Shell (and a bit of vim)

Shell Architecture, Streams, and Text Processing

Intro to Software Systems

January 21, 2026

Layers of Abstraction

The Kernel

The core of the OS. Manages CPU, memory, and hardware. It speaks binary/syscalls.

The Shell

An interface to the OS. It interprets your text commands and translates them into system calls for the Kernel .

Terminal vs. Shell

The Terminal

- A program that accepts text input and displays text output.
- It sends keystrokes to the shell and renders the characters the shell sends back.

The Shell

- Interface between user and the kernel.
- The program running *inside* the terminal
- Shells can be graphical (like Windows Explorer) or text-based (like CMD, PowerShell, or Bash).
- It parses syntax, expands variables, and executes logic.

Essential Commands

Navigation

- `pwd` : Print working dir.
- `cd <dir>` : Change dir.
- `ls` : List contents.
  `-l` (long), `-a` (hidden)

View Files

- `cat` : Print entire file.
- `head -n` : First n lines.
- `tail -n` : Last n lines.

Create

- `touch <file>` : Create empty file (or update timestamp).
- `mkdir <dir>` : Create dir.
  `-p` (create parents)

Delete

- `rm <file>` : Remove file.
  `-r` (recursive), `-f` (force)
- `rmdir <dir>` : Remove empty dir.

Copy & Move

- `cp <src> <dst>` : Copy.
  `-r` (recursive for dirs)
- `mv <src> <dst>` : Move / Rename.

Shortcuts

- `alias ll='ls -la'`
  Create shorthand.
- `func() { cmds; }`
  Reusable logic.

Standard Streams & File Descriptors

In Unix, "Everything is a File." When a command runs, it gets three open file descriptors (FDs):

Stdin (0)	**Stdout (1)**	**Stderr (2)**
Standard Input	Standard Output	Standard Error
(Keyboard)	(Screen)	(Screen)

Anatomy of a Command

Shell Parsing Precedence (inner → outer):

```
( grep "err" log.txt | head ) && echo "Found" || echo "None" ; date
```

Execution Order:

1. `|` Pipeline (left to right)
2. `( )` Subshell completes
3. `&& ||` Logical chain
4. `;` Sequential (always runs)

Operators:

- `|` : Pipe stdout to next stdin.
- `( )` : Group in subshell.
- `&&` : Run if prev succeeded.
- `||` : Run if prev failed.
- `;` : Run next regardless.

Redirection vs. Piping

The Critical Distinction

Redirection (>) connects a process to a **FILE**.
Piping (|) connects a process to another **PROCESS**.

Redirection (to File)

```
# Output goes to 'list.txt'
ls -l > list.txt
```

Piping (to Command)

```
# Output goes to 'grep' input
ls -l | grep ".txt"
```

Advanced Redirection

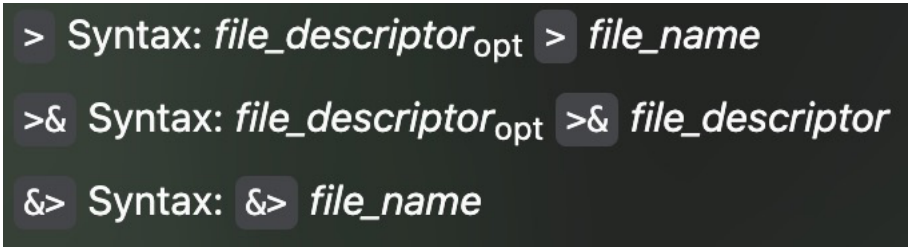
By default, > only redirects Stdout (1). Errors (2) still print to the screen.

Listing 1: Redirecting Errors Only

```
# '2>' grabs FD 2
gcc program.c 2> errors.log
```

Listing 2: Redirecting Everything

```
# Point FD 2 to where FD 1 is
./script.sh > all.log 2>&1
```



Append Mode: Use >> to add to a file instead of overwriting it.

Here Documents (Heredoc)

Problem: How to pass multiple lines to a command without a file?

