IOWA STATE UNIVERSITY

Department of Electrical and Computer Engineering

Lecture 33: Data Integrity & Protection I

Agenda

- Recap
- Data Integrity & Protection I
  - Crash Consistency Problem
  - Journaling

Recap

- Access paths
  - Timeline of reading a file from disk & writing to disk

	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 write			read		
read()					read write				read	
read()					read write					read

Recap

	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					

Recap

- Caching & Buffering
  - Reading and writing files are expensive, incurring many I/Os
  - FSes use system memory (DRAM) to cache reads and buffer writes
    - **page cache** in Linux
    - 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()`

Recap

- The 1st Unix File System (~1974)
  - simple
  - poor performance due to fragmentation
    - file data become non-contiguous
    - long seek time

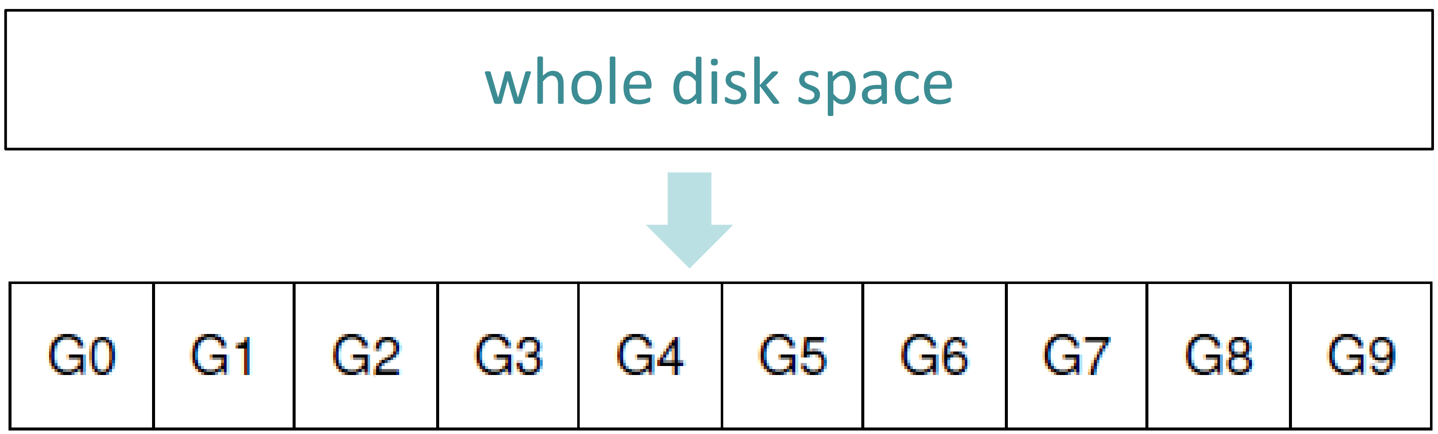
A1	A2	B1	B2	C1	C2	D1	D2

A1	A2			C1	C2		

A1	A2	E1	E2	C1	C2	E3	E4

Recap

- The Fast File System (FFS, ~1984)
  - Key insight: disk awareness
    - data 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



Agenda

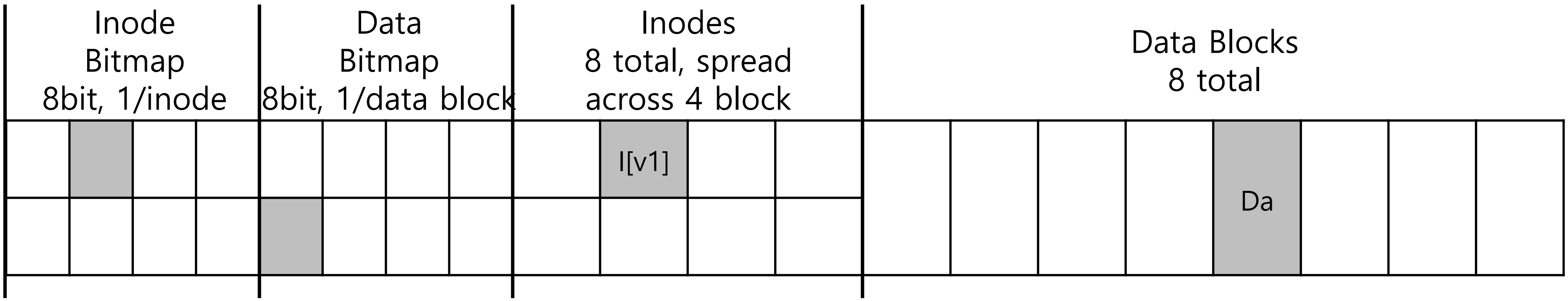
- ~~Recap~~
- Data Integrity & Protection I
  - Crash Consistency Problem
  - Journaling

