5TH CHAPTER SYSTEM CALLS FOR THE FILE SYSTEM

In this chapter we are going to look at the "system calls" for the *File System*.

- System calls which deal with the existing files, are *open()*, *read()*, *write()*, *close()* & *lseek()*.
- System calls which deal with creation of new files, are *creat()*, *mknod()*.
- System calls which deal with manipulation of *inodes*, are *chdir()*, *chroot()*, *chown()*, *chmod()*, *stat()*, *fstat()*.
- System calls which deal with change in the structure of *File System* hierarchy are *link()*, *unlink()*.
- System calls which deal with *File System* tree are *mount()*, *umount()*.
- System calls which deal with implementation of pipe are *pipe()*, *dup()*.

This chapter also introduces 3 data structures in more detail.

1) File Table
2) User File Descriptor Table
3) Mount Table.

File Table: - This has one entry allocated for every opened file in the system. This is global.
User File Descriptor Table: - This has one entry allocated for every file descriptor known to a process. Obviously this is local to each process.
Mount Table: - This has information about currently active *File System*. This is also global.

File System Calls (High Level)						
Calls which return File Descri-ptor	**Calls which use *namei ( )* algorithm**	**Calls which Assign *inodes***	**Calls which deal with File Attribs**	**Calls which deal mainly with File I/O**	**Calls which deal mainly with *File System* Structure**	**Calls which deal with *File System* Free Manipulation**
open() *creat()* *dup()* *pipe()* *close()*	*open()* *stat()* *creat()* *link()* *chdir()* *unlink()* *chroot()* *mknod()* *chown()* *mount()* *chmod()* *umount()*	*creat()* *mknod()* *link()* *unlink()*	*chown()* *chmod()* *stat()*	*read()* *write()* *lseek()*	*mount()* *umount()*	*chdir()* *chown()*
From above table, you can see, some of the algorithms say open() are categorized in more than one group						

Low Level File System Algorithms		
namei()	*ialloc(), ifree()*	*alloc(), free(), bmap()*
iget(), iput()		
Buffer Allocation Algorithms		
getblk(), brelease(), bread(), breada(), bwrite()		

✷ ***Classification Of System Calls Of File System: -*** From above figure, we can say that, *File System* is manipulated by those system calls, which can be broadly categorized into 7 bytes.

These system calls are quite "high level", because they sit on the top of "low level *File System* Algorithms", which in turn sit on the top of "Buffer Allocation Algorithms".

Still more high level abstraction is possible by aggregating these high level *File System* calls into libraries like *studio.h*, *fcntl.h* etc. And then creating high level file I/O functions like *fopen()*, *fclose()*, *fread()*, *fwrite()*, etc.

Programmers are free to use any of these functions of their will. Means they can use high-level functions or low-level system calls directly. The classification is as follows...

1) System calls which return file descriptor. E.g. *open()*.
2) System calls, which do parsing of path, name string. E.g. *creat()*.
3) System calls, which assign free *inodes*. E.g. *creat()*, *link()*.
4) System calls, which change, file attributes. E.g. *chmod()*.
5) System calls which do file I/O. E.g. *read()*, *write()*.
6) System calls, which change *File System* structure. E.g. *mount()*.
7) System calls which *File System* tree view. E.g. *chdir()*.

✷ ***System Call "open()" ->*** Whenever data in a file is "to be accessed", we have to first open the respective file. The general syntax of *open()* system call is....

fd = open (file path, flags, modes) ;

Where...

- file path -> It is string which has path of the file that we want to open. The path may be relative or absolute and must be with the file name.
- flags -> This specifies the "way" by which the file should be opened i.e. for reading or for writing or for both.
- modes -> This specifies the "file permissions" to set, if the file is being created for the first time.

The system call in turn returns an integer. If successful, the returned integer is called as "file descriptor" which can be further used for file I/O like calls, such as read, write, lseek, etc. On failure the returned integer is -1, indicating that this system call is failed to open the specified file.

Algorithm of *open()* system call is like follows :....

```
01 : Algorithm : open()
02 : input :   1) file name string (with absolute/relative path)
03 :           2) type of open (like for reading or writing or both)
04 :           3) file permissions (for creating the file)
05 : {
06 :   convert the file name string to inode (using namei()) ;
07 :   if (file does not exist or no access permission)
08 :           return (error) ; /* i.e. -1 */
09 :   make "File Table" entry for above inode, initialize the count to 1, set the offset of file ;
10 :   make the "User File Descriptor Table" entry, set the pointer to "File Table" entry accordingly ;
11 :   if ("type of open" parameter specifies "truncate file")
12 :           free all file blocks (by using free() algorithm) ;
13 :   unlock inode ; /* which was locked in namei() */
14 :   return (use file descriptor obtained) ;
15 : }
16 : output : file descriptor for specified file
```

- ✓ *Kernel* first converts the "file path string" into respective file's *inode* by using *namei()* algorithm.

- ✓ When it find the *inode* (this is in-core copy of *inode*) it checks the permission field in in-core structure with those of given 3rd parameter (if any).
- ✓ If the file does not exist or if the permissions are invalid for accessing the file, *Kernel* returns error.
- ✓ If everything up till now is "ok", then *Kernel* makes an entry in the *Kernel's File Table* i.e. global. While making this entry *Kernel* does 4 things…

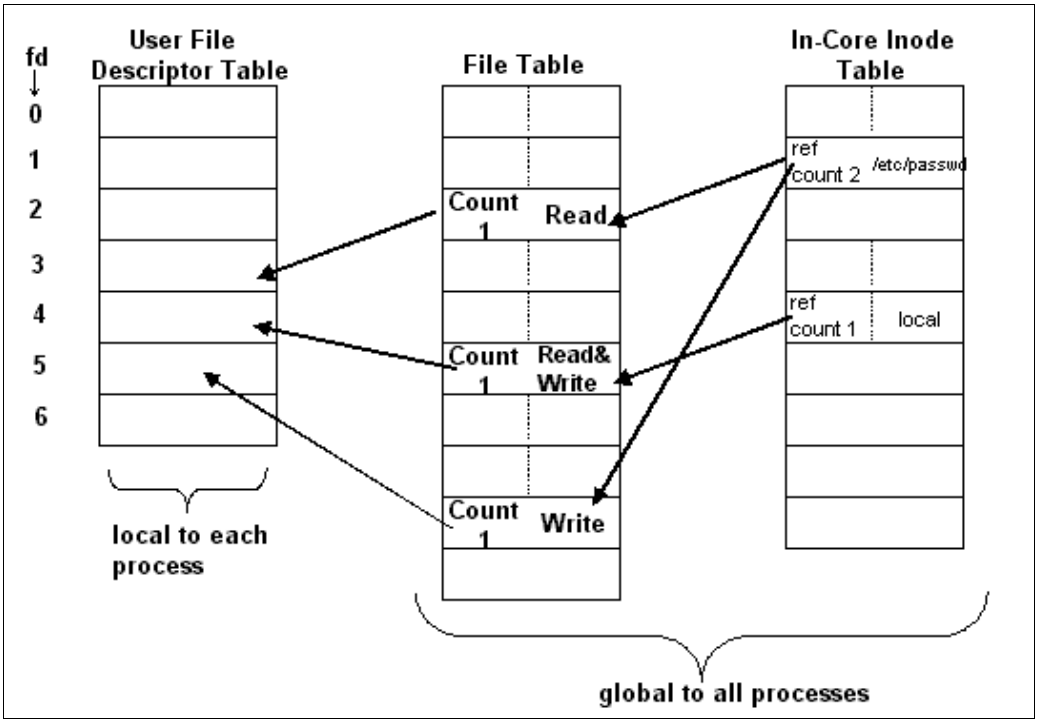
1) First, it initializes count to 1 in *File Table*.
2) Secondly, it enters a pointer to the *inode* of the open file in *File Table*.
3) Third, it sets the byte-offset field in *File Table*. This is an offset from which *Kernel* is going to do next read, write or seek operations on the open file. Here 2 cases may occur…
   a) If the file is opened 1st time for reading or writing, then *Kernel* initializes this offset to 0, indicating that read & write operation will start from the beginning of file (i.e. from 0th position in file).
   b) But if the file is opened in "write-append" mode, then *Kernel* initializes the offset field to "the size of file", means to the highest byte +1 in the file so that the writing operation will begin from "end of file".
4) *Kernel* also keeps entry of "Read" or "Write" or "Read-Write" capability (according to 2nd parameter of *open()*) of the *open()*
   - ✓ *Kernel* executing the *open()* system call on behalf of some process. So that process is going to have its private (i.e. local) "*u area*", which in turn has "*User File Descriptor Table*". Thus *Kernel* also makes corresponding entry in this "*User File Descriptor Table*", where it mainly enter a pointer to *File Table*'s entry of this opened file.
     ✷ Then *Kernel* remembers the position (i.e. index) of this entry in "*User File Descriptor Table*". This is our return value i.e. the file descriptor number.
   - ✓ Some processes may use *open()* system call to "always create" the file. Obviously if the file is already present, "always create" will overwrite the file (i.e. truncate the file). Obviously present blocks of this file are useless as the data on them is going to be overwritten. Hence it frees all present blocks by using *ifree()* algorithm.
   - ✓ Now note that, when *Kernel* uses *namei()* algorithm, it prevents another processes from accessing *inode* table and locks the *inode* of opened file. Thus when *Kernel* leaves the *open()* system call, it should make the *inode* available to the process (which is slept). Thus before leaving the *open()* system call *Kernel* unlocks the *inode*.
   - ✓ Lastly, *Kernel* returns the "remembered index in *User File Descriptor Table*" as file descriptor to the process to which process can use in its future calls of file I/O.

***Example* ->** Suppose a process executes following 3 *open()* system calls sequentially…

```
fd1 = open ("/etc/passwd", O_RDONLY ) ;
fd2 = open ("local", O_RDWR ) ;
fd3 = open ("/etc/passwd", O_WRONLY ) ;
```

In these 3 system calls, process is opening "/etc/passwd" file twice – once fpr read-only and again for write-only. Also process is opening a file named "local" for both read & write operations. Note that path of "passwd" is "absolute" because it starts from "/" while the path of "local" is "relative". Hence *namei()* searching of "passwd" file will begin from "/" and *namei()* searching of "local" file will begin from current directory.

The 3 data structures i.e. *User File Descriptor Table*, *File Table* & *In-core Inode Table*, after above 3 calls will be like following…



- ✓ When first line of code i.e. **fd = open ("/etc/passwd". O_RDONLY) ;** is executed *namei()* algorithm (and all other algorithms used inside the namei) brings "*disk inode*" of "/etc/passwd" in to the *In-core Inode Table* and initialize its "reference count" to 1. Then *Kernel* goes to *File Table* and makes entry of this file by initializing count in "*File Table*" to 1 and also makes entry of pointer to this files "*inode*" in *inode* table. Also it makes the "Read" entry of *open()* call in *File Table*. Also it sets the byte-offset field of this file. After making these 4 tasks in *File Table*, it goes to *User File Descriptor Table* and makes entry for pointer to "*File Table*" entry of this file. This entry in *User File Descriptor Table* is made of index 3, which is returned to the process as "file descriptor" i.e. the fd1.
- ✓ When second line of code i.e. **fd2 = open ("local", RDWR) ;** is executed , all above process occurs exactly same as above. Means an entry is made in *inode* table (by putting "*disk inode*" of "local" file in *inode* table) the reference count in *inode* is initialized to 1. Then the four tasks in *File Table* are done for this file by making unique entry in the *File Table*. After these four tasks, *Kernel* goes to *User File Descriptor Table*, where it finds that 3rd index is already occupied. Hence it makes entry in 4th index and return 4 as fd2.
- ✓ When third line of code i.e. **fd3 = open ("/etc/passwd", O_WRONLY) ;** is executed, then *namei()* does not read "*disk inode*" of "/etc/passwd" file because it is already in *inode* table. Thus *Kernel* just increments "reference count" to 2. But, though file is once opened before, it makes separate entry in *File Table*, and also a separate entry in *User File Descriptor Table* at index 5 and returns 5 as fd3.
  Though a same file is opened multiple times, the file has unique (i.e. only one) entry for file's *inode* in *inode* table.

1) But to indicate "how many times the file is opened" means to indicate, " how many references of the file are currently opened", *Kernel* increments the "reference count" field in the *inode* of this file. In previous example "/etc/passwd/" file is opened "twice" and hence "reference count" field in *inode* of this file becomes 2.

2) If same file is opened multiple times, unlike *inode* table, *File Table* has separate entry. Thus in previous example when we open the file "/etc/passwd/" twice, there are 2 separate entries in *File Table*.

3) The separate entries in *File Table* allows *Kernel* to deal with a file separately according to "Read", "Write" or "Read-Write" capabilities specified by the process when calling *open()* system call.
4) Interesting thing is the "count" field in "field table". If it is initialized to always 1, even if same file is called multiple times, then why it is necessary? Separate entry is itself enough, then why to mention it as 1 explicitly? The answer is that, *creat()* & *open()* system calls make "count" field 1 but *dup()* or *fork()* system calls may increment this count when same process call them for same file.
5) Similar to *File Table*, the *User File Descriptor Table* always have "separate entries" even if the same file is opened twice. This is most essential. Because process doesn't know *Kernel*'s internals. It identifies the file by "file descriptor" (i.e. fd1, fd2 or fd3 in past example) returned by *creat()* or *open()* system calls. So the same file opened twice, once for reading and once for writing, must be deal separately.
6) If we observe the *User File Descriptor Table* carefully we see that entries of "*File Table*" start from index 3, bypassing 0, 1 & 2. The reason behind this is important. When a processing is running, it has access to standard input, standard output & the standard error streams. These streams are represented by "terminal". As "everything in *UNIX* is file", terminal is also a file. Thus to give access to this file first three indices, i.e. 0, 1 & 2 are given for file descriptors of standard input, standard output & standard error streams respectively. *UNIX* system conventionally uses standard input streams descriptor to read input data, standard output stream descriptor to write output data & standard error stream descriptor to write error data messages on terminal. Note that, these 3 file descriptor are not special but are conventional only. You can use any other indices as 4, 6 or 11. but as array in C begins from 6^{th} index, the conventional 0, 1, 2 file descriptors are more natural.

