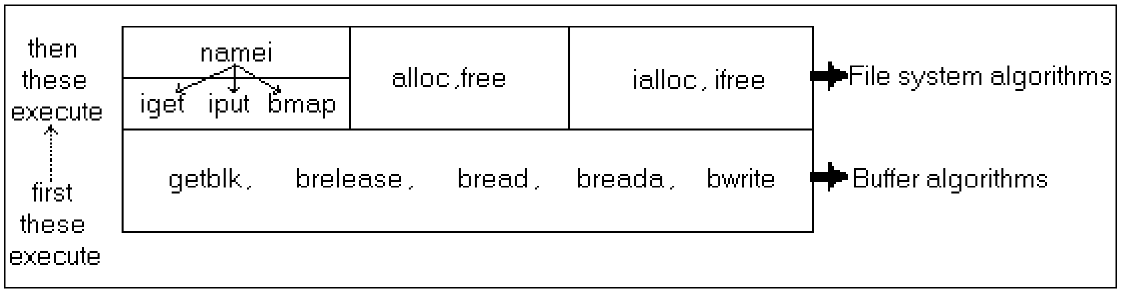
4TH CHAPTER INTERNAL REPRESENTATION OF FILES

As stated before every file on *UNIX* system has a unique *inode*. The *inode* of a file contains information about the file needed for a process to know about that file. This information includes file ownership, access permissions, file size, location of file's data in the *File System* etc.

Process accesses a file by using system calls and it identifies the file with its path as string. As seen on page 7, every file has its own unique absolute path name and it is responsibility of the *Kernel* to convert this unique absolute path name into that respective file's *inode*. ***How Kernel does this?*** will be seen later.

The algorithms in this chapter are on the top of *Buffer Cache* algorithms. Means *Buffer Cache* algorithms first get executed and then *File System* algorithms begin to execute.



Algorithms ***namei*** converts file's path into *inode*. While doing this, ***namei*** uses ***iget***, ***iput*** & ***bmap***. Here ***iget*** gives *inode*, ***iput*** releases it and ***bmap*** sets *Kernel* parameters to access the specific file whose *inode* is now in use.

Algorithms ***alloc*** & ***free*** allocate & free disk blocks for file data and algorithms ***ialloc*** & ***ifree*** assign & free *inodes* for a file.

✴ A file if executable can be executed for a class by giving "x" permission to it. But directory file is not an executable file. So giving execute permission to a directory file means allow the class to search that directory file.

✴ Inode :

Inode is a *Kernel*'s unit by which *Kernel* identifies a file. For every file there is a unique *inode*. So obviously number of file is equal to the number of *inodes* & vice versa.

Inodes are of 2 types:

1) When system opens or creates a file it also creates its *inode* and kept on the disk in "*Inode list block*" shown and explained on page 19 (2nd Chapter). This is a static form of *inode* and as it is kept on the disk it is called as ***Disk Inodes***.
2) When a file is opened or created to make operations faster, *Kernel* pulls its *inode* in "*inode list* block", into RAM. In RAM it keeps *inode* in list form. As *inodes* of this type is in memory, it is called as Dynamic and called as ***Memory Inode*** or ***In-core Inode***.

✴ Then "***Disk Inode***" is used to store the information about file when the file is not in use.
✴ While the "***In-core Inode***" is used to store information about file, when file is in use (i.e. active).

✴ Structure Of Disk *Inode* :

As a data structure, the *disk inode* consists of following fields..... (about 10 fields)

! **File Ownership Identifier**
! **File Group Identifier**
! **File Type**: - i.e. regular file or directory file or device file or block special file, character special file or pipe. If *inode* is ***free***, this field is set to 0.

! **File Access Permissions**: System protects file according to 3 classes – "Owner", "Group Owner" and "Other" (UGO). The permissions are also 3 – "r" for "read", "w" for "write" and "x" for "execute". One of these 3 permissions is given to each of above 3 classes.

! **File Access Time**: This is the time when the file was last accessed.

! **File Modification Time**: This is the time when the file was last modified.

! **File *Inode* Modification Time**: This is the time when the file's *inode* was last modified.

! **Number Of Links To The File**: This is the number of aliases of the file by which the file can identified (except its actual name).

! **File Size**: Size of the file in terms of bytes. The size is calculated from its beginning (0 offset) up to the highest ***offset + 1***. The last "one" indicates end of file. This is 32 bit integer. (Note that: Size of file is not "number of bytes" inside it. But it is the highest byte ***offset + 1***).

! **Table Of Disk Addresses:** System every time does not essentially has to write all data in a file "contiguously". It may write the data anywhere on the disk. So while reading a file from the disk, if one part of file is read contiguously from the file and next part not belongs to this file, then it is necessary that *Kernel* should know from which point on disk the next data of read file begins. So system keeps addresses of all disk blocks belonging to respective file in this table (i.e. array).

Example: - Suppose there is a file temp.txt. The owner of this file is say ABC, Group owner is XYZ then the *Disk Inode* of this file will have structure like follows….

```
Owner ABC
groups XYZ
type regular
permissions rrwx – rx – x
access time Oct 12 1998 1:33 A.M.
modification time Oct 10 1998 12:24 P.M.
inode modification time Oct 12 1998 1:33 A.M.
links 3
size 6030 bytes
disk address [ ]
```

✓ In above figure, first "r" in permissions field stands for type of file i.e. "regular" file. Not for "read". Next 3 permissions "rwx" are for user (owner), which represent all access i.e. to read or to write or to execute for user or owner. Next 2 permissions "rx" are for "Group", means group owner has access to this file for "reading" & "executing" only but not for "writing" as there is no "w". The last permission "x" is for "other" class, which has permission only to execute this file means neither to read nor to write as there are no "rw".

✓ The "links" fields shows number 3, means temp.txt file has right now 3 links in the system say *abctemp*, *xyztemp* & *othertemp*. (links are like shortcuts in windows environment. Where suppose windows directory contains *explorer.exe* and we create "File Manager" shortcut on desktop for it. Then "File Manager" is said to be link of (or alias of) *explorer.exe*. Double clicking both of them will execute the same *explorer.exe* file)

✓ It is **surprising** to note that, though *inode* contains so much information about the file, it does not contain path or name of file.

✓ The File Modification Time & The *Inode* Modification Time are two different times and their distinction is important. The contents of a file change only when its contents are modified and then saved. As modification in file data, changes "disk address table" field in *inode*, the *inode* also gets automatically modified. This means that file modification time, always.
In contrast, *inode* contents of a file get changed whenever owner is changed or group owner is changed or file permissions are changed or number of links are reduced or increased, or disk address table is changed (i.e. old data from file is deleted or new data is added), *inode* also

changes. This means that *inode* modification does not always reflect on file modification time. Means if owner or group or permissions or links are changed, *inode* modification time will change but file modification time remains unchanged because no file modification is done in above cases.

✷ **Structure Of Memory *Inode* –** Memory *inode* is also called as "In–core *inode*". Its fields are...

! **All the fields of Disk *inode*.**

! **Status field –** Contains following status:
1) Locked / unlocked
2) Reserved / not reserved
3) *inode* changed / not changed
4) File fields changed / not changed
5) Whether file is a mount point or not
: -Indicating whether *inode* is locked or not, whether a process is waiting on *inode* to get unlocked, whether In–core *inode* is changed due to change in *Disk inode*, whether file fields in In–core *inode* copy is changed due to change in file fields of *Disk inode*, whether the file (whose *inode* we are looking at) is a mount point or not.

! **The logical device number** of the *File System* to which this file (whose *inode*, we are looking at) belongs.

! **The *inode* number –** Disk *inodes* are stored in an array, though in figure on page 19 (2nd Chapter) says it is "list". *Kernel* identifies the "disk inode" by its index position in this array. This index is stored as a number of *inode* in "In-core" *inode*. As this facility is for *Kernel*, only In–core *inode* contains this field. *Disk inode* need not contain this field. (Note : *Inode* numbers start from 1 and not from 0.

! **Pointers to other In–core *inodes* –** Just like *Buffer Cache*, *inodes* are also kept on "*Free List* of inodes" as well as on "*Hash Queue* of *inodes*". The *Hash Queue* of *inodes* is identified by the logical device number & *inode* number combination (hash function). As like a buffer, *Kernel* can contain only one copy of a In–core *inode* for a *Disk inode*. But *inodes* can be simultaneously present on both the "*Free List* of *inodes*" & on "*Hash Queue* of inodes". (***Free List of inodes* + *Hash Queue of inodes* = *inode pool***).

! **A reference count number –** This indicates number of instances of the file that are active.

As seen above, many fields of in–core *inode* are similar to buffer header (explained in last chapter). Even the management of buffer pool & *inode* pool is also much similar. The *inode* lock when set to locked condition, other process cannot access it (just like buffer). Also other processes can set the status flag of In-core *inode* to indicate the *Kernel* that they should be awakened when the lock is released. (This is also same as buffer mechanism). After making desired changes to In-core *inode*, *Kernel* write these changes to the disk in corresponding *disk inode*.

The most important difference between the structure of In–core *inode* & the structure of buffer header is the field of "Reference count number". Buffer header is does not have this field. This number, as explained on last page, indicates the number of active instances of the file whose *inode* this is. When a process opens a file, it is first instance of the file and hence reference count field becomes 1. When another process opens the same file, then it is instance 2 and thus reference count becomes 2 and so on.

An in-core *inode* can be present on "*Free List* of *inode*" only if reference count field of it is 0. (Just like buffer. Buffer can be present on *Free List* if and only if it is free or made for delayed write). The reference count 0 indicates that *Kernel* can re-allocate this *inode* to another *disk inode* if required. Thus the *Free List* of *inodes* works as a pool of inactive *inodes* (as their reference count is 0, there is no instance of file corresponding to respective *inode*). So if a process attempts to open a file, and if its *inode* is not present in the *inode* pool (i.e. *Inode Free List* + *Hash Queue* of *Inodes*), then *Kernel* takes a different *inode* (whose reference count is 0) from the *inode Free List* and re-allocates it to the file for which process is trying to access. This was not the case with buffer. A buffer can be present on *Free List* of buffers if and only if it is un-locked (i.e. free).

✷ **Accessing The *Inodes*:** As *Kernel* identifies a buffer by hash function of *device number – block number combination*, *Kernel* identifies particular *inode* by "*File System* number - *inode* number" combination.

