Chapter 1: Bash Basics — Notes

By Hansika Bhambhaney

Objectives:

- Understand the function of a shell
- Trace the evolution of UNIX shells and bash
- Learn interactive bash usage and file operations
- Master input/output redirection and job control

1.1 WHAT IS A SHELL?

A shell is a command-line interface that connects the user with the UNIX or Linux operating system. It allows users to execute commands, manage files, control processes, and automate system tasks using text-based input.

🔷 ***Key Functions:***

◆ Interprets user commands
◆ Manages file systems and processes
◆ Automates workflows via scripting
◆ Enables redirection, piping, and job control

🔷 ***How It Works:***

➢ Reads user input
➢ Parses and tokenizes the command line
➢ Performs syntax checks
➢ Uses system calls like `fork()` and `exec()` to create and execute processes

🔷 ***Shell vs GUI:***

✦ Text-only interface (resource-efficient)
✦ Faster for repetitive or batch operations
✦ Lightweight compared to graphical tools

🔷 ***About Bash (Bourne Again SHell):***

1. Built on top of the original `sh`
2. Supports command history, job control, and advanced scripting
3. Handles input/output redirection with symbols like `>`, `<`, `|`
4. Pre-installed on most Linux systems

Why Bash is Important in Bioinformatics ?

- **1. Automates Pipelines Efficiently**

Bash scripts allow researchers to automate multi-step bioinformatics workflows (e.g., quality control → alignment → variant calling), saving time and reducing manual errors.

- **2. Seamless Integration with Bioinformatics Tools**

Most command-line tools (like `bwa`, `samtools`, `bcftools`, `fastqc`, `bedtools`) are designed to be used within bash environments, allowing easy chaining and scripting.

- **3. Manages Large Datasets**

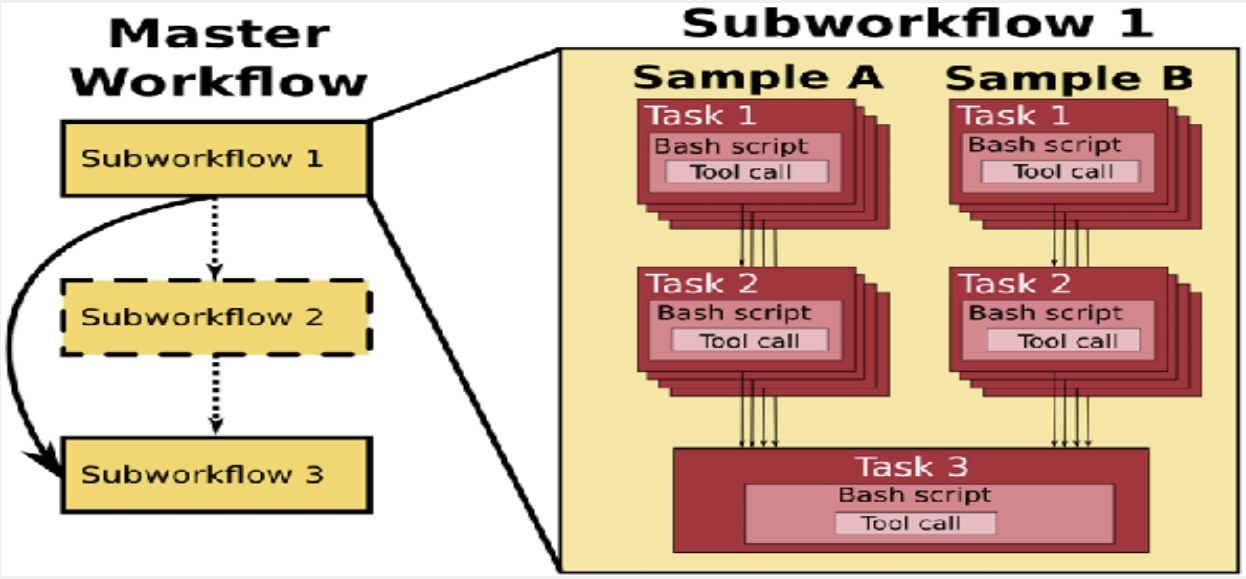
Bioinformatics deals with massive files (FASTQ, BAM, VCF, etc.). Bash handles these efficiently using loops and batch processing.

- **4. Optimized for UNIX/Linux Environments**

Most servers, clusters, and HPC systems used in bioinformatics run on UNIX/Linux, where bash is the default and most compatible shell.

- **5. Powerful Text Parsing Tools**

Built-in tools like `awk`, `grep`, `cut`, and `sed` allow fast parsing, filtering, and formatting of genomic and annotation data.



