



Course	System calls and File Management System	
Module Day 2		3 Hours
	<p>At the end of this module you will be able to know:</p> <ul style="list-style-type: none"> ▪ System calls and anatomy of system calls. ▪ Different type of system calls ▪ What is a file, File system and File management system? ▪ How files will be Stored in Hard disk, and File system layout. ▪ About file permissions and file attributes, ▪ What is Virtual file system and why do we need? ▪ About /proc file system. How to access /proc and Advantages of /proc file system, ▪ The use of file locking mechanism, ▪ System calls related to Files and directories. 	

Session Plan	
1	What is a system call, Role of a system call, Tracing a system call Available system calls
2	node structure, File Structure, File descriptor table
3	File descriptors, File I/O system calls - open, close, read, write, lseek

	Module Overview
---	-----------------

- ✓ System calls and anatomy
- ✓ Different types of system calls
- ✓ What is File System and File?
- ✓ Introduction to FMS
- ✓ Arrangement of files in hard disk
- ✓ File system layout
- ✓ File permissions
- ✓ Storing data blocks in to Inode structure
- ✓ Virtual file system
- ✓ /proc file system
- ✓ System calls related to Files with examples
- ✓ File locking
- ✓ System calls related to Directories with examples



System calls

System calls are low level functions that make operating system available to applications via a defined API (Application Programming Interface).

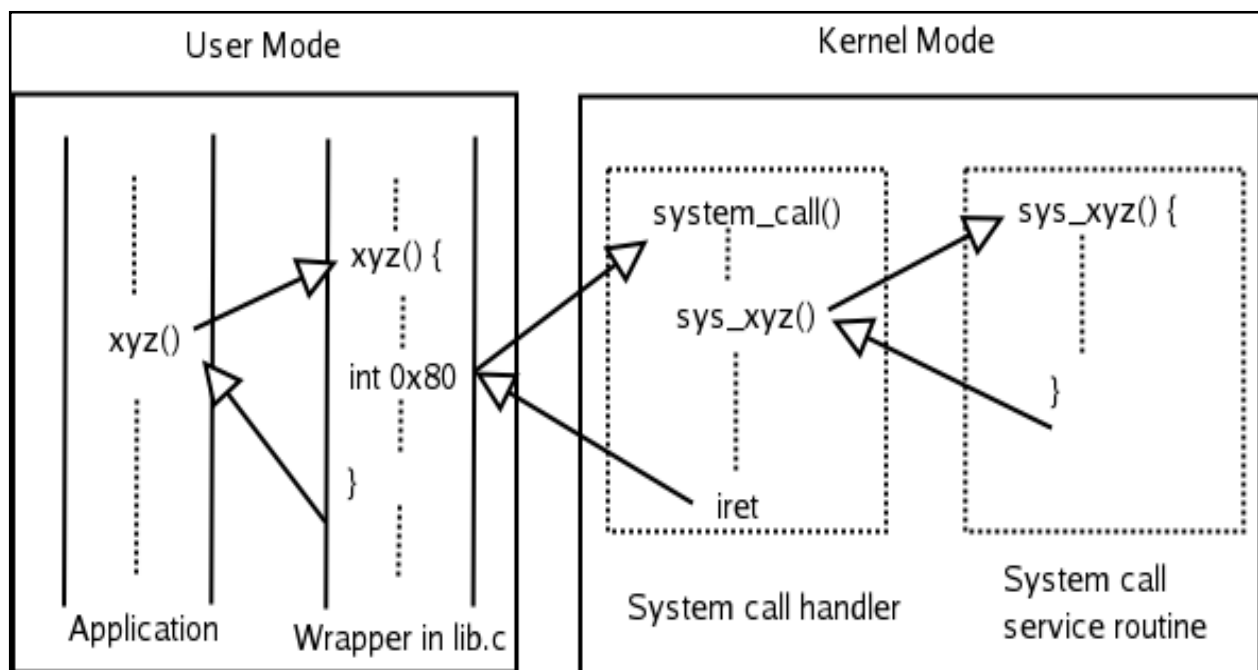
System calls represent the interface the kernel presents to user applications. In Linux all low-level I/O is done by reading and writing file handles, regardless of what particular peripheral device is being accessed—a tape, a socket, even your terminal, they are all files. Low level I/O is performed by making system calls.

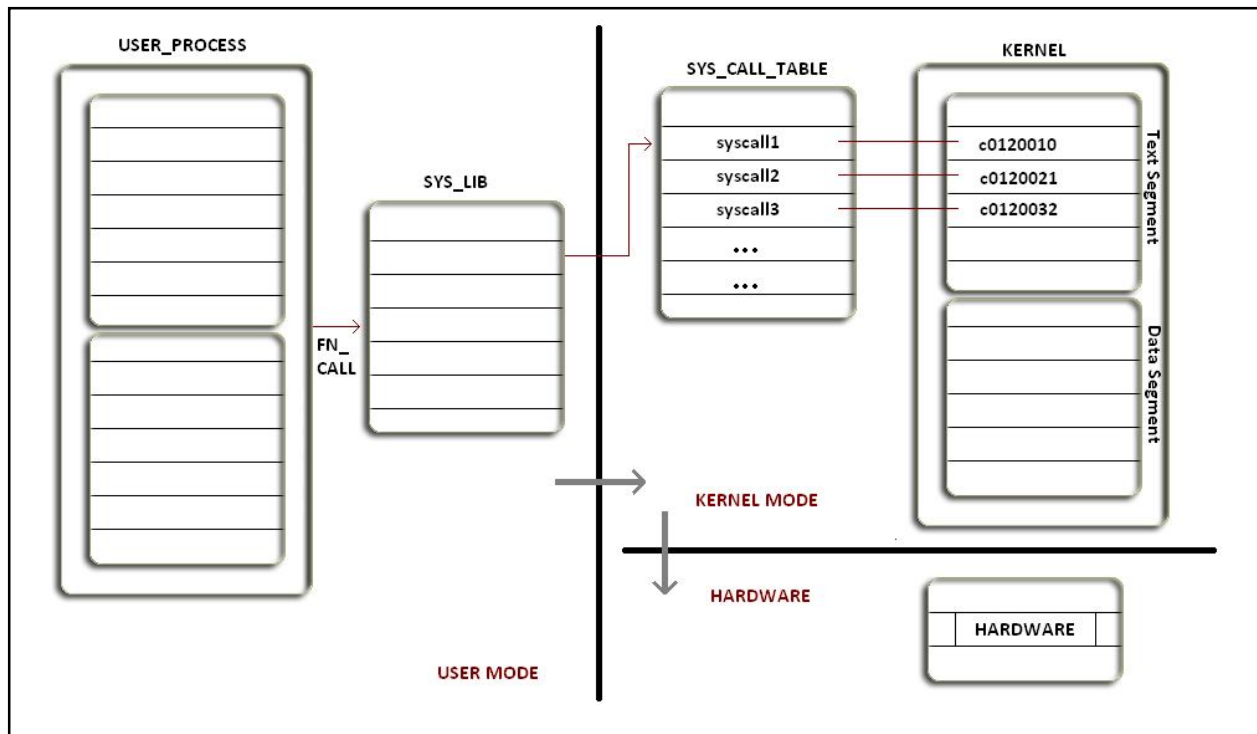
System calls acts as entry point to OS kernel. There are certain tasks that can only be done if a process is running in kernel mode.

