MONASH University
Information Technology

FIT9134
Computer architecture and operating systems

Weeks 3 & 4

Operating System II :
File Management

File Management

File management systems allow users to store information in fundamental units called 'files'. What the file actually represents is defined by the system and/or the user.

Basic file management is typically provided by O/S, while additional file management functions may be performed by specialised software such as DBMS.

File system provides *connection between logical file structure and physical implementation*, creating **logical view** for user, and hiding **physical view**.

We will use Unix as the case study for file management in an operating system.

File management in Unix

- The Unix kernel includes the code to manage files.
- Unix utilities provide high-level file management functionality (eg. copy, delete, move, etc) to the users.
- Applications can also access the low-level file management services directly, through system calls (e.g. open, read, write etc).

for general users

for programmers

File Management in Unix

- O/S maintains a ***directory structure*** for each device, to facilitate location and organization of user files, and keeps track of free space, allocating space and reclaiming it as required.

- Provides naming, access/manipulation/storage, security/protection functions for files and directories.

- File system interacts with I/O subsystem to pass on requests for operations on storage devices

Files

- Like any operating system, Unix works on the concept of files, which are stored in a *file system*. But the concept of a file in Unix is more profound than in other operating systems :

  "Everything is a file"

- In Unix, files are simply a collection of bytes stored on the storage medium. They can represent any of the following types:
  - **Ordinary** Files
    - Data (e.g. a text file, program source code)
    - Executables (e.g. a Unix command, a shell script, etc)
  - **Directories**
    - A directory is another type of file in Unix - a speical "***file***" that can contain other files and other directories.
  - **Special** Files
    - Other types of files, eg. files that represent hardware devices like hard drives.

Files – naming conventions

- Unix is ***case-sensitive***. In general, ***most Unix commands are in lowercase letters*** (ie. Unix does not like uppercase letters!).

- Unix filenames are generally made up of lowercase and uppercase letters, digits, dots (.) and commas. Using spaces (or other "special" characters) in filenames can occasionally make file-handling difficult, so try to avoid them if possible.

- There is no notion of a file "extension" in Unix (unlike O/S such as Microsoft Windows). While files can have an extension, the extension (ie. the bit after a "dot") has no special meaning, and does not necessarily define the type of the file or indicate how it should be dealt with by an application.

Files - naming conventions

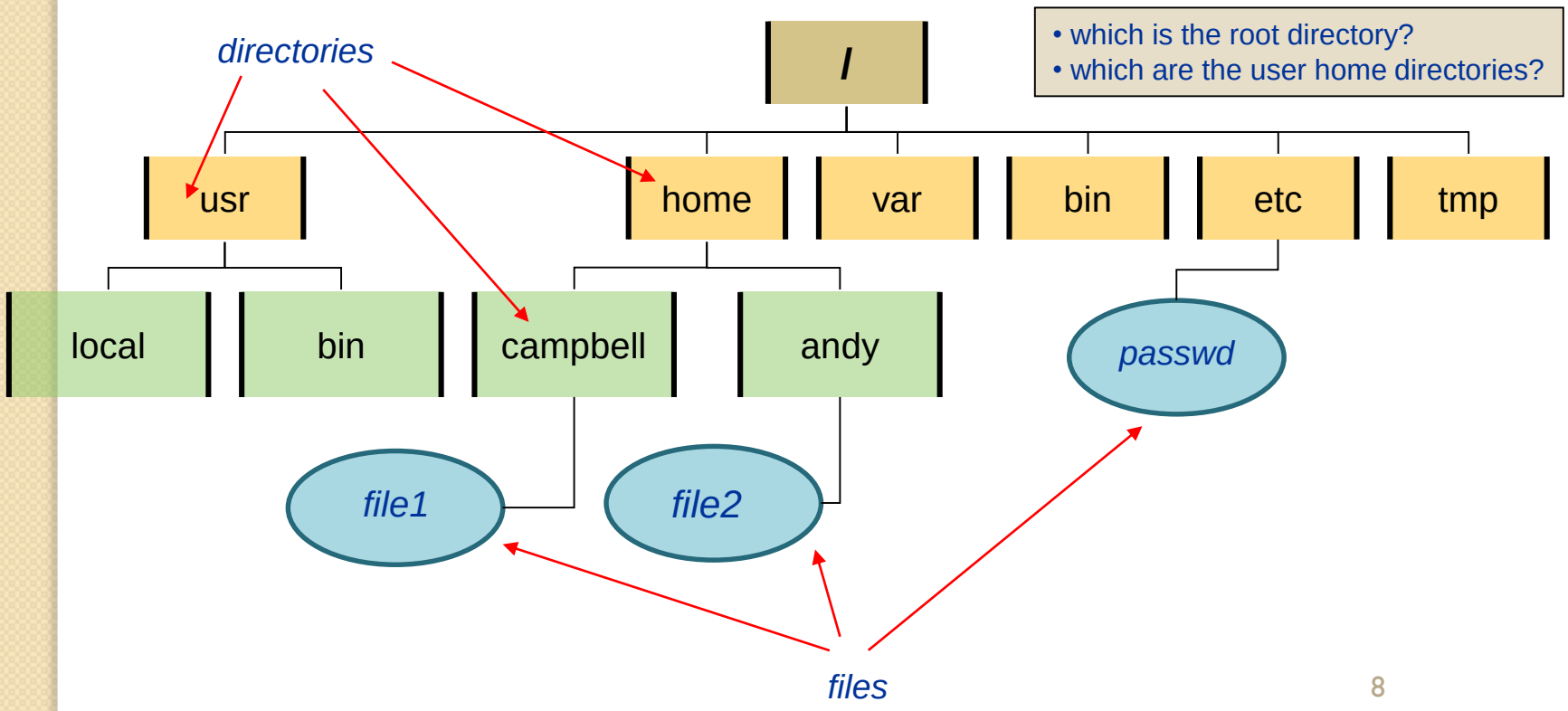
- In theory, any name can be used for a file or a directory (with the exception of the "root directory" which must always be named **/**). Unix is also very generous with the length of a filename.

