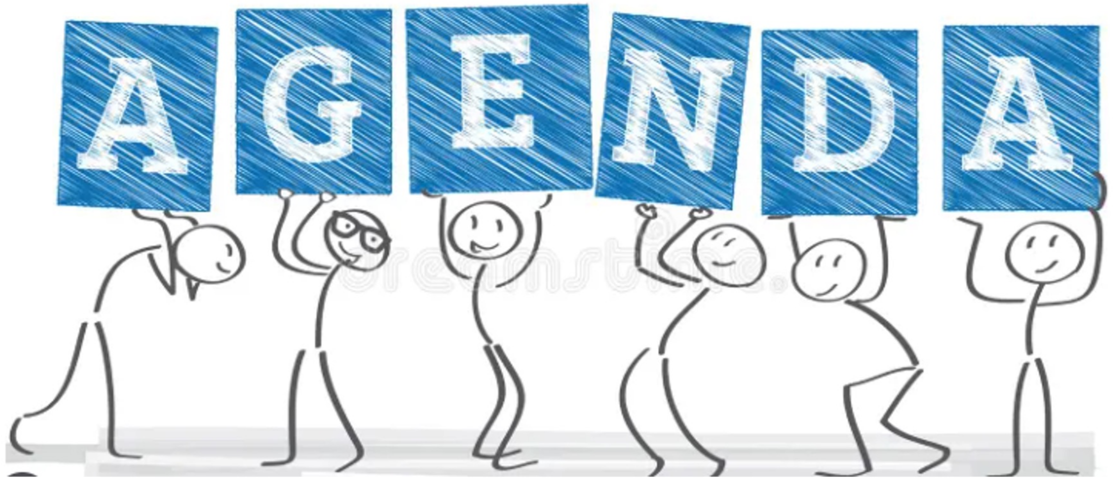
# Operating Systems

## Lecture 2.1

UNIX File Types and Permissions

Instructor: Muhammad Arif Butt, PhD

## Lecture Agenda



- Concept of Files and Directories
- Types of Files in UNIX based systems
  - Regular File
  - Directory
  - Soft/Symbolic Link
- A Deep Dive into File Permissions
- Changing File Permission using `chmod`
- Changing File Ownership using `chown`
- Where & How Linux Store passwords?
- Special Permissions in Linux

## Concept of Files and Directories

## The Concept of Files in Linux

- In the Linux ecosystem, a fundamental principle is that *everything is treated as a file*.
- This abstraction simplifies how the operating system and users interact with hardware and system resources.
- Whether you are writing to a document, reading from a keyboard, sending data to a printer, or accessing a hard drive, you are interacting with a file interface.
- This design philosophy provides a unified and consistent way to handle system resources. All these "files" reside within a hierarchical directory structure, starting from the root directory (/).

"The **UNIX philosophy** is often quoted as ***everything is a file***, but that really means ***everything is a stream of bytes***."

Linus Torvalds

## File Naming

- When a process creates a file it gives the file a name. When the processes terminates the file continues to exist and can be accessed by other processes using its name.
- The naming rules vary from system to system. Most OS allow strings of one to eight characters as legal file name allowing digits and some special characters. Some File Systems distinguish between upper and lowercase letters while others do not.
- WINDOWS
  - File names up to 255 characters.
  - Not case sensitive. File1, file1 and FILE1 all refer to same file.
  - Aware of file extensions, when a user double clicks on a file name, the program assigned to this file extension is launched with the file as parameter.
- UNIX
  - File names up to 255 characters, all acceptable except '/'.
  - Case sensitive. File1, file1 and FILE1 refer to three different files.
  - File extensions are just conventions and are not enforced by the OS.

# File Attributes

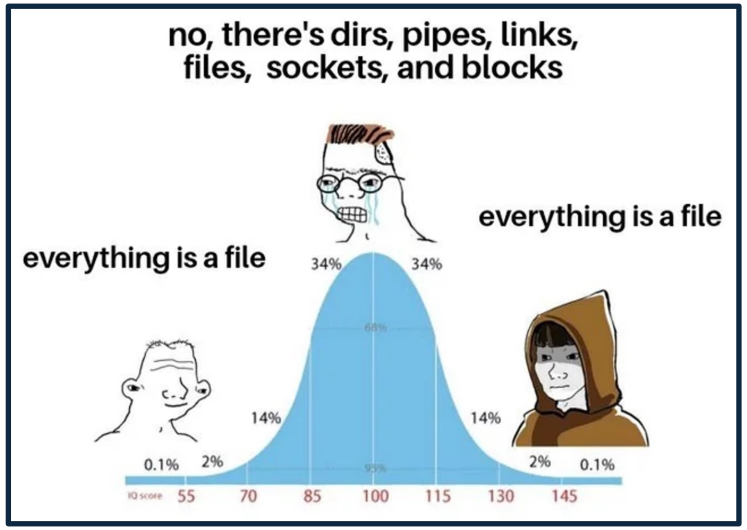
- Every file has a name and its data.
- In addition, all OS associate certain other information with each file, we call these extra items the file's attributes. List of attributes varies from system to system.
- Basic Information
  - File Name
  - File Type
  - File Organization
- Address Information
  - Starting address
  - Size used
- Access Information
  - Owner
  - Access List
  - Permitted Actions
- Time Stamps
  - Access Time (`ls -lu <filename>`)
  - Modification Time (`ls -l <filename>`)
  - Status Change (`ls -lc <filename>`)

```
$ stat /etc/passwd
File: /etc/passwd
Size: 3564        Blocks: 8          IO Block: 4096   regular file
Device: 252,1     Inode: 14288662    Links: 1
Access: (0644/-rw-r--r--)  Uid: (0/root)   Gid: (0/root)
Access: 2025-07-12 21:17:37.928071696 +0500
Modify: 2025-05-20 00:33:11.510675303 +0500
Change: 2025-05-20 00:33:11.510675303 +0500
Birth: 2025-05-20 00:33:11.510675303 +0500
```

