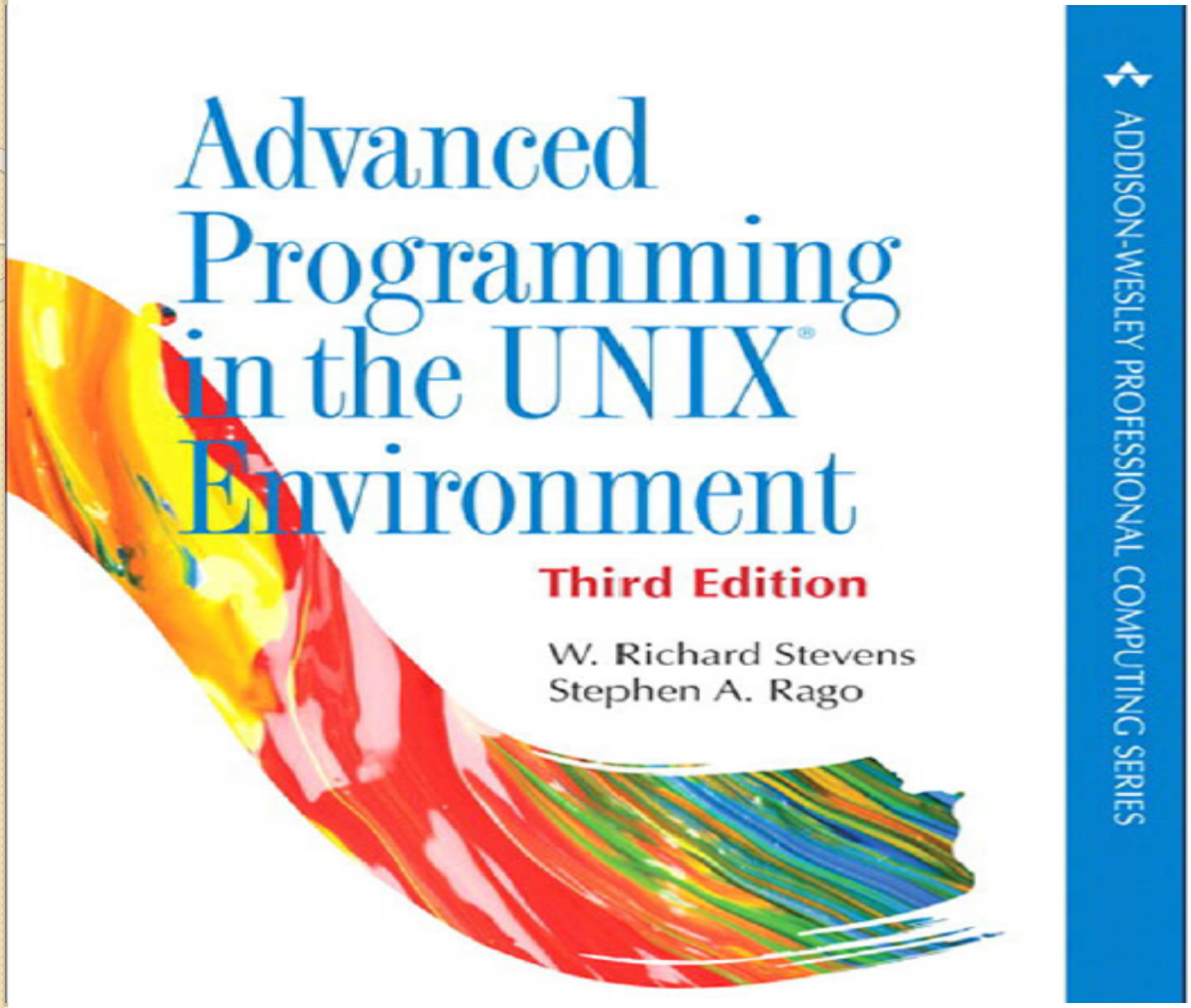
Unix System Programming
Unit I

Unix Basics:

- File Types
- The UNIX File System
- The UNIX File Attributes
- Inodes in UNIX
- Application Program Interface to Files
- UNIX Kernel Support for Files
- Relationship of C Stream Pointers and File Descriptors
- Directory Files
- Hard and Symbolic Links
- File and Record Locking
- Directory File APIs
- Device File APIs
- FIFO File APIs


Advanced Programming in the UNIX® Environment
Third Edition
W. Richard Stevens
Stephen A. Rago
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

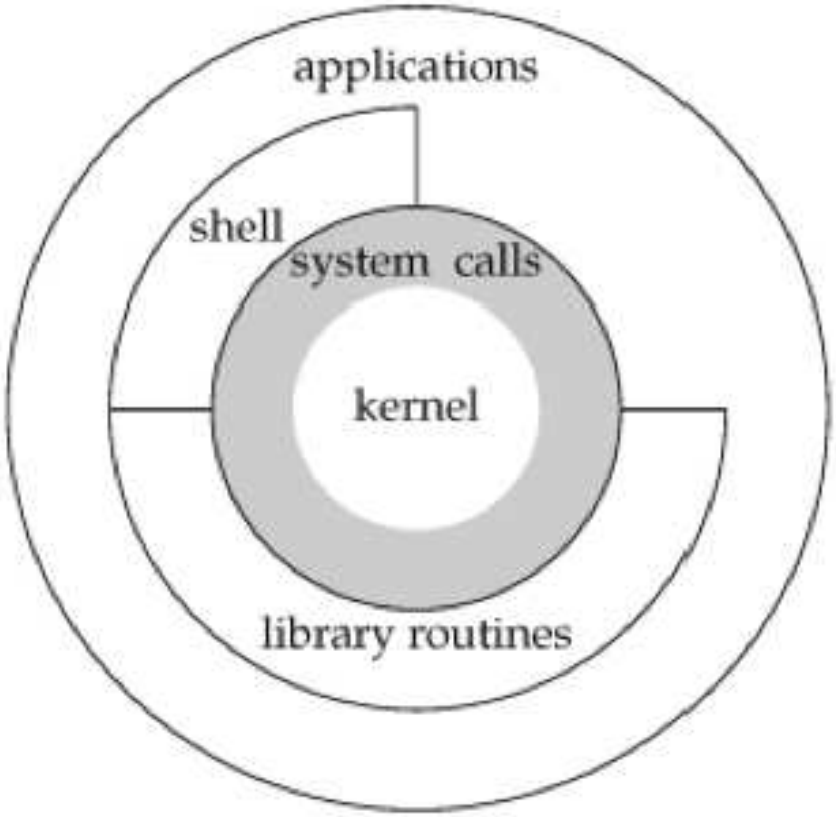
UNIX history

- http://www.unix.org/what_is_unix/history_timeline.html
- Originally developed in 1969 at Bell Labs by Ken Thompson and Dennis Ritchie.
- 1973, Rewritten in C. This made it portable and changed the history of OS
- 1974: Thompson, Joy, Haley and students at Berkeley develop the Berkeley Software Distribution (BSD) of UNIX
- two main directions emerge: BSD and what was to become "System V"

Notable dates in UNIX history

- 1984 4.2BSD released (TCP/IP)
- 1986 4.3BSD released (NFS)
- 1991 Linus Torvalds starts working on the Linux kernel
- 1993 Settlement of USL vs. BSDi; NetBSD, then FreeBSD are created
- 1994 Single UNIX Specification introduced
- 1995 4.4BSD-Lite Release 2 (last CSRG release); OpenBSD forked off NetBSD
- 2000 Darwin created (derived from NeXT, FreeBSD, NetBSD)
- 2003 Xen; SELinux
- 2005 Hadoop; DTrace; ZFS; Solaris Containers
- 2006 AWS ("Cloud Computing" comes full circle)
- 2007 iOS; KVM appears in Linux
- 2008 Android; Solaris open sourced as OpenSolaris

UNIX Basics: Architecture



System Calls and Library Functions

- System calls are entry points into kernel code where their functions are implemented.

  e.g. write( ).

- Library calls are transfers to user code which performs the desired functions.

  e.g. printf().

Different shells

- sh

The Bourne shell, called "sh," is one of the original shells, developed for Unix computers by Stephen Bourne at AT&T's Bell Labs in 1977. Its long history of use means many software developers are familiar with it. It offers features such as input and output redirection, shell scripting with string and integer variables, and condition testing and looping.

bash

- The popularity of sh motivated programmers to develop a shell that was compatible with it, but with several enhancements. Linux systems still offer the sh shell, but "bash" -- the "Bourne-again Shell," based on sh -- has become the new default standard
- Shell scripts are complex sets of commands that automate programming and maintenance chores; being able to reuse these scripts saves programmers time. Conveniences not present with the original Bourne shell include command completion and a command history.

csh and tcsh

- Developers have written large parts of the Linux operating system in the C and C++ languages. Using C syntax as a model, Bill Joy at Berkeley University developed the "C-shell," csh, in 1978. Ken Greer

- Tcsh fixed problems in csh and added command completion, in which the shell makes educated "guesses" as you type, based on your system's directory structure and files. Tcsh does not run bash scripts, as the two have substantial differences.

Files and Directories

- The UNIX filesystem is a tree structure, with all partitions mounted under the root (/).

