

glibc File System

20251029 Wed

Copyright (c) 2025 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

I used Microsoft Copilot to prepare this document



Microsoft Copilot

<https://copilot.microsoft.com> ⓘ

Microsoft Copilot: Your AI companion

Microsoft Copilot is your companion to inform, entertain and inspire. Get advice, feedback and straightforward answers. Try Copilot now.

In Linux, **symbolic links** and **hard links** are two ways to create alternate references to files, but they behave very differently. Here's a detailed comparison to help you understand their differences and use cases:

Symbolic Link (Symlink)

A symbolic link is a special file that **points to another file or directory by name**.


Characteristics

- Points to the **path** of the target file
- Can link to **directories**
- Can link to **non-existent files** (broken symlinks)
- Works **across filesystems**
- Has a **different inode** than the target

Example

Bash



 Copy

```
ln -s /home/user/data.txt shortcut.txt
```

Creates `shortcut.txt` as a symbolic link to `data.txt`.

Hard Link


A hard link is a **direct reference to the same inode** as the original file. It's essentially another name for the same file.

Characteristics

- Points to the **same inode** as the target
- Cannot link to directories (except by root in special cases)
- Cannot link across filesystems
- File is deleted only when **all hard links** are removed
- Changes to one hard link affect all others

Example

Bash ^

 Copy

```
ln /home/user/data.txt mirror.txt
```

Creates `mirror.txt` as a hard link to `data.txt`.

Comparison Table

Feature	Symbolic Link	Hard Link
Points to	File path	File inode
Can link to directories	✓ Yes	✗ No (normally)
Can link across filesystems	✓ Yes	✗ No
Can link to non-existent	✓ Yes (broken link)	✗ No
Shares inode with target	✗ No	✓ Yes
Affects target on delete	✗ No (link breaks)	✓ Yes (file remains)
Used for	Shortcuts, flexibility	Redundancy, durability

Use Cases

- **Symbolic links:** Ideal for shortcuts, version management, linking config files, or redirecting paths.
- **Hard links:** Useful for backup systems, deduplication, and ensuring file persistence even if one name is deleted.

In Linux, both **symbolic links** and **hard links** are ways to reference files, but they interact with **inodes** very differently. Here's a breakdown of how each relates to the inode structure in the filesystem:

What Is an Inode?

An **inode** is a data structure used by the filesystem to store metadata about a file — such as its size, permissions, timestamps, and disk block locations. Every file has a unique inode number within its filesystem.



Hard Links and Inodes


- A **hard link** is a direct reference to the same inode as the original file.
- Multiple hard links to a file share the same inode number.
- The file is only deleted when all hard links are removed and no process has it open.



Example

Bash ^

Copy

 `ln original.txt hardlink.txt`

- `original.txt` and `hardlink.txt` point to the **same inode**.
- Changes to one are reflected in the other.
- `ls -li` will show the same inode number for both.


Symbolic Links and Inodes

- A symbolic link is a separate file that contains a path reference to another file.
- It has its own inode, distinct from the target file.
- The symlink's inode stores the path string, not the target's data.

Example

Bash ^

 Copy


`ln -s original.txt symlink.txt`

- `symlink.txt` has a **different inode** than `original.txt`.
- If `original.txt` is deleted, `symlink.txt` becomes a **broken link**.

Comparison Table: Inode Behavior

Feature	Hard Link	Symbolic Link
Shares inode with target	✓ Yes	✗ No
Inode stores file data	✓ Yes	✗ No (stores path)
Inode number (via <code>ls -li</code>)	Same as target	Different from target
Breaks if target deleted	✗ No	✓ Yes

In Linux and other Unix-like systems, the relationship between a **file** and an **inode** is fundamental to how the filesystem works. Here's a clear breakdown:

What Is an Inode?

An **inode** (index node) is a data structure used by the filesystem to store metadata about a file — but not its name.

Inode Contains:

- File type (regular, directory, symlink, etc.)
- Permissions (read/write/execute)
- Owner (UID) and group (GID)
- File size
- Timestamps (creation, modification, access)
- Pointers to data blocks on disk

What Is a File?

A **file** in Linux is essentially:

- A name stored in a directory
- A pointer to an inode

So, the filename is just a label — the actual content and metadata live in the inode.


Relationship Between File and Inode

Concept	Description
<u>File name</u>	Stored in a <u>directory entry</u>
<u>Inode number</u>	Points to the actual file <u>metadata</u> and <u>data</u>
<u>Multiple names</u>	Can point to the same inode (via <u>hard links</u>)
<u>Deletion</u>	Removes the name; inode and data persist until all links are gone

Example

Bash




 Copy

```
ls -li myfile.txt
```


This shows the inode number of `myfile.txt`. You can create another name pointing to the same inode:

Bash



 Copy

```
ln myfile.txt anothername.txt
```



Now both names share the same inode — they are **hard links** to the same file.



Summary

- The **inode** is the true identity of a file in the filesystem.
- The **filename** is just a reference stored in a directory.
- You can have multiple filenames (hard links) pointing to the same inode.
- Symbolic links, by contrast, point to filenames — not inodes.

