SE3331-1 (2023 Fall)

IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

File System API and Disk I/O

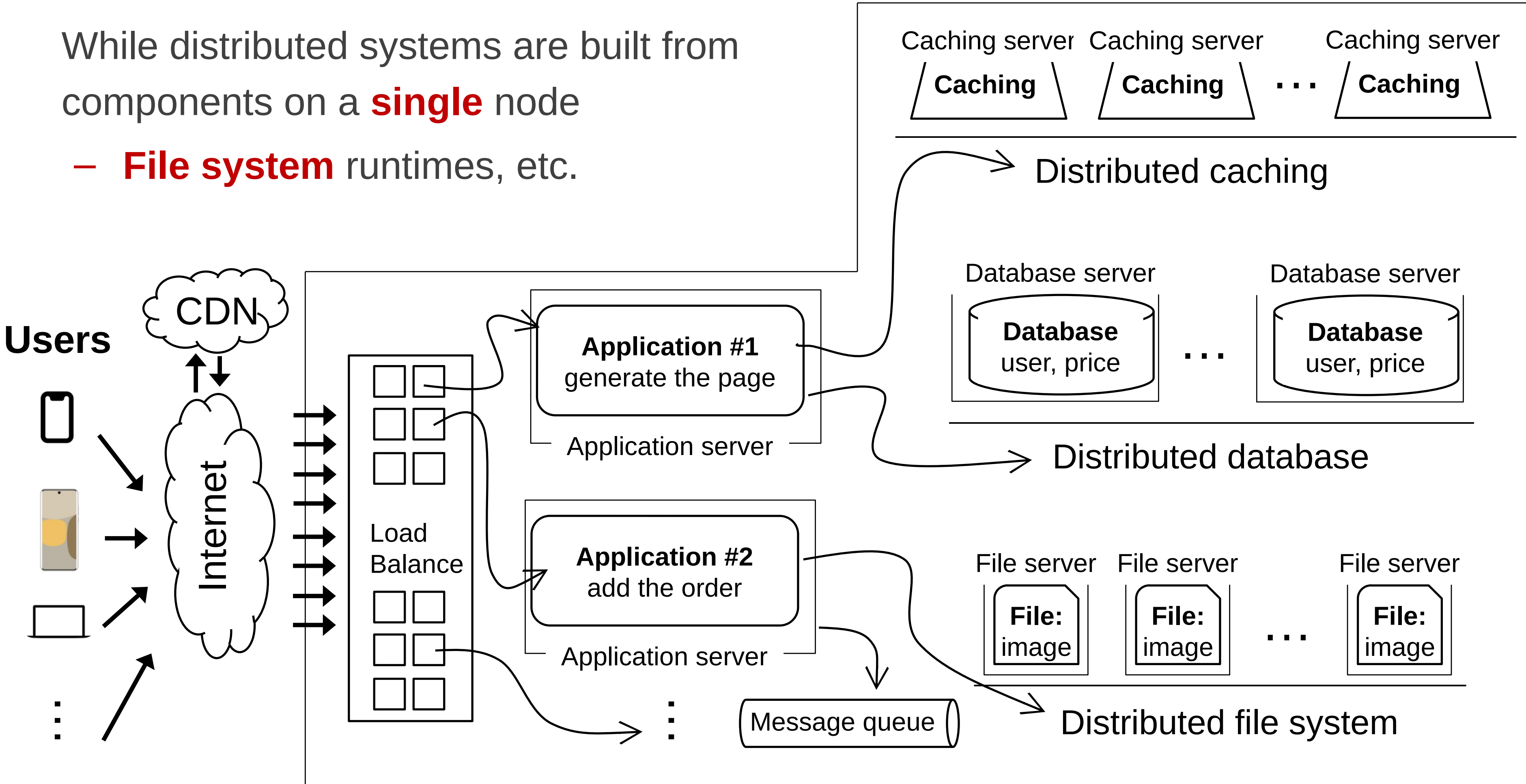
IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Review: Large-scale websites on distributed systems

