File System Management

File System Implementations

Lecture 11

Overview

- File System Implementation:
  - File system layout
  - Disk organization

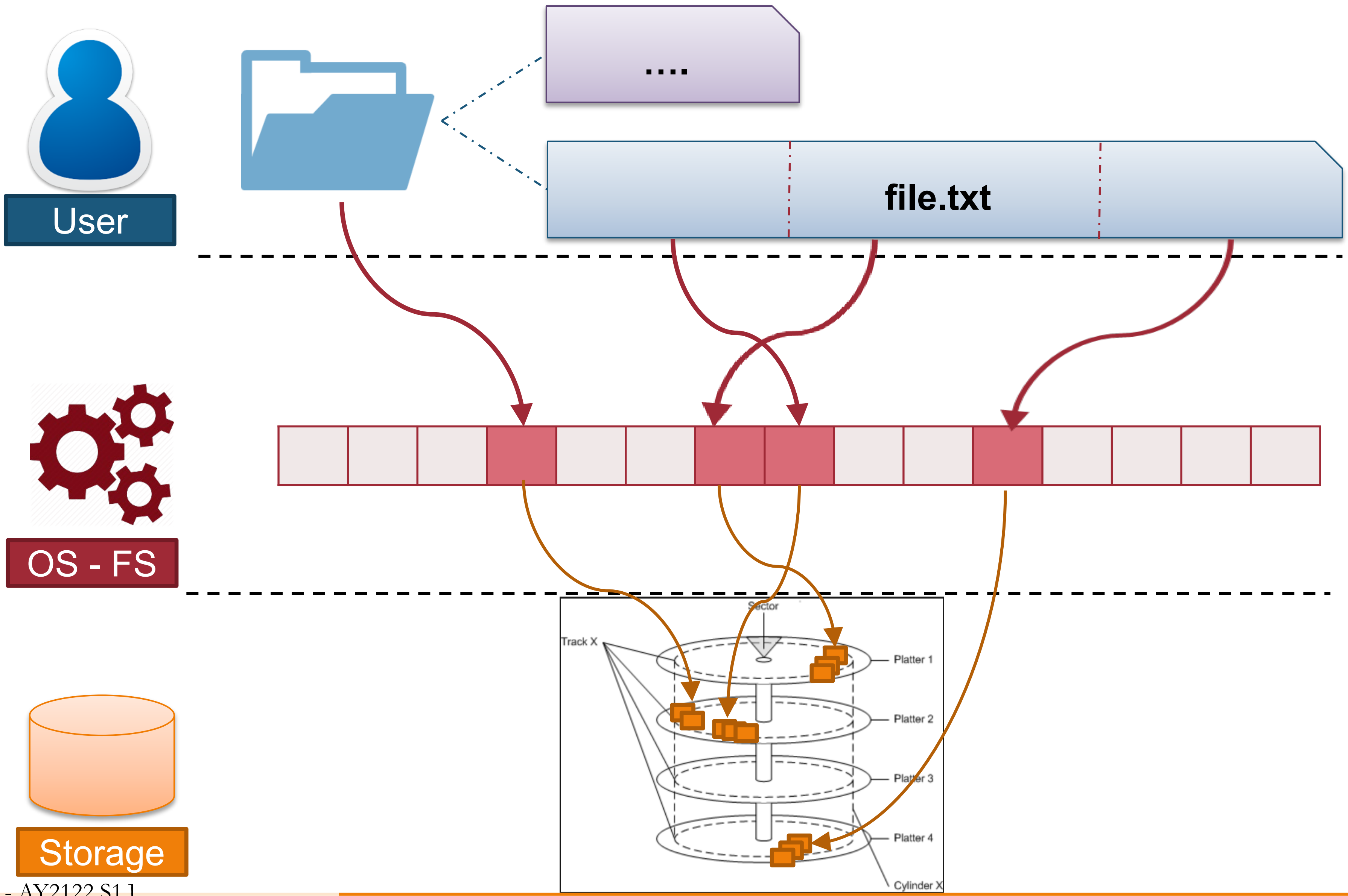
- Implementation details for:
  - File Information
  - Free Space Management
  - Directory Structure

- File System in Action

File System Implementation: Overview

- File systems are stored on storage media:
  - e.g. Hard disk, CD/DVD, SRAM etc
- Concentrate on hard disk in this lecture
  - Though the ideas are generally applicable
- General **Disk Structure**:
  - Can be treated as a 1-D array of **logical blocks**
  - Logical block:
    - Smallest accessible unit (Usually 512-bytes to 4KB)
  - Logical block is mapped into **disk sector(s)**
    - Layout of disk sector is **hardware dependent**

User ←→ OS ←→ Hardware: Views

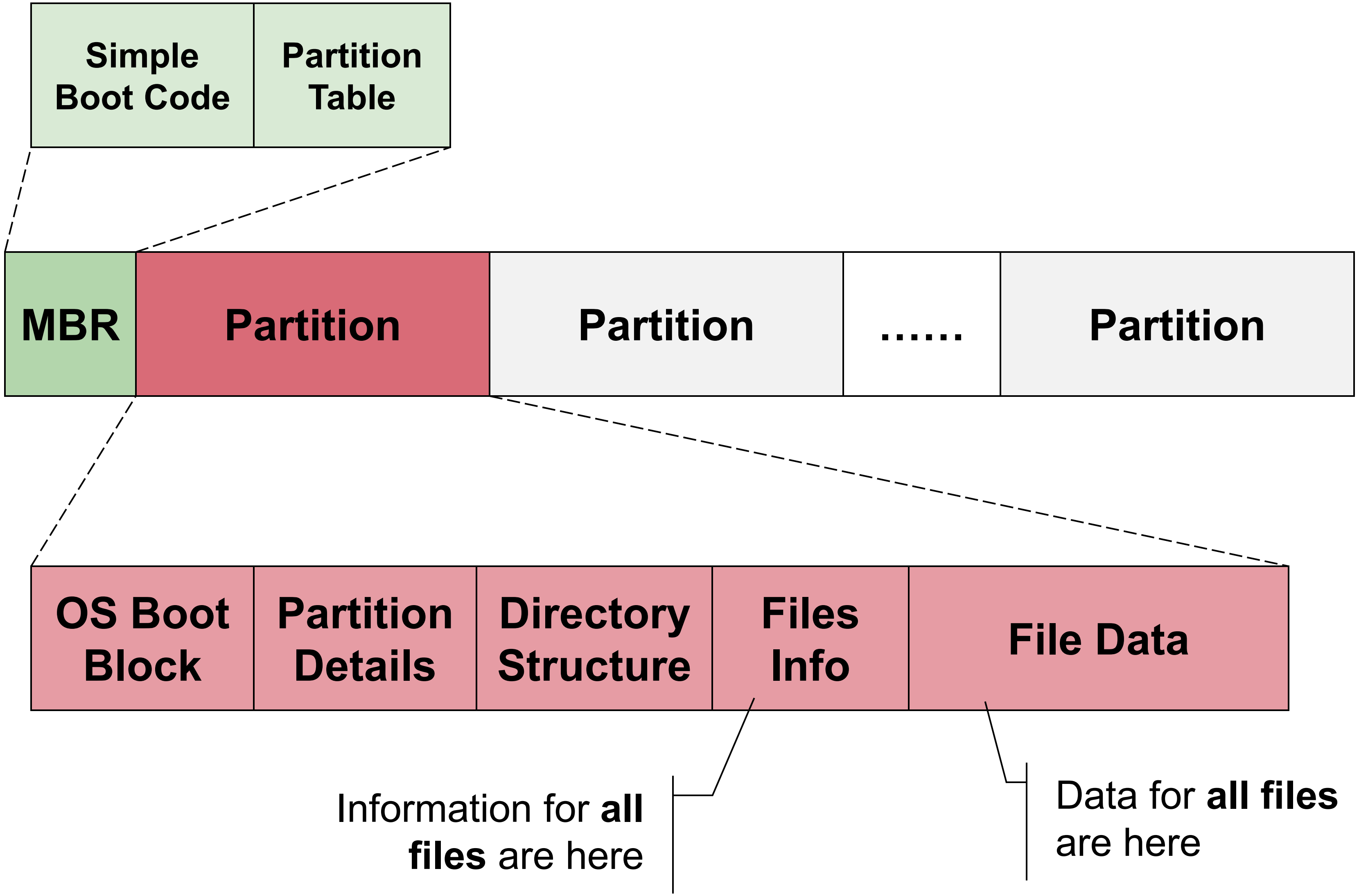


Disk Organization: Overview

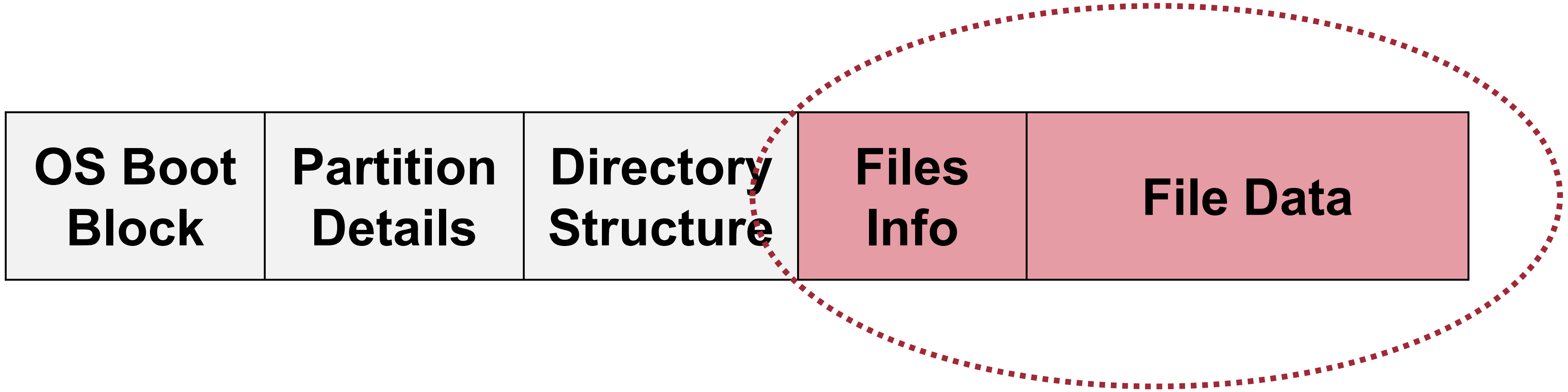
- Disk organization:
  - **M**aster **B**oot **R**ecord (**MBR**) at sector 0 with partition table
  - Followed by one or more **partitions**
    - Each partition can contain an independent **file system**

