File System Implementation

Operating Systems

Moshe Sulamy

Tel-Aviv Academic College

Very Simple File System

- File system: pure software
  - Many different file systems exist
- Start with a cast study: **vsfs**
  - Simplified version of typical UNIX file system

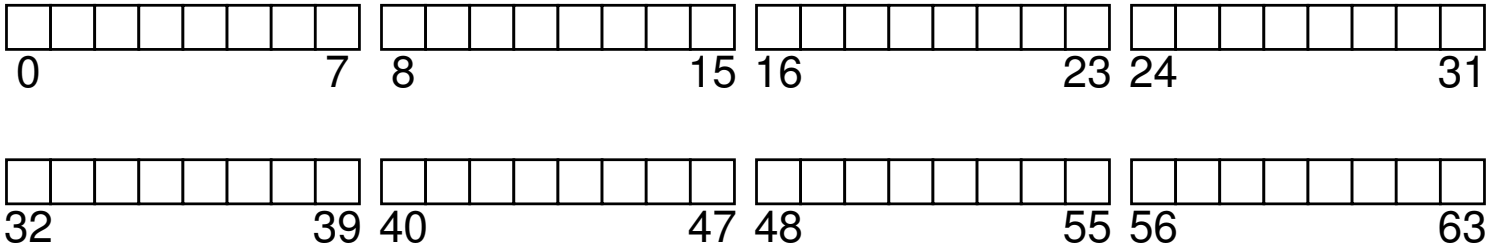
How can we build a simple file system?

Two Aspects

- **Data structures**
  - What type of on-disk structures?
- **Access methods**
  - How to map calls (`open()`, `read()`, `write()`, etc.)?
  - Read which structures during which calls?

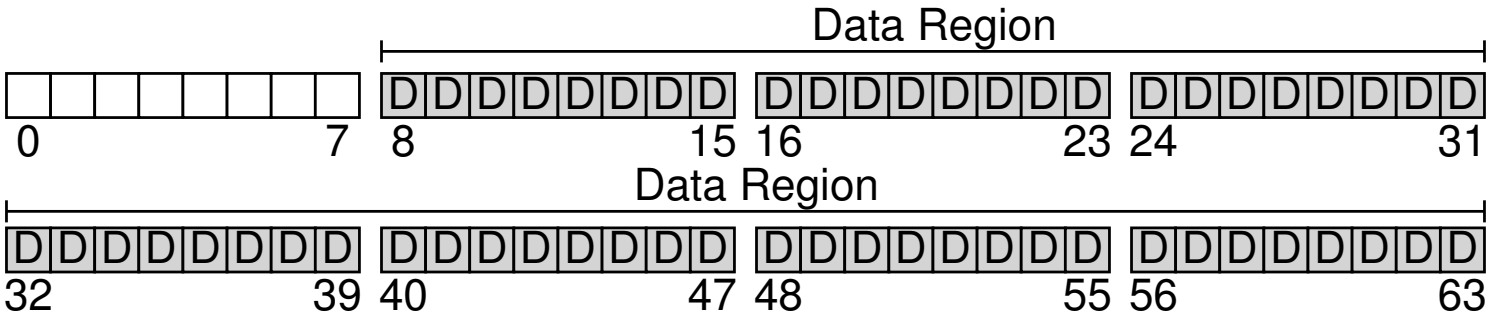
Overall Organization

- Divide disk into **blocks**
  - Addressed 0 to $N - 1$
  - Commonly-used size: 4 KB