Syntax: `command <<DELIMITER`

```
cat <<EOF
babeeee
when coming over
EOF
```

Output:

```
babeeee
when coming over
```

Use Cases:

- Multi-line input to commands.
- Embedding scripts/configs.
- Creating files inline.

Variants:

- `<<-`: Ignore leading tabs.
- `<<<`: Here string (single line).

Pro Tip

`cat <<EOF > file.txt` creates a file with the heredoc content.

Inputs

Scripts are not just static; they can pause and ask for user input.

```
echo "Enter your name:"
read username
echo "Hello $username"
```

Command: `read`

- Pauses execution.
- Waits for user to type and press Enter.
- Stores text in the specified variable.

Conditionals

```
if [ -f "config.txt" ]; then
    echo "Config found."
elif [ $user == "root" ]; then
    echo "Welcome Admin."
else
    echo "Access Denied."
fi
```

Common Tests:

- `-f file`: File exists.
- `-z str`: String is empty.
- `-eq`, `-gt`: Math comparison.

The Case Statement

Useful when matching a variable against multiple patterns (cleaner than many `elif`s).

```
case "$1" in
    start)
        echo "Starting server..."
        ;;
    stop)
        echo "Stopping server..."
        ;;
    *)
        echo "Usage: $0 {start|stop}"
        ;;
esac
```

Loops

The For Loop (Iterate lists)

```
# Using Glob Patterns
for file in *.log
do
    cat $file >> combined.log
done
```

The While Loop (Condition based)

```
count=1
while [ $count -le 5 ]
do
    echo "Attempt $count"
    count=$[$count + 1]
done
```

Regex

Globbing (Shell) $\neq$ Regex (Text Processors)

- **Globbing:** Used by shell for filenames (`*.txt`).
- **Regex:** Used by `grep`, `awk`, `vim` for content.

Common Regex Tokens:

- `^` : Start of line.
- `$` : End of line.
- `.` : Any single character.
- `*` : Zero or more.

```
# Matches "Apple", "Appple"
grep "Ap*le" file.txt
```

Grep vs. Awk

grep

"I want to find LINES."

- Best for: Filtering rows based on patterns.
- Input: Text stream.
- Output: Lines that match.

awk

"I want to manipulate COLUMNS."

- Best for: Structured data (CSV, logs).
- It is a full programming language.

Searching Like a Pro: grep

The Syntax: `grep [flags] "pattern" [path/files]`

Key Flags

- `-i` : Ignore case (Error matches error).
- `-n` : Show line numbers.
- `-r` : Recursive (Search folders).

Real Scenarios

```
# 1. Debugging (Case insensitive)
grep -i "error" app.log

# 2. Finding lost code (Recursive)
grep -r "TODO" .

# 3. Pinpointing crashes (Lines)
grep -n "segfault" main.c
```

Optimization Tip: Globbing

Don't search everything (`grep "foo" *`). It looks inside binaries/images!

- **Better:** `grep "main" *.c` (Only searches C files)

Change mode - chmod

The Problem

- By default, Linux creates files with only read/write permissions.
- They are **not** treated as executable programs for security reasons.
- To run a script, you must change its "mode" (chmod).

Method 1: Symbolic (The Easy Way)

- **Who:** `u` (user), `g` (group), `o` (others), `a` (all).
- **Action:** `+` (add), `-` (remove), `=` (set).
- **Perms:** `r` (read), `w` (write), `x` (exec).

Listing 3: Examples

```
# Add eXecutable perm to user
chmod u+x script.sh

# Remove Write from group
chmod g-w data.txt
```

Change mode - chmod

Method 2: Octal (Numeric) Permissions are represented by adding specific values together.

The Values

- **r** $= 4$
- **w** $= 2$
- **x** $= 1$

The Math

- `rw` $\rightarrow 4 + 2 =$ **6**
- `rx` $\rightarrow 4 + 1 =$ **5**
- `rwx` $\rightarrow 4 + 2 + 1 =$ **7**

Common Scenarios

Public Script (755)

User: rwx $(4 + 2 + 1 = 7)$
Group: rx $(4 + 1 = 5)$
Other: rx $(4 + 1 = 5)$

```
chmod 755 app.sh
```

Private File (600)

User: rw $(4 + 2 = 6)$
Group: — (0)
Other: — (0)

```
chmod 600 secret.key
```

Navigation & Pitfalls

1. Paths: Absolute vs. Relative

- **Absolute:** Starts from Root (/).
- **Relative:** Starts from current dir.
- **Symbols:** . (Here), .. (Parent).

```
cd /home/user/proj   # Absolute
cd ../data           # Relative
```

Why won't my script run?