As buffer mechanism has *getblk()* algorithm, the *inode* mechanism has *iget()* algorithm. So *getblk()* & *iget()* algorithms are much similar. The algorithm is as follows…

```
iget algorithm
input :  1) File System number
         2) inode number
{
   while ( not done) // i.e. inode not found
   {
         if (inode is in inode pool)
         {
               if (inode is locked)
               {
                     sleep (event : until the inode gets unlocked) ;
                     continue ; /* back to while loop */
               }
               /* here should be special processing for mount point */
               if (inode is on Free List)
                     remove from Free List ;
               increment inode reference count ;
               return (inode) ;
         }
         /* inode is not in inode pool */
         if (inode Free List is empty)
               return (error) ;
         remove new inode from the Free List ;
         reset its present File System number & inode number to that of our required inode's value ;
         remove inode from old Hash Queue ;
         then place it on new Hash Queue ;
         read inode from disk (by bread() algorithm) ;
         initiate this inode (i.e. increment reference count from 0 to 1) ;
         return (inode) ;
   } /* end of while loop */
} /* end of algorithm */
Output : locked inode
```

✷ When a process tries to access a file (say tries to open a file), *Kernel* looks for that file's *inode* in *inode* pool. While searching *Kernel* uses "hash function" of *File System* number of *inode – inode* number combination. *Kernel* first searches for that particular file's particular *inode* in *Hash Queue*, if it finds it here, it removes the same and locks it. If *Kernel* cannot find particular *inode* in *Hash Queue*, he leaves the *Hash Queue* and go to *Free List*. From *Free List* it pulls out one *inode* (which is not particular one) from the *Free List*. Then it actually reads particular *inode*' s *disk inode* and makes relevant changes in the pulled *inode*. Then from *File System* number + *inode* number, if finds out the logical disk block (i.e. disk addresses field of *inode* structure) of the file (whose *inode* this is) and accesses the file on behalf of the process.

The formula by which it finds the logical disk block number is....

block num=((*inode* num -1)/number of *inodes* per block)+start block of *inode list*.

In above formula as all are integer terms, the division also gives an integer value.

Suppose *Kernel* wants 8th *inode* (i.e. *inode* number is 8), & suppose there are 8 *inodes* per block and also suppose that the starting address of *inode list* is 2, then formulae works as...

block num = ((8 – 1) / 8) + 2
= (7 / 8) + 2
= 0 + 2
= 2

So the required 8th *inode* is in block 2. Means logical block number of 8th *inode* is 2.

Similarly if we want the block number of 9th *inode*, then above assumption & formulae gives block num 3.

If suppose there are 16 *inodes* per block and we want block number of 17th *inode* then... (Assuming starting address of *inode list* is same as above i.e. 2)

block num = ((17 -1) / 16) + 2
= (16/16) + 2
= 1 + 2
= 3

So 17th *inode* is in 3rd block

After getting this block number, now *Kernel* has device number (i.e. *File System* number) and block number. So combination can be used. Thus *Kernel* now uses ***bread*** algorithm (as on page 50, 3rd Chapter), whose input is this block number. It allows the algorithm and returns buffer for it.

We already assumed that a process is going to read file. Owing to this now we got the file's *inode*, also we got the buffer to hold file's data. But where is the data? (Because buffer returned by *getblk()* in *bread()* is yet empty and our duty is to fill it with the required file's data). So we need an address from which file's data begins. Here comes the last field of *inode* structure, which is nothing but the array of disk addresses. So *Kernel* uses following formulae to access the file *inode*'s (In-core copy) last field i.e. the disk address from which file's data begins.

Disk address = ((*inode* -1) modulo (number of *inodes* per block)) × size of *disk inode*
(or byte offset)

So, if suppose size of *disk inode* is 64 bytes (which can be obtained by C's sizeof (In-core *inode*) operator) [size of *disk inode* = sizeof (in-core *inode*) – sizeof (5 extra fields of in-core *inode*) ;] , also suppose that there are 8 *inodes* per disk block, then the disk address (or byte offset) of file's data is ....

Byte offset = ((8 – 1) % 8) × 64
= (7 % 8) × 64
= 7 × 64
= 448

So file data begins at address 448. This is a global data item for *Kernel*. Thus *Kernel*, in *bread()*, first obtains empty buffer for respective block number, then uses above byte offset (e.g. 448 in above example), starts disk head operations and reads data from the address to the end of file character. The read data is filled in the buffer and then returned by *bread()* as a valid buffer containing valid data. (see the output of bread on page 50).

Now coming back to *inode*, *Kernel* removes respective *disk inode*'s In-core *inode* from the *Free List*, then places it on correct *Hash Queue*, files its contents by file type, owner, group, permission, timings, disk addresses, etc and most importantly makes its reference count 1 (which was 0 initially) and returns the locked *inode* from *iget()* algorithm (see the output of iget() on page 58).

The *Kernel* manipulates *inode* locking mechanism & reference count mechanism independently (means these 2 process do not dependant on each other). The lock is set to "locked" condition during a system call, so that other processes get prevented from accessing or changing the *inode* while it is use. The *Kernel* releases this lock only when the system call finishes. Thus an *inode* is never locked between 2 system calls. It is always locked when system call is made and released when system call finishes.

On the contrary, *Kernel* increments the reference count field of in-core *inode* whenever access to the file (of that *inode*) is made. Means suppose at beginning the count is 0, one process opens the file, the count becomes 1, if another process opens the same file, the count becomes 2 and so on. Obviously the count will be decremented whenever process closes a file. So for above example, if second process closes the file, so the count becomes 1 (which was 2 before) and when first process closes the file, the count again becomes 0. This shows that, unlike lock, reference count can be set (i.e. incremented or decremented) across the 2 system calls. Though "lock" cannot be set across system calls, it can be released (i.e. unlocked) across the system calls to give facility to other process to access the file (i.e. sharing same file) for reading. But reference count is never decremented across system calls; reference count remains "set" across system calls to prevent other processes to access the *inode*.

Above two paragraphs shows that *Kernel* independently handle "lock mechanism" & "reference count mechanism" independently.

Up till now we know that, if everything goes smoothly, the *iget()* algorithm returns locked *inode*. Now we will discuss some error situations in *iget()* algorithm.

If *Kernel* tries to get an *inode* from *Free List* and if the *Free List* of empty, then *Kernel* gives an error. This is a sharp contrast with that of buffer mechanism. In buffer mechanism, in *getblk()* algorithm, if *Kernel* tries to get a buffer from *Free List* and if the list is empty, then the *Kernel* does not flag error, instead the process which tries to get the buffer goes to sleep. (see *getblk()* on page 43, 3rd chapter & *iget()* on page 56-57). Why the "empty *Free List*" situation for buffer & for *inode* react differently? As seen before a process cannot keep the buffer "locked" across the system calls. So when a process sleeps waiting for a buffer to become free, *Kernel* can guarantee that slept process will get buffer at some time. This is not the case for *inode*. Here process has control over *inode* on "user level" (for buffer the process control is at "*Kernel* level") and hence when a process goes to sleep waiting for an *inode* to become free, it may never wake-up. So instead of facing such "hanging" condition, *Kernel* gives an error. In short, *Kernel* can guarantee a process for free buffer hence process can sleep for waiting until buffer gets free. But for *inode Kernel* cannot guarantee a process for free *inode* (because closing a file closes the *inode* and user may not close a file for very long time) and hence keeping a process in "sleep" condition for free *inode* may "hang" the process permanently. So it is better to flag an error when a process needs a particular inode to become free when some other process already locks it.

Above situation will occur only when the "free *inode list*" is empty. Means when the allocated *inode* is not in the *inode* table.

In other situation, means when *inode* is in *inode* cache, process A wants to access its file, and hence *Kernel* found its *inode* in *Hash Queue*. But now if the *inode* is already locked by process B, then process A sleeps, but before sleeping process A sets a flag in *inode* (i.e. In-core copy of same required *inode*) to indicate the *Kernel* that when respective *inode* becomes free (i.e. unlocked), then it should be awakened. Thus when process B unlocks the *inode*, all processes, including process A, are awakened. If process A (within all awakened processes) has first priority then it gets the *inode* and immediately locks it, so other processes cannot allocate it. If process A is not at high priority, then some other process accesses the *inode* and process A again goes to sleep. But a time comes when process A surely gets the required *inode*.

When process A gets the *inode* and if its reference count is previously 0, then obviously it is also present on *Free List*. So *Kernel* removes it from the *Free List* and keeps it on correct *Hash Queue*. Now *inode* is not free. *Kernel* increment its reference count and returns locked *inode*.

So this is all about *iget()* algorithm.

✷ **Releasing The *Inodes* : -** As buffer has its releasing algorithm *brelease()*, the *inode* also has its releasing algorithm called as *iput()*. The *Kernel* uses this algorithm to release a locked *inode*.

```
iput() algorithm
input : pointer to an In-core inode
{
    lock the inode if it is not already locked ;
    decrement inode's reference count ;
    if ( reference count == 0 )
    {
        if (inode link count == 0)
        {
            free disk blocks of respective files ;
            set file type to 0 ;
            ifree ( inode) ;
        }
        if ( file is accessed OR inode is changed OR file is changed )
            update disk inode accordingly ;
        put inode on "inode Free List" ;
    }
    release inode lock ;
}
output : none.
```

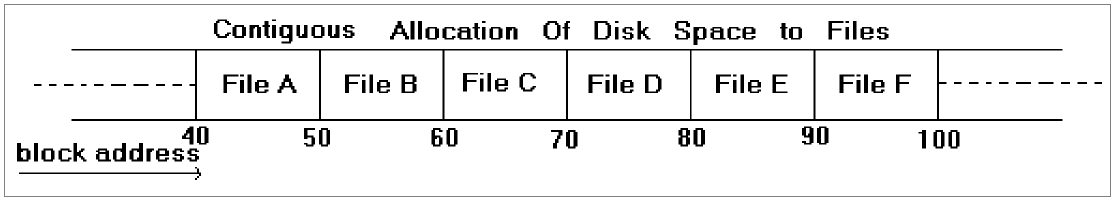
- → This algorithm gets the In-core *inode*'s pointer (memory copy of a *disk inode* of a file) as its only input.
- → It then checks whether *inode* is locked or not. If *inode* is not locked, then locks it.
- → Then it decrements the reference count by 1.
- → Then it looks, whether above decrementing makes reference count 0 or not. If reference count is 0, and if....
- → *Inode* link count (means "link" field in *inode* structure) is also 0, then it frees the disk blocks (i.e. disk addresses in *inode* structure) of respective file by algorithm *free()* - see figure on page 53.
- → Then sets the "file type" field in *inode* structure to 0 (because now *inode* is not pointing to a particular file as its disk address array is freed and as its link count & reference count, both are 0).
- → And frees the *inode* by *ifree()* algorithm.
- → Some times it may happen that the file (to which this *inode* points to) is accessed OR file contents are changed, change in *inode* means, change either in times, owner, group, access permissions, etc. OR *inode* itself is changed, then *Kernel* writes respective changes to respective *disk inode*. (i.e. updates the respective *disk inode*).
- → By doing all above things now *inode* is free, means not pointing to particular file, hence instead of putting it on *inode Hash Queue*, *Kernel* now puts it on "*inode Free List*"
- → Ultimately releases the lock, which was made in first step. This releasing of lock involves 2 situations. As this step is written outside all "if" blocks, it is common to all. Means releasing of lock is going to take place whether the reference count is 0 or not.

✷ **Structure Of Regular File:** As we discussed before, an *inode* contains "disk address" field. But until now we didn't discuss this field in detail. Note that, this is the only field by which *Kernel* can access the whole file respective to that particular *inode*.

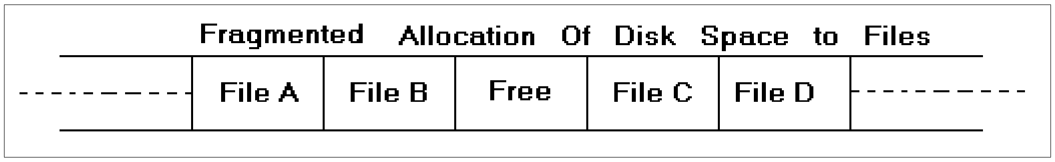
When *Operating System* is installed on disk, many files are installed on the disk. Yet after user can add different software or user itself can create or remove custom files from the disk. So disk space is filled with many files.

