The `ls` command in Linux/Unix is used to list directory contents. It has numerous options (parameters) to customize the output. Below is a detailed breakdown of the `ls` command and its parameters.

Basic Syntax

sh

```
ls [OPTIONS] [FILE/DIRECTORY]
```

Commonly Used Options

Option	Description
`-a`, `--all`	List all files, including hidden files (those starting with `.`)
`-A`, `--almost-all`	List all files except `.` (current dir) and `..` (parent dir)
`-l`	Long listing format (shows permissions, owner, size, etc.)
`-h`, `--human-readable`	Show file sizes in human-readable format (e.g., 1K, 2M)
`-t`	Sort by modification time (newest first)
`-r`, `--reverse`	Reverse the sorting order
`-S`	Sort by file size (largest first)
`-R`, `--recursive`	List subdirectories recursively
`-F`, `--classify`	Append indicator (`/` for dir, `*` for executable, etc.)
`--color`	Colorize output (default in most systems)
`-d`, `--directory`	List directories themselves, not their contents
`-i`, `--inode`	Display inode number of each file
`-n`, `--numeric-uid-gid`	Display numeric UID & GID instead of names
`-1`	List one file per line
`-m`	List files separated by commas
`-Q`, `--quote-name`	Enclose file names in double quotes
`--help`	Display help message
`--version`	Display `ls` version

Examples of `ls` Command

1. **List all files (including hidden ones)**

sh

- `ls -a`

- **Long listing format with human-readable sizes**

sh
- `ls -lh`

- **Sort files by modification time (newest first)**

sh
- `ls -lt`

- **Reverse sort order**

sh
- `ls -lr`

- **List directories only (not their contents)**

sh
- `ls -d */`

- **List files with inode numbers**

sh
- `ls -i`

- **Recursively list all files in subdirectories**

sh
- `ls -R`

- **List files sorted by size (largest first)**

sh
- `ls -lS`

- **Display files in a comma-separated list**

sh
- `ls -m`

- **List files with special indicators (`/` for dir, `*` for executable)**

sh
- `ls -F`

Understanding `ls -l` Output

When using `ls -l`, the output looks like this:

sh
```
-rw-r--r-- 1 user group 1024 May 7 10:00 file.txt
```

- **`-rw-r--r--`** → File permissions (read, write, execute for owner/group/others)
- **`1`** → Number of hard links
- **`user`** → Owner of the file
- **`group`** → Group associated with the file
- **`1024`** → File size in bytes
- **`May 7 10:00`** → Last modification time
- **`file.txt`** → File name

Combining Options

You can combine multiple options, e.g.:
sh
```
ls -lathR  # Long listing, all files, human-readable, time-sorted, recursive
```

The `cd` (Change Directory) command is one of the most fundamental and frequently used commands in Linux/Unix systems. It allows you to navigate between directories in the filesystem. Below is a detailed breakdown of the `cd` command, its syntax, options, and usage examples.

Basic Syntax
sh
```
cd [OPTIONS] [DIRECTORY]
```

Key Features

- Changes the current working directory.
- If no directory is specified, `cd` defaults to the user's home directory (`~`).
- Supports absolute paths (starting from `/`) and relative paths (relative to the current directory).

Common+ Command Usage

1. Change to a Specific Directory

sh
```
cd /path/to/directory
```
Example:
sh
```
cd /etc/nginx  # Moves to /etc/nginx
```

2. Move to Home Directory
sh
```
cd       # No arguments → goes to home directory (~)
cd ~     # Explicitly goes to home directory
cd $HOME # Uses the HOME environment variable
```

3. Move to Parent Directory (One Level Up)
sh
```
cd ..    # Moves up one directory level
```

Example:
sh
```
cd /usr/local/bin
cd ..    # Now in /usr/local
```

4. Move to Previous Directory (Go Back)

sh
```
cd -     # Switches back to the last directory you were in
```

Example:

sh
```
cd /var/log
cd /home/user
cd -     # Returns to /var/log
```

5. Using Relative Paths

- `.` → Current directory (rarely used with `cd` but useful in scripts)
- `..` → Parent directory
- `../..` → Move up two levels
- **`dirname`** → Subdirectory in the current folder

