File System Management

File System Implementations

Lecture 12

Overview

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

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

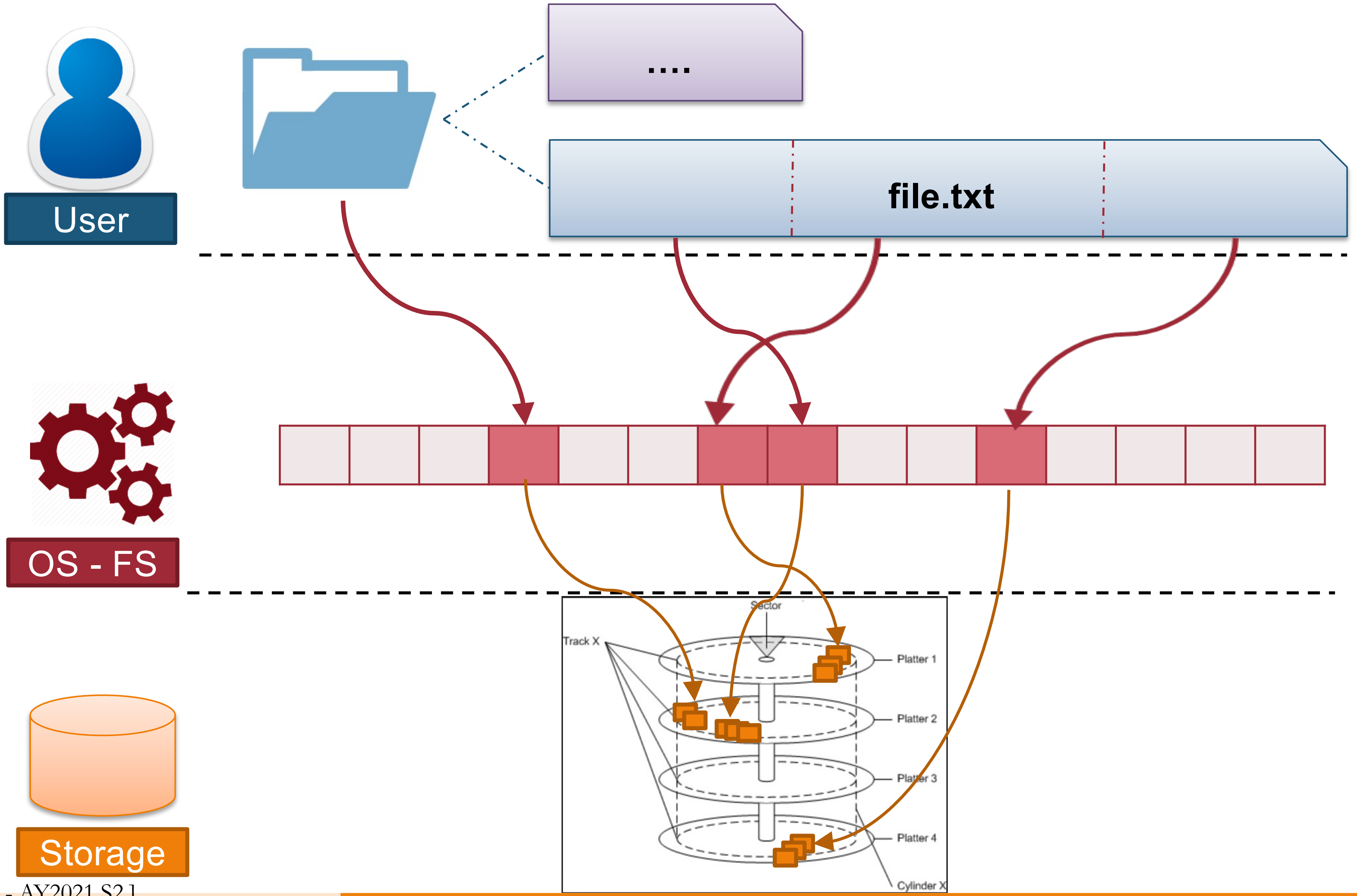
- File System in Action

- Disk I/O Scheduling

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

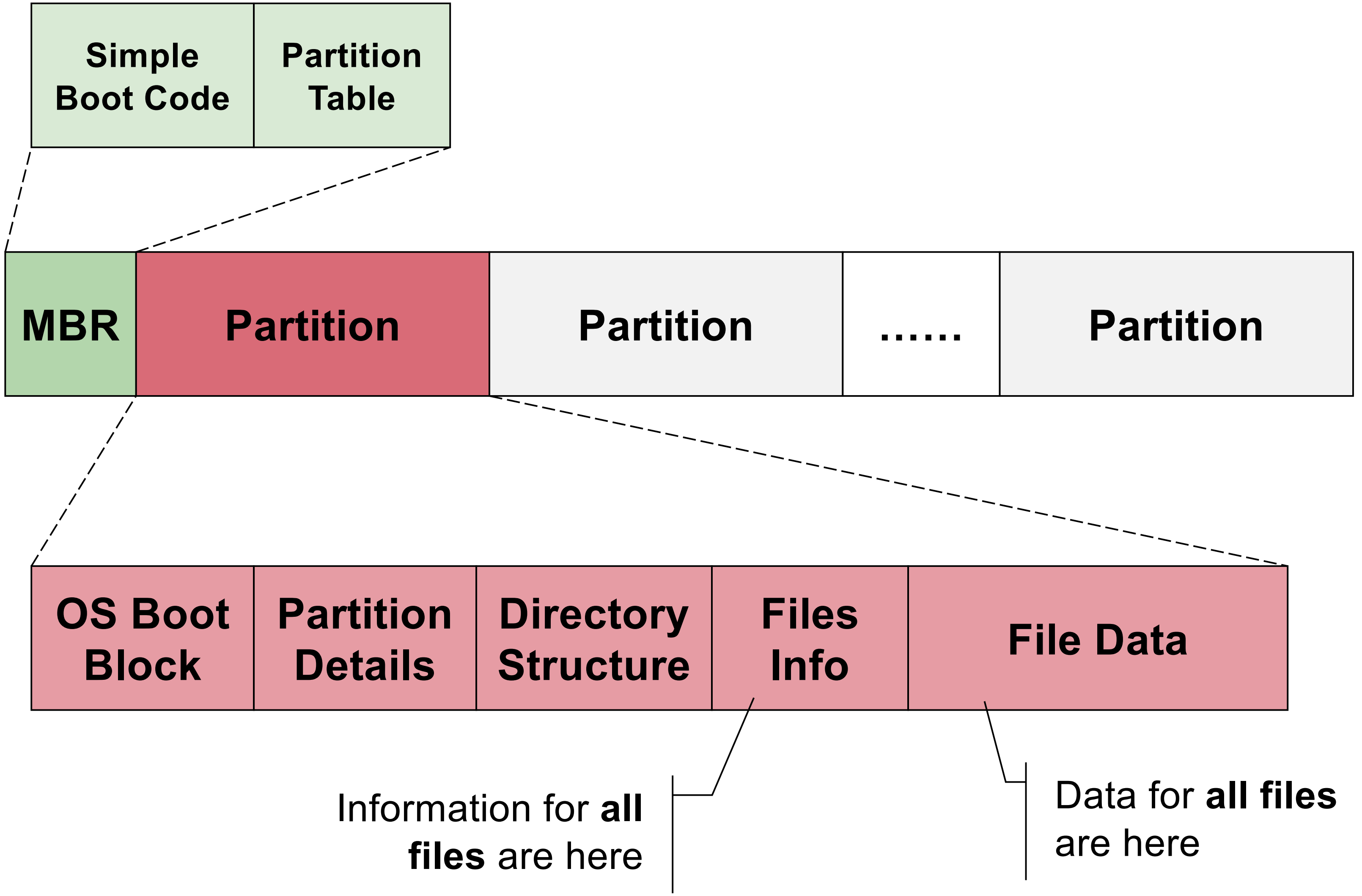


