History of Linux and the command line

Zubin

January 29, 2026

Contents

1 Operating Systems: Definition and Services

1.1 What is an Operating System?

An **operating system** (**OS**) is an intermediary between computer hardware (memory, processor, network cards, etc.) and the applications that users interact with.

- Users interact with applications.
- Applications request services from the operating system.
- The operating system manages and exploits hardware resources to provide services to the applications.

1.2 Examples of Operating Systems

- Microsoft Windows
- Apple macOS and iOS
- Google Android
- Unix and Unix-like systems such as Linux

1.3 Unix and Linux

- Unix has been around longer than Linux.
- Linux is a **Unix-like, open-source operating system**. The Linux kernel, created by **Linus Torvalds** and expanded upon by thousands of programmers, is available to the world for free.

1.4 Core Services Provided by an Operating System

1.4.1 File Management

- Managing the logical tree structure of files and their physical layout on storage devices (hard drives).

1.4.2 Memory Management

- Allocation, deallocation, and sharing of memory among multiple running processes.

1.4.3 Process Management

- Creation, execution, and termination of running applications (processes).

1.4.4 Input/Output Management

- Managing hardware like network interfaces, sound cards, video cards, printers, and other peripherals.

---

2 Genesis of Operating Systems

2.1 Early Computers (Mid-1940s)

- The first computers were built using **vacuum tubes** (evacuated glass containers that control electric current).
- These were huge machines that filled entire rooms but performed more slowly than a modern hand-held calculator.

2.2 Programming and Operation

- Programming was done manually by rearranging hardware components.
- Input/output capabilities were very limited.
- A single individual often acted as the designer, builder, programmer, and operator.

2.3 Invention of the Transistor

- The invention of the transistor led to smaller, more reliable computers.
- This innovation marked the beginning of operating systems through the appearance of **punch cards**.

2.4 Punch Cards and Role Separation

- Punch cards are cards with holes in specific locations to encode computer programs and data.
- This led to a separation of roles: programmers prepared the punch cards, and operators physically loaded them into the computer and handled the output.

2.5 Birth of Operating Systems

- Operating systems were invented to manage memory, processes (running programs), and input/output operations like reading punch cards.
- We can date the invention of operating systems to the **mid-1960s**.

---

3 UNIX Genesis

3.1 Technological Context

- The era of modern computers emerged with the appearance of **integrated circuits** and magnetic disks.
- This period also saw the development of compatible computer families, such as the **IBM System/360** (1964), which made a clear distinction between architecture and implementation.

3.2 Project MAC at MIT

- It all started with **Project MAC** (Mathematics and Computation), founded at MIT.
- It was funded by the US military's research agency (ARPA) and the National Science Foundation.
- The main goal was to develop a **timesharing system** that would allow a large community of users to access a single computer from multiple locations simultaneously.

3.3 MULTICS

- Developed by MIT, Bell Labs, and General Electric
- Stands for **Multiplexed Information and Computing Service**
- It evolved beyond timesharing to incorporate features like file sharing, file management, and system security.

3.4 Challenges of MULTICS

- The project proved much more difficult than expected.
- The system became operational in 1969 on the GE-645 computer, but its performance was far below the original targets.
- As a result, Bell Labs withdrew from the project in 1969.

3.5 Birth of UNIX

- Following the withdrawal, Bell Labs engineers **Ken Thompson** and **Dennis Ritchie** decided to create a simpler, minimal system.
- Using a little-used DEC PDP-7 machine, they began developing a single-user operating system.
- As a pun on the complexity of MULTICS, they called their system **UNICS**.

3.6 Evolution of UNIX

- In 1970, the system was enhanced to support multiple users, and its name morphed to **Unix**.
- At the time, the system was written in the **B programming language**, which was invented by Ken Thompson.

3.7 The C Programming Language

- In 1971, Dennis Ritchie improved upon B and called it **New B**.
- By 1972, the changes were so significant that Ritchie renamed his new language the **C programming language**.
- Ken Thompson then rewrote the entire Unix operating system in C.

