02 Linux File Commands & Git Fundamentals

Complete Guide: Linux File Commands & Git Fundamentals

Part 1: Introduction to Git Bash

What is Git Bash?

Git Bash is a command-line interface for Windows that emulates a Unix-like terminal environment. It allows you to use Git commands and Linux/Unix commands on Windows systems.

Why Learn Command Line?

- **Efficiency**: Execute tasks faster than GUI
- **Automation**: Script repetitive tasks
- **Remote Access**: Essential for server management
- **Version Control**: Industry-standard for code collaboration

---

Part 2: Linux File Commands in Git Bash

2.1 Navigation Commands

`pwd` - Print Working Directory

Shows your current location in the file system.

```shell
$ pwd
/c/Users/YourName/Documents
```

Intuition: Think of this as asking "Where am I?" in the file system.

`ls` - List Directory Contents

Displays files and folders in the current directory.

```shell
# Basic listing
$ ls

# Detailed listing (permissions, size, date)
$ ls -l

# Show hidden files (files starting with .)
$ ls -a

# Combined: detailed + hidden
$ ls -la

# Human-readable file sizes
$ ls -lh
```

Intuition: Like opening a folder in Windows Explorer, but in text form.

`cd` - Change Directory

Navigate between folders.

```shell
# Move into a folder
$ cd Documents

# Move up one level
$ cd ..

# Move up two levels
$ cd ../..

# Go to home directory
$ cd ~

# Go to root directory
$ cd /

# Return to previous directory
$ cd -
```

Intuition: Walking through rooms in a house. `cd` moves you between rooms, `..` is the door to the previous room.

2.2 File and Directory Creation

`mkdir` - Make Directory

Creates new folders.

```shell
# Create single directory
$ mkdir my_project

# Create multiple directories
$ mkdir folder1 folder2 folder3

# Create nested directories
$ mkdir -p parent/child/grandchild
```

The `-p` flag: Creates all parent directories if they don't exist. Without it, you'd need to create each level separately.

`touch` - Create Empty Files

Creates new files or updates timestamps.

```shell
# Create single file
$ touch index.html

# Create multiple files
$ touch file1.txt file2.txt file3.txt

# Create file with path
$ touch documents/notes.txt
```

Intuition: Like creating a new blank document in Windows.

2.3 File Operations

`cp` - Copy Files/Directories

```shell
# Copy file
$ cp source.txt destination.txt

# Copy to different directory
$ cp file.txt ../backup/

# Copy directory recursively
$ cp -r folder1 folder2

# Copy multiple files to directory
$ cp file1.txt file2.txt folder/
```

The `-r` flag: Recursive - copies folders and all their contents.

`mv` - Move/Rename Files

```shell
# Rename file
$ mv oldname.txt newname.txt

# Move file to directory
$ mv file.txt documents/

# Move and rename
$ mv file.txt documents/newname.txt

# Move multiple files
$ mv file1.txt file2.txt folder/
```

Intuition: `mv` is like cut-paste in Windows. It removes from source and places in destination.

`rm` - Remove Files/Directories

```shell
# Delete file
$ rm file.txt

# Delete multiple files
$ rm file1.txt file2.txt

# Delete directory and contents
$ rm -r folder_name

# Force delete (no confirmation)
$ rm -rf folder_name

# Interactive delete (asks confirmation)
$ rm -i file.txt
```

⚠ **WARNING**: `rm` is permanent! There's no recycle bin. Be extremely careful with `rm -rf`.

2.4 Viewing and Editing Files

`cat` - Concatenate and Display Files

```shell
# Display file contents
$ cat file.txt

# Display multiple files
$ cat file1.txt file2.txt

# Number all lines
$ cat -n file.txt
```

`less` - View Files Page by Page

```shell
$ less largefile.txt
```

Navigation in `less`:

- `Space`: Next page
- `b`: Previous page
- `/search_term`: Search forward

- `q`: Quit

`head` and `tail` - View File Portions

```shell
# First 10 lines
$ head file.txt

# First 20 lines
$ head -n 20 file.txt

# Last 10 lines
$ tail file.txt

# Last 20 lines
$ tail -n 20 file.txt

# Follow file in real-time (useful for logs)
$ tail -f logfile.txt
```

`echo` - Display Text or Write to Files

```shell
# Print to terminal
$ echo "Hello World"

# Write to file (overwrites)
$ echo "Hello World" > file.txt

# Append to file
$ echo "New line" >> file.txt

# Display variable
$ echo $PATH
```

