

# File System API and Disk I/O

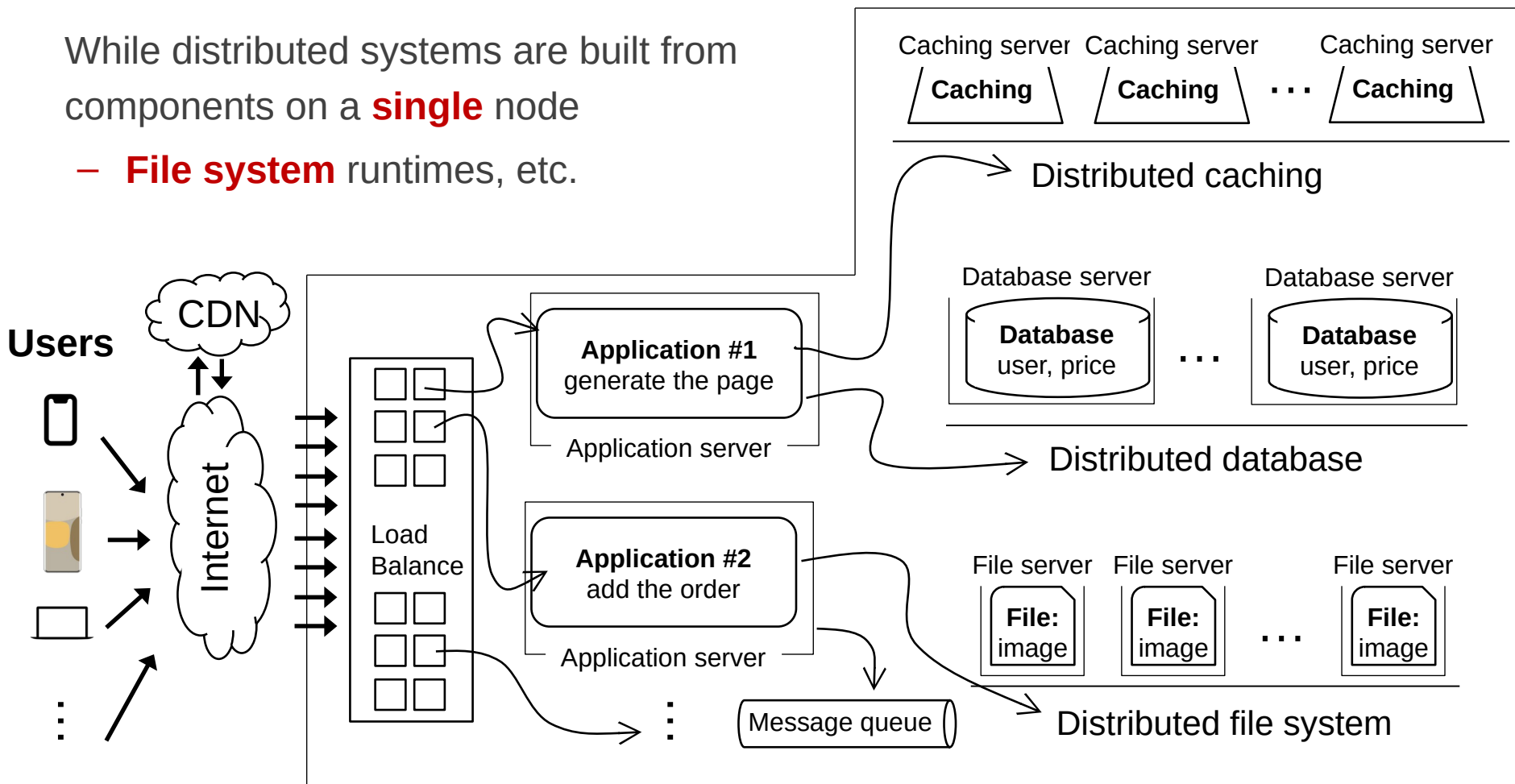
IPADS, Shanghai Jiao Tong University

<https://www.sjtu.edu.cn>

