CS 477:

Advanced Operating Systems

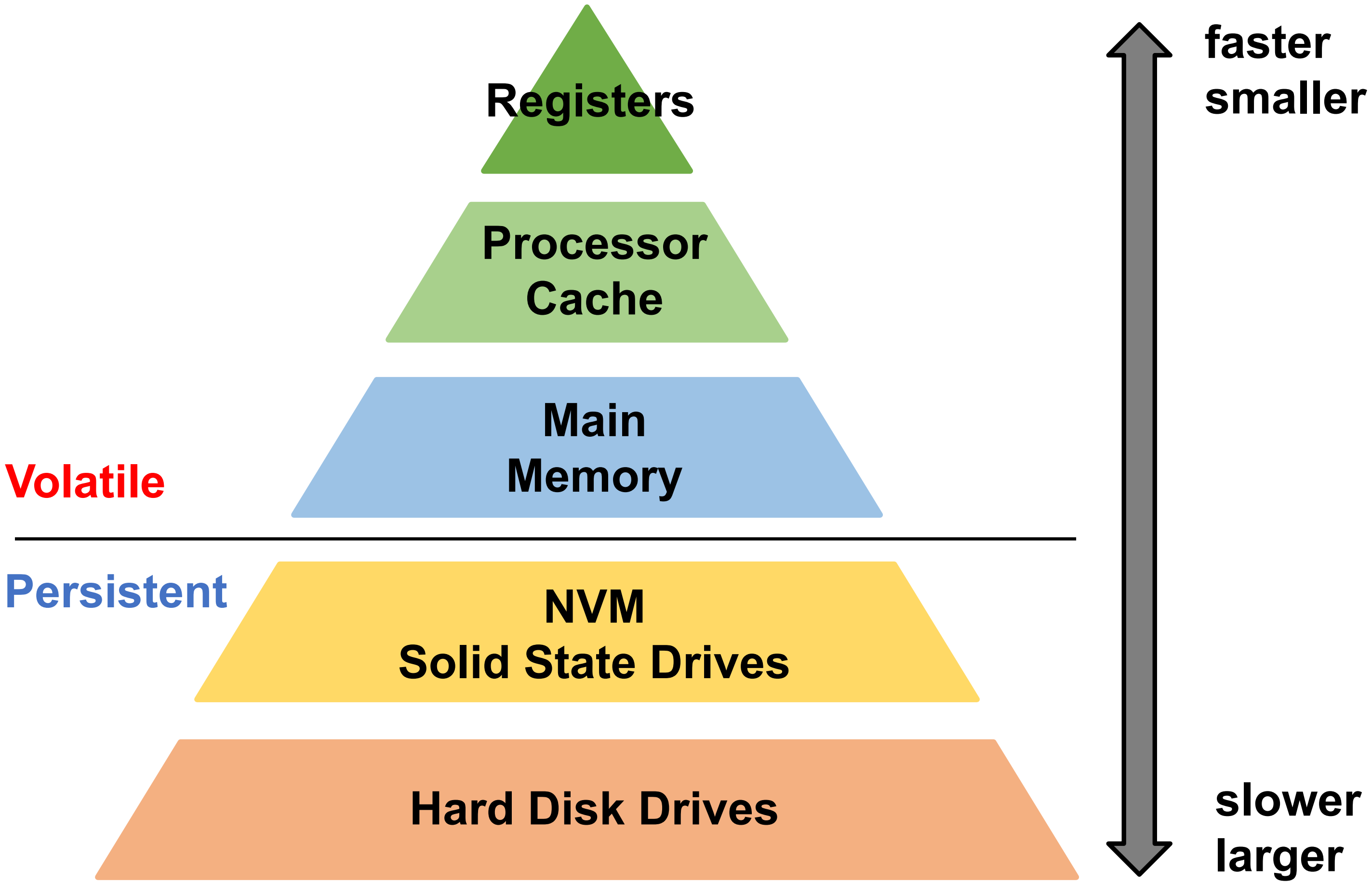
Hard Disk & Log-structured file system

Administrivia

This week

- Storage Device
- File System
- Log-Structured File Systems

Memory hierarchy and storage devices



Hard-Disk Drives

- Storage devices has been the performance bottleneck for years

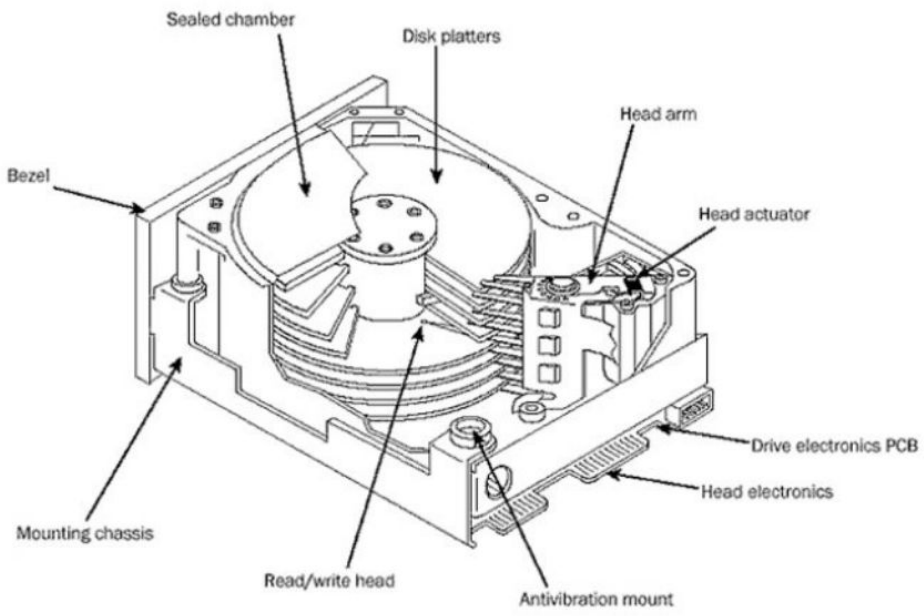
Processor:	Calculations per second	7.5
Memory (RAM):	Memory operations per second	7.5
Graphics:	Desktop performance for Windows Aero	7.7
Gaming graphics:	3D business and gaming graphics performance	7.7
Primary hard disk:	Disk data transfer rate	5.9

