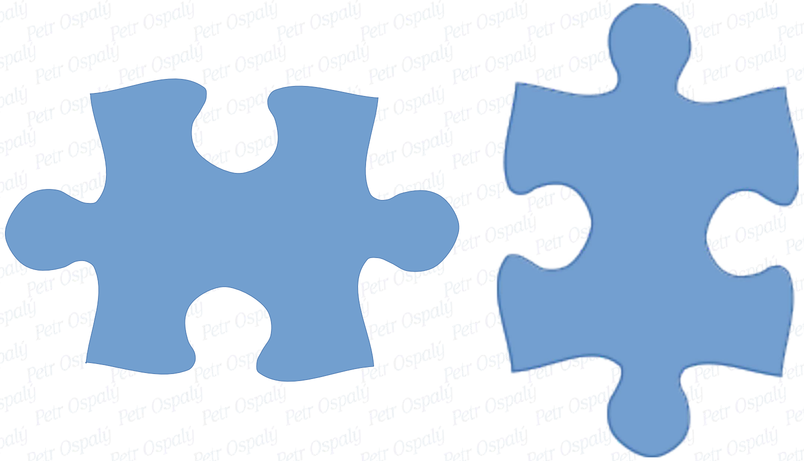


Files and Filesystems

Petr Ospalý

August 2025

# Files (1/2)

- Linux is **UNIX-like** OS
- „On a UNIX system, everything is a file; if something is not a file, it is a process.“
- File is an abstract structure to **represent** data
- File has a **path**, a **name**, **metadata** and the actual **data** (or their representation – special files)
- There are multiple types of files – their type is stored in metadata
- Type of file is not defined by extension like in Win/DOS

# Files (2/2)

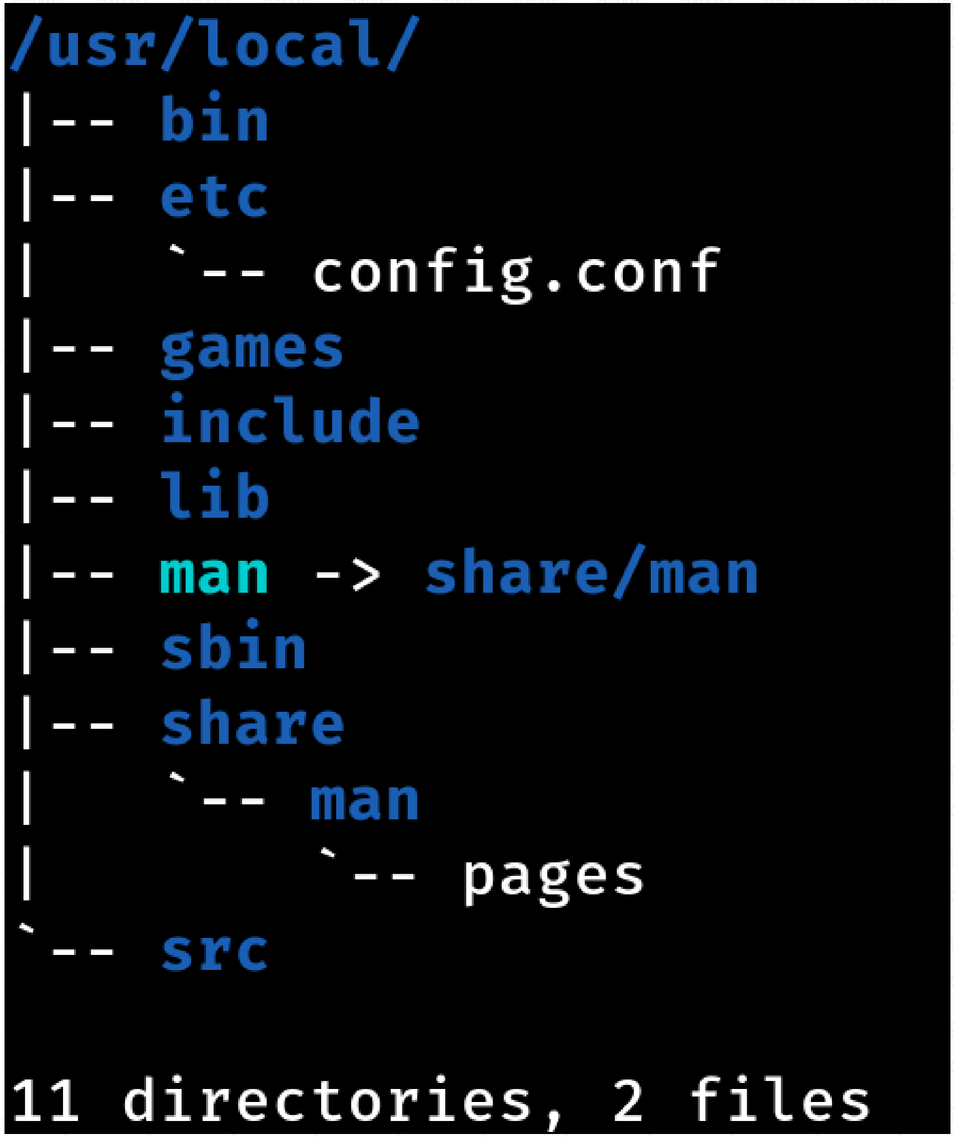
- Extension can be completely omitted (e.g. *textfile* vs *textfile.txt*)
- Regular files: plain text or raw binary (data, image, video, pdf, document etc.)
- Directory („folder“) is another type of file
- Disk (block device) is represented as a file
- Character devices – e.g. screen of terminal - are files too
- There are also pipes, sockets and other special files

# Directory (1/2)

- Directory is a type of file where „*data*“ is a **list** of files – directory „*contains*“ files
- It enables organizing files in a tree-like structure
- On *NIX system the forward slash “/” will signal that the file is directory, e.g.: *projects/*
- Directories can be nested – tree-like **hierarchy**
- Each directory in the “**path**” is separated by “/”

# Directory (2/2)

- Example of a directory structure in „*/usr/local*“
- Directory can contains regular files and other directories
- */usr/local/etc/config.conf* is a full file path
- */usr/* is a top-level directory while *local/* is subdirectory



```
/usr/local/
|-- bin
|-- etc
|   `-- config.conf
|-- games
|-- include
|-- lib
|-- man -> share/man
|-- sbin
|-- share
|   `-- man
|       `-- pages
`-- src

11 directories, 2 files
```

# Path (1/2)

- Each file has a name which must be unique inside a directory
- There could be two or more files of the same name if they differ in the path
- Path is a composite of a file name and directories in which it resides ("*/usr/local/etc/*" **+** "*config.conf*")
- E.g. these directories might exist simultaneously: */bin/*, */usr/bin/* and */usr/local/bin/*

# Path (2/2)

- Any path starting with slash “/” is so-called **absolute path**
- Single slash is also a path - it is absolute path of the **root** of the directory tree
- Path which does not start with slash is called **relative path**
- Relative path is relative to your **current directory**

# Filesystem (1/4)

- **Filesystem is a system how to organize and store files** (different OS will have different “**native**” FS – most of OS will support plethora of different filesystems)
- Simple filesystem can be a **FAT** (File Allocation Table) from DOS - it is primitive and limited (easily fragmented, maximum size is 4GB per file, etc.)
- **exFAT** is modern replacement and suited for SD cards and flash drives
- **NTFS** on Windows is advanced **journaling** FS with a feature list similar to Linux and other modern OS (like encryption, compression, acls, etc.)
- *Linux is capable of reading and writing to Windows/DOS FS but these do not support all *NIX metadata (e.g. ownerships and permissions)*

# Filesystem (2/4)

- True *NIX FS can store full file permissions and metadata needed for proper function of any UNIX-like OS
- Even amongst *NIX OSes there are many vastly different types (ext2-4, zfs, btrfs, hammerfs etc.)
- Most of them will have **journal** (all writes are logged in a journal for consistency and resilience)
- ZFS (ported to Linux) and btrfs (native to Linux) are more than just FS: they provide features not seen elsewhere (copy-on-write, effective cheap snapshots, extra protection against bit-rot with checksumming, volumes, raids, encryption, compression etc.)

# Filesystem (3/4)

- Running OS can have deployed (mounted) multiple FS at the same time
- UNIX had unified look on all its files via **hierarchical tree-like filesystem structure**
- All FS are mounted under this tree structure as directory (so no letters C:, D:, E:,… which is on Windows)
- **Files are flat** (just bytes), **ordinary text is primary data type** (anyone can open, read and modify configuration with a text editor, log files are also simple text files – or they should be…, most tools will process plain text)

# Filesystem (4/4)

- All *NIX FS (despite differences) share the concept of **inode** - which is structure holding many information about a file
- Every (*NIX) FS has finite amount of inodes - new file will allocate free inode from the pool and “deleted” file will deallocate inode back
- Inode has unique identifier inside the FS and it also contains metadata like timestamps, ownership, permissions and **link count**
- Finally it holds the information where the actual **file data** are stored in the filesystem (pixel bytes of a JPEG for example)

# Partitions

- Storage (disks, block devices) can be partitioned into multiple **partitions**
- To be able locate and use partitions a **partition table** must be created on the disk
- **MBR** - Master Boot Record partition table is an old scheme from DOS era and still commonly in use - all table records are stored at the beginning of the disk and take less than 512 Bytes
- Modern **GPT** – GUID Partition Table is more flexible and resilient (it has duplicated structures and not just in first 512 Bytes)

# Mountpoints

- Filesystems are created on **block devices** and those can be whole disks or partitions - FS is then **mounted** in the running OS (onto directory)
- There can be multiple FS at once - all filesystems are mounted somewhere under the **root filesystem** (“/”), e.g.:

```
mount -t btrfs -o subvol=/home,... /dev/sdb /home
         ^FS      ^options           ^disk    ^mountpoint
```

- Example of all mounted FS in the OS:

```
/dev/sda1 on /boot type ext4 (rw,relatime,seclabel)
/dev/sda2 on / type btrfs (rw,seclabel,...,subvol=/root)
/dev/sdb on /home type btrfs (rw,...,subvol=/home)
```

# Inode and hardlinks

- Inode does not contain filename - filenames are stored as data of a directory inode (which “contains” the files)
- The inode keep tracks of **links** - these links are called **hardlinks**
- If file has multiple links then it only means that it is available under multiple names (in the same FS) - **data are stored only once**
- File is deleted from storage only if link count drops to zero – **inode is freed**
- Hardlinked file will share the same underlying inode which means that changing data on a file will also alter content of all its hardlinked “*copies*”

# Special directory hardlinks

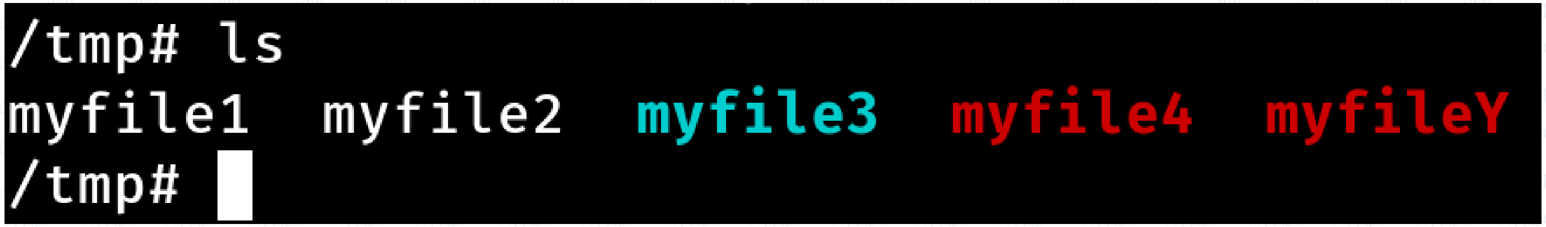
- Hardlinks cannot cross filesystem boundary and they cannot be used on directories to avoid loops in directory tree
- **Nevertheless** every directory has inside two special hardlinks:
  single **dot** (“./”) and **two-dots** (“../”) links
- One-dot hardlink references the directory itself
- Two-dots hardlink references parent directory (one level up in the path)

# Symlinks

- Symlink is a **symbolic link** - it will **not** raise link count in the inode
- Symlink is just another type of file - its data is a file path – even across FS boundary, directory or even to itself (useless though – cyclic reference)
- It is similar to URL link but for files - it also can be stale and point to nothing – **broken link**
- Windows has similar feature - symlinks to folders or applications

# Little bit of shell: **ls** (1/2)

- The following examples will be using the output of command „`ls`“ (list directory contents)
- Usage without any argument:

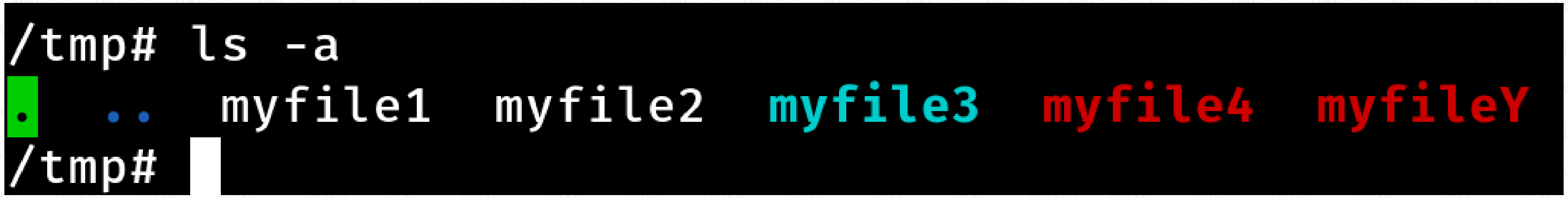


```
/tmp# ls
myfile1  myfile2  myfile3  myfile4  myfileY
/tmp# 
```

- Adding argument “`-l`” will create more descriptive long listing
- Adding argument „`-i`“ will add column with inode numbers
- So we can use it as: “`ls -l -i`“ or shorten it to: “`ls -li`“

# Little bit of shell: **ls** (2/2)

- There is a convention on *NIX systems where files starting with a dot „**.**“ are „**hidden**“
- These are so called **dot-files** (do not confuse with hidden files in Windows – very different semantics)
- And so “`ls`” command will not show them but adding argument “`-a`” will tell the “`ls`” command to show **all** files:



```
/tmp# ls -a
.   ..   myfile1  myfile2  myfile3  myfile4  myfileY
/tmp#
```

- There is more happening behind the scene – the colorful output is achieved with more trickery by **aliasing** the `ls` command – aliases are out of scope for the moment
- All put together we can use the following to get the final outputs:
  “`ls --color=auto -aliF`”

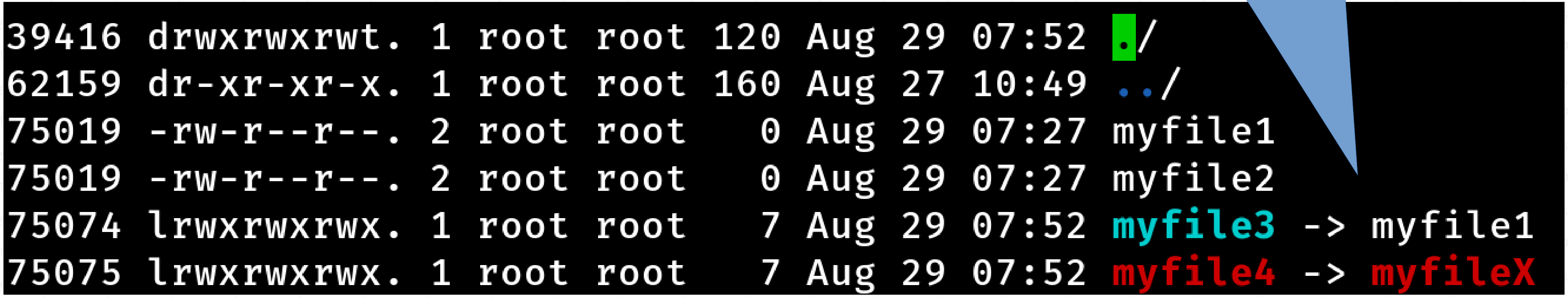
# Example (1/2)

Inode numbers:

Notice: *myfile1* and *myfile2* have the same inode number!