Note: File names may consist of any character except / and NUL as pathnames are a sequence of zero or more filenames separated by /'s.

- Directories are special "files" that contain mappings between inodes and filenames, called directory entries.
- All processes have a current working directory from which all relative paths are specified. (Absolute paths begin with a slash, relative paths do not.)

User Identification

- User IDs and group IDs are numeric values used to identify users on the system and grant permissions appropriate to them.

Standard I/O

- file descriptors: Small, non-negative integers which identify a file to the kernel. The shell can redirect any file descriptor.
- kernel provides unbuffered I/O through e. g. open read write lseek close
- kernel provides buffered I/O through e.g. fopen fread fwrite getc putc

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Figure 1.3 List all the files in a directory

```c
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Figure 1.4 Copy standard input to standard output

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Figure 1.5 Copy standard input to standard output, using standard I/O

Processes

- Programs executing in memory are called processes.
- Programs are brought into memory via one of the six exec(3) functions.
- Each process is identified by a guaranteed unique non-negative integer called the processes ID.
- New processes can only be created via the fork(2) system call.
- process control is performed mainly by the fork(2), exec(3) and waitpid(2) functions.

File I/O

1. Introduction about Unix
2. File Descriptors
3. open Function
4. creat Function
5. close Function
6. lseek Function
7. read Function
8. write Function
9. File Sharing
10. Atomic Operations
11. dup and dup2 Functions
12. fcntl Function
13. ioctl Function
14. /dev/fd

File I/O

In Unix there are five functions to perform file I/O:

- open,
- read,
- write,
- lseek,
- close.

open()

A file is opened or created by calling the open function

```
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /* mode_t mode   */ );
```

Returns: file descriptor if OK, -1 on error

open()

- The pathname is the name of the file to open or create.
- The second argument is oflag argument.
  - This argument is formed by ORing together one or more of the following constants from the <fcntl.h> header:
    - **O_RDONLY Open for reading only.**
    - **O_WRONLY Open for writing only.**
    - **O_RDWR Open for reading and writing.**

Most implementations define

O_RDONLY as 0, O_WRONLY as 1, and O_RDWR as 2, for compatibility with older programs.

Optional constants

- O_APPEND Append to the end of file on each write.
- O_CREAT Create the file if it doesn't exist. This option requires a third argument to the open function, the *mode, which specifies the access permission bits of the new* file.

Optional constants

- O_EXCL Generate an error if O_CREAT is also specified and the file already exists.

- O_TRUNC If the file exists and if it is successfully opened for either write-only or read–write, truncate its

  length to 0.

- O_NOCTTY If the pathname refers to a terminal device, do not allocate the device as the controlling terminal for this process.

- O_NONBLOCK : If the pathname refers to a FIFO, a block special file, or a character special file, this option sets the non blocking mode for both the opening of the file and subsequent I/O.

Different files:

a) FIFO file- A FIFO is similar to a pipe. A FIFO (First In First Out) is a one-way flow of data. FIFOs have a name, so unrelated processes can share the FIFO.

b) Device or special files- are used for device I/O on UNIX and Linux systems. They appear in a file system just like an ordinary file or a directory.

- On UNIX systems, there are two flavors of special files for each device:
  - character special files
  - block special files.

When a *character special file* is used for device I/O, data is transferred one character at a time. This type of access is called raw device access.

When a *block special file* is used for device I/O, data is transferred in large fixed-size blocks. This type of access is called block device access.

Filename and Pathname Truncation

What happens if NAME_MAX is 14 and we try to create a new file in the current directory with a filename containing 15 characters?
Silently truncating the filename beyond the 14th character.
BSD-derived systems returned an error status, with errno set to ENAMETOOLONG.
Silently truncating the filename presents a problem that affects more than simply the creation of new files.
If NAME_MAX is 14 and a file exists whose name is exactly 14 characters, any function that accepts a *pathname argument, such as open* or stat, has no way to determine what the original name of the file was, as the original name might have been truncated.
With POSIX.1, the constant _POSIX_NO_TRUNC determines whether long filenames and long pathnames are truncated or whether an error is returned
For example, SVR4-based systems do not generate an error for the traditional System V file system, S5.
For the BSD-style file system (known as UFS), SVR4-based systems generate an error.
DOS-compatible file system, as DOS silently truncates filenames that don't fit in an 8.3 format.
BSD-derived systems and Linux always return an error.

creat( )

A new file can also be created by calling the creat function.

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

Returns: file descriptor opened for write-only if OK, -1 on error

Note that this function is equivalent to

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

close ( )

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, -1 on error

Closing a file also releases any record locks that the process may have on the file.

Note: When a process terminates, all of its open files are closed automatically by the kernel. Many programs take advantage of this fact and don't explicitly close open files.

lseek( )

- Every open file has an associated "current file offset," normally a non-negative integer that measures the number of bytes from the beginning of the file.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns: new file offset if OK, -1 on error

- If *whence is SEEK_SET, the file's offset is set to offset bytes from the beginning of* the file.
- If *whence is SEEK_CUR, the file's offset is set to its current value plus the offset.*
- If *whence is SEEK_END, the file's offset is set to the size of the file plus the offset.*

Example:

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

To determine the current offset, we can seek zero bytes from the current position

Note: off_t is defined in sys/types.h header file and is used to measure the file offset in bytes from the beginning of the file.

- lseek only records the current file offset within the kernel—it does not cause any I/O to take place.
- This offset is then used by the next read or write operation.
- lseek is defined in unistd.h

Example 1:

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

Example 2:

```
#include <unistd.h>
#include <fcntl.h>
 #include <sys/types.h>
 int main()
{
      int file=0;
      char buffer[19];
       if((file=open("testfile.txt",O_RDONLY)) < -1)
      return 1;
       if(read(file,buffer,19) != 19)
      return 1;
```

```
        printf("%s\n",buffer);
        if(lseek(file,10,SEEK_SET) < 0) return 1;
        if(read(file,buffer,19) != 19) return 1;
        printf("%s\n",buffer); return 0;
}
```

Output:

```
$ cat testfile.txt
This is a test file that will be used to
demonstrate the use of lseek.
 $ ./testfile
 This is a test file
 test file that will
```

Example 3

```
#include <fcntl.h>
int main(void)
{
if ((fd = creat("file.hole", FILE_MODE)) < 0)
if (write(fd, buf1, 10) != 10)
if (lseek(fd, 16384, SEEK_SET) == -1)
if (write(fd, buf2, 10) != 10)
exit(0);
}
```

Output:

$ **ls -l file.hole** ***check its size***

-rw-r--r-- 1 sar 16394 Nov 25 01:01 file.hole

$ **od -c file.hole** ***let's look at the actual contents***

0000000 a b c d e f g h i j \0 \0 \0 \0 \0 \0

0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

*

0040000 A B C D E F G H I J

0040012

Read(): Read the specified no of bytes from the file

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);

                                Returns: number of bytes read, 0 if end of file, -1 on error
```

If the read is successful, the number of bytes read is returned. If the end of file is encountered, 0 is returned.

read():

There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read.
  For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a terminal device. Normally, up to one line is read at a time.
- When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
- When interrupted by a signal and a partial amount of data has already been read.

The read operation starts at the file's current offset.
Before a successful return, the offset is incremented by the number of bytes actually read.

Atomic operation

Atomic operation refers to an operation that is composed of Multiple steps.

- If the operation is performed atomically, either all the steps are performed, or none is performed.
- It must not be possible for a subset of the steps to be performed.

Example:

i) open(fd, O_RDONLY | O_APPEND)

```
if (lseek(fd, 0L, 2) < 0)          /* position to EOF */
   err_sys("lseek error");
if (write(fd, buf, 100) != 100)    /* and write */
   err_sys("write error");
```

ii) open(fd, O_RDONLY | O_CREAT)

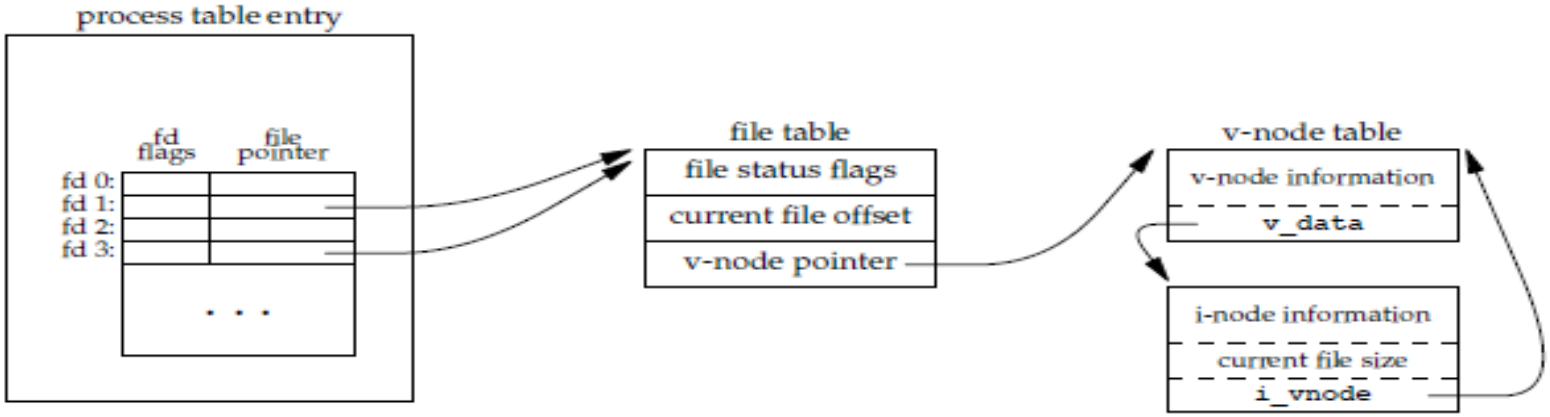
```
if ((fd = open(path, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(path, mode)) < 0)
```

dup and dup2 functions

An existing file descriptor is duplicated by either of the following functions:

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, -1 on error



- After a successful return, the old and new file descriptors may be used interchangeably.
- They refer to the same open file description and thus share file offset and file status flags;
- The two file descriptors do not share file descriptor flags
- for example, if the file offset is modified by using lseek on one of the file descriptors, the offset is also changed for the other.

dup( )

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main() {
 int fd1=0,fd2=0;
char buf[10]="abcdef";
```

```
 if((fd1=open("t12.txt",O_RDWR,0))<0)
  printf("error");
 fd2=dup(fd1);

printf("%d %d \n",fd1, fd2);
write(fd1,buf,6); return 0;
}
```

```
o/p
fd1=3, fd2=4
vi  t12.txt
abcdef
```

dup2( )

- The **dup2**() system call performs the same task as **dup**(), newfd is the copy of oldfd.
- With dup2 we specify the value of the new descriptor.
- If the file descriptor *newfd* was previously open, it is silently closed before being reused.

Note the following points:

1. If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.

2. If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then **dup2**() does nothing, and returns *newfd*.

The call

```
dup(filedes);
```

is equivalent to

```
fcntl(filedes, F_DUPFD, 0);
```

Similarly, the call

```
dup2(filedes, filedes2);
```

is equivalent to

```
close(filedes2);
fcntl(filedes, F_DUPFD, filedes2);
```

Note: dup2 is an atomic operation, whereas the alternate form involves two function calls (close and the fcntl)

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{int fd11=0,fd12=0;
char buf[10]="abcdef";
if((fd12=open("t12.txt",O_RDWR,0))<0)
printf("error");
if((dup2(fd12,fd11)<0))
printf("error");
printf("%d  %d \n",fd11, fd12);
write(fd11,buf,6);
return0;
}
```

Advantages of dup/dup2

- Used in file sharing, especially in implementing inter process communication.
- Helps in redirection of standard i/o descriptors
- Used in opening a std i/o stream functions i.e fdopen( )
- Helps the child process to inherit the property of it's parent
- Used to implement atomic inheritance.
- Used in record locking
- Used to implement pipes(popen()) in IPC

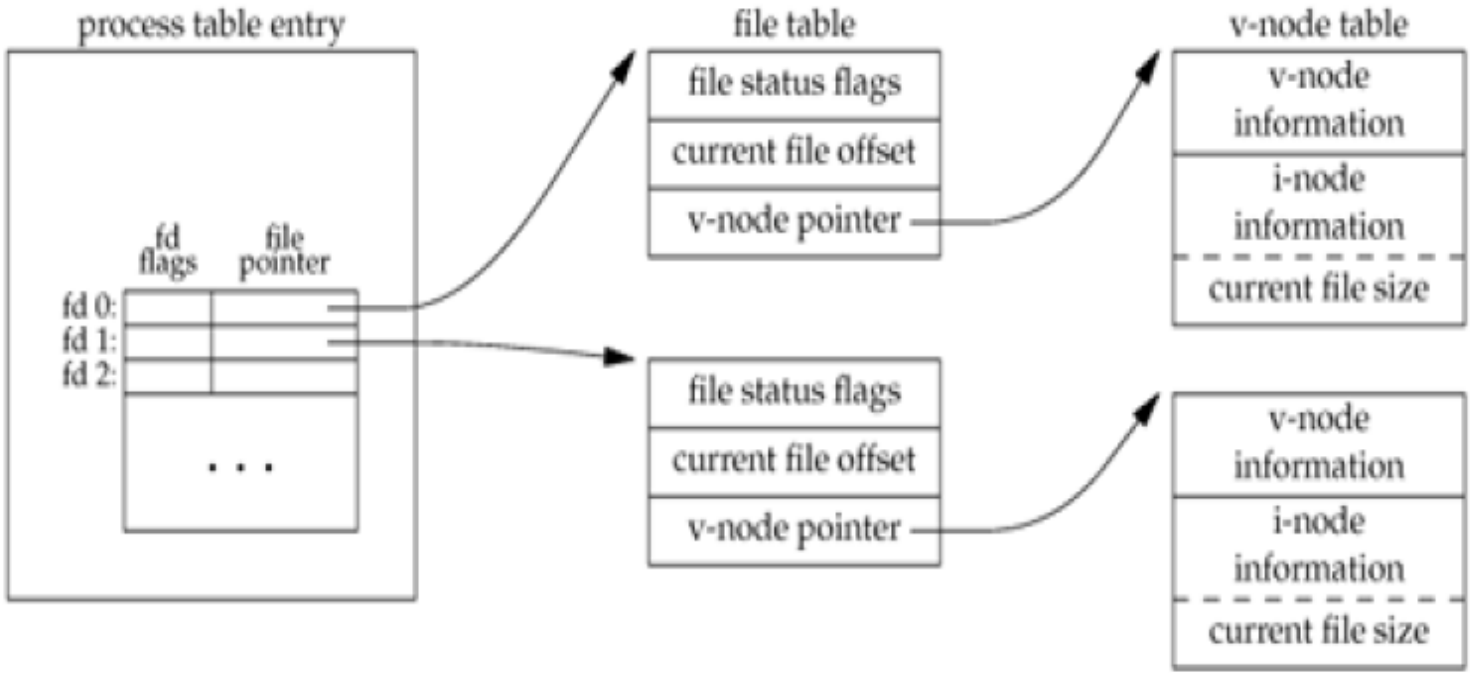
File Sharing

The kernel uses three data structures to represent an open file, and the relationships among them determine the effect one process has on another with regard to file sharing.
Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor.
Associated with each file descriptor are
a. The file descriptor flags (close-on-exec)
b. A pointer to a file table entry

File Sharing

The kernel maintains a file table for all open files.
Each file table entry contains
a. The file status flags for the file, such as read, write, append, sync, and nonblocking;
b. The current file offset
c. A pointer to the v-node table entry for the file
2. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file.
For most files, the v-node also contains the inode for the file.
This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available.
For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, and so on.

Kernel data structures for open files



fcntl( )

- fcntl() is a file control function and fcntl.h header file accompanies the fcntl()system call and provides symbolic constants (read "macro definitions") for its arguments.

- The fcntl function can change the properties of a file that is already open.
- Syntax

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );

                    Returns: depends on cmd if OK (see following), -1 on error
```

The fcntl function is used for five different purposes.

1. Duplicate an existing descriptor (*cmd* = *F_DUPFD or F_DUPFD_CLOEXEC)*
2. Get/set file descriptor flags (*cmd* = *F_GETFD* or *F_SETFD)*
3. Get/set file status flags (*cmd* = *F_GETFL* or *F_SETFL)*
4. Get/set asynchronous I/O ownership (*cmd* = *F_GETOWN or F_SETOWN)*
5. Get/set record locks (*cmd* = *F_GETLK, F_SETLK, or F_SETLKW)*

- First argument is the file descriptor
- The third argument is an integer, used for record locking

2nd argument has the following values

- F_DUPFD : Duplicate the file descriptor *fd*. *The new file descriptor is* returned as the value of the function.

- F_DUPFD_CLOEXEC : Duplicate the file descriptor and set the FD_CLOEXEC file descriptor flag associated with the new descriptor. Returns the new file descriptor.

- F_GETFD Return the file descriptor flags for *fd as the value of the function*. Currently, only one file descriptor flag is defined FD_CLOEXEC flag.

- F_SETFD Set the file descriptor flags for *fd*. *The new flag value is set from* the third argument (taken as an integer).

- F_GETFL Return the file status flags for *fd as the value of the function.*

we must first use the O_ACCMODE mask to obtain the access-mode bits and then compare the result against any of the five values.

File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete (data and attributes)

- F_SETFL: Set the file status flags to the value of the third argument (taken as an integer).
- The only flags that can be changed are O_APPEND, O_NONBLOCK, O_SYNC, O_DSYNC, O_RSYNC, O_FSYNC, and O_ASYNC.
- F_GETOWN : Get the process ID or process group ID currently receiving the SIGIO and SIGURG signals.

- F_SETOWN Set the process ID or process group ID to receive the SIGIO and SIGURG signals.
- The following four commands have special return values: F_DUPFD, F_GETFD, F_GETFL, and F_GETOWN.
- The first command returns the new file descriptor, the next two return the corresponding flags, and the final command returns a positive process ID or a negative process group ID.

Example: Program to test the file status flags for a file

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
        int val,  fd;
              if(argc!=2)
        {
                printf("Usage: %s <descriptor> \n", argv[0]);
                exit(1);
        }

             fd=open(argv[1], O_RDONLY, 0);
             printf("%d", fd);
```

```
val=fcntl(fd, F_GETFL, 0);

        switch(val & O_ACCMODE)
        {
                case O_RDONLY:
                        printf("read only");
                        break;
                case O_WRONLY:
                        printf("write only");
                        break;
                case O_RDWR:
                        printf("read write");
                        break;
                default:
                        printf("unknown access mode");
        }
```

O_ACCMODE is equal to 3 so bits 1 and 2 are on.

```
  00000000 (O_RDONLY)
& 00000011 (O_ACCMODE)
  --------
  00000000  <-- the result being compared
```

where 00000000 equals read-only so (accessMode == O_RDONLY) returns true.