- A file system generally contains:
  - OS Boot-Up information
  - Partition details:
    - Total Number of blocks
    - Number and location of free disk blocks
  - Directory Structure
  - Files Information
  - Actual File Data

Generic Disk Organization: Illustration



Implementing File



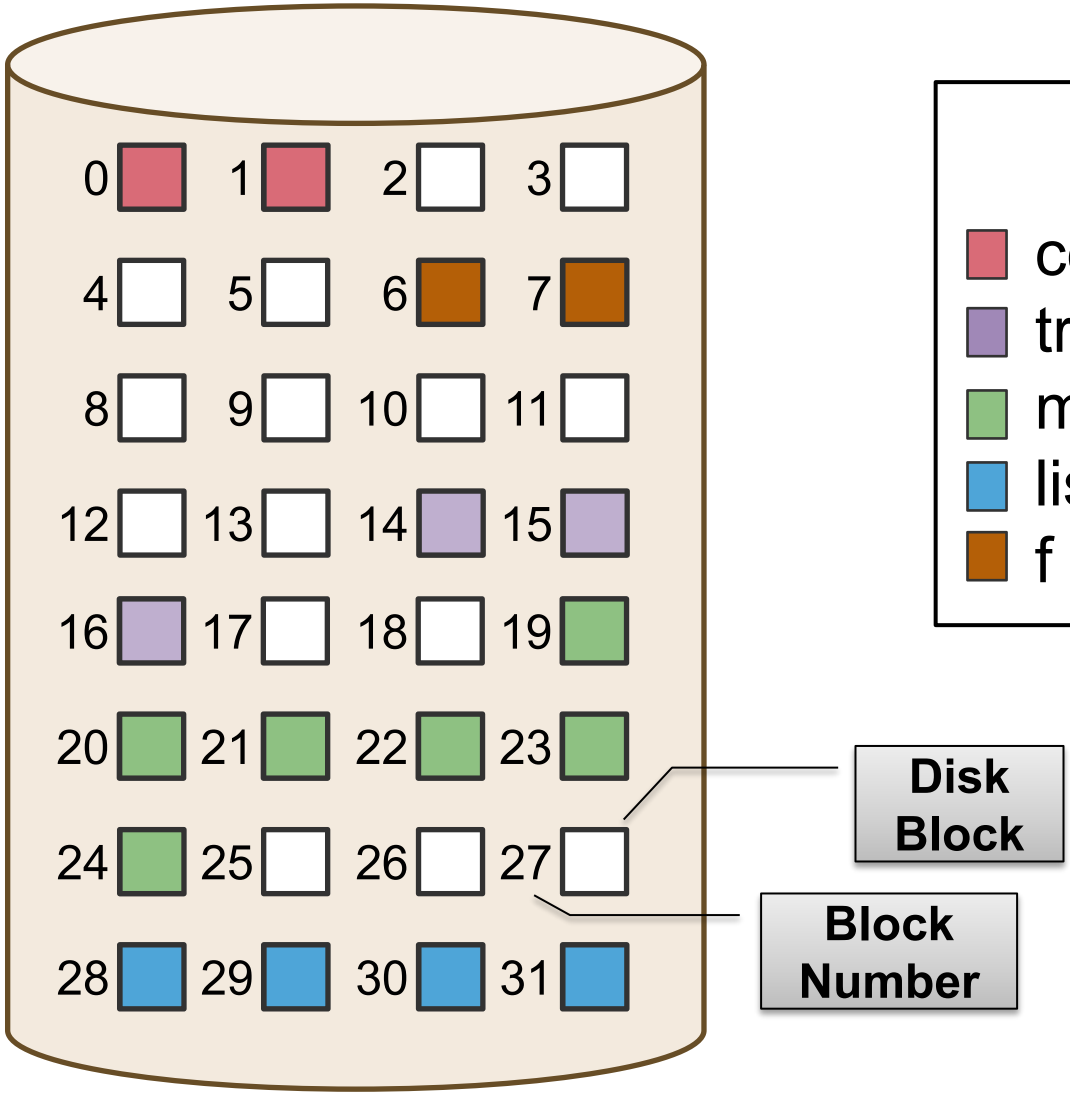
File Implementation: Overview

- Logical view of a file:
  - A collection of logical blocks
- When file size != multiple of logical blocks
  - Last block may contain wasted space
  - i.e. **internal fragmentation**
- A good file implementation must:
  - Keep track of the logical blocks
  - Allow efficient access
  - Disk space is utilized effectively
- Basically focuses on **how to allocate** file data on disk

File Block Allocation 1: **Contiguous**

- **General Idea:**
  - Allocate consecutive disk blocks to a file
- **Pros:**
  - Simple to keep track:
    - Each file only needs: Starting block number + Length
  - Fast access (only need to seek to first block)
- **Cons:**
  - **External Fragmentation**
    - Think of each file as a variable-size "partition"
    - Over time, with file creation/deletion, disk can have many small "holes"
  - File size need to be specified in advance

Contiguous Block Allocation



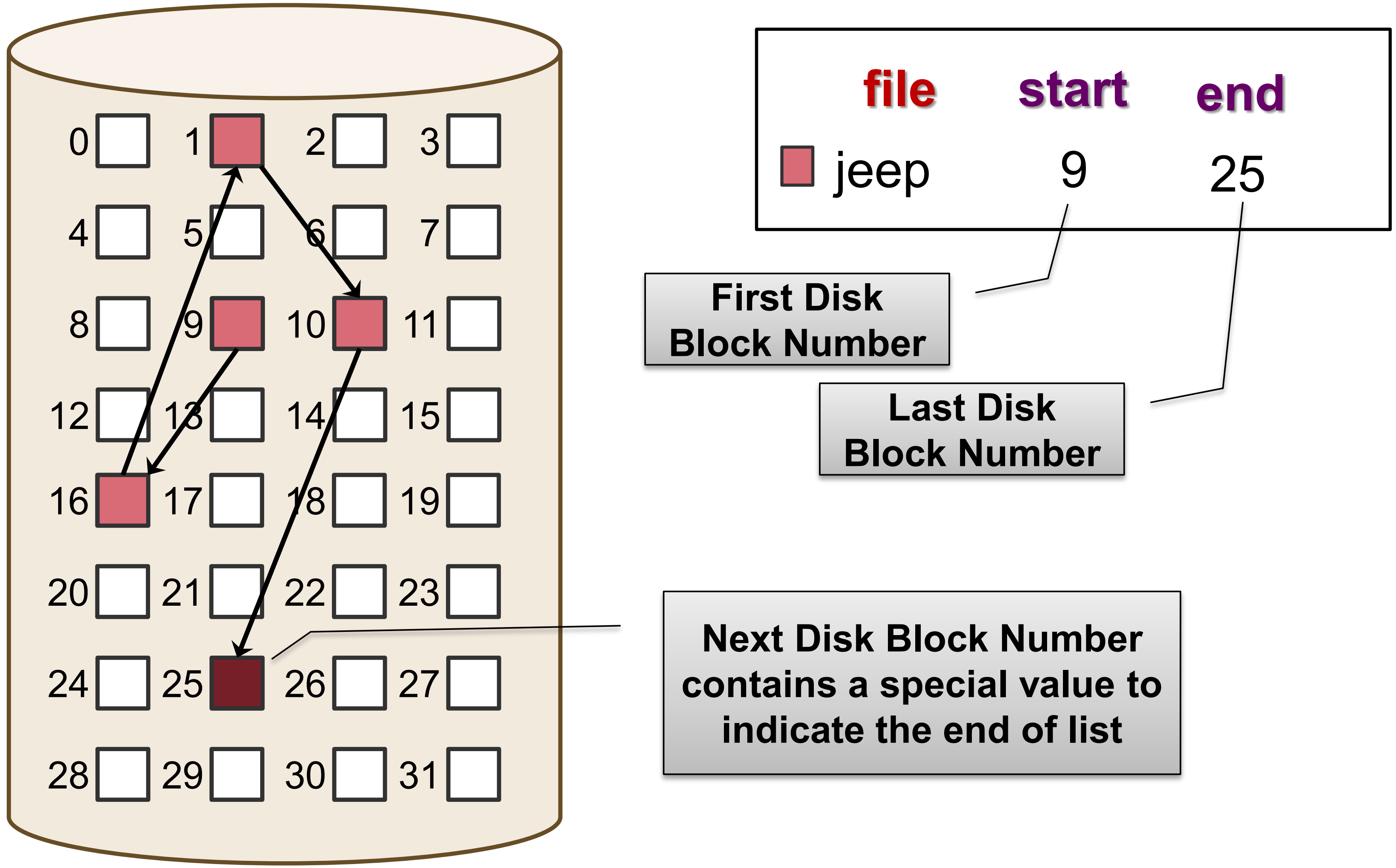
Directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

File Block Allocation 2: **Linked List**

- **General Idea:**
  - Keep a linked list of disk blocks
  - Each disk block stores:
    - The next disk block number (i.e. act as **pointer**)
    - Actual file data
  - File information stores:
    - First and last disk block number
- **Pros:**
  - Solve fragmentation problem
- **Cons:**
  - Random access in a file is very slow
  - Part of disk block is used for pointer
  - Less reliable (what if one of the pointers is incorrect?)

