

CS 318 Principles of Operating Systems

Fall 2021

Lecture 15: File Systems

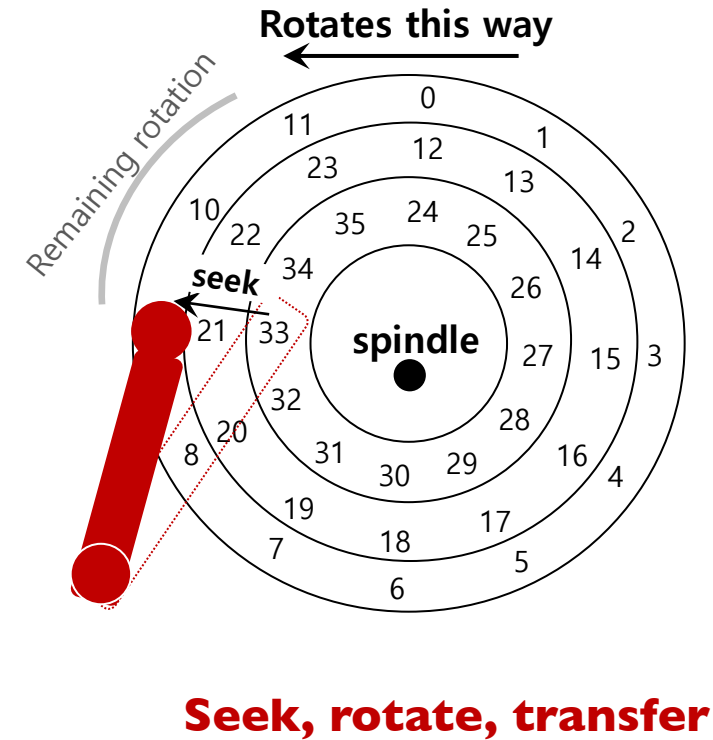
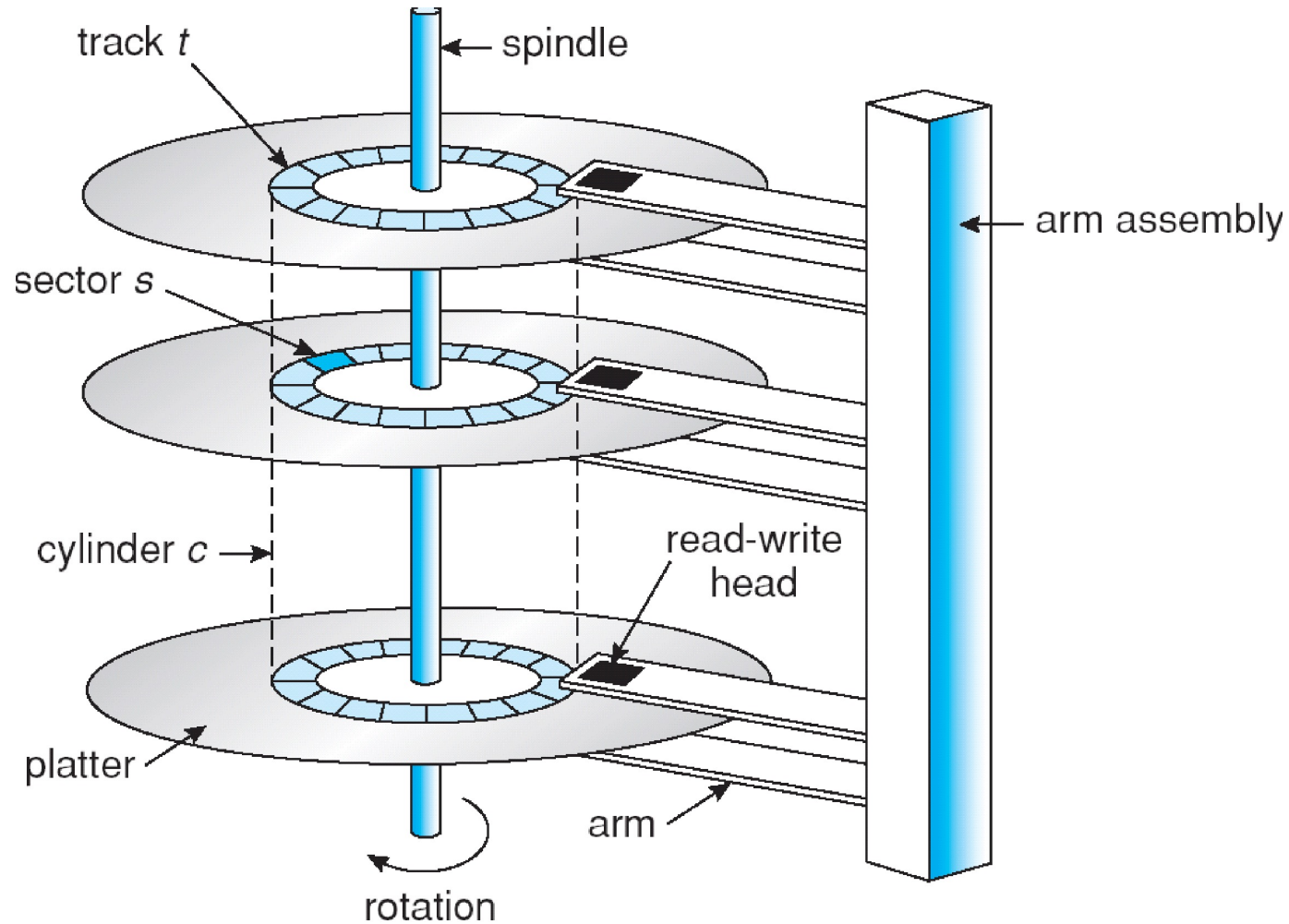
Prof. Ryan Huang



