Lesson 1: The File Concept

1. What is a File? (The Deep Definition)

To a computer, a file is a **contiguous logical address space**.

- **Logical:** This means how the *user* sees it. You see a single, solid document.
- **Contiguous:** This means the data is in one long, unbroken line.
- **Address Space:** This is just a set of "spots" where data lives.

Analogy: Imagine a long piece of movie film. To you, it's one movie (logical). It's one long strip (contiguous). Every frame has a number so the projector knows where it is (address space).

2. File Attributes (The Metadata)

A file isn't just the data inside; it's also the "tags" that describe it. These are stored in the **directory structure** on the disk.

- **Name:** The human-readable label (e.g., "Note.txt").
- **Identifier:** A unique "secret number" the computer uses. This is the real way the system finds the file.
- **Type:** Tells the system if it's a photo, a program, etc..
- **Location:** A **pointer** (like a GPS coordinate) that tells the computer exactly where on the physical disk the data is hiding.
- **Size:** How many bytes it currently uses.
- **Protection:** Rules about who can read, write, or run the file.
- **Time, Date, and User ID:** For security and tracking usage.
- **Checksum:** (Extended attribute) A mathematical "signature" used to check if the file is broken or corrupted.

3. File Types (The Extension Table)

The system uses **extensions** (the letters after the dot) to know what a file does.

File Type	Usual Extension	Function
Executable	.exe, .com, .bin	Ready-to-run programs.
Object	.obj, .o	Machine language that isn't a full program yet.
Source Code	.c, .java, .py	Code written by humans.
Batch	.bat, .sh	A list of commands for the computer to follow.
Word Processor	.doc, .rtf, .tex	Formatted text documents.
Library	.lib, .dll	Snippets of code shared by many programs.
Archive	.zip, .tar	Files squished together to save space.
Multimedia	.mp3, .avi, .mov	Sound and video files.

4. File Structure (How bytes are organized)

Who decides how the bytes are arranged? Either the **Operating System** or the **Program** you are using.

- **The Operating System:** If the OS decides the structure, it "understands" the file. For example, if you try to open a file the OS doesn't recognize, it might refuse to run it. The benefit is that the OS can provide built-in tools to search or sort those specific file types.
- **The Program:** In most modern systems, the OS ignores the structure and treats every file as a **"None" (sequence of bytes)**. The **Program** (like Word or a game) then takes that long sequence and "simulates" a complex structure by inserting its own **control characters**.

The arrangement of bytes in a file depends entirely on the level of structure the system or application imposes on the raw data.

1. None: Sequence of Bytes

In this most basic form, the file is simply a **contiguous logical address space** where bytes are placed one after another without any internal markers or dividers recognized by the operating system.

- **How they are arranged:** Bytes are stored linearly from position 0 to position *n*. The system does not know where one "piece" of information ends and the next begins.
- **Why use it:** This is the most flexible method. It allows a program to put any kind of data into a file without the Operating System (OS) interfering or forcing a specific format.

2. Simple Record Structure

Here, the file is divided into chunks called "records." The arrangement follows one of three logic types:

- **Lines:** Bytes are arranged into groups separated by a special "newline" control character.
- **Fixed Length:** The file is a series of identical-sized blocks (e.g., every 100 bytes is a new record). The computer finds record #5 by simply skipping the first 400 bytes.
- **Variable Length:** Each record can be a different size. This usually requires a small "header" at the start of each record to tell the system how many bytes follow.

3. Complex Structures

These files have internal maps and specific sections.

- **Formatted Documents:** These contain internal codes for fonts, images, and layout (e.g., a `.tex` or `.doc` file).
- **Relocatable Load Files:** These are programs organized into segments (code, data, stack) so the OS knows exactly how to place them in memory.

Example of Simulation: A program can take a "None" sequence of bytes and treat it as a "Complex" structure by deciding that the first 10 bytes are the "Title," the next 4 bytes are the "Security Code," and everything after is "Image Data". The OS doesn't know this; it just sees bytes. Only the program has the "key" to understand the arrangement.

