Complete Guide to I/O System Calls in C

File Descriptors, Standard Streams, and Low-Level I/O Operations

Understanding Linux/Unix System Programming

Systems Programming Masterclass

Compiled: January 15, 2026

Abstract

Abstract: This comprehensive guide explains Input/Output System Calls in C programming on Linux/Unix systems. We explore the fundamental concepts of file descriptors, standard streams, and the five essential I/O system calls: create, open, close, read, and write. Understanding these low-level operations is crucial for system programming, understanding how operating systems work, and writing efficient, low-level code.

Contents

1 Introduction to System Calls

What Are System Calls?

System calls are interfaces between user programs and the operating system kernel. They allow programs to request services from the kernel for operations they don't have direct access to, such as:

- File operations (create, read, write, delete)
- Process management
- Memory allocation
- Device communication
- Network operations

1.1 Why Learn I/O System Calls?

- **Direct access** to operating system functionality
- **Complete control** over file operations
- **Understanding** how high-level I/O functions work internally
- **Essential for** system programming and embedded systems
- **Better performance** in certain scenarios
- **Fundamental knowledge** for Linux/Unix programming

1.2 Difference Between Library Functions and System Calls

Aspect	Library Functions	System Calls
Location	User-space libraries	Kernel-space
Performance	Faster (no context switch)	Slower (context switch required)
Examples	`printf()`, `fopen()`	`write()`, `open()`
Buffering	Usually buffered	Usually unbuffered
Flexibility	Higher level, easier to use	Lower level, more control
Portability	More portable	Less portable (OS-specific)

1.3 Document Structure

- **Section 2:** File Descriptors and Standard Streams
- **Section 3:** The 5 Essential I/O System Calls
- **Section 4:** Practical Examples and Use Cases
- **Section 5:** Advanced Topics and Best Practices
- **Section 6:** Common Pitfalls and Debugging

2 File Descriptors and Standard Streams

2.1 What is a File Descriptor?

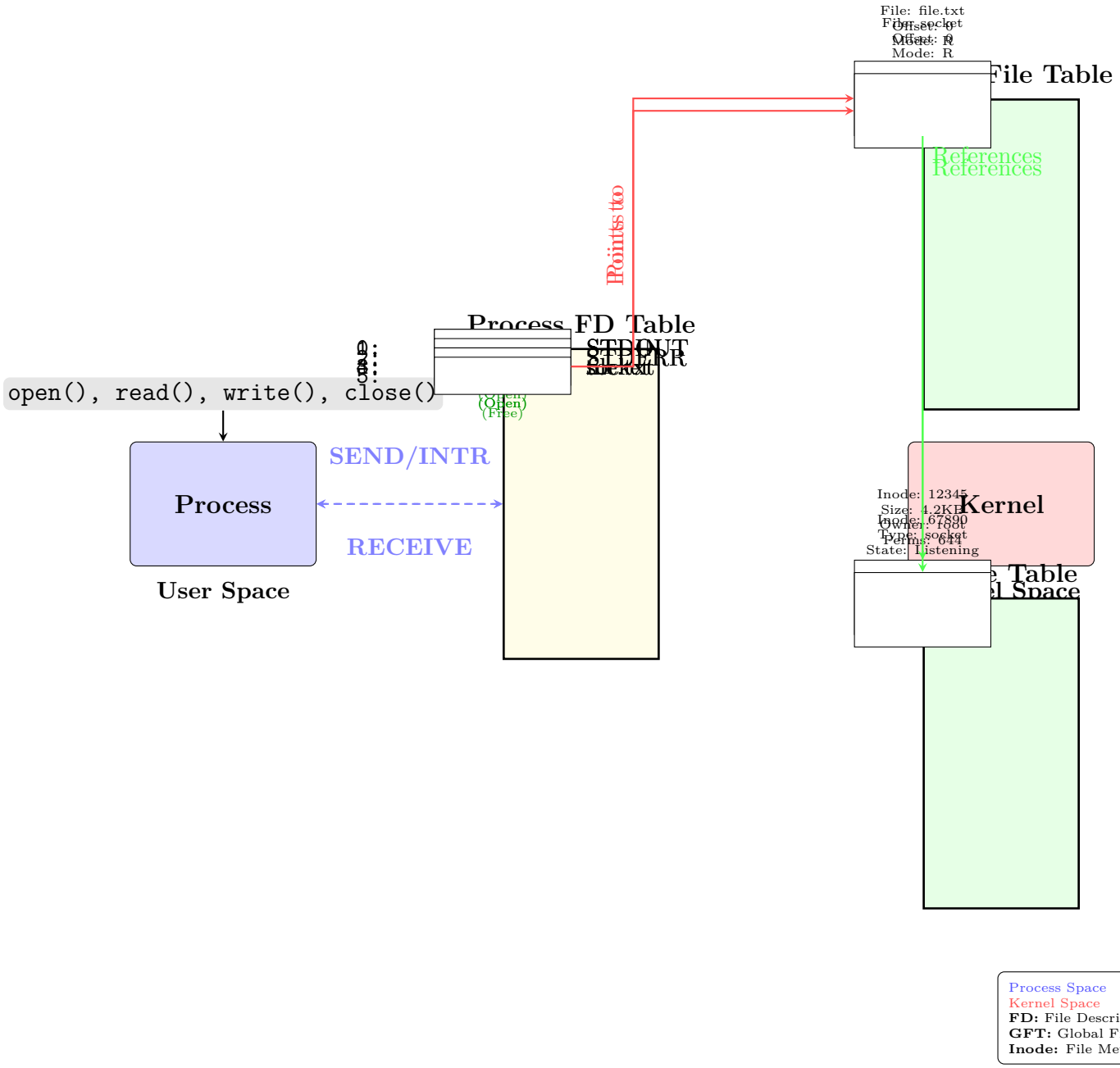
File Descriptor Definition

A **file descriptor** (FD) is a non-negative integer that uniquely identifies an opened file within a process. It's an index into the kernel's **file descriptor table** for that process.

