Unix File System, FFS, LFS, File System Caching

EE469, April 6

Yiying Zhang

Tim Berners-Lee

- www

2017

Readings

- Chapters 10-11
- Comet: Chapter 41, 43

Roadmap

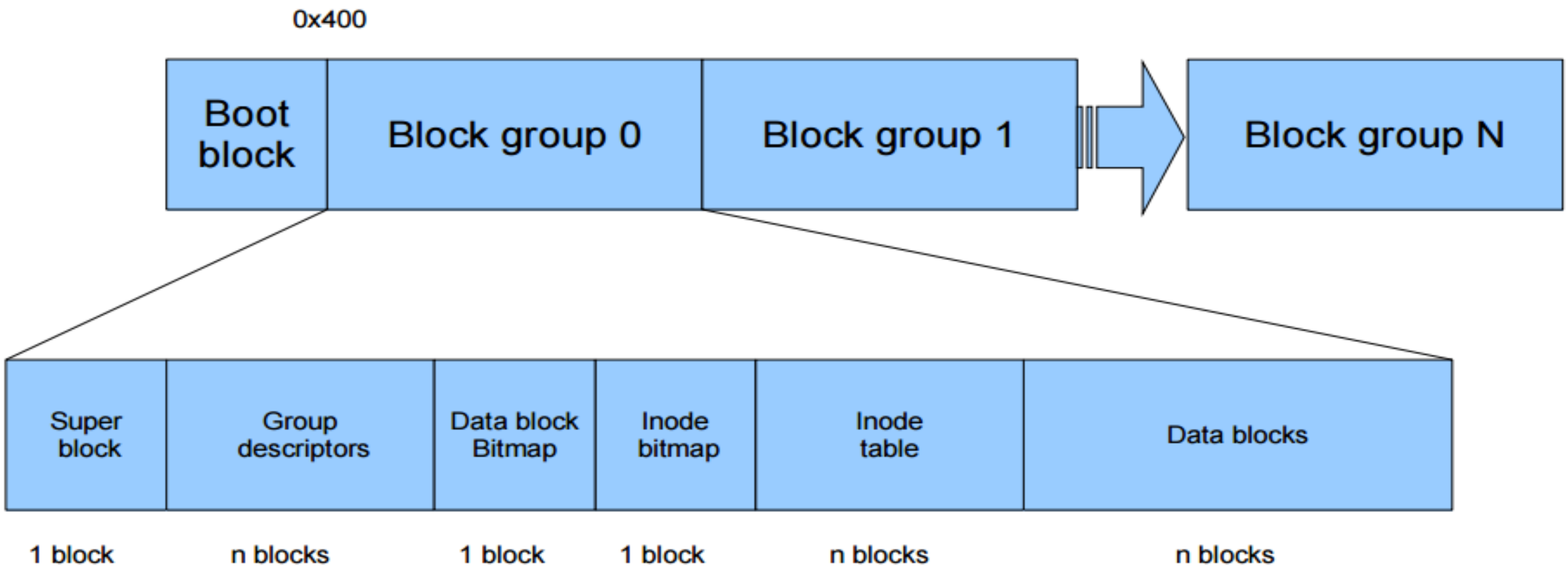
- Functionality (API)
  - Basic functionality
    - Disk layout
    - File operations (open, read, write, close)
  - Directories
- Performance
  - Disk allocation
  - File system designs
  - Buffer cache
- Reliability
  - FS level
  - Disk level: RAID

Disk Allocation revisited – many low level details

- How to keep blocks for a file together?
- How about inode and data blocks for a file?
  - It is a good idea to keep them close?
  - If so, how?
- How about files in the same directory?
  - e.g. make

