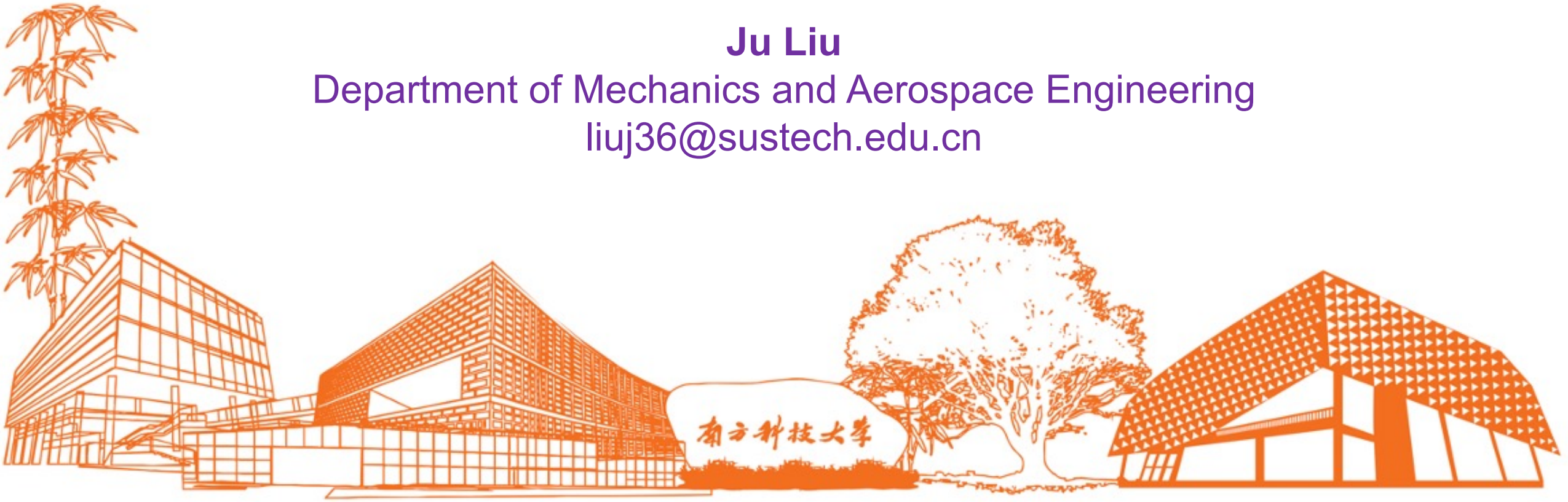
MAE 5032 High Performance Computing: Methods and Applications

Lecture 2: Unix/Linux

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn



Unix and Linux

UNIX
In early days, every computer has a different operating system.

In 1969, a team of developers in Bell Labs started working on this problem.

They develop a new OS (UNIX):

- Simple and elegant
- Written in C language insteady of assembly code
- Able to be recycled (meaning change only the kernel)

Inside view of the Bell Lab

Unix and Linux

Linus Torvalds, a young man studying computer science at the university of Helsinki, thought it would be a good idea to have some sort of freely available academic version of UNIX, and promptly start to code.

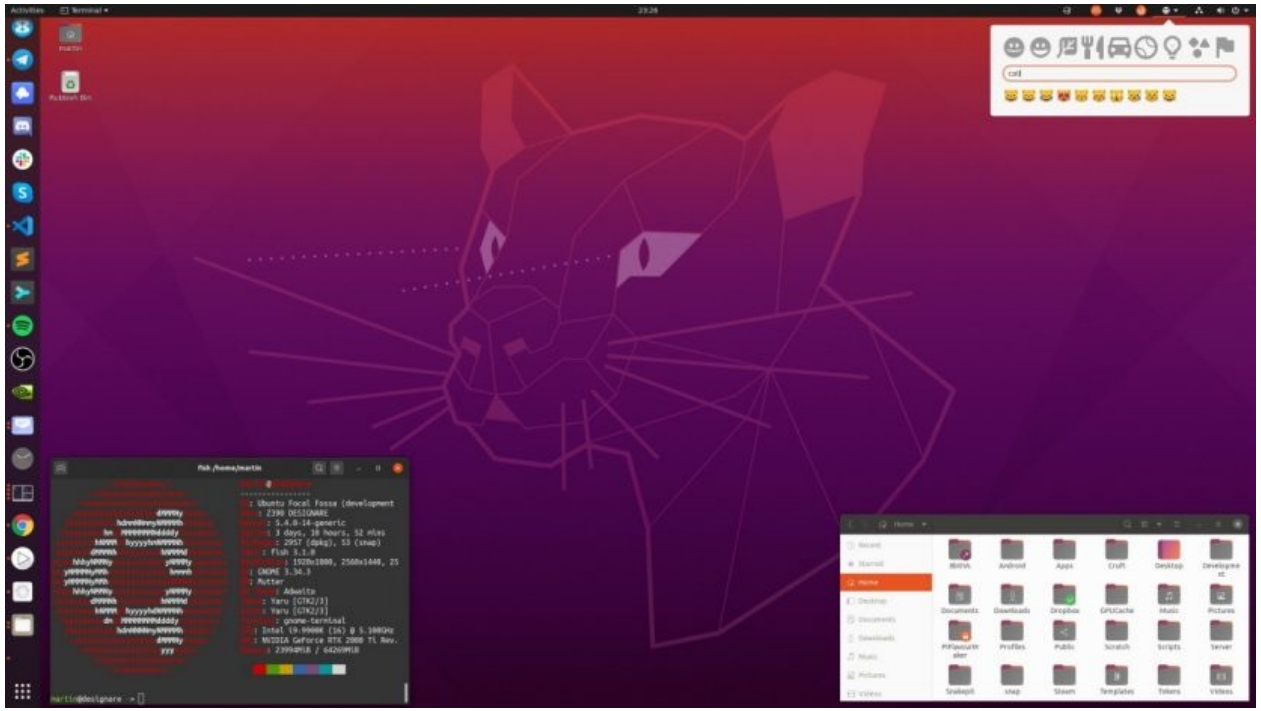
- July 1991 : Linus Torvalds from Helsinki started his hobby: Linux
- Oct. 1991 : version 0.02
- 2002 : version 2.2
- Today : look at www.kernel.org

Linux is an ideal operating system for programmers (everything a good programmer can wish is available : compilers, libraries, …)

In this class, UNIX = Linux.

Linux for non-experienced users

- Linux has distributions like Windows : RedHat, Scientific Linux, Ubuntu, etc.
- It has beautiful graphic user interface.
- Each distribution contains application softwares
- Office : openoffice
- IE : Mozilla
- etc.



GNU project

The GNU project was lunched in 1984 with the goal of developing a completely UNIX style operating system which is free.

A list of GNU software:

- GCC : The GNU C Compiler
- Gnome : The GNU desktop environment
- Emacs : A text editor
- The Gimp : GNU Image Manipulation Program
- In 2012, a fork of Linux kernel became part of GNU project.

GNU sounds like gnu (connochaetes)

Open Source

Open Source means:

User can see binary and source

Users are encouraged to read and understand the source code

Users can debug and recompile source

Closed Source means:

Uers are only given binaries

Uers cannot read the source

Uers cannot recompile the code

Linux

Advantages:

- Linux is free and more secure than Windows
- Open sources
- Necessary for HPC
- Rare for a Linux system to slow down or crashes
- Easy to debug codes

Disadvantages:

- Not very user friendly and learning curve can be sharp for beginners
- Is an open source product trustworthy?