In Linux, a **directory** is a special type of file that stores information about other files and directories. It plays a key role in the hierarchical structure of the Linux filesystem.

What Is a Directory File?

- A **directory file** contains a list of entries.
- Each entry maps a filename to an inode number.
- The inode number points to the actual file metadata and data blocks.

So, a directory is essentially a lookup table: it tells the system where to find the inode for a given name.



How It Works

- When you run `ls` the system reads the directory file to list names.
- When you open a file like `notes.txt`, the system:
 1. Looks up `notes.txt` in the directory
 2. Finds its inode number
 3. Uses the inode to access the file's metadata and content



Technical Details

- Directories are implemented as files with a specific format understood by the filesystem (e.g., ext4, xfs).
- You can inspect a directory's inode using:

Bash ^

Copy

```
ls -ld mydir
```

- You can view raw directory contents using `ls`, but not with `cat`, since it's not a regular text file.

Filesystem Hierarchy


Linux follows the [Filesystem Hierarchy Standard \(FHS\)](#) GeeksForGeeks, which organizes directories like:

Directory	Purpose
<code>/</code>	Root of the filesystem
<code>/home</code>	User directories
<code>/etc</code>	Configuration files
<code>/bin</code>	Essential binaries
<code>/var</code>	Variable data (logs, caches)

Each of these is a directory file containing entries for other files and directories.

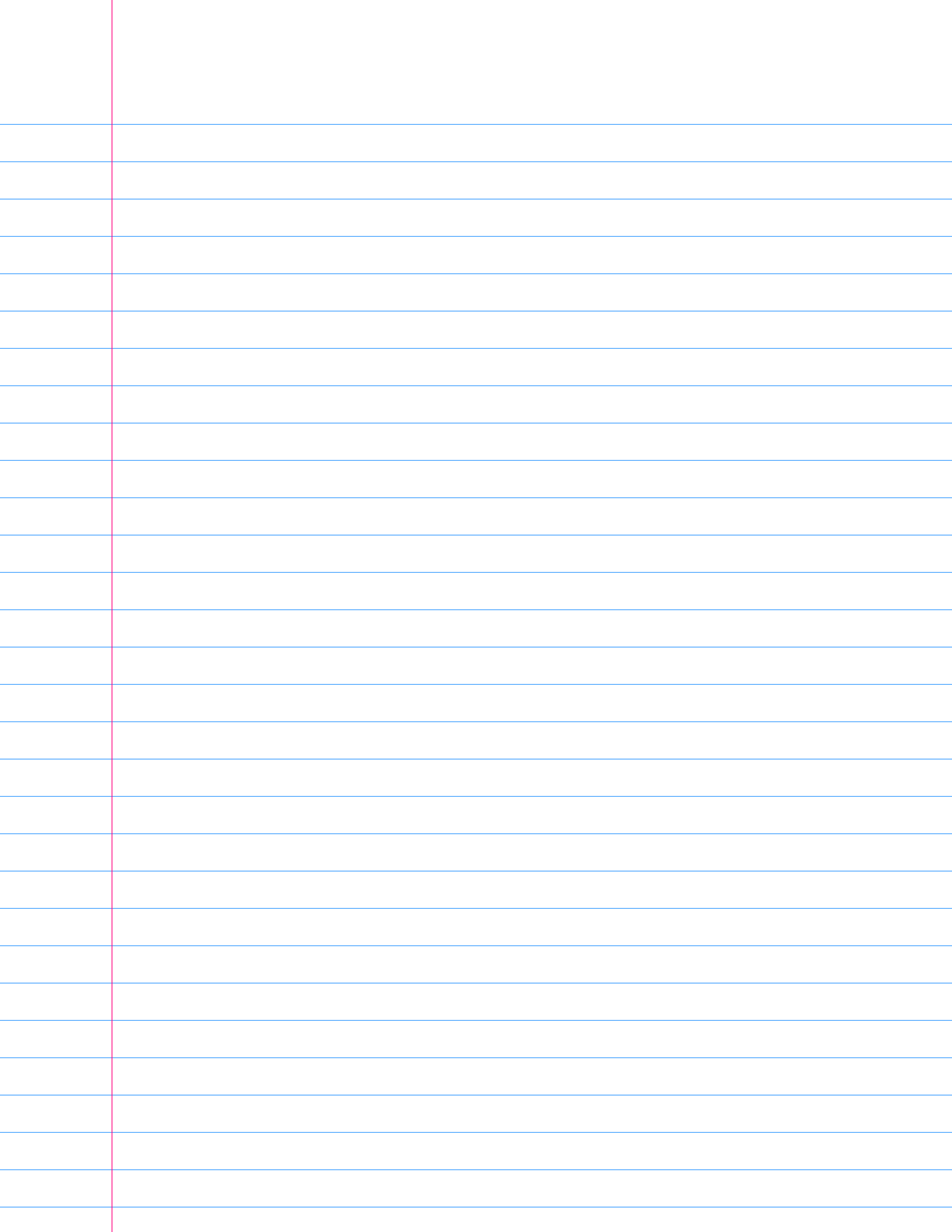
Example: Listing Directory Inodes

Bash ^

 Copy

```
ls -li /home
```

Shows inode numbers for each entry in `/home`.