2.1.1 Key Characteristics

- **Integer value** (0, 1, 2, 3, 4, ...)
- **Process-specific** (each process has its own FD table)
- **Kernel-managed** (maintained by the OS)
- **Lowest available** integer is always used for new files
- **Standard FDs** (0, 1, 2) are reserved for special purposes

2.2 Kernel Data Structures for File I/O


File: file.txt
Offset: 0
Mode: R
File Table
References
Points to
Process FD Table
STDOUT
STDERR
(Open)
(Free)
open(), read(), write(), close()
Process
SEND/INTR
RECEIVE
User Space
Inode: 12345
Size: 1.2KB
Owner: root
Perms: 644
Kernel
Inode: 67890
Type: socket
State: Listening
Table
Space
Process Space
Kernel Space
FD: File Descriptor
GFT: Global File T
Inode: File Metada

How File Descriptors Work

1. **System Call**: Process calls `open()` to request file access
2. **FD Allocation**: Kernel finds lowest free FD in Process FD Table
3. **GFT Entry**: Kernel creates entry in Global File Table with file state
4. **Inode Lookup**: GFT entry references file metadata in Inode Table
5. **Pointer Chain**: FD → GFT entry → Inode entry
6. **I/O Operations**: All `read()`/`write()` use FD to traverse chain
7. **Cleanup**: `close()` removes FD entry, may free GFT entry

Key Points:

- Each process has its own FD table
- Multiple FDs can point to same GFT entry (file sharing)
- GFT entries track per-open-file state (offset, mode)
- Inode table contains persistent file metadata

2.2.1 Standard Streams in Action

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

int main() {
    char buffer[100];

    // Write to stdout (FD 1)
    write(STDOUT_FILENO, "Enter your name: ", 17);

    // Read from stdin (FD 0)
    int bytes_read = read(STDIN_FILENO, buffer, sizeof(buffer)-1);

    if (bytes_read > 0) {
        buffer[bytes_read] = '\0';
        // Write to stdout
        write(STDOUT_FILENO, "You entered: ", 13);
        write(STDOUT_FILENO, buffer, bytes_read);

        // Write to stderr (FD 2)
        write(STDERR_FILENO, "[DEBUG] Name captured\n", 22);
    }

    return 0;
}
```

Listing 1: Demonstrating Standard File Descriptors

2.3 File Descriptor Table Visualization

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

```c
#include <fcntl.h>
#include <stdlib.h>

int main() {
    int ifd, ofd, pipefd[2];

    printf("Kernel Process File Descriptor Table\n");
    printf("==================================================\n");
    printf("FD\tType\t\tDescription\n");
    printf("--------------------------------------------------\n");

    // 1. Standard FDs (0, 1, 2)
    printf("%d\tStandard\tSTDIN (Keyboard/Input)\n", STDIN_FILENO);
    printf("%d\tStandard\tSTDOUT (Terminal/Output)\n", STDOUT_FILENO);
    printf("%d\tStandard\tSTDERR (Terminal/Error)\n", STDERR_FILENO);

    // 2. Open Files (Allocates 3 and 4)
    ifd = open("in.txt", O_RDONLY | O_CREAT, 0644);
    ofd = open("out.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    printf("%d\tFile\t\tin.txt (Input File)\n", ifd);
    printf("%d\tFile\t\tout.txt (Output File)\n", ofd);

    // 3. Create a Pipe (Allocates 5 and 6)
    if (pipe(pipefd) == -1) {
        perror("Pipe failed");
        return 1;
    }
    printf("%d\tPipe Read\tInternal Buffer Read End\n", pipefd[0]);
    printf("%d\tPipe Write\tInternal Buffer Write End\n", pipefd[1]);

    // Cleanup
    close(ifd); close(ofd);
    close(pipefd[0]); close(pipefd[1]);

    return 0;
}
```

Listing 2: Visualizing File Descriptor Table

Expected Output

```
Kernel Process File Descriptor Table
==================================================
FD      Type            Description
--------------------------------------------------
0       Standard        STDIN (Keyboard/Input)
1       Standard        STDOUT (Terminal/Output)
2       Standard        STDERR (Terminal/Error)
3       File            in.txt (Input File)
4       File            out.txt (Output File)
5       Pipe Read       Internal Buffer Read End
6       Pipe Write      Internal Buffer Write End
```

2.4 Key Concepts Summary

Important Rules

- **Rule 1:** `open()` always returns the smallest unused file descriptor
- **Rule 2:** Standard FDs (0,1,2) are automatically opened for every process
- **Rule 3:** After `close()`, the FD becomes available for reuse
- **Rule 4:** File descriptors are inherited by child processes (fork)
- **Rule 5:** `dup()` and `dup2()` create copies of file descriptors

3 The 5 Essential I/O System Calls

3.1 1. Create System Call

create() - Create a New File

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

3.1.1 Function Details

- **Header:** `<fcntl.h>` and `<unistd.h>`
- **Purpose:** Create a new empty file or truncate an existing file
- **Permissions:** `mode` specifies file permissions (octal)
- **Equivalent to:** `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)`
- **Limitation:** Always opens file for writing only

3.1.2 Permission Bits (mode)

Octal	Symbolic	Description
0400	r——	Owner can read
0200	-w——-	Owner can write
0100	–x——	Owner can execute
0040	—r——	Group can read
0020	—-w—-	Group can write
0010	——x—	Group can execute
0004	———r–	Others can read
0002	———-w-	Others can write
0001	————x	Others can execute

3.1.3 create() Example

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

int main() {
    // Create a new file with permissions rw-r--r--
    int fd = creat("newfile.txt", 0644);

    if (fd == -1) {
        perror("Error creating file");
        return 1;
    }

    printf("File 'newfile.txt' created successfully\n");
    printf("File descriptor: %d\n", fd);

