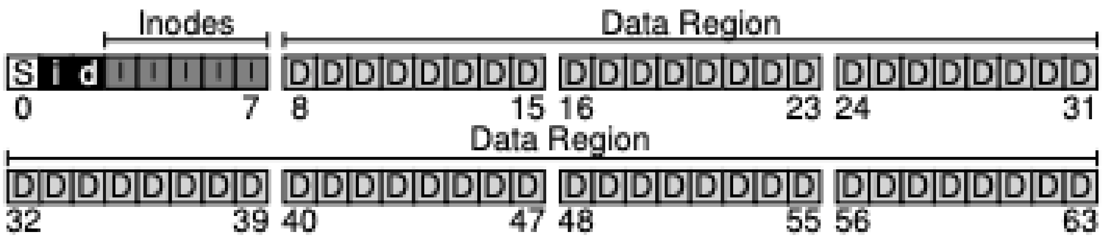
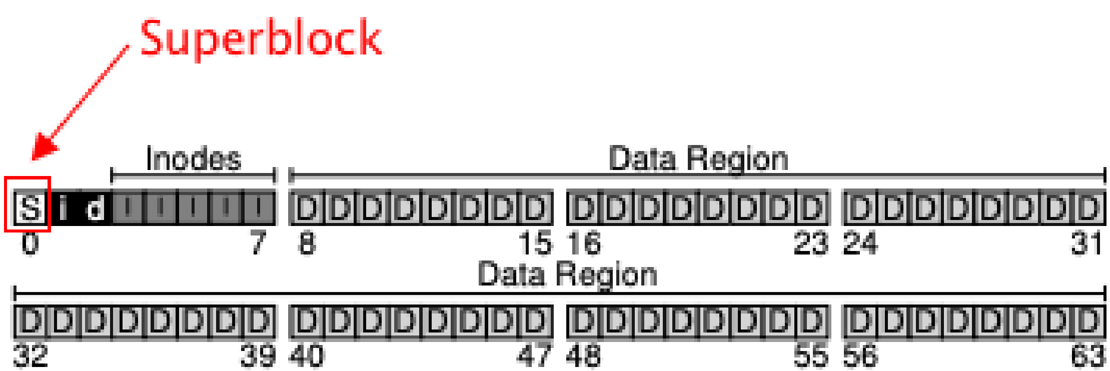
Index-based File System



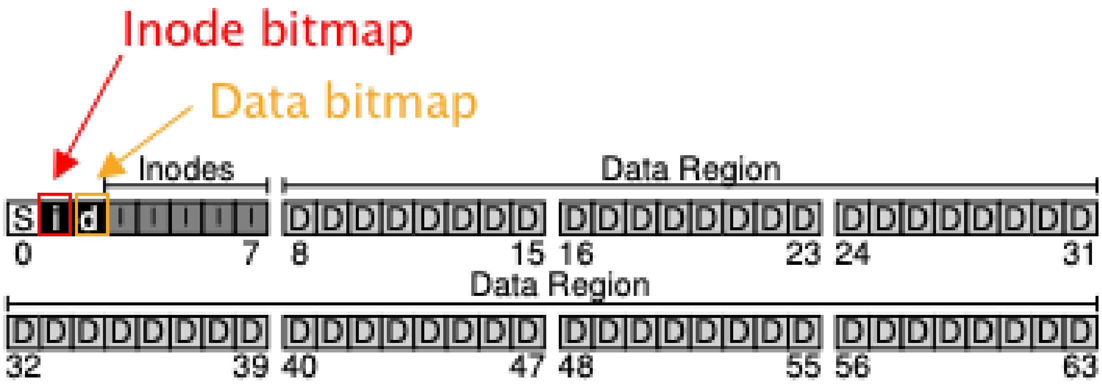
- Has following parts
  - Superblock
  - Inode Bitmap
  - Data Bitmap
  - Inodes
  - Data Region
- Each block in file system is 4KB
- Uses a large amount of metadata per file (especially for large files)