# Review: Large-scale websites on distributed systems

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

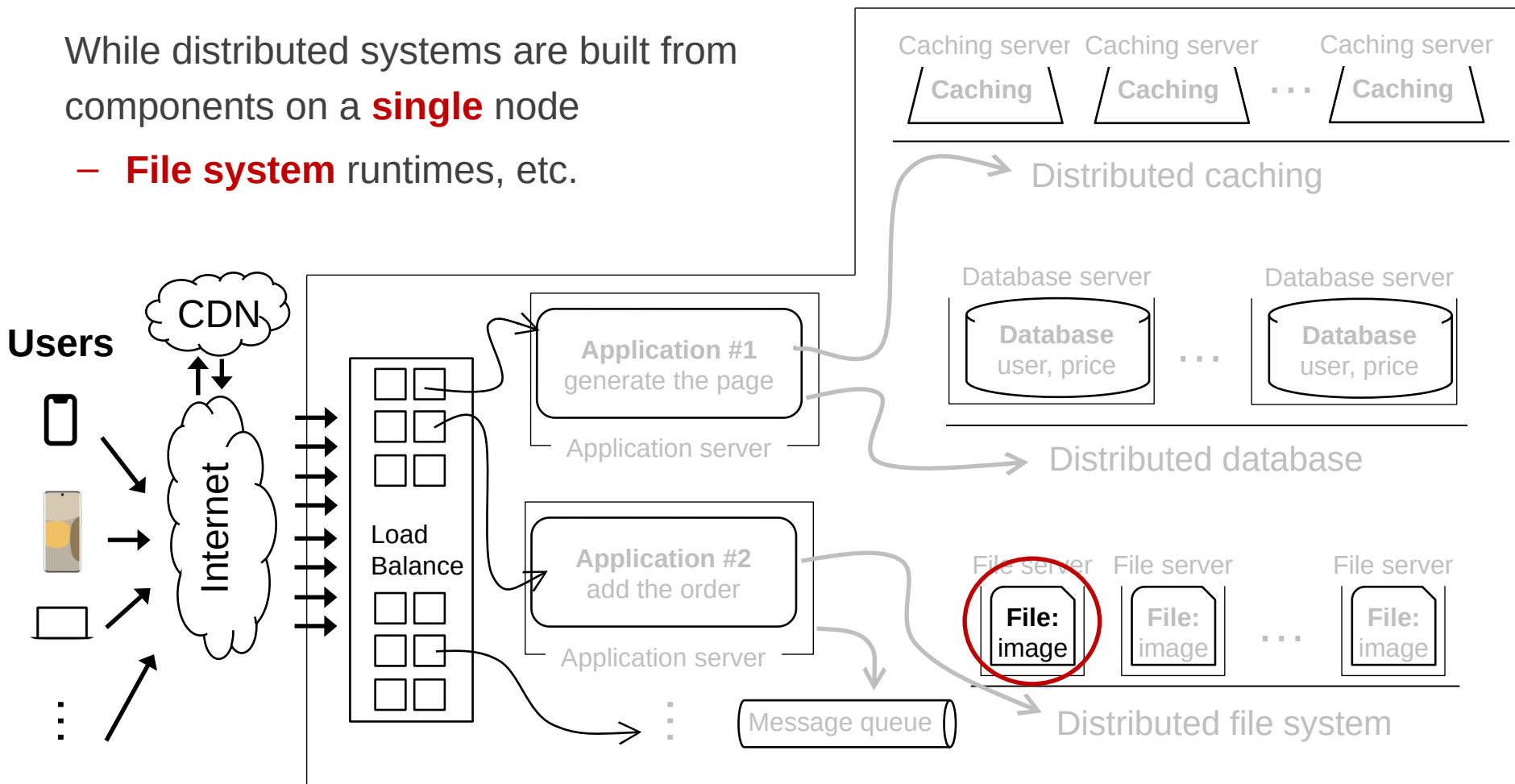
- **File system** runtimes, etc.



