UNIT-3: Linux Files

File concept:

A file, can be defined as a container which is used to store information or it can also be termed as a sequence of characters.

- Kernal it self is treated as a file. files in LINUX don't have an eof character.

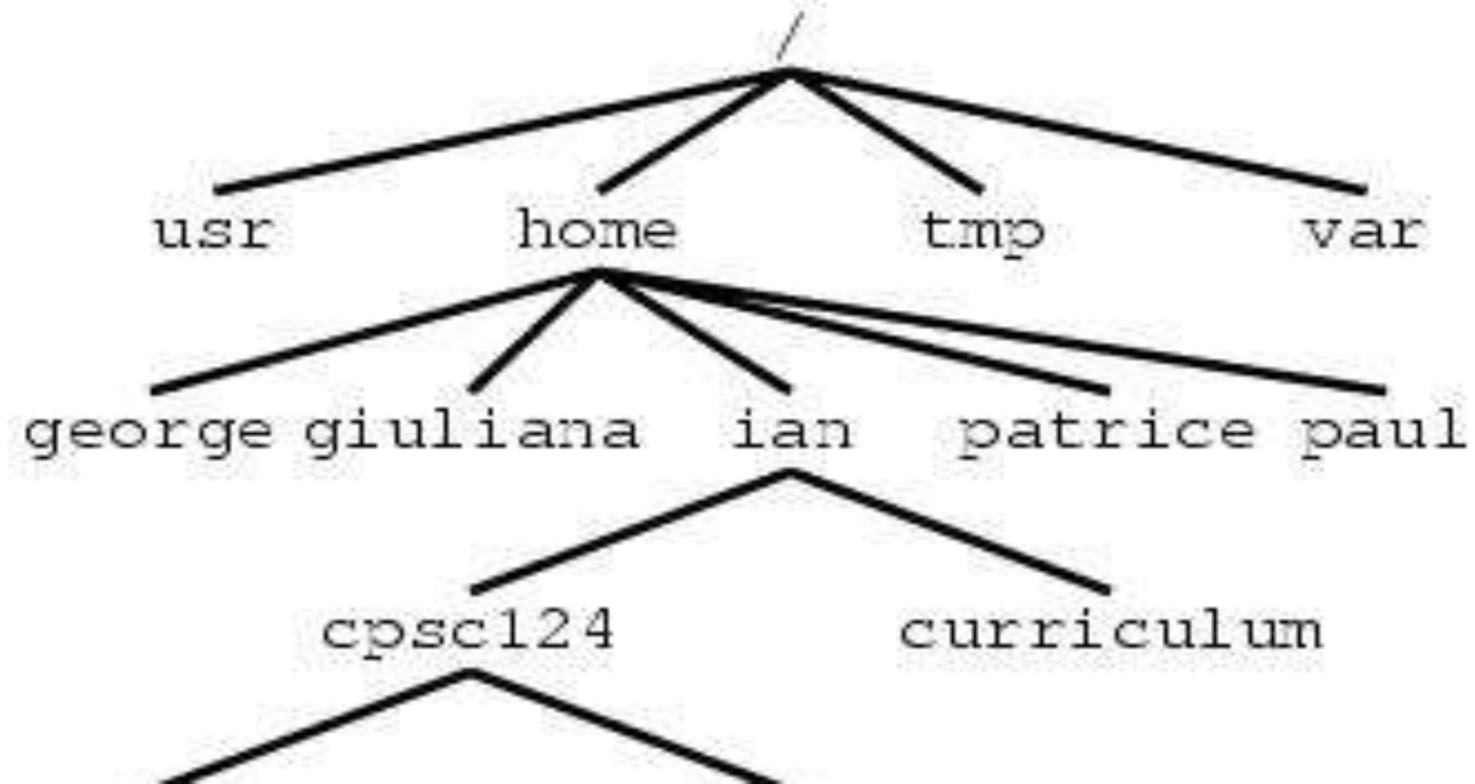
A file can be divided into 3 ways:

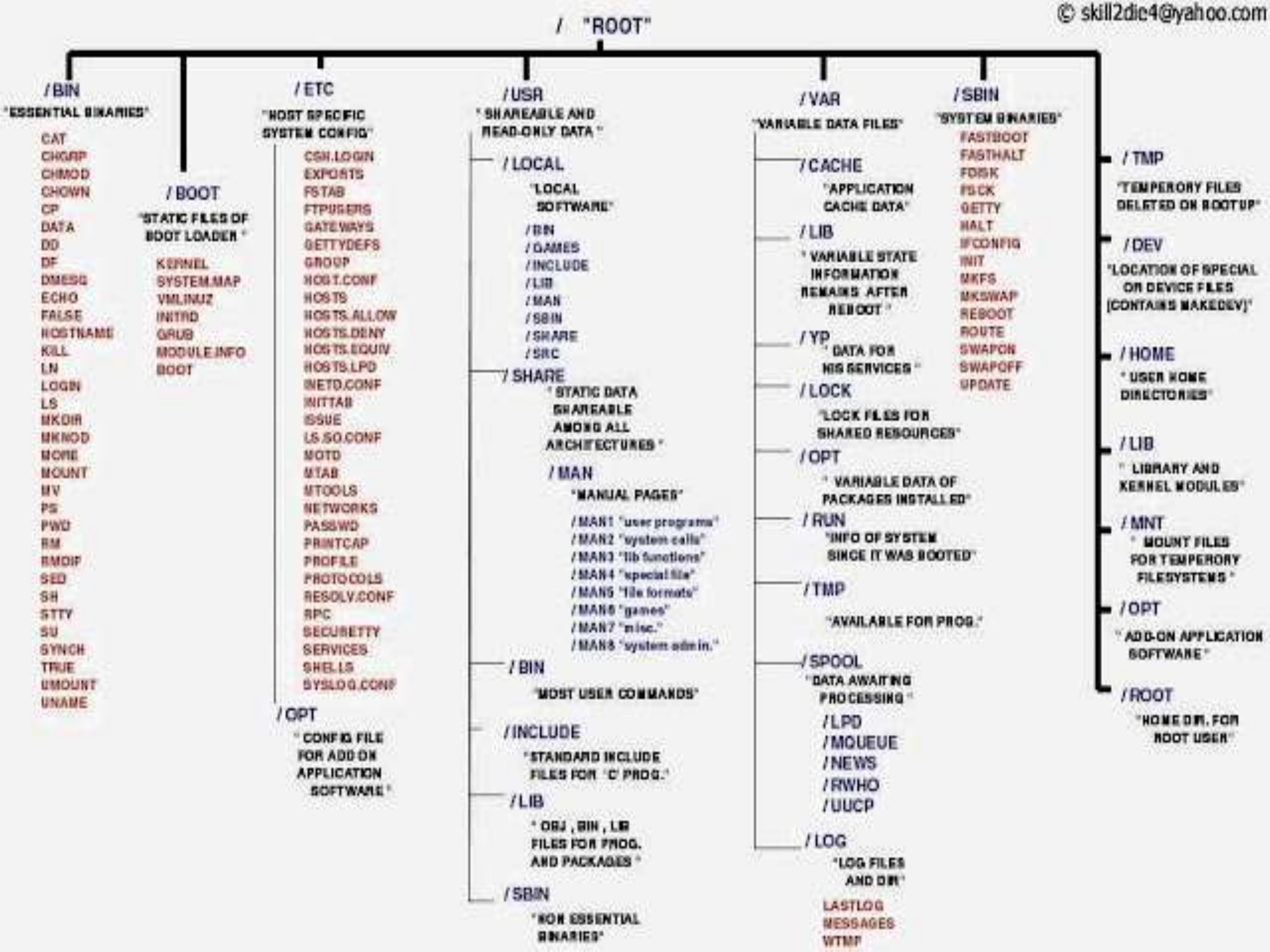
1. Regular files: The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary. It contains only data as a stream of characters.

2. Directory file: A file 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, but only the kernel

File system structure

The Unix file system is a tree hierarchical data structure that allows users to store information by name. At the top of the hierarchy is the root directory, which always has the name /. A typical Unix file system might look like:




© skill2die4@yahoo.com
/ "ROOT"
/ BIN
"ESSENTIAL BINARIES"
CAT
CHGRP
CHMOD
CHOWN
CP
DATA
DD
DF
DMESG
ECHO
FALSE
HOSTNAME
KILL
LN
LOGIN
LS
MKDIR
MKNOD
MORE
MOUNT
MV
PS
PWD
RM
RMDIR
SED
SH
STTY
SU
SYNCH
TRUE
UMOUNT
UNAME
/ BOOT
"STATIC FILES OF BOOT LOADER "
KERNEL
SYSTEM.MAP
VMLINUZ
INITRD
GRUB
MODULE.INFO
BOOT
/ ETC
"HOST SPECIFIC SYSTEM CONFIG"
CSH.LOGIN
EXPORTS
FSTAB
FTPUSERS
GATEWAYS
GETTYDEFS
GROUP
HOST.CONF
HOSTS
HOSTS.ALLOW
HOSTS.DENY
HOSTS.EQUIV
HOSTS.LPD
INETD.CONF
INITTAB
ISSUE
LS.SO.CONF
MOTD
MTAB
MTOOLS
NETWORKS
PASSWD
PRINTCAP
PROFILE
PROTOCOLS
RESOLV.CONF
RPC
SECURETTY
SERVICES
SHELLS
SYSLOG.CONF
/ OPT
" CONFIG FILE FOR ADD ON APPLICATION SOFTWARE "
/ USR
" SHAREABLE AND READ-ONLY DATA "
/ LOCAL
"LOCAL SOFTWARE"
/ BIN
/ GAMES
/ INCLUDE
/ LIB
/ MAN
/ SBIN
/ SHARE
/ SRC
/ SHARE
" STATIC DATA SHAREABLE AMONG ALL ARCHITECTURES "
/ MAN
"MANUAL PAGES"
/ MAN1 "user programs"
/ MAN2 "system calls"
/ MAN3 "lib functions"
/ MAN4 "special file"
/ MAN5 "file formats"
/ MAN6 "games"
/ MAN7 "misc."
/ MAN8 "system admin."
/ BIN
"MOST USER COMMANDS"
/ INCLUDE
"STANDARD INCLUDE FILES FOR 'C' PROG."
/ LIB
" OBJ , BIN , LIB FILES FOR PROG. AND PACKAGES "
/ SBIN
"NON ESSENTIAL BINARIES"
/ VAR
"VARIABLE DATA FILES"
/ CACHE
"APPLICATION CACHE DATA"
/ LIB
" VARIABLE STATE INFORMATION REMAINS AFTER REBOOT "
/ YP
" DATA FOR NIS SERVICES "
/ LOCK
"LOCK FILES FOR SHARED RESOURCES"
/ OPT
" VARIABLE DATA OF PACKAGES INSTALLED"
/ RUN
"INFO OF SYSTEM SINCE IT WAS BOOTED"
/ TMP
"AVAILABLE FOR PROG."
/ SPOOL
" DATA AWAITING PROCESSING "
/ LPD
/ MQUEUE
/ NEWS
/ RWHO
/ UUCP
/ LOG
"LOG FILES AND DIR"
LASTLOG
MESSAGES
WTMP
/ SBIN
"SYSTEM BINARIES"
FASTBOOT
FASTHALT
FDISK
FSCK
GETTY
HALT
IFCONFIG
INIT
MKFS
MKSWAP
REBOOT
ROUTE
SWAPON
SWAPOFF
UPDATE
/ TMP
"TEMPERORY FILES DELETED ON BOOTUP"
/ DEV
"LOCATION OF SPECIAL OR DEVICE FILES (CONTAINS MAKEDEV)"
/ HOME
" USER HOME DIRECTORIES"
/ LIB
" LIBRARY AND KERNEL MODULES"
/ MNT
" MOUNT FILES FOR TEMPERORY FILESYSTEMS "
/ OPT
" ADD-ON APPLICATION SOFTWARE"
/ ROOT
"HOME DIR. FOR ROOT USER"

•Files in Linux systems are stored in tree-like hierarchical file system.

• The root of a file system is the root ("/") directory.

• The leaf nodes of a file system tree are either empty directory files or other types of files.

• **Absolute path name of a file consists of the names of all the directories, starting from the root.**

• **Ex: /usr/divya/a.out**

• **Relative path name may consist of the "." and ".." characters. These are references to current and parent directories respectively.**

• **Ex: ../../.login denotes .login file which may be found 2 levels up from the current directory**

• A file name may not exceed NAME_MAX characters (14 bytes) and the total number of characters of a path name may not exceed PATH_MAX (1024 bytes).

• POSIX.1 defines _POSIX_NAME_MAX and _POSIX_PATH_MAX in <limits.h> header

•File name can be any of the following character set only

A to Z	a to z	0 to 9	_

• Path name of a file is called the **hardlink.**

• A file may be referenced by more than one path name if a user creates one or more hard links to the file using **ln command.**

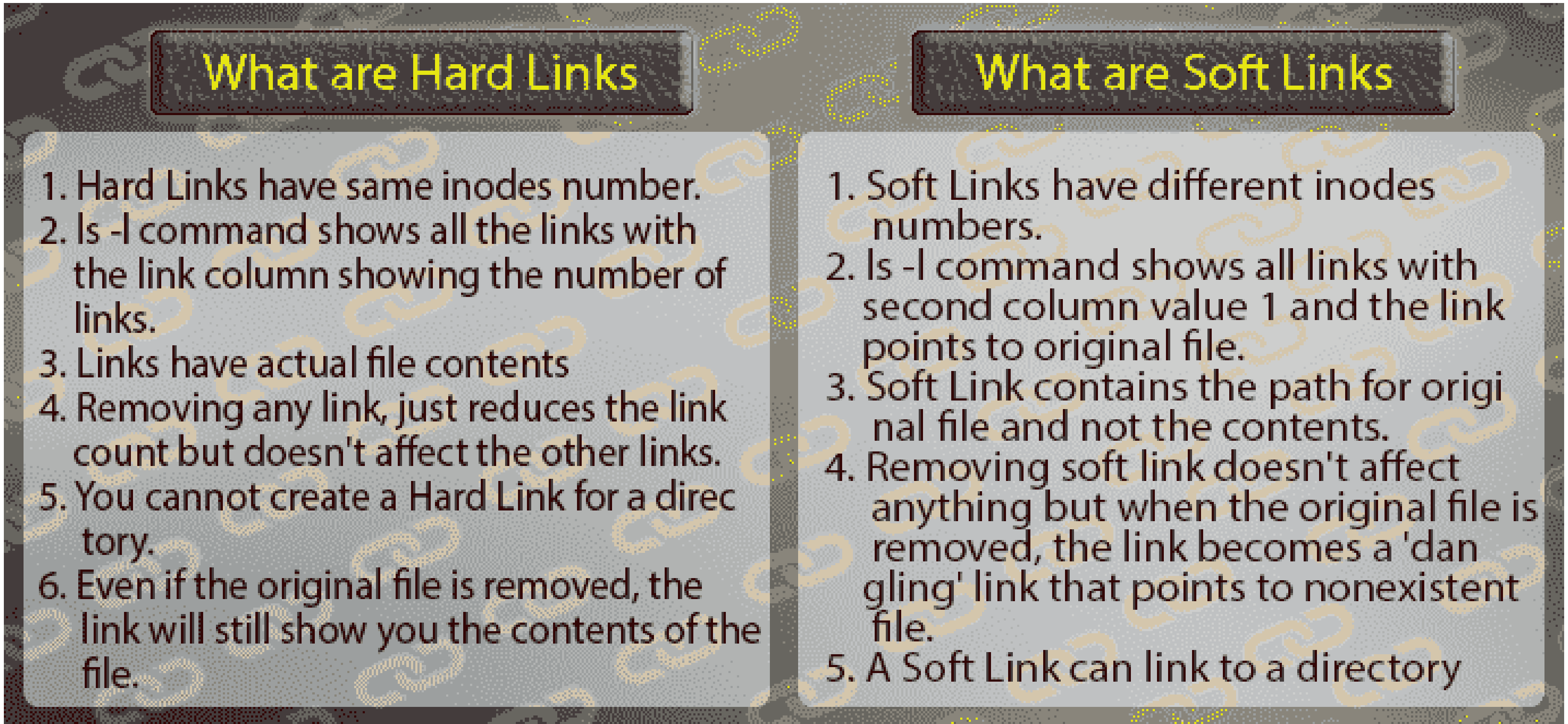
ln /usr/foo/path1 /usr/prog/new/n1

• If the –s option is used, then it is a symbolic (soft) link .

The following files are commonly defined in most UNIX systems

FILE	**Use**
/etc	Stores system administrative files and programs
/etc/passwd	Stores all user information's
/etc/shadow	Stores user passwords
/etc/group	Stores all group information
/bin	Stores all the system programs like cat, rm, cp,etc.

/dev	Stores all character device and block device files
/usr/include	Stores all standard header files.
/usr/lib	Stores standard libraries
/tmp	Stores temporary files created by program(Suppose your doing some work, during that work you need some temporary file to store ur data)



What are Hard Links

1. Hard Links have same inodes number.
2. ls -l command shows all the links with the link column showing the number of links.
3. Links have actual file contents
4. Removing any link, just reduces the link count but doesn't affect the other links.
5. You cannot create a Hard Link for a direc tory.
6. Even if the original file is removed, the link will still show you the contents of the file.

What are Soft Links

1. Soft Links have different inodes numbers.
2. ls -l command shows all links with second column value 1 and the link points to original file.
3. Soft Link contains the path for origi nal file and not the contents.
4. Removing soft link doesn't affect anything but when the original file is removed, the link becomes a 'dan gling' link that points to nonexistent file.
5. A Soft Link can link to a directory

Example:Hard links

➢Lets first create a "Test" directory and inside we create a new file "sample1".

```
$ mkdir Test
 $ cd Test
$ touch sample1
```

➢Now, create a hard link to sample1. Name the hard link sample2.

```
$ ln sample1 sample2
```

➢Display inodes for both files using i argument of the ls command.

```
$ ls -il sample1 sample2
1482256 -rw-r--r-- 2 bruno bruno 21 May 5 15:55 sample1
1482256 -rw-r--r-- 2 bruno bruno 21 May 5 15:55 sample2
```

➢From the output, you can notice that both sample1 and sample2 have the same inode number (1482256). Also, both files have the same file permissions and the same size.

➢Now Remove the original sample1

```
$ rm sample1
```

➢After removing Hard Link just have a look at the content of the "link" sample2.

```
$ cat sample2
```

➢You will still be able to see the contents of the file.

Symbolic links:

Lets create a soft link for the file sample2 using below command.

```
$ ln -s sample2 sample3
```

- Display inodes for both using i argument of ls command.

```
$ ls -il sample2 sample3
1482256 -rw-r--r-- 1 bruno bruno 21 May 5 15:55 FileB
1482226 lrwxrwxrwx 1 bruno bruno 5 May 5 16:22 FileC -> FileB
```

- From the output, you can notice that the **inodes are different** and the symbolic link has an "l" before the rwxrwxrwx. The permissions are different for the link and the original file because it is just a symbolic link.

- Now list the contents:

```
$ cat sample2 $ cat sample3
```

- Now remove the original file

```
$ rm sample2
```

- And then check the Test directory:

```
$ ls
```

- It will still display symbolic link sample3 but if you try to list the contents, it will tell you that there is no such file or directory.

```
$ cat sample3
```

Inode

Each file is associated with an *inode*, which is identified by an integer number, often referred to as an *i-number* or *inode number*.