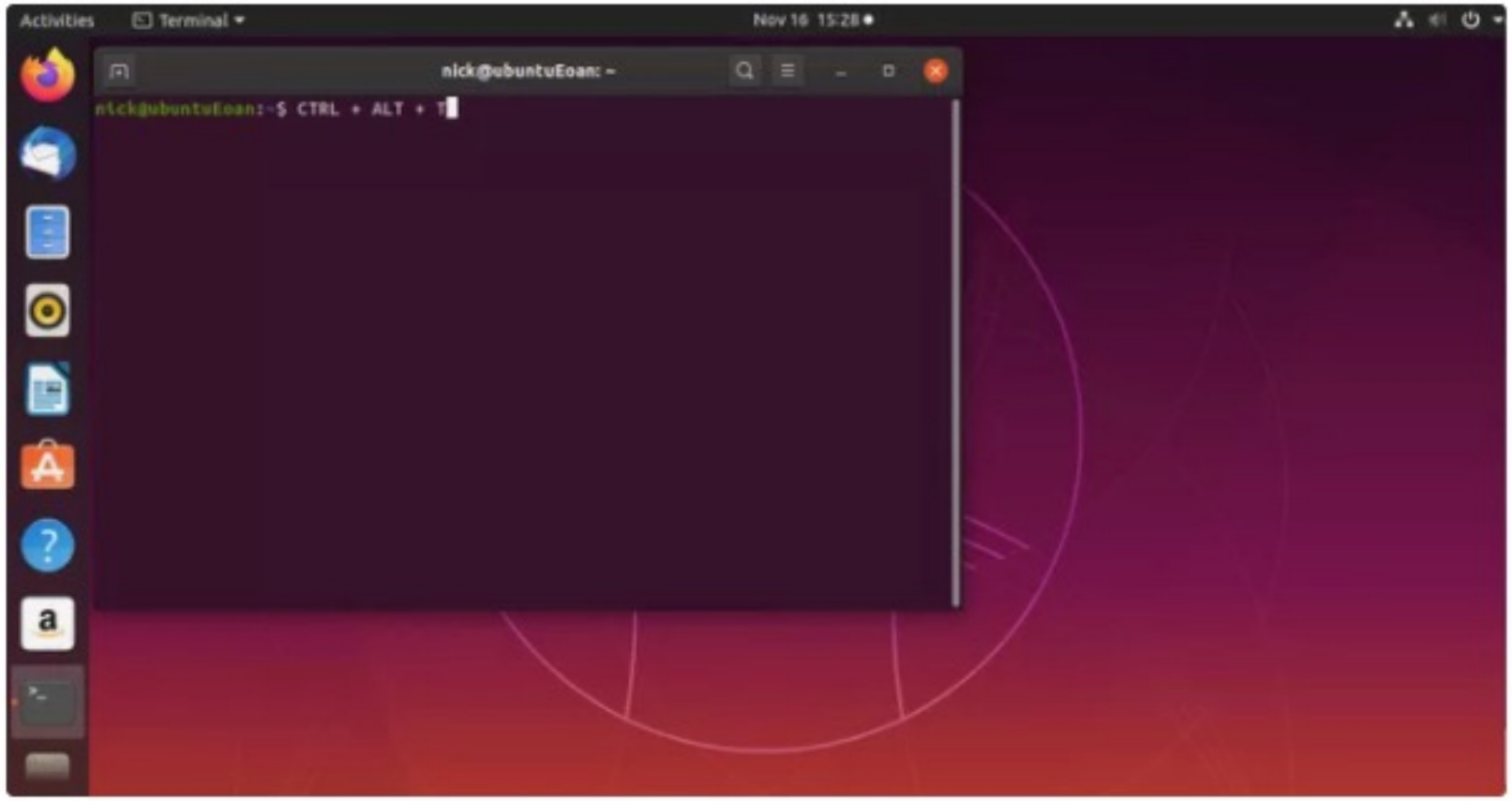
Outline

1. Basic Commands
2. File attributes and permissions
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

Open a terminal

01 Open a Linux Terminal Using Ctrl+Alt+T

of 05



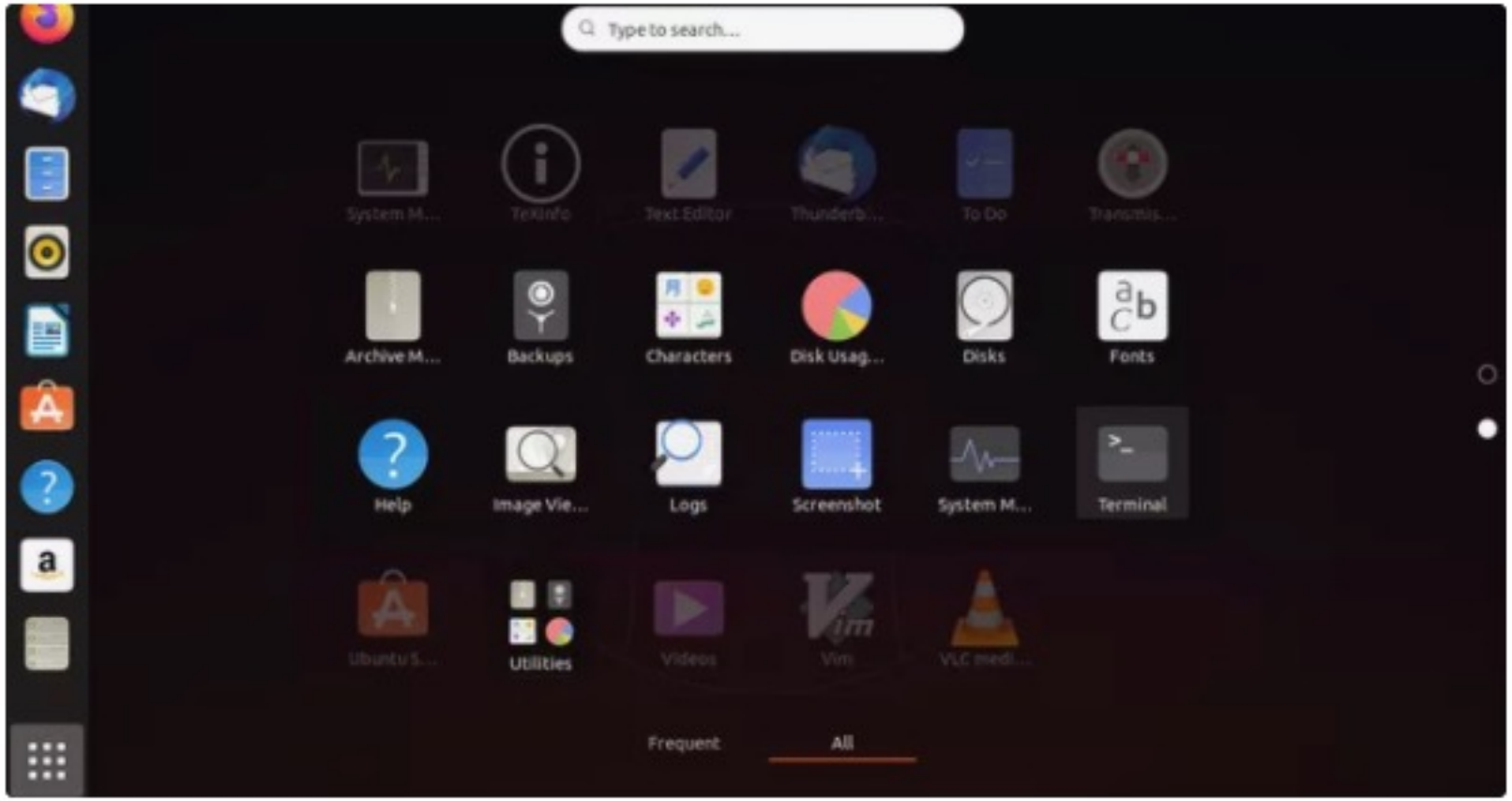
Open a terminal

02 of 05 Search Using the Ubuntu Dash

Open a terminal

03 Navigate the GNOME App Launcher

of 05



You will
know
the truth,
and
the truth
will
set you free.
John 8:32

Basic Unix Commands

Unix Command	what it does
`ls`	list directory contents
`cd`	change directory
`df`	display free disk space
`cat`	concatenate and print files
`more (less)`	a filter for paging through text one screenful at a time.
`head`	display first lines of a file
`pwd`	return working directory name
`cp`	copy files
`awk`	a pattern-directed scanning and processing language

Basic Unix Commands

Unix Command	Function
`rm`	remove directory entries
`chmod`	change file modes or Access Control Lists
`tail`	display the last part of a file
`touch`	change file access and modification times
`mkdir`	make directories
`rmdir`	remove directories
`find`	walk a file hierarchy
`grep`	print lines matching a pattern
`chown/chgrp`	change file owner and group

`ls`: list directory contents

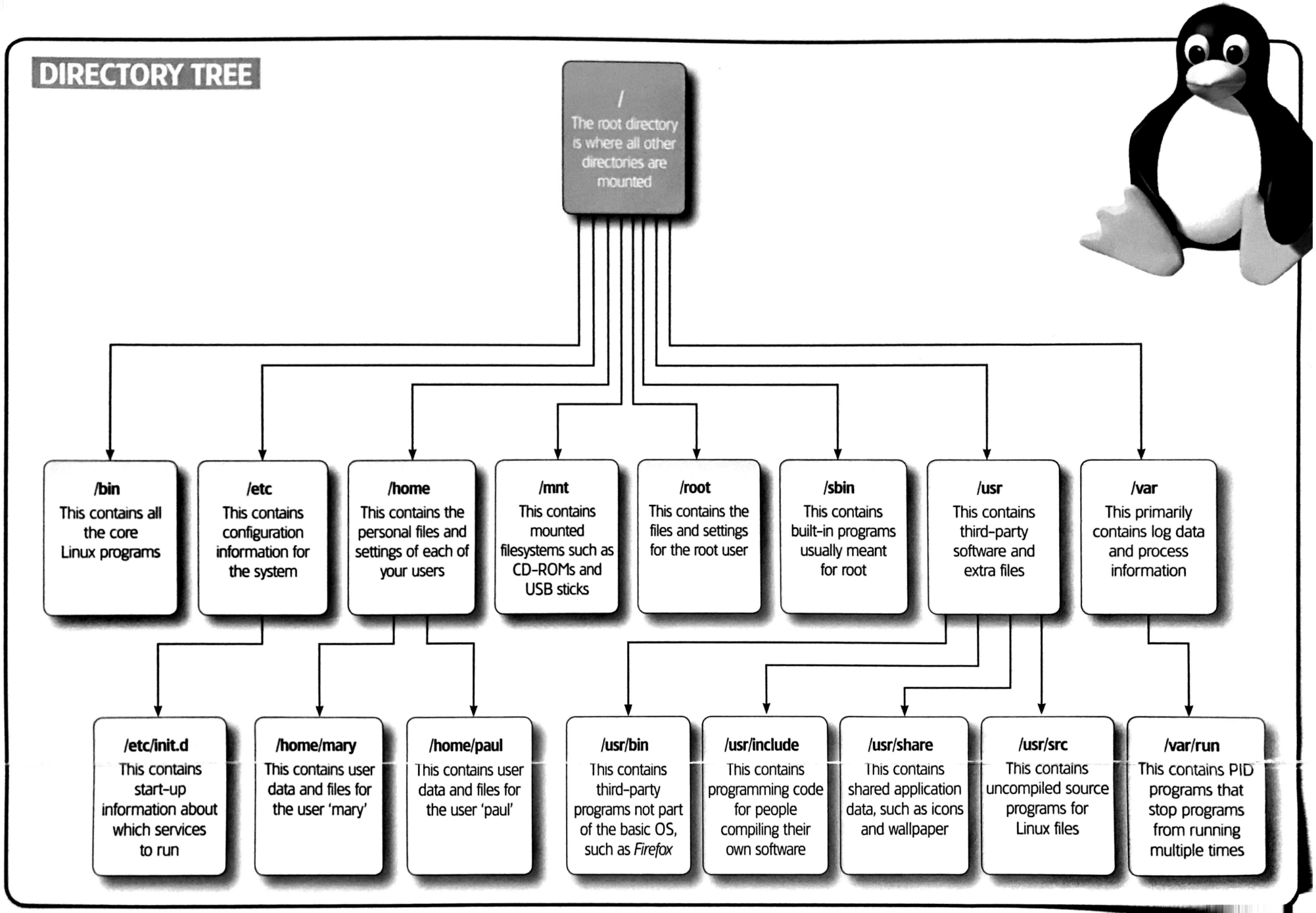
```
ls [-options] [names]
```

- The brackets around **`options`** and **`names`** in the general form of the **`ls`** command means that something is optional
- This type of description is common in the documentation for Unix commands
- To use command line options precede the option letter(s) with a hyphen
- The **`ls`** command supports many options

`ls`	list files in current directory
`ls /`	list files in the root directory
`ls .`	list files in the current directory
`ls ..`	list files in the parent directory
`ls /usr`	list files in the directory /usr
`ls -l .`	long format (include file times, owner and permissions)
`ls -a .`	all (shows hidden files as well as regular files)
`ls -F .`	include special char to indicate file types

In Unix, hidden files have names that start with " **.** "

- Absolute Path : path that starts from the root directory (/)
- Relative Path : path that does NOT start from the root directory


DIRECTORY TREE
/
The root directory is where all other directories are mounted
/bin
This contains all the core Linux programs
/etc
This contains configuration information for the system
/home
This contains the personal files and settings of each of your users
/mnt
This contains mounted filesystems such as CD-ROMs and USB sticks
/root
This contains the files and settings for the root user
/sbin
This contains built-in programs usually meant for root
/usr
This contains third-party software and extra files
/var
This primarily contains log data and process information
/etc/init.d
This contains start-up information about which services to run
/home/mary
This contains user data and files for the user 'mary'
/home/paul
This contains user data and files for the user 'paul'
/usr/bin
This contains third-party programs not part of the basic OS, such as Firefox
/usr/include
This contains programming code for people compiling their own software
/usr/share
This contains shared application data, such as icons and wallpaper
/usr/src
This contains uncompiled source programs for Linux files
/var/run
This contains PID programs that stop programs from running multiple times

cd: change directories

```
lslogin$ ls
foo.c        mydata      typescript
lslogin$ cd mydata
lslogin$ pwd
/Users/sunit/workdir/mydata
lslogin$ ls
backup.fasta data.fasta    moredata
lslogin$ cd ..
lslogin$ pwd
/Users/sunit/workdir
```

pwd: print working directory name

"**cd**" with no arguments takes you to your HOME directory.

`cat`: concatenate and print files

```
lslogin$ cat data.fasta
>c01_009 499 amino acids  MW=55632 D  pI=5.38 numambig=0
MPGGFILAIDEGTTSARAIIYNQDLEVLGIGQYDFPQHYPSPGYVEHNPDEIWNAQMLAI
KEAMKKAKIESRQVAGIGVTNQRETTILWDAISGKPIYNAIVWQDRRTSNITDWLKENYF
GMIKDKTGLIPDPYFSGSKIKWILDNLPNVRSKAEKGEIKFGTIDTYLIWKLTNGKIHVT
IGITRGTTKAHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQAD
ILGIRVVRPKVMETASMGVAMLAGLAINYWNSLNELKQKWTVDKEFIPSINKEERERRYN
AWKEAVKRSLGWEKSLGSK*
```

`more`: display a file's contents one screen at a time

```
lslogin$ more data.fasta
>c01_009 499 amino acids  MW=55632 D  pI=5.38 numambig=0
MPGGFILAIDEGTTSARAIIYNQDLEVLGIGQYDFPQHYPSPGYVEHNPDEIWNAQMLAI
KEAMKKAKIESRQVAGIGVTNQRETTILWDAISGKPIYNAIVWQDRRTSNITDWLKENYF
GMIKDKTGLIPDPYFSGSKIKWILDNLPNVRSKAEKGEIKFGTIDTYLIWKLTNGKIHVT
IGITRGTTKAHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQAD
ILGIRVVRPKVMETASMGVAMLAGLAINYWNSLNELKQKWTVDKEFIPSINKEERERRYN
AWKEAVKRSLGWEKSLGSKLPNVRSKAEKGEIKFGTIDTYLIWKLTNGRDVIEIMEKESG
TKINILKVDGGGAKDNLLMQFQILDNLPNVRSKAEKGEIKFGTIDTYLIWKLTNGTSAIG
AHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQADTDWLKENYF
LGWEKSLGSKLPNVRSKAEKGEIKFGTIDTYLIWKLTNGRDVIEIMIKFGTIDTYLIWTG
GMIKDKTGLIPDPYFSGSKIKWILDNLPNVRSKAEKGEIKFGTIDTYLIWKLTNGKIHVT
IGITRGTTKAHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQAD
DKTGLIPDPYFSGSKIKWILDNLPNVRSKAEKGEIGVAMLAGLAINYWNSLNELKFKIHE
NILKVDGGGAKDNLLMQFQILDNLPNVRSKAEKGEIKFGTIDTYLFQFQILDNLPNVRSK
AEKGEIKFGTIDTYLIWILDVIEIMEKESGTKINILKVDGGGKVDGGIAYQNRDVIEIME
LWDAISGKPIYNDNLPNVRSKAEKGEIGVAMLAGLAINNLPNVRSKAEKGEIGKGEIGVA
--More--(75%)
```