Disk Organization: Overview

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

OS Boot Block	Partition Details	Directory Structure	Files Info	File Data

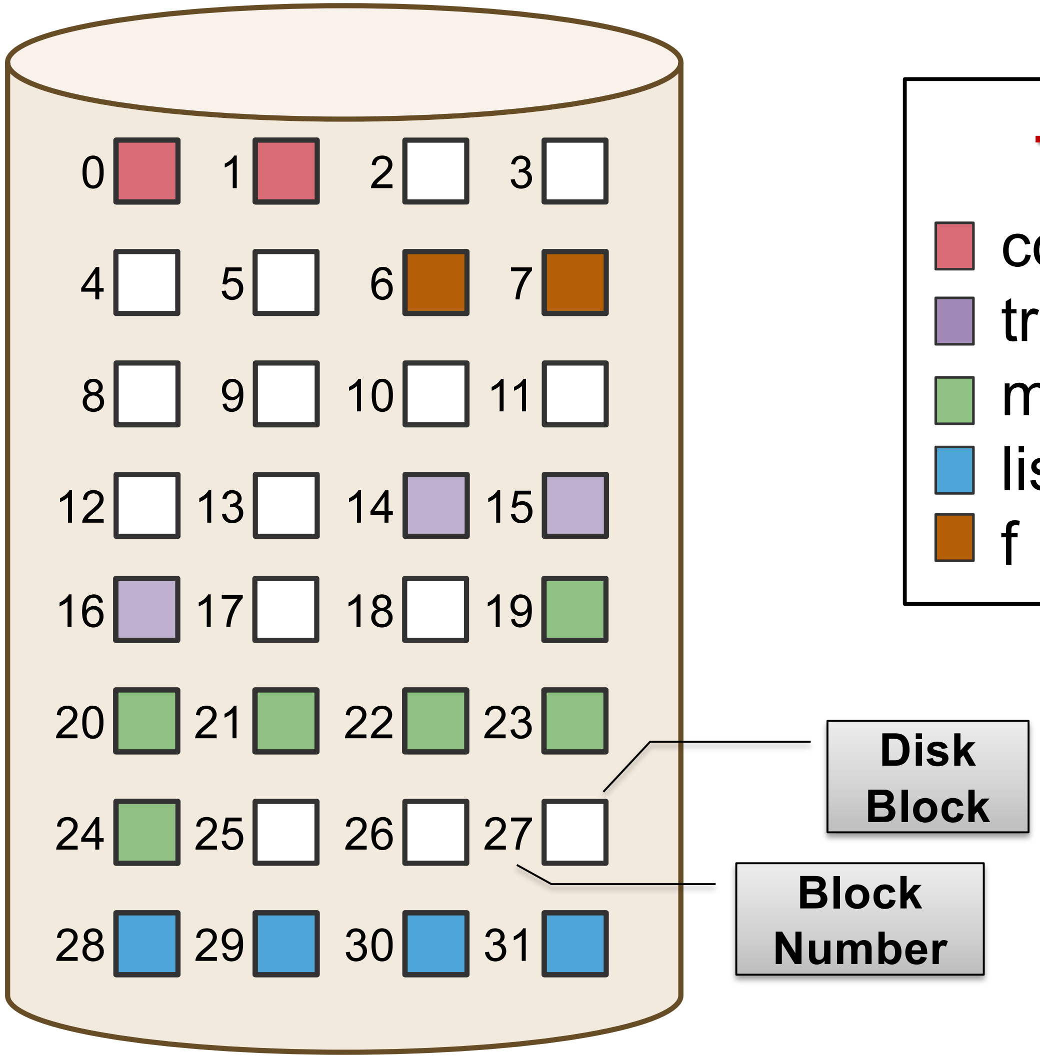
File Implementation: Overview

- 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



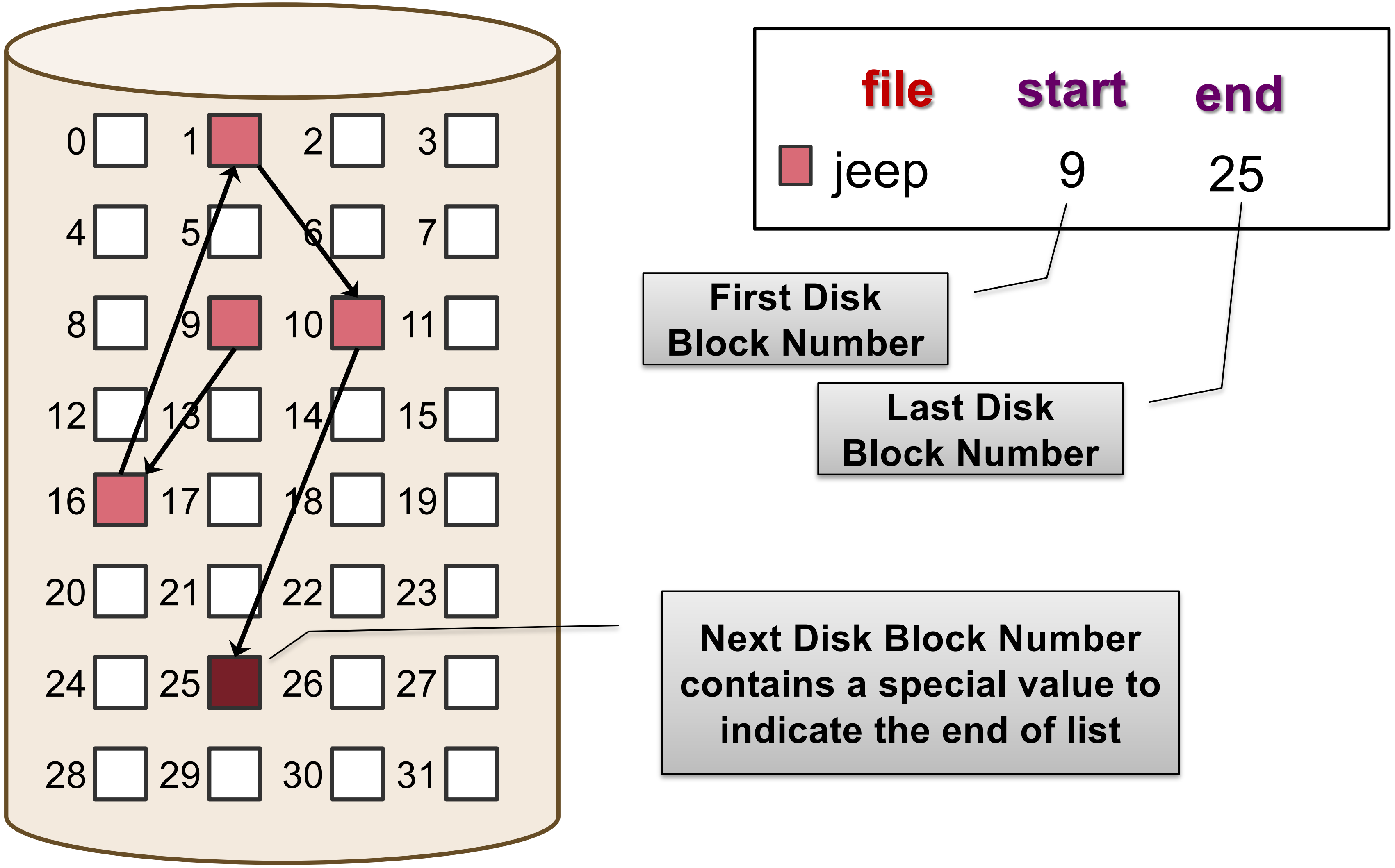
Directory

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?)

