Understanding Unix File Permissions and Extended Access Control Lists

Juboraj Ahmed

December 2025

1 Introduction

When you work on a shared computer system, you quickly realize that not everyone should have access to every file. The Unix operating system solved this problem decades ago with a simple permission model: owner, group, and others—each getting read, write, and execute rights. It worked beautifully for small systems with a handful of users. But as systems grew, with research labs, corporate servers, and cloud environments hosting hundreds of users collaborating on projects, this simple model began to show its cracks.

This assignment explores those limitations and introduces Extended Access Control Lists (ACLs) as a more flexible solution. Through this research, we'll understand not just *how* these systems work, but *why* they matter for modern computing.

2 Section 1: Understanding the Bottlenecks of Standard Unix Permissions

2.1 The Three-Tier Permission Model

Traditional Unix permissions are elegantly simple. Every file has an owner (user), an associated group, and a permission set for everyone else. Each entity gets three permission bits: read (r), write (w), and execute (x). You can see this in action with a command like `ls -l`:

```
-rw-r--r-- 1 alice research 4096 Jan 8 14:23 project_report.txt
```

Here, Alice owns the file. Members of the "research" group can read it. Everyone else can only read it too. The beauty lies in its simplicity—nine bits of information (rwxrwxrwx) tell you exactly who can do what.

But this simplicity comes with a cost.

2.2 Why This Model Breaks in Multi-User Environments

Consider a realistic scenario: A university research department runs a shared server. Professor Sarah needs to store a dataset that should be:

- Readable and writable by her graduate assistant, Tom
- Readable (but not writable) by three undergraduate students
- Readable by Dr. James from a collaborating institution
- Hidden from everyone else on the system

With standard Unix permissions, you're stuck. You could:

1. **Add everyone to a single group** and give them all read-write access (but then the undergraduates can modify the data—a disaster)
2. **Create multiple groups** (but you can only attach one group to the file, so Dr. James still wouldn't fit)
3. **Duplicate the file** into different directories with different ownership (wasteful and hard to maintain)
4. **Use "other" permissions** and hope no one unauthorized logs in (a security nightmare)

None of these solutions work cleanly. This is the core architectural limitation: **a file can belong to exactly one group, and group permissions are one-size-fits-all.**

2.3 Real-World Failure Scenarios

2.3.1 Scenario 1: The Project Collaboration Problem

A software development team needs to share source code. The project should be writable by developers in the "engineering" group but readable (not writable) by the "marketing" group for documentation purposes. Standard permissions force an uncomfortable choice: either marketing can't see the code, or they can accidentally delete it.

2.3.2 Scenario 2: The System Administrator's Dilemma

A log file `/var/log/app.log` is written by the application user "appuser" but needs to be readable by the "sysadmin" group *and* the "audit" group for monitoring. Since the file can only have one owning group, at least one team loses clean access, requiring workarounds like world-readable permissions (dangerous) or file duplication (wasteful).

2.3.3 Scenario 3: The External Collaborator Issue

A bioinformatics lab shares research data with a visiting scientist. Without adding them to an internal group (which could give them unintended access to other shared resources), there's no way to grant them specific file permissions. They either get nothing or too much.

2.4 The Administrative Overhead

As a system gets larger, administrators are forced to create hundreds of Unix Groups rather than twelve Logical Groups to allow many different permission combinations to be accommodated. This creates significant problems for system administration: as the number of groups has increased, there are more groups that can contain expired members, have misleading permissions and thus require an enormous amount of time to complete a full security review.

The fundamental issue is that **the permission model's granularity is fixed at the group level, not the user level.** If you need different permissions for multiple individual users, you're already outside the design scope of standard Unix permissions.

3 Section 2: The Mechanism of Extended Access Control Lists

3.1 What Are Extended ACLs?

Extended Access Control Lists (ACLs) don't replace standard Unix permissions—they extend them. Think of ACLs as a more flexible layer built on top of the basic model, allowing you to define permissions for specific users and groups without relying solely on group membership.

An ACL is essentially a list of rules, each stating: "User X can do Y to this file" or "Group A can do Z to this file." The operating system checks this list when someone tries to access a file, offering granularity the three-tier model simply cannot provide.

3.2 How ACL Data Integrates with the File System

To understand how ACLs fit into Unix architecture, we need to look at the file system's data structure—specifically, the **inode**.

Every file in Unix has an inode, a special data structure that stores metadata about the file: its size, ownership, timestamps, and of course, the traditional permission bits. On modern file systems like ext4, XFS, and Btrfs, the inode also contains a field that points to extended attributes.