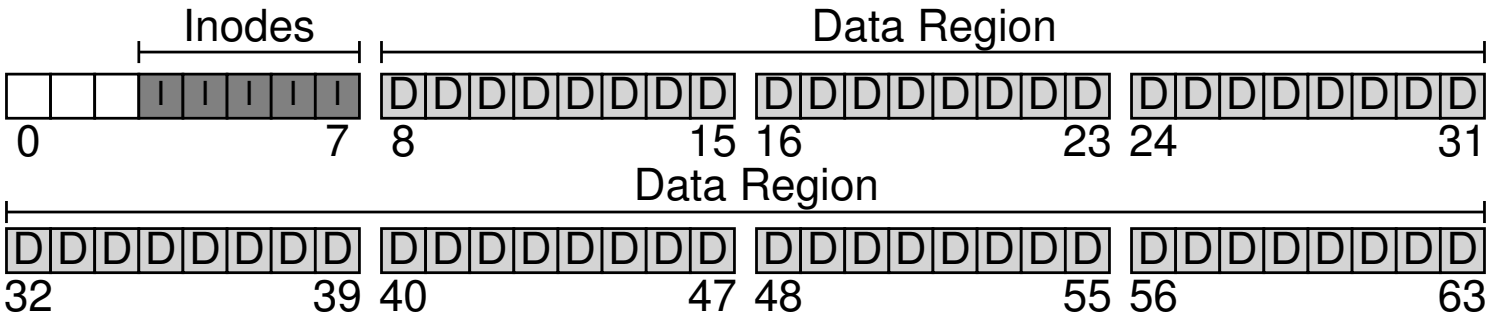
Overall Organization

- Reserve **data region** for user data
  - e.g., fixed portion: 54 of 64 blocks



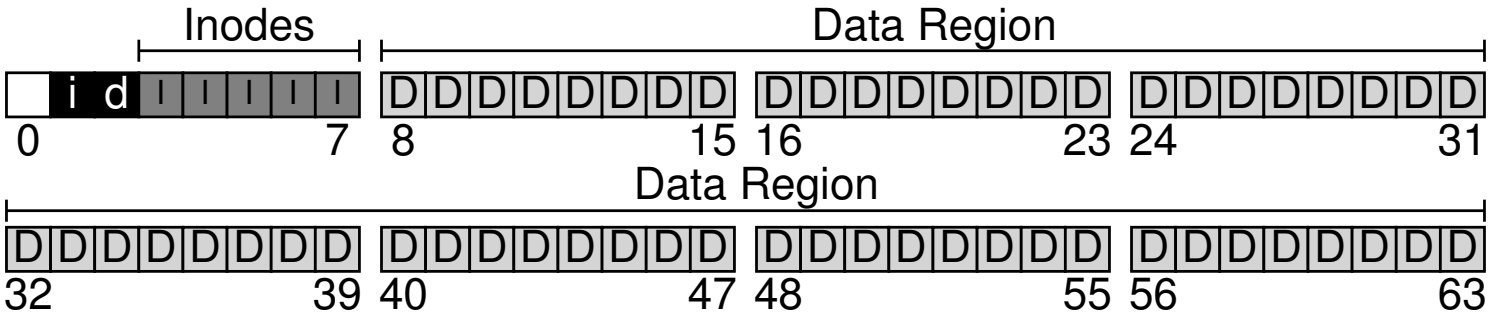
Overall Organization

- **Metadata** on file
  - Data blocks, size, owner, access writes, times, etc.
  - Usually in **inode** structure
  - ∼256 bytes per inode