Now suppose a new process (while the previous one is already running) starts executing following code lines…

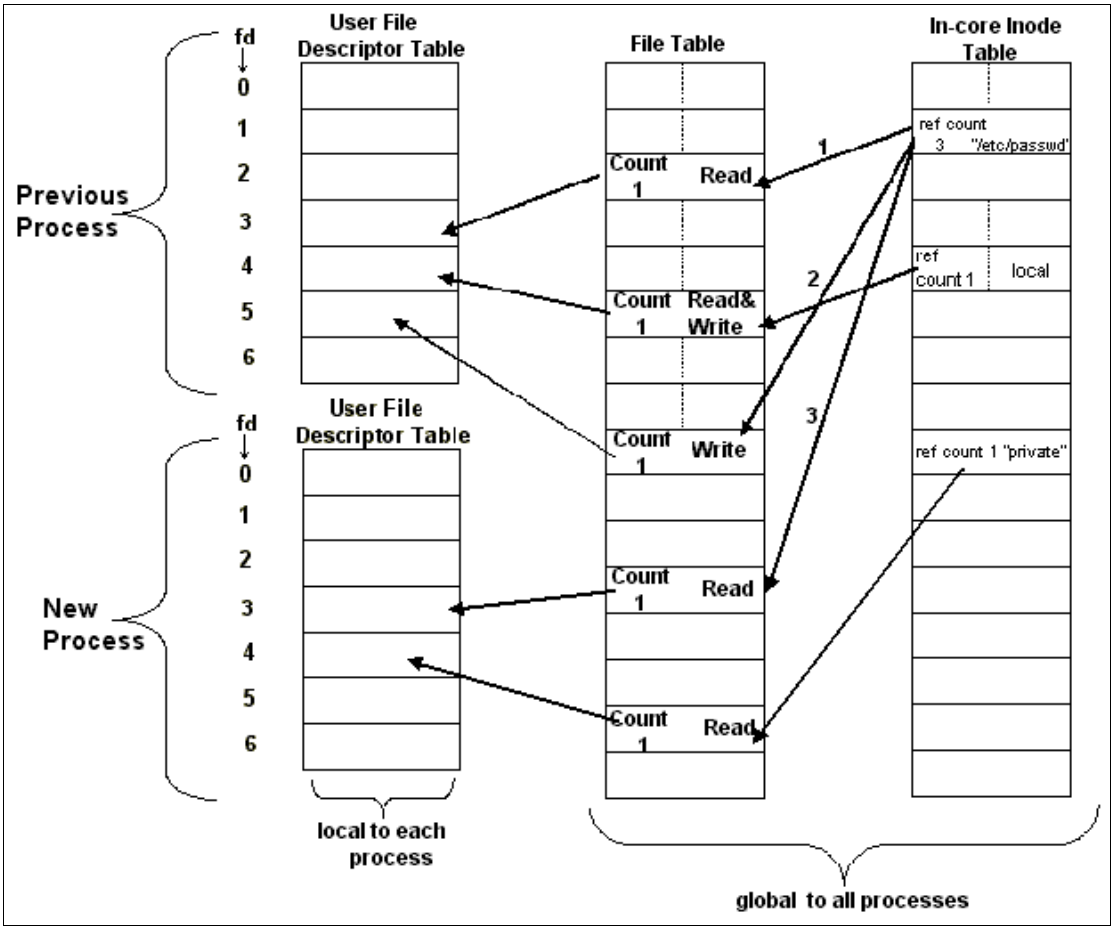
```
fd1 = open ("/etc/passwd", O_RDONLY) ;
fd2 = open ("private", O_RDONLY) ;
```

The new process will have its own *User File Descriptor Table*, which will return file descriptors for above 2 files to this new process.

But *Kernel File Table* & *inode* table are global data structures and hence common to all processes. Thus though "/etc/passwd" is opened again (i.e. third time) it will have its unique entry in *File Table* with again count set to 1. But as there is only one *inode* table already has entry of *inode* of "/etc/passwd", no new entry in *inode* table is made for this file, but only "reference count" will be now incremented to 3.

But as "private" file is newly opened, its *inode* table entry is made by reading its "disk inode".

Diagram-:



After examining both figures, we found that there is always "one to one" relation between *User File Descriptor Table* & *File Table* And there is "many to one" relationship between *File Table* & *inode* table.

So a question may arise, why contents of "file table" (i.e. count, *inode* pointer, byte offset, RW capabilities) are not made member of *User File Descriptor Table* and thus eliminating the need of separate *File Table*?

The answer is given by Ken Thomson. He said that as *User File Descriptor Table* is made local to each process, process has freedom to operate on files irrespective of other processes. But this scenario imposes one problem. *UNIX* is multitasking & multi-user OS. So file sharing must be one of its features. Including everything just *User File Descriptor Table* makes files sharing impossible. Because one process cannot have another process's *User File Descriptor Table*'s access. So there should be such a global data structure, which will allow sharing files to all process. That is why "file table" is created. Now processes can share files using *File Table* through *dup()* & *fork()* system calls.

✴ **System Call *read()* ->** The syntax of read system call is as follows....

number = read (fd, buffer, count) ;

Where...

fd -> file descriptor returned by open() system call.

buffer -> The memory in which "read data" is to be kept. (Obviously it is empty initially).

count -> How much data is to be kept, in buffer. This is in terms of bytes.

number -> This is the return value of read() system call. This specifies the "actual read bytes" which may be equal to "count" or not.

The algorithm of reading regular file by using ***read()*** is as follows .....

```
01 : Algorithm : read()
02 : input :   1) user file descriptor
03 :           2) address of a buffer (local to process)
04 :           3) number of bytes to read
05 : {
06 :   get "File Table entry" by using file descriptor ;
07 :   check file's accessibility ;
08 :   set parameters in process's u area for (a) user address (b) byte count (c) I/O to user etc. ;
09 :   get "inode table entry" from "got file table" ;
10 :   lock the inode ;
11 :   set byte offset in u area from the File Table offset ;
12 :   while (count is not satisfied)
13 :   {
14 :           convert file offset to disk block (using bmap()) ;
15 :           calculate offset into disk block  & also calculate number of bytes to read ;
16 :            if (number of bytes to read is 0) /* try to read EOF*/
17 :                    break ;
18 :          read disk block (by using bread() or if required breada()) ;
19 :          copy data from system buffer to user address ; /* i.e. to local buffer 2nd parameter */
20 :          update "u area" fields accordingly ;
21 :          release buffer by brelease() ; /* which was locked in bread() */
22 :      }
23:       unlock the inode;
24:       update "file table" offset entry accordingly ;
25:       return (number of actual read bytes) ;
26: }
27: output : number of bytes actually read (means number of bytes copied to process)
```

- ✓ At the beginning, *Kernel* gets "file descriptor", which is an index into "*User File Descriptor Table*". Thus using "file descriptor", *Kernel* goes to "*User File Descriptor Table*" and gets "*File Table*" entry.
- ✓ Then using "*File Table*" entry, it goes to "*File Table*" and checks capabilities such as "read only", "write only" or "both".
- ✓ Then it goes to process's *u area* and sets following I/O parameters...

a) I/O mode :-> Whether "read", "write" or "both" (taken from *File Table*)
b) Count -> Number of bytes to read/write.
c) Address -> Address of process's local buffer or *Kernel* buffer (where the data is to be copied).
d) Offset -> byte offset in file from which I/O is to be started. (This is taken from "*File Table*").
e) Flag -> Indicating whether the I/O will go to process's local memory or *Kernel*'s memory (i.e. buffer).

By setting these fields in process's *u area*, *Kernel* avoids special function creation and also avoids above fields as parameters to such a special function.

- ✓ Then from "*File Table*" entry, it finds "inode table" entry and goes to "*inode* table". After entering into the *inode* table, it gets respective file's *inode*.
- ✓ Then it locks the *inode*.
- ✓ The field (d) of process's *u area* is set at this step, by using "*File Table*" entry, where there is file's offset from which file I/O is to be started.
- ✓ Now *Kernel* enters into "while loop", in which it uses "count" field of process's *u area* as "testing condition". The loop continues until the "count" is not satisfied.
- ✓ Inside the loop, *Kernel* first converts file's byte offset into a disk block number by using bmap algorithm.

- ✓ Using remaining 3 outputs of *bmap()* algorithm (out of 4, one is returned in above step), it notes the byte offset in that disk block and by subtracting "beginning of block" from above byte offset, it calculates "how many bytes" it can read actually.
- ✓ If "number of bytes of read", calculated in above step is 0, it breaks out of the "while loop".
- ✓ Now if the "number of bytes to read" are greater than 0, then it reads the block using *bread()* algorithm. But if it finds that "next block" reading is expected, then it uses *breada()* algorithm to read the block.
- ✓ I n *bread()* / *breada()*, data is read into *Kernel*'s buffer. Process usually has its own "local buffer", where the "read data" is to be kept. Hence *Kernel* copies data from its buffer to the process's local buffer.
- ✓ As files byte offset is now moved up to the "last read byte", and as *Kernel* now knows "how many bytes are actually read", it updates according fields in process's *u area*. i.e. It updates file's byte offset, read count & address to write into process's buffer. (Here note that, the "read count" is changed, because process if wants to read 1044 bytes and it 600 are "actually read", then "read count" now becomes 424). Similarly suppose process, locally has a buffer say **pBuffer[ ]** of 1024 size, and suppose its beginning address is say 2022, as now 600 bytes are actually read, the address moves to 2622, from which the next copying of data should be started.
- ✓ Then it releases the *Kernel*'s buffer (i.e. used in *bread()* / *breada()* by *getblk()*).
- ✓ Above loop keeps on executing, until the process's required "number of bytes" are read.
- ✓ After completion of process's reading process, i.e. by satisfying the process's needed read procedure, *Kernel* comes out of the loop. Also note that, it also comes out of the loop by breaking the loop if the "read byte" are 0.
- ✓ After coming out of the loop, *Kernel* unlocks the *inode*.
- ✓ Then it updates the "byte offset" entry in the "*File Table*", from which next I/O is to be started.
- ✓ Finally *Kernel* returns total number of "read" bytes.

Note -> The completion of "while loop" in read system call takes places in either of the following 4 conditions...

1) If process's request is completely satisfied (means *read()* algorithm reads the equal number of bytes as specified by the process in *read()*'s parameter.
2) When the file (whose reading is going on) contains "no more data" to read.
3) When *Kernel* encounters an error while reading the disk block either by *bread()* or by *breada()*.
4) When *Kernel* encounters an error while copying "read data" from *Kernel*'s buffer to process's local buffer.

✶ Now we will see one program written in C, which opens a file and uses returned file descriptor in 3 subsequent calls to read with different local buffers and with different "number of bytes to read".

```
# include <stdio.h>
# include <fcntl.h>
void main (void)
{
        /* variable declarations */
        int fd ; /* file descriptor */
        char LittleBuffer [20], BifBuffer [1024] ;
        /* code */
        fd = open ("/temp1.txt", O_RDONLY) ;
        read (fd, LittleBuffer, 20) ;
        read (fd, BigBuffer, 1024) ;
          read (fd, LittleBuffer, 20) ;
}
```

Here *open()* system call takes file name and according to O_RDONLY mode, returns a file descriptor in "fd" variable. Successful return value (i.e. not -1) in "fd" variable indicates that, "*inode* table" entry, "*File Table*" entry & "*User File Descriptor Table*" entry are made successfully for above process.

First call to read -> *Kernel* verifies that the "file descriptor" is legal and the file is opened for reading or not.

Then in "*u area*" of the process, it stores...
-> Address of process's local buffer to LittleBuffer.
-> The byte count to 20 and
-> The starting byte offset of file to 0 (as default).

Then it calculates that the file byte offset 0 lies in the 0th block in file and retrieves the entry of 0th block in the *inode* of that file.

Now *Kernel* reads the entire block of 1024 bytes (Here we assume that system's default block size is 1024 bytes) into its own buffer (i.e. system buffer or say *Kernel* buffer) and copies only 20 bytes of them to the process's local buffer, LittleBuffer.

Then as 20 bytes are read from file, *Kernel* resets "byte offset" in "*u area*" of process to 20 (which was by default initially 0). And also decrements the "count" of "*u area*" to 0 (because process's request was for 20 bytes and those all are read. Thus request is fulfilled). Also it sets offset of LittleBuffer to 20th position.

As "read" of 20 bytes has been satisfied, now *Kernel* sets "*File Table*'s" byte offset field to 20, so that next time I/O should begin from 20th byte offset.

Finally "*read()* system call" returns 20 as the "number of actually read bytes".

Second call to read() -> The process is same as above. Means for this call, again, *Kernel* stores BigBuffer in "address" field, 1024 in "byte count" field and 20 in "file byte offset" field, of process's "*u area*" (Here 20 is taken from "*File Table*" which was set during previous call to *read()*). If the time between 2 *read()* calls of same process is "small", there is possibility that buffer will directly found on *Free List* and *Kernel*'s reading from disk can be saved. But this time request is for 1024 bytes, block is of 1024 size and offset is at 20, hence out of 1024 requested size, *Kernel* can read only 1024-20 i.e. 1004 bytes from this block and copies them to BigBuffer address. Corresponding updates are made in "*u area*" and also in "*File Table*", so that next I/O of this file will begin at 1004th position in BigBuffer and from 1024th offset position in file.

But this time, request was for 1024 bytes, out of 1004 are read and 20 are remaining. Means process's request is not satisfied and hence *Kernel* loop back into the "while loop" of *read()* algorithm.

When it loops back, now "*File Table*" has offset set to 1024. It converts this to file block, which was now not 0th but the 1st lock (as 1024 of first block are already read – i.e. 20 from first call read and 1004 from next call to read). So in *inode* it skips 0th block and looks in 1st block (sequentially 2nd first block). It looks for 1st block's buffer in "*Buffer Cache*" and if not found, then reads from disk into a buffer. Finally it copies remaining 20 bytes (which were unsatisfied for previous iteration) from system buffer to process's BigBuffer address from 1004th position onwards.

Now process' request is satisfied, hence *Kernel* can leave the call, but before leaving, now offset in BigBuffer is set to 1024 (1004 from first iteration & 20 from next iteration) and size of BigBuffer is already 1024 i.e. exhausted. In "*File Table*", it sets the file byte offset to 1044 (20+1004+20).

Third call to read() -> Process is same as for above two calls. Now *Kernel* starts from 1044th file byte offset read 20 bytes (requested in 3rd call) and copies then to LittleBuffer (overwriting previous data) from 0th position, because 1st call exhausted it. While leaving the system it sets *File Table*'s offset field to 1064 (20 [1st] + 1004 [2nd iteration 1] + 20 [2nd iteration 2] + 20 [3rd]).

Note: -

- ***i.*** For all above 3 calls "file byte offset" field of "*File Table*" was incremented by the number of bytes actually read, because all the 3 calls use same file descriptor having its unique connection with unique entry in *File Table*.
- ***ii.*** Standard I/O libraries of compiler are written in such a way that *Kernel*'s buffer size (i.e. block size) is encapsulated from processes, so that processes can read data irrespective of this

knowledge. These libraries have their own buffering mechanism, which try to reduce performance penalties.

✷ How *Kernel* Determine Chance Of read-ahead?

If

a process reads 2 blocks sequentially, then *Kernel* assumes all next read operations will be sequential.

Thus during each iteration of loop, *Kernel* saves "next logical block number" in the in-core *inode* and during actual next iteration, it compares the current logical block number with the previously saved value. If they are equal, *Kernel* calculates the physical block number and saves this physical block number value in process's "*u area*" for use in *breada()* algorithm. But if process does not read up to "end of block", *Kernel* does not call *breada()* algorithm for the next block.

✷ If we look at the figure of inode in 4th Chapter, some direct blocks contain 0, what happens while reading such blocks? Some block numbers in *inodes* "table of disk addresses" may contain 0, though next blocks of same file may contain nonzero values. This simply means that for that portion of file there is no data, though later portions (as later blocks may be nonzero) may contain data.

So while reading such blocks, *Kernel* satisfies the request, allocates an "arbitrary buffer" (because as there is no actual block for "0" entry, no actual buffer can be allocated), clears buffer contents to 0 and copies this buffer to process's local buffer. Obviously process gets "0 data" situation for such block.

But note that, this is not the same treatment for "EOF". When *Kernel* encounter EOF of a file, means there are "no next valid blocks", it does not return "0 data" but returns "no data" to the process and process can thus judge that this is EOF.

