The File

- We will look at some fundamental primitives UNIX provides for handling files from within programs.

- These primitives consist of a small set of system calls that give direct access to the I/O facilities provided by the UNIX kernel.

- They form the building blocks for all UNIX I/O and any other file mechanism will ultimately be based around them.

- The names of these primitives are :

  open - opens a file for reading or writing
  creat - creates an empty file
  close - closes a previously opened file
  read - extracts information from a file
  write - places information into a file
  lseek - moves to a specified byte in a file
  unlink - removes a file

- Example

```
/* rudimentary example program */
#include <fcntl.h>

int main (void)
{
        int fd, nread;
        char buf[1024];

        /* open file for reading */
        fd = open ("data", O_RDONLY);

        /* read in the data */
        nread = read (fd, buf, 1024);

        /* close the file */
        close (fd);
}
```

- In the open system call, the second argument is an integer constant defined in **fcntl.h**.

- There are three constants defined in **fcntl.h** that are of interest to us:

  O_RDONLY - open file for reading only
  O_WRONLY - open file for writing only
  O_RDWR - open file for both reading and writing

- If the open call succeeds and the file is opened, the return value **filedes** will contain a non-negative integer - the file descriptor. The operating system identifies the file to be used via the file descriptor.

- All programs start out with three files already open: the standard input, standard output, and standard error files. These file descriptors are 0 (**stdin**), 1 (**stdout**), and 2 (**stderr**), respectively.

UNIX file systems

- We have seen that files may be organized into groups called directories and directories may themselves be grouped together into an object called a file system.

- File systems allow the directory structure to be accommodated across several distinct, physical disks or disk partitions, while retaining the uniform user view of the structure.

- File systems are also called demountable volumes because it is possible to dynamically introduce complete subsections of the hierarchy anywhere into the tree structure.

- More to the point, it is also possible to dynamically disconnect, or unmount, a complete file system from the hierarchy and therefore, make it temporarily inaccessible to users.

- For example, a file system on a floppy disk must be mounted on to the hierarchy before any of the files on it can be accessed.

- UNIX file systems (UNIX System V) are divided into a number of logical blocks. These blocks are typically **1024-bytes** in size. Older versions of UNIX use 512-byte file systems.

- Every such file system contains four distinct sections: a **bootstrap area**, a file system **super block**, a number of blocks reserved for **inode structures** and an area reserved for **data blocks** that make up the files on that particular file system.

- The first block (logical block 0) is reserved for use as a bootstrap block. That is, it may contain a hardware-specific boot program that is used to load UNIX at system start up time.

- The file system super block (**logical block 1**) contains all the vital information about the file system, for example the total file system size (r blocks), the number of blocks reserved for inodes (n - 2), and the date and time that the file system was last updated.

- inode structures are 64-bytes in size and consequently 16 fit into each n - 2 blocks allocated for inodes, assuming a 1024-byte block size.

- There is one inode for each file and it contains information about a file's location, length, access modes, relevant dates, owner, and the like.

- File systems are created using the **mkfs** program and it is when this program is executed that the sizes of the inode areas and data areas for the file system must be specified.

- The following is an example of the steps that would be required to prepare and mount a 1.44 MB, 3.5" floppy disk (A drive) for use on the **LINUX** system:

    1. Format the floppy disk using **fdformat**

        /usr/bin/fdformat /dev/fd0

    2. Build an empty file system on with **mkfs**

        /sbin/mkfs /dev/fd0 2880

    3. Mount it with **mount**

        /bin/mount /dev/fd0 /mnt

    4. Copy files to it or create files on it

        cp /dir/file /mnt/dir/file

    5. Unmount it with **umount**

        /bin/umount /dev/fd0

- For DOS disks you may simply use the **mcopy**, **mdir** family of commands to move files back and forth.

- One of the more powerful UNIX file systems facilities is the emergence of distributed file systems, where users on several machines have simultaneous access to the same files.

- One utility to do this is called **NFS** (Network File Systems) which allows directory hierarchies on remote machines to be "mounted" onto a directory on the local system. The mechanism is transparent which means that programs do not need changing to use the remote files.

UNIX special files

- The peripheral devices attached to UNIX systems (disks, terminals, printers, tape units, etc) are represented by file names in the file system.

- Unlike ordinary disk files, reads and writes to these special files cause data to be transferred directly between the system and the appropriate peripheral device.

- Typically, these special files are stored in a directory called **/dev**. So, for example

  /dev/tty00
  /dev/tty01
  /dev/tty02

  could be names given to three of a system's terminal ports, and

  /dev/lp
  /dev/rmt0

  could refer to a line printer and magnetic tape unit respectively.
- The **LINUX** system uses the following scheme for serial ports

  /dev/cua0, /dev/ttyS0 - COM 1
  /dev/cua1, /dev/ttyS1 - COM 2
  /dev/cua2, /dev/ttyS2 - COM 3
  /dev/cua3, /dev/ttyS3 - COM 4

- Special files may be used at command level or within programs just like ordinary files. For example,

  cat foo > /dev/lp
  cat foo > /dev/ttyS0

would cause the file called **foo** to be written to the line printer and modem, respectively (access permissions permitting).

- From within a program, open, close, read and write can all be used on special files. For example,

```
#include <fcntl.h>

int main (void)
{
        int i, fd;

        fd = open ("/dev/ttyS0", O_WRONLY);
        for (i = 0; i < 100; i++)
                write (fd, "x", 1);

        close (fd);
}
```

would cause 100 x's to be written to terminal port **ttyS0**.

Block and Character special files

- UNIX special files are divided into two categories: block devices and character devices.

    1. Block special files include devices such as disk or magnetic tape units. The transfer of data between these devices and the kernel occurs in standard-sized blocks, a typical block size being1024 bytes.

       All block devices will support random access.

    2. Character special files include devices such as terminal lines, modem lines and printer devices that do not share this structured transfer mechanism.

       Random access may, or may not, be supported.

       Data transfer is not done in fixed sized blocks, but in terms of byte streams of arbitrary length.

- It is important to note that file systems can only exist on block devices.

- **UNIX** uses two operating system configuration tables called the block device and character device tables to associate a peripheral device with the device-specific code to drive the particular peripheral.

- Both tables are indexed using an integer called the major device number which is stored in the special file's inode.

- The sequence for transmitting data to and from a peripheral device is as follows:

    1. read or write system calls access the special file's inode in the normal way.

    2. The system checks a flag within the inode structure to see whether the device is a block or character device. The **major number** is also extracted.

    3. The **major number** is used to index into the appropriate device configuration table and the device specific driver routine is called to perform the data transfer.

- In this way accesses to peripheral devices can be entirely consistent with accesses to normal disk files.

- A second integer value called the **minor device number** is also stored in the inode and is passed to the device driver routines in order to identify exactly which port is being accessed on those devices which support more than one peripheral port.

- For example, on an 8-line terminal board each terminal line would share the same **major number** (and consequently the same set of driver routines), but would each have their own unique **minor number** in the range 0 to 7 to identify the particular line being accessed.