While distributed systems are built from components on a **single** node

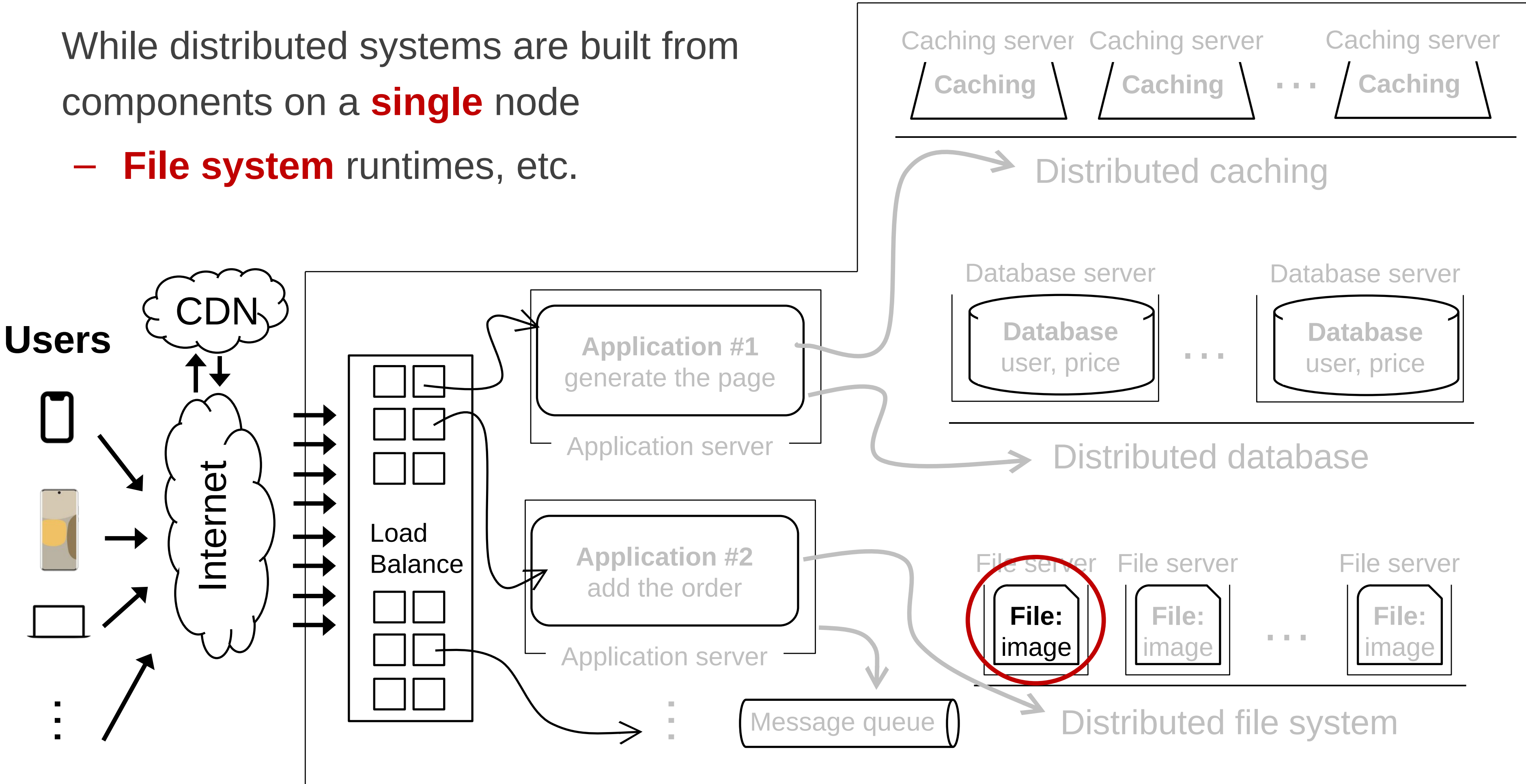
- – **File system** runtimes, etc.



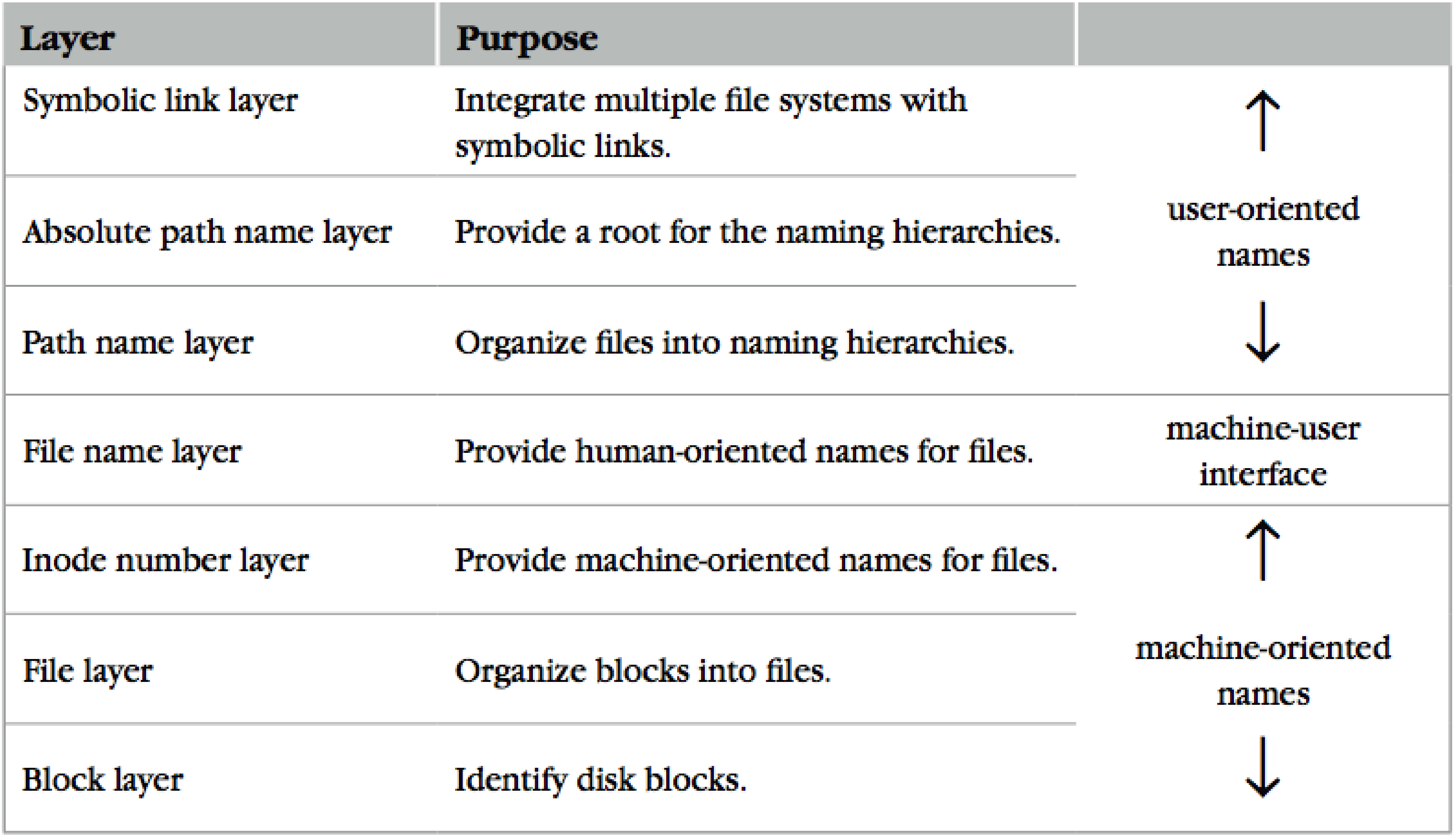
Review: Large-scale websites on distributed systems

While distributed systems are built from components on a **single** node

- **File system** runtimes, etc.

