3.2 File and Record Locking:

UNIX systems allow multiple processes to read and write the same file concurrently which provides data sharing among processes. It also renders difficulty for any process in determining when data in a file can be overridden by another process.

In some of the applications like a database manager, where no other process can write or read a file while a process is accessing a database file. To overcome this drawback, UNIX and POSIX systems support a file locking mechanism.

File locking is applicable only for regular files. It allows a process to impose a lock on a file so that other processes cannot modify the file until it is unlocked by the process.

A process can impose a write lock or a read lock on either a portion of a file or an entire file.

The difference between write locks and read locks is that when a write lock is set, it prevents other processes from setting any overlapping read or write locks on the locked region of a file. On the other hand, when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region of a file.

The intention of a write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region. A write lock is also known as an *exclusive lock*.

The use of a read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region. Other processes are allowed to lock and read data from the locked regions. Hence, a read lock is also called a *shared lock*.

3.2.1 Mandatory Lock

Mandatory locks are enforced by an operating system kernel.

If a mandatory exclusive lock is set on a file, no process can use the *read* or *write* system calls to access data on the locked region.

If a mandatory shared lock is set on a region of a file, no process can use the *write* system call to modify the locked region.

It is used to synchronize reading and writing of shared files by multiple processes: If a process locks up a file, other processes that attempts to write to the locked regions are blocked until the former process releases its lock.

Mandatory locks may 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 the runaway process is killed or the system is rebooted.

System V.3 and V.4 support mandatory locks.

3.2.2 Advisory Lock

An advisory lock is not enforced by a kernel at the system call level.

This means that even though lock (read or write) may be set on a file, other processes can still use the *read* or *write* APIs to access the file.

To make use of advisory locks, processes that manipulate the same file must cooperate such that they follow this procedure for every read or write operation to the file:

a. 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 later.
b. After a lock is acquired successfully, read or write the locked region release the lock
c. The drawback of advisory locks are that programs that create processes to share files must follow the above file locking procedure to be cooperative. This may be difficult to control when programs are obtained from different sources.

All UNIX and POSIX systems support advisory locks.

UNIX System V and POSIX.I use the *fcntl* API for file locking. The prototype of the *fcntl* API is:

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd_flag, …);
```

The *fdesc* argument is a file descriptor for a file to be processed. The *cmd flag argu*ment defines which operation is to be performed.

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 to *fcntl is* an address of a *struct flock-typed* variable. This variable specifies a region of a file where the lock is to be set, unset, or queried. The *struct flock* is declared in the <fcntl.h> as:

```
struct flock
{
    short l_type;   // what lock to be set or to unlock file
    short l_whence;      // a reference address for the next field
    off_t l_start;  //offset from the l_whence reference address
    off_t l_len;    // how many bytes in the locked region
    pid_t l_pid;    //PID of a process which has locked the file
};
```

The possible values of l_type are:

l_ type value	Use
F_RDLCK	Sets a 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

The possible values of l_whence and their uses are:

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

3.2.3 Lock Promotion and Lock splitting:

If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512.

The previous read lock from 0 to 256 is now covered by the write lock, and the process does not own two locks on the region from 0 to 256. This process is called *lock promotion.*

Furthermore, if the process now unlocks the file from 128 to 480, it will own two write locks o the file: one from 0 to 127 and the other from 481 to 512. This process is called *lock splitting*.

The procedure for setting the mandatory locks for UNIX system V3 and V4 are:

The following *file_lock.C* program illustrates a use of *fcntl* for file locking:

```
#include <iostream.h>
 #include <stdio.h>
#include <sys/types.h>
 #include <fcntl.h>
#include <unistd.h>

int main (int argc, cnar* 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, FSETLK,&fvar)==-1) {
/* Set lock falls, 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(fdosc, F_SETLKW,&fear)==-1) perror("fcntl");
}
return 0;
) /* main */
```

3.3 Directory File APIs

Directory files in UNIX and POSIX systems are used to help users in organizing their files into some structure based on the specific use of file.

They are also used by the operating system to convert file path names to their inode numbers.

Directory files are created in BSD UNIX and POSIX.1 by mkdir API:

```
#include <sys/stat.h>
#include <unistd.h>
int mkdir ( const char* path_name, mode t mode );
```

1. The path_name argument is the path name of a directory to be created.
2. The mode argument specifies the access permission for the owner, group and others to be assigned to the file.
3. The return value of mkdir is 0 if it succeeds or -1 if it fails.

UNIX System V.3 uses the *mknod* API to create directory files.

UNIX System V.4 supports both the *mkdir* and *mknod APIs* for creating directory files.

The difference between the two *APIs is* that a directory created by ***mknod*** **does not contain the "."** and ".." links. On the other hand, a directory created by *mkdir* has the "." and ".." links created in one atomic operation, and it is ready to be used.

A directory file is a record-oriented file, where each record stores a file name and the mode number of a file that resides in that directory.

The following portable functions are defined for directory file browsing. These functions are defined in both the <dirent.h> and <sys/dir.h> headers.