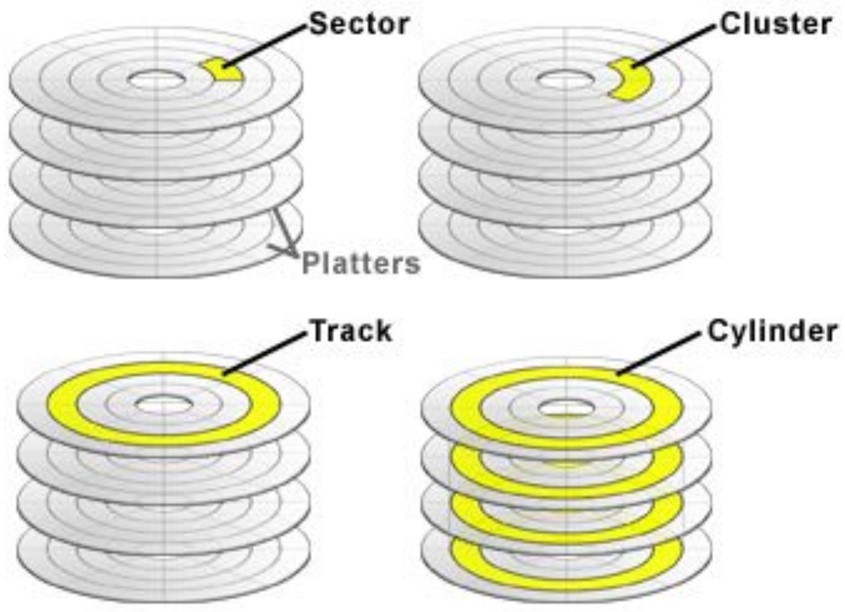
5.9

Determined by lowest subscore

Hard-Disk Drives

- Disk is slow because of mechanical parts inside

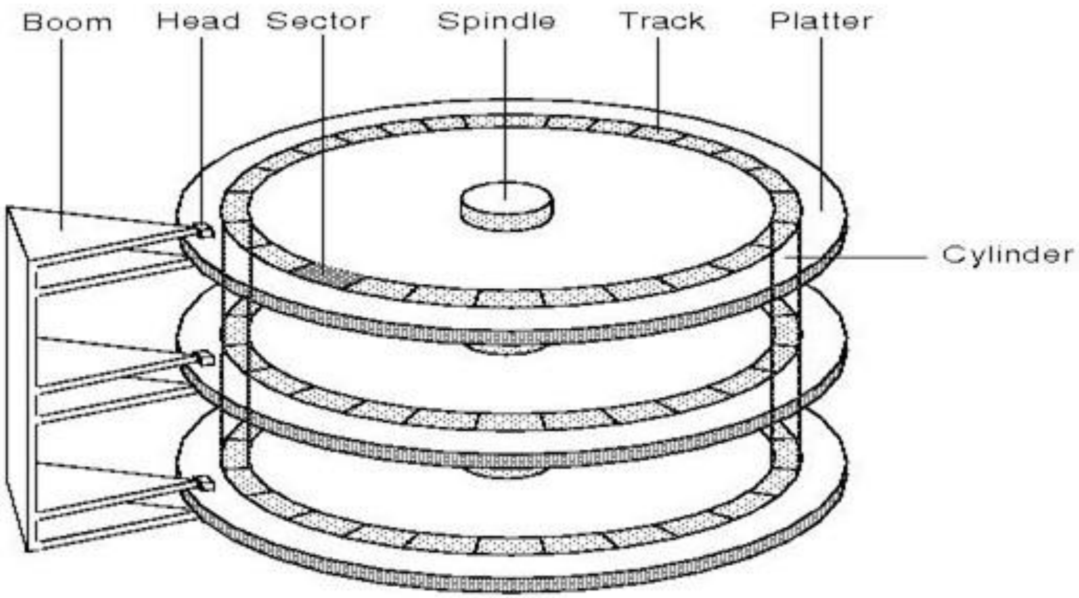




- Random accesses may incur disk head movement

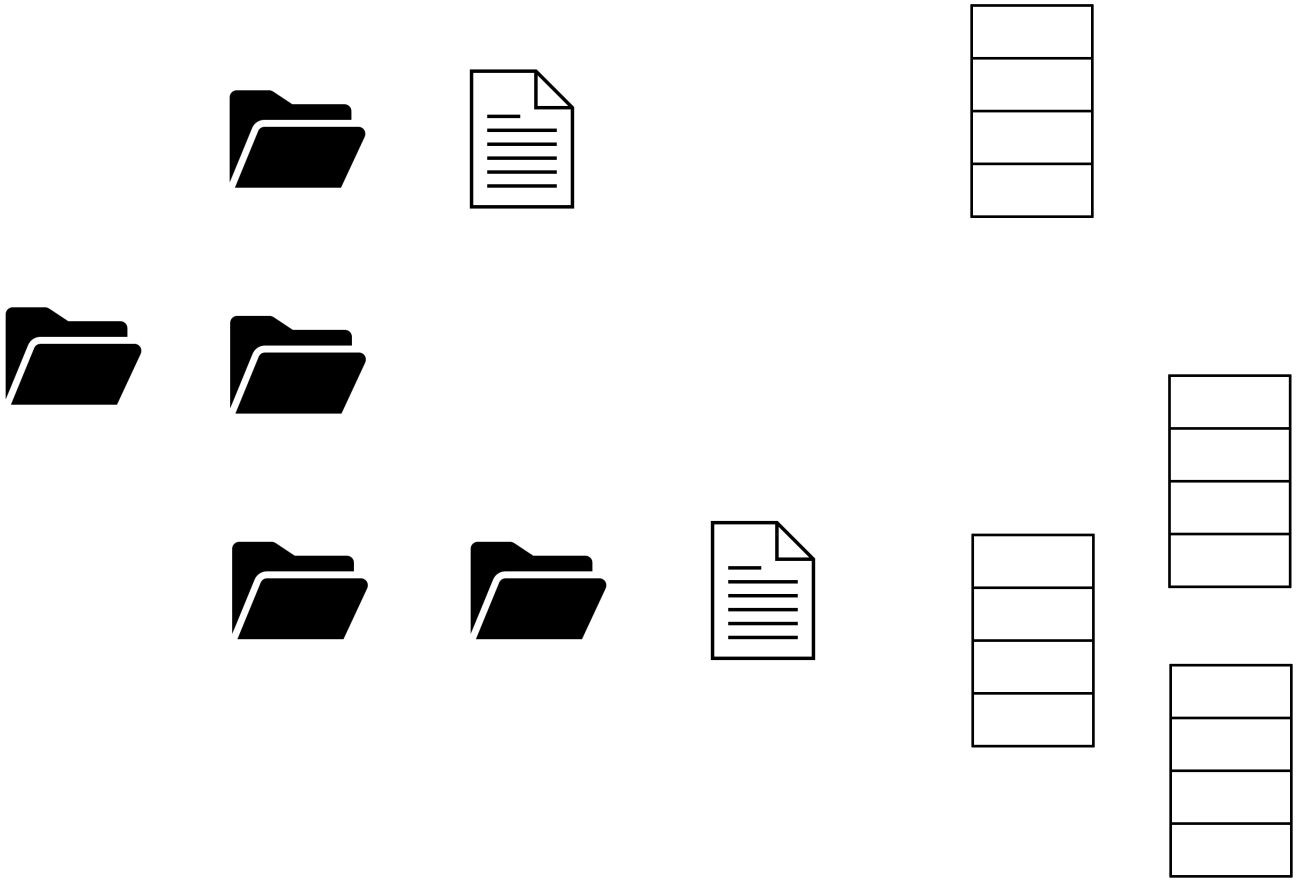
How to organize data stored on storage devices

