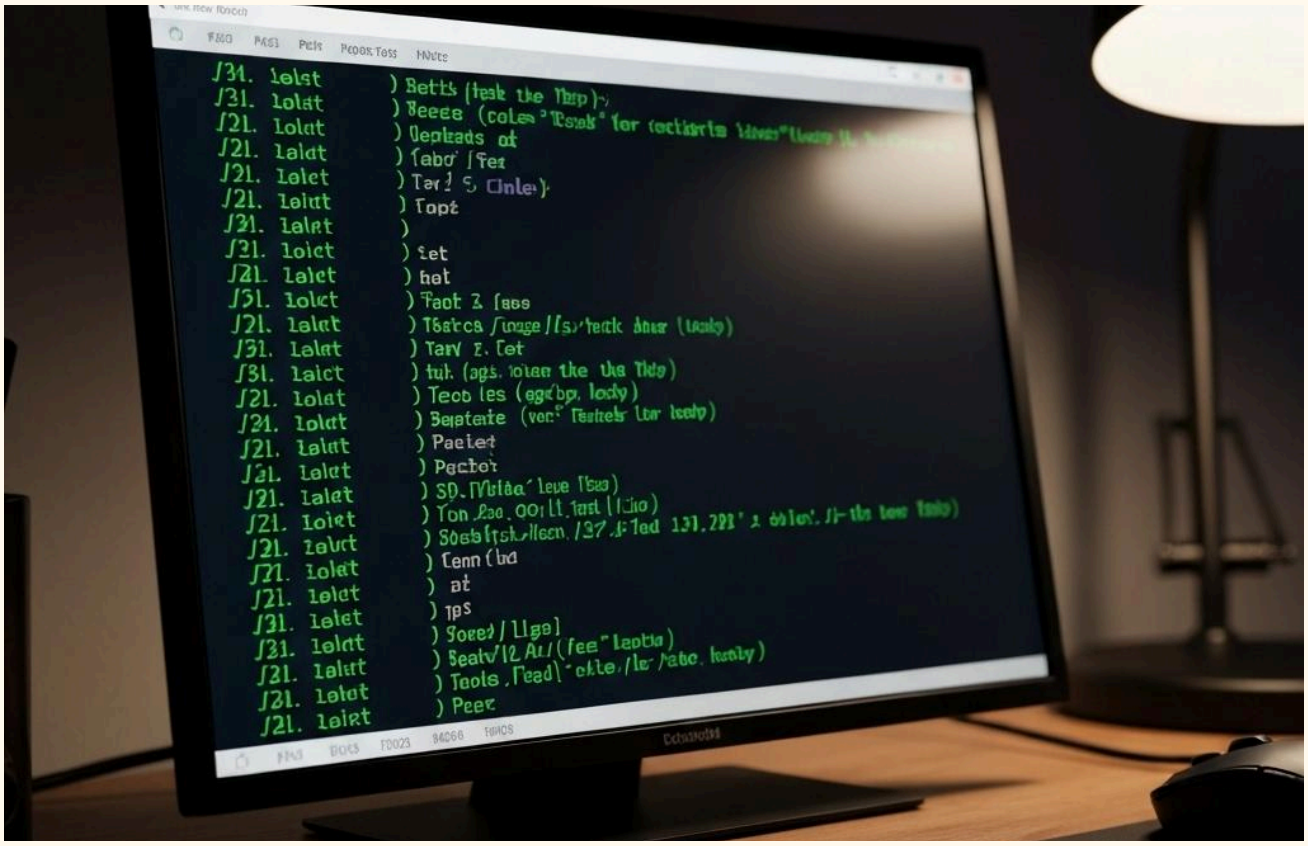
AIOps Study Journal · RHEL Series

Doc ID: SJ-RHEL-02 · Version 1.0

Paths · Inodes · Links

How Linux Identifies · Maps · Connects Files

By Sameer ur Rehman



1 · INTRODUCTION

In Linux, every file, directory, device, and socket lives inside one unified structure — a giant tree that begins at the **root directory `/`**.

Whether it's your wallpaper, a log file, or your keyboard driver, everything connects somewhere under `/`.

To move confidently inside this digital forest, you need to understand three pillars that form the base of the Linux filesystem:

Concept	Purpose
Paths	How Linux locates files
Inodes	How Linux identifies and tracks files
Links	How multiple names can point to the same file

If the filesystem were a city:

- **Paths** are street addresses.
- **Inodes** are the building registration numbers.
- **Links** are nicknames or shortcuts that lead to the same building.

Once you master these, the mysteries of file permissions, ownership, and storage usage will become clear.

---

2 · PATHS IN LINUX

A path is simply the address leading to a file or folder.

Every file you work with in Linux — from your wallpapers to kernel drivers — has a **path** that tells you *where* it lives in the filesystem.

Think of it like a set of directions on a map: “Start here, go there, and end up at this file.”

Linux understands two major kinds of paths:

- **Absolute Paths** — full addresses starting from the root `/`.
- **Relative Paths** — short directions based on where you currently stand.

Let’s start with the simplest question.

2.1 How to Read a Path

A **path** describes a location.

For example:

```
/home/sameer/task1.txt
```

This tells Linux:

- Start at `/home` — the users’ directory.
- Then open the folder **sameer**.
- Inside it, find the file **task1.txt**.

Tip for beginners:
When reading any path aloud, insert the phrase “which contains” between each step.

So `/home/sameer/task1.txt` means:

> “Start in `/(root)`, then `home`, which contains `sameer`, which contains `example.txt`.”

This small mental habit makes it easier to visualize where a file lives.

Example

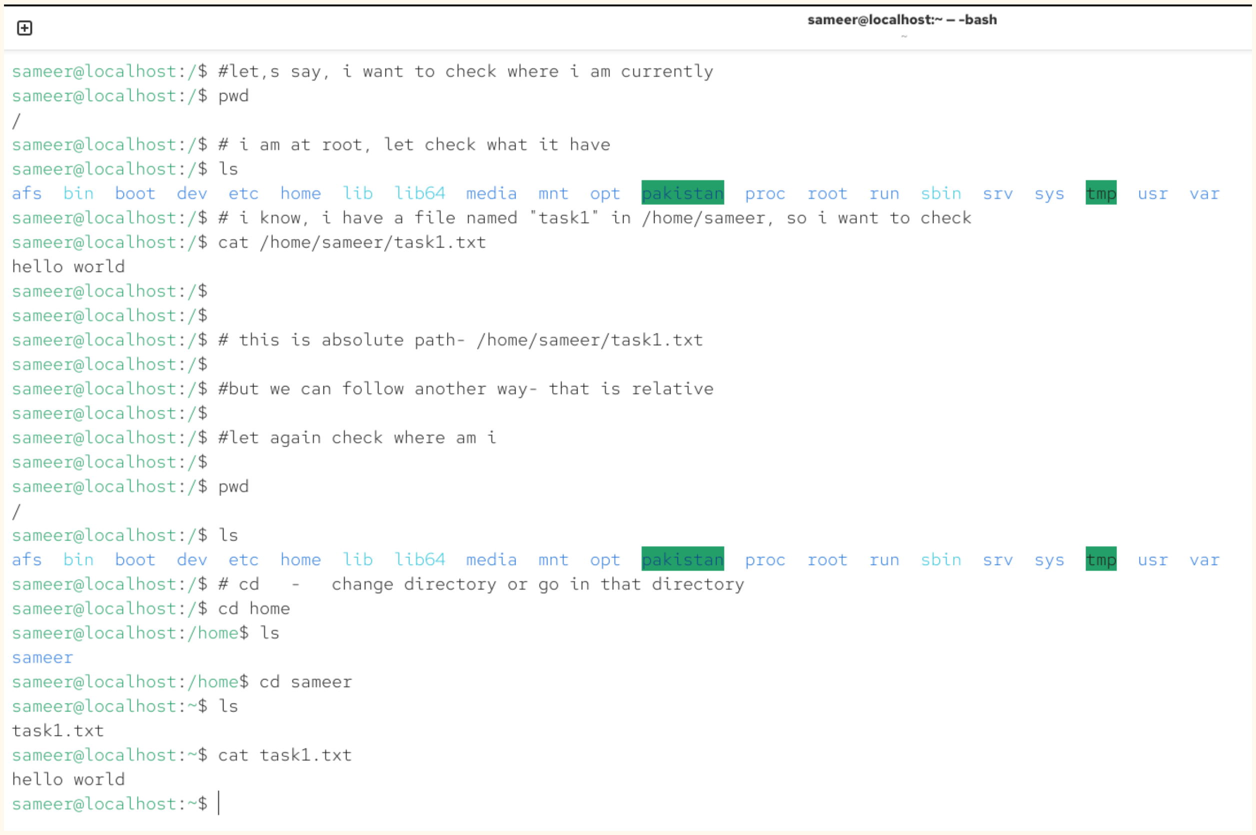
Let's say I have a folder named **"sameer"** inside my **/home** directory, and I want to look for a file called **task1** in it.