3.8 Spread of UNIX

- The C source code for Unix was distributed to universities and research centers for educational purposes.
- From 1975 onward, a very active community emerged around Unix and C.
- Other notable developers included:
  - Douglas McIlroy (McElroy)
  - Joseph Ossanna
  - Rudd Canaday

3.9 Key Milestones

- 1978: Brian Kernighan and Dennis Ritchie published the book *The C Programming Language*.
- 1983: Thompson and Ritchie received the **Turing Award**, the highest distinction in computer science, for their invention.

3.10 Legacy of UNIX

- The concepts introduced by Unix are ubiquitous today and form the foundation for many modern operating systems.
- Derivatives of Unix include:
  - macOS
  - iOS
  - Android

– Linux, which is installed on the vast majority of today's servers and connected objects.

---

4 Linux genesis and history: GNU, Stallman, GPL, Linus Torvalds, Linux

4.1 The GNU Project

- In 1983, the same year Ritchie and Thompson received the Turing Award, **Richard Stallman** of MIT launched the **GNU Project**.
- GNU is a recursive acronym standing for **GNU is Not Unix**.
- The project aimed to develop a free, open, and collaborative software system compatible with Unix, contrasting with the proprietary nature of Unix owned by Bell Labs.

4.2 Richard Stallman and the GPL

- Richard Stallman is a strong advocate for free and open-source software.
- In 1989, he conceived the **GNU General Public License** (**GPL**), designed to preserve the freedom to use, study, modify, and distribute software.
- By 1990, the GNU Project had created many tools (text editors, GUI, libraries, and the **GNU C Compiler** (**GCC**)), but it lacked a free operating system kernel to run them on.

4.3 Linus Torvalds and the Linux Kernel

- **Linus Torvalds**, a student at the University of Helsinki, was frustrated by proprietary OS licenses.
- On August 25, 1991, he announced a "free operating system" project (just a hobby) to the community.
- This became the **Linux kernel**, developed on an 80386 processor using the GNU C Compiler.

4.4 Integration and Growth

- A community formed quickly, integrating GNU software with the Linux kernel.
- In 1992, the first **Linux distributions** were released (Linux kernel + GNU tools).

- By 1993, there were over 100 developers, and popular distributions like **Debian** emerged.

4.5 Widespread Adoption

- **Late 1990s:** Major manufacturers (Dell, IBM, HP) announced Linux compatibility.
- **2000s:** Increasing deployment on web servers.
- **2010s:** Linux and Unix-based systems dominated:
  - Internet servers (70%)
  - Smartphones (90%, via Android and iOS)
  - Supercomputers (99%)
- Linux is also prevalent in game consoles, routers, and IoT devices.

---

5 Command line interface, prompt, command options and files data, command cal as example

5.1 Definition and Interaction

- A **Command Line Interface** (**CLI**) is a human-machine interface where communication takes place in text mode.
- The user types a command to request an operation, and the computer displays the result or further questions in text.
- It is central to the interaction between users and computing equipment.

5.2 The Command Prompt

- When ready to receive input, the system displays a **command prompt**.
- This prompt usually contains information like the user's name, computer name, current directory, or date.
- It ends with a character such as `$`, `#`, or `>`.

5.3 Command Structure

- Unix and Linux systems include hundreds of simple applications usable from the CLI.
- The basic structure of a command is: `command options files_or_data`

- **Command:** The name of the application (often written in C).
- **Options:** Modify the command's execution (separated by spaces).
- **Files or Data:** The inputs for the program.

5.4 Example: The `cal` Command

- Typing `cal` and pressing Enter runs the application that displays a calendar.
- Adding the `-j` option (e.g., `cal -j`) displays the calendar in Julian days (number of days elapsed since January 1st).

---

6 First commands: echo 'hello world', date, cal, history, whoami, hostname, uptime, clear, command not found, man, command options

6.1 The Command Prompt

The command prompt is the visual indicator that the system is ready to receive input. On WebLinux, it appears as a tilde, a space, a dollar sign, and a space (`~ $`), followed by a blinking cursor. When you type a command and press **Enter**, the shell interprets the command and displays the result.