Links:

- *myfile1* is a regular file
- *myfile2* was created as a hardlink
- *myfile3* is a symlink to *myfile1*
- *myfile4* is a broken symlink (to non-existent target)



```
39416 drwxrwxrwt. 1 root root 120 Aug 29 07:52 ./
62159 dr-xr-xr-x. 1 root root 160 Aug 27 10:49 ../
75019 -rw-r--r--. 2 root root   0 Aug 29 07:27 myfile1
75019 -rw-r--r--. 2 root root   0 Aug 29 07:27 myfile2
75074 lrwxrwxrwx. 1 root root   7 Aug 29 07:52 myfile3 -> myfile1
75075 lrwxrwxrwx. 1 root root   7 Aug 29 07:52 myfile4 -> myfileX
```

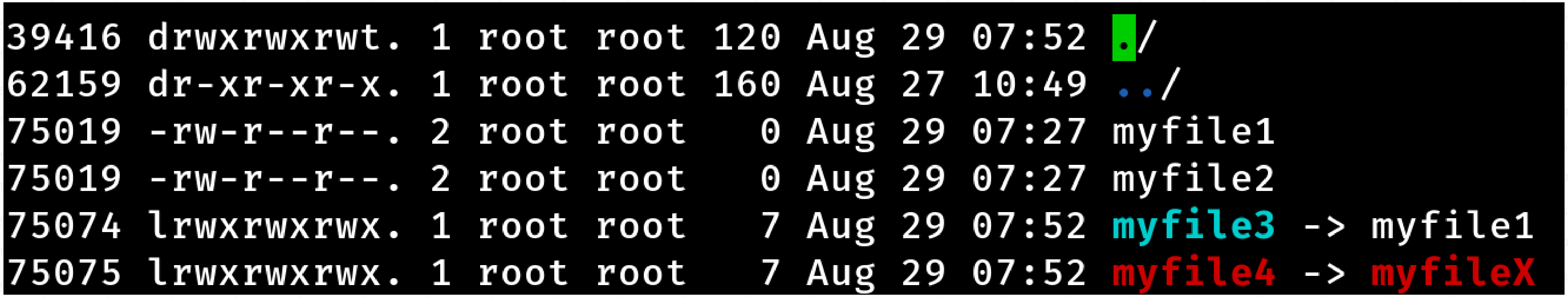
# Example (2/2)

This is "**permissions**" column (explained in a moment) but notice the first letter per every line – that denotes the **type of a file**:
„**d**" means directory, „**-**" is regular file, „**l**" for symlink, „**b**" block device, „**c**" special device, „**p**" for pipe, „**s**" for socket

Special directory hardlinks:

- Single dot (pointing to itself)
- Two-dots (pointing one level up)

Notice: *Both are appended with a slash „/" - meaning - they are directories (links to directories)*



```
39416 drwxrwxrwt. 1 root root 120 Aug 29 07:52 ./
62159 dr-xr-xr-x. 1 root root 160 Aug 27 10:49 ../
75019 -rw-r--r--. 2 root root   0 Aug 29 07:27 myfile1
75019 -rw-r--r--. 2 root root   0 Aug 29 07:27 myfile2
75074 lrwxrwxrwx. 1 root root   7 Aug 29 07:52 myfile3 -> myfile1
75075 lrwxrwxrwx. 1 root root   7 Aug 29 07:52 myfile4 -> myfileX
```

# File Ownership

- All files have **owner** and belong to a **group**
- Owner is “usually” a user on the system – but in terms of file ownership it is just an **UID** number
- UID stands for User Identification
- Some UIDs might not be actually assigned to any user on the system – so an UID number will be shown instead of a name
- Same applies to groups and **GID** – Group Identification

# Briefly on Users & Groups

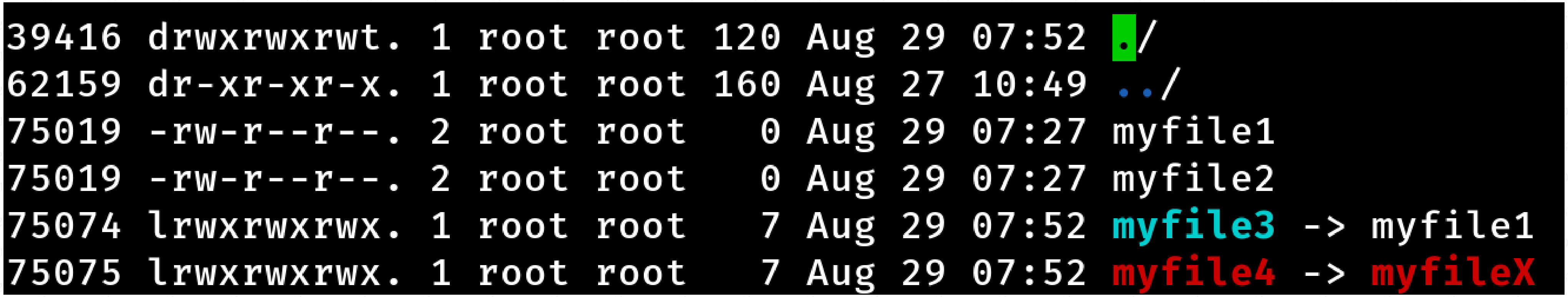
- *NIX systems are **multi-user** environments and both files and processes are **insulated** per their user and group membership
- Users cannot overwrite each other’s files unless they explicitly allow it
- User with UID zero is **root** – superuser – it has unlimited access to the OS – *it can do anything!*
- User accounts are stored in */etc/passwd* file (passwords are in */etc/shadow*)
- Groups are in */etc/group* (rare group passwords are in */etc/gshadow*)
- CLI utils for reference: `useradd/adduser`, `groupadd`, `passwd`, `gpasswd`, `usermod`, `groupmod`, `su`, `sudo` and others

# Example

Ownership – two columns:

1. column is user name (or UID)
2. column is group name (or GID)

*In this case the owner is **root** and files belong to group named also **root**.*



```
39416 drwxrwxrwt. 1 root root 120 Aug 29 07:52 ./
62159 dr-xr-xr-x. 1 root root 160 Aug 27 10:49 ../
75019 -rw-r--r--. 2 root root   0 Aug 29 07:27 myfile1
75019 -rw-r--r--. 2 root root   0 Aug 29 07:27 myfile2
75074 lrwxrwxrwx. 1 root root   7 Aug 29 07:52 myfile3 -> myfile1
75075 lrwxrwxrwx. 1 root root   7 Aug 29 07:52 myfile4 -> myfileX
```

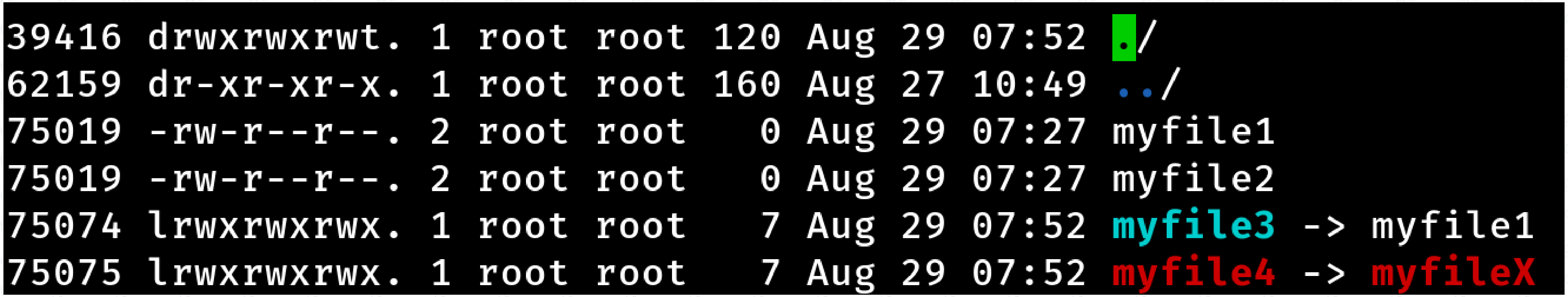
# Little bit of shell: **useradd**

- Creating new user named „*friday*“

```
useradd -md /home/friday -g users -s /bin/bash --uid 1001 friday
```

• `useradd`	← command name
`-md /home/friday`	← creating home dir
`-g users`	← adding to group “users”
`-s /bin/bash`	← default shell for the user
`--uid 1001`	← setting UID number
`friday`	← actual username

# Little bit of shell: **chown**



```
39416 drwxrwxrwt. 1 root root 120 Aug 29 07:52 ./
62159 dr-xr-xr-x. 1 root root 160 Aug 27 10:49 ../
75019 -rw-r--r--. 2 root root   0 Aug 29 07:27 myfile1
75019 -rw-r--r--. 2 root root   0 Aug 29 07:27 myfile2
75074 lrwxrwxrwx. 1 root root   7 Aug 29 07:52 myfile3 -> myfile1
75075 lrwxrwxrwx. 1 root root   7 Aug 29 07:52 myfile4 -> myfileX
```

- Within the directory in the picture above we run the following:

`chown friday myfile1` ← changing the UID only

`chown -h friday:users myfile4` ← changing UID and GID both

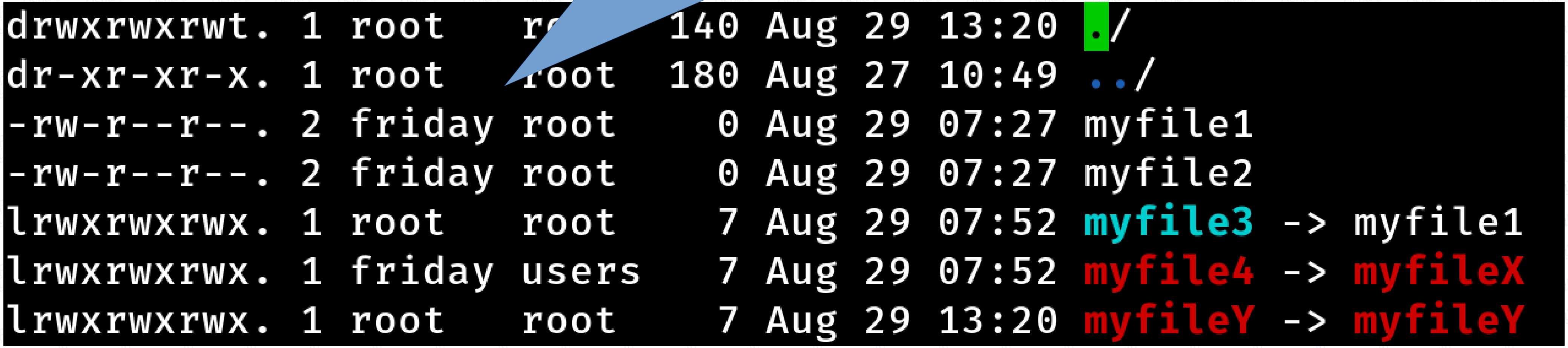
Notice: We need to use *-h* to **not** dereference because *myfile4* is broken symlink

# Result

New ownerships:

- *myfile1* belongs to user “*friday*” now
- *myfile2* was affected too because it is just hardlink
- *myfile4* belongs to user “*friday*” and also group “*users*”

Without “**-h**” option the second **chown** command would fail because it would try to change ownerships on nonexistent *myfileX*...



```
drwxrwxrwt. 1 root    ro       140 Aug 29 13:20 ./
dr-xr-xr-x. 1 root     root    180 Aug 27 10:49 ../
-rw-r--r--. 2 friday root        0 Aug 29 07:27 myfile1
-rw-r--r--. 2 friday root        0 Aug 29 07:27 myfile2
lrwxrwxrwx. 1 root    root       7 Aug 29 07:52 myfile3 -> myfile1
lrwxrwxrwx. 1 friday users       7 Aug 29 07:52 myfile4 -> myfileX
lrwxrwxrwx. 1 root    root       7 Aug 29 13:20 myfileY -> myfileY
```

# File Permissions (1/9)

- Traditional UNIX file permissions have three separate categories of access: **user (u)**, **group (g)** and **others (o)**
- Each category has the same scheme of permissions: **read (r)**, **write (w)** and **execution (x)**
- In `ls` command we can see them as "**---**" up to "**rwx**" and anything in between
- They are actually implemented with **three bits** each - number two (0 or 1 per bit) on power of three (bits) gives eight possible values **(2^3 == 8)**
- This means that we can have something like this: "**`rw-rw-r--`**" or "**`rw-------`**" where every three letters represent user, group and other permissions respectively
- The letters can be replaced with **octal** values (0-7 for each three bits)

# File Permissions (2/9)

- We can visualize the values of bits like this:

```
BIN    000 001 010 011 100 101 110 111
SYM    --- --x -w- -wx r-- r-x rw- rwx
OCT      0   1   2   3   4   5   6   7
```

- First row shows values in binary per each bit
- Second row shows the mnemonic representation
- Third row is in octal base – for those used to the numeric representation it is often more quick and pleasant to use the octals, e.g.: instead of **rwxrwxrwx** we use **777**

# File Permissions (3/9)

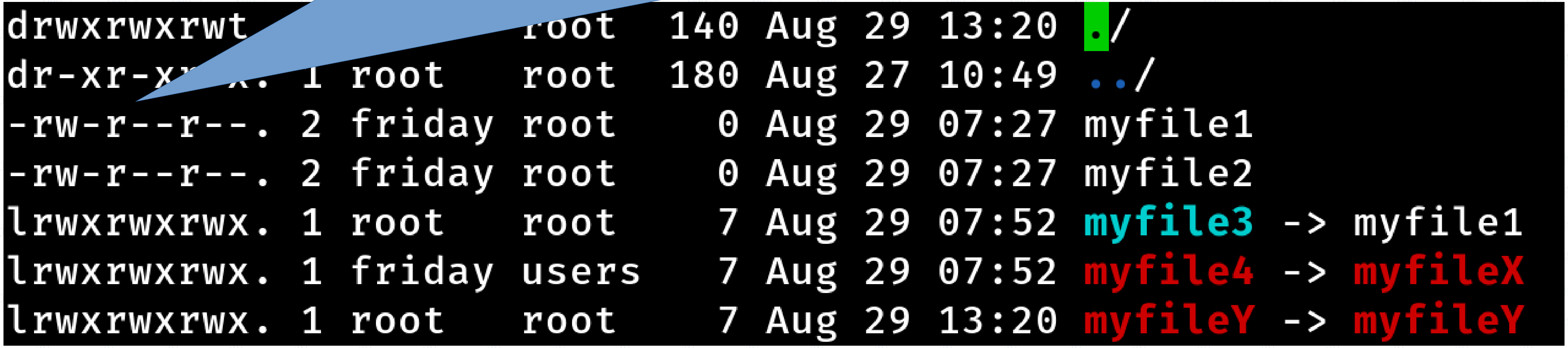
- Permission rights have slight semantic difference for directories and non-directories (not really if we realize that directory is a file...)
- **Read right** enables the user (group or others) to read the content of a regular file or list the content of a directory
- *The ability to read file has consequence of being able to create a copy of the file and change the permissions on said copy...*
- **Write right** enables modification of the content of a file (truncating, appending, rewriting etc.) and its metadata (e.g. permissions...)
- *Write right on a directory let you write (create) new files inside a directory or delete them (even if they don’t belong to the owner of directory and regardless of permissions on those files themselves)*

# File Permissions (4/9): Example

Permissions:

- *myfile1* has read and write rights for user but only read rights for group and others
- *myfile3* is symlink and those are usually “pass-through” so with full rights but the referenced file will preserve its actually permissions as-is!

