System Programming - Linux
Week 5 - Input / Output Operations (Files)

Lecturer: Dr. Aggrey Obbo (PhD)

April 18, 2025

Contents

1 Introduction

A key element of Linux system administration and programming is the Linux file system. These lecture notes focus on the basic ideas of files and how Linux structures and controls data. The notes explore the hierarchical organization of files, various file types, and the important function of inodes. In addition, it examines how the Linux kernel manages file operations and the different system calls to facilitate interaction with the file system. An understanding of these concepts is vital for anyone intending to engage with Linux systems.

Objectives

Upon completion of this lecture, students will be able to:

- Understand the Linux-specific implementation of the file concept.
- Identify and differentiate between various file types in the Linux environment.

- Describe the structure of the Linux file system hierarchy.
- Explain the role and structure of inodes in storing file metadata within Linux.
- Detail the kernel mechanisms and VFS layer supporting file operations in Linux.
- Utilize key file I/O system calls in Linux for file manipulation.
- Retrieve and interpret file status information using the stat family of system calls.
- Implement file and record locking mechanisms using lockf and fcntl in Linux.
- Understand and manipulate file permissions and ownership using relevant system calls.
- Create and differentiate between hard and soft links using Linux system calls.

2 The Linux File System

A File system in Linux is a sequence of bytes, treated uniformly regardless of content. And a file system to any operating system is like a tool that manages, stores, and retrieves data for a computing system. Linux uses a hierarchical, tree-like structure to organize all files and directories on a Linux system [1][2]. On top of the "tree" is the root directory that is represented by a single forward slash "/". The structure then branches off into various subdirectories as in Figure 1 below.

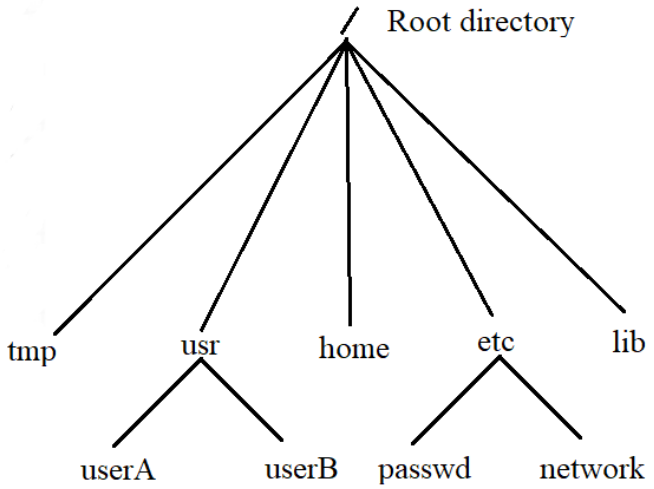

Figure 1: Linux file structure

2.1 Definition of Key Terms

Below is a definition of some of the key words used in Linux file systems;

- Paths: In Linux, paths are used to find files and directories.
  An absolute path begins at the root directory (/) and gives the full location. For instance, /home/user/documents/report.txt directly refers to the report.txt file.
  A relative path is determined by the present working directory. If you are currently in /home/user, the relative path to that same file would be documents/report.txt.
  Special paths consist of . (representing the current directory) and .. (indicating the parent directory). For example, if you're located in /home/user/documents, using ../ would take you to /home/user.
- Mounting: The Linux mount command is utilized to connect/mount a file system, such as on a USB flash drive to a designated directory for instance, /usr/local within the file system structure. This is meant to allow access of files and directories from the mounted device to both the user and the system.

- Inode: An inode or index node serves as a unique identifier for a specific piece of metadata on a given filesystem. The index node is the representation of any file or directory based on the parameters that can include: size, permission, ownership, and its location of the file and directory.
- Versioning: Versioning file systems enable the storage of previously saved versions of a file. For example copies of a file are stored based on previous commits to the disk in an hourly manner to create a backup.
- Journaling: Journaling file systems keep a log called the journal, that keeps track of the changes made to a file but not yet permanently committed to the disk so that in case of a system failure the lost changes can be brought back.