# Review: Large-scale websites on distributed systems

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

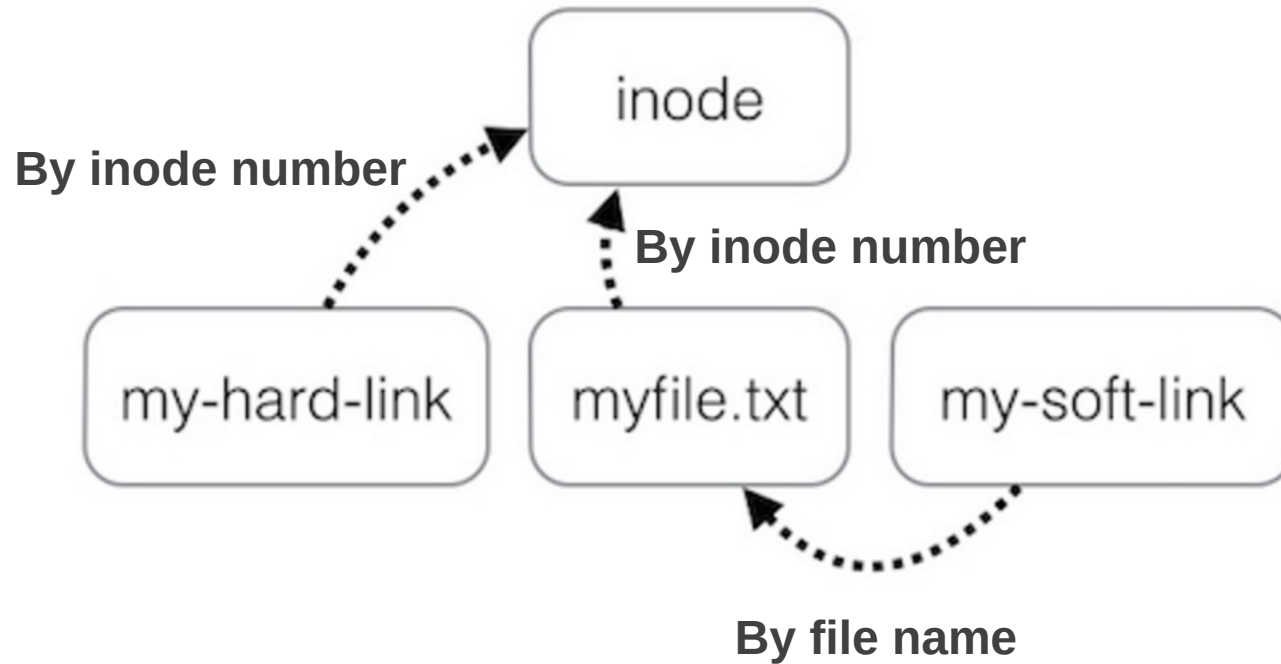
- **File system** runtimes, etc.



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

Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	↑
Absolute path name layer	Provide a root for the naming hierarchies.	user-oriented names
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.	↑
File layer	Organize blocks into files.	machine-oriented names
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

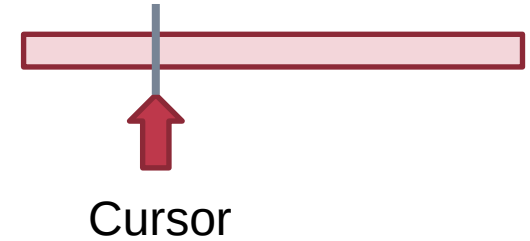
- Keep track of operation position within a file
- Can be changed by the **SEEK** operation

