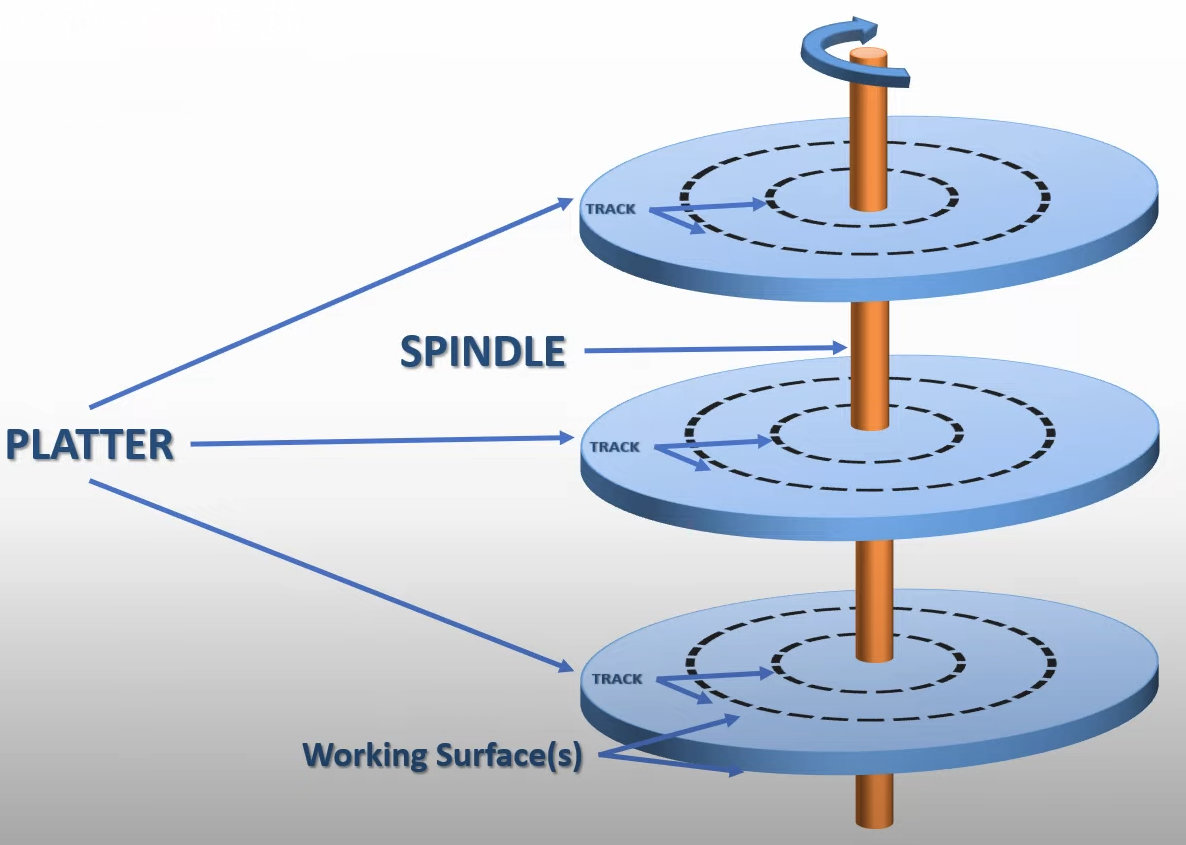
**Disk, Disk Partition and Filesystem [ dumpe2fs, tune2fs, fdisk, cfdisk, parted, gparted, lvm, mkfs, fsck, e2fsck ]**