The "Why":

- **Programs decide** when they need total control over how data is packed to save space or improve speed.
- **Operating Systems decide** when they need to enforce standards, such as ensuring a file is "executable" before allowing it to run.

5. Essential File Operations (The Mechanics)

A file is an **Abstract Data Type (ADT)**. This means it is a set of data *plus* the specific things you can do to it.

- **Create:** Find space on the disk and add an entry to the directory.
- **Write:** Put data into the file. The system must track a "write pointer" to know where the next byte goes.
- **Read:** Take data out. It uses a "read pointer".
- **Reposition (Seek):** Skipping to a different part of the file (like fast-forwarding a song).
- **Delete:** Remove the directory entry and free up the disk space.
- **Truncate:** Keeping the "attributes" (the name/size) but erasing the content.

6. The "Open" and "Close" Process (Crucial!)

You cannot just "use" a file on a disk; it’s too slow.

- **Open(Fi):** The computer searches the disk's directory, finds the file, and moves its info into **Memory (RAM)** for fast access.
- **Close(Fi):** The computer takes the updated info from memory and saves it back to the disk's directory.

The Open-File Table: The OS keeps a list of every open file. It tracks:

1. **File Pointer:** Remembers the last place you read/wrote.
2. **File-open Count:** If 5 programs have a file open, the count is 5. The OS only "closes" the file and clears it from memory when the count reaches **zero**.
3. **Disk Location Cache:** A shortcut to the disk address so the computer doesn't have to search again.
4. **Access Rights:** Per-process info on whether you can edit or just view.

7. File Locking (Managing Traffic)

If two programs change a file simultaneously, the data can become "corrupted" or broken. Locking **mediates** (manages) this traffic so everyone stays safe.

- **Mandatory Locking(Compulsory):** Access is denied depending on locks held and requested. If a program has a mandatory lock on a file, and another program tries to access it, the OS simply says "Access Denied". No one else can touch it until it's free.

- **Advisory Locking(Consultative):** Processes can find the status of locks and decide what to do. The OS shows a "Busy" sign, programs check the sign and **decides for itself what to do**—it can wait, or it can try to open the file anyway.

Lesson 2: Access Methods

1. What is an Access Method?

An **Access Method** is the strategy or "rulebook" the computer follows to find and read information inside a file.

Analogy: Imagine a file is a very long **cassette tape** (Sequential) versus a **CD** (Direct).

- On a tape, you have to play the whole thing to get to the end.
- On a CD, you can click "Track 5" and the laser jumps there instantly.

2. Sequential Access (The "Line" Method)

This is the simplest way to access a file. The computer processes information in order, one piece after another. You cannot jump ahead without passing everything in between.

The Operations (The Commands):

- **read next:** The computer reads the very next chunk of data and automatically moves its "bookmark" (the pointer) forward.
- **write next:** The computer adds new data to the end of the file and moves the "bookmark" forward.
- **reset:** The computer moves the "bookmark" all the way back to the beginning of the file.
- **no read after last write (rewrite):** This is a safety rule. If you just finished writing new data at the end, you can't immediately read "nothingness" beyond it. Because a sequential file follows a strict order, you cannot immediately "read" the space after your last write because the computer hasn't put anything there yet. To see what you just wrote or to change it, you must "reset" (go back to the start) or perform a "rewrite" to reposition the pointer.

Why use it? It is perfect for things like log files or simple text documents where you always start at the top and read to the bottom.

3. Direct Access (The "Jump" Method)

This is also called **Relative Access**. Here, the file is viewed as a numbered sequence of **blocks** or **records**. The computer can jump to any block number instantly without reading the ones before it.

Key Term: Relative Block Number The computer doesn't see "page numbers." It sees **n**, which is the **relative block number**.

- **Block 0** is the first chunk of the file.
- **Block 1** is the second, and so on.
- The number is "relative" to the start of the file, not the whole disk.

The Operations:

- **read n:** Jump directly to block *n* and read it.
- **write n:** Jump directly to block *n* and write data there.
- **position to n:** Move the "bookmark" to block *n* but don't read or write yet.
- **rewrite n:** Change the data that is already inside block *n*. In sequential access, you usually have to rewrite the whole file to change one word in the middle.

