EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha

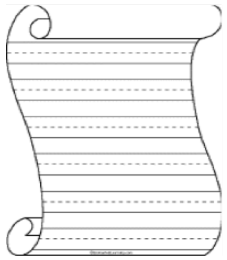
Recap: inodes

```
/*
 * Definitions for on-disk data structures.
 */
struct fs_direntry {
    char name[FS_MAXFILENAME + 1];          // name of this file or directory
    uint32_t inode_block;                   // disk block that stores the inode for
                                            // this file or directory
};

struct fs_inode {
    char type;                              // file ('f') or directory ('d')
    char owner[FS_MAXUSERNAME + 1];
    uint32_t size;                          // size of this file or directory
                                            // in blocks
    uint32_t blocks[FS_MAXFILEBLOCKS];      // array of data blocks for this
                                            // file or directory
};
```

Implementing transactions with logging

- Write-ahead logging
  - Write updates to append-only log ***before*** applying updates to file system
  - Write commit sector to end of log to commit the set of changes



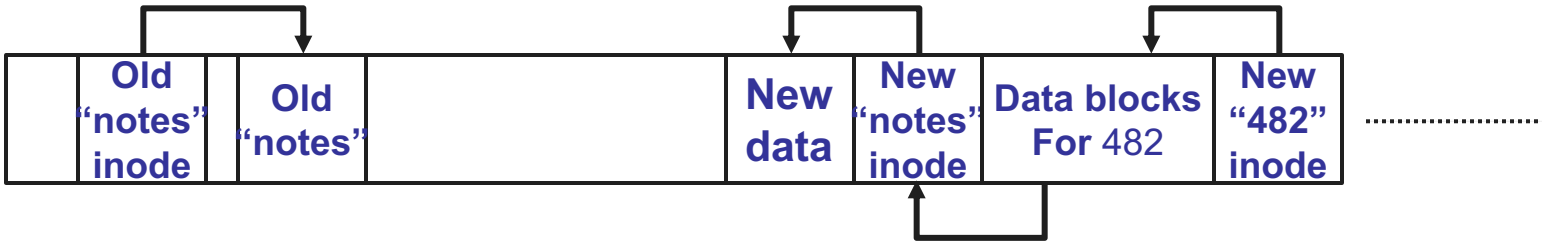
- Eventually, copy new data from log to the in-place version of the file system

- Again, update committed by single sector write

Case study: Log-structured file system

- Goal: Make (almost) all I/Os sequential
  - File system can write to any free disk block
  - In general, not possible for reads; leverage caching
- Basic idea: Treat disk as an append-only log
  - Append all writes to log
- What does it take to update the data in `/home/barisk/482/notes`?