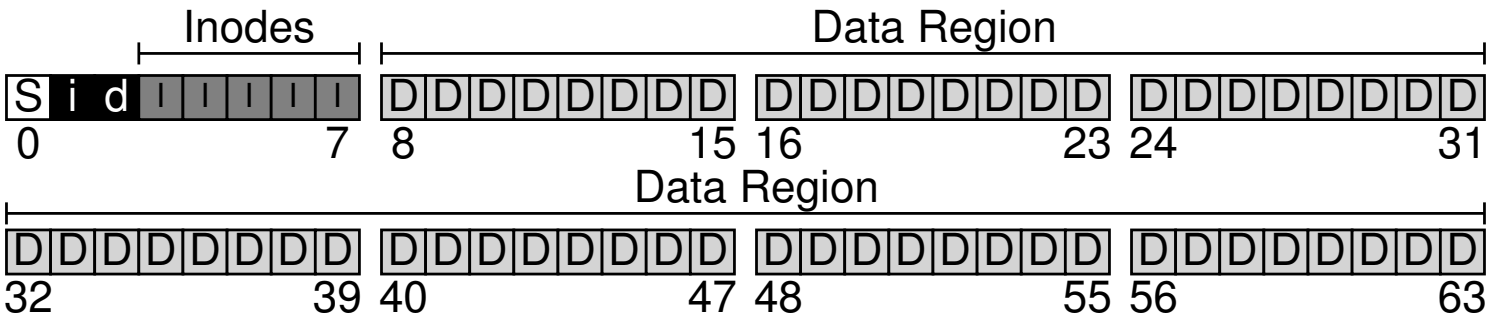
Overall Organization

- **Allocation structure**
  - Track whether inodes or data blocks are free or allocated
  - **Free list**
  - **Bitmap**
    - One for data region and one for inode table
    - Each bit indicates free (0) or in-use (1)



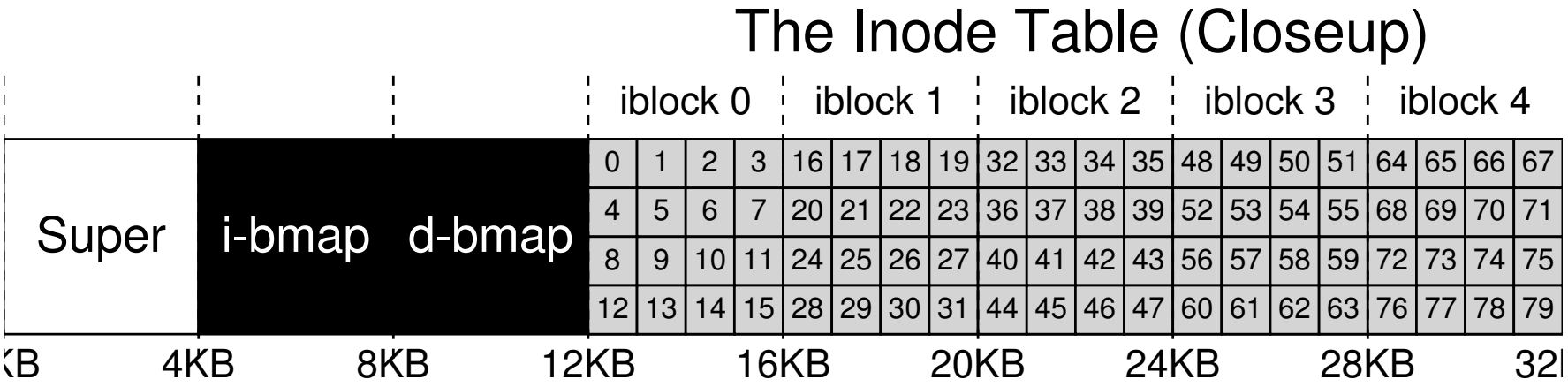
Overall Organization

- Remaining block: **superblock**
  - Information about this file system
  - e.g., number of inodes and data blocks, where inode table begins, magic number to identify file system type, etc.
- On mount, OS reads superblock first to initialize