```

```
    // Write some data to the file
    char *data = "Hello, World!\n";
    write(fd, data, 14);

    // Close the file descriptor
    close(fd);

    return 0;
}
```

Listing 3: Using creat() System Call

3.1.4 How creat() Works Internally

1. Kernel checks if file already exists
2. If exists and permissions allow, truncates to zero length
3. If doesn't exist, creates new file on disk
4. Creates entry in Global File Table
5. Allocates lowest unused FD in process FD table
6. Returns FD or -1 on error

3.2 2. Open System Call

open() - Open or Create a File

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

3.2.1 Function Details

- **Header:** `<fcntl.h>` and `<unistd.h>`
- **Purpose:** Open existing file or create new file
- **Flexible:** More flexible than `creat()` (can open for read/write)
- **Flags:** Control how file is opened (read/write, create, append, etc.)
- **Mode:** Required only when `O_CREAT` is used

3.2.2 Common Open Flags

Flag	Value	Description
`O_RDONLY`	0	Open for reading only
`O_WRONLY`	1	Open for writing only
`O_RDWR`	2	Open for reading and writing
`O_CREAT`	0100	Create file if it doesn't exist
`O_EXCL`	0200	With `O_CREAT`, fail if file exists
`O_TRUNC`	01000	Truncate file to zero length
`O_APPEND`	02000	Append to file on each write
`O_NONBLOCK`	04000	Non-blocking I/O
`O_SYNC`	010000	Synchronous I/O (wait for write completion)

3.2.3 open() Examples

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno;

int main() {
    int fd;

    // Example 1: Open existing file for reading
    fd = open("existing.txt", O_RDONLY);
    if (fd == -1) {
        printf("Error %d: %s\n", errno, strerror(errno));
    } else {
        printf("Opened for reading, fd = %d\n", fd);
        close(fd);
    }
```

```c

    // Example 2: Create new file (or open if exists) for writing
    fd = open("data.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open failed");
    } else {
        printf("Created/opened for writing, fd = %d\n", fd);
        write(fd, "Hello\n", 6);
        close(fd);
    }

    // Example 3: Open for append (adds to end of file)
    fd = open("log.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
    if (fd != -1) {
        printf("Opened for append, fd = %d\n", fd);
        write(fd, "Log entry\n", 10);
        close(fd);
    }

    // Example 4: Exclusive creation (fails if file exists)
    fd = open("unique.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);
    if (fd == -1) {
        if (errno == EEXIST) {
            printf("File already exists (exclusive mode)\n");
        }
    } else {
        printf("Exclusively created, fd = %d\n", fd);
        close(fd);
    }

    return 0;
}
```

Listing 4: Various open() Usage Patterns

3.2.4 Path Types

Absolute vs Relative Paths

- **Absolute Path:** Full path from root directory
  - `open("/home/user/file.txt", ...)`
  - Always works from any directory
- **Relative Path:** Path relative to current working directory
  - `open("file.txt", ...)` (current directory)
  - `open("../parent/file.txt", ...)` (parent directory)
  - Depends on where program is executed

3.3 3. Close System Call

close() - Close a File Descriptor

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

3.3.1 Function Details

- **Header:** `<unistd.h>`
- **Purpose:** Close file descriptor and release resources
- **Important:** Always close files when done to prevent resource leaks
- **Safe:** `close()` on already closed FD returns -1 with `EBADF`
- **Special:** Closing FD 0, 1, or 2 affects standard streams

3.3.2 close() Examples

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

int main() {
    int fd1, fd2;

    // Open two files
    fd1 = open("file1.txt", O_RDONLY);
    if (fd1 == -1) {
        perror("open file1");
        return 1;
    }
    printf("Opened file1, fd = %d\n", fd1);

    fd2 = open("file2.txt", O_WRONLY | O_CREAT, 0644);
    if (fd2 == -1) {
        perror("open file2");
        close(fd1);  // Close first file before returning
        return 1;
    }
    printf("Opened file2, fd = %d\n", fd2);

    // Use the files...
    char buffer[100];
    int n = read(fd1, buffer, sizeof(buffer)-1);
    if (n > 0) {
        write(fd2, buffer, n);
    }

    // Close files in reverse order (good practice)
    if (close(fd2) == -1) {
        perror("close file2");
    } else {
        printf("Closed file2 (fd=%d)\n", fd2);
    }

    if (close(fd1) == -1) {
```

```
        perror("close file1");
    } else {
        printf("Closed file1 (fd=%d)\n", fd1);
    }

    // Demonstrate FD reuse
    int fd3 = open("file3.txt", O_WRONLY | O_CREAT, 0644);
    printf("Opened file3, fd = %d (reused from closed fds)\n", fd3);
    close(fd3);

    return 0;
}
```