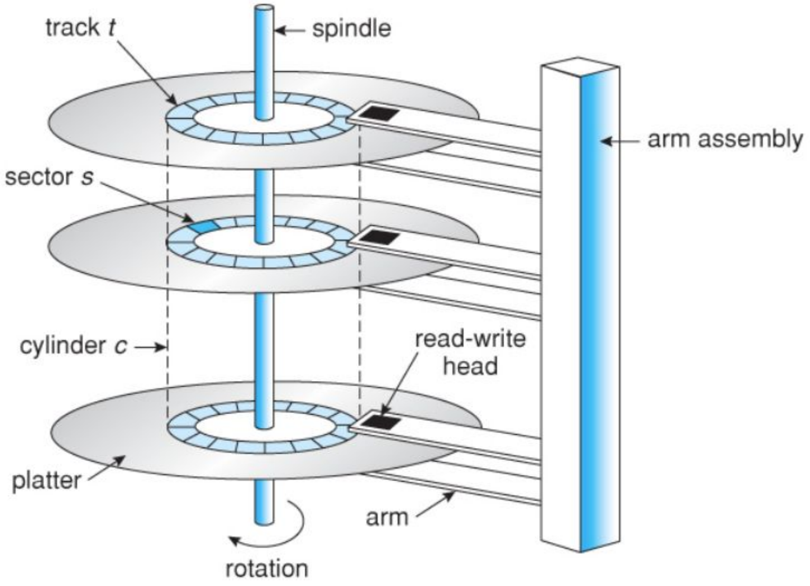
- Block abstraction
  - Split the storage device into fixed-sized chunks (**blocks**)
  - Number the blocks for access
  - Mapping the block number to the cylinder, header, sector



How to organize data stored on storage devices

- File abstraction





File system abstraction

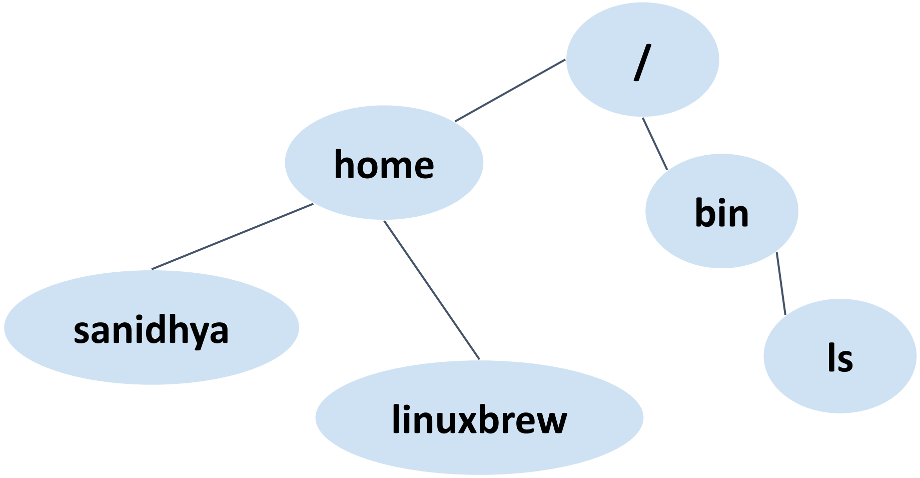
- Addresses need for long-term information storage:
  - Store large amounts of information
  - Do it in a way that outlives the program
  - Can support concurrent accesses from multiple processes
- Presents applications with **persistent**, **named** data
- Two main components:
  - **Files**
  - **Directories**

The file abstraction: File

- A file is **named** collection of related information that is recorded in secondary storage
- Or, a linear persistent array of bytes
- Has two parts:
  - **Data**: what a user or application puts in it
    - Array of bytes
  - **Metadata**: Information added and managed by the OS
    - Size, owner, security information, modification time, etc.

The file abstraction: Directory

- A special file that stores the mapping between human-friendly names of files and their inode numbers
- Contains subdirectories:
  - List of directories, files
  - / indicates the root;



$ tree /home

```
/home
├── linuxbrew
├── lost+found  [error opening dir]
└── sanidhya
    ├── 2023
    │   ├── 05-mvcc.pdf
    │   ├── 1000coresfullstory.pdf
    │   ├── 13-kernels.pptx
    │   ├── 14360135-1080p.mp4.crdownload
    │   ├── 17.pptx
    │   ├── 1 eBPF Instruction Set Specification, v1.0 — The Linux Kernel documentation.pdf
    │   ├── 2013_programming_massively_parallel_processors_a_hands-on_approach_2nd.pdf
```

