CS 318 Principles of Operating Systems

Fall 2021

Lecture 16: File System Implementation

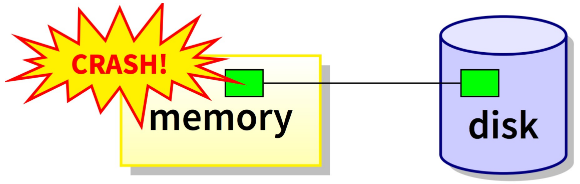
Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

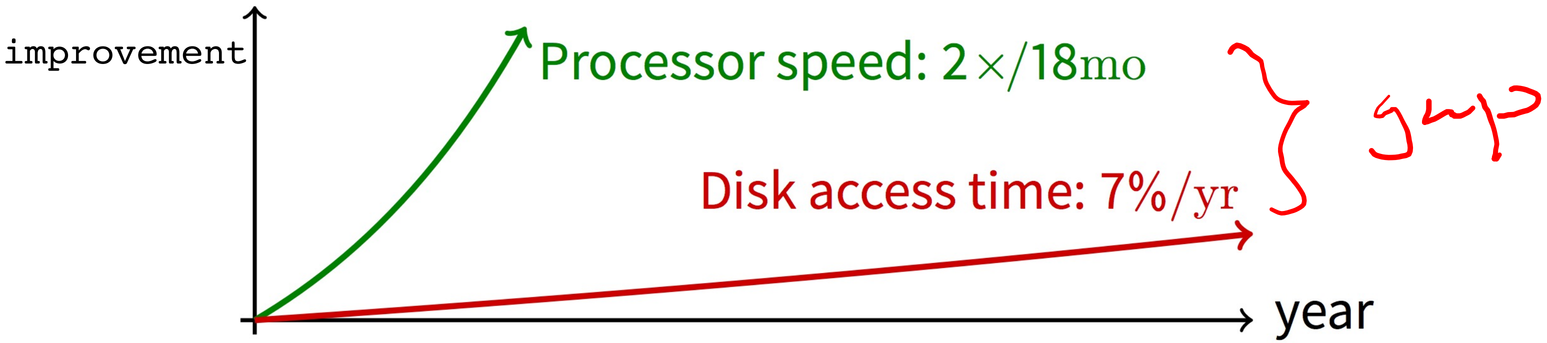
Why disks are different

Disk = First state we've seen that doesn't go away

- So: Where all important state ultimately resides



Slow (milliseconds access vs. nanoseconds for memory)



Huge (100–1,000x bigger than memory)

- How to organize large collection of ad hoc information?
- File System: Hierarchical directories, Metadata, Search

Disk vs. Memory

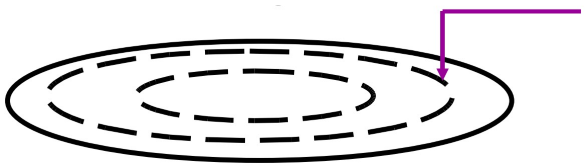
	Disk	MLC NAND Flash	DRAM
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	100 MB/s	550–2500 MB/s	> 1 GB/s
Sequential write	100 MB/s	520–1500 MB/s*	> 1 GB/s
Cost	$0.03/GB	$0.35/GB	$6/GiB
Persistence	Non-volatile	Non-volatile	Volatile

