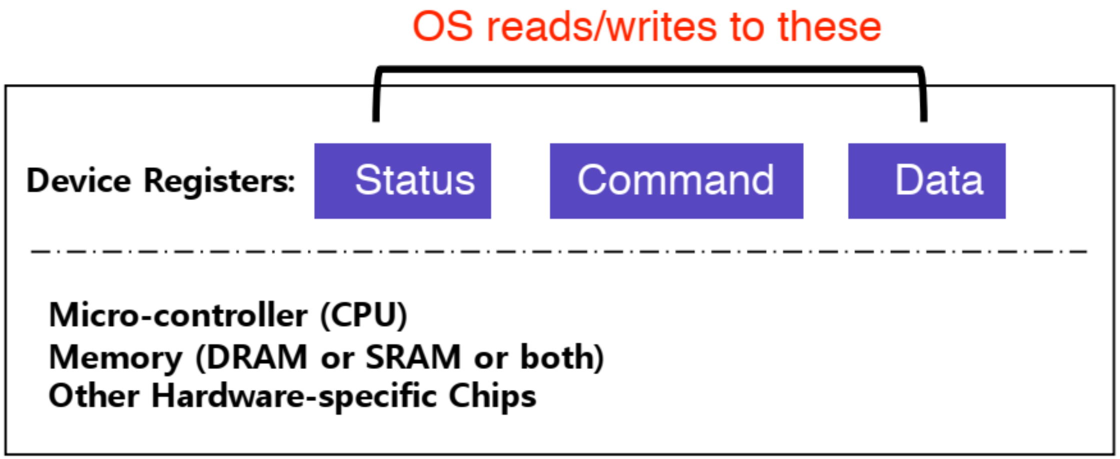
File Systems (part 1)

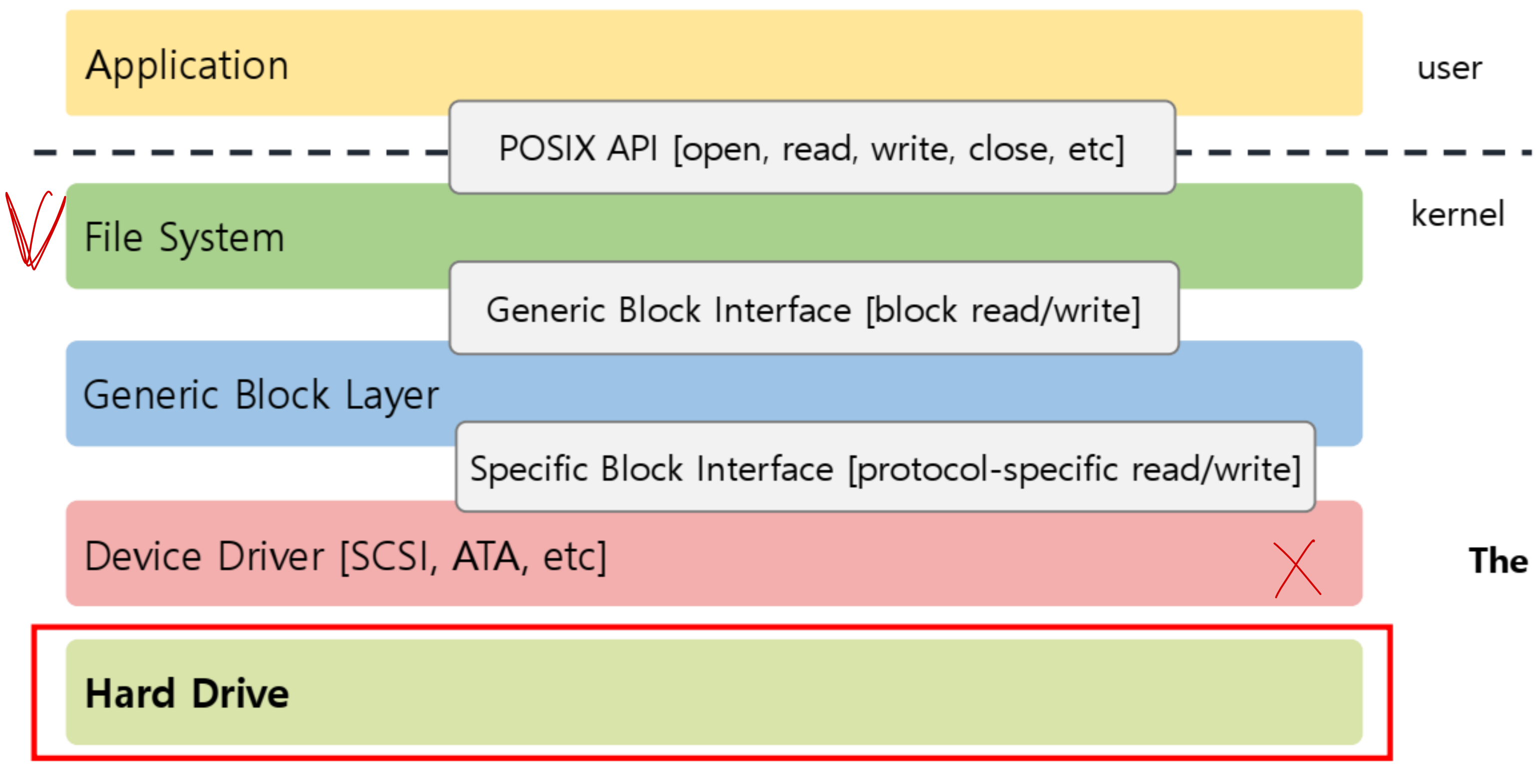
Instructor: Youngjin Kwon

Abstract model of device



- Status checks: how? → Polling vs Interrupt
- Data transfer: programmed I/O vs DMA
- Control: Special instruction vs Memory mapped IO → Common

File System Abstraction



The File System Stack

File System fun

- File systems: traditionally hardest part of OS
  - More papers on FSes than any other single topic

- Main tasks of file system:
  - Don't go away (ever) → Persistent
  - Associate bytes with name (files)
  - Associate names with each other (directories)
  - Can implement file systems on disk, over network, in remote memory, in non-volatile ram (NVRAM), on tape
  - We'll focus on disk and generalize later

(ex. rollback)

Disk vs Memory

technique? -> "journaling"

crash

one sector at a time...