Examples:
sh
```
cd ./Documents  # Moves into "Documents" (same as `cd Documents`)
cd ../Downloads # Moves from current dir to sibling "Downloads"
cd ../../       # Moves up two directory levels
```

6. Using Environment Variables

sh
```
cd $VARIABLE_NAME
```

Example:
sh
```
cd $HOME        # Goes to home directory (~)
cd $PWD         # Goes to current directory (redundant)
```

7. Using Tilde (~) Expansion

- `~` → Home directory
- `~username` → Home directory of the specified user

Examples:

sh
```
cd ~             # Goes to current user's home
cd ~john         # Goes to user "john"'s home (if permitted)
```

Special Notes

- If the directory name has **spaces or special characters**, enclose it in quotes:

  sh

- `cd "My Documents"`
- `cd 'Folder with spaces'`
- **Tab Completion** → Press `Tab` to auto-complete directory names.
- **`cd` Alone** → Always returns to the home directory (`~`).

Common Errors & Fixes

Error	Reason	Solution
`cd: no such file or directory`	Directory doesn't exist	Check spelling, use `ls`
`cd: permission denied`	Lack of permissions	Use `sudo` (if admin) or check `ls -ld dir`
`cd: not a directory`	Target is a file, not a dir	Use `ls` to verify

Summary Table

Command	Action
`cd`	Go to home directory
`cd ~`	Go to home directory
`cd /path/to/dir`	Go to an absolute path
`cd dir`	Go to a subdirectory (relative path)
`cd ..`	Move up one directory
`cd ../..`	Move up two directories
`cd -`	Switch to the previous directory
`cd $HOME`	Go to home directory (using variable)
`cd ~username`	Go to another user's home

Final Notes

- `cd` is a shell builtin (not a separate executable), so `man cd` won't work. Use `help cd` instead.
- Works the same in **Bash, Zsh, and most Unix-like shells**.

pwd command

The **`pwd`** (Print Working Directory) command is a fundamental Linux/Unix command used to display the full path of the current working directory. It helps users know their location in the filesystem hierarchy.

Basic Syntax

sh
```
pwd [OPTIONS]
```

Key Features

- Prints the **absolute path** (full path from the root `/`) of the current directory.
- Useful in scripts and when navigating complex directory structures.
- Works in **Bash, Zsh, and most Unix-like shells**.

Common Usage & Examples

1. Basic Usage (Print Current Directory)

sh
```
pwd
```

Output Example:
sh
```
/home/user/Documents
```

2. Show Physical Path (Avoid Symlinks) with `-P`

sh
```
pwd -P
```

- Resolves symbolic links (shows the actual path, not the linked one).
  Example:
  If you're inside a symlinked directory like `/var/www/html` (which points to `/srv/web`),
- `pwd` → `/var/www/html`
- `pwd -P` → `/srv/web`

3. Show Logical Path (Default Behavior) with `-L`

sh
```
pwd -L
```

- Displays the linked path (default behavior).
  Example:
  If inside a symlink `/var/www/html`,
- `pwd -L` → `/var/www/html` (same as `pwd`)

How `pwd` Works

- The shell maintains an environment variable `$PWD` (Present Working Directory).
- `pwd` retrieves and prints this value.
- The `-P` flag forces a filesystem check (ignores symlinks).

Common Errors & Fixes

Error	Reason	Solution
`pwd: too many arguments`	Extra arguments passed	Use just `pwd` or `pwd -P`
`pwd: invalid option`	Unsupported flag	Use only `-P` or `-L`

Comparison: `pwd` vs `echo $PWD`

Command	Behavior
`pwd`	Built-in command, follows `-P`/`-L` flags
`echo $PWD`	Displays the shell's stored path (may differ if `cd` changes symlinks)

Example:

sh
```
cd /var/www/html   # Suppose this is a symlink to /srv/web
pwd                # Output: /var/www/html
pwd -P             # Output: /srv/web
echo $PWD          # Output: /var/www/html
```

Summary Table

Command	Description
`pwd`	Print current working directory (may include symlinks)
`pwd -P`	Show the physical path (resolves symlinks)
`pwd -L`	Show the logical path (default, keeps symlinks)
`echo $PWD`	Prints the shell's stored path

Use Cases

1. **Scripting** → Helps verify the working directory before file operations.
2. **Debugging** → Confirms where a script is executing.
3. **Navigation** → Useful when working in deep directory structures.