Notes: hit <space> to see the next page
hit "q" to quit, "/" to search, read the man page.
"less" is an enhanced version of "more" on Linux

`head` & `tail`: display first or last lines of a file

- head displays the top of a file
  - head -n displays the top n lines
  - default is 10
- tail displays the bottom of a file
  - tail -n displays the bottom n lines
  - default is 10
  - tail +n displays the file starting at line n

tail –f output_file

head & **tail**: display first or last lines of a file

tail –f output_file

-f The **-f** option causes **tail** to not stop when end of file is reached, but rather to wait for additional data to be appended to the input. The **-f** option is ignored if the standard input is a pipe, but not if it is a FIFO.

cp: copy files

```
lslogin$ ls
bin/            my_data/          data.fasta
lslogin$ cp data.fasta backup.fasta
lslogin$ ls
bin/            my_data/          data.fasta      backup.fasta
lslogin$
```

copy one or multiple files to a directory : cp [options] source1 source2 source3 … directory

copy a file : cp source destination

cp: copy files

```
lslogin$ ls
bin/            my_data/          data.fasta
lslogin$ cp data.fasta backup.fasta
lslogin$ ls
bin/            my_data/          data.fasta      backup.fasta
lslogin$
```

use the –r option to copy recursively (for folders)

mv: move or rename files

```
lslogin$ ls
bin/            my_data/          data.fasta      backup.fasta
lslogin$ mv backup.fasta my_data
lslogin$ ls
bin/            my_data/          data.fasta
lslogin$ ls my_data
backup.fasta
lslogin$ mv data.fasta Dec15.fasta
lslogin$ ls
bin/            my_data/          Dec15.fast
```

mv [options] source destination

if source and destination are in the same folder, mv changes the name

`rm`: deletes files - *permanently*

```
lslogin$ ls
bin/            my_data/          data.fasta      backup.fasta
lslogin$ rm backup.fasta
lslogin$ ls
bin/            my_data/          data.fasta
```

`rm -i` : interactive, ask for confirmation
`rm -r` : recursively remove directories
`rm -f` : override lack of write permission

Note: There is no Recycle bin or Undelete Key!
Thou shalt know what thou are doing...

`mkdir`: make directories

```
lslogin$ ls
my_data      data.fasta      backup.fasta      prog
lslogin$ ls -F
my_data/      data.fasta      backup.fasta      prog*
lslogin$ alias ls 'ls -F'
lslogin$ ls
my_data/      data.fasta      backup.fasta      prog*
lslogin$ mkdir new_data
lslogin$ ls
my_data/      new_data/      data.fasta      backup.fasta    prog*
```

UNIX Commands: groups

```
lslogin$ id
uid=1003(rob) gid=10(staff)

lslogin$ groups
staff sysadmin wwwdevel
```

Use **groups** to see all the group names to which you belong.

grep: extracts lines from a file that match a given string or pattern

```
lslogin$ cat sequence.fas
>c01_009 499 amino acids  MW=55632 D  pI=5.38 numambig=0
MPGGFILAIDEGTTSARAIIYNQDLEVLGIGQYDFPQHYPSPGYVEHNPDEIWNAQMLAI
KEAMKKAKIESRQVAGIGVTNQRETTILWDAISGKPIYNAIVWQDRRTSNITDWLKENYF
DYSNASRTMLFNINKLEWDREILELLKIPESILPEVRPSSDIYGYTEVLGSSIPISGDAG
DQQAALFGQVAYDMGEVKSTYGTGSFILMNIGSNPIFSENLLTTIAWGLESKRVTYALEG
SIFITGAAVQWFRDGLGREPKICKSBUTTASVPDTGGVYFVPAFVGLGAPYWDPYARGLI
IGITRGTTKAHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQAD
ILGIRVVRPKVMETASMGVAMLAGLAINYWNSLNELKQKWTVDKEFIPSINKEERERRYN
AWKEAVKRSLGWEKSLGSK*

lslogin$ grep AA sequence.fas  Search in sequence.fas file for AA
DQQAALFGQVAYDMGEVKSTYGTGSFILMNIGSNPIFSENLLTTIAWGLESKRVTYALEG
SIFITGAAVQWFRDGLGREPKICKSBUTTASVPDTGGVYFVPAFVGLGAPYWDPYARGLI
```

grep: extracts lines from a file that match a given string or pattern

Options	Description
`-i`	Ignore case distinctions on Linux and Unix
`-w`	Force PATTERN to match only whole words
`-v`	Select non-matching lines
`-n`	Print line number with output lines
`-h`	Suppress the Unix file name prefix on output
`-r`	Search directories recursivly on Linux
`-R`	Just like -r but follow all symlinks
`-l`	Print only names of FILEs with selected lines
`-c`	Print only a count of selected lines per FILE
`--color`	Display matched pattern in colors

Examples:

grep –r dander /home/dan

grep –n danger example.txt

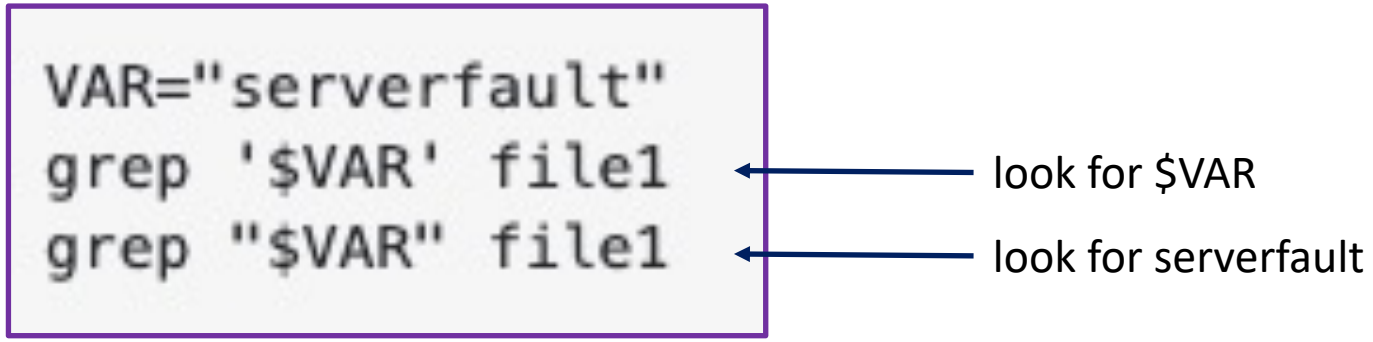
grep –i DanGer example.txt

grep: extracts lines from a file that match a given string or pattern

Usage of quotes

1. grep hello world file1 will look for word “hello” in files named world and file1.
2. grep “hello world” file1 will look for “hello world” in file1.

The choice between single and double quotes is only important if the search string contains variables to be evaluated.



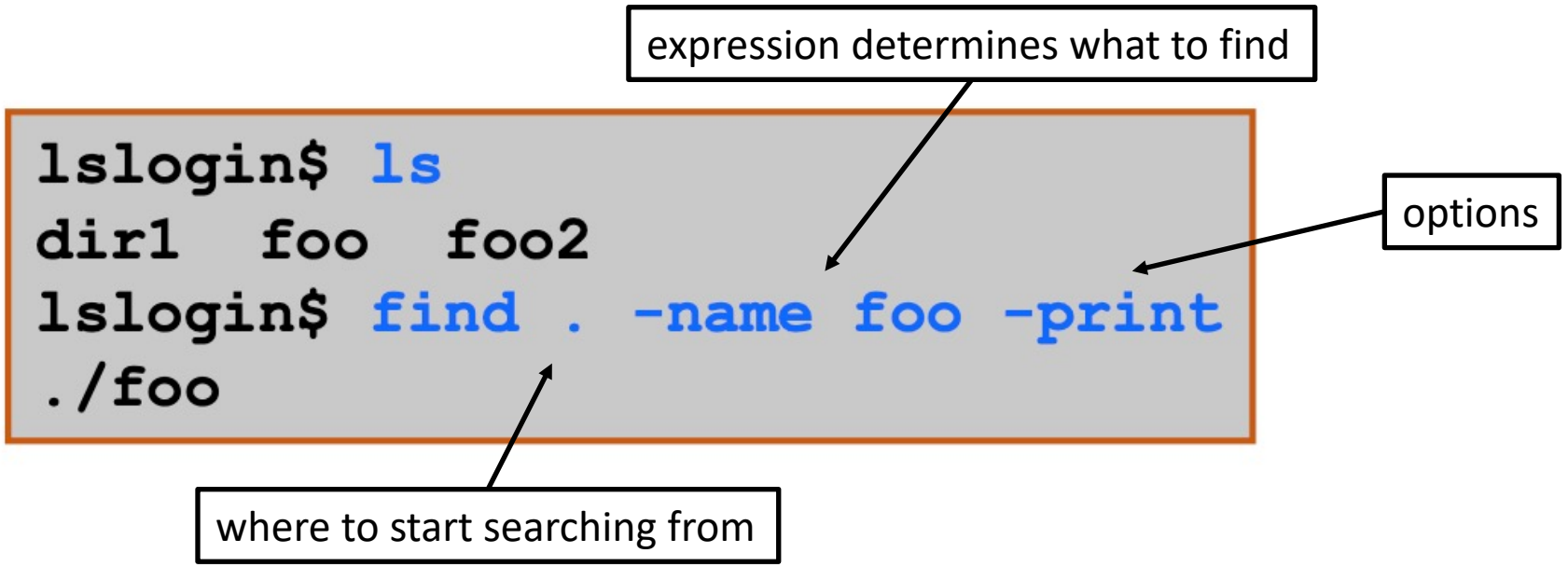
grep: cont.

```
lslogin$ cat sequence.fas
>c01_009 499 amino acids  MW=55632 D  pI=5.38 numambig=0
MPGGFILAIDEGTTSARAIIYNQDLEVLGIGQYDFPQHYPSPGYVEHNPDEIWNAQMLAI
KEAMKKAKIESRQVAGIGVTNQRETTILWDAISGKPIYNAIVWQDRRTSNITDWLKENYF
DYSNASRTMLFNINKLEWDREILELLKIPESILPEVRPSSDIYGYTEVLGSSIPISGDAG
DQQAALFGQVAYDMGEVKSTYGTGSFILMNIGSNPIFSENLLTTIAWGLESKRVTYALEG
SIFITGAAVQWFRDGLGREPKICKSBUTTASVPDTGGVYFVPAFVGLGAPYWDPYARGLI
IGITRGTTKAHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQAD
ILGIRVVRPKVMETASMGVAMLAGLAINYWNSLNELKQKWTVDKEFIPSINKEERERRYN
AWKEAVKRSLGWEKSLGSK*

lslogin$ grep '[ST].[RK]' sequence.fas
MPGGFILAIDEGTTSARAIIYNQDLEVLGIGQYDFPQHYPSPGYVEHNPDEIWNAQMLAI
KEAMKKAKIESRQVAGIGVTNQRETTILWDAISGKPIYNAIVWQDRRTSNITDWLKENYF
DQQAALFGQVAYDMGEVKSTYGTGSFILMNIGSNPIFSENLLTTIAWGLESKRVTYALEG
IGITRGTTKAHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQAD
```

a regular expression search

`find`: walk a file hierarchy

- At its simplest, find searches the filesystem for files whose name matches a specific pattern
- However, it can do a lot more and is one of the most useful commands in Unix (as it can find specific files and then perform operations on them)



`find`: walk a file hierarchy

```
1   $find ./A1 -name file1.txt
```

```
[vaishali@localhost ~]$ find ./A1 -name file1.txt
./A1/B2/file1.txt
[vaishali@localhost ~]$
```

```
1   $find -name '*.txt'
```

```
[vaishali@localhost B2]$ find -name '*.txt'
./file1.txt
./vaishali.txt
./file.txt
[vaishali@localhost B2]$
```

man: manual of Linux

man [command]

display the whole manual page of a particular command

command name

syntax

all options

```
LS(1)                          User Commands                          LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information about the FILEs (the current directory by default).  Sort entries alpha-
       betically if none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              scale sizes by SIZE before printing them; e.g., '--block-size=M'  prints  sizes  in
              units of 1,048,576 bytes; see SIZE format below
```

Hit enter to page down
Hit q to exit

which: locate the executable file

which [-a] command

display the absolute path of an executalbe file

```
-> which ls
/bin/ls
```

-a : display all matching exectuables, not just the first

```
-> which -a mpicc
/Users/juliu/lib/mpich-3.3.2/bin/mpicc
/usr/local/bin/mpicc
```

PATH variable

PATH is ***not*** a command; instead, it is an ***environemental variable*** of the Linux/Unix system that tells the directories to search for executable files.