I can simply type:

`ls /home/sameer/task1`

or i can check what i have:

`cat /home/sameer/task1`



sameer@localhost:~ -- -bash

```
sameer@localhost:/$ #let,s say, i want to check where i am currently
sameer@localhost:/$ pwd
/
sameer@localhost:/$ # i am at root, let check what it have
sameer@localhost:/$ ls
afs  bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  pakistan  proc  root  run  sbin  srv  sys  tmp  usr  var
sameer@localhost:/$ # i know, i have a file named "task1" in /home/sameer, so i want to check
sameer@localhost:/$ cat /home/sameer/task1.txt
hello world
sameer@localhost:/$
sameer@localhost:/$
sameer@localhost:/$ # this is absolute path- /home/sameer/task1.txt
sameer@localhost:/$
sameer@localhost:/$ #but we can follow another way- that is relative
sameer@localhost:/$
sameer@localhost:/$ #let again check where am i
sameer@localhost:/$
sameer@localhost:/$ pwd
/
sameer@localhost:/$ ls
afs  bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  pakistan  proc  root  run  sbin  srv  sys  tmp  usr  var
sameer@localhost:/$ # cd   -   change directory or go in that directory
sameer@localhost:/$ cd home
sameer@localhost:/home$ ls
sameer
sameer@localhost:/home$ cd sameer
sameer@localhost:~$ ls
task1.txt
sameer@localhost:~$ cat task1.txt
hello world
sameer@localhost:~$ |
```

Key Takeaway:

Every slash `/` separates one level of the filesystem tree — like folders within folders.

2.2 Absolute Path — The "Full Address"

An **absolute path** tells Linux the **exact route** from the root directory (`/`) to a file or folder — no assumptions, no shortcuts.

It’s called “absolute” because it always starts at the absolute beginning of your filesystem.

Let’s say you’re currently working inside the **/etc/** directory, busy with some configuration files. Now suddenly, you need to check another file — for example, **task1.txt** that’s stored in **/home/sameer**.

You don’t need to move from your current location. Just use the **absolute path** like this:

```
cat /home/sameer/task1.txt
```

No matter where you are in the system, this command will still work — because the absolute path starts from the root (/) and points directly to the file’s exact location.

That’s why understanding the **Linux filesystem hierarchy** is so important — it helps you know *what is where*, and lets you reach any file easily using its absolute path.

That’s the power of an **absolute path** — it works the same way for everyone, from any location.

Key Takeaway:
Absolute paths are global truths.
They don’t depend on you — they depend on the filesystem structure.

They’ll work the same whether you’re in `/home`, `/var`, or `/tmp`.

2.3 Relative Path — “From Where I Stand”

A relative path doesn’t start from `/`.
Instead, it starts from your current working directory — the place you’re standing in.

Path Type	Starts From	Example	Works Anywhere?	Notes
Absolute Path	Root `/`	`/home/sameer/docs/TASK1.txt`	Yes	Full address from root — always valid
Relative Path	Current directory `.`	`../reports/file.txt`	No	Depends on current location (pwd)

Relative path = “Go two streets back, then turn left.”
The directions only make sense if you know where you start.

Think of a relative path like giving directions from where you currently are.

It’s like saying:

“Go two streets back, then turn left.”

These directions only make sense if you know your starting point.

Let’s go step by step and see how it behaves in real use.

When you type:

```
cd sameer
```

nothing prints on the screen — but you’ve now moved inside the folder named documents.

Next, if you run:

```
ls task1.txt
```

you’ll see:

```
task1.txt
```

That’s because the file is present inside your current directory.

Now try this:

```
cat ../file.txt
```

and you’ll get something like:

```
Hello world
```

This time, you’re reading a file that lives one level above your current folder.
The `..` part means “go to the parent directory,” while `.` always refers to “the current one.”

Key Takeaway:

Use relative paths when writing scripts or working on shared projects — they make your code portable.

No matter where the project folder moves, the paths will still work correctly as long as the relative structure stays the same

3 · INODES – THE HIDDEN INDENTITY OF FILES

An inode (index node) is a small data record that stores everything about a file except its name and content.

Each file or directory has one unique inode number.

While filenames can change or move, the inode remains the file's real fingerprint inside the filesystem.

Think of it like a person's National ID card: the name can change, but the ID number stays the same.

Don't get confuse:

Inode = Metadata + Disk Pointers

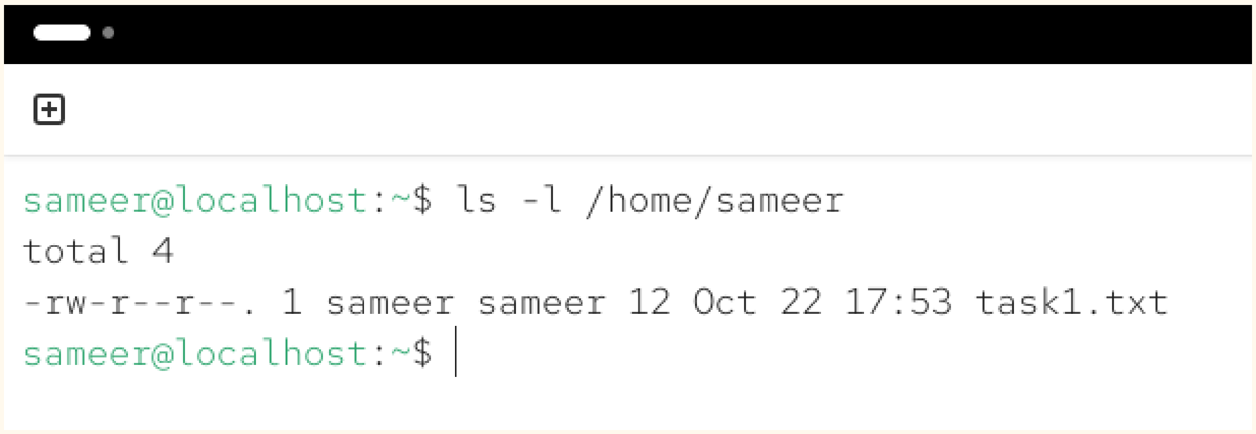
Directory = Name point that Inode Number

So, when you open a file, Linux first finds the inode number from the directory, then uses that inode to locate the actual data blocks.