- Try to avoid using special characters.

- However, important system files and directories are generally given the same "standard" names on Unix systems. Examples of some common/typical ***system directories*** include (but are not limited to) **`/etc`**, **`/bin`**, **`/home`**, **`/bin`**, **`/mnt`**, **`/usr`**, **`/var`**, **`/tmp`**, **`/proc`** and **`/lib`**.
  - you should not modify/delete these system directories/files unless you know what you are doing!

The UNIX File System

- UNIX stores files on the disk in a ***hierarchical*** structure.
- The top of the hierarchy is referred to as the ***root directory*** and is always named */*
- Eg. a typical Unix file system *might* look like :



The Working Directory

- When working on a UNIX system, you are always working within a particular directory in the file system. This is called either the *working directory* or the *current directory*.

- When you first log in to the system, your working directory is set to a directory that is unique to you as a user; your ***home directory***. The home directory is the directory which belongs to you and will contain your files (and you have full control over these files).

Paths (or Pathnames)

- A "path" or "pathname" represents the position/location of a file (or directory) in the file system hierarchy. Directories in the path are separated by the / symbol,
  - Eg. **/home/andy/week1**, **/etc/passwd**, **../week2**

- Special directory names used in paths:
  - . = ***current*** directory
  - .. = ***parent*** directory (the directory immediately above the current directory in the hierarchy)
  - ~ = the user's ***home*** directory

Absolute and Relative Paths

- A path that begins with */* is called an *absolute path* and indicates the "absolute" position of the file regardless of the directory in which the user is currently working, eg. **/home/andy/letter**, **/home/david/week3**

- A path that starts from the current directory is called a *relative path* and indicates the position of the file ***relative*** to the directory in which the user is currently working, eg. **week3/exerciseQ2, ../myfolder/test/mydocument, ../../week5**

- We generally regard a path like **~/week3** to be an absolute path (even if it does not start with a */*). Can you see why?

Ownership of files

- Files are "owned" by a user. We can check a file's ownership using the **ls** command with the long format, eg :

```
$ ls –l          ← Note : the '$' (the command prompt) is not part of the command!
drwxr-xr-x 18 cheng  fit9134   2048 Nov 12  2013 WWW
drwx------  3 cheng  fit9134   4096 Sep 26  2013 mail
-rw-r--r--  1 cheng  fit9134   1313 May  7 13:15 letter1
drwxr-xr-x  5 cheng  fit9134   2048 Aug 16  2013 tmp
drwxr-xr-x  4 cheng  fit9134   2048 Aug  5 13:48 week6
```

** more about this concept later **

A special file : /etc/passwd

- On Unix systems, there is a special file named "*passwd*" in the directory /etc. This file holds important information about all the ***users*** in the system. For instance, you may find in this file entries such as :

```
cheng:x:500:700:Andy Cheng:/home/cheng:/bin/tcsh
cwilson:x:501:701:Campbell Wilson:/home/cwilson:/bin/bash
```

- There is also a similar file, /etc/group, which holds information about all the ***groups*** in the system.

More on Files

- File system is "flat" under Unix kernel (i.e. there is no real distinction between directories and files under Unix kernel).
  - a device is represented as a file.
  - a program is a file.
  - a directory is really also a file (!)

- however, in the user interface, the user is typically presented with a hierarchical view (eg. the folders as shown in a graphical File Manager)

More on Files

- The kernel does not identify files by names; it uses a unique number to identify a file called the *inode number.*

- The **`stat`** command shows detailed info about files, eg:

```
$ stat /home/cheng/recipe
  File: `/home/andy/recipe'
  Size: 11          Blocks: 8          IO Block: 4096   regular file
Device: 841h/2113d Inode: 556730      Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/   andy)   Gid: ( 1000/   andy)
Access: 2015-08-09 19:32:27.875891467 +1000
Modify: 2015-08-09 19:32:27.875891467 +1000
Change: 2015-08-09 19:32:27.875891467 +1000
```

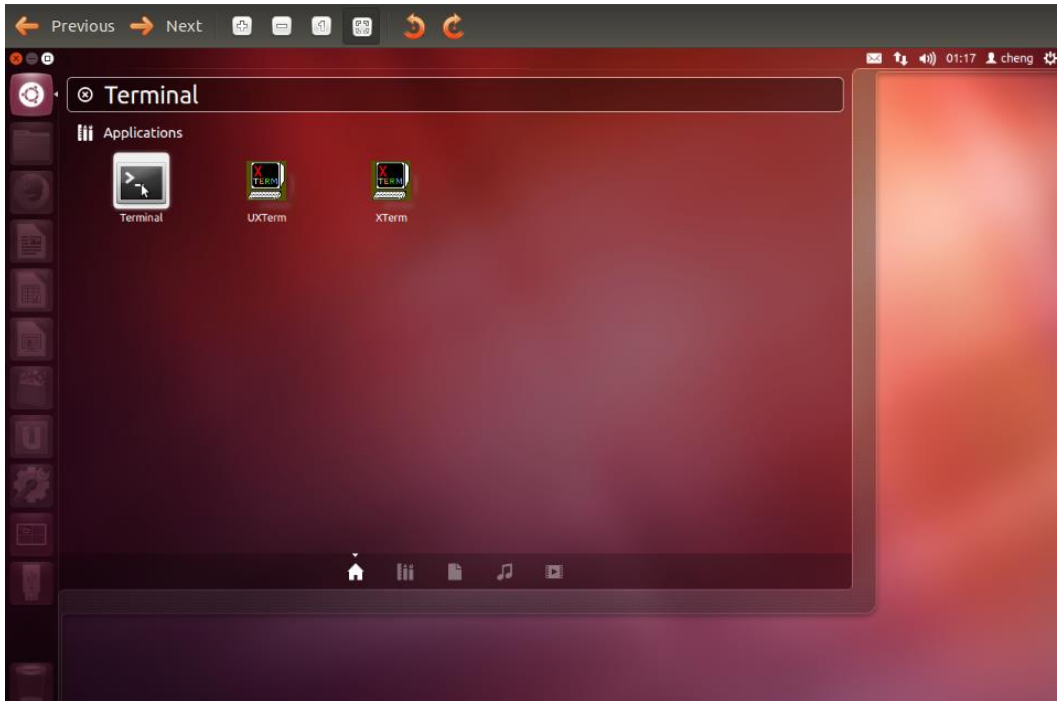
Using Unix Commands

- The original Unix O/S only allows users to interact with it via text commands (ie. user types in commands via a terminal). This is known as the "***Command-Line***" interface.