To view an environmental variable, do

echo $PATH

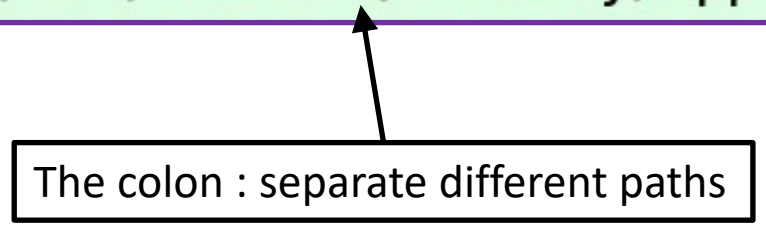
echo : prints whatever follows it on screen

$: get the value of the variable name immediately follows it.

```
-> echo $PATH
/Users/juliu/lib/mpich-3.3.2/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:
/sbin:/Library/TeX/texbin:/Library/Apple/usr/bin
```

PATH variable

```
-> echo $PATH
/Users/juliu/lib/mpich-3.3.2/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:
/sbin:/Library/TeX/texbin:/Library/Apple/usr/bin
```



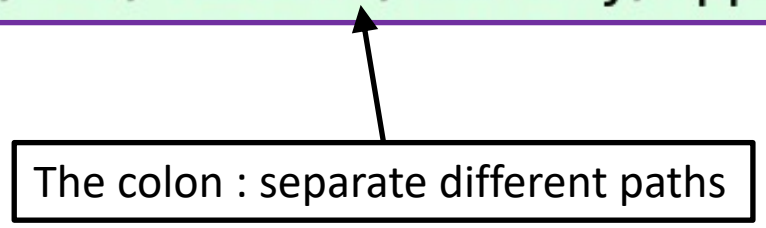
The system will search the given command starting from the first directory in PATH, and will exectuable the first one it found.

```
-> which -a mpicc
/Users/juliu/lib/mpich-3.3.2/bin/mpicc
/usr/local/bin/mpicc
```

```
-> which mpicc
/Users/juliu/lib/mpich-3.3.2/bin/mpicc
```

PATH variable

```
-> echo $PATH
/Users/juliu/lib/mpich-3.3.2/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:
/sbin:/Library/TeX/texbin:/Library/Apple/usr/bin
```



This can be useful is you want to execute your own executable file instead of the system default one.

```
-> export PATH=~/build/build_linux_shell_scripts:$PATH
```

change your PATH variable

1. Go to your home directory
2. Display all files (including the hidder file)
3. Make sure the file .bashrc exists
4. Open .bashrc by a text editor
5. Add `export PATH=your_desired_path:$PATH` into the file
6. Save and exit the editor
7. Run `source ~/.bashrc`

Tai-Yi and Qi-Ming Login

- SSH only
- UNIX/Linux users

  ssh username@172.18.6.175

  ssh username@172.18.6.67

```
$ ssh user@host

The authenticity of host 'host (12.18.429.21)' can't be established.

RSA key fingerprint is 98:2e:d7:e0:de:9f:ac:67:28:c2:42:2d:37:16:58:4d.

Are you sure you want to continue connecting (yes/no)?
```

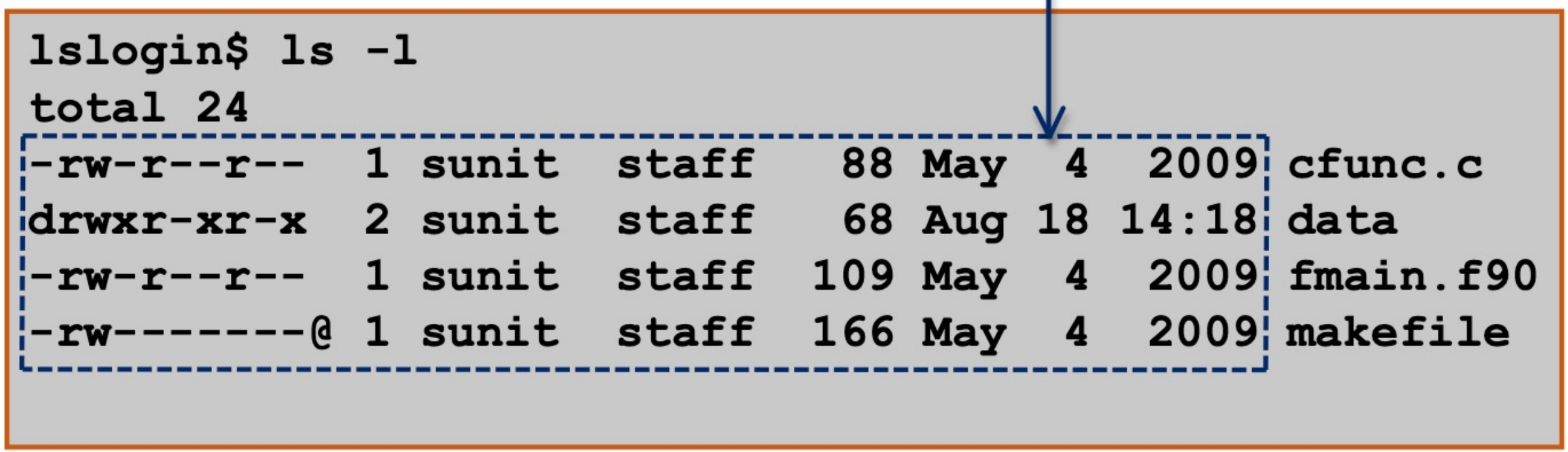
the first time when you login, you will be asked for security confirmation.

Outline

1. Basic Commands
2. File attributes and permissions
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

File Attributes

The **`ls -l`** command (long listing) displays specific attributes about each file or directory.



```
lslogin$ ls -l
total 24
-rw-r--r--   1 sunit   staff    88 May   4   2009 cfunc.c
drwxr-xr-x   2 sunit   staff    68 Aug 18 14:18 data
-rw-r--r--   1 sunit   staff   109 May   4   2009 fmain.f90
-rw-------@ 1 sunit   staff   166 May   4   2009 makefile
```

File Attributes cont.

Every file has a specific list of attributes:

- Access Times
  - when the file was created
  - when the file was last changed
  - when the file was last read
- Size
- Owners
  - user (UID)
  - group (GID)
- Permissions

Interacting with the shell

Customization

- Each shell supports some customization.
  - user prompt settings
  - environment variable settings
  - aliases
- The customization takes place in startup files which are read by the shell when it starts up
  - Global files are read first - these are provided by the system administrators (e.g.. /etc/profile)
  - Local files are then read in the user's HOME directory to allow for additional customization

File Time Attributes

- Time Attributes

`ls -l <filename>`	when the file was last changed

```
-> stat /
  File: '/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: fd00h/64768d    Inode: 128         Links: 24
Access: (0555/dr-xr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2022-02-15 15:11:56.462507715 +0800
Modify: 2022-01-28 01:23:22.509320161 +0800
Change: 2022-01-28 01:23:22.509320161 +0800
 Birth: -
```

- Display date-related attributes with the **`stat`** command

File Permissions

- Each file has a set of permissions that control who can access the file
- There are three different types of permissions:
  - read abbreviated r
  - write abbreviated w
  - execute abbreviated x
- In Unix, there are permission levels associated with three types of people that might access a file:
  - owner (you)
  - group (a group of other users that you set up)
  - world (anyone else browsing around on the file system)

File Permissions Display Format

The first column specifies the type of file:

Symbol	File type	Example
-	plain file	
d	directory	
c	character device	
b	block device	
l	symbolic link	

File Permissions Display Format cont.

- Meaning for Files:
  - r - allowed to read
  - w - allowed to write
  - x - allowed to execute
- Meaning for Directories:
  - r - allowed to see the names of the files
  - w - allowed to add and remove files
  - x - allowed to enter the directory

Changing File Permissions

- The **chmod** command changes the permissions associated with a file or directory
- Basic syntax is:

```
chmod mode file
```

- The **mode** can be specified in two ways:
  - symbolic representation
  - octal number
- Both methods achieve the same result (user's choice)
- Multiple symbolic operations can be given, separated by commas

chmod: Symbolic Representation

- *Symbolic* Mode representation has the following form:

[ugoa] [+-=] [rwxX…]

u=user	**+** add permission	**r**=read
g=group	**-** remove permission	**w**=write
o=other	**=** set permission	**x**=execute
a= all		**X**= see below

- The **X** permission option is very handy - it sets to execute only if the file is a directory or already has execute permission

chmod: Symbolic Mode Examples

```
lslogin$ ls -al foo
-rw-------  1 karl support ...
```

```
lslogin$ chmod g=rw foo
lslogin$ ls -al foo
-rw-rw----  1 karl support ...
```

```
lslogin$ chmod u-w,g+x,o=x foo
lslogin$ ls -al foo
-r--rwx--x  1 karl support ...
```

chmod: Octal Representation

- Octal Mode uses a single argument string which describes the permissions for a file (3 digits)
- Each digit of this number is a code for each of the three permission levels (user, group, world)
- Permissions are set according to the following numbers:
  - Read = 4
  - Write = 2
  - Execute = 1

	Permission Level
0	no permissions
1	execute only
2	write only
3	write and execute (1+2)
4	read only
5	read and execute (4+1)
6	read and write (4+2)
7	read, write and execute (4+2+1)

- Sum the individual permissions to get the desired combination

umask: set default permission

- umask is used to set default permissions for files and directories

```
$ umask          # display current value (as octal)
0022
$ umask -S       # display current value symbolically
u=rwx,g=rx,o=rx
```

umask: set default permission

- umask is used to set default permissions for files and directories

```
$ umask 007     # set the mask to 007
$ umask         # display the mask (in octal)
0007            #   0 - special permissions (setuid | setgid | sticky )
                #   0 - (u)ser/owner part of mask
                #   0 - (g)roup part of mask
                #   7 - (o)thers/not-in-group part of mask
$ umask -S      # display the mask symbolically
u=rwx,g=rwx,o=
```

Outline

1. File attributes and permissions
2. Basic Commands
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

Regular Expressions

grep : get regular expression.

grep understands basic regular expressions.

egrep understands extended regular expressions.

```
egrep –ni –color “some string pattern” file-to-search
```

Regular Expressions

- In addition to grep, a number of Unix commands support the use of regular expressions to describe patterns:
  - sed
  - awk
  - perl