[lec22] Disk Layout of a FS

- Disk layout of a file system

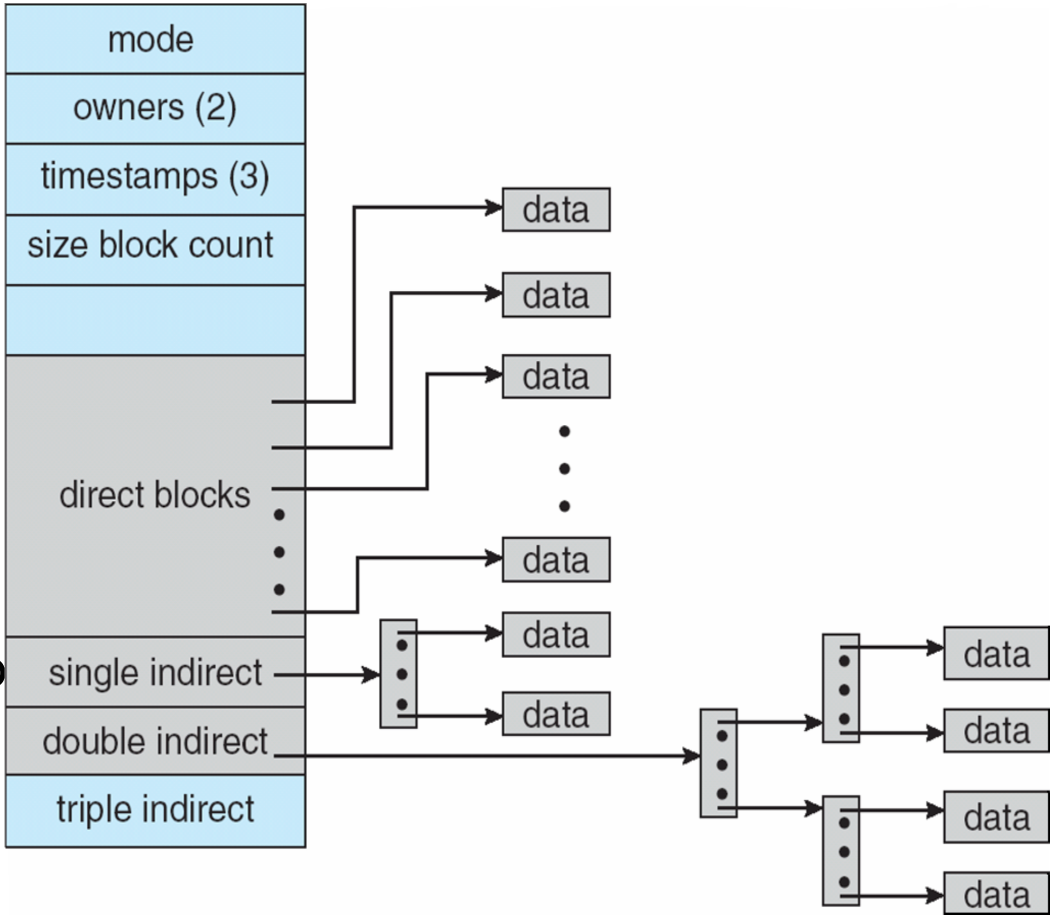


[lec22] Disk Allocation

- Context:
  - A file has logical bytes/blocks
  - Different files share physical block space on disk
- Original goals are
  - How to allocate blocks for a file – for access performance
    - Random vs. sequential accesses
  - How to index the blocks for a file – for finding the blocks on disk
- Disk allocation methods:
  - Contiguous
  - Extent-based
  - Linked files / FAT
  - Single-level indexing
  - Multi-level indexing

[lec22] Multi-Level Indexed Files (UNIX)

- 13 Pointers in a header
  - 10 direct pointers
  - 11: 1-level indirect
  - 12: 2-level indirect
  - 13: 3-level indirect
- Pros & Cons
  - In favor of small files
  - Can grow
  - Limit is 16G (w/ 1K block)
  - Random access has up to 4 seeks
- How many disk accesses to load block 23, 5, 340?

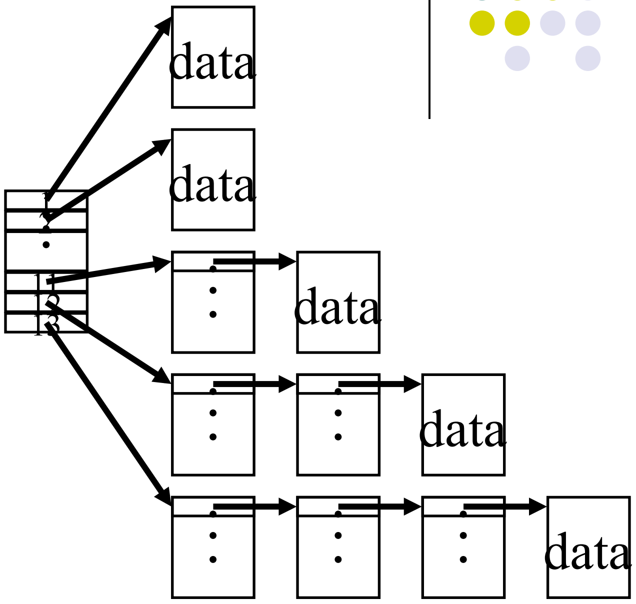


Analogy in memory management?