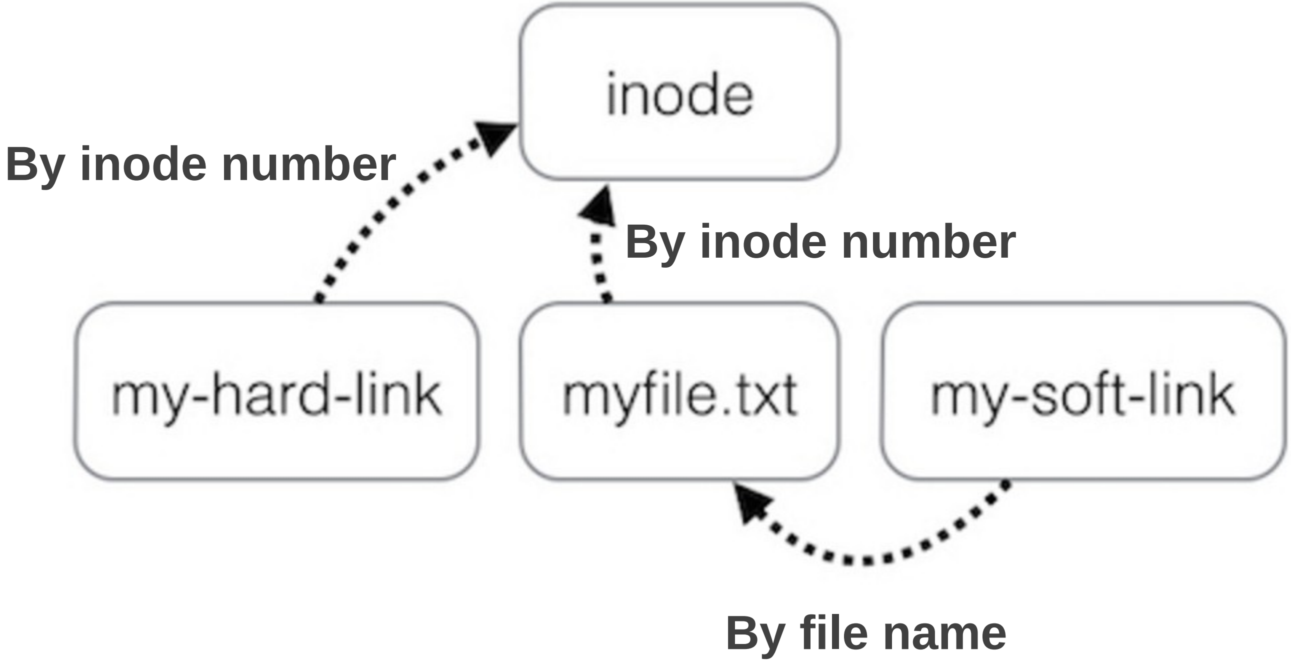


Review: The Naming Layers of the UNIX FS (version 6)



Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	↑ user-oriented names ↓
Absolute path name layer	Provide a root for the naming hierarchies.	
Path name layer	Organize files into naming hierarchies.	
File name layer	Provide human-oriented names for files.	machine-user interface
Inode number layer	Provide machine-oriented names for files.	↑ machine-oriented names ↓
File layer	Organize blocks into files.	
Block layer	Identify disk blocks.	

Review: Two Types of Links (Synonyms)



Summary of File System's 7 Layers

File name is not part of a file

- Name is **not** a part of an inode
- Name is data of a directory, and metadata of a file system
- One inode can have several names (hard links)

Hard links are equal

- If a file has two names, both are links (instead of "a link and a name")

Directory size is small

- Only mapping from name to inode number
- The term "folder" is somewhat misleading

Implementing the file system API

Implementing the File System API

API

- `CHDIR, MKDIR`
- `CREAT, LINK, UNLINK, RENAME`
- `SYMLINK`
- `MOUNT, UNMOUNT`
- `OPEN, READ, WRITE, APPEND, CLOSE`
- `SYNC`

Implemented as system calls to user applications

- Kernel has many sets of function pointers implementing the API
- Each set is specific to a FS (chose at mount point)

Sidebar: open() vs. fopen()

Difference between open() and fopen()?

- open() returns an fd; fopen() return a FILE*
- open() is a system call of OS; fopen() is an API of libc

Questions

- Which one can be used on both Windows and Linux?
- Which one has better performance?
  - fopen() provides you with buffering I/O that may turn out to be a lot faster than what you're doing with open()

File Meta-data -- inode

Owner ID

- User ID and group ID that own this inode (can be changed by chown)

Types of permission

- Owner, group, other
- Read, write, execute

Time stamps

- Last access (by READ)
- Last modification (by WRITE)
- Last change of inode (by LINK)

```
struct inode
  integer block_nums[N]
  integer size
  integer type
  integer refcnt
  integer userid
  integer groupid
  integer mode
  integer atime
  integer mtime
  integer ctime
```

OPEN a File

Check **user's permission**

Update **last access time**

Return a short name for a file

- File descriptor **(fd)**
- fd is used by **`READ, WRITE, CLOSE`**, etc.

File Descriptor

Each process starts with three **default open files**

- Standard in(stdin)： **`fd = 0`**,
  standard out(stdout): **`fd = 1`**,
  standard error(stderr): **`fd = 2`**

Can also use fd to name opened **devices**

- Keyboard, display, etc.
- Allow a designer not to worry about input/output
  - Just read from **fd 0** and write to **fd 1**

Each process has its **own fd name space**

Why File Descriptor?

