1. `open()` System Call

The `open()` system call is used to open a file and obtain a file descriptor.

Syntax:

```
int open(const char *pathname, int flags, mode_t mode);
```

- `pathname`: Name of the file to open.
- `flags`: Specifies the mode (e.g., `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`).
- `mode`: File permissions (used when `O_CREAT` is specified).
- **Returns**: A **file descriptor** (non-negative integer) on success, `-1` on failure.

Example:

```
int fd = open("file.txt", O_RDONLY);
if (fd == -1) {
    perror("Error opening file");
}
```

2. `close()` System Call

The `close()` system call is used to close an open file descriptor, releasing system resources.

Syntax:

```
int close(int fd);
```

- `fd`: File descriptor to close.
- **Returns**: `0` on success, `-1` on failure.

Example:

```
close(fd);
```

3. `read()` System Call

The `read()` system call is used to read data from a file.

Syntax:

```
ssize_t read(int fd, void *buf, size_t count);
```

- `fd`: File descriptor to read from.
- `buf`: Buffer to store the data.
- `count`: Maximum number of bytes to read.
- **Returns**: Number of bytes read, `0` for EOF, `-1` on error.

Example:

```
char buffer[100];
ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
```

4. `write()` System Call

The `write()` system call is used to write data to a file.

Syntax:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd`: File descriptor to write to.
- `buf`: Buffer containing the data.
- `count`: Number of bytes to write.
- **Returns**: Number of bytes written, `-1` on failure.

Example:

```
char msg[] = "Hello, world!";
write(1, msg, sizeof(msg));  // Writes to stdout
```

Summary Table

System Call	Purpose
`open()`	Opens a file and returns a file descriptor.
`close()`	Closes an open file descriptor.
`read()`	Reads data from a file into a buffer.
`write()`	Writes data from a buffer to a file.

These system calls are fundamental for **file handling and inter-process communication** in Linux

`pipe()` System Call in Linux

The `pipe()` system call is used for **Inter-Process Communication (IPC)**, allowing data transfer between processes. It creates a unidirectional data channel.

Syntax:

```
int pipe(int fd[2]);
```

- `fd[0]` → **Read end** of the pipe.
- `fd[1]` → **Write end** of the pipe.
- **Returns**: `0` on success, `-1` on failure.

How `pipe()` Works

1. **Parent process creates a pipe**.
2. **`fork()` is used to create a child process**.
3. **Parent writes to `fd[1]` (write end)**.
4. **Child reads from `fd[0]` (read end)**.

Example: Using `pipe()` with `fork()`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    int fd[2];
    char buffer[100];
    pid_t pid;

    if (pipe(fd) == -1) {
        perror("pipe failed");
        exit(1);
    }

    pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }

    if (pid == 0) { // Child process
        close(fd[1]); // Close write end
        read(fd[0], buffer, sizeof(buffer));
```

```
        printf("Child received: %s\n", buffer);
        close(fd[0]);
    } else { // Parent process
        close(fd[0]); // Close read end
        char msg[] = "Hello from parent!";
        write(fd[1], msg, sizeof(msg));
        close(fd[1]);
    }

    return 0;
}
```

Explanation

- The parent process **creates a pipe** (`pipe(fd)`).
- It **forks** a child process.
- The **parent writes** a message to `fd[1]` (write end).
- The **child reads** the message from `fd[0]` (read end).
- Both processes close the unused ends of the pipe.

Key Points

- **Unidirectional**: One end writes, the other reads.
- **Blocking**: `read()` waits if there's no data, and `write()` waits if the buffer is full.
- **Used for IPC** between parent and child processes.