So in this case only user can write to *myfile3* despite symlink is showing write bits on group and others (because the referenced file is myfile1…).



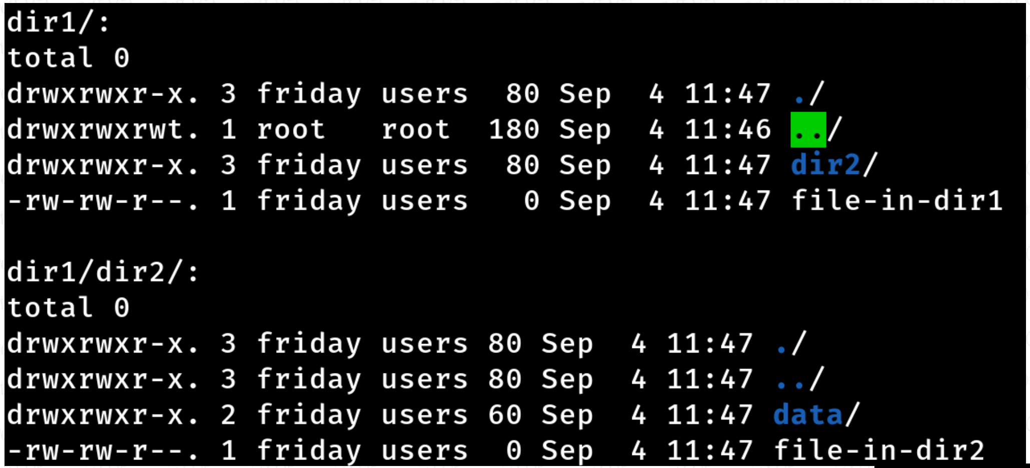
```
drwxrwxrwt                    root   140 Aug 29 13:20 ./
dr-xr-x      x. 1 root        root   180 Aug 27 10:49 ../
-rw-r--r--. 2 friday root       0 Aug 29 07:27 myfile1
-rw-r--r--. 2 friday root       0 Aug 29 07:27 myfile2
lrwxrwxrwx. 1 root   root       7 Aug 29 07:52 myfile3 -> myfile1
lrwxrwxrwx. 1 friday users      7 Aug 29 07:52 myfile4 -> myfileX
lrwxrwxrwx. 1 root   root       7 Aug 29 13:20 myfileY -> myfileY
```

# File Permissions (5/9)

- **Execution right** has specific functionality:
  - on a **regular file**: it tells the operating system that this file can be executed (e.g. program to run)
  - on a **directory**: it enables the directory to be „**traversed**“ – to be entered and reach its **subdirectories**

    e.g.: path **`/dir1/dir2/data/`** where `/dir1/` and `dir2/` have both permissions set to `--x--x--x` (111) – we cannot list contents of either `dir1` nor `dir2` but if we know that `data/` directory is buried there then we can reach it and actually list its content if `data/` has read permissions

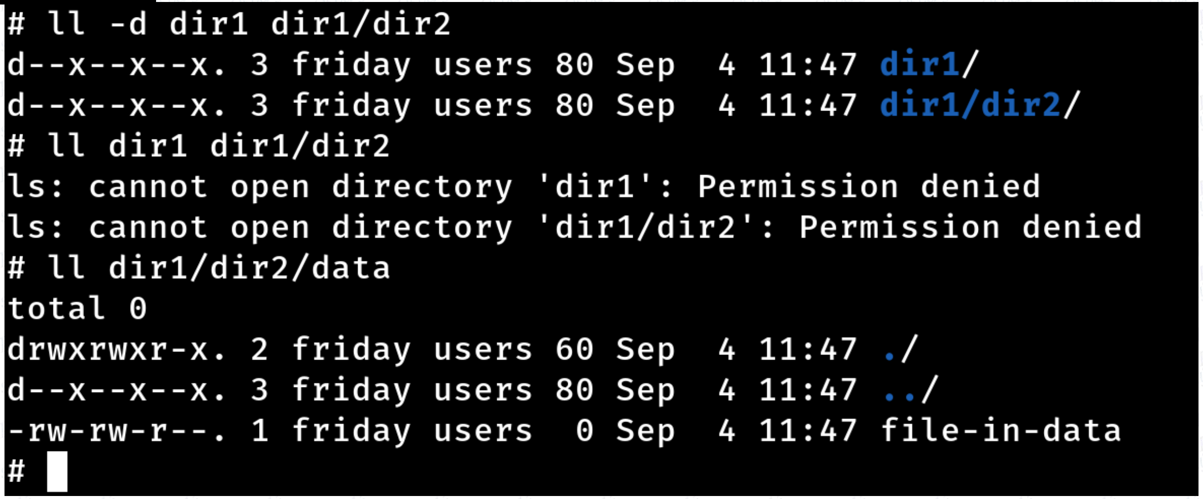
# File Permissions (6/9): Example



```
dir1/:
total 0
drwxrwxr-x. 3 friday users   80 Sep  4 11:47 ./
drwxrwxrwt. 1 root   root   180 Sep  4 11:46 ../
drwxrwxr-x. 3 friday users   80 Sep  4 11:47 dir2/
-rw-rw-r--. 1 friday users    0 Sep  4 11:47 file-in-dir1

dir1/dir2/:
total 0
drwxrwxr-x. 3 friday users 80 Sep  4 11:47 ./
drwxrwxr-x. 3 friday users 80 Sep  4 11:47 ../
drwxrwxr-x. 2 friday users 60 Sep  4 11:47 data/
-rw-rw-r--. 1 friday users  0 Sep  4 11:47 file-in-dir2
```

Content of ***dir1/*** - notice the permissions of *dir2/* – they are **775** – both read and execute rights (same is for *dir1/* at this moment)

Content of ***dir2/*** - notice the ***data/*** subdirectory



```
# ll -d dir1 dir1/dir2
d--x--x--x. 3 friday users 80 Sep  4 11:47 dir1/
d--x--x--x. 3 friday users 80 Sep  4 11:47 dir1/dir2/
# ll dir1 dir1/dir2
ls: cannot open directory 'dir1': Permission denied
ls: cannot open directory 'dir1/dir2': Permission denied
# ll dir1/dir2/data
total 0
drwxrwxr-x. 2 friday users 60 Sep  4 11:47 ./
d--x--x--x. 3 friday users 80 Sep  4 11:47 ../
-rw-rw-r--. 1 friday users  0 Sep  4 11:47 file-in-data
# 
```

Here we did set both *dir1/* and *dir2/* permissions as **111**

ls fails to read their content because read permission is missing...

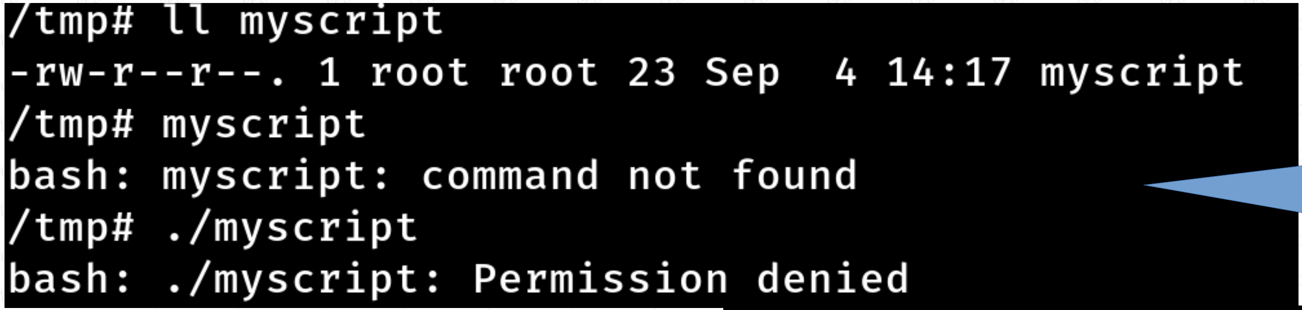
…but we can still read content of ***data/*** directory

# File Permissions (7/9): Example

- Let’s create <u>very</u> simple shell script:

```
#!/bin/sh
echo SUCCESS
```

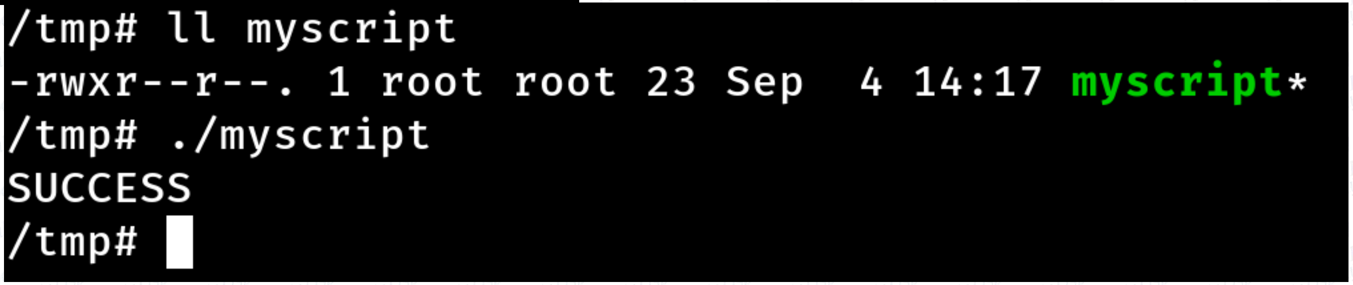
Saved as ***/tmp/myscript***



```
/tmp# ll myscript
-rw-r--r--. 1 root root 23 Sep  4 14:17 myscript
/tmp# myscript
bash: myscript: command not found
/tmp# ./myscript
bash: ./myscript: Permission denied
```

The execution failed twice:

1. Shell tried to find the command in its ***PATH*** (off-topic)
2. Missing execution bit (on-topic)



```
/tmp# ll myscript
-rwxr--r--. 1 root root 23 Sep  4 14:17 myscript*
/tmp# ./myscript
SUCCESS
/tmp# █
```

Added execution right let us run the script successfully („**./**“ is needed unless ***PATH*** is modified – see later)

# File Permissions (8/9)

- We can set **special permissions** on the files which uses **another three bits** of information (they go before the regular permissions bits, e.g. 1777 where 1 is „**sticky bit**“ – one of the special permissions)
- The bits and symbol representation (usually only one of the bit is used on a file):

`BIN`	`000`	`001`	`010`	`100`
`SYM`	`-`	`t`	`Sg`	`s`
`OCT`	`0`	`1`	`2`	`4`

  and where:
  - `t:` stands for **sticky bit** (this on a directory prevents non-owners to delete files inside)
  - `s:` **setuid** (a bit that makes an executable run with the privileges of the owner of the file)
  - `Sg:` **setgid** (same but with the privileges of the group of the file – explored elsewhere)

Flipping on special permissions will make symbol representation less neat, e.g. **1777** will be **rwxrwxrwt** instead of *trwxrwxrwx* in the **ls** output

- In numeric mode they can be written as a single digit between **0-7** (as with regular permissions) – or omitted (as they usually are) – bit combinations are rare

# File Permissions (9/9)

- Let’s skip **setuid** and **setgid** for the moment (it will be covered elsewhere)
- **Sticky bit** is common and used on “globally” writable directories like **/tmp/** where it prevents *unprivileged* users (non-root) from removing or renaming any file which does not belong to them
- **/tmp/** is the standard place of **system-wide temporary directory**
- Temporary directories are used for *throwaway* files which are needed during **lifetime** of running process (e.g. file locks) or the system and for all sorts of short-term usage, but which **do not** need to **survive reboot**
- **Temporary directories start empty and clean** upon system reboot and there is more than one usually – sprinkled whenever there is a need for them – modern Linux will usually mount them with a <u>special memory filesystem</u>: **tmpfs** (e.g. here: /tmp/, /dev/shm/, /run/, /root/tmp/...)