Superblock



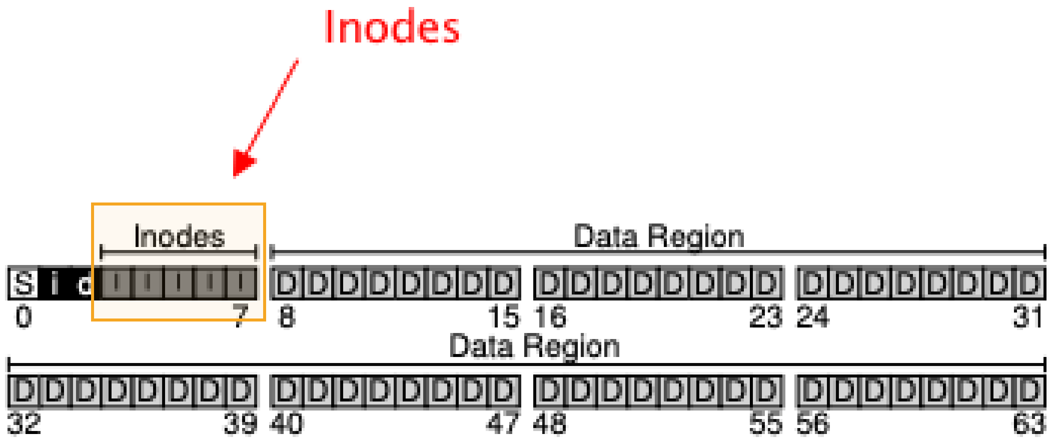
- contains information about the file system, including
  1. the number of inodes and data blocks in a particular file system
  2. the magic number of some kind to identify the file system type (e.g NFS, FFS, VSFS)
- The OS reads superblock first to initialize various parameters, and then attach volume to the file-system tree

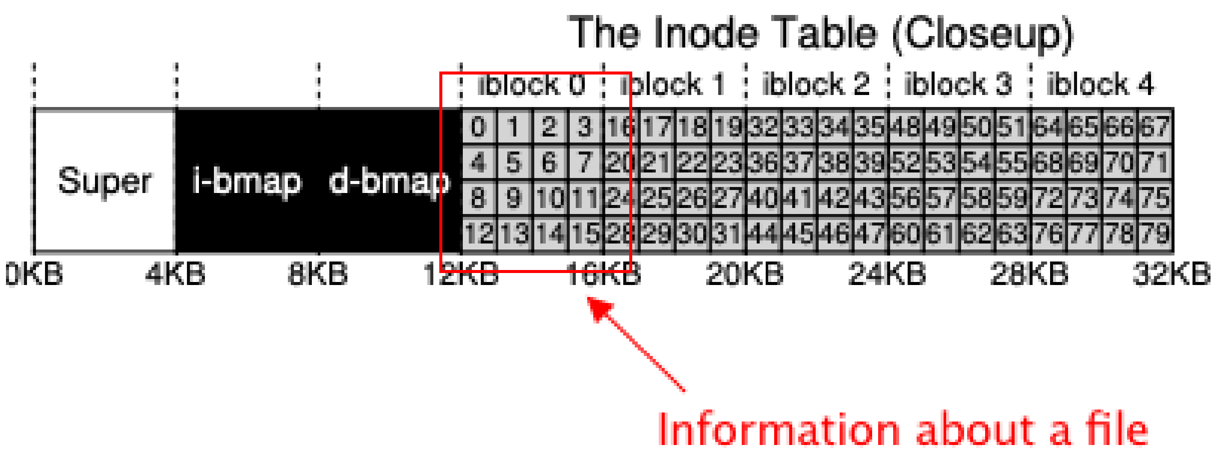
Bitmap



- Tracks whether inode or data blocks are free or allocated
- Is a simple and popuar structure
- Uses each bit
  - `0` means free
  - `1` means in use
- **Data Bitmap** is bitmap for data region
- **Inode Bitmap** is bitmap for inode region