JOHNS HOPKINS

WHITING SCHOOL
of ENGINEERING

Recap: I/O & Disks



File System Fun

File systems: a challenging OS design topic

- More papers on FSES than any other single topic

Main tasks of file system:

- Don't go away (ever)
- Associate bytes with name (files)
- Associate names with each other (directories)
- Can implement file systems on disk, over network, in memory, in non-volatile ram (NVRAM), on tape, w/ paper.
- We'll focus on disk and generalize later

Today: files, directories

Files

File: **named bytes on disk**

- data with some properties
- contents, size, owner, last read/write time, protection, etc.

How is a file's data managed by the file system?

- Next lecture's topic
- Basic idea (in Unix): a struct called an **index node or inode**
 - describe where on the disk the blocks for a file are placed
 - Disk stores an array of inodes, inode # is the index in this array

File Types

A file can also have a type

- Understood by the file system
 - Block, character, device, portal, link, etc.
- Understood by other parts of the OS or runtime libraries
 - Executable, dll, source, object, text, etc.

A file's type can be encoded in its name or contents

- Windows encodes type in name (.com, .exe, .bat, .dll, .jpg, etc.)
- Unix encodes type in contents (magic numbers, initial characters, e.g., #! for shell scripts)

Basic File Operations

Unix

`creat(name)`

`open(name, how)`

`read(fd, buf, len)`

`write(fd, buf, len)`

`sync(fd)`

`seek(fd, pos)`

`close(fd)`

`unlink(name)`

Windows

`CreateFile(name, CREATE)`

`CreateFile(name, OPEN)`

`ReadFile(handle, ...)`

`WriteFile(handle, ...)`

`FlushFileBuffers(handle, ...)`

`SetFilePointer(handle, ...)`

`CloseHandle(handle, ...)`

`DeleteFile(name)`

`CopyFile(name)`

`MoveFile(name)`

File Access Methods

FS usually provides different access methods (ways for accessing data in a file):

- **Sequential access**
 - read bytes one at a time, in order
 - by far the most common mode
- **Random access**
 - random access given block/byte number
- **Record access**
 - file is array of fixed- or variable-length records
 - read/written sequentially or randomly by record #
- **Indexed access**
 - file system contains an index to a particular field of each record in a file
 - reads specify a value for that field and the system finds the record via the index

What file access method does Unix, Windows provide?

