◆ Your Code (with comments)

```
#include <unistd.h>   // contains system call wrappers like write(), read(),
close()
#include <stdio.h>    // for printf() and other standard I/O library functions
#include <string.h>   // for strlen()
#include <stdlib.h>   // for exit()

// print   <-- looks like an accidental leftover word

int main() {
    // Using printf() would go through the C standard library (buffered I/O):
    // printf("Hello Linux\n");

    // Instead, here we call write() directly, which is a system call wrapper.
    // write(fd, buffer, count)
    // fd = 1 means standard output (stdout)
    // fd = 0 means standard output (stdin)
    // fd = 2 means standard output (stderr)
    if (write(1, "Hello Linux\n", strlen("Hello Linux\n")) < 0)
        exit(-1);   // exit with error if write failed

    return 0;
}
```

◆ System Calls vs. Library Functions

- **System Call (syscall):**
  A direct request from a user program to the Linux kernel (e.g., `write()`, `read()`, `fork()`).
  These switch CPU mode from **user space** → **kernel space**, where the OS can access hardware safely.
- **Wrapper Functions (glibc):**
  In C, you don't usually invoke system calls by raw `syscall` numbers.
  Instead, the GNU C Library (glibc) provides **wrappers**.
  - Example: `write()` in `<unistd.h>` is a **wrapper** around the `sys_write` system call.
  - It does some minor error handling (like setting `errno`) and then calls the kernel.
- **Library Functions (not syscalls):**
  Functions like `printf()` are higher-level **C library functions** that may internally use

system calls (`write`) but add extra features like buffering, formatting, etc.

◆ Example Comparison

- `printf("Hello Linux\n");`
  → goes through stdio buffering in libc, eventually calls `write(1, buffer, length)` internally.
- `write(1, "Hello Linux\n", 12);`
  → direct syscall wrapper, unbuffered, goes straight to kernel.

◆ Manual Pages (`man` command in Linux)

Linux manual pages (man pages) are grouped into **sections**:

- `man 1` → user commands (e.g., `ls` , `cat`)
- `man 2` → system calls (kernel interface functions)
- `man 3` → library functions (glibc, standard C library)
- `man 5` → file formats and configuration files
- `man 7` → conventions, overviews (e.g., `man 7 signal`)
- `man 8` → system administration commands

So in your case:

- `man 2 write` → shows the **system call interface** for `write()` .
- `man 3 printf` → shows the **C library function** for `printf()` .
- `man 2 intro` → introduction to system calls (good overview).
- `man man` → manual for the `man` command itself.
- `man -k something` → search for relevant man pages (like a keyword search).

◆ System Call Wrappers and `man 2 syscall`

1. System Calls vs Wrappers

- **System calls** = direct interface to the Linux kernel (file I/O, processes, memory, etc.).
- Each syscall has a **number** (e.g., `__NR_write = 1` on x86_64).
- **Wrappers (glibc)** = C functions (`write()` , `read()`) that prepare registers, invoke the syscall instruction, and translate errors into `errno` .

- Example:

```
// Wrapper: write() from unistd.h
write(1, "Hello\n", 6);

// Raw syscall (no wrapper):
syscall(SYS_write, 1, "Hello\n", 6);
```

---

2. `man 2 syscall` and Tables

`man 2 syscall` documents the **generic syscall() function** and shows **how arguments are passed per architecture**.

Table A: Instruction & Return

```
Arch/ABI    Instruction    Syscall# reg    Return reg
i386        int $0x80      eax             eax
x86_64      syscall        rax             rax
arm64       svc #0         x8              x0
```

- Tells you:
  - Which **instruction** to enter kernel mode.
  - Which register holds the **syscall number**.
  - Where the **return value** comes back.

Table B: Argument Registers

```
Arch/ABI    arg1   arg2   arg3   arg4   arg5   arg6
i386        ebx    ecx    edx    esi    edi    ebp
x86_64      rdi    rsi    rdx    r10    r8     r9
```

- Tells you: which registers must contain syscall arguments.

---

3. Example: `write(int fd, const void *buf, size_t count)`

On x86 (i386, old 32-bit)

- Syscall number in `eax = 4` (`__NR_write`)
- Args: `ebx = fd` , `ecx = buf` , `edx = count`
- Trap with `int 0x80`

Assembly:

```
mov eax, 4       ; __NR_write
mov ebx, 1       ; fd = 1 (stdout)
mov ecx, msg     ; buffer address
mov edx, 13      ; length
int 0x80         ; enter kernel
```

On x86-64 (modern 64-bit)

- Syscall number in `rax = 1` (`__NR_write`)
- Args: `rdi = fd` , `rsi = buf` , `rdx = count`
- Trap with `syscall`

Assembly:

```
mov rax, 1       ; __NR_write
mov rdi, 1       ; fd = 1
mov rsi, msg     ; buffer
mov rdx, 13      ; length
syscall          ; enter kernel
```

---

4. Why This Matters

- In **normal C**, you just call `write()` .
- Wrappers hide all the register setup and error handling.
- If you bypass libc (using `syscall()` or raw assembly), you must follow these tables exactly.
- Useful for:
  - Writing minimal programs (no libc).
  - Kernel-level debugging.
  - Understanding how user space talks to the kernel.