Inodes store information about files and directories (folders), such as file

1.ownership

2 .type of file

3.Group owner

4.File access permissions

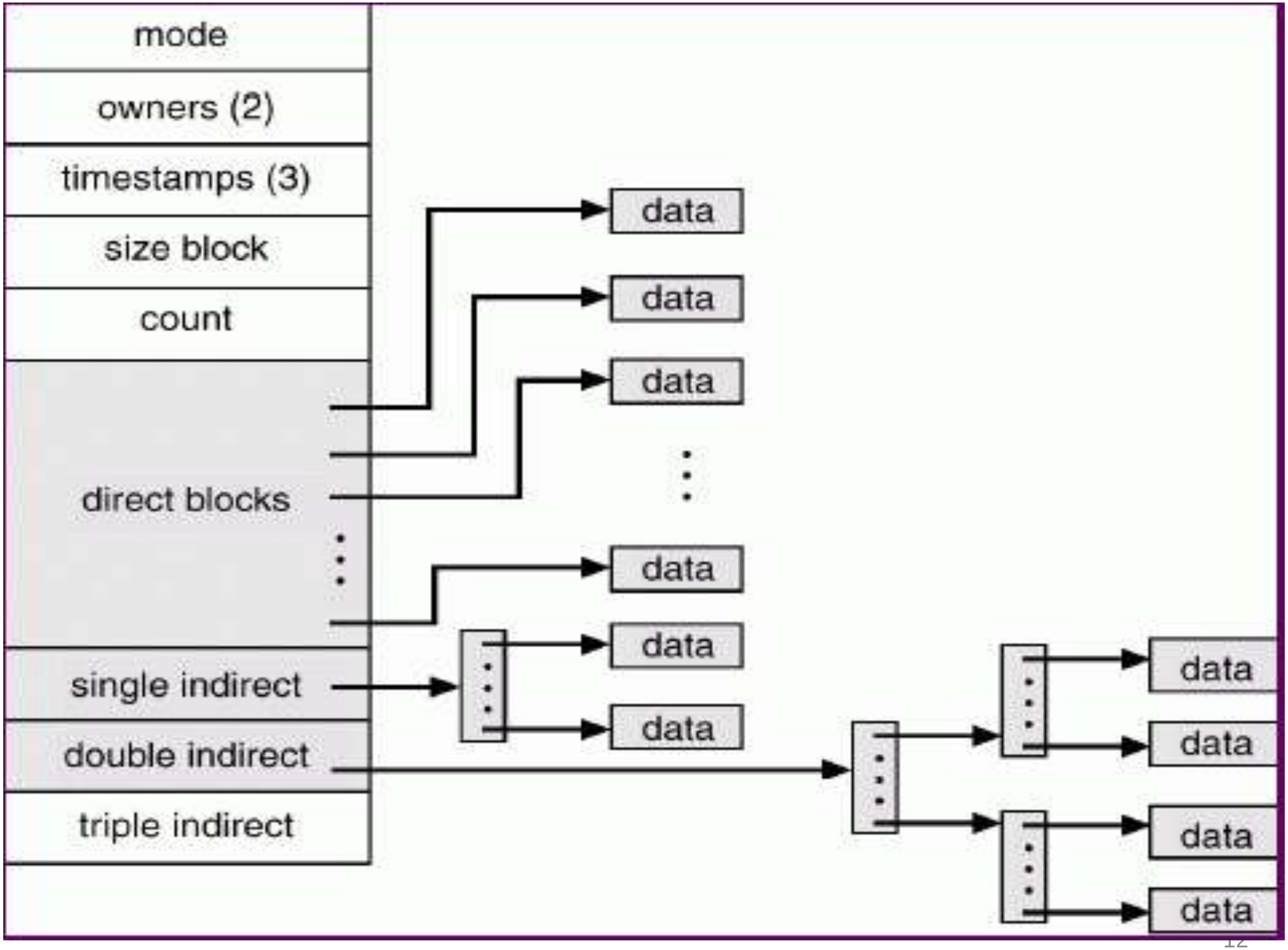
5.Date and time of last access

6.Number of links

7.Size of the file

8.Address of the blocks where file is physically present

A file's inode number can be found using the ls -i command. The ls -l command displays some of the inode contents for each file.


mode
owners (2)
timestamps (3)
size block
count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

File Type

A file in a UNIX or POSIX system may be one of the following types:

- regular file
- directory file
- FIFO file
- Character device file
- Block device file

Regular file

- A regular file may be either a text file or a binary file
- These files may be read or written to by users with the appropriate access permission
- Regular files may be created, browsed through and modified by various means such as text editors or compilers, and they can be removed by specific system commands

Directory file

- It is like a folder that contains other files, including sub-directory files.
- It provides a means for users to organise their files into some hierarchical structure based on file relationship or uses.
- Ex: **/bin directory contains all system executable programs, such as cat, rm, sort**
- A directory may be created in UNIX by the **mkdir command**

  Ex: **mkdir /usr/foo/xyz**
- A directory may be removed via the **rmdir command**

  Ex: **rmdir /usr/foo/xyz**

The content of directory may be displayed by the **ls command**

Device file

Block device file	**Character device file**
It represents a physical device that transmits data a block at a time.	It represents a physical device that transmits data in a character-based manner.
Ex: hard disk drives and floppy disk drives	Ex: line printers, modems, and consoles

A physical device may have both block and character device files representing it for different access methods.

An application program in turn may choose to transfer data by either a character-based(via character device file) or block-based(via block device file)

- A device file is created in UNIX via the **mknod command**

Ex: **mknod /dev/cdsk c 115 5**

```
mknod      /dev/cdsk      c      115      5
```

Here , c - character device file

115 - major device number

5 - minor device number

➢It is a special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer.

- The size of the buffer is fixed to PIPE_BUF.
- Data in the buffer is accessed in a first-in-first-out manner.
- The buffer is allocated when the first process opens the FIFO file for read or write
- The buffer is discarded when all processes close their references (stream pointers) to the FIFO file.
- Data stored in a FIFO buffer is temporary.
- A FIFO file may be created via the **mkfifo command.**

The following command creates a FIFO file (if it does not exists)

mkfifo /usr/prog/fifo_pipe

The following command creates a FIFO file (if it does not exists)

mknod /usr/prog/fifo_pipe p

- FIFO files can be removed using **rm command.**

Symbolic link file :

- BSD UNIX & SV4 defines a symbolic link file.
- A symbolic link file contains a path name which references another file in either local or remote file system.
- POSIX.1 does not support symbolic link file type
- A symbolic link may be created in UNIX via the **ln command**
- Ex: **ln -s /usr/divya/original /usr/raj/slink**
- It is possible to create a symbolic link to reference another symbolic link.
- **rm, mv and chmod commands will operate only on the symbolic link arguments directly and not on the files that they reference.**

The type of the file can be determined with the macros given in table

Macro name	file type
S_ISREG	regular file
S_ISDIR	directorl speciocky file
S_ISCHAR	character special file
S_ISBLK	block special file
S_ISFIFO	fifo file
S_ISLNK	symbolic link
S_ISSOCK	socket file

The Standard I/O

The Standard I/O:

- The functions which are already present in the library is known as the standard I/O functions.
- Three file streams are automatically opened when a program is started. They are stdin, stdout and stderr.
- These are declared in stdio.h and represent the standard input, output and error output, respectively, which correspond to the low level file descriptors 0, 1 and 2.

In this next section, we'll look at:

- fopen, fclose
- fread, fwrite
- fflush
- fseek
- fgetc, getc, getchar
- fputc, putc, putchar
- fgets, gets

Opening a Stream: fopen()

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

➢The fopen library function is the analog of the low level open system call.

➢fopen opens the file named by the filename parameter and associates a stream with it.

➢The mode parameter specifies how the file is to be opened. It's one of the following strings:

· "r" or "rb"	Open for reading only
· "w" or "wb“	Open for writing, truncate to zero length
· "a" or "ab“	Open for writing, append to end of file
· "r+" or "rb+" or "r+b"	Open for update (reading and writing)
· "w+" or "wb+" or "w+b"	Open for update, truncate to zero length
· "a+" or "ab+" or "a+b"	Open for update, append to end of file

- The b indicates that the file is a binary file rather than a text file. Note that, unlike DOS, UNIX doesn't make a distinction between text and binary files. It treats all files exactly the same, effectively as binary files. It's also important to note that the mode parameter must be a string, and not a character. Always use "r", and never 'r'.
- If successful, fopen returns a non−null FILE * pointer. If it fails, it returns the value NULL, defined in stdio.h.

Fread :

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

- The fread library function is used to read data from a file stream. Data is read into a data buffer given by ptr from the stream.
- Both fread and fwrite deal with data records. These are specified by a record size, size, and a count, nitems, of records to transfer.
- It returns the number of items (rather than the number of bytes) successfully read into the data buffer. At the end of a file, fewer than nitems may be returned, including zero.

fwrite

#include <stdio.h>

size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE *stream);

The fwrite library call has a similar interface to fread. It takes data records from the specified data buffer and writes them to the output stream.

➢It returns the number of records successfully written.

Important Note that **fread and fwrite are not recommended for use with structured data.**

fclose:

An open stream is closed by calling fclose.

```
#include <stdio.h>
int fclose(FILE *fp);
```

Returns: upon successful complition 0 is returned. other wise, EOF is returned and the global variable errno is set to indicate the error.

- Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded. If the standard I/O library had automatically allocated a buffer for the stream, that buffer is released.

fflush:

```
#include <stdio.h>
int fflush(FILE *stream);
```

➢The fflush library function causes all outstanding data on a file stream to be written immediately.

➢You can use this to ensure that, for example, an interactive prompt has been sent to a terminal before any attempt to read a response.