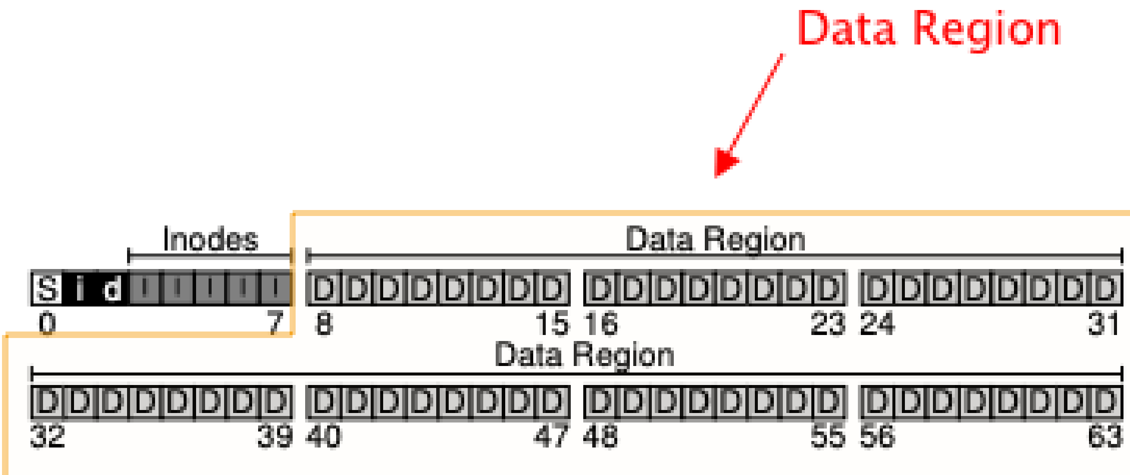
Inode





- Is a short form for **index node**
- Contains disk block location of the object's data [7]
- Contains all the information you need about a file (i.e. metadata)
  - File Type
    - e.g. regular file, directory, etc
  - Size
  - Number of blocks allocated to it
  - Protection information
    - such as who owns the file, as well as who can access it
  - Time information
    - e.g. When file was created, modified, or last accessed
  - Location of data blocks reside on disk

Data Region



- Is the region of disk we use for user data

1 Block

- Size of each block is 4KB

2 lseek

- **Syntax:** `off_t lseek(int fildes, off_t offset, int whence)`
  - `fildes` - file descriptor
  - `offset` - file offset to a particular position in file

3 Kilobyte

- 1 kilobyte is 1024 bytes

4 file

- is an array of bytes which can be created, read, written and deleted
- low-level name is called **inode number** or **i-number**

5 Reading a File From Disk

Example

When

`open("/foo/bar", O_READONLY)`

is called

- the goal is to find the inode of the file `bar` to read its basic information (i.e. includes permission, information, file size etc)
- done by traversing the pathname and locate the desired inode
- Steps
  1. Begin traversal at the root of the file system, in the **root directory**
  2. Find **inode** of the root directory by looking for `i-number`
     - **i-number** is found in it's parent directoy
     - for root directory, there is no parent directory
     - it's inode number is 2 (for UNIX file systems)
  3. Read the **inode** of root directory
  4. Once its **inode** is read, look inside to find pointers to data blocks
  5. Recursively traverse the pathname until the desired inode is found (e.g `foo` → `bar`)
  6. Issue a `read()` system call to read from file
     - `fd` with offset `0` reads the first file block (e.g. `bar data[0]`)
     - `lseek(..., offset_amt * size_of_file_block)` is used to offset/move to desired block in `bar`
  7. Trasnfer data to `buf` data block
  8. Close `fd`. No I/O is read.