In glibc, directory handling is implemented through a set of functions that wrap low-level system calls like `open()`, `read()`, and `close()` to provide a portable interface for working with directories. These functions are defined in the `dirent.h` header and implemented in glibc's source under the `sysdeps` and `dirent` directories.

Key glibc Directory Functions

Function	Purpose
<code>opendir()</code>	Opens a directory stream
<code>readdir()</code>	Reads entries from a directory stream
<code>closedir()</code>	Closes the directory stream
<code>rewinddir()</code>	Resets stream to beginning
<code>seekdir()</code>	Moves stream to a specific location
<code>telldir()</code>	Returns current location in stream

These functions operate on a `DIR*` type, which is an opaque structure internally defined by glibc.



Internal Implementation

- The `DIR` structure typically contains:
 - A file descriptor (`fd`) for the open directory
 - A buffer for directory entries
 - Position tracking for `seekdir()` and `telldir()`
- glibc uses the `getdents()` or `getdents64()` syscall under the hood to read raw directory entries from the kernel.
- The entries returned are parsed into `struct dirent` , which includes:
 - `d_name` : filename
 - `d_ino` : inode number
 - `d_type` : file type (regular, directory, symlink, etc.)



Source Location

You can explore glibc's directory implementation in its source code:

- `dirent/` — contains high-level directory functions
- `sysdeps/unix/sysv/linux/` — contains platform-specific syscall wrappers



Example Usage

C ^

Copy

```
#include <dirent.h>
#include <stdio.h>

DIR *d = opendir(".");
struct dirent *entry;

while ((entry = readdir(d)) != NULL) {
    printf("%s\n", entry->d_name);
}
closedir(d);
```

This lists all entries in the current directory using glibc's directory API.

In glibc, the `DIR` structure is an opaque type used to represent a directory stream — a handle for reading entries from a directory. It's defined in the header `<dirent.h>`, but its internal structure is hidden from user programs to maintain portability and abstraction.

What Is `DIR`?

- `DIR` is a typedef for a structure that holds state information about an open directory.
- You use it with functions like `opendir()`, `readdir()`, and `closedir()` to iterate over directory contents.

Example usage:

```
C ^ 📄 Copy  
  
#include <dirent.h>  
#include <stdio.h>  
  
DIR *d = opendir(".");  
struct dirent *entry;  
  
while ((entry = readdir(d)) != NULL) {  
    printf("%s\n", entry->d_name);  
}  
closedir(d);
```



Internal Structure (Simplified Overview)

While the full definition is internal to glibc and may vary by version, the structure typically includes:

- A file descriptor (`fd`) for the open directory
- A buffer to hold raw directory entries
- Position indicators for `seekdir()` and `telldir()`
- Possibly a lock for thread safety

In glibc source, it's defined in `dirent/dirent.h` and implemented in `dirent/opendir.c`, `readdir.c`, etc.

```
typedef struct __dirstream DIR;
```

__dirstream definition code

The `__dirstream` structure is the internal implementation behind the `DIR` type in glibc, used for directory stream operations like `opendir()`, `readdir()`, and `closedir()`. While the full source code is copyrighted and I can't reproduce it verbatim, I can summarize its key components and explain how it works.

Summary of `struct __dirstream`

This structure is defined in glibc's internal header (typically `dirent/dirent.h`) and includes fields like:

- `int fd` — the file descriptor for the open directory
- `size_t allocation` — size of the buffer allocated for reading entries
- `size_t size` — number of bytes currently in the buffer
- `off_t offset` — current offset within the buffer
- `char *buffer` — pointer to the buffer holding raw directory entries
- `struct dirent *current` — pointer to the current parsed entry
- Possibly a lock for thread safety

These fields allow glibc to wrap the low-level `getdents()` syscall and present a portable, buffered interface for reading directory contents.



How It Works

When you call `opendir()`, glibc:

1. Opens the directory using `open()`
2. Allocates a `__dirstream` structure
3. Initializes the buffer and state

When you call `readdir()`, glibc:

1. Uses `getdents()` to fill the buffer
2. Parses entries into `struct dirent`
3. Returns one entry at a time

dirent structure

The `struct dirent` in Linux is defined in `<dirent.h>` and represents a single entry in a directory stream. It's used with functions like `readdir()` to retrieve information about files and subdirectories within a directory.