➢It's also useful for ensuring that important data has been committed to disk before continuing.

➢You can sometimes use it when you're debugging a program to make sure that the program is writing data and not hanging. Note that an implied flush operation is carried out when fclose is called, so you don't need to call fflush before fclose.**);**

Fseek:

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

- The fseek function is the file stream equivalent of the lseek system call. It sets the position in the stream for the next read or write on that stream.
- The meaning and values of the offset and whence parameters are the same as for lseek. However, where lseek returns an off_t, fseek returns an integer: 0 if it succeeds, −1 if it fails, with errno set to indicate the error. So much for standardization!

- whence can be one of the following:
- ❑ SEEK_SET: offset is an absolute position
- ❑ SEEK_CUR: offset is relative to the current position
- ❑ SEEK_END: offset is relative to the end of the file
- lseek returns the offset measured in bytes from the beginning of the file that the file pointer is set to,
- –1 on failure.
- The type off_t, used for the offset in seek operations, is an implementation-dependent
- integer type defined in sys/types.h.

fgetc, getc, getchar

```
#include <stdio.h>
        int fgetc(FILE *stream);
        int getc(FILE *stream);
        int getchar();
```

- The fgetc function returns the next byte, as a character, from a file stream. When it reaches the end of the file or there is an error, it returns EOF. You must use ferror or feof to distinguish the two cases.
- The getc function is equivalent to fgetc, except that you can implement it as a macro, in which case the stream argument must not have side effects (i.e. it can't affect variables that are neither local nor passed to the functions as parameters). Also, you can't then use the address of getc as a function pointer.
- The getchar function is equivalent to getc(stdin) and reads the next character from the standard input.

fputc, putc, putchar

```
#include <stdio.h>
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

- The fputc function writes a character to an output file stream. It returns the value it has written, or EOF on failure.
- As with fgetc/getc, the function putc is equivalent to fputc, but you may implement it as a macro.
- The putchar function is equivalent to putc(c,stdout), writing a single character to the standard output. Note that putchar takes and getchar returns characters as ints, not char. This allows the end of file (EOF) indicator to take the value –1, outside the range of character numbers codes.

fgets, gets

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

- The fgets function reads a string from an input file stream (one line at a time). It writes characters to the string pointed to by s until a newline is encountered, n–1 characters have been transferred or the end of file is reached, whichever occurs first.
- gets() reads line from keyboard
- Any newline encountered is transferred to the receiving string and a terminating null byte, \0, is added. Only a maximum of n–1 characters are transferred in any one call, because the null byte must be added to finish the string, and make up the n bytes.

Formatted I/O

Formatted Output:

Formatted output is handled by the four printf functions.

```
#include <stdio.h>
int printf(const char *restrict format, ...);
 int fprintf(FILE *restrict fp, const char *restrict format, ...);
```

Both return: number of characters output if OK, negative value if output error.

```
int sprintf (char *restrict buf, const char *restrict format, ...);
 int snprintf (char *restrict buf, size_t n, const char *restrict format, ...);
```

Both return: number of characters stored in array if OK, negative value if encoding error.

The printf function writes to the standard o/p, fprintf writes to the specified stream& sprintf places the formatted characters in the array buf. The sprintf function automatically appends a null byte at the end of the array, but this null byte is not included in the return value.

Example:

```
printf("Some numbers: %d, %d, and %d\n", 1, 2, 3);
```

This produces, on the standard output:

```
Some numbers: 1, 2, and 3
```

- Here are some of the most commonly used conversion specifiers:

❑ %d, %i: Print an integer in decimal

❑ %o, %x: Print an integer in octal, hexadecimal

❑ %c: Print a character

❑ %s: Print a string

❑ %f: Print a floating-point (single precision) number

❑ %e: Print a double precision number, in fixed format

❑ %g: Print a double in a general format

- It’s very important that the number and type of the arguments passed to printf match the conversion specifiers in the format string.

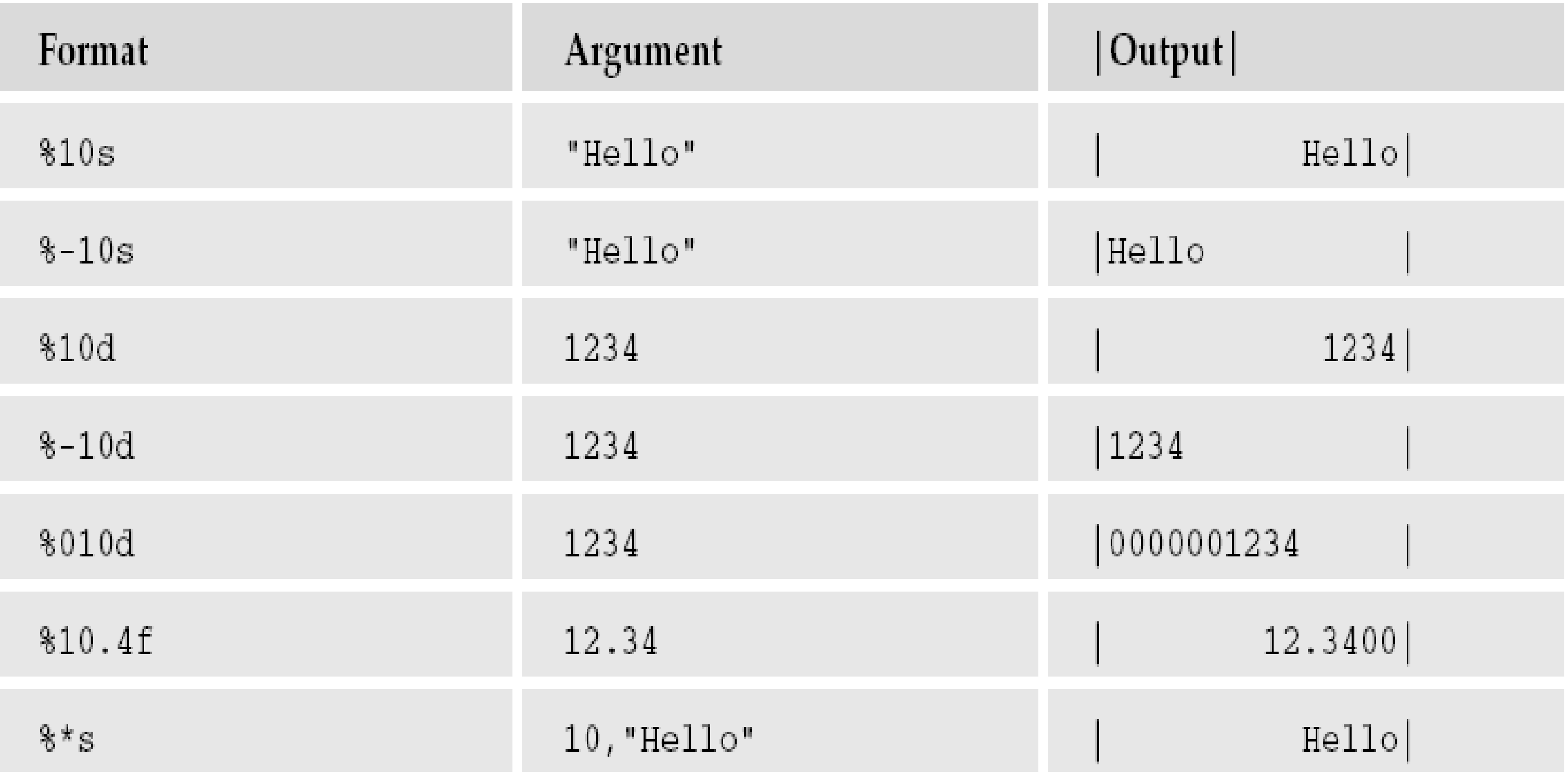
Example:

```
char initial = ‘A’;
char *surname = “Matthew”;
double age = 13.5;
printf(“Hello Mr %c %s, aged %g\n”, initial, surname,
    age);
```

This produces

- Hello Mr A Matthew, aged 13.5

- Field specifiers are given as numbers immediately after the % character in a conversion specifier.
- The following table contains some more examples of conversion specifiers and resulting output.



Format	Argument	|Output|
%10s	"Hello"	| Hello|
%-10s	"Hello"	|Hello |
%10d	1234	| 1234|
%-10d	1234	|1234 |
%010d	1234	|0000001234 |
%10.4f	12.34	| 12.3400|
%*s	10,"Hello"	| Hello|

- Note that a negative field width means that the item is written left-justified within the field.
- A variable field width is indicated by using an asterisk (*). In this case, the next argument is used for the width.
- A leading zero indicates the item is written

  with leading zeros.

Formatted Input:

Formatted input is handled by the three scanf functions.

