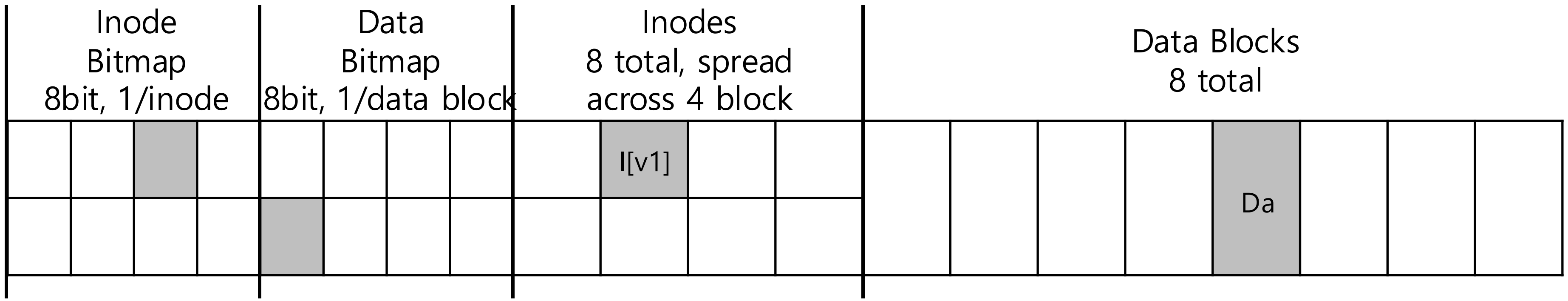
Crash Consistency

Crash Consistency

- Unlike most data structure, file system data structures must **persist**
  - They must survive over the long haul, stored on devices that retain data despite power loss
- One major challenge faced by a file system is how to update persistent data structure despite the presence of a **power loss** or **system crash**
- We'll begin by examining the approach taken by older file systems
  - **fsck**(file system checker)
  - **journaling**(write-ahead logging)

A Detailed Example

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

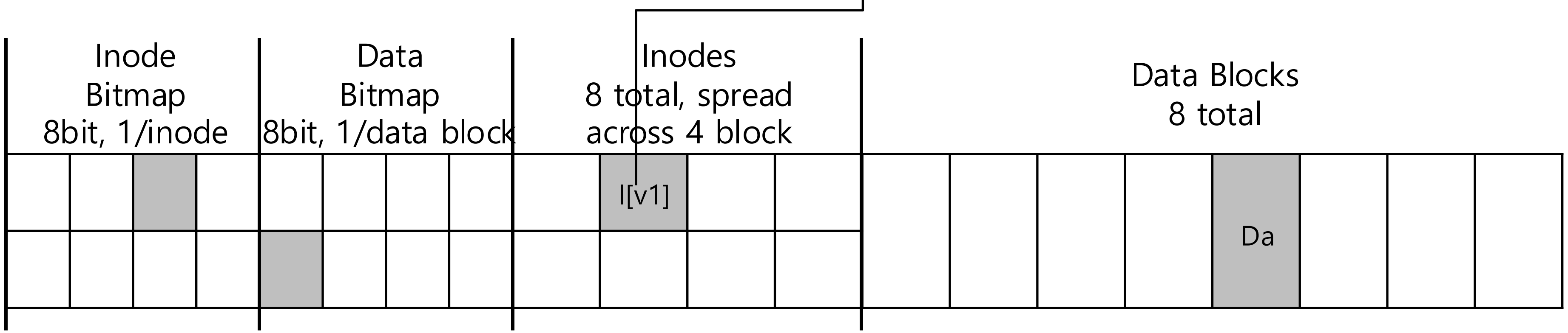


- Before append a single data block
  - single inode is allocated (inode number 2)
  - single allocated data block (data block 4)
  - The inode is denoted I[v1]

A Detailed Example

- Inside of I[v1] (inode, before update)