Crash Consistency Problem

- Unlike many in-memory data structures, FS data structures must **persist** on disk
  - need to be consistent all the time
    - if the FS data structures is corrupted (i.e., becomes inconsistent), user data may be lost
- Challenge: how to update persistent data structures safely and maintain consistency in face of unexpected failure events (e.g., power loss, system crashes)

Crash Consistency Problem

- Example
  - Workload
    - Append a single data block(4KB) to an existing file
    - open() → lseek() → write() → close()



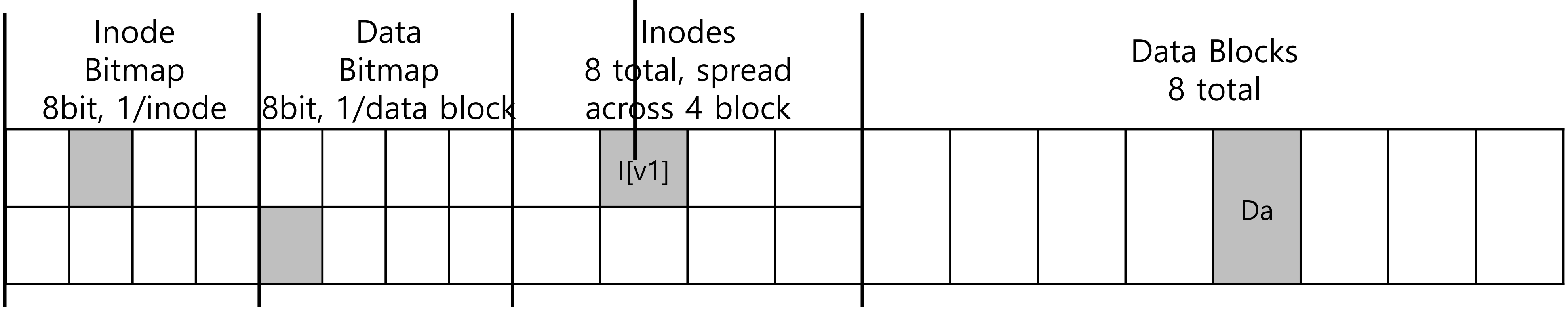
  - Before appending a single data block
    - single inode is allocated (inode number 1) (count from 0)
    - single allocated data block (data block 4) (count from 0)
    - The inode is denoted I[v1]

Crash Consistency Problem

- Example (cont')
  - Content of I[v1] before update

```
owner               : remzi
permissions         : read-only
size                : 1
pointer             : 4
pointer             : null
pointer             : null
pointer             : null
```

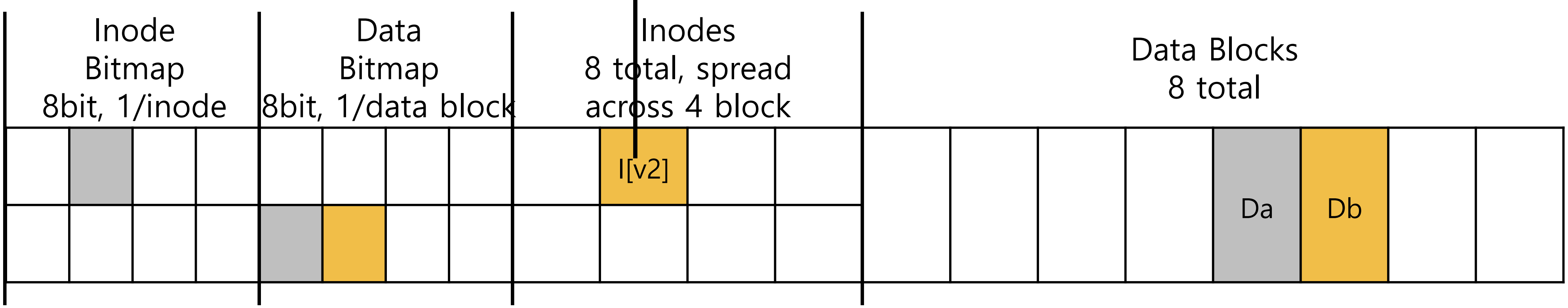
- Size of the file is 1
  - one block allocated
- First direct pointer points to block4 (Da)
- All 3 other direct pointers are set to `null`(unused)