```
#include <stdio.h>
 int scanf (const char *restrict format, ...);
int fscanf (FILE *restrict fp, const char *restrict format, ...);
int sscanf (const char *restrict buf, const char *restrict format, ...);
```

All three return: number of input items assigned,

EOF if input error or end of file before any conversion.

example:

```
int num;
scanf(“Hello %d”, &num);
```

- This call to scanf will succeed only if the next five characters on the standard input are Hello.
- Then, if the next characters form a recognizable decimal number, the number will be read and the value assigned to the variable num.

conversion specifiers are

- %d: Scan a decimal integer
- %o, %x: Scan an octal, hexadecimal integer
- %f, %e, %g: Scan a floating-point number
- %c: Scan a character (whitespace not skipped)
- %s: Scan a string
- %[]: Scan a set of characters
- %%: Scan a % character

- Like printf, scanf conversion specifiers may also have a field width to limit the amount of input consumed

- A size specifier (either h for short or l for long) indicates whether the receiving argument is shorter or longer than the default. This means that %hd indicates a short int, %ld a long int, and%lg a double precision floating-point number.
- A specifier beginning with an asterisk indicates that the item is to be ignored.
- Use the %c specifier to read a single character in the input.
- string must be sufficient to hold the longest string in the input stream.
- It’s better to use a field specifier, or a combination of fgets and sscanf, to read in a line of input and then scan it. This will prevent possible buffer overflows that could be exploited by a malicious user.

- Given the input line,
- Hello, 1234, 5.678, X, string to the end of the line
- this call to scanf will correctly scan four items:

```
char s[256];
int n;
float f;
char c;
```

- scanf(“Hello,%d,%g, %c, %[^\n]“, &n,&f,&c,s);
- The scanf functions return the number of items successfully read, which will be zero if the first item fails.
- If the end of the input is reached before the first item is matched, EOF is returned.
- If a read error occurs on the file stream, the stream error flag will be set and the error variable, errno, will be set to indicate the type of error.

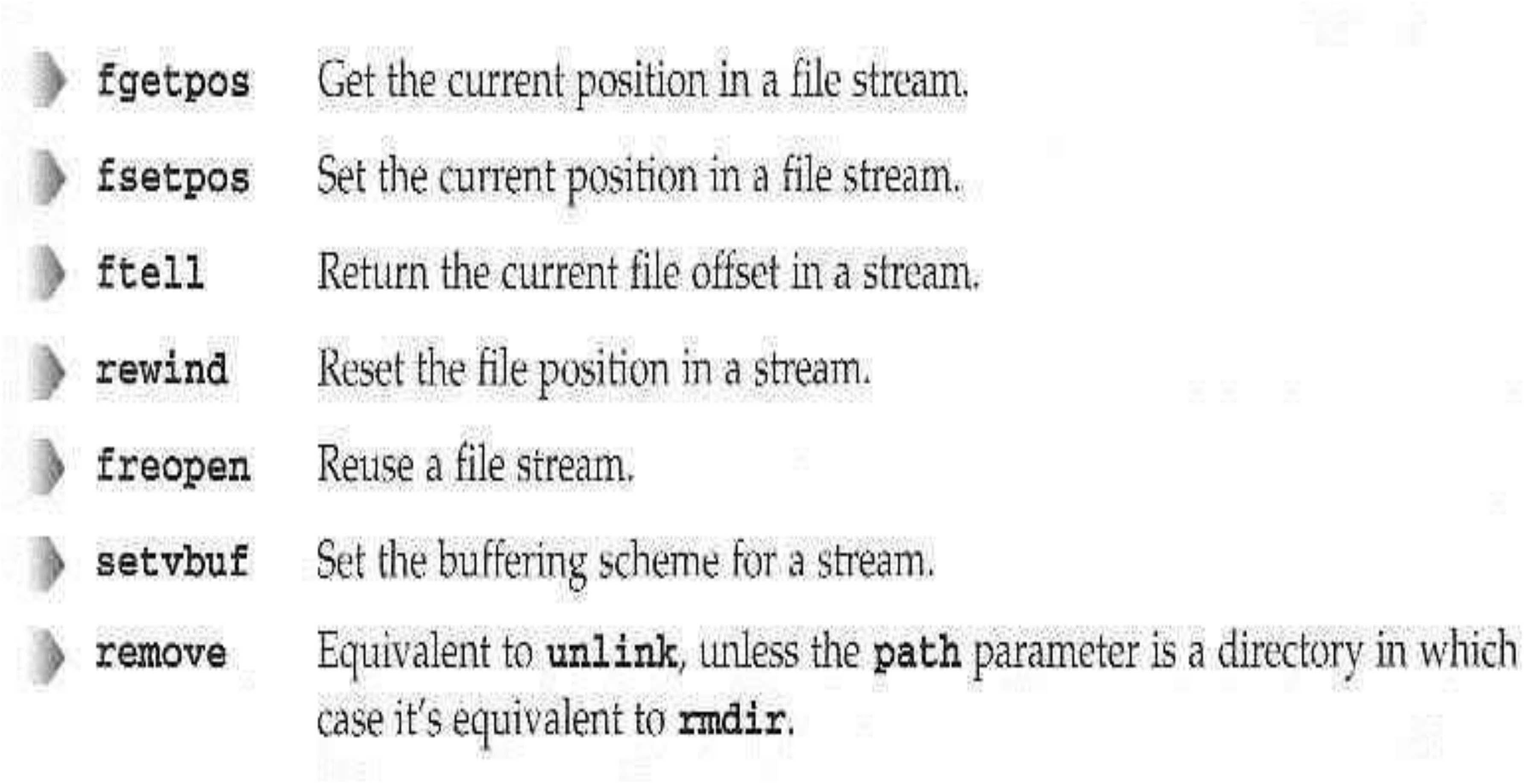
Library functions

List of functions

Function	**Description**
Clearerr	check and reset stream status
Fclose	close a stream
Fdopen	stream open functions
Feof	check and reset stream status
Ferror	check and reset stream status
Fflush	flush a stream
Fgetc	get next character or word from input stream
Fgetpos	reposition a stream
Fgets	get a line from a stream
Fileno	return the integer descriptor of the argument stream
Fopen	stream open functions
Fprintf	formatted output conversion
Fpurge	flush a stream
Fputc	output a character or word to a stream
Fputs	output a line to a stream

Other Stream Functions

Other library functions use either stream paramters or the standard streams **stdin, stdout, stderr:**

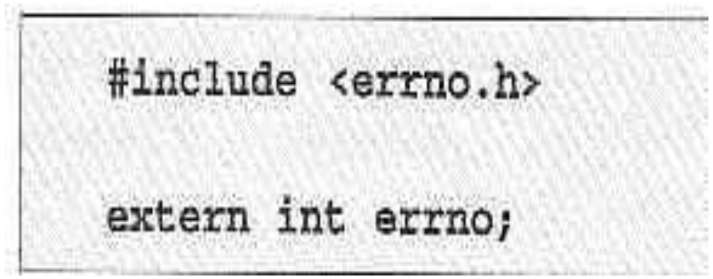


`fgetpos`	Get the current position in a file stream.
`fsetpos`	Set the current position in a file stream.
`ftell`	Return the current file offset in a stream.
`rewind`	Reset the file position in a stream.
`freopen`	Reuse a file stream.
`setvbuf`	Set the buffering scheme for a stream.
`remove`	Equivalent to **`unlink`**, unless the **`path`** parameter is a directory in which case it's equivalent to **`rmdir`**.

Stream Errors

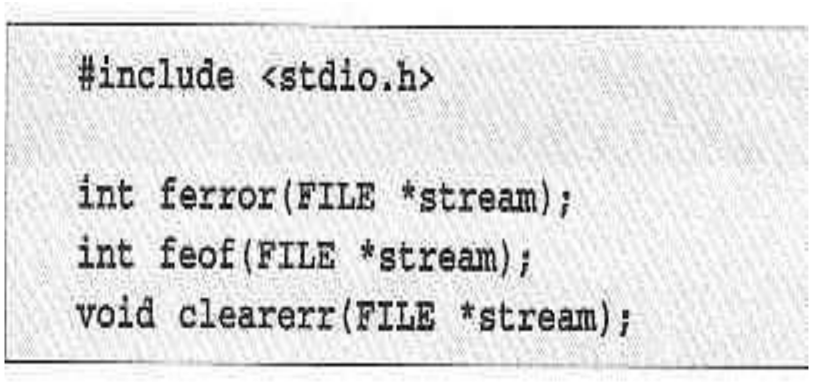
To indicate an error, many of the **stdio library functions return out of range values, such as null pointers or the constant EOF.**

In these cases, the error is indicated in the external variable **errno:**



```
#include <errno.h>

extern int errno;
```

You can also interrogate the state of a file stream to determine whether an error has occurred, or the end of file has been reached.



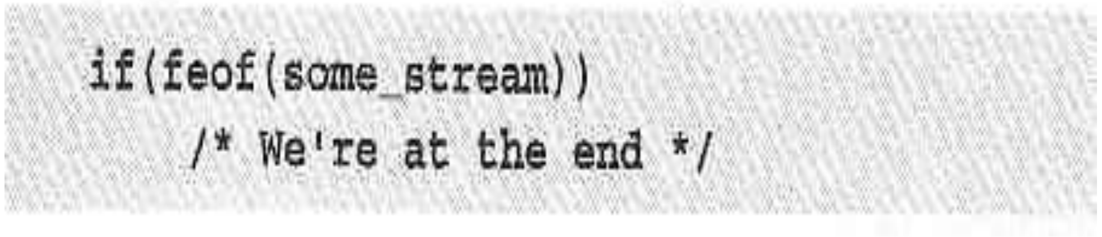
```
#include <stdio.h>

int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
```

Stream Errors

The **ferror function tests the error indicator for a stream and returns non-zero if its set, zero otherwise.**

The **feof function tests the end-of-file indicator within a stream and returns non-zero if it is set zero otherwise.**

You use it like this:



```
if(feof(some_stream))
    /* We're at the end */
```

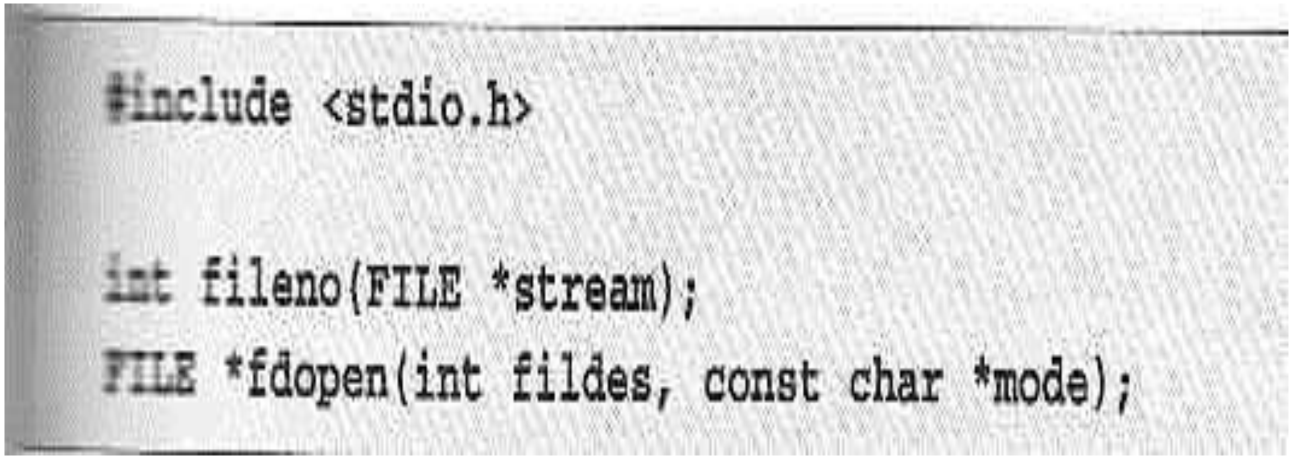
The **clearerr function clears the end-of-file and error indicators for the stream to which stream points.**

Streams and File Descriptors

Each file stream is associated with a low level file descriptor.

You can mix low-level input and output operations with higher level stream operations, but this is generally unwise.

The effects of buffering can be difficult to predict.