3 Linux File Structure

Linux uses a robust permission system to control who can read, write, and execute files and directories. The root directory(/) is the top-level directory from which all other directories and files originate. It's the base of the entire file system. Directories are special files that contain other files and directories, forming the branches of the file system tree. While the files contain data, programs, or other information. Examples of other directories include the /etc (Etcetera) directory which contains system-wide configuration files and directories for various applications and services and /home which contains subdirectories for each user on the system.

3.1 Layers of the Linux File System

The Linux file system comprises three layers: Logical, virtual and physical layers. This hierarchical structure as shown in Figure 2 below allows for flexibility and accommodates a variety of storage technologies.

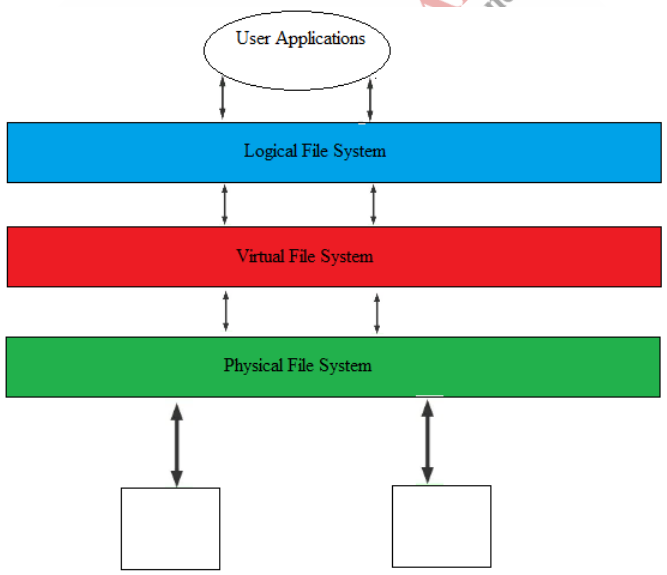

Figure 2: Linux file system layers

- Logical File System: The Logical File System acts as the interface between the user applications and the file system itself. It facilitates essential operations such as opening, reading, and closing files. Essentially, it serves as the user-friendly front-end, ensuring that applications can interact with the file system in a way that aligns with user expectations.
- Virtual File System: Beneath this layer lies the Virtual File System (VFS), which serves as an abstraction layer to facilitate uniform access to the various file system types. The Virtual File System (VFS) is a crucial layer that enables the concurrent operation of multiple instances of physical file systems. It provides a standardized interface, allowing different file systems to coexist and operate simultaneously. This layer abstracts the underlying complexities, ensuring compatibility and cohesion between various file system implementations.

- Physical File System: At the bottom is the Physical File System, responsible for managing the actual storage devices and overseeing the physical arrangement of data blocks. It takes care of fundamental aspects of data storage and retrieval, interfacing directly with the hardware components. This layer guarantees the effective distribution and use of physical storage resources, enhancing the file system's overall performance and dependability.

Characteristics of a File System

Characteristics of a File System Space Management: how the data is stored on a storage device. Pertaining to the memory blocks and fragmentation practices applied in it. Filename: a file system may have certain restrictions to file names such as the name length, the use of special characters, and case sensitive-ness. Directory: the directories/folders may store files in a linear or hierarchical manner while maintaining an index table of all the files contained in that directory or subdirectory. Metadata: for each file stored, the file system stores various information about that file's existence such as its data length, its access permissions, device type, modified date-time, and other attributes. This is called metadata. Utilities: file systems provide features for initializing, deleting, renaming, moving, copying, backup, recovery, and control access of files and folders. Design: due to their implementations, file systems have limitations on the amount of data they can store.