1.2 Evolution of UNIX Shells

- **Early UNIX Shells:**
  - **Thompson Shell (`sh`)**
    1. First shell developed in early UNIX (1971)
    2. Very basic functionality — no scripting or flow control
  - **Bourne Shell (`sh`) – 1979**
    1. Added control structures (`if`, `for`, `while`) and scripting capabilities
    2. Became the standard scripting shell for UNIX
- **Feature-Enhanced Shell:**
  - **C Shell (`csh`)**
    - Developed at UC Berkeley
    - C-like syntax, command history (`!!`, `!n`), aliases, job control
    - Preferred for interactive use but poor for scripting
- **Modern Standard:Bourne Again Shell (`bash`)**
  - Command history and auto-completion
  - Arithmetic expressions and arrays
  - Brace expansion (`{a,b,c}`), command substitution

1.3 Commands, Arguments, and Options

❖ The command initiates an action, while options and arguments refine and control its behavior.

```
command [OPTIONS] arguments
```

1.Command → The program to execute (e.g., `ls`, `lp`)

2.Options → Flags that alter command behavior (usually start with `-`)

3.Arguments → The targets (e.g., files, folders, usernames)

1.3.1 WHAT ARE OPTIONS?

In shell commands modify how a command behaves. They usually start with a - (dash) and sometimes require additional arguments.

1.***ls -l***

`ls` → The command used to list files and directories.

`-l` → An option that displays the output in a detailed "long" listing format, including file permissions, ownership, size, and modification date.

2.***lp -d lp1 -h myfile***

lp → The command responsible for sending a file to the printer.

-d lp1 → An option specifying the destination printer (lp1) .

-h → An option that suppresses the banner.

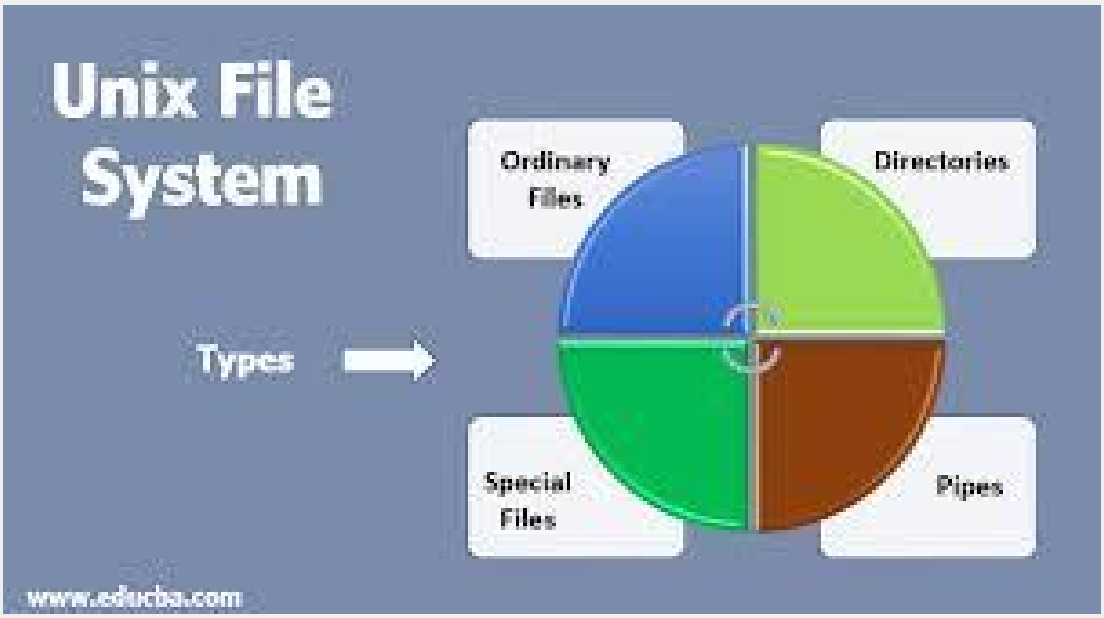
myfile → The file (argument) to be printed.