✷ Inode locking mechanism during read -> When *Kernel* executes "*read()*" system call, it locks the respective file's *inode* for the "whole duration of call". Then if process "sleeps" due to some reason (i.e. during buffer algorithms) and if some other process opens the same file, then *Kernel* looks for "flag to open". If the "flag to open system call" is O_RDONLY, it just increments the "reference count" field of in-core copy of this file's *inode* and copies data to the process's local buffer, which can then see the data "only for reading".

But if the "flag to open call" is O_WRONLY or O_RDWR, then for this 2nd process, *Kernel* opens the file, increments the "reference count" and copies the data to process's local buffer. Now process can write to or modify the contents of file seen in local buffer. But if process tries to save the changes, then from local buffer, changes are passed to system buffer of file, but before going to disk, *Kernel* realizes that "slept process" already locked the *inode* and thus 2nd process does not allow to save the changes, until "slept process" wakes up and unlocks the *inode* by completing its read operation.

Note that, *inode* locking and unlocking mechanism is "depending on system call" and not on "closing the file". Because some process may open the file and does not close it for long time, then all other processes may wait until the "closing of file" by first process. This could be disastrous especially for a file like "/etc/passwd", which is used for "user login". If one user starts manipulating "/etc/passwd" for login, then other users may have to wait until first user's login completes. This must not happen. Thus "*inode* locking" is done during system call and "*inode* unlocking" is done as soon as system call finishes. It is not dependent on "closing of file".

Due to this strategy, some times it may happen that one process opens the file by *open()* system call. As system call is finished, another process may write something to this file and thus first process after its read call may see some unexpected data. Thus system does not guarantee that data in a file will remain same after opening a file. If you want to enforce this strategy, it is better to use "file & record locking" feature given by the system.

✷ ***What will happen for following code?***

```
# include <fcntl.h>
void main (void)
{
        // variable declarations
        int fd1, fd2 ;
        char Buffer1 [512], Buffer2 [512] ;
        // code
        fd1 = open ("/etc/passwd", O_RDONLY) ;
        fd2 = opne ("/etc/passwd", O_RDONLY) ;
        read (fd1, Buffer1, sizeof (Buffer1)) ;
        read (fd2, Buffer2, sizeof (Buffer2)) ;
}
```

As "file descriptor" are different, though the file is "same", they will have unique entries in "*File Table*", hence by default both "*File Table*" entries will have offset 0 and hence will read data from 0th to 511th position in file and thus contents of Buffer1 & Buffer2 will be exactly the same. So this is the example of opening same file and reading it by two different file descriptors.

✷ **System Call *write()* ->** The syntax of *write()* system call & meanings of parameters & return value are same. **number = write (fd, buffer, count) ;**

The algorithm of writing a file is also quite similar to the algorithm of reading a file.

- ✓ But if the file does not contain a block, that corresponds to the "byte offset", on which writing is to be done, then *Kernel* allocates a new block using algorithm *alloc()*.
- ✓ If such a new block is required, then obviously file is "growing" and thus this new block of this file requires entry in the *inode*'s "Table of disk addresses" i.e. in the "13 member array".
- ✓ It also can happen that, the new block is beyond the 10 direct blocks and thus may require indirect block. Example : Suppose process wants to write at byte number 10,240. suppose it is the highest byte yet written to the file (obviously beyond current EOF). At the beginning *Kernel* tries to find logical block for 10,240 by *bmap()*, but as it is not there, *Kernel* determines 2 things ... 1) There is no such a block for this file 2) And also there is no "indirect block" (because 10240 is beyond the scope of 10 direct blocks) too.
- ✓ So *Kernel* first assigns one disk block for "indirect block" and writes its number at 10th index in "13 member array" of in-core *inode* of this file.
- ✓ Then it assigns another disk block for "actual writing" – so called as "data block" and writes its number at the 0th position in "indirect block" chosen in previous step.
- ✓ Similar to "read", "write" also has a while loop, in which, during every iteration, it writes one block to disk, until process's writing satisfies.
- ✓ During writing, *Kernel* must take care of 2 things:
    1) Whether it is writing "entire block" to disk or
    2) Whether it is writing "only part of a block" to disk.

1) If it is writing "entire block" to disk, then it either writes "newly" on an empty portion of disk or overwrites "old" portion of disk.
2) But if it writes a "part of block" on disk, then process does not want "existing data" at that position on disk to get overwritten. Hence first *Kernel* read that portion of disk into the block and then writes.
    - ✓ The writing process, as explained above, is usually block by block. But occasionally *Kernel* may use the "delayed write" method, if it anticipates that another process may require the same block for read or write. So by using "delayed write", it avoids extra "disk operation". Delayed

write is the method of choice for…a) Pipes and b) Temporary type regular file which are going to be deleted after process exits.

- ✓ Just like *read()* system call, *write()* system call also keep the *inode* of file locked during its entire duration. Because *Kernel* may change the *inode* structure according to dynamic growth of file.
- ✓ When writing completes, *Kernel* updates "file size" field in the *inode*.

✷ **File Locking & Record Locking ->** The original *UNIX* system developed by ***T & R*** did not have this feature. Because both of them did not anticipate, "exclusive access to a file". But as database technology grow, ***System V*** adds this feature to *UNIX*.

File locking is the capability given to one process to prevent both reading & writing of the same file partially or entirely by another processes.

Record locking, on other hand, is the capability given to one process to prevent reading & writing on particular record in a file by other processes.

[Note that, record is nothing but part of a file from byte offset X to byte offset Y].

For this type of File & Record locking, *System V* uses *fcntl()* system call.

fcntl (fd, cmd, arg) ;

where fd is "file descriptor" of file to be locked. cmd is the type of locking operation and arg is for arguments. arg may inclues :-1) type of lock, i.e. lock to read OR lock to write 2) byte offset – when record locking is required and locking operations i.e. cmd includes – 1) testing of locks belonging to other process but return immediately (means don't sleep) if unsuccessful.

When a locked file or a file whose record is locked, gets closed, *Kernel* automatically releases all "set locks".

Note that, if process exits before closing a file, then *Kernel* closes the open file and then releases the locks.

✷ **System Call *lseek()* ->** This system call is used to adjust the position of file I/O. Up till now our reading & writing was sequential means started from 0^{th} byte offset of file at beginning and then proceeding as per the prior byte offset setting for next I/O. But *lseek()* system call allows us to start I/O from any byte offset in the file and thus allow random access to the file. The syntax is …

position = lseek (fd, offset, reference) ;

Where…

fd is "file descriptor".

offset is the byte count in the file for which you want to do I/O (in other words up to which byte you want).

reference is the point which indicate "from which point offset should be considered". This has 3 values :

0 -> a) offset should be considered from beginning of file.

2 -> b) offset should be considered from end of file.

c) offset should be considered from current position of read or write offset point in file.

The return value position is the byte offset from which next I/O will start.

Note that, *lseek()* system call has nothing to do with the movement of disk arm nor a particular disk sector. *Kernel* simply sets the byte offset field in "*File Table*". Which is further used as starting byte for next I/O.

✷ **System Call *close()* ->** This system call is used to close a file when process no longer wants to access it. The syntax is ….

close (fd) ; where the fd is "file descriptor" of a file which is being closed.

Algorithmically, closing a file requires updating of *User File Descriptor Table*, *File Table* & *In-core Inode Table*.

-> First changes are made in *File Table*, there are 2 cases….

1) Usually "count" field in "*File Table*" is 1, which is decremented to 0, and entries are made empty.

2) But if *dup()* or *fork()* system calls made this "count" more than 1, then the "count" field is decremented by 1. Entries are retained.

-> Then changes are made in the *In-core Inode Table*. There are also 2 cases…

1) If only one file is right now referencing the *inode*, then "reference count" in *inode* entry is 1, which after close becomes 0. As "reference count" becomes 0, now *inode* of this file appears on "*Free List* of *inodes*" because *Kernel* releases it by *iput()*. Also entry in "*inode* table" for this *inode* is freed but not made empty.
2) But if other processes are also referencing the same file (means same *inode*), then "reference count" field in *inode* has value more than 1. Thus *Kernel* just decrements the reference count and as other processes still needs this *inode*, its entry in *inode* table is retained.

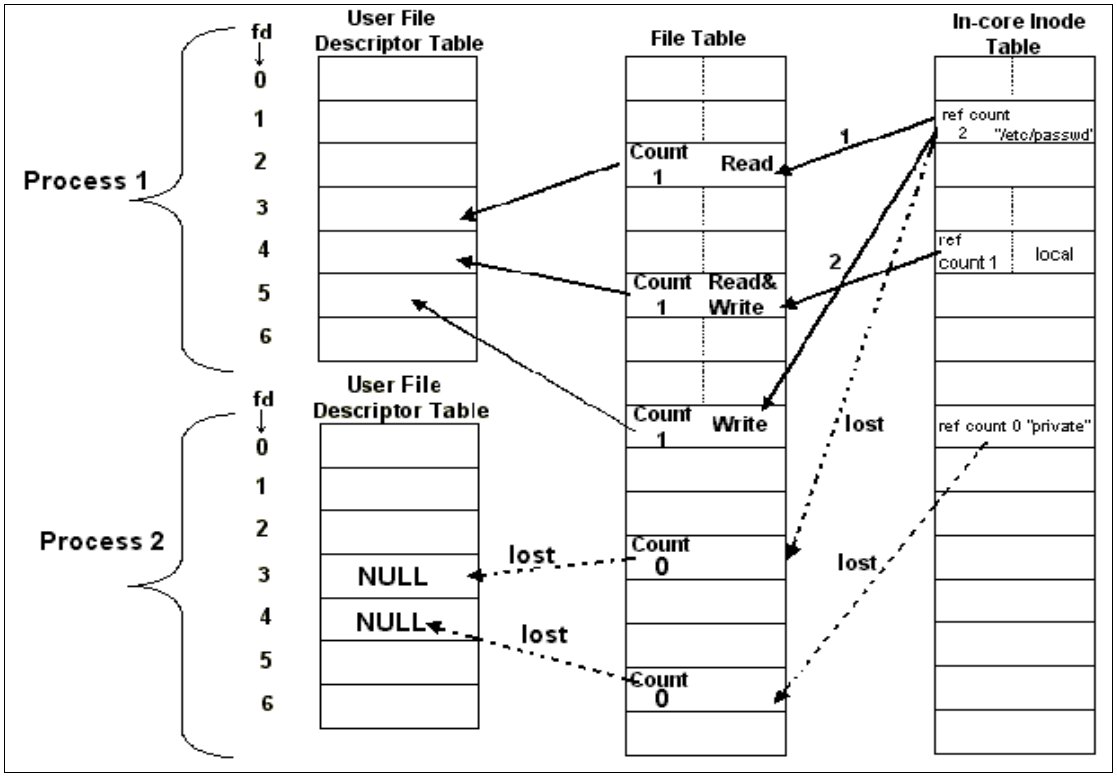
-> Final changes are made in *User File Descriptor Table* when "*close()*" system calls completes, the entry in *User File Descriptor Table* is made empty i.e. NULL. So if same file descriptor is used further by the process after call to *close()*, *Kernel* flags error.

Sometimes a process may exit without closing a file. In such cases *Kernel* explicitly looks for all open files of exited process, closes them and hence ensures that "no process can keep a file open, after it terminates".

As an example, see the program. If process 2 after opening "/etc/passwd" and "/private" files, issue following commands….

close(fd1) ;
close(fd2) ;

Then entries for file descriptor 3 & 4 are made empty (i.e. NULL. The "count" field in "*File Table*" entry is now 0 and these entries are also empty. The "reference count" field in *inode* of "/etc/passwd" is decremented by 1 and thus becomes 2 because process 1 has 2 references to this *inode*. But as "/private" file has no other references for its *inode*, the "reference count" field in its *inode* become 0, but its entry in *inode* table is not made empty, because still it is on *Free List* and some process may access the file and thus *Kernel* can reclaim this entry.



✶ **System Call *Creat()* ->** Usually *open()* system call is used to open an existing file (though if used with O_CREAT or O_TRUNC flags, it can create a new file if not exist and can truncate a same file if exists).

But as convention *creat()* system call is used to create a new file. The syntax is…

fd = creat (pathname, modes) ;

Where fd is as usual the "file descriptor" returned by this system call after creating the file. pathname is null terminated string of file name with absolute or relative path.

modes are same as in *open()* system call. If no file with the same name previously existed, *Kernel* creates a new file with specified name & permission setting given by "modes" parameter. If the file is already existing, then *Kernel* truncates the file (this is contrast to *open()*. In *open()*, O_TRUNC flag must be set to truncate an existing file). While truncating the file, *creat()* system call releases all existing data blocks of that file and sets file size to 0 in in-core *inode* of this file. (**Note : *creat()* system call is used to create only regular files and not for special files**). The algorithm of *creat()* system call is as follows .....

```
01 : algorithm : creat
02 : input :   1) file name
03 :           2) permission settings
04 : {
05 :   get inode file name (by namei) ; /* for existing */
06 :   if (file already exists) /* inode is got */
07 :   {
08 :           if (access not permitted)
09 :           {
10 :                   release inode by iput() ;
11 :                   return (error) ; /* usually -1 */
12 :           }
13 :   }
14 :   else /* if file yet not existed */
15 :   {
16 :           assign a free inode from ialloc() ;
17 :           create new directory entry in parent directory (i.e. enter new file name and attach
 new
18 :           inode to it in directory file) ;
19 :   }
20 :   allocate File Table entry for inode and initialize "count" field in File Table ;
21 :   if (file exists at this time)
22 :           free all file data blocks by free() ;
23 :   unlock the inode ;
24 :   return (user file descriptor) ;
25 : }
26 : output : file descriptor.
```

On most steps *creat()* system call follows *open()* system call.

- ✓ First *Kernel* parses the file path string and gets the *inode* for the file anticipating that file may already exists.
- ✓ If *inode* is received from *namei()*, then obviously file exists. Now if file already exists, it checks "file permissions" and if access to this file for this process is not permitted, it releases the *inode* and returns error.
- ✓ But if *namei()* returns "no *inode*", then file does not exist and thus *Kernel* enters in "else" block. ✷ For this case, while parsing the path by *namei()*, it reaches up to last component and stops before the actual file name. As this component is a directory, it enter into that *directory file* and note the byte offset of first empty directory slot and saves this offset in process's "*u area*". This is the place where *Kernel* will add the newly created file's name & its *inode*

number's entry. Sometimes it may happen that while reading the *directory file*, it may not found any empty slot. In such case *Kernel* explicitly creates an empty slot after the last existed "filled slot". It also saves the *inode* of this directory in process's "*u area*" and keeps this *inode* locked. Thus obviously this directory is going to be parent directory of newly created file.
Note that, *Kernel* still does not enter "file name + *inode* number" entry in empty slot, because if some error occurs in next processing, it will be able to "undo" changes made up till now.
As usual, *Kernel* checks, whether the process has write permission for this *directory file* (Recall that write permission for directory means allow to create subdirectories & files).