3.1. What an inode store

Whenever you list files with `ls -l`, most of that information (like owner, permissions, and timestamps) comes directly from the inode.

Here's what an inode typically stores:



```
sameer@localhost:~$ ls -l /home/sameer
total 4
-rw-r--r--. 1 sameer sameer 12 Oct 22 17:53 task1.txt
sameer@localhost:~$ 
```

Now let's break this line down piece by piece:

Field	Example	Meaning	Stored in Inode?
`-rw-r--r--.`	File type (`-` = regular file) + permissions	Who can read, write, execute	Yes
`1`	Link count	Number of hard links to this inode	Yes
`sameer`	Owner name (UID → name)	User who owns the file	Yes
`sameer`	Group name (GID → name)	Group owning the file	Yes
`12`	File size (in bytes)	Size of file content	Yes
`Oct 22 17:53`	Last modification time (mtime)	When content was last changed	Yes
`task1.txt`	Filename	Label stored in directory entry	

3.2 What Happens Internally

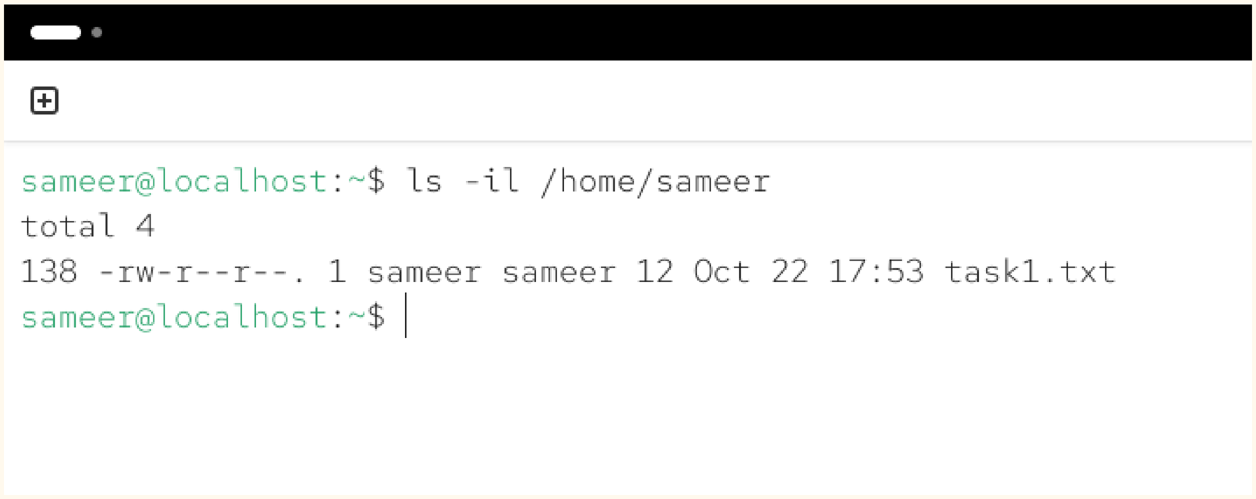
`/home/sameer` is a **directory file** containing an entry like:

```
task1.txt → inode 138
```

- The kernel looks up that inode number (e.g., `138`) in the filesystem's **inode table**, reads its metadata, and prints the above details.

- The **inode** holds everything except the file name itself.

When you run **`ls -il`**, it works just like `ls -l` — showing permissions, owner, size, and timestamps — **but** it also prints the **inode number** at the start of each line.



```
sameer@localhost:~$ ls -il /home/sameer
total 4
138 -rw-r--r--. 1 sameer sameer 12 Oct 22 17:53 task1.txt
sameer@localhost:~$ 
```

You can remember what an inode stores using the mnemonic **POT-TIP**

Here's how it expands:

Letter	Meaning	Description
P	Permissions	Who can read, write, or execute
O	Owner	User ID (UID) and Group ID (GID)
T	Timestamps	atime, mtime, ctime, btime
T	Type	File, directory, or link
I	Inode + Link Count	File's unique ID and number of hard links
B	Block Pointers	Addresses of data blocks on disk

3.3. What it does not store

Filename — kept inside directory entries
Path — derived from directory hierarchy
File content — stored in data blocks

Note: Renaming a file doesn't change its inode; only the directory entry's label updates.

Understanding Directory Entries

Here, it's essential to understand the relationship between directory entries.

In Linux, **everything is a file** — whether it's a text file, a device, or a directory.
A directory is **not magic**; it's just a *special file type* that stores the **list of names and inode numbers** of the items inside it.

Example	Meaning
`/home/user/note.txt`	Regular file
`/dev/sda`	Device file
`/home/user/`	Directory file

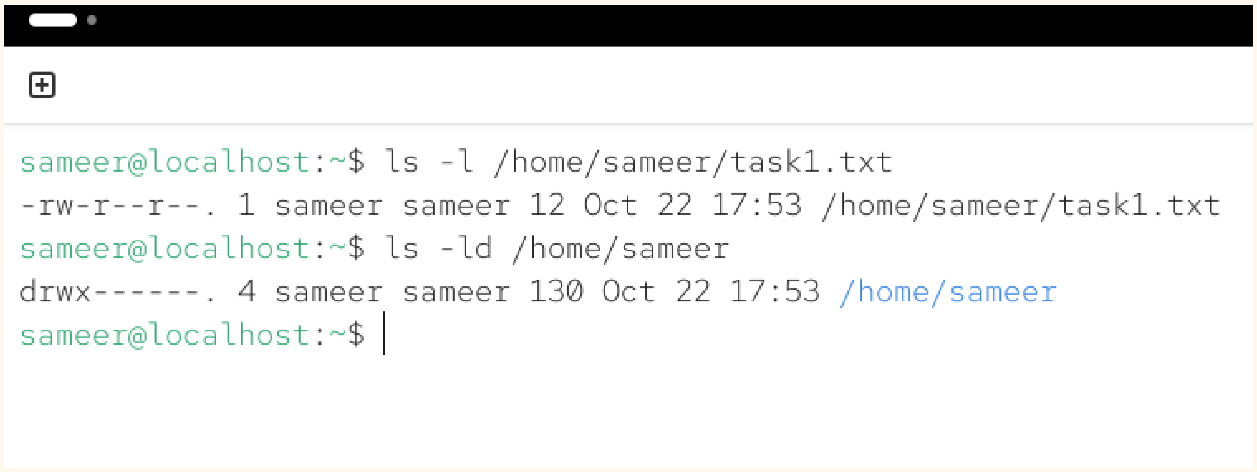
Checking Directory Type

To **check that `/home/sameer` is a directory file (type `d`)**, you can use this commands

```
ls -ld /home/user/
```

The first letter `d` means it's a directory.

If it were a normal file, it would show `-` instead of `d`



```
sameer@localhost:~$ ls -l /home/sameer/task1.txt
-rw-r--r--. 1 sameer sameer 12 Oct 22 17:53 /home/sameer/task1.txt
sameer@localhost:~$ ls -ld /home/sameer
drwx------. 4 sameer sameer 130 Oct 22 17:53 /home/sameer
sameer@localhost:~$
```

So, Directory = Special File

Each directory file has its own inode and data blocks.
But instead of user content, those blocks store directory entries — pairs of:

```
filename → inode number
```

Example (conceptually):

```
/home/sameer
 ├── task1.txt  → inode 12345
 └── task2.txt → inode 12346
```

So the /home directory's *data block* doesn't hold text — it holds this mapping table.

