

History of Linux and the command line

Zubin

January 30, 2026

Contents

1	Compiler vs interpreter	3
1.1	The Interpreter	3
1.2	The Compiler	3
1.3	Summary	3
2	Memory representation:	
	RAM, cells, word, byte, bit, memory address	4
2.1	Types of Memory	4
2.2	Structure of RAM	4
2.3	Memory Addressing	4
2.4	Memory and C Programming	4
3	Manage the memory with the command line:	
	free, top, htop	5
3.1	The free Command	5
3.2	The top Command	5
3.3	The htop Command	5
4	Memory consumption of a program using htop, virtual memory	6
4.1	Memory Usage by Data Type	6
4.2	Virtual vs. Physical Memory	6
4.3	Memory Management Tips	6
5	Interactive programs in C using scanf, fflush	7
5.1	Basic Interactive Program with scanf	7
5.2	The Problem of Buffered Output	7
5.3	The Solution: Flushing the Buffer with fflush	7
5.4	Important Considerations	8
6	Use scanf and file redirection to simulate an input	8
6.1	Creating an Input File	8

6.2	Method 1: Using Pipes	9
6.3	Method 2: Input Redirection	9
7	Don't use scanf, use fgets getline or readline	9
7.1	The Problem with scanf	9
7.2	Better Alternatives	10

1 Compiler vs interpreter

In the C programming language, we use a compiler. To understand the difference between an interpreter and a compiler, we can use an analogy of landing on a planet where inhabitants speak a strange language called "Gobbledygook". To get a mechanic to repair your spaceship, you need a translator.

1.1 The Interpreter

If you choose an interpreter:

- **Process:** The interpreter reads your first instruction, translates it immediately, and the mechanic executes it. Then it reads the second, translates, and executes, and so on.
- **Characteristics:** The interpreter stays with you, translating line by line.
- **Pros:** It allows you to correct mistakes as you go (interactive).
- **Cons:** It is a slow process because the mechanic waits for translation between steps.
- **Etymology:** "Inter" means between. The interpreter is always between your program and the computer.

1.2 The Compiler

If you choose a compiler:

- **Process:** The compiler takes your complete list of instructions and translates the whole lot at once. It then hands the translated list back to you and leaves.
- **Characteristics:** You hand the complete list to the mechanic, who executes them all in one go very quickly.
- **Pros:** Execution is very fast and efficient.
- **Cons:** Takes extra preparation time initially. If there is a mistake, it is too late to fix it during execution.
- **Etymology:** "Compile" means to pile together. It piles together your entire program and translates it all at once.

1.3 Summary

- **Interpreter:** Runs slowly, starts right away, allows you to see how things are going.
- **Compiler:** Takes preparation time, but runs very quickly and efficiently.

2 Memory representation:

RAM, cells, word, byte, bit, memory address

How does the computer remember where it has stored the value for a certain variable? To answer this, we need to understand computer memory.

2.1 Types of Memory

One role of the operating system is to manage the computer's memory.

- **RAM (Random Access Memory):** Temporary, volatile memory used to execute programs. It is quick to access.
- **Non-volatile Memory:** Permanent storage, such as the hard drive, used for storing files.

2.2 Structure of RAM

Most programs use RAM during execution.

- **Bit:** A single binary memory cell (0 or 1).
- **Word:** A group of bits forming the fundamental unit of data moved between RAM and the processor.
- **Word Size:** The number of bits in a word (e.g., 8, 16, 32, 64 bits).
- **Byte:** A unit consisting of 8 bits.

2.3 Memory Addressing

To locate data, memory cells are grouped into words, and each word is assigned an address.

- **Memory Address:** A whole number describing the location of a word in memory (similar to house addresses on a street).
- **Example:** If a computer has 8-bit words:
 - Address 0 points to the first word (first 8 bits).
 - Address 1 points to the second word (next 8 bits).

2.4 Memory and C Programming

In C, it is possible to access these memory addresses directly.

- **Access:** You can obtain the address where a variable's value is stored.
 - **Optimization:** This allows for low-level memory management and optimization of execution speed.
-

3 Manage the memory with the command line: free, top, htop

To manage memory, we first need to see how much memory is used by the programs running on the Linux system.

3.1 The free Command

The free command displays the amount of free and used system memory.

- **Options:** Use `-b` (bytes), `-k` (kilobytes), `-m` (megabytes), or `-g` (gigabytes) to set the unit.
- **Example:** `free -m` displays the table in megabytes.
- **Output:** The Mem line shows the total, used, and free memory.

3.2 The top Command

The top command shows memory usage per process.

- **Statistics:** The header shows total used and free memory.
- **VSZ (Virtual Size):** Represents the virtual memory for each program (sometimes labeled **VIRT**).
- **Sorting:** Press the **M** key to sort processes by memory usage.
- **Visualization:** Pressing the **S** key (in this specific version) switches to a view showing only memory usage.

3.3 The htop Command

htop provides a visual representation of system resources.

- **Visuals:** Bars indicate used and free memory.
- **Sorting:**
 1. Press **F6** to select "Sort by".
 2. Select **M_SIZE** (Memory Size) and press **Enter**.

3. This sorts the list by the **VIRT** column.

Monitoring memory helps identify programs putting too much pressure on the system, which can then be killed if necessary.

4 Memory consumption of a program using htop, virtual memory

We can analyze how much memory a C program uses by running it and monitoring it with htop. To do this effectively, the program must pause (e.g., waiting for user input via scanf) so it stays active in memory.