Linked List Allocation

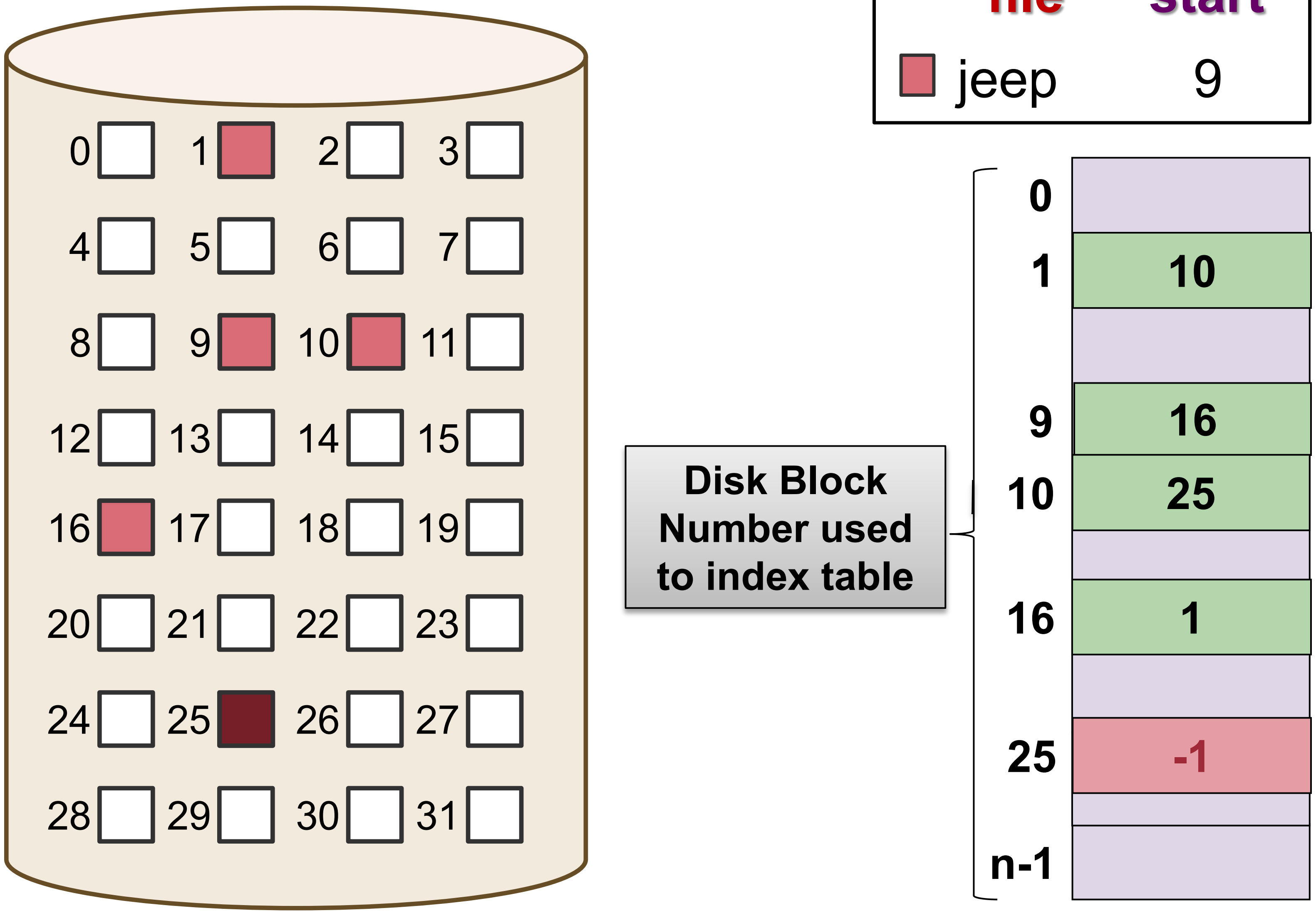


File Block Allocation 2: Linked List V2.0

- **General Idea:**
  - Move all the block pointers into a single table
    - known as **F**ile **A**llocation **T**able **(FAT)**
    - FAT is in memory at all time
  - Simple yet efficient
    - Used by MS-DOS
- **Pros:**
  - Faster Random Access
    - The linked list traversal now takes place in memory
- **Cons:**
  - FAT keep tracks of **all disk blocks** in a partition
    - Can be huge when disk is large
    - Consume valuable memory space

FAT Allocation

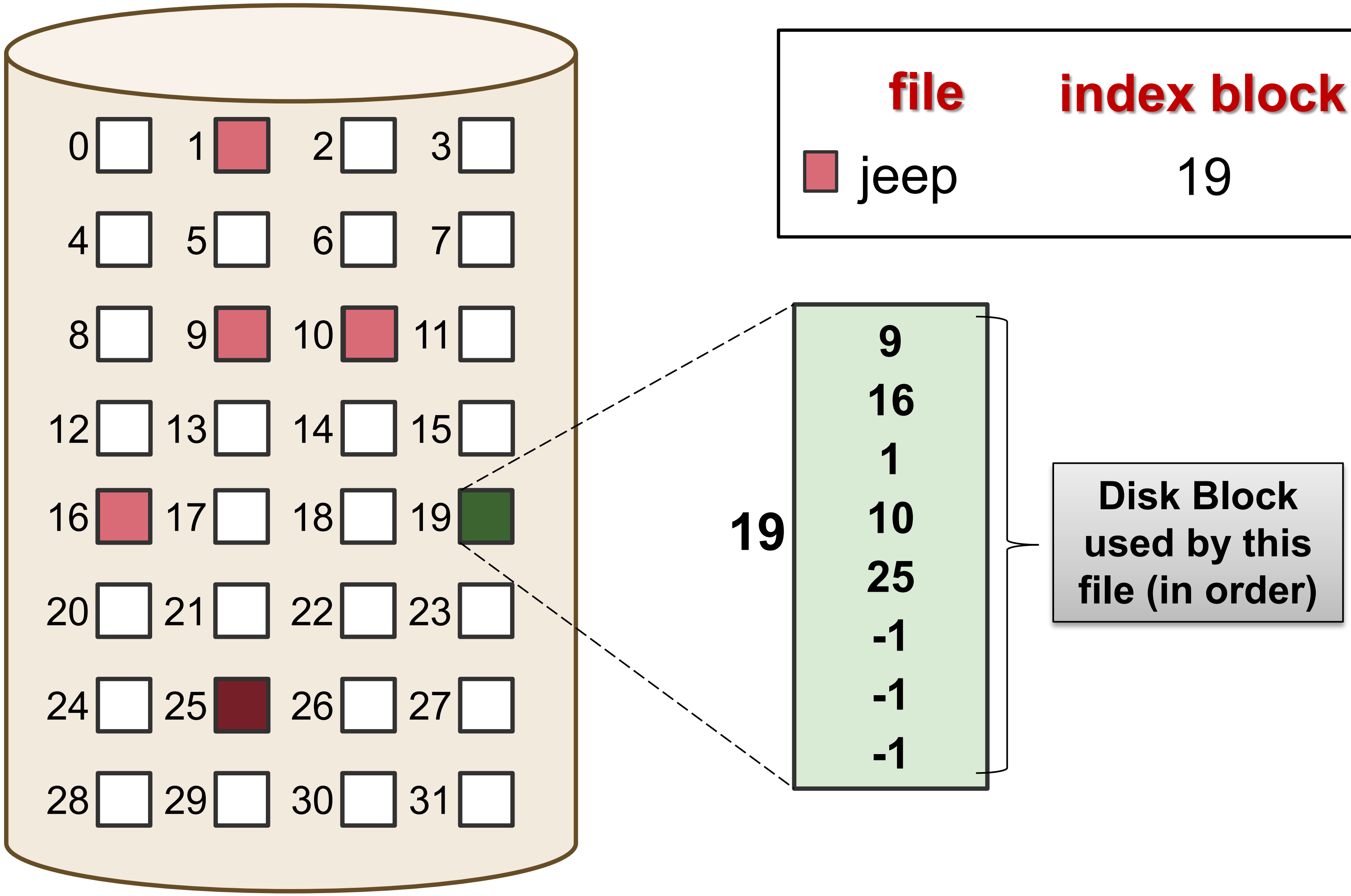
	file	start
	jeep	9



File Block Allocation 3: **Indexed Allocation**

- **General Idea:**
  - Each file has an **index block**
    - An **array of** disk block addresses
    - IndexBlock[N] == Nth Block address
- **Pros:**
  - Lesser memory overhead
    - Only index block of opened file needs to be in memory
  - Fast direct access
- **Cons:**
  - Limited maximum file size
    - Max number of blocks == Number of index block entries
  - Index block overhead

Indexed Allocation



file	index block
jeep	19

Free Space Management

OS Boot Block	Partition Details	Directory Structure	Files Info	File Data

Free Space Management: Overview

- To perform file allocation:
  - Need to know which disk block is free
  - i.e. maintain a **free space list**
- Free space management:
  - Maintain free space information
  - **Allocate:**
    - Remove free disk block from free space list
    - Needed when file is created or enlarged (appended)
  - **Free:**
    - Add free disk block to free space list
    - Needed when file is deleted or truncated

Free Space Management: **Bitmap**

- Each disk block is represented by 1 bit
  - E.g. 1 == free, 0 == occupied
- Example:

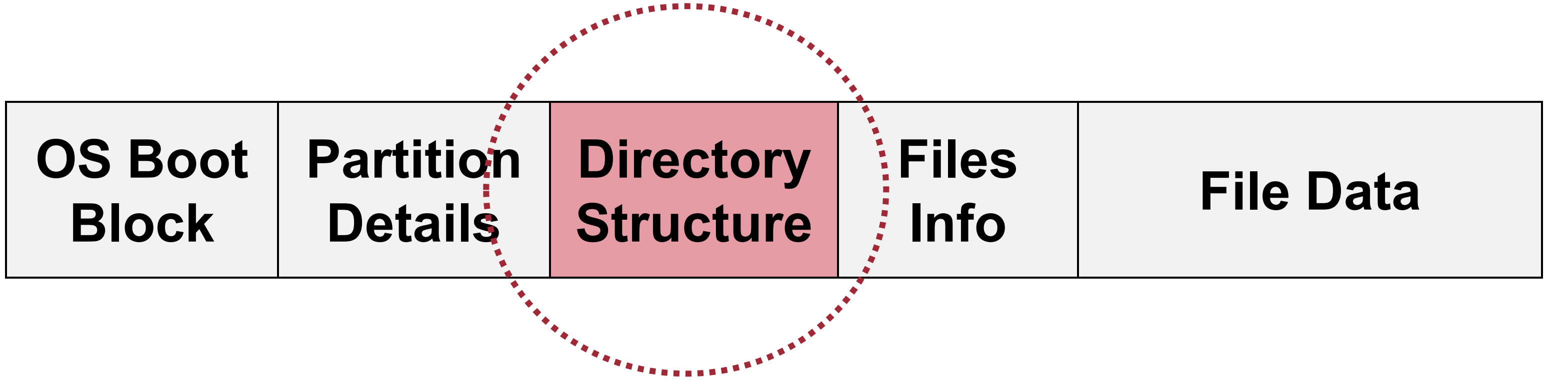
```
0 1 0 1 1 1 0 0 1 0 1 1 .......
```

  - Occupied Blocks = **0, 2, 6, 7, 9, …**
  - Free Blocks = **1, 3, 4, 5, 8, 10, 11, …**
- **Pros:**
  - Provide a good set of manipulations
    - E.g. can find the first free block, n-consecutive free blocks easily by bit level operation
- **Cons:**
  - Need to keep in memory for efficiency reason