  Examples: runoo+b matches runoob, runooob, runooooooob, etc.
  runoo*b matches runob, runoob, runooooob, etc.
  colou?r matches color or colour.
- General search pattern characters:
  - Any character (except a metacharacter) matches itself
  - "." matches any character except a newline
  - "*" matches zero or more occurrences of the single preceding character
  - "+" matches one or more of the proceeding character
  - "?" matches zero or one of the proceeding character
- Additional special characters:
  - "()" parentheses are used to quantify a sequence of characters
  - "|" works as an OR operator
  - "{}" braces are used to indicate ranges in the number of occurrences

Regular Expressions

- In addition to grep, a number of Unix commands support the use of regular expressions to describe patterns:
  - sed
  - awk
  - perl

  Examples: “g(la|oo)d” will match glad or good
  “go\{1,2\}d” will match god or good
- General search pattern characters:
  - Any character (except a metacharacter) matches itself
  - “.” matches any character except a newline
  - “*” matches zero or more occurrences of the single preceding character
  - “+” matches one or more of the proceeding character
  - “?” matches zero or one of the proceeding character
- Additional special characters:
  - “()” parentheses are used to quantify a sequence of characters
  - “|” works as an OR operator
  - “{}” braces are used to indicate ranges in the number of occurrences

Regular Expressions

```
-> cat regular_express.txt
"Open Source" is a good mechanism to develop programs
apple is my favorite food
Football game does not use feet only
Oh! The soup taste good ^M
The symbol '*' is represented as start
<Happy>
Goooogle is good!
Go Go Go
However, this dress is about $ 333 dollars ^L.
\\ This is commen.t
# End of file
```

Regular Expressions

```
-> cat regular_express.txt
"Open Source" is a good mechanism to develop programs
apple is my favorite food
Football game does not use feet only
Oh! The soup taste good ^M
The symbol '*' is represented as start
<Happy>
Goooogle is good!
Go Go Go
However, this dress is about $ 333 dollars ^L.
\\ This is commen.t
# End of file
```

```
-> grep -n 'the' regular_express.txt
```

Regular Expressions

```
-> cat regular_express.txt
"Open Source" is a good mechanism to develop programs
apple is my favorite food
Football game does not use feet only
Oh! The soup taste good ^M
The symbol '*' is represented as start
<Happy>
Goooogle is good!
Go Go Go
However, this dress is about $ 333 dollars ^L.
\\ This is commen.t
# End of file
```

```
-> grep -ni 'the' regular_express.txt
4:Oh! The soup taste good ^M
5:The symbol '*' is represented as start
```

Regular Expressions

```
-> cat regular_express.txt
"Open Source" is a good mechanism to develop programs
apple is my favorite food
Football game does not use feet only
Oh! The soup taste good ^M
The symbol '*' is represented as start
<Happy>
Goooogle is good!
Go Go Go
However, this dress is about $ 333 dollars ^L.
\\ This is commen.t
# End of file
```

```
-> grep -n --color 'g.o' regular_express.txt
1:"Open Source" is a good mechanism to develop programs
4:Oh! The soup taste good ^M
7:Goooogle is good!
```

Regular Expressions

```
-> cat regular_express.txt
"Open Source" is a good mechanism to develop programs
apple is my favorite food
Football game does not use feet only
Oh! The soup taste good ^M
The symbol '*' is represented as start
<Happy>
Goooogle is good!
Go Go Go
However, this dress is about $ 333 dollars ^L.
\\ This is commen.t
# End of file
```

```
-> grep -n --color 'go*' regular_express.txt
1:"Open Source" is a good mechanism to develop programs
3:Football game does not use feet only
4:Oh! The soup taste good ^M
7:Goooogle is good!
```

Regular Expressions

Regular Expressions

- If you really want to match a period '.', you need to escape it with a backslash "\."

Regexp	Matches	Does not match
a.b	axb	abc
a\.b	a.b	axb

Regular Expressions

```
-> cat regular_express.txt
"Open Source" is a good mechanism to develop programs
apple is my favorite food
Football game does not use feet only
Oh! The soup taste good ^M
The symbol '*' is represented as start
<Happy>
Goooogle is good!
Go Go Go
However, this dress is about $ 333 dollars ^L.
\\ This is commen.t
# End of file
```

```
-> grep -n --color 'n\.t' regular_express.txt
10:\\ This is commen.t
```

Regular Expressions

Regular Expressions

- A character class, also called a character set can be used to match only one out of several characters
- To use, simply place the characters you want to match between square brackets []
- You can use a hyphen inside a character class to specify a range of characters
- Placing a caret (^) after the opening square bracket will negate the character class. The result is that the character class will match any character that is not in the character class
- Examples:
  - [abc] matches a single a b or c
  - [0-9] matches a single digit between 0 and 9
  - [^A-Za-z] matches a single character as long as it is not a letter

Regular Expressions

```
-> cat regular_express.txt
"Open Source" is a good mechanism to develop programs
apple is my favorite food
Football game does not use feet only
Oh! The soup taste good ^M
The symbol '*' is represented as start
<Happy>
Goooogle is good!
Go Go Go
However, this dress is about $ 333 dollars ^L.
\\ This is commen.t
# End of file
```

```
-> grep -n --color '[a-z]es' regular_express.txt
3:Football game does not use feet only
5:The symbol '*' is represented as start
9:However, this dress is about $ 333 dollars ^L.
```

Regular Expressions

```
-> cat regular_express.txt
"Open Source" is a good mechanism to develop programs
apple is my favorite food
Football game does not use feet only
Oh! The soup taste good ^M
The symbol '*' is represented as start
<Happy>
Goooogle is good!
Go Go Go
However, this dress is about $ 333 dollars ^L.
\\ This is commen.t
# End of file
```

```
-> grep -n --color '[^l-q]es' regular_express.txt
5:The symbol '*' is represented as start
9:However, this dress is about $ 333 dollars ^L.
```

grep: cont.

```
lslogin$ cat sequence.fas
>c01_009 499 amino acids  MW=55632 D  pI=5.38 numambig=0
MPGGFILAIDEGTTSARAIIYNQDLEVLGIGQYDFPQHYPSPGYVEHNPDEIWNAQMLAI
KEAMKKAKIESRQVAGIGVTNQRETTILWDAISGKPIYNAIVWQDRRTSNITDWLKENYF
DYSNASRTMLFNINKLEWDREILELLKIPESILPEVRPSSDIYGYTEVLGSSIPISGDAG
DQQAALFGQVAYDMGEVKSTYGTGSFILMNIGSNPIFSENLLTTIAWGLESKRVTYALEG
SIFITGAAVQWFRDGLGREPKICKSBUTTASVPDTGGVYFVPAFVGLGAPYWDPYARGLI
IGITRGTTKAHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQAD
ILGIRVVRPKVMETASMGVAMLAGLAINYWNSLNELKQKWTVDKEFIPSINKEERERRYN
AWKEAVKRSLGWEKSLGSK*

lslogin$ grep '[ST].[RK]' sequence.fas
MPGGFILAIDEGTTSARAIIYNQDLEVLGIGQYDFPQHYPSPGYVEHNPDEIWNAQMLAI
KEAMKKAKIESRQVAGIGVTNQRETTILWDAISGKPIYNAIVWQDRRTSNITDWLKENYF
DQQAALFGQVAYDMGEVKSTYGTGSFILMNIGSNPIFSENLLTTIAWGLESKRVTYALEG
IGITRGTTKAHIARAILESIAYQNRDVIEIMEKESGTKINILKVDGGGAKDNLLMQFQAD
```

a regular expression search

Regular Expressions

- Since certain character classes are used often, a series of shorthand character classes are available for convenience:

  \d a digit. eg [0-9]
  \D a non-digit, e.g.. [^0-9]
  \w a word character (matches letters and digits)
  \W a non-word character
  \s a whitespace character
  \S a non-whitespace character

```
-> grep -n --color '\d' regular_express.txt
9:However, this dress is about $ 333 dollars ^L.
```

Regular Expressions

- Since certain character classes are used often, a series of shorthand character classes are

```
-> grep -n --color '\W' regular_express.txt
1:"Open Source" is a good mechanism to develop programs
2:apple is my favorite food
3:Football game does not use feet only
4:Oh! The soup taste good ^M
5:The symbol '*' is represented as start
6:<Happy>
7:Goooogle is good!
8:Go Go Go
9:However, this dress is about $ 333 dollars ^L.
10:\\ This is commen.t
11:# End of file
```

Regular Expressions

Regular Expressions

- More shorthand classes are available for matching boundaries:

```
-> grep -n --color "^H" regular_express.txt
9:However, this dress is about $ 333 dollars ^L.
```

^	the beginning of a line
$	the end of a line
\b	a word boundary
\B	a non-word boundary

Regular Expressions

Regular Expressions

- More shorthand classes are available for matching boundaries:

```
-> grep -n --color "y$" regular_express.txt
3:Football game does not use feet only
```

^	the beginning of a line
$	the end of a line
\b	a word boundary
\B	a non-word boundary

Regular Expressions

Regular Expressions Examples

- “notice” a string that has the text "notice" in it
- “F.” matches an “F” followed by any character
- “a.b” matches “a” followed by any 1 char followed by “b”
- “^The” matches any string that starts with "The"
- “oh boy$” matches a string that ends in the substring "oh boy";
- “^abc$” matches a string that starts and ends with "abc" -- that could only be "abc" itself!
- “ab*” matches an “a” followed by zero or more “b”'s ("a", "ab", "abbb", etc.)
- “ab+” similar to previous, but there's at least one “b” ("ab", "abbb", etc.)
- “(b|cd)ef” matches a string that has either "bef" or "cdef"
- “a(bc)*” matches an *“a”* followed by zero or more copies of the sequence "bc"
- “ab{3,5}” matches an *“a”* followed by three to five *“b”*'s ("abbb", "abbbb", or "abbbbb")
- “[Dd][Aa][Vv][Ee]” matches "Dave" or "dave" or "dAVE“, does not match "ave" or "da"

Regular Expressions

Search for a cell phone number:

```
-> egrep -n --color "\b1[3|4|5|7|8][0-9]{9}\b"
```

Search for an email address:

```
-> egrep -n --color "\b\w*@\w*" demo
```

Search on the stack overflow website for more sophisticated ways.

Could be a job interview question for programming jobs.

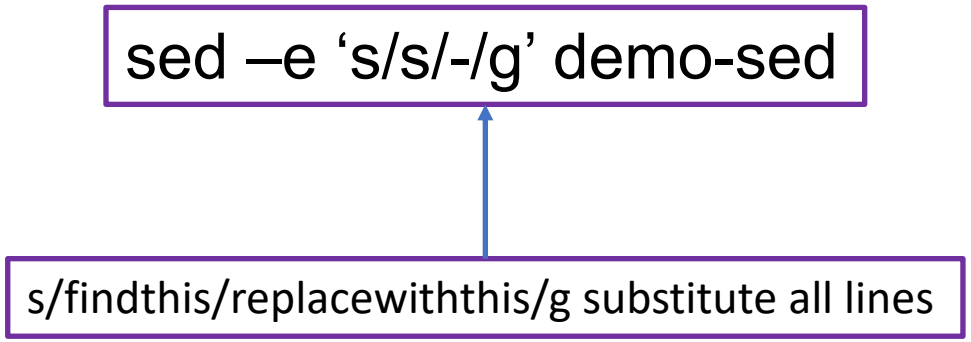
Regular Expressions

sed : stream editor

Modify a stream of data in some fashion

The following command will substitute all “s” in the demo-sed file by “-”

Read Chapter 10 of the book “Unix in a nutshell” for more details.