- ✓ Now we will come back to "else" block….
  So for now *Kernel* know that file is not existing hence it allocates new *inode* for this new file by using *ialloc()*.
- ✓ Now it writes "file name + newly allocated *inode's* number" entry in the previously seen "empty slot" in the *directory file*. As mentioned before the byte offset of this "empty slot" is taken from "*u area*" of the process, where it was previously saved.
  ✷ The *inode* of the parent directory, which was previously kept "locked" during *namei()* in the "*u area*", is now not needed to be kept locked. Hence *Kernel* releases it. (Note that, *Kernel* releases the *inode* of parent directory, not the *inode* of newly create file).
- ✓ Now 2 things are updated in memory :- 1) the file's *inode* and 2) changed *directory file* (due to addition of new file's entry). These 2 changes must be written to disk.
  - Thus *Kernel* writes file's *inode* to disk by using *bwrite()*.
  - And the only, it writes the *directory file* to disk by using *bwrite()*.
    The above sequence or order is must. Because if *inode* is written before *directory file* (as above) and if system crashed before writing the *directory file*, there will be an allocated *inode* which will have no file, but system works normal. But if *directory file* is written first and if system crashed before writing *inode*, then there will be a "file" with "no *inode*" and thus "file consistency" is lost.
- ✓ After coming out of the "else" block, *Kernel* makes entry in "*File Table*" for the *inode* and initializes "count" field in "*File Table* entry". Note that, this line of code is common to 2 situations…a) if file exists and if permissions to truncate it are valid b) if a new file is created.
- ✓ If file exists & truncation is allowed (i.e. access

permissions are valid) then to allow its truncation, the file's existing blocks must be freed. This is done by *free()* algorithm. Thus now file is new.

- ✓ Now here *Kernel* unlocks the file's *inode*.
- ✓ From here onwards algorithm follows "open" system call. Means it makes entry of "*File Table*" in "*User File Descriptor Table*" of process and return its *inode* to the process as valid "file descriptor".

✷ A note about truncation of an existing file ->

- ✓ The "old file" which is going to be truncated as "new file", must allow permission of "write" for this process.
- ✓ As truncation is going to be done only for file's data, the name is going to be same and thus file attributes like ownership, group, permissions are also going to be same.
- ✓ Thus if process specifies some new "permissions", *Kernel* ignores them, retaining old permissions as they are.
- ✓ As the file is already in same directory, *Kernel* also does not care about directory's "write" permission. Because this entry is also going to be retained.

✷ System Call *mknod()* ("mk" stands for make, "nod" stands for "node".) -> *mknod()* is the system call used to create special files (unlike *creat()*) like pipes, device files and directory files. The syntax is…..

mknod (pathname, type & permissions, device) ;

Where…

pathname-> same as explained before
type & permissions -> Where "type" is either directory or pipe, etc. "permissions" include access permissions for newly created special file.
device -> This specifies the "Major Number" and the "Minor Number", if the file is device file (e.g. – Block special or character special).

The algorithm for *mknod()* system is as follows:.....

```
01 : algorithm : mknod
02 : input :   1) path name (i.e. node name)
03 :           2) type & permissions
04 :           3) major & minor numbers for device file if any
05 : {
06 :   if (new node is not "named pipe" and if the user is not "super user")
07 :           return (error) ;
08 :   get inode of parent directory by namei() ;
09 :   if (new node already exists)
10 :   {
11 :           release parent directory's inode by iput() ;
12 :           return (error) ;
13 :   }
14 :   assign free inode for new node by ialloc() ;
15 :   create new directory entry in parent directory ;
16 :   release parent directory's inode by iput() ;
17 :   if (new node is Block special or character special device)
18 :           write major & minor number into the inode ;
19 :   release new node's inode by iput() ;
20 : }
21 : output : nothing.
```

The algorithm is very much similar to the algorithm & explanation of *creat()*. Getting, saving & releasing parent directory's *inode* are explained in *creat()* algorithm but not actually written in the algorithm. Here these steps are actually written in the algorithm.

Different steps are...

- ✓ If the new node is not named pipe and if the user is not super user, then return error.
- ✓ As this is a special file, after getting inode for it by *ialloc()*, algorithm must set the "file tupe's field in the *inode* to the type (2nd parameter) of this special file. E.g. if *directory file* is to be created, the "file type" field in its *inode* should be set to "directory".
- ✓ Finally, if the new node file is for block special or character special device, the algorithm must set Major number & minor number fields in its *inode* to given values (i.e. the 3rd parameter).
- ✓ ✷ If the "special file" is "*directory file*", the file is created but as yet it does not have entry of "**.**" & "**..**", its format is wrong. To make its format right, steps of *mkdir()* system call must be followed. These steps includes...
    1. Take two empty slots in new *directory file*.
    2. Enter "**.**" & newly created *inode*'s number as first entry (because "**.**" represents "this" directory) in first empty slot.
    3. Similarly enter "**..**" & parent directory's *inode* number (before releasing it in *mknod()* algorithm) as second entry (because "**..**" represents "parent directory" of "this directory") in second empty slot.
       Now the format of new *directory file* is valid.

✷ **System Call *chdir()* ->** This system call is used to change process's current directory (i.e. to change the process's execution environment).

When system is booted, the first process i.e. process 0 makes root (i.e. "/") as its current directory. Process 0 then calls *iget()* on root's *inode* and when *inode* number of root is got, process 0 saves it in its "*u area*" & releases the *inode* lock. Now onwards every new process (including shell) is created by *fork()* system call and the new process inherits the "current directory" of its parent process and saves it in its own "*u area*". As every process calls *iget()* on "current directory" of parent, the "reference count" field of parent directory's *inode* is incremented accordingly.

Now we will see the algorithm of *chdir()* from above background...

The syntax is ... **chdir (pathname);** where "pathname" is as usual. This pathname will now become the new current directory for the process, which calls this system call.

```
01 : algorithm : chdir()
02 : input : pathname string
03 : {
04 :   get inode for new directory by using namei() ;
05 :   if (the inode is not of directory or if process has no permission to access this directory)
06 :   {
07 :           release inode by iput() ;
08 :           return (error) ;
09 :   }
10 :   unlock the new directory's inode ;
11 :   release "old" current directory's inode by iput() ;
12 :   place new directory's inode into "current directory" slot of process's "u area" ;
13 : }
14 : output : nothing
```

- ✓ The *Kernel* uses *namei()* algorithm to parse the filename and gets *inode* for the "new" directory.
- ✓ Then it checks "whether the received *inode* is really of a directory or not" and it also checks "whether the process has permission to access this *directory file* or not". If any of this condition is there, *Kernel* releases the *inode* of this "new" directory and returns error.
- ✓ It releases "lock" of this "new" directory *inode* but keeps it allocated and also increments its reference count.
- ✓ Process's "*u area*" currently has "old" directory as the current directory. c*hdir()* is going to change that. Hence it releases this "old" directory's *inode* by *iput()* and stores *inode* of the "new" directory at that place.
- ✓ So now "new" directory is the "current directory" of the process and searching of file when path is not started with root (i.e. when path is relative) will being from this directory.

  Note -> The *inode* of this "new" directory will be released only when...
  1) either another call to *chdir()* is made.
  2) or when process exits.

✷ **System Call *chroot()* ->** *chroot()* system call changes the current root to a new root logically.

A process usually uses "global *File System* root" for file searching when path begins with "/" (i.e. absolute path). A process by calling *chroot()* may tell the *Kernel* that "this is my "new" root and hence when searching files for absolute path name, now onwards, use this "new" root as the logical root".

Recall that, *Kernel* contains a global variable which points to the *inode* of the global root. This gets allocated to the first process i.e. process 0 by *iget()* when the system boots. The syntax of *chroot()* is ....

chroot (pathname) ;

Where pathname is the “new” directory name, which will behave as “new root”. The algorithm is same like *chdir()* algorithm. *Kernel* stores the “new root” in process’s “*u area*” and unlocks its *inode* but keeps it allocated and increments its reference count too.

The difference is that as the “old root” i.e. (the default global root) is stored in a global variable, this algorithm does not need to release its *inode*.

So now onwards, the new directory (i.e. pathname) will be the “new root” of this process and also of its all child processes, and thus searching of files for absolute paths will begin from this new root.

✴ **System Call *chown()* ->** This system call changes ownership (of both, i.e. ownership user & ownership group) of a file. This is the operation on file’s *inode* and not on the file itself. The syntax is…

chown (pathname, owner, group) ;

Where “pathname” is as usual the path of file whose ownership is to be changed. “owner” is the string of new owner and “group” is the string of new group.

-> *Kernel* first uses *namei()* to parse the pathname and gets *inode* of the file.

-> Note that the “process owner” who is using this call, must be a super user, or must be owner of the file itself.

-> Then *Kernel* sets “owner” & “group” fields in the *inode* and then releases the *inode* by *iput()*.

-> After changes in the file’s ownership, the “old” owner looses “ownership” of this file.

✴ **System Call *chmod()* ->** This system call is used to change the mode (i.e. access permissions) for the file. The syntax is….

chmod (pathname, mode) ;

Where “pathname” is path of a file where modes are to be changed and “mode” is the combination of new modes (usually is octal).

-> The algorithmic flow is exactly same as that of *chown()*.

-> Just difference is that, in *chmod()* algorithm “access permission” field in the *inode* is set to the new modes (2nd parameter).

✴ **System Calls *stat()* & *fstat()* ->** *Inode* of a file contains various status flags like type, file owner, access permissions, file size, number of links, *inode* number, file access times (means all are “status fields” except the last “is member array”).

A process when wants this information, it can query the *Kernel* to give it to the process by *stat()* or *fstat()* system calls. Both works similar. The syntaxes are…

stat (pathname, statusbuffer) ;

fstat (file descriptor, statusbuffer) ;

Where “pathname” is the path of file, whose *inode*’s status you want and statusbffer is address of an *inode* data structure to store the status information returned by *Kernel*. “ file descriptor” parameter of *fstat()* is as usual.

-> Algorithmically, *Kernel* just uses *namei()* to get the *inode* of the file and then copies its all status fields into the supplied “*inode* data structure”.

✴ **Pipes ->** As seen before “pipe” allows flow of data in FIFO manner. They can also be used for synchronization of processes. Pipe even works when it does not know which processes are running on either side of it.

There are 2 types of pipe…

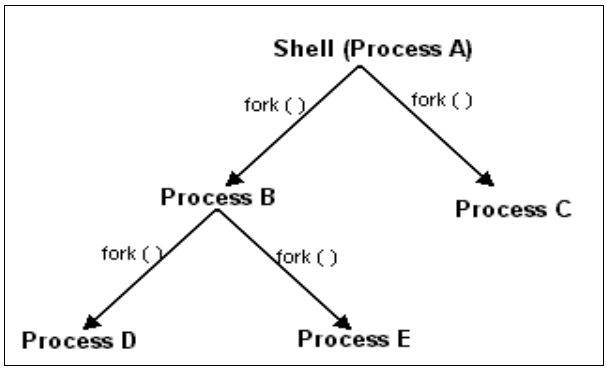
1) Name Pipe
2) Un-named Pipe.

: Both are identical almost. The difference like in “initialization of pipe” by process.

Means “Named Pipe” is opened by usual “*open()*” system call just like regular file. And “Un-named Pipe” is opened by “*pipe()*” system call. After creation, pipe is manipulated just like other

regular files. Means *read()*, *write()*, *close()* calls are as usual. (✶ *mknod()* system call is used to create the named pipe).

Another major difference occurs when we use pipe "multiprocessing environment". Means suppose "the shell" is process A and if it creates process B & process C by *fork()* system call as....



And then process B creates a pipe and also creates its two child processes, process D & process E.

Now if the pipe created by process B is a "named pipe", then all these processes, even all processes in the system can share the named pipe. Because "named pipe" is as good as the regular file having access permission for all processes.

But if the pipe created by process B is an "Un-name pipe" then the process which created it (i.e. process B) and children of it (i.e. process D & process E) can share the pipe. Process A, process C cannot share this Un-named pipe.

Our discussion of pipe will begin for "un-named pipes", because they are most commonly used.

✶ **System Call *pipe()* ->** The syntax of creation of pipe is....

pipe (fdptr) ; /* int fdptr [2] = {fd1, fd2} ; */

Where fdptr is pointer of an integer array of 2 elements, where elements are 2 file descriptors, one is for "reading" & another is for "writing".

As pipe is not existed before its use, *Kernel* must assign an *inode* for it after its creation. *Kernel* also allocates respective "*File Table*" entries (i.e. 2 entries in *File Table*) and also returns 2 respective file descriptors. Thus one file descriptor will be for reading and another will be for writing on the pipe. So everything is just like regular file and thus process need not know it is reading/writing a regular file or a pipe. The algorithm of *pipe()* system call is as follows.....

```
01 : algorithm : pipe()
02 : input :   pointer to an integer array with empty (i.e. mostly -1) elements.
03 : {
04 :           assign new inode from pipe device by using ialloc() ;
05 :           allocate 2 File Table entries, one for reading and other for writing ;
06 :           initialize File Table entries, i.e. "count" filed, pointer of inode etc ;
07 :           allocate 2 "User File Descriptor Table" entries and allow them to point to respective
08 :           "File Table" entries ;
09 :           set inode's reference count to 2 ;
10 :           initialize "count field" in pipe's inode to 1 for both "File Table" entires ;
11 :           /*  The pipe's inode has 4 extra fields ....
12 :           1) Byte offset in pipe
13 :           2) Reader count, for reader processes.
14 :           3) Writer count, for writer processes.
15 :           4) Read pointer & Write pointer.  */
16 : }
```

17 : output : pointer to an integer array, elements of which are now valid file descriptors,
18 : one for reading and other for writing (when they are parameters, they were
19 : empty) 0th element – Reader, 1st element – Writer.

✷ **Pipe Device ->** It is just like a *File System* (mostly kept in ***/dev*** directory) having "*disk inode* block" and "data blocks" parts of *File System* (i.e. pipe *File System* does not have boot block & *Super Block* parts). Thus *Kernel* can read "*disk inode* block" of pipe device and can assign an *inode* for a pipe. "Data block" part allows *Kernel* to assign blocks too.

We can imagine that, during system installation, system keeps some blocks aside, link them together, call one of them as "*disk inode* block of pipe" and call other blocks as "data blocks of pipe" and then "together" give a name "pipe device" and keeps it inside ***/dev*** directory.

So that, when a process requires a pipe, *Kernel* can enter in /dev and then picks up "pipe device" and treats it as new *File System*, which does not have boot block & *Super Block*, but has "*disk inode* block" and "data blocks" structures.

Usually system administrator gives a "pipe device" during system installation. But unlike regular files, *Kernel* cannot re-assign a "pipe *inode*" and "pipe data block" to another process while one process is already using a pipe. So pipe resource is non-sharable.

Above concept of "virtual device *File System*" is important because same logic is used for "mounting" another *File System*.

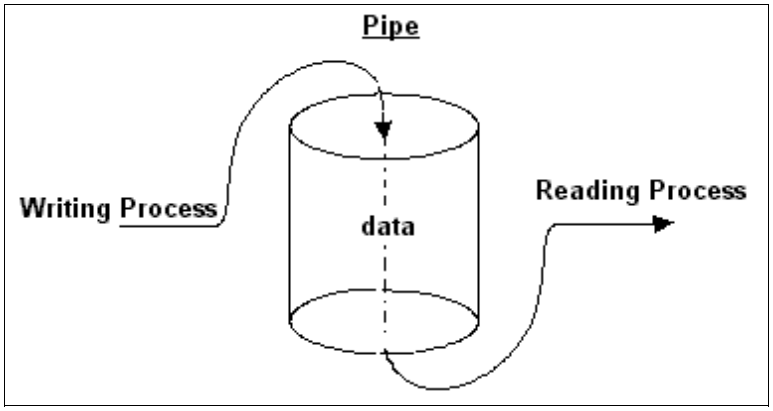
Now we will see the actual algorithm...