Here's the standard definition of `struct dirent` as implemented in glibc:

```
C ^ Copy  
  
struct dirent {  
    ino_t      d_ino;      // Inode number  
    off_t      d_off;      // Offset to the next dirent  
    unsigned short d_reclen; // Length of this record  
    unsigned char d_type;    // Type of file (e.g., regular, directory, symlink)  
    char        d_name[];    // Null-terminated filename  
};
```



```
char d_name[]; // Null-terminated filename
```

🧠 Field Breakdown

Field	Description
<code>d_ino</code>	Inode number of the file
<code>d_off</code>	Offset to the <u>next directory entry</u>
<code>d_reclen</code>	Total length of <u>this <code>dirent</code> structure</u>
<code>d_type</code>	File type (e.g., <code>DT_REG</code> , <code>DT_DIR</code> , <code>DT_LNK</code>)
<code>d_name</code>	<u>Filename (null-terminated string)</u>

Note: `d_name` is a flexible array member, meaning its actual size depends on the length of the filename.

🔧 Example Usage

C ^

📄 Copy

```
#include <dirent.h>
#include <stdio.h>

DIR *dir = opendir(".");
struct dirent *entry;

while ((entry = readdir(dir)) != NULL) {
    printf("Name: %s, Inode: %lu\n", entry->d_name, entry->d_ino);
}

closedir(dir);
```

This code lists all entries in the current directory, showing their names and inode numbers.



File Type Constants

The `d_type` field can be one of the following:

Constant	Meaning
<code>DT_REG</code>	Regular file
<code>DT_DIR</code>	Directory
<code>DT_LNK</code>	Symbolic link
<code>DT_FIFO</code>	FIFO (named pipe)
<code>DT SOCK</code>	Socket
<code>DT_CHR</code>	Character device
<code>DT_BLK</code>	Block device
<code>DT_UNKNOWN</code>	Unknown type

st_mode

In Linux and other Unix-like systems, `st_mode` is a field in the `struct stat` structure that encodes both the **file type** and **file permissions**. It's returned by system calls like `stat()`, `fstat()`, and `lstat()`.



st_mode

Breakdown

The `st_mode` field is a bitmask that includes:

♦ File Type Bits (use `S_IFMT` mask)

Constant	Octal	File Type
<code>S_IFREG</code>	<code>0100000</code>	Regular file
<code>S_IFDIR</code>	<code>0040000</code>	Directory
<code>S_IFLNK</code>	<code>0120000</code>	Symbolic link
<code>S_IFCHR</code>	<code>0020000</code>	Character device
<code>S_IFBLK</code>	<code>0060000</code>	Block device
<code>S_IFIFO</code>	<code>0010000</code>	FIFO (named pipe)
<code>S_IFSOCK</code>	<code>0140000</code>	Socket

Use `st_mode & S_IFMT` to isolate the file type.

◆ Permission Bits

Constant	Octal	Meaning
<code>S_IRUSR</code>	<code>0400</code>	Read permission for owner
<code>S_IWUSR</code>	<code>0200</code>	Write permission for owner
<code>S_IXUSR</code>	<code>0100</code>	Execute permission for owner
<code>S_IRGRP</code>	<code>0040</code>	Read permission for group
<code>S_IWGRP</code>	<code>0020</code>	Write permission for group
<code>S_IXGRP</code>	<code>0010</code>	Execute permission for group
<code>S_IROTH</code>	<code>0004</code>	Read permission for others
<code>S_IWOTH</code>	<code>0002</code>	Write permission for others
<code>S_IXOTH</code>	<code>0001</code>	Execute permission for others



Example: Checking File Type and Permissions

C ^

Copy

```
#include <sys/stat.h>
#include <stdio.h>

void check_mode(const char *path) {
    struct stat st;
    if (stat(path, &st) == 0) {
        if ((st.st_mode & S_IFMT) == S_IFDIR)
            printf("It's a directory\n");
        if (st.st_mode & S_IRUSR)
            printf("Owner can read\n");
        if (st.st_mode & S_IWUSR)
            printf("Owner can write\n");
        if (st.st_mode & S_IXUSR)
            printf("Owner can execute\n");
    }
}
```

In Linux, `access()` is a system call used to check whether the calling process has permission to access a file in a specified way — **without actually opening or modifying the file**.

`access()`

System Call Overview



Prototype (from `<unistd.h>`)

C ^

Copy

```
int access(const char *pathname, int mode);
```



Parameters

- `pathname` : Path to the file or directory
- `mode` : One or more of the following flags:

Flag	Value	Meaning
<code>F_OK</code>	<code>0</code>	Check if the file exists
<code>R_OK</code>	<code>4</code>	Check read permission
<code>W_OK</code>	<code>2</code>	Check write permission
<code>X_OK</code>	<code>1</code>	Check execute/search permission

You can combine flags using bitwise OR (`|`), like `R_OK | W_OK`.



Example Usage

C ^

Copy

```
#include <unistd.h>
#include <stdio.h>

int main() {
    if (access("myfile.txt", R_OK | W_OK) == 0) {
        printf("File is readable and writable\n");
    } else {
        perror("access");
    }
    return 0;
}
```


does access check group and other permissions

Yes — the `access()` system call **does check group and other permissions**, but with a specific caveat:

✓ What `access()` Checks

`access(pathname, mode)` checks whether the **calling process** has the requested access (read, write, execute) to the file at `pathname`. It evaluates permissions based on:

- The **real user ID (UID)** and **real group ID (GID)** of the process
- The file's permission bits: owner, group, and others

So if the process is **not the owner**, `access()` will check the **group** and then **others** fields in the file's mode (`st_mode`).



