File Descriptors and System Calls

1. File Descriptors

- **Definition**: A file descriptor (FD) is an integer that identifies an open I/O stream like files, network connections, terminals, or pipes.
- **File Descriptor Table**: It is an array of kernel objects where file descriptors are indexes.
- **Components**:
  - **File Reference**: Path of the file, e.g., `/home/example.txt`.
  - **File Position**: Offset in the file indicating the current position.
- **Types of Files**:
  - **Seekable**: Position can be explicitly changed (e.g., disk files).
  - **Non-Seekable**: Position cannot be changed (e.g., pipes, terminals).

---

2. System Calls

- **Definition**: A request made by a user program to the OS for performing privileged operations.
- **Steps**:
  - Save CPU state → Switch to kernel mode → Perform the operation → Return result to user space.
- **Why Expensive?**: It involves context switching, state saving, and kernel intervention.

---

3. System Calls vs. Function Calls

Aspect	Function Call	System Call
Domain	User-level only	Kernel-level intervention
Security	Trusted environment	OS has super-privileges
Speed	Faster	Slower due to context switching

- **Example**:
  - Function Call: `fopen()`
  - System Call: `open()`

---

4. Sequence of a System Call

- Steps:
  1. Use a **software interrupt** to switch to kernel mode.
  2. System call number (e.g., `read()`) is placed in **EAX register**.
  3. Kernel function performs the requested action.
  4. CPU restores state and returns control to the user program.

---

5. Important System Calls

- **File Management**: `open()`, `read()`, `write()`, `close()`, `lseek()`.
- **Process Management**: `fork()`, `exec()`, `getpid()`.
- **Advanced Calls**: `pipe()`, `dup()`, `unlink()`.

---

6. Standard Input, Output, and Error

- **FD 0**: Standard Input (stdin)
- **FD 1**: Standard Output (stdout)
- **FD 2**: Standard Error (stderr)

---

7. Example Code

Open and Read a File:

```
c

#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int fd = open("file.txt", O_RDONLY);
    if (fd < 0) { perror("Error"); return 1; }

    char buffer[20];
    int sz = read(fd, buffer, 10);
    buffer[sz] = '\0'; // Null-terminate the string
    printf("Read: %s\n", buffer);

    close(fd);
    return 0;
}
```

Linux Device Drivers

1. Device Drivers

- **Definition**: Software that allows the OS to communicate with hardware devices.
- **Main Role**: Translates user-level commands into hardware-specific operations.

---

2. Why Use Device Drivers?

- They act as **interfaces** between:
    - **Applications ↔ Operating System ↔ Hardware**.

---

3. Classes of Device Drivers

1. **Character Devices**:
    - Handle data **one byte at a time**.
    - Examples: `/dev/tty` (terminal), `/dev/console`.
2. **Block Devices**:
    - Handle data in **blocks** (512 bytes or larger).
    - Examples: Hard disks, USB drives.
3. **Network Devices**:
    - Transfer **packets** of data.
    - Example: Ethernet (`eth0`).

---

4. Loading Device Drivers

- **Kernel Modules**:
    - Drivers can be loaded **dynamically** into the kernel without recompiling.
    - Commands:
        - **Load Module**: `insmod <module>`
        - **Remove Module**: `rmmod <module>`

---

5. Writing a Simple Device Driver

Hello World Driver Example:

```
c

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");

static int __init hello_init(void) {
    printk(KERN_INFO "Hello, world\n");
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "Goodbye, world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- **Commands**:
    1. Compile with `make`.
    2. Load module: `insmod hello.ko`.
    3. Remove module: `rmmod hello`.

---

6. Security Concerns

- **Buffer Overflow**: Ensure memory is not overwritten.
- **Privilege Restriction**: Only root users can load modules.
- **Uninitialized Memory**: Zero out kernel memory before exposing it to user space.

---

7. Device Management

- **Inserting Driver**:
    - Previously: Kernel had to be **recompiled**.
    - Now: Use **runtime modules** (simpler and efficient).

---

8. Important System Calls

- **File System**:
    - `open()`, `read()`, `write()`, `close()`.
- **Device Control**:
    - Uses ioctl system calls for custom device management.