Directories

Problem: referencing files

“Spend all day generating data, come back the next morning, want to use it.”

– F. Corbato, on why files/dirs invented

Users remember where on disk their files are (disk sector no.)?...

- E.g., like remembering your social security or bank account #

...People want human digestible names

- We use directories to map names to file blocks

Directories serve two purposes

- For users, they provide a structured way to organize files
- For FS, they provide a convenient naming interface that allows the separation of logical file organization from physical file placement on the disk

A Short History of Directories

Approach 1: Single directory for entire system

- Put directory at known location on disk. **If one user uses a name, no one else can.**
- Many ancient personal computers work this way

Approach 2: Single directory for each user

- Still clumsy, and running `ls` on 10,000 files is a real pain

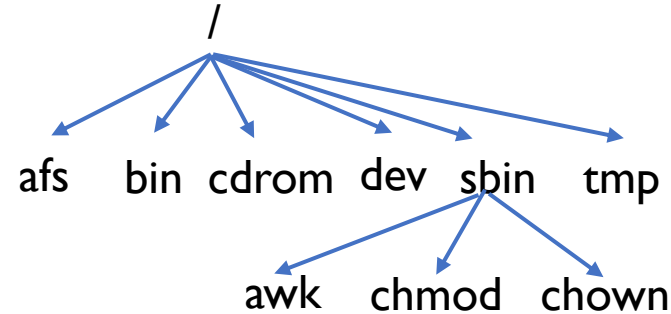
Approach 3: Hierarchical name spaces

- Allow directory to map names to files or other dirs
- File system forms a tree (or graph, if links allowed)

Hierarchical Directory

Used since CTSS (1960s)

- Unix picked up and used really nicely



Large name spaces tend to be hierarchical

- ip addresses, domain names, scoping in programming languages, etc.

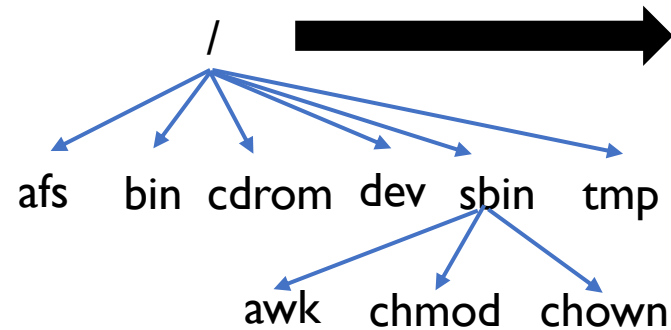
Directory Internals

A directory is a list of entries

- `<name, location>` tuple, location is typically the *inode #* (more next lecture)

Directories stored on disk just like regular files

- File type set to directory
- User's can read just like any other file
- Only special syscalls can write (why?)
- Inodes at fixed disk location
- File pointed to by the location may be another directory
- Makes FS into hierarchical tree



File content for '/'

```
<afs,1021>
<tmp,1020>
<bin,1022>
<cdrom,4123>
<dev,1001>
<sbin,1011>
...
```

Simple, plus speeding up file ops speeds up dir ops!