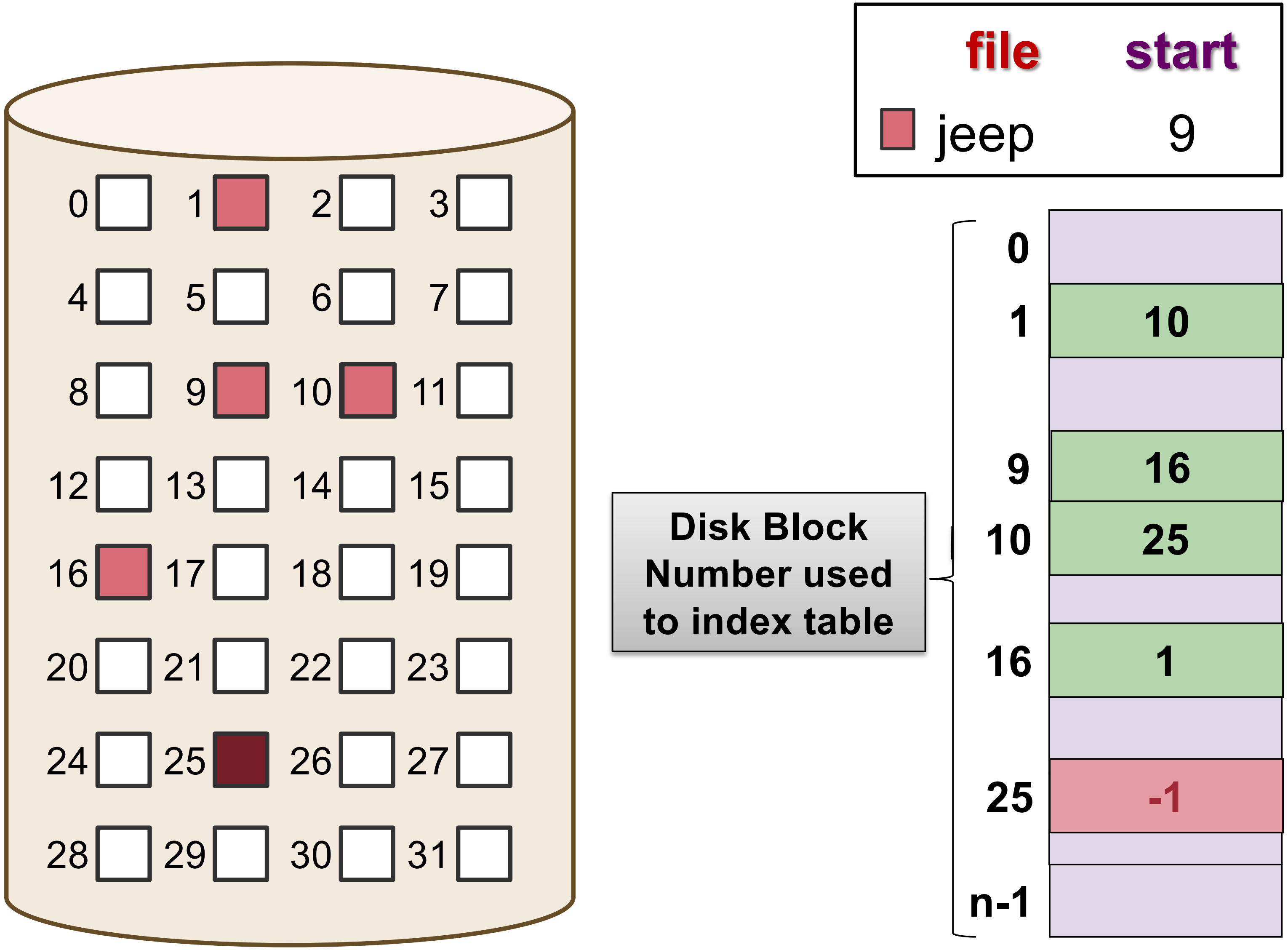
Linked List Allocation



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

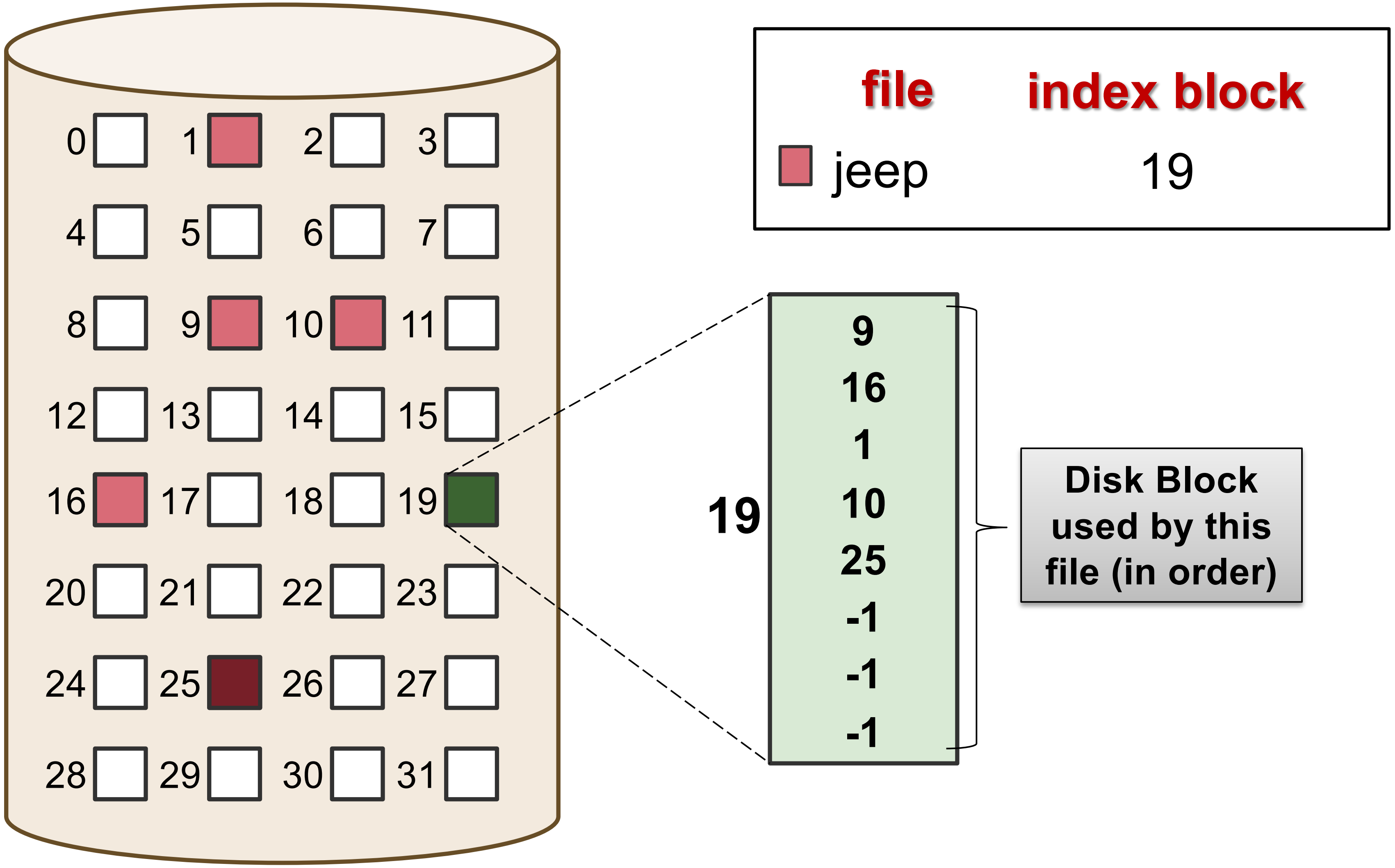
FAT Allocation



File Block Allocation 3: **Indexed Allocation**

- **General Idea:**
  - Each file has an **index block**
    - An **array of** disk block addresses
    - IndexBlock[N] == N^{th} 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



Indexed Block Allocation: Variation

- Several schemes to:
  - Allow larger file size
- Linked scheme:
  - Keep a **linked list** of index blocks
  - Each index block contains the pointer to next index block
- 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

Indexed Block Allocation: Variation (cont)

- Combined scheme:
  - Combination of direct indexing and multi-level index scheme
  - Example: Unix I-node has:
    - 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)

Unix Indexed Node (I-Node): Illustration

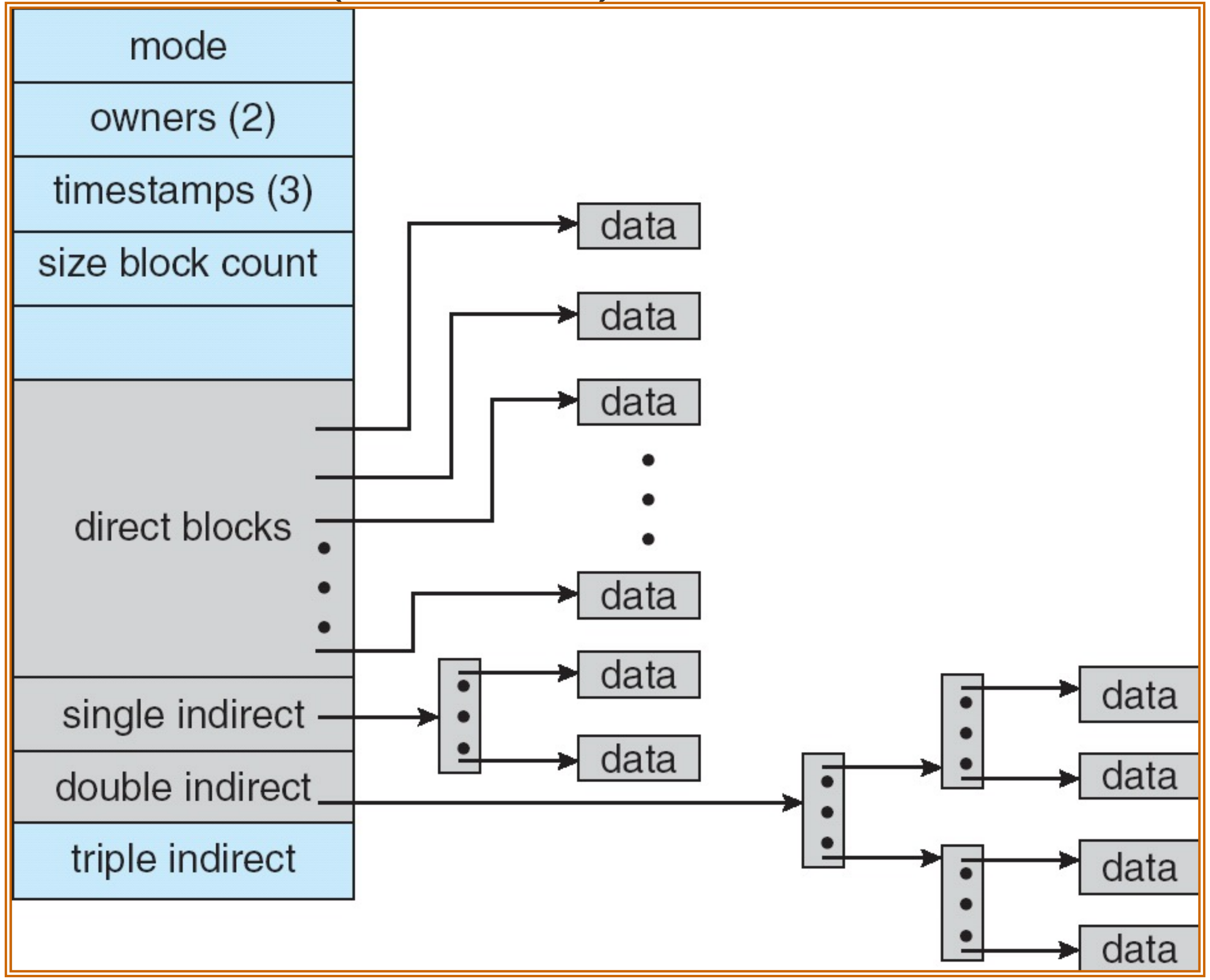
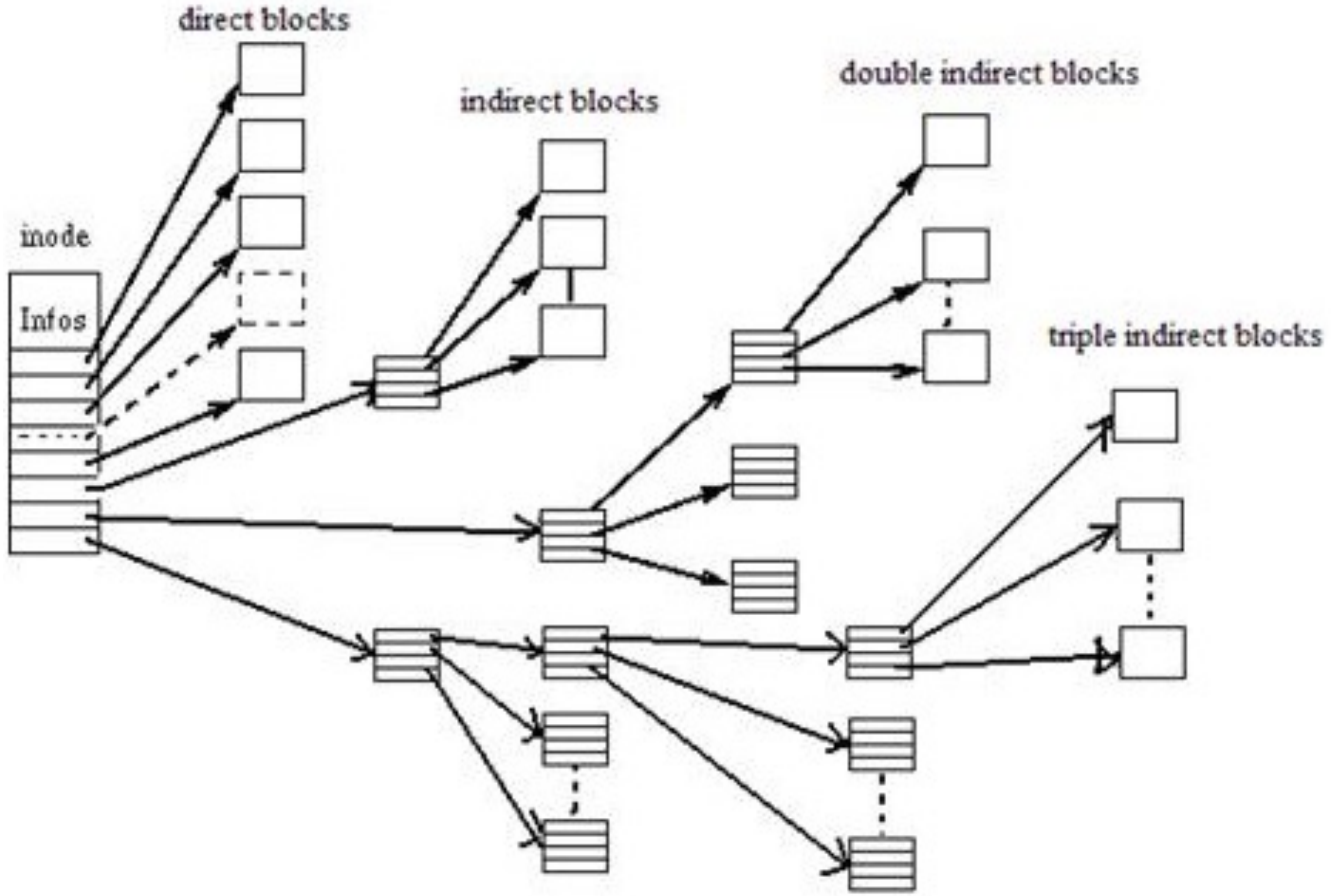