The file abstraction: Directory

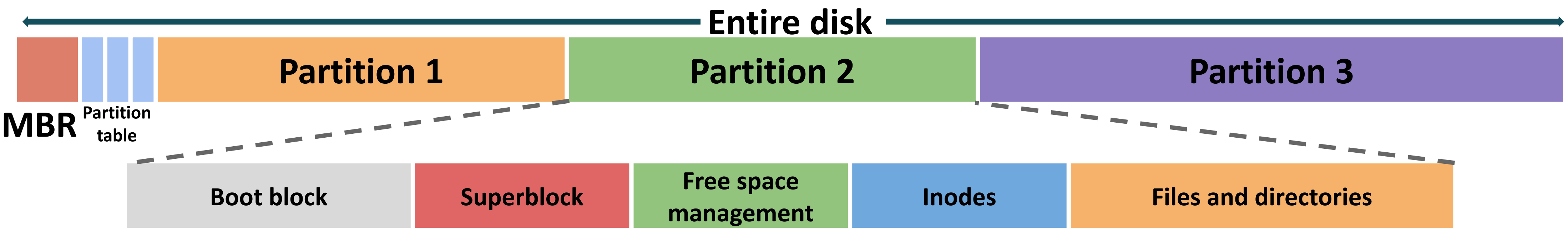
- **Links** are file pointers, i.e., they do not contain data themselves but a reference to another file
  - **Hardlinks:** Maps a file's path to the file's inode number
    - Mirror copy of the original file
    - Same inode number as that of the original file
  - **Symbolic (soft) link:** Logically maps a file's path to a different file path
    - Actual link to the original file
    - New inode number allocated on using soft link

File system implementation

- File system manages data for users
- **Given:** a large set (N) of blocks
- **Need:** data structures to encode file hierarchy and per file metadata
  - Overhead (metadata vs file data size) should be low
  - Internal fragmentation should be low
  - Efficient access of file contents: external fragmentation, # metadata access
  - Implement file system APIs
- Several choices are available (similar to virtual memory)

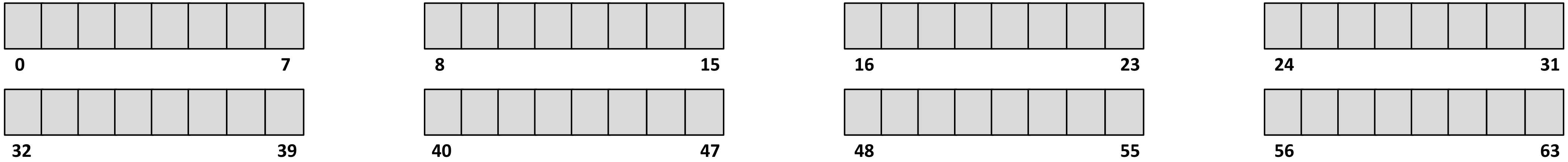
File system layout

- File system is stored on disks
  - Disk can be divided into one or more partitions
  - Sector 0 of disk: master boot record (MBR), which contains:
    - Bootstrap code (loaded and executed by the firmware)
    - Partition table (addresses of where partition start and end)
  - First block of each partition has a boot block
    - Loaded by executing code in MBR and executed on boot



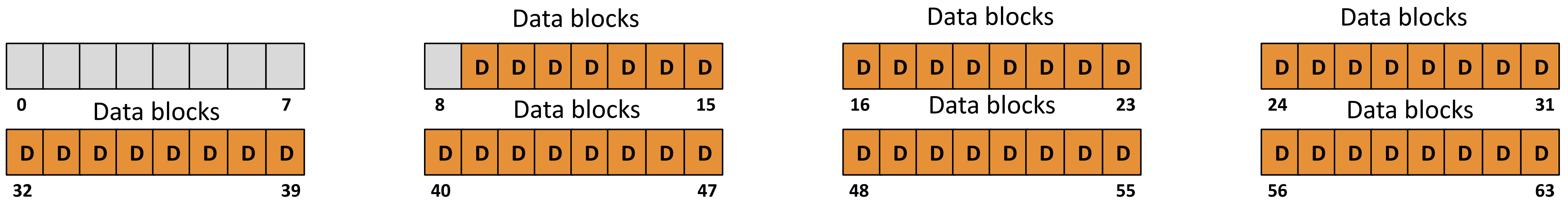
Peeking inside a partition (storage block)

- Persistent storage modeled as a sequence of N blocks
  - From 0 to N-1: 64 blocks, each of 4KB
  - Some blocks store data



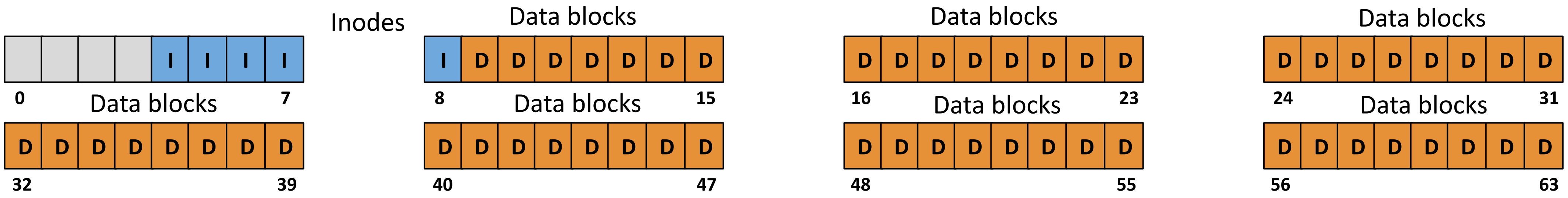
Peeking inside a partition (storage block)

- Persistent storage modeled as a sequence of N blocks
  - From 0 to N-1: 64 blocks, each of 4KB
  - Some blocks store data



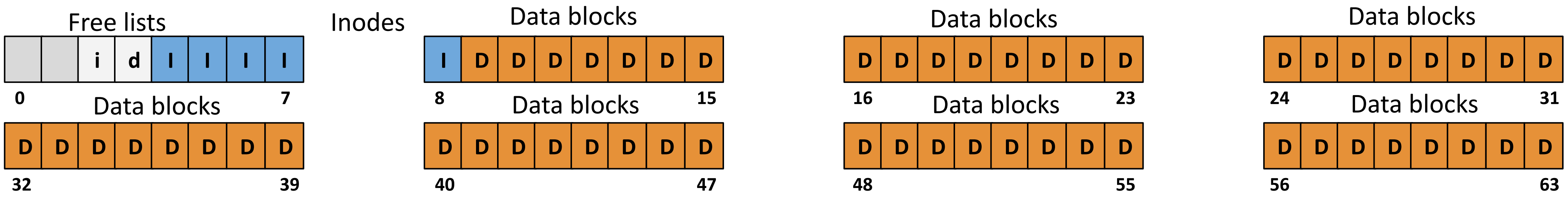
Peeking inside a partition (storage block)

- Persistent storage modeled as a sequence of N blocks
  - From 0 to N-1: 64 blocks, each of 4KB
  - Some blocks store data
  - Other blocks store metadata:
    - An array of inodes
      - At 256 bytes, 16 per block: with 5 blocks for inodes, file system can have up to 80 files