The same for the others.

```
  00000001 (O_WRONLY)
& 00000011 (O_ACCMODE)
  ---------
  00000001 <-- the result being compared
```

O_WRONLY is 1 so (accessMode == O_WRONLY) is "is 1 equal to 1" which naturally returns true.

```
            if(val & O_APPEND)
                printf(", append");
        if(val & O_NONBLOCK)
                printf(", nonblocking");
        if(val & O_SYNC)
                printf(", synchronous writes");

        putchar('\n');
        exit(0);
}
```

Output

```
arl@arl-Lenovo-H30-50:~$ ./a.out 0 </dev/tty
read only

arl@arl-Lenovo-H30-50:~$ ./a.out 1 >temp

arl@arl-Lenovo-H30-50:~$ cat temp
write only

arl@arl-Lenovo-H30-50:~$ fghh ./a.out 2 2>>tr
arl@arl-Lenovo-H30-50:~$ cat tr
fghh: command not found

arl@arl-Lenovo-H30-50:~$ ./a.out 5 5<>t1.txt
read write
```

The shell clause 5<>t1.txt opens the file for reading and writing on file descriptor 5

/dev/fd

Newer systems provide a directory named /dev/fd whose entries are files named 0, 1, 2, and so on.
Opening the file /dev/fd/*n is equivalent to duplicating descriptor n, assuming that descriptor n is* open.

In the function call
fd = open("/dev/fd/0", mode); /* equivalent to fd = dup(0); */
most systems ignore the specified mode, whereas others require that it be a subset of the mode used when the referenced file (standard input, in this case) was originally opened.

The descriptors 0 and fd share the same file table entry. For example, if descriptor 0 was opened read-only, we can only read on fd.

Even if the system ignores the open mode, and the call

```
fd = open("/dev/fd/0", O_RDWR);
```

succeeds, we still can't write to fd.

We can also call creat with a /dev/fd pathname argument, as well as specifying O_CREAT in a call to open.

This allows a program that calls creat to still work if the pathname argument is /dev/fd/1.

Some systems provide the pathnames /dev/stdin, /dev/stdout, and /dev/stderr.
These pathnames are equivalent to /dev/fd/0, /dev/fd/1, and /dev/fd/2.
The main use of the /dev/fd files is from the shell. It allows programs that use pathname arguments to handle standard input and standard output in the same manner as other pathnames.
For example, The command

```
filter file2 | cat file1 - file3 | lpr
```

First, cat reads file1, next its standard input (the output of the filter program on file2), then file3.
If /dev/fd is supported, the special handling of - can be removed from cat, and we can enter

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

O_sync( ) :The sync function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.
The function sync is normally called periodically (usually every 30 seconds) from a system daemon, often called update. This guarantees regular flushing of the kernel's block buffers.

O_async( ):Enable signal-driven I/O: generate a signal when input or output becomes possible on this file descriptor. This feature is available only for terminals, pseudo terminals, sockets, and pipes and FIFOs

what does read() do when no bytes are available to read?

Blocking Read

By default, read() waits until at least one byte is available to return to the application; this default is called “blocking” mode.

Non Blocking read()

In this read() on a slow file will return immediately, even if no bytes are available.

what does write() do when no bytes are available to write?

Blocking write:

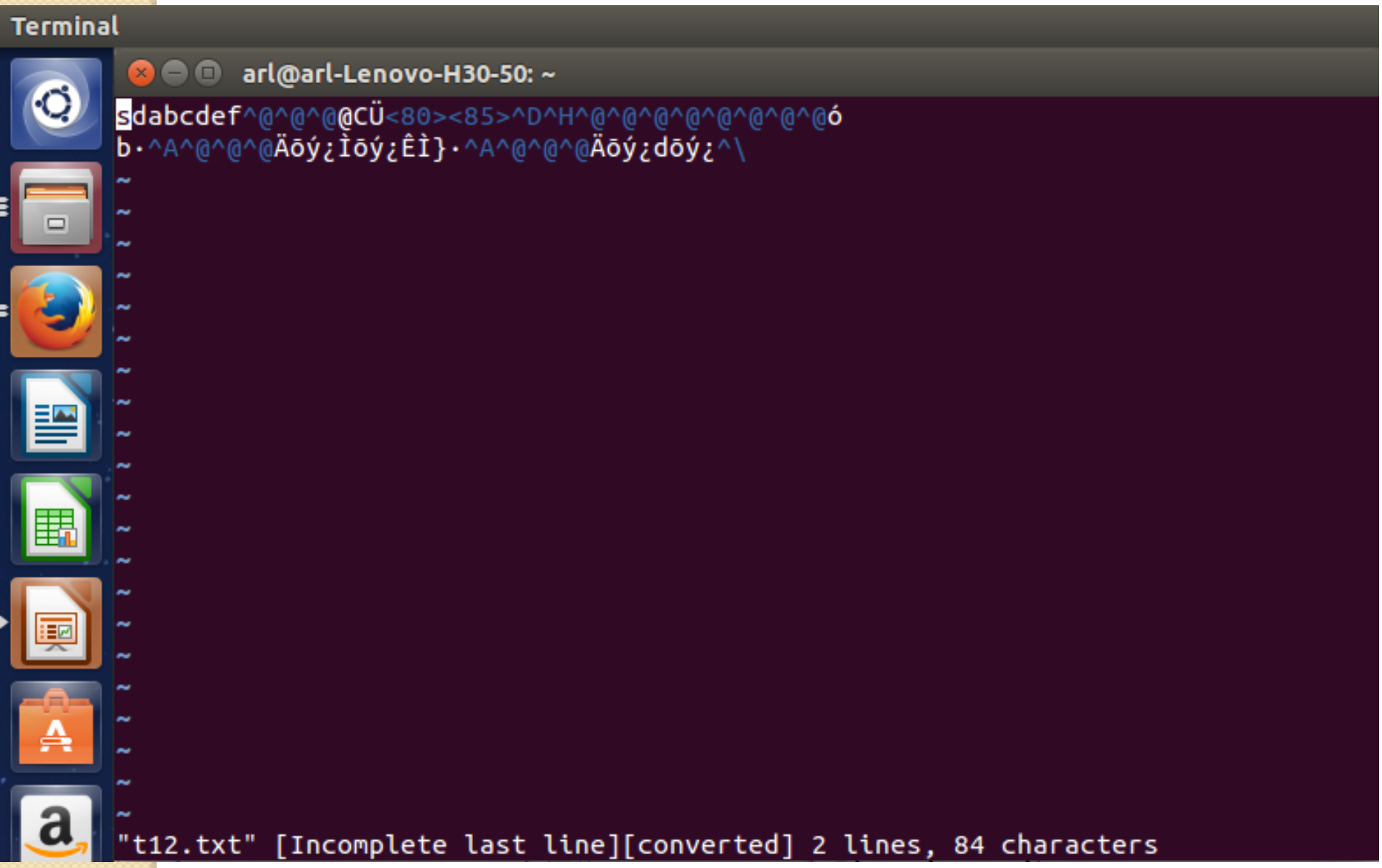
Normally, write() will block until it has written all of the data to the file.

Non blocking write:

If that particular file descriptor (or file structure) is in non-blocking mode, however, write() will write as much data into the file as it can, and then return. This means that it will store as few as 0 bytes, or as much as the full number that was requested.

