File Management

Application Programming vs System Programming

Application Programming vs System Programming

Application Programming	System Programming
Using **Std. C Libraries**	Using **System Calls**
High level functions	Low level functions
Man page **Level 3**	Man page **Level 2**
Libraries maintains buffers to read or write data from system calls. So libraries also called **Buffered I/O**.	System calls doesn't have buffers.
Libraries doesn't create performance penalty.	System calls create performance penalty.
Example: hello.c	Example: hello.c

File System

File System

- The file system
  - manages files,
  - allocating file space,
  - administrating free space,
  - controlling access to files
  - and retrieving data for user.
- The internal representation of file is given an ***inode table***.

In Linux, Everything is a File

Symbol	Name
-	Regular File
d	Directory
b	Block Device
c	Character Device
l	Symbolic Link (shortcut)
s	Socket
P	pipe

File Descriptor

- File Descriptor is an unsigned integer number it can associate with corresponding file.
- In Linux, Every running process have three file descriptors.

File Descriptor	Associated File
0	STD INPUT (Keyboard)
1	STD OUTPUT (Monitor)
2	STD ERROR (Monitor)

Basic File Operation System calls

Basic File Operation System calls

open() -> open and possibly create a file

read() -> read from a file descriptor

write() -> write to file descriptor

close() -> close a file descriptor

open() system call

Name of the File mentioned in Absolute PATH or Relative PATH

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

RETURN

Success: File Descriptor

ERROR: Negative Number

FLAGS

O_RDONLY, O_WRONLY, or O_RDWR

read() system call

Read from
File descriptor

Read/write
Buffer

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

No. of bytes to read

RETURN

Success: Positive Integer Number to represent No. of bytes read.
EOF: Return ZERO
ERROR: Negative Number

write() system call

Write to
File descriptor

Read only
Buffer

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

No. of bytes to write

RETURN

Success: Positive integer Number to represent No. of bytes written.
EOF: Return ZERO
ERROR: Negative Number

close() system call

Close File descriptor

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

RETURN

Success: Return Zero

ERROR: Negative Number

System call ERROR Handling

System call ERROR Handling

- Most of the system calls RETURN **“ZERO”** indicates SUCCESS.
- Most of the system calls RETURN **“Negative Number”** indicated ERROR.
- Few system calls RETURN **“Positive Number”** indicates SUCCESS.

Programmer RETURN proper ERROR using perror() library.

Special File Operation System calls

Special File Operations system calls

- lseek() -> reposition read/write file offset
- stat(), fstat(), lstat() -> get file status

lseek() system call

Open
File descriptor

SEEK_SET: offset bytes.
SEEK_CUR: current location + offset bytes.
SEEK_END: size of the file + offset bytes.

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

RETURN

Success: offset location as measured in bytes from the beginning of the file.
ERROR: Negative Number

Offset of the open file descriptor

stat(), fstat(), lstat() system calls

```
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

```
struct stat {
        dev_t     st_dev;      /* ID of device containing file */
        ino_t     st_ino;      /* Inode number */
        mode_t    st_mode;     /* File type and mode */
        nlink_t   st_nlink;    /* Number of hard links */
        uid_t     st_uid;      /* User ID of owner */
        gid_t     st_gid;      /* Group ID of owner */
        dev_t     st_rdev;     /* Device ID (if special file) */
        off_t     st_size;     /* Total size, in bytes */
        blksize_t st_blksize;  /* Block size for filesystem I/O */
        blkcnt_t  st_blocks;   /* Number of 512B blocks allocated */

        struct timespec st_atim;  /* Time of last access */
        struct timespec st_mtim;  /* Time of last modification */
        struct timespec st_ctim;  /* Time of last status change */
};
```

File System Layout

OS vs File System

DOS: FAT32 (**F**ile **A**llocation **T**able)

Windows: FAT, NTFS (**N**ew **T**echnology **F**ile **S**ystem)

LINUX: ext2, ext3 and ext4 (Extended file system), JFS (Journaling file system), btrfs (b-tree File System)

File System Layout

Boot block	Super block	Inode list	Data Block

Boot Block: contain bootstrap code that is read into machine to boot, or initialize, the operating system.

Super Block: Describes the state of a file system – How large it is, how many files it can store, where to find free space on the file system.

Inode (index node)list: inode represents the type of the file.

Data Block: contain file data and administrative data.

File System - Kernel Data Structures

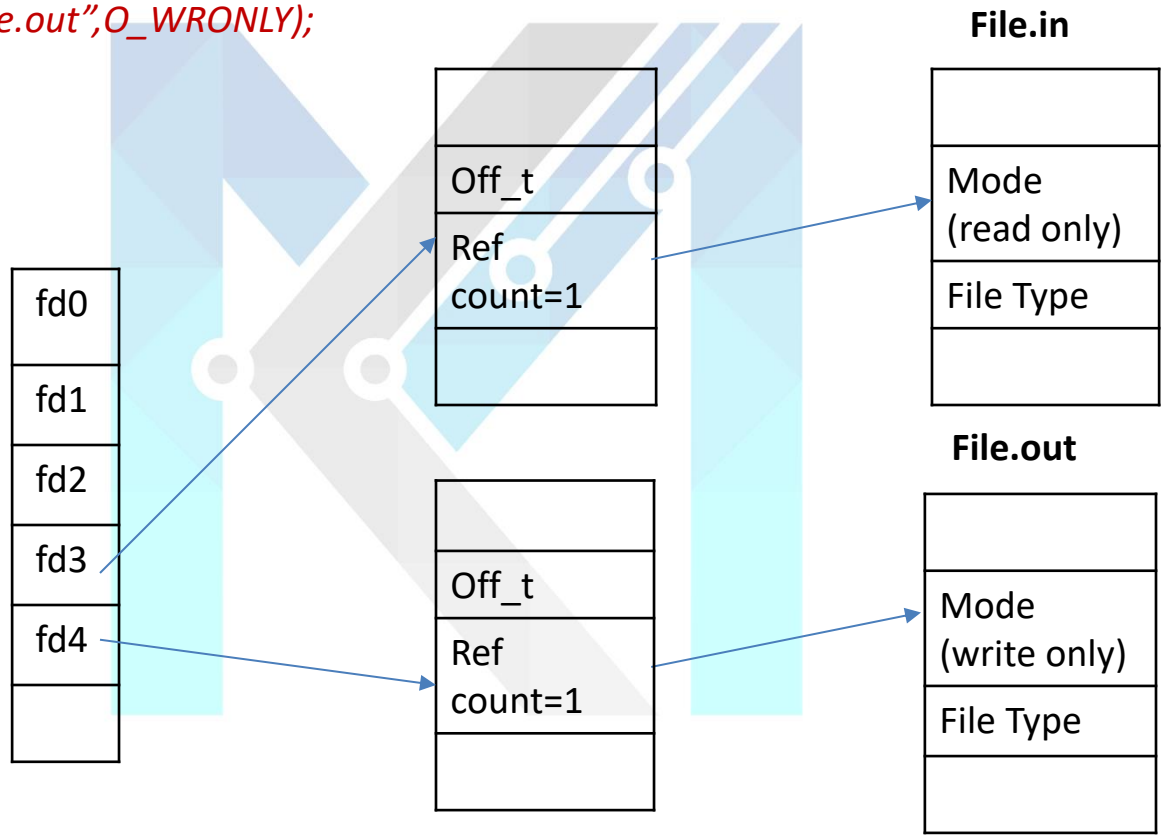
- ***User File Descriptor table:*** Each process has its own separate *descriptor table* whose entries are indexed by the process's open file descriptors. Each open descriptor entry points to an entry in the *file table.*
- ***File table:*** Each file table entry consists of (for our purposes) the current file position, a *reference count* of the number of descriptor entries that currently point to it, and a pointer to an entry in the *inode table*.