Now storing files on disk faces 2 problems. :

1) If *Operating System* decides the file size, and all files must be equal or less than this predefined size then the good news is that, the address field in *inode* structure can contain just the starting block address number and accessing file will become very easy, because *Kernel* knows that file size is not going to be larger than starting block address number + max file size (predefined). Also there is another advantage – fixing the file size will allow files to occupy the disk in "contiguous manner" and hence there will be no disk space fragmentation.
   But this scenario makes *Operating System* less ambitious & constraint in 4 situations ...
   i) If user wants to create or install a file larger than the pre-defined fixed file size, then such a file cannot be installed.
   ii) If a file on the disk is just less than the pre-defined max file size, then well, it will fit into the *File System*, but it user, later wants to add some data to this file by editing it, and if adding data to the file is going to increase the file size than the pre-defined max file size, then system will not allow the user to do this and user will annoyingly avoid to use such constraint system.
   iii) To overcome this problem in (ii), if system tries to make pre-defined fixed file size large enough to satisfy the user, then a new problem will arise. If a file is too small than that of the pre-defined fixed file size, then it is wasting of disk space for such a small sized file.



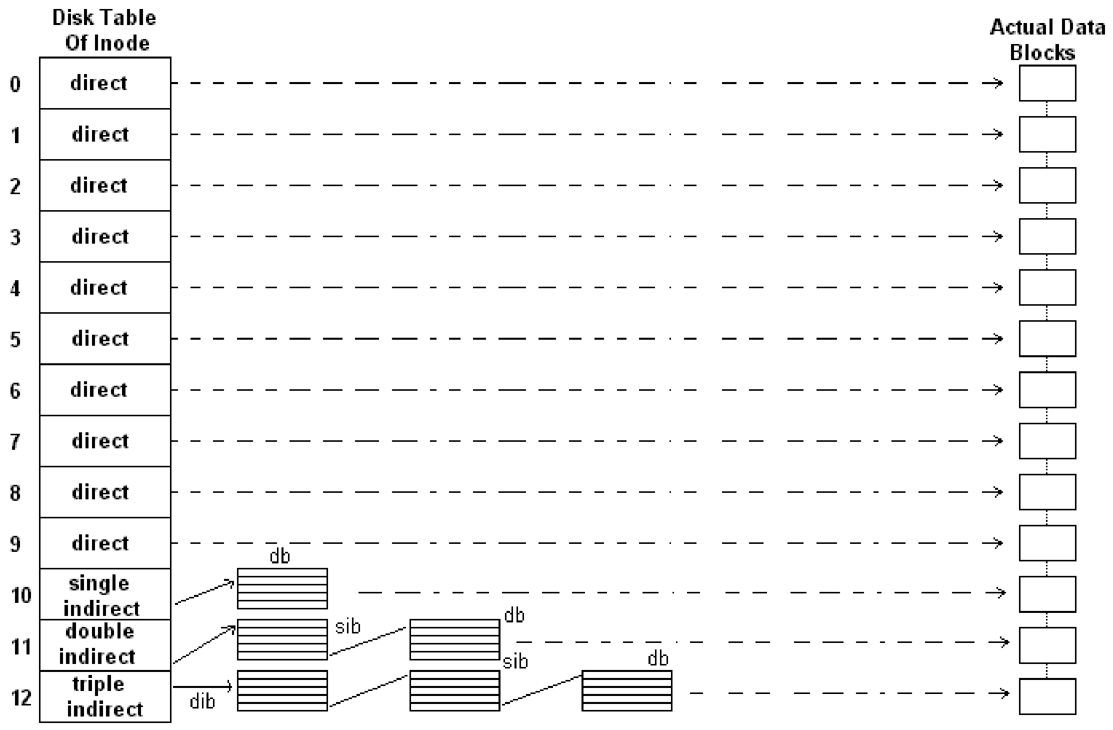
   iv) If system avoids to make such a pre-defined max file size and considers only to keep files in contiguous manner, though files are of any size, then well all above 3 constraints are avoided. But here *Kernel* have to watch expansion & reduction of every file for every time and also do necessary dynamic expansion of a file slot when user wants to add more data to the file. This also involved <u>pushing</u> of all other "next" files <u>away</u> until user's wish to add the data fulfills. Similarly when user wants to delete data portion from the file or wants to delete the complete file, then too, *Kernel* has to do a lot of dynamic work of shrinking file slot as well as <u>pulling</u> the "next" files <u>more closer</u> to the file's slot. All this overload on *Kernel* for single file reduces the system performance to worst level.
2) So contiguous file allocation, though easier to implement, has lot of constraints. Thus system uses another scenario for this purpose. Here *Kernel* allocates <u>1 block at a time</u> for a file and allows remaining data to spread throughout the *File System* wherever free space is seen.



   So now if file A is going to be edited for adding data in it, then the <u>free</u> space after file B, can be allocated for file A again. So a single file's data may be spanned throughout the disk or partition. This scenario is not also full proof, but is <u>more flexible</u> than "contiguous allocation" scenario. This makes file accessing more complicated and has following consequences...
   a) One file's data cannot be accessed by addressing a single location of starting block address of the file. Because file's data is fanned anywhere on the disk. Obviously the "disk address" field of *inode* structure is now <u>array</u> (or table) each element if which is a disk address of file's data. So whole array is full of disk addresses of those blocks, which have respective file's fragmented data.

b) Upon deletion of one or more files, the disk space gets much fragmented. To overcome this, a system may choose <u>either</u> encouraging user to do periodic de-fragmentation of the disk (i.e. Win9x comes with a “defragmenter” tool) <u>or</u> system itself does this periodic de-fragmentation by running garbage collection procedures.

c) We said in (a), that the system will keep all data block addresses in an array of disk addresses in *inode*. But conceptually array has a fixed size limit, and it will not know <u>in advance</u>, how may disk block addresses are to be stored for respective file data. To overcome this problem, *UNIX* system designers keep <u>the array size limited to 13</u>, but used it in such a way that a very large (even gigabyte) sized file’s data also can be accessed. ***How?*** : See the figure on next page. The array of “Disk Addresses contains 13 elements. First 10 elements (i.e. 0 to 9) are called as “Direct Blocks”. Means they contain block addresses of actual data in a file. Thus the relation between a “direct block” & “data block” is shown with a continuous arrow.
Figure:



[✶ Simply speaking first 10 blocks are for 1 kb. 11th block contains address of such a block which contains 256 1 kb blocks. 12th block contains address of such a block which has 256 blocks (of 256 containing 1 kb block) and so on.]

The 11th element is a “Single Indirect Block”. This does not contain data block address directly. Instead it has address of such a block which in turn has addresses of actual disk block.

The 12th element is a “Double Indirect Block”. Similar to above, this also does not contain address of actual data block. Instead it contains address of such a block, which in turn contains address of “Single Indirect Block”, each of which contain addresses of actual data blocks.

The 13th element is a “Triple Indirect Block”. Similar to above this also does not contain address of actual data block. Instead it contains address of such a block, which it turn contains addresses of “double indirect block”, each of which contain address of such a block, which in turn contains addresses of “single indirect block”, each of which contains address of such a block, which is actually a data block.

Above discussion will be cleared by one example, suppose in our *File System* 1 logical block is capable of holding 1KB, and our machine supports 32 bit integer (i.e. *int* has 4 bytes). So obviously 1 block can have addresses up to 256. (***how?*** 1KB = 1024 bytes and 1024/4 = 256).

Now we will take a file (of some obituary size, say 10 GB). The *inode*'s, "disk block table field" will have first 10 elements as direct blocks. As one block is assumed of 1KB, these 10 direct blocks will cover 10 KB data of file. Now system assigns a special block and keeps 256 addresses of file's data blocks in it. This special block is called as "Single Indirect Block". The address of this special block is now kept as 11th element of *inode*'s "disk block table". So now this 11th element (11th element is called as "single indirect block") alone can cover 256 KB. So up till now file's 10KB + 256KB = 266KB data is taken care off. Then system assigns another special block and keeps 256 addresses of "single indirect blocks" and the address of this special block is kept in 12th element of *inode*'s disk block table. So now file's
(1 KB * 10 blocks (which are first 10 blocks)) + (1 KB * 256 blocks) + (1 KB * 256 * 256) = 65802 KB data is taken care off. This 12th element is called as "Double Indirect Block". Then system assigns another special block and keeps 256 addresses of "double indirect blocks" in it. The address of this block is then kept in 13th (last) element of *inode*'s disk block table. This 13th element is now called as "Triple Indirect Block". This element can take care of (1KB * 10) + (1KB * 256) + (1KB * 256 * 256) + (1KB * 256 * 256 * 256) = 16843018 KB which is 16 GB round about. (Because 16843018 KB is round about 16448 MB which is round about 16 GB).

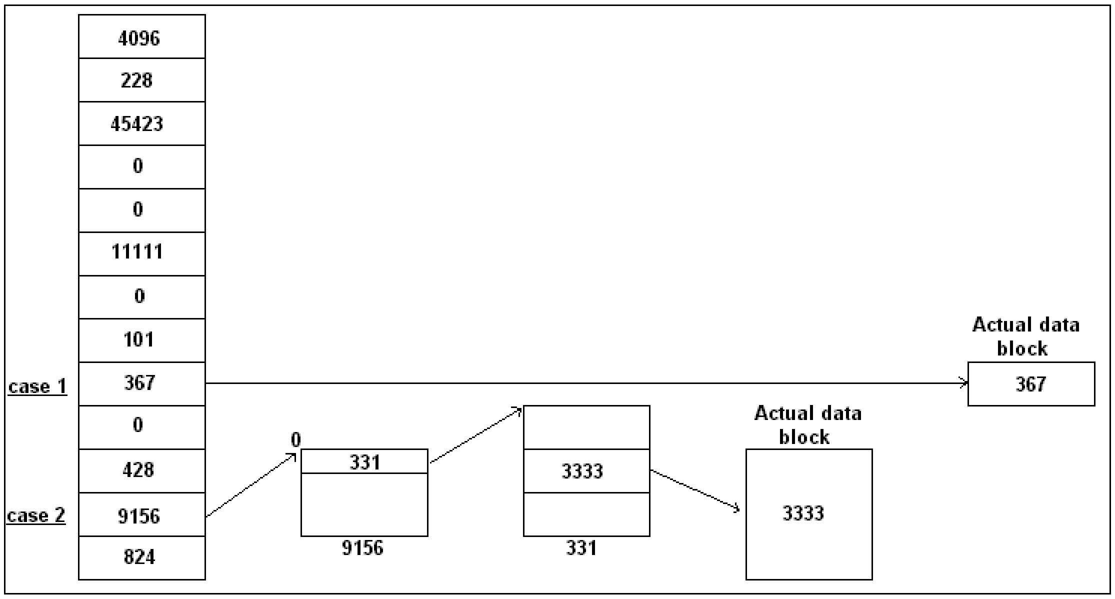
10	direct blocks with 1KB each	=	10 KB	-
1	single indirect block	=	256 KB	-
1	double indirect block	=	65536 KB	64 MB
1	triple indirect block	=	16772116 KB	16 GB

(Here it is assumed that 1 logical block is made up 1 KB)

Above method can be extended further for "Quadruple Indirect Block" or "Quintuple Indirect Block", but the current structure up to "Triple Indirect Block" is more ....more than sufficient.