Ex: write(fd, buf,25)

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
        int fd11=0,fd12=0;
        char buf[10]="sdabcdef";
        if((fd12=open("t12.txt",O_RDWR,0))<0)
            printf("error");

        write(fd12,buf,60);
        return 0;
}
```


Terminal
arl@arl-Lenovo-H30-50: ~
sdabcdef^@^@^@@CÜ<80><85>^D^H^@^@^@^@^@^@^@^@ó
b·^A^@^@^@Äōý¿Ìōý¿ÊÌ}·^A^@^@^@Äōý¿dōý¿^\
"t12.txt" [Incomplete last line][converted] 2 lines, 84 characters

Today's task: Reference http://www.linux-mag.com/id/308/

1) Execute the program to check the file status flags for a given file

2) open a file descriptor in non blocking mode

Note:

By default a file will be opened in blocking mode but using fcntl change blocking to non blocking mode

You can open a file descriptor as non-blocking by adding a flag to the open(), and you can change a file descriptor between blocking and non-blocking via the fcntl() call.

Files and directories

- **This chapter mainly deals with additional features of the file system and the properties of a file.**
  - **We'll start with the stat functions and go through each member of the stat structure, looking at all the attributes of a file.**
  - **In this we'll describe each of the functions that modify these attributes: change the owner, change the permissions, and so on.**
  - **We'll also look in more detail at the structure of a UNIX file system and symbolic links.**
  - **We finish this chapter with the functions that operate on directories, and we develop a function that descends through a directory hierarchy.**

stat/fstat/lstat functions

All these functions returns the information about a file.

- Given a pathname, the stat function returns a structure of information about the named file.
- The fstat function obtains information about the file that is already open on the descriptor fd.
- The lstat function is similar to stat, but when the named file is a symbolic link, lstat returns information about the symbolic link, not the file referenced by the symbolic link.

Syntax

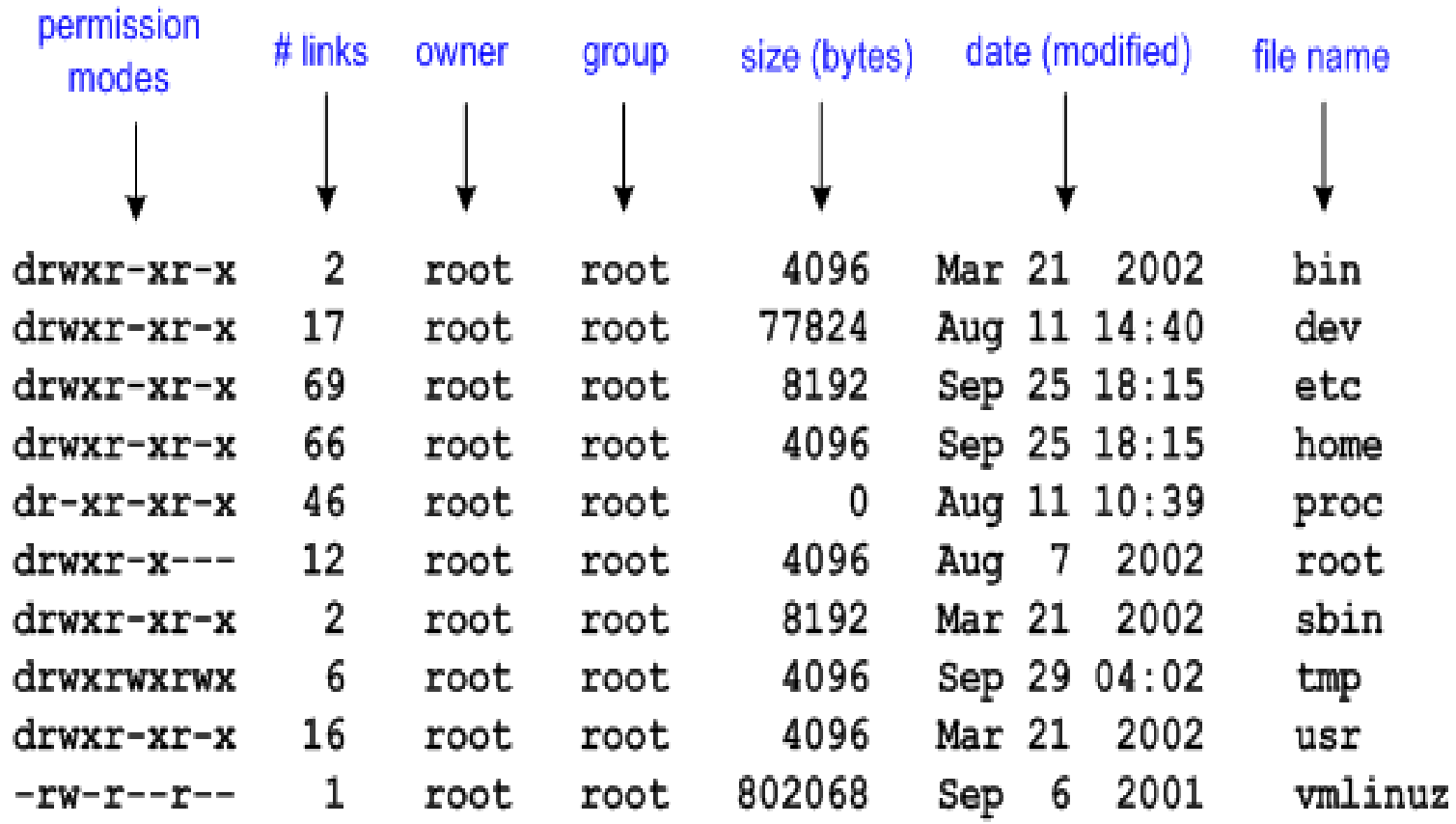
```
#include <sys/stat.h>

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int fd, struct stat *buf);

int lstat(const char *restrict pathname, struct stat *restrict buf);

int fstatat(int fd, const char *restrict pathname,
            struct stat *restrict buf, int flag);
```

All four return: 0 if OK, –1 on error

- The buf argument is a pointer to a structure that we must supply.
- The definition of the structure can differ among implementations, but it could look like

```
struct stat {
mode_t   st_mode; /* file type & mode (permissions) */
ino_t   st_ino; /* i-node number (serial number) */
dev_t   st_dev; /* device number (file system) */
dev_t   st_rdev; /* device number for special files */
nlink_t   st_nlink; /* number of links */
uid_t   st_uid; /* user ID of owner */
gid_t   st_gid; /* group ID of owner */
off_t   st_size; /* size in bytes, for regular files */
time_t   st_atime; /* time of last access */
time_t    st_mtime; /* time of last modification */
time_t   st_ctime; /* time of last file status change */
blksize_t st_blksize; /* best I/O block size */
blkcnt_t st_blocks; /* number of disk blocks allocated */
};
```

Attributes of a file (ls –l)



Note: This listing is for example purposes and not necessarily an accurate or complete representation of the root (/) directory.

From right to left, the attribute fields are:

- **file name**: the name associated with the file (recall, this can be any type of file)
- **modification date**: the date the file was last modified, i.e. a "time-stamp". If the file has not been modified within the last year (or six months for Linux), the year of last modification is displayed.
- **size**: the size of the file in bytes (i.e. characters). This is the number of characters in the file, not necessarily the size on disk, since files are written to disk in 1024 byte blocks. Note also if the file is a directory, this is the size of the structure needed to manage the directory hierarchy.
- **group**: associated group for the file
- **owner**: the owner of the file
- **number of links**: the number of other links associated with this file
- **permission modes**: the permissions assigned to the file for the owner, the group and all others.

File types

- Two different types of files exist:
  - regular files
  - directories.

- Most files on a UNIX system are either regular files or directories, but there are additional types of files. The types are

1. Regular file. The most common type of file, which contains data of some form.

2. Directory file.

- A file that contains the names of other files and pointers to information on these files.
- Any process that has read permission for a directory file can read the contents of the directory,
- Only the kernel can write directly to a directory file.

3. Block special file. A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.

4. Character special file. A type of file providing unbuffered I/O access in variable-sized units to devices.
Note: All devices on a system are either block special files or character special files.

5. FIFO. A type of file used for communication between processes. It’s sometimes called a named pipe.

6. Socket. A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host.

7. Symbolic link. A type of file that points to another file.

- The type of a file is encoded in the st_mode member of the stat structure. We can determine the file type with the macros shown in Figure.

- The argument to each of these macros is the st_mode member from the stat structure.

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

Program to print the type of a file

```
#include <sys/stat.h>
int main(int argc, char *argv[])
{
        int i;
        struct stat buf;
        char *ptr;
        for (i = 1; i < argc; i++) {
                printf("%s: ", argv[i]);
                if (lstat(argv[i], &buf) < 0)
                 {
                        err_ret("lstat error");
                        continue;
                }
                if (S_ISREG(buf.st_mode))
                ptr = "regular";
```

```
else if (S_ISDIR(buf.st_mode))
        ptr = "directory";
else if (S_ISCHR(buf.st_mode))
        ptr = "character special";
else if (S_ISBLK(buf.st_mode))
        ptr = "block special";
else if (S_ISFIFO(buf.st_mode))
        ptr = "fifo";
else if (S_ISLNK(buf.st_mode))
        ptr = "symbolic link";
else if (S_ISSOCK(buf.st_mode))
        ptr = "socket";
else
        ptr = "** unknown mode **";
printf("%s\n", ptr);
}
exit(0);
}
```

Sample output

```
$ ./a.out /etc/passwd /etc /dev/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: block special
/dev/cdrom: symbolic link
```

User Identification

- **The *user ID from our entry in the password file is a numeric value that identifies us to* the system.**
- **This user ID is assigned by the system administrator when our login name is assigned, and we cannot change it.**
- **The user ID is normally assigned to be unique for every user.**
- **We call the user whose user ID is 0 either *root or the super user. The entry in the* password file normally has a login name of root, and we refer to the special privileges of this user as super user privileges.**

Group user

- Our entry in the password file also specifies our numeric *group ID. This, too, is assigned* by the system administrator when our login name is assigned.

Typically, the password file contains multiple entries that specify the same group ID. Groups are normally used to collect users together into projects or departments.

This allows the sharing of resources, such as files, among members of the same group.

Supplementary groups

In addition to the group ID specified in the password file for a login name, most versions of the UNIX System allow a user to belong to other groups.
This practice started with 4.2BSD, which allowed a user to belong to up to 16 additional groups

```
int main()
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Every process has six or more IDs associated with it.