ACLs are stored as extended attributes within the inode metadata. Here's how it works:

- **Traditional approach (UFS/older systems):** The inode contains a field called `i_shadow` that points to a separate "shadow inode." This shadow inode then stores the full ACL list. Multiple files can share the same shadow inode if they have identical ACLs, saving disk space.
- **Modern approach (ext4, XFS, Btrfs):** The inode itself is larger and has a field named `i_file_acl` (originally designed specifically for ACLs) that points to an extended attribute block. If the block size is large enough, ACL data fits directly in the inode; otherwise, it's stored in separate extended attribute blocks on disk.

This integration is important: **because ACL data lives alongside file metadata in the inode, permission checks happen at the kernel level without requiring extra file reads.** The operating system can evaluate both standard permissions and ACL entries efficiently when a process tries to access a file.

3.3 The Structure of a POSIX ACL Entry

A POSIX ACL entry has three essential components:

1. **Entry Type (Tag):** This identifies the category of the rule:
   - ACL_USER_OBJ – The file owner (maps to traditional "owner" bits)
   - ACL_USER – A specific named user (e.g., alice, bob)
   - ACL_GROUP_OBJ – The file's owning group (maps to traditional "group" bits)
   - ACL_GROUP – A specific named group (e.g., developers, analysts)
   - ACL_MASK – An upper-bound permission limit for all non-owner ACL entries
   - ACL_OTHER – Everyone else (maps to traditional "other" bits)
2. **Qualifier (Optional):** For named users and groups, this specifies *which* user or group:
   - For ACL_USER, this is a UID (user ID)
   - For ACL_GROUP, this is a GID (group ID)
   - The traditional entries (ACL_USER_OBJ, ACL_OTHER, etc.) don't have qualifiers
3. **Permission Set:** Three bits representing read (r), write (w), and execute (x), just like traditional Unix permissions

Example ACL entry: `user:alice:rw-` means "User alice can read and write, but not execute."

When you run the command `setfacl project_report.txt`, you'll see output like this:

```
file: project_report.txt
owner: alice
group: research
user::rw-
user:bob:r--
user:tom:rw-
group::r--
group:marketing:r--
mask::rw-
other::---
```

This tells us:

- Alice (the owner) can read and write
- Bob can only read
- Tom can read and write
- The research group can read
- The marketing group can read
- The mask limits all non-owner entries to read and write (not execute)
- Others have no access

3.4 Comparing Traditional chmod and Modern setfacl/getfacl

The difference between managing standard permissions and ACLs reflects their underlying architecture.

Using chmod (Standard Permissions):

The `chmod` command modifies the nine permission bits in the inode directly. It understands symbolic notation like `u+rwx` (owner plus read, write, execute) or numeric notation like 755 (owner: 7, group: 5, others: 5). Because it only works with the fixed three tiers, its syntax is inherently limited:

```
chmod 750 project_report.txt
```

- **Owner:** read, write, execute (7)
- **Group:** read, execute (5)
- **Others:** nothing (0)

Every user either gets group permissions or falls into the "other" category. There's no way to say "only Bob gets read access" while others in the group get more.

Using setfacl and getfacl (ACLs):

These commands work with the extended attribute blocks, allowing precise per-user and per-group rules. The syntax is more explicit but also more powerful:

- **Grant user "alice" read and write access:**
  `setfacl -m u:alice:rw project_report.txt`
- **Grant group "marketing" read-only access:**
  `setfacl -m g:marketing:r project_report.txt`
- **View the full ACL:** `getfacl project_report.txt`
- **Remove Alice's permissions:** `setfacl -x u:alice project_report.txt`

When you run `chmod`, it updates both the traditional bits *and* the ACL mask, keeping the two systems in sync. When you use `setfacl`, you're directly modifying the extended attribute data.

Aspect	chmod	setfacl/getfacl
Scope	Modifies three permission tiers	Adds/removes individual ACL entries
User-level granularity	None (only group level)	Yes (can target specific users)
Number of groups	One owning group	Multiple named groups possible
Syntax	Symbolic (u+rwx) or numeric (755)	Explicit (u:username:rwx)
File size impact	Minimal (9 bits)	Larger (extended attributes)

Table 1: Comparison of chmod vs setfacl/getfacl

4 Section 3: Why Extended ACLs Are Necessary

4.1 Closing the Gap Between Simple and Complex

The traditional Unix permission model was designed for simplicity. In the 1970s, when Unix systems had small user bases and relatively simple organizational structures, three permission categories were sufficient. The elegance of rwxrwxrwx meant anyone could understand file permissions at a glance.