So can we say that *UNIX Operating System* can access 16 GB file? The answer is **No**. Because *UNIX* is capable of doing so, but the *inode* structure specifies file size (page 54 & 55) as ***int*** type. If this is a 16 bit integer, then this size can be maximum up to 2^{16} bytes means 2147483648 bytes, i.e. 2097152 KB, i.e. 2048 MB, i.e. 2 GB. But if this is a 32 bit integer, then this size can be maximum up to 2^{32} bytes means 4294967296 bytes, i.e. 4194304 KB, i.e. 4096 MB, i.e. 4 GB. So *UNIX File System* (on 32 bit processor with 32 bit integer size) can access a file with maximum upto 4 GB size.

To understand the concept, we will loot at one or two examples.



Consider that for a file, its *inode* structure is as in above figure. Assume that 1 block is of 1 KB i.e. 1024 bytes.

Case 1 ^ Suppose we want to access file's data at position 9000 byte offset. The system uses ***bmap*** algorithm to find the block number to search for. So first, it is easy that 9000 byte offset will fall in 8th block (by 0 index, 8th block), because 1 block is 1 KB means 1024 bytes , obviously 9000 will fall in 8th block. In 8th block the address is 367, so system directly takes this number 367 and as it is of "direct block type", there is no further indirection, and hence system directly takes 367th block from buffer pool and proceeds for reading or writing of it.

Case 2 ^ Now suppose we want to access file's at position 350,000 byte offset. By ***bmap*** algorithm system comes to know that, this offset will require "double indirection block" i.e. 11th block in *inode*'s disk table. ***How?*** 350,000 bytes means round about 3401 KB. Obviously direct blocks cover up to 19 KB, single indirect block covers up to 256 KB and double indirect block covers up to 65 MB. So the required block 3401 KB (i.e. round about 3 MB), is concerned with 11th block, which is a double indirect block. Now system looks in 11th block, there it finds address 9156. Then it searches for 9156th block. Now this 9156th block is itself a double indirect block. In this it gets 331 in its 6th block, so system searches for block no 331. But 331 is itself a single indirect block, where it searches for 3333 block number. 3333 is a direct block where it gets the required byte offset address. This description is according to the figure.

✷ How system does above calculations?

I. System starts looking in disk block array of *inode* of the respective file. 1 direct block is 1kb (i.e.1024 bytes), hence 10 blocks will be able to access 1040*10=10240 bytes. Our requirement is 3,50,000th bytes, so obviously we should go further.

II. The 11th block is a single indirect block. Means it can access 256 blocks of 1k, means it will be able to access 2, 62,144 bytes of file. Adding initial 10,240 to 2, 62,144 we get 272384. But still our requirement is 3,50,000th byte, which is greater than above; hence, we should go further.

✷ Algorithm *bmap()*:- Process accesses data in a file by using byte offset, starting at 0 up to the last byte offset (which depends upon the file size). But *Kernel* converts this "byte offset view of a file" into "block wise view of a file". So according to *Kernel*, file starts at logical block 0 to the last logical block of file (obviously this depends upon the file size). For this *Kernel* accesses *inode* of particular file and using it, *Kernel* converts logical file block into appropriate physical disk block.

The algorithm *bmap()* is used by *Kernel* to convert the byte offset of a file into the physical disk block.

```
algorithm : bmap /* block map of logical file byte offset to the system block */
input :  1) inode of the file
         2) byte offset of the file
{
                 calculate logical block number corresponding to the byte offset ;
output 2 →       calculate the starting byte of above block ; /* used for I/O */
output 3 →       calculate number of bytes to copy to user ; /* from above 2 */
output 4 →       check whether "read-ahead" can be applicable or not and mark the inode /* */
                 determination level of indirection ;
                 while ( still not exactly at the proper level of indirection)
                 {
                          calculate index into inode or indirect block number from logical block number in
                            file ;
                          get disk block number from above inode or from above indirect block number ;
```

```
                        release buffer from previous disk read if any ; /* algorithm brelease */
                        if (no more level of indirect remain)
output 1 →                  return (block number) ;
                        read indirect disk block ; /* algorithm bread */
                        adjust logical block number in file according to level of indirection ;
                  }
        }
output :          1) physical disk block number corresponding to file
                  2) byte offset into block
                  3) bytes of I/O in block
                  4) read-ahead block number
```

- ✓ Algorithm *bmap()* has 2 inputs, one the *inode* of the file (received by using *iget()*) and second byte offset in the file to which user wants to look (set by read or seek operation by the process of user).
- ✓ Then system calculates logical block number of the file's current byte offset. See **case 1** on page 63. Means suppose user wants to read file's data form byte offset 9000. In the disk address array of 13 elements, this byte offset will found at index 8 (starting from 0) (because 1 array element size = 1024 bytes). According to figure on page 63, the 8th element has address number 367. So this is the logical disk block number.
- ✓ Now system calculates starting byte of "above found" block. This is easy, as each block is of 1024 bytes, 8th block will start at 1024*8 i.e. 8192. So the starting byte will be 8192.
- ✓ Then system calculates how many bytes user wanted to see from this file. Here 2 cases may appear. :

  Case A ^ User may want to see file data, which is less than the block size. Means 8th block begins at 8192 (i.e. 1024*8) and this block extends up to 9216 (i.e. 1024*9). So suppose the user need is to see 200 characters (1 character = 1byte) from 9000th byte, then 9000+200 gives 9200, which is less than 9216, hence well, the "wishful" data is within the limit of this block. So 200 bytes will be copied to the user starting from 9000 to 9200 (or from 9000 to block end means to 9216).

  Case B ^ suppose user wants to read full-screen data starting from 9000. Now "full-screen" approx means 80 characters horizontally & 20 lines vertically. So user wants to read 1600 characters, i.e. 1600 bytes. Now adding 1600 bytes to 9000 gives 10600, which is beyond 9216, means next block (i.e. 9th) is also involved. Here "read-ahead" algorithm may require anticipating that user may want to read next block from 10600 to its end (means up to 10240).
- ✓ By above way of case B, system also anticipates, whether "read-ahead" is going to be required or not. Accordingly it marks the *inode*.
- ✓ For above example, we don't need further indirect blocks because our task gets completed in "direct blocks" only. But if we refer to **case 2** page 64, where we want to read data at 350,000-byte offset, then we need indirect blocks and hence level of indirection is calculated as explained on pages 64 to 66.
- ✓ If presently found block and the level of indirection do not match, then algorithm enters in "while loop".
- ✓ In while loop, first it calculates index into the *inode*'s direct block or indirect block from logical block number found (before entering into the while loop). So this logical block number becomes starting point.
- ✓ Then it gets corresponding physical disk block number (from either of above 2 conditions).
- ✓ If any pending read operation is going on, then it releases the buffer of this pending read operation, by using *brelease()* algorithm.
- ✓ If no more levels of indirection are necessary, then it returns the found block number. (If level ends at direct block level).

- ✓ Else it reads indirect disk block by *bread()*, if necessary. (If level requires indirect level).
- ✓ Adjust logical block number accordingly and goes back to loop.

✷ This algorithm gives 4 outputs : ....
1) The block number (i.e. physical block number).
2) Byte offset into the block.
3) Bytes of I/O in block.
4) Read-ahead block number if required.

✷ Out of 4 outputs, one is usual return value (obviously local variable) but remaining outputs are obviously global.

III. The 12th block is a double indirect block, which contains 256 single indirect block and single indirect block (each) can access 256 blocks of 1 KB.
So out of 256 single indirect blocks, system looks in 0th block, in 0th block it gets address of single indirect block. Out of 256 single indirect blocks 74 (i.e. 1024*75=76800 bytes). We took 75 because block are counted from 0 to block 74 is total 75 blocks. Now if we add 76800 to previous 272384, we get 3,49,184. This shows that next block i.e. 75th (means total 76th) block will have our required byte 350000. We can confirm it by adding one more 1024 to the 3,49,184, which gives 3,50,208, which is greater than our required. Means 74th element end at 3,49,184 and 75th element begin by 3,49,185 and end at 3,50,208. Within this range our required byte of 350000 resides.

This is the reason that element 0 of double indirect block and element 75 of single indirect block are chosen. Now at this 75th location the stored address is 3333, which is the address of direct block, which actually is going to have the file's data at our required 350000th position.

This discussion shows that addresses kept at elements are not important (because these addresses can be any number assigned by system), but the most important is the location of element in array. E.g. 0th in double indirect and 75th in single indirect.

Now in direct block 3333, where the byte offset 350000 is located? The answer is easy; just subtract 350000 and the starting byte offset off 3333, which is 3,49,184. We get answer 815. Counting from 0, we can say that required 3,50,000 byte offset is located at byte 816 in direct block 3333.

If we carefully examine the figure on page 63, some blocks shows their entries as **0**. This means that these logical block entries contain "no data". This means that when process wrote this file, the byte offsets corresponding to those blocks, never wrote with any data. Hence block numbers remain at their initial default value of 0. Although those blocks contain no data, no disk space is wasted. We can deliberately create such blocks in a file by using *lseek()* & *write()* function calls together.

As we had seen accessing a large byte offset requires referencing double or even triple indirect blocks. Obviously *Kernel* has much more work to do, which may slow down the system performance. But in actual reality most of the *UNIX* files (i.e. those installed along with system installation) are less than 10 KB, which directly fit in first 10 direct blocks of *inode list* and hence most of the times *UNIX* looks much faster than other operating systems.

Other possible way (1) ^ Above we saw the *inode list* with assumption, that 1 block is containing 1 KB of data. But BSD *UNIX* implementation allows 1 block to contain 4 KB or even 8 KB, making large file assessment more faster in single operation. But this may lead to another problem of disk degramentation. ***HOW?*** Suppose 1 block contains 8 KB, and if a file is of 12 KB. Allow such a file will acquire one complete block (8 KB) and half of the next block (4 KB). So remaining half of the second block is wasted. Because no other file's data will acquire that space as it is allocated to the 12 KB sized file. So ***Berkeley*** implementation gave a solution of assigning a totally separate block for many files, which will contain that data of file, which does not fit in the *inode* block. Thus in above example, instead of wasting second block, the remaining 4 KB data of file will be saved in this new block. This solution is under utilization for BSD *UNIX*, but not yet solely approved by other flavors of *UNIX*. This new block which is going to contain such fragments of file (i.e. the data which is small to fit in a block and wastes a block) is termed as "Fragment Block".