6 inode

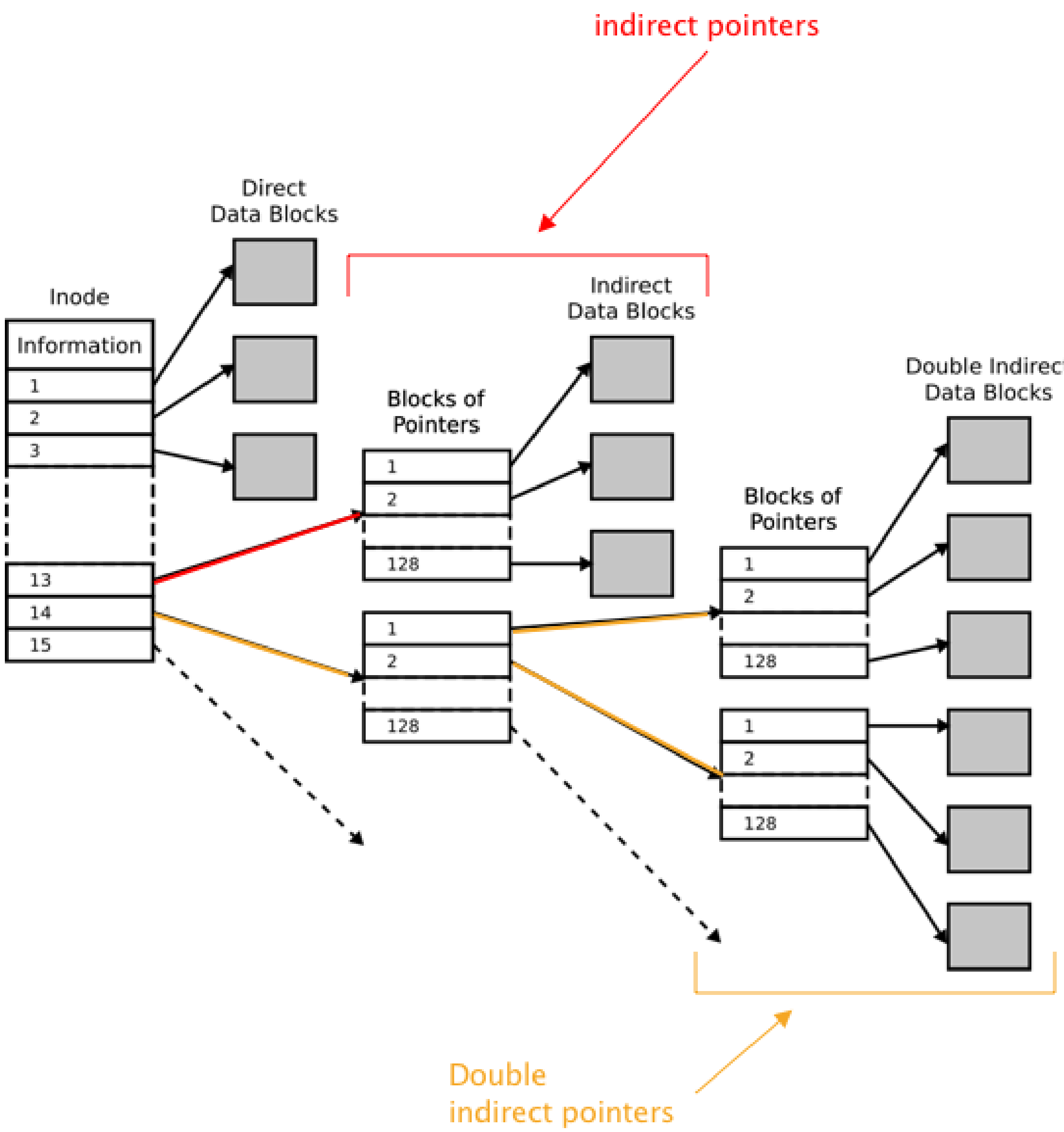
- total size may vary
- inode pointer has size of 4 byte
- Has 12 **direct pointers** to 4KB data blocks
- Has 1 **indirect pointer** [when file grows large enough]
- Has 1 **double indirect Pointer** [when file grows large enough]
- Has 1 **triple indirect Pointer** [when file grows large enough]