*: Flash write performance degrades over time

Disk Review

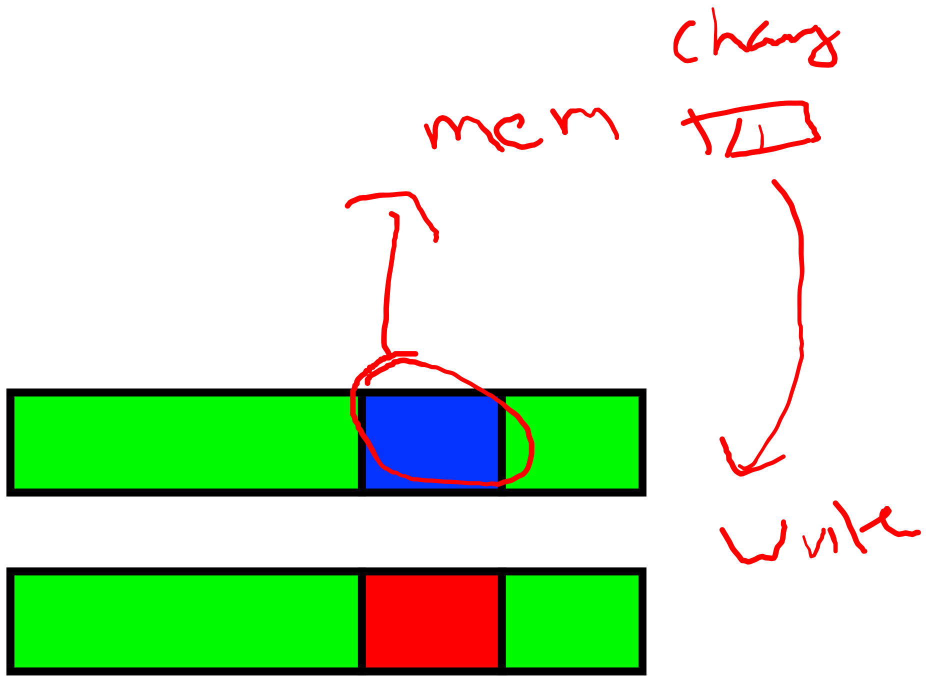
Disk reads/writes in terms of sectors, not bytes

- Read/write single sector or adjacent groups



How to write a single byte? "Read-modify-write"

- Read in sector containing the byte
- Modify that byte
- Write entire sector back to disk
- Key: if cached, don't need to read in



Sector = unit of atomicity.

- Sector write done completely, even if crash in middle (disk saves up enough momentum to complete)

Larger atomic units have to be synchronized by OS

Some Useful Trends (1)

Disk bandwidth and cost/bit improving exponentially

- Similar to CPU speed, memory size, etc.

Seek time and rotational delay improving very slowly

- Why? require moving physical object (disk arm)

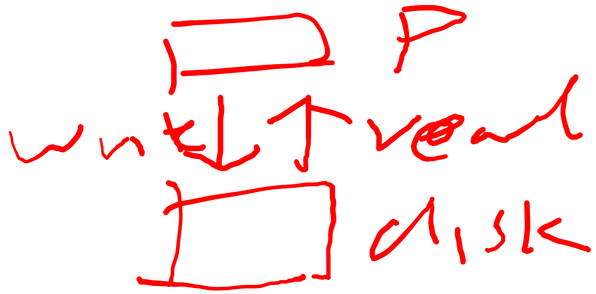
Disk access is a huge system bottleneck & getting worse

- Bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
- Trade bandwidth for latency if you can get lots of related stuff.

Some Useful Trends (2)

Desktop memory size increasing faster than typical workloads

- More and more of workload fits in file cache
- Disk traffic changes: mostly writes and new data



Memory and CPU resources increasing

- Use memory and CPU to make better decisions
- Complex prefetching to support more IO patterns
- Delay data placement decisions reduce random IO

Goal

Want: operations to have as few disk accesses as possible & have minimal space overhead (group related things)

What's hard about grouping blocks?

Like page tables, file system metadata constructs mappings

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

File Systems vs. Virtual Memory

In both settings, want location transparency

- Application shouldn't care about particular disk blocks or physical memory locations

In some ways, FS has easier job than VM:

- CPU time to do FS mappings not a big deal (why?) ➔ no TLB
- Page tables deal with sparse address spaces and random access, files often denser (0 . . . filesize − 1), ~sequentially accessed

In some ways, FS's problem is harder:

- Each layer of translation = potential disk access
- Space a huge premium! (But disk is huge?!?!)
  - Cache space never enough; amount of data you can get in one fetch never enough
- Range very extreme: Many files < 10 KB, some files GB

Some Working Intuitions

FS performance dominated by # of disk accesses

- Say each access costs ~10 milliseconds
- Touch the disk 100 times = 1 second
- Can do a billion ALU ops in same time!