Crash Consistency Problem

- Example (cont')
  - After update: I[v2]

```
owner         : remzi
permissions   : read-only
size          : 2
pointer       : 4
pointer       : 5
pointer       : null
pointer       : null
```



- Data bitmap is updated
- Inode is updated (I[v2])
- New data block is allocated (Db)

Crash Consistency Problem

- Example (cont')
  - FS performs three separate writes to the disk
    - Update inode from I[v1] to I[v2]
    - Data bitmap
    - Data block (Db)
  - These writes usually don't happen immediately
    - dirty inode, bitmap, and new data may sit in main memory for a while
      - **page cache** in Linux
  - If a crash happens after one or two of these write have taken place, but not all three, the FS could be in an inconsistent state

Crash Consistency Problem

- Example (cont')
  - Crash Scenarios
    - Imagine only a single write succeeds: three possible outcomes
      - (1) Only the data block(Db) is written to disk
        - The data is on disk, but there is no inode
        - Thus, it is as if the write never occurred
        - This case is not a problem at all

Crash Consistency Problem

- Example (cont’)
  - Crash Scenarios
    - Imagine only a single write succeeds: three possible outcomes
      - (2) Only the updated inode(I[v2]) is written to disk
        - The inode points to the disk address (5, Db)
        - Db has not been written
        - We will read **garbage** data (old contents of address 5) from the disk

Crash Consistency Problem

- Example (cont’)
  - Crash Scenarios
    - Imagine only a single write succeeds: three possible outcomes
      - (3) Only the updated bitmap (B[v2]) is written to disk
        - The bitmap indicates that block 5 is allocated
        - But there is no inode that points to it
        - space leak: block 5 would never be used by the file system

Crash Consistency Problem

- Example (cont')
  - Crash Scenarios
    - If TWO writes succeed: another three possible outcomes
      - (1) The inode(I[v2]) and bitmap(B[v2]) are written to disk, but not data(Db)
        - The file system metadata is completely consistent
        - Problem : Block 5 has garbage in it
      - (2) The inode(I[v2]) and the data block(Db) are written, but not the bitmap(B[v2])
        - We have the inode pointing to the correct data on disk
        - Problem : inconsistency between the inode and the old version of the bitmap(B1)
          - the data block(Db) may be overwritten mistakently (since the bitmap doesn't mark it as unused)

Crash Consistency Problem

- Example (cont')
  - Crash Scenarios
    - If TWO writes succeed: another three possible outcomes
      - (3) The bitmap(B[v2]) and data block(Db) are written, but not the inode(I[v2])
        - Problem : inconsistency between the inode and the data bitmap
        - We have no idea which file it belongs to

Crash Consistency Problem

- Ideally, an FS should transit from one consistent state to another consistent state **atomically**

- Unfortunately, we can't do this easily
  - the three updates are issued to non-contiguous locations, cannot be merged into one single writes
    - have to issue three separate writes
  - failure events may occur between these writes

Agenda

- ~~Recap~~
- Data Integrity & Protection I
  - ~~Crash Consistency Problem~~
  - Journaling

Journaling

- Used in Ext3 & Ext4 (and many other FSes)
  - no journaling in Ext2
- Also called Write-Ahead-Logging (WAL)
  - common in database community
- Basic Idea
  - Do not write to the main FS data structures directly
  - Write to a "journal" data structure first
  - Update the main FS data structures only after all relevant writes are safely stored in the journal

Agenda

- ~~Recap~~
- Data Integrity & Protection I
  - ~~Crash Consistency Problem~~
  - ~~Journaling~~

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.