## 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

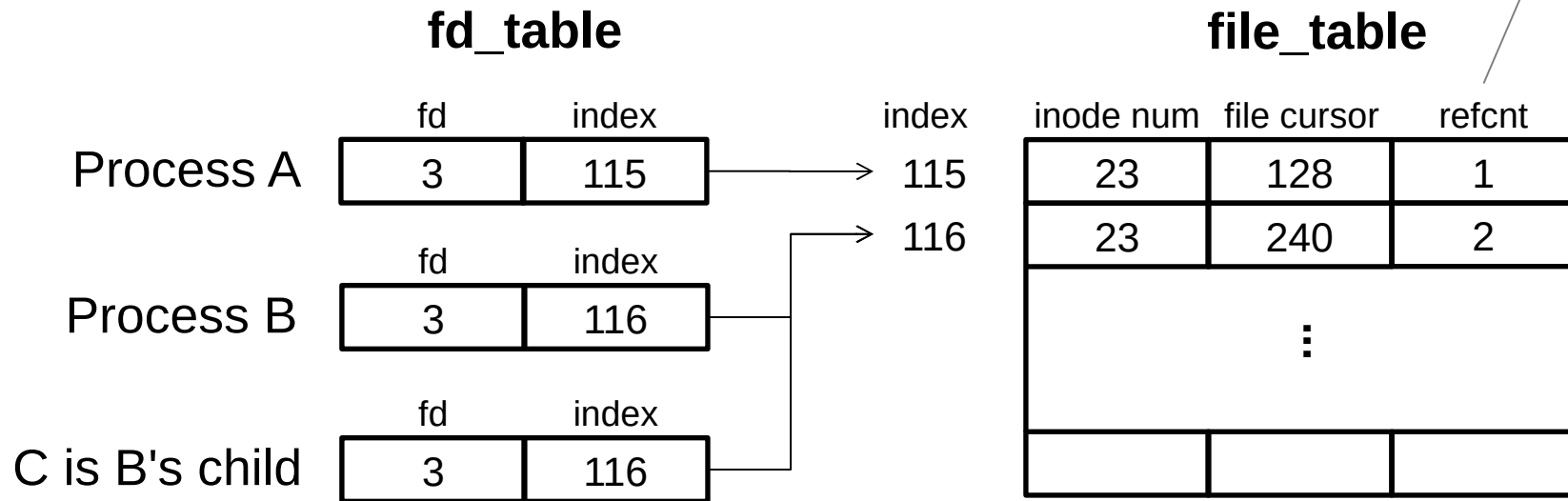
- Records information for opened files
- inode num, file cursor, ref\_count of opening processes
- Children can share the cursor with their parent

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!*



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**

0 *inode\_number* ← PATH\_TO\_INODE\_NUMBER(filename, wd)

1. **if** *inode\_number* = FAILURE **and** *flags* = O\_CREATE **then** // Create the file?

2.     *inode\_number* ← CREATE(filename, mode) // Yes, create it.

3. **if** *inode\_number* = FAILURE **then**

4.     **return** FAILURE

5. *inode* ← INODE\_NUMBER\_TO\_INODE(*inode\_number*)

6. **if** PERMITTED(*inode*, *flags*) **then**

// Does this user have the required permissions?

7.     *file\_index* ← INSERT(*file\_table*, *inode\_number*)

8.     *fd* ← FIND\_UNUSED\_ENTRY(*fd\_table*)