Ex: tasks interacting with hardware, If a process wants to do such kind of task then it would require itself to be running in kernel mode which is made possible by system calls.



Anatomy of system call





A System Call is an explicit request to the kernel made via a software interrupt. On x86 machines, the interrupt call '0x80' call to a system call handler. The system call handler in turn calls the system call interrupt service routine (ISR).

Following steps are performed before triggering a system call on x86:

1. Put the system call number in EAX register.
2. Set up the arguments to the system call in EBX, ECX, etc.
3. Call the relevant interrupt (for Linux, 80h).
4. The result is usually returned in EAX.

On x86, there are six registers that are used for the arguments that the system call takes. The first argument goes in EBX, the second in ECX, then EDX, ESI, EDI, and finally EBP. Files (in Linux everything is a file) are referenced by an integer file descriptor

We will learn 5 basic system calls:

1. sys_open
2. sys_close
3. sys_read
4. sys_write
5. sys_lseek.



Basic system calls

sys_open - open a file

System call number (in EAX): 5

Arguments:

EBX: The pathname of the file to open/create

ECX: set file access bits (can be OR'd together):

O_RDONLY open for reading only

O_WRONLY open for writing only

O_RDWR open for both reading and writing

O_APPEND open for appending to the end of file

O_TRUNC truncate to 0 length if file exists

O_CREAT create the file if it doesn't exist

EDX: set file permissions.

Returns in EAX: file descriptor.

On errors: -1.

sys_close - close a file by file descriptor reference

System call number (in EAX): 6

Arguments:

EBX: file descriptor.

Returns in EAX:

On errors: -1.

sys_read - read up to count bytes from file descriptor into buffer

System call number (in EAX): 3

Arguments:

EBX: file descriptor.

ECX: pointer to input buffer.

EDX: buffer size, max. count of bytes to receive.

Returns in EAX: number of bytes received.

On Errors: -1 or 0 (no bits read).

sys_write - write (up to) count bytes of data from buffer to file descriptor reference.

System call number (in EAX): 4

Arguments:

EBX: file descriptor.

ECX: pointer to output buffer.

EDX: count of bytes to send.

Returns in EAX: number of bytes written successfully

On Errors: -1 or 0 (no bits written).

sys_lseek - change file pointer.

System call number (in EAX): 19

Arguments:

EBX: file descriptor.

ECX: offset, given in number from the following parameter.

EDX: either one of

SEEK_SET 0 - beginning of file.

SEEK_CUR 1 - current position.

SEEK_END 2 - end of file.

Returns in EAX: current file pointer position.

On Errors: beginning of file position.

Error Handling

System calls set a global integer called `errno` on error.

The constants that `errno` may be set to are (partial list):

EPERM: operation not permitted.

ENOENT: no such file or directory (not there).

EIO: I/O error

EEXIST: file already exists.

ENODEV: no such device exists.

EINVAL: invalid argument passed.

Tracing a system call

Tracing of a system call means analyzing list of system calls used in a command or in executed user program.

This is helpful for debugging of an error or to know the internal flow of program.

This can be accomplished by using **strace** command.

Syntax: **strace <command>**

Ex: `strace ls`

`strace ./a.out`

```
[utl@ubuntu-Desktop: Linux] $./a.out
Hello world
[utl@ubuntu-Desktop: Linux] $strace ./a.out
execve("./a.out", [ "./a.out" ], [ /* 43 vars */ ]) = 0
brk(0) = 0x95bb000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7700000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=85720, ...}) = 0
mmap2(NULL, 85720, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb76eb000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"... , 512)
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7538000
mmap2(0xb76e5000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0xb76e5000
mmap2(0xb76e8000, 11036, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb76e8000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7537000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7537900, limit:1048575, seg_32bit:1, ts:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb76e5000, 8192, PROT_READ) = 0
mprotect(0xb76e8000, 4096, PROT_READ) = 0
mprotect(0xb7723000, 4096, PROT_READ) = 0
munmap(0xb76eb000, 85720) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb76ff000
write(1, "Hello world\n", 12) = 12
exit_group(0) = ?
```



Different types of system calls

File management system call:

System calls responsible for file accessing or management

open(): opens a file.

close(): closes a file.

read(): Used to read the content of a opened file to a buffer.

write(): Used to write some data in buffer to a file.

link() : links two files.

dup(): duplicates a file descriptor.

Process management system calls.

System calls responsible for process management.

fork () :- Used to create a new process

exec() :- Execute a new program

wait():- wait until the process finishes execution

exit():- Exit from the process

getpid():- get the unique process id of the process

getppid():- get the parent process unique id

nice():- to bias the existing property of process

Communication system calls

Used to communicate between processes.

pipe(): Used to create a pipe used to communicate between two process.

mkfifo(): used to create a FIFO.

msgsnd(): used to send a message to message queue.

msgrcv(): used to receive a message from message queue.

socket(): used to create a socket

listen(): waiting for an incoming request.