6.2 User and System Identification

- **whoami**: Displays the current username. On WebLinux, this is simply `user`.
- **logname**: Similar to `whoami`, it prints the name of the current user.
- **id**: Displays the user ID (UID) and group ID (GID). For the default user, these are typically 1000.
- **hostname**: Displays the name of the computer (e.g., `openrisk`).
- **uname**: Prints system information. By default, it prints the kernel name (`Linux`).
- **uname -a**: The `-a` option prints all system information, including the network node hostname, kernel release, and version.

6.3 Basic Interaction and Output

- **echo**: Prints the arguments passed to it. For example, `echo hello` prints "hello".
- **echo $0**: Displays the name of the shell interpreter (e.g., `sh`).
- **clear**: Clears the terminal screen.

- **history**: Displays a list of previously executed commands. You can use the **Up** and **Down** arrow keys to navigate through this history.

6.4 Time and Date

- **uptime**: Displays how long the system has been running, along with the current time and load averages.
- **cal**: Displays a calendar. The `-j` option displays Julian dates (days numbered from 1 to 366).
- **date**: Displays the current date and time. It supports formatting options:
  - `date +"%T"`: Displays time only.
  - `date +"%A %d %B %Y"`: Displays full weekday, day, month, and year.

6.5 Getting Help and Manuals

- **–help**: Many commands support this option to display usage summaries (e.g., `whoami -help`).
- **man**: The manual command. Typing `man <command>` usually opens the manual page for that command.
  - On full Linux systems, this provides detailed documentation.
  - On WebLinux, manual pages may be removed to save space, resulting in "no manual entry".
  - Press **q** to exit the manual viewer and return to the prompt.

---

7 Interactive commands: top, htop, nano, vim, how to get back to the prompt

7.1 Interactive Programs

Unlike basic commands that print output and return to the prompt immediately, interactive commands launch programs that take over the terminal interface. The command prompt (e.g., `~ $`) disappears, and the user must interact with the running program. To return to the command line, one must know how to exit these specific applications.

7.2 System Monitoring: top and htop

- **top**: Provides a real-time view of process activity, CPU usage, and memory usage.

- **To Exit:** Press **q** or use the interrupt key combination **Ctrl+C** (often denoted as `^C`).

- **htop**: A more advanced, colorful version of `top` with graphical bars for system resources.
  - **To Exit:** Press **F10**, **q**, or **Ctrl+C**.

7.3 Text Editors: nano and vim

- **nano**: A simple file editor. It displays a shortcut menu at the bottom where the caret symbol (`^`) represents the Control key.
  - **To Exit:** Press **Ctrl+X** (indicated as `^X`).
  - Note: **Ctrl+C** does not exit nano.
- **vim**: A famous but complex editor. It operates in different modes (e.g., Insert mode for typing, Normal mode for commands).
  - **To Exit:**
    1. Press **Esc** to ensure you are in Normal mode.
    2. Type `:q` (colon, q) and press **Enter**.
    3. If changes were made and you want to force quit without saving, type `:q!` and press **Enter**.

7.4 Summary of Exit Strategies

If you are stuck in a program and want to return to the prompt:

1. Try pressing **q** (quit).
2. Try pressing **Ctrl+C** (interrupt).
3. Look for on-screen help (e.g., `^X` or **F10**).
4. If in `vim`, use **Esc** followed by `:q!`.

---

8 Filesystem

8.1 Evolution of Storage

One role of the operating system is to manage files. Historically, data was stored on punch cards, where huge physical volumes were required to store what now fits on a small flash drive (e.g., 4.3 billion characters). The advent of disk drives revolutionized programming by allowing millions of files to be stored and accessed instantly without physical handling.

8.2 The File System Tree

The file system is organized as a tree hierarchy:

- **Root** (`/`): The starting point of the file system.
- **Directories**: Files are grouped into folders. A path is denoted using slashes, e.g., `/folder/subfolder/file`.

8.3 Linux Directory Structure

Based on the Filesystem Standard (FSSTND) initiated in 1993, most Linux distributions use the following directories:

- **/bin**: Basic executable commands (binaries) for a minimal system.
- **/sbin**: System binaries for the **super user** (root).
- **/home**: Contains directories for standard users (e.g., `/home/user`).
- **/root**: The home directory specifically for the root user.
- **/etc**: Configuration files. Stands for **Editable Text Configuration**.
- **/lib**: Libraries required by binaries in `/bin` and `/sbin`.
- **/tmp**: Temporary files. Content is usually deleted upon restart.
- **/var**: Variable files such as logs (`/var/log`), databases, and emails.
- **/usr**: **Unix System Resources** (not "user"). Contains non-essential applications and libraries.
- **/dev**: Device files. Linux treats devices as files (e.g., `/dev/mem`, `/dev/audio`).

---

9 pwd, cd, ls, absolute path, relative path (1)

9.1 Navigating the File System

Navigation is a core skill in the Linux command line. The primary commands for this are `pwd`, `ls`, and `cd`.

9.2 Print Working Directory (pwd)

- **pwd**: Stands for "print working directory".
- It displays the absolute path of the directory you are currently located in.
- When you first log in, you are typically in your home directory (e.g., `/home/user`).
- The tilde (`~`) in the command prompt represents this home directory.

9.3 Listing Files (ls)

- **ls**: Lists the files and directories in the current working directory.
- **ls -a**: Lists all files, including **hidden files**.
  - Hidden files and directories start with a dot (e.g., `.config`).
  - This view reveals the special directories `.` (current) and `..` (parent).
- **ls -al**: Combines options to show a detailed list of all files, including hidden ones.
  - Lines starting with `d` are directories.
  - Lines starting with `-` are regular files.

9.4 Changing Directories (cd)

- **cd**: Stands for "change directory".
- **cd /**: Moves to the root directory.
- **cd** (without arguments): Returns to the user's home directory.
- **cd ..**: Moves up one level to the parent directory.

9.5 Absolute vs. Relative Paths

- **Absolute Path**:
  - Always starts with the root directory slash (`/`).
  - Describes the full path from the root to the destination (e.g., `/home/user`, `/sys/module`).
  - It works from anywhere in the file system.
- **Relative Path**:
  - Does **not** start with a slash.
  - Describes the path relative to the current working directory.
  - `.` (dot): Refers to the current directory (e.g., `./sys` is the same as `sys`).
  - `..` (dot dot): Refers to the parent directory.
  - You can chain them (e.g., `../../` moves up two levels).

---

10 pwd, cd, ls, absolute path, relative path (2)

This section covers how to determine a file's type, find the absolute path of a file or directory, and locate an installed program on the system.

10.1 Getting File Type (file)

The `file` command determines the type of a file.

- **Source Code:** Running `file program.c` on a C source file will identify it as such.

```
$ file program.c
program.c:  C source, ASCII text
```

- **Plain Text:** After creating a new file named `f` with some text using an editor like `nano`, the `file` command will identify it as ASCII text.

```
$ file f
f:  ASCII text
```

- **Directory:** The command can also identify directories.

```
$ file /bin
/bin:  directory
```

- **Symbolic Link:** Some files are symbolic links, which are pointers to other files. The `file` command reveals this relationship.

```
$ file /bin/cat
/bin/cat:  symbolic link to /bin/busybox
```

- **Executable Binary:** Following the symbolic link, we can inspect the actual program file. Executable files are often in ELF (Executable and Linkable Format).

```
$ file /bin/busybox
/bin/busybox:  ELF 32-bit LSB executable, OpenRISC...
```

  This output indicates it is a 32-bit executable program for the OpenRISC processor architecture.

- **Other System Files:** The file system contains many types of files. For example, in `/sys/kernel`:
  - `file notes` might return something complex like `X11 SNF font data, MSB first`.
  - `file fscaps` might return something simple like `ASCII text`.

10.2 Finding the Real Path (realpath)

The `realpath` command resolves all symbolic links and references to `.` and `..` to print the canonical absolute path of a file or directory.