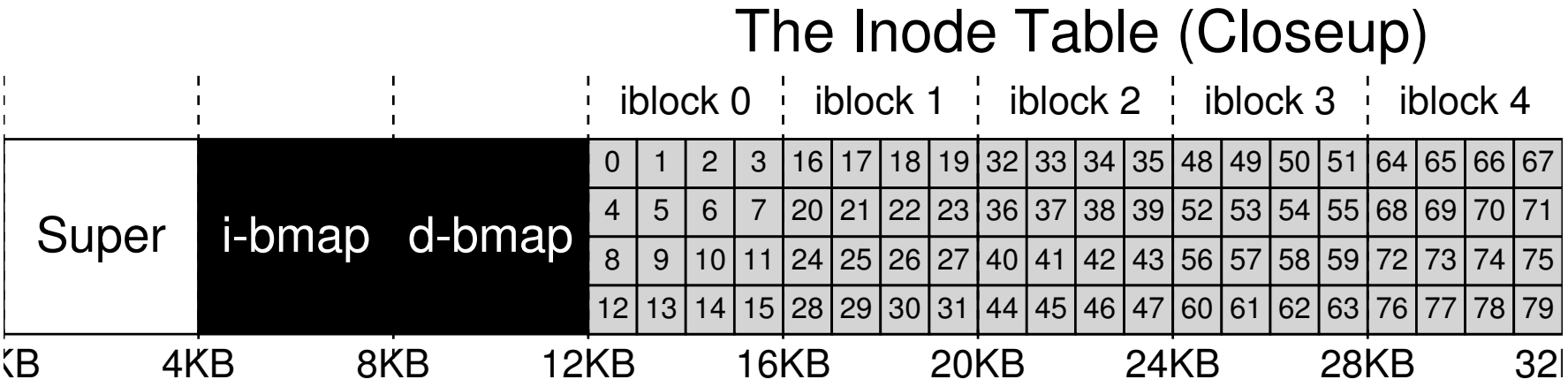
The Inode

- **Index node**
  - Referred to by a number: **i-number** (**low-level name**)
  - Example: 20KB inode table → 80 inodes
  - Read inode 32:



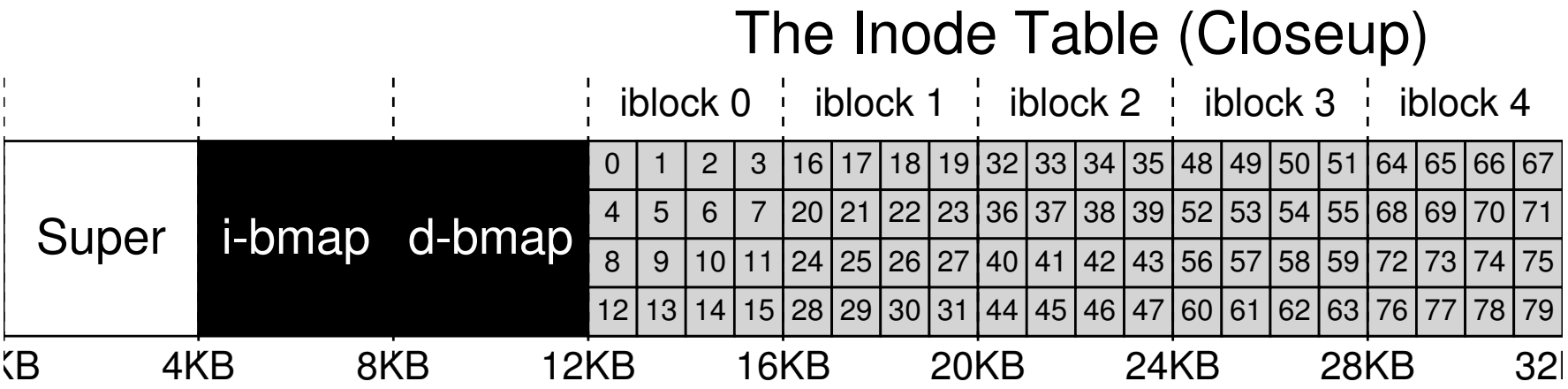
The Inode

- **Index node**
  - Referred to by a number: **i-number** (**low-level name**)
  - Example: 20KB inode table → 80 inodes
  - Read inode 32: $32 \cdot sizeof(inode) + startAddr = 8KB + 12KB$



The Inode

- **Index node**
  - Referred to by a number: **i-number** (**low-level name**)
  - Example: 20KB inode table $\rightarrow$ 80 inodes
  - Read inode 32: $32 \cdot sizeof(inode) + startAddr = 8KB + 12KB$
  - Read disk sector $\frac{20 \times 1024}{512} = 40$



The Inode

- Where are the data blocks?
  - One or more **direct pointers**
  - Each pointer refers to one disk block
  - Limited: no support for big files