Final Notes

- `pwd` is a **shell builtin** (check with `type pwd`).
- For more details, run:

  sh

- `man pwd   # or `help pwd` (if builtin)`

output redirection with > and >>

Output Redirection in Linux (> and >>)

Output redirection allows you to control where command output goes—either to a file instead of the terminal. The two primary operators are:

Operator	Description
>	Overwrites a file (or creates it if it doesn't exist)
>>	Appends to a file (or creates it if it doesn't exist)

---

1. > (Overwrite Redirection)

Overwrites the file if it exists, or creates a new one.
Syntax
sh
```
command > file.txt
```

Examples

Save `ls` output to a file (overwrite)

sh
```
ls > files_list.txt
```

- If `files_list.txt` exists, it gets **overwritten**.
- If it doesn't exist, it gets **created**.

Redirect `echo` to a file
sh
```
echo "Hello, World!" > message.txt
```

- Replaces `message.txt` with `"Hello, World!"`.

Discard output (send to `/dev/null`)
sh
```
command > /dev/null  # Silences output
```

---

2. >> (Append Redirection)

Appends output to a file (does not overwrite).

Syntax

sh
```
command >> file.txt
```

Examples

Append `date` to a log file

sh
```
date >> log.txt
```

- Adds the current date/time to `log.txt` without deleting previous content.

Add multiple lines to a file

sh
```
echo "Line 1" >> notes.txt
echo "Line 2" >> notes.txt
```

- `notes.txt` will contain:

- `Line 1`
- `Line 2`

Key Differences: `>` vs `>>`

Feature	>	>>
File Creation	Creates if missing	Creates if missing
Existing Content	Overwrites	Appends
Use Case	New file/replace	Logging/accumulating data

Common Use Cases

- **Logging** → Append output to a log file:

sh

- `echo "Script started at $(date)" >> script.log`

- **Saving Command Output** → Store results for later:

sh
- `df -h > disk_usage.txt`

- **Error Redirection** → Combine with `2>` (stderr):

sh
- `command > output.txt 2> errors.txt`

- **Combining Outputs** → Redirect both stdout and stderr:

sh
- `command &> all_output.txt`

Advanced Redirection

Redirect `stderr` Only (`2>`)

sh
```
command 2> error.log  # Only errors go to error.log
```

Redirect Both `stdout` and `stderr` (`&>`)

sh
```
command &> output_and_errors.log
```

Redirect `stdout` to One File and `stderr` to Another
sh
```
command > output.txt 2> errors.txt
```

Summary Table

Command	Effect
`cmd > file`	Overwrites `file` with output
`cmd >> file`	Appends output to `file`
`cmd 2> file`	Redirects errors to `file`
`cmd &> file`	Redirects **both** output and errors to `file`
`cmd > file 2>&1`	Older syntax for `&>` (redirects all output)

Important Notes

- If the file doesn't exist, both `>` and `>>` **create it**.
- `>` is **destructive** (erases existing content).
- `>>` is **safe for logs** (keeps old data).
- Use `set -o no clobber` to prevent accidental overwrites with `>`.

Final Tips

- Check redirections with `ls` or `cat`:

sh

- `cat output.txt`

- To **append and print to terminal**, use `tee -a`:

sh

- `command | tee -a log.txt`

Utilize grep to search for text within file

Using `grep` to Search for Text in Files

`grep` (Global Regular Expression Print) is a powerful Linux command-line tool for searching text patterns in files. It supports **regular expressions** and offers many options for filtering results.

Basic Syntax

sh

```
grep [OPTIONS] "PATTERN" [FILE...]
```

Key Features

- Search for **text patterns** in files.
- Support for **regular expressions** (advanced pattern matching).
- Display **matching lines** with file names (if multiple files).
- Can **count matches**, **ignore case**, **show line numbers**, etc.

Common `grep` Usage Examples

1. Search for a Word in a File

sh
```
grep "error" logfile.txt
```

- Displays all lines containing `"error"` in `logfile.txt`.