Access cost dominated by movement, not transfer:

- 1 sector: $5ms + 4ms + 5\mu s\ (\approx 512\ B/(100\ MB/s)) \approx 9ms$
- 50 sectors: $5ms + 4ms + .25ms = 9.25ms$
- Can get 50x the data for only ~3% more overhead!

Observations that might be helpful:

- All blocks in file tend to be used together, sequentially
- All files in a directory tend to be used together
- All names in a directory tend to be used together

Problem: How to Track File's Data

Disk management:

- Need to keep track of where file contents are on disk
- Must be able to use this to map byte offset to disk block
- Structure tracking a file's sectors is called an index node or inode
- *inodes* must be stored on disk, too why?

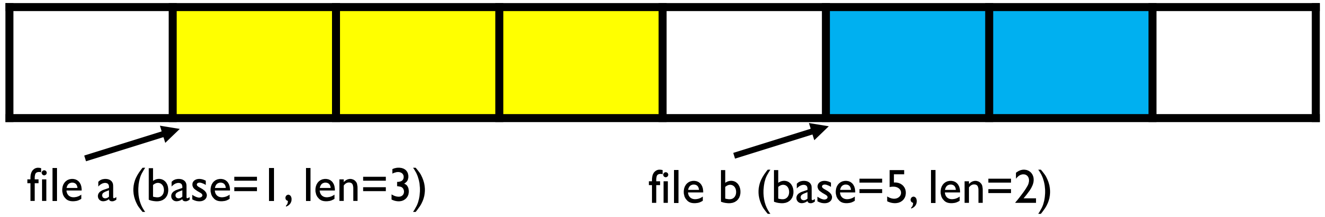
Things to keep in mind while designing file structure:

- Most files are small
- Much of the disk is allocated to large files
- Many of the I/O operations are made to large files
- Want good sequential and good random access (what do these require?)

Straw Man: Contiguous Allocation

"Extent-based": allocate files like segmented memory

- When creating a file, make the user pre-specify its length and allocate all space at once
- Inode contents: location and size



What happens if file c needs 2 sectors?

Example: IBM OS/360

Pros?

- Simple, fast access, both sequential and random

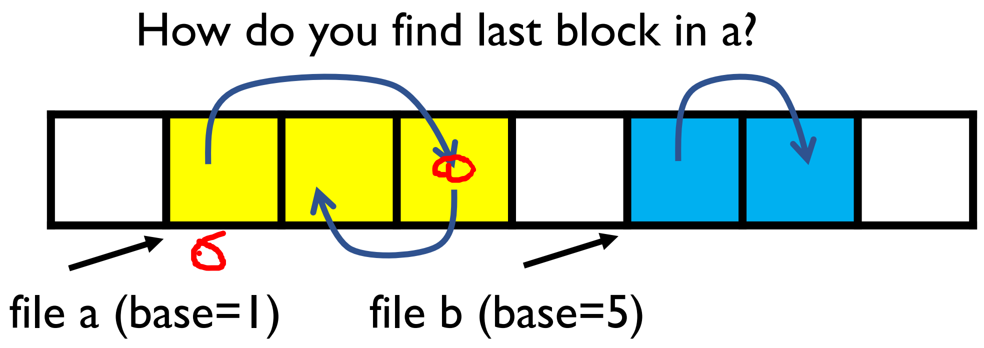
Cons? (Think of corresponding VM scheme)

- Files may not dynamically grow after creation
- External fragmentation

Straw Man #2: Linked Files

Basically a linked list on disk.