Example

Suppose a file has these permissions:

Bash



Copy

```
-rw-r----- 1 alice devs 1234 Oct 29 14:00 report.txt
```

- Owner: `alice`
- Group: `devs`
- Others: no access

If a process runs under:

- UID of `bob`
- GID of `devs`

Then:

- `access("report.txt", R_OK)` will return `0` (success) — because `bob` is in the group and group has read permission.

If `bob` were in a different group:

- `access()` would fall back to checking "others" — and fail in this case.

✓ Return Values of `access()`

Return Value	Meaning
0	Access is allowed
-1	Access is denied or error occurred

If `access()` returns -1, it sets `errno` to indicate the reason. Common `errno` values include:

<code>errno</code>	Meaning
EACCES	Permission denied
ENOENT	File does not exist
ENOTDIR	A component of the path is not a directory
EROFS	Write access requested on read-only filesystem
EINVAL	Invalid mode argument

global errno

In Linux and POSIX systems, `errno` is a **global variable** that stores the error code from the most recent system call or library function that failed. It's defined as:

C ^

Copy

```
extern int errno;
```

However, you **should not** access `errno` directly. Instead, include `<errno.h>` and rely on the standard interface, because `errno` may be implemented as a macro or thread-local storage depending on the system.



Key Points About `errno`

- **Global per thread:** On modern systems, `errno` is thread-local, meaning each thread has its own copy.
- **Set only on error:** Functions set `errno` when they fail (return `-1` or `NULL`), but they don't reset it on success.
- **Not cleared automatically:** You must reset it manually if needed, e.g., `errno = 0;`
- **Used with:**
 - `perror()` — prints a message based on `errno`
 - `strerror(errno)` — returns a string describing the error

In Linux and POSIX systems, `errno` is a global variable that stores the **error code** when a system call or library function fails. It helps you understand **why** an operation failed.

What Is `errno`?

- Defined in `<errno.h>`
- Set by system calls and some library functions when they return `-1` or `NULL`
- Not cleared automatically — only updated on error
- Can be printed using `perror()` or `strerror(errno)`

Common `errno` Values

Code	Value	Meaning
<code>EPERM</code>	1	Operation not permitted
<code>ENOENT</code>	2	No such file or directory
<code>ESRCH</code>	3	No such process
<code>EINTR</code>	4	Interrupted system call
<code>EIO</code>	5	I/O error
<code>ENXIO</code>	6	No such device or address
<code>E2BIG</code>	7	Argument list too long
<code>EACCES</code>	13	Permission denied
<code>EEXIST</code>	17	File exists
<code>ENOTDIR</code>	20	Not a directory
<code>EISDIR</code>	21	Is a directory
<code>EINVAL</code>	22	Invalid argument
<code>ENOSPC</code>	28	No space left on device

The `perror()` function in C is used to print a human-readable error message to `stderr` based on the current value of `errno`. It's a quick way to understand why a system call or library function failed.

Prototype (from `<stdio.h>`)

C ^

 Copy

```
void perror(const char *s);
```

- `s`: A custom message prefix (often the name of the function that failed)



Example Usage

C ^

Copy

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

int main() {
    if (access("nonexistent.txt", R_OK) == -1) {
        perror("access"); // Prints: access: No such file or directory
    }
    return 0;
}
```

- If `access()` fails, `errno` is set (e.g., `ENOENT`)
- `perror()` prints: `"access: "` followed by the error message from `strerror(errno)`



Notes

- Output goes to `stderr`, not `stdout`
- It's useful for debugging system call failures
- You can also use `strerror(errno)` if you want to format the message yourself

Permission Mode Constants

These constants are used with system calls like `chmod()`, `stat()`, and `fchmod()` to set or check file permissions.

♦ Owner (User) Permissions

Constant	Octal	Meaning
<code>S_IRUSR</code>	0400	Read permission for owner
<code>S_IWUSR</code>	0200	Write permission for owner
<code>S_IXUSR</code>	0100	Execute/search permission for owner


♦ Group Permissions

Constant	Octal	Meaning
<code>S_IRGRP</code>	0040	Read permission for group
<code>S_IWGRP</code>	0020	Write permission for group
<code>S_IXGRP</code>	0010	Execute/search permission for group

Constant	Octal	Meaning
<code>S_IROTH</code>	0004	Read permission for others
<code>S_IWOTH</code>	0002	Write permission for others
<code>S_IXOTH</code>	0001	Execute/search permission for others

Example: Setting Permissions

C ^

 Copy

```
chmod("myfile.txt", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```


This sets:

- Owner: read and write
- Group: read
- Others: read

Equivalent to `chmod 644 myfile.txt`.

Special Mode Bits

Constant	Octal	Meaning
<code>S_ISUID</code>	04000	Set user ID on execution
<code>S_ISGID</code>	02000	Set group ID on execution
<code>S_ISVTX</code>	01000	Sticky bit (restricts deletion)

Used for executable files and shared directories  `/tmp`.

in linux, what are the different file systems

Linux supports a wide range of file systems, each designed for different use cases such as performance, reliability, journaling, and compatibility. Popular ones include ext4, XFS, Btrfs, and FAT32.