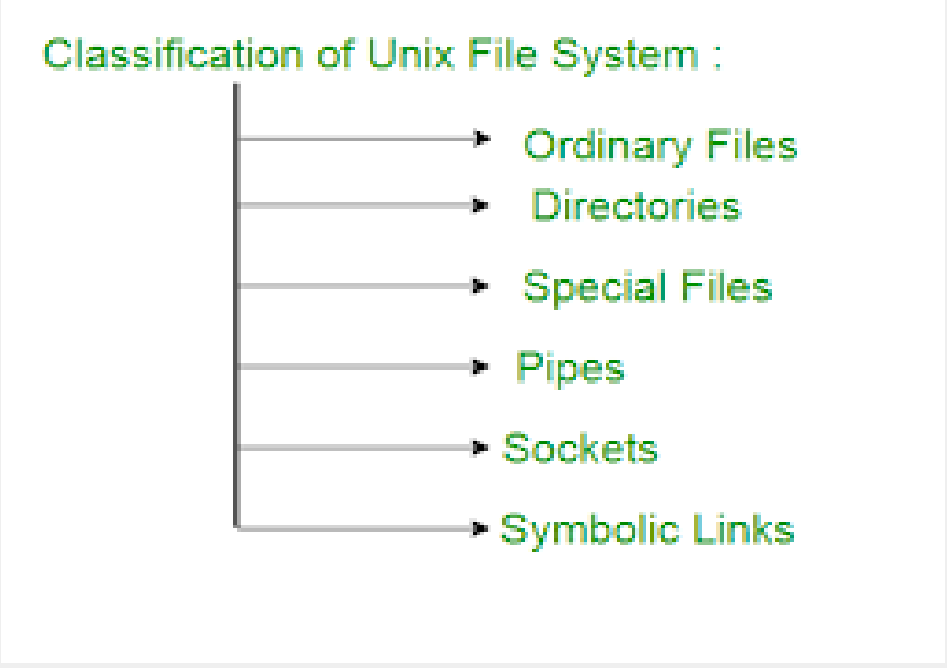
3.grep -i -n "hello world" file.txt

- grep → Command (searches for text)
- -i → Makes search case-insensitive

1.4 WHAT ARE FILES AND ITS TYPES?

- In UNIX/Linux, everything is treated as a file — including programs, text, directories, and even hardware devices. Understanding file types is fundamental when working with the shell.





1.4.1 Types of Files in UNIX/Linux:

- **1. Regular Files**
    - Contain readable data or binary content.
    - Examples: text files (`notes.txt`), data files (`genes.fasta`), logs.
    - Not executable unless explicitly made so.
    - Can be created or viewed using commands like `touch`, `cat`, `nano`, etc.

```
cat notes.txt          # Display contents of a text file
touch report.txt       # Create an empty text file
nano genes.fasta       # Open a FASTA file in terminal editor
```

🔹 **2. Executable Files**

- These are files that can be run as programs.
- Can be compiled binaries or bash scripts (`.sh`).
- Must have execution permission using `chmod`.

```
chmod +x script.sh     # Make script executable
./script.sh            # Run executable shell script
file /bin/ls           # Show that ls is a binary executable
```

3. Directories

- Special files that contain other files or subdirectories.
- Form the hierarchical UNIX file structure.
- Use navigation commands to move or explore them.