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by exec functions

The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in.

The effective user ID, effective group ID, and supplementary group IDs determine our file access permissions

The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID, respectively, when a program is executed.

A file can have 3 permissions Read, write and Execute
A process can have 6 id's associated with it.
Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.

Every file has an owner and a group owner.
The owner is specified by the st_uid member of the stat structure; the group owner, by the st_gid member.

- When we execute a program file, the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID.

- However, we can also set a special flag in the file's mode word (st_mode) that says, "When this file is executed, ***set the effective user ID of the process to be the owner of the file (st_uid)."***

- ***Similarly, we can set another bit in the file's mode word that causes the effective group ID to be the group owner of the file (st_gid).***

- These two bits in the file's mode word are called the *set-user-ID bit and the set-group-ID bit.*

- For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges.
- This happens regardless of the real user ID of the process that executes the file.
- As an example, the UNIX System program that allows anyone to change his or her password, passwd(1), is a set-user-ID program.
- This is required so that the program can write the new password to the password file, typically either /etc/passwd or /etc/shadow, files that should be writable only by the superuser.
- Because a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully.

File access permissions

- The st_mode value also encodes the access permission bits for the file – could be any of the file types(regular files, directories, character special files, and so on).
- There are nine permission bits for each file, divided into three categories.

The nine file access permission bits, from <sys/stat.h>

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	userexecute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	groupexecute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	otherexecute

- The first rule is that *whenever we want to open any type of file by name, we must have execute* permission in each directory mentioned in the name, including the current directory, if it is implied.
- This is why the execute permission bit for a directory is often called the search bit.
- For example, to open the file /usr/include/stdio.h, we need execute permission in the directory /, execute permission in the directory /usr, and execute permission in the directory /usr/include.
- We then need appropriate permission for the file itself, depending on how we're trying to open it: read-only, readwrite, and so on.

- If the current directory is /usr/include, then we need execute permission in the current directory to open the file stdio.h. This is an example of the current directory being implied, not specifically mentioned.
- It is identical to our opening the file ./stdio.h.
- Note that read permission for a directory and execute permission for a directory mean different things.
- Read permission lets us read the directory, obtaining a list of all the filenames in the directory.
- Execute permission lets us pass through the directory when it is a component of a pathname that we are trying to access. (We need to search the directory to look for a specific filename.)

- The read permission for a file determines whether we can open an existing file for reading: the O_RDONLY and O_RDWR flags for the open function.
- The write permission for a file determines whether we can open an existing file for writing: the O_WRONLY and O_RDWR flags for the open function.
- We must have write permission for a file to specify the O_TRUNC flag in the open function.
- We cannot create a new file in a directory unless we have write permission and execute permission in the directory.
- To delete an existing file, we need write permission and execute permission in the directory containing the file.
- We do not need read permission or write permission for the file itself.
- Execute permission for a file must be on if we want to execute the file using any of the six exec functions. The file also has to be a regular file.

- The file access tests that the kernel performs each time a process opens, creates, or deletes a file depend on the owners of the file (st_uid and st_gid), the effective IDs of the process (effective user ID and effective group ID), and the supplementary group IDs of the process, if supported.
- The two owner IDs are properties of the file, whereas the two effective IDs and the supplementary group IDs are properties of the process.

The tests performed by the kernel are as follows

- If the effective user ID of the process is 0 (the superuser), access is allowed.
- This gives the superuser free rein throughout the entire file system.

- If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the
- file), access is allowed if the appropriate user access permission bit is set. \

- Otherwise, permission is denied.
- By *appropriate access permission bit, we mean that if the process is opening the file* for reading, the user-read bit must be on. If the process is opening the file for writing, the userwrite bit must be on. If the process is executing the file, the user-execute bit must be on.

- If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.

- If the appropriate other access permission bit is set, access is allowed. Otherwise, permission isdenied.

- These four steps are tried in sequence.
- Note that if the process owns the file (step 2), access is granted or denied based only on the user access permissions; the group permissions are never looked at.
- Similarly, if the process does not own the file, but belongs to an appropriate group, access is granted or denied based only on the group access permissions, other permissions are not looked at

Ownership of New Files and Directories

- The user ID of a new file is set to the effective user ID of the process.
- The group ID of a new file can be the effective group ID of the process.
- The group ID of a new file can be the group ID of the directory in which the file is being created.

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char **argv)
{
    if(argc != 2)
        return 1;

    struct stat fileStat;
    if(stat(argv[1],&fileStat) < 0)
        return 1;

    printf("Information for %s\n",argv[1]);
    printf("---------------------------\n");
    printf("File Size: \t\t%d bytes\n",fileStat.st_size);
    printf("Number of Links: \t%d\n",fileStat.st_nlink);
    printf("File inode: \t\t%d\n",fileStat.st_ino);

    printf("File Permissions: \t");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n\n");

    printf("The file %s a symbolic link\n", (S_ISLNK(fileStat.st_mode)) ? "is" : "is not");

    return 0;
}
```

Note:

1. Only a super user process can change the real user ID. Normally, the real user ID is set by the login(1) program when we log in and never changes. Because login is a super user process, it sets all three user IDs when it calls setuid.

2. The effective user ID is set by the exec functions only if the set-user-ID bit is set for the program file.

access()

- determine accessibility of a file

Syntax:

int access(const char **path*, int *mode*);

- The value of *mode* is either the bitwise-inclusive OR of the access permissions to be checked (R_OK, W_OK, X_OK) or F_OK

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission

- When we open a file, the kernel performs its access tests based on the effective user and group IDs.

- Sometimes, however, a process wants to test accessibility based on the real user and group IDs.

- This is useful when a process is running as someone else, using either the set-user-ID or the set-group-ID feature.

- The check is done using the calling process's *real* UID and GID, rather than the effective IDs as is done when actually attempting an operation

- Example…

```
#include <fcntl.h>
Int main(int argc, char *argv[])
{
if (argc != 2)
err_quit("usage: a.out <pathname>");
if (access(argv[1], R_OK) < 0)
err_ret("access error for %s", argv[1]);
else
printf("read access OK\n");

if (open(argv[1], O_RDONLY) < 0)
err_ret("open error for %s", argv[1]);
else
printf("open for reading OK\n");
exit(0);
}
```

Results

$ **ls -l a.out**
-rwxrwxr-x 1 sar 15945 Nov 30 12:10 a.out

$ **./a.out a.out**
read access OK
open for reading OK

$ **ls -l /etc/uucp/Systems**
-r-------- 1 root 1315 Jul 17 2002 /etc/
shadow

$ ./a.out /etc/uucp/Systems
access error for /etc/shadow: Permission denied
open error for /etc/shadow: Permission denied

$ su **// *become superuser***
Password: **// *enter superuser password***

 # chown uucp a.out **// *change file's user ID to uucp***
 # chmod u+s a.out **// *and turn on set-user-ID bit***

 # ls -l a.out **// *check owner and SUID bit***
-rwsrwxr-x 1 uucp 15945 Nov 30 12:10 a.out

 # exit **// *go back to normal user***
$ ./a.out /etc/uucp/Systems
access error for /etc/shadow: Permission denied

chown: change the ownership

A chown command is used to change file owner and group information.

A chmod command is used to change file access permissions such as read, write, and access.

example..

chown owner-user **file**

chown owner-user:owner-group **file**

```
# ls -l demo.txt
-rw-r--r-- 1 root root 0 Aug 31 05:48 demo.txt

# chown vivek demo.txt

# ls -l demo.txt
-rw-r--r-- 1 vivek root 0 Aug 31 05:48 demo.txt
```

(Same way, try to change group id)

umask ( ) :

- The umask function sets the file mode creation mask for the process and returns the previous value.
- Syntax:

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
                    Returns: previous file mode creation mask
```

- The cmask argument is formed as the bitwise OR of any of the nine constants from S_IRUSR, S_IWUSR, and so on.

The file mode creation mask is used whenever the process creates a new file or a new directory.

```
#include <fcntl.h>
Int main(void)
{
umask(0);
if (creat("foo“, S_IRUSR|S_IWUSR|S_IRGRP|
S_IWGRP|S_IROTH |S_IWOTH) < 0)
printf("creat error for foo");
umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

if (creat("bar“, S_IRUSR|S_IWUSR|S_IRGRP| S_IWGRP
|S_IROTH |S_IWOTH) < 0)
printf("creat error for bar");
exit(0);
}
```

- Setting the **umask** to **0 means** that newly created files or directories created will have read privileges initially revoked.
- umask value is usually set once, on login, by the shell's start-up file, and never changed. (The UMASK value can be set in /etc/profile for all the new users but only by root)
- Users can set the umask value to control the default permissions on the files they create.
- This value is expressed in octal, with one bit representing one permission to be masked off.

The umask file access permission bits

Mask bit	Meaning
0400	user-read
0200	user-write
0100	user-execute
0040	group-read
0020	group-write
0010	group-execute
0004	other-read
0002	other-write
0001	other-execute

Octal digit in umask command	Permissions disabled during file creation
0	none, all original permissions will remain
1	execute permission is disabled
2	write permission is disabled
3	write and execute permission are disabled
4	read permission is disabled
5	read and execute permissions are disabled
6	read and write permission are disabled
7	all permissions are disabled

Note: umask values are completely reverse of chmod values

Permissions can be denied by setting the corresponding bits.