Common Linux File System Types

Here's a breakdown of the most widely used file systems in Linux and what they're best known for:

ext4 (Fourth Extended File System)

- **Most common default** on many Linux distributions
- Supports large files (up to 16 TB) and volumes (up to 1 EB)
- Journaling for data integrity
- Backward compatible with ext3 and ext2

XFS

- High-performance journaling file system
- Excellent for large files and parallel I/O workloads
- Used in enterprise environments and Red Hat-based systems

Btrfs (B-tree File System)

- Modern copy-on-write file system
- Supports snapshots, checksums, compression, and RAID
- Still evolving; not always recommended for mission-critical systems

ext3 / ext2

- ext3: predecessor to ext4, with journaling
- ext2: older, no journaling — useful for flash storage due to low write overhead


FAT32 / exFAT

- Used for USB drives and cross-platform compatibility
- Limited features (no journaling, limited file size)
- exFAT supports larger files and is better for modern flash storage

NTFS

- Windows-native file system
- Supported via `ntfs-3g` driver in Linux
- Useful for dual-boot systems or external drives

ZFS (Zettabyte File System)

- Advanced features: snapshots, deduplication, compression
- High reliability and data integrity
- Not included by default due to licensing, but  available via third-party packages

ISO 9660 / UDF

- Used for optical media (CDs, DVDs)
- ISO 9660: older standard
- UDF: newer, supports larger files and better compatibility

Specialized or Niche File Systems

File System	Use Case
<code>tmpfs</code>	Temporary in-memory storage
<code>overlayfs</code>	Container layering (e.g., Docker)
<code>aufs</code>	Union file system (older Docker)
<code>F2FS</code>	Optimized for flash memory
<code>ReiserFS</code>	Legacy journaling file system

access permission of a symbolic link

In Linux, **symbolic links (symlinks)** have a unique behavior when it comes to access permissions:


Key Facts About Symbolic Link Permissions

- **Symlinks don't have traditional access permissions** like regular files or directories.
- The `access()` system call and most file operations **follow the symlink** to its target and check permissions on the **target file**, not the link itself.
- The symlink's own `st_mode` (from `lstat()`) will show `0777` permissions, but these are **ignored** by the kernel.

Example

Suppose you have:

Bash ^

 Copy

```
ln -s /etc/passwd mylink
```



Example

Suppose you have:

Bash ^

 Copy

```
ln -s /etc/passwd mylink
```

Then:

C ^

 Copy

```
access("mylink", R_OK); // Checks read access on /etc/passwd
```

Even if `mylink` appears to have `rxwxrwxrwx` permissions, the actual access depends on `/etc/passwd`.

Special Notes

- Use `lstat()` to inspect the symlink itself (not the target).
- Use `access()` or `stat()` to check the target's permissions.
- Symlink permissions are mostly relevant for **directory listing** or **removal**, not for controlling access.

unlink symbolic and unlink hard link

Unlinking a **symbolic link** and a **hard link** in Linux both use the `unlink()` system call or the `rm` command, but they behave differently due to how each type of link works.

Symbolic Link vs Hard Link

Feature	Symbolic Link (<code>symlink</code>)	Hard Link
Points to	Path of another file	Same inode as target file
Breaks if target is deleted	✓ Yes	✗ No
Shares inode	✗ No	✓ Yes
Can cross filesystems	✓ Yes	✗ No



Unlinking a Symbolic Link

- **Command:** `rm symlink_name` or `unlink symlink_name`
- **Effect:** Deletes the symlink itself, **not** the target file
- If the symlink is broken (target missing), `rm` still removes the symlink



Example

Bash ^

Copy

```
ln -s /etc/passwd mylink
rm mylink # Removes the symlink, /etc/passwd remains untouched
```



Unlinking a Hard Link

- **Command:** `rm hardlink_name` or `unlink hardlink_name`
- **Effect:** Removes one reference (directory entry) to the file
- File is deleted **only when all hard links are removed**



Example

Bash ^

Copy

```
ln original.txt hardlink.txt
rm hardlink.txt # original.txt still exists and file data is intact
```



Summary

- `unlink()` removes a **name** from the filesystem.
- For symlinks: it removes the link file.
- For hard links: it removes a reference to the inode; the file is deleted only when the last link is gone.

symbolic link is a special file, how the file content is written where, what

Great question! A **symbolic link (symlink)** is indeed a special type of file in Linux and other Unix-like systems. Let's break down how it works under the hood:

What Is a Symbolic Link?

A symbolic link is a file that contains a **path to another file or directory**. It acts like a shortcut — when you access the symlink, the system transparently redirects you to the target.

Where Is the Content Stored?

- The **content of a symlink is the path to its target**, stored as a string.
- This path is saved **in the data blocks of the symlink's inode**.
 - For **short paths** (typically under ~60 bytes), the path is stored **directly in the inode** (called "fast symlinks").
 - For **longer paths**, the path is stored in a separate data block on disk, and the inode points to that block.