```
#include <stdio.h>

int fileno(FILE *stream);
FILE *fdopen(int fildes, const char *mode);
```

Kernel Support for files

In UNIX system V, the kernel maintains a file table that has an entry of all opened files and also there is an inode table that contains a copy of file inodes that are most recently accessed.

The steps involved are : 1. The kernel will search the process descriptor table and look for the first unused entry. If an entry is found, that entry will be designated to reference the file .The index of the entry will be returned to the process as the file descriptor of the opened file.

2. The kernel will scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

If an unused entry is found the following events will occur:

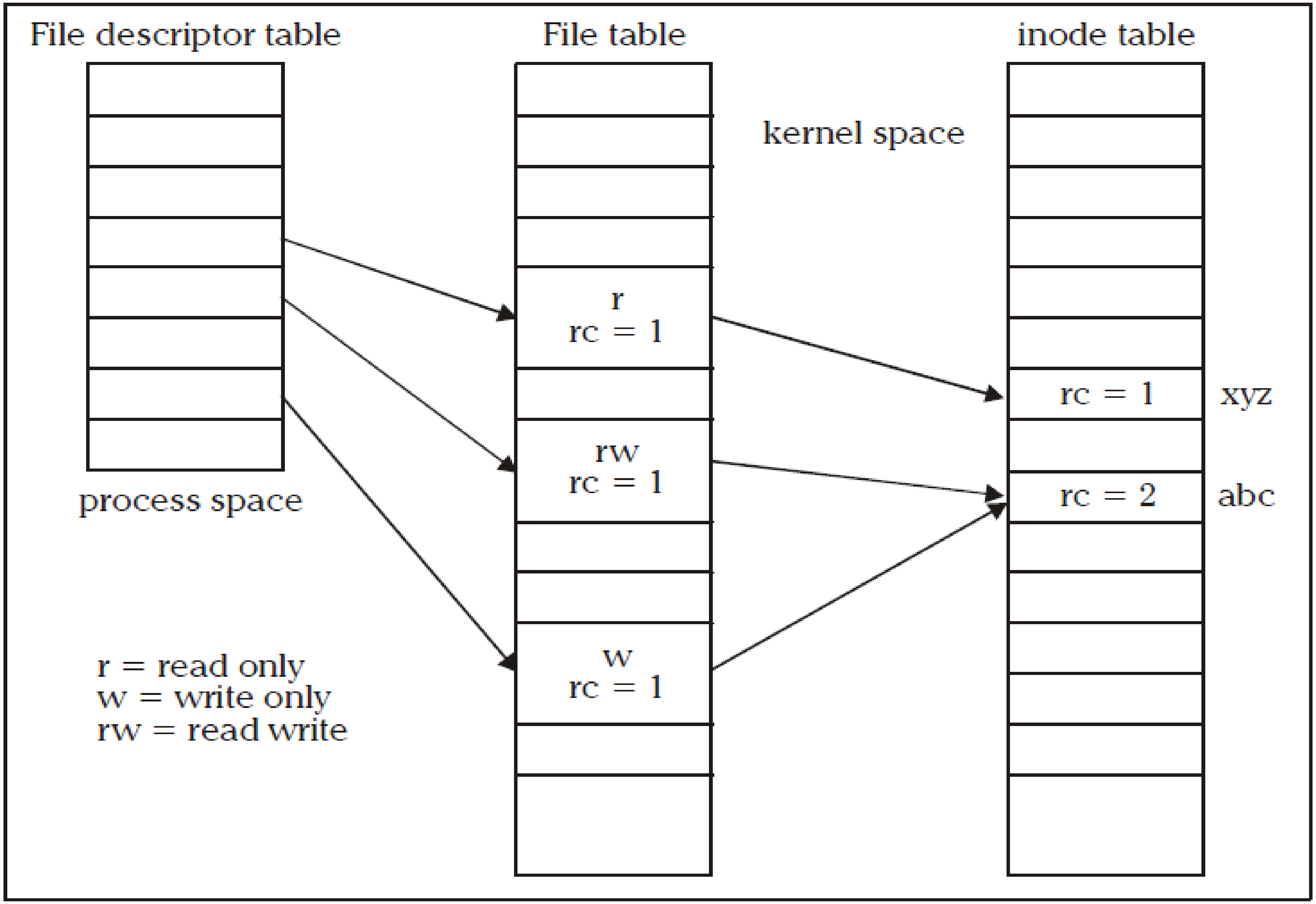
• The process file descriptor table entry will be set to point to this file table entry.

• The file table entry will be set to point to the inode table entry, where the inode record of the file is stored.

System calls

- A system call, sometimes referred to as a kernel call, is a request in a Linux operating system made by an active process for a service performed by the kernel.
- The kernel is a program that constitutes the core of an operating system, and it has complete control over all resources on the system and everything that occurs on it. When a user mode process (i.e., a process currently in user mode) wants to utilize a service provided by the kernel , it must switch temporarily into kernel mode, also called system mode, by means of a system call.
- Kernel mode has root (i.e., administrative) privileges, including root access permissions. This allows the operating system to perform restricted actions such as accessing hardware devices or the memory management unit (MMU).
- The file table entry will contain the current file pointer of the open file. This is an offset from the beginning of the file where the next read or write will occur.

Data Structure for File Manipulation



- The file table entry will contain an open mode that specifies that the file opened is for read only, write only or read and write etc. This should be specified in open function call.
- The reference count (rc) in the file table entry is set to 1. Reference count is used to keep track of how many file descriptors from any process are referring the entry.
- The reference count of the in-memory inode of the file is increased by 1. This count specifies how many file table entries are pointing to that inode.
- If either (1) or (2) fails, the **open system call returns -1 (failure/error)**
- Normally the reference count in the file table entry is 1,if we wish to increase the rc in the file table entry, this can be done using fork,dup,dup2 system call.
- When a open system call is succeeded, its return value will be an integer (file descriptor). Whenever the process wants to read or write data from the file, it should use the file descriptor as one of its argument.

The following events will occur whenever a process calls the close function to close the files that are opened.

1. The kernel sets the corresponding file descriptor table entry to be unused
2. It decrements the rc in the corresponding file table entry by 1, if rc not equal to 0 go to step 6.
3. The file table entry is marked as unused.
4. The rc in the corresponding file inode table entry is decremented by 1, if rc value not equal to 0 go to step 6.
5. If the hard link count of the inode is not zero, it returns to the caller with a success status otherwise it marks the inode table entry as unused and de-allocates all the physical dusk storage of the file.
6. It returns to the process with a 0 (success) status.

File descriptors

In computer programming, a **file descriptor** (FD) is an abstract indicator for accessing a file. The term is generally used in POSIX operating systems

In POSIX, a file descriptor is an integer, specifically of the C type *int*. There are three standard POSIX file descriptors, corresponding to the three standard streams, which presumably every process should expect to have:

Integer value	Name
0	Standard input (*stdin*)
1	Standard output (*stdout*)
2	Standard error (*stderr*)

- Generally, a file descriptor is an index for an entry in a kernel-resident array data structure containing the details of open files.
- In POSIX this data structure is called a file descriptor table, and each process has its own file descriptor table. The process passes the file descriptor to the kernel through a system call, and the kernel will access the file on behalf of

- The process itself cannot read or write the file descriptor table directly.

- On Linux, the set of file descriptors open in a process can be accessed under the path /proc/PID/fd/, where PID is the process identifier.

- In Unix-like systems, file descriptors can refer to any Unix file type named in a file system. As well as regular files, this includes directories, block and character devices (also called "special files"), Unix domain sockets, and named pipes.

- File descriptors can also refer to other objects that do not normally exist in the file system, such as anonymous pipes and network sockets.

Low –level file access

UNIX system calls are used to manage the file system, control processes, and to provide inter process communication.

Types of system calls in UNIX:

1.Open	2.creat
3.read	4.write
5.Close	6.lseek
7.stat	8.fstat
9.ioctl	10.umask
11 .dup and dup2	12.fcntl

open function:

This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file.

The prototype of open function is

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

#include<sys/fcntl.h>

int open(const char *pathname, int accessmode, mode_t permission);
```

•If successful, open returns a nonnegative integer representing the open file descriptor.

• If unsuccessful, open returns –1.

• The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.