1 -> 2 -> 3 -> 4

doesn't guaranteed.

Write in a single operation

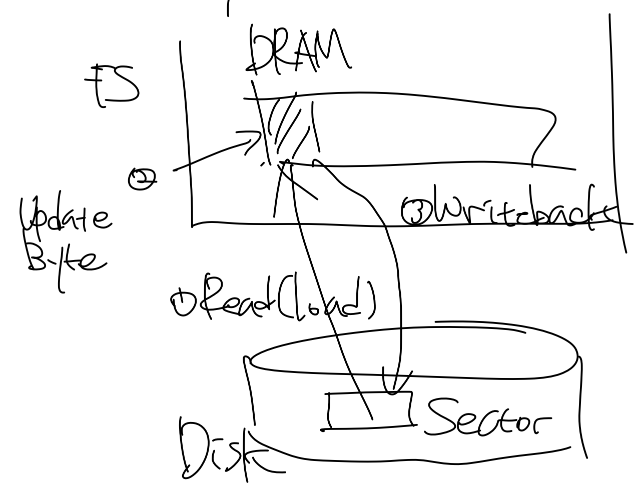
	Disk	MLC NAND Flash	DRAM
Smallest write	sector	page	byte
Atomic write	sector	page	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	100 MB/s	550–2500 MB/s	$>$ 10 GB/s
Sequential write	100 MB/s	520–1500 MB/s*	$>$ 10 GB/s
Cost	\$0.03/GB	\$0.32/GB	\$10/GiB
Persistence	Non-volatile	Non-volatile	Volatile

Disk review

"fsynch API"

- How to write a single byte? "Read-Modify-Write"

if cached → don't need to read in.



- Sector: unit of atomicity
  - Sector write done completely, even if crash in middle
  - What if writing multiple sectors?

Files: named bytes on disk

Blocks vs Sectors(?)

- File abstraction:
  - User's view: named sequence of byte
  - FS's view: collection of disk blocks
  - File system's job: map name & offset to disk blocks
    - File, offset → FS → logical block address

(name) indexing

- File operations (aka APIs based on use cases):
  - Create, delete a file
  - Read from a file, write to the file

File system design goal

- Wanted:
  - Performance: Operations to have as few disk accesses as possible
  - Space efficiency: Have minimal space overhead (group related things)
  - Consider underlying storage device characteristics

Per process

- Analogy to memory system
  - Page table: map virtual page # to physical page #
  - File **metadata**: map file byte offset to disk block address
  - Directory: map name to disk address or file #

inode

File System Workload

- File sizes
  - Are most files small or large? small
  - Which accounts for more total storage: small or large files? large

File System Workload

- File sizes
  - Are most files small or large?
    - SMALL
  - Which accounts for more total storage: small or large files?
    - LARGE

File System Workload

- File access
  - Are most accesses to small or large files? small
  - Which accounts for more total I/O bytes: small or large files?

large

How to allocate large files?

: Contiguous blocks.

File System Workload

- File access
  - Are most accesses to small or large files?
    - SMALL
  - Which accounts for more total I/O bytes: small or large files?
    - LARGE

Working intuitions

- How are files used?
  - Most files are read/written sequentially
  - Some files are read/written randomly
    - Ex: database files, swap files
  - Some files have a pre-defined size at creation
  - Some files start small and grow over time
    - Ex: system logs

Prethought of File System Design

- For small files:

- For large files:

- May not know at file creation
  - Whether file will become small or large
  - Whether file is persistent or temporary
  - Whether file will be used sequentially or randomly

each file has to associate metadata

Pre-thought of File System Design

1B file → "absurd"

~~4KB~~ metadata

make metadata small.

typically (128 - 256 B)

- For small files:
  - Small blocks for storage efficiency
  - Files used together should be stored together
  - Metadata size matters
    - IO for metadata vs IO for data
- For large files:

- May not know at file creation
  - Whether file will become small or large
  - Whether file is persistent or temporary
  - Whether file will be used sequentially or randomly

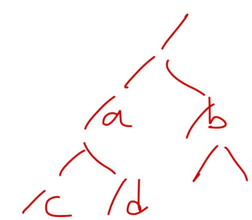
Pre-thought of File System Design

- For small files:
  - Small blocks for storage efficiency
  - Files used together should be stored together
  - Metadata size matters
    - IO for metadata vs IO for data
- For large files:
  - Large blocks for storage efficiency
  - Contiguous allocation for sequential access
  - Efficient lookup for random accesses
- May not know at file creation
  - Whether file will become small or large
  - Whether file is persistent or temporary
  - Whether file will be used sequentially or randomly

File System Components

- Directory

  name → metadata
  - Group of named files or subdirectories
  - Mapping from file name to file metadata location
- Path
  - String that uniquely identifies file or directory
  - Ex: /cse/www/education/courses/cs330
- Links
  - Hard link: link from name to metadata location
  - Soft link: link from name to alternate name



- Mount
  - Mapping from name in one file system to root of another (FS)

UNIX File System API

- create, link, unlink, createdir, rmdir
  - Create file, link to file, remove link
  - Create directory, remove directory
- open, close, read, write, seek
  - Open/close a file for reading/writing
  - Seek resets current position
- **fsync**
  - File modifications can be cached
  - fsync forces modifications to disk (like a memory barrier)

Main Points

- File layout
- Directory layout
- How to locate file and directory blocks?