- Keep a linked list of all free blocks
- Inode contents: a pointer to file's first block
- In each block, keep a pointer to the next one



Examples (sort-of): Alto, TOPS-10, DOS FAT

Pros?

- Easy dynamic growth & sequential access, no fragmentation

Cons?

- Linked lists on disk a bad idea because of access times
- Random very slow (e.g., traverse whole file to find last block)
- Pointers take up room in block, skewing alignment

Example: DOS FS (simplified)

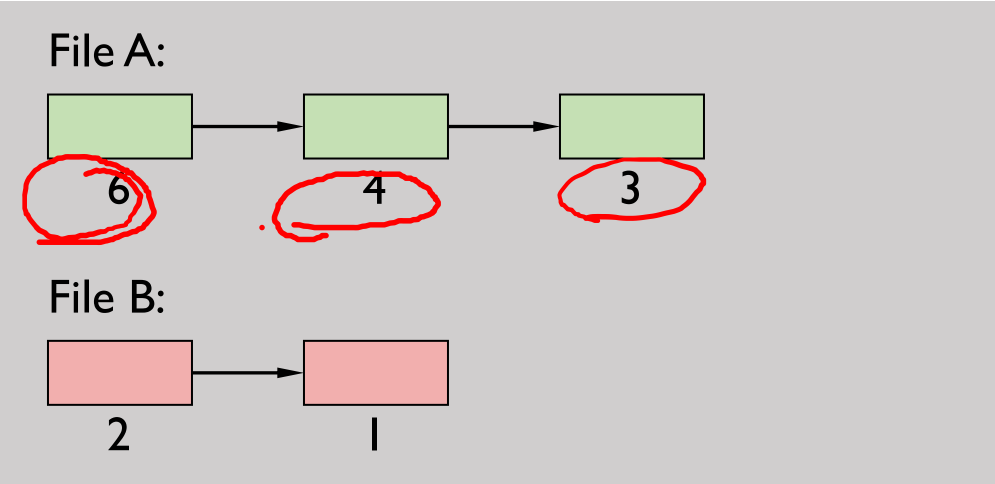
Linked files with key optimization: *puts links in fixed-size "file allocation table" (FAT) rather than in ~~each data block~~.*

Directory (5)

A: 6	
B: 2	

FAT (16-bit entries)

0	free
1	EOF
2	1
3	EOF
4	3
5	EOF
6	4
	...



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

FAT Discussion

Entry size = 16 bits (initial FAT16 in MS-DOS 3.0)

- What's the maximum size of the FAT? 65,536 entries
- Given a 512 byte block, what's the maximum size of FS? 32MiB
- One solution: go to bigger blocks. Pros? Cons?

Space overhead of FAT is trivial:

- 2 bytes / 512 byte block = ~ 0.4% (Compare to Unix)

Reliability: how to protect against errors?

- Create duplicate copies of FAT on disk
- State duplication a very common theme in reliability

Bootstrapping: where is root directory?

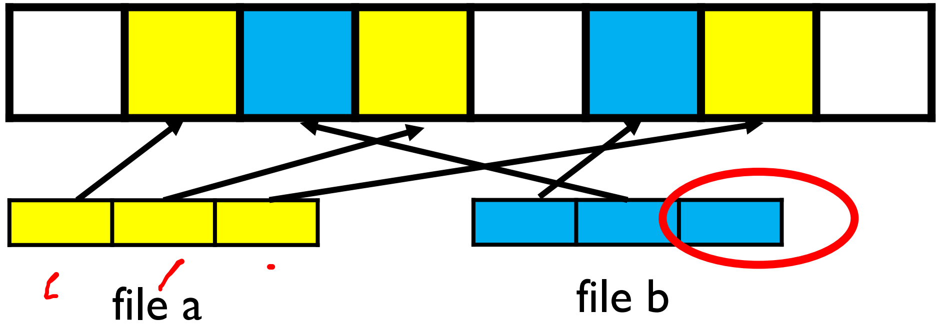
- Fixed location on disk:

FAT	FAT (opt)	Root dir	…

Another Approach: Indexed Files

Each file 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 (static or dynamic?)
- Allocate array to hold file's block pointers on file creation
- Allocate actual blocks on demand using free list



Pros?