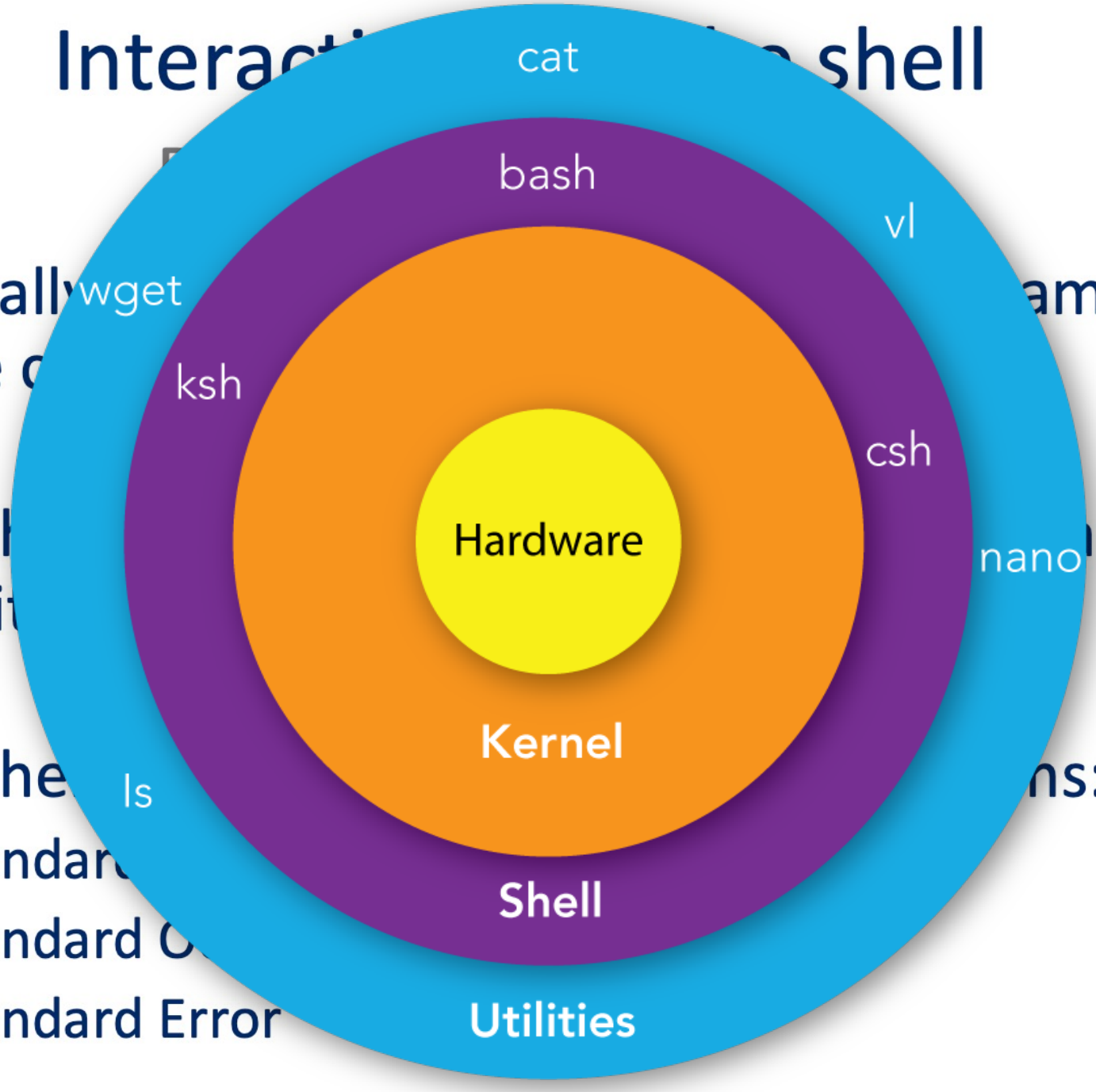
Introduction to Unix

Outline

1. File attributes and permissions
2. Basic Commands
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

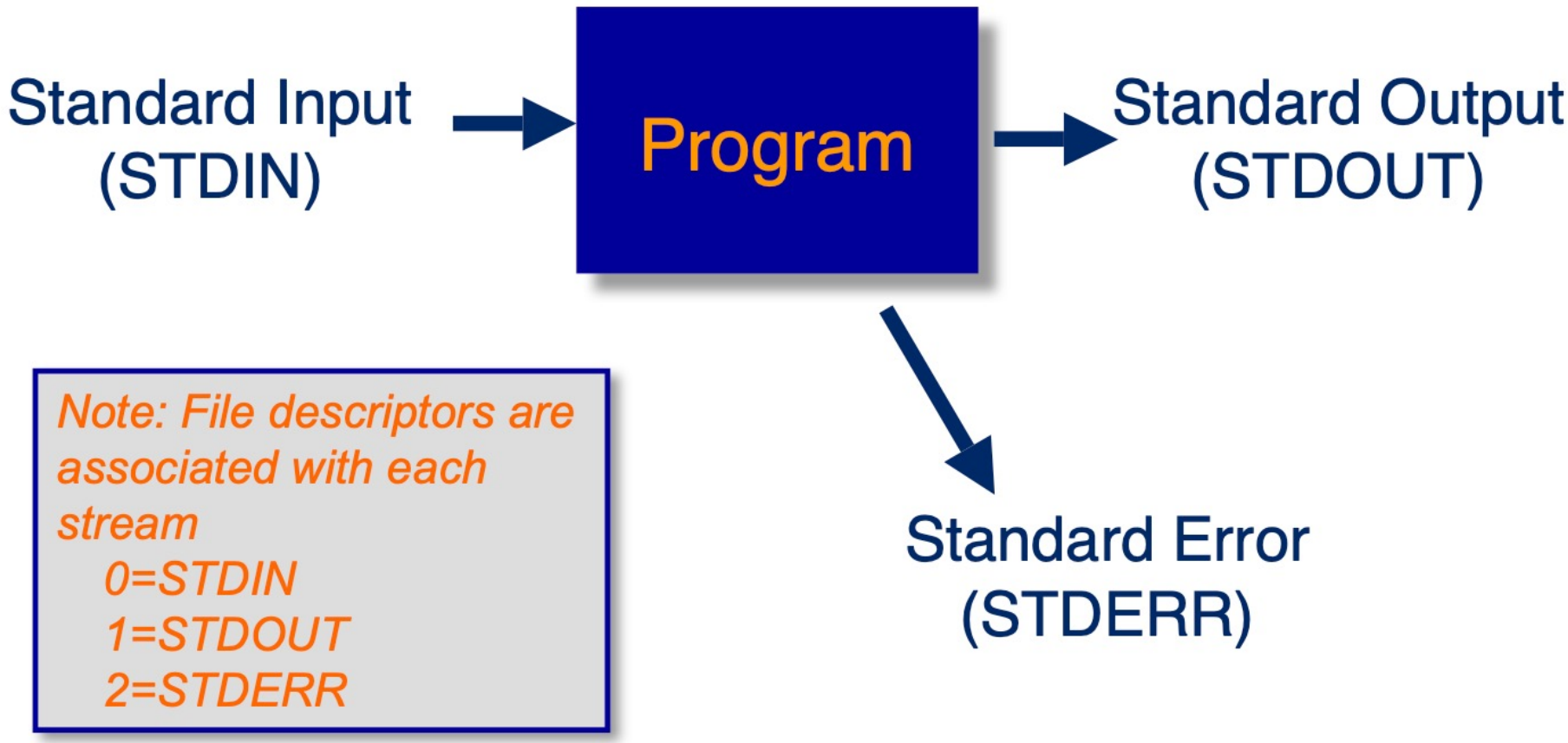
Interacti shell

- Typicall am and some c
- The sh nd runs it
- The she ns:
  - Standar
  - Standard O
  - Standard Error



Interacting with the shell

Programs and Standard I/O



Note: File descriptors are associated with each stream

0=STDIN
1=STDOUT
2=STDERR

Interacting with the shell

Defaults for I/O

- When a shell runs a program for you:
  - standard input is your keyboard
  - standard output is your screen or window
  - standard error is your screen or window
- If standard input is your keyboard, you can type stuff in that goes to a program
- To end the input you press Ctrl-D (^D) on a line by itself, this ends the input stream
- The shell is a program that reads from standard input
- Any idea what happens when you give the shell ^D?

Interacting with the shell

UNIX: Shell Flavors

- There are two main ‘flavors’ of shells:
  - Bourne created what is now known as the standard shell: “sh”, or “bourne shell”. Its syntax roughly resembles Pascal. Its derivatives include “ksh” (“korn shell”) and now, the most widely used, “bash” (“bourne again shell”)
  - One of the creators of the C language implemented a shell to have a “C-programming” like syntax. This is called “csh” or “C-shell”. Today’s most widely used form is the very popular “tcsh”
- Shells can run interactively or as a shell script

Shell variables

- Shell variables can only contain letters (a-z A-Z), numbers (0-9), or the underscore (_).

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

Correct.

```
2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
```

Wrong.

- Use = to define a variable (without spacing)
  e.g. machine_name=Kohn

- Access the variable value by $
  echo $machine_name

- Unsetting a variable directs the SHELL to remove the variable from the list of variables that it tracks.
  unset machine_name

Shell variables

Three types of variables:

- Local variables : a variable that is present within the current instance of shell command prompt. It is not available to programs started by the shell.

- Environmental variables : a variable available to any child process of the shell. Some functions need environmental variables to function correctly.

Define environmental variable : export var_name=var_value

It can be tedius to type 172.18.6.175 every time to enter Tai-Yi.

```
export TAIYI=172.18.6.175

ssh mae-liuj@$TAIYI
```

Shell variables

Three types of variables:

- Local variables : a variable that is present within the current instance of shell command prompt. It is not available to programs started by the shell.

- Environmental variables : a variable available to any child process of the shell. Some functions need environmental variables to function correctly.

Define environmental variable : export var_name=var_value

It can be tedius to remember the data directory on Tai-Yi.

```
export DATA=/data/mae-liuj

cd $DATA
```

```
-> env
TERM_PROGRAM=Apple_Terminal
SHELL=/bin/bash
TERM=xterm-256color
TMPDIR=/var/folders/w8/xz7sb8b54z1_zyrt75n8pyd80000gn/T/
TERM_PROGRAM_VERSION=433
TERM_SESSION_ID=217223CD-AE30-4FB1-9988-14DCB6E048D7
USER=juliu
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.F1SfIGgjKC/Listeners
PATH=/Users/juliu/lib/mpich-3.3.2/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
:/Library/TeX/texbin:/Library/Apple/usr/bin
LaunchInstanceID=E8E6312D-FEA7-44D0-AB21-8EEDF6335D39
PWD=/Users/juliu
EDITOR=vim
XPC_FLAGS=0x0
XPC_SERVICE_NAME=0
SHLVL=1
HOME=/Users/juliu
LOGNAME=juliu
VISUAL=vim
LC_CTYPE=en_US.UTF-8
SECURITYSESSIONID=186a9
FFLAGS=-w -fallow-argument-mismatch -O2
MACHINE_NAME=kolmogorov
_=/usr/bin/env
```

list all environmental variables

shell type

we talked about this before

home directory (~)

Interacting with the shell

Shell Startup Files

- sh,ksh:
- ~/.profile
- bash:
  ~/.bash_profile
  ~/.bash_login
- ~/.profile
- ~/.bashrc
- ~/.bash_logout
- csh:
- ~/.cshrc
- ~/.login
- ~/.logout
- tcsh:
- ~/.tshrc
- ~/.cshrc
- ~/.login
- ~/.logout

- These are hidden files in home directory
- .bashrc is executed for interactive ***non-login shells***
- .bash_profile or .bash_login or .profile is exectued for ***login shells*** (i.e. when you use password to enter shell)
- In login shell, only one of the above will be loaded.
- In Mac OS terminal by default runs a login shell every time.

change your PATH variable

1. Go to your home directory
2. Display all files (including the hidder file)
3. Make sure the file .bashrc exists
4. Open .bashrc by a text editor
5. Add `export PATH=/Users/juliu/lib/mpich-3.3.2/bin:$PATH`
6. Save and exit the editor
7. Run source ~/.bashrc.

source is the command that loads the .bashrc configuration.

Interacting with the shell

Wildcards for Filename Abbreviation

- When you type in a command line the shell treats some characters as special (*metacharacters)*
- These special characters make it easy to specify filenames
- The shell processes what you give it, using the special characters to replace your command line with one that includes a bunch of file names

see Chapter 7 of Unix in a nutshell

Interacting with the shell

The special character *

- “*” matches anything.
- If you give the shell “*” by itself (as a command line argument), the shell will remove the * and replace it with all the filenames in the current directory.
- “a*b” matches all files in the current directory that start with a and end with b.
- This looks like regular expressions but isn’t quite the same.