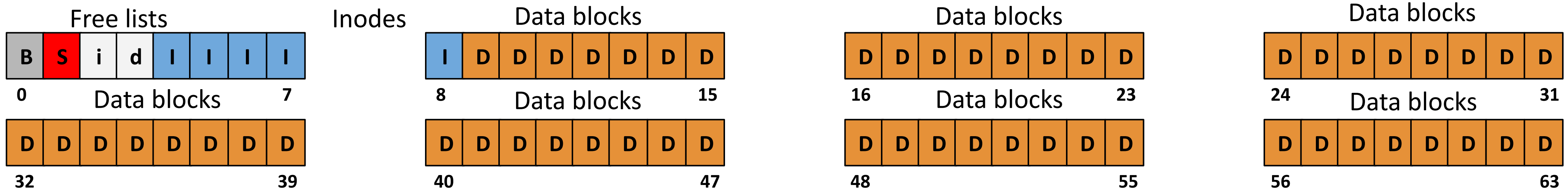
Peeking inside a partition (storage block)

- Persistent storage modeled as a sequence of N blocks
  - From 0 to N-1: 64 blocks, each of 4KB
  - Some blocks store data
  - Other blocks store metadata:
    - An array of inodes
      - At 256 bytes, 16 per block: with 5 blocks for inodes, file system can have up to 80 files
    - Bitmap tracking free inodes and data blocks (free lists)



Peeking inside a partition (storage block)

- Persistent storage modeled as a sequence of N blocks
  - From 0 to N-1: 64 blocks, each of 4KB
  - Some blocks store data
  - Other blocks store metadata:
    - An array of inodes
      - At 256 bytes, 16 per block: with 5 blocks for inodes, file system can have up to 80 files
    - Bitmap tracking free inodes and data blocks (free lists)
    - Boot block and superblock are at the beginning of the partition



Example: Updating a block in a file system

Inode bitmap

0	1	0	0	0	0

data bitmap

0	0	0	0	1	0

Inode information   data blocks

```
-- Iv1 --  --  --  -- | -- -- -- -- D1 --
```

owner:	sanidhya
permission:	read-write
size:	1
pointer:	4
pointer:	NULL
pointer:	NULL

- Suppose we append a data block to a file
- Add new data block **D2**

Example: Updating a block in a file system

Inode bitmap

0	1	0	0	0	0

data bitmap

0	0	0	0	1	0

Inode information data blocks

`-- Iv1 -- -- -- -- | -- -- -- -- D1 D2`

owner:	sanidhya
permission:	read-write
size:	1
pointer:	4
pointer:	NULL
pointer:	NULL

- Suppose we append a data block to a file
- Add new data block **D2**
- Update inode

Example: Updating a block in a file system

Inode bitmap

0	1	0	0	0	0

data bitmap

0	0	0	0	1	0

Inode information   data blocks

```
-- Iv2 --  --  --  -- | -- -- -- -- D1 D2
```

owner:	sanidhya
permission:	read-write
size:	**2**
pointer:	4
pointer:	**5**
pointer:	NULL

- Suppose we append a data block to a file
- Add new data block **D2**
- Update inode
- Update data bitmap

Example: Updating a block in a file system

Inode bitmap

0	1	0	0	0	0

data bitmap

0	0	0	0	1	1

Inode information   data blocks

```
-- Iv2 --  --  --  -- | -- -- -- -- D1 D2
```

owner:	sanidhya
permission:	read-write
size:	2
pointer:	4
pointer:	5
pointer:	NULL

- Suppose we append a data block to a file
- Add new data block **D2**
- Update inode
- Update data bitmap

What if a crash or power outage occurs between writes?

If only a single write is written to disk

Inode bitmap

0	1	0	0	0	0

data bitmap

0	0	0	0	1	1

Inode information data blocks

`-- Iv2 -- -- -- -- | -- -- -- -- D1 D2`

- Data block (D2) is written to disk:
  - Data is written; **no way** to get to it: D2 still appears as a **free block**
  - Write is lost, but FS (meta)data structures are consistent
- Just the inode (`Iv2`) is written to disk:
  - On following the block pointer, garabase is read
  - **Inconsistent FS:** data bitmap says block is free, while inode says it is used
- Updated data bitmap is written to disk:
  - **Inconsistent FS:** data bitmap says data block is used, but no inode points to it

If two writes are written to disk

Inode bitmap

0	1	0	0	0	0

data bitmap

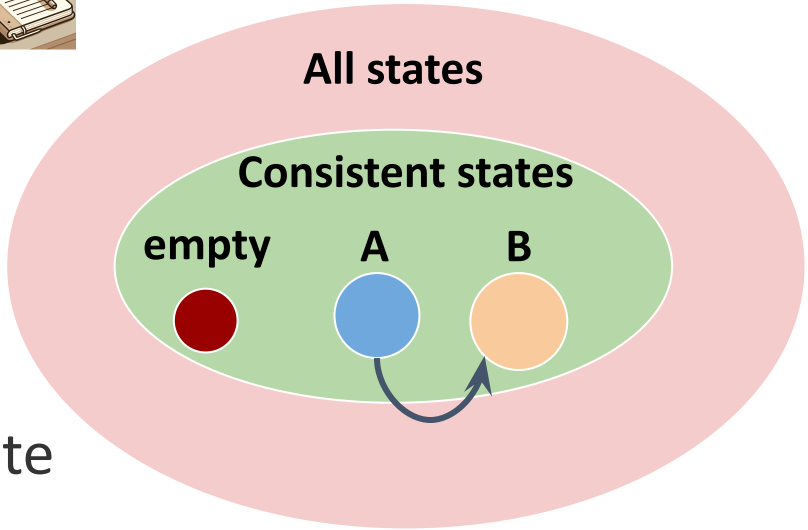
0	0	0	0	1	1

Inode information data blocks

```
-- Iv2 --  --  --  -- | -- -- -- -- D1 D2
```

- Inode and data bitmap updates succeed
  - **Good news:** file system is consistent
  - **Bad news:** reading new block will return garbage
- Inode and data block updates succeed
  - **Inconsistent FS**
- Data bitmap and data block succeed
  - **Inconsistent FS**