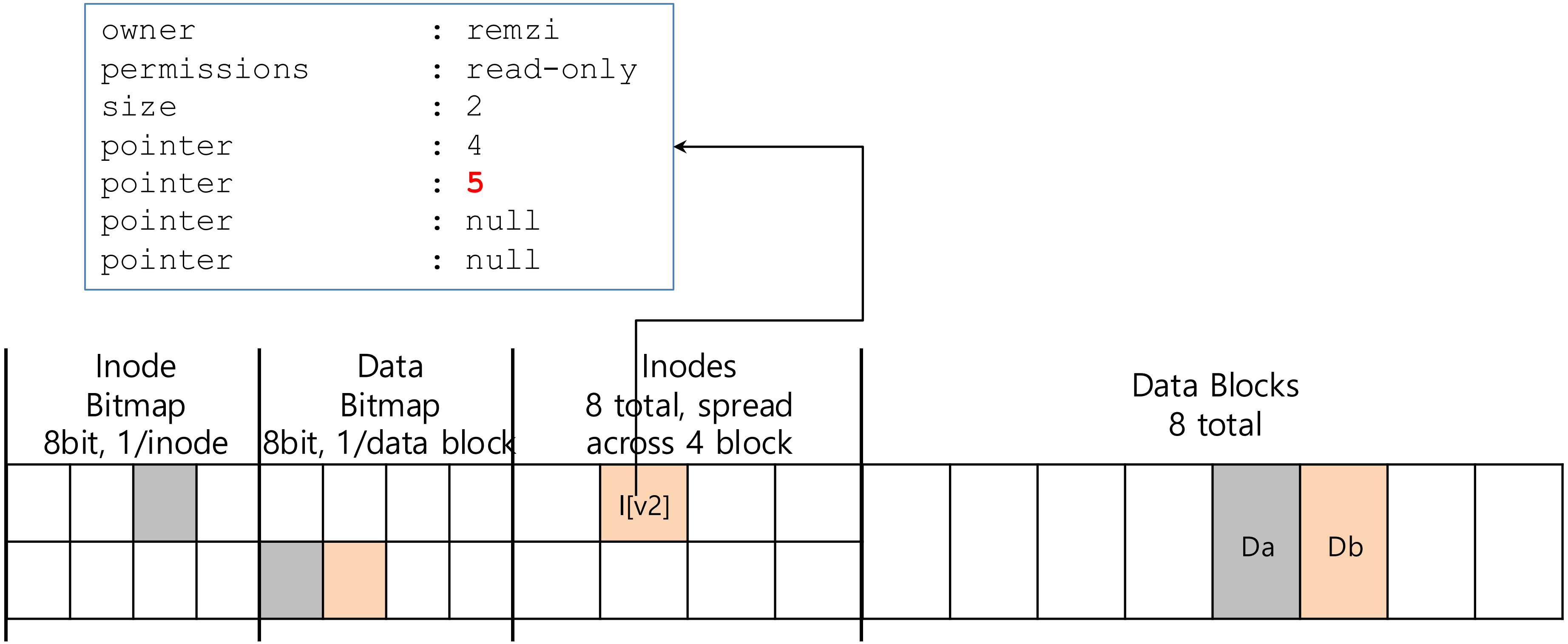
```
owner          : remzi
permissions    : read-only
size           : 1
pointer        : 4
pointer        : null
pointer        : null
pointer        : null
```



- 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)

A Detailed Example

- After update



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

A Detailed Example

- To achieve the transition, the system perform three separate writes to the disk
  - One each of inode I[v2]
  - Data bitmap B[v2]
  - Data block (Db)
- These writes usually don't happen immediately
  - dirty inode, bitmap, and new data will sit in main memory
  - **page cache** or **buffer cache**
- If a crash happens after one or two of these write have taken place, but not all three, the file system could be left in a funny state

Crash Scenario

- Imagine only a single write succeeds; there are thus three possible outcomes
  1. Just 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
  2. Just the updated inode(I[v2]) is written to disk
     - The inode points to the disk address (5, Db)
     - But, the Db has not yet been written there
     - We will read **garbage** data(old contents of address 5) from the disk
     - **Problem : file-system inconsistency**