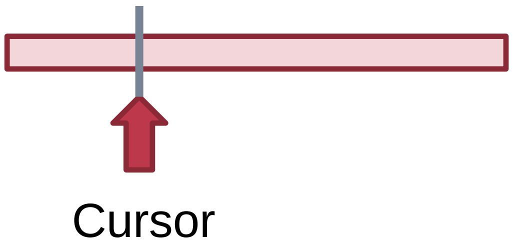
Other options

- Option-1: OS returns an inode pointer
- Option-2: OS returns all the block numbers of the file

Reasons and considerations

- Security: user can never access kernel's data structure
- Non-bypassability: all file operations are done by the kernel
  - Aka., *complete mediation*

File Cursor



File cursor

- Keep track of operation position within a file
- Can be changed by the **SEEK** operation

Case-1: Sharing file cursor

- Parent passes its fd to its child
    - In UNIX, a child process inherits all open fds from its parent
- Allow both parent and child to share one output file

Case-2: Not sharing file cursor

- Two processes open the same file

fd_table & file_table

One **file_table** for the whole system

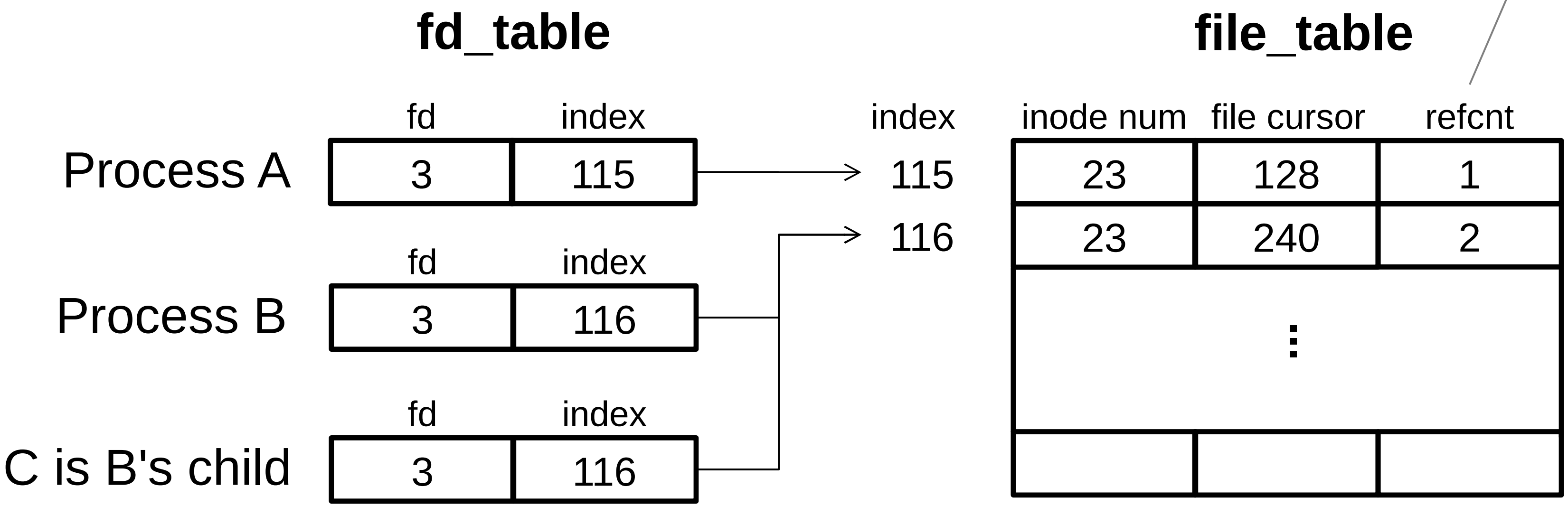
- Records information for opened files
- inode num, file cursor, ref_count of opening processes
- Children can share the cursor with their parent

One **fd_table** for each process

- Records mapping of **fd** to index of the **file_table**

File Cursor Sharing

Note: this refcnt is not the refcnt of inode!



fd_table

Process A

fd	index
3	115

Process B

fd	index
3	116

C is B's child

fd	index
3	116

file_table

index	inode num	file cursor	refcnt
115	23	128	1
116	23	240	2
	⋮		

Process A, B and C all open just one file with inode number 23

Process A and B open the same file, not share file cursor

Process B and C share the file cursor

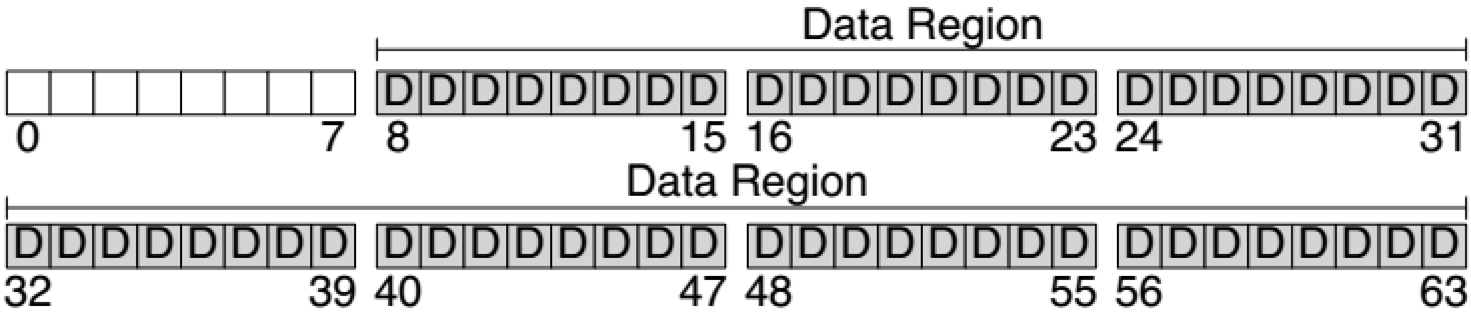
OPEN Implementation

```
procedure OPEN(string filename, integer flags, integer mod)-> integer
inode_number ← PATH_TO_INODE_NUMBER(filename, wd)
if inode_number = FAILURE and flags = O_CREATE then    // Create the file?
    inode_number ← CREATE(filename, mode)              // Yes, create it.
if inode_number = FAILURE then
    return FAILURE
inode ← INODE_NUMBER_TO_INODE(inode_number)
if PERMITTED(inode, flags) then                        // Does this user have the required permissions?
    file_index ← INSERT(file_table, inode_number)
    fd <- FIND_UNUSED_ENTRY(fd_table)                  // Find entry in file descriptor table
    fd_table[fd] ← file_index                          // Record file index for file descriptor
    return fd                                          // Return fd
else return FAILURE                                    // No, return a failure
```