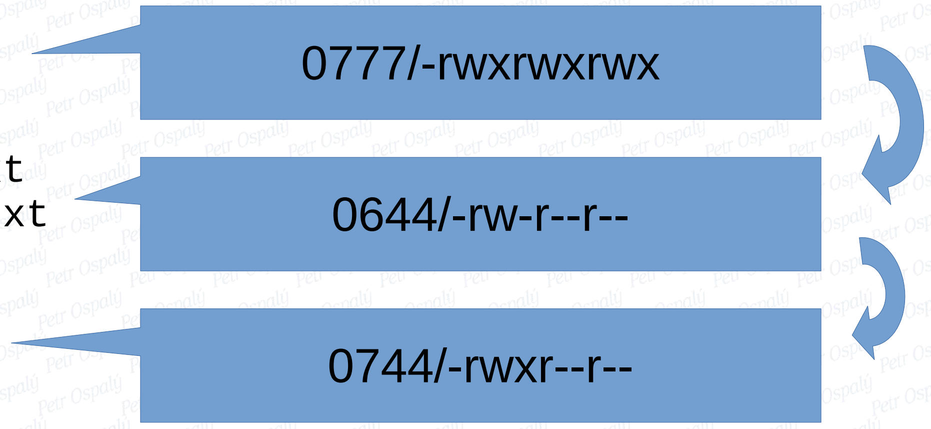
# Little bit of shell: **chmod**

- Command **chmod** serves to set and modify the file permissions – it can have rather complex usage in comparison with commands mentioned
- There are two main modes often used:
  - `chmod [ugoa][+-=][rwxXst],... myfile.txt` → uses symbols and enables updates
  - `chmod [0-7]... myfile.txt` → uses numerics and sets all at once
- Best is to see on examples (every bullet has equivalent alternatives):

  -
    ```
    chmod a=rwx file.txt
    chmod ugo=rwx file.txt
    chmod 777 file.txt
    ```

  -
    ```
    chmod go-w,a-x file.txt
    chmod go-wx,u-x file.txt
    chmod 644 file.txt
    ```

  -
    ```
    chmod u+x file.txt
    chmod 744 file.txt
    ```



# Little bit of shell: **umask**

- When file is created – the program decides on its permissions (usually **0666**) but before the file is actually committed to FS an **umask** (*file mode creation mask*) comes to play…
- **umask**: it is command which shows or set “umask” for your default permissions (e.g. **0022**):
  `BIN 111 111 111 111` (full permissions **7777**)
  `BIN 000 000 010 010` (umask **0022** – notice only two bits are true… writes for non-user)
  `BIN 111 111 101 101` (negated mask **7755**, logical **XOR**)
  Finally we apply the negated mask to the actual file mode wanted by program (e.g. 0666) by applying logical operation **AND**:
  `BIN 000 110 110 110` (program creation mode **0666**)
  `BIN 111 111 101 101` (negated/complement mask **7755**)
  `BIN 000 110 100 100` (result after AND: **0644**)

  umask (e.g. 0022) is basically a filter which filters out certain bits...
- *Usually OS will prevent programs to set execution bits on file creation despite umask values*

# Significant directories

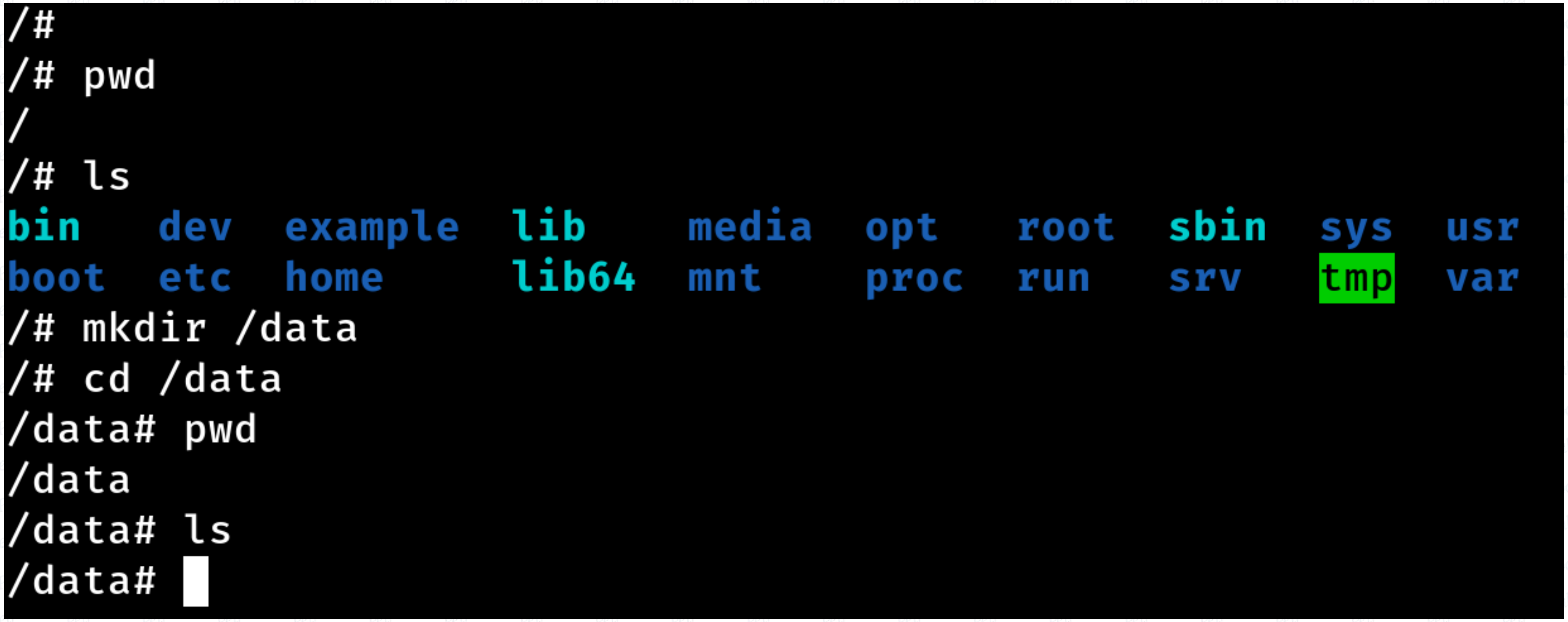
- **Linux Filesystem Hierarchy Standard (FHS)**
- **/boot** – bootloader files, kernel image, initial ramdisk and other needed for boot
- **/dev** – files representing devices
- **/proc** – virtual filesystem, user space interface with Linux kernel and processes
- **/bin, /sbin** – places with important binaries (programs) often used early in boot process
- **/lib, /lib64** – shared libraries
- **/etc** – canonically a place where all configuration is stored
- **/home** – usual path for user „*home*“ directories
- **/root** – home directory of superuser
- **/tmp** – temporary files
- **/usr** – majority of installed software will be here (/usr/bin, /usr/sbin, /usr/lib…)
- **/usr/local, /opt** – reserved for machine-specific custom software installations
- **/var, /var/log** – place for „*variable*“ data, e.g. log and database files

# Shell commands so far...

- We came across quite a few CLI tools during this presentation:
  - `ls, useradd, chown, chmod, umask`
  - some mentioned only for reference (more on them elsewhere)
- We will need much more though to complete our excursion to files and filesystems:
  - `cd, pwd, mkdir, touch, rm, rmdir, ln, stat, unlink, cp, mv`
- Luckily they are simple to use and understand

# Shell basics (1/10)

- Take a look at this CLI session and see if you can make sense of it (explanation will be on the next slide)

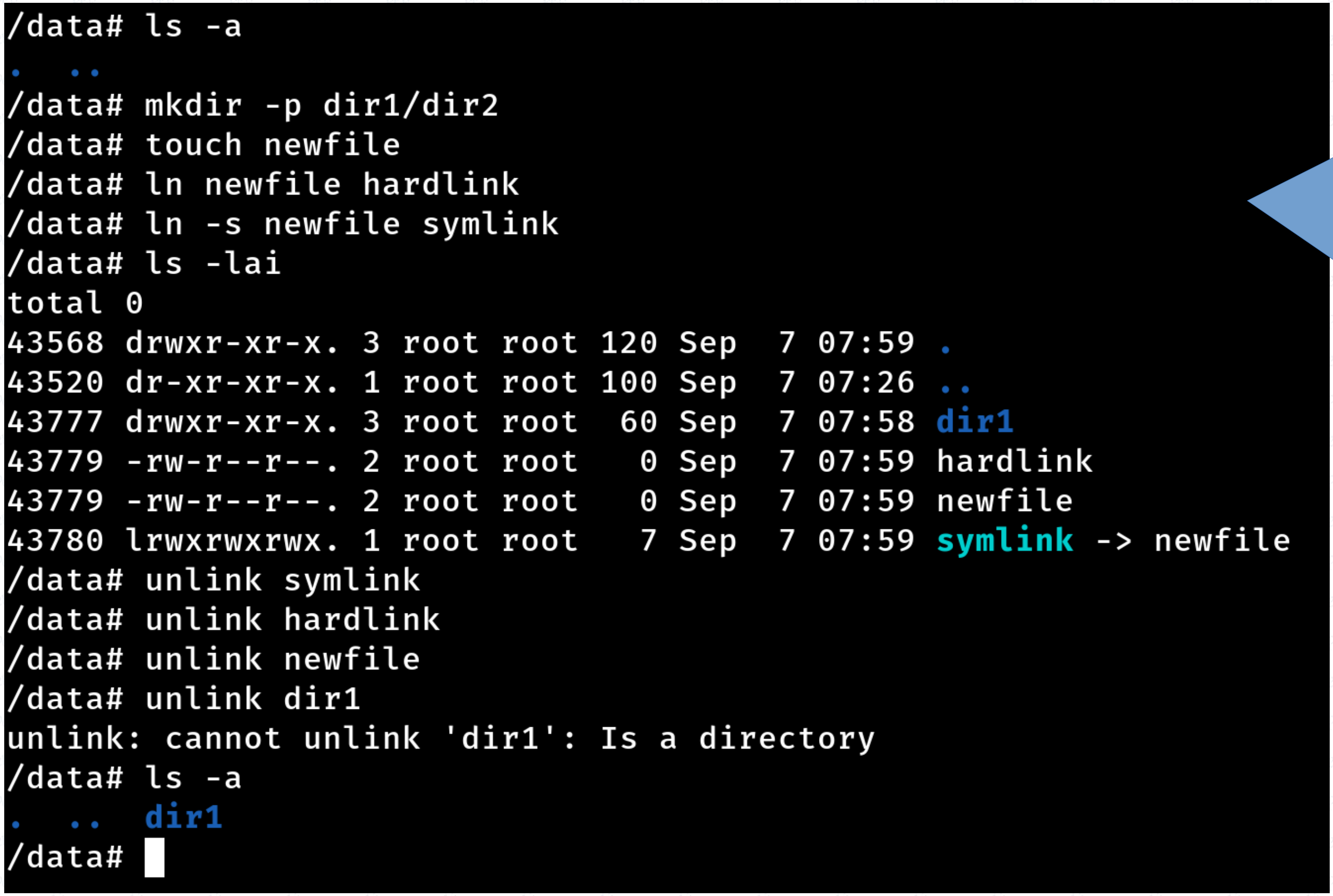


