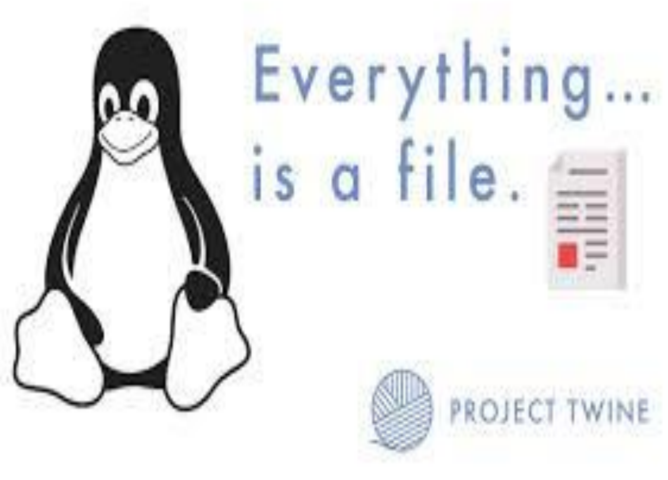
UNIT - II

UNIX FILES



By,
Dr. Kuldeep P. Sambrekar

Content

- Introduction: File Types,
- The UNIX and POSIX File System,
- The UNIX and POSIX File Attributes,
- Inodes in UNIX System V,
- Application Program Interface to Files,
- UNIX Kernel Support for Files,
- General File APIs,
- Directory File APIs ,
- Device File APIs,
- FIFO File APIs ,
- Symbolic Link File APIs,
- File and Record Locking.

- Files are the building blocks of any OS.

- When you execute a command in UNIX, the kernel fetches the executable file from the file system and also any files required for reading and writing.

- Files in UNIX and POSIX cover a wide range of file types.

- They also provide a common set of interfaces to files.

- This chapter covers
  - File Types
  - File attributes and their uses.
  - Kernel and process specific data structures used for file manipulation.

FILE TYPES

- Regular file
- Directory file
- FIFO file
- Character device file
- Block device file

Regular file

- It may be text or binary file
- They can be created, browsed and modified by various means such as text editors and compilers.
- They can be created with command like vi
- To remove regular files use rm command

Directory file

- Folder that contains other files and subdirectories
- Provides a means to organize files in hierarchical structure
- To create : mkdir command
- To remove : rmdir command
- To display : ls command

Block device file :

Physical device which transmits data a block at a time

EX: hard disk drive, floppy disk drive

- **Character device file :**

Physical device which transmits data in a character-based manner

EX: line printers, modems, consoles

- To create : mknod command

  ```
  mknod    /dev/cdsk  c  115  5
  ```

  /dev/cdsk : name of the device

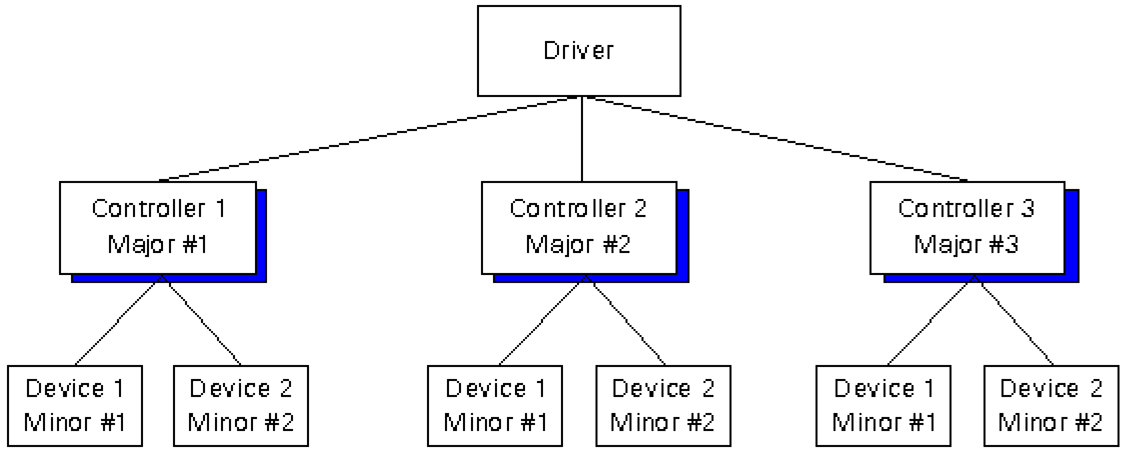
  c -- character device b -- block device

  115 — major device number

  5 — minor device number

Major device number : an index to the kernel’s file table that contains address of all device drivers

Minor device number : indicates the type of physical file and I/O buffering scheme used for data transfer



```
mknod  /dev/cdsk      c        115    5
```

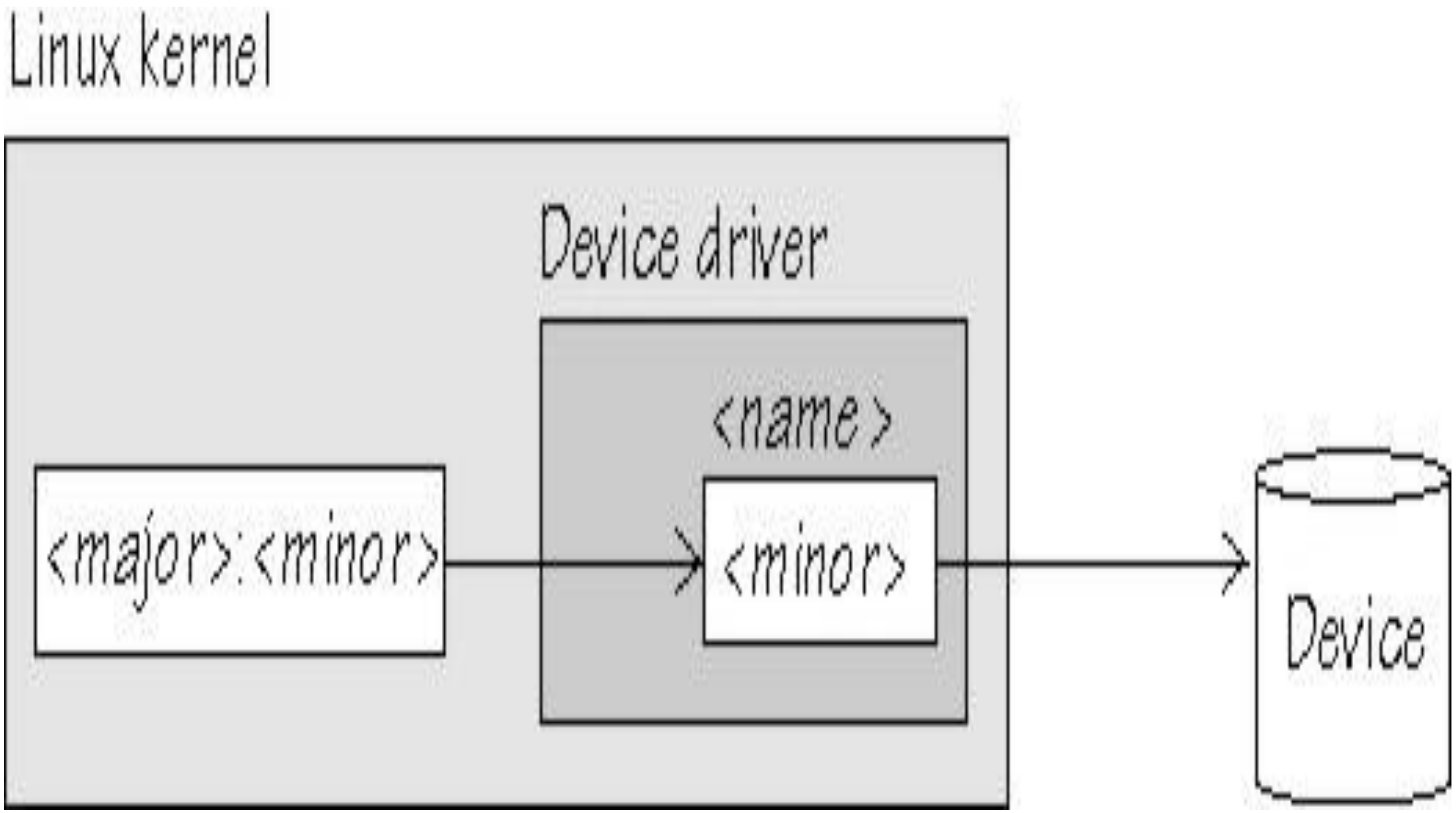
115 – major device number: An index to a kernel table that contains the address of device driver functions known to the system.

5 – minor device number: An integer value to be passed as an argument to a device driver function when it is called. It tells which actual device it is talking to.

c – Character device file

b – Block device file

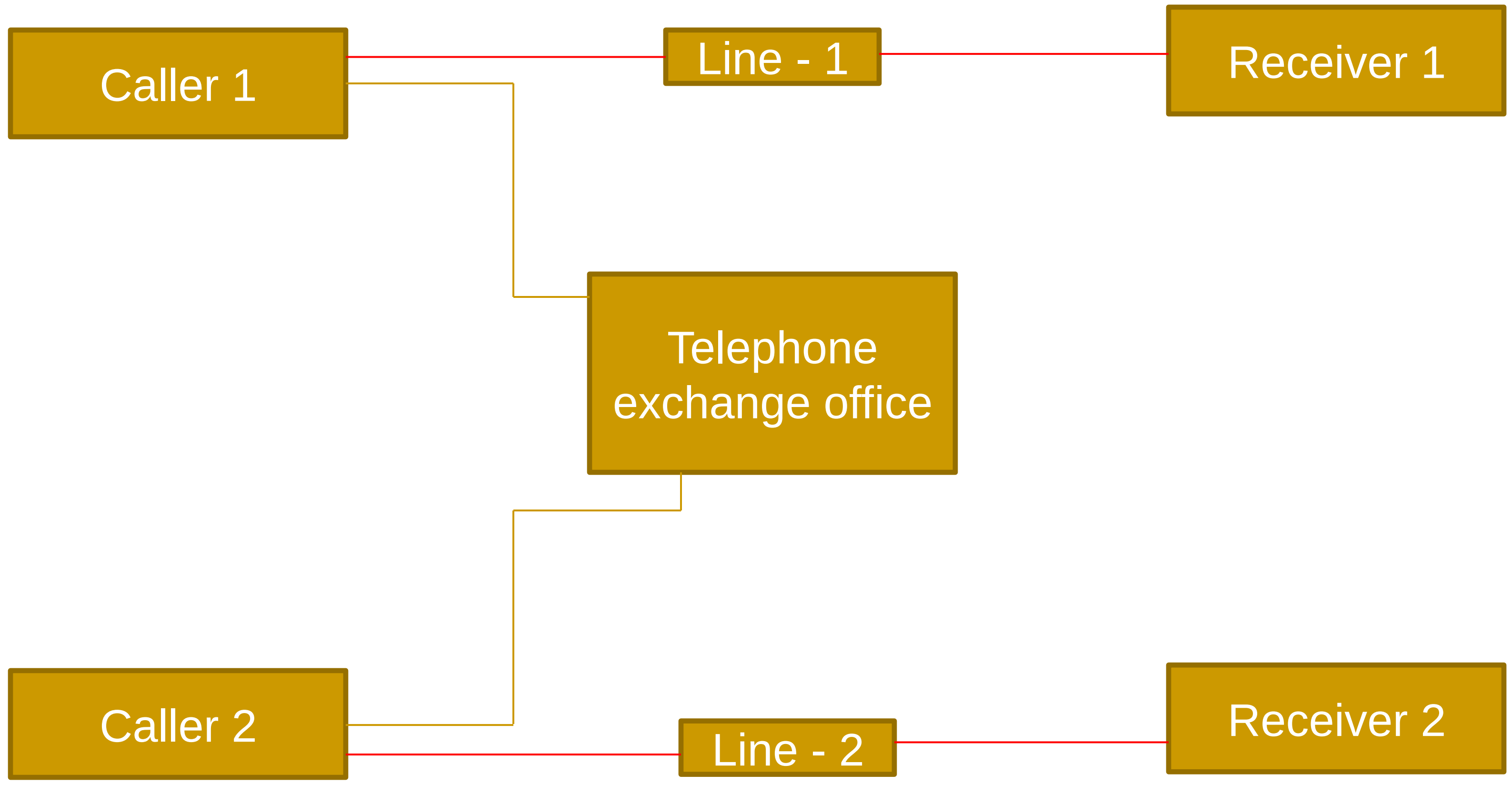
Example



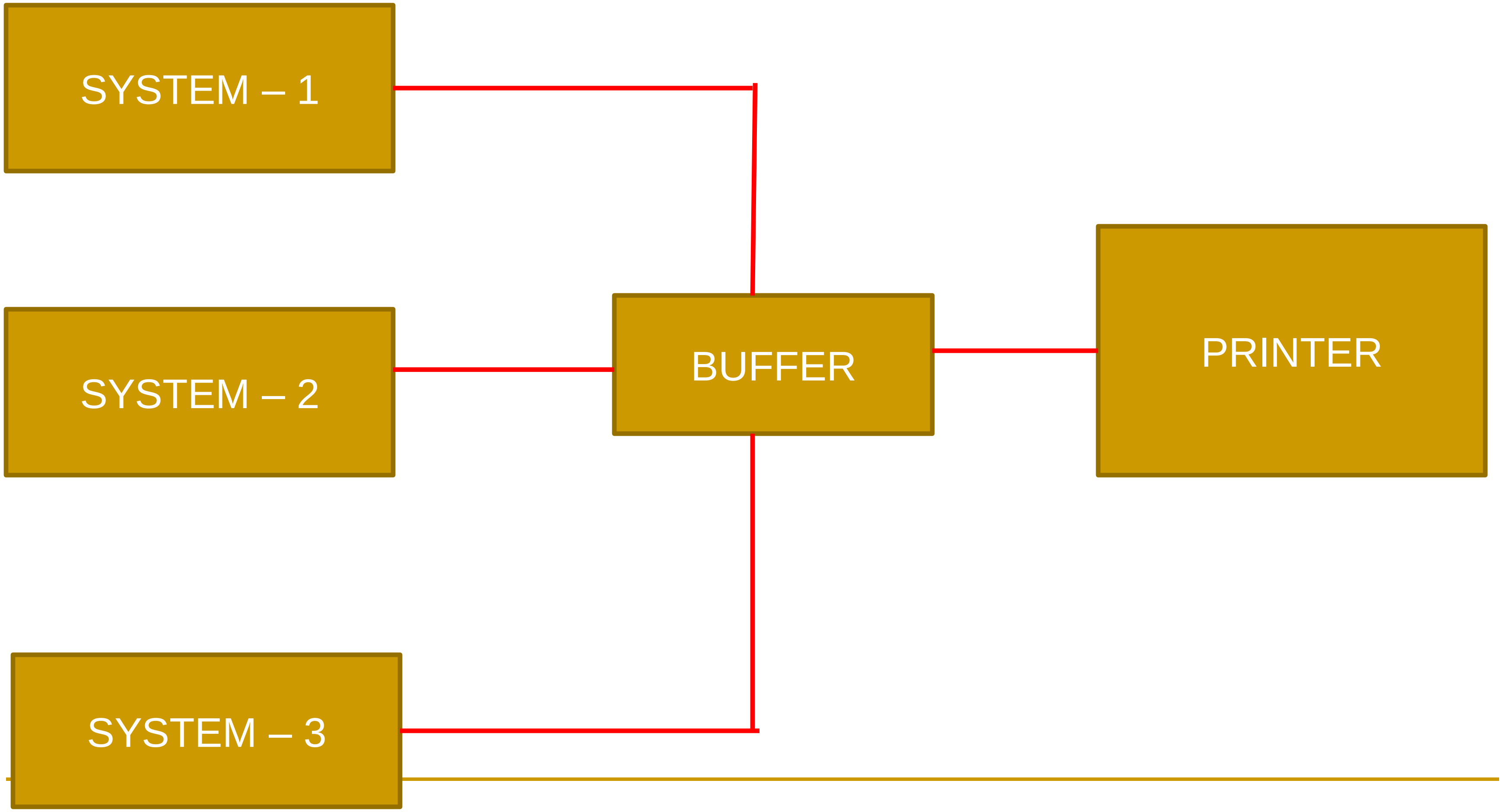
FIFO file

- Special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer
- Max size of buffer – PIPE_BUF
- The storage is temporary
- The file exists as long as there is one process in direct connection to the fifo for data access

Example


Caller 1
Line - 1
Receiver 1
Telephone exchange office
Caller 2
Line - 2
Receiver 2

Technical example



- **To create : mkfifo OR mkfifo**

  mkfifo /usr/prog/fifo_pipe

  mknod /usr/prog/fifo_pipe p

- **A fifo file can be removed like any other regular file**

Symbolic link file

- A symbolic link file contains a pathname which references another file in either the local or a remote file system
- To create : ln command with –s option

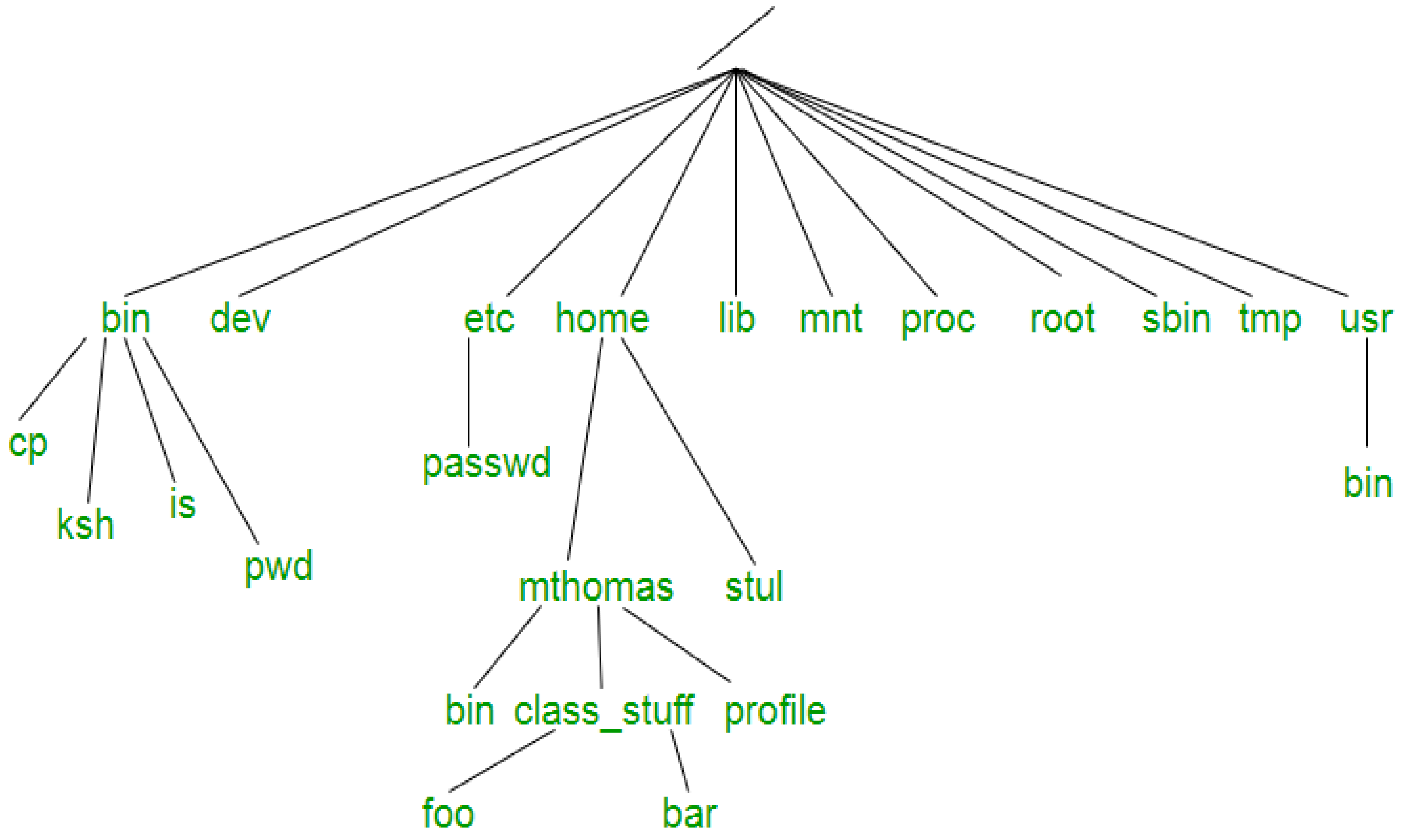
```
ln –s  /usr/abc/original  /usr/xyz/slink


cat  /usr/xyz/slink
```

/*will print contents of /usr/abc/original file*/

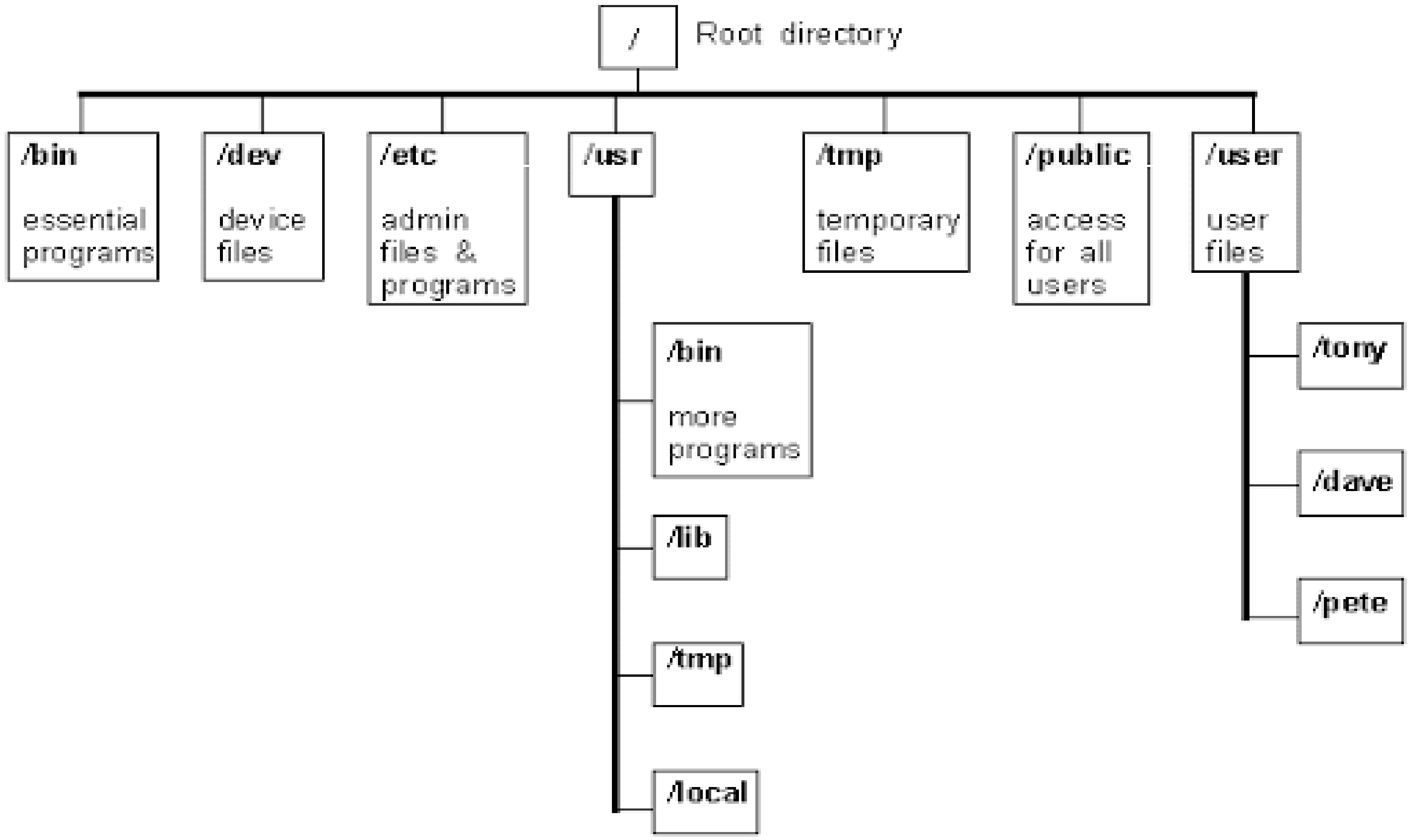
UNIX and POSIX file systems

- They have a tree-like hierarchical file system
- “/” – denotes the root
- “.” – current directory
- “..” – parent directory
- NAME_MAX – max number of characters in a file name
- PATH_MAX -- max number of characters in a path name

FILE SYSTEM


/
bin
dev
etc
home
lib
mnt
proc
root
sbin
tmp
usr
cp
ksh
is
pwd
passwd
bin
mthomas
stul
bin
class_stuff
profile
foo
bar

Example


/
Root directory
/bin
essential programs
/dev
device files
/etc
admin files & programs
/usr
/tmp
temporary files
/public
access for all users
/user
user files
/bin
more programs
/lib
/tmp
/local
/tony
/dave
/pete

Common UNIX files

/etc	:	Stores system administrative files & programs
/etc/passwd	:	Stores all user information
/etc/shadow	:	Stores user passwords
/etc/group	:	Stores all group information

/bin	: Stores all system programs
/dev	: Stores all character and block device files
/usr/include	: Stores standard header files
/usr/lib	: Stores standard libraries
tmp	: Stores all temporary files created by programs

UNIX and POSIX file attributes

- All the attributes of a file can be displayed by the command ls –l(long listing of files)
- File type : type of file
- Access permission : the file access permission for owner group and others
- Hard link count : number of hard links of a file

UID	: the file owner user ID
GID	: the file group ID
File size	: the file size in bytes
Last access time	: the time the file was last accessed
Last modify time	: the time the file was last modified

- **Last change time** : the time the file access permission UID ,GID or hard link count was last changed

Inode number	: the system inode number of the file
File system ID	: the file system ID where the file is stored

Attributes of a file that remain unchanged

- File type
- File inode number
- File system ID
- Major and minor device number

Other attributes are changed using UNIX commands or system calls

Inodes in UNIX

- ➢ In UNIX System V, a file system has an inode table which keeps track of all files.
- ➢Each inode table is an inode record contains all attributes of a file including inode no and physical disk address.
- ➢ Each inode no is unique to a file system only.
- ➢ An OS does not keep the name of a file in its inode record. The mapping from file names to inode nos is done through directory files.

Inodes in UNIX

- To access a file, for example /usr/joe
  - The UNIX kernel knows the /directory inode no – it is kept in process-U area.
  - It will scan the "/" directory file to find the inode no of usr file.
  - Once it gets the inode no, it checks if the calling process has permission to search usr directory.
  - It then looks for the joe file.
- Whenever a new file is created in directory, entry is made in inode table and directory file.
- Inode tables are kept in their file systems on disk, but the UNIX kernel maintains an in-memory inode table to contain a copy of the recently accessed inode records.

APPLICATION PROGRAM INTERFACE TO FILES

- **Files are identified by path names**
- **Files must be created before they can be used.**
- **Files must be opened before they can be accessed by application programs. open system call is for this purpose, which returns a file descriptor, which is a file handle used in other system calls to manipulate the open file**

Application Program Interface to Files

- Files are identified by path names
- Files must be created before they can be used.

File type	**UNIX Command**	**UNIX and POSIX.1 System call**
Regular Files	Vi, emacs..etc	Open, creat
Directory Files	mkdir	mkdir, mknod
FIFO Files	mkfifo	mkfifo, mknod
Device Files	mknod	Mknod
Symbolic Files	ln –s	Symlink

UNIX KERNEL SUPPORT FOR FILES

Example:

Consider:

1. If you want to open a file.

What will you do?

Just double click on the file and the file gets opened…..

But have you ever thought how this process works in the background????

UNIX KERNEL SUPPORT FOR FILES

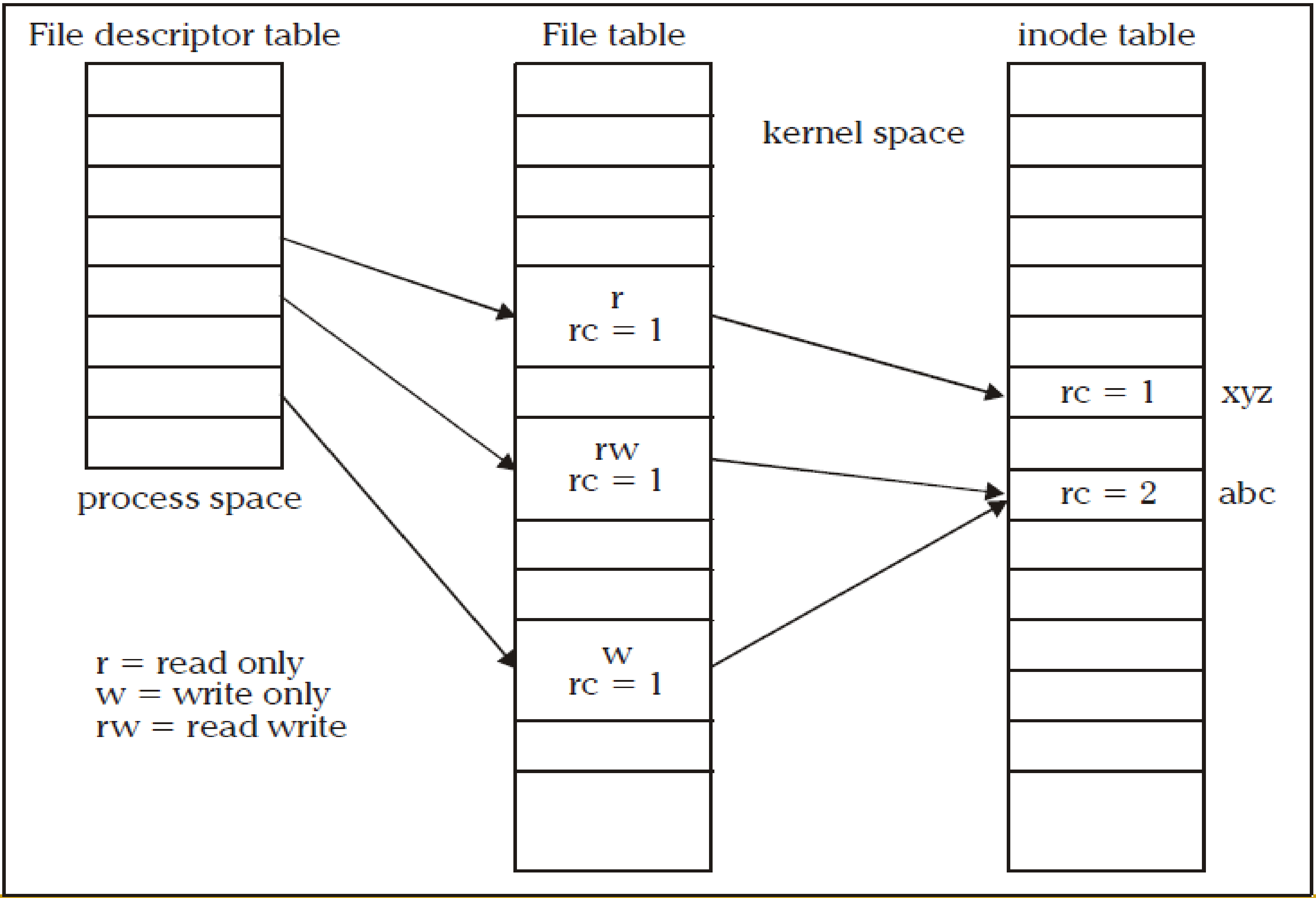
- Whenever the process calls the **open** function to open a file to read or write, the kernel will resolve the pathname to the file inode number.

File table – All open files

Inode Table – copy of file inodes most recently accessed

File descriptor Table – All files opened by the Process, OPEN_MAX

UNIX Kernel Support for Files



UNIX Kernel Support for Files

The steps involved are :

1. The kernel will search the process file descriptor table and look for the first unused entry.

2. If an entry is found, that entry will be designated to reference the file .The index of the entry will be returned to the process as the file descriptor of the opened file.

3. The kernel will scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

UNIX Kernel Support for Files

If an unused entry is found the following events will occur:

1.The process file descriptor table entry will be set to point to this file table entry.

2.The file table entry will be set to point to the inode table entry, where the inode record of the file is stored.

3.The file table entry will contain the current file pointer of the open file. This is an offset from the beginning of the file where the next read or write will occur.

4.The file table entry will contain an open mode that specifies that the file opened is for read only, write only or read and write etc. This should be specified in open function call.

UNIX Kernel Support for Files

If an unused entry is found the following events will occur:

5.The reference count (rc) in the file table entry is set to 1. Reference count is used to keep track of how many file descriptors from any process are referring the entry.

6.The reference count of the in-memory inode of the file is increased by 1. This count specifies how many file table entries are pointing to that inode.

If either (1) or (2) fails, the **open** system call returns -1 (failure/error)

Close function

1. The kernel sets the FD entry to be unused
2. Decrement the reference count in file table by 1. If still non-zero, go to 6.
3. The file table entry is marked as unused.
4. Decrement the reference count in file inode table by 1, if still non-zero, go to 6.
5. If hard-link count of inode is non zero, return to caller. Otherwise it marks the inode table entry as unused and deallocates all the physical disk storage of the file, as all the file path names have been removed by some process.
6. It returns to the process with a 0 (success) status.