Updating Data in /home/barisk/482/notes



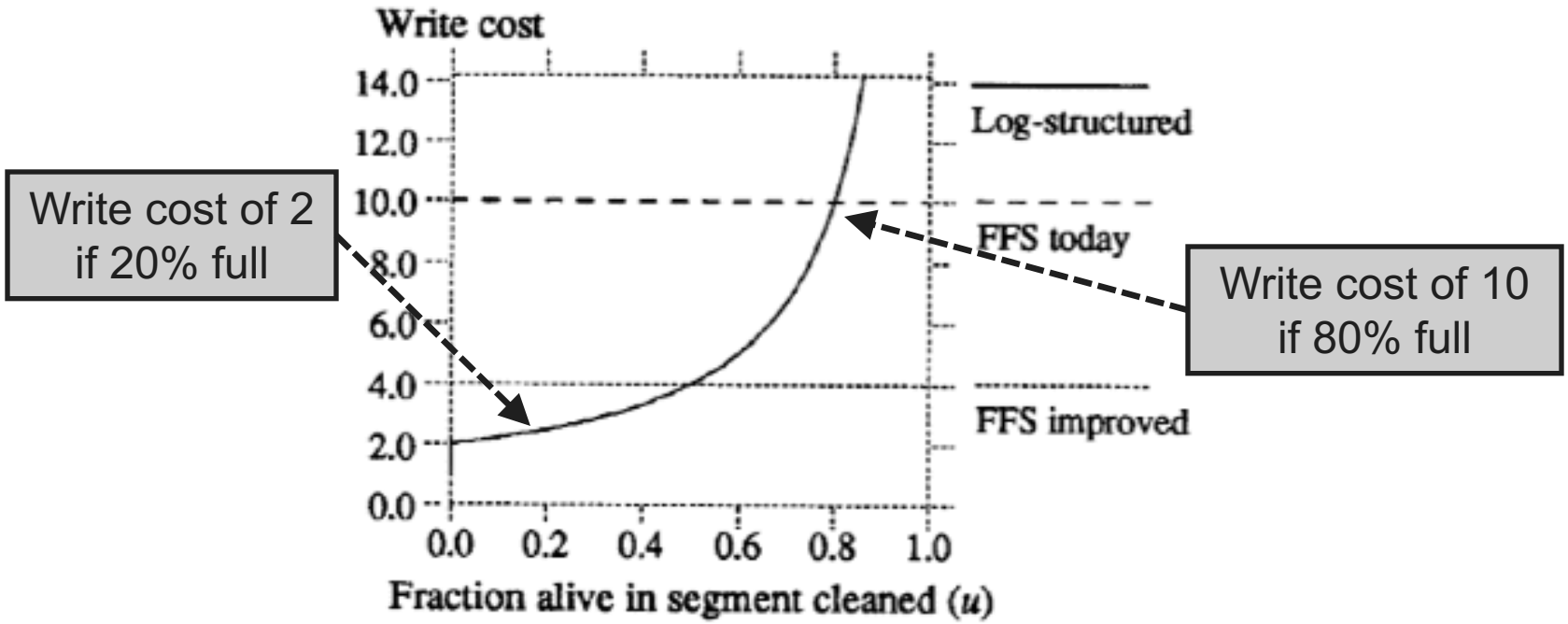
The inode map

- New data structure: inode map (indirection!)
  - Directory entries contain inode number
  - inode map translates inode number to disk block
- Where should inode map be stored?
  - Fixed location
- Problems with storing inode map in fixed location on disk?
  - All writes are not perfectly sequential

LFS: Other challenges

- LFS append-only quickly runs out of disk space
  - Need to recover deleted/overwritten blocks
- Need aggressive defragmentation of the disk
  - The whole point of LFS is to have large contiguous areas where you can write sequentially
- Cleaning is expensive if high disk space utilization

Write Cost Comparison

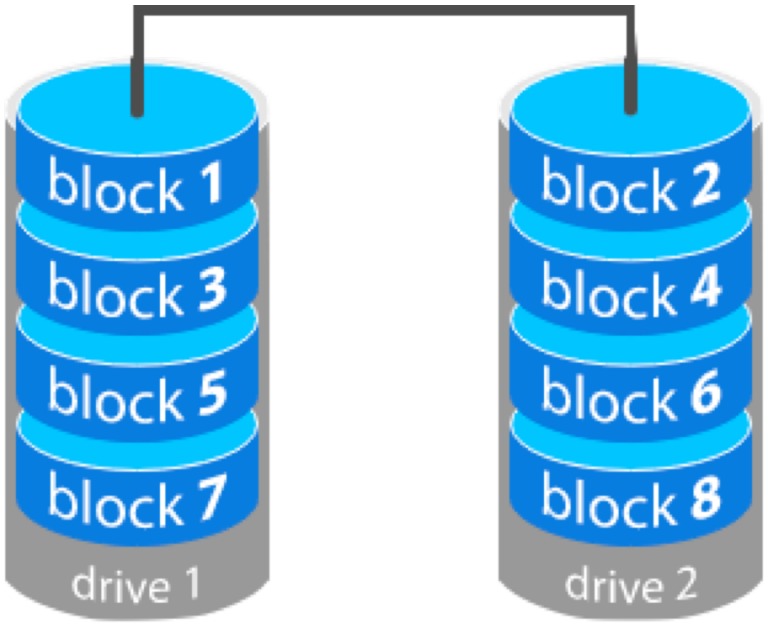


RAID

- Redundant Array of Inexpensive Disks (RAID)
  - Sits in between hardware and the file system

- Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
  - Files are striped across disks
  - Each stripe portion is read/written in parallel
  - Bandwidth increases with more disks