**Introduction**

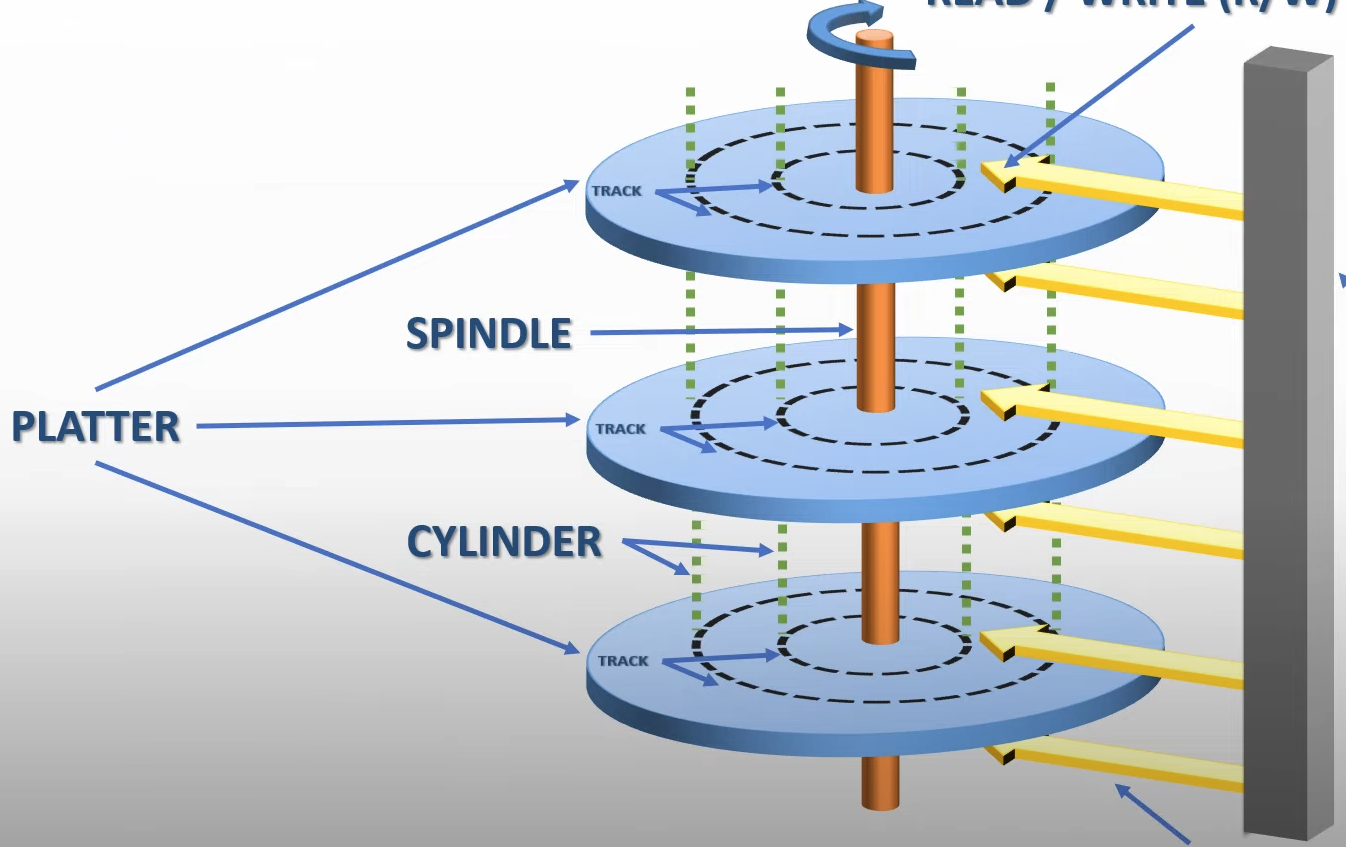
Disk partitioning is an essential step in managing storage efficiently, allowing me to organize the hard drive into separate sections for different purposes. Tools like fdisk and cfdisk offer command-line simplicity, while parted provides advanced options for resizing and creating partitions. For a more visual approach, gparted makes things even easier with its graphical interface.

**Types of Disk**

1. **Hard Disk Drive (HDD)**

****

* The platters are circular disks made of non-magnetic material and coated with magnetic film.
* The platter has 2 working surfaces where the data can be written.
* The spindle motors are responsible for rotating the platters to read or write data in the respective track and sector.