Free Space Management: **Linked List**

- Use a linked list of disk blocks:
  - Each disk block contains:
    - A number of free disk block numbers, or
    - A pointer to the next free space disk block
- **Pros:**
  - Easy to locate free block
  - Only the first pointer is needed in memory
    - Though other blocks can be cached for efficiency
- **Cons:**
  - High overhead
    - Can be mitigated by storing the free block list in free blocks!

Implementing Directory



Directory Structure: Overview

- The main tasks of a directory structure:
  1. Keep tracks of the files in a directory
     - Possibly with the file metadata
  2. Map the file name to the file information
- Remember:
  - File must be opened before use
    - Something like `open( "data.txt" );`
  - The purpose of the open operation:
    - Locate the file information using pathname + file name
- Path name
  - List of directory names traversed from root
  - E.g. `/dir2/dir3/data.txt`

Directory Structure: Overview (cont)

- Given a full path name:
  - Need to recursively search the directories along the path to arrive at the file information
- Example:
  - Full path name: **`/dir2/dir3/data.txt`**
  1. Find "**`dir2`**" in directory "/"
     - Stop if not found (or incorrect type)
  2. Find "**`dir3`**" in directory "**`dir2`**"
     - Stop if not found (or incorrect type)
  3. Find "**`data.txt`**" in directory "**`dir3`**"
     - Stop if not found (or incorrect type)
- Sub-directory is usually stored as file entry with special type in a directory

Directory Implementation: **Linear List**

- Directory consists of a list:
  - Each entry represents a file:
    - Store file name (minimum) and possibly other metadata
    - Store file information or pointer to file information

- Locate a file using list:
  - Requires a linear search
    - Inefficient for large directories and/or deep tree traversal
  - Common solution:
    - Use cache to remember the latest few searches
      - User usually move up/down a path

Directory Implementation: **Hash Table**

- Each directory contains a
  - Hash table of size **`N`**
- To locate a file by file name:
  - File name is hashed into index **`K`** from **`0`** to **`N-1`**
  - **`HashTable[K]`** is inspected to match file name
    - Usually chained collision resolution is used
    - i.e. file names with same hash value is chained together
      - to form a linked list with list head at **`HashTable[ K ]`**
- **Pros:**
  - Fast lookup
- **Cons:**
  - Hash table has limited size
  - Depends on good hash function

Directory Implementation: **File Information**

- File information consists of:
  - File name and other metadata
  - Disk blocks information
    - As discussed in the file allocation schemes earlier

- Two common approaches:

1. Store everything in directory entry
   - A simple scheme is to have a fixed size entry
     - All files have the same amount of space for information
2. Store only file name and points to some data structure for other info

How is it done?

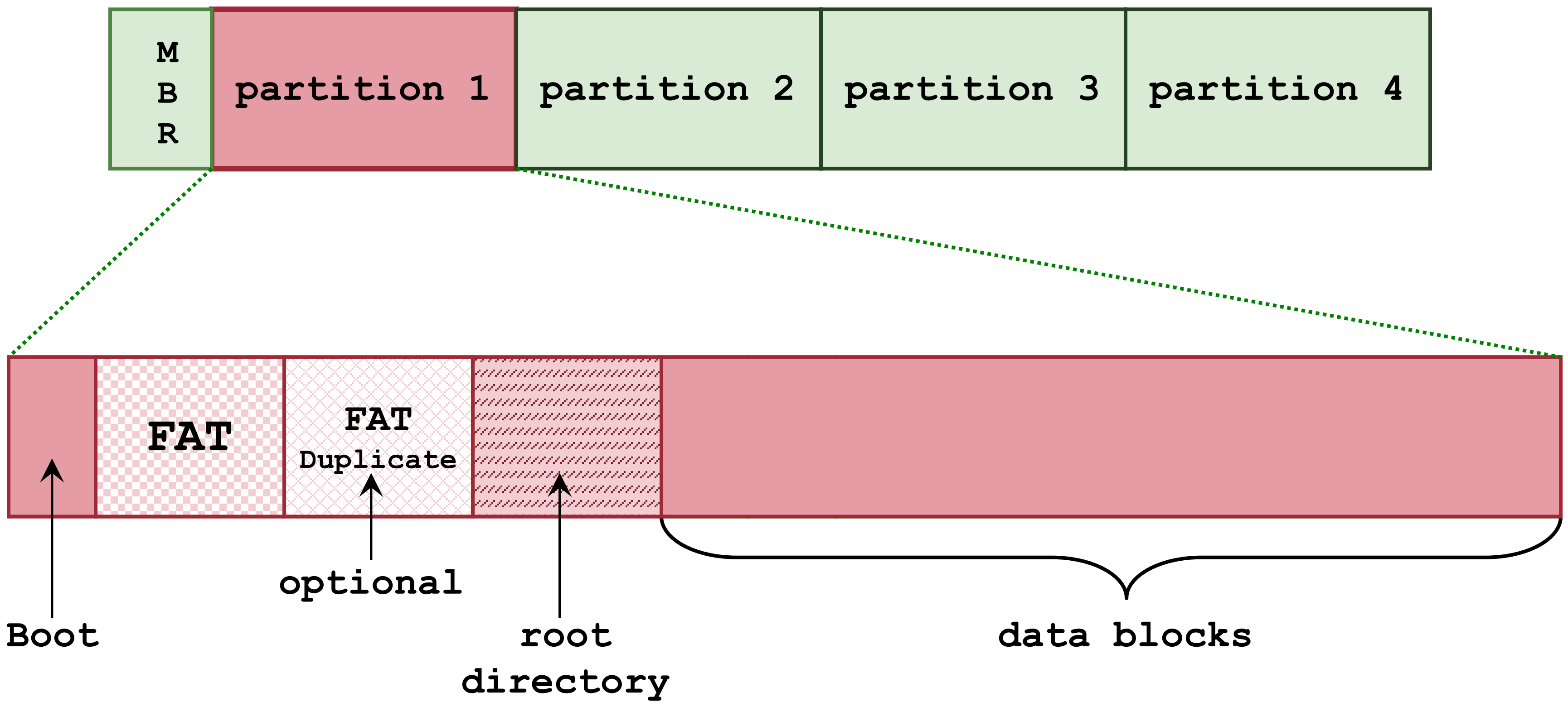
CASE STUDIES

Microsoft FAT File System

- Brief history:
  - Used in MS-DOS starting in 1980s
  - Shipped as default file system until Windows XP (2001)
  - Several major versions:
    - FAT12 → FAT16 → FAT32
  - FAT32 still very prevalent nowadays
    - Supported across all major OSes
    - Used in portable drives, gaming console, digital camera, etc

- Simple and serves as a good introduction

Microsoft FAT File System Layout



File Data and File Allocation Table

- File data are allocated to:
  - A number of data blocks / data block clusters
  - Allocation info is kept as a **linked list**
    - All data block pointers kept separately in the File Allocation Table

- File Allocation Table (FAT):
  - One entry per data block/cluster
  - Store disk block information
    - Free? Next block (if occupied)? Damaged?
  - OS will cache in RAM to facilitate linked list traversal

File Allocation Table: Illustration

- FAT entry contains either:
  - **FREE** code (block is unused)
  - **Block number** of next block
  - **EOF** code (i.e. NULL pointer)
  - **BAD** block (block is unusable, i.e. disk error)

- Example:
  - Block 3 → 5 → 8 → EOF
  - Block 4 and 9 are free
  - Block 1 is unusable

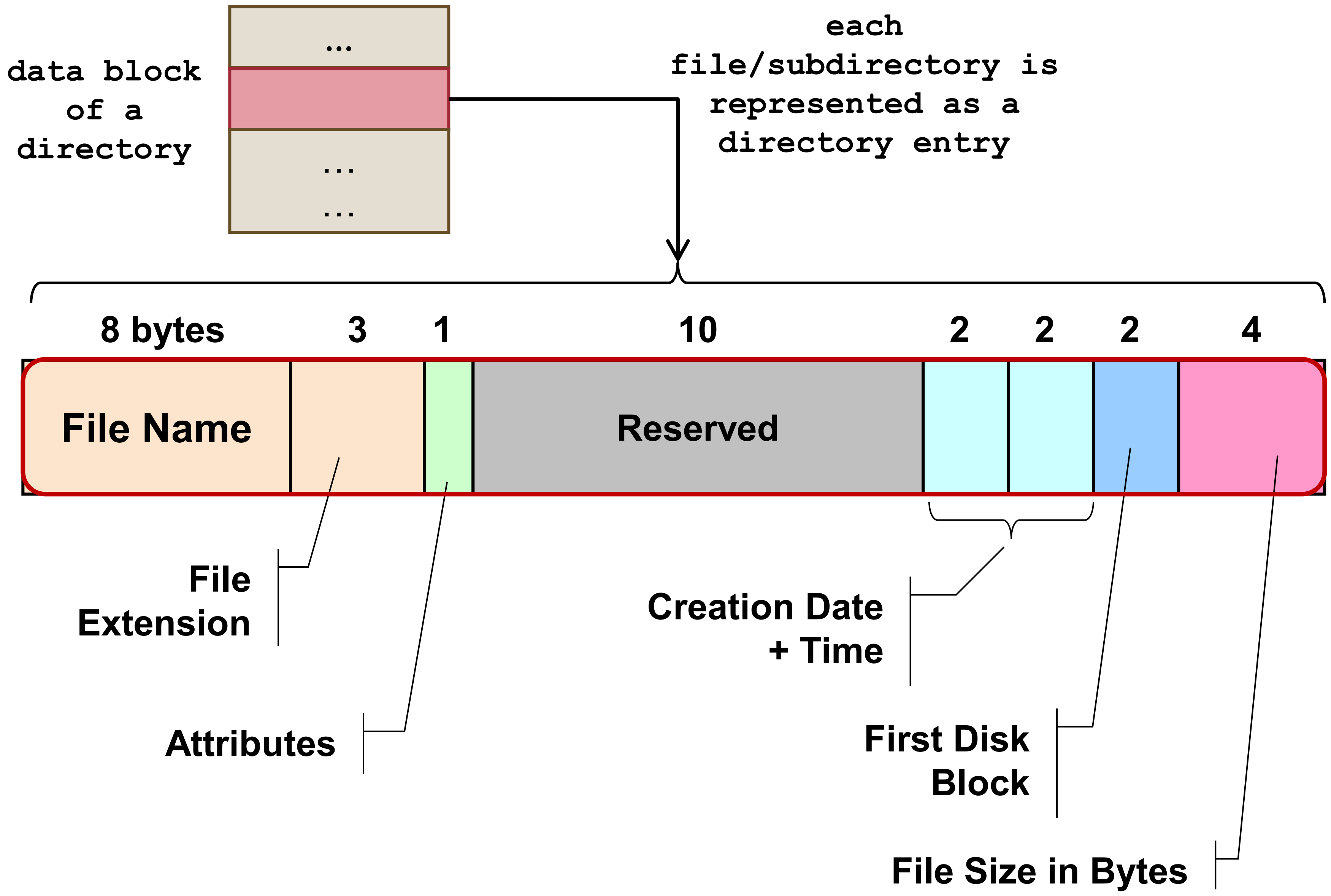
0	
1	BAD
2	EOF
3	5
4	FREE
5	8
6	7
7	EOF
8	EOF
9	FREE
	…
	…
	…

FAT

Directory Structure and File Information

- Directory (Folder) is represented as:
  - Special type of file
  - Root directory is stored in a special location
    - Other directories are stored in the data blocks
  - Each file/subdirectory within the folder:
    - Represented as **directory entry**
- Directory Entry:
  - Fixed-size 32-bytes per entry
  - Contains:
    - Name + Extension
    - Attributes (Read-Only, Directory/File flag, Hidden etc)
    - Creation Date + Time
    - First disk block + File Size

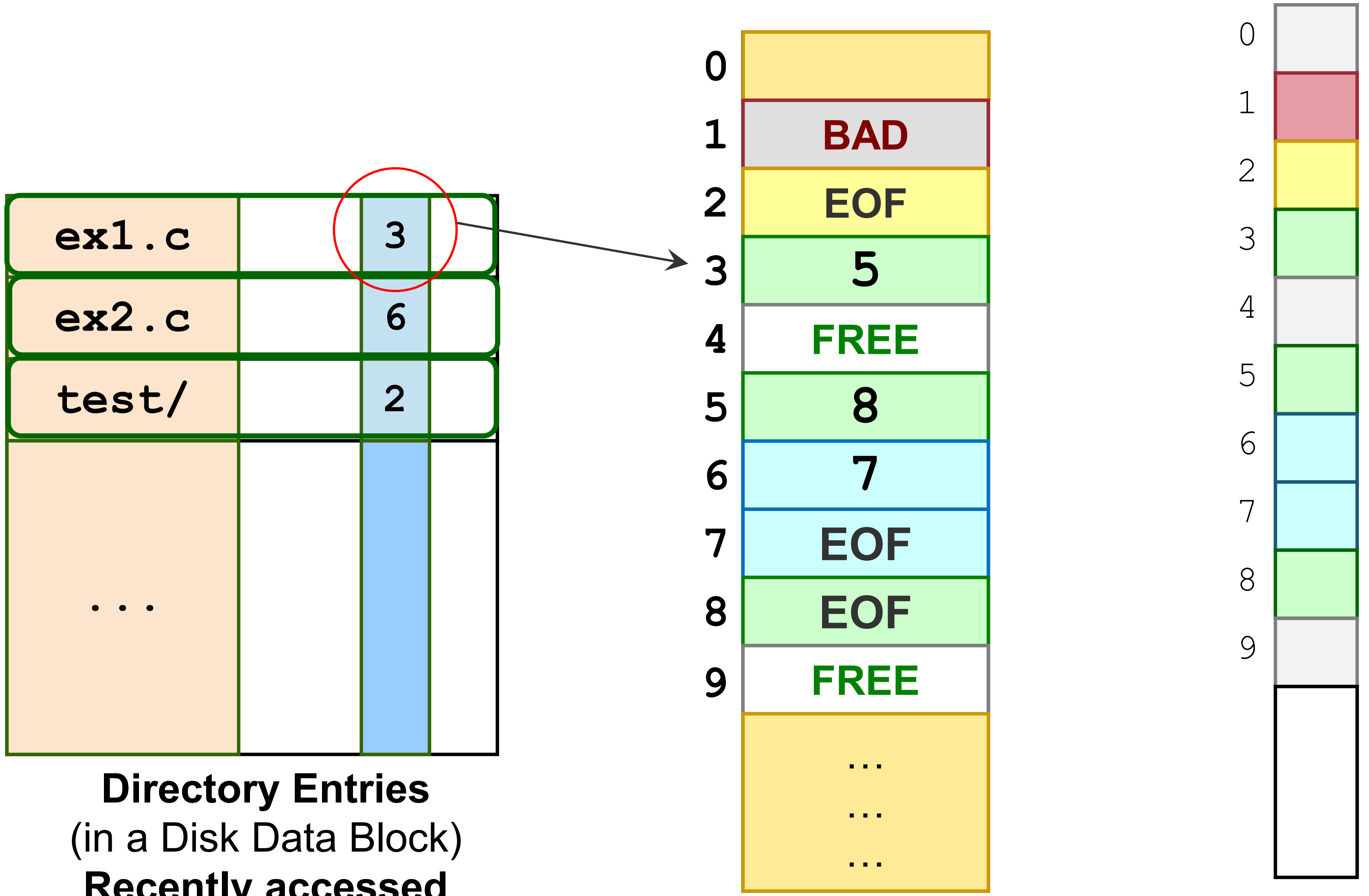
Directory Entry Illustration



Directory Entry Fields

- File Name + Extension
  - Limited to 8+3 characters
  - The first byte of file name may have special meaning:
    - Deleted, End of directory entries, Parent directory, etc.
- File Creation Time and Date:
  - Year is limited to 1980 to 2107
  - Accuracy of second is ±2 seconds
- First Disk Block Index:
  - Different variants uses different number of bits:
    - 12, 16 and 32 bits for FAT12, FAT16 and FAT32 respectively

FAT FS: Putting the parts together…



Directory Entries
(in a Disk Data Block)
Recently accessed directory can be cached

FAT
(in RAM)

Data Blocks
(in Disk)

FAT FS: Putting the parts together…

1. Use first disk block number stored in directory entry to find the starting point of the linked disk blocks
2. Use FAT to find out the subsequent disk blocks number
   - Terminated by special value (**EOF**)
3. Use disk block number to perform actual disk access on the data blocks

- For a directory, the disk blocks contain:
  - Directory entries for the files/subfolders within that directory

Extended-2 File System (Ext2)

- One of the most popular file systems used in Linux

- A nice case study:
  - Uses many techniques discussed
  - Embedded a few traditional Unix FS ideas
    - Serves as a good starting point to understand other Unix related FS

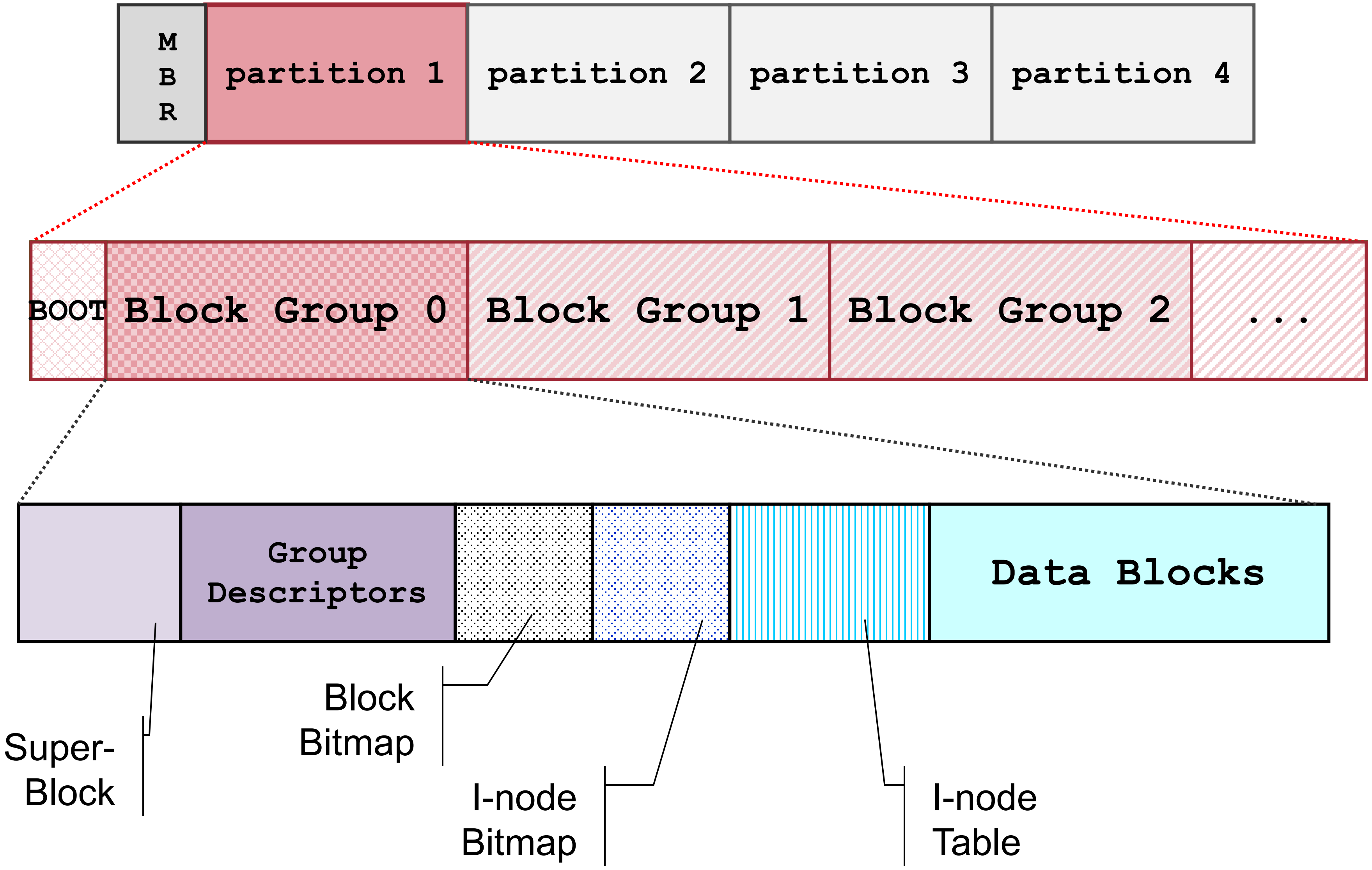
- However, Ext2 is quite intricate:
  - Will concentrate on important/relevant parts only

Ext2 FS: Overview

- Disk space is split into **Blocks**
  - Correspond to one or more disk sector
  - i.e. Similar to disk cluster in FAT FS
- Blocks are grouped into **Block Groups**

- Each file/directory is described by:
  - A **single** special structure known as **I-Node** (Index Node)
- I-Node contains:
  - File metadata (access right, creation time etc)
  - Data block addresses

Ext2 FS: Layout



Partition Information

- **Superblock**
  - Describe the whole file system
  - Includes:
    - Total I-Nodes number, I-Nodes per group
    - Total disk blocks, Disk Blocks per group
    - etc
  - Duplicated in each block group for redundancy
- **Group Descriptors**
  - Describe each of the block group
    - Number of free disk blocks, free I-nodes
    - Location of the bitmaps
  - Duplicated in each block group as well

Partition Information (cont)

- **Block Bitmap**
  - Keep track of the usage status of blocks of this block group (1 = Occupied, 0 = Free)
- **I-Node Bitmap**
  - Keep track of the usage status of I-Nodes of this block group (1 = Occupied, 0 = Free)
- **I-Node table**
  - An array of I-Nodes
    - Each I-Node can be access by a unique index
  - Contains only I-Nodes of this block group

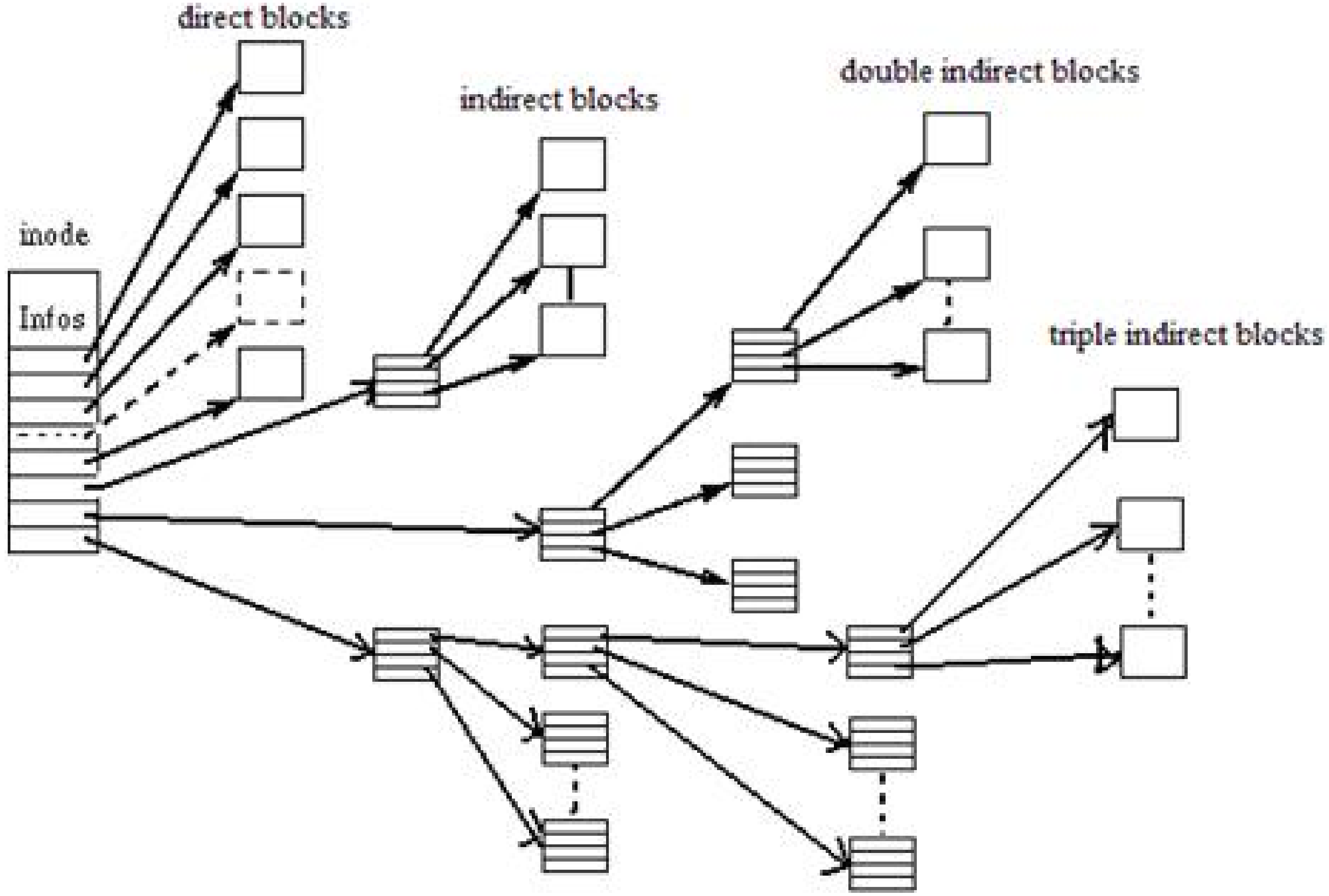
Ext2: I-Node Structure (128 Bytes)

Mode (2)	File type (regular, directory, special, etc) + File permission
Owner Info (4)	User Id (2 bytes) + Group Id (2 bytes)
File Size (4/8)	File Size in bytes. Larger for regular file (8 bytes)
Timestamps (3 x 4)	Creation, Modification & Deletion timestamps
Data Block Pointers (15 x 4)	Indices of data blocks. 12 x direct, 1 x indirect, 1 x double indirect, 1 x triple indirect
Reference Count (2)	Number of time this I-Node is referenced by directory entry
… Other … … Fields …	

Multilevel Data Blocks

- Allow larger file size
- Multilevel index:
  - Similar idea **as multi-level paging**
  - First level index block points to a number of **second level index blocks**
    - Each second level index blocks point to actual disk block
  - Can be generalized to any number of levels
- I-node has a combination of direct indexing and multi-level index scheme

I-Node Structure: Data Blocks



- 12 **direct pointers** that point to disk block directly
- 1 **single indirect block**
  - which contains a number of direct pointers
- 1 **double indirect block**
  - which points to a number of **single indirect blocks**
- 1 **triple indirect block**
  - which points to a number of **double indirect blocks**
- A combination of efficiency (for small file) and flexibility (still allow large file)

I-Node Data Block Example

- The design of I-Node allows:
  - Fast access to small file
    - The first 12 disk blocks is directly accessible
  - Flexibility in handling huge file
- Example:
  - Each disk block address is 4 bytes
  - Each disk block is 1KiB
    - So, indirect block can store 1KiB/4 = 256 addresses
  - Maximum File Size:

    = Direct blocks + single indirect + double indirect + triple indirect

    = 12 x 1 KiB + 256 x 1KiB + 256^2 x 1KiB + 256^3 x 1KiB

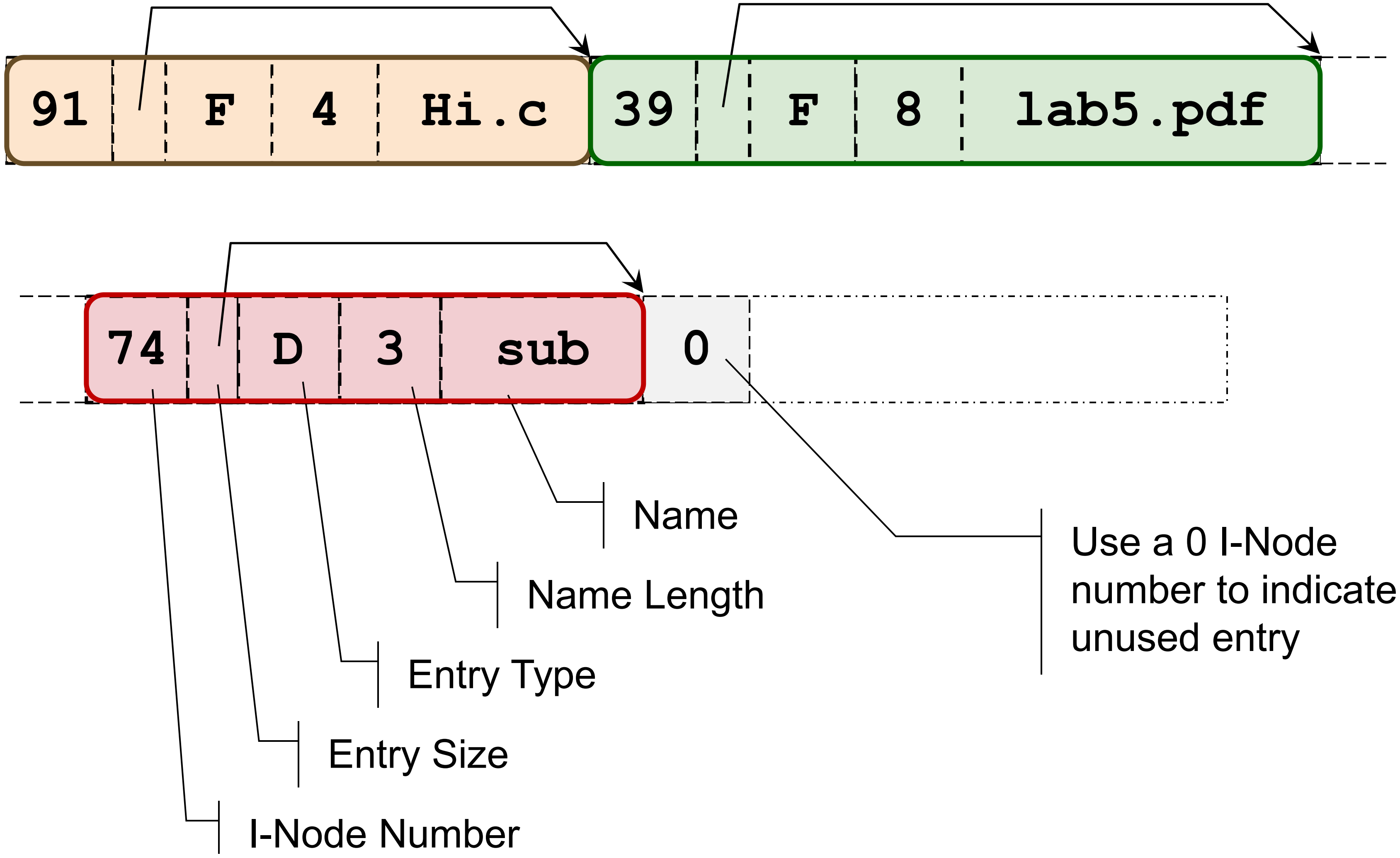
    = 16843020 KiB (16 GiB)

Directory Structure

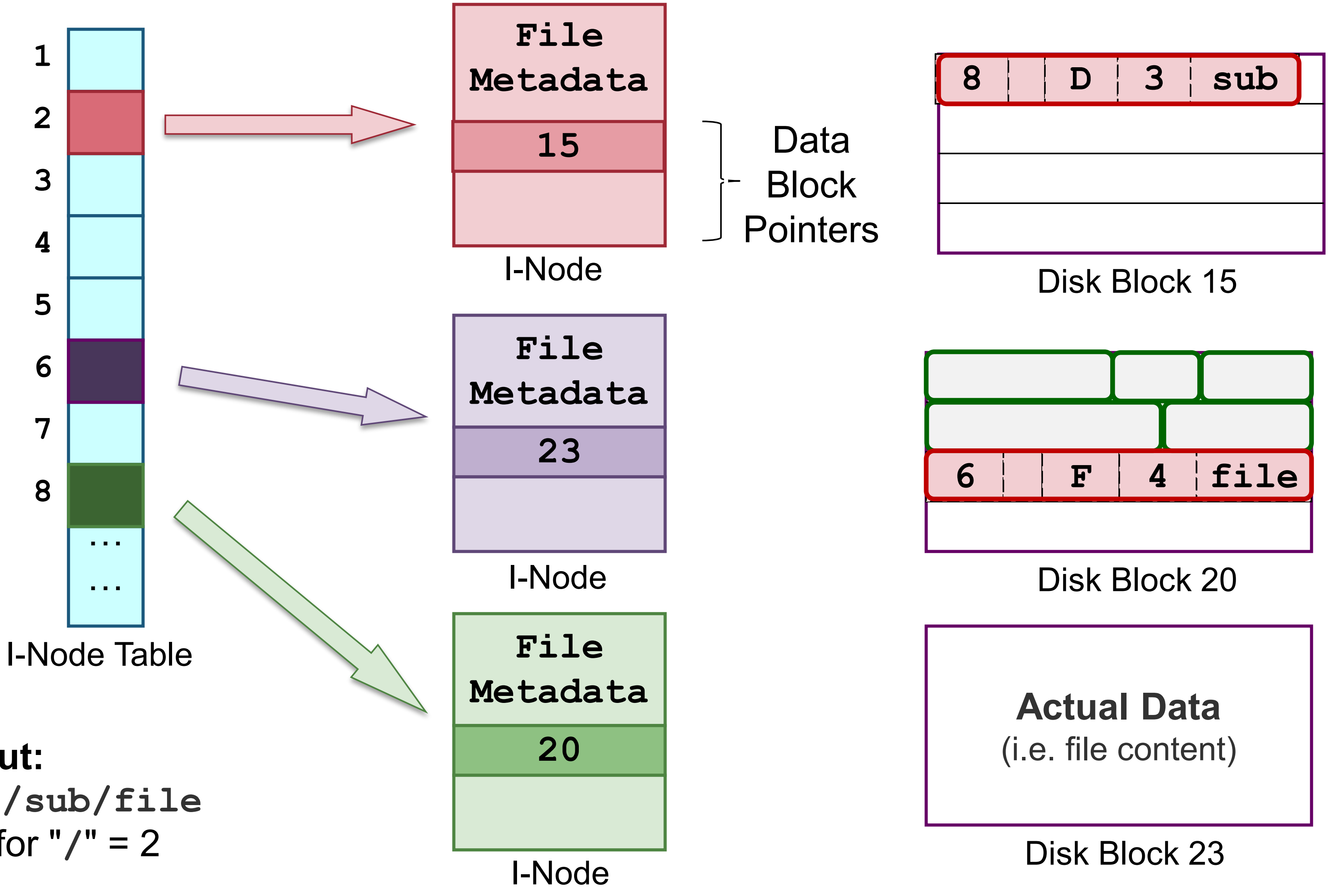
- Data blocks of a directory stores:
  - A **linked list of directory entries** for file/subdirectories information within this directory

- Each directory entry contains:
  - I-Node number for that file/subdirectory
  - Size of this directory entry
    - For locating the next directory entry
  - Length of the file/subdirectory name
  - Type: File or Subdirectory
    - Other special file type is also possible
  - File/Subdirectory name (up to 255 characters)

Directory Entry (Illustration)



Ext2: Putting The Parts Together



❖ **Try it out:**
Access `/sub/file`
I-Node for "/" = 2

Ext2 FS: Putting the parts together…..

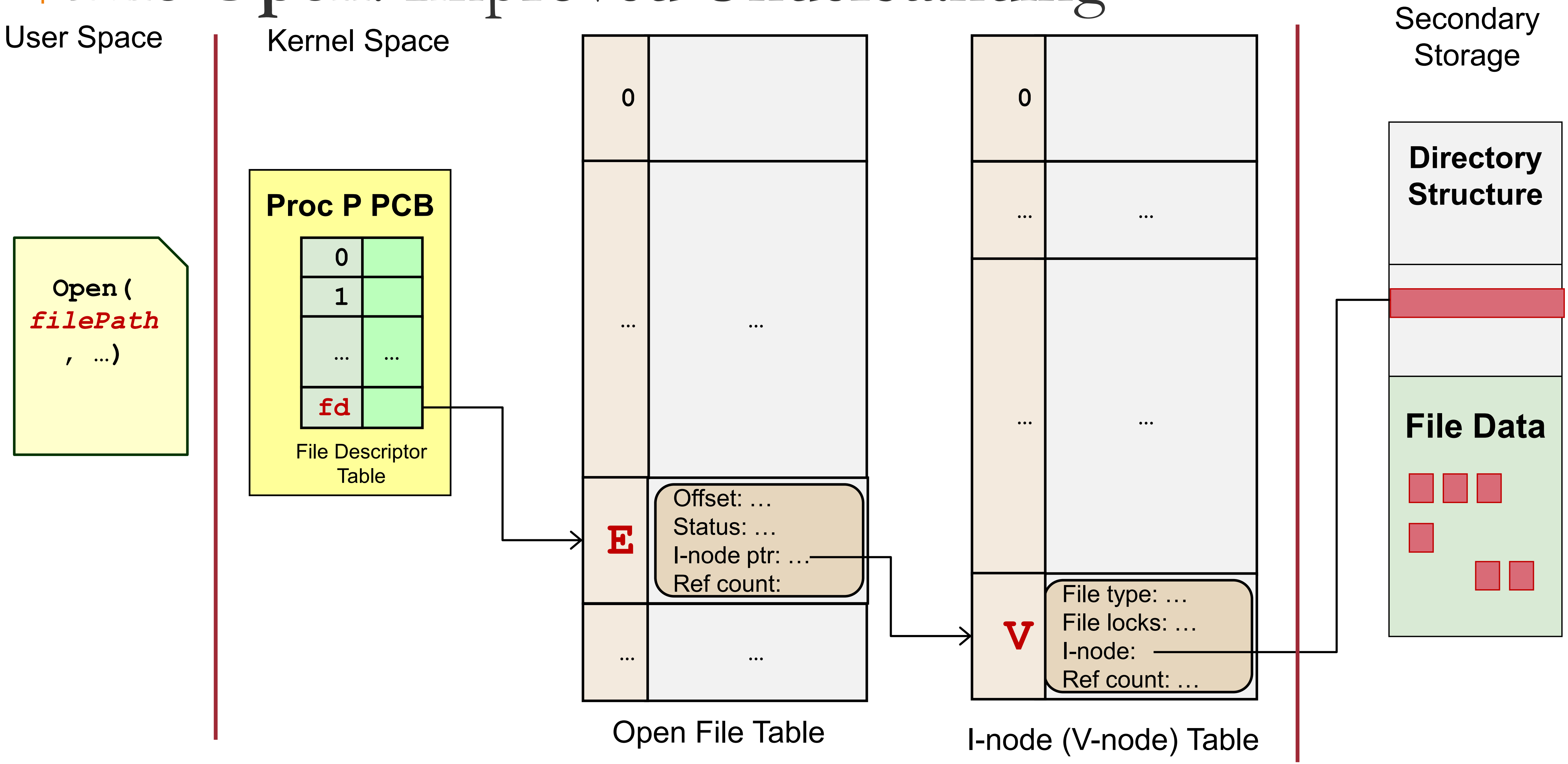
- Give a pathname, e.g. "`/sub/file`"