File System Design

- Data structures
  - Directories: file name -> file metadata
  - File metadata: how to find file data blocks
    - **Indexing structure**
  - Free map: list of free disk blocks

    ↳ bitmap
- How do we organize these data structures?
  - Device has non-uniform performance
    - Random vs Sequential, Append vs Update

Design Challenges

- Index structure (cover in filesys_part1)
  - How do we locate the blocks of a file?
- Index granularity
  - What block size do we use?
- Free space (cover in filesys_part2)
  - How do we find unused blocks on disk?
- Locality (cover in filesys_part2)
  - How do we preserve spatial locality?
- Reliability (cover in crash consistency)
  - What if machine crashes in middle of a file system op?

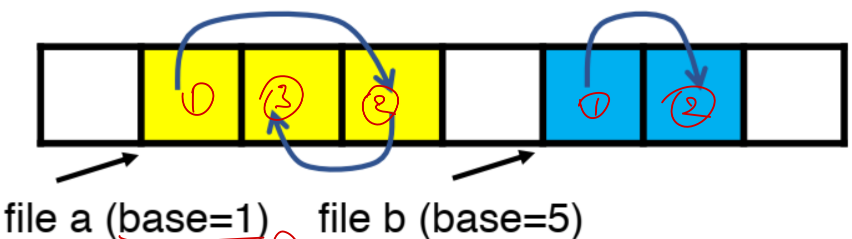
Implementation

How to design indexing structure?

Linked files

- Linked list index structure
  - Simple, easy to implement   ex) FAT32
  - Still widely used (e.g., thumb drives)
  - File metadata (called **Inode**) points file's first block

How do you find last block in a?



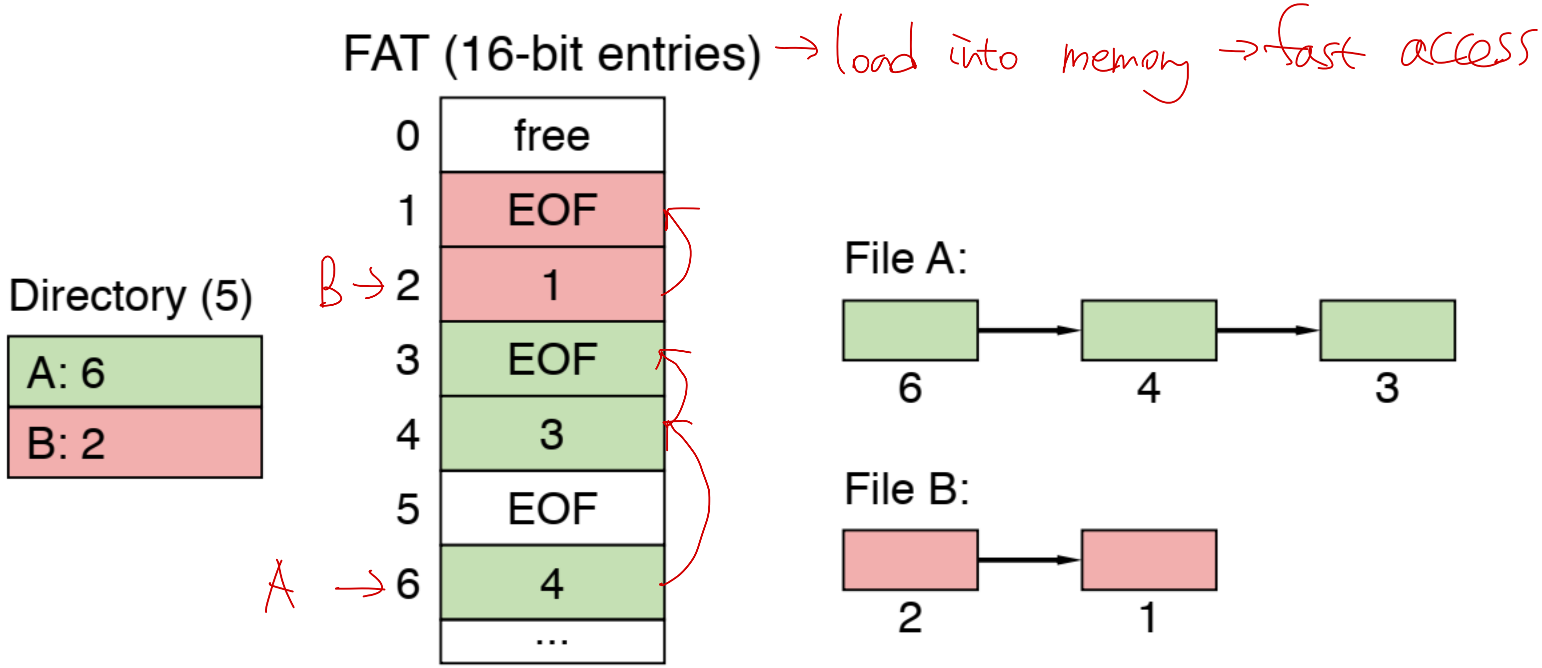
first block

- File table:
  - Linear map of all blocks on disk
  - Each file a linked list of blocks

Microsoft FAT (simplified)

Problem: Slow when random access.

- Linked files with key optimization: *puts links in fixed-size "file allocation table" (FAT) rather than in the blocks.*



Still do pointer chasing, but can cache entire FAT in DRAM so can be cheap compared to disk access

FAT

- Pros: Simple Design , Good at Sequential access

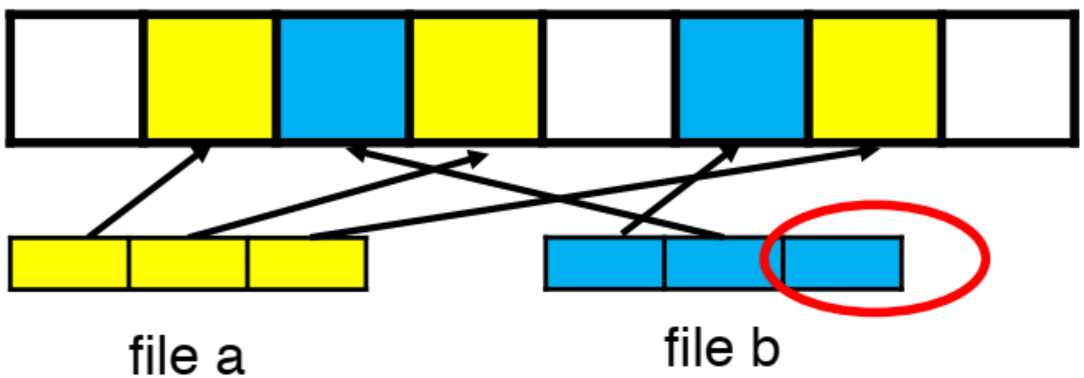
  Cheap & fast Comparing Disk access

  Direct access possible , reliable

- Cons: Bad at random access

Indexed files

- **Each file metadata has an array holding all of its block pointers**
  - Just like a page table, so will have similar issues
  - Max file size fixed by array's size
  - Allocate array to hold file's block pointers on file creation
  - Allocate actual blocks on demand using free list



Berkeley UNIX FFS (Fast File System)

first find inode by looking at...

- inode table

  유사한

  inode

  – Analogous to FAT table
- inode
  – Metadata
    - File owner, access permissions, access times, …
  – Set of 12 data pointers
  – With 4KB blocks => max size of files? 48KB → too small
  – How to overcome?
    - Add more data pointers to inode? → takes too much space

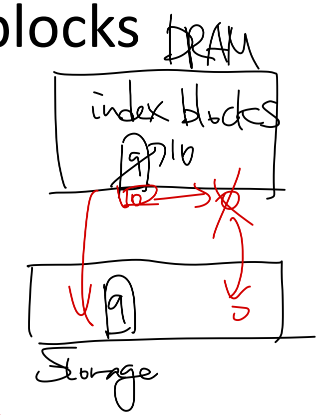
→ Solution: Multi-level

limit: have to read multiple blocks → Sol: Cache into DRAM

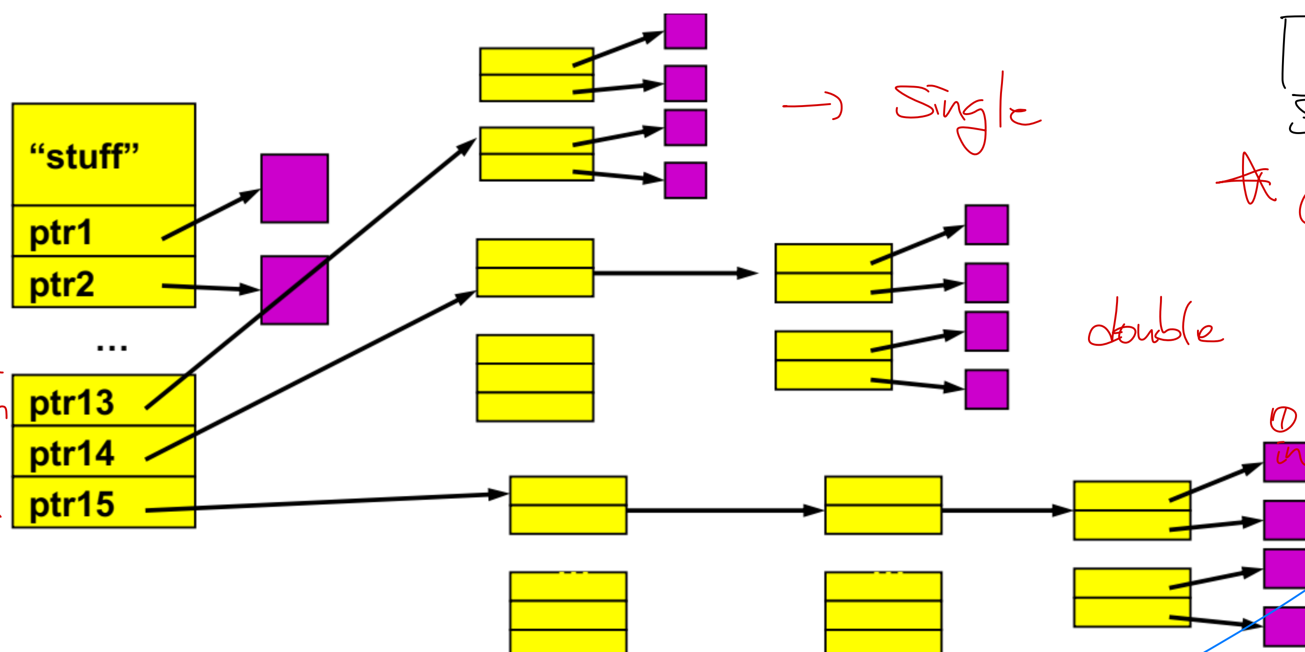
Multi-level Indexed files

but still

- **inode = 15 block pointers + "stuff"**
  - first 12 are direct blocks: solve problem of first blocks access slow
  - then single, double, and triple indirect block



☆ Consistency problem



12 data blocks

pointers

→ Single

double

triple

① index → ② data

actually not in storage

③ read → garbage

Solution: Data store first, index later.

FFS inode

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4KB blocks => max size of 48KB files
- Indirect block pointer
  - pointer to disk block of data pointers (Let’s say 4B)
- Indirect block: 1K data blocks => 4MB (+48KB)

1K x 4KB = 4MB

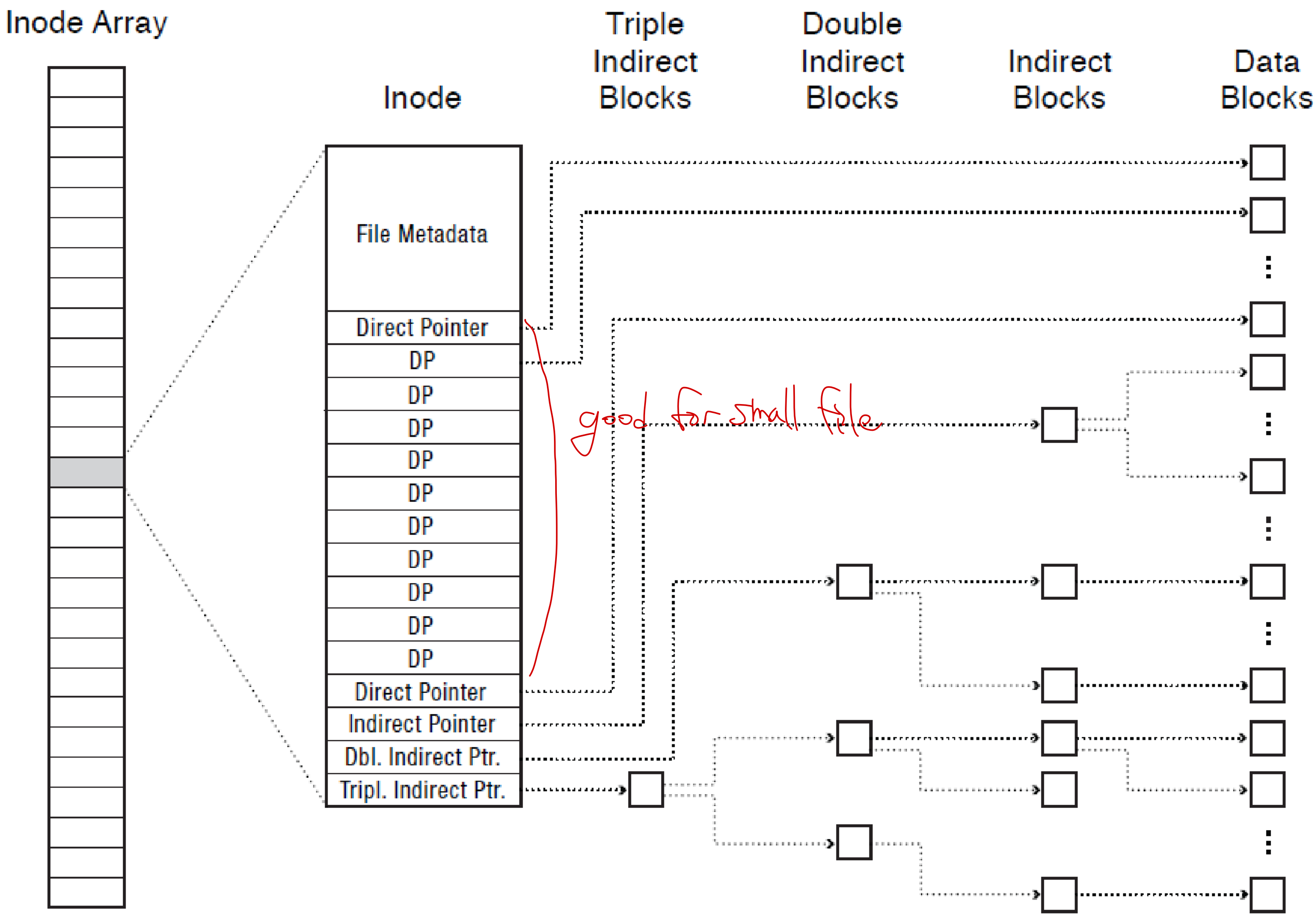
FFS inode

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4KB blocks => max size of 48KB
- Indirect block pointer
  - pointer to disk block of data pointers (Let’s say 4B)
  - 4KB block size => 1K data blocks => 4MB
- Doubly indirect block pointer
  - Doubly indirect block => 1024 * indirect block
  - 4GB (+ 4MB + 48KB)

FFS inode

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4KB blocks => max size of 48KB
- Indirect block pointer
  - pointer to disk block of data pointers
  - 4KB block size => 1K data blocks => 4MB
- Doubly indirect block pointer
  - Doubly indirect block => 1K indirect blocks
  - 4GB (+ 4MB + 48KB)
- Triply indirect block pointer
  - Triply indirect block => 1K doubly indirect blocks
  - 4TB (+ 4GB + 4MB + 48KB)