Error and signal handling system calls

sys_alarm(): set an alarm clock for delivery of a signal

signal(): used to handle a signal generated.

sys_pause(): used to wait for an signal.

kill(): used to send a signal from one process to another process.

strerror(): used to check for error and if any returns the type of error using string.

err(): prints the error in formatted format.

errno(): returns the last error number generated.

System call Vs library functions

A library function is linked to the user program and executes in user space while a system call is not linked to a user program and executes in kernel space.

A library function execution time is counted in user level time while a system call execution time is counted as a part of system time.

Library functions can be debugged easily using a debugger while System calls cannot be debugged as they are executed by the kernel.



Introduction to File Management System

File: A collection of data or information that has a name. All information stored in a computer hard disk must be in a file. Files are collection of data items stored on disk.

Files are always associated with devices like hard disk, floppy disk etc. All information about file is stored in a data structure called **Inode Structure**.

Following are general rules for both Linux and UNIX like systems:

- All file names are case sensitive. So filename sample.txt, Sample.txt, SAMPLE.txt all are three different files.
- You can use upper and lowercase letters, numbers, "." (dot), and "_" (underscore) symbols.
- You can use other special characters such as blank space, but they are hard to use and it is better to avoid them.
- In short, filenames may contain any character except / (root directory), which is reserved as the separator between files and directories in a pathname. You cannot use the null character.
- No need to use. (dot) in a filename. Some time dot improves readability of filenames. And you can use dot based filename extension to identify file.
For example:
.sh = Shell file.
.tar.gz = Compressed archive.
- Most modern Linux and UNIX limit filename to 255 characters (255 bytes). However, some older version of UNIX system limits filenames to 14 characters only.
- A filename must be unique inside its directory. For example, inside /home/linux directory we cannot create a demo.txt file and demo.txt directory name. However, other directory may have files with the same names.

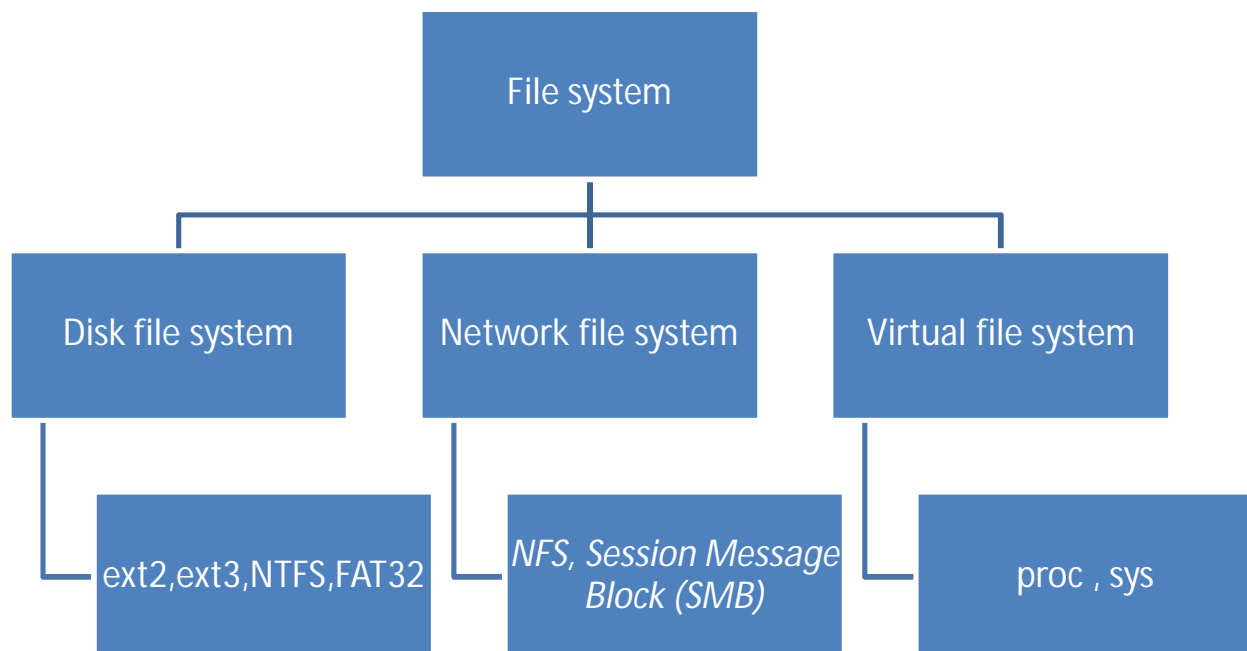
Directory: Directory is a group of files used to organize your data files. There are two types of directories are there in Linux:

- **Root Directory:** Strictly speaking, there is only one root directory in your system, which is denoted by / (forward slash). It is root of your entire file system and cannot be renamed or deleted.
- **Sub Directory:** It is a directory under root directory which can be created, and renamed by the user.

File System: The way of arranging files in Memory (Hard disk) is called File System. In a computer, a file system is the way in which files are named and where they are placed logically for storage and retrieval. All Operating Systems have file systems in which files are placed in a hierarchical (tree) structure. The following are a few of the file systems.

- / - Special file system that incorporates the files under several directories including /dev, /sbin, /tmp etc,
- /usr - Stores application programs,
- /var - Stores log files, mails and other data,
- /tmp - Stores temporary files.

Linux supports numerous file system types. They can be broadly classified in to 3 groups.



- **Ext2:** This is like UNIX file system. It has the concepts of blocks, inodes and directories.
- **Ext3:** It is ext2 file system enhanced with journaling capabilities. Journaling allows fast file system recovery. Supports POSIX ACL (Access Control Lists).
- **ISO fs** (iso9660): Used by CDROM file system.
- **Proc fs:** The proc file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime.
- **NFS:** Network file system allows many users or systems to share the same files by using a client/server methodology. NFS allows sharing all of the above file systems.