7 Indirect Pointers

- Is allocated to data-block if file grows large enough
- Has total size of 4 KB or 4096 bytes
- Has $4096/4 = 1024$ pointers
- Each pointer points to 4KB data-block
- File can grow to be $(12 + 1024) \times 4\text{K} = 4144\text{KB}$

8 Double Indirect Pointers

- is allocated when single indirect pointer is not large enough
- each pointer in first pointer block points to another pointer block
- has 1024^2 pointers
- each of 1024^2 pointers point to 4KB data block
- File can grow to be $(12 + 1024 + 1024^2) \times 4\text{K} = 4198448\text{KB}$ or $\approx 4.20\text{GB}$



9 Triple Indirect Pointers

- is allocated when double indirect pointer is not large enough
- has 1024^3 pointers
- each of 1024^3 pointers point to 4KB data block
- File can grow to be $(12 + 1024 + 1024^2 + 1024^3) \times 4K = 4299165744KB$ or $\approx 4.00TB$

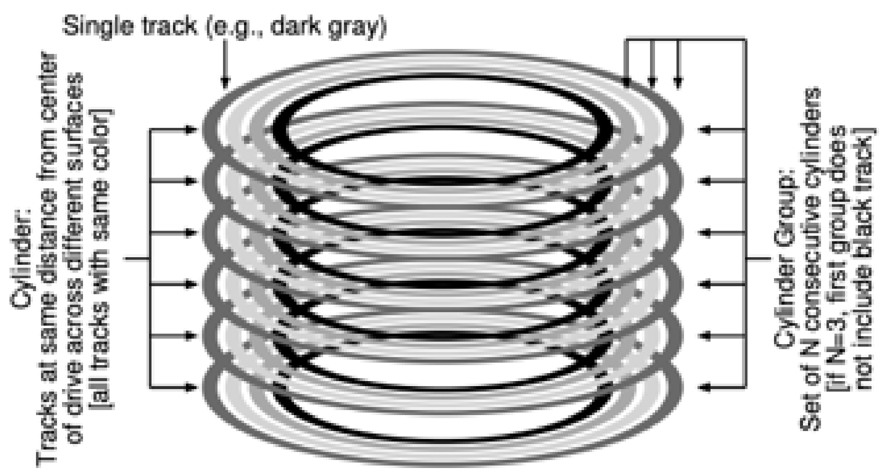
10 Old UNIX File system

- was simple, and looked like the following on disk

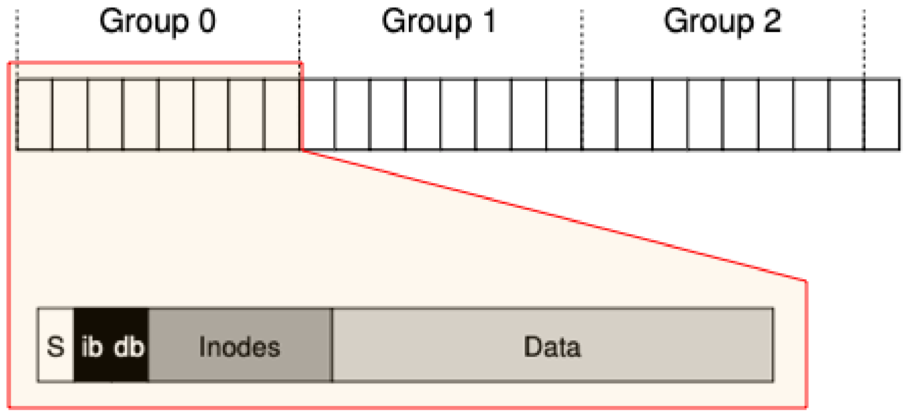
- has terrible performance
- suffers from **external fragmentation**
- had small data block (512 bytes) and transfer of data took too long

11 Fast File System

- modern file system has same APIS (`read()`, `write()`, `open()`, `close()`)
- divides into a number of **cylinder groups**



- each **block group** or **cylinder group** is consecutive portion of disk's address



12 Bitmap

- Are excellent way to manage free space
- tracks whether inodes/data block of the group are allocated

13 FFS Policies: Allocating Files and Directories

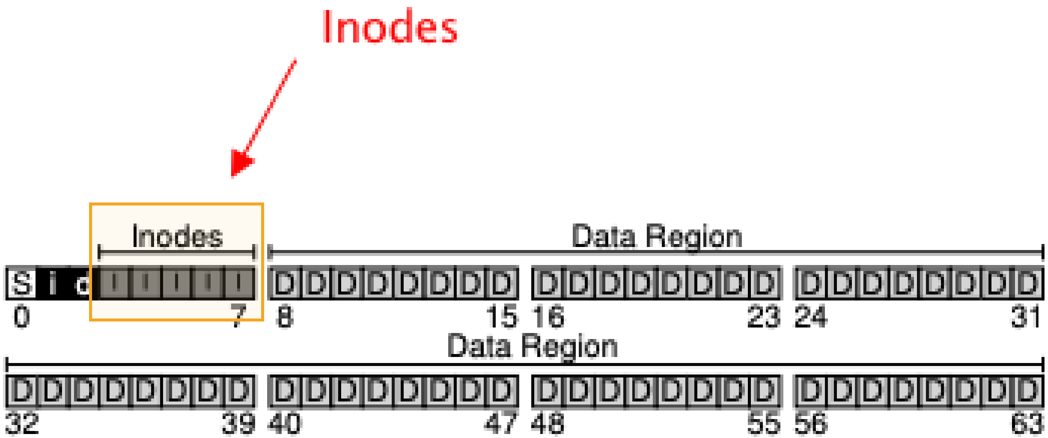
- Basic Idea: keep related stuff together, and keep related stuff far apart
- Directories Step
  1) Find the **cylinder group** with a low number of allocated directories and a high number of free inodes
     - low number of allocated directories → to balance directories across groups
     - high number of free nodes → to subsequently be able to allocate a bunch offiles
  2) Put directory data and inode to the **cylinder group**
- Files Step
  1) Allocate the data blocks of a file in the same **cylinder group** as its inode
  2) Place all files in the same directory in the cylinder group of the directory they are in

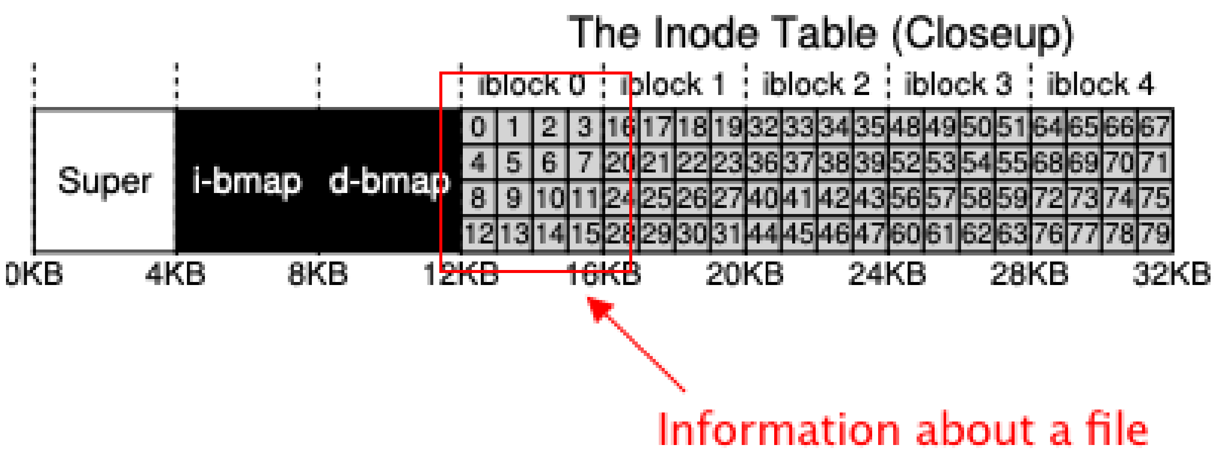
     Example

     On putting `/a/c`, `/a/d`, `/b/f`, FFS would place

     - `/a/c`, `/a/d` as close as possible in the same **cylinder group**,
     - `/b/f` located far away (in some other **cylinder group**)