- Modern day Unix usually also provides a ***Graphical*** interface, allowing user to interact via pointing-and-clicking using devices such as a mouse. This makes it easier for the casual user to interact with the O/S.

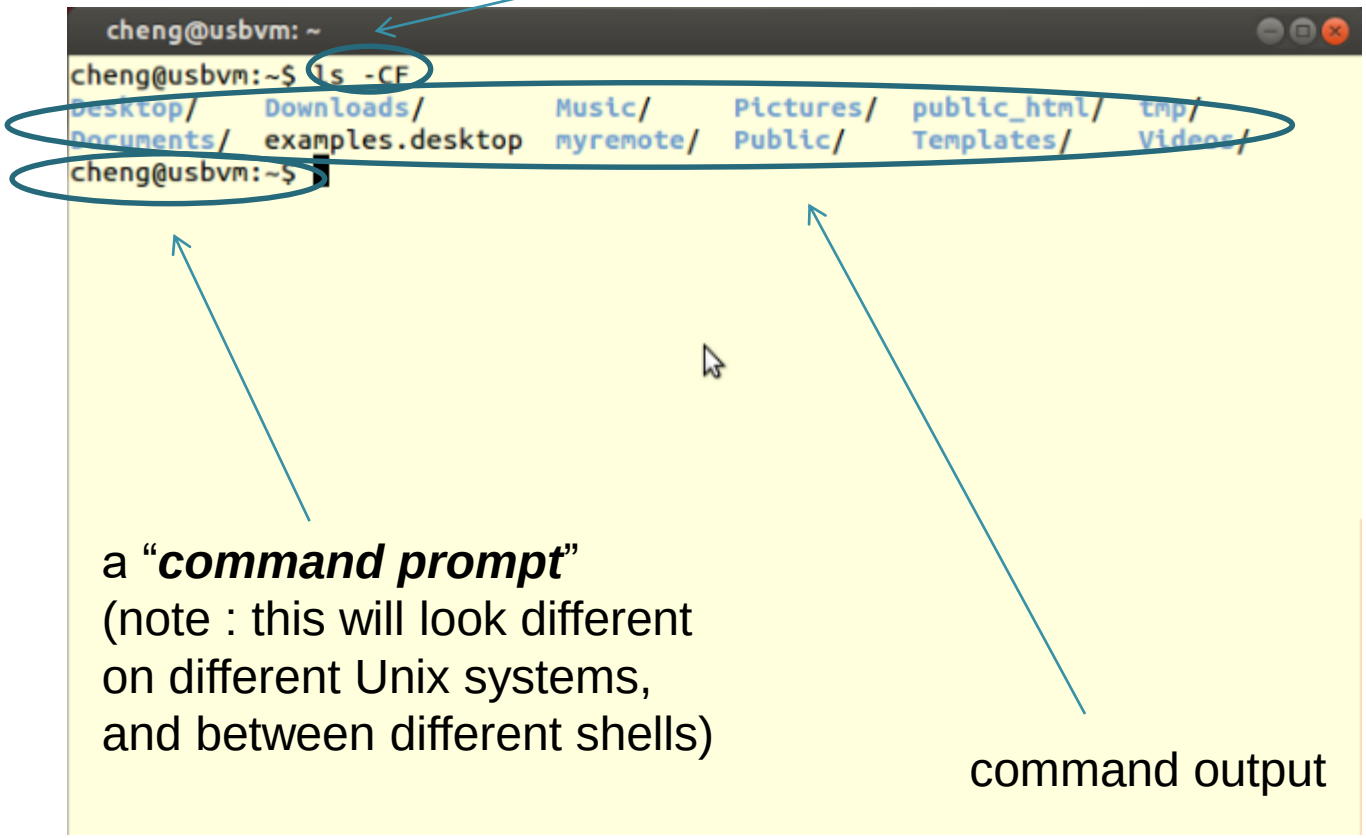
Graphical or Command-Line Interface???

- The command-line interface is still popular among advanced Unix users, because it is generally faster & more flexible.
  - A "*Terminal Window*" is now typically provided for this purpose, eg. in Ubuntu, the user can activate that window via the Dash Home menu or shortcut-key (***ctr-alt-T***)