- Linux also supports Microsoft NTFS, vfat, and many other file systems. You can find out what type of file systems currently mounted with **mount** command:

File Management System: This is used to manage the all file systems in the hard disk. File management describes the fundamental methods for naming, storing and handling files. We can call FMS as File manager. The file manager handles the all files on secondary storage media. To perform these tasks file manager must:

- be able to identify the numerous files by giving unique names to them,
- Maintain a list telling where exactly each file is stored, how many sectors on the medium it occupies, and in which order those sectors (Hard disk is divided as sectors. Each sector size is typically 512bytes) make up the file,
- provide simple and fast algorithms to read and write files in cooperation with the device manager,
- give and deny access rights on files to users and programs,
- allocate and de-allocate files to processes in cooperation with the process manager (PMS),
- Provide users and programs with simple commands for file handling and etc.

In Linux FMS will support you to work with different File Systems by using Virtual File System. We will discuss this topic in coming sections.



Arrangement of Files in Hard Disk

In Linux a file system is divided into two categories:

- **User data:** As shown in the below figure, typically 75% of the hard disk will be reserved for data blocks. This store the actual data contained in files.
- **Meta data:** This stores the file system structural information such as Super blocks, inode structures, and directories.

Linux Partitions: In the above diagram, the entire hard disk is divided in to several partitions. Disk partitioning is the creation of separate divisions of a hard disk drive. Once a disk is divided into several partitions, directories and files of different categories may be stored in different partitions. Many new Linux sys admin (or Windows admin)

create only two partitions / (root) and swap for entire hard drive. This is really a bad idea. You need to consider the following points while partitioning the disk:

- **Security:** Separation of the operating system files from user files may result into a better and secure system.
- **Ease of use:** Make it easier to recover a corrupted file system or operating system installation.
- **Testing:** Boot multiple operating systems such as Linux, Windows and FreeBSD from a single hard disk. And many more.

In Linux there are some file systems that need their own partitions like:

- **/usr:** This is where most executable binaries, the kernel source tree and much documentation will be placed.
- **/var:** This is where spool directories such as those for mail and printing go. In addition, it contains the error log directory.
- **/tmp:** This is where most temporary data files stored by apps.
- **/boot:** This is where your kernel images and boot loader configuration will be placed.
- **/home:** All users will be under this directory.



File System Layout

The file system layout will be having the different sections as shown in the below figure.

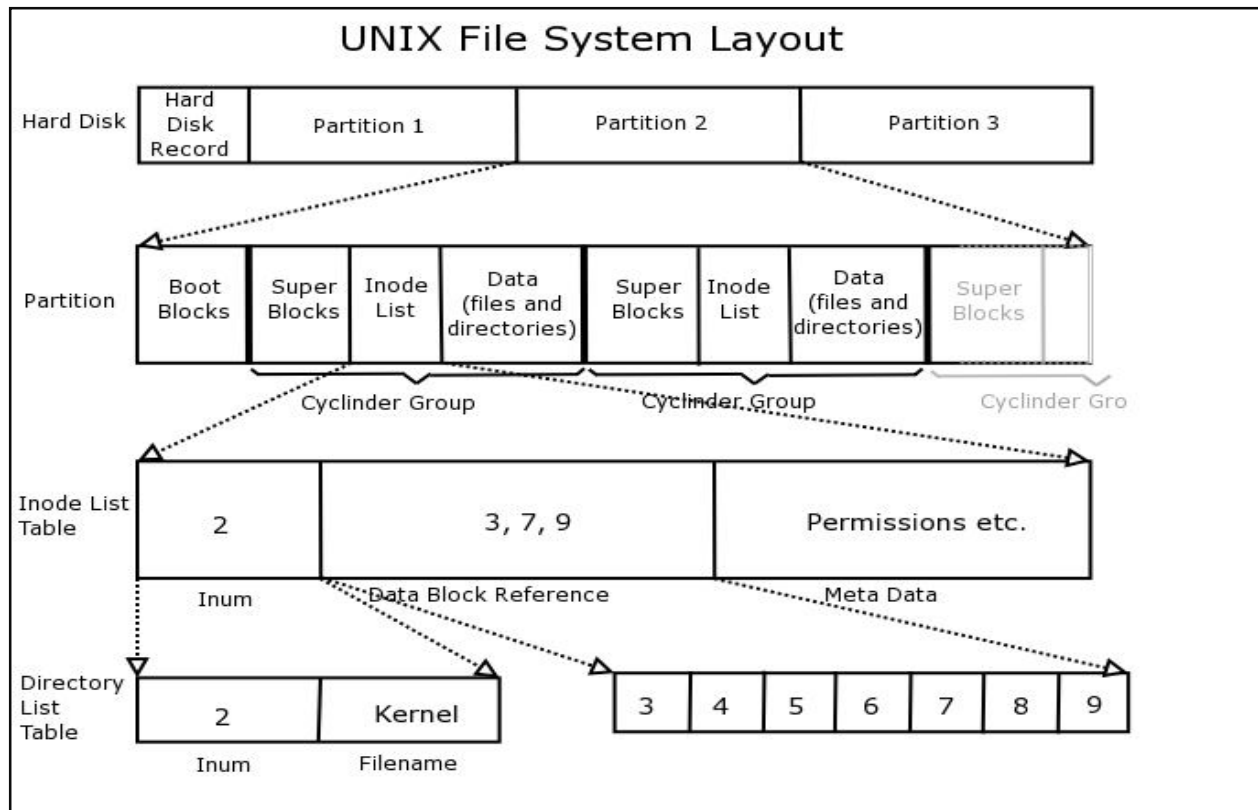


Fig: File system Layout

Boot Block: Beginning of the file system, typically first sector and contains the boot strap code that is read into the machine to boot or initialize OS.

Super Block: Each file system is different and they have type like ext2, ext3 etc. Further each file system has size like 5 GB, 10 GB and etc. In short each file system has a superblock, which contains information about file system such as:

- File system type
- Size
- Status
 - How large it is, number of files it can stores, where to find free place on the file and other system.
- Information about other metadata structures:
 - Having linked list of Free Inodes, Allocated Inodes, Free Data Blocks, and Allocated Data Blocks of that partition.

Inode Block: Under Linux, each any every file will be having the following attributes:

- Permissions,

- Owner,
- File type,
- Groups,
- File size,
- Time stamp:
 - File access time,
 - Creation time,
 - Change time.
- Number of links
- Address of data blocks, etc.