2. Case-Insensitive Search (`-i`)

sh
```
grep -i "warning" system.log
```

- Finds `"warning"`, `"Warning"`, `"WARNING"`, etc.

3. Show Line Numbers (`-n`)

sh
```
grep -n "failed" app.log
```

- Outputs matched lines with their line numbers:
- `42:Error: failed to connect`
- `105:Connection failed`

4. Count Matches (`-c`)

sh
```
grep -c "success" report.txt
```

- Prints only the **number of matches** (e.g., `3`).

5. Search in Multiple Files

sh
```
grep "404" access.log error.log
```

- Searches for `"404"` in both files.

6. Search All Files in a Directory (`-r` Recursive)

sh
```
grep -r "TODO" /home/user/projects/
```

- Searches for `"TODO"` in **all files** under `/home/user/projects/`.

7. Invert Match (`-v` – Exclude Lines)

sh
```
grep -v "success" results.txt
```

- Shows lines **not containing** `"success"`.

8. Search for Whole Words (`-w`)

sh
```
grep -w "port" config.txt
```

- Matches `"port"` but not `"export"` or `"portal"`.

9. Use Regular Expressions (`-E` Extended Regex)

sh
```
grep -E "[0-9]{3}-[0-9]{2}-[0-9]{4}" data.txt
```

- Finds **SSN-like patterns** (e.g., `123-45-6789`).

10. Display Filename Only (`-l`)

sh
```
grep -l "critical" *.log
```

- Lists **only filenames** containing `"critical"`.

11. Show Context Around Matches (`-A`, `-B`, `-C`)

- `-A 3` → Show **3 lines after** the match.
- `-B 2` → Show **2 lines before** the match.
- `-C 1` → Show **1 line before & after** the match.

Example:

sh
```
grep -C 2 "segmentation fault" kernel.log
```

Advanced `grep` Usage

Pipe Output from Another Command

sh
```
cat log.txt | grep "error"
```

or

sh
```
ps aux | grep "nginx"
```

Search Compressed Files (`zgrep`)

sh
```
zgrep "exception" /var/log/syslog.2.gz
```

Highlight Matches (`--color=auto`)

sh
```
grep --color=auto "important" notes.txt
```

- Highlights `"important"` in color.

Summary Table

Command	Description
`grep "text" file`	Basic search
`grep -i "text" file`	Case-insensitive search
`grep -n "text" file`	Show line numbers
`grep -c "text" file`	Count matches
`grep -r "text" /dir/`	Recursive search
`grep -v "text" file`	Exclude matches
`grep -w "text" file`	Whole-word match
`grep -E "regex" file`	Extended regex
`grep -l "text" *.log`	List filenames only
`grep -A 3 "text" file`	Show 3 lines after match
`grep -B 2 "text" file`	Show 2 lines before match
`grep -C 1 "text" file`	Show 1 line before & after

Common Errors & Fixes

Error	Solution
`grep: file.txt: No such file`	Check file path
`Binary file matches`	Use `grep -a` to search binary files
`Pattern not found`	Check spelling or use `-i`

Final Notes

- `grep` is **fast** and **versatile** for text searches.
- For **more complex patterns**, learn **regular expressions** (`man grep`).
- Combine with `>` to save results:

  sh
- `grep "error" log.txt > errors.txt`

cp command

`cp` Command in Linux (Copy Files & Directories)

The `cp` (copy) command is used to **copy files and directories** in Linux/Unix systems. It supports various options for controlling how files are duplicated.

Basic Syntax

sh
```
cp [OPTIONS] SOURCE DEST
cp [OPTIONS] SOURCE... DIRECTORY
```

Key Features

- Copies files/directories from **SOURCE** to **DEST**.
- Can **overwrite** existing files (use `-i` for confirmation).
- Supports **recursive copying** (`-R` for directories).
- Preserves file **attributes** (`-p` for permissions, timestamps).

Common `cp` Usage Examples

1. Copy a File to Another Location

sh
```
cp file.txt /backup/
```

- Copies `file.txt` to `/backup/`.