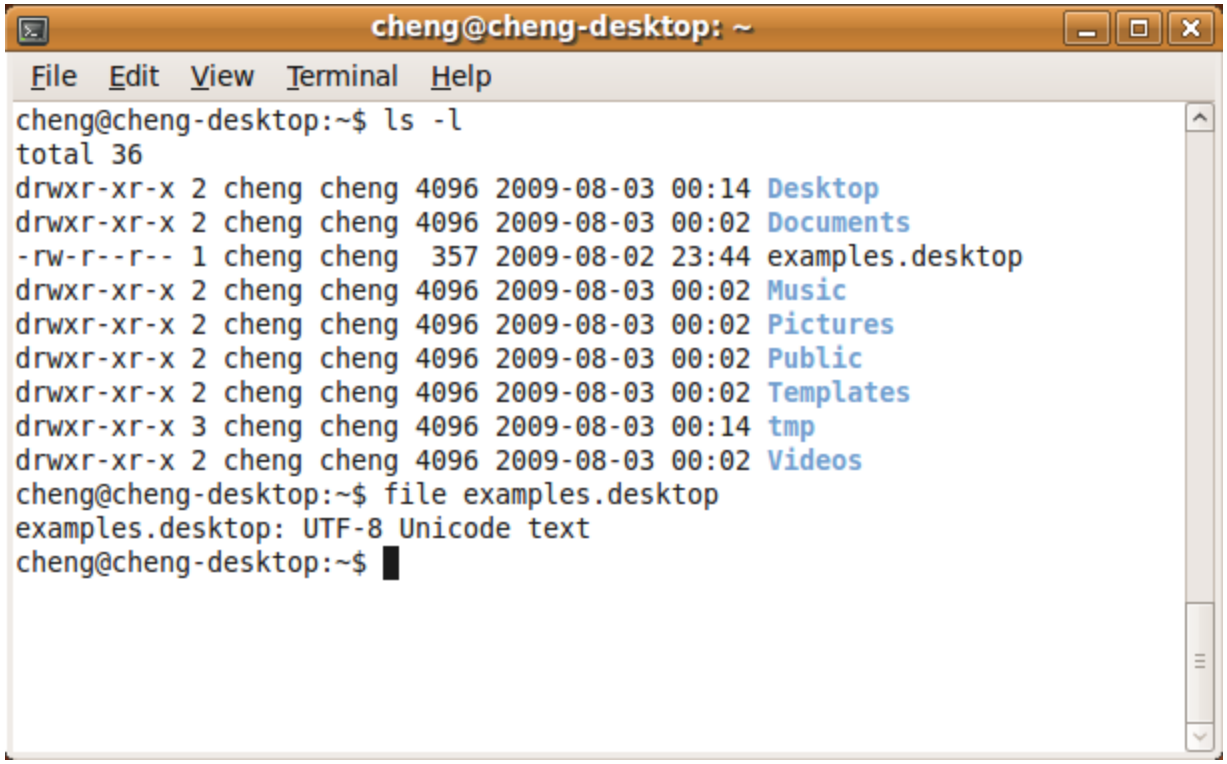


Command-Line Example

- In the command-line window, commands are typed in by the user, eg. :



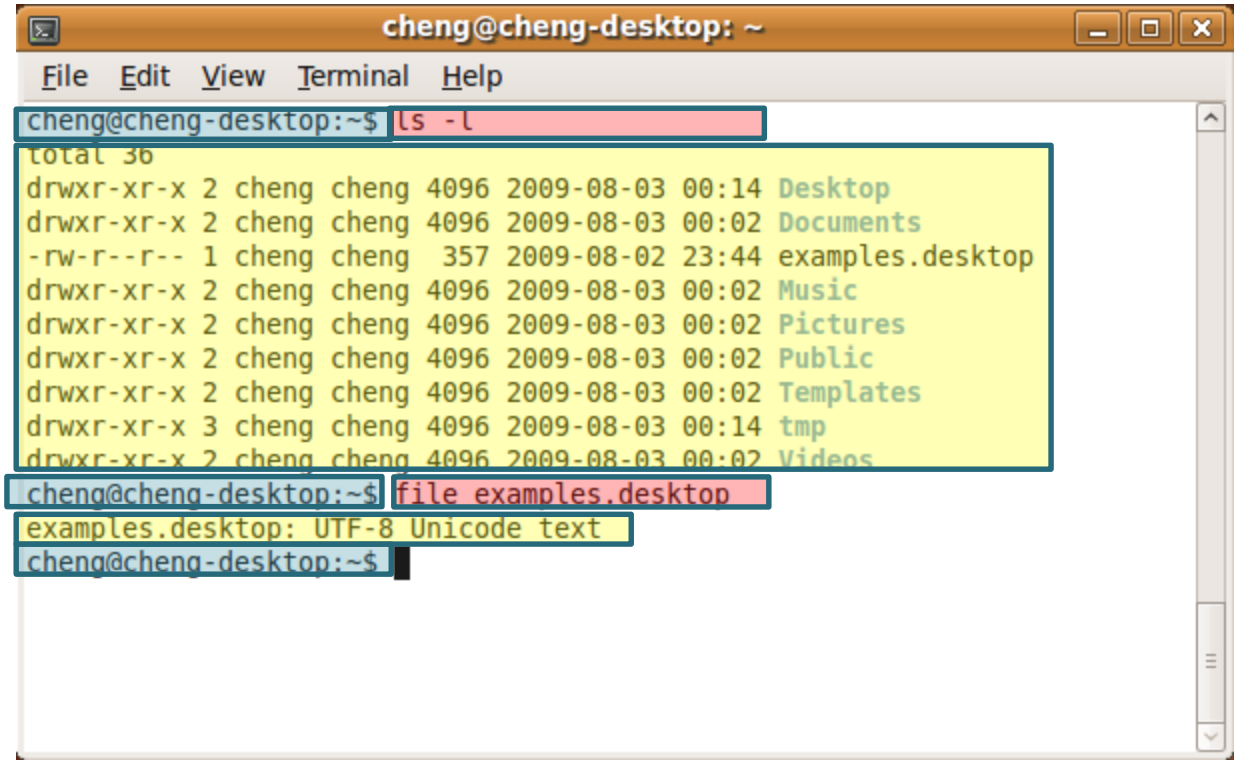
A simple command example



In the diagram above, can you identify examples of these?

