so here we'll go through the syntax of BRE.

Regular Expression

Single character, e.g. a, b, c	Matches itself
.	Matches **any character**
^	**Matches the beginning** of the pattern space, such as the beginning of a file
$	Like ^, but **matches the end**
[list]	Matches **any of the character in the list**
[^list]	Matches any of the character **not** in the list

Regular Expression

?	The preceding item is optional and matched **at most once**.
*	The preceding item will be matched **zero or more** times.
+	The preceding item will be matched **one or more** times.
{n}	The preceding item is matched n or more times.
{n,}	The preceding item is matched n or more times.
{,m}	The preceding item is matched at most m times. This is a GNU extension.
{n,m}	The preceding item is matched at least n times, but not more than m times.

Regular Expression

- Some regex examples:
  - ‘a\{3\}b’ -- matches ‘aaab’
  - ‘^ap’ -- matches all words starting by “ap”
  - ‘.*’ -- matches all strings
  - ‘le$’ -- matches all words ending by “le”
  - ‘\$’ -- matches for a dollar sign (\ for escaping)
- Examples with grep and sed:
  - cat file.txt | sed ‘s/a\{2\}b/bba/g’ # replace all ‘aab’ to ‘bba’
  - cat file.txt | sed ‘s/\/\/.*$//g’ # replace all strings starting with // to empty lines
  - Cat file.txt | grep “[^]” # omit all empty lines
- *You can find a bunch of examples about them, they might be useful in HW1!*

Exercise

- Material: https://www.csie.ntu.edu.tw/~coderalo/mirrorlist.txt
- Get the RTT information: ping -c 3 -q $url
- You should list the mirrors in ascendant order according to the average RTT.