Week 1

Shell(slide)

prompt

$ You are in a shell, most likely POSIX (sh) compatible.

You are in a root shell. With great power comes great responsibility.

% You are probably in the C shell.

> You are on a continuation line e.g. inside a string.

builtins

```
$ which ls
/bin/ls
$ which cd
$
```

'which': to find where the command store in.
Because 'cd' is build in command so which can not find it(but actually can found, it in /usr/bin/cd).

Shell expansion
'?' single character in filename
e.g. image???.jpg matches

cp [-rfi] SRC... DEST copy files
-r recursive
-f overwrite readonly
-i ask before overwriting (interactive)
mv [–nf] SRC... DEST move files
-n no overwrite

-f force overwrite
e.g. mv hibro testdir (testdir is a directory)
(also could use to re-name e.g. mv hello.c hi.c will rename hello.c to hi.c)

Pipe
standard input/output
Internally, programs read (fd, buffer, size) and write (fd, buffer, size).
Each program starts with three file descriptors open:
0 = standard input
1 = standard output
2 = standard error

pipe

$ ls -1 | head
head [-n NUM] e.g. ls -1 | head -n -2 will show first two line
tail [-n NUM]

grep [-iv] EXPRESSION reads standard input and prints only lines that match the regular expression to standard output. With -i it is case-insensitive, and with -v it only prints lines that do not match the expression.
grep (flag: -i 忽视大小写; -v 反向匹配; -E extension grep)

redirect
$ cat infile | sort > outfile will first sort the infile then put the output into outfile
$ sort < infile > outfile do the same job.
$ COMMAND > FILE overwrites FILE
$ COMMAND >> FILE appends to FILE

error redirect
$ COMMAND > FILE 2> FILE2 will put error message to FILE2
$ COMMAND > FILE 2>&1 will put error message to standard output.
ignore output:
$ COMMAND > /dev/null

tee
$ ls | tee FILE will put the ls output into FILE

subshell to argument
$ COMMAND $(SOMETHING)
$ echo $(echo Hi | sed –e s/Hi/Hello/)

Regular expressions (正则表达)

? 匹配文件名中的 0 个或 1 个字符， * 匹配零个或多个字符，+匹配一个或多个

^[a-zA-Z0-9_-]{3,15}$

^ 表示匹配字符串的开头。 [a-zA-Z0-9_-] 表示字符集，包含小写字母、大写字母、数字、下划线和连接字符 -。{3,15} 表示前面的字符集最少出现 3 次，最多出现 15 次，从而限制了用户名的长度在 3 到 15 个字符之间。$ 表示匹配字符串的结尾

File System

/etc/passwd 文件包含了系统中每个用户的基本信息，包括用户名、用户 ID、主目录、登录 shell 等。

/etc/group 文件包含了系统中每个用户组的信息，包括组名、组 ID、组成员等。

`/bin` stands for binaries, that is programs that you can run.

`/usr` is a historical accident and a bit of a mess.

In the earliest days:

/bin was only for binaries needed to start the system - or at least the most important binaries that needed to live on the faster of several disk drives, like your shell.

/usr/bin was where most binaries lived which were available globally, for example across all machines in an organization.

/usr/local/bin was for binaries installed by a local administrator, for example for a department within an organization.

/etc stores system-wide configuration files and typically only root (the administrator account) can change things in here. For example, system wide SSH configuration lives in /etc/ssh

/lib contains dynamic libraries - windows calls these .dll files, POSIX uses .so. For example, /lib/x86_64-linux-gnu/libc.so.6 is the C library, which allows C programs to use functions like printf.

/home is the folder containing users' home directories, for example the default user vagrant gets /home/vagrant. The exception is root, the administrator account, who gets /root.

/sbin (system binaries) is another collection of programs, typically ones that only system administrators will use. For example, fdisk creates or deletes partitions on a disk and lots of programs with fs in their name deal with managing file systems. /sbin/halt, run as root (or another user that you have allowed to do this), shuts down the system; there is also /sbin/reboot.

`/tmp` is a temporary filesystem that may be stored in RAM instead of on disk (but swapped out if necessary), and that does not have to survive rebooting the machine.

`/var` holds files that vary over time, such as logs or caches.

`/dev`, `/sys` and `/proc` are virtual file systems. One of the UNIX design principles is that almost every interaction with the operating system should look to a program like reading and writing a file, or in short *everything is a file*. For example, `/dev` offers an interface to devices such as hard disks (`/dev/sda` is the first SCSI disk in the system, and `/dev/sda1` the first partition on that), memory (`/dev/mem`), and a number of pseudoterminals or ttys that we will talk about later.
`/proc` provides access to running processes;
`/sys` provides access to system functions. For example, on some laptop systems, writing to `/sys/class/backlight/acpi`

In any case, `/usr` and its subfolders are for normally read-only data, such as programs and configuration files but not temporary data or log files.

This indicates the file type, green is an executable program, blue is a link to another file

If you ever really need a root shell then `sudo bash` gets you one - with `#` instead of `$` as prompt to warn you that you are working as root.
sudo apt update fetches the new package list from the repository. This way, apt can tell you if any packages have been updated to new versions since you last checked.

sudo apt upgrade upgrades every package that you already have installed to the latest version in your local package list (downloaded when you do an apt update).