Inode: An inode is a data structure on a traditional Unix-style (Linux) file system such as ext3 or ext4. An inode stores basic information about a regular file, directory, or other file system object. In short the inode identifies the file and its attributes. In Linux each file in the file system is represented by an inode. So every file will be having corresponding Inode number. To check the inode number of a file, execute the following command on shell:

< shell >\$ ls -li file_name

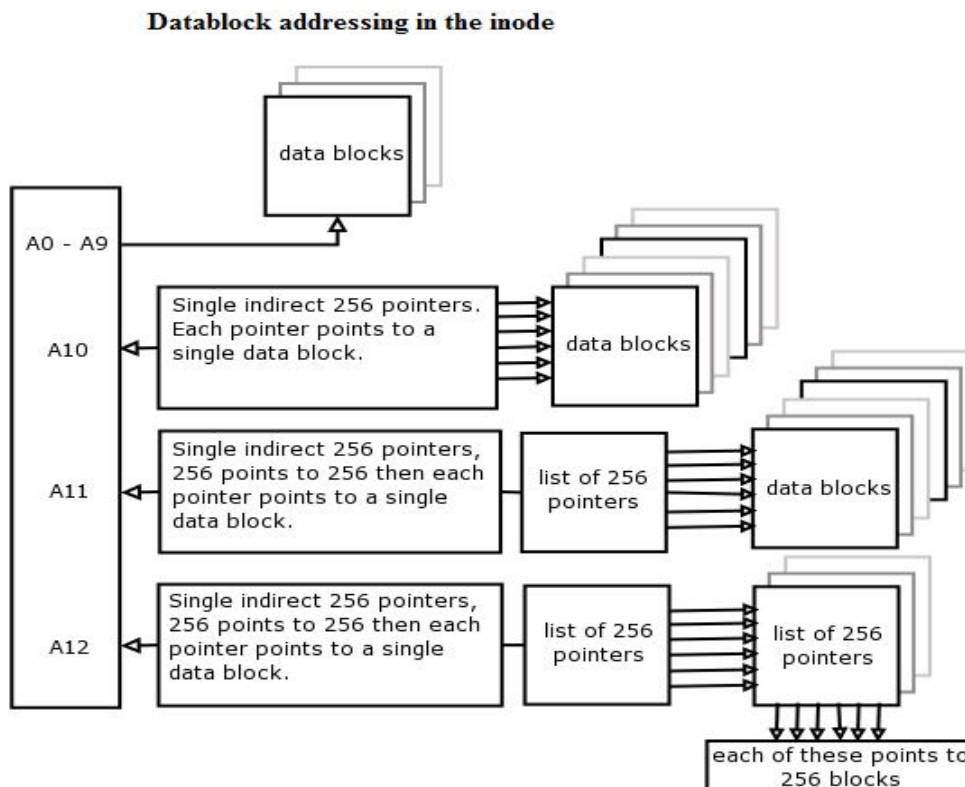
File Types: In Linux a file may have the following types:

- Regular file
- directory file
- symbolic link
- Block-oriented Device file
- Character-oriented Device file
- Pipe and named pipe
- Socket

File Permissions: When you create a file under Linux System, the default file permissions will be given to that file by administrator. In Linux all users will be categorised as Owner, Groups, Others for particular file. Each user will be having different permissions to access that file. To check all file permissions regarding with particular file, execute the following command on shell:

< shell >\$ ls -l sample.txt

Inode addressing: All file information is stored in data blocks, and all references to that data blocks will be there in corresponding inode of that file.



Every inode structure will contain group of pointers referring to data blocks of the file. The inode block numbers work in a group of 13. The first 10 point directly to a data block which contain the actual file data - this is called direct addressing = 10Kb.

Then the 11th block points to a data block that contains 256 pointers, each of those pointers point to a single address block as above. This is called single indirect block addressing = 256Kb.

The 12th block points to a data block that has 256 pointers and each of those pointers point to a data block with 256 pointers, and each of those pointers to a data block as per point 1 above. This is called double indirect block addressing = 64Mb.

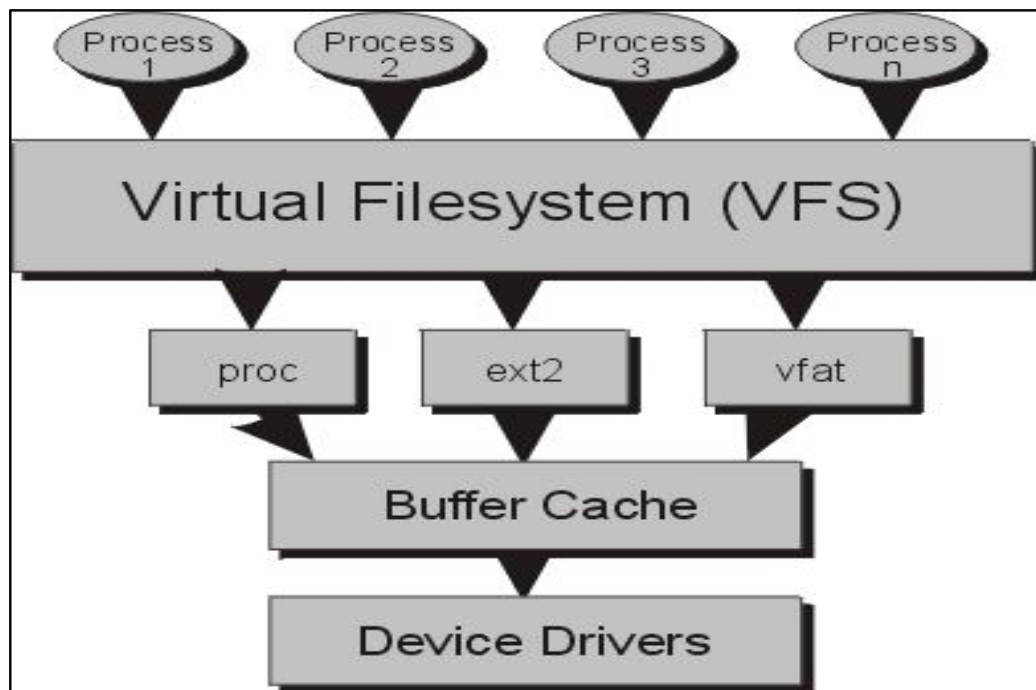
The 13th point has a triple indirect setup where the 256 pointers each point to 256 pointers that each point to 256 pointers that each point to a data block as per point 1 above = 16GB.