- For a simple file in the home directory, it returns the full path.

```
$ realpath f
/home/user/f
```

- For a symbolic link, it returns the path of the file it points to.

```
$ realpath /bin/cat
/bin/busybox
```

10.3 Locating Executables (which)

The `which` command shows the full path of an executable program by searching the directories listed in the user's `PATH` environment variable. This is useful for finding where a command is installed.

- `which realpath` → `/usr/bin/realpath`
- `which cat` → `/bin/cat`
- `which ls` → `/bin/ls`
- `which file` → `/usr/bin/file`
- You can even find the location of the `which` command itself:
  `which which` → `/usr/bin/which`

---

11 Touch, rm, names with spaces

11.1 Creating Files (touch)

We have seen how to create files using a text editor like `nano`. A simpler way to create an empty file without opening an editor is using the `touch` command.

- **Command:** `touch filename`
- **Example:** `touch file` creates an empty file named "file".

11.2 Deleting Files (rm)

To delete (remove) files, use the `rm` command.

- **Command:** `rm filename`
- **Example:** `rm f` deletes the file named "f".
- **Warning:** Deleted files are removed permanently and do not go to a trash bin.

11.3 Filenames with Spaces

Spaces in filenames can cause issues because the shell interprets spaces as separators between arguments.

11.3.1 The Problem

If you try to create a file with spaces like `touch my space file`, the shell interprets this as a request to create three separate files: "my", "space", and "file".

11.3.2 The Solutions

To handle spaces correctly, you have two main options:

1. **Backslash Escape:** Use a backslash (`\`) before the space to tell the shell it is part of the name.

   ```
   $ touch my\ file
   ```

2. **Quotes:** Enclose the entire filename in single (`' '`) or double (`" "`) quotes.

   ```
   $ touch 'my file'
   ```

To remove these files, you must use the same methods (e.g., `rm 'my file'` or `rm my\ file`).

11.3.3 Best Practice

It is generally recommended to avoid spaces in filenames to prevent these issues. Use underscores (`_`) or hyphens (`-`) instead (e.g., `my_file`).

11.4 Getting Unstuck (Quotes and Brackets)

If you open a quote (single or double), parenthesis, or bracket but forget to close it before pressing Enter, the shell will wait for you to finish the input. The prompt changes (often to `>`), and standard commands like `ls` or `Ctrl+C` might not work immediately.

How to exit:

1. **Close the pair:** Type the missing closing character (quote, bracket, etc.) and press Enter. The command will likely fail (e.g., "command not found"), but you will return to the main prompt.
2. **Control+D:** Pressing `Ctrl+D` sends an "End of File" (EOF) signal. You might need to press it twice or press Enter first to ensure you are on a new line.

---

12 cat less

This section covers how to read and create files using `cat` and output redirection, and how to view long outputs using the `more` and `less` commands.

12.1 Reading and Writing Files (cat, >)

- **Reading Files:** The `cat` command (short for "concatenate") is used to print the contents of a file to the terminal.

```
$ cat file.txt
here is a text file
with a second line
```

- **Output Redirection (>):** The greater-than sign (>) redirects the output of a command into a file instead of printing it to the screen. This will overwrite the file if it already exists.

```
$ echo "hello world" > file2.txt
```

This command creates `file2.txt` containing the text "hello world". This can also be used to copy file contents:

```
$ cat file2.txt > file3.txt
```

- **Creating Files with cat:** You can use `cat` to create a file directly from your keyboard input.

```
$ cat > anotherfile.txt
this is another file
with a new second line
Ctrl+D
```

After typing the command, the terminal waits for your input. Press **Ctrl+D** on a new line to signal the end of the file and return to the prompt.

12.2 Viewing Long Output (Pipes, more, less)

When the output of a command is too long to fit on the screen (e.g., `cat /etc/services`), it scrolls by too quickly. To manage this, we can "pipe" the output to a pager program. The pipe operator (`|`) sends the output of one command to the input of another.

12.2.1 The `more` Command

`more` is a basic pager that displays text one screen at a time.

- **Usage:** `cat /etc/services | more` or `more filename.txt`
- **Navigation:**
  - **Spacebar**: Go to the next page.
  - **q** or **Ctrl+C**: Quit.
- It is a simple tool, and `less` is often preferred.

12.2.2 The `less` Command

`less` is a more advanced and powerful pager ("less is more").

- **Usage:** `ls /usr/bin | less` or `less filename.txt`
- **Navigation:**
  - **Arrows (Up/Down)**: Scroll line by line.
  - **j** / **k**: Move down / up one line.
  - **Spacebar**: Scroll forward one page.
  - **g** (lowercase): Go to the beginning of the text.
  - **G** (uppercase): Go to the end of the text.
  - **q**: Quit.

12.2.3 Searching in `less`

`less` allows you to search through the text.

- **/pattern**: Search forward for the given pattern.
- **?pattern**: Search backward for the given pattern.
- **n**: Jump to the next match.
- **N** (uppercase): Jump to the previous match.

```
$ ls /usr/bin | less
(Inside less, type /joe and press Enter to search for "joe")
```

12.2.4 Options for `less`

`less` can be customized with command-line options.

- **-N**: Display line numbers.
- **-M**: Display a more detailed status line, including the percentage through the file.

```
$ ls /usr/bin | less -NM
```

This will show the list of files with line numbers and a status bar at the bottom.

---

13 mkdir, rm -r

This section explains how to create and delete directories (folders).

13.1 Creating Directories (mkdir)

The `mkdir` command is used to "make directories".

- **Single Directory:** To create one directory.

  ```
  $ mkdir folder
  ```

- **Multiple Directories:** To create several directories at the same level.

  ```
  $ mkdir d1 d2 d3
  ```

- **Nested Directories (-p):** To create a directory and any necessary parent directories that do not already exist, use the `-p` (parents) option.

  ```
  $ mkdir -p D1/D2/D3
  ```

  This command creates directory `D3` inside `D2`, which is inside `D1`.

13.2 Deleting Directories (rm -r)

The standard `rm` command does not delete directories. To do so, you must use the recursive option.

- **Recursive Deletion (-r):** The `-r` (or `-recursive`) option tells `rm` to delete a directory and all of its contents, including subdirectories.

  ```
  $ rm -r folder
  ```

  Warning: This command is extremely powerful and deletes files and directories permanently without confirmation. Use it with great care.

- **Interactive Deletion (-i):** To add a layer of safety, combine the `-r` option with `-i` (interactive). This will prompt you to confirm every deletion.

```
$ rm -ri folder
rm:  descend into directory 'folder'?  y
rm:  remove regular empty file 'folder/f1'?  y
rm:  remove directory 'folder'?  y
```

- **Automating Confirmation (yes):** The `yes` command repeatedly outputs 'y'. You can pipe its output to an interactive command to automatically answer "yes" to all prompts.

  ```
  $ yes | rm -ri folder
  ```

  This command will automatically confirm and delete the `folder` directory and all its contents...

---

14 Unblock yourself in the command line

This section explains how to get back to the command prompt, even if you are stuck inside a program.

14.1 Basic Exit Strategies

There are several standard ways to exit a program or an input mode:

- **Ctrl+C:** This is the most common way to interrupt and terminate a running program (e.g., the `rain` program). It works in most, but not all, cases.
- **Program-Specific Commands:** Some programs have their own exit commands. For example, in `vim`, you must press **Esc** and then type `:q!` and press **Enter** to force quit. **Ctrl+C** will not work.
- **Closing Input:** If you have an open single quote (`'`), double quote (`"`), or bracket, the shell will wait for you to close it. You must type the matching character to finish the input and return to the prompt.
- **Ctrl+D (End of File):** This signal is used to indicate the end of input.
  - It is used to exit from commands that read from standard input, such as `cat > file.txt`.
  - Sometimes it needs to be pressed on a new line, and occasionally it needs to be pressed twice.

14.2 The Worst Case: Killing a Stuck Process

If a program is completely unresponsive and the basic methods fail, you can forcefully terminate it from a second terminal.