```
/#
/# pwd
/
/# ls
bin    dev   example  lib    media  opt   root  sbin  sys  usr
boot   etc   home     lib64  mnt    proc  run   srv   tmp  var
/# mkdir /data
/# cd /data
/data# pwd
/data
/data# ls
/data# 
```

# Shell basics (2/10)

- A file „name“ will always be either relative or absolute pathname (e.g.: *dir1/file.txt*, *file.txt*, *../file.txt*, */file.txt*, etc.)
- **pwd**: prints current directory (where you are located)
- **mkdir**: create new directory (argument is a directory name; there is useful option: “**`-p`**“ which will create all intermediary dirs)
- **cd**: change directory (argument is directory path – relative or absolute)
- **ls**: This command was covered already but let’s remind some useful options: **`-l`** (long listing), **`-i`** (show inodes), **`-1`** (output in columns), **`-a`** (show hidden dot-files)

# Shell basics (3/10)



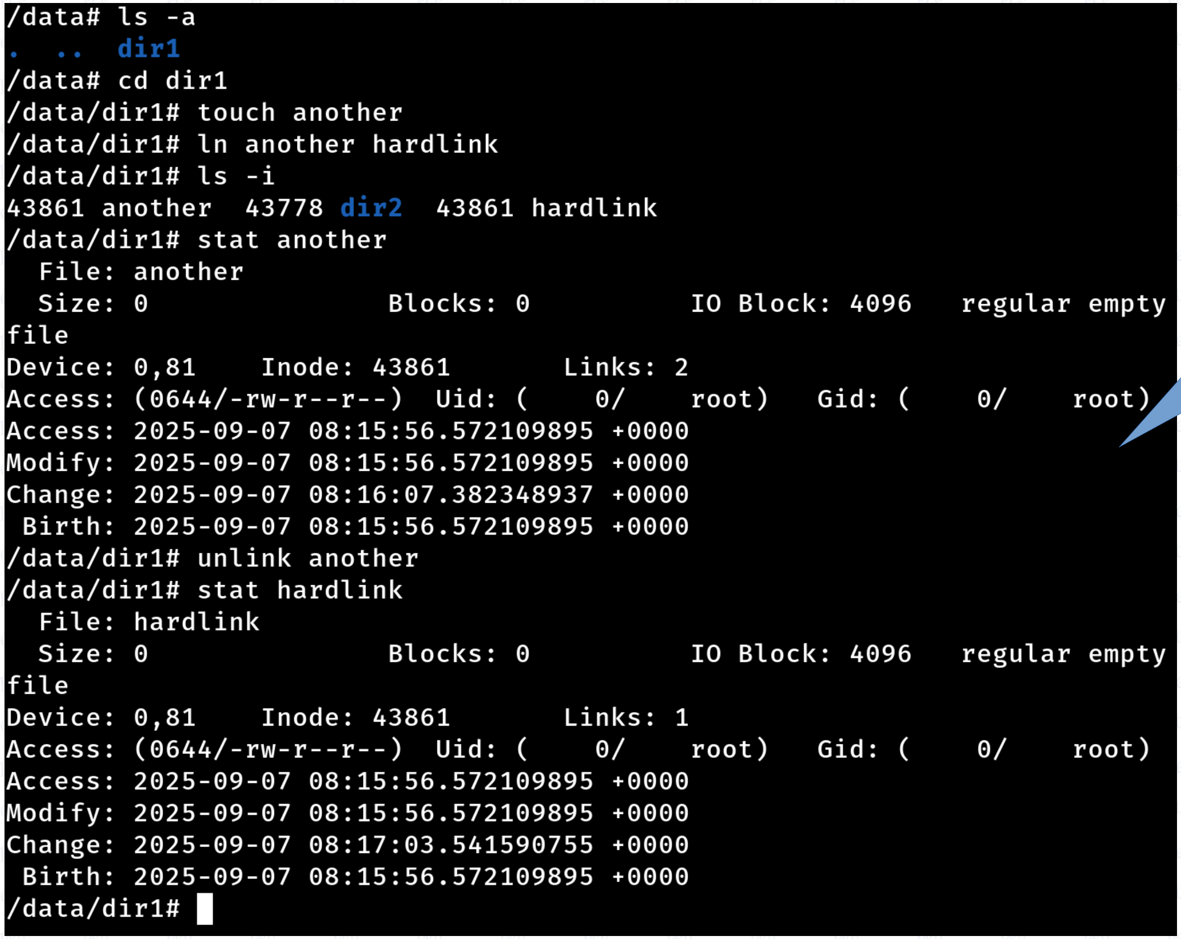
```
/data# ls -a
.  ..
/data# mkdir -p dir1/dir2
/data# touch newfile
/data# ln newfile hardlink
/data# ln -s newfile symlink
/data# ls -lai
total 0
43568 drwxr-xr-x. 3 root root 120 Sep  7 07:59 .
43520 dr-xr-xr-x. 1 root root 100 Sep  7 07:26 ..
43777 drwxr-xr-x. 3 root root  60 Sep  7 07:58 dir1
43779 -rw-r--r--. 2 root root   0 Sep  7 07:59 hardlink
43779 -rw-r--r--. 2 root root   0 Sep  7 07:59 newfile
43780 lrwxrwxrwx. 1 root root   7 Sep  7 07:59 symlink -> newfile
/data# unlink symlink
/data# unlink hardlink
/data# unlink newfile
/data# unlink dir1
unlink: cannot unlink 'dir1': Is a directory
/data# ls -a
.  ..  dir1
/data# 
```

1. creating directory structure *dir1/dir2*
2. creating file *„newfile“*
3. creating hardlink and symlink with **ln** command
4. removing all new files with **unlink** command
5. failing to remove directory with **unlink**

# Shell basics (4/10)

- **touch**: simple command to create empty files but also to "touch" existing file and change their access and modification times in their metadata
- **ln**: This command without an option will create **hardlink**:
```
ln <original file> <hardlink name>
```
  or **symlink** with the **-s** option:
```
ln -s <original file> <symlink name>
```
- **unlink**: Removes the file link – which will reduce "link" count from inode – so effectively removes the file link – it cannot unlink directories...

# Shell basics (5/10)



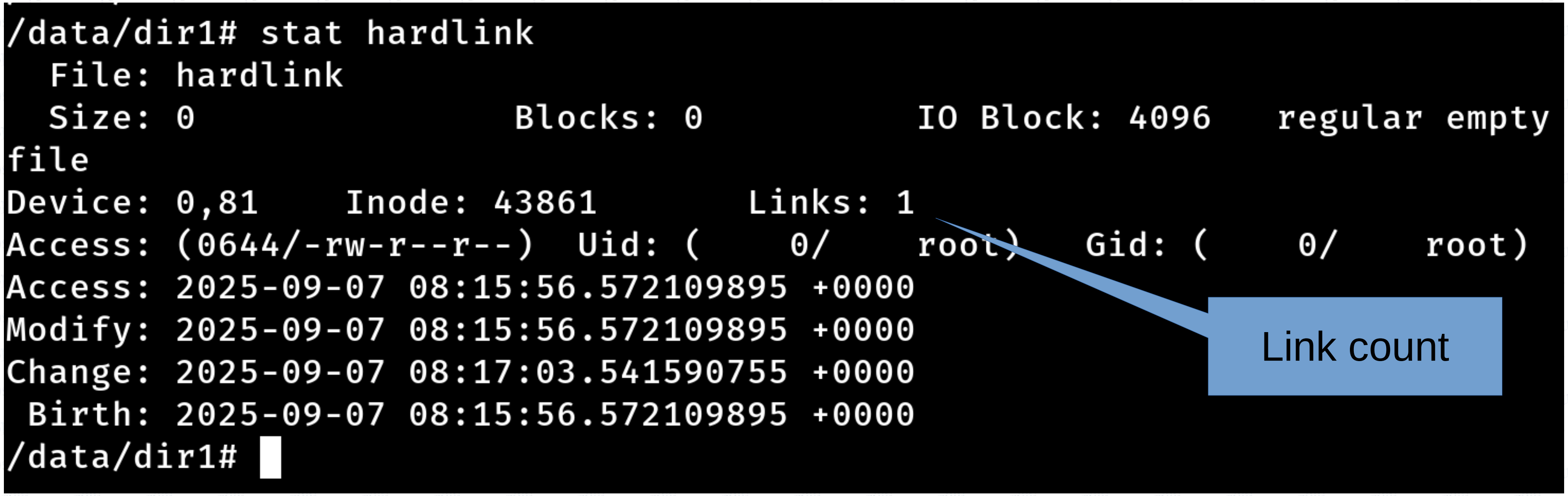
```
/data# ls -a
.  ..  dir1
/data# cd dir1
/data/dir1# touch another
/data/dir1# ln another hardlink
/data/dir1# ls -i
43861 another  43778 dir2  43861 hardlink
/data/dir1# stat another
  File: another
  Size: 0               Blocks: 0          IO Block: 4096   regular empty
file
Device: 0,81    Inode: 43861       Links: 2
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2025-09-07 08:15:56.572109895 +0000
Modify: 2025-09-07 08:15:56.572109895 +0000
Change: 2025-09-07 08:16:07.382348937 +0000
 Birth: 2025-09-07 08:15:56.572109895 +0000
/data/dir1# unlink another
/data/dir1# stat hardlink
  File: hardlink
  Size: 0               Blocks: 0          IO Block: 4096   regular empty
file
Device: 0,81    Inode: 43861       Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2025-09-07 08:15:56.572109895 +0000
Modify: 2025-09-07 08:15:56.572109895 +0000
Change: 2025-09-07 08:17:03.541590755 +0000
 Birth: 2025-09-07 08:15:56.572109895 +0000
/data/dir1# 
```

Command **stat** will show you detailed metadata information on a given file (actually the inode) – notice the **timestamps** and **links count** (showing two because of hardlink)

We deleted the „original“ file and used **stat** command on its **hardlink** – as you can see **links count dropped to one**, also modification timestamp changed but otherwise it is the same – because it is the same inode...

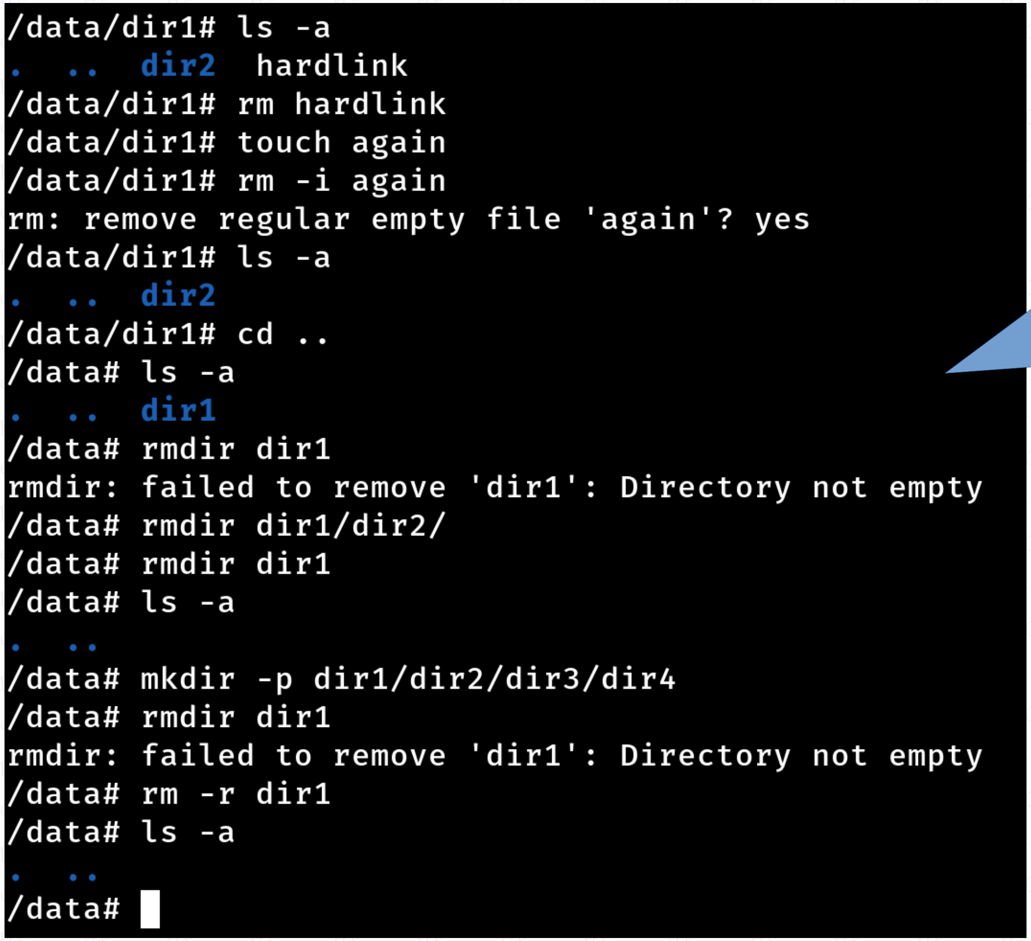
# Shell basics (6/10)

- **stat**: it will show inode metadata in full on a given file name (file link – same thing); the option “**-f**” will show information on filesystem where file is stored (quick way to find FS name)



```
/data/dir1# stat hardlink
  File: hardlink
  Size: 0               Blocks: 0          IO Block: 4096   regular empty
file
Device: 0,81    Inode: 43861       Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2025-09-07 08:15:56.572109895 +0000
Modify: 2025-09-07 08:15:56.572109895 +0000
Change: 2025-09-07 08:17:03.541590755 +0000
 Birth: 2025-09-07 08:15:56.572109895 +0000
/data/dir1# 
```

# Shell basics (7/10)



```
/data/dir1# ls -a
.  ..  dir2  hardlink
/data/dir1# rm hardlink
/data/dir1# touch again
/data/dir1# rm -i again
rm: remove regular empty file 'again'? yes
/data/dir1# ls -a
.  ..  dir2
/data/dir1# cd ..
/data# ls -a
.  ..  dir1
/data# rmdir dir1
rmdir: failed to remove 'dir1': Directory not empty
/data# rmdir dir1/dir2/
/data# rmdir dir1
/data# ls -a
.  ..
/data# mkdir -p dir1/dir2/dir3/dir4
/data# rmdir dir1
rmdir: failed to remove 'dir1': Directory not empty
/data# rm -r dir1
/data# ls -a
.  ..
/data# █
```

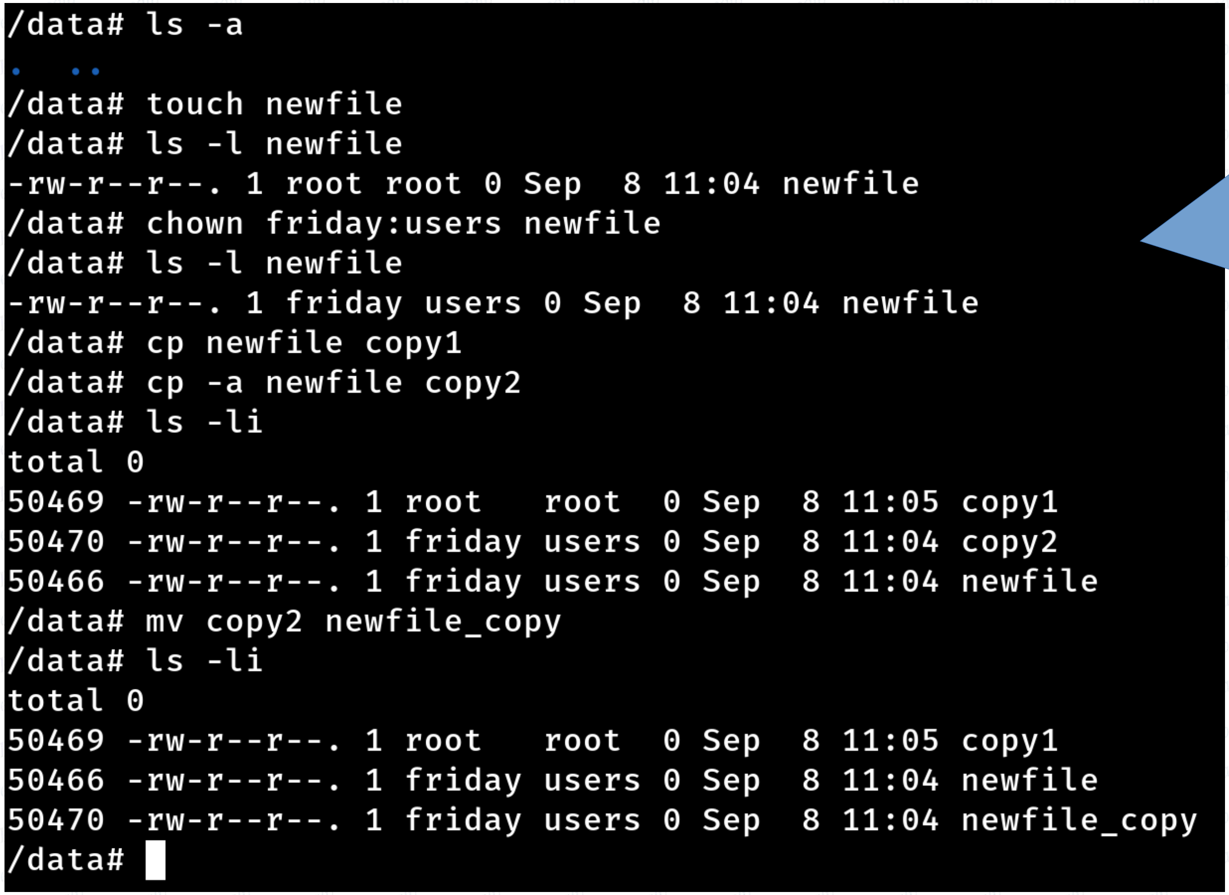
1. removing file with command **rm** (in this case same result as with **unlink**)
2. creating new test file and removing it with the option "**-i**" – that will ask **interactively** the user for permission
3. **cd ..** uses **two-dots** link to move up in directory structure
4. **rmdir** will fail to delete directory if it is **not empty**
5. **rm** with the option "**-r**" will delete content of a directory **recursively** – all files – careful...

# Shell basics (8/10)

- **rmdir**: Safe deletion command for directories – it will remove only empty; in the case of chained empty directories like “*dir1/dir2*” we could shorten two commands: `rmdir dir2; rmdir dir1;`
  to just (with option **-p**): `rmdir -p dir1/dir2`
- **rm**: Where **unlink** and **rmdir** fails, **rm** can be used – it can be dangerous when used mindlessly… options to remember:
  `-r` ← **recursive deletion** for directories and their content
  `-i` ← **interactive option** which will always ask for permission
  `-f` ← **forcefully delete file(s)** => ignore missing files and don’t ask for permission…

  *If you see command (scripts or copy-paste from web) starting with “**rm -rf**” then focus and be **super vigilant** (especially if there is “*****” asterisk…) and get to know what are you about to do...*

# Shell basics (9/10)