2.5 File Permissions (Understanding ls -l)

When you run `ls -l`, you see:

```
-rw-r--r-- 1 user group 1234 Jan 15 10:30 file.txt
```

Breaking it down:

- `-rw-r--r--`: Permissions
  - First character: File type (`-` = file, `d` = directory)
  - Next 3 (`rw-`): Owner permissions (read, write, no execute)
  - Next 3 (`r--`): Group permissions (read only)
  - Last 3 (`r--`): Others permissions (read only)
- `1`: Number of links
- `user`: Owner name
- `group`: Group name
- `1234`: File size in bytes
- `Jan 15 10:30`: Last modified date/time
- `file.txt`: Filename

2.6 Search and Find

`grep` - Search Inside Files

SHELL

```
# Search for text in file
$ grep "search_term" file.txt

# Case-insensitive search
$ grep -i "search_term" file.txt

# Search recursively in directory
$ grep -r "search_term" folder/

# Show line numbers
$ grep -n "search_term" file.txt

# Count matches
$ grep -c "search_term" file.txt
```

`find` - Locate Files

```shell
# Find files by name
$ find . -name "*.txt"

# Find directories
$ find . -type d -name "folder_name"

# Find files modified in last 7 days
$ find . -mtime -7

# Find files larger than 1MB
$ find . -size +1M
```

Part 3: Git Fundamentals {#git-fundamentals}

3.1 Understanding Version Control

What is Git? Git is a distributed version control system that tracks changes in your code over time.

Why Git?

- **Track Changes**: See what changed, when, and by whom
- **Collaboration**: Multiple people can work simultaneously
- **Backup**: Every clone is a full backup
- **Branching**: Experiment without affecting main code
- **History**: Revert to any previous state

3.2 Git Configuration

Before using Git, configure your identity:

```shell
# Set your name
$ git config --global user.name "Your Name"

# Set your email
$ git config --global user.email "your.email@example.com"

# View configuration
$ git config --list

# View specific setting
$ git config user.name
```

The `--global` flag: Applies settings to all repositories. Without it, settings apply only to current repository.

3.3 Creating a Repository

Method 1: Initialize New Repository

```shell
# Create project folder
$ mkdir my_project
$ cd my_project

# Initialize Git repository
$ git init
```

What happens: Git creates a hidden `.git` folder that stores all version history.

Method 2: Clone Existing Repository

```shell
# Clone from GitHub/remote
$ git clone https://github.com/username/repository.git

# Clone to specific folder name
$ git clone https://github.com/username/repository.git my_folder
```

3.4 The Git Workflow: Three States

Understanding Git requires knowing the three states of files:

1. **Working Directory**: Your actual files where you make changes
2. **Staging Area (Index)**: Files marked to be included in next commit
3. **Repository (.git directory)**: Committed snapshots of your project

```
Working Directory → (git add) → Staging Area → (git commit) → Repository
```

Intuition: Think of it like preparing a package:

- **Working Directory**: Items scattered on your desk
- **Staging Area**: Items you've put in the box
- **Repository**: Sealed packages in storage

3.5 Basic Git Commands

`git status` - Check Repository State

```shell
$ git status
```

Output explains:

- Which branch you're on
- Which files are modified
- Which files are staged
- Which files are untracked

Always run `git status` - it's your compass in Git.

`git add` - Stage Changes

```shell
# Stage single file
$ git add file.txt

# Stage multiple files
$ git add file1.txt file2.txt

# Stage all changes
$ git add .

# Stage all txt files
$ git add *.txt

# Stage directory
$ git add folder/
```

Intuition: Selecting which changes you want to include in your next snapshot (commit).

`git commit` - Save Snapshot

```shell
# Commit with message
$ git commit -m "Add login feature"

# Commit with detailed message (opens editor)
$ git commit

# Stage and commit modified files (not new files)
$ git commit -am "Update documentation"
```

Writing Good Commit Messages:

- Start with a verb: "Add", "Fix", "Update", "Remove"
- Be specific: "Fix login button alignment" not "Fix bug"
- Present tense: "Add feature" not "Added feature"

`git log` - View History

```shell
# Full log
$ git log

# Compact view
$ git log --oneline

# Last 5 commits
$ git log -5

# Graphical view
$ git log --graph --oneline --all

# Commits by author
$ git log --author="John"
```

3.6 Understanding Branches

What is a Branch? A branch is an independent line of development. It's like creating an alternate timeline where you can experiment without affecting the main project.