Why use it? This is essential for **Databases**. If you are looking for one specific customer in a file of 1 million, you don't want to read 999,999 names first. You want to jump straight to their "ID number" (their block).

Mapping Relative Block Numbers to Physical Locations

To find a specific piece of data, the operating system (OS) must translate a **Relative Block Number (n)** into a physical address on the disk.

1. **The Base Address:** When a file is opened, the OS looks at its **Location** attribute (a pointer to the file's start on the device). This physical starting point is cached in the **Open-File Table** for speed.
2. **The Offset:** The **Relative Block Number** acts as an "offset." If you want to read block n, the computer doesn't search the whole disk; it simply takes the starting physical address and "skips" forward by n blocks.
3. **The Result:** This simple addition (Start Address + n) tells the disk's hardware exactly which physical track and sector to read, allowing the "jump" that makes **Direct Access** so fast.

4. Simulating Sequential Access on a Direct File

If you have a file that allows "jumping" (Direct Access), you can make it *behave* like a Sequential file if you want to. The computer does this using a simple math formula or code logic.

How the logic works: The computer keeps a hidden variable called **cp** (Current Position).

1. **Reset:** To start over, it just sets `cp = 0`.
2. **Read Next:** It performs `read cp` and then immediately does `cp = cp + 1`.
3. **Write Next:** It performs `write cp` and then immediately does `cp = cp + 1`.

Why do this? It gives the programmer flexibility. They can jump around when they need to, but use a simple "next" command when they just want to read in order.

5. Index and Relative Files (The "Address Book" Method)

A Step-by-Step Example

The diagram in the lecture shows an **Index File** containing names like Adams, Arthur, Asher, and Smith.

1. **The Goal:** You want to find "Smith's" social security number.
2. **The Search:** The computer does not search the huge **Relative File**. Instead, it searches the small, organized **Index File** for the name "Smith".
3. **The Pointer:** The Index File tells the computer that "Smith" is linked to a specific **logical record number** (a "jump" address).
4. **The Jump:** The computer follows that arrow and jumps directly to that specific spot in the **Relative File** to pull the data (Social Security and Age).

Analogy: The **Index** is like the index at the back of a textbook. You look up a word, find the page number, and jump straight to that page in the "Relative File" (the rest of the book).

Lesson 3: Directory Structure

A **directory structure** is a collection of nodes that contains information about all the files in a system. Think of it as the "address book" or "map" for the storage device.

1. Where It Lives and Why It Matters

The directory structure and the files themselves both reside on the **disk**. Because this information is so important, backups of both are often kept on **tapes** to prevent data loss.

The directory has two main jobs:

1. **For the User:** It provides a way to name and organize files so we can find them.
2. **For the System:** It provides a way to translate a human name (like `photo.jpg`) into a physical location on the disk.

The "Big Picture": Partitions and Volumes

To understand why these are significant, you have to see the hierarchy of how a disk is built:

1. **Disk:** The physical hardware.
2. **Partitions (Slices/Minidisks):** The disk is subdivided into smaller "virtual disks".
3. **Volume:** An entity that contains a file system and its own **device directory** or **volume table of contents** to track its files.
4. **Directory:** The map within that volume.
5. **Files:** The individual containers of data.
6. **Index:** The internal "map" for a specific large file.

2. Operations Performed on a Directory

The Operating System provides specific actions you can perform on the directory itself, not just the files inside it:

- **Search for a file:** Finding the entry for a specific file name.
- **Create a file:** Adding a new entry to the directory.
- **Delete a file:** Removing an entry from the directory.
- **List a directory:** Showing all the files currently "recorded" in that directory.
- **Rename a file:** Changing the name in the directory entry.
- **Traverse the file system:** Moving through the directory "tree" to look at different sections of the disk.