Listing 5: Proper close() Usage

3.3.3 What Happens on close()?

1. Kernel decrements reference count in Global File Table
2. If reference count reaches 0:
   - Flushes any pending writes (if buffered)
   - Updates file metadata (size, modification time)
   - Removes entry from Global File Table
   - For disk files, data is now safely stored
3. Marks FD as available in process FD table
4. Returns 0 on success, -1 on error

3.3.4 Important close() Behavior

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

int main() {
    // Open same file twice - gets different FDs
    int fd1 = open("test.txt", O_RDONLY);
    int fd2 = open("test.txt", O_RDONLY);

    printf("fd1 = %d, fd2 = %d\n", fd1, fd2);

    // Each has independent file position
    char c;
    read(fd1, &c, 1);
    printf("fd1 read: %c\n", c);

    read(fd2, &c, 1);
    printf("fd2 read: %c (starts from beginning)\n", c);

    // Close fd1 doesn't affect fd2
    close(fd1);
    printf("Closed fd1\n");

    // fd2 still works
    read(fd2, &c, 1);
```

```c
    printf("fd2 still works, read: %c\n", c);

    close(fd2);

    // Try to close invalid FD
    if (close(999) == -1) {
        perror("close(999)");
    }

    // Close stdin (FD 0) - be careful!
    // close(STDIN_FILENO);
    // Now read() from stdin will fail!

    return 0;
}
```

Listing 6: Understanding close() Behavior

3.4 4. Read System Call

read() - Read from a File Descriptor

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

3.4.1 Function Details

- **Header:** `<unistd.h>`
- **Purpose:** Read data from file descriptor into buffer
- **Returns:** Number of bytes read, 0 on EOF, -1 on error
- **Blocking:** By default, blocks if no data available
- **Position:** Updates file position (except for pipes, sockets)

3.4.2 read() Examples

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main() {
    int fd;
    char buffer[1024];
    ssize_t bytes_read;

    // Open file for reading
    fd = open("input.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    printf("Reading file (fd=%d)...\n", fd);

    // Read file in chunks
    int total_bytes = 0;
    while ((bytes_read = read(fd, buffer, sizeof(buffer))) > 0) {
        total_bytes += bytes_read;
        printf("Read %zd bytes (total: %d)\n", bytes_read, total_bytes);

        // Process the buffer (e.g., write to stdout)
        write(STDOUT_FILENO, buffer, bytes_read);
    }

    // Check why loop ended
    if (bytes_read == 0) {
        printf("Reached end of file (EOF)\n");
    } else if (bytes_read == -1) {
        perror("read error");
    }

    close(fd);

```

```
    // Example 2: Reading from stdin
    printf("\nEnter something (press Ctrl+D for EOF):\n");
    bytes_read = read(STDIN_FILENO, buffer, sizeof(buffer)-1);

    if (bytes_read > 0) {
        buffer[bytes_read] = '\0';
        printf("You entered: %s", buffer);
    } else if (bytes_read == 0) {
        printf("EOF on stdin\n");
    }

    return 0;
}
```

Listing 7: Using read() System Call

3.4.3 Important read() Behaviors

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

int main() {
    // Create a test file
    int fd = open("test.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    write(fd, "ABCDEFGHIJ", 10);
    close(fd);

    // Open two file descriptors to same file
    int fd1 = open("test.txt", O_RDONLY);
    int fd2 = open("test.txt", O_RDONLY);

    char c;

    // Each FD has independent file position
    read(fd1, &c, 1);
    printf("fd1 read: %c\n", c);  // A

    read(fd2, &c, 1);
    printf("fd2 read: %c\n", c);  // A (independent position)

    read(fd1, &c, 1);
    printf("fd1 read: %c\n", c);  // B

    // Using lseek to change position
    lseek(fd1, 4, SEEK_SET);  // Move to position 4 (0-based)
    read(fd1, &c, 1);
    printf("fd1 after lseek: %c\n", c);  // E

    // fd2 position unchanged
    read(fd2, &c, 1);
    printf("fd2 read: %c\n", c);  // B

    close(fd1);
    close(fd2);

    // Example: Partial read
    fd = open("test.txt", O_RDONLY);
    char buf[5];
```

```c
    ssize_t n;

    // Request 5 bytes, might get less
    n = read(fd, buf, 3);  // Only ask for 3
    printf("Requested 3, got %zd bytes\n", n);

    // Request more than available
    n = read(fd, buf, 20);  // File has 7 bytes left
    printf("Requested 20, got %zd bytes (until EOF)\n", n);