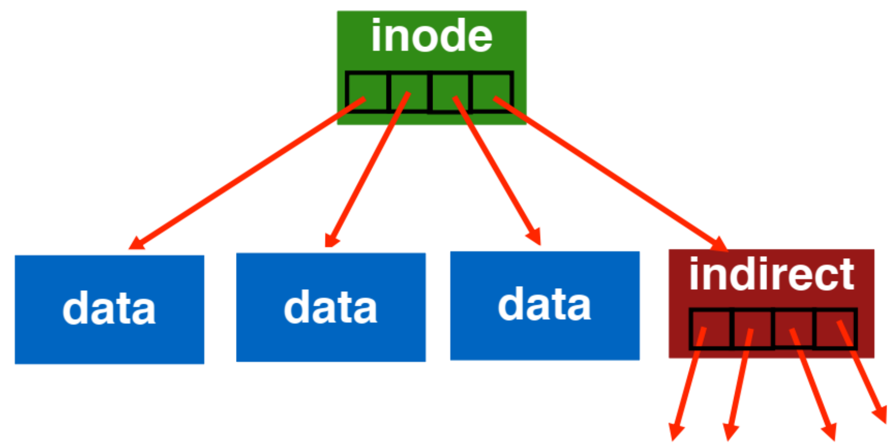
Berkeley UNIX FFS (Fast File System)

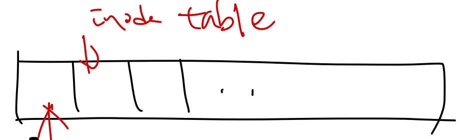


More about Inode

type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

inode





More about Inode

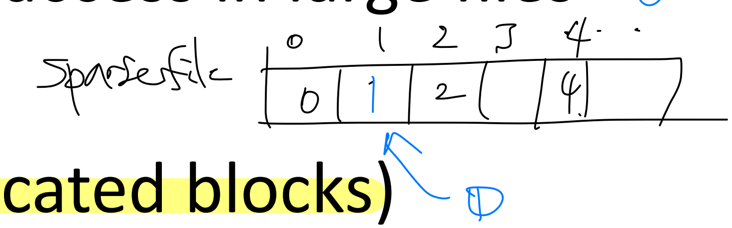
Super block

- **inodes are stored in a fixed-size array** → Inode table?
  - Size of array fixed when disk is initialized; can't be changed
  - Lives in known location, originally at one side of disk:

  - The *index* of an inode in the inode array called an i-number
  - Internally, the OS refers to files by *i-number*
  - When file is opened, inode brought in memory
  - Written back when modified and file closed or time elapses

FFS Asymmetric Tree

- Small files: "shallow tree"
  – Efficient storage for small files
- Large files: "deep tree"
  – Efficient lookup for random access in large files
- **Sparse files**:
  – Files that have holes (unallocated blocks)
  – Blocks are allocated when used
  – How to represent sparse files?
    - only fill index pointers if needed (fill on demand)



Sparse file representation in index scheme: NULL

Named Data Translations in File Systems

$f(x)$

file name offset $\xrightarrow{\text{directory}}$:returns → *file number offset* $\xrightarrow{\text{index structure}}$ *storage block*