- ✓ *Kernel* first assigns a new *inode*, taking it from "*disk inode list* of pipe" of "pipe device" by using *ialloc()* algorithm.
- ✓ *Kernel* then allocates 2 *File Table* entries, one for "reading for pipe" and another for "writing on pipe". Accordingly it also updates *inode*'s fields in the *inode* of this pipe in "*In-core Inode Table*".
- ✓ Then it initializes "*File Table*" entries. Means in both "*File Table*" entries, *Kernel* keeps pointer of pipe's *inode*, set "count" field to 1 ,"capability flags" to "Read" & "Write" respectively.
- ✓ Then it sets the pipe *inode*'s reference count to 2. This step is important, because as "*File Table*" has 2 entries for pipe, ultimately process is going to have 2 file descriptors. This is nothing but the logic of "opening 1 file (i.e. pipe) with 2 file descriptors)". Hence pipe *inode*'s reference count is set to 2.
- ✓ The most important difference between "regular file's *inode*" and "pipe's *inode*" is that, for "regular file", the byte offset field is in "*File Table*" but for "pipe", the byte offset field is kept in "pipe's *inode*" itself.

  Hence *Kernel* sets "byte offset" field for pipe in pipe's *inode*, which indicates, from which position, the next I/O is going to be started. Obviously processes cannot use *lseek()* system call to set "byte offset" of a pipe. Thus "random access of pipe" is impossible. So pipe has only "sequential access".
- ✓ Lastly, *Kernel* initializes "Reader Count" & "Writer Count" fields in "pipe's *inode* to 1 each". These counts indicate how many reader processes & writer processes are working for this pipe.
- ✓ Return Values -> Corresponding to 2 *File Table* entries, *Kernel* creates 2 "*User File Descriptor Table*" entries, keeps pointer of respective "*File Table*" entries here and then return indices of this table as "file descriptors".

  Note that, initially array of file descriptors was passed to this system call as "invalid values". When system call finishes successfully these invalid values become "valid" and return to the calling process.

✷ **Reading & Writing Pipes ->**



"Writing process" is at the left side of pipe & "reading process" is at the right side of pipe. So "writing process" fills the pipe by putting data into it and "reading process" reads the data by pulling from it.

Thus processes access data from a pipe by FIFO manner. The figure does not mean that number of writing processes must be equal to the number of reading processes. If the number of writing & reading processes is greater than 1, then processes must co-ordinate the mechanism of using such a pipe.

With reference to *Kernel*, pipe is just like any other file. It assigns "data blocks" taken from the "pipe device" to store the data, during writing on pipe. The difference in regular file & pipe is that, the pipe uses only "direct disk blocks" from its *inode* while regular files may have "indirect disk blocks" too. So for greater efficiency, pipe's data storage is made small, limited to 10 KB (because there are 10 direct disk blocks on pipe's *inode*) assumed that block size of system is 1024 bytes. But this limited storage is for "present instance, means when "reading processes" empties the data from pipe, the writing process may fill it again. Obviously such repetitive work requires synchronization of process with the pipe. To do such repetitive activity *Kernel* manipulates 10 direct blocks on pipe's *inode* as "circular queue", having "Read Pointer" & "Write Pointer". Pipe's *inode* stores these 2 pointers, i.e. Read Pointer & Write Pointer of circular queue.

Four Cases Of Reading & Writing Pipe ->

1) Writing on a pipe having enough room for writing data.
2) Reading from a pipe having enough data to satisfy user's request.
3) Writing on a pipe, which does not have enough room for writing data.
4) Reading from a pipe, which does not have enough data to satisfy the user's request.

1) Writing On A Pipe Having Enough Room For Writing Data -> Here the sum of the number of bytes written and the number of bytes already present inside the pipe, is less than equal to the capacity of the pipe.

The *Kernel* follows same "writing algorithm" for the pipe as that of regular file, except that, *Kernel* increases pipe size automatically after every write processing. This is contrast with the regular file. *Kernel* will increase the size of regular file only when it writes data beyond current EOF.

Means suppose a regular file's EOF is at 5 KB. Data in it ends at 4 KB. Then remaining 1 KB may be empty. This may happen when a process writes 4 KB data in a regular file but when by *lseek()* advance to 5 KBth offset, writes something there of 1 byte size and then stops writing. So this is EOF. So when next time writing on the file can begin from 4 KBth byte + 1, filling 1 KB empty area. In this case too, file size will remain same of 5 KB if process does not write beyond that.

This is not the case for pipe. For pipe, every "write" processing increments size of pipe automatically.

Now a question arises, **if *Kernel* increments pipe size automatically during every "*write()*" system call, then what will happen when "10 direct block" limit finishes?** Here *Kernel* will just adjust the "file offset" field in process's "*u area*" to 0 to point to the beginning of pipe.

But note that *Kernel* never overwrites the data in pipe. Still it can reset the "byte offset" to 0 because it does not allow the data to be over-flown
beyond pipe's max capacity.

When writing of "writer process" finishes, *Kernel* updates pipe's "writer process" such that, the next writing will begin from the point where last writing stopped.

Then *Kernel* awakens all slept reading processes, which went to sleep until writing completes.

2) **Reading From A Pipe Having Enough Data To Satisfy The User's Request ->** When a "reading process" reads a pipe, first it checks whether the pipe is empty or not. If pipe contains data, then *Kernel* reads data from that pipe as it is reading from a regular file. Means by using *read()* system call. Its reading starts from the "pipe read pointer" stored in the pipe's *inode*. Means "pipe read pointer" also indicates "last read" action if any. As reading progresses, it is just like "draining contents of pipe", means as reading goes on, *Kernel* decrements size of the pipe according the number of bytes it actually "read" from the pipe. If necessary it can adjust the "offset" field in process's "*u area*" to go to the beginning of pipe if necessary.

When reading by "reading process" completes, *Kernel* awakens all those "writing processes" which went to sleep until reading of pipe completes.

Kernel also save the "current read offset" in the pipe's *inode* (Again recall that, for pipe, byte offset is stored in *inode* and not in *File Table*).

3) **Writing On A Pipe Which Does Not Have Enough Room To Keep Writing Data ->** *Kernel* here just marks the *inode* and goes to sleep waiting for "reading process" to drain the data from the pipe. When a process starts reading data, *Kernel* wakes up and also awakens all "writing processes" to start their writing on pipe. Obviously there may be "race conditions" between "writing processes".

So it is possible that process A & B sleep for draining of pipe by a reader process. Process A already put some data in pipe. As reading starts both will wakeup and during "race" between A & B, if B is scheduled first, then it may write its data after the data of A. So when A gets scheduled the written data inside the pipe is not contiguous with respect to A's data.

4) **Reading From A Pipe Which Does Not Have Enough Data To Satisfy User's Request ->** The reading will complete successfully but data does not satisfy the user's request. But this is the case when pipe at least has some data, less than the user's request.

But when pipe is empty, then "reading process" sleeps waiting some process to write in the pipe. When writer process starts writing the pipe, all slept "reader processes" wakes up for reading and again "race condition" will occur, and in "multi-writer" & "multi-reader" environment a reader process may read "unexpected data".

Above situation is true for "Un-named pipes". But if there is "Named pipe" and if it is opened with "No Delay" option, then if pipe is empty, reader process returns immediately instead of going to sleep.

✴ **Conclusion Of 4 Cases ->** The possible "Race Conditions" in (3) & (4) cases enforce the use of "synchronization between processes" when Multi-Reader & Multi-Writer processes work with a single pipe.

✴ **Opening A Named Pipe ->** Unlike "Un-named pipes", "named pipe" has directory entry and thus are accessed by "path name strings". So named pipes are more closer to regular files. As like regular files, named pipes permanently exist in the *File System*, while un-named pipes are transient. Thus to remove named pipe *unlink()* system call is used, which is not for un-named pipes.

Algorithm for opening a named pipe is identical to the algorithm for opening a regular file. But in case of named pipe, *Kernel* increments "read or write counts" in the pipe's *inode* to indicate how many processes opened this named pipe.

Note that, it is foolish thing to open a pipe for reading when there is no "writing process". Similarly it makes no sense in opening a pipe for writing when there is no "reading process".

Thus, if by mistake, a pipe is opened for reading and if there is no "writing process", then the "reading process" goes to sleep until some "writing process" opens this pipe for writing. Same is for opening a pipe for writing when there is no "reading process".

When respective process opens the pipe for reading / writing the processes waiting for writing / reading are awakened.

Sleeping of processes while opening a pipe, and if respective counter processes are not available, can be avoided by using "no delay" option. Where processes return immediately without sleeping when counter processes are unavailable.

✷ **Closing Of Pipes ->** For a process, closing of a pipe is similar to the closing of a regular file.

But *Kernel* needs some special processing for closing a pipe. Especially *Kernel* needs special processing for reading the pipe's *inode*.

While closing of a pipe, *Kernel* decrements the "reader & writer count" fields in pipe's *inode* accordingly.

1) If the "writer count" drops to 0, means if there remain no "writing process", and if "reader count" is greater than 0, means there are "reading processes" sleeping and waiting for "writing processes" to write on pipe, then as soon as "writer count" drops to 0, kernel awakens these "slept reading processes" which return from *read()* system call without reading any data, but no error.
2) If the "reader count" drops to 0, means if there remains no "reader process", and if "writer count" is greater than 0, means there "writing processes" sleeping & waiting for "reader processes" to read from pipe, then as soon as "reader count" drops to 0, *Kernel* awakens these "slept writing processes" which return from "write" system call with error.
3) When both "reader & writer" counts drops to 0, then *Kernel* realizes that there are no processes accessing the pipe. Thus *Kernel* frees all data blocks of pipe, adjusts pipe's *inode* such that pipe is now empty and releases the *inode*. It also frees the "*disk inode*" of an un-named pipe (because un-named pipe is not a permanent file) so that re-assignment become possible (this is not for named pipe, which is permanent).

✷ **Some Important Points About Pipe ->** Though pipe & regular file is similar for a process, the *Kernel* implementation of pipe is different. This is because the "byte offset" entry, which is in "*File Table*" for regular file and in "*inode*" for pipe.

Actually named pipe has permanent existence (just like any other regular file) due to its entry in directory table. So just like other regular files its "byte offset" entry can be kept in pipe's "*File Table*". But it is not done. Because system designers needed to consider "named" & "un-named" pipes consistent as "Pipe" only.

For the case of un-named pipe, as it is not permanent but transient, *open()* call will create different "*File Table*" entries & thus sharing is not possible (*dup()* & *fork()* do not work for pipe). So for un-named pipe sharable thing is *inode* and hence "byte offset" entry is made in *inode* instead of in *File Table*. To make this consistent with the idea of "Generalized Pipe", similar logic is used for named pipe too though not necessary for them.

✷ Example Of Creating, Reading & Writing An Un-named Pipe ->

```
# include <stdio.h>
char string [ ] = "hello" ;
void main (void)
{
        // variables
        char buffer [1024] ;
        char * ptr1, * ptr2 ;
        int fd [2] ;
```

```
        ptr1 = string ;
        ptr2 = buffer ;
        while (* ptr1 != '\0')
        {
                * ptr2 = * ptr1 ;
                ptr1 ++ ;
                ptr2 ++ ;
        }
        pipe (fd) ;
        for ( ; ; )  /* infinite loop */
        {
                write (fd [1], buffer, 6) ;
                read (fd [0[, buffer, 6) ;
        }
}
```

Here a global string variable is copied to the buffer. Then *pipe()* system call is made to get 2 file descriptors, one for reader (0^{th} element of array) and other for writer (1^{st} element of array). Then an infinite "for loop" is used, in which process writes data to one end of pipe (i.e.fd [1]).

Note that, though we expect that on one end of pipe there is a "writer process" on the other end, there is a "reader process", *Kernel* does not care whether these two processes on either sides of pipe are same or different. In above example, on both sides of pipe, "writer" & "reader" process is same.

✶ Example Of Creating, Reading & Writing A Named Pipe ->

```
# include <stdio.h>
# include <fcntl.h> /* for mknod() */
char string [ ] = "hello" ;
void main (int argc, char * argv [ ] )
{
        // variables
        int fd ;
        char buffer [256] ;
        // code
        /* current named pipe */
        mknod ("temp", <a dummy number>, 0) ; /* 0 indicates RW permissions to all */
        if (argc == 2)
                fd = open ("temp", O_WRONLY) ;
        else
                fd = open ("temp", O_RDONLY) ;
        for (  ;  ;  )  /* infinite loop */
        {
                if (argc == 2)
                        write (fd, string, 6) ;
                else
                        read (fd, buffer, 6) ;
        }
}
```

Here a "command line argument" having process is used, expecting that user will give 2 arguments of which second will be considered as dummy.

By using *mknod()* system call a named pipe with "temp" name is created and given Read/Write access to all users.

If number of arguments is 2, then pipe is opened as "write only". Else the pipe is opened as "read only".

Then an infinite loop is used and inside it, if number of arguments are 2, then the global string is written in the pipe. Else process reads whatever contents of pipe into the buffer.

Here too, processes on both sides of pipe are the "same above program". But as it is a named pipe, processes on either sides of pipe may be "any". They need not to be related (unlike un-named pipe).

✴ **System Call *dup()* ->** This system call duplicates i.e. copies a file descriptor into the first free slot of the *User File Descriptor Table* and returns a new file descriptor. It works for all file types. The syntax is ....

newfd = dup (fd) ;

Where "fd" is the file descriptor that we wanted to duplicate and "newfd" is the new file descriptor returned by the *dup()* system call.

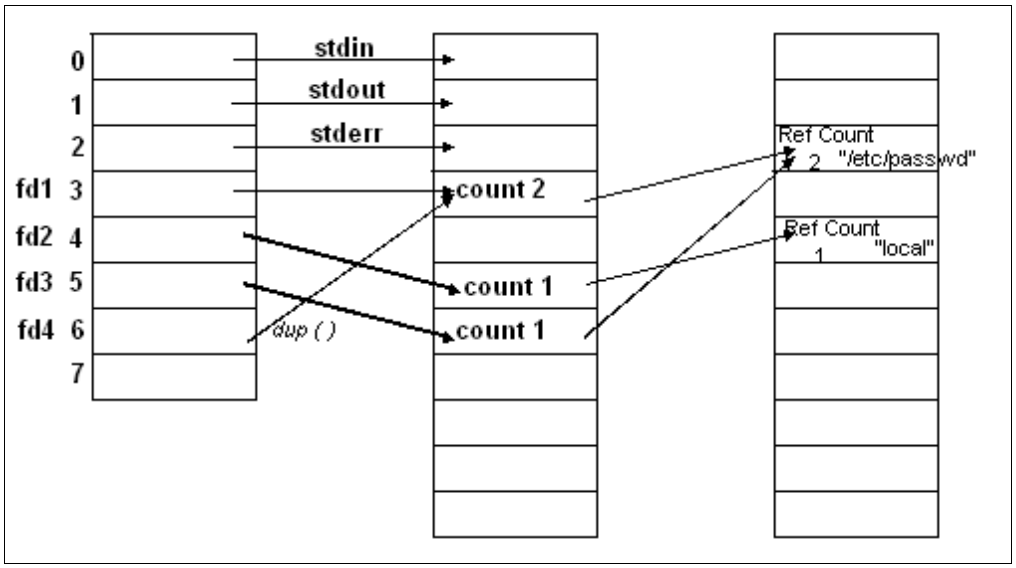
As *dup()* system call duplicates a file descriptor but points or references to the same file as that of old file descriptor, both file descriptor reference a same file and hence point to same *File Table* entry.