2. Copy and Rename a File

sh
```
cp file.txt newfile.txt
```

- Copies `file.txt` to `newfile.txt` (in the same directory).

3. Copy Multiple Files to a Directory

sh
```
cp file1.txt file2.txt /backup/
```

- Copies both files to `/backup/`.

4. Copy a Directory Recursively (`-R` or `-r`)

sh
```
cp -R /source/folder /destination/
```

- Copies **all files & subdirectories** inside `folder`.

5. Preserve File Attributes (`-p`)

sh
```
cp -p original.txt backup/
```

- Keeps **permissions, ownership, and timestamps**.

6. Interactive Mode (`-i` – Confirm Before Overwriting)

sh
```
cp -i file.txt /backup/
```

- Asks before overwriting existing files.

7. Force Overwrite (`-f`)

sh
```
cp -f file.txt /backup/
```

- **Overwrites** without asking (use with caution).

8. Copy Only Newer Files (`-u`)

sh
```
cp -u source/*.txt backup/
```

- Copies only if the source is **newer** than the destination.

9. Show Progress (`-v` Verbose Mode)

sh
```
cp -v file.txt /backup/
```

- Displays what is being copied:

- `'file.txt' -> '/backup/file.txt'`

10. Create Symbolic Links Instead of Copying (`-s`)

sh
```
cp -s /original/file.txt /link/
```

- Creates a **symlink** (shortcut) instead of copying.

Advanced `cp` Usage
Copy All Files Matching a Pattern
sh
```
cp *.txt /backup/
```

- Copies all `.txt` files to `/backup/`.

Copy Files from Multiple Directories

sh
```
cp dir1/file.txt dir2/file2.txt /backup/
```

Copy Excluding Certain Files (`rsync` Alternative)

sh
```
cp -R --exclude="*.tmp" /source/ /destination/
```

Summary Table

Command	Description
`cp file dest/`	Basic file copy
`cp file newfile`	Copy & rename
`cp -R dir/ dest/`	Copy directory recursively
`cp -i file dest/`	Confirm before overwrite
`cp -f file dest/`	Force overwrite
`cp -p file dest/`	Preserve permissions
`cp -u file dest/`	Update if newer
`cp -v file dest/`	Verbose mode (show progress)
`cp -s file link/`	Create symlink instead

Common Errors & Fixes

Error	Solution
`cp: cannot stat 'file': No such file`	Check file path
`cp: -r not specified; omitting directory`	Use `-R` for directories
`cp: overwrite 'file'?`	Use `-i` (interactive) or `-f` (force)
`Permission denied`	Use `sudo` (if admin) or check permissions

Final Notes

- By default, `cp` **overwrites** existing files without warning (use `-i` for safety).
- For **large-scale copying**, consider `rsync` for better performance.
- Use `man cp` for full documentation.

mv command

`mv` Command in Linux (Move/Rename Files & Directories)

The `mv` (move) command is used to **move or rename files and directories** in Linux/Unix systems. It is also used to **overwrite files** (unlike `cp`, which creates copies).

Basic Syntax

sh
```
mv [OPTIONS] SOURCE DEST
mv [OPTIONS] SOURCE... DIRECTORY
```

Key Features

- **Moves** files/directories from `SOURCE` to `DEST`.
- **Renames** files/directories if `DEST` is a new name.
- Can **overwrite** existing files (use `-i` for confirmation).
- Works **faster than `cp`** (no data duplication, just metadata update).

Common `mv` Usage Examples

1. Move a File to Another Directory

sh
```
mv file.txt /backup/
```

- Moves `file.txt` to `/backup/`.

2. Rename a File

sh
```
mv oldname.txt newname.txt
```

- Renames `oldname.txt` to `newname.txt`.

3. Move Multiple Files to a Directory

sh
```
mv file1.txt file2.txt /backup/
```

- Moves both files to `/backup/`.

4. Move a Directory

sh
```
mv old_dir/ new_dir/
```

- Renames `old_dir/` to `new_dir/` (if `new_dir/` doesn't exist).
- If `new_dir/` exists, moves `old_dir/` inside it.

5. Interactive Mode (`-i` – Confirm Before Overwriting)

sh
```
mv -i file.txt /backup/
```

- Asks before overwriting existing files.