Directory is a block

Directories Are Files

inode

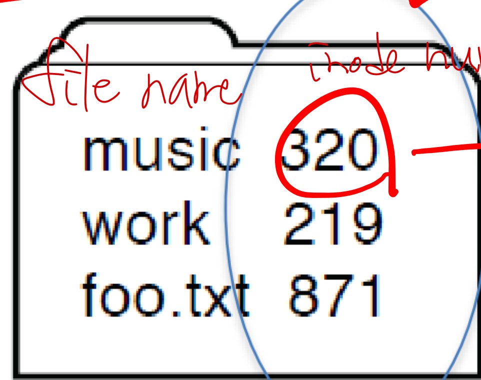
for file

for directory

4KB

file name

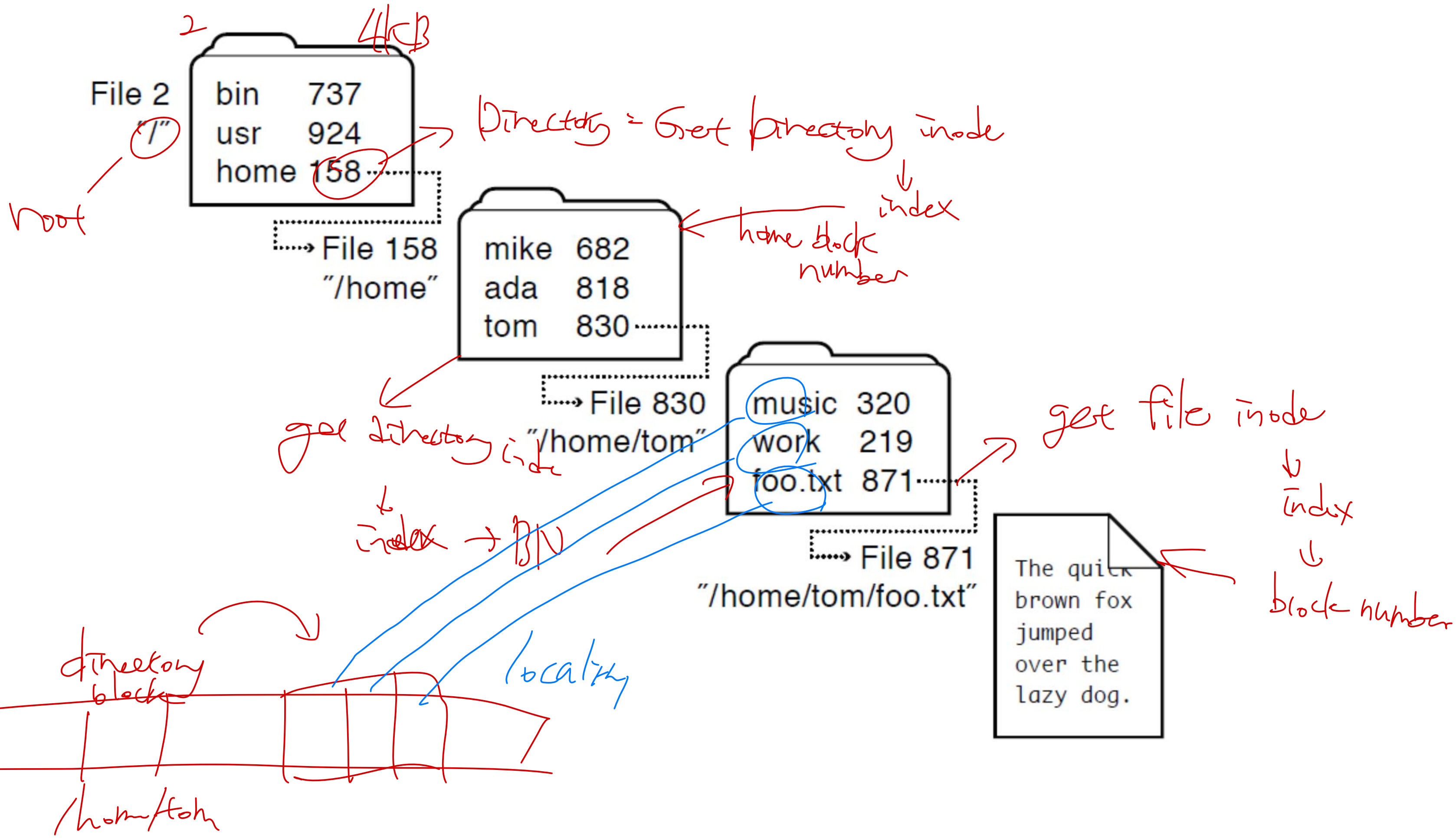
inode number



file's inode number

block

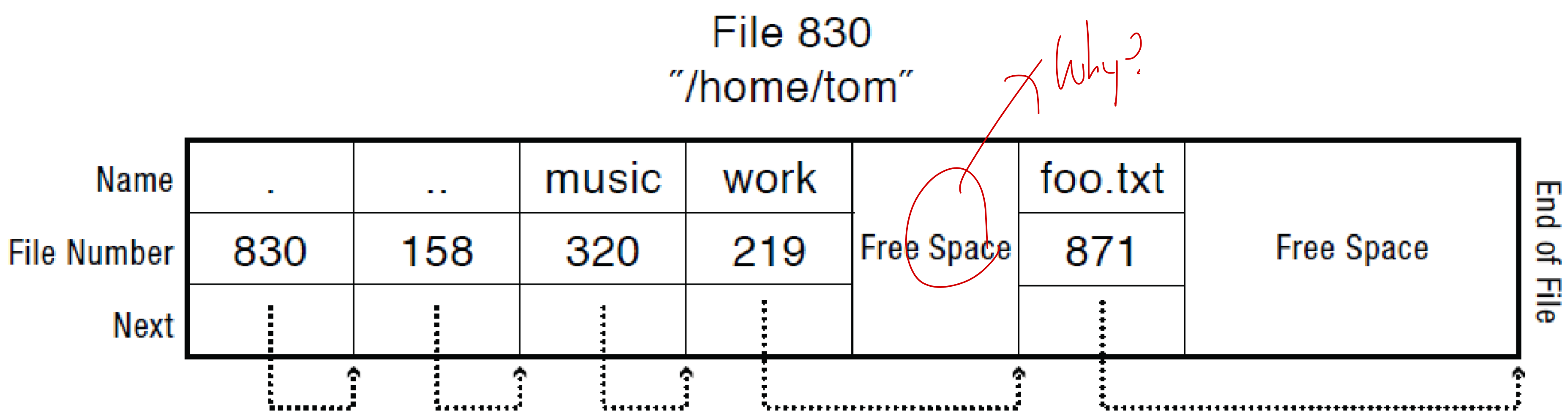
Recursive Filename Lookup



Directory Layout

Directory stored as a file

Linear search to find filename (small directories)