Interacting with the shell

Understanding *

- The echo command prints out whatever you tell it:
  - > echo hi
  - hi
  - > ls
  - dir1 foo foo2
- What will the following command do?

> echo *

```
dir1 foo foo2
```

SHELL alias

- A shell alias is a shortcut to reference a command.
- Good for avoid typing long commands.
- Reduce keystronkes and guard correctness.

alias name=value

```
# User specific aliases and functions
alias ll='ls -lhrGS'
alias rm='rm -i'
alias mv='mv -i'
alias cdw='cd /work/mae-liuj'
alias cdd='cd /data/mae-liuj'
alias cds='cd /scratch'
alias rmsol='rm -rf SOL_* dot_SOL_*'
alias h5dump='/work/mae-liuj/lib/hdf5-1.8.16/bin/h5dump'
```

Interacting with the shell

Shell Stream Redirection

- A very powerful function in Unix is redirection for input and output:
  - The shell can attach things other than your keyboard to standard input (stdin)
    - A file (the contents of the file are fed to a program as if you typed it) - common in scientific programming
    - A pipe (the output of another program is fed as input as if you typed it)
  - The shell can attach things other than your screen to standard output (stderr)
    - A file (the output of a program is stored in file)
    - A pipe (the output of a program is fed as input to another program

Interacting with the shell

Stream Redirection

- To tell the shell to store the output of your program in a file, follow the command line for the program with the “>” character followed by the filename:
-

  ls > lsout

- The command above will create a file named lsout and place the output of the ls command in the file

Interacting with the shell

Stream Redirection

- To have the shell get standard input from a file, use the “<“ character:

  sort < nums

- The command above would sort the lines in the file nums and send the result to stdout

- The beauty of redirection is that you can do both forms together:

  sort < nums > sortednums

Interacting with the shell

Modes of Output Redirection

- There are two modes of output redirections:
  - “>” the create mode
  - “>>” the append mode

- For example:
  - the command ls > foo will create a new file named foo (deleting any existing file named foo).
  - if you use “>>” instead, the output will be appended to foo:

```
ls /etc >> foo
ls /usr >> foo
```

Interacting with the shell

Stream Redirection

- Many commands send error messages to standard error (stderr) which is different from stdout.
- However, the “>” output redirection only applies to stdout (not stderr)
- To redirect stderr to a file you need to know what shell you are using:
  - BASH
    - “2>” redirects stderr (eg. ls foo blah gork 2> erroroutput)
    - “&>” redirects stdout and stderr (eg. ls foo &> /dev/null)
  - TCSH
    - “>&” merges stdout and stderr and sends to a file:
      ls foo blah >& saveboth
    - “>>&” merges stdout and stderr and appends to a file:
      ls foo blah >>& saveboth

Note: File descriptors are associated with each stream

0=STDIN
1=STDOUT
2=STDERR

范例1-"太乙"-vasp

注意:建议采用2018.4版本

```
#!/bin/sh
#BSUB -J N_F                          ##job name
#BSUB -q short                        ##queue name
#BSUB -n 80                           ##number of total cores
#BSUB -R "span[ptile=40]"             ##40 cores per node
#BSUB -W 12:00                        ##walltime in hh:mm
#BSUB -R "select[hname!='r13n18']" ##exclusive r13n18
#BSUB -e err.log                      ##error log
#BSUB -o H.log                        ##output log
module load intel/2018.4 mpi/intel/2018.4 vasp/5.4.4
mpirun vasp_std &>log
```

Introduction to Unix

Outline

1. File attributes and permissions
2. Basic Commands
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

Unix pipes

Unix Pipes

- A pipe is a holder for a stream of data
- A Unix pipeline is a set of processes chained by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) of the next one

Unix pipes

Building Commands

- More complicated commands can be built up by using one or more pipes
- Use the "|" character to pipe two commands together
- The shell takes care of all the hard work for you
- Example:

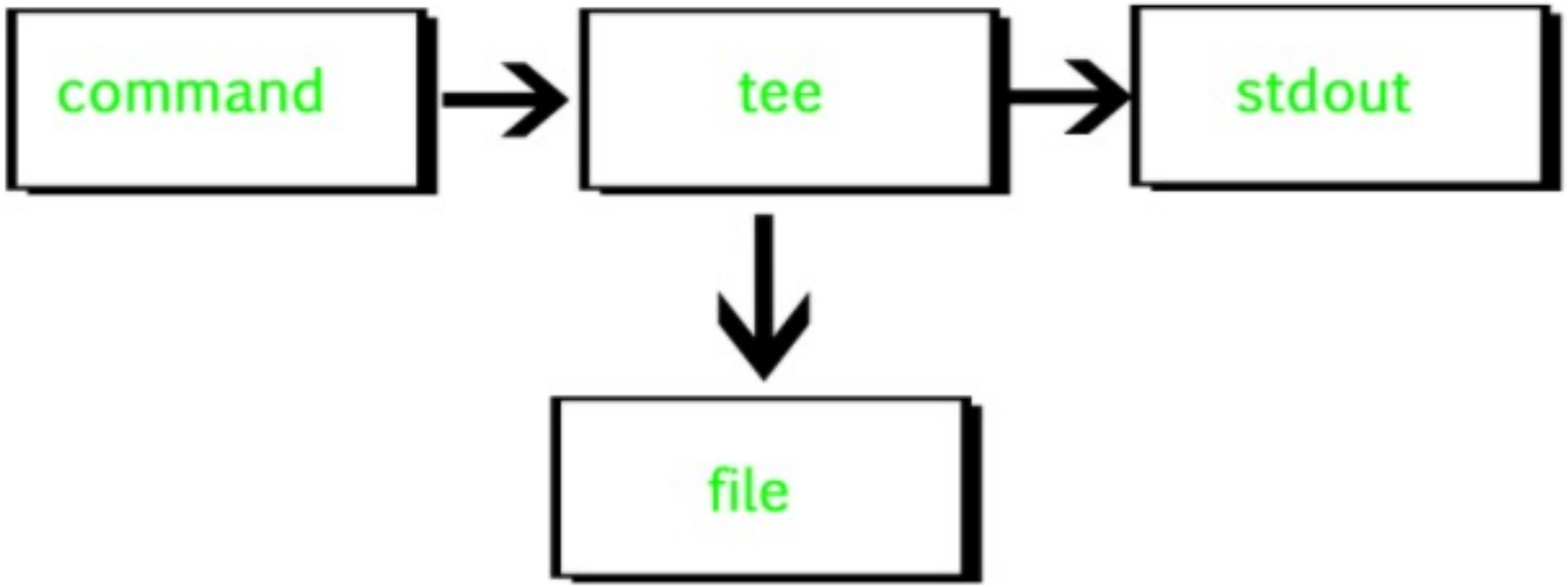
```
> cat apple.txt
core
worm seed
jewel

> cat apple.txt | wc
3 4 21
```

Note: the wc command prints the number of newlines, words, and bytes in a file

Unix pipes

Building Commands

tee : reads the standard input and writes it to both the standard output and one or more files.




```
-> ls -l | tee output.txt
```

```
-> ls -l | wc -l | tee output.txt
```

Unix pipes

Building Commands

xargs : converts input from standard input into arguments to a command

```
find /path -type f -print | xargs rm
```

Introduction to Unix

Outline

1. File attributes and permissions
2. Basic Commands
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

Job Control

Job Control

- The shell allows you to manage jobs
  - place jobs in the background
  - move a job to the foreground
  - suspend a job
  - kill a job
- If you follow a command line with “&”, the shell will run the job in the background
  - this is you useful if you don’t want to wait for the job to complete
  - you can type in a new command right away
  - you can have a bunch of jobs running at once
  -

    > cat foo | sort | uniq > saved_sort &

Job Control

Background jobs

- Handy for programs you need throughout a session: emacs &
- For commands that take a lot of time: make all &> make.out &
- If the job will run longer than your session: nohup make all &> make.out &

Job Control

Listing Your Jobs

- The command jobs will list all background jobs:

  ```
  > jobs
  [1] Running  cat foo | sort | uniq > saved_ls &
  ```

- The shell assigns a number to each job (in this case, the job number is 1)

Job Control

Managing Jobs

- You can kill the foreground job by pressing ^C (Ctrl-C).

- You can also kill a job in the background using the kill command (and the appropriate job index)

  > kill %1

Note: it's important to include the "%" sign to reference a job number.

Introduction to Unix

Outline

1. File attributes and permissions
2. Basic Commands
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

UNIX Environment variables

Unix Environment Variables

- Unix shells maintain a list of environment variables which have a unique name and a value associated with them
  - some of these parameters determine the behavior of the shell
  - also determine which programs get run when commands are entered (and which libraries they link against)
  - provide information about the execution environment to programs
- We can access these variables:
  - set new values to customize the shell
  - find out the value of some to help accomplish a task

UNIX Environment variables

Environment Variables

- To view environment variables, use the env command
- If you know what you are looking for, you can use your new friend grep:
-

```
> env | grep PWD
PWD=/home/karl
```

- Use the echo command to print variables; the “$” prefix is required to access the value of the variable:

```
> echo $PWD
/tmp
```

- Can also use environment variables in arbitrary commands:

```
Koomie@canyon--> ls $PWD
foo1  foo2
```

UNIX Environment variables

Special Environment Variable: PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it is not a build-in command, the shell tries to find a program whose name matches the desired command
- How does the shell know where to look on the filesystem?
- The PATH variable tells the shell where to search for programs (non built-in commands)

UNIX Environment variables

Special Environment Variable: PATH

- Example PATH Definition:

  -> echo $PATH
  /home/karl/bin/krb5:/opt/intel/compiler70/ia32/bin:/home/karl/bin:/usr/local/apps/mpich/icc/bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin

- The PATH is a list of directories delimited by colons (":“)
  - It defines a list and search order
  - Directories specified earlier in the PATH take precedent; once the matching command is found, the search terminates

- You can add more search directories to your PATH by changing the shell startup files
  - BASH: export PATH="$PATH":/home/karl/bin
  - TCSH: set path = (/home/karl/bin $path)

UNIX Environment variables

Other Important Variables

- PWD current working directory
- MANPATH determines where to find man pages
- HOME home directory of user
- MAIL where your email is stored
- TERM what kind of terminal you have
- PRINTER specifies the default printer name
- EDITOR used by many applications to identify your choice of editors (eg. vi or emacs)
- LD_LIBRARY_PATH specifies a search path for dynamic runtime libraries

UNIX Environment variables

Setting Environment Variables

- The syntax for setting Unix environment variables depends on your shell:
  - BASH: use the export command
    > export PRINTER=scully
    > echo $PRINTER
    scully
  - TCSH: use the setenv command
    > setenv PRINTER mulder
    > echo $PRINTER
    mulder

- Note: environment variables that you set interactively are only available in your current shell
  - If you spawn a new shell (or login again), these settings will be lost
  - To make permanent changes, you should alter the login scripts that affect your particular shell (eg. .login, .profile, .cshrc, etc...)