- a *command*
- a *command output*
- a *command prompt*

A simple command example



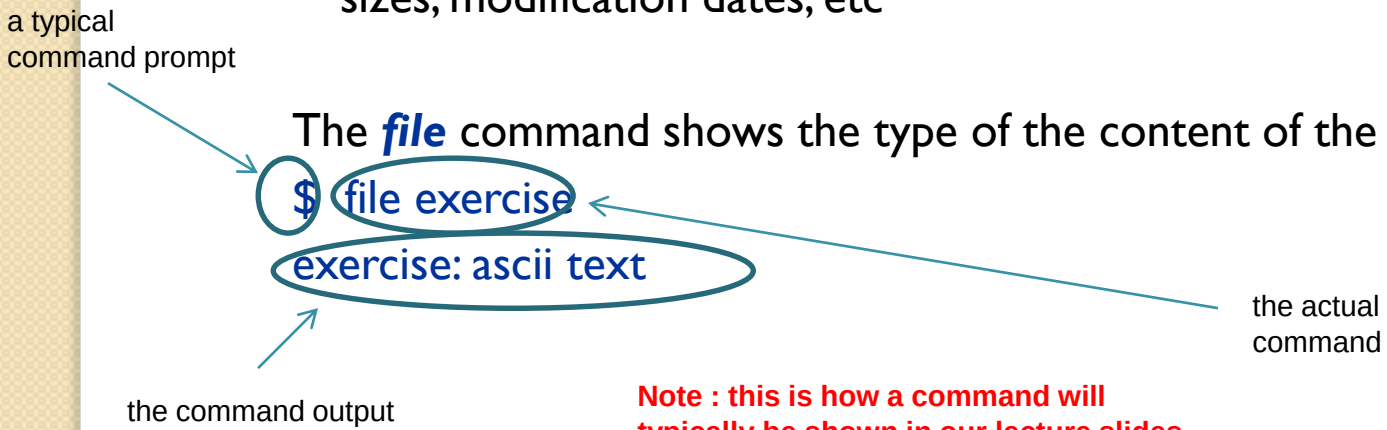
In the diagram above, can you identify examples of these?

- a *command*
- a *command output*
- a *command prompt*

File types – as shown by ls -l

Character	File type
-	regular (ordinary) file
d	directory
b	buffered special file (e.g. a disk drive)
c	unbuffered special file (e.g. a terminal)
l	symbolic link
p	pipe
s	socket

The command ***ls –l*** will show things such as file types, permissions, file sizes, modification dates, etc



Pre-reading for Week 4

- The next few slides (**Slides #23-28** – on file permissions and how to change them) are important for your Week 4's lab tasks. Please make sure you study them (plus do some onlint research on these topics) ***before*** attending the labs.

File permissions

- Three levels of permissions :
  - the ***user***
  - the user's ***group*** and
  - ***others*** who have account on the system
- Three kinds of permissions (for each level) :
  - ***read***, ***write*** and ***execute***
  - these have the usual meaning for ordinary files (for directory files – x means something different – more later)

Hence 9 different combinations in total

File permissions

- A total of nine (9) binary ***bits*** representing the permissions:

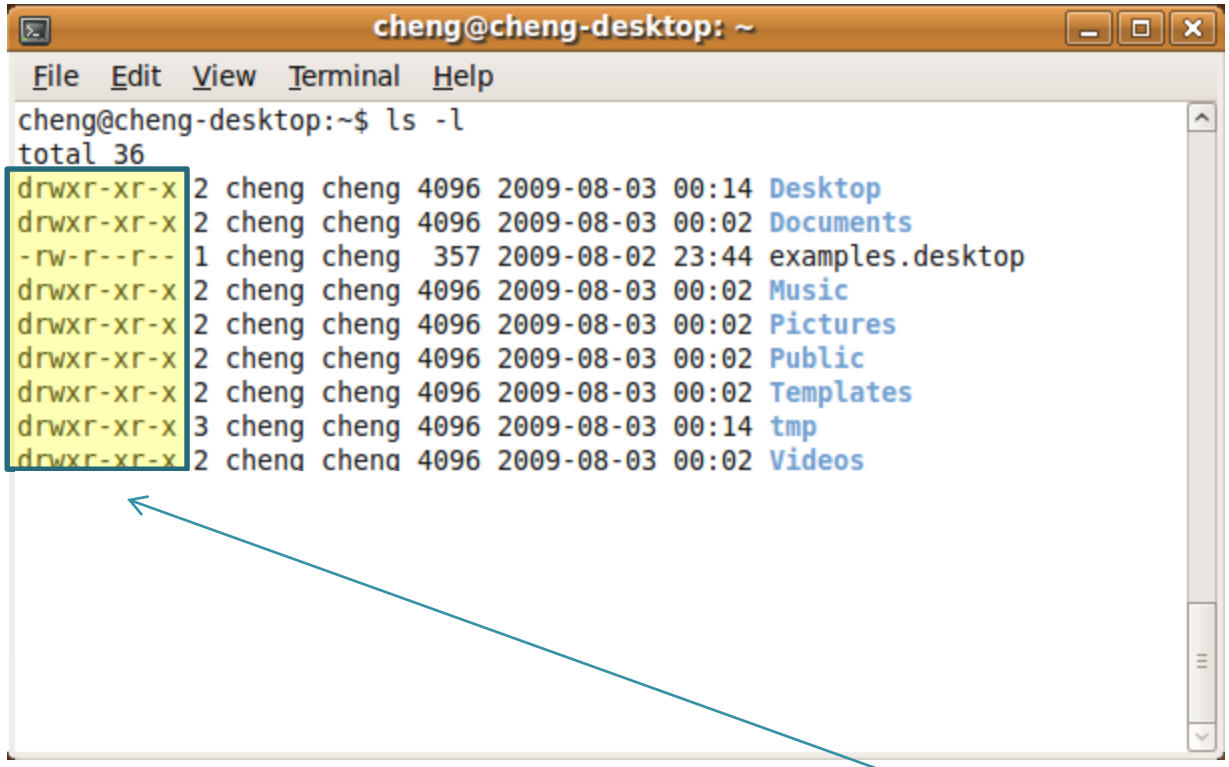
user	group	others
r/- w/- x/-	r/- w/- x/-	r/- w/- x/-

  - a "**-**" indicates the permission is "off", eg. **rwxrw-r--**
  - an example to follow shortly

- A user can choose to restrict access to his files/directories, so that other users may or may not access them.

- A ***Superuser*** (the "root" user) has access to all files irrespective of permissions.