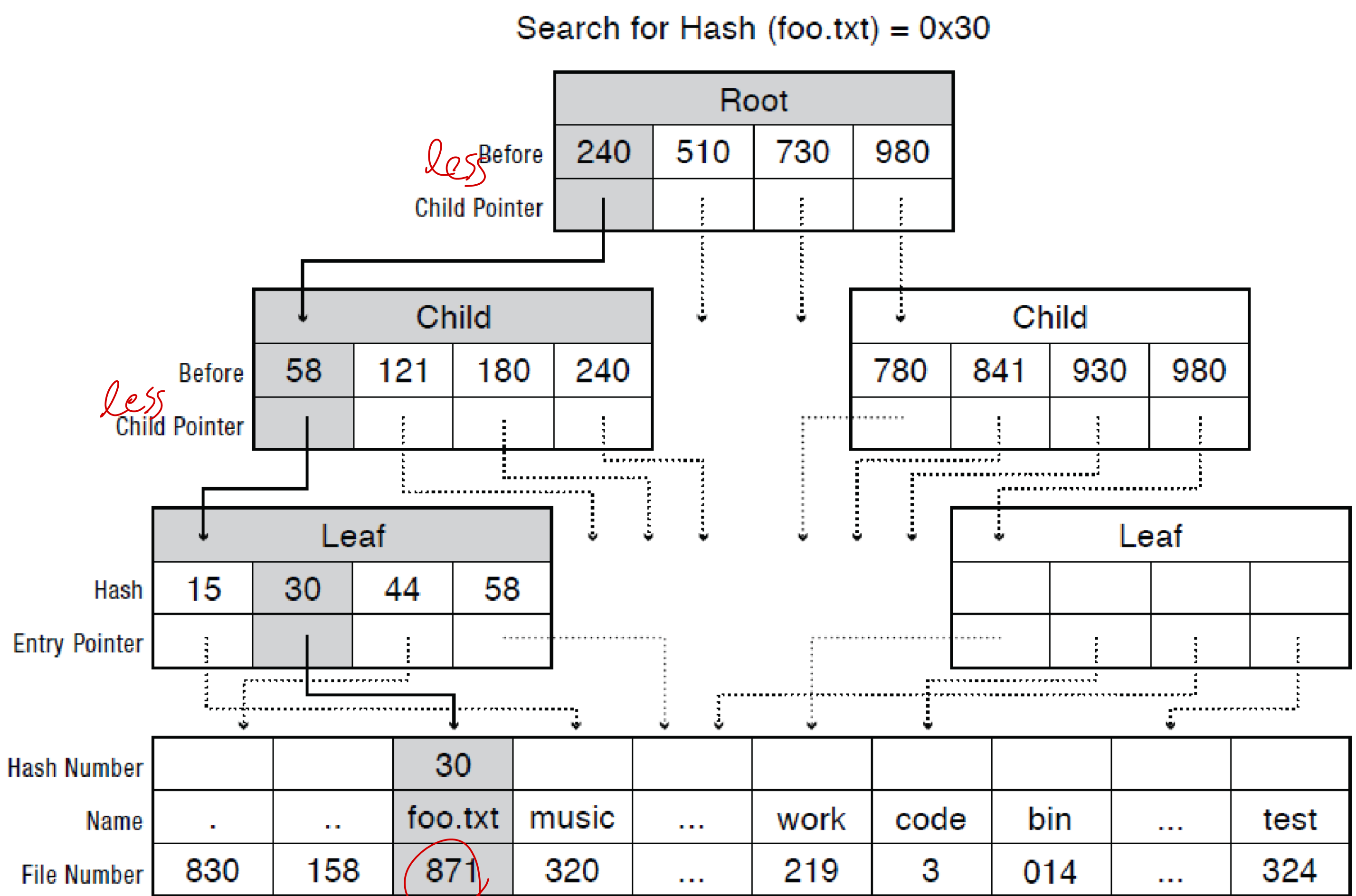


Actually B+ tree
OS

Large Directories: B Trees

30

B tree vs B+ tree

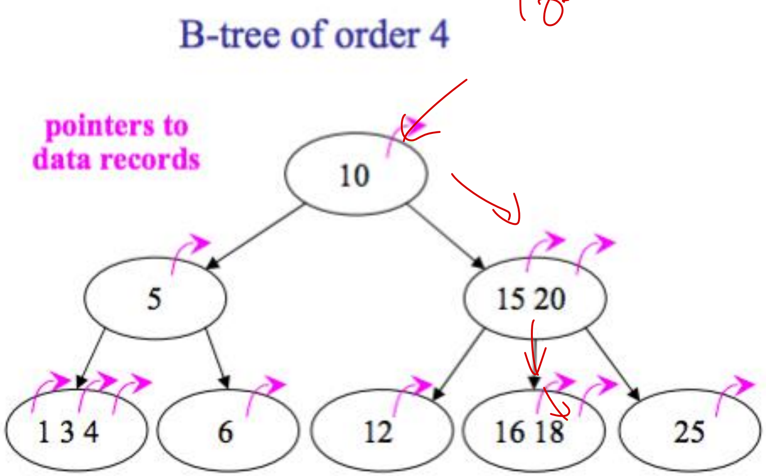


→ find inode → find block num → access

B tree vs B+ tree

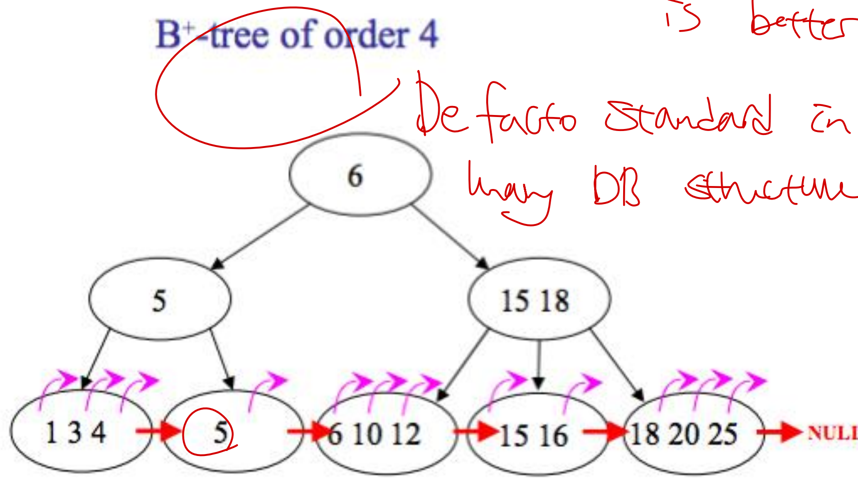
- In B+ tree, data is accessed only in leaf node

18



⊕ Traversal (ex. Sequential read is better)

De facto standard in many DB structure



Sometimes finish in internal node

ex. 5

Always goes to the leaf node

Large Directories: Layout