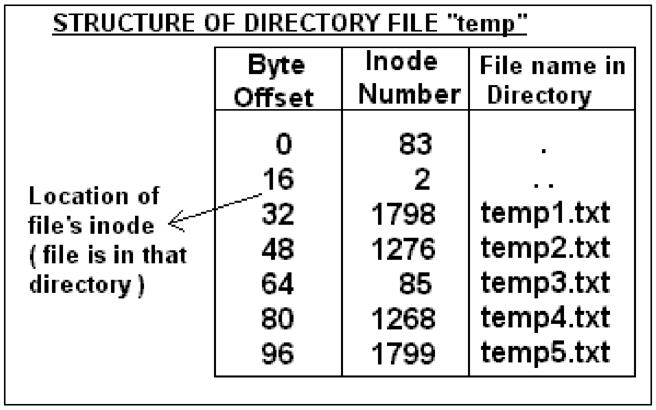
Other possible way (2) ^ In our discussion of *inode*, we seen that *inode* does not actually contain file's data. But *inode* contains addresses of logical blocks where the actual file data is located. Some implementation tried to keep actual file's data in *inode*. This strategy states that give the *inode* block size equal to that of respective file. Then a small portion of this block will contain the inode structure and all remaining block will contain the file's data. This is a good idea for system performance, because only one disk access will be required to access whole file. But drawback is that, *inode* size for every file vary and thus file integrity of *Operating System* will lost.

✷ *Directories:*

As we said on **page 2** (1st Chapter), that, in *UNIX* directories & devices are also considered as files. In *UNIX*, directories are the entities, which gave the system its hierarchical (tree like) *File System* structure.

Directory File contains some sequential entries. 3 things are important....

1) ***Byte offset in Directory File :-*** This is the number (unsigned integer hence always +ve) which gives location of a file's *inode* in that *Directory File*. The number starts from 0 and is exactly divisible by 16. So the offset numbers are 0, 16, 32, 48, 64, 80, ... etc.



STRUCTURE OF DIRECTORY FILE "temp"

Byte Offset	Inode Number	File name in Directory
0	83	.
16	2	..
32	1798	temp1.txt
48	1276	temp2.txt
64	85	temp3.txt
80	1268	temp4.txt
96	1799	temp5.txt

Location of file's inode (file is in that directory)

This is */temp* directory, created directly under root directory. This directory contains 5 files, *temp1.txt* to *temp5.txt*.

So when we ask what is the location of "temp3" file's *inode*, in *Directory File* "temp", we should say that it is located at byte offset 64.

As byte offset is an "offset", means the distance of a file from the beginning of this *Directory File*, every *Directory File* will have same 0 to nth offset from 0 to the number files in that directory, given a number of offset as exact multiple (or exactly divisible by 16) of 16.

Also note that, the first two file names in every directory are dot ("**.**") and dot-dot ("**..**"), where : "**.**" Represents the present or current *Directory File* and "**..**" represents the parent *Directory File* (whose subdirectory is the present directory.

✷ The entry of "***.***" & "***..***" in the *Directory File* of "/" root directory is done by ***mkfs*** program during *File System* creation while installed *UNIX* operating system.

In our example of "temp" directory, "**.**" will represent the "temp" directory itself and the "**..**" will represent the parent directory of "temp" directory, which "/". (Because we already said that "temp" is created directory under root directory). Out byte offset of "**.**" file is at 0 and of "**..**" file is at 16.

Note that byte offset number is the location of the file's *inode* in that directory. It should not be misunderstood with *inode* number of the file in that directory. *Inode* number of a file has a separate entry in *Directory File*.

2) ***Inode Number :-*** As seen on page 53, *inode* number is a unique number given by the system to a specific file. So this number is the "identifier" of a file for *Kernel*. Its entry is made in the *Directory File* along with its name. So in our example "temp3.txt" file's *inode* number is 85, and this *inode*'s location in that directory is at byte offset 64. ***You may ask here that, suppose***

we created such a file which is not in any directory, then where its entry will go? This is ***impossible***, because if you don't create a file in any other directory, then automatically it will get created under root directory (i.e. "/") and its entry will be found in "root *Directory File*". *Inode* number is of 2 byte size.

(✶ ***Where is the inode number of "/" file?*** Obviously it will be written in *Directory File* of "/" and will be denoted next to "**.**" in that file, because "." Represents current directory.

3) File Names: - These are names of files located under present directory. Then names are just null. terminated strings. Though *UNIX* system does not force a limit on the length of the file, under *Directory File*, names are restricted to 14 characters when written in *Directory File*.

So *UNIX* says that each entry in a *Directory File* is of 16-byte size. (14 bytes for 14 characters in file name and 2 bytes for *inode* number).

Above line may surprise us, where is the size of byte offset number? Why isn't it included in total entry size?

Note that, each entry in *Directory File* contains only 2 things: 1) the *inode* number of a file (file is supposed to be present in that directory) 2) the name of the file in that directory.

Then why the tabular structure shown on page 67 looks like the 3 entities in each entry?

To get answer of this question, consider following three lines of a "C" code…

fprintf (file, "%2d%145", 83, ".");
fprintf (file, "%2d%145", 2, "..");
fprintf (file, "%2d%145", 1798, "temp1.txt");

The output will be ….

```
8 3                        .  2                           .  .
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

The actual output will be in straight horizontal line, because no new line character is used in *fprintf()*.

[**About file name entry in *Directory File* : -** The file name written in a *Directory File* is a null terminated character string, with the full path. Path is separated by "/" character indicating subdirectories. Obviously all names (except the last) must be a directory name].

The *Directory File* has structure like above output. Means the format strings %2d & %14s make the entry of 16 bytes (2+14). Obviously if first entry begins at 0 i.e. beginning of the file, then the file name will take position on 15th byte, obviously next entry will being at 16th byte. Then next entry's file name "**..**" will occupy 30th & 31st byte and thus the third entry will begin at 32nd byte.

Sp the *Directory File* is written in such a way that a new entry will begin at the exact multiple of 16. As this task is achieved automatically by formatted file output function, *Kernel* will find the entry for a specific file at the offset number (i.e. file pointer position) which is exact multiple of 16. Hence there is no need of making "byte offset number" as one of the content of entry in *Directory File*. Thus entry in *Directory File* is made up of only 2 things and byte offset number can be automatically found due to the "formatted" nature of *Directory File*.

In our sample code, we hadn't used "\n" new line character. This is because the limit of entry is 16 byte. Suppose a file has name of 20 characters, then system will take first 14 of them to fulfill the 16 byte size. If "\n" is added, then obviously entry size will become 17 bytes ("\n" holds 1 byte). This indirectly also tells us that *Directory File* will grow horizontally as there are no new line operators.

Above discussion also tells us a very important thing, when a process needs to open a file, *Kernel* will go to the path specified by process, opens the *Directory File*, and searches the entry for file name by jumping from 0 to 16 to 32 and so on. When file name matches, it will take *inode* number of that file, then it will look into the *inode* structure of that file in its global *inode list* and then as explained on **page 61 to 66**, will look for logical blocks in which file data is located and finally will assign a buffer (from buffer pool) for each block of that file and then will put contents of those buffers on screen one by one.

So structure of *Directory File* is utmost important and must be maintained by consistent through out. That is why any user (even administrator – a super user) does have "write permission" for *Directory File*. It is the *Kernel*, which only has write permission for *Directory File*.

For a directory user is given all 3 permissions : read, write and execute, but they are concerned with the contents of the directory and not at all concerned with the actual *Directory File*.

Thus ....

✷ *Read permission to the user for directory means*, allow the user to see contents i.e. files in that directory.

✷ *Write permission to the user for directory means*, allow the user to create or delete subdirectories & files in that directory.

✷ *Execute permission to the user for directory means*, allow the user to search the directory i.e. allow the user to open subdirectories in that directory.

For *Kernel*, on other hand, writing permission for directory means, to actually write in *Directory File*. *Kernel* writes to *Directory File* just like writing in any other regular file. But *Kernel* follows strict "formatting" strategy while writing on *Directory File* to maintain the consistency of *File System*. In other words, a *Directory File* is just as a regular file for *Kernel* but a special i.e. "not to touch" type file for all users of the system including the super-user.

As *Directory File* is just like regular file for *Kernel*, it also has *inode* number (which is seen in *Directory File* along with "**.**") and all levels of direct & indirect block depending upon size of *Directory File*.

Thus, a *Directory File* is a "database" for *Kernel* of the files inside that directory. The idea of treating directories & devices as "file", is from the brain of ***Dennis Ritchie*** (one of the creator of *UNIX*).

The *Directory File* structure given by *UNIX* system becomes so popular that, ***Microsoft*** adopted this idea as it is for its DOS & Windows *Operating System*.

In the second paragraph of this page (Above discussion....), we said that, when a file is to be opened, *Kernel* looks for the entry of the file name in the *Directory File*, according to the path given by a process that wants to open the file. So here *Kernel* will look for file name, but actually *Kernel* not at all interested in the file name. Rather it is interested in the *inode* number of file name. So the path of a file given by a process should ultimately lead to the *inode* number. *Kernel* uses a special algorithm called as ***namei*** to convert a file path to file's *inode*.

✷ Conversion Of Path To Inode : - As seen in last paragraph, the algorithm for this is ***namei***, which is as follows ....

```
namei algorithm /* converts path string to inode */
input : path name string
{
  if (path name string starts from root i.e. "/")
          working inode = root inode (using iget) ;
  else
          working inode = current directory's inode (using iget) ;
  while (there are more components in path string)
  {
          read next component from input parameter ;
          verify :        1) the working inode is of directory
                  2) the access permissions for it are OK ;
          if (working inode is root inode & component is "..")
                  continue ; /* go back to while */
          read the Directory File (using working inode) by repetitive use of
                  bmap, bread, brelease ;
          if (component matches any entry in Directory File)
          {
                  get inode number of matched component ; /* here there should be code for mount
```