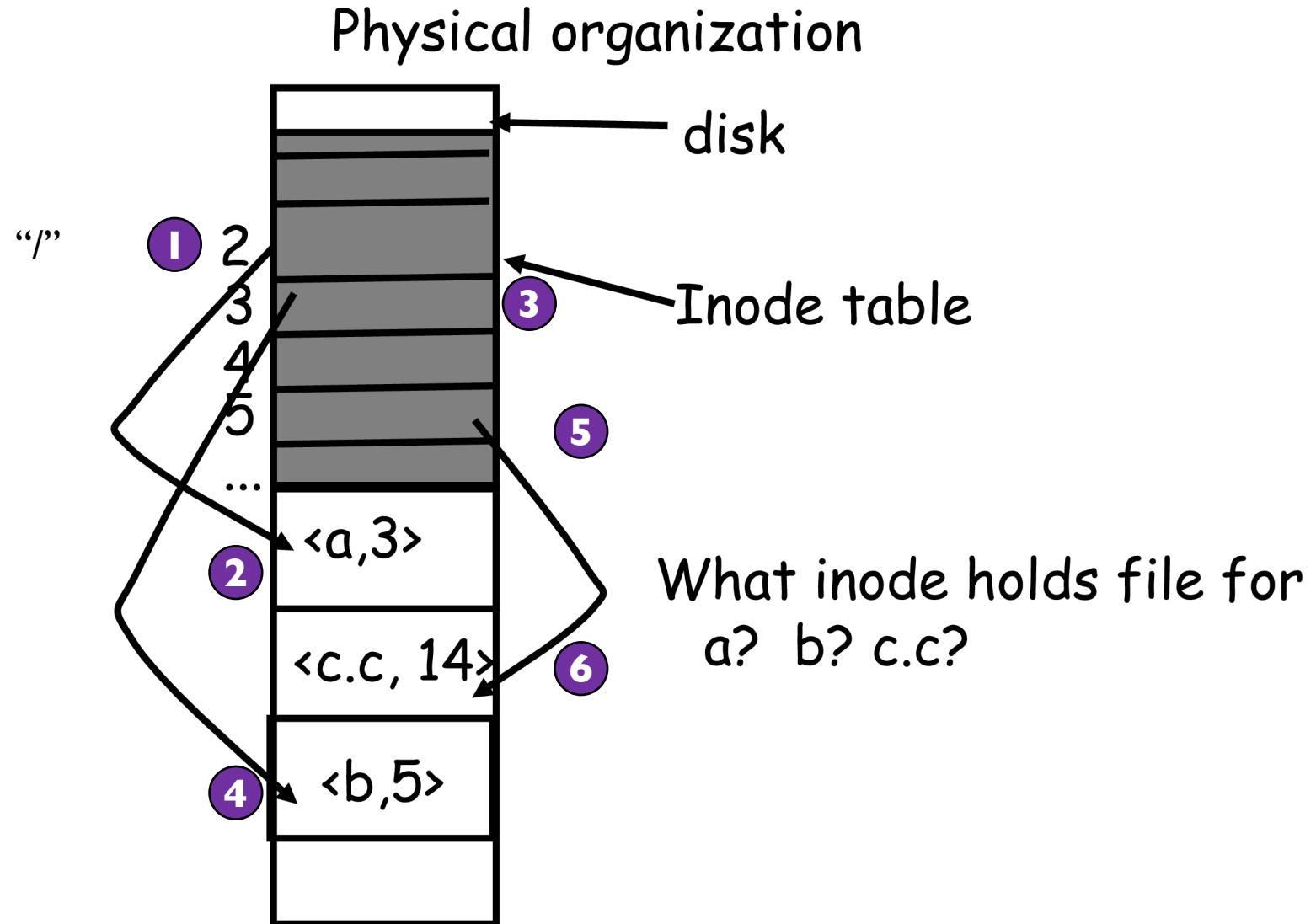
Path Name Translation (v1)

Let's say you want to open “/one/two/three”

What does the file system do?