RAID-0: Striping

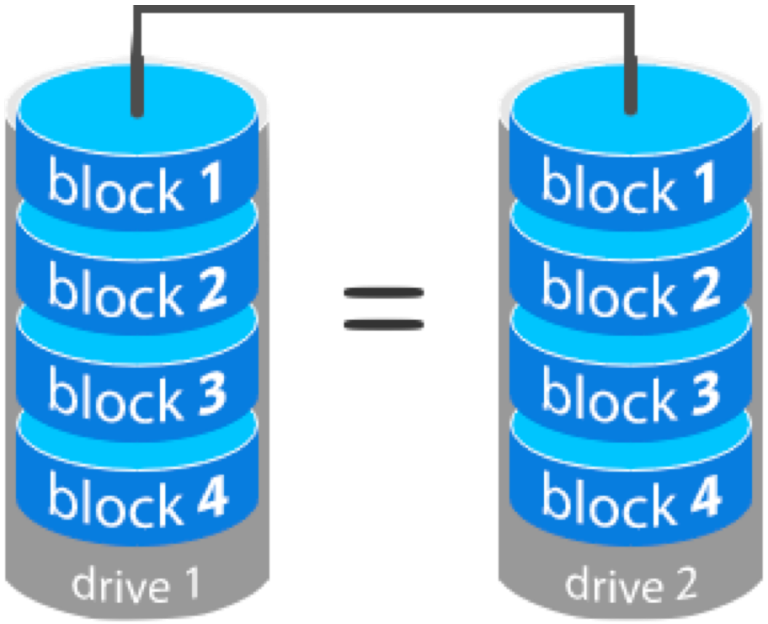


RAID Challenges

- Small writes (less than a full stripe) don't benefit from striping

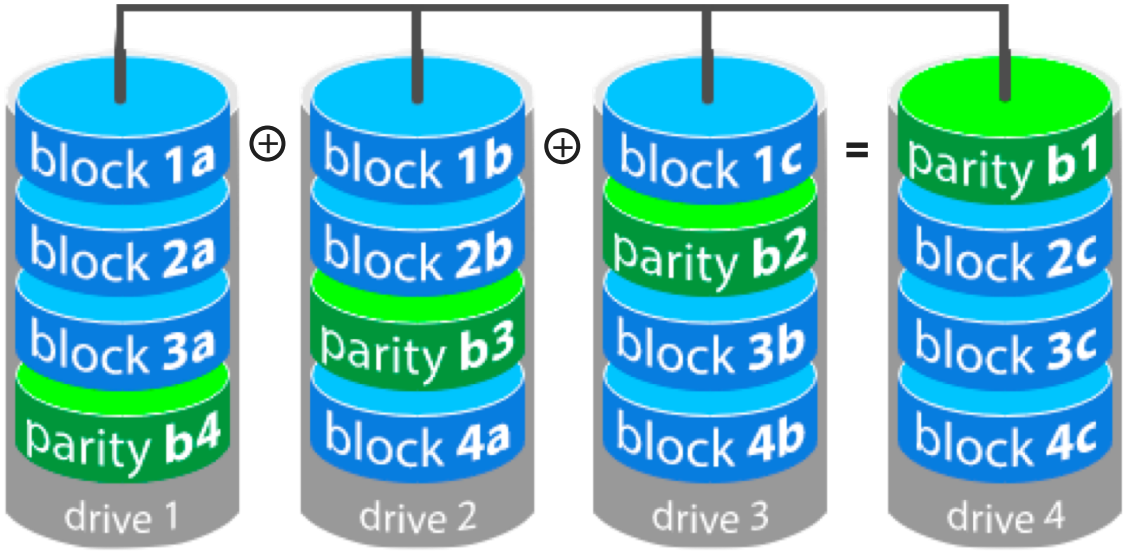
- Reliability
  - More disks increase chance of failure (MTBF)
  - Example:
    - » Say 1 disk has 10% chance of failing in one year
    - » With 10 disks, chance of any 1 disk failing in one year is $1 - (1 - 0.1)^{10} = 65\%$!

RAID-1: Mirroring



RAID with parity

- Improve reliability by storing redundant parity
  - In each stripe, use one block to store parity data
    - » XOR of all data blocks in stripe



  - Can recover any data block from all others + parity

RAID Levels

- RAID 0: Striping
  - Good performance but no reliability
- RAID 1: Mirroring
  - Maintain full copy of all data
  - Good read performance, but 100% storage overhead and no benefit to writes

RAID Levels

- RAID 5: Floating parity
  - Introduces some overhead, but disks are "inexpensive"
  - Parity blocks for different stripes written to different disks
  - No single parity disk → no bottleneck at that disk
  - Not ideal for small writes (less than a full stripe)
    - » Need to read entire stripe, update with small write, then write entire stripe out to disks