Some common umask values are

1. 002 to prevent others from writing your files.

2. 022 to prevent group members and others from writing your files

3. 027 to prevent group members from writing your files and others from reading, writing, or executing your files.

chmod()

- The chmod, fchmod, and fchmodat functions allow us to change the file access permissions for an existing file.

To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have super user permissions.

The *mode is specified as the bitwise OR of the constants*

mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

```c
#include "apue.h"

int
main(void)
{
    struct stat     statbuf;

    /* turn on set-group-ID and turn off group-execute */

    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */

    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");

    exit(0);
}
```

Before the execution

```
$ ls -l foo bar
-rw-------  1 sar          0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar          0 Dec  7 21:20 foo
```

After the execution

```
$ ls -l foo bar
-rw-r--r--  1 sar          0 Dec  7 21:20 bar
-rw-rwSrw-  1 sar          0 Dec  7 21:20 foo
```

In this example, we have set the permissions of the file bar to an absolute value, regardless of the current permission bits.

For bar explicitly turned on the set-group-ID bit and turned off the group-execute bit.

The chmod functions automatically clear two of the permission bits under the following conditions:

1. On systems, such as Solaris, that place special meaning on the sticky bit when used with regular files, if we try to set the sticky bit (S_ISVTX) on a regular file and do not have super user privileges, the sticky bit in the *mode is automatically* turned off.

2. The group ID of a newly created file might potentially be a group that the calling process does not belong to.

Sticky bit

- A Sticky bit is a permission bit that is set on a file or a directory that lets only the owner of the file/directory or the root user to delete or rename the file. No other user is given privileges to delete the file created by some other user.

History:

The sticky bit is not a new concept. In fact, it was first introduced in in 1974 in the Unix operating system.

The purpose of sticky bit back then was different.
It was introduced to minimize the time delay introduced every time when a program is executed.

When a program is executed, it takes time to load the program into memory before the user can actually start using it.
If a program, for example an editor is used frequently by users the the start-up time delay was an overhead back then.

To improve this time delay, the sticky bit was introduced. The OS checked that if sticky bit on an executable is ON, then the text segment of the executable was kept in the swap space.

This made it easy to load back the executable into RAM when the program was run again thus minimizing the time delay.

Stick bit to a directory If the bit is set for a directory, a file in the directory can be removed or renamed only if the user has write permission for the directory and meets one of the following criteria:

- Owns the file
- Owns the directory
- Is the super user

Ex: The directories /tmp and /var/tmp (try to create, rename and delete a directory)

Chown, fchown and lchown

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);

int fchownat(int fd, const char *pathname, uid_t owner, gid_t group,
             int flag);

int lchown(const char *pathname, uid_t owner, gid_t group);
```

All four return: 0 if OK, -1 on error

File size

- The st_size member of the stat structure contains the size of the file in bytes.

This field is meaningful only for regular files, directories, and symbolic links.

Most contemporary UNIX systems provide the fields st_blksize and st_blocks as a file size. For a symbolic link, the file size is the number of bytes in the filename. Ex: usr/lib:

```
lrwxrwxrwx  1 root          7 Sep 25 07:14 lib -> usr/lib
```

Unix Filesystem

Unix divides physical disks into logical disks called *partitions*.

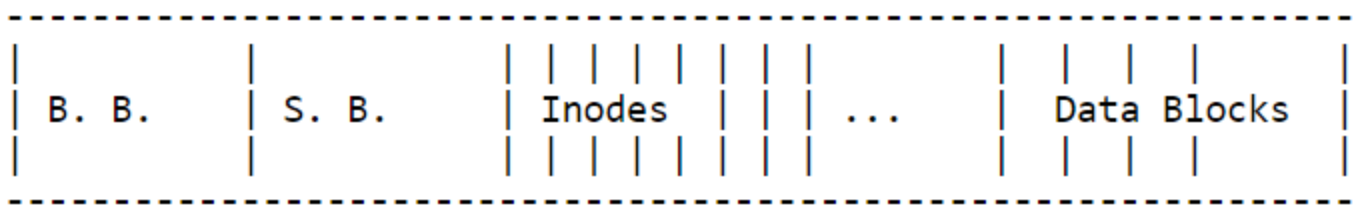
Each partition is a standalone file system.

Each filesystem contains:

1. a ***boot block*** located in the first few sectors of a file system. The boot block contains the initial bootstrap program used to load the operating system.
2. a ***super block*** describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, the inode number of the root directory, magic number, etc.
3. a linear array of ***inodes*** (short for ``index nodes''). There is a one to one mapping of files to inodes and vice versa. An inode is identified by its ``inode number'', which contains the information needed to find the inode itself on the disk

Users think of files in terms of file names, Unix thinks of files in terms of inodes.

4. ***data blocks*** blocks containing the actual contents of files



An inode is the ``handle'' to a file and contains the following information:

- file ownership indication
- file type (e.g., regular, directory, special device, pipes, etc.)
- file access permissions. May have setuid (sticky) bit set.
- time of last access, and modification
- number of links (aliases) to the file
- pointers to the data blocks for the file
- size of the file in bytes (for regular files), major and minor device numbers for special devices.

An integral number of inodes fits in a single data block.

Information the inode does not contain: pathname of file (short or full)

soft / hard links to a file

- To make links between files you use ln command. To copy a file you use cp command.

  The main advantage is **access permission**. If you want to change the permissions for a file called foo, you only have to do it on the original(if you link).

  With copies you have to find all of the copies and change permission on each of the copies.

Other Advantages

a. Create a shortcut for example /webroot/ home/httpd
   can be soft linked at /home/httpd
b. Ease of management
c. Save disk space etc

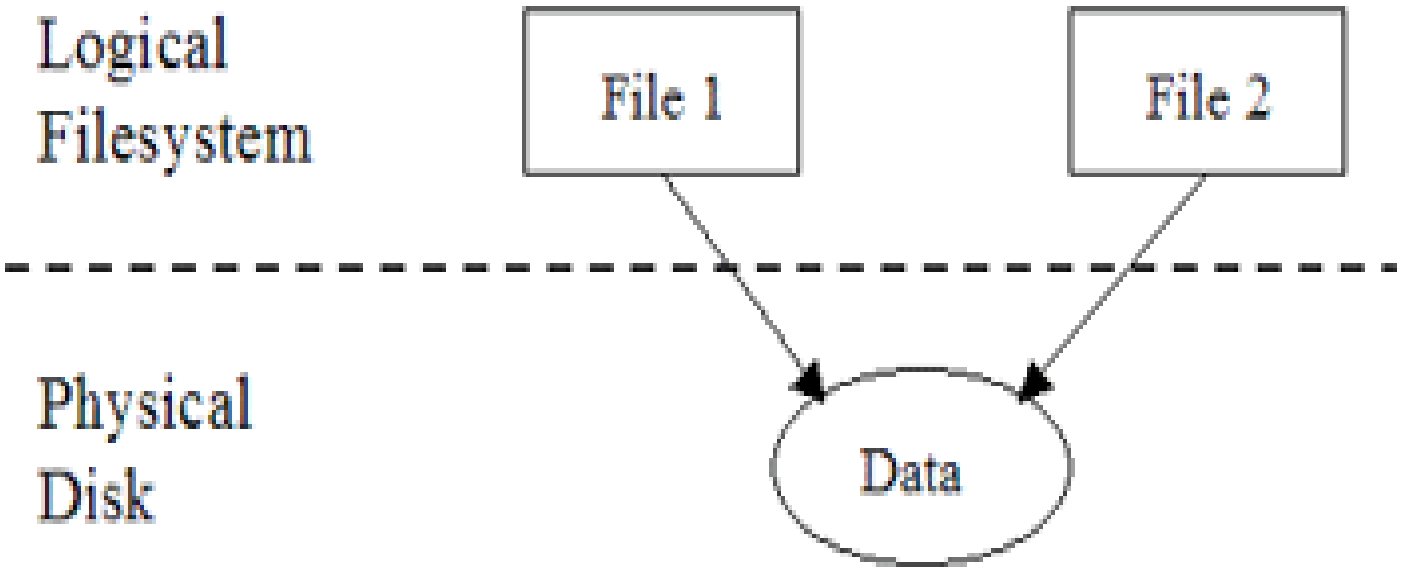
There are two types of links

Hard link
Soft link

Hard link and Soft link

- Hard links: two filenames pointing to the same inode and the same data blocks.
- Hardlinks must be on the same filesystem, softlinks can cross filesystems.
- Hardlinked files stay linked even if you move either of them (unless you move one to another file system triggering the copy-and-delete mechanism).
- Hardlinked files are co-equal, while the original is special in softlinks, and deleting the original deletes the data. The data does not go away until *all* hardlinks are deleted.

Hard Linked Files



Example

$ touch blah1
$ echo “hello good morning" > blah1
$ ln blah1 blah1-hard *// ln is a command to create hard link*
$ ls -l blah1 blah1-hard

$ mv blah1 blah1-new *// Changing the name of blah1*

does not matter:

// **blah1-hard points to the inode, the contents, of the file - that wasn't changed.**

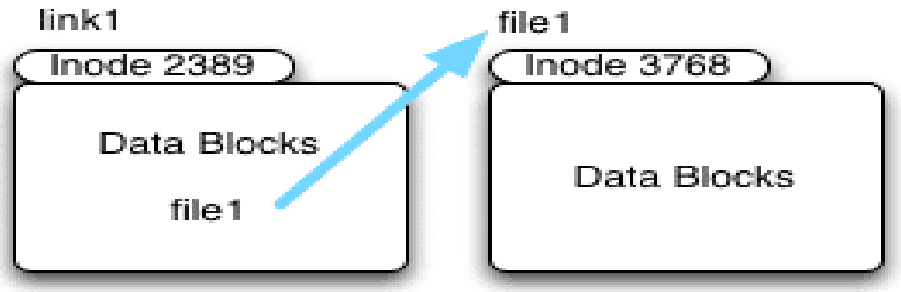
$ cat blah1-hard
hello good morning

Limitations of hard link

Hard links normally require that the link and the file reside in the same file system.

Only the super user can create a hard link to a directory (when supported by the underlying file system).

Soft-link(Symbolic link)



soft link - path of file which it points to

A symbolic link is an indirect pointer to a file, There are no file system limitations on a symbolic link and what it points to.

Anyone can create a symbolic link to a directory. Symbolic links are typically used to ‘‘move’’ a file or an entire directory hierarchy to another location on a system.

Symbolic or soft links: there are two inodes: one contains the actual data of the file, the other serves as a pointer to the first file, containing only the first file's name.

Symbolic links can be used across file systems, even on different computers.

They are used on almost every Unix system for their flexibility.

ln –s is used to create symbolic link.

Example:

$ echo "Dog" > blah2

ln -s blah2 blah2-soft

ls –l

blah2

blah2-soft -> blah2

$ mv blah2 blah2-new

$ ls blah2-soft blah2-soft

$ cat blah2-soft

cat: blah2-soft: No such file or directory

Note: The contents of the file could not be found because the soft link points to the name, that was changed, and not to the contents. if blah2 is deleted, blah2-soft is just a link to a non-existing file.

Observe the following during hard and soft link

- Hard-link
  ln old.txt new.txt(check file sizes of both old and new)
  Check the inode no of both files
  Check the link count of old.txt file
  Soft-link
  ln –s t1.txt t2.txt(check file sizes of both t1 and t2 files)
  Check the inode no of both files
  Check the link count of t1.txt file

Programs on links

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char* argv[]){
        if(argc==3){
                printf("\n%s %s", argv[1], argv[2]);
                if((link(argv[1],argv[2]))==0)
                        printf("\nHard link created");
                else
                        printf("\nError on hard link parameter");
        }
        else if(argc==4){
                if((symlink(argv[2],argv[3]))==0)
                        printf("\nSoft link created");
                else
                        printf("\nError on soft link parameter");
        }

        return 0;

}
```