Introduction to Unix

Outline

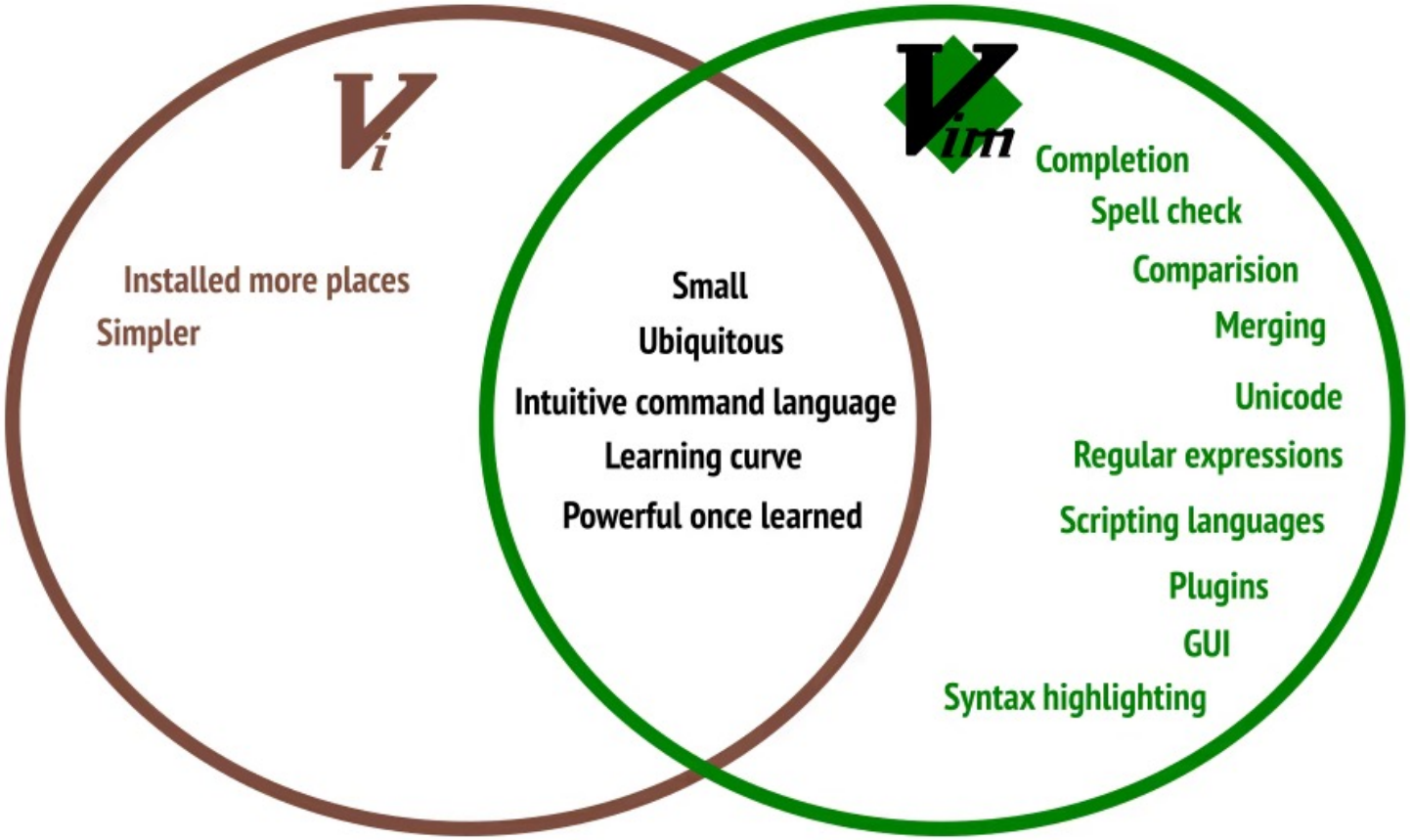
1. File attributes and permissions
2. Basic Commands
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

Text Editors

- For programming, we need to make use of available Unix text editors
- The two most popular and available editors are vi and emacs
- You should familiarize yourself with at least ~~one of the two~~ vi (and this let's you enter into the editor wars which is a never-ending debate in the programming community)

Text Editors

Vi Overview



Install vim:

```
sudo apt-get install vim
```

Text Editors

Vi Overview

- Fundamental thing to remember about vi is that it has two different modes of operation:
  - Insert Mode
  - Command mode
- The insert mode puts anything typed on the keyboard into the current file
- The command mode allows the entry of commands to manipulate text. These commands are usually one or two characters long, and can be entered with few keystrokes
- Note that vi starts out in the command mode by default

Text Editors

Vi Overview

- Quick Start Commands
  - > vi
  - Press i to enable insert mode
  - Type text (use arrow keys to move around)
  - Press Esc to enable command mode
  - Press :w <filename> to save the file
  - Press :q to exit vi

Text Editors

Useful vi commands

- :q! – exit without saving the document. Very handy for beginners
- :wq – save and exit
- / <string> – search within the document for text. n goes to next result
- dd – delete the current line
- yy – copy the current line
- p – paste the last cut/deleted line
- :1 – goto first line in the file
- :$ - goto last line in the file
- $ – end of current line, ^ – beginning of line
- % – show matching brace, bracket, parentheses

Text Editors

Additional vi References

- http://www.eng.hawaii.edu/Tutor/vi.html
- http://staff.washington.edu/rells/R110/
- Vi Commands Reference card: http://tnerual.eriogerg.free.fr/vimqrc.pdf

Introduction to Unix

Outline

1. File attributes and permissions
2. Basic Commands
3. Regular expressions
4. Interacting with the shell
5. Unix pipes
6. Job control
7. UNIX Environment variables
8. Text Editors
9. Shell scripting

Shell Scripting

Unix Scripting

- Scripting is “easy” - you just place all the Unix commands in a file as opposed to typing them interactively

- Handy for automating certain tasks:
  - staging your scientific applications
  - performing limited post-processing operations
  - any repetitive operations on files, etc...

- Shells provide basic control syntax for looping, if constructs, etc…

Shell Scripting

Unix Scripting

- Shell scripts must begin with a specific line to indicate which shell should be used to execute the remaining commands in the file:
  - BASH:
    #!/bin/bash
  - TCSH
    #!/bin/tcsh
- Comment lines can be included if they start with #
- In order to run a shell-script, it must have execute permission. Consider the following script:

  > cat hello.sh
  #!/bin/bash
  echo “hello world”

  > ./hello.sh
  ./hello.sh: Permission denied.

- > chmod 700 hello.sh
  > ./hello.sh
  hello world

Shell Scripting

Unix Scripting: Arithmetic Operations

- Simple arithmetic syntax depends on the shell:
  - TCSH
  - set i1=10
  - set j1=3
  - @ k1 = $i1 + $j1 # Note space between @ and k1
  - echo "The sum of $i1 and $j1 is $k1"
  - BASH
  - i1=2
  - j1=6
  - k1=$(($i1*$j1))
  - echo "The multiplication of $i1 and $j1 is $k1"

- Note, you can also use the expr command (for both shells). For example:
  - TCSH: set z=`expr $i1 + $j1`
  - BASH: z=`expr $i1 + $j1`

consult man page on `expr` for more details

Shell Scripting

Unix Scripting: Conditionals

- Syntax for conditional expressions depends on your choice of shell:
- BASH (general format):

```
if [ condition_A ]; then
        code to run if condition_A true
elif [ condition_B ]; then
        code to run if condition_A false and
condition_B true
else
        code to run if both conditions false
fi
```

- TCSH (general format):
  -

```
if (condition) then
  commands
else if (other condition) then
  commands
else
  commands
endif
```

Shell Scripting

Unix Scripting: String Comparisons

- string1 = string2 Test identity
- string1 !=string2 Test inequality
- -n string the length of string is nonzero
- -z string the length of string is zero

BASH Example:

```
today="monday"
if [ "$today" = "monday" ] ; then
    echo "today is monday"
fi
```

TCSH Example:

```
set today="friday"
if ( "$today" != "monday" )  then
    echo "today is not monday"
endif
```

Shell Scripting

BASH Integer Comparisons

- int1 –eq int2 Test identity
- int1 –ne int2 Test inequality
- int1 –lt int2 Less than
- int1 –gt int2 Greater than

BASH Example:

```
x=13
y=25
if [ $x -lt $y ]; then
  echo "$x is less than $y"
fi
```

Shell Scripting

Unix Scripting: Common File Tests

- -d file Test if file is a directory
- -f file Test if file is not a directory
- -s file Test if the file has non zero length
- -r file Test if the file is readable
- -w file Test if the file is writable

BASH Example:

```
if [ -f foo ]; then
  echo "foo is a file"
fi
```

Shell Scripting

Unix Scripting: For loops

- These are useful when you want to run the same command in sequence with different options
- sh example:
  - for VAR in test1 test5 test7b finaltest; do
  - runmycode $VAR > $VAR.out
  - done
- csh example:
  - foreach VAR (test1 test5 test7b finaltest)
  - runmycode $VAR > $VAR.out
  - end
- sh one-liner (note seq is not standard):
- for i in `seq 1 5`; do echo $i; done
- 1
- 2
- 3
- 4
- 5

Shell Scripting

Quoting in Unix

- We've seen that some metacharacters are treated special on the command line: * ?
- What if we don't want the shell to treat these as special - we really mean *, not all the files in the current directory
- To turn off special meaning - surround a string with double quotes:

```
> echo here is a star "*"
here is a star *
```

Shell Scripting

Use of Quotes

- You have to be careful with the use of different styles of quotes in your commands or scripts

- They have different functions:
  - Double quotes inhibit wildcard replacement only
  - Single quotes inhibit wildcard replacement, variable substitution and command substitution
  - Back quotes cause command substitution

Shell Scripting

Single Quotes

- Single quotes are similar to double quotes, but they also inhibit variable substitution and command substitution
- Means that special characters do not have to be escaped:

- > echo 'This is a quote \" '
- This is a quote \”

Shell Scripting

Back Quotes

- If you surround a string with back quotes, the string is replaced with the result of running the command in back quotes:

- > echo `ls`
- foo fee file?

- > echo "It is now `date` and OU is still questionable”
  It is now Tue Sep 19 11:24:25 CDT 2006 and OU is still questionable

Shell Scripting

More Quote Examples

- Some Quoting Examples:
  - $ echo Today is date
  - Today is date
  - $ echo Today is `date`
  - Today is Thu Sep 19 12:28:55 EST 2002
  - $ echo ”Today is `date`”
  - Today is Thu Sep 19 12:28:55 EST 2002
  - $ echo ’Today is `date`’
  - Today is `date`

“ “ = double quotes
‘ ‘ = single quotes
` ` = back quotes

Shell Scripting

Command-Line Parsing

- To build generic shell scripts, consider using command-line arguments to provide the inputs you need internally (syntax again depends on the choice of shell)

- Syntax:
  - $# refers to the number of command-line arguments
  - $0 refers to the name of the calling command
  - $1, $2, ..., $N refers to the Nth argument
  - $* refers to all command-line parameters

  - echo "Calling command is: $0"
  - echo "Total # of arguments is: $#"
  - echo "A list of all arguments is: $*"
  - echo "The 2nd argument is: $2"

  - > ./foo.sh texas rose bowl
  - Calling command is: ./foo.sh
  - Total # of arguments is: 3
  - A list of all arguments is: texas rose bowl
  - The 2nd argument is: rose

In tcsh, you can also reference individual arguments with $argv: eg. $1 = $argv[1]

Shell Scripting

More UNIX Commands for Programmers

– man –k	Search man pages by topic
– time	How long your program took to run
– date	print out current date/time
– test	Compare values, existence of files, etc
– tee	Replicate output to one or more files
– diff	Report differences between two files
– sdiff	Report differences side-by-side
– wc	Show number of lines, words in a file
– sort	Sort a file line by line
– gzip	Compress a file
– gunzip	Uncompress it
– strings	Print out ASCII strings from a (binary)
– ldd	Show shared libraries program is linked to
– nm	Show detailed info about a binary obj
– tar	Archiving utility
– uniq	Remove duplicate lines from a sorted file
– which	Show full path to a command
– file	Determine file type

Unix Scripting

References/Acknowledgements

- National Research Council Canada (Rob Hutten, Canadian Bioinformatics Resource)
- Intro. to Unix, Dave Hollinger, Rensselaer Polytechnic Institute
- Bash Reference Manual, http://www.faqs.org/docs/bashman/bashref.html
- Advanced Bash-Scripting Guide, http://db.ilug-bom.org.in/Documentation/abs-guide/
- TCSH Reference, http://www.tcsh.org/tcsh.html/top.html
- Unix in a Nutshell, A. Robbins, O'Reilly Media, 2006.

ssh and scp

tar

top

install a lib with sudo permission