4.1 Memory Usage by Data Type

By declaring arrays of different types with one million elements, we can observe how memory consumption changes in the VIRT (Virtual Memory) column of htop:

- **Basic Program:** A simple program might use ≈ 952 KB.
- **Char Array (1,000,000 chars):** Usage increases to $\approx 1,800$ KB.
- **Int Array (1,000,000 ints):** Usage increases significantly to $\approx 4,728$ KB (approx. 4 MB).
- **Double Array (1,000,000 doubles):** Usage doubles compared to integers, reaching $\approx 8,632$ KB (approx. 8 MB).

4.2 Virtual vs. Physical Memory

An important observation is that even when a program reserves a large amount of memory (high VIRT), the system's physical memory usage (the bar at the top of htop) may not increase correspondingly.

- **Virtual Memory:** The amount of memory the program has asked for or reserved.
- **Physical Memory:** The actual RAM hardware used.
- **Lazy Allocation:** Linux is smart; if a program reserves memory (like a large array) but never writes to or reads from it, the system does not immediately allocate physical RAM. Physical memory is consumed only when the virtual memory is actually used.

4.3 Memory Management Tips

- **Optimization:** Ensure that large arrays or variables are used efficiently.

- **Dynamic Allocation:** If you allocate memory dynamically in C, always remember to **free** it to return it to the system for other programs.
-

5 Interactive programs in C using `scanf`, `fflush`

This section explains how to create interactive C programs and handle a common issue related to terminal output buffering.

5.1 Basic Interactive Program with `scanf`

An interactive program prompts the user for input and reads it from the terminal.

- **Prompting:** Use `printf` to display a question to the user.
- **Reading Input:** Use `scanf` to wait for and read the user's input.
- **Example:** A program can ask for a family name, first name, and age.
 - `scanf("%s", familyName)` reads a string into a character array.
 - `scanf("%d", &age)` reads an integer into an integer variable.
- Typically, each prompt with `printf` ends with a newline character (`\n`) to move the cursor to the next line for the user's input.

5.2 The Problem of Buffered Output

The `printf` function in C uses a buffer, which is a temporary memory area.

- **Buffering:** Output is not always sent directly to the screen. It is first stored in a buffer. The buffer is "flushed" (its contents are sent to the terminal) when certain conditions are met.
- **Newline Trigger:** A common trigger for flushing the buffer is encountering a newline character (`\n`).
- **The Issue:** If you write a `printf` statement for a prompt without a trailing `\n`, the text may not appear on the screen immediately. The program might wait for the user's input with `scanf` before the prompt is even visible. All the buffered prompts might appear in a single line only after the program finishes or the buffer is flushed for another reason.

5.3 The Solution: Flushing the Buffer with `fflush`

To ensure prompts appear immediately without a newline, you can manually flush the output buffer.

- **The Command:** The `fflush(stdout)` function forces the standard output buffer (`stdout`) to be written to the screen.
- **Implementation:** Place `fflush(stdout);` immediately after each `printf` statement that you want to appear on the screen right away.
- **Example:**

```
printf("What is your first name? ");  
fflush(stdout);  
scanf("%s", firstName);
```

- This ensures the question is visible before the program waits for the `scanf` input, even without a `\n`. The user's answer will then appear on the same line as the question.

5.4 Important Considerations

- **System Variability:** This buffering behavior is not consistent across all Linux systems. Some systems or C libraries might flush output more frequently, even without a newline.
 - **Best Practice:** To write portable and predictable interactive programs, always ensure the output buffer is flushed. The two main ways are:
 1. End your prompt strings with `\n`.
 2. Call `fflush(stdout);` after the prompt.
-

6 Use scanf and file redirection to simulate an input

We can automate or simulate user input for interactive programs using pipes and file redirection. This is useful for testing or when running programs in scripts.

6.1 Creating an Input File

First, create a file containing the inputs expected by the program, separated by newlines.

- **Example:** Create a file named `answers` with the following content:

```
sharrock  
remi  
18
```

- You can create this using `nano` or `cat > answers`.

6.2 Method 1: Using Pipes

You can pipe the output of a command (like `cat`) into your program.

- **Command:** `cat answers | ./program`
- **Explanation:** The shell takes the output of `cat answers` and feeds it as standard input to `./program`. The `scanf` functions in the program read from this stream just as if it were typed on the keyboard.

6.3 Method 2: Input Redirection

You can redirect a file directly into the program's standard input using the less-than sign (`<`).

- **Command:** `./program < answers`
 - **Explanation:** This tells the shell to open the file `answers` and connect it to the input of `./program`.
 - **Note:** This is the reverse of output redirection (`>`), which sends output to a file. Here, we send a file to the input.
-

7 Don't use `scanf`, use `fgets` `getline` or `readline`

Using `scanf` for string input can be dangerous and lead to unexpected behavior for both programmers and users.

7.1 The Problem with `scanf`

`scanf` splits input based on whitespace (spaces, tabs, newlines).

- **Scenario:** A program asks for Family Name, First Name, and Age sequentially.
- **Input:** If the user types "Sharrock Remi 18" at the first prompt and hits Enter.
- **Result:**
 - Family Name becomes "Sharrock".
 - First Name automatically becomes "Remi" (consuming the next part of the buffer).
 - Age automatically becomes "18".
- The program skips waiting for the subsequent inputs because `scanf` consumed the space-separated values from the first line. This behavior makes `scanf` risky for robust user interaction.

7.2 Better Alternatives

To handle string input more safely, consider these functions:

- **fgets:** Reads the entire line, including spaces. Note that it also captures the newline character (`\n`) at the end, which you may need to remove.
- **getline:** A POSIX-standard function (available on most Unix-like systems) that reads an entire line.
- **GNU readline:** A powerful function specific to Linux/GNU systems.

For robust C programming, it is recommended to research these functions or take a dedicated C programming course.