// Find entry in file descriptor table

9.     *fd\_table*[*fd*] ← *file\_index*

// Record file index for file descriptor

10.    **return** *fd*

// Return fd

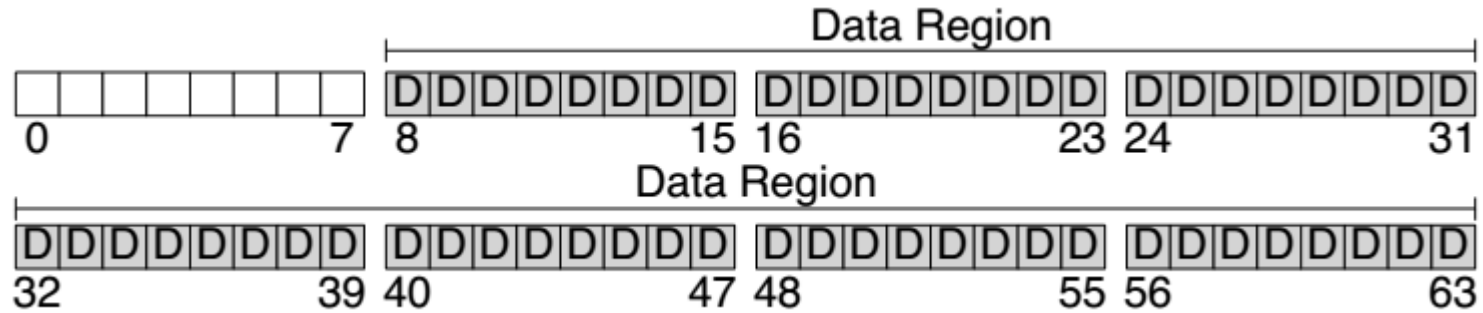
11. **else return** FAILURE

// No, return a failure

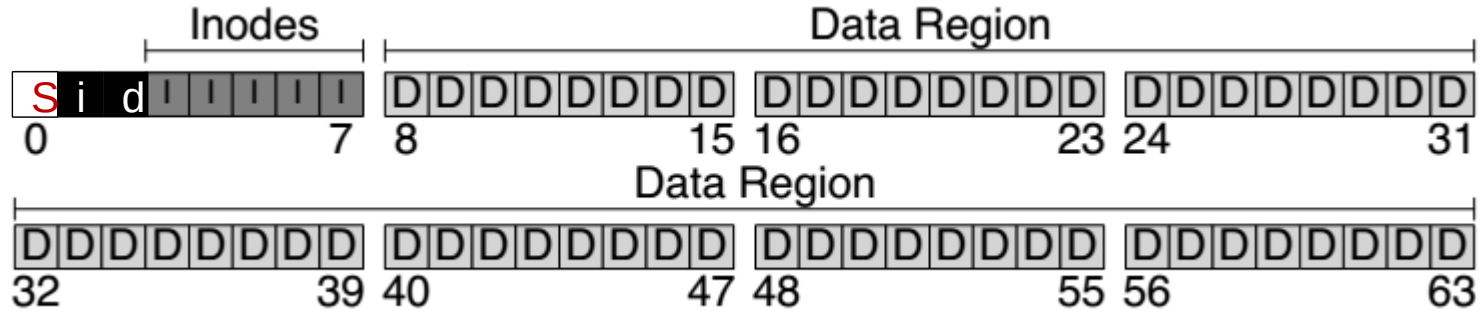
## READ Implementation

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

## Disk Layout of a Simple File System



# 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

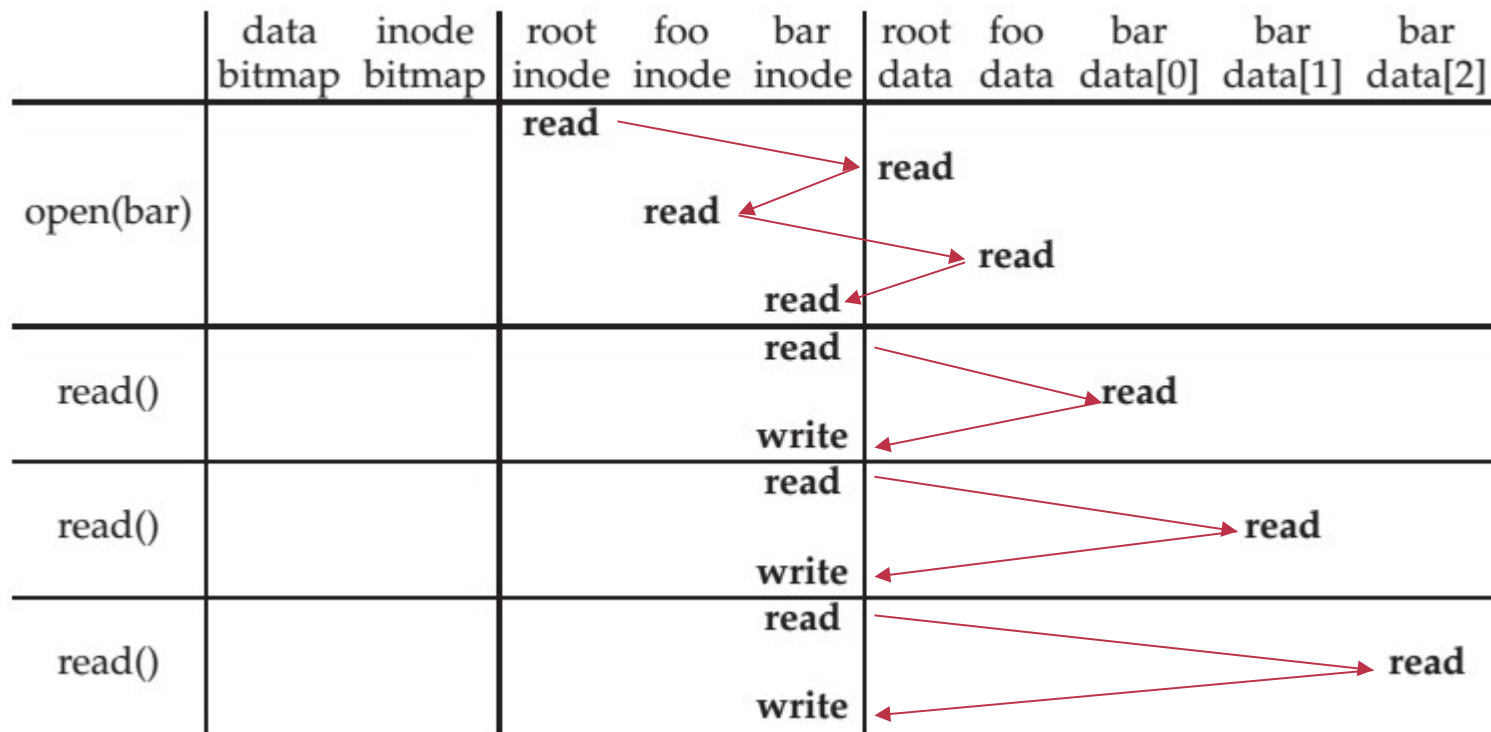