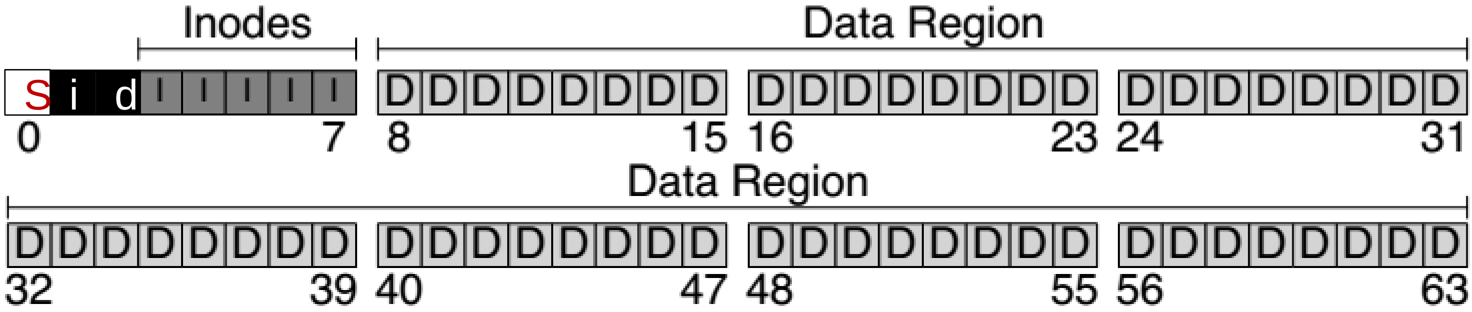
READ Implementation

```
procedure READ(integer fd, character[] &buf, integer n)-> integer
    file_index ← fd_table[fd]
    cursor ← file_table[file_index].cursor
    inode ← INODE_NUMBER_TO_INODE(file_table[file_index].inode_number)
    m ← MINIMUM(inode.size – cursor, n)
    atime of inode ← now( )
    if m = 0 then return END_OF_FILE
    for i from 0 to m – 1 do
        b ← INODE_NUMBER_TO_BLOCK(cursor + i, inode_number)
        COPY(b, buf, MINIMUM(m-i, BLOCKSIZE)
        i ← i + MINIMUM(m – i, BLOCKSIZE)
    file_table[file_index].cursor ← cursor + m
    return m
```

Disk Layout of a Simple File System


Data Region
D D D D D D D D D D D D D D D D D D D D D D D D
0 7 8 15 16 23 24 31
Data Region
D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D
32 39 40 47 48 55 56 63

At the Head of a Disk Partition



i: inode free block bitmap

d: data free block bitmap

S: super-block

- How many inodes: 80
- How many data blocks: 56
- Where the inode table begins: block 3
- …
- A magic number to identify the file system type

The super-block is used when the file system is mounted

Questions

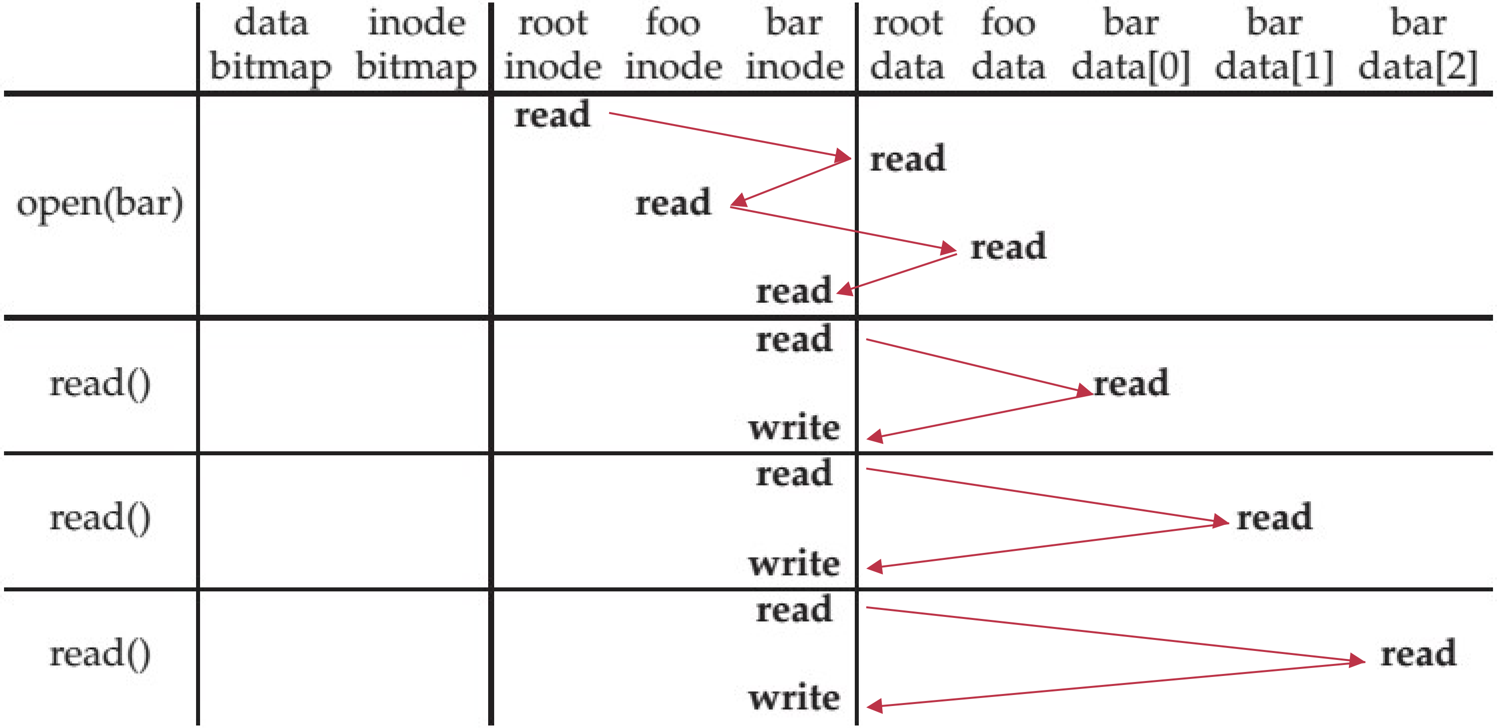
How many read and write in a `OPEN`?