- Logical File System: The Logical File System functions as the bridge between user applications and the underlying file system. It enables key operations like opening, reading, and closing files. In essence, it acts as an approachable front-end, making certain that applications can engage with the file system in a manner that meets user expectations.

- Virtual File System: The Virtual File System (VFS) serves as an essential layer that facilitates the simultaneous functioning of various physical file system instances. It offers a uniform interface that allows multiple file systems to exist and operate at the same time. This layer simplifies the underlying intricacies, maintaining compatibility and unity among different file system implementations.

- Physical File System: The Physical File System manages the actual storage and organization of physical memory blocks on the disk. It addresses the fundamental aspects of data storage and retrieval while communicating directly with hardware components. This layer guarantees the effective assignment and use of physical storage resources, which enhances the overall performance and dependability of the file system.

3.2 File System Types

Linux has a variety of file systems, each possessing distinct advantages. Each may have its features, performance characteristics, and use cases. Some common Linux file system types include:

- Ext2/Ext3/ext4: The ext4 file system structures data in blocks collected into block groups. It employs extents to manage large files effectively and uses delayed allocation to minimize fragmentation. The inclusion of journaling with checksums improves reliability. Ext4 accommodates large file systems and files, providing better performance along with features such as nanosecond timestamps and adaptable allocation methods. It is the typical default choice.

- XFS: XFS is a robust 64-bit journaling file system that was originally created by SGI for their IRIX operating system and later adapted for Linux. It functions like a meticulously organized librarian. Like a library, XFS breaks it down into smaller, easier-to-handle sections known as allocation groups (AGs) rather than having a single, extensive catalog. This structure enables the computer to find and save information significantly faster, and particularly when working with numerous files or large applications. Its design provides outstanding scalability for I/O threads, file system bandwidth, and the dimensions of both files and the file system itself, particularly when distributed across multiple storage devices.

  The key features of XFS include metadata journaling, ensuring data consistency after crashes, and support for very large file systems. It utilizes extent-based allocation to reduce fragmentation and optimize performance for large files.

- Btrfs, pronounced "Butter FS," is a contemporary file system for Linux that strives to be versatile and offer advanced functionalities. One of its main advantages is its Copy-on-Write (CoW) feature. When a file is modified, instead of altering the original directly, Btrfs creates a duplicate, makes the necessary changes, and

then updates the old version. This approach helps safeguard against data corruption and facilitates easy snapshots, allowing the system to effortlessly revert to a prior state if an issue arises.

Btrfs also supports features like built-in RAID (combining multiple drives for better performance or redundancy), volume management (resizing partitions without unmounting), and checksumming (verifying data integrity to detect and sometimes repair errors). While powerful, it's still evolving, and its performance and stability can vary depending on the specific use case. However, for many Linux users, Btrfs offers a compelling alternative with its focus on data protection and advanced management capabilities. Btrfs is a modern file system offering advanced features like snapshots, compression, and volume management. It also offers sophisticated features such as snapshots and data compression.

Btrfs provides features such as integrated RAID (which merges multiple drives for enhanced performance or redundancy), volume management (allowing for partition resizing without needing to unmount), and checksumming (ensuring data integrity to identify and sometimes fix errors). Although it is robust, it is still being developed, and its performance and reliability can differ based on the specific application. Nevertheless, for numerous Linux users, Btrfs presents an attractive alternative due to its emphasis on data safeguarding and advanced management features.

- ZFS is a robust and sophisticated file system that was originally created by Sun Micro-systems. Unlike conventional file systems, ZFS merges the functionalities of a file system and a volume manager. This enables it to handle storage devices (such as hard drives) and arrange its files simultaneously, providing excellent versatility.

  One of the main characteristics of ZFS is its strong emphasis on data integrity. It utilizes checksums to continuously check the health of data and can automatically correct errors if redundancy is configured. ZFS also uses Copy-on-Write (CoW), which helps to avoid data corruption and facilitates easy snapshots.