Consistency solution: Journaling 



Goal: limit the amount of required work after crash

Goal: Get correct state, not just consistent state

Approach: Turns multiple disk updates into a single disk write

- "Write ahead" a short note to a "log", specifying changes about to be made to the FS data structures
- If a crash occurs while updating FS data structures, consult the log to determine what to do
  - No need to scan the entire disk

More on journaling 

- A logbook to maintain significant events, decision, or change in course
  - Historical record of the journey
  - Reference in case if something goes wrong or unexpected situations arise
- Recording events on logbook → journaling in file systems
  - Before making any changes to the data (read/write/delete etc.), record the intended changes in special area called the “**journal**”
- Journal is a **special area** on the disk that stores data in a **write-ahead fashion**, also called **write-ahead logging**
- Journaling uses **transactions’ atomicity** to provide crash consistency

Data journaling: an example

Inode bitmap

0	1	0	0	0	0

data bitmap

0	0	0	0	1	0

Inode information data blocks

`-- Iv1 -- -- -- -- | -- -- -- -- D1 --`

- Let’s add a new block **D2** to the file
- Three easy steps:
  - Write to the **log**, these 5 blocks: **`TxBeg | Iv2 | Bv2 | D2 | TxEnd`**
    - Write each record to a block for ensuring atomicity
  - Write the blocks **`Iv2`**, **Bv2**, **D2** to the file system structure place (**checkpoint**)
    - Bv2 is bitmap information
  - Mark the transaction free in the journal (i.e., remove it)

Data journaling: an example

Inode bitmap

0	1	0	0	0	0

data bitmap

0	0	0	0	1	0

Inode information data blocks

```
-- Iv1 --  --  --  -- | -- -- -- -- D1 --
```

- Let's add a new block **D2** to the file
- Three easy steps:
  - Write to the **log**, these 5 blocks: **`TxBeg | Iv2 | Bv2 | D2 | TxEnd`**
    - Write each record to a block for ensuring atomicity
  - Write the blocks **`Iv2`**, **`Bv2`**, **D2** to the file system structure place (**checkpoint**)
  - Mark the transaction free in the journal (i.e., remove it)
- **If crash happens before the log is updated: Ignore changes if no commit**
- **If crash happens after the log is updated: Replay changes in log back to the disk**

Improving file system performance for HDD

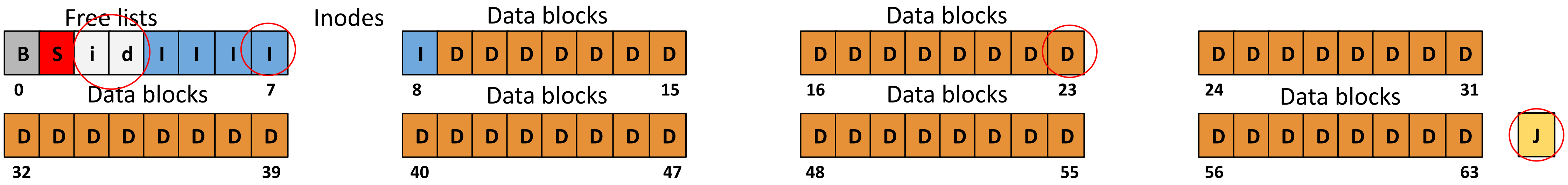
- Improving read performance
  - DRAM size is now larger and larger
  - Use DRAM to cache data from HDD to reduce disk access

- Improving write performance
  - Use DRAM to buffer data is fine, but need to sync them to HDD
  - Try batching data as much as possible to turn small random updates into large sequential writes

Nature of disk writes in traditional file systems

- Appending a block to an existing file
  - Update inode of that file
  - Update inode bitmap
  - Update data block bitmap
  - Update data blocks
  - Update journaling

So many random accesses might incur multiple seeks inside HDD!



Log-structured file system (LFS)

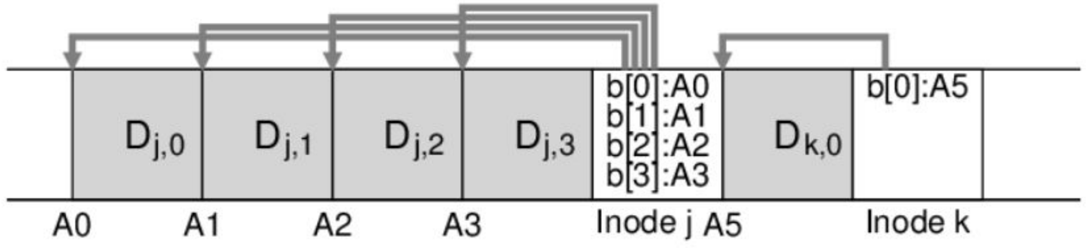
- Key idea:

  Instead of adding a log to the existing disk, use entire disk as a log

- Buffer all updates (including metadata) into an *in-memory segment*
- When a segment is full, write to the disk in a long sequential transfer to unused part of the disk
- Never overwrite existing data

Writing to disk seqentially

- Write data blocks and metadata blocks sequentially



- New version of the inode keeps track of the latest data blocks
- Buffer the logs in DRAM, and write to disk when the log has sufficient size

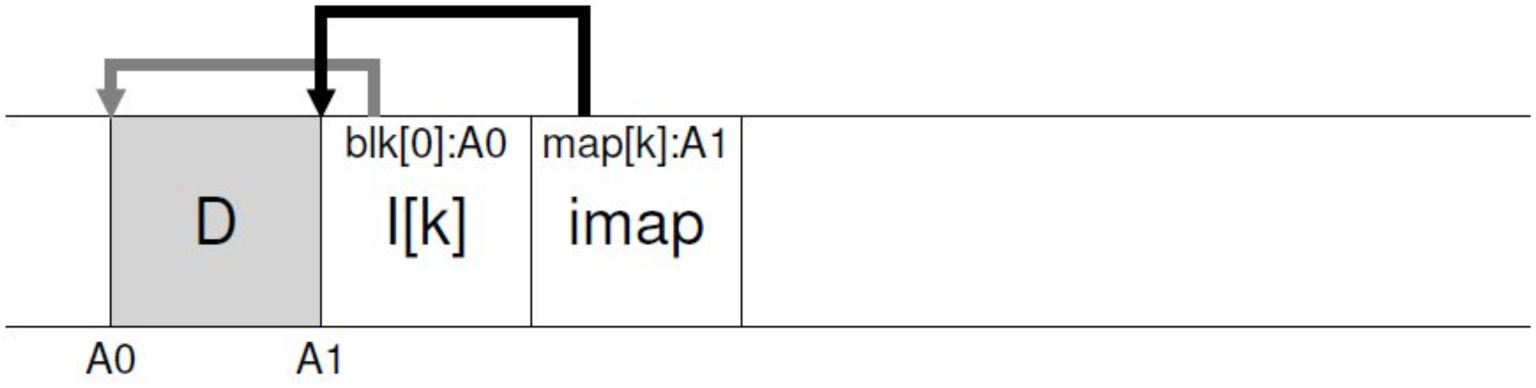
Finding inodes in LFS

- Inodes are scattered throughout the disk
  - No lingered kept in fixed region on the disk
  - No fixed patterns on the disk to keep track of them
- Each inode even has different versions
  - Each update to a file need a separate version of inode
  - Need to keep track of the latest version of each inode

How to keep track of the versions of the inodes?