Indirect Blocks

- Block as the basic address unit, BLOCKSIZE is constant
- 13 three-byte pointers point either directly or indirectly to the disk blocks containing the data contents of the file.
  - Pointers 0-9: addresses of direct blocks containing file data
  - Pointer 10: address of a single indirect block, a block containing the addresses of direct blocks
  - Pointer 11: address of a double indirect block, a block containing the addresses of single indirect blocks which contain the addresses of direct blocks
  - Pointer 12: address of a triple indirect block

Indirect blocks addressing ranges



- Assume blocksize = 1K
  - a block contains 1024 / 4 = 256 block addresses
- direct block address: 10K
  - indirect block addresses: 256K
  - double indirect block addresses: 256 * 256K = 64M
  - tripe indirect block addresses: 256 * 64M = 16G

What happens in accessing block 23, 5, 340?

UNIX File System

- UNIX (1969)
  - One of the most popular operating systems
  - Evolving since escaping from Bell lab early 70's
  - Written in C with small kernel
- Other important events in 1969?
  - Man landed on the Moon
  - Internet was born (4 nodes!)
  - Linus Torvalds was born

UNIX File System (UFS)

- Overall structure of the file storage and control on UNIX
- One of the most significant aspects of UNIX

UFS Overview

- Anything can be viewed as a file: devices, networking
- All files as streams of bytes in UNIX kernel
- Hierarchical, directory-based
- Four types of files
  - regular file: ASCII files
  - directory-type file: map file names to the contents in a directory
  - special file: printers, terminals, other devices
  - named pipe: FIFO

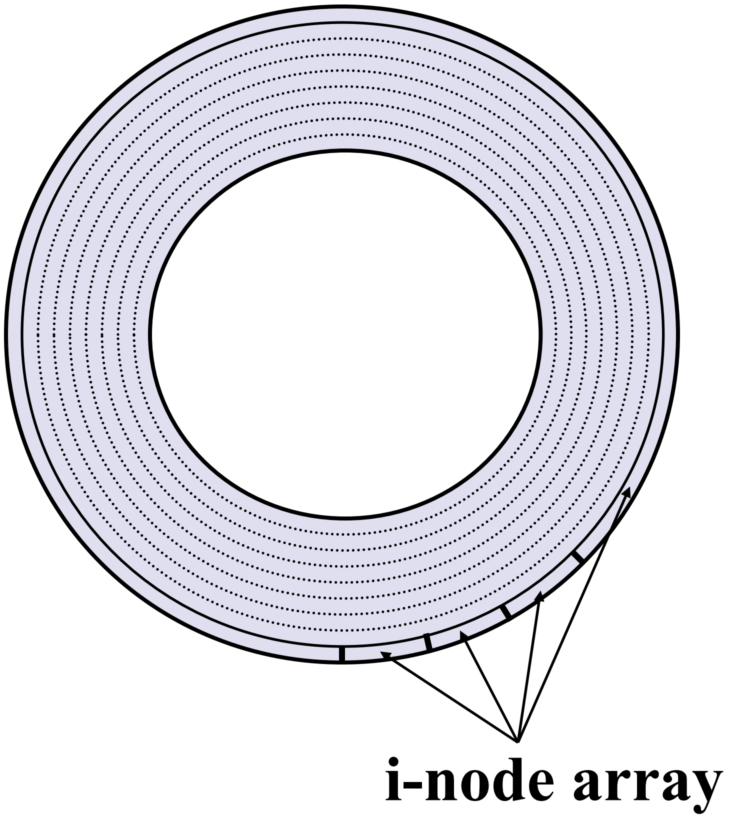
UFS Overview (cont)

- inode: index node representing a file
  - Every access to the file must make use of the information of the inode.

- UNIX supports multiple file systems
  - one in charge of UNIX system startup
  - others can be “mounted” or “removed”
    - on disk, CD-ROM, floppy, over network

Early Unix Disk Layout

- An array of inodes in outermost cylinders
- inode number is index into the inode array
- Problems
  - inodes are far away from data blocks
  - fixed max number of files



Disk Organization of UFS

- Boot Block: the first block in a UNIX file system, contains the boot program and other initialization information or unused
- Super Block: always the second block, contains specific information about the file system
- Inode array (similar to inode map):
  - list of inodes for the file system
  - Contiguous
  - always follows the super block
  - number specified by the system admin at format time
- Data Blocks: immediately follow the i-list and consume the rest of the blocks

UFS Free Blocks Organization

- All free blocks appear in *free-block chain*.
- free-block chain is a linked list of *free-block address blocks*.
- The superblock contains the head of the free-block chain