So the max size is actually 16GB, BUT this is limited by the inode sizing not by the structure of the operating system.



Virtual File System

Definition: Virtual File system is a kernel software layer that handles all system calls related to different file systems. Its main strength is providing a common interface to the different file systems.



- The above diagram shows the relation between the kernels VFS and other different file systems. This file system can also be called as Virtual File System Switch.
- VFS is a kernel software layer that handles all system calls related to file systems. Its main strength is providing a **common interface** to several kinds of file systems.
- Without VFS, it is not possible to communicate with the different File Systems.
- Every file system on the hard disk should register itself with VFS.
- To manage the all file systems that are mounted at any given time, it maintains data structures that describe the whole file system.
- The VFS also keeps a cache of directory lookups so that the inodes for frequently used directories can be quickly found.
- It is not possible to communicate with different file systems, without VFS in the system.



/proc File System

When you execute the below command in shell prompt:

```
< shell >$ mount -l
```

It will display the all file systems which are currently mounted to the system. And you can identify the /proc file system as shown below:

```
proc on /proc type proc (rw,noexec,nosuid,nodev)
```

Notice that the field **nodev** indicates that this file system is not associated with a hardware device such as disk drive. Files in the /proc file system don't correspond to actual files on a physical device. Instead, they are magic objects that behave like files but provide access to parameters, data structures, and statistics in the kernel.

The “contents” of these files are not always fixed blocks of data, as ordinary file contents are. Instead, they are generated on the fly by the Linux kernel when you read from the file. Physically all the file systems are present in the hard disk permanently. But /proc file system will be created in the Hard disk and after some time the data in that files system will be deleted automatically.

Accessing the file /proc/meminfo will likely give you different results each time, because memory usage is nearly always fluctuating.

How to use: proc must implement a directory structure, and the file contents within, it must then define a unique and persistent inode number for each directory and files it contains. It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode. When data is read from one of these files, proc collects the appropriate information, formats it into text form and places it into the requesting process's read buffer.

Examples: The Proc File System can be used to gather useful information about the system and the running kernel. Some of the important files are listed below:

- **/proc/meminfo** - information about the physical RAM, Swap space etc.
- **/proc/mounts** - list of mounted file systems
- **/proc/devices** - list of available devices
- **/proc/filesystems** - supported file systems
- **/proc/modules** - list of loaded modules
- **/proc/version** - Kernel version



Working with files (System calls)

System Call: Processes access kernel resources using system calls. These are C functions developed as part of Kernel. To access any internal structure of kernel, system calls are the way. There are several system calls provided to access kernel resources. Most of system calls will come under three categories as given below.

- File System,
- Process Management and
- Inter process communication.

Now let us discuss the system calls related to files.

Required header files:

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

Creating a file: To create a new file, we have to use “creat” system call. The prototype of “creat” system call is given below.

```
int creat(const char *file, int perms);
```

file: File name to be created,

perms: initial permissions to the new file.

If the file name already exists, it is truncated and new one is created. On success of creating new file, returns file descriptor. On failure -1 will be returned.

EX: /***** WAP to create a file with name “sample.txt” *****/

```
int main()
{
    int fd;
    fd = creat( "sample.txt", 0744);
    if(fd == -1) {
        printf("fails to create a file\n");
        exit(0);
    }
    Printf(" sample.txt is created successfully in current
    directory\n");
}
```

Opening a file: A system call “open” is used to open a existing file. And also we can create & open a file using this system call by providing related flags. Here prototype of this system call is given below.

```
int open(const char *file, int modes, int perms);
```

file: file name to open
modes: The file has to be open in which mode:
0 or O_RDONLY for reading
1 or O_WRONLY for writing
2 or O_RDWR for read and write
64 or O_CREAT for creating.
Perms: different permissions to different users.

If we want to give more than one modes, we have to use ‘|’ symbol. This system call returns file descriptor on success, and -1 on error.

EX: /***** create and open “sample.txt” in read & write modes using open call *****/

```
int main()  
{  
    Int fd;  
    fd = open("sample.txt", O_CREAT | O_RDWR, 0744);  
    if(fd == -1) {  
        printf("error in opening a file:\n");  
        exit(0);  
    }  
    return 0;  
}
```

Writing to file: “write” system call is used to write data to file. The prototype of this system call is given below:

```
size_t write(int fd, const void *buf, size_t nbytes);
```

fd: file descriptor of the opened file or created file,
buf: has contents which will be written to file. Must give address in this field,
nbytes: Number of bytes to write to file from buf address

This will return -1 on failure, and on success returns number of bytes written successfully to the file.

EX: /***** Create a file "sample.txt" and write hello in it. *****/

```
int main()
{
    int fd;
    fd = creat( "sample.txt", 0744);    // file will be opened
in write mode only.
    if(fd == -1) {
        printf("error in creating a file\n");
        exit(0);
    }
    write(fd, "hello", 5);
    return 0;
}
```

Note: when you create a process, three default files (**stdin**, **stdout**, **stderr**) will be open for that process. Here **stdin** means input (keyboard) to the process; **stdout** means output (screen) of the process and **stderr** means error messages in the process has to sent to screen. The default file descriptors of these files are:

FD	Name
0	Stdin
1	Stdout
2	Stderr

The corresponding file descriptors will be loaded to the file descriptor table in that process. For that process we can open maximum 256 files (including stdin, Stdout, stderr) or we can save 256 file descriptors in that FD table.

FD	Name
0	Stdin
1	Stdout
2	Stderr
.	.
.	.
.	.
255	.

Fig: FD Table

The above shown FD table will be created for new process. When you open any file in that process like:

```
fd = open("file_name", O_RDWR, 0777);
```

Now fd will be added to FD table of that process and the value assigned for fd is 3. Now FD table looks like this:

FD	Name
0	Stdin
1	Stdout
2	Stderr
3	file_name
.	.
.	.
255	.

EX: /***** WAP to write hello to monitor screen *****/

```
int main()
{
    write(1, "hello", 5);
    return 0;
}
```

Reading from file: "read" system call is used to read data from file. The prototype of this system call is given below:

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

fd: file descriptor of the opened file only,

buf: To save the data read from file,

nbytes: number of bytes to read from file.

On success returns number of bytes read successfully, 0 on end of file, and on failure returns -1.

Note: Parameters to read and write functions are same. But read system call reads data from file and keeps in buffer and write system call writes data from buffer to file.

Ex: /***** WAP to open already existing file and read 1st character and display *****/

```

int main()
{
    int fd; char ch;
    fd = open("sample.txt", O_RDONLY, 0744);
    if(fd == -1) {
        printf("error in opening a file\n");
        exit(0);
    }
    read(fd, &ch, 1);
    printf("%c\n", ch);
    return 0;
}

```

Closing a file: To close a file, we have to use “close” system call. The prototype is given below:

<pre>int close(int fd);</pre>

fd: file descriptor of a file which is to be closed.
Returns 0 on success and -1 on failure.

Note: In above example programs we are using file and closing. If we are not closing a file, that will be closed automatically after completion of process. If you want to close a file between the executions of program, you have to use this system call.

Ex: /** WAP to create a file “sample.txt”, write hello to it and close the file. Open the same file for reading and display the contents character by character till the end.*****/

```

int main()
{
    int fd, n; char ch;
    fd = creat("sample.txt", 0744);
    if(fd == -1) {
        printf("error in creating a file\n");
        exit(0);
    }
    write(fd, "hello", 5);
    close(fd);
    fd = open("sample.txt", O_RDONLY, 0744);
    if(fd == -1) {
        /**Remove these lines and

```

```

        printf("error in opening file\n");
        exit(0);
    }
    while( (n = read(fd, &ch, 1)) > 0)
        printf("%c\n", ch);
    close(fd);
    return 0;
}
//*****check output
//*****
//*****

```

Note:

- When you open a file the offset (cursor) in the file will be at beginig of the file.
- For writing 1byte data to file, the offset will be incrementing 1byte forward.
- For reading 1byte data from file, the offset will be incrementing 1byte forward.

In the above program remove the lines and check the output of the program. Here we will not get output as "hello", because after write operation the offset in the file will be at end of file. So when we are closing and opening file, the offset will be at beginning of the file.

Seeking offset in file: If you want to read middle of the data in file, we have to move the offset in file. To-do this we have to use "lseek" system call. The prototype is given below.

```

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

```

fd: file descriptor of opened file
offset: bytes to seek offset from whence (+ve or -ve value)
whence: from where to seek offset in the file.
0: from beginning of the file
1: from current position
2: from end of file

Returns offset value from beginning of the file on success, -1 on failure.

Ex: /***** WAP to read the 4th character in the file "sample.txt" *****/

```

int main()
{
    int fd; char ch;
    fd = open( "sample.txt", O_RDWR, 0744);
    if( fd == -1) {
        printf("error in opening file\n");
        exit(0);
    }
}

```



```

    }
    write(fd, "hello prabha...", 15);
    lseek(fd, 4, 0);
    read(fd, &ch, 1);
    printf("4th character is: %c\n", ch);
    close(fd);
    return 0;
}

```

Checking Accessibility to file: To check accessibility permissions of the file, we have to use "access" system call. The prototype is given below:

```
int access( const char *file_name, int mode);
```

file_name: The file name to check accessibility.

Mode: permissions to check accessibility. This may be:

F_OK:	file Exist or not
R_OK:	read permissions
W_OK:	write permissions
X_OK:	execute permissions

Returns 0 on success and -1 on failure. You can give more than one mode to check accessibility by using ' | ' symbol.

Ex: /***** WAP to check if /etc/passwd can be accessed in read & write mode *****/

```

int main()
{
    int a;
    a = access( "/etc/passwd", R_OK | W_OK);
    if( a == 0)
        printf("/etc/passwd file is read & write
permitted\n");
    else
        printf("/etc/passwd file is not read & write
permitted\n");
    return 0;
}

```

Creating link to file: link system call is used for giving another name to a file. After successful of this call, the file will have one more link. The prototype of this call is given below:

```
int link( unsigned char *src, char *dst);
```

src: Source file. This must exist

dst: src will be having one more name as dst. This must not exist

This system call fails if dst is already exist. On failure, this will return -1 and on success 0. After successful link, the modification on src will reflect in dst and vice versa. "ln" command does same thing as link call does. But "ln" is used in command line.

Ex: /***** WAP to create a link for "sample.txt" to "rebel.txt" *****/

```
int main()
{
    link( "sample.txt", "rebel.txt");
    return 0;
}
```

Note: After this program, both files will be sharing the same inode numbers. To check the inode numbers of both files execute the below command:

<shell>\$ ls -li

Unlink the file: To unlink or to delete the file, we have to use "unlink" system call. The prototype is given below.

```
int unlink(const char *file);
```

file: file to be unlinked.

Returns 0 on success and -1 on error.

Ex: /***** WAP to have a file "a.txt" link to "b.txt", and unlink "a.txt" *****/

```
int main()
{
    if ( ( link("a.txt", "b.txt")) == -1) {
        printf("error in creating link\n");
        exit(0);
    }
    unlink("a.txt");
    return 0;
}
```

I/O Redirection: As you know already that, input to a process is always from file descriptor Zero (0) and output of a process is sent to file descriptor one (1). Now to redirect I/O of a process, we have to replace the new file descriptor in place of 0 and 1. To replace new file descriptor, first we have to close the existing file descriptor. To close and replace **close** and **dup** system calls are used respectively. The syntax of dup function is shown below.

Dup is a system call used for duplicating a file descriptor. This call is widely used in implementing redirection and piping.

<pre>int dup(int fd);</pre>

Here fd -> file descriptor whose duplicate is done.

On success this system call returns the new file descriptor and returns -1 on error.