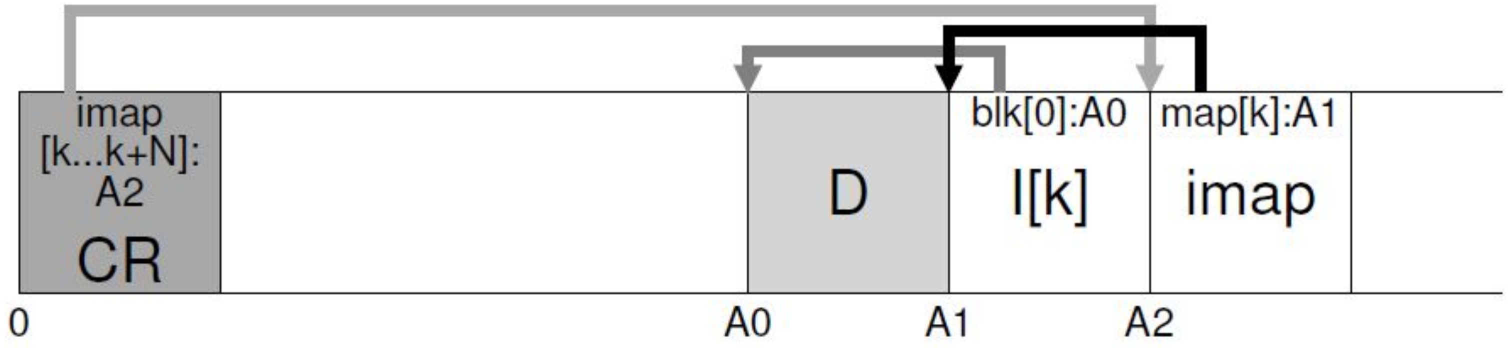
The inode map (imap)

- Book-keeping the latest version of inodes on the disk
  - Takes an inode number as input and produces the disk address of the most recent version of the inode
  - The imaps resides together in the log with inode and data block updates



Finding imaps in LFS

- LFS has fixed region (Checkpoint Region) for imap lookup
  - Maintaining pointers to the latest of the inode map
  - Only updated periodically (For example, every tens of seconds)

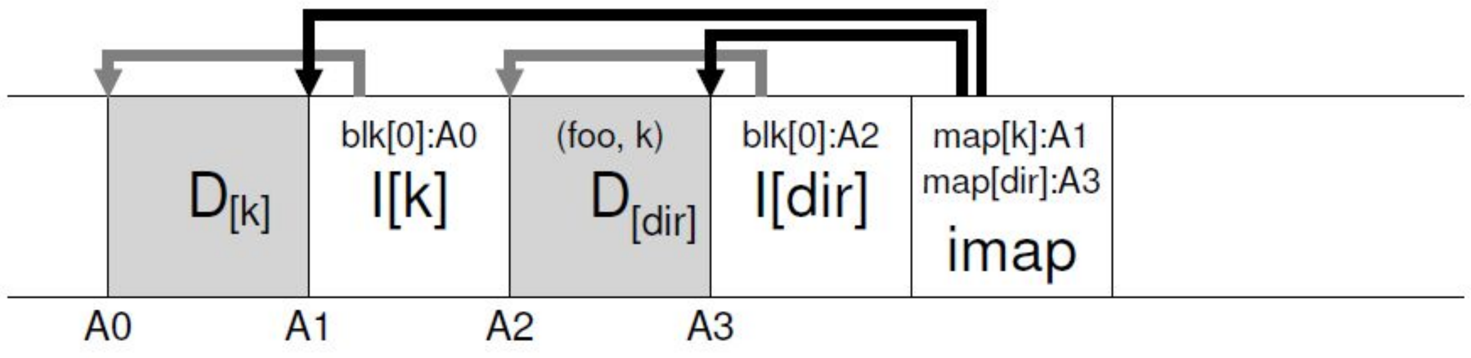


Reading a file from the disk

- Read from the checkpoint region to get imaps
  - Also cache imaps into DRAM
- Read the mode recent inode from the imap
- Read a block form the file with its direct or indirect pointers

What about directories

- Similar to traditional file systems
    - A directory is a file whose data blocks contains file/subdirectory names and their inode numbers as pairs