- Both sequential and random access easy

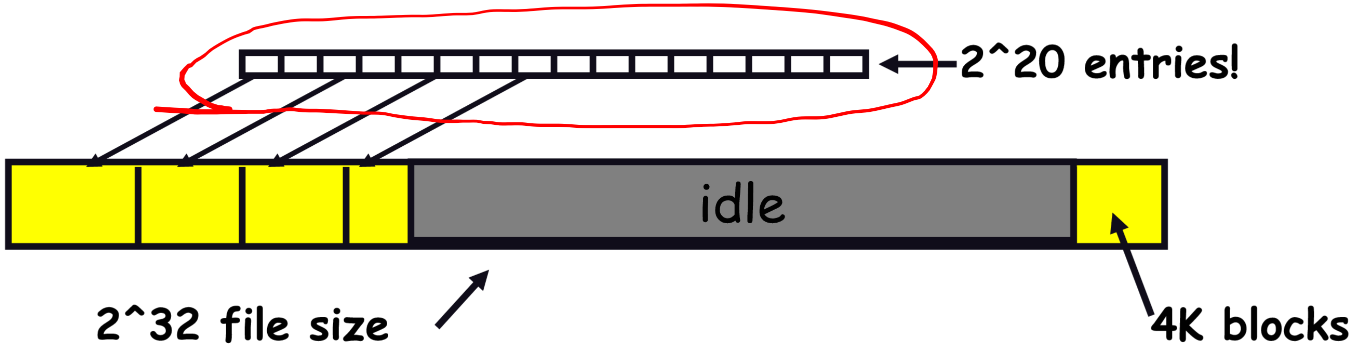
Cons?

- Mapping table requires large chunk of contiguous space
- ...Same problem we were trying to solve initially

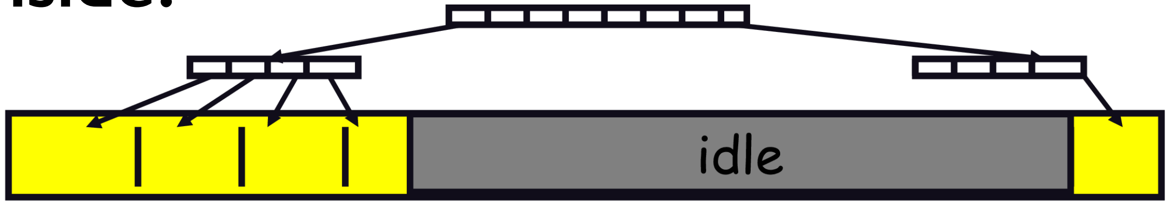
Indexed Files

Issues same as in page tables

- Large possible file size = lots of unused entries
- Large actual size? table needs large contiguous disk chunk