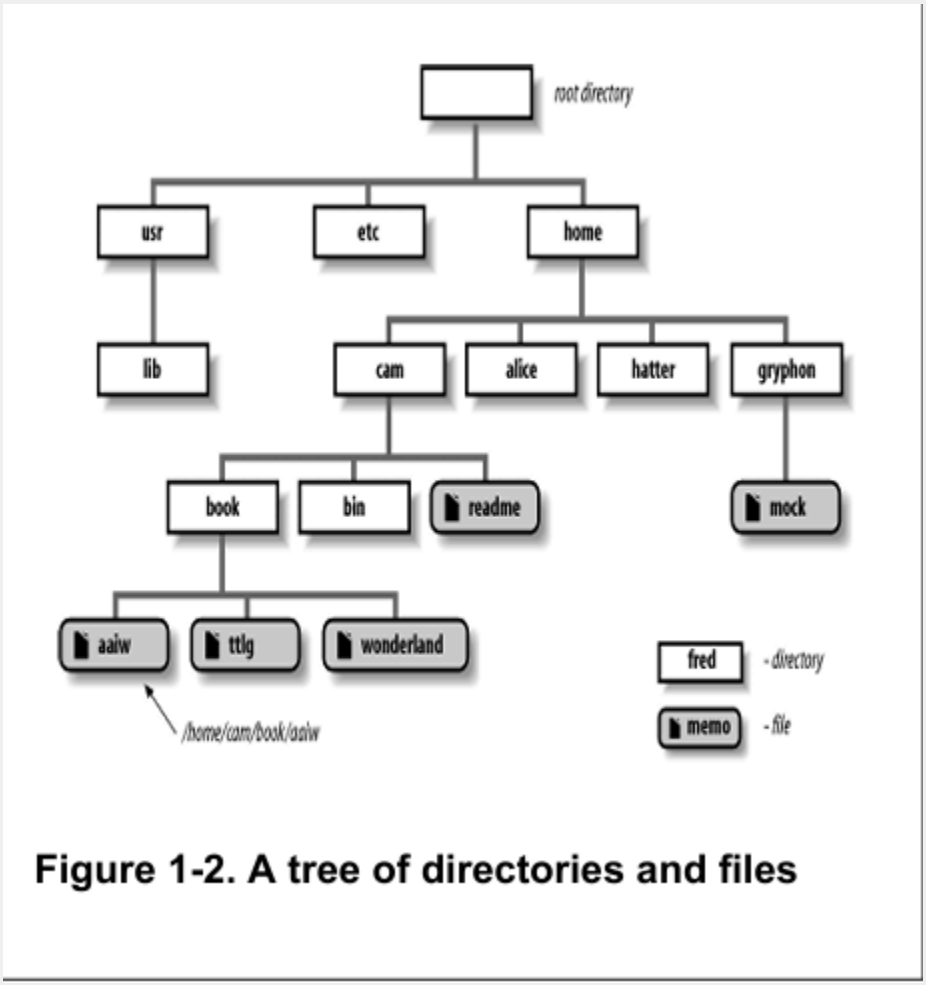


Figure 1-2. A tree of directories and files

```
mkdir projects          # Create a new directory
cd projects             # Move into directory
ls -l                   # List contents of current directory
pwd                     # Show full path of working directory
```

1.5 Understanding the Working Directory in UNIX/Linux

The working directory, also called the current directory, refers to the location in the filesystem where the user is presently operating while using the terminal. Rather than specifying complete file paths every time, users can take advantage of relative pathnames, which simplifies navigation and file operations.

- **Absolute Pathname:**
  Refers to the full path of a file or directory starting from the root (`/`).
  Example:

```
/home/cam/book/memo
```

- **Relative Pathname:**

  Refers to the location of a file relative to the current working directory.

  Example:
  If the working directory is `/home/cam`, then the file can be accessed as:

```
book/memo
```

1. **Check current directory** → pwd
2. **Move between folders** → cd folder_name
3. **List contents** → ls -l
4. **Create/Delete folders** → mkdir, rmdir, rm -r
5. **Copy/Move files** → cp, mv
6. **Find files** → find, grep

1.6 Filename Substitution (Globbing) and Brace Expansion

In bash, filename substitution is the process where the shell expands wildcard patterns into matching filenames before executing a command. This process is known as globbing.

🔹 **1. Wildcard Characters (Globbing)**

Wildcards are used to match patterns in filenames and directory names.

✦ `*` (Asterisk)

- Matches zero or more characters.

```bash
ls *.txt
```

Lists all files ending in `.txt` (e.g., `data.txt`, `notes.txt`)

✦ `?` (Question Mark)

- Matches exactly one character.

```bash
ls file?.sh
```

Matches `file1.sh`, `fileA.sh`, but **not** `file10.sh`

✦ `[abc]` (Character Class)

- Matches one character from the set inside brackets.

```bash
ls report[123].csv
```

1.6.1 Brace Expansion

Brace expansion is a built-in Bash feature that generates strings based on defined patterns. Unlike wildcards (globbing), which match existing files or directories, brace expansion produces literal text combinations, regardless of whether the resulting strings correspond to actual files.

- **1. Basic Brace Expansion**
  SYNTAX

```
preamble{item1,item2,...}postscript
```

1.**Preamble**: Prefix added before each expanded item

2.Options: Comma-separated values inside braces {}

3.Postscript: Suffix added after each expanded item.

EXAMPLE: The shell combines each option with the prefix b and suffix s.

```bash
echo b{ed,olt,ar}s
```

OUTPUT:

```nginx
beds bolts bars
```

- **2. Nested Brace Expansion**

Bash supports nesting braces to create more complex combinations.

EXAMPLE: The inner set `{d,n,k}` is expanded first, then merged with `ar` and `ed`.

```bash
echo b{ar{d,n,k},ed}s
```

OUTPUT:

```nginx
bards barns barks beds
```

- **3. Combining Brace Expansion with Wildcards**

Brace expansion can be used with wildcards to reduce repetitive patterns in commands.