## Types of Files in UNIX

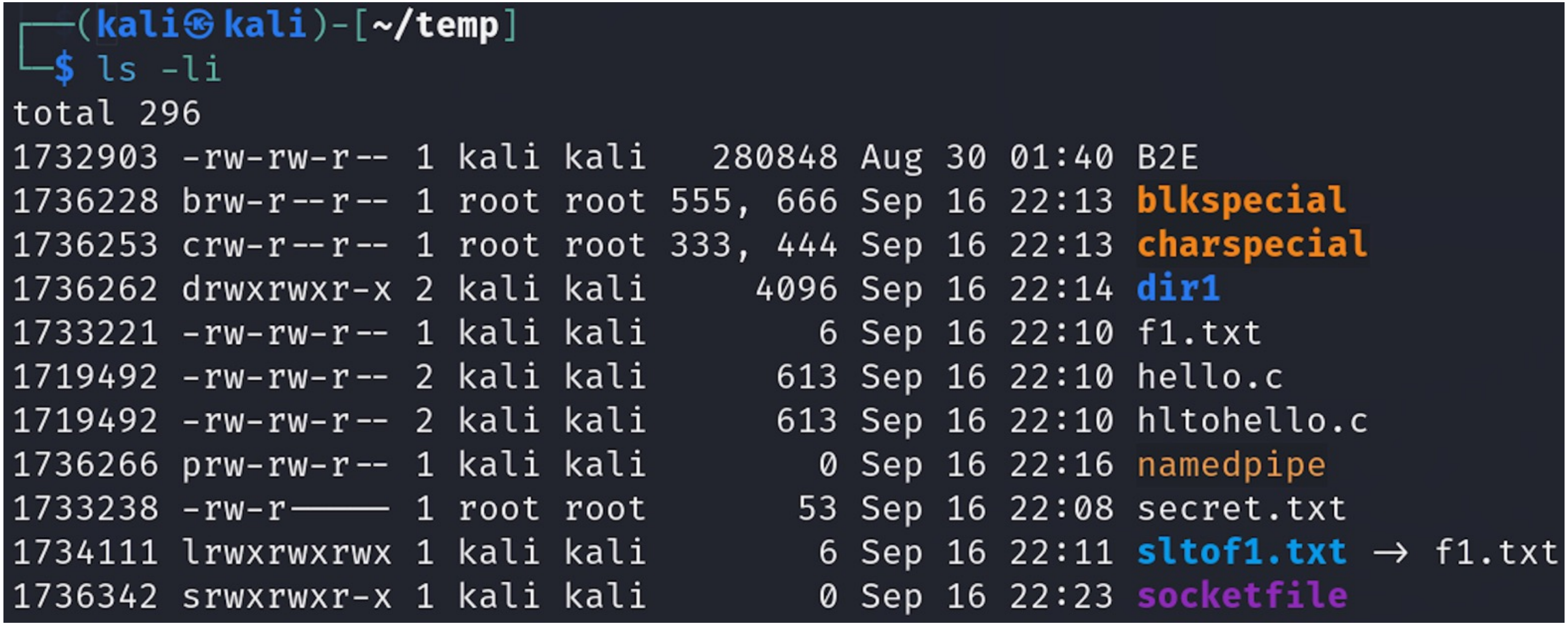
# Types of Files in UNIX

- **Regular file ( - )** Files that contain information entered in them by a user, an application program or a system utility program (ASCII text, binary, image, compressed etc).
- **Directory ( d )** Contains a list of file names plus pointers to associated i-nodes. Directories are actually ordinary files with special write protection privileges so only the file system can write into them, while read access is available to user programs.
- **Symbolic Link ( l )** Links let you give a file more than one name.



- **Block Special File ( b )** A block special file consists of as sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed, i.e. we can directly access block 154 without first having to read blocks 0 to 153. Block special files are typically used for disks. e.g. `/dev/hda1`, `/dev/lp`.
- **Character Special File ( c )** Used to communicate with h/w that input or output one character at a time. Keyboard, printers, mice, plotters, networks are examples of character special files.
- **Named Pipe (p)** A file that passes data between processes. It stores no data itself, but passes data between process writing data into pipe and process reading from pipe. (`ls -l /dev | less`)
- **Socket (s)** A stylized mechanism for inter-process communications (UNIX domain sockets)

# Types of Files in UNIX (cont…)



```
┌──(kali㉿kali)-[~/temp]
└─$ ls -li
total 296
1732903 -rw-rw-r-- 1 kali kali   280848 Aug 30 01:40 B2E
1736228 brw-r--r-- 1 root root 555, 666 Sep 16 22:13 blkspecial
1736253 crw-r--r-- 1 root root 333, 444 Sep 16 22:13 charspecial
1736262 drwxrwxr-x 2 kali kali     4096 Sep 16 22:14 dir1
1733221 -rw-rw-r-- 1 kali kali        6 Sep 16 22:10 f1.txt
1719492 -rw-rw-r-- 2 kali kali      613 Sep 16 22:10 hello.c
1719492 -rw-rw-r-- 2 kali kali      613 Sep 16 22:10 hltohello.c
1736266 prw-rw-r-- 1 kali kali        0 Sep 16 22:16 namedpipe
1733238 -rw-r----- 1 root root       53 Sep 16 22:08 secret.txt
1734111 lrwxrwxrwx 1 kali kali        6 Sep 16 22:11 sltof1.txt -> f1.txt
1736342 srwxrwxr-x 1 kali kali        0 Sep 16 22:23 socketfile
```

# What are Regular Files?

- A regular file is the most common file type in Linux that contains user data or program instructions stored as a sequence of bytes. It appears with a hyphen **(-)** as the first character in the file type field when viewed with `ls -l`.
- Regular files can contain any type of data including text, binary executables, images, documents, or any other form of information that applications can read and write.
- UNIX allows you to add extensions like `.txt` or `.jpg` to filenames, but the OS doesn't use them to determine how to handle the file. They're just helpful hints for users to know what type of content is inside the file.
- Instead of relying on file extensions, UNIX checks the first few bytes of a file (called "magic number") to determine if it's a program that can be run. This prevents users from accidentally trying to execute a text file or image as if it were a program.

| Regular File Types | Extension |
|---|---|
| Executable files | **exe, com, bin** |
| Machine language, but not linked | **obj, o** |
| Source code | **c, java, py, asm** |
| Batch files specific to cmd interpreter | **bat, sh** |
| Text files | **txt, doc** |
| Word processor files | **docx, doc, rtf, tex** |
| Library files | **lib, a, so, dll** |
| Print or view | **ps, pdf, jpg** |
| Archive files | **arc, zip, tar** |
| Multimedia files | **mpeg, mov, mp3, avi** |

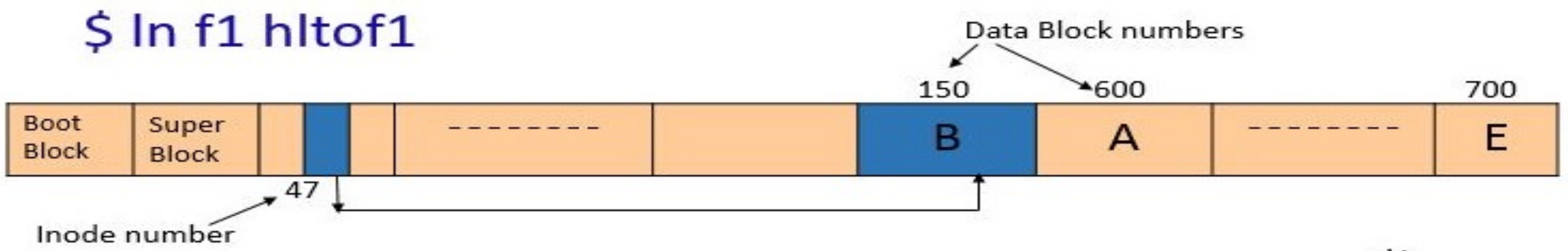
# What are Directories?