```
                    point crossing.*/
                release working inode (using iput) ; /* because it is now unnecessary */
                now working inode = inode of matched component ; /* using iget again */
            }
            else /* means if component doesn't match */
                return (no inode in there) ;
    } /* end of while loop */
    return (working inode) ;
}
output : a locked inode of desired file.
```

Now we will explain this algorithm by taking one example. Suppose we want to get *inode* of file named temp3.txt in /temp directory. (See on page 67-68). So the input parameter path string will be either */temp/temp3.txt* or *temp/temp3.txt* according to "absolute path" or "relative path" respectively.

The algorithm, which converts this path to inode, is called as "***namei***".

From Chapter 1, page 8, we can say that, every process's program is stored inside a directory and that directory becomes the "execution environment" of the process. In other words, every process (except the init or process 0) is a child process of its current parent process. So the "execution environment" of any child process is the current directory of its current parent process. (Note – The current directory of the parent process does not mean the directory where the program of parent process is stored. But it is the directory from which the parent is executed).

The pointer to this current directory is made in the "u area" of the process. So while searching for a given file path, *Kernel* has 3 choices depends upon the nature of the specified file path.

- ✓ If the give path is "absolute path", then *Kernel* has to start its search right from the root directory i.e. "/".
- ✓ But when the given path is "relative path", then *Kernel* always starts its search from "execution environment" of the process. Means from the current directory of the of the parent process of this process. *Kernel* finds this current directory's pointer in *u area* of the running process and starts searching for it.
    - ❖ Here one point must be kept in the mind, that the pointer to the current directory is not the pointer to its name but actually it is pointer to its *inode*.
    - ❖ So in case of "relative path", *Kernel* finds *inode* of current directory in *u area* of the process and then using this *inode* starts searching the given path.
    - ❖ Now a question may arise, ***if Kernel needs inode of current directory, if there is "absolute path", then where the inode of "/" is located?*** The answer is that, the *inode* of "/" is located "globally" and hence *Kernel* has no problem to access it while searching the absolute path.

So when *Kernel* starts searching from "/", it uses a global variable to access *inode* of "/" and when *Kernel* starts searching from "relative path", then it gets *inode* of current directory from *u table* of process.

The term "current directory" is slightly fishy here. We know that program can be executed to make it as "process" by typing name of the program and then pressing enter key. Here 2 cases must be considered….

1) Switch to the directory where the program is located (i.e. either by *cd* or by *chdir*) and then type program's name and press enter. This seems very logical to start a program by going at the program's location.
2) But some programs are so commonly used, that every time switching to the directory of the program is very time consuming. For this reason, path of such programs are written in system startup files (in DOS, the file is *autoexec.bat* and in *UNIX*, the file is *.profile*). Now when you reboot the machine after entering path into *.profile* file, now onwards the program located inside

the specified path can be executed from anywhere in the system. Means now you don't have to switch to the program's directory to execute the program. So you can execute the special programs from any directory. This happens because system startup file sets the execution environment to the path specified in the *.profile*.

Now coming back to our discussion of "current directory", suppose we have a program "abc", located in */temp/progs* directory. We switch to */temp/progs* and type *abc* and press enter. Now *abc* process is started. If we ask what is the current directory of *abc*, we say that */temp/progs* is the current directory of *abc*.

This answer is very much accurate. But the reason of answer is very important. We said that "current directory" of *abc* is */temp/progs* because "our program *abc* is inside it". But this is not correct reason. The correct reason is....

Recall from chapter 1, every process has a parent process except "process 0". When system starts up, process 0 begins automatically and then it starts its child process *init* & then *init* starts the *shell*. Appearance of command prompt (say # or $) indicate that shell is started. Now any program started from command prompt is the child process of shell.

When we start abc from "/temp/progs" command prompt, the "abc" process has its parent process the shell. Now shell is that process which is most commonly required one. So its path is in *.profile* file. So shell can start any where. Thus the directory of shell, here is /temp/progs and hence the current directory of abc is /temp/progs. Because "current directory of any process (except process 0) is the directory of its parent when the process is created".

Processes can change their current directory by using *chdir()* system call (means "change directory"). Thus search for path always begins at current directory except for the case, when path is given with initial "/" character. In this case search begins from root directory.

Coming back to our discussion of ***namei*** algorithm, now we can say ... 1) If path does not start with "/" character, the search will begin from current directory and pointer to this current directory is stored in "process *u area*" 2) If path begins with "/" character, then the search will begin from "root directory" and the system root *inode* is stored in a global variable. In this case too, current directory is stored in "process *u area*".

Sometimes process may use *chroot()* (i.e. change root) system call to change the notion of *File System* root (obviously temporarily), in this case the changed temporary root s stored in "*process u area*".

Algorithm: -

- ✓ If path begins from root, the "working *inode*" variable will be assigned to root *inode*. To get the root *inode*, *Kernel* uses iget() algorithm (see on page 56-57, to get the root *inode*).
- ✓ If path does not begin with root, then the "working *inode*" variable is set to "current directory *inode*". The *Kernel* uses *iget()* algorithm to get the "current directory *inode*".
- ✓ Now using this "working *inode*" variable, a loop starts. The loop condition every time checks how many components are left in path string.
- ✓ Inside the loop, next path component is read.
- ✓ Then "working *inode*" is checked, whether this *inode* is of directory, whether access permissions are ok or not and so on.
- ✓ If working *inode* is of root (i.e. root *inode*) and the component is "**..**", then control is send back to beginning of loop, because "**..**" means parent directory & "/" has no parent directory.
- ✓ Else, means if component is not "**..**", then obviously working *inode* will be a directory. Hence *Kernel* reads this directory (i.e. the working *inode*) by using bmap, bread & brelease algorithms repetitively.
- ✓ If the component matches with the entry in directory (working *inode*) file (see directory file on page 67-68), then it gets *inode* number of the matched component.
- ✓ Then it releases the current working *inode* (because here onwards search ends and working *inode* is not necessary) by using iput algorithm.

- ✓ Now "working *inode*" is newly set to the *inode* number matched with the component. For this *Kernel* again uses iget algorithm with *inode* number (found from *directory file*) as the parameter to *iget()*.
- ✓ Now if the component does not match with the entry in *directory file* (by looking inside the directory's *inode*), then obviously the given path is invalid, and thus there is no *inode* and hence algorithm returns "no inode" situation.
- ✓ If everything works fine, then the "working *inode*" now contains valid *inode* number, which is returned to the *Kernel*.

✷ ***Some Points To Remember: -*** The variable "working *inode*" is a local variable, to store the currently found *inode* during iteration of loop in the algorithm. The *inode* where the search begins is obviously the 1st working *inode* and search begins here while working with this "working *inode*", *Kernel* assures following things…

a) Whether the working *inode* is a directory or not. If not, then *Kernel* flags error that the given component in path is a file and not the directory. (Remember that *directory file* is a not leaf node & ordinary file is always a leaf node).

b) The process, which is looking for the path, must have "execute permission" for all directories mentioned inside the path. Again recall that "execute permission" for a directory means permission to traverse all of its subdirectories. If permission is not allowed, *Kernel* flags error of "no access permission".

c) The use ID of the process must match with the owner or group ID of that process. OR at least the path of file (to search for) must have "execute permissions" to "all users (i.e. "others")". Otherwise search fails.

2) When *Kernel* searches *inode* of a directory, to look for a file name entry inside that *directory file*, *Kernel* uses linear search method. For this *Kernel* starts from 0th offset of that *directory file* (see the figure of *directory file* on page 67). Then it converts this byte offset to the appropriate logical disk block number by using *bmap()* algorithm. Then it reads this logical disk block by using *bread()* algorithm while reading the disk block, it tries to match the path component with the entries in that block. If match is found, it gets the *inode* number of matched component (recall that *directory file* contains *inode* number (stored in 2 bytes) corresponding to file name entry) and then releases this block by using the *brelease()* algorithm. At the same time it releases the "old working *inode*" by using *iput()* algorithm. Now finally it assigns newly found matched *inode* number to the "working *inode*" variable by using the *iget()* algorithm. Thus now new matched *inode* becomes the "working *inode*". Then it loops back to search condition "whether still some components are remaining in path", if yes, it repeats the whole above procedure and if not, means no components remain in path, then returns this "working *inode*" as the *inode* of the file for which the search was started.

3) I f *Kernel* does not found any match with the directory entries in the directory block, then it releases the block (by *brelease()*), jumps to next offset (i.e. 16) and repeats all above process in (2) by *bmap()*, *bread()* & *brelease()*. This repetition goes on until all directory entries get finished and search reaches ***EOF*** of *directory file*.

4) We will make all above things more clearly by taking one example. ***Suppose a process wants to open /temp/temp1.txt file (either for reading or writing) OR wants to search for the same above file.***

While passing the path /temp/temp1.txt, *Kernel* first found "/" character
It makes "working *inode*" as root *inode*.

↓

Checks whether "/" for whether a directory or not

↓

As "/" is a directory, it passes next component, which is "temp".

↓

It ensures whether the process has proper permissions to execute the "/" directory.

↓

It opens *directory file* of "/" directory and starts looking from byte offset 0 for the entry of name "temp". For this it takes each entry one at a time, allocates & de-allocates blocks, reads blocks (all by *bmap()*, *bread()*, *brelease()*).

↓

When entry for "temp" is found, *Kernel* releases current working *inode* (which was for "/") by *iput()* and assigns it to the *inode* of "temp". [For this it uses 2 byte entry of *inode* number situated prior to name "temp". Then it takes this *inode* number of "temp", passes it as parameter to *iget()* and gets *inode* for this "temp" and finally assigns it to working *inode*].

↓

Again it ensures whether "temp" is directory or not and whether process has permission to execute "temp" directory or not.

↓

Then it goes to path, looks for next component, which is "temp1.txt".

↓

It opens the *directory file* of "temp" and starts looking for the entry of "temp1.txt" by same above method.

↓

On finishing match with "temp1.txt", it takes inode number of temp1.txt, releases current working *inode* (which was for "temp") by *iput()*, then uses *iget()* to get *inode* of temp1.txt and assigns returned *inode* to "working *inode*".

↓

Again it loops back to search path, look for next component, but as no components are left, (means path ends or say exhausted), the working *inode* is the required *inode* and returns it to process.

↓

By taking the *inode* of required file, *Kernel* goes to "in-core *inode list*" looks for blocks in the 13 member array of *inode*, uses "buffer pool" algorithms to get logical disk blocks, keeps addresses of these blocks in global file table and puts a pointer (which points to the address of logical blocks of this file) in process's user file descriptor table and finally returns the "position number" (i.e. the location where the pointer is kept) to the process. The process gets this number as "index" or "file descriptor" of the file being opened and gets it as return value of *open()* system call.

Now onwards, this file descriptor or say index can be used to read this file (by *read()* system call) or to write on this file (by *write()* system call)

5) Repetitive use of *bmap()*, *brelease()* & *bread()* algorithms in "linear search" method looks or seems to be slower or time consuming. But according to ***Ritchie***, as it is associated with "fixed size" (16 byte per entry) of *directory file*, the mechanism is in actually more fast.

✷ Inode Assignment To New File :- i.e. how a new inode to created when creat() system call is used? : Up till now we were dealing with a situation, where we already have a file & its *inode*. So we just used “already available” *inode*.

But what about creating a new *inode* (i.e. in other words, creating a new file)?

Before continuing with “*inode* assignment to a new file, we will first look at the contents of *Super Block*.

✷ ***Contents Of Super Block / Structure Of Super Block***.

The structure of *Super Block* contains following 10 fields….

1) Size of the *File System*
2) The number of free blocks in the *File System*.
3) A list of available free blocks in the *File System* (not actual blocks but this numbers only).
4) The index number of the next free block in the free block list.
5) The size of the *inode list*.
6) The number of free *inodes* in the *File System*.
7) A list of free *inodes* in the *File System* (not actual *inodes*, but their numbers only).
8) The index number of the next free *inode* in the free *inode list*.
9) Lock fields for free block and for the free *inode lists*.
10) A flag, which indicates that the *Super Block* has been modified or not.

Now we will turn our attention to the topic of “new *inode* assignment to a newly created file”. The algorithm for this is called as ***ialloc*** which is as follows:….

```
01 : ialloc algorithm /* allocate inode to a new file */
02 : input : File System
03 : {
04 :   while (not done)
05 :   {
06 :           if (Super Block is locked) /* means no access to inode list in Super Block */
07 :           {
08 :                   sleep (event : Super Block becomes free) ;
09 :                   continue ; /* go back to while loop */
10 :           }
11 :           if (inode list in Super Block is empty)
12 :           {
13 :                   lock Super Block ;
14 :                   get “remembered inode” for search of free inode ;
15 :                   search the disk inode block for free inodes until Super Block (i.e. the list of free
                          inodes in super Block) becomes full, or no more free inodes (i.e. by bread() &
                          brelease()) ;
16 :                   unlock Super Block ;
17 :                   wake up (event : Super Block becomes free) ;
18 :                   if (no free inodes found on disk)
19 :                           return (no inode) ;
20 :                   set remembered inode for next (future) free inode search ;
21 :           }
22 :           /* there are inodes in Super Block inode list */
23 :           get inode number from Super Block ;
24 :           get inode (algorithm iget) ;
25 :           if (inode not gets free after all (means) inode is got from above 2 steps, but it is not free ) /*
!!! */
```

```
        {
                write inode to disk ;
                release inode (algorithm iput) ;
                continue ; /* go back to while loop */
        }
        /* inode becomes free or is free */
        initialize inode ;
        write inode to disk ;
        decrement the "File System free inode count" in Super Block ;
        return (inode) ;
    } /* end of while loop */
}
output : locked inode
```

As seen before, *Kernel* uses *iget()* algorithm to allocate a known *inode* to an existing file (obviously we have *File System* & *inode* number of this existing file). Thus the *inode* number was previously decremented.

But this time, we have a new file, means we have only "*File System*" and we don't have "*inode* number" yet!

As mentioned before in 2nd chapter (on page 19), the *File System* has linear list of *inodes* called as "*disk inode list*".

The *inode* is said "free", when its "type" field (see figure of *inode* on page 54) is 0.

So obviously we will say that, when *Kernel* needs a new, free *inode*, *Kernel* will search this "*disk inode list*". But this type of search will be very time consuming, because of least "one disk read operation" will be required for one *inode* and the "*disk inode list*" might contain thousands of *inodes* (according to the number of files in that *File System*.

To improve performance, the above method of search is avoided. Instead one array of number of free *inodes* (i.e. number of each free *inode*) is kept in the *Super Block* as one of its field.

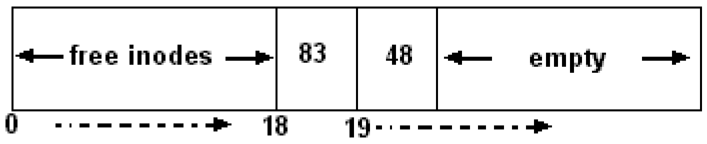
✷ ***ialloc() algorithm*** **:-**

1) *Kernel* first checks whether some other process locked the free *Super Block* or not.
2) If *Super Block* is already locked, then the process, which requires new inode, goes to sleep on the event of waiting for *Super Block* to get free.
3) And *Kernel* goes back to white loop and starts the whole process again.
4) But if *Super Block* is not locked, then *Kernel* can look inside the *Super Block*. Now while looking inside the *Super Block*, if *Kernel* finds that the "free *inode* list" array inside the *Super Block* is empty, and then *Kernel*'s duty is to first fill the *inode list* array of *Super Block*.
5) For this purpose *Kernel* first locks the *Super Block*, so, no other process would access it.
6) Then it gets the "*inode* number" of "remembered inode". Obviously at the beginning, this will be 0.
7) Then *Kernel* starts searching of disk for free *inodes* and start filling of *Super Block*'s *inode* array by the *inode* number of "found free inodes". *Kernel* identifies "free inodes" by viewing 0 at the "file type" field of *inode*. *Kernel* continues this task, until, either the array of super lock gets filled or there remains no free *inodes* on disk.
8) As filling of array of *Super Block* is completed, now *Kernel* unlocks the *Super Block*. And awakes all those processes which slept on the event of "locked *Super Block*".
9) Sometimes it may happen that, *Kernel* may not found any free *inode* on disk. In such a case *Kernel* returns "no *inode*" situation.
10) But if everything works smoothly up to step (8), then *Kernel*, after filling the array, remembers the highest numbered *inode* (that it already filled in the array). This *inode* is called as "remembered inode" which is the last one saved in *Super Block*. This highest numbered *inode*, later, used by *Kernel* to start next search in the while loop.

11) We discussed the condition of "empty *inode list* in *Super Block*", from step (4) to (10). But if there are *inodes* in *inode list* of *Super Block*, then *Kernel* simply takes next *inode* number (remember that *Kernel* only takes number) from the list.
12) Then it uses this number as input to *iget()* algorithm and then by using *iget()* it gets the *inode*. Means it allocates a "free in–core inode" (obviously taken from *Free List* of inodes) for the newly got *inode*. If necessary it can read this *inode* from the *disk inode* from the *disk inode list*. Then it copies this *inode* to the in–core copy, initializes the fields in the *inode* and returns the locked *inode* to *disk inode list* to indicate that, this *inode* is now "in use".
13) Very rarely (indicated by three exclamation marks on line number 27 of the algorithm), race condition may occur. A "non zero" file type field in *inode* indicates that *inode* is already assigned. Means not free means the received *inode* from iget () is yet not free, then in such cases, *Kernel* writes this "received inode" back to disk, then release the *inode* by input () and go back to the starting point of loop and starts whole procedure again.
14) The most important point to remember that, whenever *Kernel* gets a free *inode* (at step (12)) it decrements the free *inode* now on words will not be considered as "free inode". Rather consider that this *inode* is now "in use".

Examples ^ We will consider 2 examples of allocating new *inodes*.....

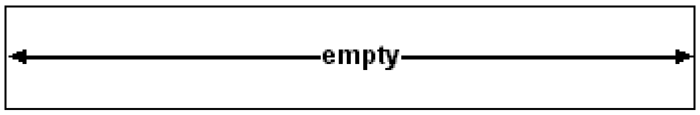
1)



Above figure is of "free *inode* number list" in *Super Block*. Suppose "at present time" there is 20 free *inodes* in the list indexed from 0 to 19. Remaining list is empty.

When *Kernel* looks for free *inode*, it takes the last *inode* i.e. 19th *inode* (whose number is 48) and thus now on words list will contain 19 *inodes* whose last one is 18th *inode*. So *Kernel* after taking 19th *inode*, decrements the 8th field of *Super Block* (which is "index number of next free *inode* in the free *inode list* of *Super Block*". See page 74-75) to 18, indicating that the next free *inode* is now index 18 (whose number is 83).

2)

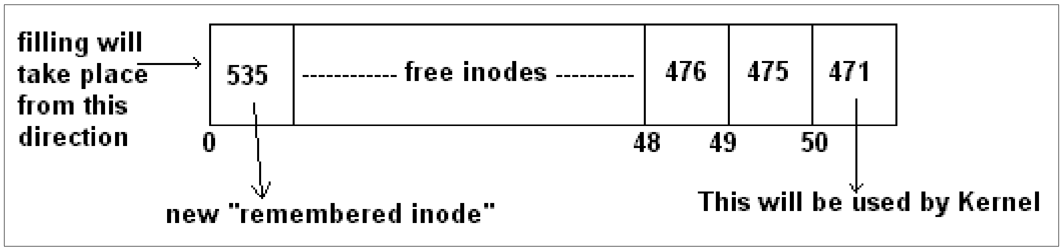


Now the "free *inode* list" in *Super Block* is empty. Suppose the last time "remembered inode" is of number 470. (Note that this is the *inode* number, not the index). As the list is empty, this 470 numbered *inode* is also already used.

Now when *Kernel* starts filling this list, it will do it from the ***inode* number = "remembered inode" +1**. Means the first one to fill will of number 471. Filling starts in such away that, the first "filled one" (i.e. 471 numbered) will be pushed to right side and newer free *inodes* will get added from left side. So suppose 51 *inodes* are added (assuming max capacity of *Super Block* free *inode list* is 51), then 471 numbered *inode* will be of index 50, and the highest numbered *inode* will be at index 0. We will assume that *inode* of number 535 is at index 0.

After complete filling, now *Kernel* takes the last *inode* i.e. of index 50, means *inode* number 471 for its new *inode* and remembers *inode* of index 0, means *inode* number 535 as the new "remembered *inode*" for next search if required.

✴ Above explanation also tells one important thing. The *inode* numbers at the end of the "list array" (means at high index) are smaller than the *inode* numbers at the beginning of the "list array" (means at low index). <u>In other words</u>, the array is filled in "<u>decreasing order</u>".


filling will take place from this direction
535
----------- free inodes ----------
476
475
471
0
48
49
50
new "remembered inode"
This will be used by Kernel

Also remember that, list will contain only free *inodes*, hence though 0th index *inode* number is 535, <u>don't expect</u> that 1st index will have *inode* number 534. It may be there if and only if it is free (i.e. not assigned. Means its "file type" field is 0).

✴ Freeing Of An Inode : - The algorithm for free the *inode* is known as *ifree()*, which is as follows :...