What Happens When a File is Renamed or Moved

- The **inode number of file stays the same** — the file's metadata and data blocks do **not** change.
- Only the **directory entry (name → inode mapping)** is modified:
    - The **old directory entry** is removed.

    - A **new directory entry** is created in the target directory (or with the new name).

- The inode itself remains untouched, so permissions, ownership, timestamps, and data are all preserved.
- but, if two partitions use the **same filesystem type** (e.g., both formatted as `ext4`), they are still **different filesystems** in Linux terms — because each partition maintains its **own independent inode table** and **block space**, so the inode number changed.

3.4. Connecting Directory Entries and Data Storage

So far, we've learned that a **directory** stores only *names* and *inode numbers*.
But what about the **actual content** of those files — the text, images, or program data inside them?
That information doesn't live in the inode or directory; it resides in another structure called **data blocks**.

Each inode contains **pointers** that tell the filesystem *where* a file's data blocks are located on disk.
This is how Linux links everything together:

```
Directory entry → Inode → Data blocks
```

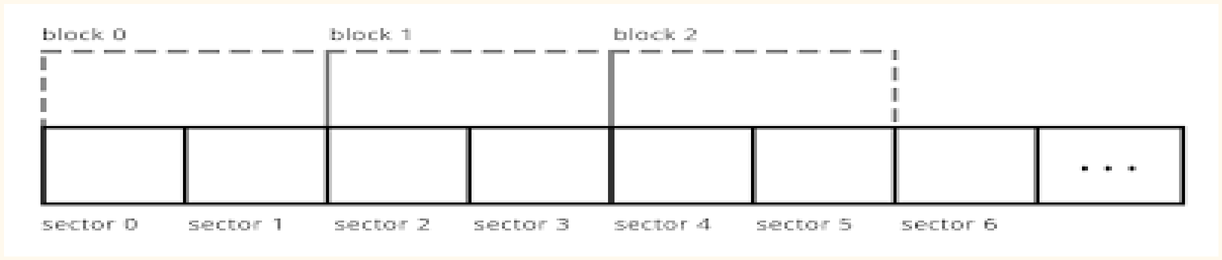
In short, the directory tells **"what is the file's name and inode number"**,
and the inode tells **"where the file's data lives."**

Sectors and Blocks

Unit	Typical Size	Role
Sector	512 bytes	Smallest hardware read/write unit on a disk
Block	4 KiB (4096 bytes)	Smallest filesystem allocation unit (ext4, XFS, etc.)

Sectors are like **bricks**, and blocks are **rooms built from those bricks**.
The filesystem never stores files smaller than one block.

Even a **1-byte** file still occupies **one full 4 KiB block** — because disk space is allocated per block, not per byte.



Example — Small File on Disk

```
echo "A" > note.txt
```

- The file has **1 byte** of content.
- Linux still allocates **one complete 4 KiB block** for it.
- It also uses **one inode** to store metadata like permissions and timestamps.

Insight:

Every file consumes **one inode + at least one block**, no matter how small it is.

3.5 Where Inodes and Blocks Live

When we create a partition and format it with a filesystem (for example, using `mkfs.ext4`), Linux organizes the disk into several well-defined regions.

(We'll discuss how partitions and filesystems actually work in detail in a later episode.)

Each region has a specific job — like departments in an office.

Component	What It Does
Boot Block	The very first section of the partition. It can hold bootloader code that helps the system start up.

Superblock	Acts like the filesystem's control room — it records key details such as block size, total blocks, and number of inodes.
Inode Table	A database of all inodes — one entry for every file and directory, storing metadata (permissions, timestamps, owners, etc.).
Data Blocks	The actual workspace where file contents and directory entries are stored. Each file's inode points to these blocks.

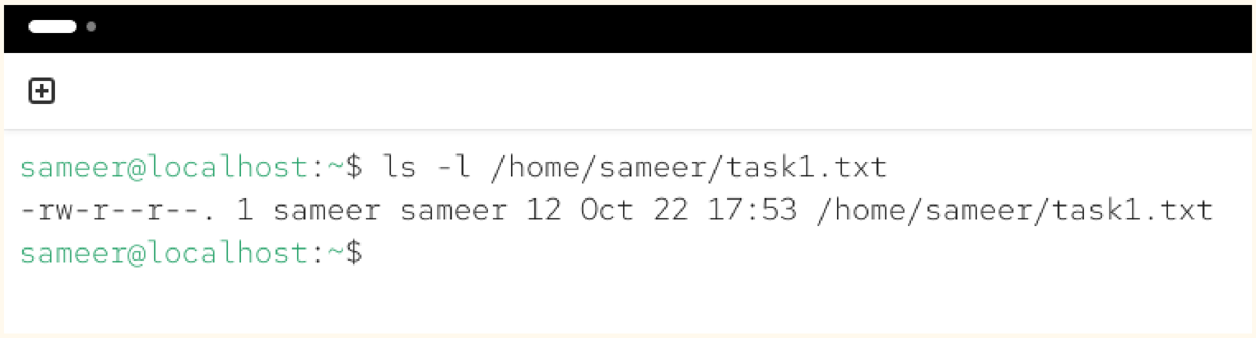
Think of it like this:

- The Boot Block is the doorbell and nameplate of the house.
- The Superblock is the blueprint and directory map.
- The Inode Table is the index book listing who lives where.
- The Data Blocks are the rooms holding the real stuff.

The **inode table** stores all inode structures, each identified by a unique inode number.

Every inode contains pointers that **link to the physical addresses of data blocks** on the disk — this is how the filesystem knows *where* the actual file content lives.

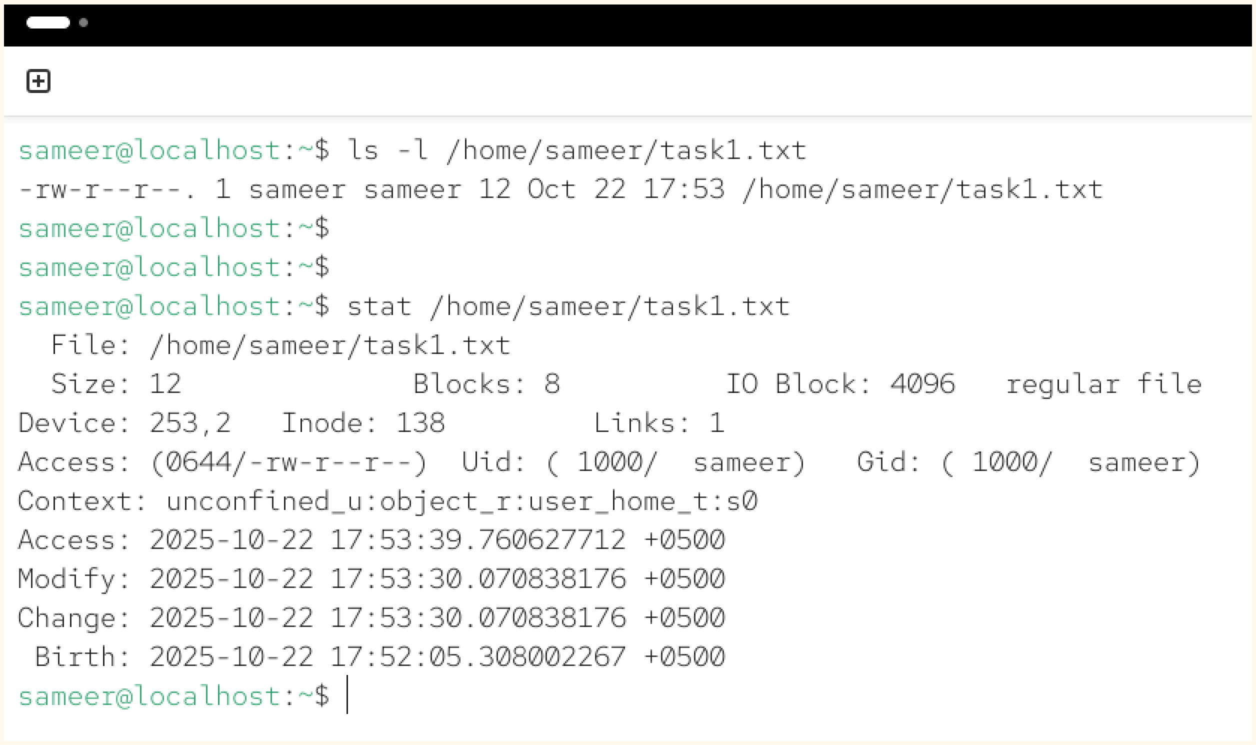
Let understand this by an example



```
sameer@localhost:~$ ls -l /home/sameer/task1.txt
-rw-r--r--. 1 sameer sameer 12 Oct 22 17:53 /home/sameer/task1.txt
sameer@localhost:~$
```