- In Linux, directories are special files that contain mappings between human-readable filenames and their corresponding `inode` numbers.
- In Linux, an `inode` number uniquely identifies a data structure (`inode` block) that contains the metadata/attributes as well as pointers to the data blocks of a specific file.
- It appears with a hyphen **`(d)`** as the first character in the file type field when viewed with `ls -l`.
- The opposite figure shows a pictorial view of a directory, having files and sub-directories. We will scuba dive to this in our next session ☺
- **WHY Do We Need Directories?**
  - Directories provide hierarchical organization, allowing logical grouping of related files
  - Enable multiple files with the same name to coexist in different directories
  - Directories inherit and enforce permissions for contained files
- In contrast to Linux, Windows stores file attributes directly in directory entries

| Inode# | Filename |
|---|---|
| 54 | . |
| 6 | .. |
| **47** | **f1.txt** |
| 49 | f2.txt |
| 34 | dir1 |
| 35 | dir2 |
| | |

```
/home/user/
├── Documents/
│   ├── work/
│   └── personal/
├── Pictures/
└── Downloads/
```

# What are Hard Links?



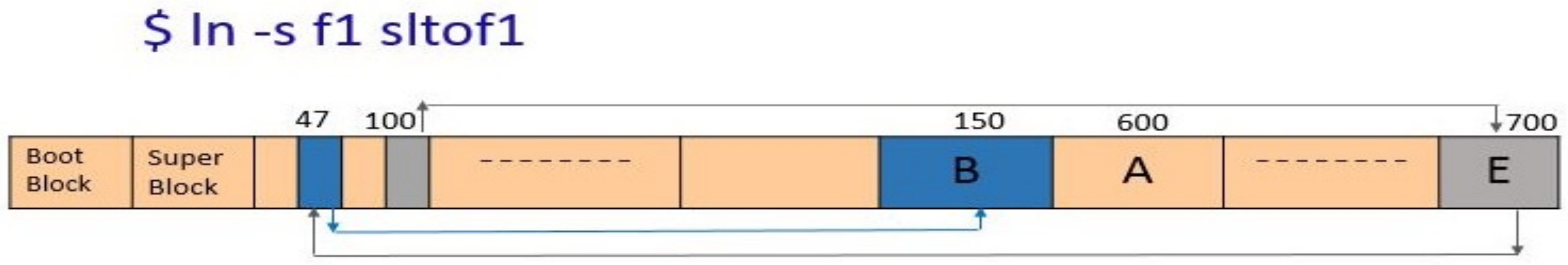
```
$ ln f1 hltof1
```

d1

| | |
|---|---|
| 47 | F1 |
| 47 | Hltof1 |
| | |
| | |

- A hard link is created using the **`ln original_file link_name`** command, which creates a new directory entry pointing to the same inode (no new data block or inode is allocated).
- The hard link appears as a regular file (-) with identical permissions, ownership, size, and timestamps, and shares the same inode number as the original file.
- When you create a file, it has one hard link, and creating additional hard links makes the file accessible via different paths and names. It is efficient way to have multiple names for the same file without duplicating data.
- With each hard link creation, the link count (shown in the third column of `ls -l`) is incremented; deleting any hard link only decrements this count, and data remains accessible until the link count reaches zero.
- Modifying any hard link updates the timestamps and content of all links since they reference the same inode.
- Regular users cannot create hard links to directories because it would create loops in the filesystem hierarchy, which is prohibited by the filesystem.
- Hard links cannot span across different filesystems/partitions because inode numbers are only unique within a single filesystem.

# What are Soft Links?



d1

| | |
|---|---|
| 47 | f1 |
| 100 | sltof1 |
| | |
| | |

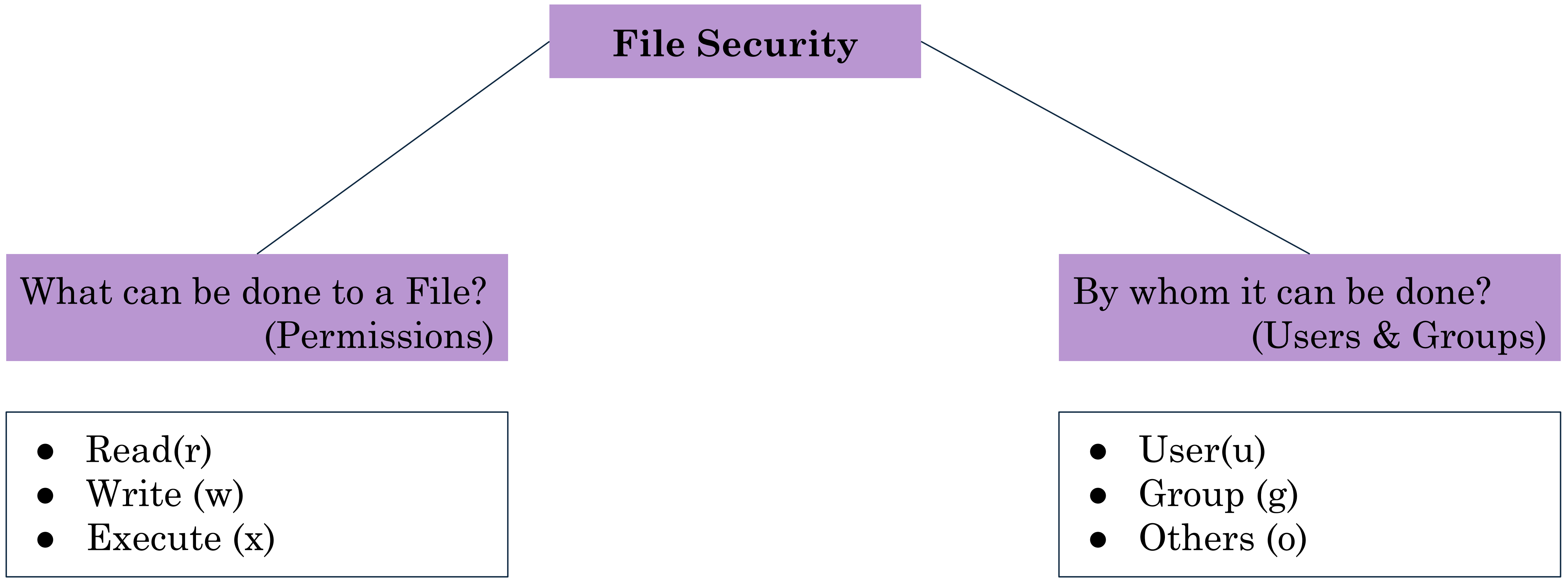
- A soft link is created using the **`ln -s original_file link_name`** command, which creates a new file containing the pathname to the target file (allocates a new inode and data block).
- The soft link appears as a symbolic link (`l`) with its own permissions and inode number, showing the target path when viewed with `ls -l`, and is distinct from the original file.
- Creating or deleting soft links does not affect the target file's link count; if the original file is deleted, the soft link becomes "broken" or "dangling" but continues to exist.
- Modifying the target file updates its timestamps, but the soft link's own timestamps remain unchanged unless the link itself is modified.
- Soft links can point to directories without filesystem restrictions, making them useful for creating shortcuts to folders anywhere in the system.
- Soft links can span across different filesystems and partitions because they store the target path as text rather than referencing inode numbers.

## File Permissions in UNIX

## File Protection & Security

- When information is kept in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection)
- File owner/creator should be able to control:
    - What can be done to the file / directory
    - By whom it can be done
- Different OS support different types of access to files and directories:
    - Read
    - Write
    - Execute
    - Append
    - Delete
    - List

# File Protection in Linux


File Security
What can be done to a File? (Permissions)
By whom it can be done? (Users & Groups)
● Read(r)
● Write (w)
● Execute (x)
● User(u)
● Group (g)
● Others (o)

## Users and Groups

- **Users** - Every user of a system is assigned a unique UID. User's names and UIDs are stored in `/etc/passwd` file. Users cannot read, write or execute each others files without permissions
- **Groups** - Users are assigned to groups with unique GID. GIDs are stored in `/etc/group` file. Each user is given his own primary group by default. He/she can belong to other secondary groups to gain additional access. All users in a group can share files that belong to that group.
- In context to UNIX Files, users are divided into three classes:
  - **User/owner:** The user who created the file. Any file you create, you own.
  - **Group:** The owner of a file can grant access of a file to the members of a specific group
  - **Others:** All users who are neither owner, nor group falls in others category

# Shell Commands (Users & Groups)

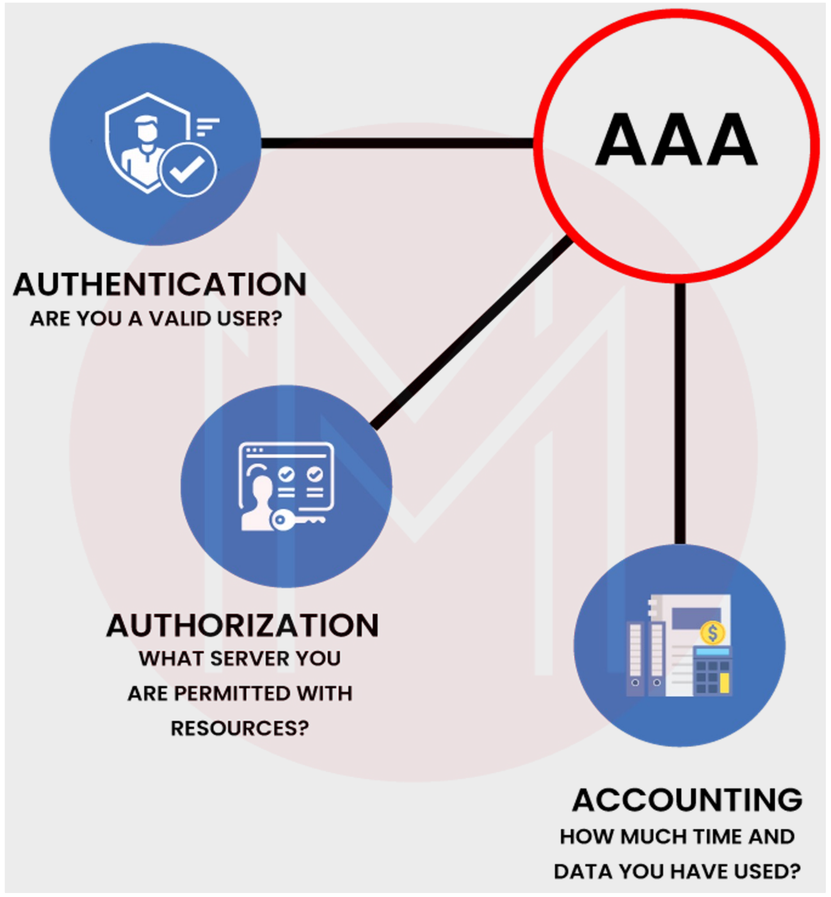
| User Management Commands | Description |
|---|---|
| **adduser <username>** | Interactive command to create a new user (/etc/passwd) |
| **useradd -m -d /home/user2 user2** | Low-level command to add user |
| **passwd [username]** | Used to change the password of currently logged in user or the user specified. The hash of the password is stored inside /etc/shadow file |
| **deluser <username>** | User we want to delete should be logged out. It don't deletes user HOME Dir |
| **userdel -r <username>** | Low level command that also deletes HOME directory with –r option |
| **usermod** | To modify user: (personal info (-c), default user shell (-s), user name (-l), lock user (-L), unlock user (-U), primary group (-g), secondary group (-G) |
| **groupadd <group-name>** | Create a new group. (/etc/group) |
| **groupdel <group-name>** | To delete group |
| **groupmod** | To modify group: change group name (-n) |
| **chage** | Used to change password expiry info of a user in /etc/shadow file |
| **chsh** | Used to change default user shell in /etc/passwd file |
| **chfn** | Used to change user personal info in /etc/passwd file |
| **finger** | shows user info in detail |
| **id** | Displays UID, primary GID and secondary GIDs you belong to |
| **su** | Switch user: To login using any username if we know its password (e.g: su -root). Using '-' option will also give you the target user environment. You will find yourself in the target user HOME Directory and his default login shell |

# AAA Architecture

- **Authentication** is the process of verifying the identity of a user or system trying to access a resource. In a Linux based machine every user must have an entry in the local user database files `/etc/passwd` along with `/etc/shadow`, `/etc/group` and `/etc/gshadow`
  - Something you know (password, passphrase, PIN)
  - Something you have (NIC, ATM, Passport)
  - Something you are (Biometrics)
  - Somewhere you are (Geographic location, IP, MAC address)
  - Something you do (signatures, pattern unlock)

- **Authorization** determines what an authenticated user or process is allowed to do, specifying access levels or permissions based on the user role or identity. (DAC, MAC, RBAC)

- **Accountability**, also known as auditing, involves tracking and recording user activities and resource usage. This information is used for monitoring, analysis, and compliance purposes.


AAA
AUTHENTICATION
ARE YOU A VALID USER?
AUTHORIZATION
WHAT SERVER YOU ARE PERMITTED WITH RESOURCES?
ACCOUNTING
HOW MUCH TIME AND DATA YOU HAVE USED?

# Access Control Models

- **Discretionary Access Control (DAC):** In DAC, *owner* of the object decides who all have what all kinds of access (`rwx`) on his object. This is the default for most Operating Systems.
- **Mandatory Access Control (MAC):** In MAC, *system* decides who all have what all kinds of access (`rwx`) on the objects. Every object has a security classification associated with it (e.g, top secret, secret, confidential, restricted, unclassified). Every user has a security clearance to access a specific class of object.
- **Role-Based Access Control (RBAC):** Users are assigned *roles*, and roles have specific permissions, simplifying management and enhancing security.

# Discretionary Access Control

Every file stored on the file system of a computer must be secured. No unauthorize person should be able to access it. The default Authorization scheme in all UNIX based systems is Discretionary Access Control (DAC), in which the owner of the file. traditional privilege scheme defines what actions a user or process can perform on files, directories, and other system resources. The following screenshot shows the long listing of a directory contents having seven different file types in Linux.

**Inode Number:** Cannot be changed

**Type of file**: Cannot be changed

**Permissions**: Can be changed using **`chmod`**

**Link Count**: Can be changed using **`ln`** or **`rm`**

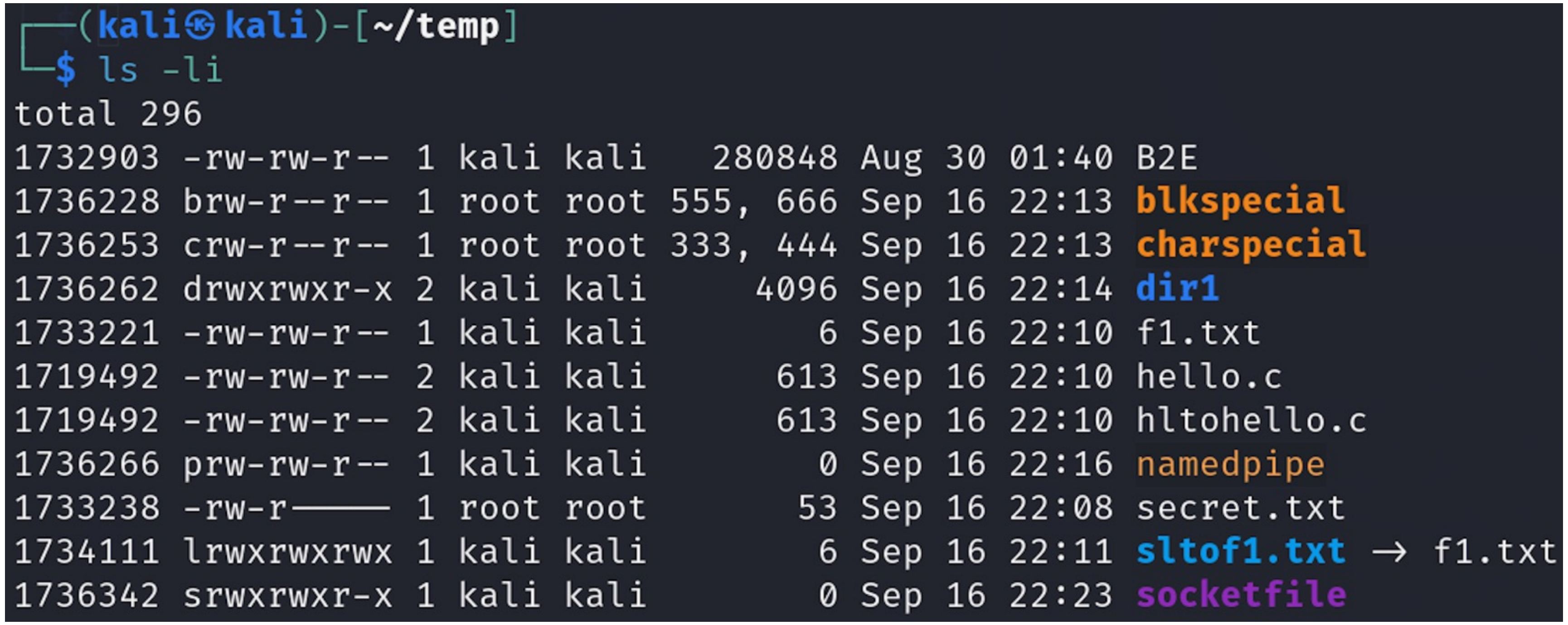
**Owner**: Can be changed using **`chown`**

**Group**: Can be changed using **`chgrp`**

**Size**: Can be changed by writing to file

**Date/Time**: Can be changed using **`touch`**

**Name:** Can be changed using **`mv`**



```
┌──(kali㉿kali)-[~/temp]
└─$ ls -li
total 296
1732903 -rw-rw-r-- 1 kali kali   280848 Aug 30 01:40 B2E
1736228 brw-r--r-- 1 root root 555, 666 Sep 16 22:13 blkspecial
1736253 crw-r--r-- 1 root root 333, 444 Sep 16 22:13 charspecial
1736262 drwxrwxr-x 2 kali kali     4096 Sep 16 22:14 dir1
1733221 -rw-rw-r-- 1 kali kali        6 Sep 16 22:10 f1.txt
1719492 -rw-rw-r-- 2 kali kali      613 Sep 16 22:10 hello.c
1719492 -rw-rw-r-- 2 kali kali      613 Sep 16 22:10 hltohello.c
1736266 prw-rw-r-- 1 kali kali        0 Sep 16 22:16 namedpipe
1733238 -rw-r----- 1 root root       53 Sep 16 22:08 secret.txt
1734111 lrwxrwxrwx 1 kali kali        6 Sep 16 22:11 sltof1.txt → f1.txt
1736342 srwxrwxr-x 1 kali kali        0 Sep 16 22:23 socketfile
```

## What do you mean by `rwx` Permissions

**For Files:**

- **READ**: Enables users to open files and read its contents using `less, more, head, tail, cat, grep, sort, view` commands
- **WRITE**: Enables users to open a file and change its contents using editors like `vi, vim, peco`
- **EXECUTE**: Enables users to execute files as a program or script (`a.out, script.py`)

**For Directories:**

- **READ:** Allows users to list directory contents using `ls` command
- **WRITE:** Allows users to create new files and directories, as well as delete files they own within a directory using `mkdir, touch, cp, mv` commands
- **EXECUTE:** Allows users to search in the directory and change to it using the `cd` command. Moreover, without execute permissions on a directory, read/write permissions are meaningless

## How Permissions are Checked?

- Whenever a user access a file/directory, the permissions are applied in following fashion:
    - If you are the user/owner, the user/owner permissions apply.
    - If you are in the group, the group permissions apply.
    - If you are neither owner nor group member, then others permissions apply

- To maintain the file type and permissions, all UNIX based systems use a 16 bit architecture as shown:

| File Type (4) | Special Permissions (3) | User (3) | Group (3) | Others (3) |
|---|---|---|---|---|
| 1000 | 000 | 110 | 100 | 000 |

## Changing File Permissions

## Default Permissions

- Whenever a user create a file or directory, the new permissions on the file are set by the creator program (`vim, touch, mkdir, mknod`) having the **`mode`** specified by the programmer in the `open()` system call along with the system's **`umask`** setting.
- The programs that create files typically have a mode of 0666, while the programs that create directories typically have a mode of 0777
- To check or change the current umask setting of your system, use the following command:

```
$ umask
022
$ umask 077
```

- For both files and directories the final permissions can be calculated by **`mode & ~umask`**
- **Examples:**
  - File created with mode 0666, umask 0022:

    `0666 & ~0022 = 0666 & 7755 = 0644`

  - Directory created with mode 0777, umask 0022:

    `0777 & ~0022 = 0777 & 7755 = 0755`

## Changing File Permissions

- The access rights for any given file can be modified by using the change mode (**`chmod`**) command
- **`chmod`** takes two lists as its arguments: permission changes and filenames.

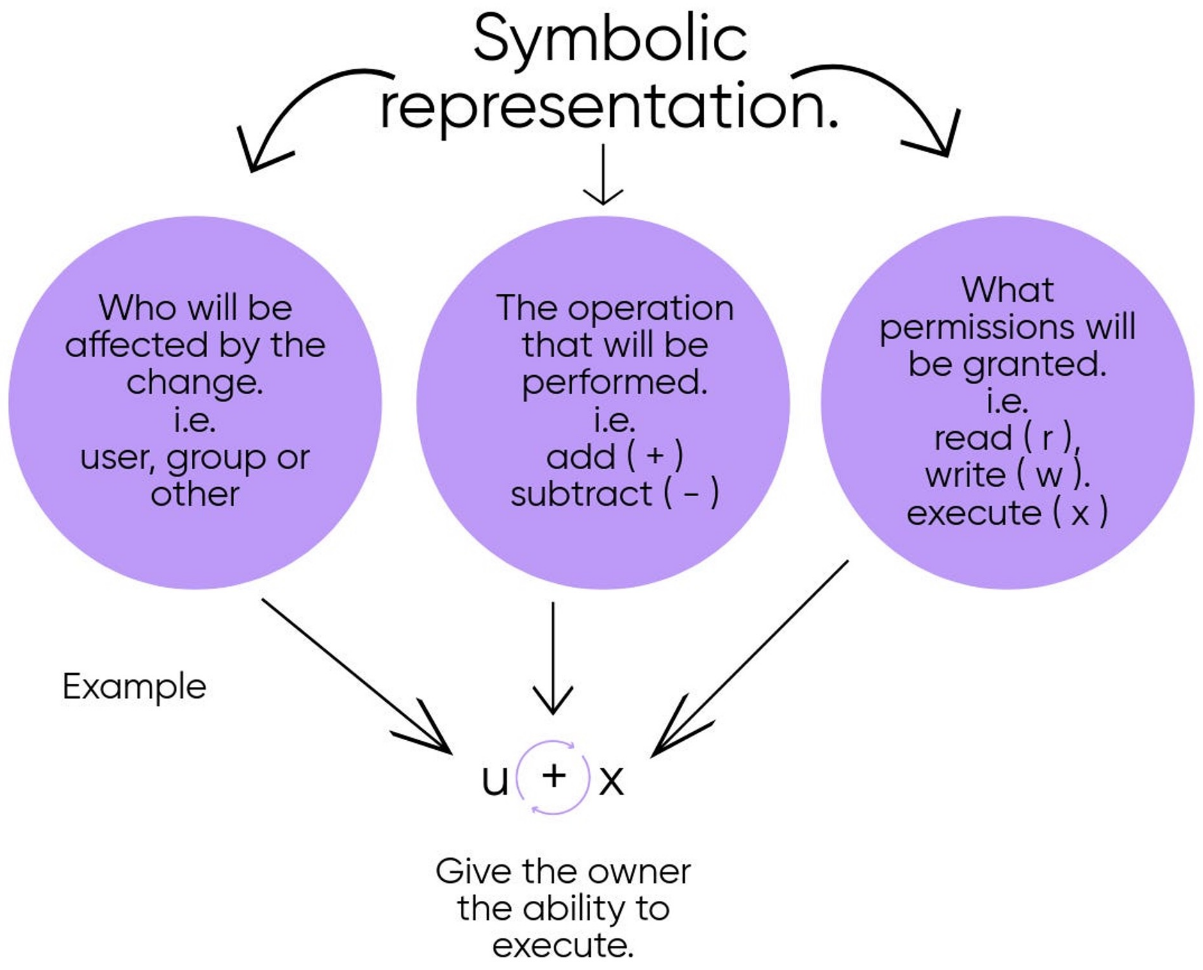
```
chmod <mode> <filename/dirname>
```

- We can use two different modes
  - Symbolic
  - Octal

## Symbolic Method

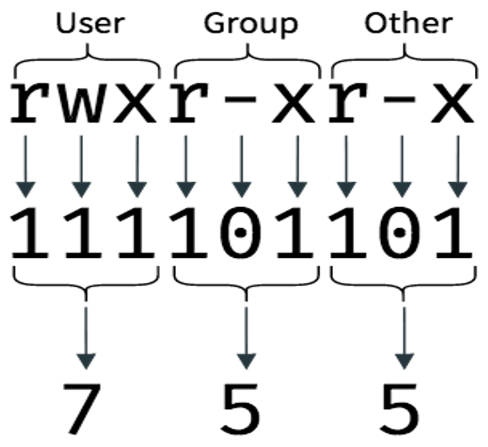
- **Symbols for Level**
  - u - Owner of a file
  - g - Group to which the user belongs
  - o - All other users
  - a - All Can replace u, g, or o
- **Symbols for Permissions**
  - + Add the following permissions
  - - Remove the following permissions
  - = Assigns entire set of permissions
- **Examples**

```
chmod u=rwx     filename
chmod g=rx      filename
chmod g+x       filename
chmod o-w       filename
chmod a+r       filename
chmod a+r-x     filename
chmod g=rw,o-w  filename
```


Symbolic representation.
Who will be affected by the change. i.e. user, group or other
The operation that will be performed. i.e. add ( + ) subtract ( - )
What permissions will be granted. i.e. read ( r ), write ( w ). execute ( x )
Example
u + x
Give the owner the ability to execute.

# Octal Method

- With the `chmod` command, we can specify a *three digit* octal number as `mode`, which will completely reset the permissions. You can't add/remove individual settings, as we can do in symbolic method of changing permissions.
- Each octal digit represent the permission that applies to owner, group and other respectively.
- When translated into a binary number, each octal digit becomes three binary digits, representing `rwx` permissions.



- **Examples**
  - ➢ `chmod 440     filename`
  - ➢ `chmod 750     filename`
  - ➢ `chmod 660     filename`

| r | w | x | Octal | Permissions |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | No Permissions |
| 0 | 0 | 1 | 1 | Execute Only |
| 0 | 1 | 0 | 2 | Write Only |
| 0 | 1 | 1 | 3 | Write and Execute |
| 1 | 0 | 0 | 4 | Read Only |
| 1 | 0 | 1 | 5 | Read and Execute |
| 1 | 1 | 0 | 6 | Read and Write |
| 1 | 1 | 1 | 7 | Read, Write and Execute |

## Changing File Ownership

# Changing file ownership with chown

- The `chown` ("change owner") command changes the owner and optionally the group of a file or directory.
- Requires `sudo` if you are not the current owner.
- To change only the owner of a file:

```
chown <new_owner> <file/directory>
```

- To change both the user and the group owner at the same time:

```
chown <new_owner>:<new_group> <file/directory>
```

- To change ownership of a directory and all files/subdirectories inside it (recursive):

```
chown -R <new_owner>:<new_group> <directory>
```

## Changing file ownership with chgrp

- The `chgrp` ("change group") command changes only the group owner of a file or directory. The user owner is not affected.
- Managing permissions for collaboration. It allows you to share files with a team (group) without changing the file's primary owner.
- To change the group of a file or directory:

```
chgrp <new_group> <file/directory>
```

- To change the group recursively for a directory:

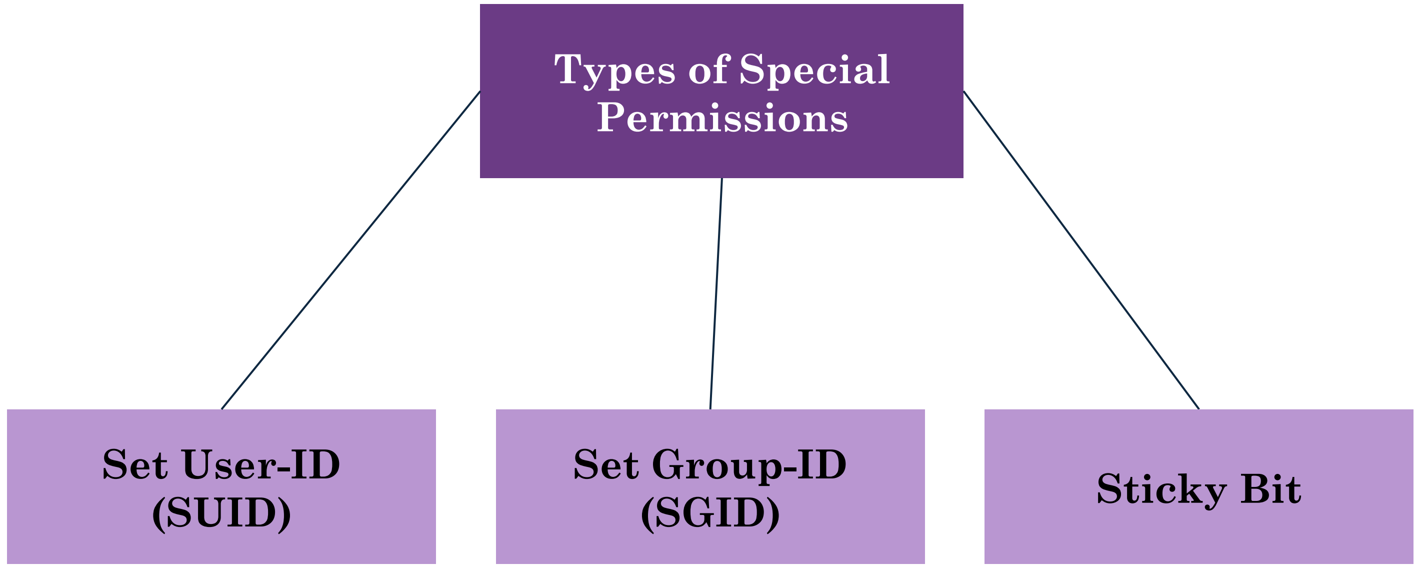
```
chgrp -R <new_group> <directory>
```

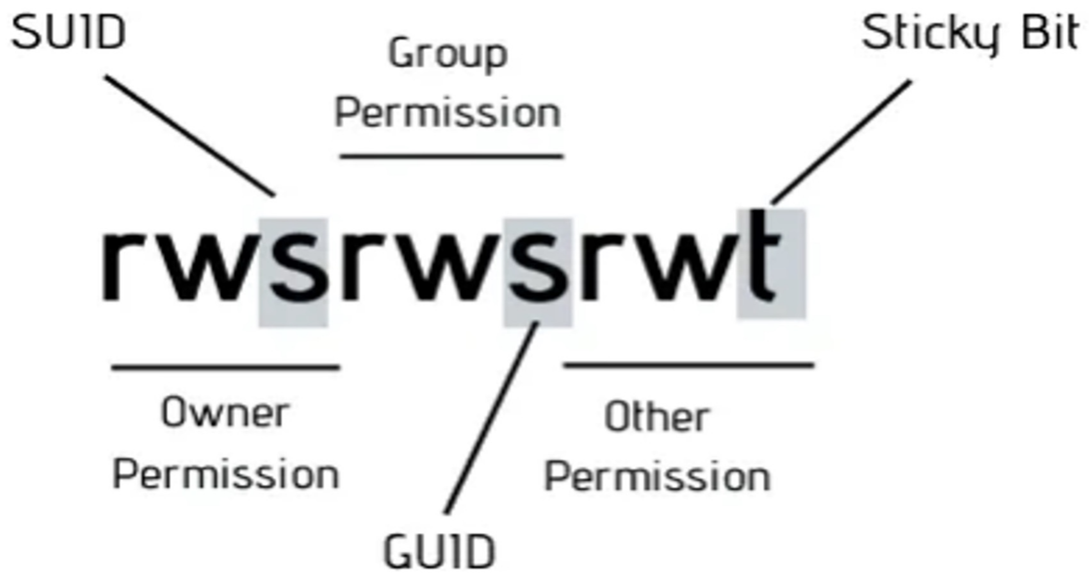
## Special Permissions

## Special Permissions

In Linux, special permissions extend standard file access controls and include SUID, SGID, and the Sticky Bit:

- **SUID Bit:** Set User ID bit allows a file to be executed with the file owner's privileges, commonly used for programs needing elevated access (e.g., /usr/bin/passwd).
- **SGID Bit:** Set Group ID bit causes executed files to run with the file group's privileges, and for directories, newly created files inherit the directory's group.
- **Sticky Bit:** When a directory has this bit set, it ensures that only the file owner or root can delete or rename files within it (e.g., /tmp).





# Set-User-ID Bit

- The SUID bit is typically set on executable files, and has no meanings on a directory.
- When an executable has the SUID bit set, it runs with the privileges of the file's owner, rather than the user who launched it.
- This is commonly used for programs requiring elevated privileges, such as the `passwd` command. Although normal users cannot write to `/etc/shadow` file, the `passwd` binary is owned by root and has the SUID bit set, allowing it to update the file securely on the user's behalf.
- It is represented by an **s** in the execute portion of owner permissions, or a capital **S** in case if the owner execute permission is off.
- To check the executable files in our system having their SUID bit set:

```
$ find / -type f -perm -4000 2>/dev/null
```

- To set the SUID bit of a program:

```
$ chmod u+s <myexe>
```

# Set-Group-ID Bit

- The SGID bit is typically set on executable files or directories.
- When an executable file has the SGID bit set, it runs with the privileges of the file's group, rather than the group of the user who runs it (`locate`, `wall`). This is useful when a program needs consistent access to resources owned by a specific group.
- When the SGID bit is set on a directory (`/var/mail/`), all new files or subdirectories created within it inherit the group ownership of the directory, instead of the user's default group. This is commonly used in shared directories to maintain consistent group collaboration.
- It is represented by an **s** in the execute portion of group permissions, or a capital **S** in case if the group execute permission is off.
- To check the executable files and directories in our system having their SGID bit set:

```
$ find / -type f -perm -2000 2>/dev/null
$ find / -type d -perm -2000 2>/dev/null
```

- To set the SGID bit of a program:

```
$ chmod g+s <myexe or dir>
```

## Sticky Bit

- The sticky bit is typically set on directories, having no meaningful effect on regular files in modern Linux systems (historically used for keeping executables in memory).
- When the sticky bit is set on a directory (`/tmp/`, `/var/tmp/`), only the file owner, directory owner, or root can delete or rename files within that directory, even if other users have write permissions to the directory. This prevents users from deleting each other's files in shared writable directories.
- It is represented by a `t` in the execute portion of other permissions, or a capital `T` if the other execute permission is off.
- To check the all the directories in our system having their sticky bit set, run the following command

```
$ find / -type d -perm -1000 2>/dev/null
```