Obviously, "count" field in "*File Table*" gets incremented.

Suppose a process executes following piece of code....

fd1 = open ("/etc/passwd". O_RDONLY) ;
fd2 = open ("local". O_RDONLY) ;
fd3 = open ("/etc/passwd". O_WRONLY) ;
fd4 = dup (fd1) ;

The data structures "*User File Descriptor Table*", "*File Table*" and "*In-core Inode Table*" after above code's execution will be...



If we assume that first file descriptor i.e. fd1 is at index 3 (as 0, 1, 2 are for standard terminal) which is for "/etc/passwd", then second file descriptor i.e. fd2 is of index 4 which is for file "local" and the third file descriptor i.e. fd3 is at index 5, which is again for "/etc/passwd" file.

Then *dup (fd1)*, duplicates fd1 descriptor and returns fd4. As the first empty slot in "*User File Descriptor Table*" is at index 6, fd4 returns index 6. Now both fd1 & fd4 points to same "*File Table*"

entry for "/etc/passwd" which is O_RDONLY and hence "count" field in this *File Table* entry is incremented to 2 which was 1 when there was no *dup()* call for fd1.

So arrows drawn from index 3 & index 6 point to same entry in "*File Table*", breaking "one to one" relationship between "*User File Descriptor Table*" & "*File Table*" seen up till now.

We will take another example of a "C" code having *dup()* system call....

```
# include <stdio.h>
# include <fcntl.h>
void main (void)
{
        // variables
        int i, j ;
        char Buffer1 [512] , Buffer2 [512] ;
        // code
        i = open ("/etc/passwd", O_RDONLY) ;
        j = dup (i) ;
        read (i, Buffer1, sizeof (Buffer1)) ;
        read (j, Buffer2, sizeof (Buffer2)) ;
        close (i) ;
        read (j, Buffer2, sizeof (Buffer2)) ;
}
```

"i" is the file descriptor returned by *open()* system call when opening "/etc/passwd" file. "j" is the file descriptor returned by *dup()* system call while duplicating the "i" file descriptor.

In process's "*u area*" both i & j file descriptors will point to same *File Table* entry and hence will point to same "byte offset" of the file.

Thus first *read()* system call will read the file from 0 to 511th byte and second read call will read from 512th to 1023rd byte.

This is in sharp contrast with the example of "Reading of same file via two file descriptors". As in that example the two file descriptors were created with 2 separate *open()* calls, both have different "*File Table*" entries and hence Buffer1 & Buffer2 contain identical data as both *read()* calls starts from 0th byte of file.

Here, though there are 2 file descriptors and though both are reading the same file, the contents of Buffer1 & Buffer2 will never be identical. Because as one file descriptor, out of the two, is created by duping the other by *dup()*, both will not have separate *File Table* entries. Rather both will point to same *File Table* entry. Hence second *read()* call will take over the place where first *read()* call ends. This is due to the fact that when first *read()* call ends, the "offset" field in *File Table* is set to the last read byte and during second *read()* call, process's "*u area*" is already updated to the byte where last *read()* call ends. Hence second *read()* call will start reading the file where last *read()* call ends.

This example shows another important fact. If one of the file descriptor is closed (in this example "i" file descriptor is closed), the other file descriptor can take over all file I/O just as normal as if is the only one.

In real world examples, processes usually close "stdout" file descriptor (i.e. index 1) and *dup()* any desired file descriptor. As *dup()* creates next file descriptor at first empty slot in "*User File Descriptor Table*", it will be created at index 1 (because closing *stdout* makes that slot empty) and then treat the file as if it is "stdout". So can be used further in "indirection" or "pipe" logic.

✷ **Mounting A *File System* ->** During system installation, disk driver partitions disk in logical sections. Each logical section is given a device file name and usually stored under /dev directory (just like /dev/pipedevice).

In 2nd chapter, we said that *File System* has boot block, *Super Block*, *disk inode list* & data blocks. Now in "data blocks" part of the primary *File System*, there may be another logical section on

disk having its own boot block, *Super Block*, *disk inode list* & data blocks. We will call this logical section of disk as "secondary *File System*". (Note – this name is given just for understanding purpose).

So *mount()* system call is used to connect such a secondary *File System* to the primary *File System*. In other words, *mount()* system call connects the logical *File System* in a specified section of disk to the existing *File System* hierarchy. Similarly *umount()* system call is used to disconnect the *File System* from hierarchy.

Usually "secondary file systems" are located on another partition of disk. If such a partition is not directly accessible in primary *File System* hierarchy, then *mount()* system call is used and after connection we can access that partition in our existing *File System* hierarchy. Thus such blocks now become "secondary *File System*" (of course logical) instead of just disk blocks.

The syntax of *mount()* system call is....

mount (special pathname, directory pathname, options) ;

Where ....

special pathname -> It is the path of "Block Device Special File" of the disk section containing the logical secondary *File System*, which we want to mount.

directory pathname -> It is the path of *directory file*, in the existing file hierarchy, where the mountable *File System* is to be mounted. (This directory is usually called as *Mount Point*).

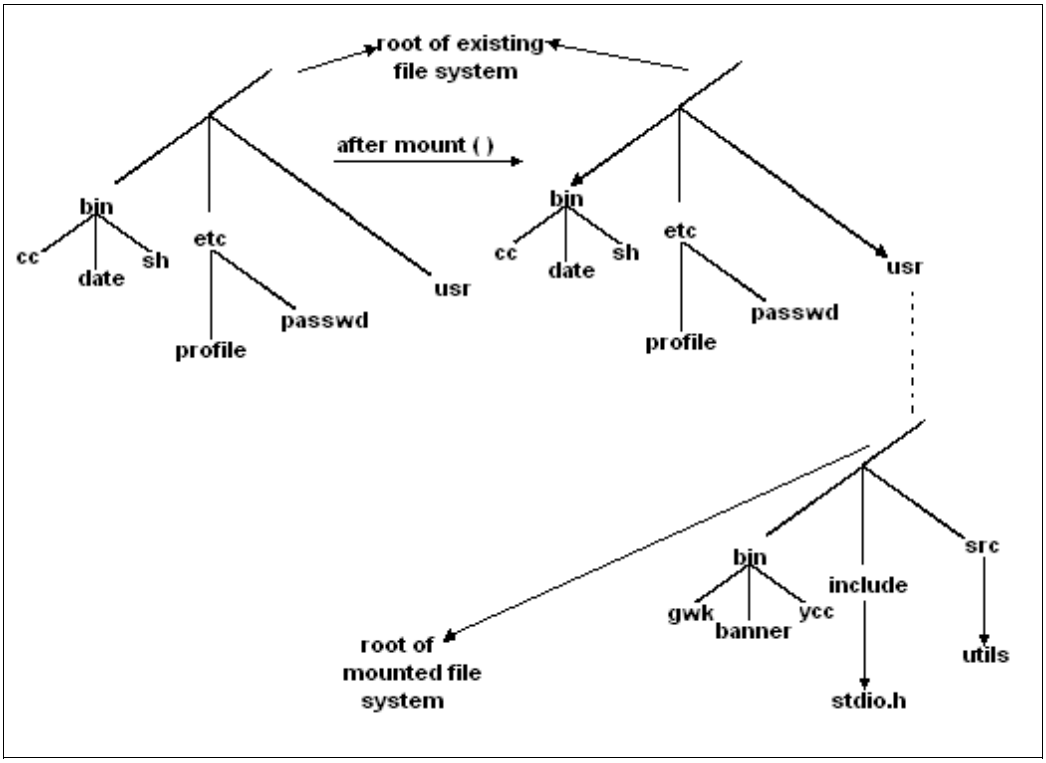
options -> This indicates whether the mountable *File System* should be mounted as "read-only" or not.

Example -> Suppose in "/dev" directory there is a file "dsk1", which has mountable *File System* (i.e. secondary *File System*) and we want to mount it in "/usr" directory, as "read-only". Then the mount system call will be.....

mount ("/dev/dsk1", "/usr", 0) ;

As "/dev/dsk1" is "Block Special" type device file, it represents partition or partition of a disk containing some data. So *mount()* system call force *Kernel* to assume that the special portion of disk contains *File System* having its own *Super Block*, *inode list*, disk blocks & a *root inode*. (Note – this root *inode* is of root of the mountable *File System*, and not the existing *File System*'s root).

So after completion of *mount()* system call, the root of the mountable (now we should say "mounted", because *mount()* call is completed) *File System* is accessed by the name "/usr".



Now any process can access data in mounted *File System* as if it is a part of existing *File System*. So processes don't need to know that the mounted file system is say "attached" and not the "fixed" part of existing *File System*.

But this does not mean that a process can create link for a program present in mounted *File System*, on the exiting *File System*. This is because system V does not allow files to get linked across multiple file systems. So *link()* is such a system call which checks *File System* and allows to create link for a program of exiting *File System* on exiting *File System*.

✷ **Mount Table ->** *Kernel* maintains a special data structure for *mount()* & *umount()* system calls, called as the *Mount Table*. This table contains entries of every mounted *File System* in exiting *File System*. Each entry contain …

1) Device number of the mounted *File System* (recall that device files have their major & minor numbers in their inodes).
2) Pointer to a buffer which contains *Super Block* of the mounted *File System*. (Recall that mounted *File System* has its own *Super Block*, *inode list* block, data blocks etc).
3) Pointer to the root *inode* of the mounted *File System* (Recall that mounted *File System* has its own root. In one previous example "/" of "/dev/dsk1" is the root *inode* of mounted file system).
   This may confuse us that our exiting *File System*'s root i.e. "/" & "/" of "/dev/dsk1" are one and the same. But if you see figure on last page, the difference will be clear.
4) Pointer to the directory (i.e. mount point) *inode*. (i.e. in our previous example "usr" of one root file system).
   The most important point of mount system call is that, it associates "mount point inode" to the "mounted *File System*'s root inode", such that process can see as if the mounted *File System* is already a part of existing root *File System* rather that just a temporary attachment.

The algorithm of *mount()* system call is as follows –

```
01 : algorithm : mount()
02 : input :   1) file name of Block Special file
03 :           2) directory name of "mounted on"
04 :                   3) options (read - only).
```

```
05 : {
06 :   if (user is not super user)
07 :        return (error) ;
08 :   get inode of the Block special file by namei() ;
09 :   make legality checks;
10 :   get inode of "mounted on" directory using namei() ;
11 :   if (it is not directory or reference count > 1)
12 :   {
13 :            release inode of Block special file by iput() ;
13 :            release inode of "mounted on" directory by iput() ;
14 :            return (error);
15 :   }
16 :   find empty slot in Mount Table ;
17 :   involve block device driver's open routine ;
18 :
get a free buffer from buffer pool ;
19 :   read "Super Block" of mounted File System into this buffer ;
20 :   initialize Super Block's fields ;
21 :   get root inode of "mounted file system" by iget() and save it in the Mount Table ;
22 :                                        /* other Mount Table entries too */
23 :   mark inode of "mounted on" directory as "mount point" ;
24 :   release inode of Block special file by iput() ;
25 :     unlock inode of mount point directory ;  /* don't release*/
26 : }
27:  output : nothing
```

- ✓ First *Kernel* checks the level of user. As *mount()* can be called by super user only, if the user is not the super user, *Kernel* returns error.
- ✓ Then *Kernel* gets inode of the *Block Special File* (1st parameter) by using *namei()* algorithm. This *Block Special File* has the "mountable *File System*", i.e. the secondary *File System*.
- ✓ Then *Kernel* makes "legality checks" on the returned *inode*.
- ✓ Then *Kernel* gets *inode* of the "mounted on" directory by using *namei()* algorithm. This is the directory on which the new *File System* is going to be mounted. (Obviously this directory is part of existing *File System*).
- ✓ After getting *inode* of "mounted on" directory, *Kernel* checks whether the received *inode* is really directory or not. *Kernel* also checks "reference count" field of this *inode*. This reference count must not be greater than 1. Because 1 indicates that the process, which is calling mount system call, is the only process accessing this directory. Because if this count is greater than 1, then some other process is also accessing the directory. As the "mounted *File System*" tree is going to be lodged in this directory, one should prevent other processes to create or to remove subdirectories under this directory.

  Thus if reference count of "mounted on" directory is greater than 1 or if "mounted on" directory is really not a directory, *Kernel* releases both *inodes* (i.e. of *Block Special File* and of *directory file*) and return error.
- ✓ *Kernel* then finds "empty slot" in the *Mount Table* and if not found it allocates new empty slot in the *Mount Table*.

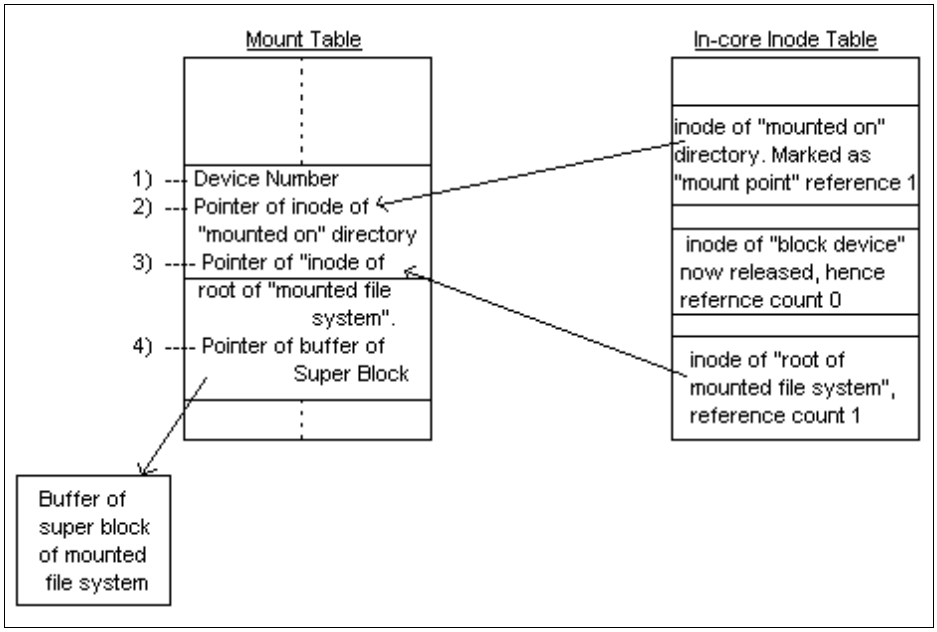
  It also marks this slot as "in use", so that other processes should not create their *Mount Table* entry in this slot.

  Then it writes device number of *Block Special File* (received by reading major & minor number in the *Block Special File*'s *inode*).