Multi-Level Index

- **Indirect pointers**
  - Point to a block that contains more pointers
  - Each points to user data
  - Inode: fixed number of direct pointers, single indirect pointer
- File grows large: allocate indirect block
  - Point inode's indirect pointer to it

Multi-Level Index

- Multi-level index approach:
  - Double indirect pointer: points to block of indirect pointers
  - Triple indirect pointer: points to block of double indirect pointers
- Example:
  - Block size 4KB, 4-byte pointers
  - 12 direct pointers, both single and double indirect block

Multi-Level Index

- Multi-level index approach:
  - Double indirect pointer: points to block of indirect pointers
  - Triple indirect pointer: points to block of double indirect pointers
- Example:
  - Block size 4KB, 4-byte pointers
  - 12 direct pointers, both single and double indirect block
  - Can accommodate 4GB file $((12 + 1024 + 1024^2) \times 4KB)$

Multi-Level Index

- Many file systems use multi-level index
  - Linux ext2 and ext3, NetApp's WAFL, UNIX file system
  - SGI XFS and Linux ext4 use **extents**
- **Extents**
  - Disk pointer plus length
  - Avoids large metadata per file (pointer for every block)

Multi-Level Index

- Measurement summary:

Most files are small	∼2K most common size
Average file size growing	Almost 200K
Most bytes are stored in large files	Few big files use most of space
File systems contain lost of files	Almost 100K on average
File systems are roughly half full	Even as disks grow
Directories are typically small	Most have 20 or fewer entries

Linked-Based Approaches

- Use **linked list**
  - One pointer inside inode $\rightarrow$ first block of file
  - End of data block $\rightarrow$ another pointer
- Performs poorly for some allocations
  - e.g., read last block of file, random access
  - Solution? instead of next pointers, in-memory table of links
- Used by **FAT** (**file allocation table**) file system
  - Directory entries instead of inodes (hard links impossible)
  - Classic Windows file system before **NTFS**

Directory Organization

- Directory: list of (entry name, inode number) pairs
- Two extra files: "dot" & "dot-dot" for current and parent dirs
  - e.g., `dir` has three files:

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

Free Space Management

- File system must track free inodes and data blocks
- In vsfs: two bitmaps
  - New file: search through bitmap for free inode, allocate it
  - **Pre-allocation**: commonly used
    - Look for and allocate contiguous blocks for file

Access Paths

- Open a file (`/foo/bar`), read it, close it
  - Issue an `open("/foo/bar", O_RDONLY)`
    - **Traverse** pathname to locate desired inode
    - Begin at root: well-known, usually inode 2
    - Read block that contains inode 2
    - Look inside it - read data block to find inode number of `foo`
    - Read inode and data blocks of `foo` to find `bar`
  - Read from the file, repeat:
    - Read `bar` inode to find data block
    - Read data block
    - Write to inode - update access time
  - Close the file

Access Paths

- Writing to a file:
  - Open file (as before)
  - Each write generates five I/Os:
    1. Read data bitmap
    2. Write updated data bitmap (newly-allocated block to use)
    3. Read the inode
    4. Write updated inode with new block location
    5. Write the actual block itself

Access Paths

- Creating a file:
  - Read inode bitmap
  - Write updated inode bitmap with allocated inode
  - Write inode itself
  - Write data to directory containing the file
  - Read and write directory inode to update it
  - Directory needs to grow? Additional I/O
    - To data bitmap, new directory block

Caching and Buffering

- Simple operations: huge number of I/Os
  - e.g., long pathname can lead to hundreds of reads
  - Just to open a file!

What can a file system do to reduce the costs of many I/Os?

Caching and Buffering

- Simple operations: huge number of I/Os
  - e.g., long pathname can lead to hundreds of reads
  - Just to open a file!

What can a file system do to reduce the costs of many I/Os?