3. Goals of a Good Directory Organization

To be useful, a directory must achieve three things:

- **Efficiency:** You should be able to find a file very quickly.
- **Naming Convenience:**
    - Two different users should be able to use the same name for different files without a conflict.
    - The same file should be able to have several different names (aliases).
- **Grouping:** You should be able to group files logically by their properties (for example, putting all "Java programs" in one spot and all "games" in another).

4. Types of Directory Structures

A. Single-Level Directory

This is the simplest possible structure where **all files** for **all users** are kept in one single list.

- **The Naming Problem:** Since there is only one list, no two files can have the same name. If User A names a file `test`, User B cannot use that name.
- **The Grouping Problem:** As the number of files grows, it becomes a giant, messy pile. There is no way to separate your work from someone else's.

B. Two-Level Directory

To solve the naming problem, the system creates a **Master File Directory (MFD)** that points to individual **User File Directories (UFDs)**.

- **How it works:** When a user logs in, the system searches the Master directory to find that specific user's private directory.
- **Naming:** Now, User 1 and User 2 can both have a file named `a` because they are in different lists.
- **Path Name:** To find a file, you now need a "Path" (e.g., `User1/test`).
- **The Limit:** While this is efficient for searching and solves name conflicts, it still has **no grouping capability**. A single user still cannot create "folders within folders" to organize their own files.

C. Tree-Structured Directories

This is the modern system used by Windows, Mac, and Linux. It allows a directory to contain both **files** and **other subdirectories**.

- **The Root:** The very top of the tree is called the **root**.
- **Path Names:**
    - **Absolute Path Name:** Starts from the root (e.g., `/spell/mail/prog/list`).
    - **Relative Path Name:** Starts from where you are currently "standing" (the **Current/Working Directory**).

- **Navigation and Management:**
  - `cd` **(Change Directory):** Used to move your "current position" in the tree.
  - `mkdir`**:** Creates a new subdirectory in your current spot.
  - `rm`**:** Deletes a file.
- **The "Subtree" Rule:** In a tree structure, if you delete a directory (like `mail`), the system deletes the **entire subtree** rooted by that directory (all the files and folders inside it).

Lesson 4: File-System Mounting

Mounting is the process of "attaching" a separate file system to your existing one so the computer can actually use it. It is the process of taking a **group of files** and making them available to a **user or group**. Mounting allows the administrator to decide exactly *where* in the directory tree a set of files appears, which determines which users can see and use them.

A file system cannot be accessed until this connection is made. If you have a disk full of data but haven't "mounted" it, the Operating System acts as if those files do not exist. Mounting is the "activation" step.

1. The Mechanic: Plugging in the "Bridge"

Think of your computer's main file system as a large **tree**. If you plug in a USB drive or add a new hard drive, it is like having an **extra branch** that isn't connected to the tree yet.

- **Unmounted Partition:** This is a disk or section of a disk that has its own internal folders and files but is currently "invisible" to the main system.
- **The Mount Point:** To make it visible, you must pick a specific "spot" (an existing empty folder) on your main tree. This spot is called the **mount point**.

2. How the OS Does It

1. The Operating System (OS) looks at the **unmounted partition**.
2. It identifies the **root** (the very top) of that partition's private tree.
3. The OS then "glues" that root onto the **mount point** directory you selected in your main system.
4. **The Result:** From now on, whenever you click on that folder (the mount point), the OS automatically diverts you to the files on the new disk. It feels like one big, seamless tree, even though the data lives on two different physical devices.

3. Why is it Significant?

Without mounting, every time you plugged in a disk, the computer wouldn't know where to put it in your folder list. Mounting allows the system to remain organized. You can have multiple disks "plugged in" at different spots in your tree, and the OS handles the translation between the **human path** you see and the **physical hardware** where the data actually sits.

4. The "Existing" vs. "Unmounted" State (The Diagram Example)

Your lecture provides a specific visual example in **Figure 10-11**:

- **The Existing System (a):** Imagine your main system has a root directory (`/`) with a folder called `users`. Inside `users`, you have two people: `bill` and `fred`. Bill has his own files, and Fred has a folder called `help`.
- **The Unmounted Partition (b):** You have a separate disk (or partition). On this disk, there are users named `sue` and `jane`. Jane has folders called `prog` and `doc`. Right now, this disk is "floating" alone; Bill and Fred cannot see Sue or Jane's files.