How many read and write in a OPEN?

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)`



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 write	read	read		read	read			
					read write		write			
				write						
write()	read write				read					
					write		write			
write()	read write				read					
					write			write		
write()	read write				read					
					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

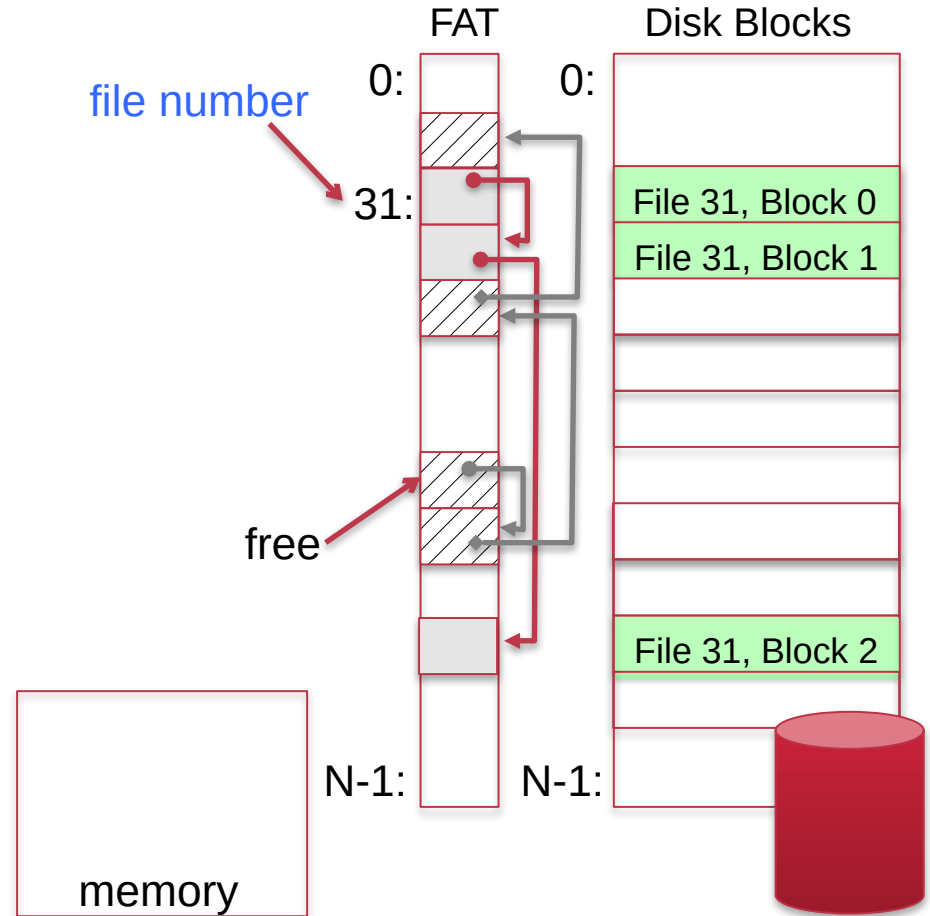
FAT is **a linked list** 1-1 with blocks