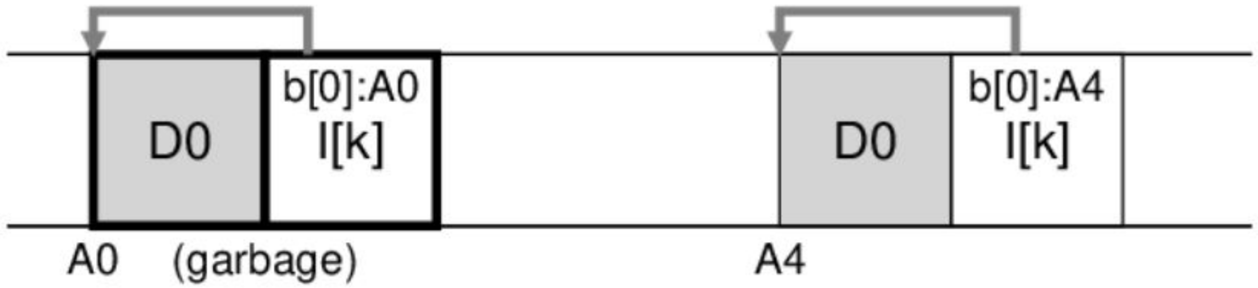


Garbage collection

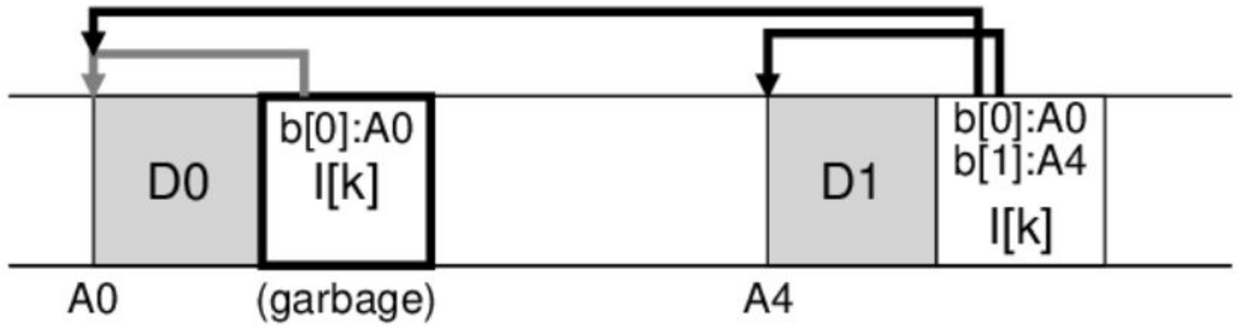
- LFS keeps writing newer version of file and inode to new locations
  - Leaving the old versions of both data blocks, inodes, and imaps all over the places (garbage)
- Need to identify the obsolete data blocks, inodes, and imaps
  - Keep the latest live version and periodically clean old versions

Garbage collection

- Case 1: Overwrite a data block



- Case 2: Append a new block to an existing file



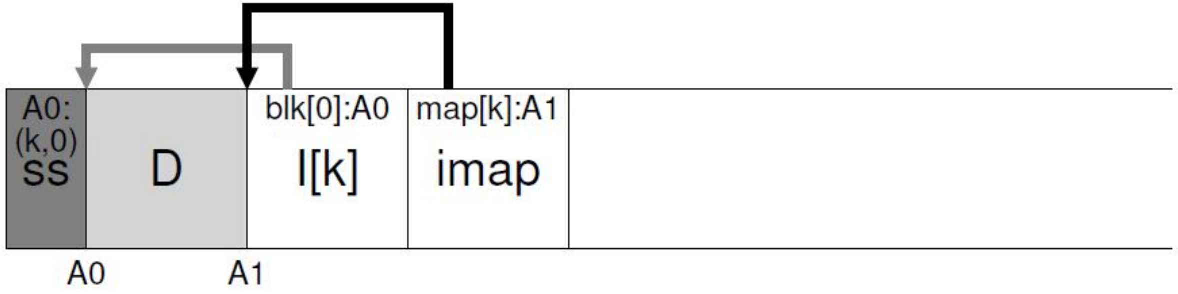
Garbage collection

- How to collect the garbage and free up space?
  - Only keep the latest live versions and periodically clean old versions
  - Perform garbage collection in a segment-by-segment basis

- Possibility of extending LFS to support **snapshot** and **versioning**
  - Keep some old verions of inode, imap, and data blocks
  - Rollback to previous version of file by writing new imaps pointing to previous versions of the inode

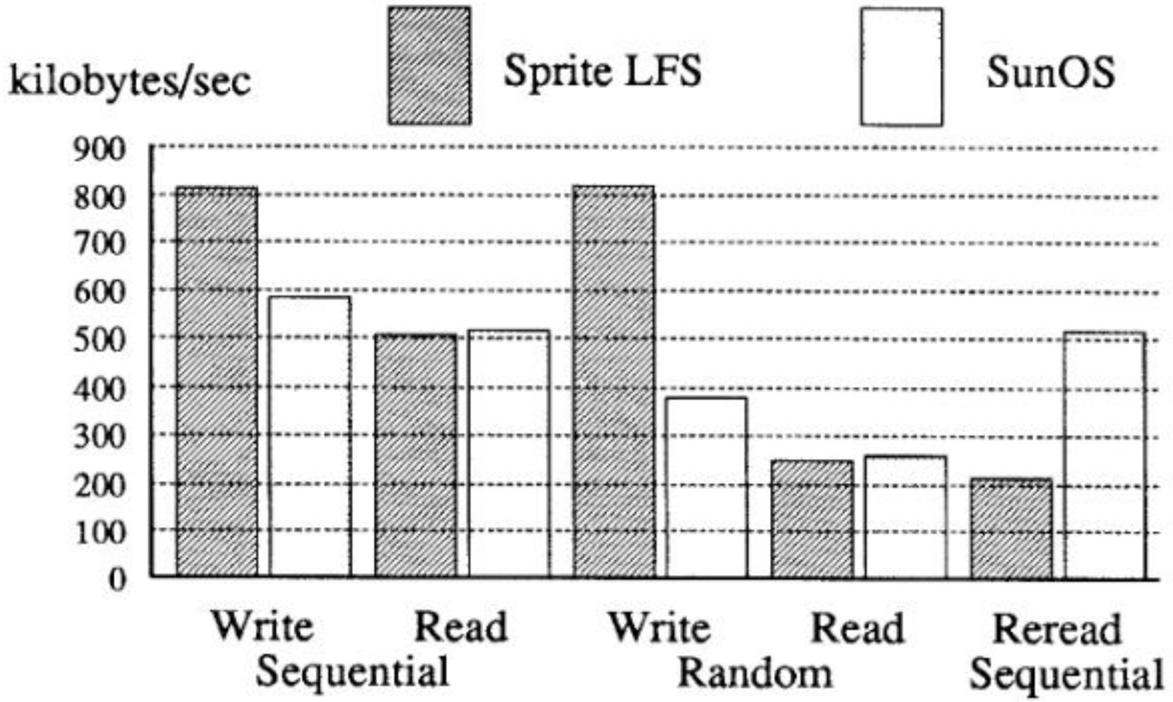
Garbage collection

- Determine which blocks are alive and which are dead
  - A piece of per-segment metadata called Segment summary block (SS)
  - Inode number and offset for each data block are recorded




```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

Evaluation


kilobytes/sec
Sprite LFS
SunOS
900
800
700
600
500
400
300
200
100
0
Write
Read
Sequential
Write
Read
Random
Reread
Sequential

LFS – Summary

- DRAM size is big enough to cache data on disk
- Write performance dominates file systems’ performance
- Put file systems as a big “log”, making writes in a sequential manner
- Imaps, checkpoint regions keeps track of the latest versions of the file system
- Garbage collection periodically to free up space on the disk