Visual Representation:

```
main:     A---B---C---F---G
               \       /
feature:        D---E
```

Why Branch?

- **Isolation**: Work on features without breaking main code
- **Collaboration**: Multiple features developed simultaneously
- **Experimentation**: Try ideas without commitment
- **Organization**: Separate concerns (features, bugs, experiments)

`git branch` - Manage Branches

```shell
# List all branches (* indicates current)
$ git branch

# Create new branch
$ git branch feature-login

# Delete branch (safe - won't delete if unmerged)
$ git branch -d feature-login

# Force delete branch
$ git branch -D feature-login

# Rename current branch
$ git branch -m new-name

# List all branches with last commit
$ git branch -v
```

`git checkout` - Switch Branches

```shell
# Switch to existing branch
$ git checkout feature-login

# Create and switch to new branch
$ git checkout -b new-feature

# Switch back to main/master
$ git checkout main

# Discard changes in file (restore from last commit)
$ git checkout -- file.txt
```

⚠ **Note**: Newer Git versions use `git switch` for branch switching:

```shell
$ git switch feature-login
$ git switch -c new-feature  # create and switch
```

Understanding Checkout Deeply

When you `checkout` a branch:

1. Git updates your working directory to match that branch's state
2. Git points HEAD to that branch
3. New commits will be added to this branch

HEAD: A pointer to the current branch/commit you're on.

3.7 Merging Branches

`git merge` - Combine Branches

```shell
# First, switch to branch you want to merge INTO
$ git checkout main

# Then merge the feature branch
$ git merge feature-login
```

What Happens During Merge:

1. Git finds the common ancestor
2. Git combines changes from both branches
3. Git creates a new merge commit

Types of Merges

1. Fast-Forward Merge Happens when target branch hasn't changed since feature branch was created.

```
Before:

main:     A---B
               \
feature:        C---D

After:

main:     A---B---C---D
```

2. Three-Way Merge Happens when both branches have new commits.

```
Before:
main:     A---B---E
               \
feature:        C---D

After:
main:     A---B---E---F (merge commit)
               \       /
feature:        C---D
```

Handling Merge Conflicts

When Conflicts Occur: Same file, same location, different changes in both branches.

Conflict Markers in File:

```
<<<<<<< HEAD
code from current branch
=======
code from merging branch
>>>>>>> feature-branch
```

Resolution Steps:

SHELL

```
# 1. Open conflicted file
# 2. Edit file to resolve conflicts (remove markers)
# 3. Stage resolved file
$ git add conflicted-file.txt

# 4. Complete merge
$ git commit -m "Merge feature-branch and resolve conflicts"
```

3.8 Working with Remote Repositories

`git remote` - Manage Remote Connections

```shell
# List remotes
$ git remote

# List remotes with URLs
$ git remote -v

# Add remote
$ git remote add origin https://github.com/username/repo.git

# Remove remote
$ git remote remove origin

# Rename remote
$ git remote rename origin upstream

# Show remote details
$ git remote show origin
```

"origin": Conventional name for the main remote repository.

`git push` - Upload to Remote

```shell
# Push current branch to remote
$ git push origin main

# Push and set upstream (first time)
$ git push -u origin main

# Push all branches
$ git push --all

# Push tags
$ git push --tags

# Force push (dangerous - overwrites remote)
$ git push -f origin main
```

What Push Does:

1. Sends your commits to remote repository
2. Updates remote branch to match your local branch

When to Use `-u` (upstream): First time pushing a branch. It links your local branch to remote branch, so later you can just use `git push`.

`git pull` - Download from Remote

```shell
# Pull current branch from remote
$ git pull origin main

# Pull with rebase instead of merge
$ git pull --rebase origin main
```

What Pull Does: `git pull` = `git fetch` + `git merge`

1. Downloads new commits from remote
2. Merges them into your current branch

`git fetch` - Download Without Merging

```shell
# Fetch from origin
$ git fetch origin

# Fetch specific branch
$ git fetch origin main

# Fetch all remotes
$ git fetch --all
```

Fetch vs Pull:

- **Fetch**: Downloads changes but doesn't merge (safe)
- **Pull**: Downloads and automatically merges (can cause conflicts)

Best Practice: Fetch first, review, then merge manually.