✴ Note that, writing of device number of *Block Special File* (i.e. of mountable *File System*) into the empty slot of *Mount Table* is done as immediately as possible because in later part of code, this process may sleep, and other process may attempt to mount some other *File System*. (Process may sleep while opening device driver's open routines in next step).
So by marking the "*Mount Table*" entry as "in use", *Kernel* avoids two processes to mount *File Systems* in same *Mount Table* slot. And by writing device number in the slot it prevents other processes from mounting same *File System* again. (It shows that at a time only 1 *File System* can be mounted once only).

- ✓ Then *Kernel* calls *open()* procedure for the block special device driver. This procedure checks legality of device and then initializes the hardware (i.e. Flippy, CD-ROM, other Hard Disks, etc. on which mountable *File System* is present). Process may sleep during this step.
- ✓ Then *Kernel* allocates a buffer (to read data from the mountable *File System*) from the buffer pool. Note that, this is done by variation of *getblk()*, not the same *getblk()* we had seen in *Buffer Cache*.
- ✓ Then it reads "*Super Block*" of mounted *File System* into this buffer using a variation of *read()* algorithm. Not the same *read()* system call we up till seen.
- ✓ Then *Kernel* initializes the fields of *Super Block*.
  Especially,
  - It clears the lock fields for *Super Block*'s "free block list" and "free *inode list*".
  - Sets number of free *inodes* in the *Super Block* to 0. This step is particularly important. This step forces *ialloc()* algorithm to search the disk for free *inodes*, every time, even after system crash. (important in "autmount" option).
  - If *mount()* system call is made with "read-only" option, then *Kernel* sets a flag in this *Super Block*.
- ✓ *Kernel* gets *inode* of the root of mounted *File System* and stores pointer of it in the *Mount Table*. Also it stores pointer of "mounted on" directory in the *Mount Table*. This is necessary when process cross mount point during *namei()* file name parsing.
  To any process, the "mounted on" directory (i.e. "/usr" in our example) and the "root *inode* of mounted *File System*" are equivalent and *Kernel* establishes their equivalence by putting them in the same *Mount Table* entry.
- ✓ Finally *Kernel* makes the *inode* of "mounted on" directory as "mount point", so that other processes may identify it as mount point.
- ✓ As now onwards the *Block Special File*'s *inode* is not necessary, it is released by *iput()*.
- ✓ But the *inode* of "mounted on" directory is unlocked only and not released. Thus allowing other processes to traverse the mounted *File System*. But as it marked "mounted on", other processes, which don't have permission, can't write on it.
- ✓

✴ **Figure Of Data Structures After *mount()* ->**



✴ **Crossing Mount Points In File Path Name Search - >** If a *File System* is mounted and processes starts using the mounted *File System* as if it is the part of existing root file system, then there are some cases, especially during "search", where the "mount point" directory is crossed.

Suppose following sequence of shell commands is executed...

mount /dev/dsk1 /usr -> 1)
cd /usr/src/utils -> 2)
cd ../../.. -> 3)

- ✓ In first command, as our previous example, /dev/dsk1 *File System* is mounted on /usr directory.
- ✓ ***cd*** command, then uses "*chdir()*" system call and *Kernel* starts parsing pathname "/usr/src/utils" by *namei()*. Here while parsing it considers "/" as root of existing *File System*, then takes component "usr" which is our mount point. So in this command, there is crossing of mount points from "mounted on" *File System* (i.e. primary, existing *File System*) to the "mounted" *File System* (i.e. secondary *File System*).

  So after crossing the mount point, now the current environment is "utils" directory, which is in "mounted" *File System*.

- ✓ The third command ***cd*** results *Kernel* to switch from "utils" to its parent directory, means in "src", then to the parent of "src", means in "usr" and then to the parent of "usr" i.e. in "/". In the last switching (means in last ".."), *Kernel* again crosses the mount point. But this time crossing of mount point is from "mounted" *File System* to the "mounted on" *File System*.

✴ **Case 1) -> Crossing Of "mount point" from the "mounted on" *File System* to the "mounted" *File System*. ->** In the algorithm of *iget()*, on line 14, there is a comment - **/* here should be special processing for mount point */**. Now its time to consider this special processing. The special lines added to the algorithm after above comment are...

```
if (inode is a "mount point)
{
        find Mount Table entry for mount point ;
        get new File System number from Mount Table ;
        use mounted File System's root inode number in search ;
        continue ; /* loop back again */
```

```
}
```

The algorithm of "mount point crossing" in *iget()* states ...
-> If *inode* is marked as "mount point", then...
a) Find the *Mount Table* entry for this mount point.
b) From its *Mount Table* entry note the device number (i.e. *File System* number).
c) Also get the "mounted *File System*'s root *inode* number" from the *Mount Table*.
d) And then continue the search onwards now in mounted *File System*.

In our case, when *Kernel* arrives at /usr, and gets *inode* for "usr", it finds that, it is marked as "mount point" and then follows above steps.

✷ **Case 2) -> Crossing of "mount point" from the "mounted *File System*" to the "mounted on" file sytem ->** Now see, for the *namei()* algorithm. There are changes for special processing of mount point in *namei()*.

a) Add a goto identifier.
b) Add a "if" block.

a) Add a goto identifier -> Goto line 14, and after this line add following identifier..

ComponentSearch :

b) Add an "if" block -> Now go to line 19 (after adding above goto identifier, this will be line 20), and after it add following code...

```
if (found inode is of root and if working inode is also root and if the component name is "..")
{
        /* then this is crossing of mount point */
        get Mount Table entry for working inode ;
        release working inode by iput() ;
        working inode = "mounted on" File System's inode ;
        lock "mounted on" File System's inode ;
        increment reference count of working inode ;
        goto "ComponentSearch" for ".." ; /* i.e. goto identifier */
}
```

The algorithm states that...

- ✓ When a component matches with a directory entry, *Kernel* gets *inode* number of matched component, as usual in *namei()*.
- ✓ But now it checks this *inode* for "mount point". If found *inode* is of "root" and if "working *inode*" is also of "root" and the component is "..", then *Kernel* identifies that this *inode* is a "mount point".
- ✓ Then it finds the *Mount Table* entry whose device number is equal to the device number of last found *inode*.
- ✓ Then it gets the *inode* of "mounted on" directory and continues its search for ".." component, by using "mounted on" directory's *inode* as current working *inode*.

  According to our example, now current directory is "/usr/src/utisl". So when "cd **../../..**" command is given *namei()* starts parsing of "**../../..**" pathname. As pathname does not begin with "/", it is a relative pathname and hence search begins from current directory, i.e. from /usr/src/utils. So this is the starting "working *inode*" i.e *inode* of "utils".

  When first "**..**" component arrives, *Kernel* goes to its parent directory (recall that "**..**" stands for parent directory) means to "/usr/src".

  When second ".." component arrives, *Kernel* goes to its parent directory, means to "/usr". So now the working *inode* is of "usr".

When third ".." component arrives, *Kernel* finds that this is a root *inode* "usr", also by previous search, the working *inode* is also "usr" and the component is ".." so this must be the mount point.
So *Kernel* finds *Mount Table* entry for "usr" mount point, releases the current working *inode*, makes current working *inode* equals to the "mounted on" *File System*'s *inode*, locks the *inode*, increments the reference count of new working *inode* and goes to the "goto identifier" to start further searching in "mounted on" *File System*. Note that, ".." of "usr" is "/", which is root of our existing (primary) *File System*.

✴ **Unmounting A *File System* ->** The system call used for un-mounting a *File System* is *umount()*, whose syntax is .....

umount (special file-name) ;

Where "special file name" is the *File System* to be un-mounted.

The algorithm is...

```
01 : algorithm: umount()
02 : input: special filename of File System to be un-mounted
03 : {
04 :   if (user is not super user)
05 :         return (error) ;
06 :   get inode of special file name by namei ( );
07 :   extract major & minor number of un-mountable device ;
08 :   get Mount Table entry ;
09 :   release inode of special file by iput() ;
10 :   remove "shared text entries" from "region table" for file belonging to un-mountable File System
;
11 :   update Super Block, inodes and flush buffer ;
12 :   if (files from un-mountable File System are still in use)
13 :         return (error)  ;
14 :   get root inode of mounted File System from Mount Table;
15 :   lock inode ;
16 :   release inode; /* which was not released in mount() */
17 :   invoke close entries for special device ;
18 :   invalidate buffers from un-mounted File System ;
19 :   get inode of "mount point" directory from Mount Table ;
20 :   lock inode;
21 :   clear "mount point" flag marking from inode ;
22 :   release inode; /* which was not released in mount() */
23 :   free buffer used for mountable File System's Super Block ;
24 :   free Mount Table's slot for mountable File System ;
25 : }
26 : output : nothing
```

→ If user is not the super user, call to *umount()* fails and *Kernel* returns error.
→ *Kernel* by using *namei()*, gets the *inode* of the special file.
→ From this *inode Kernel* extracts Major number & Minor number of the device of special file.
→ By using device number (major number), it finds the *Mount Table* entry.
→ Releases *inode* of special file by *iput()*.

- → Some processes may use “include files” i.e. headers or libraries from the un-mountable *File System*. So *Kernel* removes such file’s “shared text entries” from the “region table”. Note that, these “shared text entries” must not be operational.
- → Then *Kernel* updates respective *Super Block* and write back to disk. Also it updates respective *inodes* and writes them back to disk. Some buffers may be “delayed write”, hence it flushes such buffers from buffer pool.
- → Some files from the un-mountable *File System* may be active (active files can be identified by positive reference count in their *inodes*) and there may be some files from un-mountable *File System*, which are yet open (means not close). In both these cases *umount()* fails and returns error.
- → As “root *inode*” of un-mountable *File System* is still there since *mount()*, it releases this *inode*.
- → Then as *mount()* calls device’s open routines, now *umount()* calls device’s *close()* routines.
- → As after *umount()* call, data on buffers of mounted *File System* is not going to be needed, it invalidate all such buffers. But while invalidating buffers,it moves them to the beginning of “*Free List* of buffers”, so that valid buffers should stay longer time in the buffer pool.
- → Then it gets the “mounted on” directory’s *inode* and removes “mount point” mark from it, which was set during *mount()* call.
- → Then it releases this “mounted on” directory’s *inode*.
- → Finally it frees the buffer of *Super Block* of un-mountable *File System* (which was allocated in *mount()* call).
- → And then free the “*Mount Table*” slot of this un-mountable *File System*.

✷ **System Call *link()*** :- > The *link()* system call links a file to new name in the *File System*’s directory structure, thus creating a new directory entry for an existing *inode*. The syntax of *link()* system call is…

Link (source filename, target filename) ;

Where “source filename” is the name of an existing file, while “target filename” is the new, additional name to the source file given after completion of *link()* system call.

Processes can access the file with any of the name, either by existing name or by its linked name. as *Kernel* does not know by which name you are calling the file (either original or linked), no special treatment is given to the “linked name” file.

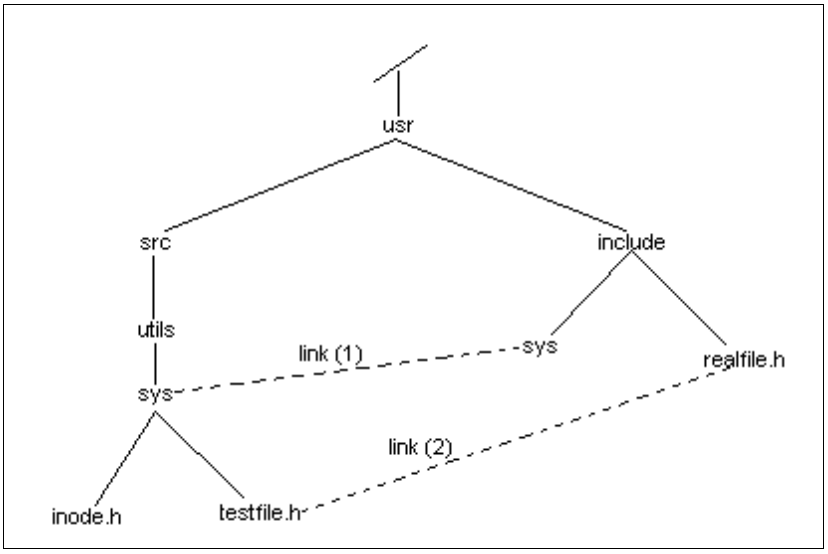
Suppose following 2 lines of code are executed….

link (“/usr/src/utils/sys”, “/usr/include/sys”) ;

link (“/usr/include/realfile.h”, “/usr/src/utils/sys/testfile.h”) ;

Here “/usr/src/utils/sys/testfile.h”, “/usr/include/sys/testfile.h” and “/usr/include/realfile.h” are referring to the same file. ***How?*** See following figure…

SEE FIGURE ON NEXT PAGE



In the first call "/usr/src/utils/sys" is given a new name "/usr/include/sys". And in second call "/usr/include/realfile.h" is given a new name "/usr/src/utils/sys/testfile.h".

So, "/usr/include/realfile.h" is an original file (shown by dark line). But "/usr/include/sys/testfile.h" is also the same, if you follow the link (1) in figure. Also "/usr/include/realfile.h" is same too, if you follow link (2) in figure.

Note that, through a link can be created on any type of file, only super user is allowed to create link for *directory file*. Although newer versions have *mkdir()* system call which eliminates the need of linking the directories.

The power to super user to link directory was necessary, because ordinary user may link a directory to another which is below it and this will fall *Kernel* in infinite loop while traversing the path of such linked directories.

The algorithm for *link()* system call is...

```
01 : algorithm : link()
02 : input :   1) existing file name
03 :           2) new file name
04 : {
05 :   get inode for existing file name by namei() ;
06 :   if (too many links for existing file are already there or if user is not super user for linking a
07 :   directory)
08 :   {
09 :           release inode of the existing file by iput() ;
10 :           return (error) ;
11 :   }
12 :   increment "link count" on the inode of existing file ;
13 :   update disk copy of inode accordingly ;
14 :   unlock the existing file's inode ;
15 :   get inode of the "parent directory" by namei() ;
16 :   /* this is the directory which is going to contain link */
17 :   if (the name given to new file already exists or the same named file is existing on different
18 :       File System)
19 :   {
20 :           undo the changes done above l
21 :           return (error) ;
22 :   }
```

```
23 :   create new directory entry in the parent directory for this new file ;
24 :   release parent directory's inode by iput() ;
25 :   release existing file's inode by iput() ;
26 }
27 : output : nothing
```

→ First *Kernel* gets the *inode* of existing file by using *namei()* algorithm.
→ Then looking inside the *inode*, it gets current number of links to the existing file. If this "link count" is too much, means if currently the existing file already has too many links, the algorithm fails, releases the *inode* of existing file, and returns error. Also if the user is not super user & the link is for directory, then as only super user has power to link directory, algorithm fails, releases *inode* of existing file & returns error.
→ On success from above 2 conditions, *Kernel* is ready to make link. Thus it increments "link count" in the *inode* of existing file.
→ It also immediately updates disk copy of this *inode*. This step is important and see later.
→ Then it unlocks this *inode*. This is also very important.
→ Now creating links means to create a new entry in a parent directory. So *Kernel* gets *inode* of the parent directory by searching the path of "new filename" using *namei()* algorithm.
→ If this same name already exists inside the parent directory or a file with the same name already exist in different *File System* (**Recall from *mount()* that, *link()* cannot span over different *File Systems***) , *Kernel* undoes the changes made above, means decrements the incremental link count of existing file and returns error.
→ But if it succeeds from above conditions, it creates a directory entry in the parent directory and puts directory entry of new filename & its *inode*, inside that parent directory.
→ Finally it releases both *inodes* by *iput()* and algorithm finishes….
Thus at the end of *link()* call, the "link count" field in existing file's *inode* will be +1 that of the beginning.
Note that "reference count" field & "link count" field in *inode* are totally different. Reference count keeps track of how many processes had accessed the file, while "link count" keeps track of how many directory entries (i.e. new file names) this file has. So even if reference count reaches to 0, the link count of existing file is at least 2. One for "existing filename" and second for "new file name".