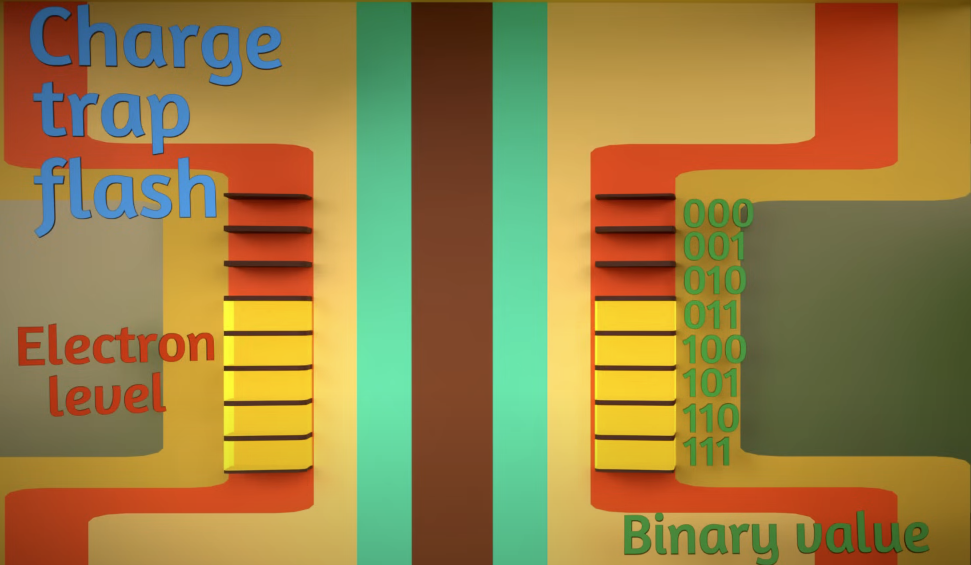


* The collection of tracks at the same distance from edges of the platter form a cylinder.
* Each track consists of various sectors onto which data is written.
* The data is written in the respective sector using the actuator arm (head).
* When the data has to be written or read from the HDD, the head first finds the track on which the data has to be written or read and then it finds the sector where the data has to be written or read from.

1. **Solid State Drive (SSD)**

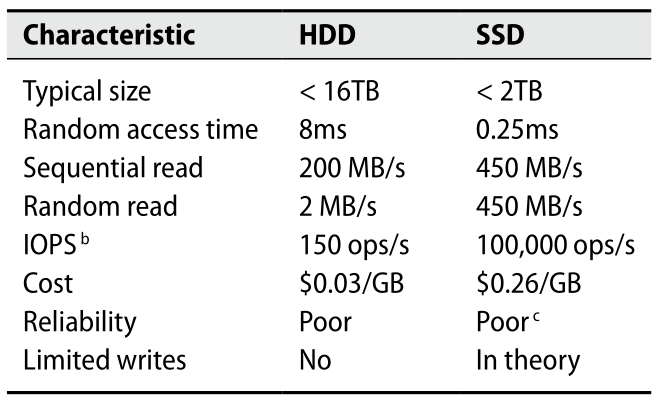
****

* The SSD uses NAND flash technology, it consists of memory cells (above diagram) that can trap different levels of electrons.



* The charge trap can store up to 3 bits of electron (1-7).
* There are a total of 40,000 columns and 50,000 rows, moreover 100 layers like this are stacked on top of it for data storage and redundancy.
* To write or read data, the rows are accessed using bitline selectors and to select the layers control gate selectors are used.
* The above layout is copied and 8 more layers like this are created and stacked on top of each other and packed into a small chip.

**HDD Vs SSD**

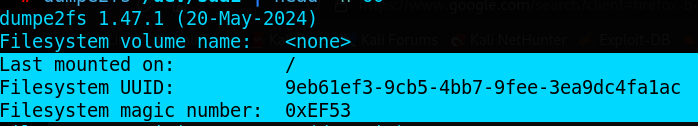


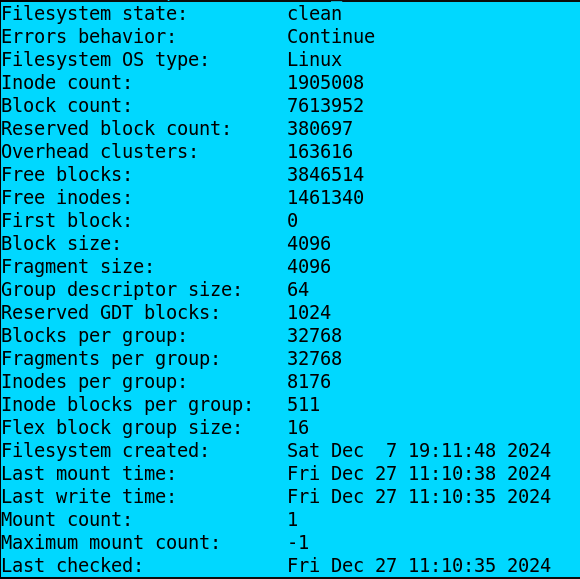
**Types of Interfaces**

| **Features** | **Small Computer System Interface (SCSI)** | **Parallel ATA (PATA)** | **Serial ATA (SATA)** | **Non-Volatile Memory Express (NVMe)** | **Serial Attached SCSI (SAS)** |
| --- | --- | --- | --- | --- | --- |
| Year | 1975 | 1986 | 2003 | 2013 | 2004 |
| Data Rate | 80 MBps | 133 Mbps | 6 Gbps | 32 Gbps | 12 Gbps |
| Communication Type | Parallel | Parallel | Serial | Serial | Serial |
| Connector Type | 50-pin or 68-pin or 80-pin | 40-pin | 7-pin | PCIe slot attachment | 7-pin |

**Dumpe2fs**

The dumpe2fs is used to display information about the ext2/ext3/ext4 file system. Some of the useful information of this command are illustrated below





*dumpe2fs <device name> (or) dumpe2fs /dev/sdb*

**Tune2fs**

The tune2fs is used to adjust the tunable file system parameter on the ext2/ext3/ext4 file system.