    // At EOF
    n = read(fd, buf, 5);
    printf("At EOF: requested 5, got %zd bytes\n", n);  // 0

    close(fd);

    return 0;
}
```

Listing 8: Understanding read() Behavior

3.4.4 read() Return Value Summary

Return Value	Meaning
`n > 0`	Successfully read n bytes
`0`	End of File (EOF) reached
`-1`	Error occurred (check errno)
`0 < n < count`	Partial read (common with pipes, sockets)

3.4.5 Common read() Errors

- `EBADF`: fd is not a valid file descriptor
- `EINTR`: Interrupted by signal before reading any data
- `EINVAL`: fd is attached to object unsuitable for reading
- `EIO`: I/O error (low-level hardware error)
- `EISDIR`: fd refers to a directory

3.5 5. Write System Call

write() - Write to a File Descriptor

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

3.5.1 Function Details

- **Header:** <unistd.h>
- **Purpose:** Write data from buffer to file descriptor
- **Returns:** Number of bytes written, -1 on error
- **Blocking:** May block if device/buffer is full
- **Atomic:** Write up to PIPE_BUF bytes (usually 4096) is atomic

3.5.2 write() Examples

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    int fd;
    ssize_t bytes_written;
    const char *text = "Hello, System Calls!\n";

    // Example 1: Write to a file
    fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    bytes_written = write(fd, text, strlen(text));
    printf("Wrote %zd bytes to file\n", bytes_written);

    // Write more data
    const char *more_text = "Second line of text.\n";
    bytes_written = write(fd, more_text, strlen(more_text));
    printf("Wrote %zd more bytes\n", bytes_written);

    close(fd);

    // Example 2: Append to file
    fd = open("output.txt", O_WRONLY | O_APPEND);
    const char *append_text = "Appended line.\n";
    write(fd, append_text, strlen(append_text));
    close(fd);

    // Example 3: Write to stdout and stderr
    write(STDOUT_FILENO, "This goes to stdout\n", 20);
    write(STDERR_FILENO, "This goes to stderr\n", 20);

    // Example 4: Copy file using read/write
```

```
    int src_fd = open("input.txt", O_RDONLY);
    int dst_fd = open("copy.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    if (src_fd != -1 && dst_fd != -1) {
        char buffer[4096];
        ssize_t n;

        printf("Copying file...\n");
        while ((n = read(src_fd, buffer, sizeof(buffer))) > 0) {
            ssize_t written = write(dst_fd, buffer, n);
            if (written != n) {
                perror("write error during copy");
                break;
            }
        }

        if (n == -1) {
            perror("read error during copy");
        } else {
            printf("File copied successfully\n");
        }

        close(src_fd);
        close(dst_fd);
    }

    return 0;
}
```

Listing 9: Using write() System Call

3.5.3 Important write() Behaviors

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

int main() {
    // Example 1: Partial writes
    int fd = open("partial.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    char large_buffer[10000];
    memset(large_buffer, 'A', sizeof(large_buffer));

    // write() may write less than requested
    ssize_t written = write(fd, large_buffer, sizeof(large_buffer));
    printf("Requested %zu, wrote %zd bytes\n", sizeof(large_buffer),
    written);
    close(fd);

    // Example 2: O_APPEND behavior
    fd = open("append.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    write(fd, "Line 1\n", 7);
    close(fd);

    // Open with O_APPEND - all writes go to end
    int fd1 = open("append.txt", O_WRONLY | O_APPEND);
    int fd2 = open("append.txt", O_WRONLY);  // Without O_APPEND

```

```c
    // Both write at current position
    write(fd1, "fd1: Line 2\n", 12);  // Goes to end (due to O_APPEND)

    lseek(fd2, 0, SEEK_SET);  // Move to beginning
    write(fd2, "OVERWRITE", 9);  // Overwrites beginning!

    close(fd1);
    close(fd2);

    // Example 3: Writing to pipes
    int pipefd[2];
    pipe(pipefd);

    char msg[] = "Hello through pipe!";
    written = write(pipefd[1], msg, sizeof(msg));
    printf("Wrote %zd bytes to pipe\n", written);

    // Pipe has limited buffer (usually 64KB)
    // Write may block if pipe is full

    close(pipefd[0]);
    close(pipefd[1]);

    // Example 4: Error cases
    fd = open("/dev/full", O_WRONLY);  // Device that always returns ENOSPC
    if (fd != -1) {
        written = write(fd, "test", 4);
        if (written == -1) {
            perror("write to /dev/full");
        }
        close(fd);
    }

    // Write to closed FD
    fd = 999;  // Invalid FD
    if (write(fd, "test", 4) == -1) {
        perror("write to invalid fd");
    }

    return 0;
}
```

Listing 10: Understanding write() Behavior

3.5.4 write() Return Value Summary

Return Value	Meaning
`n == count`	All bytes successfully written
`0 < n < count`	Partial write (may retry remaining)
`-1`	Error occurred (check errno)
`0`	Rare (usually means count was 0)

3.5.5 Common write() Errors

- `EBADF`: fd is not a valid file descriptor open for writing
- `EFBIG`: File size limit exceeded

- `EINTR`: Interrupted by signal before writing any data
- `EIO`: Low-level I/O error
- `ENOSPC`: No space left on device
- `EPIPE`: Broken pipe (reader closed)

4 Practical Examples and Use Cases

4.1 Example 1: Simple File Copy

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

#define BUFFER_SIZE 4096

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
        return 1;
    }

    const char *source = argv[1];
    const char *dest = argv[2];

    // Open source file
    int src_fd = open(source, O_RDONLY);
    if (src_fd == -1) {
        perror("Error opening source file");
        return 1;
    }

    // Open destination file
    int dest_fd = open(dest, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dest_fd == -1) {
        perror("Error opening destination file");
        close(src_fd);
        return 1;
    }

    // Copy data
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read, bytes_written;
    int total_bytes = 0;

    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        bytes_written = write(dest_fd, buffer, bytes_read);

        if (bytes_written != bytes_read) {
            perror("Error writing to destination");
            close(src_fd);
            close(dest_fd);
            return 1;
        }

        total_bytes += bytes_written;
    }

    if (bytes_read == -1) {
        perror("Error reading from source");
    } else {
        printf("Copied %d bytes from '%s' to '%s'\n",
               total_bytes, source, dest);
    }