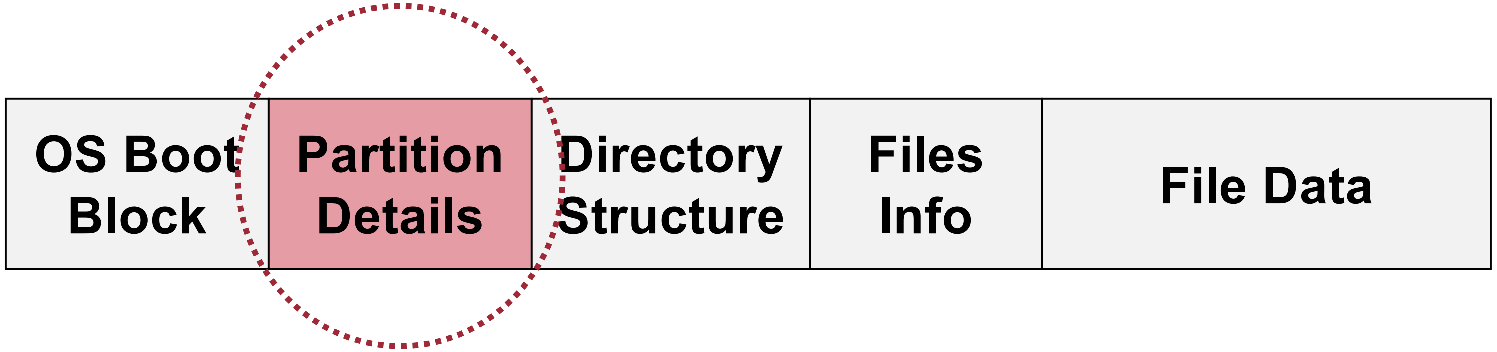
Image taken from "Operating System Concepts" 7th Edition by Silberschatz, Galvin and Gagne

I-Node Unrolled



- 12 **direct pointers** that point to disk block directly
- 1 **single indirect block**
  - contains a number of **direct pointers**
- 1 **double indirect** block
  - points to a number of **single indirect blocks**
- 1 **triple indirect block**
  - points to a number of **double indirect blocks**
- The combination ensures efficiency for small files and flexibility (still support larger files)

Free Space Management



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:

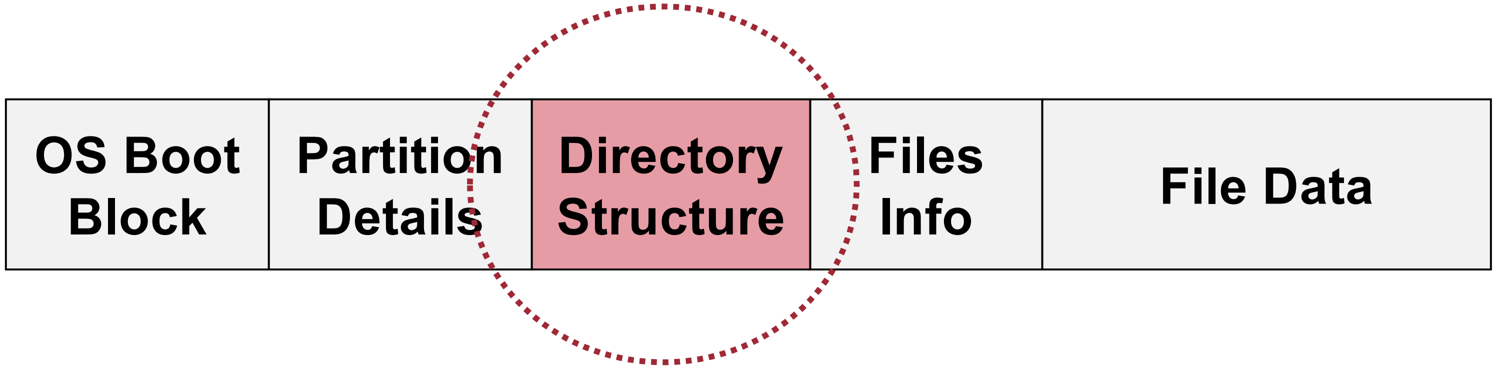
```
0 1 0 1 1 1 0 0 1 0 1 1 .......
```

  - 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
    - 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

File locked and loaded!