  Although ZFS is highly reliable and provides features like integrated RAID-like capabilities (referred to as ZFS RAID or RAID-Z), compression, and deduplication (which minimizes storage space by finding duplicate files), it can be more demanding in terms of resources compared to simpler file systems and may necessitate a deeper technical understanding to manage effectively on Linux. Nevertheless, for those who prioritize data security and sophisticated storage management, ZFS stands out as an exceptional choice.

4 File Permissions

Linux file permissions control access and modification of files and directories [2]. They are represented by a string of characters when one uses the ls -l command.

-rw-r–r– 1 user group 1024 Apr 10 23:00 abcfile.txt drw-r–r-x 1 user group 1024 Apr 10 23:00 abcfile.txt

The first character (-) indicates the file type (a regular file in this case; d would signify a directory). The next nine characters represent the permissions for three categories: owner, group, and others.

Owner (user): The first three characters (rw-) define the permissions for the user who owns the file.

r (read): The owner can open and read the contents of abcfile.txt. w (write): The owner can modify the contents of abcfile.txt. x (execute): The owner can execute abcfile.txt In this example, the owner has read and write permissions but not execute permission. Group: The next three characters (r–) define the permissions for the group that the file belongs to. In this case, users belonging to the group can only read myfile.txt. They cannot modify or execute it. Others: The last three characters (r–) define the permissions for all other users on the system who are neither the owner nor members of the file's group. Here, everyone else can only read myfile.txt.

4.1 Adding Users and Groups

To add users the useradd or adduser command is used. This can only be done with a root privilege. Although these commands both leed to the same outcome, useradd provides more low-level control, while adduser is more user-friendly with prompts. The username has to be provide where other options can be included: -m for setting a home directory, setting a password, -G for group memberships.

The purpose of groups is to collect users for easier permission management. The groups are created with the groupadd command. The group name is specified, and optional parameters can include the group ID (GID) with -g flag. Users can are added to existing groups using the usermod -aG groupname username command.

Example

sudo groupadd powerusers
sudo usermod -aG newgroup user1

The above code creates a group called “powerusers” and line 2 above adds user1 to the group.

4.2 Granting Permissions

The three categories of access rights: Read, Write, and Execute Permissions. The permissions can be changed using the chmod command. For example, to give the owner execute permission, you could either use
chmod u+x abcfile.txt.
or set specific permissions using numerical mode (where read=4, write=2, execute=1),using
chmod 755 abcfile.txt
This would give the owner read, write, and execute permissions (4+2+1=7), and the group and others read and execute permissions (4+1=5).

4.3 Other attributes stored in the inode

User ID (UID) and Group ID (GID): Identifying users and groups.
chown(): Changes ownership
lchown(): Changes the ownership of the symbolic link itself, not the target.
fchown(): Changes ownership of an open file.
umask: Setting the default file creation mask.
chmod(): Modifying permissions using symbolic and numeric modes.
fchmod(): Modifying permissions of an open file

5 Linux File Types

In Linux, everything is treated as a file, including regular files, directories, device files, symbolic links, named pipes, and sockets. Linux employs a diverse range of file types, each serving a specific purpose within the operating system. The most common type are

- Regular files: Regular files hold data such as text documents, images, executables, and libraries.
- Directories. Directories store data, and act as containers for other files and sub-directories, forming the hierarchical structure of the Linux file system.
- Symbolic links (or soft links): These are special files that act as pointers or containing a pathname to another file or directory. If the original file is moved or deleted, the symbolic link will become broken. They are created using the ln -s command. They can cross file system boundaries.