*tune2fs <option> <device name> (or) tune2fs -c 5 /dev/sda*

**Options**

c - To change the maximum mount count.

I - To change the inode size used by the file system.

j - To add ext3 journal to the file system.

l - To list the contents of the file system super block.

u - To set the user who can use the reserved file system block.

**Types of File System**

A filesystem is a method and data structure that an operating system uses to manage, organize, store, and retrieve data on storage devices like hard drives, SSDs, or USB drives. It determines how data is stored, how files are named, and how directories are structured. The file system ensures that data is saved in a structured way and can be accessed quickly and efficiently.

**Virtual File System ( VFS )**

Virtual File System (VFS) is like a translator between your operating system and the different types of filesystems (like ext4, NTFS, FAT32, etc.) on your storage devices. It acts as an abstraction layer between user-space processes and the various filesystems and storage devices.

Imagine you have different types of storage (e.g., a local hard drive, a USB stick, or a network drive), each using its own language (its own filesystem). The VFS enables us to communicate with all these different types of physical storage and file systems without any difficulty. It doesn’t matter what type of filesystem is behind the scenes because the VFS provides a common way to access, read, and write files.

**Components of VFS**

**Superblock Object:**

* The superblock is a data structure that contains essential metadata about a mounted file system. This metadata can include:
  + File system type (e.g., ext4, NFS, FAT)
  + File system size
  + Free space available
  + Pointers to operations that the file system supports (e.g., read, write, mount).
  + The location of the inode table (which is stored in a separate block in the filesystem) and other file system-specific details
* Every file system has its own superblock. When a file system is mounted, its superblock is loaded into memory.
* The superblock ensures that the kernel can interact with the file system and manage file operations properly.

**Inode (Index Node) Object:**

* An inode is a data structure that stores metadata about a file or a directory. It doesn't store the file name or data content but holds the following information:
  + File size: The total size of the file.
  + Permissions: Who can read, write, or execute the file.
  + Owner: The user or group that owns the file.
  + Timestamps: Creation, modification, and last access times.
  + Pointers to data blocks: These point to the locations on disk where the actual file content is stored.
* Inodes are used to locate actual content of files using its metadata and pointers.
* When a file is accessed, VFS will find the file’s inode using dentry to obtain this metadata and work with the file’s data.

**Dentry (Directory Entry) Object:**

* A dentry represents a directory entry (mapping between file name and its inode) and is responsible for mapping a file's name to its inode.
* The dentry helps VFS to resolve the full path of the file by looking up directory entries. To resolve the following path /home/user/file.txt the following process takes place
* VFS starts with the root directory (/) and checks the dentry cache for the entry corresponding to the root.
* If the dentry for / is cached, VFS uses it; if not, VFS searches the root directory’s blocks on disk to resolve it.
* Once / is resolved, VFS moves to the next component in the path: home. It checks the dentry cache for home.
* If the dentry for home is cached, VFS uses it; if not, VFS looks up the directory blocks of / to find the inode number of home.
* Next, VFS resolves the user. It checks the dentry cache for the user under the home directory.
* If the dentry for the user is cached, VFS uses it; otherwise, it searches home’s directory blocks to get the inode number of the user.
* Finally, VFS resolves file.txt by checking the dentry cache for file.txt under the user directory.
* If file.txt is found in the cache, VFS uses it; otherwise, VFS looks up the user directory’s blocks to get the inode number of file.txt.
* The dentry for file.txt contains the inode number, and VFS uses this number to retrieve the corresponding inode.
* The inode for file.txt contains metadata such as file permissions, timestamps, and pointers to the data blocks.
* With the inode, VFS can now perform the requested file operation (e.g., read, write).

**File Object:**

* The file structure represents an open file and is used to manage the state of the file during its use in the system.
* It includes:
  + File descriptor: A handle used by the application to interact with the file.
  + Current file position: The location within the file where the next read/write operation will happen.
  + File operations: These are the specific functions (like read, write, close) associated with the file, and they are tailored for the underlying file system.
  + Pointer to the inode: The file structure holds a reference to the inode that stores the metadata and the actual data of the file.
* The file structure is created when a file is opened and is removed once the file is closed.