6. Force Overwrite (`-f`)

sh
```
mv -f file.txt /backup/
```

- **Overwrites** without asking (use with caution).

7. Show Progress (`-v` Verbose Mode)

sh
```
mv -v file.txt /backup/
```

- Displays what is being moved:

- `'file.txt' -> '/backup/file.txt'`

8. Move Only If Newer (`-u`)

sh
```
mv -u file.txt /backup/
```

- Moves only if `file.txt` is **newer** than the destination.

9. Prevent Overwriting (`-n`)

sh
```
mv -n file.txt /backup/
```

- **Does not overwrite** if the destination exists.

Advanced `mv` Usage

Move All Files Matching a Pattern

sh
```
mv *.txt /backup/
```

- Moves all `.txt` files to `/backup/`.

Move Files from Multiple Directories

sh
```
mv dir1/file.txt dir2/file2.txt /backup/
```

Rename with Wildcards

sh
```
mv photo_*.jpg vacation_*.jpg
```

- Renames `photo_1.jpg` → `vacation_1.jpg`, etc.

Summary Table

Command	Description
`mv file dest/`	Basic move
`mv file newname`	Rename file
`mv dir/ newdir/`	Rename/move directory
`mv -i file dest/`	Confirm before overwrite
`mv -f file dest/`	Force overwrite
`mv -v file dest/`	Verbose mode (show progress)
`mv -u file dest/`	Move only if newer
`mv -n file dest/`	No overwrite

Common Errors & Fixes

Error	Solution
`mv: cannot stat 'file': No such file`	Check file path
`mv: permission denied`	Use `sudo` (if admin) or check permissions
`mv: overwrite 'file'?`	Use `-i` (interactive) or `-f` (force)
`mv: target is not a directory`	Use `/` at the end of directory names

Final Notes

- `mv` **does not create copies** (unlike `cp`), so **files are deleted from the source**.
- By default, `mv` **overwrites** existing files without warning (use `-i` for safety).
- Use `man mv` for full documentation.

rm command

`rm` Command in Linux (Remove Files & Directories)

The `rm` (remove) command is used to **delete files and directories** in Linux/Unix systems. It is a powerful command that **permanently deletes** data (unlike moving to a "trash" bin).

Basic Syntax

sh
```
rm [OPTIONS] FILE...
rm [OPTIONS] DIRECTORY...
```

Key Features

- **Deletes files permanently** (no easy recovery).
- Can **delete directories recursively** (`-r` or `-R`).
- Supports **force deletion** (`-f`) and **interactive prompts** (`-i`).
- **Use with caution** – deleted files are not recoverable by default.

Common `rm` Usage Examples

1. Delete a Single File
sh
```
rm file.txt
```

- Permanently removes `file.txt`.

2. Delete Multiple Files

sh
```
rm file1.txt file2.txt file3.txt
```

- Deletes all listed files.

3. Delete All Files Matching a Pattern

sh
```
rm *.tmp
```

- Removes all files ending with `.tmp`.

4. Delete a Directory Recursively (`-r` or `-R`)

sh
```
rm -r my_folder/
```

- Deletes `my_folder/` and **all its contents** (subdirectories & files).

5. Force Deletion (`-f`) – No Warnings

sh
```
rm -f file.txt
```

- **Skips confirmation prompts** (dangerous but useful in scripts).

6. Interactive Mode (`-i`) – Confirm Before Deleting

sh
```
rm -i *.log
```

- Asks for confirmation before each deletion:

- `rm: remove regular file 'error.log'? y`

7. Verbose Mode (`-v`) – Show Deleted Files

sh
```
rm -v old_*
```

- Prints each deleted file:

- `removed 'old_report.txt'`
- `removed 'old_data.csv'`

8. Delete Empty Directories (`rmdir` Alternative)

sh
```
rm -d empty_dir/
```

- Only works if the directory is **empty**.

Advanced `rm` Usage

Delete Files Older Than X Days

sh
```
find /path/to/files -mtime +30 -exec rm {} \;
```

- Deletes files **older than 30 days**.

Delete Files Except Certain Ones

sh
```
rm !(important.txt|keep_this.file)
```

- Uses **extended globbing** (enable with `shopt -s extglob` in Bash).