Directory files

- Directory is a record oriented file.
- Record data type is struct dirent in UNIX V and POSIX.1, and struct direct in BSD UNIX

Directoryfunction	Purpose
opendir	Opens a directory file
readdir	Reads next record from file
closedir	closes a directory file
rewinddir	Sets file pointer to beginning of file

- Unix system also provides telldir and seekdir function

Directory File APIs

- **Why do we need directory files?**
- **To aid users in organizing their files into some structure based on the specific use of files**
- **They are also used by the operating system to convert file path names to their inode numbers**

- To create

int **mkdir (const char*** path_name , **mode_t** mode);

- **The mode argument specifies the access permission for the owner, group, and others to be assigned to the file.**

Difference between mkdir and mknod

- Directory created by mknod API does not contain the “.” and “..” links. These links are accessible only after the user explicitly creates them.
- Directory created by mkdir has the “.” and “..” links created in one atomic operation, and it is ready to be used.
- One can create directories via system API’s as well.

- A newly created directory has its user ID set to the effective user ID of the process that creates it.

- Directory group ID will be set to either the effective group ID of the calling process or the group ID of the parent directory that hosts the new directory.

FUNCTIONS

Opendir:

DIR* opendir (const char* path_name);

This opens the file for read-only

Readdir:

Dirent* readdir(DIR* dir_fdesc);

The dir_fdesc value is the DIR* return value from an opendir call.

Closedir :

int closedir (DIR* dir_fdesc);

It terminates the connection between the dir_fdesc handler and a directory file.

Rewinddir :

void rewinddir (DIR* dir_fdesc);

Used to reset the file pointer associated with a dir_fdesc.

- Rmdir API:

int rmdir (const char* path_name);

Used to remove the directory files. Users may also use the unlink API to remove directories provided they have super user privileges.

These API's require that the directories to be removed must be empty, in that they contain no files other than "." and ".." links.

Device file APIs

- Device files are used to interface physical devices (ex: console, modem) with application programs.
- Device files may be character-based or block-based
- The only differences between device files and regular files are the ways in which device files are created and the fact that lseek is not applicable for character device files.

To create:

```
int mknod(const char* path_name, mode_t
                            mode,int device_id);
```

- The mode argument specifies the access permission of the file
- The device_id contains the major and minor device numbers. The lowest byte of a device_id is set to minor device number and the next byte is set to the major device number.

MAJOR AND MINOR NUMBERS

- **When a process reads from or writes to a device file, the file's major device number is used to locate and invoke a device driver function that does the actual data transmission.**
- **The minor device number is an argument being passed to a device driver function when it is invoked. The minor device number specifies the parameters to be used for a particular device type.**

- A device file may be removed via the unlink API.
- If O_NOCTTY flag is set in the open call, the kernel will not set the character device file opened as the controlling terminal in the absence of one.
- The O_NONBLOCK flag specifies that the open call and any subsequent read or write calls to a device file should be non blocking to the process.

FIFO File APIs

- **These are special device files used for inter process communication.**
- **These are also known as named pipes.**
- **Data written to a FIFO file are stored in a fixed-size buffer and retrieved in a first-in-first-out order.**
- **To create:**

```
int mkfifo( const char* path_name, mode_t
                                    mode);
```

How is synchronization provided?

- When a process opens a FIFO file for read-only, the kernel will block the process until there is another process that opens the same file for write.
- If a process opens a FIFO for write, it will be blocked until another process opens the FIFO for read.

This provides a method for process synchronization

- **If a process writes to a FIFO that is full, the process will be blocked until another process has read data from the FIFO to make room for new data in the FIFO.**
- **If a process attempts to read data from a FIFO that is empty, the process will be blocked until another process writes data to the FIFO.**
- **If a process does not desire to be blocked by a FIFO file, it can specify the O_NONBLOCK flag in the *open* call to the FIFO file.**

- UNIX System V defines the O_NDELAY flag which is similar to the O_NONBLOCK flag. In case of O_NDELAY flag the read and write functions will return a zero value when they are supposed to block a process.
- If a process writes to a FIFO file that has no other process attached to it for read, the kernel will send a SIGPIPE signal to the process to notify it of the illegal operation.

- If Two processes are to communicate via a FIFO file, it is important that the writer process closes its file descriptor when it is done, so that the reader process can see the end-of-file condition.
- **Pipe API**

Another method to create FIFO files for

inter process communications

int pipe (int fds[2]);

- Uses of the fds argument are:
- fds[0] is a file descriptor to read data from the FIFO file.
- fds[1] is a file descriptor to write data to a FIFO file.

- The child processes inherit the FIFO file descriptors from the parent, and they can communicate among themselves and the parent via the FIFO file.

Symbolic Link File APIs

- These were developed to overcome several shortcomings of hard links:
  - Symbolic links can link from across file systems
  - Symbolic links can link directory files
  - Symbolic links always reference the latest version of the file to which they link
  - Hard links can be broken by removal of one or more links. But symbolic link will not be broken.

To create :

```
int symlink (const char* org_link, const
                                   char* sym_link);
int readlink (const char* sym_link, char* buf,
                                   int size);
int lstat (const char* sym_link, struct stat*
                                   statv);
```

FILE AND RECORD LOCKING

- UNIX systems allow multiple processes to read and write the same file concurrently.
- It is a means of data sharing among processes.
- Why we need to lock files? It is needed in some applications like database access where no other process can write or read a file while a process is accessing a data base.
- Unix and POSIX systems support a file-locking mechanism.
- File locking is applicable only to regular files.

Shared and exclusive locks

- A read lock is also called a shared lock and a write lock is called an exclusive lock.
- These locks can be imposed on the whole file or a portion of it.
- A write lock prevents other processes from setting any overlapping read or write locks on the locked regions of a file.
- The intention is to prevent other processes from both reading and writing the locked region while a process is modifying the region.

- A read lock allows processes to set overlapping read locks but not write locks. Other processes are allowed to lock and read data from the locked regions.
- A mandatory locks can cause problems: If a runaway process sets a mandatory exclusive lock on a file and never unlocks it, no other processes can access the locked region of the file until either a runaway process is killed or the system is rebooted.

- **If a file lock is not mandatory, it is an advisory. An advisory lock is not enforced by a kernel at the system call level**
- **The following procedure is to be followed**
  - **Try to set a lock at the region to be accessed. if this fails, a process can either wait for the lock request to become successful or go do something else and try to lock the file again.**
  - **After a lock is acquired successfully, read or write the locked region.**
  - **Release the lock after read or write operation to the file.**

Advisory locks

- A process should always release any lock that it imposes on a file as soon as it is done.
- An advisory lock is considered safe, as no runaway processes can lock up any file forcefully. It can read or write after a fixed number of failed attempts to lock the file
- Drawback: the programs which create processes to share files must follow the above locking procedure to be cooperative.

FCNTL file locking

- int fcntl (int fdesc, int cmd_flag, …);

Cmd_flag	Use
F_SETLK	Sets a file lock. Do not block if this cannot succeed immediately.
F_SETLKW	Sets a file lock and blocks the calling process until the lock is acquired.
F_GETLK	Queries as to which process locked a specified region of a file.

For file locking, the third argument is an address of a struct flock-typed variable.

This lock specifies a region of a file where the lock is to be set, unset or queried.

```
struct flock
{
        short   l_type;
        short   l_whence;
        off_t   l_start;
        off_t   l_len;
        pid_t   l_pid;
};
```

l_type and l_whence fields of flock

l_type value	Use
F_RDLCK	Sets as a read (shared) lock on a specified region
F_WRLCK	Sets a write (exclusive) lock on a specified region
F_UNLCK	Unlocks a specified region

l_whence value	*Use*
SEEK_CUR	The l_start value is added to the current file pointer address
SEEK_SET	The l_start value is added to byte 0 of file
SEEK_END	The l_start value is added to the end (current size) of the file

- **The l_len specifies the size of a locked region beginning from the start address defined by l_whence and l_start. If l_len is 0 then the length of the lock is imposed on the maximum size and also as it extends. It cannot have a –ve value.**

- **When fcntl is called, the variable contains the region of the file locked and the ID of the process that owns the locked region. This is returned via the l_pid field of the variable.**

LOCK PROMOTION AND SPLITTING

- **If a process sets a read lock and then sets a write lock on the file, the process will own only the write lock. This process is called lock promotion.**
- **If a process unlocks any region in between the region where the lock existed then that lock is split into two locks over the two remaining regions.**

Mandatory locks can be achieved by setting the following attributes of a file.

- **Turn on the set-GID flag of the file.**
- **Turn off the group execute right of the file.**
- **All file locks set by a process will be**

unlocked when process terminates.

If a process locks a file and then creates a child process via fork, the child process will not inherit the lock.

The return value of fcntl is 0 if it succeeds or -1 if it fails.

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
struct flock fvar;
int    fdesc;
```

```
while (--argc > 0)                {       /* do the
   following for each file */
if ((fdesc=open(*++argv,O_RDWR))==-1)
 {
   perror("open"); continue;
  }
fvar.l_type = F_WRLCK;
fvar.l_whence = SEEK_SET;
fvar.l_start = 0;
fvar.l_len = 0;
```

```
/* Attempt to set an exclusive (write) lock on
   the entire file */
while (fcntl(fdesc, F_SETLK,&fvar)==-1)
{
/* Set lock fails, find out who has locked the file
   */
while (fcntl(fdesc,F_GETLK,&fvar)!=-1 &&
fvar.l_type!=F_UNLCK)
{
   cout << *argv << " locked by " << fvar.l_pid<<
   " from " <<      fvar.l_start << " for "<<
   fvar.l_len << " byte for " <<
   (fvar.l_type==F_WRLCK ? 'w' : 'r')
   << endl;
        if (!fvar.l_len) break;
   fvar.l_start += fvar.l_len;
        fvar.l_len = 0;
} /* while there are locks set by other
  processes */
} /* while set lock un-successful */
```

```
// Lock the file OK. Now process data in the file
/* Now unlock the entire file */
fvar.l_type = F_UNLCK;
fvar.l_whence = SEEK_SET;
fvar.l_start = 0;
fvar.l_len = 0;
if (fcntl(fdesc, F_SETLKW,&fvar)==-1)
perror("fcntl");
}
return 0;
}  /* main */
```

QUESTIONS

- Explain the access mode flags and access modifier flags. Also explain how the permission value specified in an ‘Open’ call is modified by its calling process ‘unmask, value. Illustrate with an example **(10)**
- Explain the use of following APIs **(10)**

i) fcntl ii) lseek iii) write iv) close

- With suitable examples explain various directory file APIs (10)
- Write a C program to illustrate the use of mkfifo ,open ,read & close APIs for a FIFO file (10)
- Differentiate between hard link and symbolic link files with an example (5)
- Describe FIFO and device file classes (5)
- Explain process of changing user and group ID of files (5)

- What are named pipes? Explain with an example the use of lseek, link, access with their prototypes and argument values (12)
- Explain how *fcntl* API can be used for file record locking (8)
- Describe the UNIX kernel support for a process . Show the related data structures (10)

- **Give and explain the APIs used for the following (10)**

1. ***To create a block device file called SCS15 with major and minor device number 15 and 3 respectively and access rights read-write-execute for everyone***
2. ***To create FIFO file called FIF05 with access permission of read-write-execute for everyone***