Using ls -l to show file permissions



In the diagram above, the file permissions (marked in yellow) are shown as a ***9-bit*** pattern (**rwxrwxrwx**), plus a "***file-type***" bit at the front (normally "**-**" for file, "**d**" for directories)

File permissions : example using *ls -l*

$ ls -l examples

-rw-r--r-- 1 cheng users 357 2009-08-02 23:44 examples

Output explanations :

- The permission mode of this file is **read** and **write** for the owner, **read** only for the group and **read** only for others
- There is 1 **hard link** (more on this later)
- The user-id of the file's owner is ***cheng***
- The group-id of the file is ***users***
- The size of the file is **357** "blocks" – NB. block size can vary between systems
- The file was last modified on **2009-08-02 23:44**
- The file name is ***examples***

The option "*-l*" in the command above is to request the output in ***long*** format

There is another option, "*-h*", which will make ls display sizes in "human readable" format (eg. 8K, 555M, 4G, etc)

Change file permission (chmod)

Syntax: chmod [-R] *who* [*op*] [*permission*] *file-list*

- *who* is one of:
  - u — user — owner of the file
  - g — group — group to which the owner belongs
  - o — other — all other users
  - a — all — can be used in place of u,g,o
- *op* is one of:
  - + — add permission
  - - — remove permission
  - = — set permission (reset all other permissions)
- *permission* is one or more of : *r, w, x*

- Note : this is typical of a Unix command – the command is given with some option(s) & the actual operands)

chmod: Examples

You can also combine the options,
eg. chmod u+x,o+w temp

NB. no space here

```
ls -l temp
  -rw-rw-r--  1 cheng  users     57 Jul 12 16:47 temp

chmod a=rw temp          (set all permissions)
      ⇩
  -rw-rw-rw-  1 cheng users     57 Jul 12 16:47 temp

chmod o-w temp           (set others' permissions)
      ⇩
  -rw-rw-r--  1 cheng users     57 Jul 12 16:47 temp

chmod u+x temp           (set user's permissions)
      ⇩
  -rwxrw-r--  1 cheng users     57 Jul 12 16:47 temp
```

Note how the **chmod** commands change the file's permissions

File access for processes

- When a process executes, it has four id's:
  - a ***real*** user-id
  - an ***effective*** user-id
  - a ***real*** group-id
  - an ***effective*** group-id

- these id's determine a process's access permissions to files/directories.

Real versus Effective user id's

- *Real UID* is the *UID* of the user that created *THIS* process – ie. the user who executes/runs the program.

- *Effective UID* is used to evaluate privileges of the process to perform a particular action.

- This distinction is useful, since it allows a normal user to sometimes execute processes which require higher access privileges than he has,
  - eg. a build-in program may be available for everyone to execute, but may need some super-user privileges to carry out certain system-related tasks. So we can change the ***effective UID*** of that process to make the O/S thinks that the super-user is executing that program (instead of the user who actually runs the program).

File access for processes

- When a process tries to access a file, which of the three (***User***, ***Group***, or ***Others***) ownership permission applies?

- The general rules are as follows:
  - If the process's effective user-id is same as the owner of the file then ***User*** permissions apply
  - Otherwise, if the process's effective group-id is the same as file's group id then ***Group*** permissions apply
  - Otherwise, ***Others*** permissions apply

- In other words, what a user is permitted to do to a file depends on whether he owns the file, his group owns the file, or neither...

setuid and setgid

- A process' access privileges depend on who ***executes*** the process, not on who ***owns*** the executable program itself
  - This is safer in general, but not helpful in some (rare, but important) cases.
  - This can be overcome using special permissions: *set-user-id* and *set-group-id*
    - When a program with *setuid* permission is executed, the resulting process's effective *user-id* becomes that of owner of the program (instead of the user who executes that program).
    - Similarly with *setgid*.
    - In both cases, the real *uid* and *gid* are not affected

Example of using set-uid

- */etc/passwd* file stores the encrypted passwords (plus other info) of all registered users in the system
  - *$ ls -l /etc/passwd*

the password file

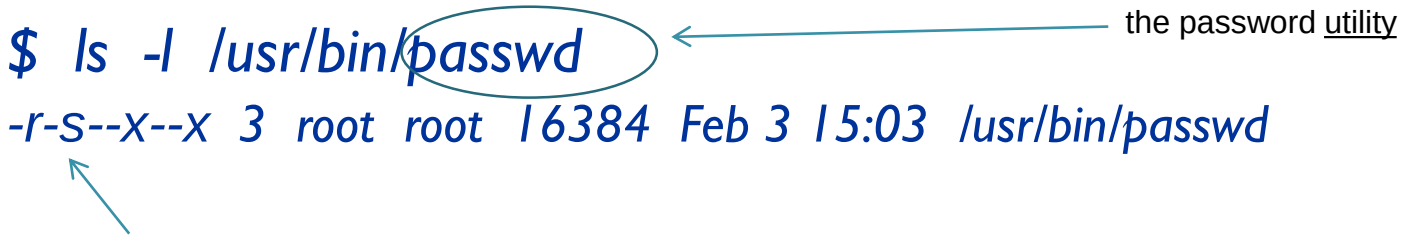
who has access to this file??

-rw-r--r-- 1 root root 29757 Jul 23 9:05 /etc/passwd

- The command *passwd* (in */usr/bin*) needs to be used by any user to change his/her own password, i.e. to modify */etc/passwd*. How can this be possible when a normal user does not have write permission to the */etc/passwd* file?
  - the solution : applying setuid on the *passwd* command

Example of using set-uid

- Let's examine the permissions of the */usr/bin/passwd* utility :



The *s* instead of the usual *x* in the permission field of the owner means:

when this command is run (by a normal user), the process will be granted the permissions corresponding to the process-owner, in this case, ***root***

- ***root*** has write access to */etc/passwd* file (see previous slide)
- this means : users cannot directly modify the */etc/passwd* file, but can change his/her own password by using the *passwd* command. This is required for obvious security reasons...

Binary, Octal & Hexadecimal notations....