But the real world is messy. Organizations have complex access requirements that don't fit neatly into "owner, group, and others." A single file might legitimately need different permissions for:

- The person who created it (typically full permissions)
- A few specific colleagues collaborating on a project
- Several teams or departments for reading or auditing
- External partners or contractors (limited access)

Extended ACLs bridge this gap. They allow systems to express real-world complexity while remaining compatible with traditional Unix thinking.

4.2 Practical Benefits

- **Elimination of Group Proliferation:** Instead of creating dozens of special-purpose groups to accommodate different permission combinations, administrators can use named ACL entries targeting specific users or groups. The system remains manageable even with hundreds of users.
- **Fine-Grained User-Level Control:** ACLs shift the granularity from the group level to the user level. You can grant Alice specific permissions without affecting other members of her group. This is impossible with standard permissions.
- **Reduced Need for File Duplication:** In scenarios where the same data needs different access levels for different users, ACLs allow a single file to serve multiple purposes without copying it to different directories with different ownerships and groups. This saves storage and reduces inconsistencies.

- **Security Through Precision:** The principle of least privilege—granting only the minimum necessary permissions—is easier to implement. You're not forced to choose between "too little" (no access) and "too much" (full group access). You can specify exactly what each user or group needs.
- **Audit Trail Clarity:** ACLs make permission policies explicit and auditable. When you run `getfacl`, you see exactly who has access and at what level, making security reviews more thorough.

4.3 Integration with Unix Philosophy

A crucial strength of extended ACLs is that they don't abandon the Unix philosophy; they extend it. The basic three-tier model still works for simple cases—files with straightforward permissions continue to use just the traditional bits. Optional ACLs provide additional flexibility to more complex situations, providing an added level of access control but will only be enabled when required.

The file system continues to support the same types of backup tools and the `chmod` command will continue to function as before. Although the operating system does all the hard work behind the scenes (kernel), it provides a single method for checking the permissions to user-space applications.

5 Conclusion

The traditional permission model of Unix has limitations due to its simplicity. For many years, this simplistic permission model was effective. However, as older Unix systems continue to evolve into modern multi-user systems, these same limitations are quickly becoming increasingly problematic for developers and system administrators alike.

With the traditional permission model, Unix allows permissions to be set at the group level only, so that only one group can own the file and no more than one group can own the file. This limitation creates conflict among system administrators and users, as they must find ways to work around these limitations.

Extended Access Control Lists offer a logical next step for Unix systems, not just a new model of managing permissions. While supporting the structural elegance of the standard 'three-tier permission model,' extended access control lists add flexibility, enabling Unix-like operating systems to adapt to modern, composite permission strategies.

By storing Extended Access Control List permissions together with other file system data within the inode itself, as well as implementing utility programs like "setfacl" and "getfacl," allows for extensive and nuanced permission control on a granular basis, which is essential for real-world organizations.

References

The Linux Kernel Organization. (2025). EXT4 filesystem: Dynamic structures. *Linux Kernel Documentation*. https://www.kernel.org/doc/html/v6.5/filesystems/ext4/dynamic.html.

USENIX. (2002). POSIX Access Control Lists on Linux. https://www.usenix.org/legacyurl/posix-access-control-lists-linux.

Ubuntu Manpages. (2002). acl(5) - Access Control Lists. *Ubuntu Manual Pages*. https://manpages.ubuntu.com/manpages/jammy/man5/acl.5.html.

Kulkarni, D. (2023). Understanding file permissions and ACL on UNIX-like systems. *Hashnode*. https://dhananjaykulkarni.hashnode.dev/file-permissions-and-access-control-lists.

GeeksforGeeks. (2018). Access Control Lists (ACL) in Linux. https://www.geeksforgeeks.org/linux-unix/access-control-listsacl-linux/.

ITTC Help. (2021). ACL Guide. *University of Kansas*. https://help.ittc.ku.edu/ACL_Guide.

Ones. (2025). Mastering Unix permissions: User, group, and other access levels. https://ones.com/blog/mastering-unix-permissions-user-group-other-access-levels/.

Ohio Supercomputer Center. (2025). HOWTO: Use POSIX ACL. https://www.osc.edu/resources/getting_started/howto/howto_manage_access_control_list_acls/howto_use_posix_acl.