14 Inode





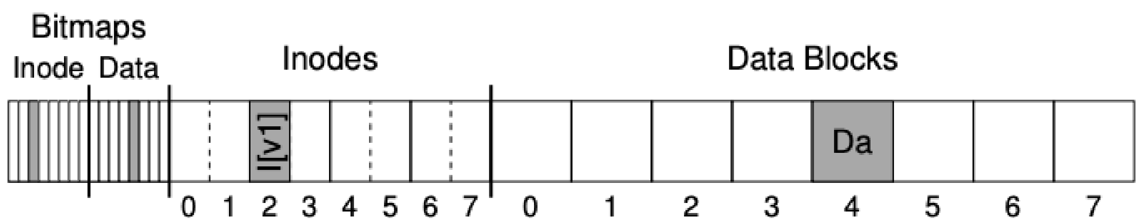
- Is a short form for **index node**
- Contains disk block location of the object's data [1]
- Contains all the information you need about a file (i.e. metadata)
  - File Type
    - e.g. regular file, directory, etc
  - Size
  - Number of blocks allocated to it
  - Protection information
    - such as who owns the file, as well as who can access it
  - Time information
    - e.g. When file was created, modified, or last accessed
  - Location of data blocks reside on disk

15 Crash Consistency

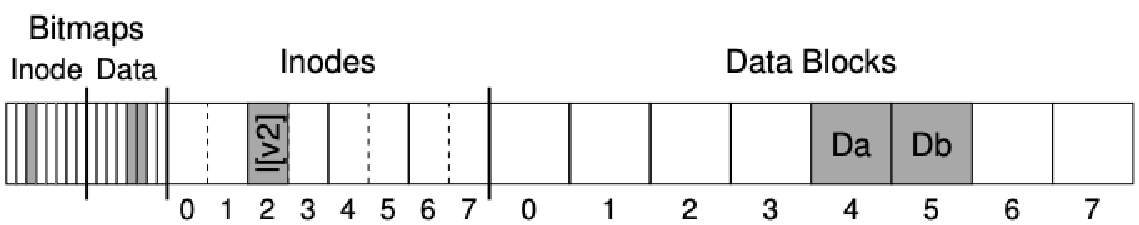
- Goal: How to update persistent data structures despite the presence of a **power loss** or **system crash**?

16 Crash Scenarios

Before



After



1) Just the data block (Db) is written to disk

- No inode that points to it
- No bitmap that says the block is allocated
- It is as if the write never occured
- There is no problem here. All is well. (In file system's point of view)

2) Just the updated inode (I[v2]) is written to disk

- Inode points to the disk where `Db` is about to be written
- No bitmap that says the block is allocated
- No `Db` is written
- Garbage data will be read
- Also creates **File-system Inconsistency**

  - Caused by on-disk bitmap telling us Db 5 is not allocated, but inode saying it does

3) Just the updated bitmap (B[v2]) is written to disk

- Bitmap indicates tht block 5 is allocated
- No inode exists at block 5
- Creates **file-system inconsistency**
- Creates **space-leak** if left as is
  - block 5 can never be used by the file system

4) Inode (I[v2]) and bitmap (B[v2]) are written to disk, and not data

- File system metadata is completely consistent (in perspective of file system)
- Garbage data will be read

5) Inode (I[v2]) and data block (Db) are written, but not the bit map

- Creates **file-system inconsistency**
- Needs to be resolved before using file system again

6) Bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2])

- Creates **file-system inconsistency** between inode and data bitmap
- Creates **space-leak** if left as is
  - Inode block is lost for future use
- Creates **data-leak** if left as is
  - Data block is lost for future use

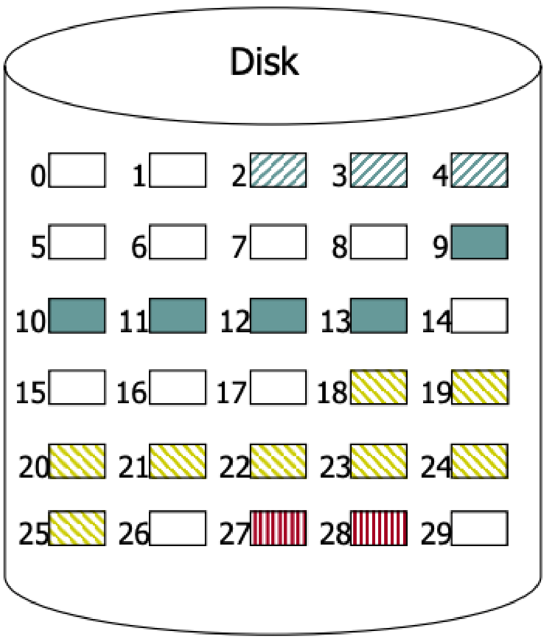
17 External Fragmentation

- Is various free holes that are generated in either your memory or disk space. [8]
- Are available for allocation, but may be too small to be of any use [8]

18 Internal Fragmentation

- Is wasted space within each allocated block [8]
- Occurs when more computer memory is allocated than is needed

sectionExtent Based File System



directory

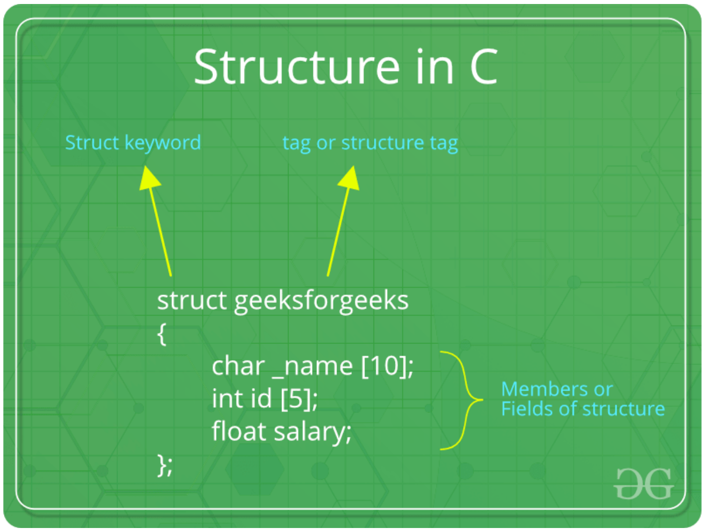
File Name	Start Blk	Length
File A	2	3
File B	9	5
File C	18	8
File D	27	2

- Is simply a disk pointer plus a length (in blocks)
  - Together, is called **extent**
- Often allows more than one extent
  - resolve problem of finding continuous free blocks
- Is less flexible but more compact
- Works well when there is enough free space on the disk and files can be laid out contiguously

Example

Linux's ext4 file system
sectionFields

- Is the members in a structure



sectionProcess List

- Is a data structure in kernel or OS
- Contains information about all the processes running in the system

sectionProcess Control Block

- Is a data structure in kernel or OS
- Contains all information about a process
- Is where the OS keeps all of a process' hardware execution state
- Generally includes
  1. Process state (ready, running, blocked)
  2. Process number
  3. Program counter: address of the next instruction
  4. CPU Registers: is saved at an interrupt
  5. CPU scheduling information: process priority
  6. Memory management info: page tables
  7. I/O status information: list of open files