✴ **Two Dead Lock Possibilities in *link()*** -> The step "unlocking the existing file's *inode*" of the incrementing its "link count" is of the paramount importance. If this step is removed ; deadlock situation occurs.

Suppose there are 2 processes A & B. Process A is executing *link()* call as….
link ("a/b/c/d", "e/f/g") ;
And process B is executing *link()* call as…
link ("e/f", "a/b/c/d/e") ;
During processing of *link()* algorithm, suppose process A is now scheduled and it finds *inode* of "d" (means *inode* of "a/b/c/d"), and allocates it.
Now process B is scheduled and it also finds & allocates *inode* of "f" (means *inode* of "e/f").
Again process A is scheduled, as "e/f" is already allocated to process B and locked by it, process A cannot get its new file name's *inode* as it has the name "e/f". Hence process A went to sleep.
If process B gets scheduled, it can also not get *inode* of its new filename, because its new filename is "a/b/c/d", which is already allocated & locked by process A. Hence process B also sleeps.
So proce
ss A is holding an *inode* which is wanted by B and process B is also holding an *inode* which is wanted by A. This is example of classic deadlock.

To avoid such deadlock, it is better, rather best to unlock the existing file's (i.e. first parameter's) *inode* after incrementing its "link count".

Even a single process can also finds itself in deadlock if it issues *link()* call as...

link ("a/b/c", "a/b/c/d") ;

Here process gets *inode* of "c" (i.e. of "a/b/c") and locks it during *namei()*. If the "unlock" statement is removed from the algorithm, when process tries to get *inode* of "d" by *namei()*, it cannot get *inode* of "a/b/c" as it is already locked by previous *namei()* so process goes to sleep and never wakes up.

What is the effect of such deadlock on the system? Though two or one (as explained above) processes go to sleep, other processes just work fine and system may seems to have no problem, until any other processes will require pathname searching including above file paths.

But if such files are of "/bin" or "/usr/bin" or "/bin/sh" then there will be disaster because these directories contain "continuously needed programs".

System Call *unlink()* -> The *unlink()* system call removes a directory entry for a file. The syntax of *unlink()* system call is....

unlink (pathname) ; /* pathname is as usual */

If above call is successful, no process can access the file referenced by "pathname". But if the file has other links, then can be accessed by these other link names.

The algorithms of *unlink()* system call can be given as...

```
01 : algorithm : unlink()
02 : input : pathname
03 : {
04 :   get "parent directory's" inode by namei() ;
05 :   /* if unlinking of current directory is going on */
06 :   if (last component of pathname is ".")
07 :           increment current directory inode's reference count ;
08 :   else
09 :           get inode of the file to be unlinked by iget() ;
10 :   if (the file is directory but user is not super user)
11 :   {
12 :           release both above inodes by iput() ;
13 :           return (error) ;
14 :   }
15 :   if (this file is a "shared text file" and it its inode's link count is currently 1)
16 :           remove "shared text entries" from region table ;
17 :   write into the parent directory ; /* inode number 0 */
18 :   release inode of parent directory by iput() ;
19 :   decrement "link count" in file's inode ;
20 :   release file's inode by iput() ;
21 :   /* here I this step, first iput() checks whether link count is 0, if it is, then iput()release file blocks
22 :      (by free()) and also frees the inode by ifree() */
23 : }
24 : output : nothing
```

→ *Kernel* first gets the *inode* of "parent directory" in which the un-linkable file resides. For this *Kernel* uses a variation of *namei()* algorithm (i.e. not the same *namei()* up till we used).

→ Then it checks whether the un-linkable file is a current directory, if it is, then the last component in the pathname should be obviously "**.**" So *Kernel* here just increments the current directory's *inode*'s reference count.

→ Else *Kernel* gets *inode* of un-linkable file using *iget()*.
→ If the un-linkable file is directory and the user is not the super user, then it releases the both above *inodes* and returns error.
→ If this un-linkable file is a "shared text file" by some other processes, and the "link count' of this un-linkable file is 1 (means it is an original file and not the link), then *Kernel* removes its "shared text entries" from region table (as also explained in *umount()*).
→ Then *Kernel* removes un-linkable file's directory entry from the directory. Here writing 0 for the *inode* of this file is enough. Also it updates this "write" on disk too.
→ Then decrements "link count" in un-linkable file's *inode*.
→ Releases both *inodes*, i.e. *inode* of parent directory and *inode* of un-linkable file by using *iput()*.
→ If the un-linkable file *inode*'s reference count & link count, both drop to 0, *Kernel* releases file's block by *free()* and also frees the *inode*. While freeing the blocks, *Kernel* first frees direct blocks & then recursively frees indirect blocks if any. It "zeros out" the block numbers in "table address" array of file's *inode* and then sets file size to 0. While freeing *inode* of the file by *ifree()*, it sets "file type" field to 0, so the *inode* is now free for reassignment. But note that, disk copy of this *inode* is still pointing to an existing file. So above changes made in in-core *inode* are immediately written to disk.

✶ ***File System*** **Consistency ->** *Kernel* takes some precautions while maintaining *File System* consistency.
1) When *Kernel* removes entry of a file from a directory, *Kernel* writes directory updated contents to the disk synchronously (means no other process can interrupt this action) after removal of that file's entry. This minimizes *File System* corruption if system failure occurs in between. Then it destroys contents of actual file (i.e. freeing file blocks, etc) and frees the *inode*.

But if this process is asynchronous, then it may happen that while directory contents are not yet written to the disk and if inode is freed and if now system crashes, then after recovery it may happen that there is a file in directory with no *inode* (because *inode* is already freed and thus may be reallocated to some other file).
2) Similarly between the two tasks, i.e. freeing disk blocks & freeing the *inode*, *Kernel* first frees the *inode* & then free the disk blocks.

In both above cases if system failure occurs somewhere in between, then ***fsck*** program corrects it.

✶ **Race Conditions For *unlink()* System call ->** Race conditions may occur during *unlink()* system call, particular when unlinking the directories.
1) During the unlinking of directory, *rmdir()* command removes the actual directory. First *rmdir()* sees that the un-linkable (or say removable) directory is empty, means it reads the directory, one entry at a time and looks that all entries have their *inode* values 0 or not. If all entries have their *inode* values 0, then & then only *rmdir()* removes the directory.

But note that, *rmdir()* works at user level and there are two system calls in between. One is *read()* (i.e. reading the directory contents to see whether it is really removable or not) and other is *unlink()* .So *Kernel* can do content switch in between these two system calls an during this context switch, some another process may get scheduled (after *rmdir()* determines that there is no directory entry and hence this directory is removable) which would create a directory entry in the removable directory.

This rare condition can be avoided by 2 things...
a) Using file & record locking
b) During *unlink()*, *inodes* of both, the parent directory and of removable file are kept locked, so that no other process can access both of them.

2) There is another race condition. Suppose process A is parsing a file name and it sleeps in *namei()* and suppose process B is unlinking a file/directory for the same above path and suppose it also sleeps. Now later suppose *Kernel* first schedules B (rather than A) and process B completes its unlinking task.

So when process A gets scheduled, it will continue parsing and will ultimately access such an in-core *inode* which is already removed and thus illegal.

To avoid this race condition, *namei()* always checks the link count of parsed file/directory is not 0, (because 0 link count, means file/dir is removed) and if it is 0, it returns error.

This checking is also not full proof. Because some other process say process C (before scheduling of process A but after completion of process B) may create a new directory in the same path to which the same *inode* (which was freed due to unlinking by B) may get allocated and thus when process A gets scheduled, it will complete its *namei()* as usual but process A will get such a file which is completely un-expected. But system consistency is maintained, though this worst security breach had occurred. This is one of the rare most race condition.

3) It may also happen that one process has opened a file and another process (or even the same process) is trying to unlink it while it is open.

Here *unlink()* will be successful because *inode* of opened file is unlocked at the end of *open()* system call. So *unlink()* can access the *inode* of opened file and will complete the removal of file as if it was not opened. Also unlink will remove directory entry for this file.

But amazing thing is that, at the end of *unlink()* call, the *inode* of removed file is released by *iput()*, but as *open()* system call made reference count of this inode incremented, *iput()* will no free the disk blocks of the removed file and thus process which opened this file can still (though it is removed) do all its "file descriptor" related tasks.

Now when the process which opened the file, now closes it, *Kernel* will now completes the task of *iput()* (from *unlink()*) and closes all file contents because closing of file drops its reference count to 0.

In worst cases, if process which opened the file and the process which unlinked the file are different, and if the process which opened file and then closed the file expects that file is still there. Hence when it (or any other process) tries to reopen it, then surprise!!!, file is gone. Because as soon as *close()* was made, reference count field in its *inode* becomes 0, and as soon as reference count becomes 0, *iput()* (in *unlink()*) completes removal of file's data blocks from the array in *inode*.

✷ ***File System* Abstractions ->** The *mount()* system call can mount multiple *File System*s (including NFS) and even file systems of different operating systems (like DOS, Windows, ISO 9660, etc).

Here a question may arise that ***whether all these different file systems suppose all UNIX file I/O system calls? Whether they also support "inode" feature of UNIX ?***

The answer is that, they may or may not! ***Then how UNIX can recognize them during mount? And how UNIX commands are operable on them?***

This becomes possible due to ***Weinberger***, who introduced ***Network File System*** and also introduced ***File System Abstraction***.

He considers "*inode*" as the major interfere between different file systems & *UNIX File System*. Thus he created so called "***Generic Inode***". This is an in-core type of *inode* which is "***System independent***" and which will point to "system specific *inode*".

So "system specific *inode*" is given by respective system and contains "*File System* specific data" such as access permissions, block layout.

While the "generic *inode*" will contain device number, *inode* number, file type, file size, owner and reference count fields.

Here we also may think that some other OS may not contain "*inode*" structure. In such cases while this OS is going to get support from *UNIX*, the OS designers will give such an object to *UNIX*, to which *UNIX* later can convert into "*File System* specific *inode*", "*File System* specific *Super Block*" and "*File System* specific directory structure". *UNIX Operating System* later on can convert these things into its own specific format and then will allow mounting of such different *Operating System*'s *File System*.

There is yet another issue! Though mounted *File System* is from different OS, *UNIX* can run its usual file I/O routines (such as *open()*, *read()*, *write()*, *close()*) on the files of different OS. ***How?***

For this purpose the OS designers which want to get support from *UNIX*, give a "*File System* structure" which contains addresses of functions which perform their own file I/O.

So when *UNIX* calls its function, in turn this function makes indirect call to above mentioned *File System* specific functions, which actually perform their file I/O.

Means suppose DOS system has *dos_open()* system call to open a file of DOS *File System*. When such a system is mounted in *UNIX*, then call to *UNIX*'s *open()* will indirectly call to *dos_open()* when trying to open DOS's file in *UNIX*.

Thus *Operating System*s which want to get support from *UNIX*, usually give usual file I/O routine addresses to *UNIX* system. These actual file I/O routines are...
1) opening a file 2) closing a file 3) reading from a file 4) writing in a file 5) parsing of file name 6) getting *inode* of a file 7) releasing *inode* of a file 8) access permission related routines 9) mounting & un-mounting, etc.

✴ **Different Situations Of *File System* Inconsistencies ->** The major computer obstacle is "Power Failure" which can create many *File System* inconsistency situations. Some of them are...

1) When *File System* is originally set up at installation, all disk blocks are on *Free List*. When a disk block is assigned for a file, *Kernel* removes it form the *Free List* and assigns its address in *inode*'s "disk block array" field. The kernel can then reassign this same block to another *inode* only when it again appear on the *Free List*. Therefore we can state a rule that, the disk block is either on *Free List* or assigned as a valid file block inside respective file *inodes*' disk block array.
   Now consider that *Kernel* frees a block, obviously this block will appear in "*Free Lit of blocks*" in *Super Block*. The *Kernel* assigns it to a new file. While assigning this block to a new file, *Kernel* wrote *inode* & disk blocks to the disk correctly but power failure (or system crash) occurs before updating the *inode* of old file to the disk.
   Now when rebooting occurs, there are 2 *inodes* (one of old file and other of new file) which contain above disk block's address in their "disk block array".
   This is *File System* inconsistency.
2) Consider the same above situation. Here *Kernel* wrote in-core *Super Block* (having above block as free) and if power failure (or system crash) occurs before updating the old *inode* on disk, then too, same block is going to appear as free (as it is in "*Free List* of blocks" in *Super Block*) and as allocated (as it is in "disk block array" of old file's *inode*.
3) As stated in (1), every block must exist at either of two locations (but never on both at a time as in (1) & (2)) i.e. either "*Free List* of blocks" in *Super Block* or in "disk block array" of some *inode*.
   Now consider that a file is shrunken. So one of its disk block gets freed. Thus this freed block appears in in-core copy of *Super Block*. Now the shrunken file is written to disk. Obviously its *inode* (both in-core copy & disk copy – as file is written on disk) will not contain the free block in its "disk block array". If system gets crashed (power failure), before updating in-core *Super Block* on disk, then the disk block is lost. It will neither be on *Super Block*'s "Free disk block list" nor in file's "disk block array".
   This is also another type of a *File System* inconsistency.
4) All files (except un-named pipe) must exist in the *File System* tree. Obviously all such files must have directory entry and also must have their "link count" non-zero.
   Now consider if a process creates a file (or named pipe) and power failure (or system crash) occurs before creating its directory entry, then *inode* will show non-zero "link count", means file is existing, but there is no directory entry of this file in any of the directory indicating file is not existing.
   This is also a *File System* inconsistency.
5) The same above problem may occur if a directory is unlinked without assuring that it is really empty. In other words, a directory is removed even it contains one or many entries.
6) Sometimes an administrator may mount an improperly formatted *File System*. In such a case *Kernel* may access a disk block thinking that of an *inode*, but in reality the disk block is of some

data rather than an *inode*. In such cases the format of received *inode* will be completely incorrect and *File System* inconsistency will occur.

7) When an *inode* number appears in 16 byte directory entry, then obviously is must be an allocated *inode*. But sometimes (rarely) system may crash after writing entry to the disk but before writing updated *inode* to the disk.
   In such cases directory entry will have *inode* number but *namei()* will fail to get the *inode*, because respective numbered *inode* will not be found on disk.
8) The same above thing as in (7) may occur, if a process unlinked a file and write freed *inode* of that file to disk but system gets crashed before updating directory contents to disk.
9) If the number of free blocks in *Super Block* does not match with the number of free blocks actually on disk, then too *File System* inconsistency occurs.
10) If number of free *inodes* in *Super Block* does not match with the number of free *inodes* actually on disk, then too *File System* inconsistency occurs.

MAINTENANCE -> All of above *File System* inconsistencies can be corrected by

a) Synchronized disk writing operations and most importantly
b) By...***fsck*** routine.