How many read and write in a `READ`?

How many read and write in a `CREATION`?

File Open & Read Timeline

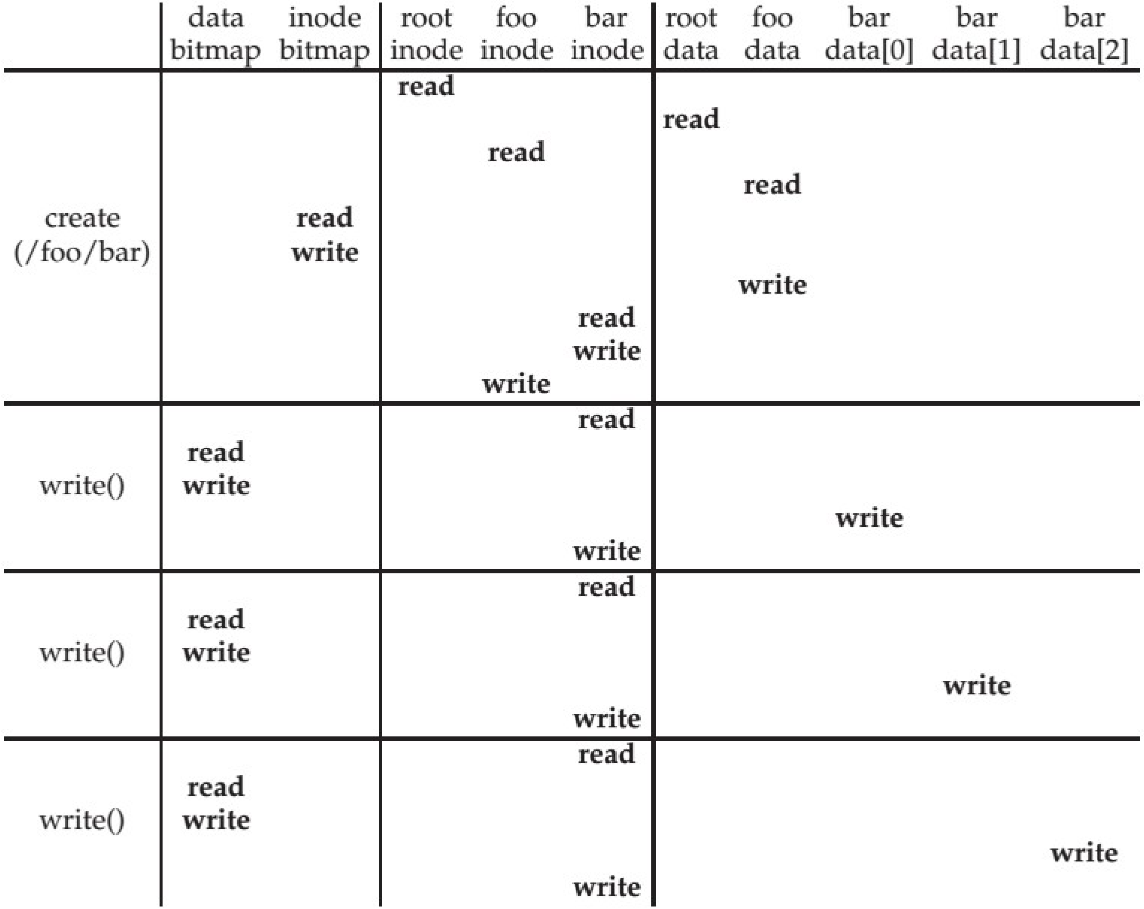
`open("/foo/bar", O_RDONLY)`



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Why "write" on bar inode in a read operation? Why no "write" on foo inodes?

File Creation Timeline



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					
write()					read					
	read									
	write									
									write	
					write					
write()					read					
	read									
	write									
										write
					write					

WRITE, APPEND & CLOSE

WRITE is **similar to** READ

- Allocate new block if necessary
- Update inode's size and mtime

APPEND

- Similar to write, directly write to the end of the file

CLOSE

- Free the entry in the fd_table
- Decrease the reference counter in file table
- Free the entry in file table if counter is 0

Failures in the middle may cause inconsistency!

Questions

When writing, which **order** is preferred?

- Update block bitmap, write new data, update inode (size and pointer)
- Update block bitmap, update inode (size and pointer), write new data
- Update inode (size and pointer), update block bitmap, write new data

SYNC

Block cache

- Cache of recently used disk blocks
- Read from disk if cache miss
- Delay the writes for batching
- Improve performance
- **Problem**: may cause **inconsistency** if fail before write

SYNC

- Ensure all changes to a file have been written to the storage device

Delete after OPEN but before CLOSE

One process has OPENed a file

Another process delete the file

- By removing the last name pointing to the file
- The reference counter is now 0

The inode is **not freed** until the first process calls CLOSE

- On Windows, it may forbid to delete an opened file

Review: Renaming

```
1 UNLINK(to_name)
2 LINK(from_name, to_name)   ❌
3 UNLINK(from_name)
```

Text edit usually save editing file in a **temp** file

- Edit in `.a.txt.swp`, then rename `.a.txt.swp` to `a.txt`

What if the computer **fails between 1 & 2**?