Original UFS Problems

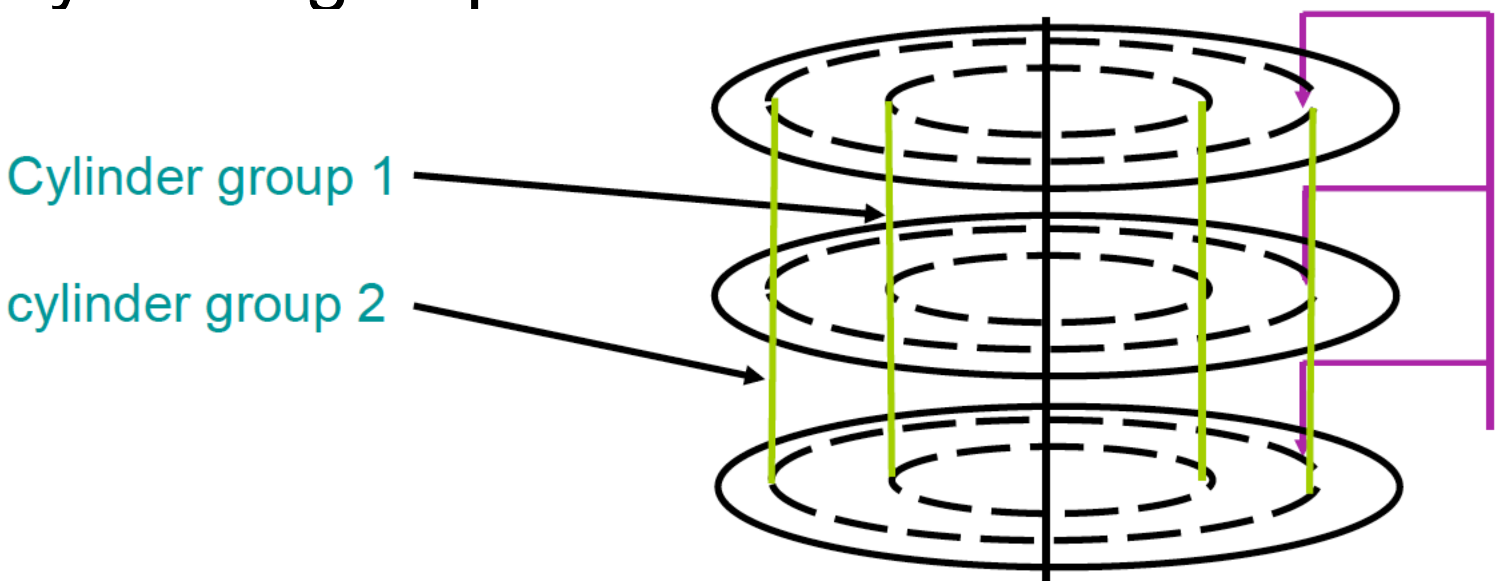
- Original Unix FS had two major problems:
  - 1. data blocks are allocated randomly in aging file systems (using linked list)
    - blocks for the same file allocated sequentially when FS is new
    - as FS “ages” and fills, need to allocate blocks freed up when other files are deleted
      - problem: deleted files are essentially randomly placed
      - so, blocks for new files become scattered across the disk!
  - 2. inodes are allocated far from blocks
    - all inodes at beginning of disk, far from data
    - traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks
  - BOTH of these generate many long seeks!

Cylinder groups

- Each disk partition is subdivided into groups of consecutive cylinders
  - data blocks from a file all placed in same cylinder group
  - files in same directory placed in same cylinder group
  - inode for file in same cylinder group as file's data
- Each cylinder group contains a **bit map** of all available blocks in the cylinder group
  - Better than linked list

Clustering related objects in FFS

- 1 or more consecutive cylinders into a "cylinder group"