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

File offset ( $o = \langle B, x \rangle$ )

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

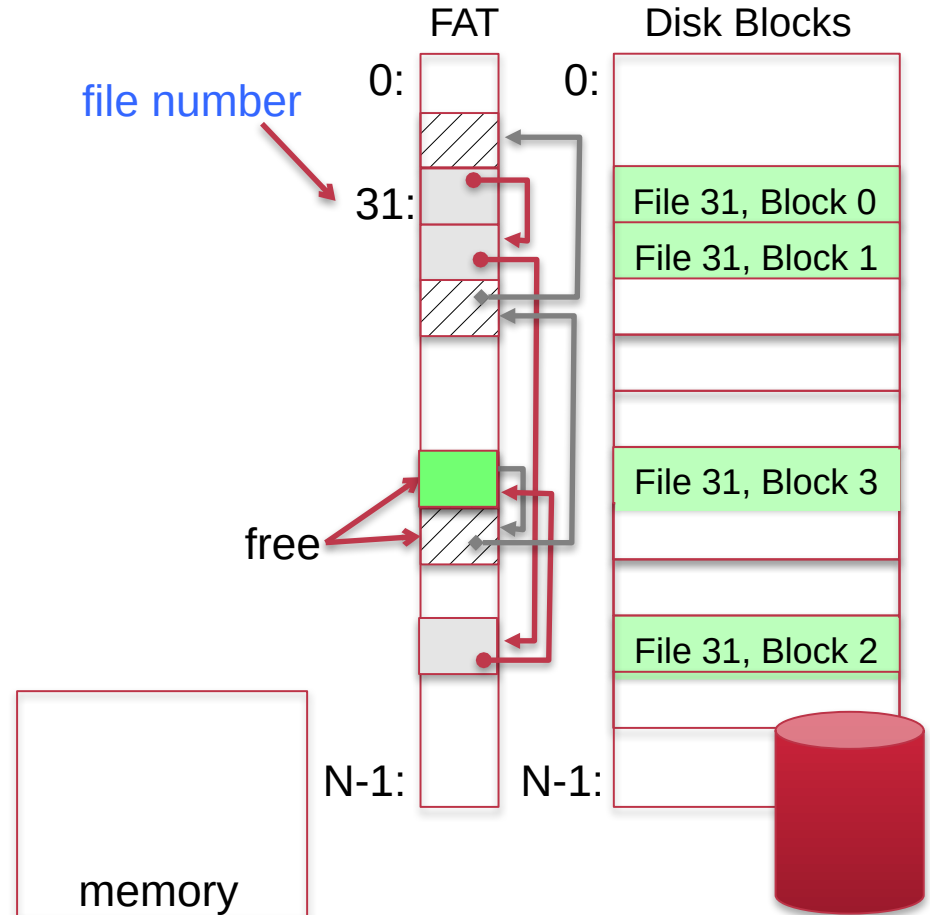
File offset ( $o = \langle B, x \rangle$ )