Instead Of:

```bash
ls *.c *.h *.o
```

USE :This expands to match all files ending with `.c`, `.h`, or `.o`.

```bash
ls *.{c,h,o}
```

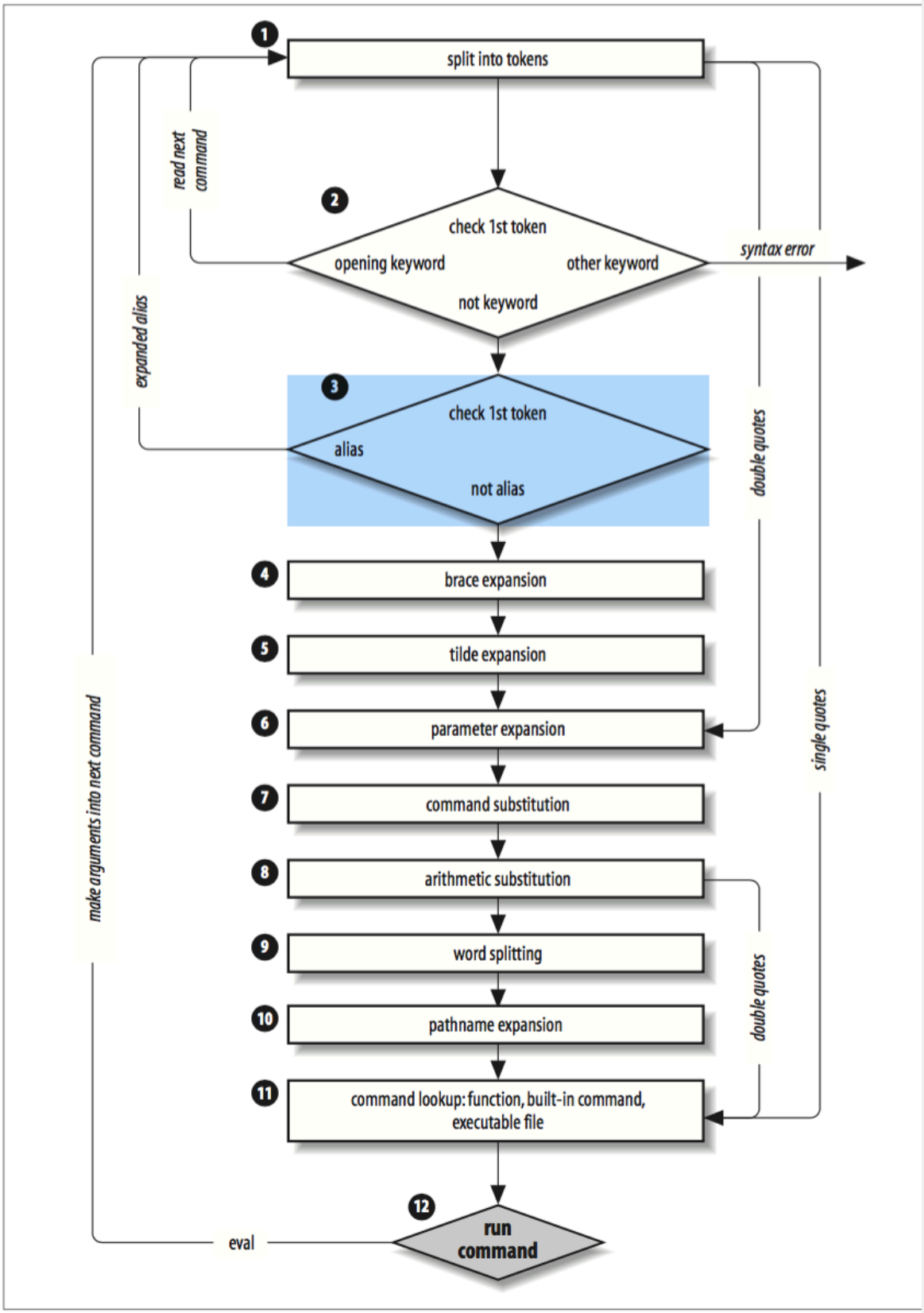


Figure 7-1. Steps in command-line processing

1.7Input and Output Redirection in Bash

Redirection is a key feature of the bash shell that allows users to control where input comes from and where output goes. It's used extensively in automation, scripting, and command chaining.

1. Understanding Standard Streams

Bash (like most UNIX shells) deals with three default data streams:

- **Standard Input (`stdin`)** — file descriptor `0`
  → Default source: keyboard

- **Standard Output (`stdout`)** — file descriptor `1`
  → Default destination: terminal screen

- **Standard Error (`stderr`)** — file descriptor `2`
  → Default destination: terminal screen (separate from stdout)

These streams can be redirected to or from files and other commands.

2. Redirection Operators in Bash

✦ `>` – Redirect Standard Output (Overwrite)

- Redirects the output of a command to a file. If the file exists, it is overwritten.

```bash
bash

echo "Hello" > output.txt
```

Creates (or overwrites) `output.txt` with the text "Hello".

✦ `>>` – Redirect Standard Output (Append)

- Appends the output to a file instead of overwriting it.

```bash
echo "Another line" >> output.txt
```

Adds the text to the end of `output.txt`

✦ `2>` – Redirect Standard Error

- Redirects only the error output to a file.

```bash
ls wrongfile 2> error.log
```

Saves the error message (e.g., "No such file") to `error.log`.