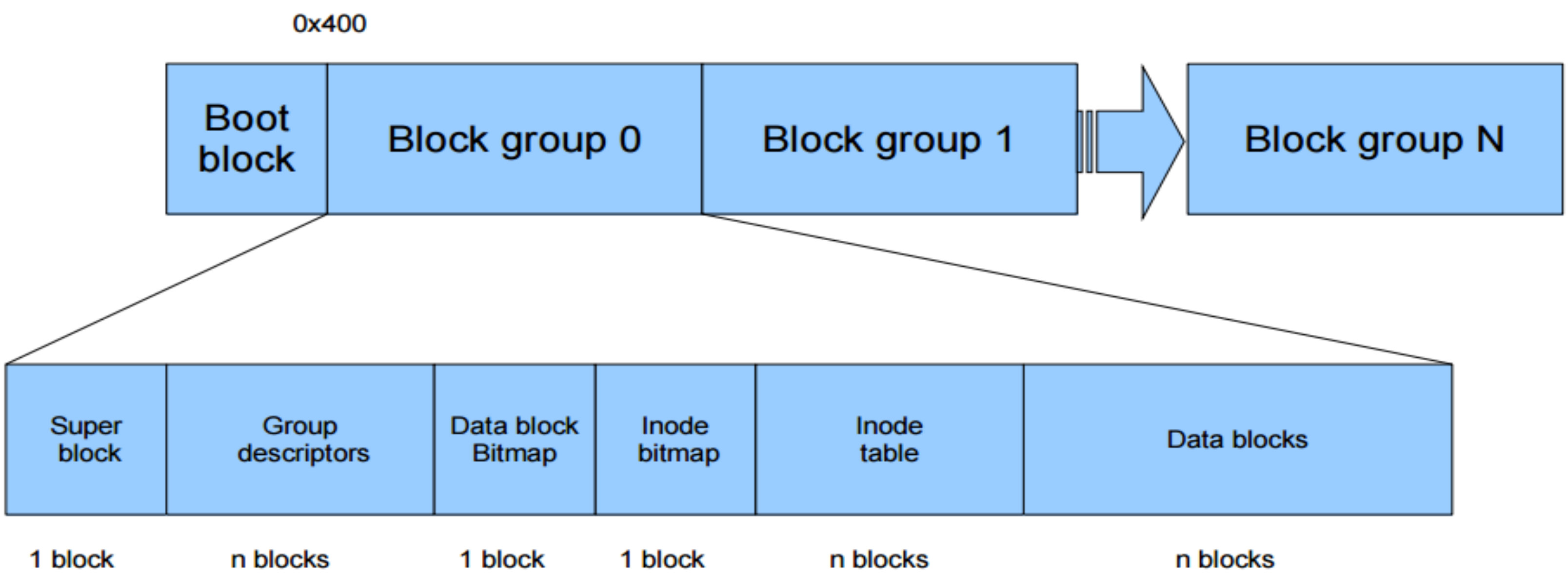


- Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
- Tries to put everything related in same cylinder group
- Tries to put everything not related in different group

THE FAST FILE SYSTEM（FFS）

- BSD 4.2 introduced the “fast file system”
  - ***Superblock*** is replicated on different cylinders of disk
  - Have one inode table per group of cylinders
    - It minimizes disk arm motions
  - Inode has now 15 block addresses
  - Minimum block size is 4K