1. Let **CurDir** = "`/`"
   - Root directory usually has a fixed I-Node number (e.g. 2)
   - Read the actual I-Node
2. Look at the next part in pathname:
   - If it is a directory, e.g. "`sub/`"
     - Locate the directory entry in **CurDir**
     - Retrieve I-Node number, then read the actual I-Node
     - **CurDir** = next part in pathname
     - Goto Step 2.
   - Else //it is a file
     - Locate the directory entry in CurDir
     - Retrieve I-Node number, then read the actual I-Node

Walkthrough on file operation: **Open**

- Process **`P`** open file **`/…/…/…/F`**:
  - Use full pathname to locate file **`F`**
    - If not found, open operation terminates with error
    - When **`F`** is located, its file information is loaded into a new entry **`E`** in system-wide table
    - Creates an entry in **`P`**'s table to point to **`E (file descriptor)`**
    - Return the file descriptor of this entry
- The returned file descriptor is used for further read/write operation

File Open: Improved Understanding



Ext2: **Common Tasks**

- Deleting a file:
  - Remove its directory entry from the parent directory:
    - Point the previous entry to the next entry / end
    - To remove first entry: Blank record is needed
  - Update I-node bitmap:
    - Mark the corresponding I-node as free
  - Update Block bitmap:
    - Mark the corresponding block(s) as free
- Question:
  - Is it possible to "undelete" a file under ext2?
  - What if the system crashes between the steps?

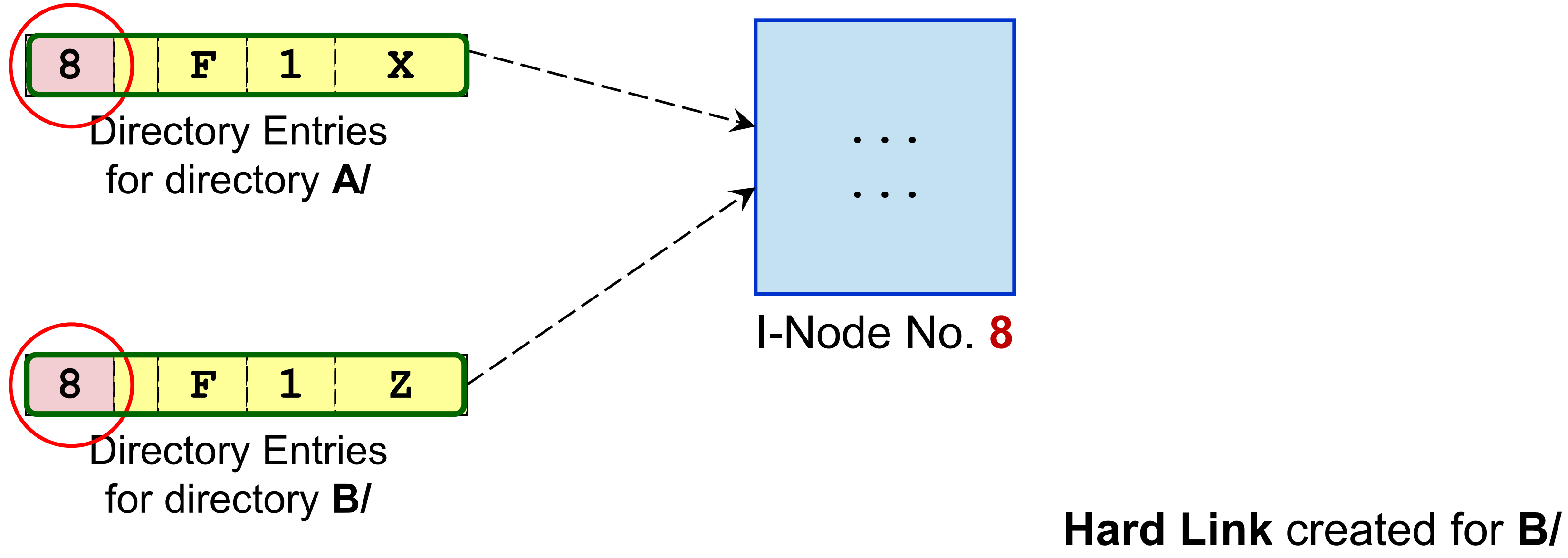
Hard / Symbolic Link with I-Node

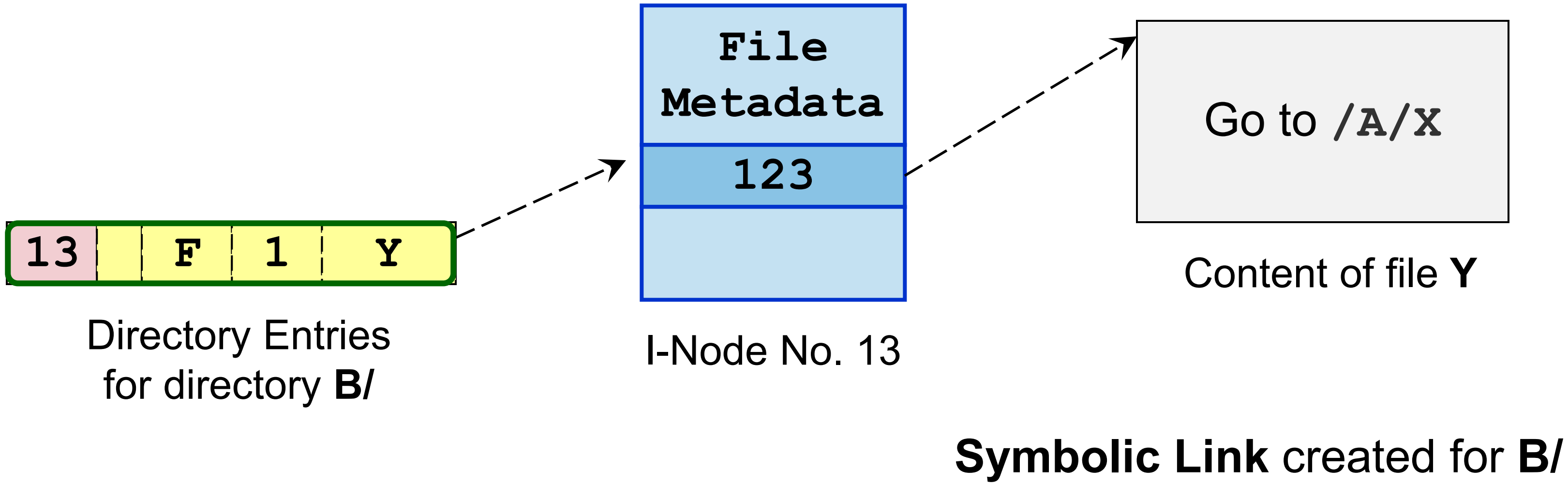
- Scenario:
  - Directory `A` contains file `X`, with I-Node# `XN`
  - Directory `B` wants to share `X`

- **Hard link:**
  - Creates a directory entry in `B` which use the **same I-Node number** `XN`
  - Can have different filename

- **Symbolic Link:**
  - Creates a **new file** `Y` in `B`
  - `Y` contains the pathname of `X`

Hard/Symbolic Link Illustration





Ext2 FS: Hard Link and Symbolic Link

- Hard Link problems:
  - With multiple references of a I-Node
    - Maintain a I-Node reference count
    - Decremented for every deletion
- Symbolic Link problems:
  - As **only the pathname is stored**, the link can be easily invalidated:
    - File name changes, file deletion etc

Summary

- Covered implementation details for file system
  - File Information
    - Allocation schemes
  - Free Space management
  - Directory Structure

- Covered the FAT & EXT2 file system

- Relook at file operations from the OS viewpoint

EXTRA TOPICS

File System Consistency Check

- Power loss / system crash can render the file system in an inconsistent state

- Tools for checking consistency:
  - Windows: **`CHKDSK`** (check disk)
  - Linux: **`fsck`** (file system check)

- Using your understanding of the file systems covered, can you deduce what are the issues can be found / fixed by such tools?

Defragmentation

- Fragmentation: File data can be scattered across many disjoint blocks on storage media
  - Seriously impact I/O performance

- In Windows:
  - Official and 3rd party software to alleviate the problem

- In Linux:
  - File allocation algorithm is more intelligent:
    - Files are allocated further apart
    - Free blocks near to existing data block are use if possible
  - Fragmentation is very low when the drive occupancy is < ~90%

Journaling

- Keep additional information to recover from system crash

- Basic Idea:
  1. Write the information and / or the actual data into a separate log file
  2. Perform the actual file operation

- Depending on the information logged:
  - Can recover to an earlier stable state or re-perform the interrupted file operation

- Most current file systems support journaling in some fashion

Interesting/Important File Systems (1/3)

- Virtual File System (VFS)
  - Provides another layer of abstraction on top of existing file systems
    - Allow application to access file on **different file systems**
    - File operations are translated by VFS automatically to the corresponding native file system

- Network File System (NFS)
  - Allows files to reside on different physical machine
  - File operations are translated into network operations

Interesting/Important File Systems (2/3)

- New Technology File System (NTFS)
  - Used in WinXP onwards
  - Some features:
    - File Encryption, File Compression
    - Versioning (different versions of a file is kept)
    - Hard/Symbolic Link
- Extended-3 / 4 File System (Ext3 / Ext4)
  - Variant on the Ext2 FS
  - Support:
    - Journaling (Keep track of changes to file)
    - In-place upgrade from Ext2
    - Expanded maximum file and file system size

Interesting/Important File Systems (3/3)

- Hierarchical File System Plus (HFS+)
  - Used in Mac OS X
  - Some features:
    - Compression, encryption support
    - Large file system, file and number of file/folder support
    - Metadata journaling