```

```
    // Cleanup
    close(src_fd);
    close(dest_fd);

    return 0;
}
```

Listing 11: File Copy Utility Using System Calls

4.2 Example 2: Simple Cat Command

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

#define BUFFER_SIZE 1024

void cat_file(const char *filename) {
    int fd;

    if (filename[0] == '-' && filename[1] == '\0') {
        // Read from stdin
        fd = STDIN_FILENO;
    } else {
        fd = open(filename, O_RDONLY);
        if (fd == -1) {
            perror(filename);
            return;
        }
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;

    while ((bytes_read = read(fd, buffer, BUFFER_SIZE)) > 0) {
        write(STDOUT_FILENO, buffer, bytes_read);
    }

    if (bytes_read == -1) {
        perror("read error");
    }

    if (fd != STDIN_FILENO) {
        close(fd);
    }
}

int main(int argc, char *argv[]) {
    if (argc == 1) {
        // No arguments, read from stdin
        cat_file("-");
    } else {
        for (int i = 1; i < argc; i++) {
            cat_file(argv[i]);
        }
    }

    return 0;
}
```

Listing 12: Simple cat command implementation

4.3 Example 3: Simple Echo Command

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int output_fd = STDOUT_FILENO;
    int append_mode = 0;

    // Check for output redirection
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], ">") == 0 && i + 1 < argc) {
            // Truncate and write
            output_fd = open(argv[i + 1],
                             O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (output_fd == -1) {
                perror("open for output");
                return 1;
            }
            // Remove redirection tokens from args
            argc = i;
            break;
        } else if (strcmp(argv[i], ">>") == 0 && i + 1 < argc) {
            // Append mode
            output_fd = open(argv[i + 1],
                             O_WRONLY | O_CREAT | O_APPEND, 0644);
            if (output_fd == -1) {
                perror("open for append");
                return 1;
            }
            argc = i;
            append_mode = 1;
            break;
        }
    }

    // Write all arguments
    for (int i = 1; i < argc; i++) {
        write(output_fd, argv[i], strlen(argv[i]));
        if (i < argc - 1) {
            write(output_fd, " ", 1);
        }
    }

    write(output_fd, "\n", 1);

    if (output_fd != STDOUT_FILENO) {
        close(output_fd);
        if (append_mode) {
            printf("Appended to file\n");
        } else {
            printf("Output written to file\n");
        }
```

```
    }

    return 0;
}
```

Listing 13: Echo command with redirection support

4.4 Example 4: Simple Head Command

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

#define DEFAULT_LINES 10

void head_file(const char *filename, int line_count) {
    int fd;

    if (filename[0] == '-' && filename[1] == '\0') {
        fd = STDIN_FILENO;
    } else {
        fd = open(filename, O_RDONLY);
        if (fd == -1) {
            perror(filename);
            return;
        }
    }

    char buffer[1];
    int lines_printed = 0;
    ssize_t bytes_read;
    char last_char = '\n';  // Start as if we just saw a newline

    while (lines_printed < line_count &&
           (bytes_read = read(fd, buffer, 1)) > 0) {

        write(STDOUT_FILENO, buffer, 1);

        if (buffer[0] == '\n') {
            lines_printed++;
        }

        last_char = buffer[0];
    }

    if (last_char != '\n') {
        write(STDOUT_FILENO, "\n", 1);
    }

    if (fd != STDIN_FILENO) {
        close(fd);
    }
}

int main(int argc, char *argv[]) {
    int line_count = DEFAULT_LINES;
    int file_index = 1;

    // Parse command line arguments
```

```
    if (argc > 1 && argv[1][0] == '-') {
        line_count = atoi(&argv[1][1]);
        if (line_count <= 0) {
            line_count = DEFAULT_LINES;
        }
        file_index = 2;
    }

    if (file_index >= argc) {
        // No filename, read from stdin
        head_file("-", line_count);
    } else {
        for (int i = file_index; i < argc; i++) {
            if (argc - file_index > 1) {
                // Multiple files, print header
                printf("\n==> %s <==\n", argv[i]);
            }
            head_file(argv[i], line_count);
        }
    }

    return 0;
}
```

Listing 14: Head command showing first N lines

5 Advanced Topics and Best Practices

5.1 Error Handling with errno

Using errno for Error Diagnostics

The `errno` variable is set by system calls on error. Always check it after a failed call.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

int main() {
    int fd;

    // Example 1: Simple error check
    fd = open("nonexistent.txt", O_RDONLY);
    if (fd == -1) {
        printf("Error %d: %s\n", errno, strerror(errno));
    }

    // Example 2: perror() for automatic message
    fd = open("/root/protected.txt", O_RDONLY);
    if (fd == -1) {
        perror("open failed");
        // Prints: open failed: Permission denied
    }

    // Example 3: Checking specific errors
    fd = open("test.txt", O_RDONLY | O_CREAT | O_EXCL, 0644);
    if (fd == -1) {
        if (errno == EEXIST) {
            printf("File already exists (exclusive mode)\n");
        } else if (errno == EACCES) {
            printf("Permission denied\n");
        } else {
            perror("unexpected error");
        }
    }

    // Example 4: Save/Restore errno for library calls
    int saved_errno;
    fd = open("file.txt", O_RDONLY);
    if (fd == -1) {
        saved_errno = errno;
        // Call some library function that might change errno
        printf("Some message\n");
        // Restore errno for accurate error reporting
        errno = saved_errno;
        perror("open");
    }

    return 0;
}
```