- To set the sticky bit of a directory

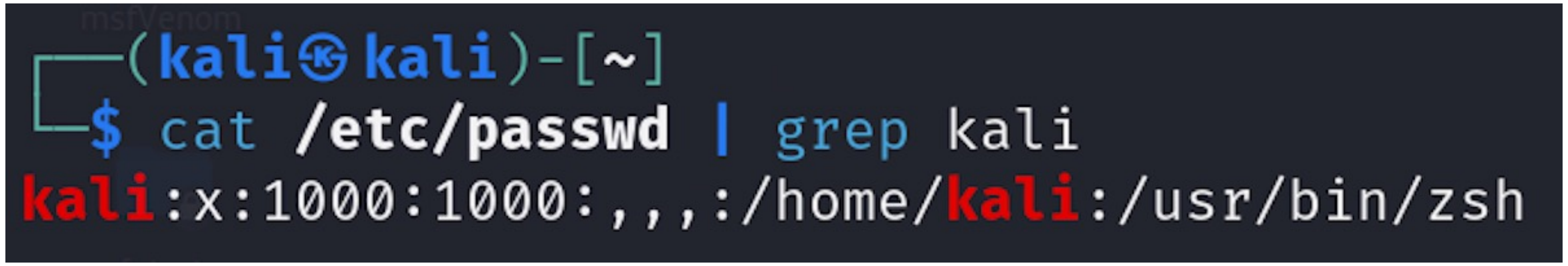
```
$ chmod o+t mydir
```

## Where & How
## Linux Stores Passwords?

# The /etc/passwd and /etc/shadow File

- To ensure password data is securely hashed and protected from compromise, modern operating systems use different approaches:
  - Linux typically uses salted algorithms like `SHA-512-crypt`, `yescrypt`, or `Argon2`;
  - macOS uses Password Based Key Derivation Function `PBKDF2` with `SHA-512`; and
  - Windows stores passwords as NT hashes (MD4-based) in the Security Account Manager (SAM) database
- The early version of UNIX in 1971, used to store the hashed value of passwords in the second field of the world readable **/etc/passwd** file. This was because this file contains user related information other than passwords and many applications require that information to function properly. In the later versions of UNIX, and in today's Linux distros, the hashed password is saved in the **/etc/shadow** file, which is readable only by super users.

**loginname:en_passwd:UID:GID:GECOS:homedir:shell**

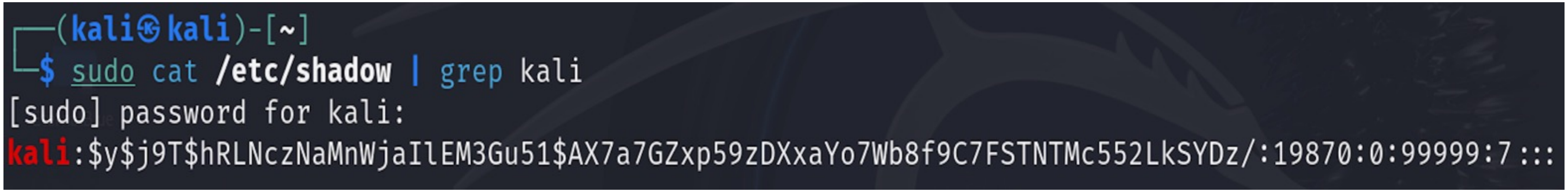


```
┌──(kali㉿kali)-[~]
└─$ cat /etc/passwd | grep kali
kali:x:1000:1000:,,,:/home/kali:/usr/bin/zsh
```

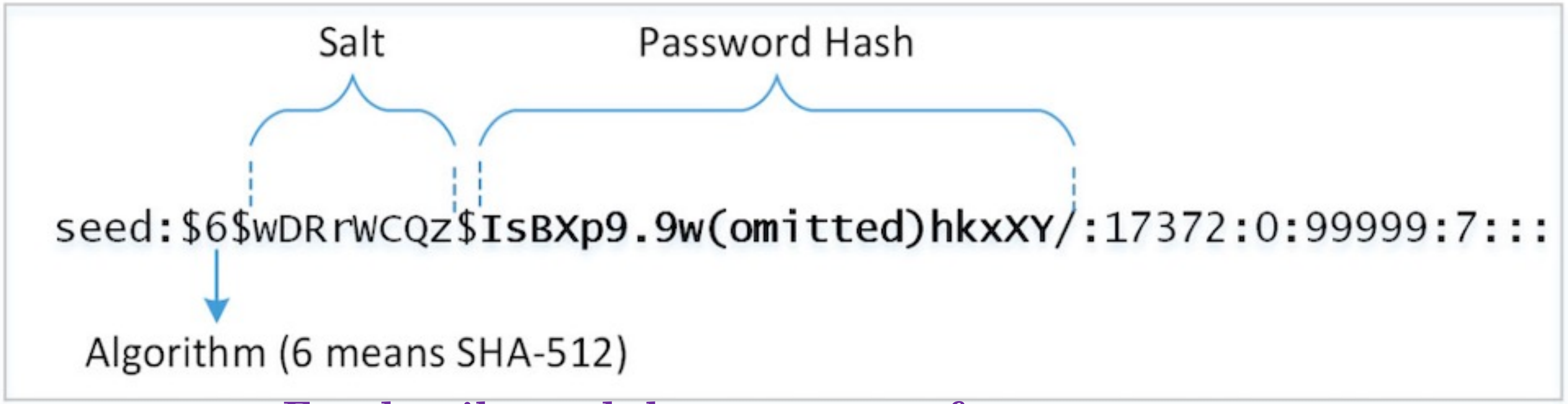
# The /etc/passwd and /etc/shadow File

The screenshot of the contents of the **/etc/shadow** file on my Kali Linux machine, with its nine fields is also shown below. Every row contains one record having nine colon separated fields:

```
user:$y$salt$hash:lastchanged:min:max:warn:inactive:expire:
```



```
┌──(kali㉿kali)-[~]
└─$ sudo cat /etc/shadow | grep kali
[sudo] password for kali:
kali:$y$j9T$hRLNczNaMnWjaIlEM3Gu51$AX7a7GZxp59zDXxaYo7Wb8f9C7FSTNTMc552LkSYDz/:19870:0:99999:7:::
```


Salt
Password Hash
seed:$6$wDRrWCQz$IsBXp9.9w(omitted)hkxXY/:17372:0:99999:7:::
Algorithm (6 means SHA-512)

**For details read the man page of shadow(5)**

# To Do

- Watch OS video on User Management:
  https://www.youtube.com/watch?v=eA3YOhtWHQk&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=16
- Watch OS video on Hard and Soft Links:
  https://www.youtube.com/watch?v=g8xZgtuYiWI&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=22
- Watch OS video on File Permissions:
  https://www.youtube.com/watch?v=tEYYasCVxRc&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=23
- Watch OS video on Special File Permissions:
  https://www.youtube.com/watch?v=6CJtdvL9P-Y&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=24

**Coming to office hours does NOT mean that you are academically weak!**