```
#include <sys/types.h>
#if defined (BSD) && !_POSIX_SOURCE
#include <sys/dir.h>
 typedef struct direct Dirent;
#else
#include <dirent.h>
typedef struct dirent Dirent;
#endif
DIR* opendir (const char* path_name);
Dirent* readdir (DIR* dir_fdesc);
int closedir (DIR* dir_fdesc);
void rewinddir (DIR* dir_fdesc);
```

The uses of these functions are:

opendir: Opens a directory file for read-only. Returns a file handle DIR* for future reference of the file.

readdir: Reads a record from a directory file referenced by *dir_fdesc* and returns that record information.

closedir: Closes a directory file referenced by *dir_fdesc*.

rewinddir: Resets the file pointer to the beginning of the directory file referenced by *dir_fdesc*. The next call to *readdir* will read the first record from the file.

UNIX systems support additional functions for random access of directory file records. These functions are not supported by POSIX.1:

telldir: Returns the file pointer of a given dir_fdesc.

seekdir: Changes the file pointer of a given dir_fdesc to a specified address.

Directory files are removed by the *rmdir* API. Its prototype is given below:

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

The following *list_dir.C* program illustrates uses of the *mkdir, opendir, readdir, closedir,* and rmdirAPIs:

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#if defined (BSD) && !_POSIX_SOURCE
#include <sys/dir.h>
  typedef struct direct Dirent;
#else
#include <dirent.h>
  typedef struct dirent Dirent;
#endif
```

```
int main (int argc, char* argv[ ])
 {
Dirent* dp;
DIR*    dir_fdesc;
while (--argc > 0 ) {          /* do the following for each file */
if ( !(dir_fdesc = opendir( *++argv ) )) {
if (mkdir( *argv, S_ IRWXU|S_IRWXG|S_IRWXO) == -1 )
 perror( "opendir" );
continue;
}
/*scan each directory file twice*/
for (int i=0;i<2;i++) {
for ( int cnt=0; dp=readdir( dir_fdesc );) {
if (i) cout << dp->d_name << endl;
if (strcmp( dp->d_name, ".") && strcmp( dp->d_name, ".. ") )
 cnt++;                              /*count how many files in directory*/
if (!cnt) { rmdir( *argv ); break;) /* empty directory */
  rewinddir( dir fdesc );    / reset pointer for second round */
  }
  closedir( dir fdesc );
 }
}
```

3.4 Device File APIs

Device files are used to interface physical devices with application programs.

Specifically, when a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer.

Device files may be character-based or block-based.

UNIX systems define the ***mknod*** API **to create device** files.

#include <sys/stat.h>

#include <unistd.h>

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

1. The path_name argument is the path name of a directory to be created.
2. The mode argument specifies the access permission for the owner, group and others to be assigned to the file.
3. The *device_id* contains the major and minor device numbers and is constructed in most UNIX systems as follows: The lowest byte of *a device_id* is set to a minor device number and the next byte is set to the major device number. For example, to create a block device file called SCSI5 with major and minor numbers of 15 and 3, respectively, and access rights of read-write-execute for everyone, the *mknod* system call is:
   mknod("SCSI5", S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO, (15<<8) 13);
4. The major and minor device numbers are extended to fourteen and eighteen bits, respectively.
5. In UNIX, if a calling process has no controlling terminal and it opens a character device file, the kernel will set this device file as the controlling terminal of the process. How-ever, if the O_NOCTTY flag is set in the *open* call, such action will be suppressed.

6. 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.

The following *test mknod.C* program illustrates use of the *mknod, open, read, write,* and *close* APIs on a block device file.

```
#include <iostream.h>
#include <stdio.h>
 #include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
int main( int argc, char* argv[ ] ) {
if(argc!=4){
        cout << "usage: " << argv[0] << " <file> <major no> <minor no>\n";
        return 0;
}
int major = atoi( argv[2]), minor = atoi( argv[3] );
     (void) mknod( argv[1], S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO, ( major <<8) | minor );
int rc=1, fd = open(argv[1], O_RDWR | O_NONBLOCK | O_NOCTTY );
char buf[256];
while ( rc && fd != -1 )
 if (( rc = read( fd, buf, sizeof( buf )) ) < 0 )
  perror( "read" );
       else if ( rc) cout << buf << endl;
close(fd);
}
```