  To create a soft link, symlink() system call is used and unlink() system call removes the symbolic link file itself.
- Pipes: Named pipes (FIFOs) are special files that provide a unidirectional communication channel between processes. One process can write data to the FIFO, and another process can read that data. They are created using the mkfifo command.
- Sockets: Sockets are files in inter-process communication (IPC) across networks or on local machines. They facilitate communication using network protocols like TCP/IP or Unix domain sockets.
- Hard links: Hard links are direct references to the inode (a data structure containing metadata about a file) of another file. They essentially make a file accessible through multiple names in the file system. Unlike symbolic links, hard links remain valid even if the original filename is changed or deleted, as long as at least one hard link exists. They cannot, however, span across different file systems. Hard links are created using the ln command.

  All hard links share the same inode number, permissions, ownership, and data. and changes through one link are reflected in all. They cannot, however, cross file system boundaries.

  The link() system call creates a new hard link. While unlink() system call removes a hard link. The file is deleted when the link count reaches zero.

- Block devices: Block devices handle data in blocks and allow for random access to fixed-size blocks of data. They include devices like SSDs and USB drives. They are also typically found in the /dev directory.
- Character devices: Character devices, on the other hand, normally reside in the /dev directory and represent devices that handle data character by character. Examples of these devices include serial ports, keyboards, and mice. Unlike character devices, these don't support random access.

6 File Descriptors

File descriptors in Linux are non-negative integers that serve as unique identifiers for open files and other I/O resources in a process. When a process opens a file, socket, pipe, or device, the Linux kernel assigns it a file descriptor, which allows it to interact with the resource.

Every process in Linux has its own file descriptor table. This table is a per-process data structure in the kernel that maps file descriptors to open files. When a process makes a system call, such as read() or write(), it sends a file descriptor, which the kernel uses as an index into the process's file descriptor database to get the related file information.

Each process's file descriptor table contains pointers to entries in the system-wide open file table, which contains information about all open files on the system, including their current position, access mode, and flags. This enables multiple processes to potentially share access to the same file while maintaining their own independent file descriptors and file positions. The first three file descriptors are standard and are inherited by child processes. They include:

- 0 (STDIN): Standard input, usually connected to the keyboard.
- 1 (STDOUT): Standard output, typically connected to the terminal.
- 2 (STDERR): Standard error, also usually connected to the terminal for displaying error messages.

When a process calls fork(), the child process receives a copy of the parent's file descriptor table. This means that both the parent and child processes will initially have the same file descriptors referring to the same open files. Subsequently, they can independently operate on these files through their respective file descriptors. On the other hand the exec() family of functions typically closes file descriptors that have the close-on-exec flag set before executing a new program.

7 Kernel Support for Files in Linux

The Linux kernel provides a comprehensive and flexible architecture for file management, allowing it to support a wide variety of file systems. The key components for file system support include:

1. Virtual File System (VFS) Layer: The VFS is a crucial abstraction layer within the Linux kernel. It provides a uniform interface for applications to access files, regardless of the underlying file system's type. This means that a program can read a file using the same system calls, whether the file resides on an ext4, XFS, or NFS file system.

   VFS defines a set of operations that file systems must implement. These operations are grouped into three main categories:

   - Inode Operations: Handle file-specific metadata such as file permissions and size.
   - File Operations: Manages reading, writing, and manipulating file data.
   - Superblock Operations: Manage file system-level information (e.g., mounting and unmounting).

2. Dentry Cache: To speed up file access, the kernel maintains a cache of directory entries (dentries). A dentry represents a component of a path for example a directory or file name.

   The dentry cache stores recently accessed directory entries, significantly reducing the time it takes to resolve file paths. This is because the kernel can often find the necessary information in the cache instead of having to traverse the file system structure on disk.

3. Page Cache (Buffer Cache): Linux employs a unified page cache, which integrates file data caching with the virtual memory system. When a process reads data from a file, the kernel copies the data into pages in memory. Subsequent reads to the same data can be served directly from the page cache, avoiding disk I/O.

Reducing disk I/O via caching file data, the page cache drastically reduces the number of slow disk operations, leading to improved system performance.

4. Write-back policies: The kernel uses write-back policies to manage when modified data in the page cache is written back to disk. These policies balance performance (keeping data in cache as long as possible) with data integrity (ensuring that changes are eventually saved).