3.9 Typical Git Workflow

Daily Workflow

SHELL

```
# 1. Start work: Update from remote
$ git pull origin main

# 2. Create feature branch
$ git checkout -b feature-user-auth

# 3. Make changes to files
#  ...  edit files  ...

# 4. Check what changed
$ git status
$ git diff

# 5. Stage changes
$ git add .

# 6. Commit with message
$ git commit -m "Add user authentication"

# 7. Push to remote
$ git push -u origin feature-user-auth

# 8. Create Pull Request on GitHub/GitLab
#  ...  use web interface  ...

# 9. After review and merge, switch back
$ git checkout main

# 10. Update local main
$ git pull origin main

# 11. Delete local feature branch
$ git branch -d feature-user-auth
```

3.10 Additional Useful Commands

`git diff` - See Changes

```shell
# Changes in working directory (unstaged)
$ git diff

# Changes in staging area
$ git diff --staged

# Changes between branches
$ git diff main feature-branch

# Changes in specific file
$ git diff file.txt
```

`git stash` - Temporarily Save Changes

```shell
# Stash current changes
$ git stash

# Stash with message
$ git stash save "Work in progress"

# List all stashes
$ git stash list

# Apply most recent stash
$ git stash apply

# Apply and remove from stash list
$ git stash pop

# Apply specific stash
$ git stash apply stash@{1}

# Delete stash
$ git stash drop stash@{0}
```

Use Case: You're working on something but need to quickly switch branches to fix a bug.

`git reset` - Undo Changes

```shell
# Unstage file (keep changes in working directory)
$ git reset file.txt

# Undo last commit (keep changes staged)
$ git reset --soft HEAD~1

# Undo last commit (keep changes unstaged)
$ git reset HEAD~1

# Undo last commit (discard changes - DANGEROUS)
$ git reset --hard HEAD~1

# Reset to specific commit
$ git reset --hard abc1234
```

HEAD~1: One commit before HEAD **HEAD~2**: Two commits before HEAD

`git revert` - Undo by Creating New Commit

```shell
# Revert specific commit
$ git revert abc1234

# Revert without auto-commit
$ git revert -n abc1234
```

Reset vs Revert:

- **Reset**: Rewrites history (use for local commits only)
- **Revert**: Creates new commit that undoes changes (safe for shared branches)

Part 4: Practical Exercises {#exercises}

Exercise 1: Linux Commands Practice

SHELL

```
# 1. Create directory structure
$ mkdir -p project/{src,tests,docs}

# 2. Create files
$ touch project/README.md
$ touch project/src/{main.js,utils.js}
$ touch project/tests/test.js

# 3. Navigate and list
$ cd project
$ ls -la

# 4. Add content
$ echo "# My Project" > README.md
$ echo "console.log('Hello');" > src/main.js

# 5. View content
$ cat README.md
$ less src/main.js

# 6. Copy and rename
$ cp README.md docs/
$ mv docs/README.md docs/documentation.md

# 7. Find files
$ find . -name "*.js"
$ grep -r "console" .
```

Exercise 2: Git Basics

SHELL

```
# 1. Initialize repository
$ git init my-app
$ cd my-app

# 2. Configure Git (if not already done)
$ git config user.name "Your Name"
$ git config user.email "your@email.com"

# 3. Create files
$ echo "# My App" > README.md
$ mkdir src
$ echo "function hello() {}" > src/app.js

# 4. Check status
$ git status

# 5. Stage and commit
$ git add .
$ git commit -m "Initial commit"

# 6. Make changes
$ echo "New line" >> README.md

# 7. View diff
$ git diff

# 8. Stage and commit
$ git add README.md
$ git commit -m "Update README"

# 9. View history
$ git log --oneline
```

Exercise 3: Branching and Merging

```shell
# 1. Create new branch
$ git checkout -b feature-navbar

# 2. Make changes
$ echo "navbar code" > src/navbar.js
$ git add .
$ git commit -m "Add navbar feature"

# 3. Switch back to main
$ git checkout main

# 4. Make different change on main
$ echo "footer code" > src/footer.js
$ git add .
$ git commit -m "Add footer"

# 5. Merge feature branch
$ git merge feature-navbar

# 6. View combined history
$ git log --graph --oneline --all

# 7. Delete feature branch
$ git branch -d feature-navbar
```

