UNIX FILES

FILE TYPES

- Regular file
- Directory file
- Fifo file
- Character device file
- Block device file

Regular file

- It mat be text or binary file
- Both the files are executable provided execution rights are set
- They can be read or written by users with appropriate permissions
- 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**

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

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

- **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

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

- **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

UNIX command	System call	Attributes changed
chmod	chmod	Changes access permission, last change time
chown	chown	Changes UID, last change time
chgrp	chown	Changes GID, last change time

touch	utime	Changes last access time, modification time
ln	link	Increases hard link count
rm	unlink	Decreases hard link count .If the hard link count is zero ,the file will be removed from the file system
vi, emac		Changes file size,last access time, last modification time

Inodes in UNIX system

- UNIX system V has an inode table which keeps track of all files
- Each entry in inode table is an inode record
- Inode record contains all attributes of file including inode number and physical address of file
- Information of a file is accessed using its inode number

- Inode number is unique within a file system
- A file record is identified by a file system ID and inode number
- Inode record doesnot contain the name of the file
- The mapping of filenames to inode number is done via directory files

112	
67	..
97	abc
234	a.out

To access a file for example /usr/abc, the kernel knows "/" directory inode number of any process, it will scan "/" directory to find inode number of "usr" directory it then checks for inode number of abc in usr . The entire process in carried out taking into account the permissions of calling process

APPLICATION PROGRAM INTERFACE TO FILES

- **Files are indentified by path names**
- **Files must 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**

- **A process can may open at most OPEN_MAX number of files**
- **The read and write system calls can be used to read and write data to opened files**
- **File attributes are queried using stat or fstat system calls**
- **File attributes are changed using chmod, chown, utime and link system calls**
- **Hard links are removed by unlink system call**

UNIX KERNEL SUPPORT FOR FILES

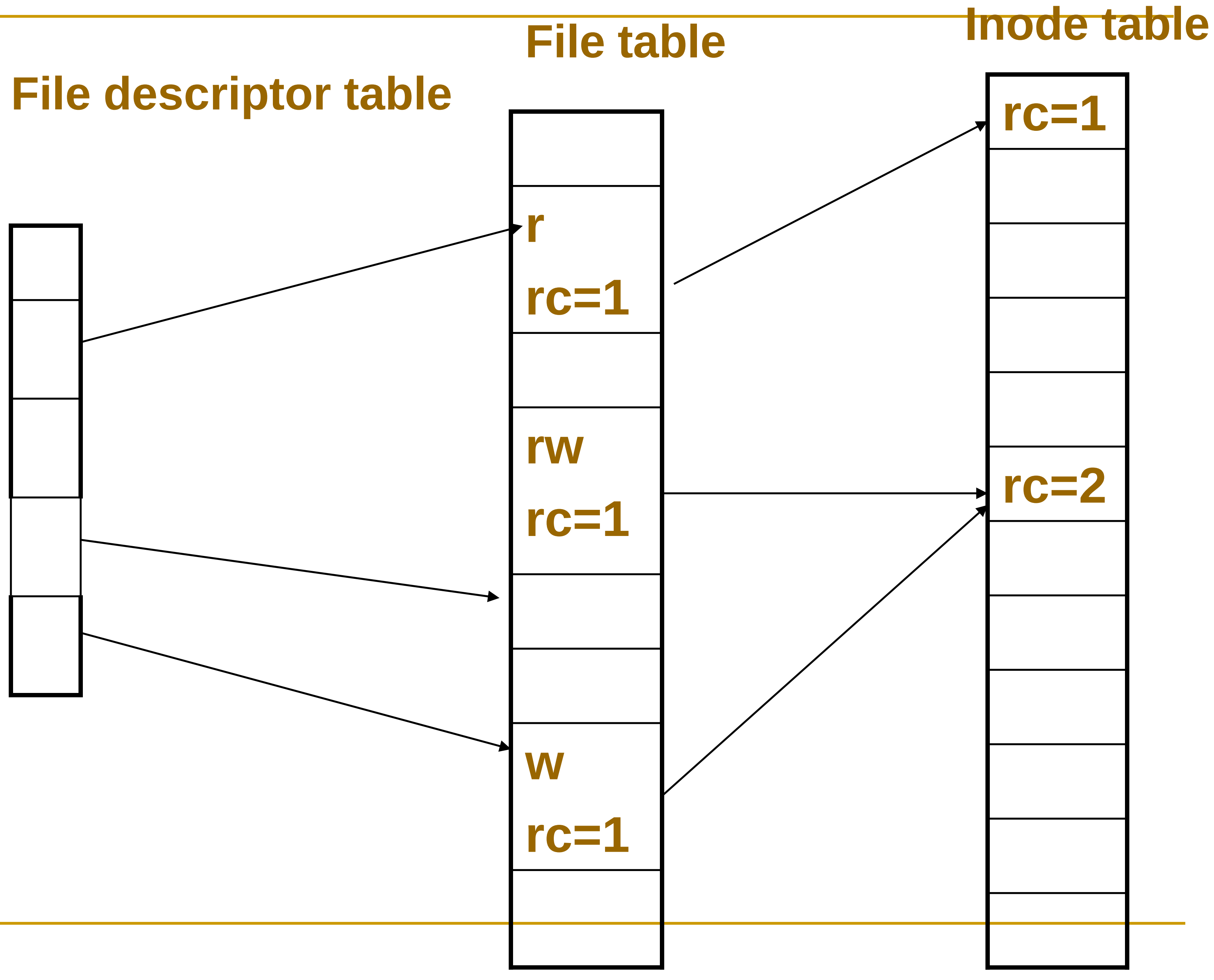
- Whenever a user executes a command, a process is created by the kernel to carry out the command execution
- Each process has its own data structures: file descriptor table is one among them
- File descriptor table has OPEN_MAX entries, and it records all files opened by the process