✦ `&>` – Redirect Both Standard Output and Error

- Combines `stdout` and `stderr` into a single file.

```bash
./run.sh &> log.txt
```

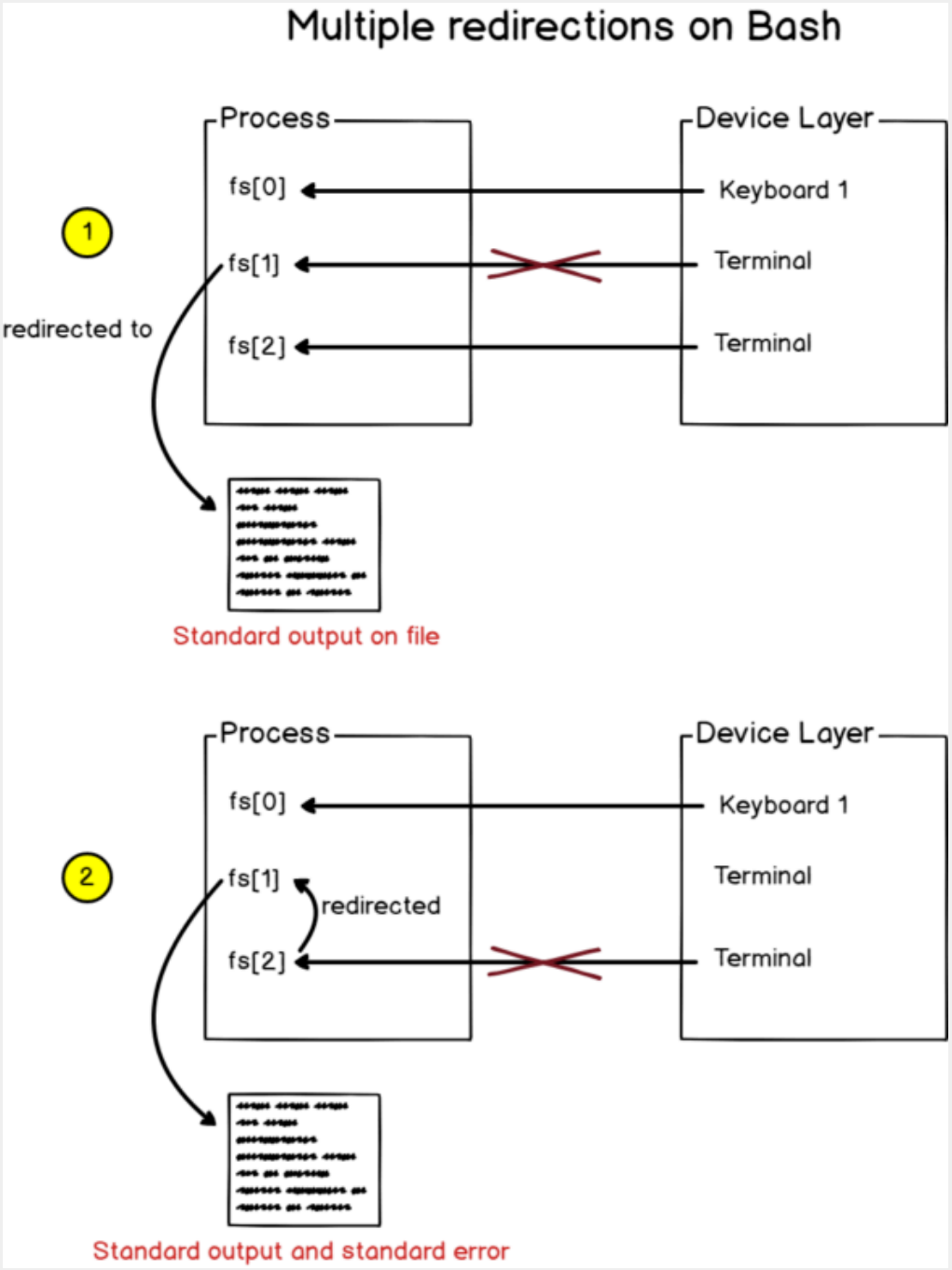
Stores all output and error messages into `log.txt`.

3. Silent Execution / Suppressing Output

- Discard output by redirecting to `/dev/null`:

```bash
command > /dev/null          # Discard stdout
command 2> /dev/null         # Discard stderr
command &> /dev/null         # Discard both
```

Useful for running scripts silently or hiding unwanted messages.