FILE SYSTEM IN ACTION

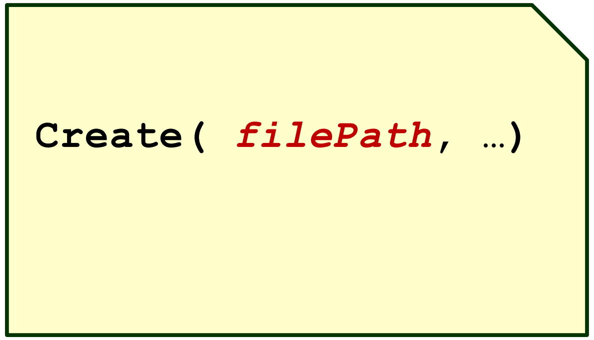
File System in Action: Overview

- Previous sections are on **static information** for a FS stored on media
- At runtime, when user interacts with file:
  - **Run-time information** is needed
  - Maintained by OS in memory
- [Recap] Common in-memory information:
  - System-wide open-file table:
    - Contain a copy of file information for each open file + other info
  - Per-process open-file table:
    - Contains pointer to system-wide table + other info
  - Buffers for disk blocks read from/written to disk

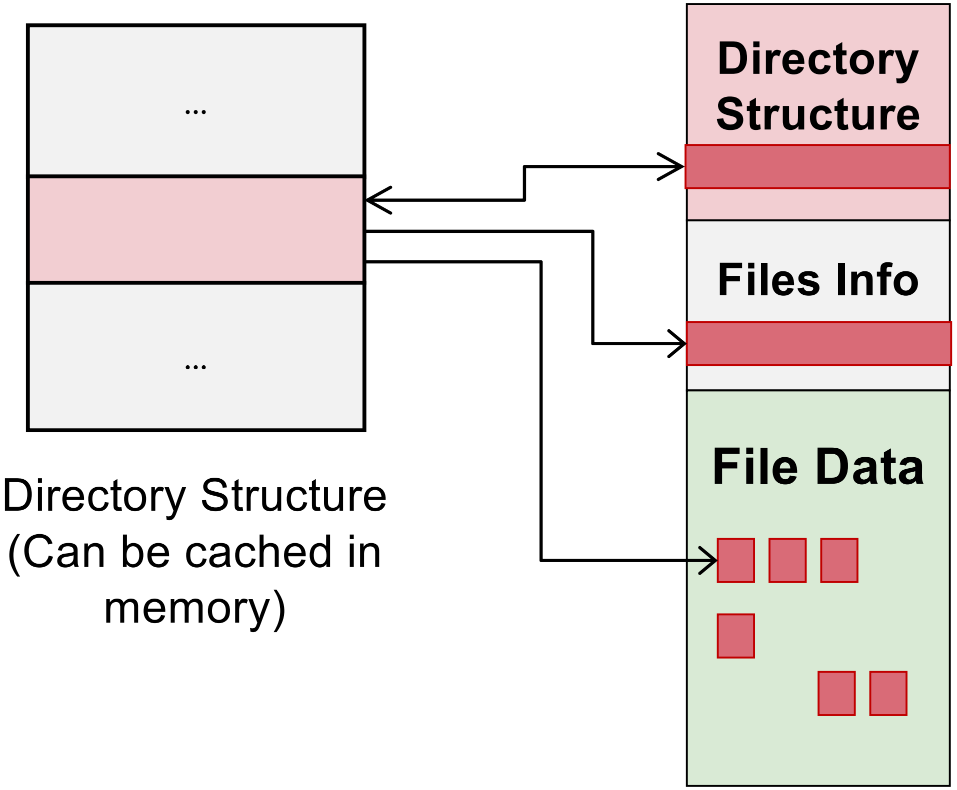
Walkthrough on file operation: Create

- Let us relook at the file operation
  - With the newly covered details
- To create a file `/…/…/parent/F`:
  - Use full pathname to locate the **parent** directory
    - Search for filename **F** to avoid duplicates
      - If found, file creation terminates with error
    - Search could be on the cached directory structure
  - Use free space list to find free disk block(s)
    - Depends on allocation scheme
  - Add an entry to **parent** directory
    - With relevant file information
    - File name, disk block information etc

File Creation: Illustration



User Program



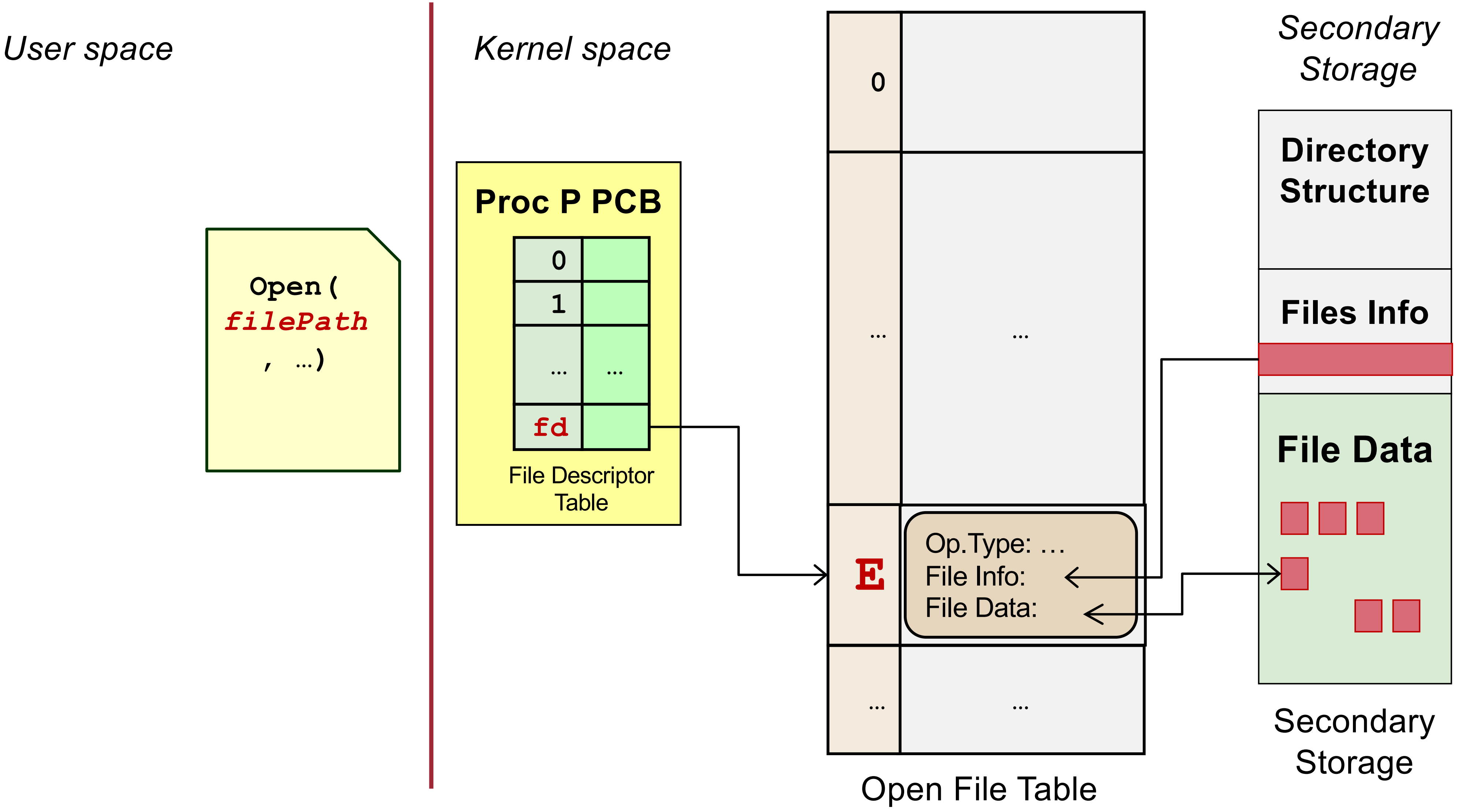
Directory Structure (Can be cached in memory)

