General File APIs
Module 3

General file APIs

- **open** Opens a file for data access
- **read** Reads data from a file
- **write** Writes data to a 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
- **unlink** Deletes a hard link of a file
- **umask** Sets default file creation mask

Open

- ***The function establishes connection between process and a file***
- ***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 de allocate system resources.
- If 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 a link of an existing file.

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

- Cannot link a directory unless the calling function has the super user privilege

stat fstat

- stat and fstat retrieve attributes 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_t       st_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_atime;
    time_t      st_mtime
    time_t      st_ctime
};
```

- If pathname specified in stat 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 attributes 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

- 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 execute permission

chmod fchmod

- The chmod and fcmod 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 and lchown

- The chown and fchown functions 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 timestamps 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 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 database.
- 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;     /*Specifies the type of the lock*/
        short    l_whence; /*specifies what the offset is relative
    to
        off_t    l_start;   /*offset of the start of the region*/
        off_t    l_len;   /*length of the region to be locked*/
        pid_t    l_pid; /* process ID of the process holding the
    lock */
};
```

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.

Directory File APIs

- Why do we need directory files?
- To help 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 a directory

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

Returns: 0 if successful, -1 on error

- **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

Returns: directory file handler if successful, NULL on error

readdir:

struct dirent* readdir(DIR* dir_fdesc);

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

Returns: pointer to the struct dirent structure if successful, NULL on error

closedir :

int closedir (DIR* dir_fdesc);

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

Returns: 0 if successful, -1 on error

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);

Returns: 0 if successful, -1 on error

- Used to remove the directory files.
- Users may also use the **unlink API to remove**

directories provided they have super user privileges.

- These APIs require that the directories to be removed **must be empty,** in that they contain no files other than "." and ".." links.

Directory structure

```
struct dirent
{
   long d_ino;                    /* inode number*/
   off_t d_off;                   /* offset to this dirent */
   unsigned short d_reclen;    /* length of this record*/
    unsigned char d_type;          /* type of file*/
   char d_name[NAME_MAX+1];      /* filename
                                     (null terminated) */
};
```

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);

Returns: 0 if successful, -1 on error

- **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);
```

Returns: 0 if successful, -1 on error

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**.
- /usr/go/test1 link /usr/joe/symlnk

To create :

int symlink (const char* org_link, const char* sym_link);

Returns: 0 if successful, -1 on error

int readlink (const char* sym_link, char* buf, int size);

Returns: actual number of characters of a pathname that is placed in the buf argument if successful, -1 on error

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***