What Does the Symlink Contain?

Let's say you run:


Bash ^

 Copy

```
ln -s /home/user/data.txt mylink
```

- `mylink` is a symlink file.
- Its content is literally the string: `/home/user/data.txt`
- You can see this with:

Bash ^

 Copy

```
cat mylink      # Follows the link and shows contents of data.txt
readlink mylink # Shows the actual path stored in the symlink
```



Example: Inspecting a Symlink

Bash ^

Copy

```
ls -l mylink
```

```
# Output: lrwxrwxrwx 1 user user 18 Oct 29 17:40 mylink -> /home/user/data.txt
```

- The `l` at the start shows it's a symlink.
- The `->` shows the path it points to.
- The size (`18`) is the length of the path string.



Summary

Feature	Symbolic Link
File type	Special file (<code>S_IFLNK</code>)
Content	Path to target (as a string)
Stored in	Inode (short) or data block (long)
Access behavior	Redirects to target file

hard link is a name, how a hard link is stored, where, what

You're absolutely right — a **hard link is just another name** for a file. But unlike symbolic links, hard links are deeply tied to the underlying filesystem structure. Let's break it down:

What Is a Hard Link?

A hard link is a **directory entry** that points directly to a file's **inode** — the data structure that stores metadata and disk block locations. Multiple hard links can point to the same inode, meaning they all refer to the **same file content**.



How and Where Is a Hard Link Stored?

◆ Inode Structure

- Every file has an inode containing:
 - File size
 - Timestamps
 - Permissions
 - Pointers to data blocks
 - A **link count** (number of hard links)

◆ Directory Entry

- A hard link is stored as a **directory entry** (filename → inode number)
- When you create a hard link:
 - A new name is added to the directory
 - It points to the **same inode** as the original file

Example

Bash ^

 Copy

```
echo "Hello" > original.txt
ln original.txt clone.txt
```

- `original.txt` and `clone.txt` both point to the same inode
 - Changes to one affect the other
 - Deleting one does **not** delete the file — only reduces the link count
-

Inspecting Hard Links

Bash ^

 Copy

```
ls -li original.txt clone.txt
```

- The `-li` flag shows the inode number
- Both files will have the **same inode number**
- `stat original.txt` shows `Links: 2`



Summary

Feature	Hard Link
Stored as	Directory entry pointing to inode
Shares inode	✓ Yes
Shares content	✓ Yes
Link count	Increases with each hard link
Breaks if target deleted	✗ No — file persists until last link is removed

structure type related to a directory entry

In Linux and POSIX systems, the structure type related to a **directory entry** is:




`struct dirent`

Defined in `<dirent.h>`, this structure represents a single entry in a directory — such as a file, subdirectory, or symbolic link.

Structure Definition (simplified)

```
struct dirent {
    ino_t      d_ino;      // Inode number
    off_t      d_off;      // Offset to the next dirent
    unsigned short d_reclen; // Length of this record
    unsigned char d_type;   // Type of file (e.g., DT_REG, DT_DIR)
    char        d_name[];  // Null-terminated filename
};
```


 Copy

Field Breakdown

Field	Description
<code>d_ino</code>	Inode number of the file
<code>d_off</code>	Offset to the next directory entry
<code>d_reclen</code>	Length of this directory entry structure
<code>d_type</code>	File type (e.g., <code>DT_REG</code> , <code>DT_DIR</code> , <code>DT_LNK</code>)
<code>d_name</code>	Filename (null-terminated string)

Example Usage

C ^

 Copy

```
#include <dirent.h>
#include <stdio.h>

int main() {
    DIR *dir = opendir(".");
    struct dirent *entry;

    while ((entry = readdir(dir)) != NULL) {
        printf("Name: %s, Type: %d, Inode: %lu\n",
            entry->d_name, entry->d_type, entry->d_ino);
    }

    closedir(dir);
    return 0;
}
```



Notes

- `d_type` may be `DT_UNKNOWN` on some filesystems (e.g., NFS), so use `stat()` or `lstat()` for reliable type info.
- `readdir()` returns a pointer to `struct dirent` for each entry in the directory.

blksize for FILE internal buffer for buffering

In C, when you use the standard I/O library (`stdio.h`) and open a file with `fopen()` , the system uses an internal buffer for efficient I/O. The size of this buffer is **not directly tied to** `st_blksize` from `struct stat` , but they are conceptually related.



FILE Internal Buffer Size

- The `FILE` object (used by `fopen` , `fread` , `fwrite` , etc.) maintains its own **user-space buffer**.
- The default buffer size is typically **4 KB or 8 KB**, depending on the system and libc implementation.
- You can **manually set** the buffer size using `setvbuf()` .



Relationship to `st_blksize`

- `st_blksize` (from `stat()`) is the **optimal block size** for I/O on that file or filesystem.
- It's a **hint** — not a requirement — for how large your buffer should be.
- Standard I/O implementations **may use** `st_blksize` to choose a good default buffer size, but this is not guaranteed.