Command — ls -l /home/sameer/task1.txt shows that Linux knows this file is 12 bytes long, but it doesn't say how much space it actually consumes on disk.

That's where stat helps. Command 2 — stat /home/sameer/task1.txt



```
sameer@localhost:~$ ls -l /home/sameer/task1.txt
-rw-r--r--. 1 sameer sameer 12 Oct 22 17:53 /home/sameer/task1.txt
sameer@localhost:~$
sameer@localhost:~$
sameer@localhost:~$ stat /home/sameer/task1.txt
  File: /home/sameer/task1.txt
  Size: 12              Blocks: 8          IO Block: 4096   regular file
Device: 253,2   Inode: 138         Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/  sameer)   Gid: ( 1000/  sameer)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2025-10-22 17:53:39.760627712 +0500
Modify: 2025-10-22 17:53:30.070838176 +0500
Change: 2025-10-22 17:53:30.070838176 +0500
 Birth: 2025-10-22 17:52:05.308002267 +0500
sameer@localhost:~$
```

What's Happening Technically

When you created this file:

```
echo "Hello" > task1.txt
```

1. **The kernel allocates one inode** (inode 138 in your case).
   - Inode stores metadata (permissions, timestamps, pointers).
2. **It writes your 12 bytes of content** into a new **data block**.
   - Each data block = 4 KiB (4096 bytes) on ext4.
3. **The inode's pointer** now points to that block's physical address.

4. `ls -l` reads the inode and shows the *logical size* → 12 bytes.

5. `stat` shows both the *logical size* (12) and *physical space used* (8 sectors = 1 block = 4 KiB).

Key Takeaway

Concept	Logical	Physical
File Size (`ls -l`)	12 bytes	How much data file contains
Blocks (`stat`)	8 × 512 = 4096 bytes	How much space filesystem allocated
Reason	Filesystem allocates per block	Smallest allocatable unit = 4 KiB
Conclusion	Even tiny files occupy a full 4 KiB block + one inode	

Insight:

Linux filesystems trade a bit of wasted space for simplicity and speed. Each file always gets at least one full 4 KiB block — making reads and writes faster and predictable for the kernel.

Key to Remember — File Size, Block, and Sector Relationship

Even a **tiny 12-byte file** still consumes **one full filesystem block** on disk.

Unit	Typical Size	Purpose
Sector	512 bytes	Smallest hardware read/write unit on disk
Block	4 KB (4096 bytes)	Smallest filesystem allocation unit used by ext4/xfs

1 Block = 8 Sectors	(8 × 512 = 4096)	Filesystem groups 8 sectors into one 4 KB block

So when you check with `stat`, you see:

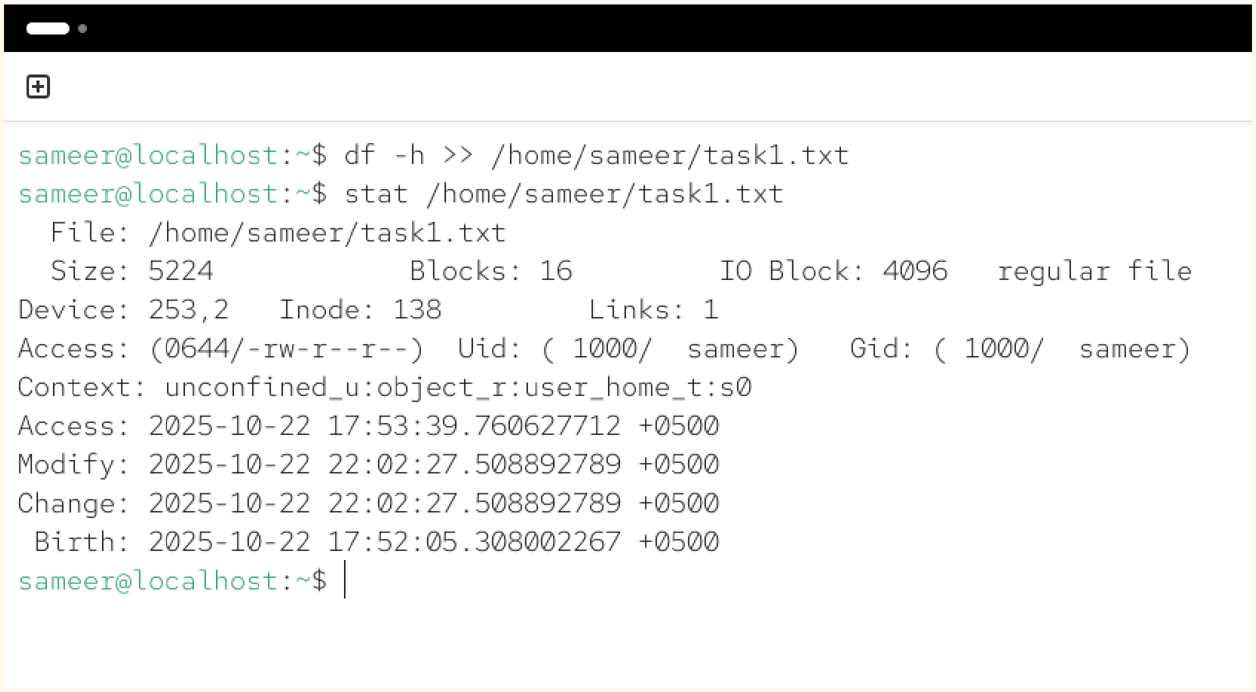
```
Size: 12        Blocks: 8        IO Block: 4096
```

This means:

- The file's **logical size** is **12 bytes** — that's the real data you wrote.
- The filesystem **allocated one full 4 KB block** to store it.
- `stat` reports **8 blocks** because it counts **512-byte sectors** (the POSIX standard unit).

`1 block = 8 sectors = 4 KB`
→ even the smallest file always takes **one full block + one inode**.