Linux does not look in . by default. You must specify the path: `./script.sh`

2. The "Space" Problem The shell splits arguments on spaces.

Bad (Fails):

```
rm My File.txt
# Deletes "My" AND "File.txt"
```

Good (Works):

```
rm "My File.txt"    # Quotes
rm My\ File.txt     # Escape
```

Golden Rule for Scripts: Always quote your variables! `rm "$file"`

Job Control: Multitasking

The Concept The shell allows you to pause, resume, and run processes in the background, letting you use a single terminal for multiple tasks.

Key Shortcuts

- `Ctrl + C`
  Kill the process (SIGINT).
- `Ctrl + Z`
  Suspend/Pause the process.

Commands

- `bg`: Resume suspended job in the **background**.
- `fg`: Bring background job to the **foreground**.
- `jobs`: List all active jobs.

Workflow: The "Pause" Button

Scenario: You are in a Python shell with active variables, but need to check a filename. Exiting would delete your data!

The Solution: Suspend & Resume

```
# 1. You define variables (Don't exit!)
$ python3
>>> count = 500
>>> f = open("... wait, what is the file called?")

# 2. Hit Ctrl + Z to SUSPEND (Pause)
[1]+  Stopped                 python3

# 3. Terminal is free! Check the file name.
$ ls
input_data.txt   script.py

# 4. Resume Python (Variable 'count' is safe!)
$ fg
>>> f = open("input_data.txt")
```

From Commands to Scripts

Definition: A script is simply a saved sequence of commands.

1. The Structure

```
#!/bin/bash
# ^ The "Shebang" (Interpreter)

echo "Starting task..."
mkdir -p output
```

2. The Execution

Scripts are not executable by default!

```
# Step A: Grant Permission
chmod +x script.sh

# Step B: Run it (requires ./)
./script.sh
```

Arguments

Don't hardcode paths. Let the user decide.

Listing 4: safe_script.sh

```
#!/bin/bash

# 1. Check if argument exists
if [ $# -ne 1 ]; then
    echo "Usage: $0 <directory>"
    exit 1
fi

# 2. Check if it is actually a directory
if [ ! -d "$1" ]; then
    echo "Error: Not a directory"
    exit 1
fi
```

Key Concepts:

- $0: Script name.
- $1: First argument.
- $#: Argument count.
- `exit 1`: Signal failure.

Vim: Modes & Navigation

Modal Editor: Different modes for different tasks.

Modes

- `Normal` (default): Navigate, delete, copy.
- `Insert` (`i`): Type text.
- `Visual` (`v`): Select text.
- `Command` (`:`): Save, quit, search.
- `Esc`: Return to Normal mode.

Navigation (Normal Mode)

- `h j k l`: Left, Down, Up, Right.
- `w` / `b`: Next / Prev word.
- `0` / `$`: Start / End of line.
- `gg` / `G`: Top / Bottom of file.
- `Ctrl+d` / `Ctrl+u`: Page down / up.
- `:n`: Go to line `n`.

Vim: Editing & Commands

Essential Commands

- `i`: Enter insert mode.
- `dd` / `yy` / `p`: Delete / Copy / Paste line.
- `u`: Undo.
- `/text`: Search.
- `:w`: Save.

Combos: Commands compose!
`d2w` = delete 2 words
`3dd` = delete 3 lines

The Most Important Command

How to quit Vim:

- `Esc` → `:q` → `Enter`
- Unsaved changes? `:q!` (discard) or `:wq` (save & quit).

Stuck? Spam `Esc` first.

"How do I exit Vim?" has 2.9M+ views on Stack Overflow.

b: