IOWA STATE UNIVERSITY

Department of Electrical and Computer Engineering

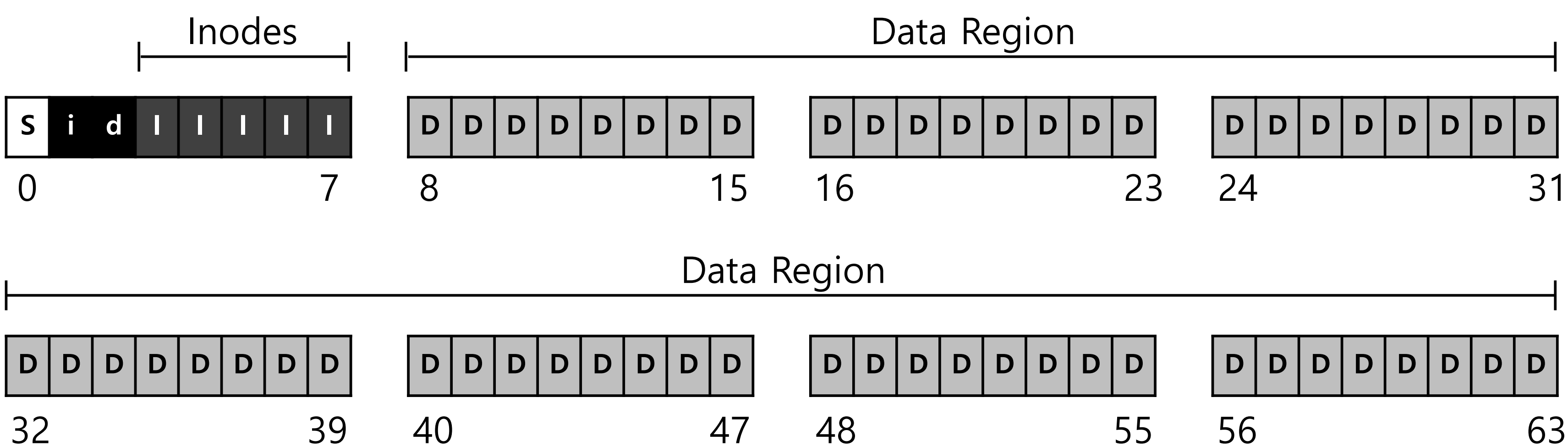
Lecture 32:
File System Implementation II

Agenda

- Recap
- File System Implementation II
  - Access Paths
  - Caching & Buffering
  - Fast File System (FFS)

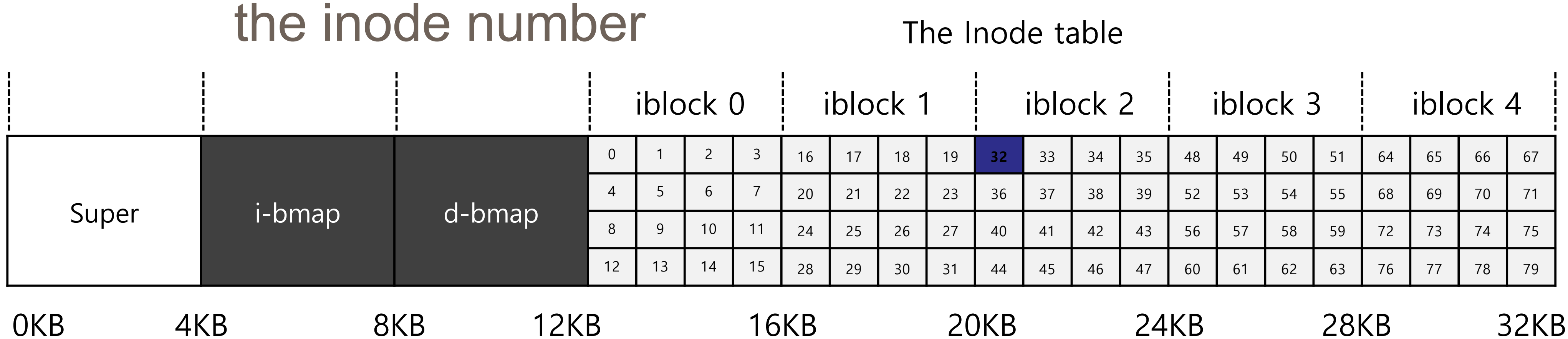
Recap

- Two important perspectives of an FS
  - Data structures
  - Access methods
- Basic design
  - data region: user data
  - metadata region: inodes, bitmaps, superblock



Recap

- Each user file is represented by one `inode`
  - `inode` contains all information about the file
    - e.g., type, size, permissions, timestamps, pointers to data blocks
    - data blocks may be indexed by multi-level indirect pointers
- Each `inode` is referred to by an inode number
  - FS calculates where the inode is on the disk based on the inode number



Agenda

- ~~Recap~~
- File System Implementation II
  - Access Paths
  - Caching & Buffering
  - Fast File System (FFS)

Access Paths

- Reading a file from disk
  - Issue `open("/foo/bar", O_RDONLY)`
    - Traverse the pathname and locate the desired inode
      - Begin at the root of the file system `(/)`
        - In many Unix file systems, the root inode number is 2
      - based on the inode number, FS finds the block in the inode table that contains inode #2
      - Read inode #2 to find pointers to its data blocks (contents of the root directory)
      - read the data blocks and find an entry containing "foo" and its inode#
      - Traverse recursively the pathname until finding the inode for "bar"
      - Check permissions, allocate a file descriptor for this process and returns file descriptor to user.

Access Paths

- Reading a file from disk (cont')
  - Issue `read()` to read from the file
    - Already found the inode for the open file (via `open`)
    - read the inode to find the location of its data blocks
    - read in the first block of the file
      - Update the inode with a new last accessed time
      - Update the file offset in the open file table for the process
  - When the file is closed:
    - File descriptor should be deallocated
    - that is all the FS really needs to do; no disk I/Os take place

Access Paths

- Reading a file from disk (cont')
  - Timeline (Time Increasing downward)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Access Paths

- Writing to a file on disk
  - Issue `write()` to update the file with new contents.
  - may allocate a new data block (unless a block is being overwritten).
  - Need to update data block, data bitmap
  - generates five I/Os (at least):
    - one to read the data bitmap
    - one to write the bitmap (to reflect its new state to disk)
    - two more to read and then write the inode
    - one to write the actual block itself
  - To create file, FS also needs to update the directory, causing additional I/O traffic

Access Paths

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write							write		
					write					
write()					read					
	read									
	write								write	
					write					
write()					read					
	read									write
	write									
					write					

Agenda

- ~~Recap~~
- ~~File System Implementation II~~
  - ~~Access Paths~~
  - Caching & Buffering
  - Fast File System (FFS)

Caching and Buffering

- Reading and writing files are expensive, incurring many I/Os
  - E.g., long pathname(/1/2/3/…./100/file.txt)
    - One to read the inode of the directory and at least one read its data
    - Literally perform hundreds of reads just to open the file
- To reduce I/O traffic, FSes use system memory (DRAM) to cache reads
  - Read I/O can be avoided by large cache
  - **page cache** in Linux

Caching and Buffering

- Write traffic has to go to disk for persistence
  - caching is less useful for writes
- But buffering writes in memory may still help improve performance
  - FS can optimize the writes in memory, e.g.:
    - batch some updates into a smaller set of I/Os
    - avoiding unnecessary I/O (e.g., overwritten in memory)
- Applications may force flush dirty data to disk by calling `fsync()`

Agenda

- ~~Recap~~
- File System Implementation II
  - ~~Access Paths~~
  - ~~Caching & Buffering~~
  - Fast File System (FFS)

The 1st Unix File System (~1974)

- Similar to the basic FS we discussed
  - a simplified view

S	**Inodes**	**Data**

- The Good Thing
  - Simple and supports the basic abstractions
  - Easy to implement
- The Problem
  - Terrible performance

The 1st Unix File System (~1974)

- Why the performance is terrible
  - major issue: treated the disk as a **random-access memory**
  - Example of random-access blocks with four files.
    - Data blocks for each file can accessed by going back and forth the disk, because they are are **contiguous.**

A1	A2	B1	B2	C1	C2	D1	D2

    - File b and d is deleted.

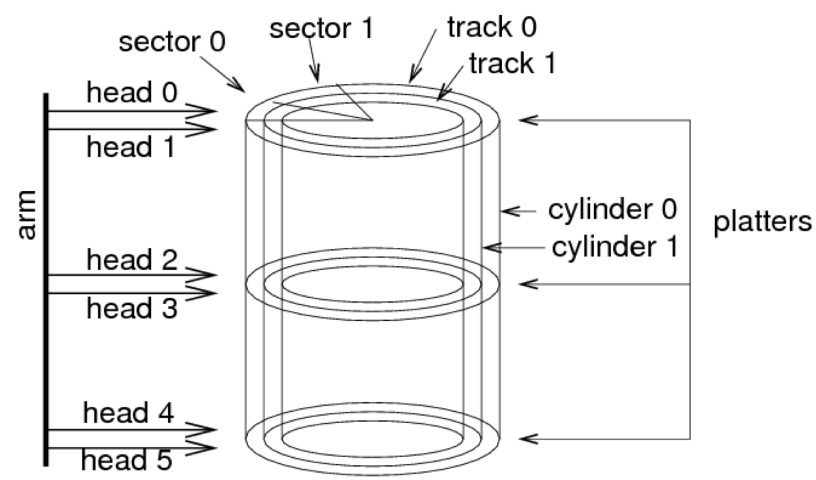
A1	A2			C1	C2		

    - File E is created with free blocks. (**spread across** the disk!)

A1	A2	E1	E2	C1	C2	E3	E4

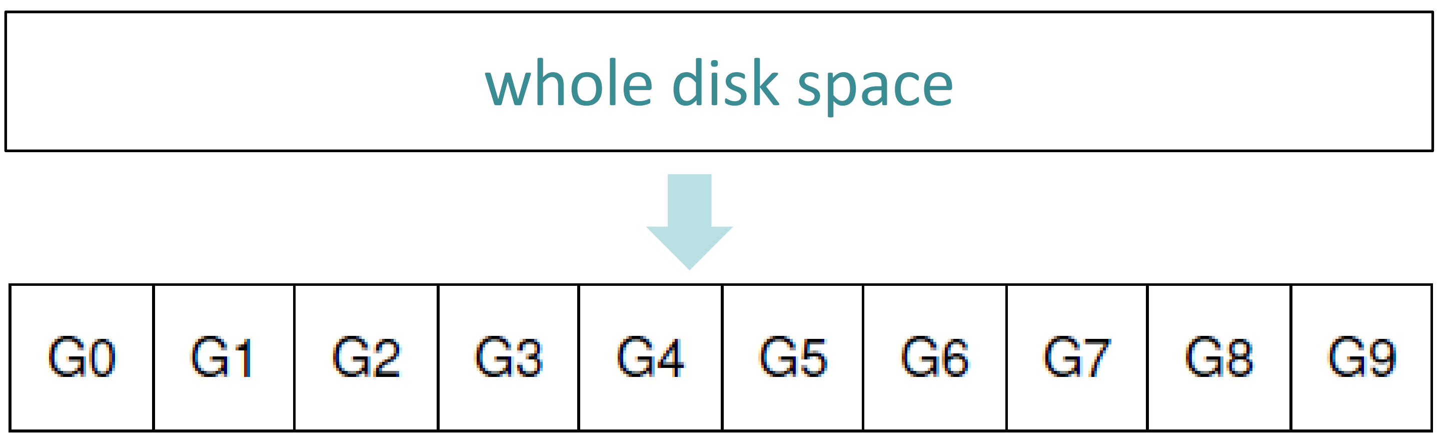
The 1st Unix File System (~1974)

- Why the performance is terrible
  - major issue: treated the disk as a **random-access memory**
  - The actual disk is different from DRAM (recap)
    - access latency varies much at different locations
      - e.g., seek time varies



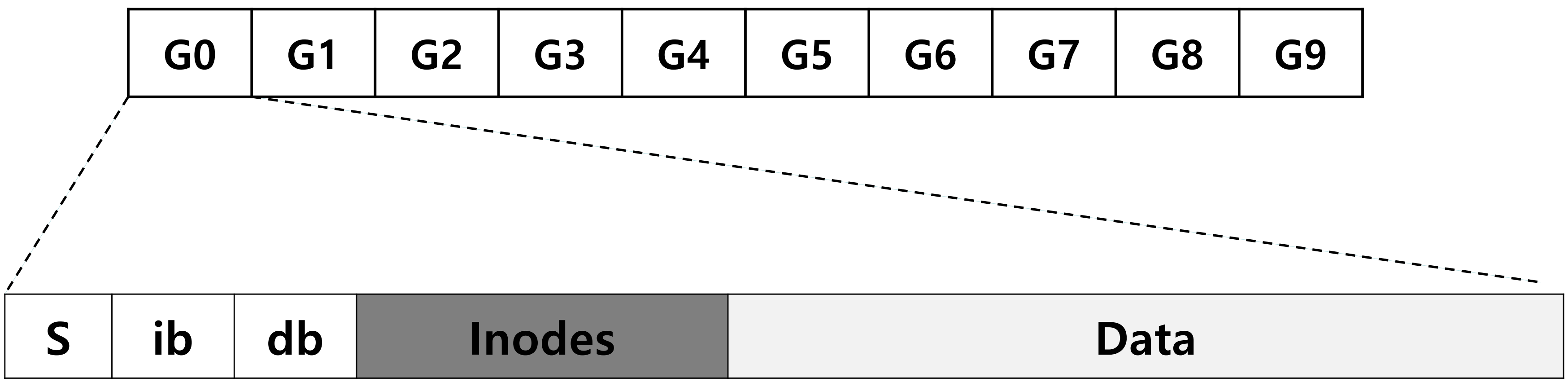
The Fast File System (FFS, ~1984)

- Key insight: disk awareness
  - FS structures and allocation polices match the internals of disks
  - Divide the disk into cylinder groups (block groups)
    - place related stuff in the same group, avoid long seek



The Fast File System (FFS, ~1984)

- Each cylinder group has a structure similar to the Unix FS



The Fast File System (FFS, ~1984)

- Heuristics to Allocate Related Stuff Together
  - For directory
    - find the cylinder group with:
      - a low number of allocated directories (b/c we want to balance directories across groups)
      - a high number of free inodes (b/c we want to be able to allocate a bunch of files later)
    - put the directory data and inode in that same group
  - For file
    - allocate the data and inode of the file in the same group
    - places all files that are in the same directory in the cylinder group of the directory they are in

The Fast File System (FFS, ~1984)

- Much Better Performance
  - e.g., 14-47% of raw disk bandwidth
- Main Lesson of FFS
  - treat disk like it’s a disk
- A watershed moment in FS history
- Many FSes today take cues from FFS

Agenda

- ~~Recap~~
- ~~File System Implementation II~~
    - ~~Access Paths~~
    - ~~Caching & Buffering~~
    - ~~Fast File System (FFS)~~

Questions?

*acknowledgement: slides include content from “Modern Operating Systems” by A. Tanenbaum, “Operating Systems Concepts” by A. Silberschatz etc., “Operating Systems: Three Easy Pieces” by R. Arpaci-Dusseau etc., and anonymous pictures from internet.