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



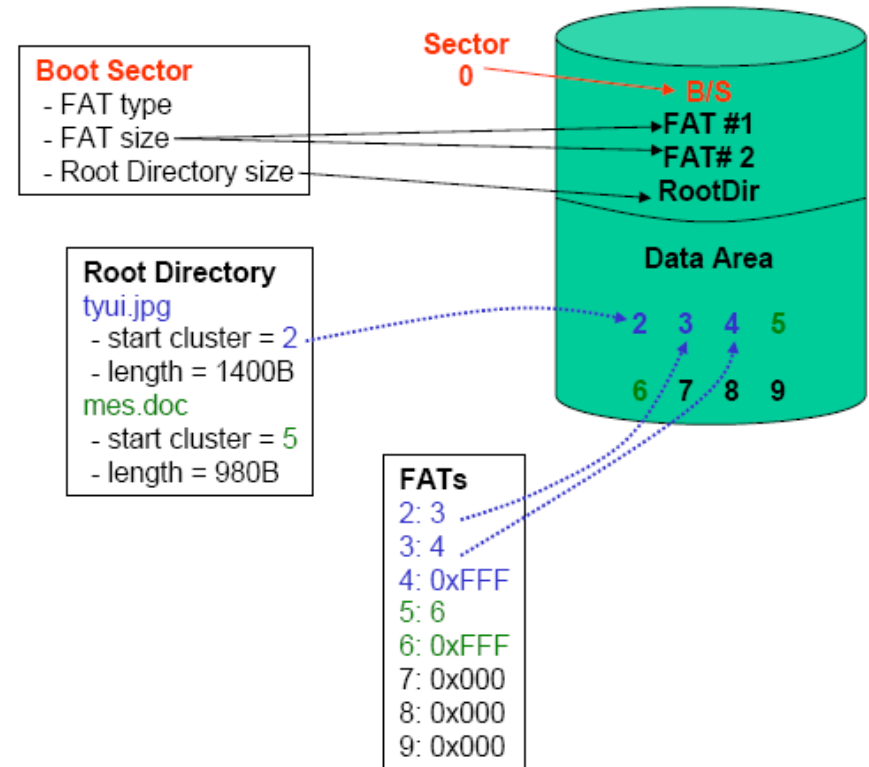
# FAT File System

## File allocation table (FAT)

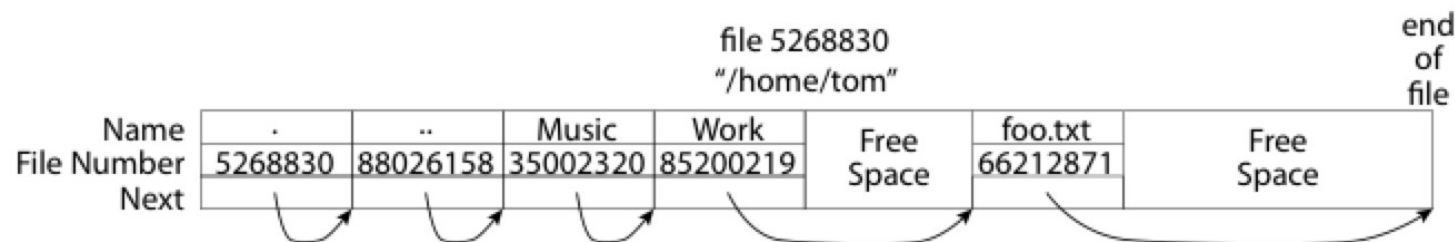
- Organize files as linked lists

## No inode

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



## What about the Directory in FAT?



**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?