- If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non symbolic link file to which it refers.
- The second argument is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.
- Generally the access modes are specified in <fcntl.h>. Various access modes are:
- **O_RDONLY - open for reading file only**
- **O_WRONLY - open for writing file only**
- **O_RDWR - opens for reading and writing file.**
- There are other access modes, which are termed as access modifier flags, and one or more of the following can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

O_APPEND - Append data to the end of file.

O_CREAT - Create the file if it doesn’t exist

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

O_TRUNC - If file exists discard the file content and set the file size to zero bytes.

O_NONBLOCK - Specify subsequent read or write on the file should be non-blocking.

O_NOCTTY - Specify not to use terminal device file as the calling process control terminal.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
int main(){
        int file_desc;
int save_errno;
file_desc = open(“/tmp/LCK.test”,
    O_RDWR | O_CREAT |
    O_EXCL, 0444);
if (file_desc == -1) {
save_errno = errno;
printf(“Open failed with error
    %d\n”, save_errno);
}
else {
printf(“Open succeeded\n”);
}
exit(EXIT_SUCCESS);
}
```

The first time you run the program, the output is

```
$ ./lock1
Open succeeded
```

but the next time you try, you get

```
$ ./lock1
Open failed with error 17
```

creat Function

The prototype of creat is

```
#include <sys/types.h>
#include<unistd.h>
```

int ***creat(const char *pathname, mode_t mode);***

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

• The first argument pathname specifies name of the file to be created.

• The second argument mode_t, specifies permission of a file to be accessed by owner group and others.

• The creat function can be implemented using open function as:

```
#define creat(path_name, mode)
 open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

read :

• The read function fetches a fixed size of block of data from a file referenced by a given file descriptor.

• The prototype of read function is:

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

• If successful, read returns the number of bytes actually read.

• If unsuccessful, read returns –1.

• The first argument is an integer, fdesc that refers to an opened file.

• The second argument, buf is the address of a buffer holding any data read.

• The third argument specifies how many bytes of data are to be read from the file.

• The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.

- 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.

write :

• The write system call is used to write data into a file.

• The write function puts data to a file in the form of fixed block size referred by a given file descriptor.

•The prototype of write is

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

• If successful, write returns the number of bytes actually written.

• If unsuccessful, write returns –1.

• The first argument, fdesc is an integer that refers to an opened file.

• The second argument, buf is the address of a buffer that contains data to be written.

• The third argument, size specifies how many bytes of data are in the buf argument.

• The return value is usually equal to the number of bytes of data successfully

close :

• The close system call is used to terminate the connection to a file from a process.

• The prototype of the close is

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

• If successful, close returns 0. If unsuccessful, close returns –1.

• The argument fdesc refers to an opened file.

• Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.

• The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

fcntl : (Manipulate file descriptor 0r File Control)

• The fcntl function helps a user to query or set flags and the close-on-exec flag of any file descriptor.

• The prototype of fcntl is

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

• The first argument is the file descriptor.

• The second argument cmd specifies what operation has to be performed.

• The third argument is dependent on the actual cmd value.

• The possible cmd values are defined in <fcntl.h> header.

cmd value	Use
F_GETFL	Returns the access control flags of a file descriptor fdesc
F_SETFL	Sets or clears access control flags that are specified in the third argument to fcntl. The allowed access control flags are O_APPEND & O_NONBLOCK
F_GETFD	Returns the close-on-exec flag of a file referenced by fdesc. If a return value is zero, the flag is off; otherwise on.
F_SETFD	Sets or clears the close-on-exec flag of a fdesc. The third argument to fcntl is an integer value, which is 0 to clear the flag, or 1 to set the flag
F_DUPFD	Duplicates file descriptor fdesc with another file descriptor. The third argument to fcntl is an integer value which specifies that the duplicated file descriptor must be greater than or equal to that value. The return value of fcntl is the duplicated file descriptor

lseek

- The lseek function is also used to change the file offset to a different value.
- Thus lseek allows a process to perform random access of data on any opened file.
- The prototype of lseek is

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

- On success it returns new file offset, and –1 on error.
- The first argument fdesc, is an integer file descriptor that refer to an opened file.
- The second argument pos, specifies a byte offset to be added to a reference location in deriving the new file offset value.
- The third argument whence, is the reference location.

Whence value	Reference location
SEEK_CUR	Current file pointer address
SEEK_SET	The beginning of a file
SEEK_END	The end of a file

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

- The output of the preceding code is:

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

stat, fstat :

- The stat and fstat function retrieves the file attributes of a given file.
- The only difference between stat and fstat is that the first argument of a stat is a file pathname, where as the first argument of fstat is file descriptor.
- The prototypes of these functions are

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

•The second argument to stat and fstat is the address of a struct stat-typed variable which is defined in the <sys/stat.h> header.

• Its declaration is as follows:

```
struct stat
 {
dev_t st_dev;               /* file system ID */
 ino_t st_ino;              /* file inode number */
mode_t st_mode;             /* contains file type and permission */
 nlink_t st_nlink;          /* hard link count */
 uid_t st_uid;              /* file user ID */
 gid_t st_gid;              /* file group ID */
 dev_t st_rdev;             /*contains major and minor device#*/
off_t st_size;              /* file size in bytes */
 time_t st_atime;           /* last access time */
time_t st_mtime;            /* last modification time */
 time_t st_ctime;           /* last status change time */ };
```

The return value of both functions is:

- 0 if they succeed
- -1 if they fail
- *errno contains an error status code*

ioctl (input/output control or control device) :

```
#include <unistd.h>
int ioctl(int fildes, int cmd, ...);
```

on success zero is returned,

On error -1 is returned

- system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls.

Ex: A CD-ROM device driver which can instruct a physical device to eject a disc would provide an ioctl request code to do that.

- ioctl is a bit of a rag–bag of things. It provides an interface for controlling the behavior of devices, their descriptors and configuring underlying services.

- Terminals, file descriptors, sockets, even tape drives may have ioctl calls defined for them and you need to refer to the specific device's man page for details.
- ioctl performs the function indicated by cmd on the object referenced by the descriptor fildes. It may take an optional third argument depending on the functions supported by a particular device.

Ex: int ioctl(int fd, SETFONT, ARIAL)

1.fd is file descriptor, the one returned by open
2.request is request code. e.g GETFONT will get current font from printer, SETFONT will set font on a printer.
3.third argument is void *. Depending on second argument, the third may or may not be present. e.g. if second argument is SETFONT, third argument may give font name as ARIAL.

umask

- The umask is a system variable that encodes a mask for file permissions to be used when a file is created.
- You can change the variable by executing the umask command to supply a new value.
- The value is a three-digit octal value. Each digit is the result of ORing values from 1, 2, or 4
- the following table. The separate digits refer to “user,” “group,” and “other” permissions, respectively.

Digit	Value	Meaning
1	0	No user permissions are to be disallowed.
	4	User read permission is disallowed.
	2	User write permission is disallowed.
	1	User execute permission is disallowed.
2	0	No group permissions are to be disallowed.
	4	Group read permission is disallowed.
	2	Group write permission is disallowed.
	1	Group execute permission is disallowed.
3	0	No other permissions are to be disallowed.
	4	Other read permission is disallowed.
	2	Other write permission is disallowed.
	1	Other execute permission is disallowed.

For example, to block “group” write and execute, and “other” write, the umask would be:

Digit	Value
1	0
2	2
	1
3	2

Values for each digit are ORed together; so the second digit will need to be 2 | 1, giving 3. The resulting umask is 032.

- To set the umask 077 type the following command at shell prompt:

```
$ umask 077
$ mkdir dir1
$ touch file
$ ls -ld dir1 file
```

Sample outputs:

```
drwx------ 2 vivek vivek 4096 2011-03-04 02:05 dir1
-rw------- 1 vivek vivek 0 2011-03-04 02:05 file
```

File and Record Locking

Linux system allow multiple processes to read and write the same file concurrently , but it results data in a file can be overridden by another process. To avoid this drawback linux supports file and record locking.

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

Record locking is used to describe the ability of a process to prevent other processes from modifying a region of a file while the first process is reading or modifying that portion of the file. It is **byte-range locking**, since it is a range of a file (possibly the entire file) that is locked.

➢To lock the entire file, we set l_start and l_whence to point to the beginning of the file and specify a length (l_len) of 0.

Types of locks:

1.Shared lock:

read locks are shard locks. Any no.of process can have read lock on the same object at a time.

2. Exclusive lock :

Write locks are exclusive locks. When one process puts write lock on the object, no other read lock or write lock is allowed on the same object.

Locking Regions

- Creating lock files is fine for controlling exclusive access to resources such as serial ports or infrequently accessed files, but it isn’t so good for access to large shared files. Suppose you have a large file that is written by one program but updated by many different programs simultaneously.
- This might occur if a program is logging some data that is obtained continuously over a long period and is being processed by several other programs. The processing programs can’t wait for the logging program to finish—it runs continuously—so they need some way of cooperating to provide simultaneous access to the same file.

- You can accommodate this situation by locking regions of the file so that a particular section of the file is locked, but other programs may access other parts of the file. This is called *file-segment, or file-region, locking*.Linux has two ways to do this: using the fcntl system call and using the lockf call.
- the fcntl and lockf locking mechanisms do not work together: They use different underlying implementations, so you should never mix the two types of call.

```
#include <fcntl.h>
int fcntl(int fildes, int command, ...);
```

- It's important to realize that this is a cooperative arrangement and that you must write the programs correctly for it to work. A program failing to create the lock file can't simply delete the file and try again. It might then be able to create the lock file, but the other program that created the lock file has no way of knowing that it no longer has exclusive access to the resource.

- fcntl operates on open file descriptors and, depending on the command parameter, can perform different tasks. The three command options of interest for file locking are as follows:
  - ❑ F_GETLK
  - ❑ F_SETLK
  - ❑ F_SETLKW
- When you use these the third argument must be a pointer to a struct flock, so the prototype is effectively this:
- **int fcntl(int fildes, int command, struct flock *flock_structure);**
- The flock (file lock) structure is implementation dependent, but it will contain at least the following members:
  - ❑ short l_type;
  - ❑ short l_whence;
  - ❑ off_t l_start;
  - ❑ off_t l_len;

The l_type member takes one of several values, also defined in fcntl.h.