- The file `to_name` will be lost, which will surprise the user
- Need **atomic** action (in later lectures)

Review: Renaming

```
1 LINK(from_name, to_name)
2 UNLINK(from_name)
```

Weaker specification without atomic actions

- 1. Changes the inode # for `to_name` to the inode # of `from_name`
- 2. Removes the directory entry for `from_name`

If fails between 1 & 2

- Must increase reference count of `from_name`'s inode on recovery

If `to_name` already exists

- The `to_name` file will always exist, even if the machine fails between 1 & 2
- Need to revoke the resource on recovery

Rename between different directories

```
$ ls -il dir1
40978804 -rw-r--r--  1 xiayubin  wheel  6  9 26 08:50 from.txt

$ ls -il dir2
40978827 -rw-r--r--  1 xiayubin  wheel  7  9 26 08:51 to.txt

$ mv dir1/from.txt dir2/to.txt

$ ls dir1

$ ls -il dir2
40978804 -rw-r--r--  1 xiayubin  wheel  6  9 26 08:50 to.txt
```

inode number

Other file systems (not inode)

Other Choices instead of inode

Method-1: use continue blocks

- Re-allocate if the file expands
- E.g., data in memory
- Why not?

Method-2: use a linked list

- Each block links to its next block
- Use special one as EOF (End of File)
- E.g., FAT32
- Why not?

How to integrate different FS?

- vnode (will discuss later)
- Interface is similar with inode

FAT (File Allocation Table) File System

File is a collection of disk blocks

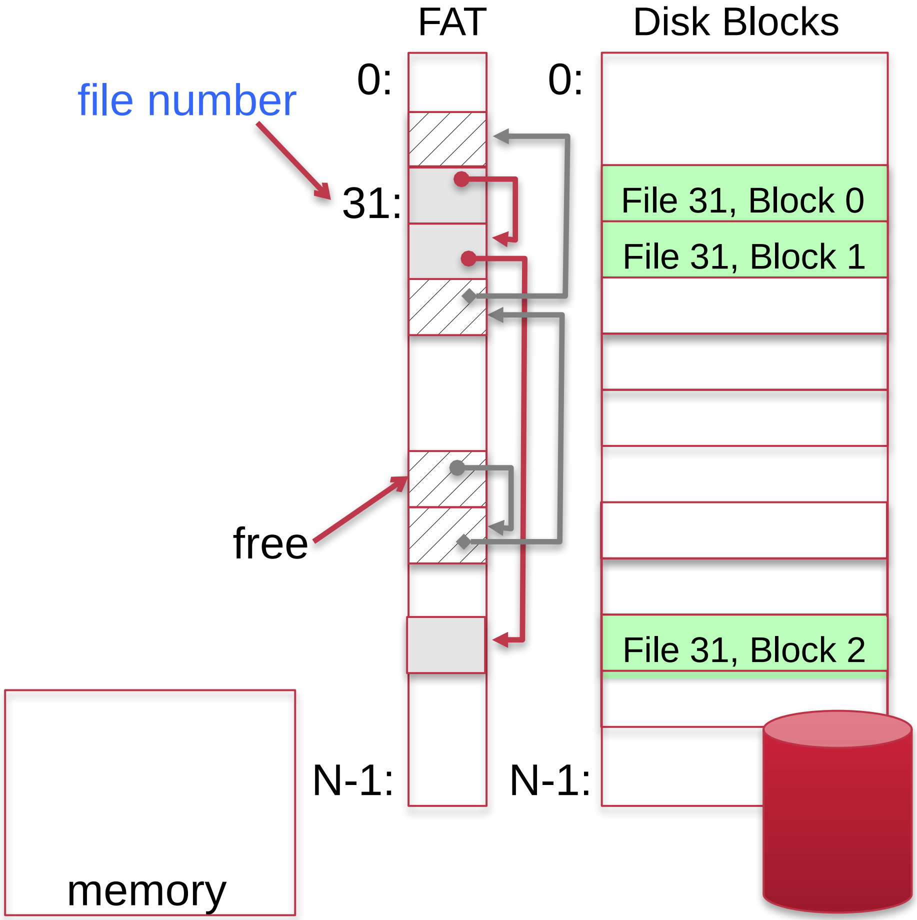
FAT is **a linked list** 1-1 with blocks

File *Number* is index of root
of block list for the file

File offset (o = < B, x >)