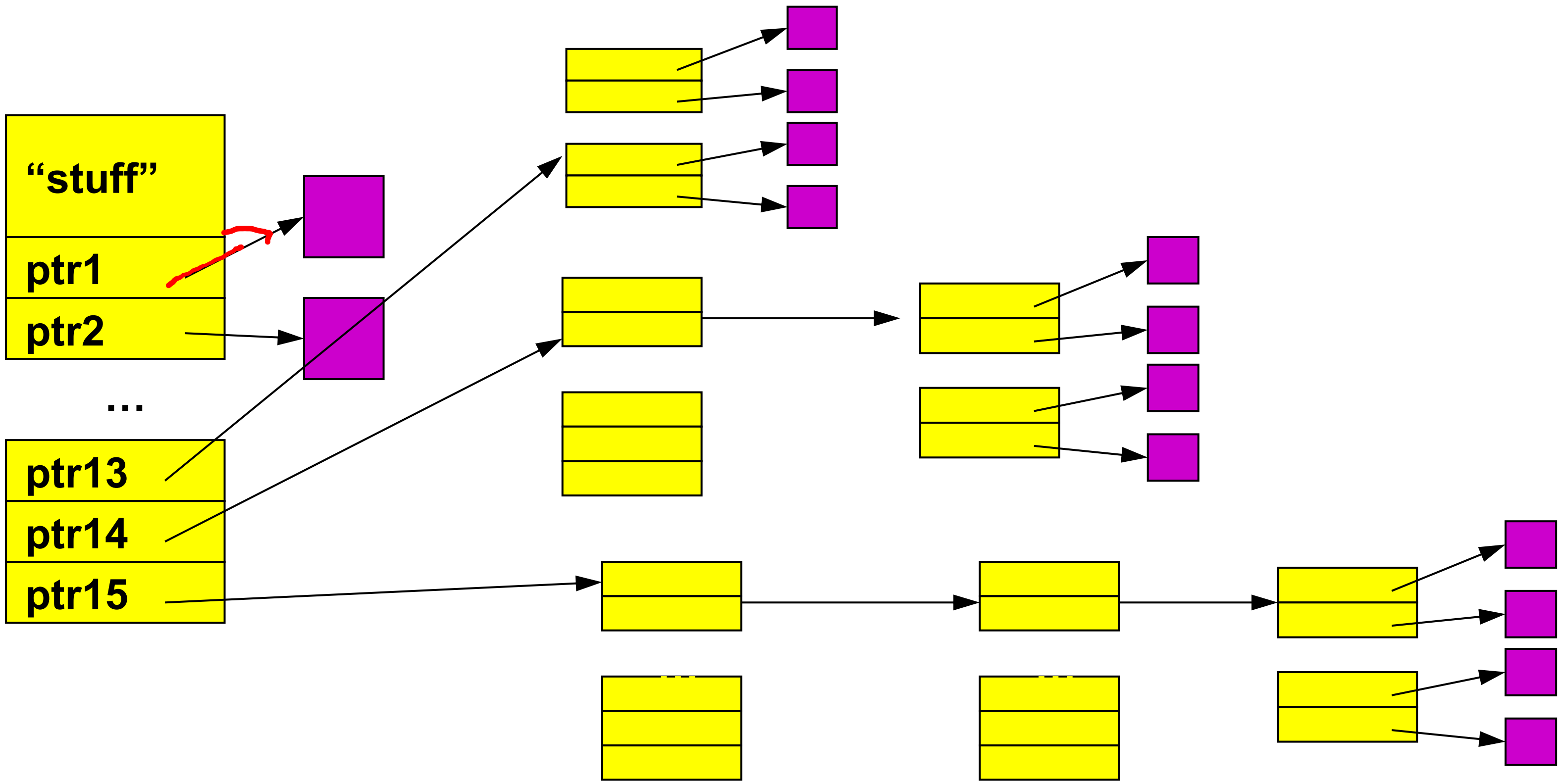
Solve identically: small regions with index array, this array with another array, ... Downside?



Multi-level Indexed Files: Unix inodes

inode = 15 block pointers + "stuff"

