File Handling in C: Exploring I/O System Calls and File Descriptors

File handling is a crucial aspect of programming, enabling the reading and writing of data to files. In the C programming language, file handling is facilitated through file descriptors and I/O system calls. This article will delve into the core concepts of file handling in C, covering file descriptors, standard file descriptors, I/O system calls, file permissions, and the distinction between functions and system calls.

🔷 Understanding File Descriptors:

File descriptors are integer values used to identify and track open files within a process. They serve as handles or references to files or I/O streams. File descriptors are fundamental in operating systems and are associated with the POSIX standard.

```
#include <fcntl.h>

int main() {
    int fileDescriptor;
    fileDescriptor = open("example.txt", O_CREAT | O_WRONLY, 0644);
    // Use the file descriptor for further operations
    // ...
    close(fileDescriptor);
    return 0;
}
```

🔷 The Three Standard File Descriptors:

C provides three standard file descriptors: STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO. They represent the default input, output, and error streams, respectively.

- STDIN_FILENO (0): Represents the standard input stream, typically the keyboard.
- STDOUT_FILENO (1): Represents the standard output stream, typically the console or terminal.
- STDERR_FILENO (2): Represents the standard error stream used for error messages and diagnostics.

```
Example: Writing to Standard Output:
#include <unistd.h>

int main() {
    write(STDOUT_FILENO, "Hello, world!\n", 14);
    return 0;
}
```

🔷 The open and close Functions

Opening a file is the operation that prepares the file for further processing. This operation is performed using the open function:

int open(const char *pathname, int oflag, [, mode_t mode]);

The function returns -1 in case of an error. Otherwise, it returns a file descriptor associated with the opened file.

🔶 **Parameters:**

- pathname – contains the file name.
- oflag – file opening options. This is actually a bit sequence, where each bit or group of bits has a specific meaning. For each of these meanings, there is a corresponding constant defined in the C header file **fcntl.h**. These constants can be combined using the **bitwise OR (|)** operator in C, allowing multiple options to be set in the oflagparameter. Below are some of these constants:
  - O_RDONLY – open for **reading only**.
  - O_WRONLY – open for **writing only**.
  - O_RDWR – open for **both reading and writing**.
  - O_APPEND – open for **appending** at the end of the file.
  - O_CREAT – create the file if it does not already exist; when used with this option, the open function must also receive the mode parameter.
  - O_EXCL – **exclusive** file creation: if O_CREAT is used and the file already exists, the open function will return an error.
  - O_TRUNC – if the file exists, its contents will be **deleted**.
- mode – used **only** when the file is being created and specifies the access permissions associated with the file. These permissions are obtained by combining constants using the **bitwise OR (|)** operator, just like in the previous option. The constants include:
  - S_IRUSR – **read permission** for the file owner (user).
  - S_IWUSR – **write permission** for the file owner (user).
  - S_IXUSR – **execute permission** for the file owner (user).
  - S_IRGRP – **read permission** for the owner's group.
  - S_IWGRP – **write permission** for the owner's group.
  - S_IXGRP – **execute permission** for the owner's group.
  - S_IROTH – **read permission** for other users.
  - S_IWOTH – **write permission** for other users.
  - S_IXOTH – **execute permission** for other users.

int creat(const char *pathname, mode_t mode);
int close(int);

🔷 Understanding File Descriptors in Linux

A **file descriptor (FD)** is a non-negative integer used by the operating system to reference open files or I/O resources such as sockets and pipes. It acts as an index into a table of open files maintained by the OS for each process.

🔶 File Descriptor Table

Each process has a file descriptor table mapping integers to file objects:

FD	Object
0	stdin (keyboard input)
1	stdout (terminal output)
2	stderr (error messages)
3+	Opened files/sockets

Are File Descriptors Unique to a Single Process or the Entire OS?

File descriptors (FDs) are **unique within each process**, not globally across the operating system. Each process has its **own file descriptor table**, maintained by the kernel. The same FD number (e.g., 3) in two different processes may refer to different files because each process has a separate file descriptor table.

🔶 Why Are They Process-Specific?

1. **Isolation:** Each process should have independent access to files without interference from others.
2. **Security:** One process cannot directly access another process's files using its file descriptors.
3. **Efficiency:** The OS can efficiently manage open files per process instead of maintaining a global FD table.

🔶 Why Can We Duplicate File Descriptors?

File descriptors can be **duplicated** using dup() or dup2() to make multiple FDs refer to the **same open file description**.

Concept:

- A file descriptor is just an **index** in a process's file descriptor table.
- It points to an **open file description**, which stores information about the open file (e.g., position, mode).
- Duplicating an FD creates a new entry in the file descriptor table, pointing to the same open file description.

🔷 What Happens When I Run ./app in Two Different Terminals?

If I open two separate terminals and run ./app in each, the behavior depends on how **stdout is handled by the OS**.

🔶 Does Each Process Have a Different stdout File Descriptor?

Each terminal (like /dev/pts/1 and /dev/pts/2) is a separate device file in the OS. When I run ./app in both terminals:

- Each process gets a separate file descriptor (1 for stdout).
- The stdout file descriptor in each process points to a different terminal file.

```
$ tty  # Check which terminal you're in
/dev/pts/1  # In first terminal
$ ./app  # Runs in /dev/pts/1
And in the second terminal:
```

```
$ tty
/dev/pts/2  # A completely separate terminal


$ ./app  # Runs in /dev/pts/2
Since each terminal is a different file, stdout from ./app in /dev/pts/1 does
NOT interfere with stdout in /dev/pts/2.
```

🔷 Common functions

```
int open(const char *pathname, int oflag, [, mode_t mode]);
intcreat (const char *pathname, mode_t mode);
int close (int filedes);
ssize_t read(int fd, void *buff, size_t nbytes);
ssize_t write(int fd, void *buff, size_t nbytes);
off_t lseek(int fd, off_t offset, int pos);

int mkdir(const char *pathname, mode_t mode)
int rmdir(const char *pathname)
```

🔶 Key Features:

- Directly interacts with the **kernel's file descriptor table**.
- Requires **manual buffering** for efficient reading/writing.
- Used for **low-level file operations**, like setting flags (O_APPEND, O_CREAT).
- Works well with **system calls** like read(), write(), and lseek().

🔶 Understanding System Calls:

System calls are functions provided by the operating system kernel that allow user programs to interact with the underlying system's resources. They provide access to low-level operations like file I/O, process management, and network communication.

🔶 Difference Between Functions and System Calls:

Functions in C are typically provided by libraries and operate within the user space of a program. System calls, on the other hand, directly interface with the operating system's kernel to access privileged operations and system resources.

🔶 Conclusion:

File handling in C involves understanding file descriptors, standard file descriptors, I/O system calls, file permissions, and the distinction between functions and system calls. With the knowledge of these concepts, I can confidently perform file operations, such as creating, opening, closing, reading, and writing files, in your C programs.

🔷 Working with standard library functions for opening, reading, writing and closing files in a more abstract level

```
            FILE *fopen(const char *filename, const char *mode)
            int fclose(FILE *stream)
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

```
size_t fwrite( void *ptr, size_t size, size_t
nmemb,  FILE *stream)
```

🔷stat() vs fstat() vs lstat()

Function	Input Type	Follows Symlink?	Use Case
stat()	File path (char *)	✅ Yes	Get info about the actual file (resolves symlink).
fstat()	File descriptor (int)	✅ Yes	Get info from an open file.
lstat()	File path (char *)	❌ No	Get info about the symlink itself.