**Mount Table:**

* The mount table is a list of all the file systems currently mounted on the system.
* Each entry in the mount table contains:
  + A reference to the superblock of the file system.
  + The mount point: The directory in the filesystem where the file system is attached.
  + The file system type and any specific mount options.
* It helps VFS keep track of which file systems are accessible and where they are mounted in the directory hierarchy.

**Working of VFS**

**File System Mounting:**

* When a file system is mounted, VFS creates a mount point and associates it with the superblock of the file system. The superblock contains metadata about the file system, such as the location of the inode table.

**Path Resolution:**

* When an application accesses a file (e.g., open("/home/user/file.txt")), VFS resolves the path by looking up each component (/, home, user, file.txt) in the dentry cache.
* If the component is not found in the cache, VFS reads the corresponding directory block from disk to resolve the inode number.
* The inode contains metadata about the file, such as permissions, size, and pointers to the file's data blocks.

**System Call Handling:**

* Once the inode is located, VFS delegates the requested operation (e.g., read(), write()) to the specific file system handler. Each file system (e.g., ext4, NFS) implements its own version of these operations.

**File Descriptors and Operations:**

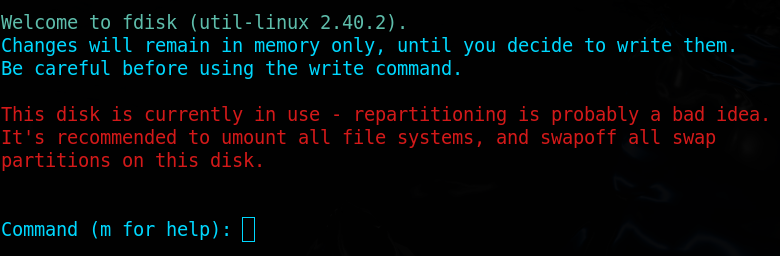
* After successfully opening the file, VFS creates a file descriptor that points to the file\_operations structure, which defines the file system-specific operations.
* VFS uses this structure to carry out the requested action on the file.

**Caching:**

* VFS improves performance through caching mechanisms:
  + Dentry cache: Stores the results of path resolution for quick access to directory entries.
  + Inode cache: Stores inodes to reduce disk I/O for file metadata.
  + Page cache: Caches file data to reduce read operations from disk.

**Fdisk**

Fdisk is a dialog-driven program for creation and manipulation of partition tables, easy to create and delete partitions and has various options in it.



*fdisk <device name> (or) fdisk /dev/sda*

**Working**

1. When you type fdisk, the shell looks for the binary in your $PATH directories.
2. fdisk opens the disk device file (e.g., /dev/sda) using system calls like open() to access the disk.
3. It reads the partition table by directly accessing the disk's first few sectors.
4. When you create or delete a partition, it modifies the in-memory partition table and writes changes back to the disk using write() and related system calls.
5. The changes are finalized when the user writes the partition table, notifying the kernel using ioctl() to re-read the partition table.

**Options :**

m – To list the help menu.

p – To print the partition.

n – To create a new partition.

d – To delete the partition.

t – To select the type of the partition.

w – To write changes to a disk.

q – To exit the fdisk.

g – To create a new empty GPT partition table.

o – To create a new empty MBR (Dos) partition table.

F – To list free unpartitioned space .