- Use system memory (DRAM) to cache important blocks
  - Early systems used fixed-size cache
    - Static partitioning of memory: can be wasteful
  - Modern systems use **dynamic partitioning**

Caching and Buffering

- Sufficiently large cache: avoid read I/O altogether
- Write traffic has to go to disk
  - Cache does not reduce write I/O

Caching and Buffering

- Sufficiently large cache: avoid read I/O altogether
- Write traffic has to go to disk
  - Cache does not reduce write I/O
- Use **write buffering**
  - Delay writes: **batch** updates to smaller set of I/Os (several updates to inode bitmap)
  - Buffer writes in memory, **schedule** subsequent I/Os
  - **Avoid** writes, e.g., file created and then deleted
- Use `fsync()` to force writes to disk

UNIX File System

- The old UNIX file system:

UNIX File System

- The old UNIX file system:
  - Superblock: volume size, number of inodes, pointer to head of free list of blocks, etc.
  - The inode region for all inodes
  - Data blocks take up most of the disk

UNIX File System

- The old UNIX file system:
  - Superblock: volume size, number of inodes, pointer to head of free list of blocks, etc.
  - The inode region for all inodes
  - Data blocks take up most of the disk

- Simple, supports basic abstractions, easy to use
- Problem: terrible performance (2% of disk bandwidth)

UNIX File System

- Disk treated as random-access memory
  - Expensive positioning costs
  - e.g., data blocks of file far away from its inode
- File system **fragmented**
  - Logically contiguous file $\rightarrow$ back and forth across the disk
- Block size too small (512 bytes)
  - Bad for data transfer
  - Positioning overhead for each block

UNIX File System

- For example, data block region with four files:

A1	A2	B1	B2	C1	C2	D1	D2

UNIX File System

- For example, data block region with four files:

A1	A2	B1	B2	C1	C2	D1	D2

- Files B & D are deleted:

A1	A2			C1	C2		

UNIX File System

- For example, data block region with four files:

A1	A2	B1	B2	C1	C2	D1	D2

- Files B & D are deleted:

A1	A2			C1	C2		

- Allocate file E, of size four blocks:

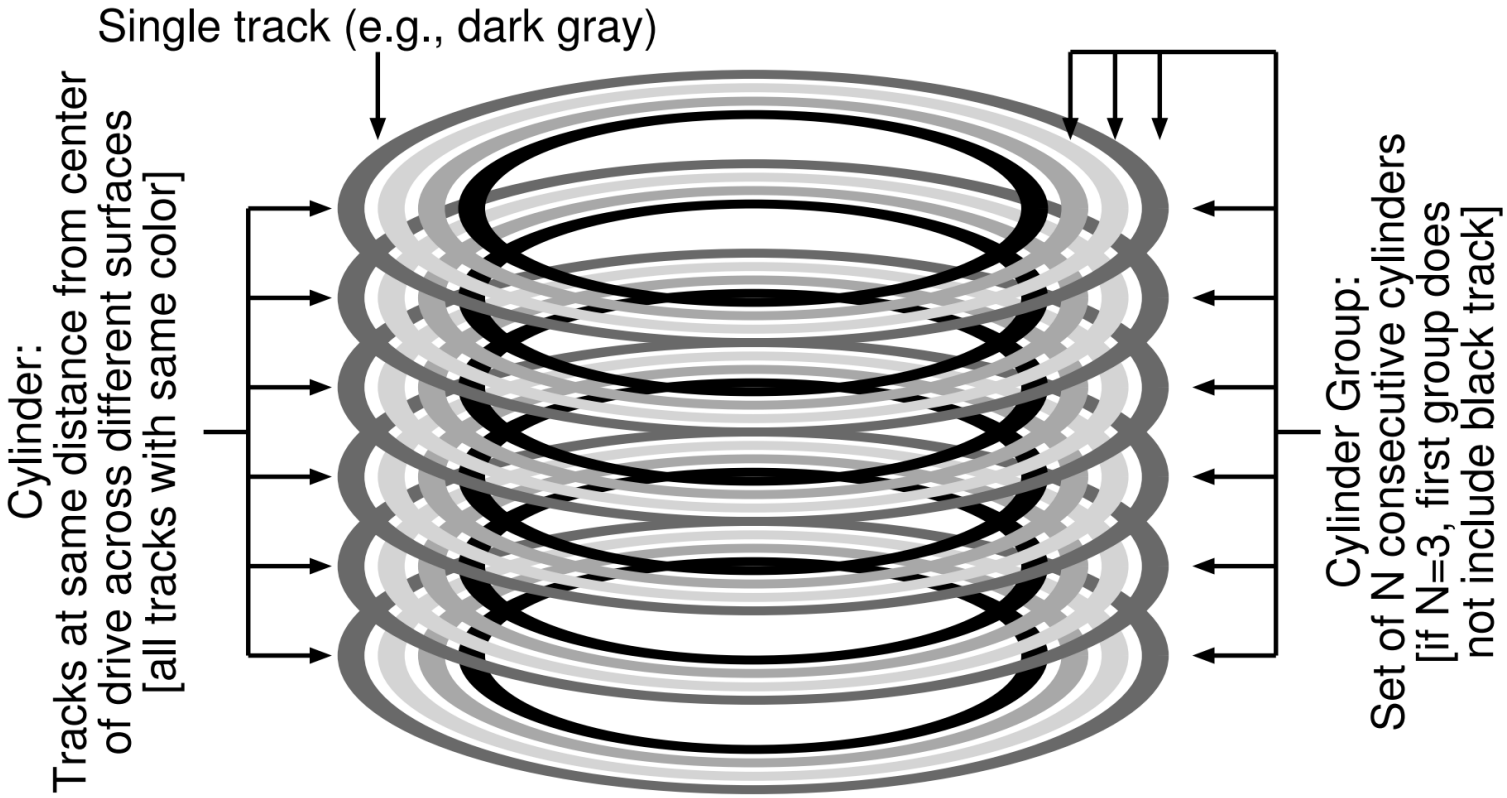
A1	A2	E1	E2	C1	C2	E3	E4

FFS

- **Fast File System**
  - Design structures and allocation to be "disk aware"
- Keep same API
  - `open()`, `read()`, `write()`, `close()`, etc.
  - Change internal implementation
  - Paved the path for new file system construction

Cylinder Group

- FFS divides disk into **cylinders** and **cylinder groups**
  - In modern file systems: **block groups**
  - e.g., Linux ext2, ext3, and ext4



Cylinder Group

- Use groups to improve seek performance
- e.g., place two files within the same group
- Allocate files and directories within each group

G0	G1	G2	G3	G4	G5	G6	G7	G8	G9

Cylinder Group

- Within a single group:
  - Copy of the **super block** (S)
    - For reliability reasons
  - Per-group **inode bitmap** (ib) and **data bitmap** (db)
  - The **inode** and **data block** regions
    - Same as vsfs

Policies

- Keep related stuff together
  - Keep unrelated stuff far apart
- Placement of directories:
  - Find group with low number of allocated directories, high number of free inodes
  - Put directory data and inode in that group
- Placement of files:
  - Allocate data blocks in same group as inode
  - Place files in same group as directory

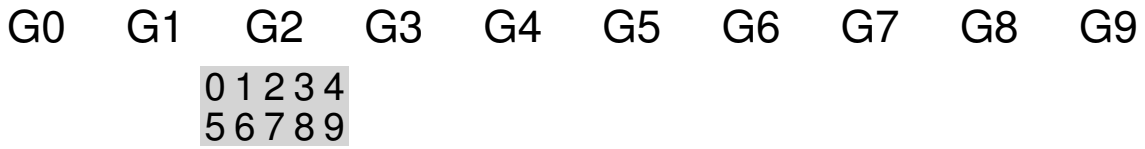
Policies

- Create 3 dirs (/, /a, /b) and four files (/a/c, /a/d, /a/e, /b/f)

```
group   inodes      data            group   inodes      data
    0   /---------  /---------           0   /---------  /---------
    1   acde------  accddee---           1   a---------  a---------
    2   bf--------  bff-------           2   b---------  b---------
    3   ----------  ----------           3   c---------  cc--------
    4   ----------  ----------           4   d---------  dd--------
    5   ----------  ----------           5   e---------  ee--------
    6   ----------  ----------           6   f---------  ff--------
    7   ----------  ----------           7   ----------  ----------
    ...                                  ...
    With name locality                   No name locality
```