- Directory entries map file names to location (inode #)
- Open directory “/”: Where? **Root directory is always inode #2**
- Search for the entry “one”, get location of “one” (in dir entry)
- Open directory “one”, search for “two”, get location of “two”
- Open directory “two”, search for “three”, get location of “three”
- Open file “three”

Unix Example: /a/b/c.c



Unix inodes and Path Search

Unix inodes are **not** directories

- Inodes describe where on the disk the blocks for a file are placed
- Directories are files, so inodes also describe where the blocks for directories are placed on the disk

Directory entries map file names to inodes

- To open “/one”, use Master Block to find inode for “/” on disk
- Open “/”, look for entry for “one”
- This entry gives the disk block number for the inode for “one”
- Read the inode for “one” into memory
- The inode says where first data block is on disk
- Read that block into memory to access the data in the file

Naming Magic

Bootstrapping: Where do you start looking?

- Root directory always inode #2 (0 and 1 historically reserved)

Special names:

- Root directory: “/”
- Current directory: “.”
- Parent directory: “..”

Some special names are provided by shell, not FS:

- User’s home directory: “~”
- Globbing: “foo.*” expands to all files starting “foo.”

Using the given names, only need two operations to navigate the entire name space:

- cd name: move into (change context to) directory name
- ls: enumerate all names in current directory (context)

Basic Directory Operations

Unix

Directories implemented in files

- Use file ops to create dirs

C library provides a higher-level abstraction for reading directories

- opendir(name)
- readdir(DIR)
- seekdir(DIR)
- closedir(DIR)

Windows

Explicit directory operations

- CreateDirectory(name)
- RemoveDirectory(name)

Very different method for reading directory entries

- FindFirstFile(pattern)
- FindNextFile()

Default Context: Working Directory

Cumbersome to constantly specify full path names

- In Unix, each process has a “current working directory” (cwd)
- File names *not* beginning with “/” are assumed to be relative to cwd; otherwise translation happens as before

Shells track a default list of active contexts

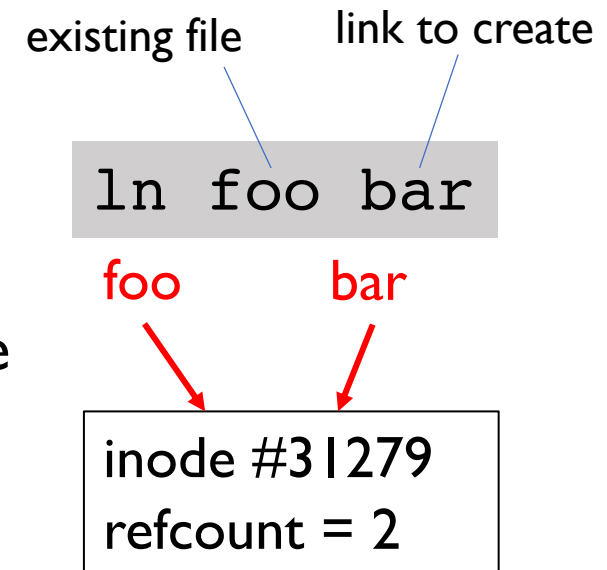
- A “search path” for programs you run
- Given a search path **A:B:C**, the shell will check in A, then B, then C
- Can escape using explicit paths: “./foo”