5. Remote Mounting (NFS and CIFS)

In advanced systems, mounting isn't just for physical disks in your hand.

- **Client-Server Model:** A "Client" (your computer) can mount a file system that actually lives on a "Server" (another computer far away).
- **Translation:** When you do this, your computer takes your standard file commands (like "Open file") and automatically translates them into **remote calls** sent over the network.
- **Protocols:** The lecture notes that **NFS** is the standard for UNIX systems, while **CIFS** is the standard for Windows.

Lesson 5: File Sharing & Protection

This lesson covers how multiple people use the same computer or network without accidentally (or intentionally) breaking each other's work.

1. Why Share Files?

On a multi-user system, file sharing is desirable because it allows people to collaborate. Instead of making ten copies of a project, everyone accesses one shared version. This sharing is managed through a **protection scheme** to ensure only the right people can see the data.

2. Identifying Users: IDs and Groups

To keep things organized, the system needs to know exactly who you are. It uses two "secret numbers":

- **User IDs (UID):** Every person has a unique ID number. This allows the computer to track exactly which files belong to you.
- **Group IDs (GID):** A manager can create a "Group" (like "Accounting" or "Students") and add many users to it.
- **Why use IDs?** It is much easier for a computer to process numbers than human names. By using IDs, the system can quickly check a "Permission List" before it lets you open a file.

3. Remote File Systems (Networking)

Sometimes, the file you want isn't even on your computer; it is on a **Server** across the network.

- **Client-Server Model:** Your computer is the **Client**. You ask the **Server** (the main computer) for a file. The server can serve many clients at once.
- **Automatic Translation:** You don't have to do anything special to open a network file. The Operating System takes your standard command (like "Open") and automatically translates it into a **remote call** sent over the network.
- **Standard Protocols:**
    - **NFS (Network File System):** The standard "language" for sharing files on UNIX systems.
    - **CIFS:** The standard "language" used for sharing on Windows.
    - (Statelessness is not strictly unique to UNIX, but **NFS** is the specific example of a **stateless** protocol used in **UNIX** systems).
    - **Why not one language?** Different operating systems (Windows and UNIX) evolved their own standards over time. However, systems like **Active Directory** or **LDAP** are used today to help provide "unified access" across these different systems.

- Why is there a need for translation?

Your computer uses standard "file calls" for local tasks, such as opening a file on your own hard drive. However, when you want to use a file on a different computer (a server), your local system must translate those commands into **remote calls**. This translation allows you to access files across a network automatically and seamlessly without needing to know special networking commands.

4. Failure Modes: What if the Server Crashes?

Sharing files over a network adds a new risk: the network could break or the server could turn off. Systems handle this in two ways:

- **Stateful:** The server remembers everything you are doing (like which page you are on). If it crashes, it loses that "state" and you get errors.
- **Stateless (NFS):** The server forgets you the moment a task is done. Every request you send includes **all** the information needed to do the job.
- **Why use Stateless?** It makes recovery very easy. If the server crashes and restarts, it doesn't need to "remember" anything to get back to work. You just send the request again.

5. Protection: The "Who" and "What"

The owner or creator of a file should control **what** can be done and **by whom**. There are six main types of access (actions) the system can control:

1. **Read:** Can you look at the content?
2. **Write:** Can you change the content?
3. **Execute:** Can you run the file (if it's a program)?
4. **Append:** Can you add new data only to the end of the file?
5. **Delete:** Can you remove the file?
6. **List:** Can you see that the file exists in the directory?

6. Access Control Lists (ACLs): The Math of Permissions

It is the "master list" the computer checks to see who has permission to use a file. To keep permissions simple, the system divides the world into three classes of users:

- **Owner (User):** The person who made the file.
- **Group:** The "team" assigned to the file.
- **Public (World):** Everyone else on the computer.