Here i increased size:



```
sameer@localhost:~$ df -h >> /home/sameer/task1.txt
sameer@localhost:~$ stat /home/sameer/task1.txt
  File: /home/sameer/task1.txt
  Size: 5224            Blocks: 16         IO Block: 4096   regular file
Device: 253,2   Inode: 138         Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/  sameer)   Gid: ( 1000/  sameer)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2025-10-22 17:53:39.760627712 +0500
Modify: 2025-10-22 22:02:27.508892789 +0500
Change: 2025-10-22 22:02:27.508892789 +0500
 Birth: 2025-10-22 17:52:05.308002267 +0500
sameer@localhost:~$
```

Explanation

Property	Before	Now
Size	12 bytes	5224 bytes
Blocks	8 (1 × 4KB block)	16 (2 × 4KB blocks)
Disk Space Used	4 KB	8 KB

So:

- At 12 bytes → only **1 block (4 KB)** was needed.
- At 5224 bytes → data exceeded one block (4096 bytes), so the filesystem allocated **a second block**.

Since each filesystem block = 8 sectors (8 × 512 = 4096),
Two blocks = 16 sectors total — and `stat` always reports blocks in **512-byte units**, not filesystem blocks.

---

4 · WHAT ARE LINKS?

When you make a movie shortcut on your desktop, you don't copy the movie — you just point to it. That's exactly what Linux links do.

A link is not a duplicate of a file — it's another name leading to the same data.

4.1 Understanding Hard and Soft Links in Linux

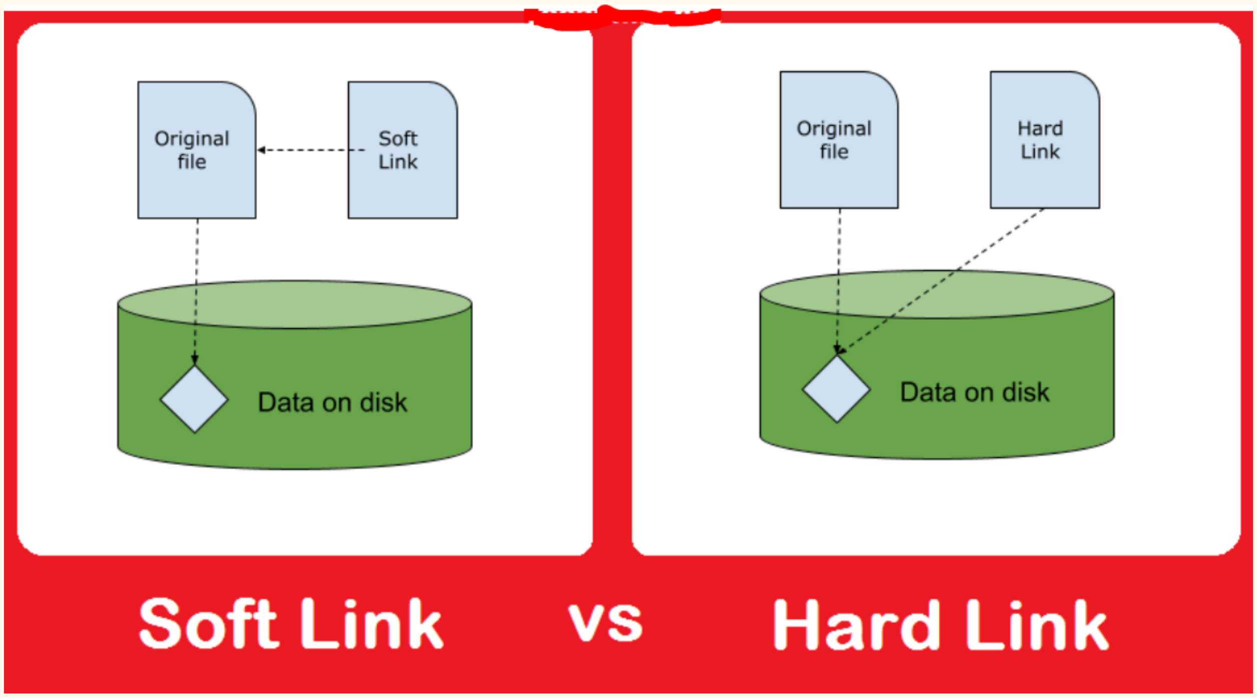
In Linux, every file is represented by an **inode** — a unique digital ID card that stores all its important details like **ownership, permissions, size, and timestamps**.

Now, when we create a **link**, we're not making a new file or copy. Instead, we're simply adding another **reference** to the same inode — just like giving a person another **nickname**.
No matter which name you use, both point to the same person — or in this case, the same file

Links in Linux are of two main types:

- **Hard Links:** Another valid name for the same inode.
- **Soft (Symbolic) Links:** A special file that stores the path of another file.

To understand this better, think of a file as a **house**:
A **hard link** is like adding another **door** to the same house — both doors open to the same space.
A **soft link** is like putting a **signboard** that says, "The house is that way."
If the house moves, the signboard becomes useless.



4.2 What Are Hard Links in Linux?

A **hard link** is a second name for the same physical file. It points directly to the same inode and data blocks.
When you create a hard link, you don't copy the file; you just add another entry in the directory that points to the same inode.

Key Characteristics of Hard Links

- They share the **same inode number** as the target file.
- They **cannot link directories** (for safety reasons).
- They **cannot cross filesystem boundaries**.

- If the original file is deleted, the hard link **still works** because both names point to the same inode and data.

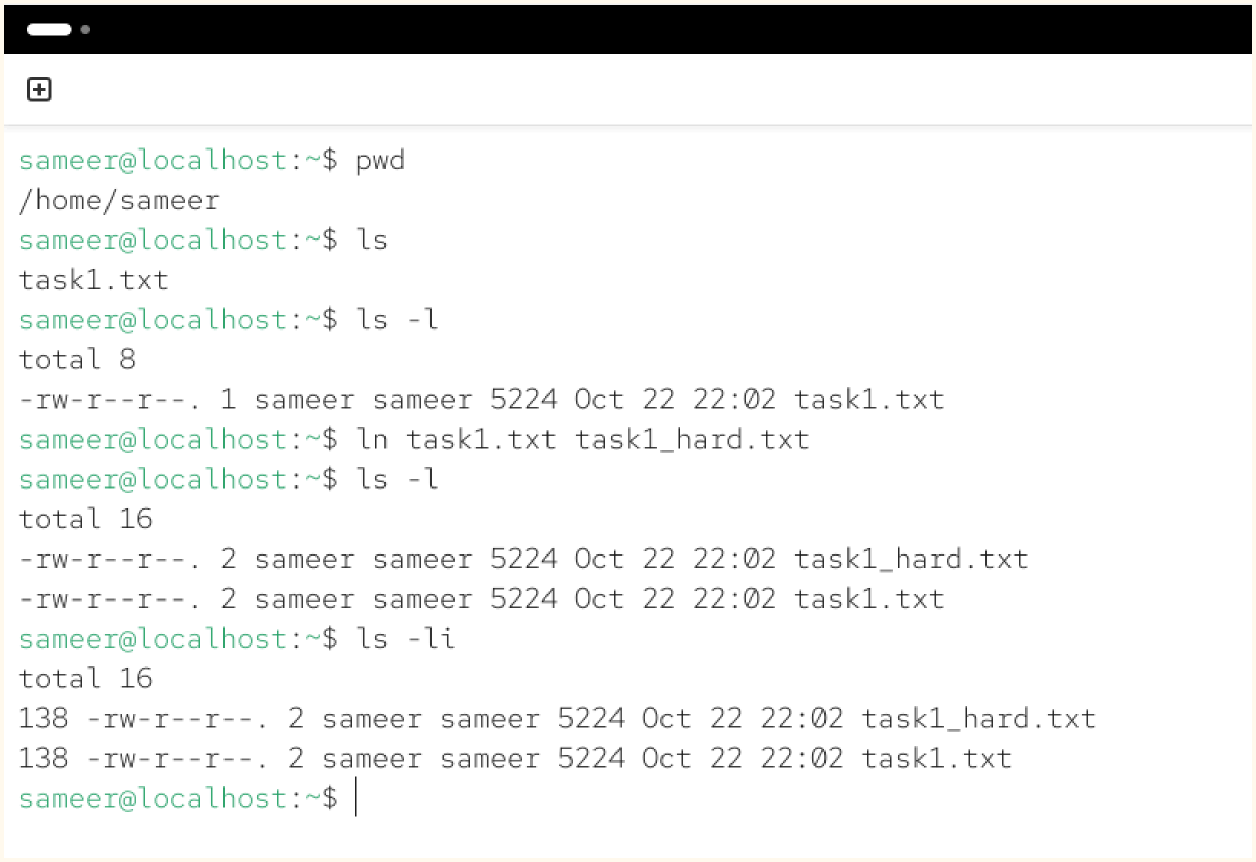
In short, the file's data remains alive as long as at least one hard link exists.