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
#define SIZE 4096
int main(int argc, char* argv[]){
        int len;
        char buf[SIZE];
        len = readlink(argv[1], buf, SIZE);
        printf("\nLength= %d", len);
        printf("\nUnlinking file");
        if(unlink(argv[1])==0)
                printf("\nLink removed");
        else
                printf("\nError");
        return 0;
}
```

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char* argv[]){
        if((link(argv[1],argv[2]))<0)
                printf("\nLink error");
        else
                printf("\nLink established from %s to %s ", argv[1], argv[2]);
        return 0;
}
```

unlink

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

Try this

```
$ ln -s /no/such/file myfile              create a symbolic link
$ ls myfile
myfile                                    ls says it's there
$ cat myfile                              so we try to look at it
cat: myfile: No such file or directory
$ ls -l myfile                            try -l option
lrwxrwxrwx  1 sar       13 Jan 22 00:26 myfile -> /no/such/file
```

l in the ls –l command indicates soft link

Other link functions

unlink(): is used to remove the link on a file and on a directory (can’t create hard link on a directory)

```
arl@arl-Lenovo-H30-50:~/link$ cd cp
arl@arl-Lenovo-H30-50:~/link/cp$ ls
cp2
arl@arl-Lenovo-H30-50:~/link/cp$ cd cp2
arl@arl-Lenovo-H30-50:~/link/cp/cp2$ ls
ff
arl@arl-Lenovo-H30-50:~/link/cp/cp2$ cd ..
arl@arl-Lenovo-H30-50:~/link/cp$ cd ..
arl@arl-Lenovo-H30-50:~/link$ cc unlink1.c
arl@arl-Lenovo-H30-50:~/link$ ./a.out cp1
Length 2
Unlinking file
Link Removedarl@arl-Lenovo-H30-50:~/link$ cd cp
arl@arl-Lenovo-H30-50:~/link/cp$ cd ..
arl@arl-Lenovo-H30-50:~/link$ ls -l cp1
ls: cannot access cp1: No such file or directory
arl@arl-Lenovo-H30-50:~/link$ ln -s cp cp1 cp21 cp 31
ln: target ‘31’ is not a directory
arl@arl-Lenovo-H30-50:~/link$ ln -s cp cp1
arl@arl-Lenovo-H30-50:~/link$ ln -s cp cp12
arl@arl-Lenovo-H30-50:~/link$ ln -s cp cp13
arl@arl-Lenovo-H30-50:~/link$ ./a.out cp
Length -1
Unlinking file
```

- readlink(): is used to read the name in the link.

rename(const char *oldname, const char *newname): To rename a file or a directory. (check what happens if new newname already exists)

File times

Three time fields are maintained for each file in a stat structure

Field	Description	Example	ls(1) option
st_atim	last-access time of file data	read	-u
st_mtim	last-modification time of file data	write	default
st_ctim	last-change time of i-node status	chmod, chown	-c

- The modification time indicates when the contents of the file were last modified.
- The changed-status time indicates when the i-node of the file was last modified.

utime( ): changes the access and modification times of the inode specified by *filename* to
the *actime* and *modtime* fields of buf respectively.

Syntax:
int utime(const char *_filename_, const struct utimbuf *_buf_);

The structure used by this function is:

```
struct utimbuf
{
time_t actime; /* access time */
 time_t modtime; /* modification time */
 };
```

The privileges required to execute these functions depend on the value of the *times* argument.

- If *times is a null pointer , the access time and modification time are both set to the current time. To do this either* the effective user ID of the process must equal the owner ID of the file or the process must have write permission for the file.

- If *times is a non-null pointer the access time and the modification time are set to the values in the structure pointed to by times*.

```
 #include <fcntl.h>
int main(int argc, char *argv[])
{
int i, fd;
struct stat statbuf;
struct utimbuf timebuf;
for (i = 1; i < argc; i++) {
if (stat(argv[i], &statbuf) < 0) {          /* fetch current
times */
printf("%s: stat error", argv[i]);
     continue;
}
if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /*
truncate */
printf("%s: open error", argv[i]);
 continue;
}
```

```
timebuf.actime=statbuf.st_atime;
timebuf.modtime= stat.st_mtime;
if(utime(argv[i], &timebuf)<0){
printf(“utime error”);
continue;
}
}
exit(0);
}
```

Before execution

$ ls -l changemod times // look at sizes and last-modification times

-rwxrwxr-x 1 sar 15019 Nov 18 18:53 changemod

-rwxrwxr-x 1 sar 16172 Nov 19 20:05 times

$ ./a.out test-file

After the execution

$ ls -l changemod times

-rwxrwxr-x 1 sar 15019 Nov 18 18:53 changemod

-rwxrwxr-x 1 sar 16172 Nov 19 20:05 times

Explanation: Initially statbuf object holds the last access and modification time of a file, then truncation happens(modification of a file), so the modification time changes but after this we are copying statbuf values into timebuf strcture. At the end calling utime function reads the values from the timebuf and set that values into the members of the stat structure.

That’s why we are getting the same time.

Try other directory related function (refer text book)

1. Creating a directory using
(int mkdir(const char **pathname, mode_t mode);*

2. Removing a directory using
int rmdir(const char **pathname);*

3. Renaming a directory using
int rename(const char *oldname, const char *newname);

Reading a directory

- Program to implement ls command

```
#include <dirent.h>
Int main(int argc, char *argv[])
{
DIR *dp; struct dirent *dirp;
if (argc != 2) printf("usage: ls directory_name");
if ((dp = opendir(argv[1])) == NULL)
printf("can’t open %s", argv[1]);
while ((dirp = readdir(dp)) != NULL)
printf("%s\n", dirp->d_name);
closedir(dp);  exit(0);  }
```

Printing special device-file numbers

```
#include "apue.h"
#ifdef SOLARIS
#include <sys/mkdev.h>
#endif

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (stat(argv[i], &buf) < 0) {
            err_ret("stat error");
            continue;
        }

        printf("dev = %d/%d", major(buf.st_dev),  minor(buf.st_dev));
```

```
        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                    (S_ISCHR(buf.st_mode)) ? "character" : "block",
                    major(buf.st_rdev), minor(buf.st_rdev));
        }
        printf("\n");
    }

    exit(0);
}
```