```
/data# ls -a
.  ..
/data# touch newfile
/data# ls -l newfile
-rw-r--r--. 1 root root 0 Sep  8 11:04 newfile
/data# chown friday:users newfile
/data# ls -l newfile
-rw-r--r--. 1 friday users 0 Sep  8 11:04 newfile
/data# cp newfile copy1
/data# cp -a newfile copy2
/data# ls -li
total 0
50469 -rw-r--r--. 1 root   root  0 Sep  8 11:05 copy1
50470 -rw-r--r--. 1 friday users 0 Sep  8 11:04 copy2
50466 -rw-r--r--. 1 friday users 0 Sep  8 11:04 newfile
/data# mv copy2 newfile_copy
/data# ls -li
total 0
50469 -rw-r--r--. 1 root   root  0 Sep  8 11:05 copy1
50466 -rw-r--r--. 1 friday users 0 Sep  8 11:04 newfile
50470 -rw-r--r--. 1 friday users 0 Sep  8 11:04 newfile_copy
/data# 
```

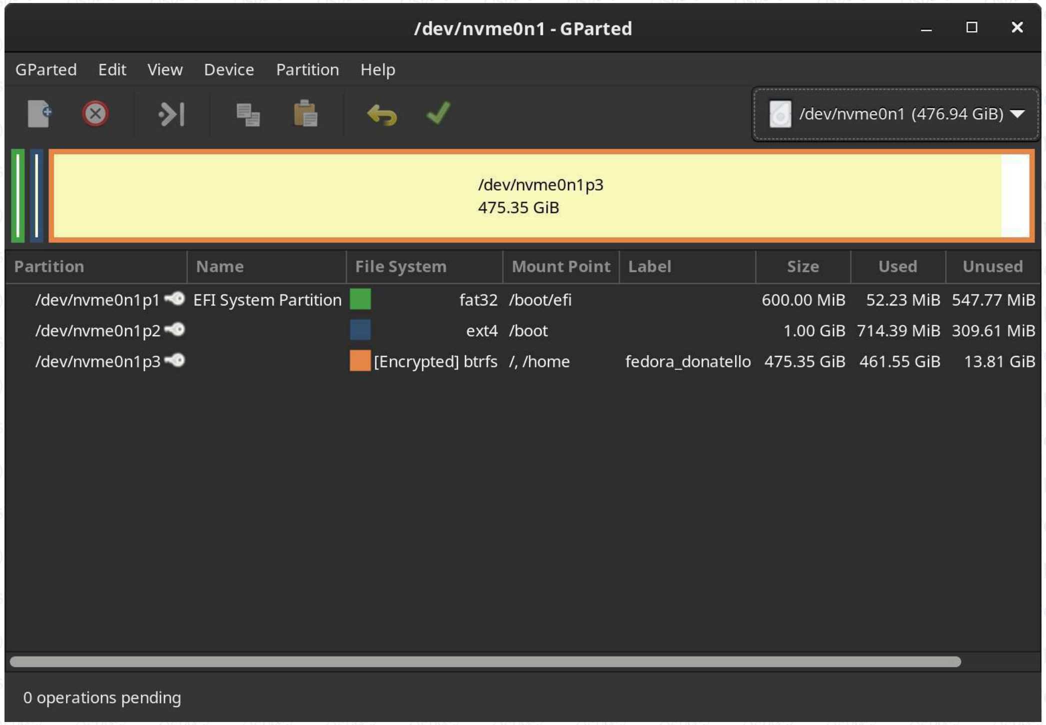
1. creating *newfile*
2. changing ownership
3. copying (**cp**) the file without preserving permissions and ownership
4. Copying (**cp**) the file while preserving the permissions with "**-a**" (this option is overkill here "**-p**" would suffice)
5. moving/renaming (**mv**) the second copy – notice the ownership is not changed...

# Shell basics (10/10)

- **cp**: It expects the original file and the new destination name, options to remember:
  `-r` ← **recursive copy** for directories and their content
  `-p` ← **preserve metadata** like permissions, ownerships, timestamps
  `-a` ← **archive** this combines multiple options into one including (**-r** and **-p**)
  `-i` ← **interactive option** which will always ask for permission
  `-n` ← **no clubber** which will not overwrite existing file
  `-f` ← **forcefully copy** => replace existing destination files and don’t ask for permission…
  ”**cp -a**” *is usually what you want most often...*
- **mv**: It will rename a file or move a file into another directory, options to remember:
  `-n` ← **no clubber** which will not overwrite existing file
  `-i` ← **interactive option** which will always ask for permission
  `-f` ← **forcefully move** => overwrite existing and don’t ask for permission…
- **ls**: New usage: “**ls -l <name>”**; you can add as many file name arguments as you want including directories but if you don’t want to list content of a directory then add option “**-d**”

# Briefly on storage (1/9)

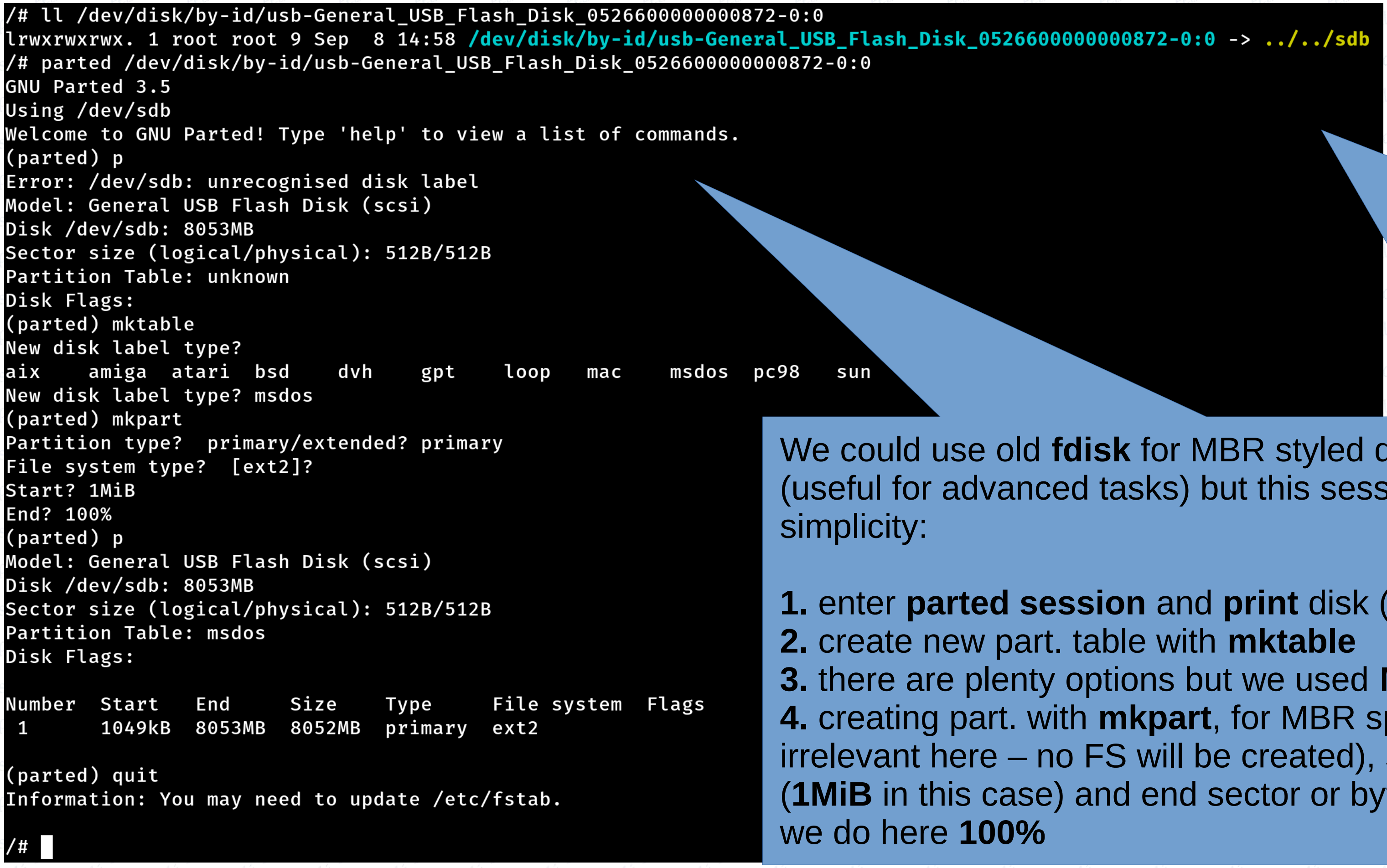
- ***Attention!** Commands in this section are dangerous… and they will delete your data if not careful…*
- If there is a **graphical app** with some **safeguards** and **guardrails** then use that!
- For example **GParted** is very capable and easy enough to use...



Partition	Name	File System	Mount Point	Label	Size	Used	Unused
/dev/nvme0n1p1	EFI System Partition	🟩 fat32	/boot/efi		600.00 MiB	52.23 MiB	547.77 MiB
/dev/nvme0n1p2		🟦 ext4	/boot		1.00 GiB	714.39 MiB	309.61 MiB
/dev/nvme0n1p3		🟧 [Encrypted] btrfs	/, /home	fedora_donatello	475.35 GiB	461.55 GiB	13.81 GiB

0 operations pending

# Briefly on storage (2/9)



```
/# ll /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0
lrwxrwxrwx. 1 root root 9 Sep  8 14:58 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0 -> ../../sdb
/# parted /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0
GNU Parted 3.5
Using /dev/sdb
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) p
Error: /dev/sdb: unrecognised disk label
Model: General USB Flash Disk (scsi)
Disk /dev/sdb: 8053MB
Sector size (logical/physical): 512B/512B
Partition Table: unknown
Disk Flags:
(parted) mktable
New disk label type?
aix    amiga  atari  bsd    dvh    gpt    loop   mac    msdos  pc98   sun
New disk label type? msdos
(parted) mkpart
Partition type?  primary/extended? primary
File system type?  [ext2]?
Start? 1MiB
End? 100%
(parted) p
Model: General USB Flash Disk (scsi)
Disk /dev/sdb: 8053MB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system  Flags
 1      1049kB  8053MB  8052MB  primary  ext2

(parted) quit
Information: You may need to update /etc/fstab.

/#
```

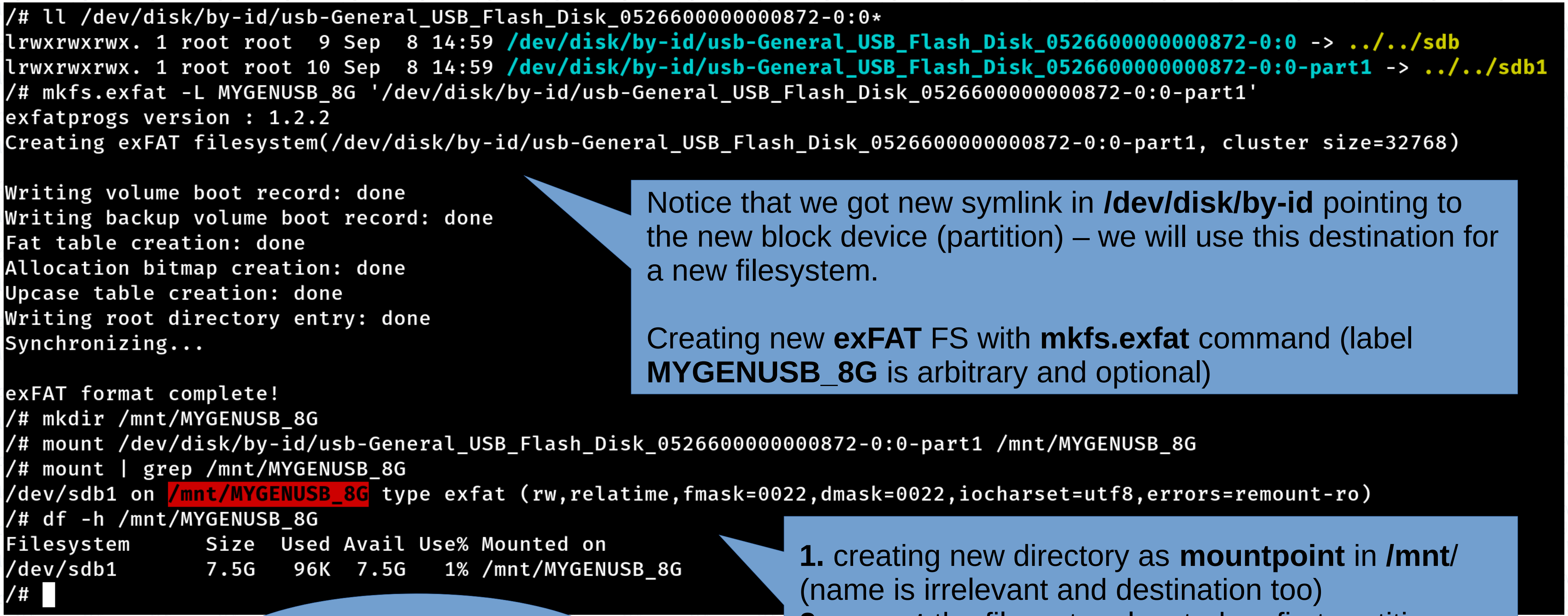
This is USB flash drive inserted into a Linux system (as */dev/sdb*).

I recommend to always use symlinks in ***/dev/disk/by-id/*** to avoid accidentally selecting wrong disk...

We could use old **fdisk** for MBR styled disks or modern **gdisk** (useful for advanced tasks) but this sessions uses **parted** for simplicity:

1. enter **parted session** and **print** disk (**p** for short)
2. create new part. table with **mktable**
3. there are plenty options but we used **MBR/msdos**
4. creating part. with **mkpart**, for MBR specify **primary**, (FS type is irrelevant here – no FS will be created), starting sector or byte (**1MiB** in this case) and end sector or byte or simply percentage like we do here **100%**

# Briefly on storage (3/9)



```
/# ll /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0*
lrwxrwxrwx. 1 root root  9 Sep  8 14:59 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0 -> ../../sdb
lrwxrwxrwx. 1 root root 10 Sep  8 14:59 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0-part1 -> ../../sdb1
/# mkfs.exfat -L MYGENUSB_8G '/dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0-part1'
exfatprogs version : 1.2.2
Creating exFAT filesystem(/dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0-part1, cluster size=32768)

Writing volume boot record: done
Writing backup volume boot record: done
Fat table creation: done
Allocation bitmap creation: done
Upcase table creation: done
Writing root directory entry: done
Synchronizing...

exFAT format complete!
/# mkdir /mnt/MYGENUSB_8G
/# mount /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0-part1 /mnt/MYGENUSB_8G
/# mount | grep /mnt/MYGENUSB_8G
/dev/sdb1 on /mnt/MYGENUSB_8G type exfat (rw,relatime,fmask=0022,dmask=0022,iocharset=utf8,errors=remount-ro)
/# df -h /mnt/MYGENUSB_8G
Filesystem      Size  Used Avail Use% Mounted on
/dev/sdb1       7.5G   96K  7.5G   1% /mnt/MYGENUSB_8G
/# 
```

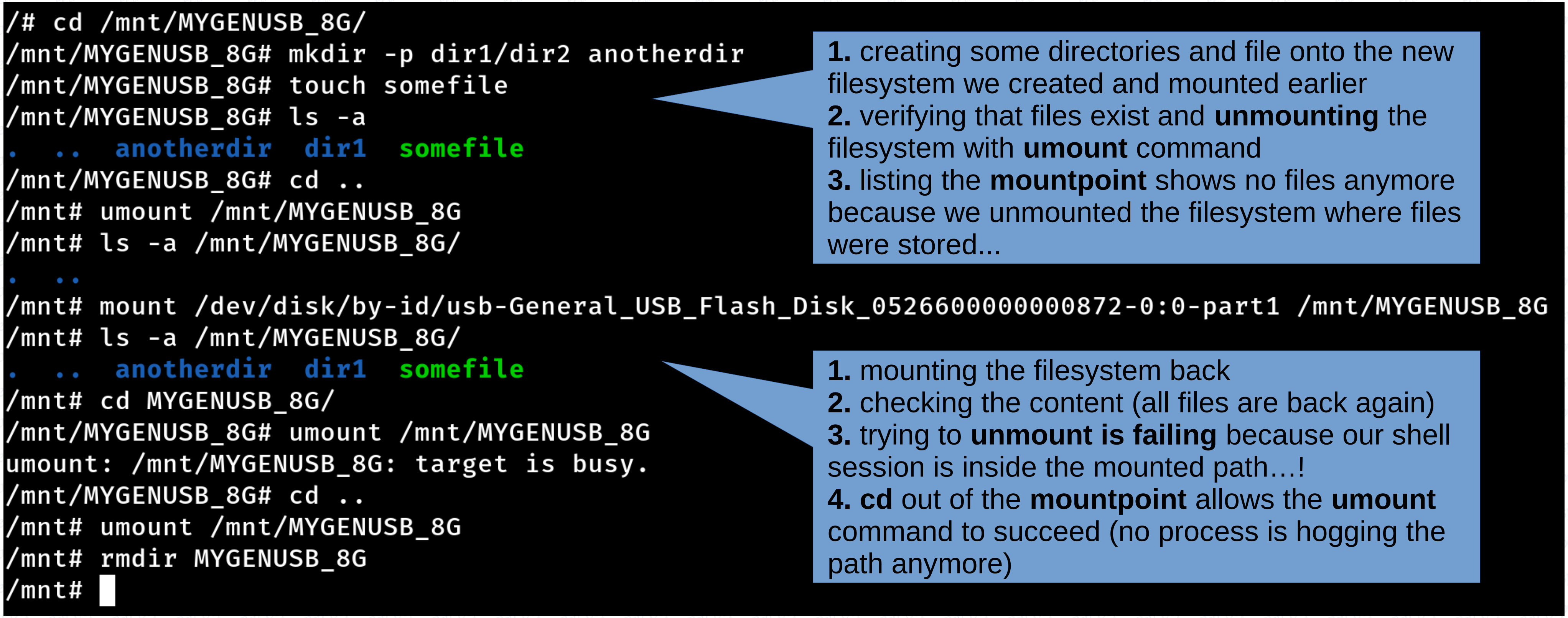
Notice that we got new symlink in ***/dev/disk/by-id*** pointing to the new block device (partition) – we will use this destination for a new filesystem.

Creating new **exFAT** FS with **mkfs.exfat** command (label **MYGENUSB_8G** is arbitrary and optional)

1. creating new directory as **mountpoint** in **/mnt/** (name is irrelevant and destination too)
2. mount the filesystem located on first partition
3. checking that is actually mounted by "**grepping**"
4. printing filesystem usage with **df** command

„**mount|grep …**"
Ignore this for now...

# Briefly on storage (4/9)