Example: Matching Buffer Size to `st_blksize`

C ^

Copy

```
#include <stdio.h>
#include <sys/stat.h>

int main() {
    struct stat sb;
    FILE *fp = fopen("myfile.txt", "r");

    if (fp && fstat(fileno(fp), &sb) == 0) {
        char *buf = malloc(sb.st_blksize);
        setvbuf(fp, buf, _IOFBF, sb.st_blksize); // Full buffering with custom size
        // Now use fp with optimal buffering
    }

    // Don't forget to free(buf) and fclose(fp) later
    return 0;
}
```

difference between readlink and realpath

The difference between `readlink()` and `realpath()` in C (and in Linux generally) lies in how they handle symbolic links and path resolution:



`readlink()`

- **Purpose:** Reads the **target path** stored in a symbolic link.
- **Does NOT resolve** the full path — just returns the raw string inside the symlink.
- **Does NOT follow** multiple symlinks or canonicalize the path.



Example

```
C ^ Copy  
  
char buf[1024];  
ssize_t len = readlink("mysymlink", buf, sizeof(buf) - 1);  
if (len != -1) {  
    buf[len] = '\0';  
    printf("Symlink points to: %s\n", buf);  
}
```

- If `mysymlink` points to `/etc/passwd`, it returns that string.
- If the symlink is broken, it still returns the stored path.



realpath()

- **Purpose:** Resolves a path to its **absolute, canonical form**.
- Follows all symlinks, resolves `.` and `..`, and returns the actual location.
- Returns `NULL` if the path doesn't exist or is broken.



Example

C ^

Copy

```
char resolved[PATH_MAX];  
if (realpath("mysymlink", resolved)) {  
    printf("Resolved path: %s\n", resolved);  
}
```

- If `mysymlink` points to `/etc/passwd`, it returns `/etc/passwd`
- If the symlink is broken, it fails



Summary

Function	Returns	Follows Symlinks	Requires Valid Target
<code>readlink()</code>	Raw path stored in symlink	✗ No	✗ No
<code>realpath()</code>	Absolute resolved path	✓ Yes	✓ Yes



readlink()

Prototype

C ^

Copy

```
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

◆ Parameters:

- `pathname` : Path to the symbolic link
- `buf` : Buffer to store the link target
- `bufsiz` : Size of the buffer

◆ Returns:

- Number of bytes placed in `buf` (not null-terminated)
- `-1` on error (e.g., if `pathname` is not a symlink)



`realpath()` Prototype

C ^

Copy

```
char *realpath(const char *pathname, char *resolved_path);
```

◆ Parameters:

- `pathname` : Path to resolve (can be relative or symlink)
- `resolved_path` : Buffer to store the absolute path (or `NULL` to allocate dynamically)

◆ Returns:

- Pointer to `resolved_path` or dynamically allocated string
- `NULL` on error (e.g., path doesn't exist)

what if unlink original target


If you **unlink (delete) the original target file** of a symbolic link or hard link, the behavior depends on the type of link:

Symbolic Link

- A **symbolic link** stores the **path** to the target file.
- If you delete the original target file:
 - The symlink becomes **broken** or **dangling**.
 - It still exists, but points to a non-existent location.
 - Accessing it (e.g., `cat symlink`) will result in an error like `No such file or directory`.

Example

Bash ^

 Copy

```
ln -s original.txt symlink.txt
rm original.txt
cat symlink.txt # Error: No such file or directory
```





Hard Link

- A **hard link** is just another name for the same inode.
- If you delete the original filename:
 - The file **still exists** as long as at least one hard link remains.
 - The data is preserved until the last hard link is removed.



Example

Bash ^

Copy

```
ln original.txt hardlink.txt
rm original.txt
cat hardlink.txt # Still works - file content is intact
```



Summary

Action	Symbolic Link	Hard Link
Delete original target	Symlink breaks	File remains via hard link
Access after deletion	Fails	Succeeds
Link count affected	No	Yes (decremented)

what if original target is provided to system call unlink

If the **original target file** is passed to the `unlink()` system call, the system will attempt to **remove that specific filename (directory entry)** from the filesystem. Here's what happens depending on the context:



`unlink("original.txt")` — What It Does

- **Removes the directory entry** named `original.txt`
- **Decrements the link count** of the file's inode
- If the link count reaches **zero** and no process has the file open:
 - The file's **data blocks and inode are freed**
 - The file is **fully deleted**



Impact on Links

- ◆ **If `original.txt` is the only hard link:**

- The file is deleted from disk.

- ◆ **If there are other hard links:**

- The file remains accessible via those links.
- Only the name `original.txt` is removed.

- ◆ **If symbolic links point to `original.txt`:**

- They become **dangling symlinks** (broken).
- The symlink itself is unaffected — it still stores the path `original.txt`.



Summary

Scenario	Result
<code>unlink("original.txt")</code>	Removes that name from the directory
Other hard links exist	File remains via those links
No other links and not open	File is deleted
Symlinks point to it	Symlinks break, but are not deleted