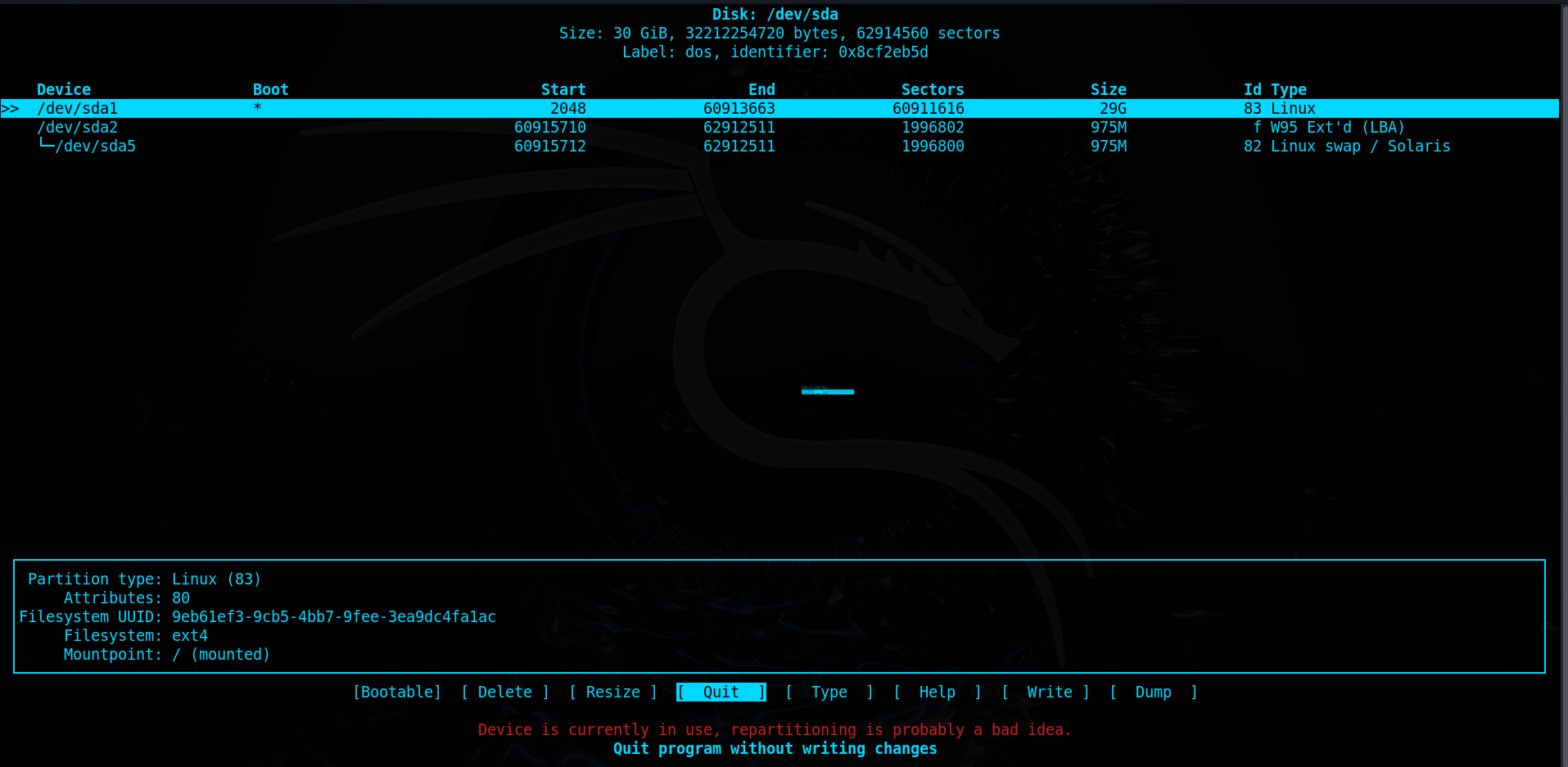
v – To verify the partition table.

l – To list available partition types.

**Note :**

**Cfdisk**

Cfdisk is a curses-based program for manipulating any block device, easy to create, delete and resize partitions not the filesystem.



*cfdisk <device name> or cfdisk /dev/sda*

**Working**

1. Like fdisk, the binary is located and executed by the shell.
2. cfdisk uses the ncurses library to render its text-based UI.
3. It interacts with the disk device file and partition table in the same way as fdisk (using open(), read(), and write()).
4. Changes to the partition table are made in memory and written back only when the user confirms.

**Options :**

b – To add a bootable flag on a selected partition.

n – To create a new partition.

h – To print the help menu.

d – To delete a selected partition.

W – To write changes to a disk.

r – To resize the disk partition (only the disk not the filesystem).

q – To exit cfdisk.

Up Arrow – To move the cursor to the previous partition.

Down Arrow – To move the cursor to the next partition.