Week3 Git

git branch (use to check what branch appear in repository) (flag: -a show all branch, remote and local; -r show remote branch; -d will delete the branch with branch name 'git branch -d <branch-name>'). * (and green colour in terminal) mean the current branch you in.

git checkout -b <new branch-name> will create a new branch in repository.

git push --set-upstream origin develop This adds an "upstream" entry on the local develop branch to say that it is linked to the copy of your repository called `origin`, which is the default name for the one you cloned the repository from.

Say you'd like to go back to how the code was before the last commit: git checkout HEAD~1

Say a commit was a horrible mistake and you'd like to apply it in reverse and undo all the changes of it: git revert HEAD(6 digits HASH)
git revert HASH adds a new commit that returns the files to the state they were before the commit with the given hash.

git reset HASH undoes commits by moving the HEAD pointer back to the commit with the given hash, but leaves the working copy alone (you can use the --hard option to change the files as well).

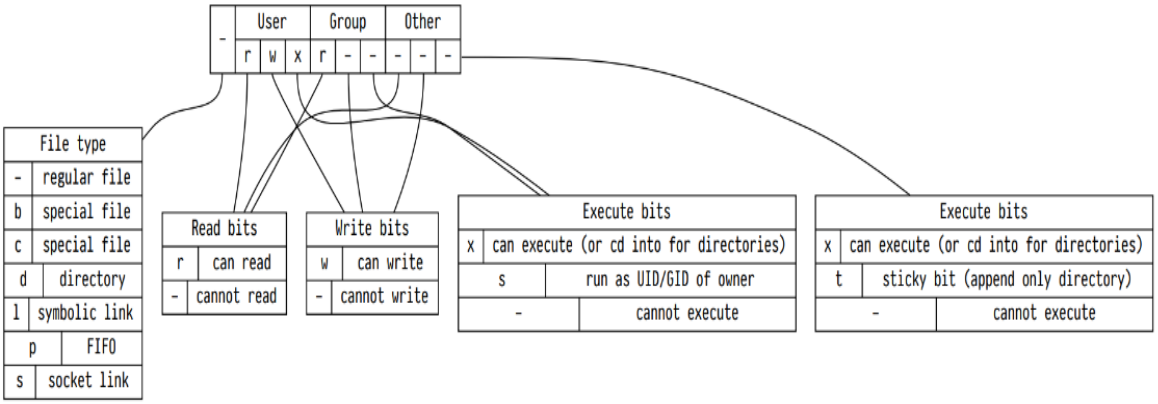
Ignore files

So we want to tell git to ignore the program and changes in it, which we do by creating a file called .gitignore and adding an expression on each line to say which file(s) or folders to ignore - you can use *.o to select all object code files, for example.

Create a file .gitignore and add the single line program to it.
Do another git status and notice that while the program is now ignored, the ignore file is marked as new. This file does belong in the repository, so add it and commit it.
Check that git status reports clean again, and that git log contains two commits.

Week 4

Permissions



su <user-name> use to change the user.
sudo add <user-name> to add user

chmod g+r / g-r <directory> (i.e.+: add/ -:remove)
u = owner; g= group; o = others; a = all; r for read; w for write; x for execute

Shell script

Start the file with the shebang #! then the path to the interpreter of the script plus any arguments:
For portable POSIX shellscripts #! /bin/sh/
For less portable BASH scripts #! /usr/bin/env bash

Basic Syntax
Shell scripts are written by chaining commands together
A; B run A then run B
A | B run A and feed its output as the input to B
A && B run A and if successful run B
A || B run A and if not successful run B
Programs return a 1 byte exit value (e.g. C main ends with return 0;)

- This gets stored into the variable ${?} after every command runs.
- 0 indicates success (usually)
- >0 indicates failure (usually)

[$? -eq 0] # works
[$? -eq 0] # doesn't work

Different shells
(Just use bash unless you care about extreme portability in which case use POSIX sh)
Typical Shells
sh POSIX shell
bash Bourne Again shell (default on Linux)
zsh Z Shell (default on Macs), like bash but with more features
ksh Korne shell (default on BSD)
Other shells
dash simplified faster bash, used for booting on Linux
Busybox sh simplified bash you find on embedded systems
Weird shells
fish More usable shell (but different incompatible syntax)
elvish Nicer syntax for scripting (but incompatible with POSIX)
nushell Nicer output(but incompatible)

Variable

To create a variable:
GREETING="Hello World!"
(No spaces around the =)
To use a variable
echo "${GREETING}"
If you want your variable to exist in the programs you start as an environment variable:
export GREETING
To get rid of a variable
unset GREETING

:? use to find the variable is set or not(e.g. ${GREETING:? “ERROR, variable not set”} will output an error message if GREETING is null/not set)

Standard variable

${0} Name of the script
${1}, ${2}, ${3} … ${n} Arguments passed to your script
${#} The number of arguments passed to your script
${@} and ${*} All the arguments

if, for loop and case statement

```
if [condition1]; then
    #command
elif[condition2]; then
    #command
else
    #command
fi

for variable in list; do
    #command
done

case expression in
    pattern1)
        #command\
        ;;
    pattern2)
        #command
```

```
        ;;
    *)
        #command
        ;;
esac
```

case statement will choose one command to run based on expression.

basename and dirname

basename use to find the file name, dirname use to find where the file locate in.

```
filePath="/var/log/myapp/application.log"
fileName=$(basename "$filePath")
```

`scriptDir=$(dirname "$0")` ($0 is the file itself)

remove file extension:
“${BASE}.c” will remove extension in variable BASE

Week4 Debug

Need use flag -g and -Og when compile file if wanna use gdb

```
gcc -Og hibro.c -g -o hibro
```