- In computing, it is often convenient to express numbers in format other than decimal notations.

- Some useful readings/links :

  - http://kb.iu.edu/data/agxz.html
  - http://www.allaboutcircuits.com/vol_4/chpt_1/4.html
  - http://en.wikipedia.org/wiki/Octal
  - http://en.wikipedia.org/wiki/Hexadecimal
  - http://www.tonymarston.net/php-mysql/converter.php

Some commonly used **Octal** (base-8) values for file permissions

```
700     ==>     /* owner:  rwx------ */
400     ==>     /* owner:  r------- */
200     ==>     /* owner:  -w------ */
100     ==>     /* owner:  --x------ */

070     ==>     /* group:  ---rwx--- */
040     ==>     /* group:  ---r------ */
020     ==>     /* group:  ----w---- */
010     ==>     /* group:  -----x--- */

007     ==>     /* others: ------rwx */
004     ==>     /* others: ------r-- */
002     ==>     /* others: ------w- */
001     ==>     /* others: ------x */

4000    ==>     /* set user id on execution */
2000    ==>     /* set group id on execution */
```

Values of 0-7 can be used to indicate if a particular bit is “on or “off”

This sort of “shortcut” is commonly used in Unix commands

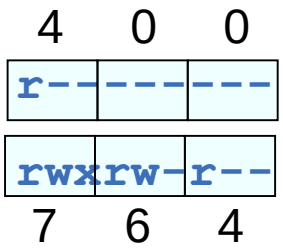
a ‘1’ means the corresponding bit is “on”

a ‘0’ means the corresponding bit is “off”

Examples :

chmod 400 file1 ====>

chmod 764 file2 ====>



(4 0 0 == 100 000 000)

(7 6 4 == 111 110 100)

Standard Input, Output and Error

- Remember, in Unix, *everything is a file…*
- Every time a shell is started, 3 files are opened automatically :

  stdin, *stdout*, *stderr*

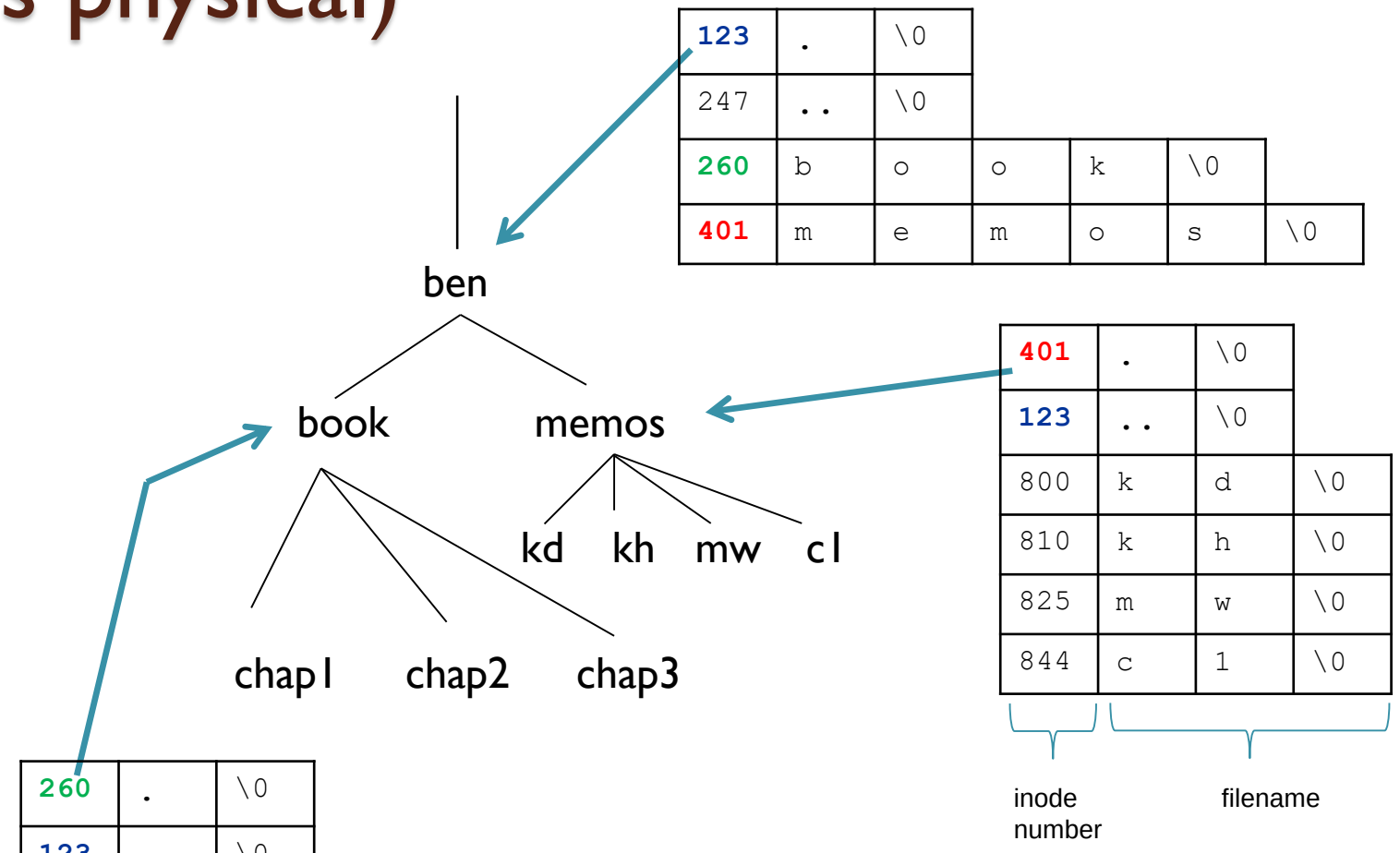
File	Default Device	File Descriptor
stdin	keyboard	0
stdout	screen	1
stderr	screen	2

- A process can then easily read/write to/from these "files"; this makes I/O programming relatively easy.