Follow list to get block #

Unused blocks 🔔 FAT free list



FAT Properties

File is a collection of disk blocks

FAT is **a linked list** 1-1 with blocks

File *Number* is index of root of block list for the file

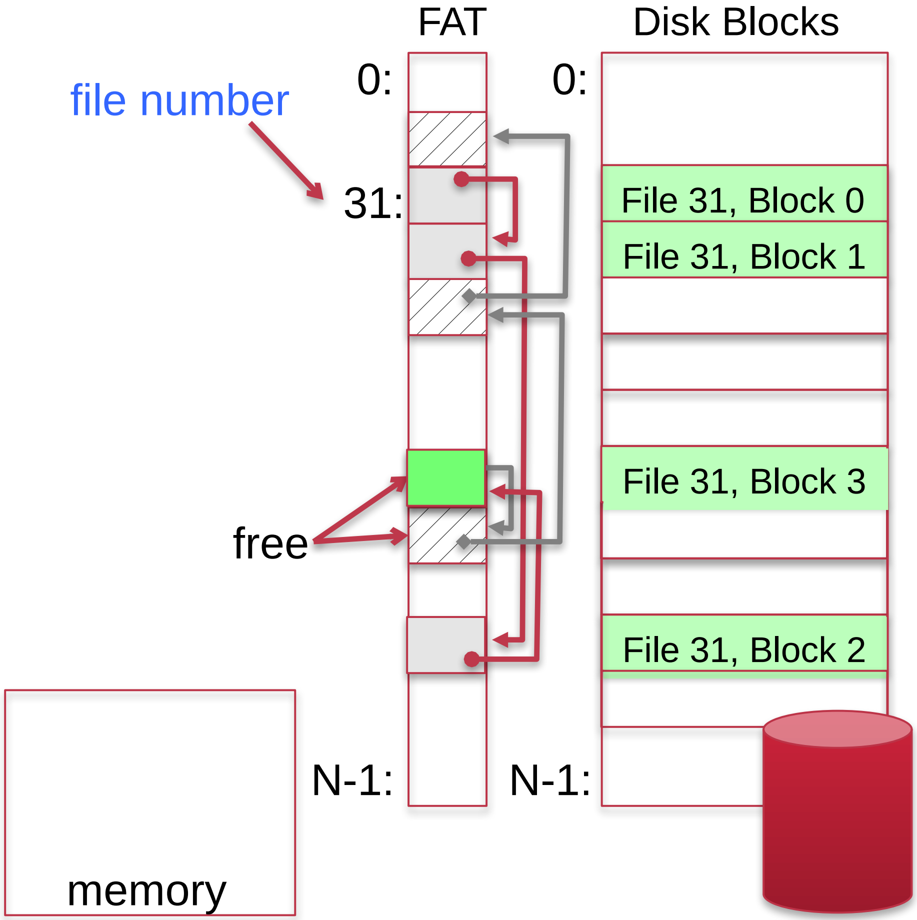
File offset (o = < B, x >)

Follow list to get block #

Unused blocks 🔔 FAT free list

Ex: file_write(31, < 3, y >)

- Grab blocks from free list
- Linking them into file



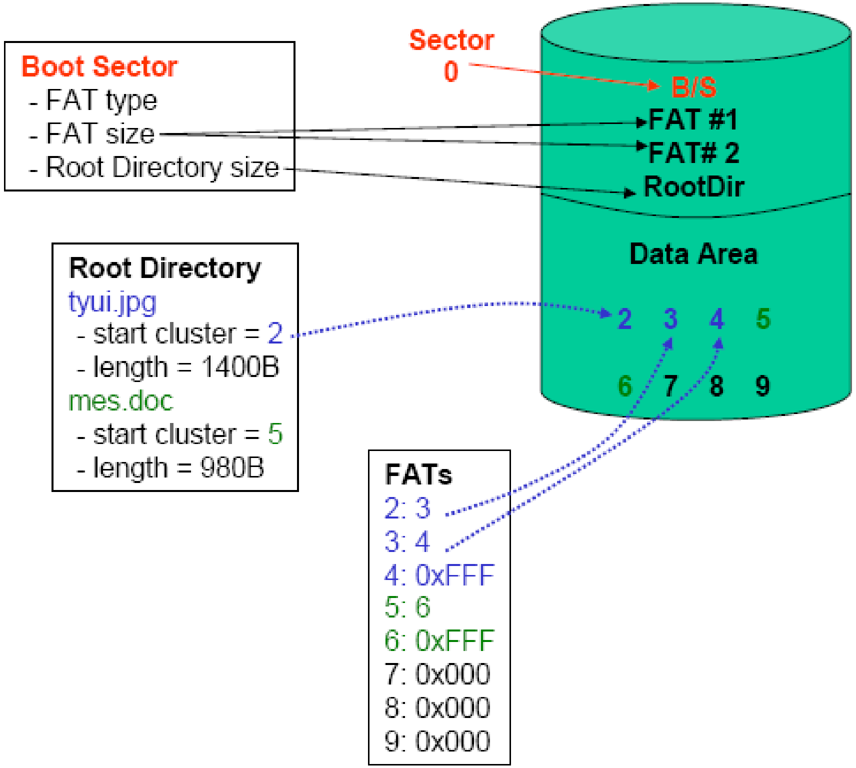
FAT File System

File allocation table (FAT)

- Organize files as linked lists

No inode

- File metadata: name & size
- Metadata are saved in dirs

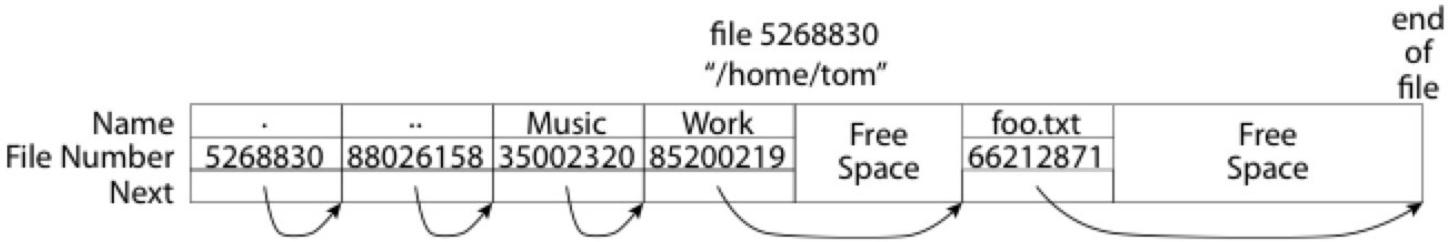


Boot Sector
- FAT type
- FAT size
- Root Directory size

Root Directory
tyui.jpg
- start cluster = 2
- length = 1400B
mes.doc
- start cluster = 5
- length = 980B

FATs
2: 3
3: 4
4: 0xFFF
5: 6
6: 0xFFF
7: 0x000
8: 0x000
9: 0x000

What about the Directory in FAT?



Name	.	..	Music	Work	Free Space	foo.txt	Free Space
File Number	5268830	88026158	35002320	85200219		66212871	
Next							

Directory: essentially a file containing <file_name: file_number> mappings

- Free space for new entries
- File attributes (metadata) are kept in directory
- Each directory is a linked list of entries

Question: Where to find root directory ("/")?

- Root dir at sector 0

Question: inode vs. FAT

What are the differences between inode and FAT?

Support hard link? Soft link?