The Binary Logic (RWX): Permissions are often written as three numbers. Each number is a sum of:

- **Read (R) = 4 (1 0 0)**
- **Write (W) = 2 (0 1 0)**
- **Execute (X) = 1 (0 0 1)**

How are they presented as numbers?

The system uses a mathematical sum based on three actions: **Read (4)**, **Write (2)**, and **Execute (1)**.

- **7 (Owner):** $4 + 2 + 1 = 7$. This means the owner has "1 1 1" in binary—Full control (RWX).
- **6 (Group):** $4 + 2 = 6$. This is "1 1 0" in binary—Read and Write only.
- **1 (Public):** This is just "1" —Execute only (Run the file, but can't see the code).

Who uses these numbers and do I see them?

The **Operating System** uses these numbers because it can process binary (0s and 1s) much faster than words.

- **Do you see them?** Usually, no. On systems like Mac OS X or Windows, you see a "File Info" or "Properties" window. This window translates those numbers into human-friendly checkboxes like **"Read & Write"** or **"No Access"**.

Analogy: Think of a movie script. The **Owner** (Writer) can change it. The **Group** (Actors) can read it. The **Public** (Audience) can only watch the movie (Execute it) but can't see the paper script or change the ending.

Important Breakdown:

1. Direct Access vs. Directory Structure: Finding vs. Navigating

The biggest point of confusion is often the difference between how a computer **finds** a file and how it **moves around inside** that file.

- **Directory Structure (The External Map):** The directory is a collection of nodes that contains information about **all files** on the disk. Its primary job is to translate a human-readable **Name** into a physical **Location** (a pointer) on the device. Think of this like the **GPS in your car**; it helps you find the building (the file) in a huge city (the disk). Without it, the computer would have no idea where your "homework.pdf" actually starts on the physical hardware.
- **Direct Access (The Internal Movement):** Once the directory has helped you "open" the building (the file), **Direct Access** is the method you use to move between rooms (blocks of data). It treats the file as a numbered sequence of **blocks or records**. You use a **relative block number (n)** to tell the system, "Jump straight to block 50," without walking through blocks 1 to 49 first.
- **The Significance:** We need the **Directory** for organization, so two users can have the same file name without crashing the system. We need **Direct Access** for speed, especially in databases where reading every single line of a million-row file just to find one name would be too slow.

2. What is an Index File?

An **Index File** is a specialized tool used to make searching inside a single, very large file even faster.

- **How it works:** Imagine you have a massive "Relative File" containing thousands of employee records. Instead of searching that giant file linearly, the computer creates a smaller **Index File**. This index contains a **Key** (like a Last Name) and a **Pointer** (the logical record number).
- **The Logic:** You search the small, organized Index for "Smith," find the record number associated with him, and then use **Direct Access** to jump straight to that spot in the big Relative File.
- **Significance:** It acts exactly like the **index at the back of a textbook**. You don't flip through 500 pages to find a topic; you look at the 5-page index, find the page number, and jump directly there.

3. Index File vs. Directory Structure: File-Level vs. System-Level

This is the most common confusion point, as they both "map" information.

Feature	Directory Structure	Index File
Scope	**System-Wide:** Maps all files on the disk.	**File-Specific:** Maps content inside one large file.
Mapping Goal	Maps **File Name** $\rightarrow$ **Physical Disk Location**.	Maps **Data Key (e.g., Name)** $\rightarrow$ **Block Number (n)**.
Storage	Lives in a special "Volume Table of Contents" on the disk.	Is often a separate file created by a specific program.
Analogy	A **Building Directory** in a lobby (tells you which room is the Doctor's office).	A **Patient Chart** (tells you which page has the blood test results).

---

4. The "Big Picture": Partitions and Volumes

To understand why these are significant, you have to see the hierarchy of how a disk is built:

1. **Disk:** The physical hardware.
2. **Partitions (Slices/Minidisks):** The disk is subdivided into smaller "virtual disks".
3. **Volume:** An entity that contains a file system and its own **device directory** or **volume table of contents** to track its files.
4. **Directory:** The map within that volume.
5. **Files:** The individual containers of data.
6. **Index:** The internal "map" for a specific large file.