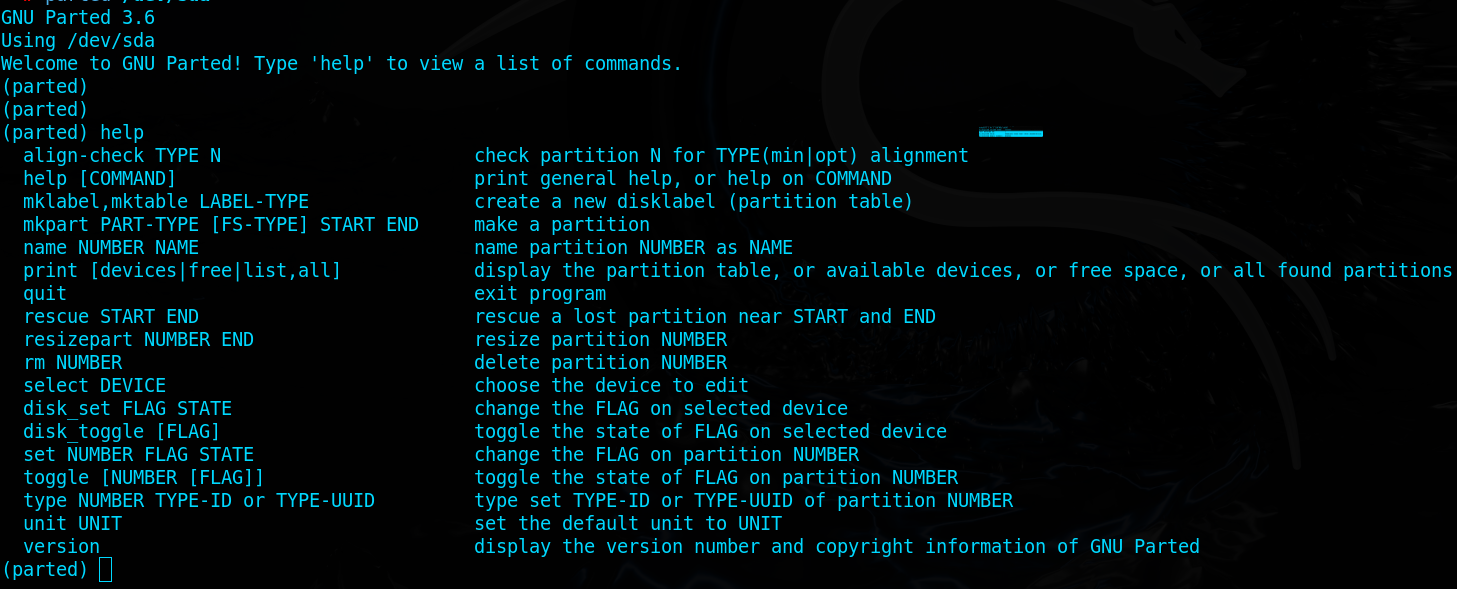
Left Arrow – To move the cursor to the previous menu item.

Right Arrow – To move the cursor to the next menu item .

**Note :**

**Parted**

Parted is a program to manipulate disk partitions, easy to create partitions, delete partitions, resize partitions and copy data to new hard drives and more.



*parted <device name> (or) parted /dev/sda*

**Working**

1. The shell finds the parted binary and starts the process.
2. parted opens the disk device file (e.g., /dev/sda) using low-level system calls.
3. It reads the GPT or MBR structures from the disk, parses them, and loads them into memory.
4. Commands like creating or resizing partitions modify the in-memory partition table, which is written back when changes are confirmed.

**Options :**

help – To display the help menu / display help about below commands.

*help <command> (or) help mktable*

mkpart – To create a new disk partition.

mktable – To create a new partition table (GPT, MBR, etc).

*mktable <type> (or) mktable gpt*

rm – To delete a partition.

*rm <partition number> (or) rm 1*

print – To print partition details

*print device (or) print free (or) print list (or) print all*

quit – To exit parted program

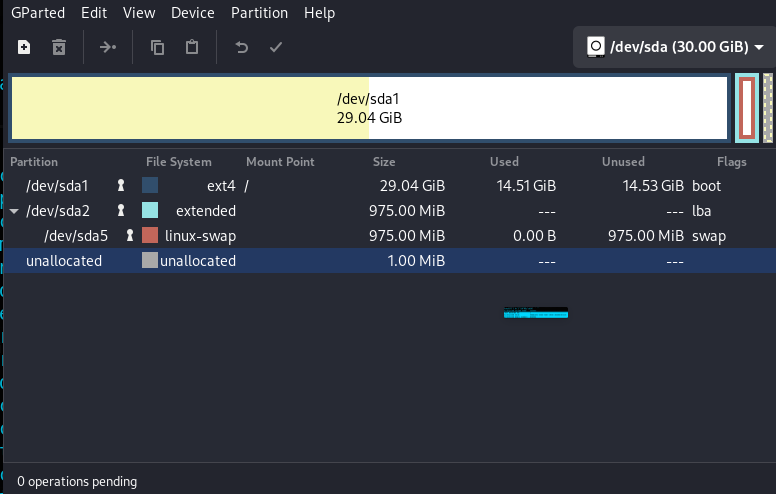
resizepart – To resize given partition

*resizepart <partition number> <size> (or) resizepart 1 10G*

**Note :**

**Gparted**

Gparted application is a GNOME partition editor for creating, reorganizing and deleting disk partitions as well as reducing the risk of loss of data.