1.8 Pipelines in Bash

Pipelines in Bash allow you to connect multiple commands together so that the output of one command becomes the input of another — enabling powerful command chains and efficient data processing directly in the terminal.

1.8.1 What is a Pipeline?

- A pipeline is created using the `|` (pipe) symbol.
- It passes the standard output (stdout) of one command directly into the standard input (stdin) of the next command.
- This enables modular and readable command structures — each command does one job, and together they perform complex operations.

1.8.2. Simple Example

```bash
cat file.txt | grep "error"
```

`cat` displays the content of `file.txt`
`grep` filters lines that contain the word “error”

1.8.3 . Multi-Command Example (Chain)

Step-by-step:

- `cat access.log` → reads the log file
- `grep "404"` → filters HTTP 404 error entries
- `sort` → organizes them alphabetically
- `uniq -c` → counts unique entries
- `sort -nr` → sorts results numerically in reverse order

```bash
cat access.log | grep "404" | sort | uniq -c | sort -nr
```

1.8.4 Why Use Pipelines?

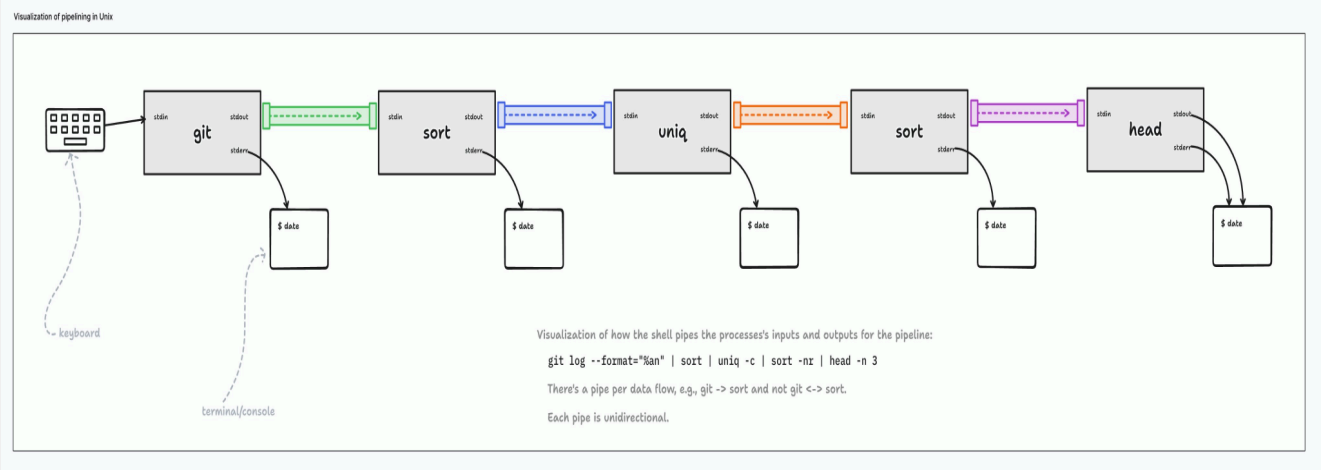
- Efficient: Avoids the use of temporary files.
- Readable: Breaks complex tasks into smaller, manageable pieces.
- Modular: Each command focuses on a single task (Unix philosophy).
- Fast: Uses in-memory streams instead of writing to disk between steps.

1.8.5. Pipes and Filters

Filters are commands that process input and produce modified output — commonly used in pipelines.

✔ Common filter commands:

- `grep` → searches for patterns
- `awk` → extracts/prints structured text
- `cut` → extracts fields/columns
- `sort` → sorts input lines
- `uniq` → removes duplicate lines
- `head`, `tail` → limit output
- **`tr`**, **`sed`** → translate or modify characters/text



1.9 Background Jobs IN BASH

1. What Are Background Jobs?

- In Bash, a background job is any command or process that runs independently of the terminal after it has been launched.
- When a job is run in the background, the shell immediately returns the prompt, allowing the user to continue working or run other commands without waiting for the first command to finish.
- This is especially useful for long-running tasks such as data processing, file transfers, or system updates.

2. Why Use Background Jobs?

- To multitask in a terminal session without opening multiple terminal windows or tabs.
- To avoid blocking the command line when running time-consuming processes.
- To allow parallel execution of multiple jobs in shell scripts or during command-line sessions.

3. Job Management in the Shell

- Every background job started in a shell is assigned a job ID and tracked by the shell's internal job control system.
- Users can view a list of all currently running background jobs in the shell session.

- Background jobs can be managed by bringing them to the foreground or resuming them after being suspended.

