Printf Under The Hood

Manuel Pérez

March 6, 2025

Contents

1 printf under the hood

1.1 The program

Will look into a really simple program that prints a text on a terminal and waits until a key is pressed:

```
sed -n 1~1p printfuth.cpp
```

```
#include <bits/posix_opt.h>
#include <cstdio>
#include <termios.h>
#include <unistd.h>

int keypress() {
  struct termios saved_state, new_state;
  int c;
  if (tcgetattr(STDIN_FILENO, &saved_state) == -1) {
    return EOF;
```

```
  }
  new_state = saved_state;
  new_state.c_cflag &= ~(ICANON | ECHO);
  new_state.c_cc[VMIN] = 1;
  new_state.c_cc[VTIME] = 0;
  if (tcsetattr(STDIN_FILENO, TCSANOW, &new_state) == -1) {
    return EOF;
  }
  c = getchar();
  tcsetattr(STDIN_FILENO, TCSANOW, &saved_state);
  return c;
}

int main() {
  std::printf("Running on terminal: %s\n", ttyname(STDIN_FILENO));
  std::printf("Hello, world!");
  keypress();
  return 0;
}
```

1.2 The assembler instructions

First things, first. The program is compiled into this (assembler):

```
g++ -g printfuth.cpp -o printfuth
objdump --disassemble=main printfuth > printfuth-disassemble
sed -n 1~1p printfuth-disassemble
```

It can be seen that what the program does is basically a call to `printf`.

1.2.1 TODO What's the purpose of `lea 0xeac(%rip),%rax`?

The LEA (Load Effective Address) instruction computes the effective address of a memory location and stores it in a register.

Here, it's storing the address where the string "Hello, world!" resides into the `rax` register. From that register it will be copied onto the `rdi` register, which is the one that must hold the first argument to a function called (according to the calling convention).

The address is expressed as an offset from the contents of the `rip` register (which is the instruction pointer). As it can be seen, the resulting address

is `0x2004`, which equals `0x12dc + 0xd28` (the value of the `rip` register plus the offset `0xd28`).

The section of the executable file where the string is stored is this:

```
readelf -x .rodata printfuth
```

1.2.2 TODO How is the argument passed to `printf`?

1.3 TODO Execution context

The program is run by a shell, whose standard output is directed to a terminal window (a terminal emulator window, to be precise) on the screen.

1.4 TODO The system calls

```
strace ./printfuth 2>&1 1>/dev/null
```

The key system call here is `write(1, "Hello, world!", 13)`.

It writes to the file with descriptor 1, which is, by default, the standard output in Linux.

1.5 Scheme

Looks like this:

- `printf` makes a `write` syscall.
- `write` syscall writes the string to file with file descriptor 1, which is, by default, the *standard output*.
- The file descriptor 1 (`STDOUT`) is handled by a driver that controls a *device* which is a pseudoterminal (something like `/dev/pty/3`).
- The pseudoterminal is connected to a terminal emulator aplication (the one we are running the command from).
- The terminal emulator receives the characters and prints them onto its window.

Still obscure parts:

- What does it mean that the file descriptor 1 is handled by a driver that controls a pseudoterminal device?

- How does the terminal emulator write characters on the window (how are shapes, sizes, etc., handled)?
- What are the details about how the pseudoterminal is connected to the terminal emulator?
- What processes are forked, exec'ed or the like for all these things to happen?