Kernel support to open system call

- Whenever an open function is called the kernel will resolve the pathname to file inode
- Open call fails and returns -1 if the file inode does not exist or the process lacks appropriate permissions
- Else a series of steps follow

- **The kernel will search the file descriptor table and look for first unused entry,which is returned as file descriptor of the opened file**
- **The kernel will scan the file table, which is used to reference the file If an unused entry is found then**
- **If an unused entry is found then**

1. **The process's file descriptor table entry will be set to point to file table entry**
2. **The file table entry will be set to point to inode table entry where the inode record of file is present**
3. **The file table entry will contain the current file pointer of the open file**
4. **The file table entry will contain an open mode which specifies the file is opened for read- only ,write-only etc.**
5. **Reference count of file table is set to 1.**
6. **The reference count of the in-memory inode of file is increased by 1.**


File descriptor table
File table
Inode table
r
rc=1
rw
rc=1
w
rc=1
rc=1
rc=2

Kernel support : read system call

- The kernel will use the file descriptor to index the process's file descriptor table to find file table entry to opened file
- It checks the file table entry to make sure that the file is opened with appropriate mode
- The kernel will use the file pointer used in file table entry to determine where the read operation should occur in file

- The kernel will check the type of file in the inode record and invokes an appropriate driver driver function to initiate the actual data transfer with a physical file
- If the process calls lseek function then the changes are made provided the file is not a character device file, a FIFO file, or a symbolic link file as they follow only sequential read and write operations

Kernel support : close system call

1. The kernel will set the corresponding descriptor table entry to unused
2. It decrements the reference count in file table entry by 1.if reference count !=0 then go to 6
3. File table entry is marked unused
4. The reference count in file inode table entry by 1.if reference count !=0 then go to 6

5. **If hard link count is non zero, it returns a success status, otherwise marks the inode table entry as unused and and deallocates all the physical disk storage**

6. **It returns 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 dirent in BSD UNIX

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

- Unix system also provides telldir and seekdir function

Hard and symbolic links

- **A hard link is a UNIX path name for a file**
- **To create hard link ln command is used**

  ln /usr/abc/old.c /usr/xyz/new.c
- **Symbolic link is also a means of referencing a file**
- **To create symbolic link ln command is used with option –s**

  ln –s /usr/abc/old.c /usr/xyz/new.c

Difference : cp and ln command

- **Cp command creates a duplicated copy of file to another file with a different path name**
- **Where as ln command saves space by not duplicating the copy here the new file will have same inode number as original file**

Difference : hard link and symbolic link

Hard link	Symbolic link
Does not create a new inode	Create a new inode
Cannot link directories unless it is done by root	Can link directories
Cannot link files across file systems	Can link files across file system
Increase hard link count of the linked inode	Does not change hard link count of the linked inode

General file APIs

Open	Opens a file for data access
Read	Reads data from file
Write	Writes data into file
Lseek	Allows random access of data in a file
Close	Terminates connection to a file
Stat,fstat	Queries attributes of a file

Chmod	Changes access permissions of a file
Chown	Changes UID and/or GID of a file
Utime	Changes last modification time and access time stamps of a file
Link	creates a hard link to a file
Ulink	Deletes a hard link to a file
Umask	Sets default file creation mask

Open

- ***The function establishes connection between <u>process</u> and a <u>file</u>***
- ***The prototype of the function***

```
#include <sys/types.h>
#include <fcntl.h>
int open (const char *pathname, int access
                              mode , mode_t permission);
```

- **Pathname** : It can be absolute path name or a relative path name
- **Access_mode** : An integer which specifies how file is to be accessed by calling process

Access mode flag	Use
O_RDONLY	Opens file for read-only
O_WRONLY	Opens file for write only
O_RDWR	Opens file for read & write

- Access modifier flag

- O_APPEND
- O_CREAT
- O_EXCL
- O_TRUNC
- O_NONBLOCK
- O_NOCTTY

- O_APPEND : appends data to end of file
- O_TRUNC : if the file already exists, discards its contents and sets file size to zero
- O_CREAT : creates the file if it does not exist
- O_EXCL : used with O_CREAT only. This flag causes open to fail if the file exists

- **O_NONBLOCK : specifies that any subsequent read or write on the file should be non blocking**

- **O_NOCTTY : specifies not to use the named terminal device file as the calling process control terminal**

Umask

- It specifies some access rights to be masked off
- Prototype:

```
    mode_t umask ( mode_t  new_umask);
 mode_t old_mask =
                 umask (S_IXGRP|S_IWOTH);
/*removes execute permission from group and
  write permission from others*/
```

- Actual_permission   =
                    permission & ~umask_value

Creat

- It is used to create new regular files

```
#include <sys/types.h>
#include <unistd.h>
Int creat (const char* pathname,mode_t mode)
```

Read

- This function fetches a fixed size block of data from a file referenced by a given file descriptor

```
#include <sys/types.h>
#include <unistd.h>
 ssize_t read (int fdesc ,void* buf, size_t size);
```

Write

- The write function puts a fixed size block of data to a file referenced by a file descriptor

```
#include <sys/types.h>
#include <unistd.h>
 ssize_t write (int fdesc ,void* buf, size_t size);
```

Close

- Disconnects a file from a process

```
#include <unistd.h>
int close (int fdesc);
```

- Close function will allocate system resources
- It a process terminates without closing all the files it has opened ,the kernel will close those files for the process

fcntl

- The function helps to query or set access control flags and the close-on-exec flag of any file descriptor

```
#include <fcntl.h>
Int fcntl (int fdesc ,int cmd);
```

- cmd argument specifies which operation to perform on a file referenced by the fdesc argument

- cmd value
- F_GETFL : returns the access control flags of a file descriptor fdesc
- F_SETFL : sets or clears control flags that are specified
- F_GETFD : returns the close-on-exec flag of a file referenced by fdesc
- F_SETFD : sets or clears close-on-exec flag of a file descriptor fdesc
- F_DUPFD : duplicates the file descriptor fdesc with another file descriptor

lseek

- the lseek system call is used to change the file offset to a different value
- Prototype :

```
#include <sys/types.h>
#include <unistd.h>
Off_t lseek(int fdesc , off_t pos, int whence)
```

- Pos :
- **specifies a byte offset to be added to a reference location in deriving the new file offset value**
- **Whence location reference**
- **SEEK_CUR current file pointer address**
- **SEEK_SET the beginning of a file**
- **SEEK_END the end of a file**

link

- The link function creates a new link for existing file
- Prototype :

```
#include <unistd.h>
int link (const char* cur_link ,const char*
                                        new_link)
```

unlink

- **Deletes link of an existing file**

```
#include <unistd.h>
int unlink (const char* cur_link );
```

- **Cannot link a directory unless the calling function has the superuser privilege**

Stat fstat

- **Stat and fstat retrieve arguments of a given file**

```
#include <sys/types.h>
#include <unistd.h>
int stat (const char* path_name,struct
                                    stat* statv)
int fstat (const int fdesc,struct stat* statv)
```

- **Struct stat**

```
{
    dev_ts       t_dev;
    ino_t        st_ino;
    mode_t       st_mode;
    nlink_t      st_nlink;
    uid_t        st_uid;
    gid_t        st_gid;
    dev_t        st_rdev;
    off_t        st_size;
    time_t       st_mtime
    time_t       st_ctime
};
```

- If pathname specified in stast is a symbolic link then the attributes of the non-symbolic file is obtained
- To avoid this lstat system call is used
- It is used to obtain attribites of the symbolic link file

```
int lstat (const char* path_name , struct stat*
                                            statv);
```

```
/* Program to emulate the UNIX ls -l command */
#include <iostream.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
static char xtbl[10] = "rwxrwxrwx";
```

```
#ifndef MAJOR
#define MINOR_BITS      8
#define MAJOR(dev)     ((unsigned)dev >>
                            MINOR_BITS)
#define MINOR(dev)     ( dev &
                              MINOR_BITS)
#endif
```