1. **Open a Second Terminal:** In environments like WebLinux, there is often a button (e.g., an arrow) to open a new terminal session.
2. **Identify the Process with `htop`:** In the second terminal, run `htop`. This will show a list of all running processes for your user.
3. **Find the Process:** Locate the stuck program in the list (e.g., `rain`). Note its **PID** (Process Identifier), which is a unique number assigned to it.
4. **Send the Kill Signal:**
   - Use the arrow keys to select the stuck process within `htop`.
   - Press the **F9** key to bring up the "send signal" menu.
   - Select signal **9** (**SIGKILL**) from the list on the left. This is a non-ignorable signal that forcefully terminates the process.
   - Press **Enter** to send the signal.
5. **Verify Termination:** The process should disappear from the `htop` list. If you switch back to the first terminal, you will see a "Killed" message, and the prompt should return.

14.3 Recovering a Stuck Terminal (WebLinux)

Sometimes, even after killing a process, the original terminal can become unresponsive.

- **The Bug:** In WebLinux, after killing a process from a second terminal, the prompt in the first terminal may return but not accept any input.
- **Standard Linux Solution:** On a full Linux system, you would typically open another terminal and kill the stuck shell process itself. Reboots are rarely necessary.
- **WebLinux Solution (The Last Resort):** If the terminal is completely stuck, the simplest way to "reboot" WebLinux is to **reload the browser page**. This will start a fresh Linux environment, but your files will be preserved.

---

15 mv: rename and move

This section explains how to move and rename files and directories using the `mv` command.

15.1 Moving Files and Directories

The `mv` command is used to move a file or directory from one location to another.

- **Move a file into a directory:**

```
$ touch f1
$ mkdir d1
$ mv f1 d1
```

  This moves the file `f1` into the directory `d1`.

- **Move a directory into another directory:**

```
$ mkdir bigdir
$ mv d1 bigdir
```

  This moves the directory `d1` (and its contents) into `bigdir`.

15.2 Using Relative Paths

You can use relative paths to move items around the filesystem.

- **Moving a file up the directory tree:** If you are inside `/home/user/bigdir/d1`, you can move `f1` back to the home directory (`~`).

```
$ pwd
/home/user/bigdir/d1
$ mv f1 ../..
```

  The path `../..` refers to the parent of the parent directory, which is `/home/user`.

- **Error case:** You cannot move the directory you are currently in.

```
$ pwd
/home/user/bigdir/d1
$ mv .  ../..
mv:  cannot move '.'  to '../../d1':  Device or resource busy
```

15.3 Renaming with mv

There is no dedicated "rename" command in Linux. The `mv` command is used for renaming. If the destination is a new name in the same directory, `mv` performs a rename.

```
$ mv bigdir smalldir
```

This command renames the directory `bigdir` to `smalldir`.

15.4 Moving and Renaming Simultaneously

You can combine moving and renaming into a single command.

- **Move and rename a file:** Move `f1` from `otherdir` to the home directory and rename it to `f2`.

```
$ mv otherdir/f1 ~/f2
```

- **Move and rename a directory:** Move `d1` from `smalldir` into `otherdir` and rename it to `d2`.

```
$ mv smalldir/d1 otherdir/d2
```

15.5 The Dangers of Overwriting

The `mv` command can be dangerous because it will overwrite existing files without warning.

- Consider two files, `f5` and `f6`, with different content.

```
$ echo "this is f5" > f5
$ echo "this is f6" > f6
```

- If you "rename" `f6` to `f5`, the original `f5` is permanently lost.

```
$ mv f6 f5
$ cat f5
this is f6
```

Warning: The original content of `f5` has been overwritten and cannot be recovered.

15.6 Safe Moving and Renaming

To prevent accidental overwriting, you can use options with the `mv` command. You can see these options with `mv -help`.

- **Interactive Mode (`-i`):** This option prompts you for confirmation before overwriting a file.

```
$ mv -i f6 f5
mv:  overwrite 'f5'?  n
```

You can type `n` (for no) to cancel the operation and prevent data loss.

- **No-Clobber Mode (`-n`):** This option prevents `mv` from overwriting any existing file. The move will simply not happen if the destination file exists.

Best Practice: Be very careful with `mv`. Using the `-i` option is a good habit to avoid losing data.