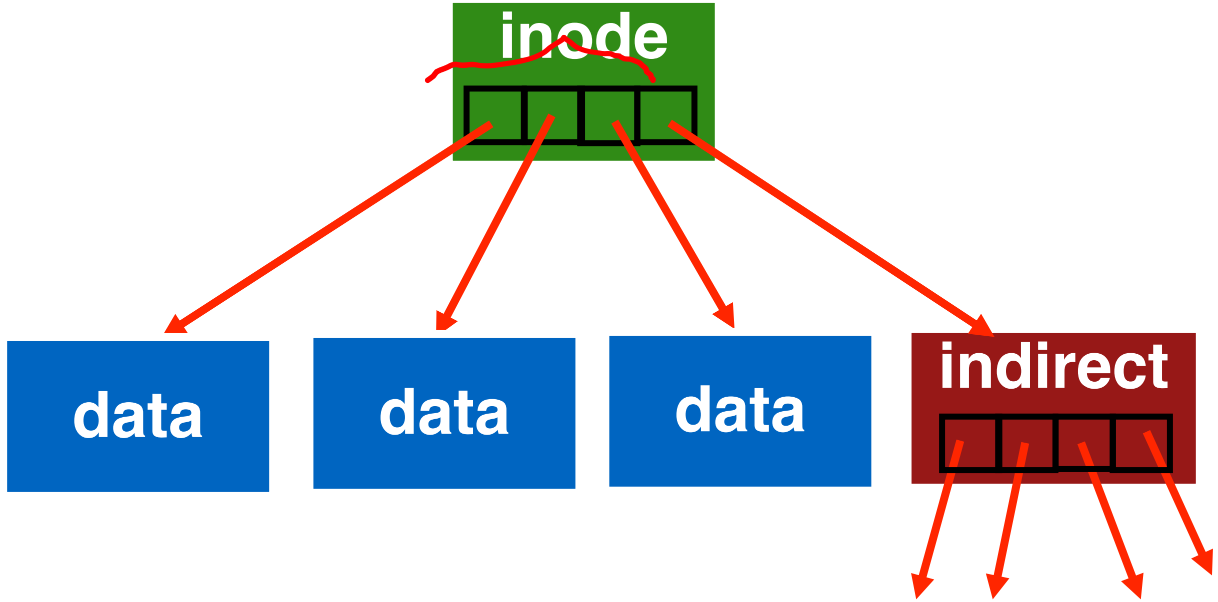
- first 12 are direct blocks: solve problem of first blocks access slow
- then single, double, and triple indirect block