Crash Scenario

- Imagine only a single write succeeds; there are thus three possible outcomes (Cont.)
  3. Just 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
     - Thus, the file system is inconsistent again
     - **Problem : space leak,** as block 5 would never be used by the file system

Crash Scenario

- There are also three more crash scenarios. In these cases, two writes succeed and the last one fails
  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)**

Crash Scenario

- There are also three more crash scenarios. In these cases, two writes succeed and the last one fails (Cont.)
  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

- What we'd like to do ideally is move the file system from on consistent state to another **atomically**
- Unfortunately, we can't do this easily
  - The disk only commits one write at a time
  - Crashes or power loss may occur between these updates
- We call this general problem the **crash-consistency problem**

Solution #1: File System Checker

File System Checker

- The File System Checker (**`fsck`**)
  - `fsck` is a Unix tool for finding inconsistencies and repairing them
  - `fsck` check super block, Free block, Inode state, Inode links, etc
  - Such an approach can't fix all problems
    - example : The file system looks consistent but the inode points to garbage data
  - The only real goal is to make sure the file system metadata is internally consistent

File System Checker

- Basic summary of what `fsck` does:
  - **Superblock**
    - `fsck` first checks if the superblock looks reasonable
      - » Sanity checks : file system size > number of blocks allocated
    - Goal : to find suspect superblock
    - In this case, the system may decide to use an alternate copy of the superblock
  - **Free blocks**
    - `fsck` scans the inodes, indirect blocks, double indirect blocks, and so on
    - The only real goal is to make sure the file system metadata is internally consistent

File System Checker

- Basic summary of what `fsck` does: (Cont.)
  - **Inode state**
    - Each inode is checked for corruption or other problem
      - » Example : type checking(regular file, directory, symbolic link, etc)
    - If there are problems with the inode fields that are not easily fixed
      - » The inode is considered suspect and cleared by `fsck`
  - **Inode Links**
    - `fsck` also verifies the link count of each allocated inode
      - » To verify the link count, `fsck` scans through the entire directory tree
    - If there is a mismatch between the newly–calculated count and that found within an inode, corrective action must be taken
      - » Usually by fixing the count with in the inode

File System Checker

- Basic summary of what `fsck` does: (Cont.)
  - **Inode Links** (Cont.)
    - If an allocated inode is discovered but no directory refers to it, it is moved to the lost+found directory
  - **Duplicates**
    - fsck also checks for duplicated pointers
    - Example : Two different inodes refer to the same block
      - If one inode is obviously bad, it may be cleared
      - Alternately, the pointed-to block could be copied

File System Checker

- Basic summary of what `fsck` does: (Cont.)
  - **Bad blocks**
    - A check for bad block pointers is also performed while scanning through the list of all pointers
    - A pointer is considered "bad" if it obviously points to something outside its valid range
    - Example : It has an address that refers to a block greater than the partition size
      - » In this case, `fsck` can't do anything too intelligent; it just removes the pointer

File System Checker

- Basic summary of what `fsck` does: (Cont.)
  - **Directory checks**
    - `fsck` does not understand the contents of user files
      - » However, directories hold specifically formatted information created by the file system itself
      - » Thus, `fsck` performs additional integrity checks on the contents of each directory
    - Example
      - » making sure that "." and ".." are the first entries
      - » each inode referred to in a directory entry is allocated?
      - » ensuring that no directory is linked to more than once in the entire hierarchy

File System Checker

- Building a working `fsck` requires intricate knowledge of the filesystem
- `fsck` have a bigger and fundamental problem: **too slow**
  - scanning the entire disk may take many minutes or hours
  - Performance of `fsck` became prohibitive
    - as disk grew in capacity and RAIDs grew in popularity
- At a higher level, the basic premise of fsck seems just a tad irrational
  - It is incredibly expensive to scan the entire disk
  - It works but is wasteful
  - Thus, as disk(and RAIDs) grew, researchers started to look for other solutions