Example — Creating a Hard Link

`ln task1.txt task1_hard.txt`

`ls -li`

You'll notice both files share the same inode number — proof that they are identical.



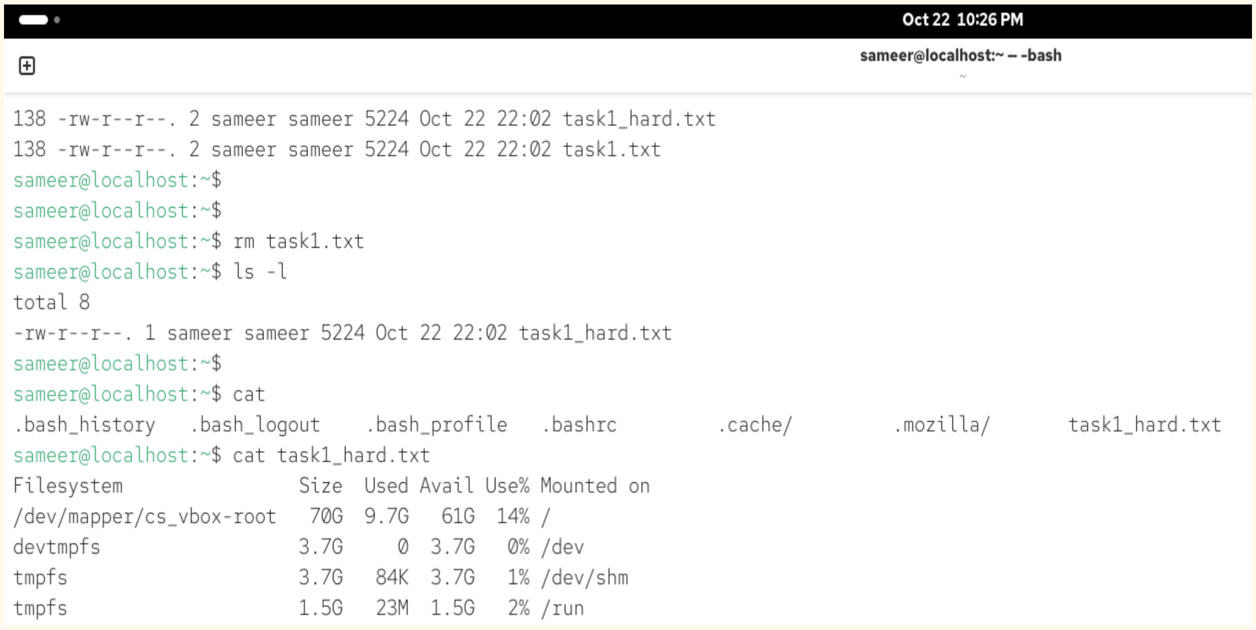
```
sameer@localhost:~$ pwd
/home/sameer
sameer@localhost:~$ ls
task1.txt
sameer@localhost:~$ ls -l
total 8
-rw-r--r--. 1 sameer sameer 5224 Oct 22 22:02 task1.txt
sameer@localhost:~$ ln task1.txt task1_hard.txt
sameer@localhost:~$ ls -l
total 16
-rw-r--r--. 2 sameer sameer 5224 Oct 22 22:02 task1_hard.txt
-rw-r--r--. 2 sameer sameer 5224 Oct 22 22:02 task1.txt
sameer@localhost:~$ ls -li
total 16
138 -rw-r--r--. 2 sameer sameer 5224 Oct 22 22:02 task1_hard.txt
138 -rw-r--r--. 2 sameer sameer 5224 Oct 22 22:02 task1.txt
sameer@localhost:~$
```

Now remove the original:

`rm task1.txt`

`cat task1_hard.txt`

Output: `Hello World`
Even after deleting `task1.txt`, the data survives because the inode still exists through `task1_hard.txt`.



```
138 -rw-r--r--. 2 sameer sameer 5224 Oct 22 22:02 task1_hard.txt
138 -rw-r--r--. 2 sameer sameer 5224 Oct 22 22:02 task1.txt
sameer@localhost:~$
sameer@localhost:~$
sameer@localhost:~$ rm task1.txt
sameer@localhost:~$ ls -l
total 8
-rw-r--r--. 1 sameer sameer 5224 Oct 22 22:02 task1_hard.txt
sameer@localhost:~$
sameer@localhost:~$ cat
.bash_history   .bash_logout    .bash_profile   .bashrc         .cache/         .mozilla/       task1_hard.txt
sameer@localhost:~$ cat task1_hard.txt
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/cs_vbox-root   70G  9.7G   61G  14% /
devtmpfs                  3.7G     0  3.7G   0% /dev
tmpfs                     3.7G   84K  3.7G   1% /dev/shm
tmpfs                     1.5G   23M  1.5G   2% /run
```

4.3 What Are Soft (Symbolic) Links in Linux?

A **soft link**, also known as a **symbolic link**, is more like a shortcut in Windows.
It doesn't point directly to the inode but instead stores the **path** of the original file as text.