- ***Inode table:*** Each entry contains most of the information in the stat structure, including the st_mode and st_size members.
- ***File table*** is global kernel structure where as ***user file descriptor table*** is allocated per process.

File System - Kernel Data Structures

In this example, two descriptors reference distinct files. **There is no sharing.**

fd3=open("file.in",O_RDONLY);

fd4=open("file.out",O_WRONLY);



User File descriptor Table (one table per process)

File Table (shared by all process)

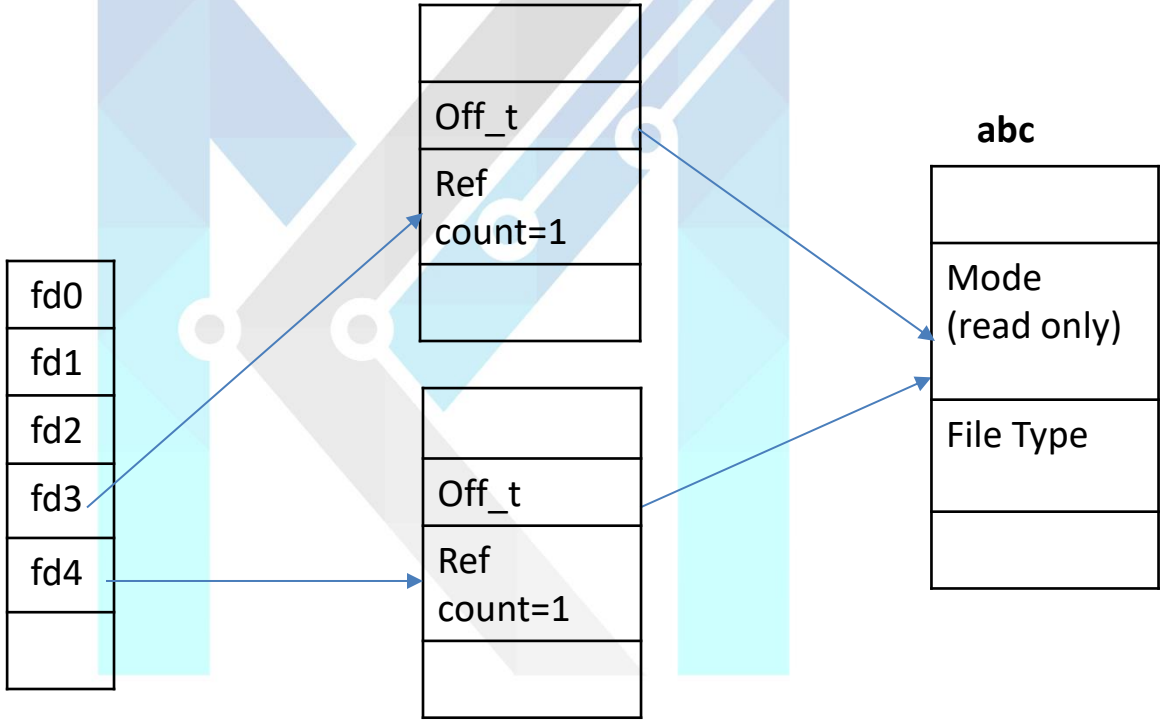
inode Table (shared by all process)

File System - Kernel Data Structures

This example shows two descriptors **sharing the same disk file** through two open file table entries

fd3=open("abc",O_RDONLY);

fd4=open("abc",O_RDONLY);

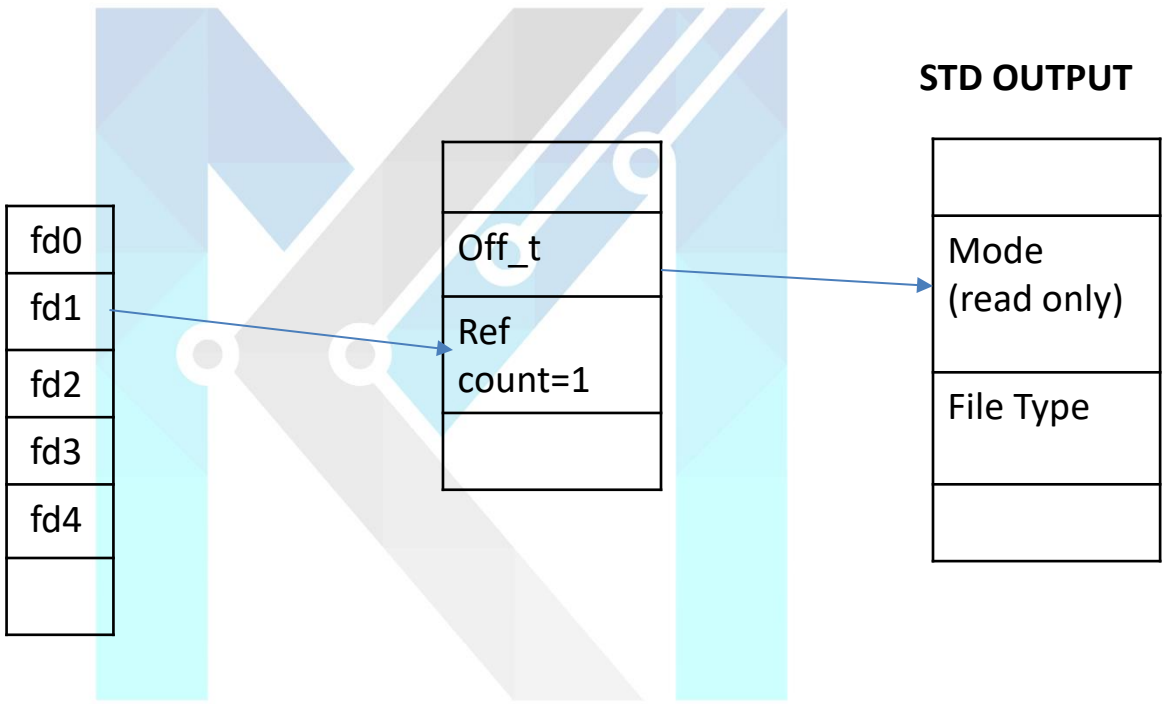


User File descriptor Table (one table per process)

File Table (shared by all process)

inode Table (shared by all process)

ls command o/p: STD OUT



User File descriptor Table
(one table per process)

File Table
(shared by all process)

inode Table
(shared by all process)

I/O Redirection – ls > ls.log ?

dup2(int oldfd, int newfd) Example:

Kernel data structures after redirecting standard output by calling **dup2(4,1).**