4. Foreground vs. Background Execution

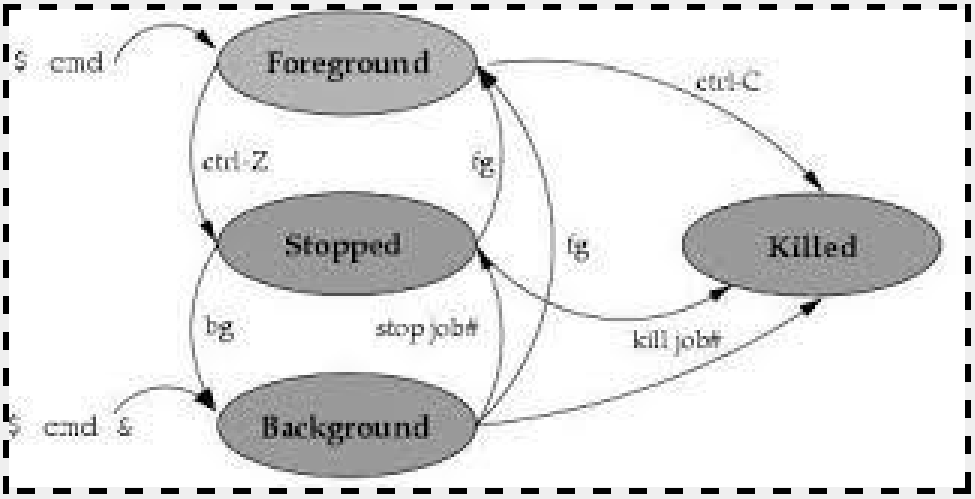
- A foreground job occupies the terminal and must complete before another command can be executed.
- A background job runs silently in the background while the terminal remains available for other tasks.
- Users can switch jobs between background and foreground as needed using job control features.

5. Monitoring and Managing Jobs

- Background jobs are not displayed by default. You must actively check for them using job control commands.
- Jobs can be stopped, paused, resumed, or terminated if needed.
- Proper job management is crucial when running multiple concurrent tasks, especially on shared systems or servers.

6. Controlling Job Priority with `nice`

- The `nice` system feature allows users to set the priority level of a job at the time it is launched.
- Lower priority jobs are given fewer CPU resources, allowing high-priority tasks to run more efficiently.



1.9 Special Characters and Quoting in Bash

Bash provides quoting mechanisms and special characters to handle spaces, special symbols, and control how variables or expressions are interpreted. Quoting ensures that the shell treats text as literal or partially literal, depending on the quote type.

1. Single Quotes (`'...'`)

Purpose:

- Single quotes are used to preserve the exact text within them.
- Nothing inside single quotes is interpreted by the shell — variables, commands, and escape sequences are treated as plain text.

Example: `echo '$HOME'`

The output will be: `$HOME` (not the value of the variable).

The shell treats everything literally, including the dollar sign.

Used when :

- When we want to prevent variable expansion or command execution.
- When handling content with special symbols that should remain untouched.

2. Double Quotes (`"..."`)

Purpose:

- Double quotes allow interpretation of variables and some special characters.
- Within double quotes, Bash will expand variables and command substitutions, but still treat the rest of the content as a single string.

Example: `echo "My home is $HOME"`

The output might be: `My home is /home/username`

The `$HOME` variable is expanded inside double quotes.

Used when:

- When we need to preserve whitespace or formatting but also want variable or command evaluation.
- Useful in script output and dynamic messages.

🔹 **3. Backslash (`\`) – Escape Character**

Purpose:

- The backslash is used to escape or suppress the special meaning of a single character.
- It tells Bash to treat the following character literally.

Examples:

- Typing `echo \$HOME` will display `$HOME` without expanding the variable.
- Typing `echo "She said: \"hello\""` will print: `She said: "hello"`
- A backslash at the end of a line (`\`) allows you to continue a long command on the next line.

Used when:

- When we need to insert quote marks inside a quoted string.
- When writing multi-line commands.
- When we need to print or pass special characters without triggering their function.

Summary – Chapter 1: Bash Basics

Chapter 1 provides a comprehensive introduction to the bash shell — a powerful command-line interface used in UNIX/Linux systems. It covers the evolution of various UNIX shells, the structure of commands, and essential file system concepts. Key features such as globbing, brace expansion, input/output redirection, and pipelines are explained in detail, along with job control and quoting mechanisms.

The chapter emphasizes how bash supports automation, scripting, and efficient resource handling, making it an indispensable tool for developers, system administrators, and bioinformaticians. By mastering bash, users gain control over system operations and the ability to automate complex workflows with minimal overhead.

References:

Newham, C. (2005). *Learning the bash Shell* (3rd ed.). O'Reilly Media. ISBN: 0-596-00965-8.