```
/# cd /mnt/MYGENUSB_8G/
/mnt/MYGENUSB_8G# mkdir -p dir1/dir2 anotherdir
/mnt/MYGENUSB_8G# touch somefile
/mnt/MYGENUSB_8G# ls -a
.  ..  anotherdir  dir1  somefile
/mnt/MYGENUSB_8G# cd ..
/mnt# umount /mnt/MYGENUSB_8G
/mnt# ls -a /mnt/MYGENUSB_8G/
.  ..
/mnt# mount /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0-part1 /mnt/MYGENUSB_8G
/mnt# ls -a /mnt/MYGENUSB_8G/
.  ..  anotherdir  dir1  somefile
/mnt# cd MYGENUSB_8G/
/mnt/MYGENUSB_8G# umount /mnt/MYGENUSB_8G
umount: /mnt/MYGENUSB_8G: target is busy.
/mnt/MYGENUSB_8G# cd ..
/mnt# umount /mnt/MYGENUSB_8G
/mnt# rmdir MYGENUSB_8G
/mnt# 
```

1. creating some directories and file onto the new filesystem we created and mounted earlier
2. verifying that files exist and **unmounting** the filesystem with **umount** command
3. listing the **mountpoint** shows no files anymore because we unmounted the filesystem where files were stored...

1. mounting the filesystem back
2. checking the content (all files are back again)
3. trying to **unmount is failing** because our shell session is inside the mounted path…!
4. cd out of the **mountpoint** allows the **umount** command to succeed (no process is hogging the path anymore)

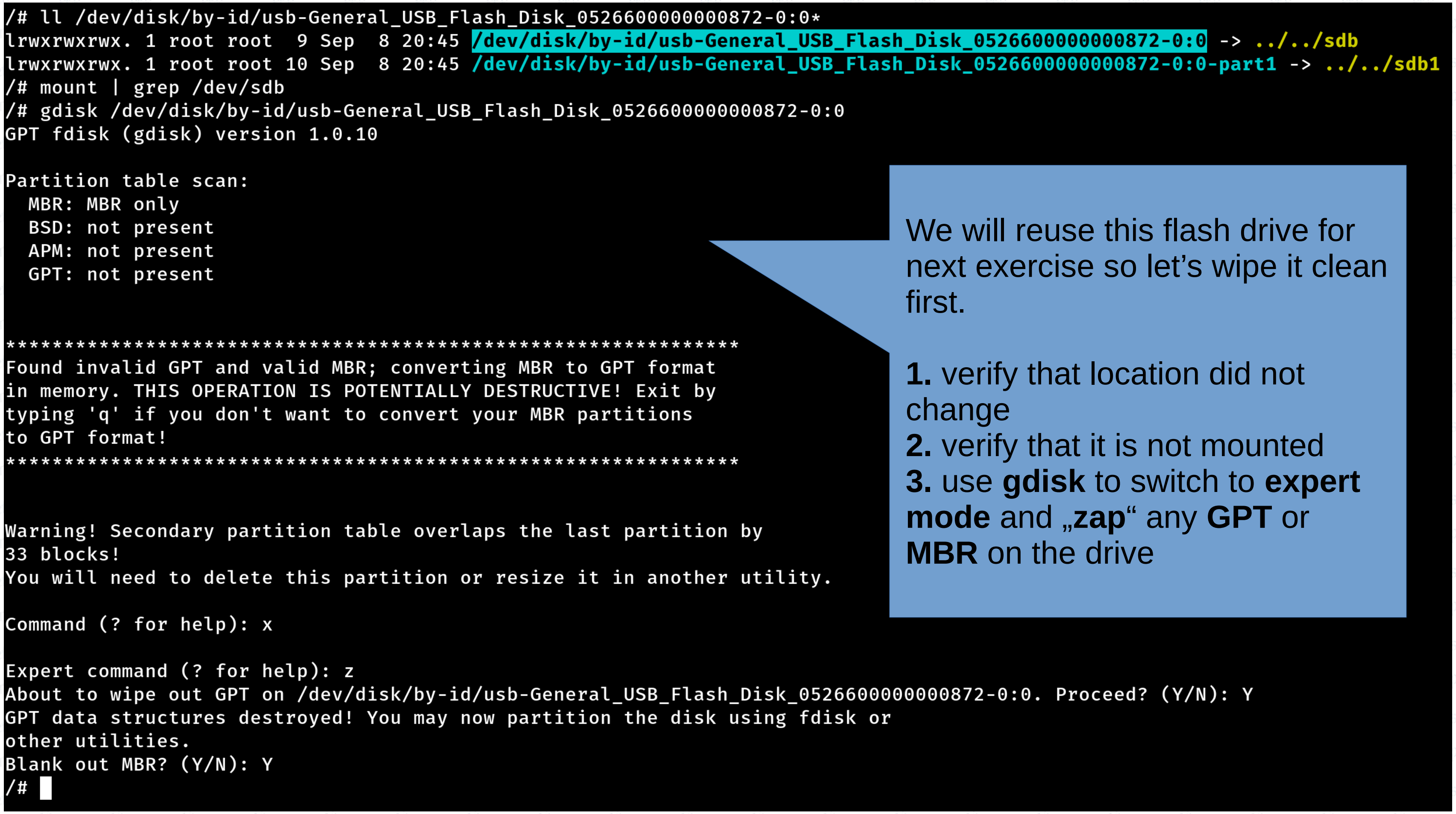
# Briefly on storage (5/9)

- **gdisk**: Not used here but when the block device have orphaned MBR or GPT structures then this command can „zap“ it and make clean slate on the disk
- **parted**: Easier to use than fdisk/gdisk with better **readline** interface (all interactive commands can be passed as arguments so it can be used in scripts too)
- **mkfs**, **mkfs.exfat**: `mkfs` accepts “**`-t <fstype>`**” argument so in our case we could do “**`mkfs -t exfat`** **...**“ with the same result; useful option for exFAT and other *FATs* is
“**`-L <label>`**” as we used here; describing other fstypes is out of scope
- **mount**: Usually “**`mount <device> <mountpoint>`**” is enough but in some cases it is needed to explicitly state the **fstype** and add extra **options**:

```
mount -t btrfs -o subvol=/home,... /dev/disk/by-id/... /home
         ^FS      ^options          ^disk               ^mountpoint
```

- **umount**: Simply to unmount the **mountpoint**

# Briefly on storage (6/9)

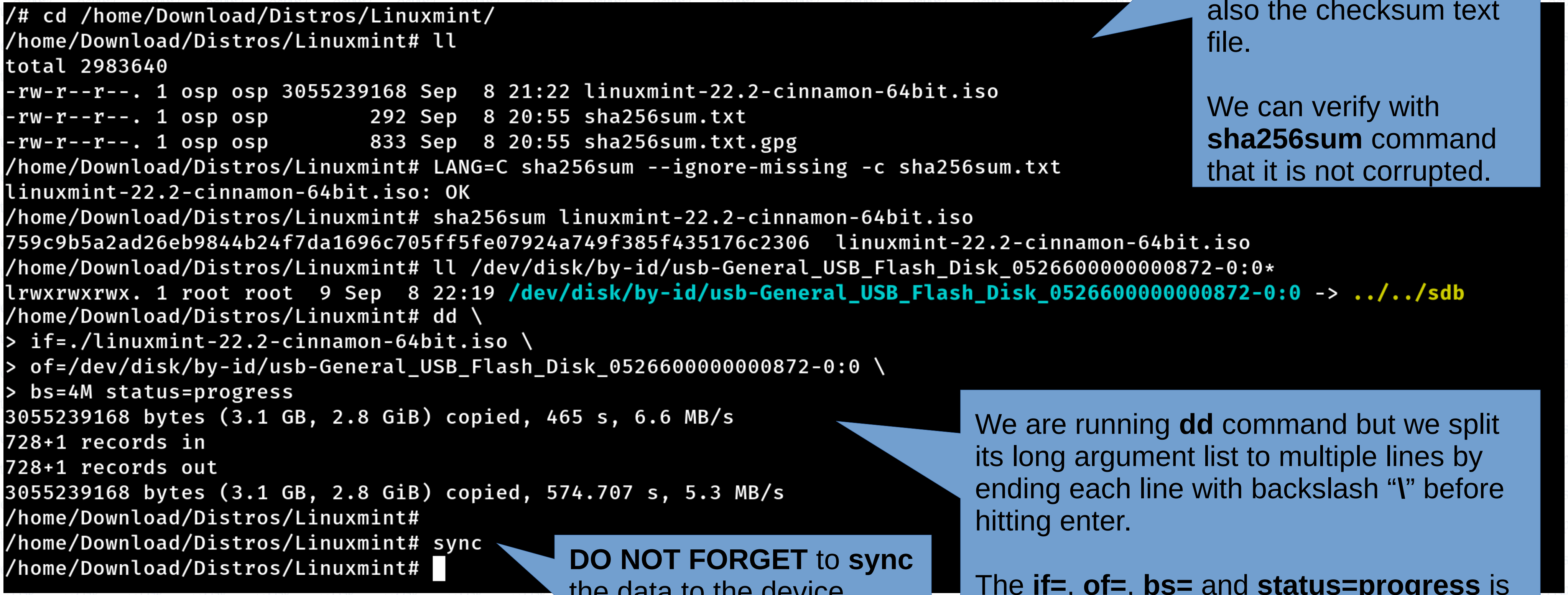


```
/# ll /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0*
lrwxrwxrwx. 1 root root  9 Sep  8 20:45 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0 -> ../../sdb
lrwxrwxrwx. 1 root root 10 Sep  8 20:45 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0-part1 -> ../../sdb1
/# mount | grep /dev/sdb
/# gdisk /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0
GPT fdisk (gdisk) version 1.0.10

Partition table scan:
  MBR: MBR only
  BSD: not present
  APM: not present
  GPT: not present


***************************************************************
Found invalid GPT and valid MBR; converting MBR to GPT format
in memory. THIS OPERATION IS POTENTIALLY DESTRUCTIVE! Exit by
typing 'q' if you don't want to convert your MBR partitions
to GPT format!
***************************************************************


Warning! Secondary partition table overlaps the last partition by
33 blocks!
You will need to delete this partition or resize it in another utility.

Command (? for help): x

Expert command (? for help): z
About to wipe out GPT on /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0. Proceed? (Y/N): Y
GPT data structures destroyed! You may now partition the disk using fdisk or
other utilities.
Blank out MBR? (Y/N): Y
/# 
```

We will reuse this flash drive for next exercise so let’s wipe it clean first.

1. verify that location did not change
2. verify that it is not mounted
3. use **gdisk** to switch to **expert mode** and „**zap**“ any **GPT** or **MBR** on the drive

# Briefly on storage (7/9)



```
/# cd /home/Download/Distros/Linuxmint/
/home/Download/Distros/Linuxmint# ll
total 2983640
-rw-r--r--. 1 osp osp 3055239168 Sep  8 21:22 linuxmint-22.2-cinnamon-64bit.iso
-rw-r--r--. 1 osp osp        292 Sep  8 20:55 sha256sum.txt
-rw-r--r--. 1 osp osp        833 Sep  8 20:55 sha256sum.txt.gpg
/home/Download/Distros/Linuxmint# LANG=C sha256sum --ignore-missing -c sha256sum.txt
linuxmint-22.2-cinnamon-64bit.iso: OK
/home/Download/Distros/Linuxmint# sha256sum linuxmint-22.2-cinnamon-64bit.iso
759c9b5a2ad26eb9844b24f7da1696c705ff5fe07924a749f385f435176c2306  linuxmint-22.2-cinnamon-64bit.iso
/home/Download/Distros/Linuxmint# ll /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0*
lrwxrwxrwx. 1 root root  9 Sep  8 22:19 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0 -> ../../sdb
/home/Download/Distros/Linuxmint# dd \
> if=./linuxmint-22.2-cinnamon-64bit.iso \
> of=/dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0 \
> bs=4M status=progress
3055239168 bytes (3.1 GB, 2.8 GiB) copied, 465 s, 6.6 MB/s
728+1 records in
728+1 records out
3055239168 bytes (3.1 GB, 2.8 GiB) copied, 574.707 s, 5.3 MB/s
/home/Download/Distros/Linuxmint#
/home/Download/Distros/Linuxmint# sync
/home/Download/Distros/Linuxmint# 
```

Here is downloaded bootable image of Linux distribution (ISO file) and also the checksum text file.

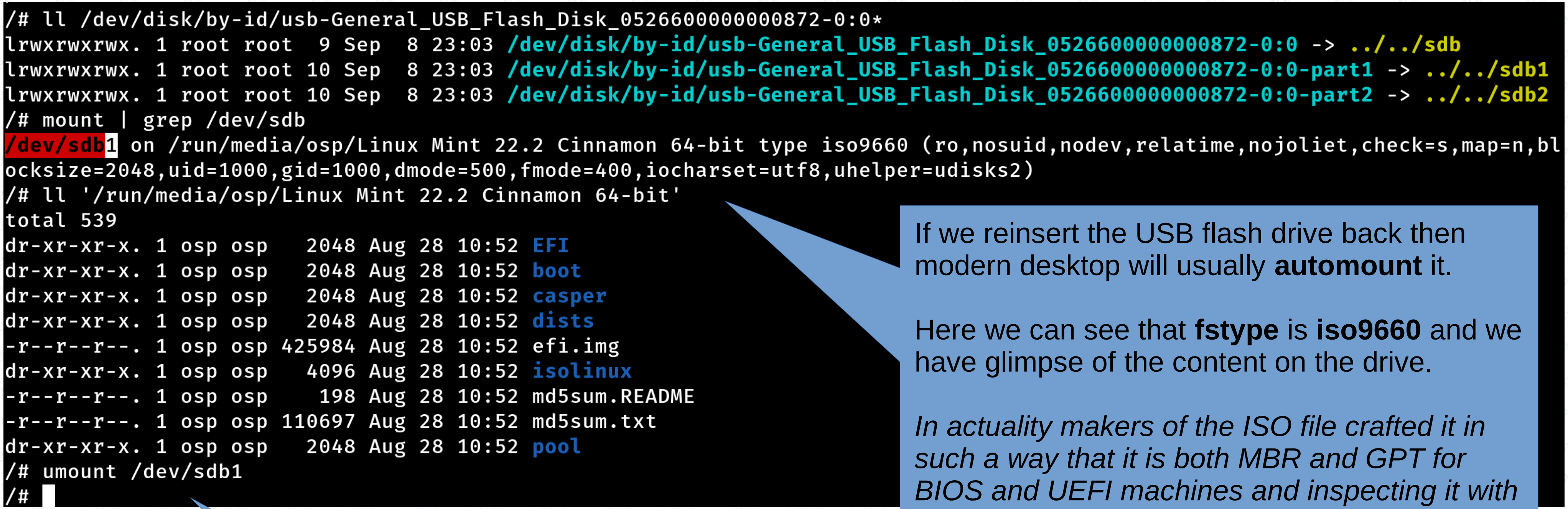
We can verify with **sha256sum** command that it is not corrupted.

We are running **dd** command but we split its long argument list to multiple lines by ending each line with backslash “****” before hitting enter.

The **if=**, **of=**, **bs=** and **status=progress** is sufficient usage for writing files with **dd**.

DO NOT FORGET to **sync** the data to the device… otherwise some might be lingering in memory only...

# Briefly on storage (8/9)