Disk Layout of FFS



Jon Ousterhout

- Systems
- Log structured file systems
- RamCloud
- Stanford

Mendel Rosenblum

- Systems
- VMWare
- Log Structured File system

Margo Seltzer

- Systems/storage/database
- BerkelyDB, provenance, file systems
- Harvard

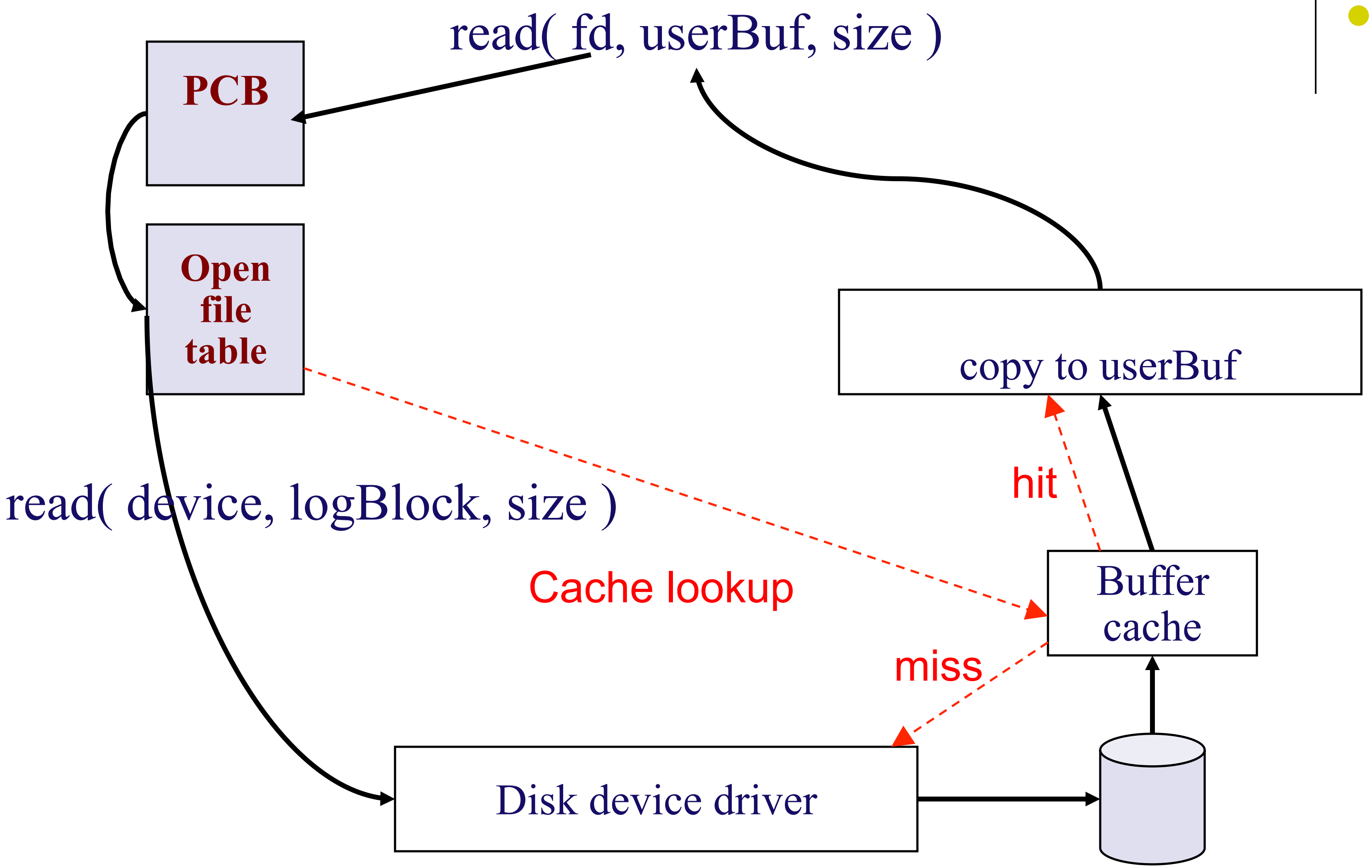
Roadmap

- Functionality (API)
  - Basic functionality
    - Disk layout
    - File operations (open, read, write, close)
  - Directories
- Performance
  - Disk allocation
  - File system designs
  - Buffer cache
- Reliability
  - FS level
  - Disk level: RAID

"Principle of locality" once more

- Locality of reference in file accesses
  - Yet another application of the principle of locality
  - What were the earlier instances in this class?
- Keep a number of disk blocks in "the much faster" memory
  - when accessing disk, check the cache first!
- File system buffer caches are maintained in software

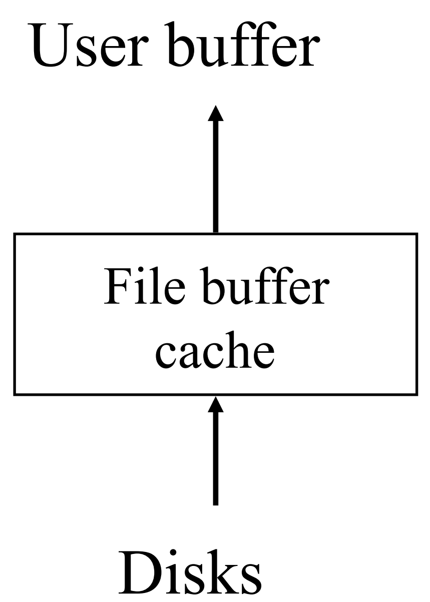
Reading A Block



Modern disk drives are addressed as large one-dimensional arrays of logical blocks

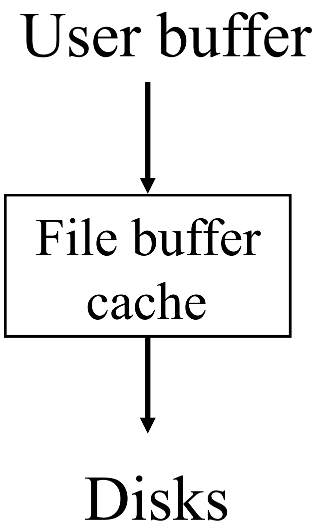
Read operations in presence of buffer cache

- read(fd, buf, n)
  - On a hit
    - copy from the buffer cache to a user buffer
  - On a miss
    - replacement if necessary
    - read a file block into the buffer cache



Write operations: Maintaining Consistency

- write(fd, buffer, n)
  - On a hit
    - write to buffer cache
  - On a miss
    - Read first
    - Then write (hit)



File persistence under file caching

- Problem: fast cache memory is volatile, but users expect disk files to be persistent
  - In the event of a system crash, dirty blocks in the buffer cache are lost !
  - Example 1: creating “/dir/a”
    - Allocate inode (from free inode list) for “a”
    - Update parent dir content – add (“a”, inode#) to “dir”

- Solution 1: use write-through cache
  - Modifications are written to disk immediately
    - (minimize “window of opportunities”)
  - No performance advantage for disk writes

File persistence under file caching

- Possible solution 2: write back cache
  - Gather (buffer) writes in memory and then write all buffered data back to storage devices
  - e.g., write back dirty blocks after no more than 30 seconds
  - e.g., write back all dirty blocks during file close

- Problem with this?