Large-File Exception

- General policy: exception for large files
  - Entirely fill block group it is placed within
  - Prevents related files from being placed in group



- For large files: spread chunks across disk
  - Hurts performance, can address by choosing chunk size carefully
  - Reduce overhead by doing more work: **amortization**

Amortization

- For example:
  - Average positioning time: 10 ms, transfer rate: 40 MB/s
  - Goal: Achieve 50% of peak disk performance
  - i.e., 10 ms transferring data for every 10 ms positioning
  - How big does a chunk have to be?

Amortization

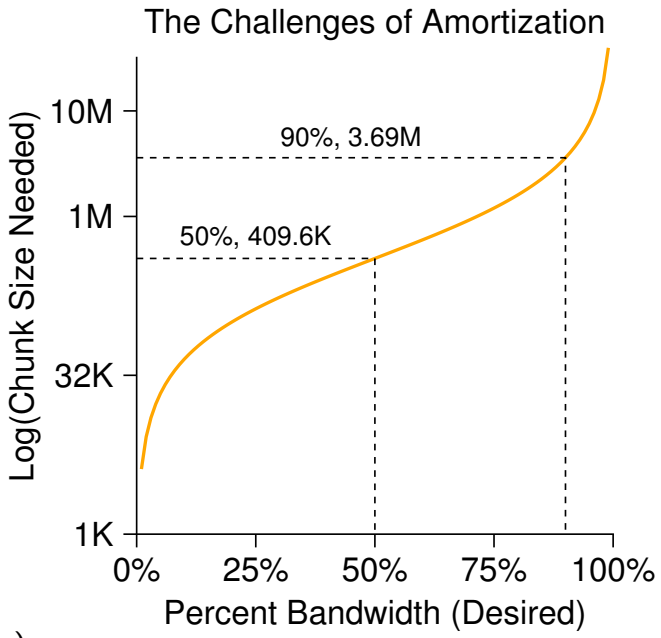
- For example:
  - Average positioning time: 10 ms, transfer rate: 40 MB/s
  - Goal: Achieve 50% of peak disk performance
  - i.e., 10 ms transferring data for every 10 ms positioning
  - How big does a chunk have to be?

- 40 MB/s $\rightarrow$ 409.6 KB
  - $(40 \cdot 1024/100)$

Amortization

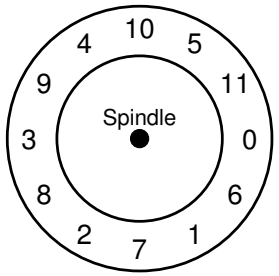
- For example:
  - Average positioning time: 10 ms, transfer rate: 40 MB/s
  - Goal: Achieve 50% of peak disk performance
  - i.e., 10 ms transferring data for every 10 ms positioning
  - How big does a chunk have to be?

- 40 MB/s $\rightarrow$ 409.6 KB
  - $(40 \cdot 1024/100)$



FFS

- Internal fragmentation
  - Most files were small (at the time)
  - Use sub-blocks of 512 bytes
- **Parameterization**
  - Skip over every other block
  - Enough time to request next block before it went past disk head



  - **Track buffer** prevents two spins to read track
- Also introduced: **long file names** and **symbolic links**

Summary

- Divide disk into blocks
  - Commonly-used size (4KB)
  - Data region for user data, metadata region for inodes
  - Allocation structure (data and inode bitmaps)
  - **Superblock**: information about file system
- Data uses **direct** and **indirect pointers**
  - Multi-level approach: pointer to block of indirect pointers
  - **Extents**: disk pointer plus length
- Access paths: huge number of I/Os
  - Cache with **dynamic partitioning**
  - **Write buffering**
- FFS: using **cylinder groups** and **large-file exception**