Listing 15: Proper Error Handling

5.2 File Positioning with lseek

lseek() - Reposition Read/Write Offset

```
off_t lseek(int fd, off_t offset, int whence);
```

5.2.1 lseek() Examples

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

int main() {
    int fd = open("test.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    // Write some data
    write(fd, "0123456789ABCDEF", 16);

    // Move to different positions
    off_t pos;
    char buffer[2];

    // SEEK_SET: offset from beginning
    lseek(fd, 5, SEEK_SET);
    read(fd, buffer, 1);
    buffer[1] = '\0';
    printf("Byte at position 5: %s\n", buffer);  // 5

    // SEEK_CUR: offset from current position
    lseek(fd, 3, SEEK_CUR);  // Move 3 bytes forward
    read(fd, buffer, 1);
    printf("3 bytes after position 5: %s\n", buffer);  // 9

    // SEEK_END: offset from end
    lseek(fd, -4, SEEK_END);  // 4 bytes from end
    read(fd, buffer, 1);
    printf("4 bytes from end: %s\n", buffer);  // C

    // Get current position
    pos = lseek(fd, 0, SEEK_CUR);
    printf("Current position: %ld\n", pos);

    // Get file size
    off_t size = lseek(fd, 0, SEEK_END);
    printf("File size: %ld bytes\n", size);

    // Rewind to beginning
    lseek(fd, 0, SEEK_SET);

```

```c
    // Write at specific position (overwrites)
    lseek(fd, 3, SEEK_SET);
    write(fd, "XXX", 3);

    // Read modified file
    lseek(fd, 0, SEEK_SET);
    char all[20];
    int n = read(fd, all, 16);
    all[n] = '\0';
    printf("Modified content: %s\n", all);  // 012XXX6789ABCDEF

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

Listing 16: Using lseek() for File Positioning

5.3 File Status with fstat

fstat() - Get File Status

```
int fstat(int fd, struct stat *statbuf);
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <time.h>

int main() {
    int fd = open("test.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    struct stat file_stat;

    if (fstat(fd, &file_stat) == -1) {
        perror("fstat");
        close(fd);
        return 1;
    }

    printf("File Information:\n");
    printf("  File size: %ld bytes\n", file_stat.st_size);
    printf("  Blocks allocated: %ld\n", file_stat.st_blocks);
    printf("  Block size: %ld bytes\n", file_stat.st_blksize);
    printf("  Inode number: %ld\n", file_stat.st_ino);
    printf("  Links: %ld\n", file_stat.st_nlink);
    printf("  Owner UID: %d\n", file_stat.st_uid);
    printf("  Group GID: %d\n", file_stat.st_gid);

    printf("  Permissions: ");
    printf((S_ISDIR(file_stat.st_mode)) ? "d" : "-");
    printf((file_stat.st_mode & S_IRUSR) ? "r" : "-");
    printf((file_stat.st_mode & S_IWUSR) ? "w" : "-");
    printf((file_stat.st_mode & S_IXUSR) ? "x" : "-");
```

```
    printf((file_stat.st_mode & S_IRGRP) ? "r" : "-");
    printf((file_stat.st_mode & S_IWGRP) ? "w" : "-");
    printf((file_stat.st_mode & S_IXGRP) ? "x" : "-");
    printf((file_stat.st_mode & S_IROTH) ? "r" : "-");
    printf((file_stat.st_mode & S_IWOTH) ? "w" : "-");
    printf((file_stat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n");

    printf("  Last access: %s", ctime(&file_stat.st_atime));
    printf("  Last modification: %s", ctime(&file_stat.st_mtime));
    printf("  Last status change: %s", ctime(&file_stat.st_ctime));

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

Listing 17: Getting File Information with fstat()

5.4 Duplicating File Descriptors with dup/dup2

dup()/dup2() - Duplicate File Descriptors

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

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

int main() {
    int fd, fd2, fd3;

    // Create a file
    fd = open("test.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    write(fd, "Original FD\n", 12);

    // dup() creates a copy with lowest available FD
    fd2 = dup(fd);
    printf("Original fd: %d, dup fd: %d\n", fd, fd2);

    // Both write to same file
    write(fd, "From original fd\n", 17);
    write(fd2, "From dup fd\n", 12);

    // They share file position
    lseek(fd, 0, SEEK_SET);
    char buffer[100];
    int n = read(fd, buffer, sizeof(buffer)-1);
    buffer[n] = '\0';
    printf("File content:\n%s", buffer);

    // dup2() duplicates to specific FD
    fd3 = dup2(fd, 10);  // Try to duplicate to FD 10
    printf("dup2 created fd: %d\n", fd3);

    // Redirect stdout to file
    int saved_stdout = dup(STDOUT_FILENO);  // Save original stdout
    dup2(fd, STDOUT_FILENO);  // Now stdout goes to file

    printf("This goes to the file, not screen!\n");

    // Restore stdout
    dup2(saved_stdout, STDOUT_FILENO);
    printf("Back to screen output\n");

    // Cleanup
    close(fd);
    close(fd2);
    close(fd3);
    close(saved_stdout);

    return 0;
}
```