Other performance optimizations

- Read-ahead (e.g. Linux)
  - For sequential access, read the requested block and the following N blocks together (why is this a good idea?)

- Write-behind:
  - Start disk write, but don’t make application wait until the disk operation completes

- Allow overlap of a process’s computation with its own disk I/O (e.g. AIO in FreeBSD)

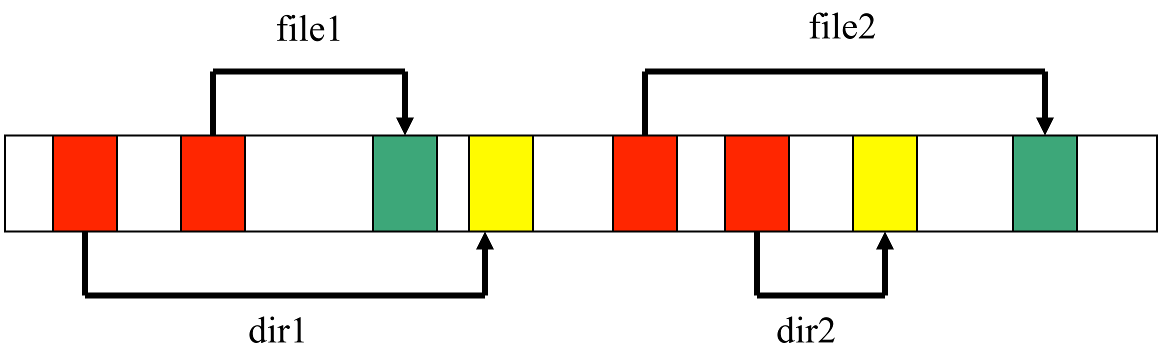
Log-Structured File Systems

- LFS was designed in response to two trends in workload and disk technology:
  1. Disk bandwidth scaling significantly (40% a year)
     - but, latency is not
  2. RAM & caches are bigger
  - So, a lot of reads do not require disk access
  - Most disk accesses are writes ⇒ pre-fetching not very useful
  - Worse, most writes are small ⇒ 10 ms overhead for 50 μs (in mem) write
  - Example: to create a new file:
    - inode of directory needs to be written
    - Directory block needs to be written
    - inode for the file has to be written
    - Need to write the file
  - Delaying these writes could hamper consistency
- Solution: LFS to utilizes full disk bandwidth

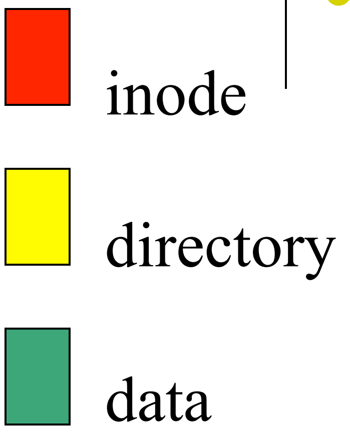
LFS Basic Idea

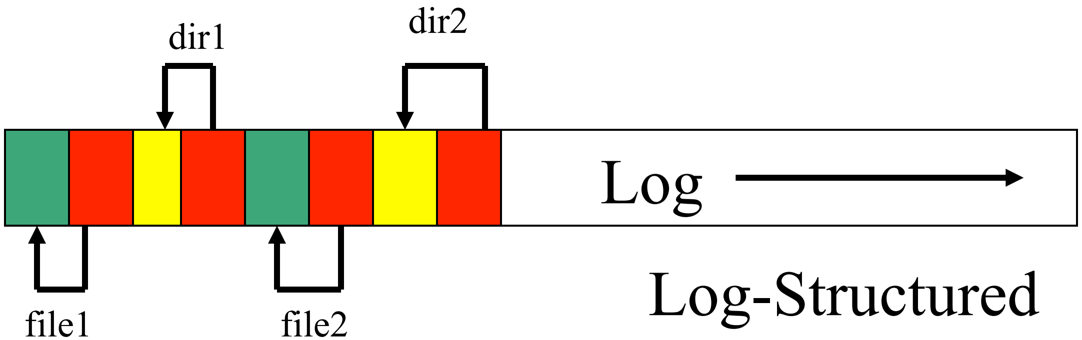
- Structure the disk as a sequential log
  - Periodically, all pending writes buffered in memory are collected in a single segment
  - The entire segment is written contiguously at end of the log
- Segment may contain inodes, directory entries, data
  - Start of each segment has a summary
  - If segment around 1 MB, then full disk bandwidth can be utilized