Directories (Unix)

- Unix directories consist of a series of directory entries.
- Each directory entry contains at least the inode number for the file and a character field containing the filename.
- The actual format of the directory is file-system dependent – and is hidden from the user.

Example: directory structure (logical vs physical)



123	.	\0				
247	..	\0				
260	b	o	o	k	\0	
401	m	e	m	o	s	\0

401	.	\0	
123	..	\0	
800	k	d	\0
810	k	h	\0
825	m	w	\0
844	c	1	\0

260	.	\0				
123	..	\0				
566	c	h	a	p	1	\0
567	c	h	a	p	2	\0
590	c	h	a	p	3	\0

Directory permissions (Unix)

- ***Read*** permission
  - Processes can list names and subdirectories within the directory
- ***Write*** permission
  - Processes can alter the directory, i.e. create and remove existing files. Note: ***write*** permission on a file is required to modify its contents (this is not altering the directory so write permission on the directory does not say anything about this) but *not* to remove it if the directory has write permission!
- ***Execute*** permission
  - Allows a user to change into the directory (*cd* from the shell or *chdir* as a system call). In addition, to open a file or execute a program a user must have ***execute*** permission on all the directories leading to the file as specified in the file's absolute pathname.

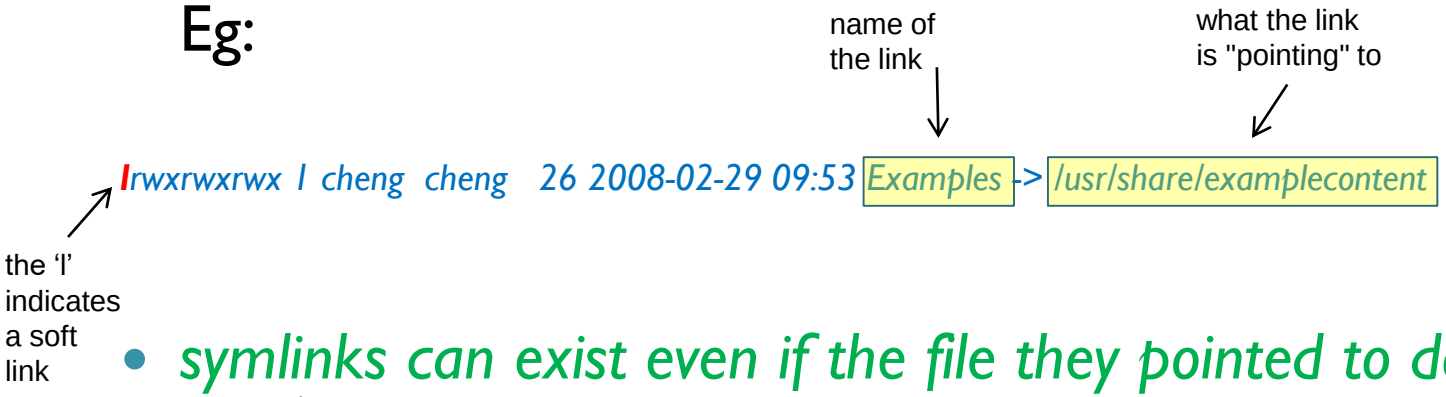
Links – Hard Links

- A ***hard*** link is a pointer/reference to a file - *every file has at least one hard link to it*.
- The link is how the operating system associates a file *name* with the address of the actual data on the storage device.
- Additional links can be created to allow sharing of files or access through a different name.
- Hard links can be created in Unix using *ln*.
- *A file exists until the last hard link to it is removed*. When the last hard link is removed, the space previously used by the file is marked for re-use.

Links – Symbolic Links

- *a **symbolic*** link (or ***symlink,*** or ***soft*** link) is a ***file*** that contains a pointer to another file.

Eg:



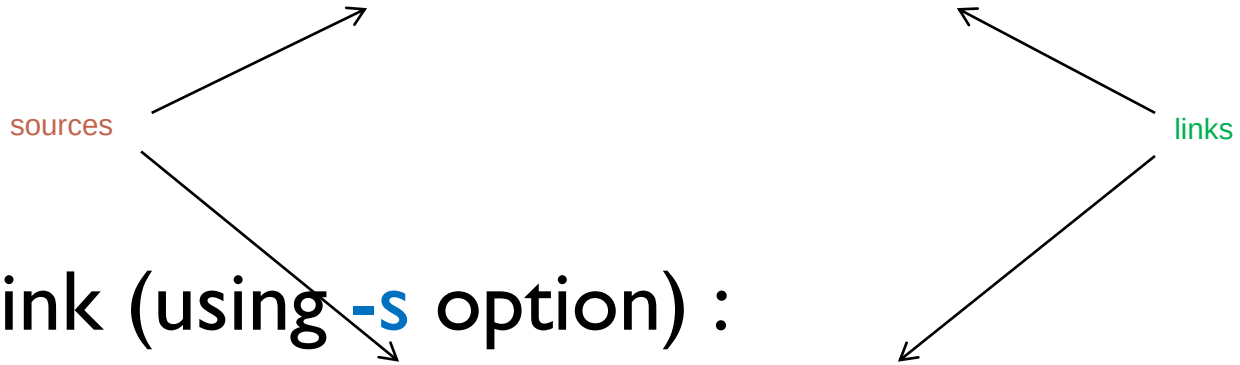
- *symlinks can exist even if the file they pointed to does not exist*!

- symlinks must be used if a link is to span filesystems (sometimes referred to as partitions). Hard links cannot be used on directories, and do not work across different filesystems.

Unix *ln* command examples

- Hard link (default) :

```
ln  ~/week4/myfile  hardlink-1
```



- Soft link (using -s option) :

```
ln  -s  ~/week4/myfile  softlink-1
```

These links provide some useful explanations about ***Unix Links*** :

- https://www.geeksforgeeks.org/soft-hard-links-unixlinux/
- http://linuxgazette.net/105/pitcher.html