Listing 18: Duplicating File Descriptors

5.5 Non-blocking I/O

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main() {
    // Set stdin to non-blocking mode
    int flags = fcntl(STDIN_FILENO, F_GETFL, 0);
    fcntl(STDIN_FILENO, F_SETFL, flags | O_NONBLOCK);

    printf("Type something (non-blocking read, Ctrl+D to exit)...\n");

    char buffer[100];
    int bytes_read;

    for (int i = 0; i < 10; i++) {
        bytes_read = read(STDIN_FILENO, buffer, sizeof(buffer)-1);

        if (bytes_read > 0) {
            buffer[bytes_read] = '\0';
            printf("You typed: %s", buffer);
            break;
        } else if (bytes_read == -1 && errno == EAGAIN) {
            // No data available yet
            printf(".");
            fflush(stdout);
            sleep(1);
        } else if (bytes_read == 0) {
            printf("\nEOF reached\n");
            break;
        }
    }

    if (bytes_read <= 0 && errno != EAGAIN) {
        printf("\nNo input received\n");
    }

    // Restore blocking mode
    fcntl(STDIN_FILENO, F_SETFL, flags);

    return 0;
}
```

Listing 19: Non-blocking I/O Example

6 Common Pitfalls and Debugging

6.1 Common System Call Errors

Error	Cause	Solution
Memory Leaks	Not closing file descriptors	Always close() when done
Race Conditions	File created between check and open	Use O_CREAT | O_EXCL
Permission Issues	Wrong file permissions	Check umask, use correct mode
Buffer Overflows	Not checking read() return value	Always check return values
Partial Reads/Writes	Assuming full buffer transferred	Handle partial operations
File Descriptor Limits	Too many open files	Close unused FDs, increase limits
Blocking Forever	Read/write on blocking FD	Use non-blocking or timeouts

6.2 Debugging Techniques

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

// Debug macro
#ifdef DEBUG
#define DBG_PRINT(fmt, ...) fprintf(stderr, "DEBUG: " fmt, ##__VA_ARGS__)
#else
#define DBG_PRINT(fmt, ...)
#endif

int safe_open(const char *path, int flags, mode_t mode) {
    DBG_PRINT("Opening %s with flags %d\n", path, flags);

    int fd = open(path, flags, mode);
    if (fd == -1) {
        DBG_PRINT("open failed: %s\n", strerror(errno));
    } else {
        DBG_PRINT("open succeeded, fd = %d\n", fd);
    }

    return fd;
}

ssize_t safe_read(int fd, void *buf, size_t count) {
    DBG_PRINT("Reading from fd %d, requesting %zu bytes\n", fd, count);

    ssize_t result = read(fd, buf, count);
    if (result == -1) {
        DBG_PRINT("read failed: %s\n", strerror(errno));
```

```c
    } else if (result == 0) {
        DBG_PRINT("read reached EOF\n");
    } else {
        DBG_PRINT("read returned %zd bytes\n", result);
    }

    return result;
}

int main() {
    // Enable debugging
    #define DEBUG 1

    int fd = safe_open("test.txt", O_RDONLY, 0);
    if (fd == -1) {
        return 1;
    }

    char buffer[100];
    ssize_t n = safe_read(fd, buffer, sizeof(buffer));

    if (n > 0) {
        write(STDOUT_FILENO, buffer, n);
    }

    close(fd);

    // Check file descriptor limits
    #ifdef DEBUG
    printf("\nFile descriptor information:\n");
    printf("STDIN_FILENO = %d\n", STDIN_FILENO);
    printf("STDOUT_FILENO = %d\n", STDOUT_FILENO);
    printf("STDERR_FILENO = %d\n", STDERR_FILENO);

    // Try to find maximum FDs
    for (int i = 3; i < 100; i++) {
        if (fcntl(i, F_GETFD) == -1 && errno == EBADF) {
            printf("First unused FD: %d\n", i);
            break;
        }
    }
    #endif

    return 0;
}
```

Listing 20: Debugging System Call Programs

6.3 Best Practices Summary

System Calls Best Practices

1. **Always check return values** - Never assume success
2. **Use proper error handling** - Check errno, use perror()
3. **Close file descriptors** - Prevent resource leaks
4. **Handle partial reads/writes** - Loop until all data transferred
5. **Use appropriate flags** - O_CREAT, O_EXCL for atomic operations
6. **Check permissions** - Use correct mode values
7. **Be aware of blocking** - Consider non-blocking I/O for responsiveness
8. **Use constants** - STDIN_FILENO instead of 0, etc.
9. **Test edge cases** - Empty files, large files, permission issues
10. **Clean up on failure** - Close opened FDs before returning

6.4 Performance Considerations

- **Buffer size matters:** Too small = many system calls, too large = memory waste
- **System calls are expensive:** Minimize context switches
- **Use read/write efficiently:** Read/write in reasonable chunks (4KB-64KB)
- **Consider memory mapping:** `mmap()` for large files
- **Batch operations:** Combine when possible
- **Avoid unnecessary seeks:** Sequential access is faster

Conclusion

I/O system calls are the foundation of file operations in Linux/Unix systems. Understanding them provides deep insight into how operating systems work and enables writing efficient, low-level code.

Key Takeaways

- **File descriptors** are integers that represent open files in a process
- **Standard streams** (0,1,2) are automatically opened for every process
- The **5 essential I/O system calls** are: create, open, close, read, write
- **Always check return values** and handle errors properly
- **System calls provide low-level control** but require careful management
- **Understanding these concepts** is crucial for system programming

Happy Systems Programming!

Master the fundamentals, control the system!

This document was compiled on January 15, 2026.
For updates and more resources, visit: `https://github.com/systems-programming`