Secondary Storage

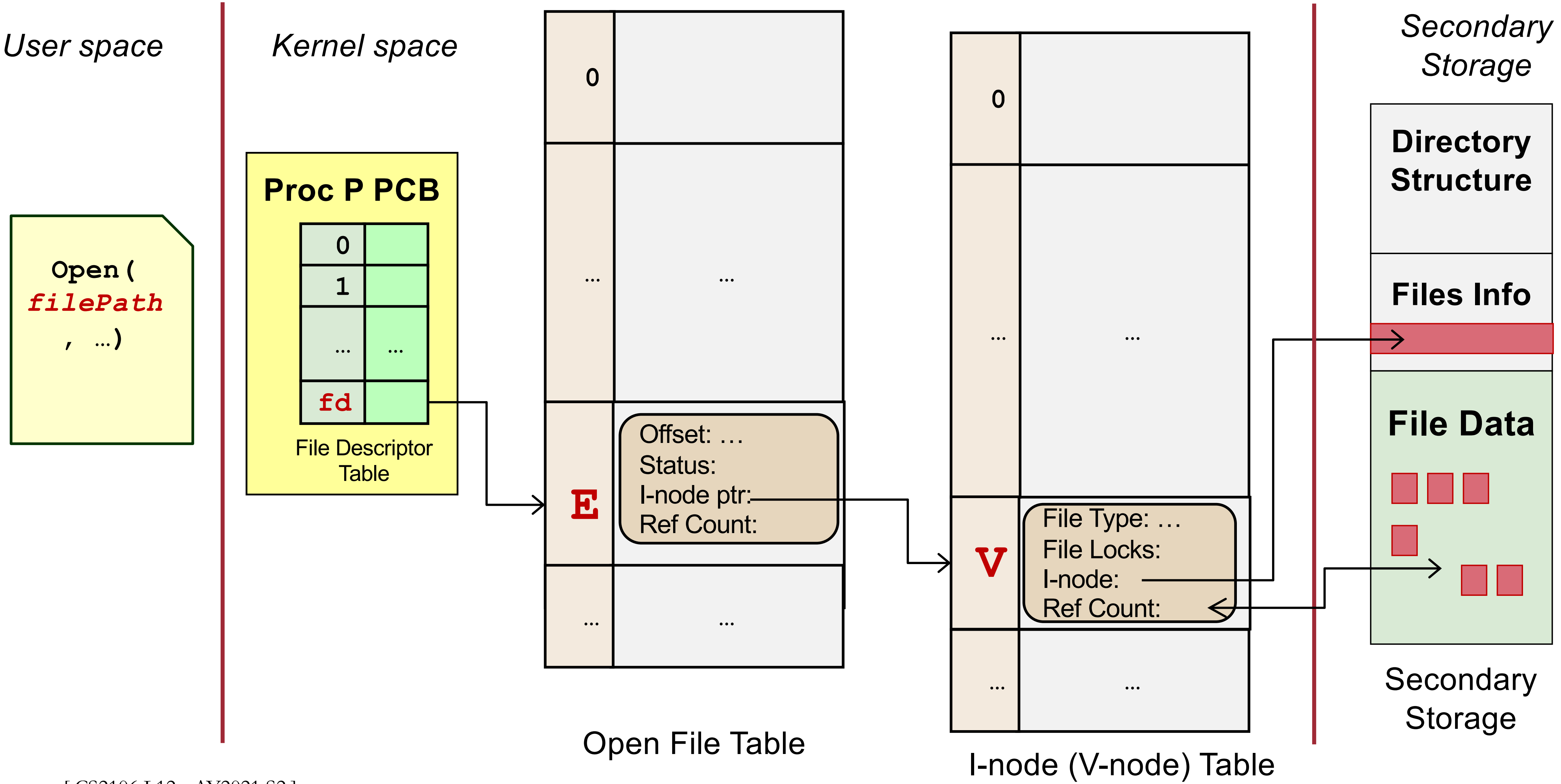
Walkthrough on file operation: **Open**

- Process `P` open file `/…/…/…/F`:
  - Search system-wide table for existing entry `E`
    - If found:
      - Creates an entry in `P`'s table to point to `E`
      - Return a pointer to this entry
    - If not found, continue to next step
  - 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`
    - Return a pointer to this entry
- The returned pointer is used for further read/write operation

File Open: Improved Understanding



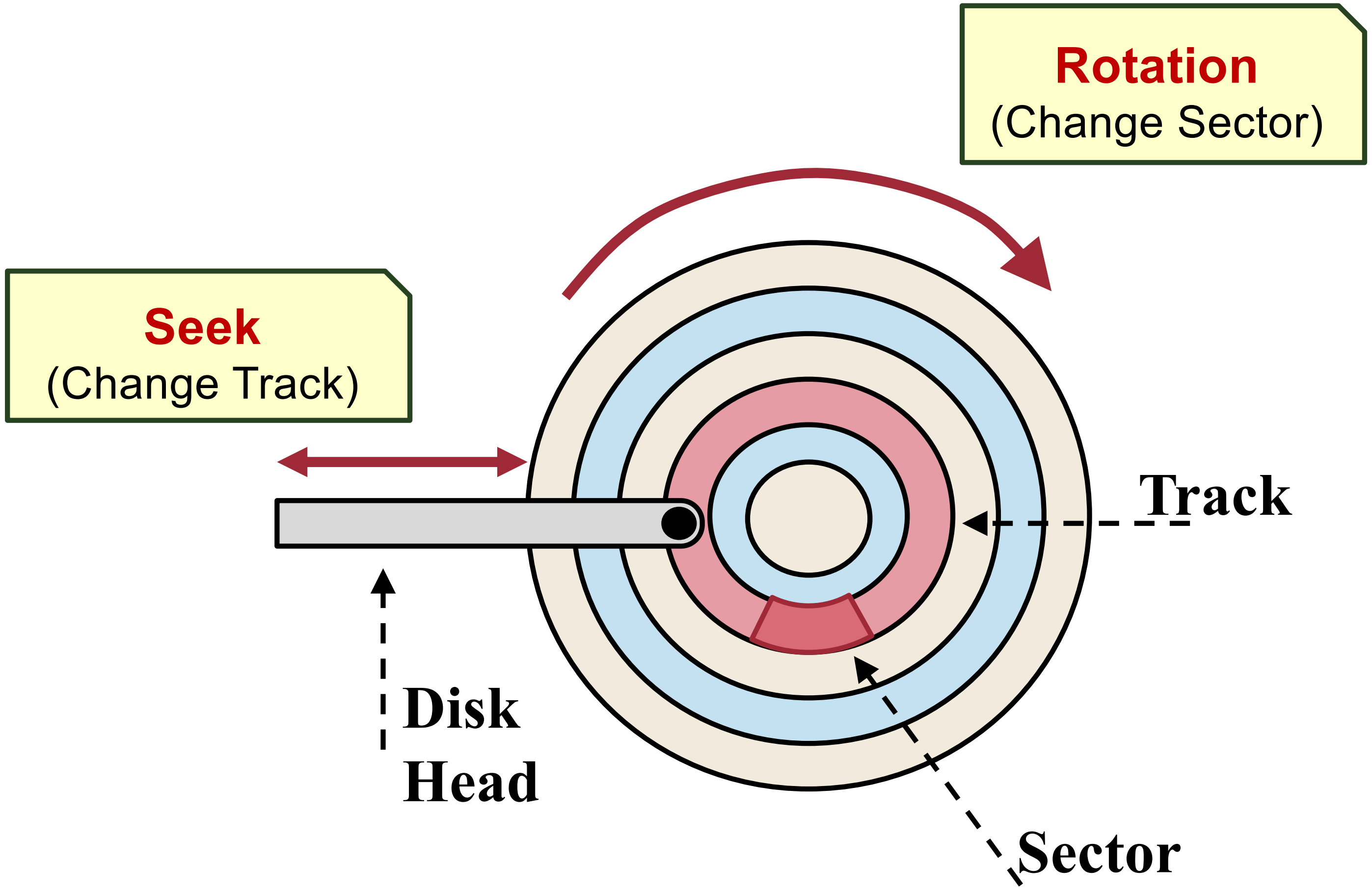
File Open in Linux (non-examinable)