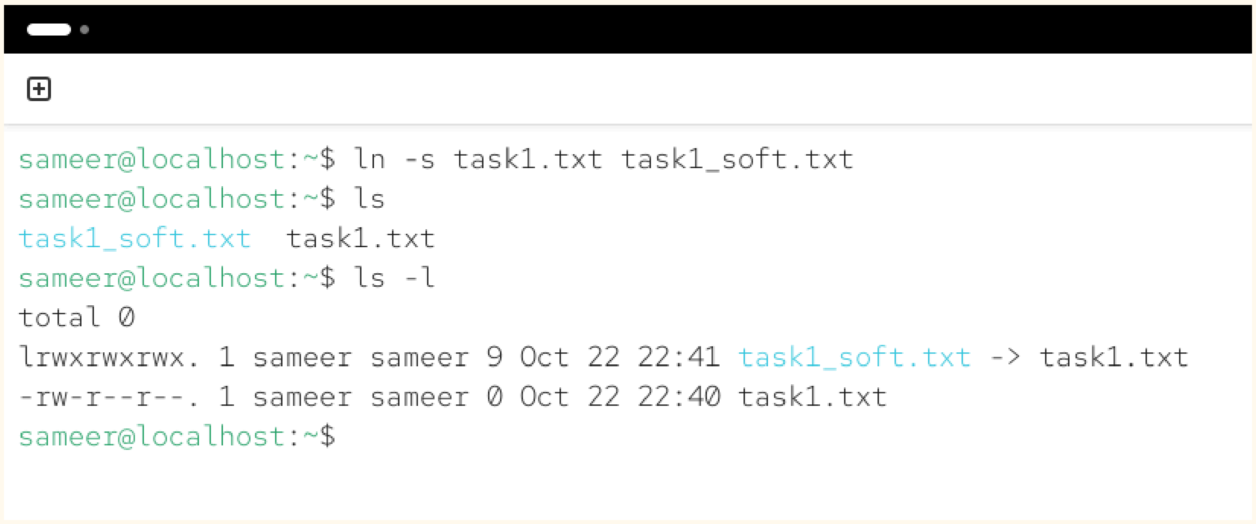
Key Characteristics of Soft Links

- They have a **different inode** than the original file.
- They can **cross filesystems** and **link directories**.
- If the original file is deleted or moved, the soft link **breaks** (becomes "dangling").
- It stores only the **path**, not the actual data.

Example — Creating a Soft Link

```
ln -s task1.txt task1_soft.txt
```

```
ls -l
```



```
sameer@localhost:~$ ln -s task1.txt task1_soft.txt
sameer@localhost:~$ ls
task1_soft.txt  task1.txt
sameer@localhost:~$ ls -l
total 0
lrwxrwxrwx. 1 sameer sameer 9 Oct 22 22:41 task1_soft.txt -> task1.txt
-rw-r--r--. 1 sameer sameer 0 Oct 22 22:40 task1.txt
sameer@localhost:~$
```

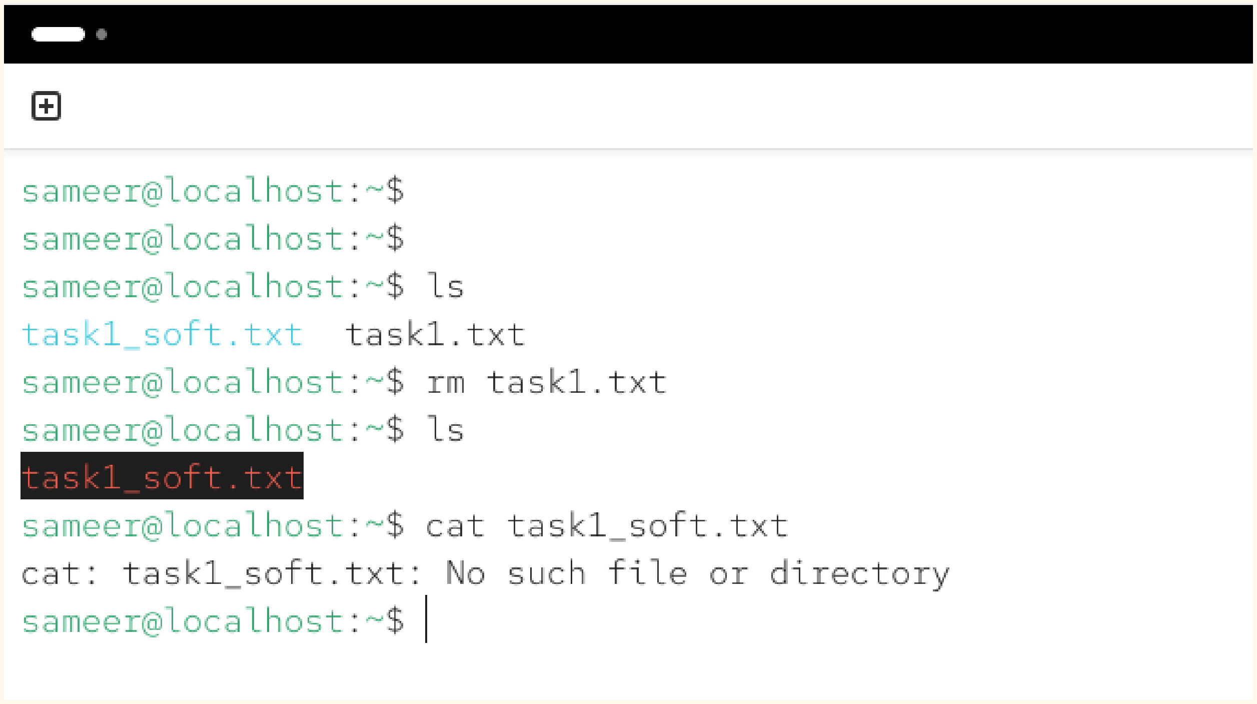
You’ll see:

`task1_soft.txt -> task1.txt`

Now remove the original file:

`rm task1.txt`

`cat task1_soft.txt`



```
sameer@localhost:~$
sameer@localhost:~$
sameer@localhost:~$ ls
task1_soft.txt  task1.txt
sameer@localhost:~$ rm task1.txt
sameer@localhost:~$ ls
task1_soft.txt
sameer@localhost:~$ cat task1_soft.txt
cat: task1_soft.txt: No such file or directory
sameer@localhost:~$
```

Output: `No such file or directory`
The soft link fails because its target path no longer exists

4.4 Difference Between Hard Links and Soft Links

Here's a quick side-by-side comparison:

Parameter	Hard Link	Soft Link
Inode	Same as the target file	Different from the target
File System	Must be in the same filesystem	Can cross filesystems
Directories	Not allowed	Allowed
Data	Points directly to data blocks	Points to file path
If Original Is Deleted	Link still works	Link breaks
Speed	Slightly faster	Slightly slower
Memory Usage	Less	More

Use **hard links** for efficiency when staying in the same filesystem.
Use **soft links** when you need flexibility — for directories or linking across partitions.

4.5 Resolving Symbolic Links in Linux

Sometimes links form chains, like this:

```
/usr/bin/python3 → /etc/alternatives/python3 → /usr/bin/python3.11
```

You can check the final resolved path using:

```
readlink -f /usr/bin/python3
```

```
or
```

`realpath /usr/bin/python3`

Command	Purpose
`readlink -f`	Follows all symbolic links until the actual file
`realpath`	Prints the canonical (absolute) path

Both help you find the real executable behind multiple linked shortcuts.

4.6 Common Issues and Fixes

Problem	Reason	Fix
Broken symlink	Target deleted or moved	Use `readlink -f` to trace and recreate
Hard link fails	Target on another filesystem	Use soft link instead
`mv` didn't change inode	Moved within same filesystem	Normal behavior

---

5 · SUMMARY

Hard and soft links are like two ways of referring to the same story:
One is **direct** (hard link), the other is **by address** (soft link).

A **hard link** gives durability — even if the original file disappears, the data lives on.
A **soft link** offers flexibility — it can point across disks, to directories, or anywhere else, but depends on the target's path.

Once you grasp inodes, directory entries, and links, you begin to see Linux not as a collection of files — but as a connected network of names, identities, and relationships.

Everything in Linux is connected like a living organism, **paths** define routes, **inodes** define identity, and **links** define relationships.
Understanding these transforms you from a user into a system administrator who sees how the filesystem really works.

Next Lesson

SJ-RHEL-03 · Ownership & Permissions

Learn how Linux controls who can read, write, or execute files — and how users, groups, and permission bits shape access across the system.