Delete Large Files Safely (Overwrite First)

sh
```
shred -u -z -n 5 sensitive_file.txt
```

- Securely deletes by overwriting (`shred` is safer for sensitive data).

Summary Table

Command	Description
`rm file`	Delete a file
`rm file1 file2`	Delete multiple files
`rm *.tmp`	Delete files matching a pattern
`rm -r dir/`	Delete directory recursively
`rm -f file`	Force delete (no warnings)
`rm -i file`	Interactive confirmation
`rm -v file`	Verbose mode (show deletions)
`rm -d dir/`	Delete empty directory

Common Errors & Fixes

Error	Solution
`rm: cannot remove 'file': No such file`	Check file path
`rm: cannot remove 'dir/': Is a directory`	Use `-r` for directories
`rm: permission denied`	Use `sudo` (if admin) or check permissions
`Argument list too long`	Use `find` or `xargs` for many files

Final Notes

- **`rm` does not move files to trash** – deletions are **permanent**.
- For safer deletions, consider:
  - `trash-cli` (moves files to trash).
  - `alias rm='rm -i'` (always prompt before deleting).
- Use `man rm` for full documentation.

mkdir

`mkdir` Command in Linux (Create Directories)

The `mkdir` (make directory) command is used to **create new directories** in Linux/Unix systems. It supports creating single or multiple directories at once, with options to set permissions and create parent directories as needed.

Basic Syntax

sh
```
mkdir [OPTIONS] DIRECTORY...
```

Key Features

- Creates **new directories** (folders).
- Can create **multiple directories** at once.
- Supports **parent directory creation** (`-p` flag).
- Allows **setting permissions** (`-m` flag).

Common `mkdir` Usage Examples

1. Create a Single Directory

sh
```
mkdir new_folder
```

- Creates `new_folder` in the current directory.

2. Create Multiple Directories

sh
```
mkdir dir1 dir2 dir3
```

- Creates `dir1`, `dir2`, and `dir3` in the current directory.

3. Create Nested Directories (`-p` Parent Flag)

sh
```
mkdir -p projects/code/python
```

- Creates the full path (`projects/code/python`) even if parent directories don't exist.

4. Set Directory Permissions (`-m` Mode Flag)

sh
```
mkdir -m 755 secure_dir
```

- Creates `secure_dir` with permissions `rwxr-xr-x` (755).

5. Verbose Mode (`-v` – Show Created Directories)

sh
```
mkdir -v temp
```

- Outputs:

- `mkdir: created directory 'temp'`

6. Create Multiple Nested Directories

sh
```
mkdir -p data/{logs,backups,config}
```

- Creates:

- `data/logs`
- `data/backups`
- `data/config`

Advanced `mkdir` Usage

Create Directories with Spaces in Names

sh
```
mkdir "My Documents"
```

- Quotes prevent the space from being interpreted as a separator.

Create a Directory with a Specific Group Ownership

sh
```
mkdir shared_dir && chgrp team shared_dir
```

- First creates `shared_dir`, then sets its group to `team`.

Use Variables to Name Directories

sh
```
dirname="backup_$(date +%F)"
mkdir "$dirname"
```

- Creates a directory like `backup_2023-12-25`.

Summary Table

Command	Description
`mkdir dir`	Create a single directory
`mkdir dir1 dir2`	Create multiple directories
`mkdir -p path/to/dir`	Create nested directories
`mkdir -m 750 dir`	Set permissions (750 = `rwxr-x---`)
`mkdir -v dir`	Verbose mode (show creation)
`mkdir --help`	Show help message

Common Errors & Fixes

Error	Solution
`mkdir: cannot create directory 'dir': File exists`	Directory already exists (use `-p` to ignore)
`mkdir: cannot create directory 'dir': Permission denied`	Use `sudo` or check parent directory permissions
`mkdir: missing operand`	Specify at least one directory name

Final Notes

- By default, `mkdir` **does not create parent directories** (use `-p` for that).
- To **remove directories**, use `rmdir` (empty dirs) or `rm -r` (non-empty dirs).
- Use `man mkdir` for full documentation.