I'm afraid you have to wait…..

DISK I/O SCHEDULING

Magnetic Disk in One Glance



Disk Scheduling: The Problem

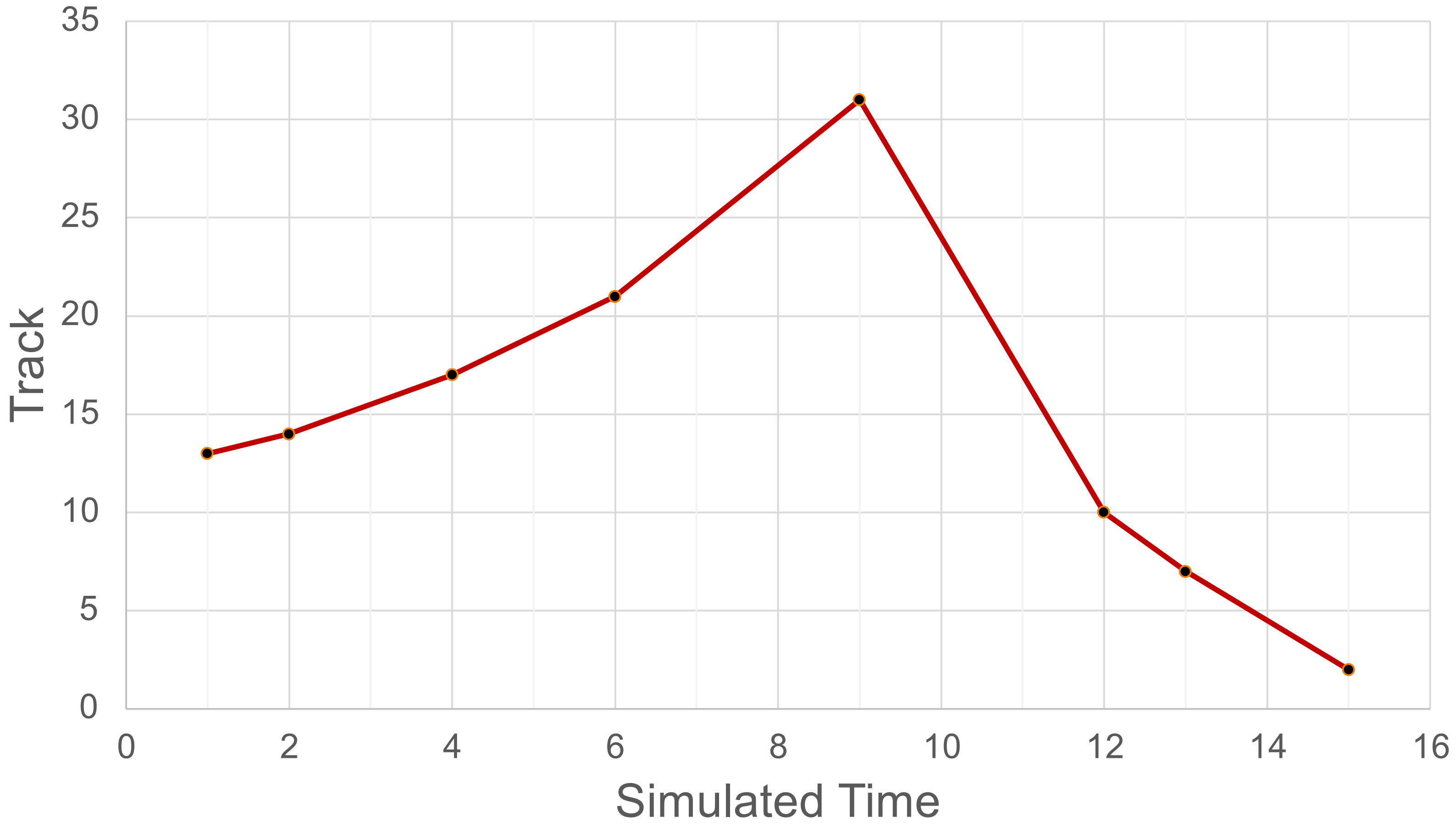
- Due to the significant seek and rotational latency, OS should schedule the disk I/O requests
- I/O (disk) scheduling:
  - Intention of reducing **overall waiting time**
  - As rotational latency is hard to mitigate, we focus on reducing the **seeking time**
  - Balance the need for high throughput while trying to fairly share I/O requests amongst processes

Disk Scheduling: Algorithms

- Consider the following disk I/O requests indicated by only the **track number (magnetic disks)**:
  - **13, 14, 2, 18, 17, 21, 15**
- A few obvious candidates:
  - **FCFS**
  - **SSF** (**S**hortest **S**eek **F**irst)
    - "SJF" modified for the disk context
  - The **SCAN** family (aka **Elevator**):
    - Bi-Direction [Innermost ←→ Outermost] (SCAN)
    - 1-Direction [Outermost→ Innermost, then seek back and start again from the outermost] (C-SCAN)
    - Very intuitive: Imagine the tracks are floors in a building, and the disk head is the elevator servicing the floors (Figure out the algorithm before lecture ☺)

SCAN: Disk Head Movement

- disk I/O requests indicated by only the **track number** :
  [13, 14, 2, 10, 17, 21, 7]



I/O Scheduling: Newer Algorithms

- **Deadline** - 3 queues for I/O requests:
  - Sorted
  - Read FIFO - read requests stored chronologically
  - Write FIFO - write requests stored chronologically
- **noop (No-operation)** - no sorting
- **cfq (Completely Fair Queueing)** - time slice and per-process sorted queues
- **bfq (Budget Fair Queuing) (Multiqueue) -** fair sharing based on the number of sectors requested

Summary

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

- Relook at file operations from the OS viewpoint

- Discussed OS responsibility in I/O scheduling
  - for hard disks

References

- **OS Concepts**, 9th Edition
  - FS abstraction: 11.1 – 11.3
  - FS implementation: 12.1 – 12.5
  - Disk Scheduling: 10.4
- **Modern Operating Systems**, 4th Edition
  - FS abstraction: 4.1, 4.2
  - FS implementation: 4.3
  - Disk Scheduling: 5.4.3.1
- **Three Easy Pieces**:
  - Chapters 39, 40