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



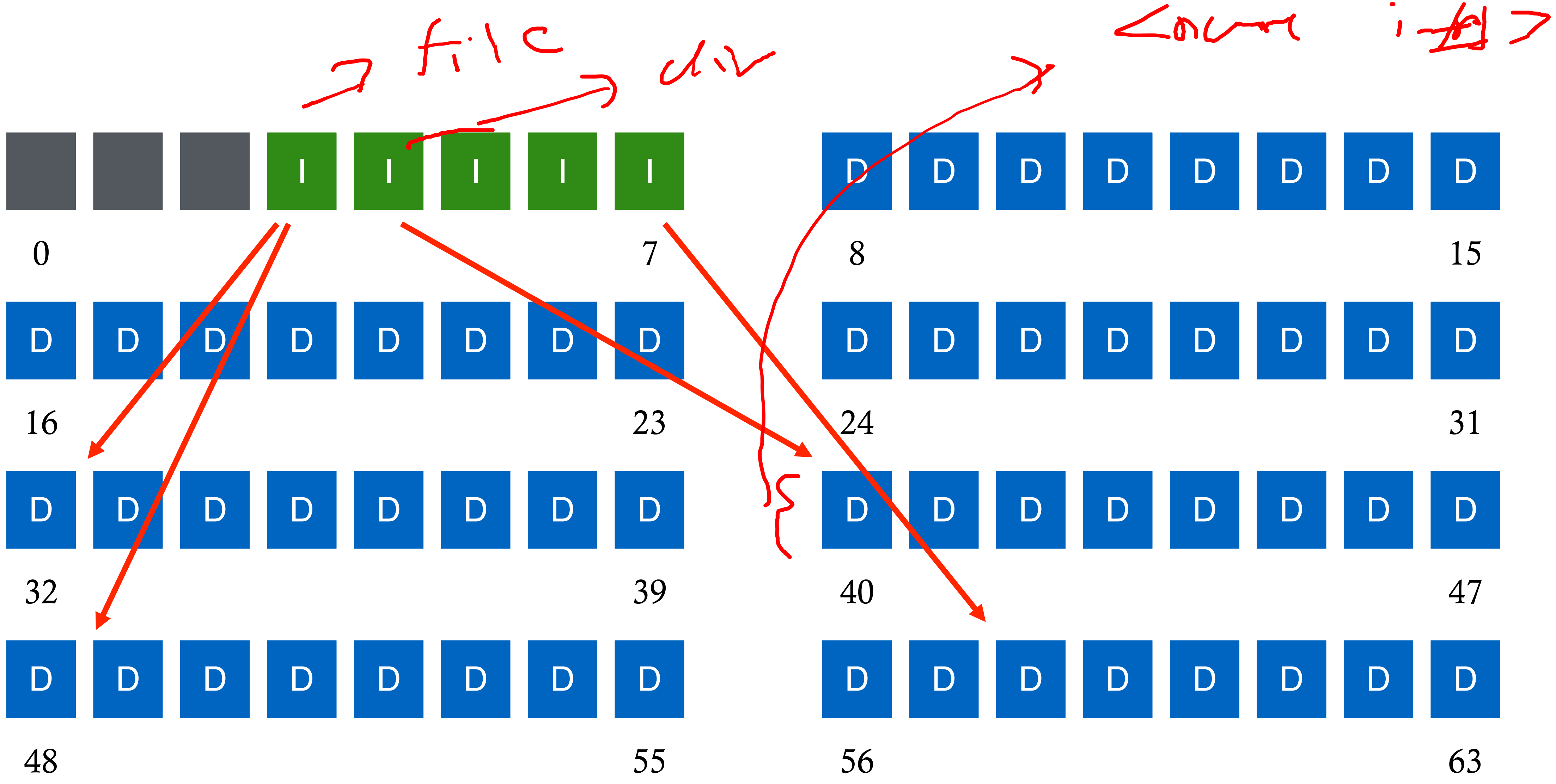
More About inodes

inodes are stored in a fixed-size array

- 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

More About inodes



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, e.g., to open "/a.txt"

- Use Master Block to find inode for "/" on disk
- (1) Read inode for "/" into memory, (2) look for entry for "a.txt"
- This entry gives the disk block number for the inode for "a.txt"
- (3) Read the inode for "a.txt" into memory
- (4) The inode says where first data block is on disk
- Read that block into memory to access the data in the file

How many disk accesses are required?

What about reading "/a/b/c.txt"

File Buffer Cache

Disk operations are slow…

Applications exhibit locality for reading and writing files

Idea: Cache file blocks in memory to capture locality

- Called the file buffer cache
- Cache is system wide, used and shared by all processes
- Reading from the cache makes a disk perform like memory
- Even a small cache can be very effective

Issues

- The file buffer cache competes with VM (tradeoff here)
- Like VM, it has limited size
- Need replacement algorithms again (LRU usually used)

Caching Writes

On a write, some applications assume that data makes it through the buffer cache and onto the disk

- As a result, writes are often slow even with caching

OSes typically do write back caching

- Maintain a queue of uncommitted blocks
- Periodically flush the queue to disk (30 second threshold)
- If blocks changed many times in 30 secs, only need one I/O
- If blocks deleted before 30 secs (e.g., /tmp), no I/Os needed

Unreliable, but practical

- On a crash, all writes within last 30 secs are lost
- Modern OSes do this by default; too slow otherwise
- System calls (Unix: fsync) enable apps to force data to disk

Read Ahead

Many file systems implement "read ahead"

- FS predicts that the process will request next block
- FS goes ahead and requests it from the disk
- This can happen while the process is computing on previous block
  - Overlap I/O with execution
- When the process requests block, it will be in cache
- Compliments the disk cache, which also is doing read ahead

For sequentially accessed files can be a big win

- Unless blocks for the file are scattered across the disk
- File systems try to prevent that, though (during allocation)

Summary

File System Layouts

- Unix inodes

File Buffer Cache

- Strategies for handling writes

Read Ahead

Next Time…

Read Chapter 41, 42