```
/# ll /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0*
lrwxrwxrwx. 1 root root  9 Sep  8 23:03 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0 -> ../../sdb
lrwxrwxrwx. 1 root root 10 Sep  8 23:03 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0-part1 -> ../../sdb1
lrwxrwxrwx. 1 root root 10 Sep  8 23:03 /dev/disk/by-id/usb-General_USB_Flash_Disk_0526600000000872-0:0-part2 -> ../../sdb2
/# mount | grep /dev/sdb
/dev/sdb1 on /run/media/osp/Linux Mint 22.2 Cinnamon 64-bit type iso9660 (ro,nosuid,nodev,relatime,nojoliet,check=s,map=n,bl
ocksize=2048,uid=1000,gid=1000,dmode=500,fmode=400,iocharset=utf8,uhelper=udisks2)
/# ll '/run/media/osp/Linux Mint 22.2 Cinnamon 64-bit'
total 539
dr-xr-xr-x. 1 osp osp   2048 Aug 28 10:52 EFI
dr-xr-xr-x. 1 osp osp   2048 Aug 28 10:52 boot
dr-xr-xr-x. 1 osp osp   2048 Aug 28 10:52 casper
dr-xr-xr-x. 1 osp osp   2048 Aug 28 10:52 dists
-r--r--r--. 1 osp osp 425984 Aug 28 10:52 efi.img
dr-xr-xr-x. 1 osp osp   4096 Aug 28 10:52 isolinux
-r--r--r--. 1 osp osp    198 Aug 28 10:52 md5sum.README
-r--r--r--. 1 osp osp 110697 Aug 28 10:52 md5sum.txt
dr-xr-xr-x. 1 osp osp   2048 Aug 28 10:52 pool
/# umount /dev/sdb1
/# 
```

If we reinsert the USB flash drive back then modern desktop will usually **automount** it.

Here we can see that **fstype** is **iso9660** and we have glimpse of the content on the drive.

*In actuality makers of the ISO file crafted it in such a way that it is both MBR and GPT for BIOS and UEFI machines and inspecting it with **parted**, **fdisk** or **gdisk** commands might give you a headache… **Treat it as a binary blob.***

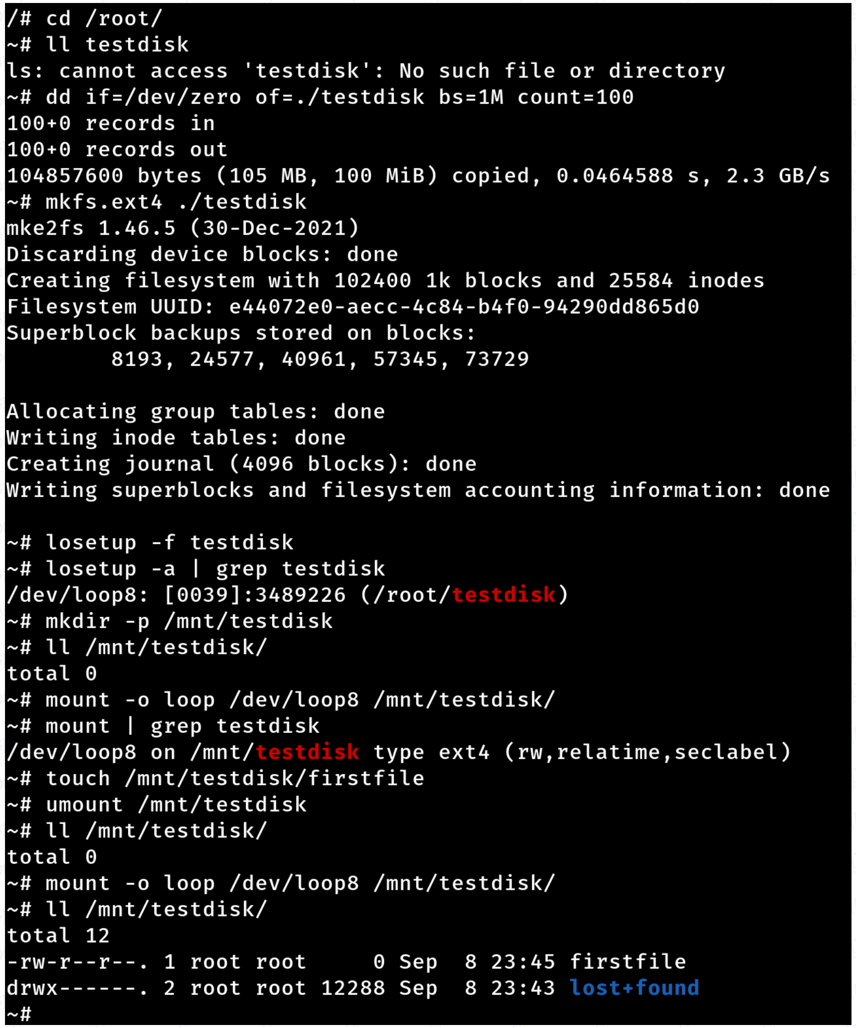
Notice that we **unmounted** with the **device** as the argument and not mountpoint… Both ways are valid.

# Briefly on storage (9/9)

- **sha256sum**: Run *sha256* checksum on a file and also check against the checksum file if provided with argument “**-c <checksums file>**”
- **sync**: It will flush all uncommitted writes to the disk(s) – always use after command like **dd**
- **dd**: <u>*This command will not ask for permission to overwrite your data*</u> – so always double-check that you are writing where you think you are writing… The most common usage was demonstrated with “baking” **ISO** file onto flash drive:
  `if=<source>` ← Input File (e.g. disk, device...)
  `of=<target>` ← Output File (e.g. disk, device...)
  `bs=<blocksize>` ← More useful in combination with “`count=<number>`”
  `status=progress` ← Show status (by default **dd** is <u>silent</u>)
  ”**dd if=/dev/zero of=./myfile bs=1M count=1024**” will create 1GiB “*empty*” file

# Loop devices (1/3)

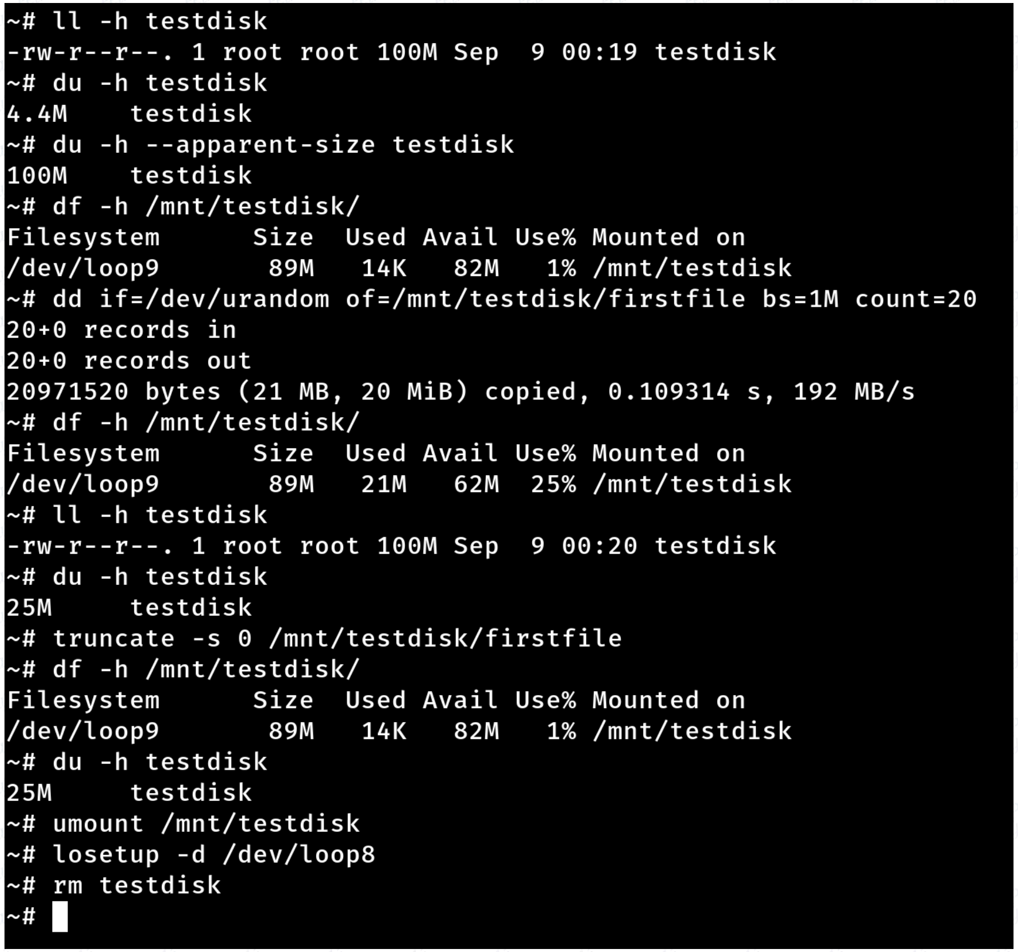
- We can create a file of any desired size with **dd** command (here we simply using special device **/dev/zero** as source)
- We can create filesystem (ext4 in this case) onto this file
- Magic happens with a **losetup** tool which will allocate for us special **loop device**
- We can then map our file onto it and mount it (with the option "**-o loop**")



```
/# cd /root/
~# ll testdisk
ls: cannot access 'testdisk': No such file or directory
~# dd if=/dev/zero of=./testdisk bs=1M count=100
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 0.0464588 s, 2.3 GB/s
~# mkfs.ext4 ./testdisk
mke2fs 1.46.5 (30-Dec-2021)
Discarding device blocks: done
Creating filesystem with 102400 1k blocks and 25584 inodes
Filesystem UUID: e44072e0-aecc-4c84-b4f0-94290dd865d0
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done

~# losetup -f testdisk
~# losetup -a | grep testdisk
/dev/loop8: [0039]:3489226 (/root/testdisk)
~# mkdir -p /mnt/testdisk
~# ll /mnt/testdisk/
total 0
~# mount -o loop /dev/loop8 /mnt/testdisk/
~# mount | grep testdisk
/dev/loop8 on /mnt/testdisk type ext4 (rw,relatime,seclabel)
~# touch /mnt/testdisk/firstfile
~# umount /mnt/testdisk
~# ll /mnt/testdisk/
total 0
~# mount -o loop /dev/loop8 /mnt/testdisk/
~# ll /mnt/testdisk/
total 12
-rw-r--r--. 1 root root     0 Sep  8 23:45 firstfile
drwx------. 2 root root 12288 Sep  8 23:43 lost+found
~#
```

# Loop devices (2/3)

- Notice that **ls** will report the size of our disk file as **100M** (that is expected – we **dd** 100 times 1M)
- (Disk Usage) tool **du** reports only **4.4M** (that is because underlying FS supports **sparse files**)
- Another (Disk Free) tool **df** can tell us filesystem usage
- This time we **dd** not zeroes but **pseudorandom** bytes from **/dev/urandom** onto our empty "*firstfile*"
- We can see the difference with **df** command in the usage of the FS
- Tool **truncate** shrinked the file back to zero bytes and **df** reports **1%** usage only again
- Although the expanded disk usage of the *testdisk* file remained (25M) – once used sectors are no longer **sparse**
- Lastly **umount** and release the loop device "**losetup -d <loop device>**"



```
~# ll -h testdisk
-rw-r--r--. 1 root root 100M Sep  9 00:19 testdisk
~# du -h testdisk
4.4M    testdisk
~# du -h --apparent-size testdisk
100M    testdisk
~# df -h /mnt/testdisk/
Filesystem      Size  Used Avail Use% Mounted on
/dev/loop9       89M   14K   82M   1% /mnt/testdisk
~# dd if=/dev/urandom of=/mnt/testdisk/firstfile bs=1M count=20
20+0 records in
20+0 records out
20971520 bytes (21 MB, 20 MiB) copied, 0.109314 s, 192 MB/s
~# df -h /mnt/testdisk/
Filesystem      Size  Used Avail Use% Mounted on
/dev/loop9       89M   21M   62M  25% /mnt/testdisk
~# ll -h testdisk
-rw-r--r--. 1 root root 100M Sep  9 00:20 testdisk
~# du -h testdisk
25M     testdisk
~# truncate -s 0 /mnt/testdisk/firstfile
~# df -h /mnt/testdisk/
Filesystem      Size  Used Avail Use% Mounted on
/dev/loop9       89M   14K   82M   1% /mnt/testdisk
~# du -h testdisk
25M     testdisk
~# umount /mnt/testdisk
~# losetup -d /dev/loop8
~# rm testdisk
~# █
```

# Loop devices (3/3)

- **df**: Disk Free command – reports filesystem usage; useful option is “**-h**” which gives “human readable” values (K, M, G...) instead of long byte values
- **du**: Disk Usage command – by default it will show real amount of bytes a file is using on the filesystem; useful options:
  `-h` → human readable format
  `-s` → summarize the result
  `-c` → show total (in the case of more than one file argument)
  `--apparent-size` → it will report the “claimed” size (what **ls** would show)
  `-x` → stays in one FS (skip mounted FS in some subdirectory)
  ”**du -shx /dir**” is useful command to calculate how much space this directory is taking on your FS without counting <u>submounted</u> filesystems and their files
- **losetup**: Tool to manage loop devices, imported options:
  `-a` → list all active loop devices
  `-f` → find new unused loop device to be attached to a file (**losetup -f <myfile>**)
  `-d` → detach the loop device (**losetup -d /dev/loopXYZ**)
- **truncate**: Used for resizing a given file – common usage: **truncate -s <newsize bytes> <file>**

# Final words

- **Everything is a file in UNIX-like system and a text file is the most prevalent data around which many tools orbit**
- **Filesystems on UNIX-like system is hierarchical tree-like structure**
- **Symlinks** are used to circumvent and “break” this tree hierarchy
- **Inode** is the universal underlying file structure on UNIX-like filesystem
- UNIX-like systems have distinct ownership and permissions
- **This was long and exhaustive presentation on a very extensive topic and yet we covered only the surface...**
- Although it should be enough of a headstart to build more skills and knowledge as time goes on