UFS vs. LFS



Unix File System





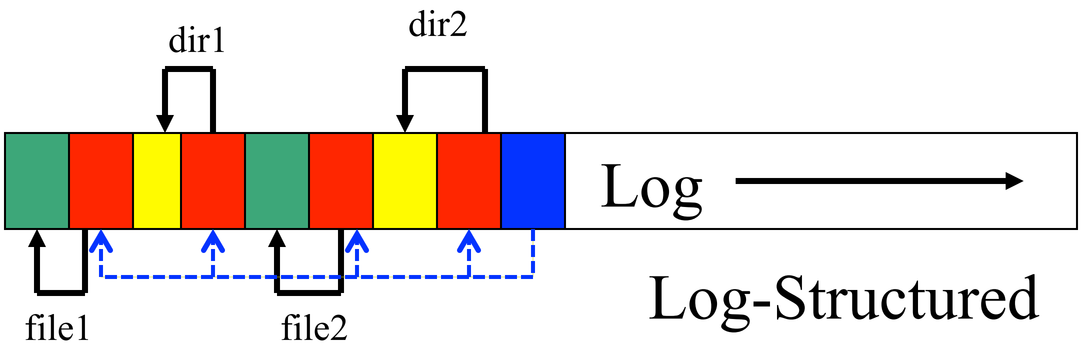
Log-Structured File System

Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

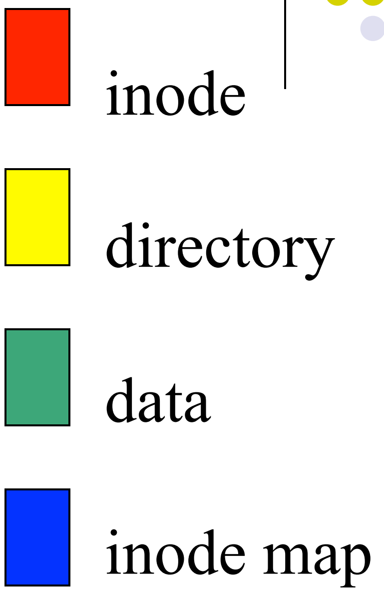
LFS: Locating Data

- FFS uses inodes to locate data blocks
  - inodes preallocated in each cylinder group
  - directories contain locations of inodes
- LFS appends inodes to end of log, just like data
  - makes them hard to find
- Solution:
  - use another level of indirection: inode maps
  - inode maps map file #s to inode location
  - location of inode map blocks are kept in a checkpoint region
  - checkpoint region has a fixed location
  - cache inode maps in memory for performance

LFS



Log-Structured File System



Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

LFS Cleaning

- Finite disk space implies that the disk is eventually full
  - Fortunately, some segments have stale information
  - A file overwrite causes inode to point to new blocks
    - Old ones still occupy space
- Solution: LFS Cleaner thread compacts the log
  - Read segment summary, and see if contents are current
    - File blocks, inodes, etc.
  - If not, the segment is marked free, and cleaner moves forward
  - Else, cleaner writes content into new segment at end of the log
  - The segment is marked as free!
- Disk organized as a circular buffer, writer adds contents to the front, cleaner cleans content from the back

An Interesting Debate

- Ousterhout vs. Seltzer
  - OS researchers have very “energetic” personalities
    - famous for challenging each others’ ideas in public
  - Seltzer published a 1995 paper comparing and contrasting LFS with FFS
    - Ousterhout published a “critique of Seltzer’s LFS Measurements”, rebutting arguments that LFS performs poorly in some situations
    - Seltzer published “A Response to Ousterhout’s Critique of LFS Measurements”, rebutting the rebuttal…
    - Ousterhout published “A Response to Seltzer’s Response”, rebutting the rebuttal of the rebuttal…
  - moral of the story:
    - *very* difficult to predict how a FS will be used
      - so it’s hard to generate reasonable benchmarks, let alone a reasonable FS design
    - *very* difficult to measure a FS in practice
      - depends on a HUGE number of parameters, including workload and hardware architecture