/* Show file type at column 1 of an output line */

```
static void display_file_type ( ostream& ofs, int
                                          st_mode )

  {
      switch (st_mode &S_IFMT)
      {
            case  S_IFDIR:   ofs << 'd'; return;
                                     /* directory file */
            case S_IFCHR:  ofs << 'c'; return;
                             /* character device file */
            case S_IFBLK:   ofs << 'b'; return;
                                 /* block device file */
```

```
        case S_IFREG:   ofs << ' '; return;
                                        /* regular file */
        case S_IFLNK:  ofs << 'l'; return;
                                  /* symbolic link file */
        case S_IFIFO:   ofs << 'p'; return;
                                          /* FIFO file */
            }
}
```

```
/* Show access permission for owner, group,
   others, and any special flags */
   static void display_access_perm ( ostream&
                                     ofs, int st_mode )
   {
       char amode[10];
       for (int i=0, j= (1 << 8); i < 9; i++, j>>=1)
      amode[i] = (st_mode&j) ? xtbl[i] : '-';

                    /* set access permission */
```

```
/* set access permission */
     if (st_mode&S_ISUID)
    amode[2] = (amode[2]=='x') ? 'S' : 's';
     if (st_mode&S_ISGID)
    amode[5] = (amode[5]=='x') ? 'G' : 'g';
     if (st_mode&S_ISVTX)
     amode[8] = (amode[8]=='x') ? 'T' : 't';

     ofs << amode << ' ';
  }
```

```
/* List attributes of one file */
    static void long_list (ostream& ofs, char* path_name)
    {
        struct stat   statv;
        struct group        *gr_p;
        struct passwd       *pw_p;
        if (stat (path_name, &statv))
        {
            perror( path_name );
            return;
        }
```

```
display_file_type( ofs, statv.st_mode );
display_access_perm( ofs, statv.st_mode );
ofs << statv.st_nlink;
                    /* display hard link count */
gr_p = getgrgid(statv.st_gid);
                    /* convert GID to group name */
pw_p = getpwuid(statv.st_uid);
                    /*convert UID to user name */

ofs << ' ' << pw_p->pw_name << ' ' <<
                    gr_p->gr_name << ' ';
```

```
if ((statv.st_mode&S_IFMT) == S_IFCHR ||
        (statv.st_mode&S_IFMT)==S_IFBLK)
      ofs << MAJOR(statv.st_rdev) << ','
          <<    MINOR(statv.st_rdev);
        else ofs << statv.st_size;
   /* show file size or major/minor no. */
    ofs << ' ' << ctime (&statv.st_mtime);
        /* print last modification time */
        ofs << ' ' << path_name << endl;
                /* show file name */
  }
```

```
/* Main loop to display file attributes one file
                                        at a time */
  int main (int argc, char* argv[])
  {
      if (argc==1)
      cerr << "usage: " << argv[0] << " <file
           path name> ...\n";
      else while (--argc >= 1) long_list( cout,
                                        *++argv);
      return 0;
  }
```

Access

- The access function checks the existence and/or access permission of user to a named file

```
#include <unistd.h>
int access (const char* path_name, int flag);
```

- The flag contains one or more bit flags
- **Bit flags USE**

Bit flags	USE
F_OK	checks whether the file exists
R_OK	checks whether calling process has read permission
W_OK	checks whether calling process has write permission
X_OK	checks whether calling process has read permission

Chmod fchmod

- **The chmod and fchmod functions change file access permissions for owner, group and others and also set-UID ,set-GID and sticky bits**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int chmod (const char* path_name, mode_t flag);
int fchmod (int fdsec, mode_t flag);
```

- **Flag argument contains new access permissions and any special flags to be set on the file**

- **Flag value can be specified as an octal integer value in UNIX, or constructed from the manifested constants defined in <sys/stat.h>**

Chown fchown lchown

- **The chown and fchown function change the user ID and group ID of files**
- **lchown changes the ownership of symbolic link file**

```
#include <unistd.h>
#include <sys/types.h>
int chown (const char* path_name,
                                        uid_t uid, gid_t gid);
int fchown (int fdesc, uid_t uid, gid_t gid);
int lchown (const char* path_name,
                                        uid_t uid, gid_t gid);
```

utime

- The function modifies the access and modification time stamps of a file

```
#include <unistd.h>
#include <sys/types.h>
#include <utime.h>
int utime (const char* path_name,
                              struct utimbuf* times);
```

- **Struct utimbuf**

```
{
  time_t actime;
  time_t modtime;
};
```

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 the 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 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 the 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 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**

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 struct flock-typed variable.

```
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 locked 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 a write lock the read lock is now covered by a 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 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 */
```

Directory File APIs

- Why do need directory files? Uses?
- 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 ownerv,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 superuser privileges.

These API’s require that the directories to be removed 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 nonblocking to the process.**

FIFO File APIs

- These are special device files used for interprocess communication.
- These are also known as named files
- 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

interprocess 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);
```

- **To QUERY the path name to which a symbolic link refers, users must use the readlink API. The arguments are:**
- **sym_link is the path name of the symbolic link**
- **buf is a character array buffer that holds the return path name referenced by the link**
- **size specifies the maximum capacity of the buf argument**

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***