5. File System Drivers: The Linux kernel supports a modular architecture for file systems. Specific file system types (like ext4, XFS, Btrfs, etc.) are implemented as kernel modules.

   These modules, which commonly have a.ko extension (for example, ext4.ko), contain the file system-specific code that interfaces with the VFS interface. When a file system is mounted, the kernel loads the relevant module, allowing it to access the file system's files.

8 System Calls for File I/O Operations

In Linux, system calls are essential for performing file I/O operations [3]. They provide the interface between user-space processes and the kernel, enabling programs to interact with the file system.

Key system calls include:

- open(): Used to open or create a file, returning a file descriptor.
- read(): Reads data from a file descriptor into a buffer.
- write(): Writes data from a buffer to a file descriptor.
- close(): Closes a file descriptor, releasing associated resources.
- Duplicating file descriptors.
  dup(): Finds the next available file descriptor.
  dup2(): Duplicates to a specific file descriptor (often used for redirection).
- lseek(): Repositions the file offset for subsequent read/write operations.
- unlink(): Deletes a file.

These calls are fundamental to how programs handle files, allowing for operations like reading configuration files, writing log data, or processing user input. They operate at a low level, directly interacting with the kernel to manage file access, permissions, and data transfer.

9 File Status Information

The stat family of system calls in Linux provides crucial file status information. These calls, include: stat(), fstat(), and lstat(). They retrieve details about files and directories. For example;

- stat(pathname, buf) - fetches status information for the file specified by pathname and stores it in the buf structure. stat() is used for accessing members of the struct stat (e.g., st_mode, st_size, st_uid, st_mtime).
- lstat(pathname, buf) behaves like stat(), but it does not follow symbolic links, instead returning information about the link itself. Important for working with symbolic links.
- fstat(fd, buf) does the same for an open file referred to by the file descriptor fd. Useful when you already have an open file.

The retrieved information is essential for various system operations and understanding file attributes. The stat command-line utility leverages system calls to present this information in a user-friendly format.

10 Concurrency Issues

Race conditions arise when multiple processes access the same file and try to read or write to a shared file simultaneously without proper coordination. The lack of synchronization can lead to unpredictable and often incorrect outcomes. One process might overwrite another's changes, or a read operation might catch the file in an inconsistent state, resulting in corrupted data.

To prevent these challenge, Linux provides various synchronization mechanisms. These include:

- File locking allows only one process to hold exclusive access to a file or a portion of it.
- Semaphores are synchronization mechanisms used to control access to shared resources in multi-threaded or multi-process environments. They are employed to coordinate file access and ensure data integrity.

The need for locking is to serialize access to crucial parts of files. Linux relies extensively on advisory locking, which means that processes must cooperate and actively acquire locks using system calls such as lockf() or fcntl() to avoid conflicts. lockf(fd, cmd, len) provides a straightforward interface for whole-file advisory locks, including commands such as F_LOCK (blocking lock), F_TLOCK (non-blocking), F_ULOCK (unlock), and F_TEST (test lock). However, it only supports whole-file locking.

fcntl(fd, cmd, flock) increases power. It employs the struct flock to provide the lock type (l_type: read, write, unlock), starting position (l_start), length (l_len, 0 for the entire file), and l_whence. The commands are F_GETLK (check if a lock exists), F_SETLK (non-blocking lock attempt), and F_SETLKW (blocking lock attempt). fcntl() locks specified byte ranges within a file, providing more precise control.

List of References

1. Fox, R. (2021). Linux with operating system concepts. Chapman and Hall/CRC.
2. Calcatinge, A., & Balog, J. (2021). Mastering Linux administration: a comprehensive guide to installing, configuring, and maintaining Linux systems in the modern data center. Packt Publishing Ltd.
3. Ward, B. (2021). How Linux works: What every superuser should know. no starch press.