**Working**

1. When launched, gparted initializes a graphical interface using the GTK library.
2. It scans all connected storage devices by executing low-level commands (lsblk, parted) or reading files like /proc/partitions and /sys/block/.
3. For any operation (e.g., resizing a partition), it spawns backend tools (parted, mkfs) to execute the requested changes.
4. It updates the GUI in real time to reflect disk modifications.

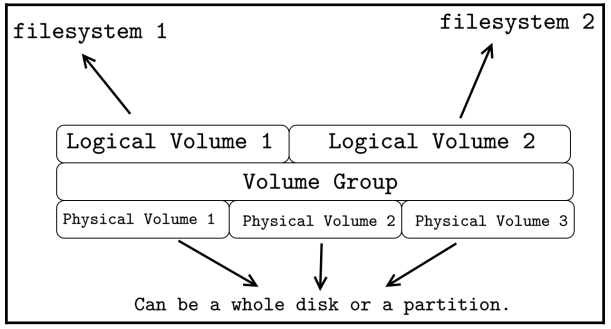
**Note :**

**LVM**

The Logical Volume Manager provides tools to create logical block devices from physical devices. **Volume Group** is a collection of one or more physical devices. Each of these physical devices are called **Physical Group**. A **Logical Group** is a virtual block device that can be used by the system or applications.

**Working**

1. The lvm command interacts with the device-mapper kernel module.
2. It reads metadata stored on physical volumes (PVs) and volume groups (VGs).
3. LVM maintains a mapping of logical to physical volumes. When you create or resize a logical volume, it updates this mapping in the metadata.
4. It uses ioctl() system calls to notify the kernel of changes to the logical volumes.

****

**Steps to create LVM**

**1. Create Physical Volume**

*pvcreate <device name> (or) pvcreate /dev/sda*

**Physical Volume Commands**

pvdisplay – To display information about available physical volume

pvchange – To change attributes of physical volume.

pvremove – To remove a created physical volume.

pvresize – To resize the physical volume.

pvck – To check metadata on physical volume.

pvs – To list all physical volumes.

pvscan – To list all physical volumes.

**2. Create Volume Group**

*vgcreate <vg name> <physical volumes > (or)*

*vgcreate volgrp0 /dev/sda3 /dev/sdb*

**Volume Group Commands**

vgdisplay – To display information about available volume groups.

vgck – To check consistency of volume groups.

vgextend – To increase the size of the created volume group.

vgrename – To rename an existing volume group.

vgremove – To remove a created volume group.

vgreduce – To remove a physical volume in the volume group.

vgs – To list all volume groups.

vgscan – It searches for all existing volume groups.

vgmerge – To merge two or more existing volume groups.

vgchange – To change the attributes of the volume group.

**3. Create Logical Volumes**

*lvcreate --size <size> --name <lv name> <vg name> (or)*

*lvcreate --size 5G --name lv\_root volgrp0*

**Logical Volume Commands**

lvdisplay – To display information about available logical volumes.

lvs – To list all available logical volumes.

lvscan – To list all logical volumes in all volume groups.

lvextend – To increase the size of the existing logical volume.

lvreduce – To reduce size of the existing logical volume when not mounted).

lvrename – To rename an existing logical volume.

lvremove – To remove a created logical volume.

lvresize – To resize the logical volume without unmounting.

lvchange – To change the attributes of the logical volume.

**Note :**

* logical volumes are represented in the device mapper directory (/dev/mapper)

**4. Create Filesystem**

*mkfs.<filesystem type> <device name> (or)*

*mkfs.ext4 /dev/mapper/lv\_root*

**5. Mounting Logical Volume**

*mount <device name> <target directory> (or)*

*mount /dev/mapper/lv\_root /logical\_vol\_folder*

**Fsck**

The fsck (short for File System Consistency Check) is a command-line utility in Linux and UNIX-like systems used to check and repair filesystem inconsistencies. It scans the filesystem for errors and attempts to fix them, ensuring the integrity and usability of the filesystem.

*fsck <option> <device name> (or) fsck /dev/sda*

***E2fsck***

The e2fsck is a command-line tool used to check and repair ext2, ext3, and ext4 filesystems. It is part of the e2fsprogs package and works directly with ext-based filesystems to detect inconsistencies, repair corruption, and ensure the filesystem is in a usable state.

*e2fsck <options> <device name> (or) e2fsck /dev/sda*