Ex: /***** Analyse the below two programs *****/

```
Int main()
{
    Int a;
    Close(0);
    Printf("\nenter the value of a: ");
    Scanf("%d", &a);
    Printf("\na = %d\n", a);
    Return 0;
}
```

```
Int main()
{
    Int a;
    Close(1);
    Printf("\nenter the value of a: ");
    Scanf("%d", &a);
    Printf("\na = %d\n", a);
    Return 0;
}
```

Ex: /***** WAP to redirect standard input of a process from a file *****/

```
Int main()
{
    Int a, fd;
    Close(0);
```

```

    fd = open("temp", 0);           // when temp file is having 25
    dup(fd);
    Printf("\nenter the value of a: ");
    Scanf("%d", &a);
    Printf("\na = %d\n", a);
    Return 0;
}

```

Ex: /***** WAP to redirect standard output of a process to a file *****/

```

Int main()
{
    Int fd;
    Close(1);
    fd = open("temp", 1);           // when temp is a empty file
    dup(fd);
    Printf("\n this line will print to file\n ");
    Return 0;
}

```



File Locking

File locking is a mechanism which allows only one process to access a file at any specific time. By using file locking mechanism, many processes can read/write a single file in a safer way.

Why do we need?

- Process "A" opens and reads a file which contains account related information. Process "B" also opens the file and reads the information in it. Now Process "A" changes the account balance of a record in its copy, and writes it back to the file.
- Process "B" which has no way of knowing that the file is changed since its last read, has the stale original value. It then changes the account balance of the same record,

and writes back into the file. Now the file will have only the changes done by process "B".

To avoid such issues, locking is used to ensure serialization.

System call: For locking and unlocking a file, we have to use "flock" system call. The prototype is given below.

<pre>int flock(int fd, int flag);</pre>
--

fd: file descriptor of the file which is to be lock or unlock.

flag: Locking: LOCK_EX

Unlocking: LOCK_UN

Returns 0 on success, and -1 on error.

Ex: /***** WAP to lock a file before using and unlock after using *****/

```
int main()
{
    int fd, a;  char ch;
    fd = open("sample.txt", O_RDWR, 0777);
    if(fd == -1) {
        printf("error in opening a file\n");
        exit(0);
    }
    a = flock(fd, LOCK_EX);
    if(a == -1)
        printf("error in locking\n");
    else
        printf("file locked\n");
    read(0, &ch, 1);
    write(fd, &ch, 1);
    a = flock(fd, LOCK_UN);
    if(a == -1)
        printf("error in unlocking\n");
    else
        printf("file unlocked\n");
    close(fd);
    return 0;
}
```

Note: To check the use of locking, write another program to do some operation on the same file "sample.txt". then run these two programs in different terminals and analyse the output.



Working with directories (System calls)

As you have seen so far, working with files is actually quite easy, and you will be glad to hear that working with directories is little different. Before going to work with directories, 1st we have to know some system calls related to directories. Those are:

- mkdir()
- opendir()
- readdir()
- rmdir()

Required headerfiles:

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

Creating a directory: To create a directory, we have to use "mkdir" system call. The prototype of this system call is given below.

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

path: directory name to be created,

mode: permissions to that directory.

Returns 0 on success and -1 on failure.

Ex: /***** WAP to create a directory in current directory*****/

```
int main()  
{  
    int var;  
    var = mkdir("sample.txt", 0777);
```

```

        if(var = -1) {
            printf("error in creating a directory\n");
            exit(0);
        }
        return 0;
    }
}

```

Note: When you create a directory, two default hidden files (file names starting with '.') namely "." and ".." will be created in that directory. Here "." file is reference to the current directory and ".." file is reference to the previous directory or parent directory of current directory. To check all hidden files in current directory, execute the below command:

<shell>\$ ls -a

Opening a directory: To open a directory, the below system call has to be used.

DIR *opendir(const char *path);

Path: directory name to open

Returns a pointer to directory, which is opened. Here the pointer is of type DIR. So before using this system call, we have to declare a pointer of this type. Then we have to collect the return value of opendir to that pointer variable. Returns NULL on failure.

Ex: /***** WAP to open current working directory *****/

```

int main()
{
    DIR *ptr;
    ptr = opendir( "." );
    if( ptr == NULL) {
        printf("error in opening current directory\n");
        exit(0);
    }
    return 0;
}

```

Reading from directory: To read a directory, we have to use "readdir" system call. Before reading directory, 1st we have to know the reference to that directory which is returned by opendir(). After getting DIR pointer, then pass that as argument to this system call. The prototype is given below:

```
struct dirent *readdir( DIR *dp);
```

dp: directory pointer, returned by readdir().

Returns pointer to the **dirent structure** on success, and NULL on failure. The dirent structure is given below.

```
struct dirent {
    ino_t      d_ino;    /* inode number */
    off_t      d_off;    /* offset to next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type; /* file type */
    char         d_name[256]; /* filename */
}
```

Here d_type specifies the file type. The different possibilities are given below:

<u>d_type</u>	-	<u>File type</u>
DT_BLK	-	Block device
DT_CHR	-	Character device
DT_DIR	-	Directory
DT_FIFO	-	Named pipe
DT_LNK	-	Symlink
DT_REG	-	Regular file
DT SOCK	-	Named socket
DT_UNKNOWN	-	Unknown file type

Ex: /***** WAP to print the 1st file name in current directory*****/

```
int main()
{
    DIR *dptr;
    struct dirent *ptr;
    dptr = opendir( ".");
    if(dptr == NULL) {
        printf("error in opening directory\n");
        exit(0);
    }
    ptr = readdir( dptr );
    If(ptr == NULL) {
        printf("error in reading directory\n");
        exit(0);
    }
    printf("%s\n", ptr->d_name);
    return 0;
}
```



```
}
```

Removing a directory: To remove or delete a directory, we have to use “rmdir” system call. The prototype of this system call is given below.

Int rmdir(const char *path);

path: directory name.

Returns 0 on success and -1 on failure.

Ex: /*** WAP to remove “sample.txt” directory in current directory*****/**

```
int main()
{
    if( (rmdir("sample.txt")) == -1) {
        printf("error while removing a directory\n");
        exit(0);
    }
    printf("directory removed successfully\n");
    return 0;
}
```