Exercise 4: Working with Remote (GitHub)

```shell
# 1. Create repository on GitHub (use web interface)

# 2. Add remote
$ git remote add origin https://github.com/username/my-app.git

# 3. Push to remote
$ git push -u origin main

# 4. Make local changes
$ echo "Update" >> README.md
$ git add .
$ git commit -m "Update README"

# 5. Push changes
$ git push

# 6. Simulate remote changes (use GitHub web editor to edit file)

# 7. Pull remote changes
$ git pull origin main
```

Exercise 5: Handling Conflicts

SHELL

```
# 1. Create branch
$ git checkout -b feature-conflict

# 2. Edit file
$ echo "Line from feature" >> README.md
$ git add .
$ git commit -m "Update from feature"

# 3. Switch to main
$ git checkout main

# 4. Edit same file
$ echo "Line from main" >> README.md
$ git add .
$ git commit -m "Update from main"

# 5. Try to merge (will cause conflict)
$ git merge feature-conflict

# 6. View conflict
$ cat README.md

# 7. Resolve conflict (edit file manually)
$ nano README.md  # or use any text editor

# 8. Stage and complete merge
$ git add README.md
$ git commit -m "Resolve merge conflict"
```

Quick Reference Cheat Sheet

Linux Commands

```shell
# Navigation
pwd                     # Current directory
ls -la                  # List with details
cd folder               # Change directory
cd ..                   # Up one level

# Files/Folders
mkdir folder            # Create directory
touch file.txt          # Create file
cp source dest          # Copy
mv old new              # Move/rename
rm file                 # Delete file
rm -r folder            # Delete folder

# Content
cat file                # Display file
less file               # Page through file
head file               # First 10 lines
tail file               # Last 10 lines
echo "text" > file      # Write to file
grep "text" file        # Search in file
```

Git Commands

```shell
# Setup
git config --global user.name "Name"
git config --global user.email "email"
git init                # Initialize repository
git clone url           # Clone repository

# Basic Workflow
git status              # Check status
git add file            # Stage file
git add .               # Stage all
git commit -m "msg"     # Commit
git log --oneline       # View history

# Branching
git branch              # List branches
git branch name         # Create branch
git checkout name       # Switch branch
git checkout -b name    # Create and switch
git merge name          # Merge branch

# Remote
git remote add origin url   # Add remote
git push -u origin main     # Push
git pull origin main        # Pull
git fetch origin           # Fetch only

# Undo
git diff                # View changes
git reset file          # Unstage
git reset --hard        # Discard changes
git stash               # Save temporarily
git stash pop           # Restore stashed
```

Common Pitfalls and Best Practices

Pitfalls to Avoid

1. **Forgetting to commit regularly**: Commit often with clear messages
2. **Working directly on main**: Always create feature branches
3. **Force pushing to shared branches**: Never use `git push -f` on main
4. **Not pulling before starting work**: Always sync before making changes

5. **Committing sensitive data**: Use `.gitignore` for passwords, keys
6. **Unclear commit messages**: "Fix stuff" tells nothing; "Fix login validation" is clear

Best Practices

1. **Commit Often**: Small, logical commits are easier to understand and revert
2. **Branch for Features**: Keep main stable, develop in branches
3. **Pull Before Push**: Avoid conflicts by staying updated
4. **Review Before Committing**: Use `git diff` and `git status`
5. **Write Clear Messages**: Future you will thank present you
6. **Use .gitignore**: Don't commit generated files, dependencies, or secrets
7. **Test Before Merging**: Ensure code works before merging to main

---

Conclusion

You now have a solid foundation in:

- Navigating and manipulating files in the command line
- Understanding Git's architecture and workflow
- Creating and managing branches
- Collaborating with remote repositories

Next Steps:

1. Practice these commands daily
2. Start a personal project using Git
3. Contribute to open-source projects
4. Learn about Git workflows (GitFlow, GitHub Flow)
5. Explore advanced topics (rebasing, cherry-picking, hooks)

Remember: **The best way to learn is by doing!** Start with simple projects and gradually tackle more complex workflows.

Additional Resources

- **Official Git Documentation**: https://git-scm.com/doc
- **GitHub Learning Lab**: https://lab.github.com/
- **Interactive Git Tutorial**: https://learngitbranching.js.org/
- **Git Cheat Sheet**: https://education.github.com/git-cheat-sheet-education.pdf

Good luck on your Git journey! 🚀