Example of locality

Hard Links

More than one dir entry can refer to a given file

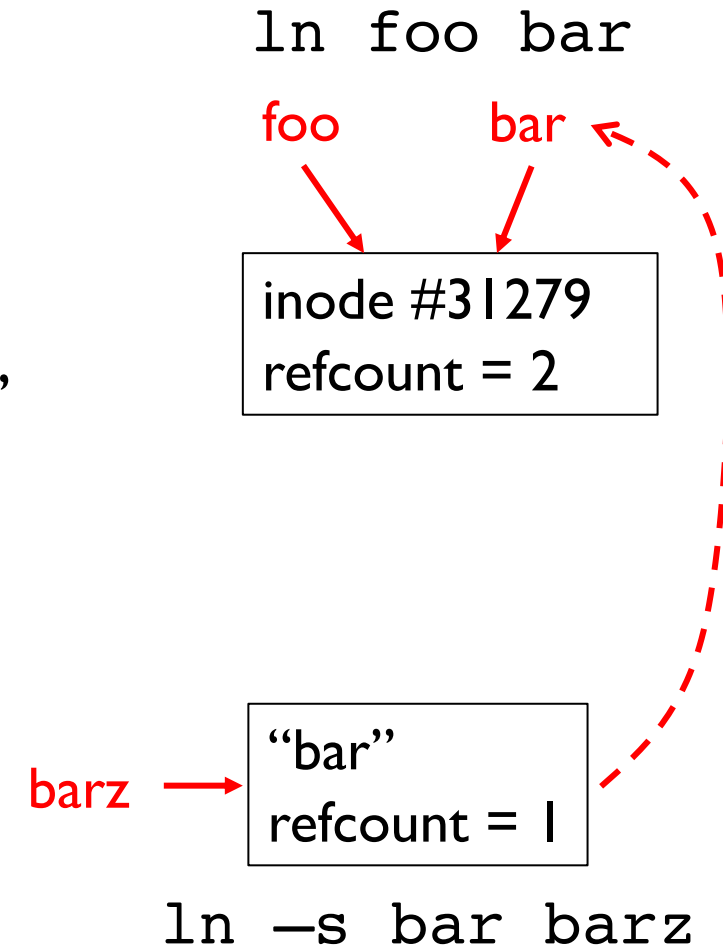
- Hard link creates a synonym for file
- Unix stores count of pointers (“hard links”) to inode
- If one of the links is removed (e.g., `rm`), the data are still accessible through any other link that remains
- If all links are removed, the space occupied by the data is freed.



Soft Links

Soft/symbolic links = synonyms for names

- Point to a file/dir name, but object can be deleted from underneath it (or never exist).
- Unix implements like directories: inode has special “symlink” bit set and contains name of link target
- When the file system encounters a soft link it automatically translates it (if possible).



File Sharing

File sharing has been around since timesharing

- Easy to do on a single machine
- PCs, workstations, and networks get us there (mostly)

File sharing is important for getting work done

- Basis for communication and synchronization

Two key issues when sharing files

- Semantics of concurrent access
 - What happens when one process reads while another writes?
 - What happens when two processes open a file for writing?
 - What are we going to use to coordinate?
- Protection

Protection

File systems implement a protection system

- Who can access a file
- How they can access it

More generally...

- Objects are “what”, subjects are “who”, actions are “how”

A protection system dictates whether a given **action** performed by a given **subject** on a given **object** should be allowed

- You can read and/or write your files, but others cannot
- You can read “/etc/motd”, but you cannot write it

Representing Protection

Access Control Lists (ACL)

For each object, maintain a list of subjects and their permitted actions

Capabilities

For each subject, maintain a list of objects and their permitted actions

Objects

	/one	/two	/three
Alice	rw	-	rw
Bob	w	-	r
Charlie	w	r	rw

ACL

Capability

Subjects

ACLs and Capabilities

Approaches differ only in how the table is represented

Capabilities are easier to transfer

- They are like keys, can handoff, does not depend on subject

In practice, ACLs are easier to manage

- Object-centric, easy to grant, revoke
- To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem

ACLs have a problem when objects are heavily shared

- The ACLs become very large
- Use groups (e.g., Unix)

Unix File Protection

What approach does Unix use in the FS?

- Answer: both

ACL: Unix file permissions

Capability: file descriptors

How are they used together?

- Conversion through `open()` system call

Converted to
capability

ACL check, expensive

```
int fd = open("file.txt", O_WRONLY);  
if (fd == -1)  
    exit(-1);
```

```
for (int i = 0; i < 100; i++)  
    write(fd, buf + i * 4, 4);
```

Use capability from then on

Summary

Files

- Operations, access methods

Directories

- Operations, using directories to do path searches

Sharing

Protection

- ACLs vs. capabilities

Next Time...

Read Chapter 41, 42