- **F_RDLCK:** A shared (or "read") lock. Many different processes can have a shared lock on the same (or overlapping) regions of the file. If any process has a shared lock, then no process will be able to get an exclusive lock on that region. In order to obtain a shared lock, the file must have been opened with read or read/write access.
- **F_UNLCK:** Unlock; used for clearing locks
- **F_WRLCK:** An exclusive (or "write") lock. Only a single process may have an exclusive lock on any particular region of a file. Once a process has such a lock, no other process will be able to get any sort of lock on the region. To obtain an exclusive lock, the file must have been opened with write or read/write access.

- The l_whence, l_start, and l_len members define a region — a contiguous set of bytes — in a file. The l_whence must be one of SEEK_SET, SEEK_CUR, SEEK_END (from unistd.h). These correspond to the start, current position, and end of a file, respectively.
- l_whence defines the offset to which l_start, the first byte in the region, is relative. Normally, this would be SEEK_SET, so l_start is counted from the beginning of the file.
- The l_len parameter defines the number of bytes in the region.
- The l_pid parameter is used for reporting the process holding a lock;
- Each byte in a file can have only a single type of lock on it at any one time, and may be locked for shared access, locked for exclusive access, or unlocked.

The F_GETLK Command

- The first command is F_GETLK. It gets locking information about the file that fildes (the first parameter)has open. It doesn’t attempt to lock the file(It does only verification and for not applying locks. The calling process passes information about the type of lock it might wish to create, and fcntl used with the F_GETLK command returns any information that would prevent the lock from occurring.
- The values used in the flock structure are described in the following table:
- l_type : Either F_RDLCK for a shared (read-only) lock or F_WRLCK for an exclusive (write) lock
- l_whence: One of SEEK_SET, SEEK_CUR, or SEEK_END LCK
- l_start :The start byte of the file region of interest

- l_len: The number of bytes in the file region of interest
- l_pid: The identifier of the process with the lock
- A process may use the F_GETLK call to determine the current state of locks on a region of a file. It should set up the flock structure to indicate the type of lock it may require and define the region it’s interested in. The fcntl call returns a value other than -1 if it’s successful.
- If the file already has locks that would prevent a lock request from succeeding, it overwrites the flock structure with the relevant information. If the lock will succeed, the flock structure is unchanged. If the F_GETLK call is unable to obtain the information, it returns -1 to indicate failure.

The F_SETLK Command

- This command attempts to lock or unlock part of the file referenced by fildes. The values used in the flock structure (and different from those used by F_GETLK) are as follows:

l_type One of the following:

- **F_RDLCK : for a read-only, or shared, lock**
- **F_WRLCK : for an exclusive or write lock**
- **F_UNLCK : to unlock a region**
- **l_pid : Unused**
- As with F_GETLK, the region to be locked is defined by the values of the l_start, l_whence, and l_len fields of the flock structure. If the lock is successful, fcntl returns a value other than -1; on failure, -1 is returned.

The F_SETLKW Command

- The F_SETLKW command is the same as the F_SETLK command above except that if it can’t obtain the lock, the call will wait until it can. Once this call has started waiting, it will return only when the lock can be obtained or a signal occurs.

- **Write a program to lock a file using system call.**

(*lock-file.c*) Create a Write Lock with *fcntl*

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
 char* file = argv[1];
 int fd;
 struct flock lock;
 printf ("opening %s\n", file);
 /* Open a file descriptor to the file. */
 fd = open (file, O_WRONLY);
 printf ("locking\n");
 /* Initialize the flock structure. */
```

```
memset (&lock, 0, sizeof(lock));
 lock.l_type = F_WRLCK;
                                /* Place a write lock on the file. */
 fcntl (fd, F_SETLKW, &lock);

 printf ("locked; hit Enter to unlock... ");
                                /* Wait for the user to hit Enter. */
 getchar ();

 printf ("unlocking\n");
                                /* Release the lock. */
 lock.l_type = F_UNLCK;
 fcntl (fd, F_SETLKW, &lock);

 close (fd);
 return 0;
```

- Compile and run the program on a test file—say, /tmp/test-file—like this:

  ```
  % cc -o lock-file lock-file.c
  % touch /tmp/test-file
  % ./lock-file /tmp/test-file
  opening /tmp/test-file
  locking
  locked; hit Enter to unlock...
  ```

- Now, in another window, try running it again on the same file.

```
% ./lock-file /tmp/test-file
    opening /tmp/test-file
    Locking
```

- Note that the second instance is blocked while attempting to lock the file. Go back to the first window and press Enter:
- Unlocking
- The program running in the second window immediately acquires the lock.

File and Directory Maintenance or Management

- The standard libraries and system calls provide complete control over the creation and maintenance of files and directories.

chmod

- You can change the permissions on a file or directory using the chmod system call.
- This forms the basis of the chmod shell program.

syntax:

#include <sys/stat.h>

int chmod(const char *path, mode_t mode);

- The file specified by path is changed to have the permissions given by mode.
- The modes are specified as in the open system call, a bitwise OR of required permissions. Unless the program has been given appropriate privileges, only the owner of the file or a superuser can change its permissions.

chown

- A superuser can change the owner of a file using the chown system call.

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

- The call uses the numeric values of the desired new user and group IDs (called from getuid and getgid
- calls) and a system value that is used to restrict who can change file ownership.
- The owner and group of a file are changed if the appropriate privileges are set.

- **Change the owner of a file**

```
 # ls -l
    tmpfile -rw-r--r-- 1 himanshu family 0 2012-05-22 20:03 tmpfile
 # chown root tmpfile
# ls -l
tmpfile -rw-r--r-- 1 root family 0 2012-05-22 20:03 tmpfile
```

Change both owner and the group

```
# ls –l
     tmpfile -rw-r--r-- 1 root family 0 2012-05-22 20:03 tmpfile
 # chown himanshu:friends tmpfile
 # ls -l
    tmpfile -rw-r--r-- 1 himanshu friends 0 2012-05-22 20:03 tmpfile
```

unlink, link, and symlink

- You can remove a file using unlink.
- The unlink system call removes the directory entry for a file and decrements the link count for it.
- It returns 0 if the unlinking was successful,

  –1 on an error.
- You must have write and execute permissions in the directory where the file has its directory entry for this call to function.

```
#include <unistd.h>
int unlink(const char *path);
int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

- The link system call creates a new link to an existing file, path1. The new directory entry is specified by path2.
- You can create symbolic links using the symlink system call in a similar fashion.
- Note that symbolic links to a file do not increment a file’s reference count and so do not prevent the file from being effectively deleted as normal (hard) links do.

mkdir and rmdir

- You can create and remove directories using the mkdir and rmdir system calls.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

- The mkdir system call is used for creating directories and is the equivalent of the mkdir program.
- mkdir makes a new directory with path as its name.
- The directory permissions are passed in the parameter

- mode and are given as in the O_CREAT option of the open system call and, again, subject to umask.

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

- The rmdir system call removes directories, but only if they are empty

chdir and getcwd

- A program can navigate directories in much the same way as a user moves around the file system.
- As you use the cd command in the shell to change directory, so a program can use the chdir system call.

```
#include <unistd.h>
int chdir(const char *path);
```

- A program can determine its current working directory by calling the getcwd function.

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

- The getcwd function writes the name of the current directory into the given buffer, buf.
- It returns NULL if the directory name would exceed the size of the buffer (an ERANGE error), given as the parameter size.
- It returns buf on success.
- getcwd may also return NULL if the directory is removed (EINVAL) or permissions changed (EACCESS) while the program is running.

Scanning Directories

- The directory functions are declared in a header file dirent.h.
- They use a structure, DIR, as a basis for directory manipulation. A pointer to this structure, called a *directory stream (a DIR *), acts in much the* same way as a file steam (FILE *) does for regular file manipulation.
- Directory entries themselves are returned in dirent structures, also declared in dirent.h, because one should never alter the fields in the DIR structure directly.

opendir

- The opendir function opens a directory and establishes a directory stream.
- If successful, it returns a pointer to a DIR structure to be used for reading directory entries.

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

- opendir returns a null pointer on failure.
- Note that a directory stream uses a low-level file descriptor to access the directory itself, so opendir could fail with too many open files.

readdir

- The readdir function returns a pointer to a structure detailing the next directory entry in the directory stream dirp. Successive calls to readdir return further directory entries.
- On error, and at the end of the directory, readdir returns NULL.
- POSIX-compliant systems leave errno unchanged when returning NULL at end of directory and set it when an error occurs.

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

- Note that readdir scanning isn’t guaranteed to list all the files (and subdirectories) in a directory if there are other processes creating and deleting files in the directory at the same time.
- The dirent structure containing directory entry details includes the following entries:
    - ❑ ino_t d_ino: The inode of the file
    - ❑ char d_name[]: The name of the file

telldir

- The telldir function returns a value that records the current position in a directory stream.
- You can use this in subsequent calls to seekdir to reset a directory scan to the current position.

```
#include <sys/types.h>
#include <dirent.h>
long int telldir(DIR *dirp);
```

seekdir

- The seekdir function sets the directory entry pointer in the directory stream given by dirp.
- The value of loc, used to set the position, should have been obtained from a prior call to telldir.

```
#include <sys/types.h>
#include <dirent.h>
void seekdir(DIR *dirp, long int loc);
```

closedir

- The closedir function closes a directory stream and frees up the resources associated with it.
- It returns 0 on success and –1 if there is an error.

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

Assignments

- Stream Errors
- Kernel support for files