```
ifree algorithm /* make inode free */
input : inode number to free
{
    increment "free inode count" ;
    if (Super Block is already locked)
            return ;
    if (inode list in Super Block is already full)
    {
            if (inode number is less than "remembered" inode)
                    set remembered inode = input inode number ;
    }
    else
            store inode number in inode list of Super Block ;
    return ;
}
output : nothing.
```

This algorithm takes the "*inode* number" of the *inode* that we want to free.

- ✓ First it increments the total number of available free *inode* count.
- ✓ Then it enters in *Super Block* to update it. But if *Super Block* is already locked, it avoids "race" condition, by immediate returning. (i.e. no sleep & wait business). Obviously in such a case, this *inode* cannot be found in *Super Block* (as *Kernel* cannot enter in *Super Block* when locked) but can be found on *disk inode list* (so when *Kernel* reads disk for free *inodes*, this *inode* will found on disk).
- ✓ But if *Super Block* is not locked, but "free *inode list*" is full, means there is no space to keep this *inode*, then *Kernel* compares this *inode*'s number with that of the "remembered *inode*'s number". If this *inode*'s number is less than that of the "remembered" *inode*'s number, then it sets remembered *inode*'s number to this *inode*'s number and returns.
  So now onwards, the newly freed *inode* number will be the "remembered" *inode* number.
  Here a question may arise ***"what about the old, discarded "remembered inode"?*** Note that, this old, discarded *inode* number is not lost. Because *Kernel* can find it on "*disk inode list*" at any time.

Examples : - We will take 3 examples of freeing *inode*

1) Consider the figure of "free *inode list*" in *Super Block* on page 77. This list has "empty" portion. Means there is room to place "free *inode*". In such case, *Kernel* just keeps free *inode* at proper position (maintaining the order), pushes *inodes* to right side accordingly, and as one free *inode* gets added to the list, it increments the index to next free *inode* and proceeds.
2) Now suppose the "free *inode list*" in *Super Block* is full as shown in figure 78. In such case there is no room to put freed *inode*. Now suppose the free *inode* number is 499. According to the figure the "remembered *inode* number" is 535. So the freed *inode* number is less than the remembered *inode* number. Here *Kernel* marks the freed *inode* (i.e. 499) as new "remembered inode" and discarded the old remembered *inode* (i.e. 535) from the list. But don't think that 535 is lost. When *Kernel* needs *inode* number 535, it will start searching the *disk inode list*, from *inode* number 499 and finds *inode* number 535 on *disk inode list*.
3) Consider the same situation as in (2). Means "free *inode list*" on *Super Block* is full , the "remembered *inode*" number is 535 and now the freed *inode* number is 601. This case is not handled in algorithm. Means *Kernel* will directly fall on the last return statement (line number 14 in algorithm) and returns. This does not mean that *inode* 601 is not free. "It is free, but not on the *Free List*". Means when *Kernel* searches *disk inode list*, starting from *inode* number 535 (as it is "remembered" one), it will found *inode* 601 on "*disk inode list*".

✷ **Race Condition in In *ialloc()* algorithm ^** The step 13 of *ialloc()* algorithm, explained on page 77, shows a rare race condition.

- ✓ Suppose there are 3 processes A, B & C.
- ✓ Suppose *Kernel* is now working with process A. For process A, *Kernel* assigns one *inode* say X by using *ialloc()* algorithm. *ialloc()* algorithm internally uses *iget()* which has its own "sleep condition" and also *iget()* internally uses *bread()* which is also has its own "sleep" condition and sub conditions of sleep (And *ialloc()* itself also uses *bread()*). So there are many chances that "*ialloc()* using" process may sleep.
- ✓ Thus suppose that process A sleeps before copying the *disk inode* into in-core *inode* during *ialloc()*.

  Now *Kernel* schedules process B. process B wants a new *inode*. Hence it also used *ialloc()*.
- ✓ Suppose now "*Free List* of *inodes*" in *Super Block* is empty. So process B starts searching "disk list of *inodes*" and starts filling the "*Free List* of *inodes*" in *Super Block*.
- ✓ Suppose also that the search of process B starts at an *inode* which is less than the *inode* being assigned to A. Obviously as A is asleep, its *inode* (which is yet not assigned completely) is considered as free and thus filling process of process B will pull it in *Super Block*'s "free *inode list*". After completion of filling of free *inodes*, process B will get a free *inode* and goes out of the picture. But it keeps the A's "due *inode*" as "free" on "free *inode list*" in *Super Block*.

  Now process A wakes up and completes its remaining task of assigning the *inode*. As these steps do not require "*Super Block*", the *Super Block* remains "untouched". So now process A is using its X *inode*.
- ✓ Now suppose, process C requests for an *inode* by *ialloc()* and suppose it picks up "X' *inode* from the "*Free List* of *inodes*" in *Super Block*. When it wants this *inode*'s in-core copy to process further, it sees that the "file type" field is already assigned (as A is right now using it). S o *Kernel* looks at this condition and tries to assign a new *inode*, which is obviously a disaster!!!

So to avoid such disasters, *Kernel* writes updated *inode* immediately to disk in *ialloc()* to reduce the chances of race. After writing updated *inode* immediately to disk, its "file type" field is set to valid file type and hence when *Kernel* searches disk for free *inodes* it won't take this updated *inode* as its "file type field" is not zero (means not free).

Another way of reducing race conditions in *ialloc()* is "locking of *Super Block*" when *Kernel* searches disk for free *inodes*. Because if *Super Block* is not kept locked during search, more than one process found the "*Free List* of *inodes*" empty and start searching *disk inode list*. As this process requires "hardware access", both processes will waste lot of CPU time by alternate sleeping & awaking.

Similarly during freeing of *inode*, if process does not check "*Super Block*'s lock", then one process may keep *inode* in it overwriting the changes made by another process which is currently searching & filling the *Free List* of *inode* of *Super Block*.

Thus "locking of *Super Block*" check is crucial both in *ialloc()* & in *ifree()*.

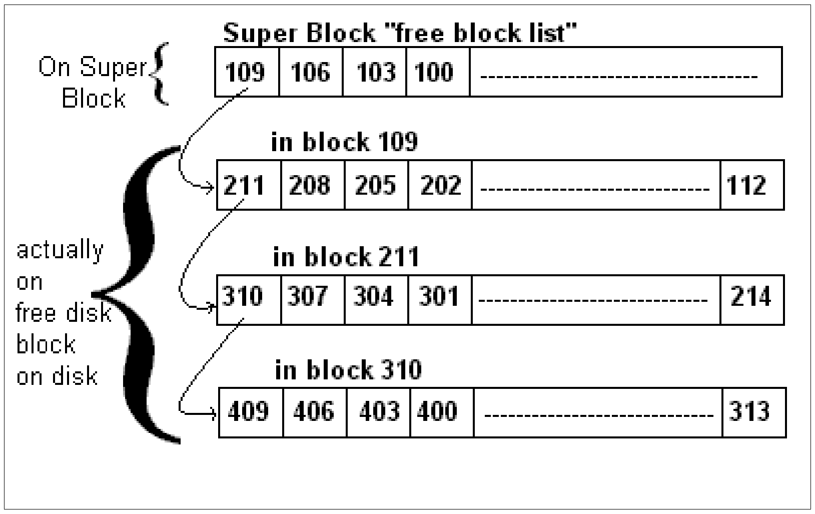
✷ Allocation Of Disk Blocks ^

When a process wants to write data to a file, ultimately (though buffered) the data is going to be written on a disk block (or data block). Thus a process when writing on a new file or when expanding an old file, will obviously require a new disk block.

Before proceeding to "new block allocation" algorithm, first we will see how disk blocks are arranged on the disk.

File data is actually stored in "data block" part of *File System* figure (see on page 19). It is very interesting to see how these data blocks are arranged. The system utility program ***mkfs*** organizes data blocks in such a way that...

1) The *Super Block* will contain array of free block numbers.
2) One element in this array is a block number and when looked into that block you will found another array of free blocks.
3) Other element in above array will also a block number, and when looked into that block you will found another array of free blocks.
4) And so on.
5) Free blocks in *Super Block*'s free block array and free blocks in block's "free block array" are at different locations and hence joined by link list.



6) All these arrays are maintained in "descending" manner. Means low index will have "large block number", while high index will have "small block number".
7) If we look at the figure on last page, we found that "*Super Block's*" free block list is an array, where first element is free block number 109. Now following the link of 109, we enter into the block 109, there we found another array of free blocks, whose first element is 211. Following the link of 211, we enter into the block 211, where we found another array of free blocks, whose first element is 310 and so on.

   You also able to see that low index elements (say 211 at 0th index in the array of block 109) is of higher value than the last element of higher index which have lower value say 112.

   So "free blocks" are kept by their numbers, first in *Super Block* and then actually in "free blocks". And all these arrays are linked by link list. This also tells us that, free blocks are those blocks which contain array of other free blocks and not any file data. Obviously when such a free block (containing array of other free blocks) is assigned to an actual file, this array must be wiped off.

The algorithm by which *Kernel* (on behalf of a process) writes data to a file by getting free block is called as *alloc()*. This algorithm assigns next free block (which is available) in the *Super Block* "free block list". Note that, once a free block is allocated, it cannot be reallocated until it again becomes free. The algorithm is as follows....

```
alloc algorithm /* free block allocation */
input : File System number (device number)
{
  while (Super Block is locked)
        sleep (event : until Super Block gets unlocked) ;
  remove the block from Super Block's free block list ;
  if (removed block is the last block)
  {
        lock Super Block ;
        read the removed block ; /* bread() algorithm */
        copy the "read" free block number (from the array in the block) to the Super Block's list ;
        release block's buffer (because bread() internally called getblk()) ;
          /* by brelease(), because bread() used this buffer */
        unlock Super Block ;
        wake up processes ; /* which slept due to lock of Super Block */
  }
  get buffer for removed block ; /* getblk() algorithm */
  zero buffer contents ; /* wipe off array info */
  decrement total count of free blocks ;
  mark Super Block as "modified" ;
  return (buffer) ;
}
output : buffer for new block
```

- ✓ The input to this algorithm is the device number (i.e. the *File System* number) from which a block is to be chosen.
- ✓ When *Kernel* enters the algorithm, it first checks whether the *Super Block* is locked or not. To ensure this locking mechanism is crucial to avoid race conditions. If the *Super Block* is locked, the process which wants the block goes to sleep until the *Super Block* gets unlocked. When *Super Block* gets unlocked, it can proved as usual to the next step of algorithm.
- ✓ Now if the *Super Block* is unlocked, then *Kernel* enters in next step of algorithm. Means it enters into the *Super Block*, looks for free block in "free block list" and remove next free block from this list.
- ✓ If the removed block is the last block in "*Free List* of blocks", means if the list is going to become empty after its removed, then *Kernel* locks the *Super Block*, reads the removed block, where it finds the "array of free blocks". It reads this array and copies it to the "free block list" of *Super Block*. So now onwards the "free block list" will not be empty. While reading the "removed block", *Kernel* uses *bread()* algorithm. Inside *bread()* there is a call to *getblk()*, which gives buffer for this "removed block" on which there was the "read" array. Obviously after copying of this array, this buffer has to be released and hence *Kernel* uses *brelease()* to release this intermediate buffer. Now *Kernel* unlocks the *Super Block* and wakes up all those processes, which want to sleep, and waiting for *Super Block* to get unlocked.
- ✓ But if the "removed block is not the last one", next step of algorithm.
- ✓ It uses *getblk()* to get buffer for this "removed" block.

- ✓ As buffer might contain some data (say the array of free buffers on it) which has to be cleared. Hence *Kernel* makes all buffer contents to 0 by wiping the contents.
- ✓ Then *Kernel* decrements the total count of free blocks by 1 (as new is now assigned).
- ✓ Due to changes made in the contents of *Super Block*, *Kernel* marks the *Super Block* as “modified” and leaves the algorithm by returning buffer to the “wanted” process.

✴^ Rarely *File System* may not contain any free blocks (means there is no last block in “*Free List*”, i.e. the “*Free List* of blocks” in *Super Block* is absolutely empty, then *Kernel* returns error to the calling process. *Kernel* does this by looking at the global “count of free block in *File System*”. If it is already 0, then there is no need for *Kernel* to enter in this algorithm and it just returns error.

✴ <u>One question may arise here that</u>, ***if the “removed block” is the “last one” and Kernel reads “array of free blocks” from this “removed block” by bread(), where it already gets the buffer (inside bread() there is a call to getblk()), then why it releases it and then gets buffer by recalling getblk() separately?***

The answer is that, *bread()* gets buffer by calling *getblk()*, where *Kernel* is sure that there is valid data on that buffer (i.e. array of free blocks). But as process wants “new block” to get allocated whose contents must be zero. Thus *Kernel* cannot use the buffer got from *bread()*. Instead it releases it and then get it by calling *getblk()* separately.

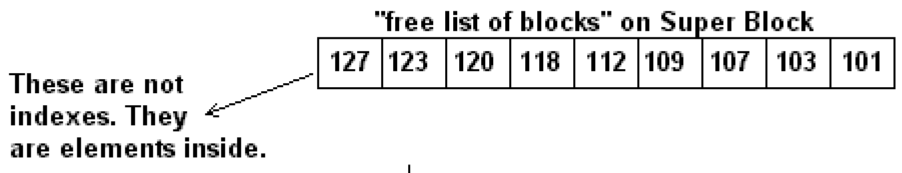
✴ If a process writes lot of data to a file, then obviously it is going to need more & more new blocks by repetitively call of *alloc()*. <u>But note that, *Kernel* assigns only one block at a time</u>.

✴ The utility ***mkfs*** organizes link list of free block arrays in such a way that the free block numbers are nearer to each other (Means free block number 109 will be followed by say 110 or 112 (if free) and not followed by 407. Because if “same” file’s data is written first on block 109 & then on 407, than the file data gets fragmented). This is necessary to reduce disk fragmentation of file’s data as possible. This helps in “system performance maintenance”, especially when a file is read sequentially.

✴ But in any case, *Kernel* makes <u>no effort to sort</u> the “free block numbers” on the list in *Super Block*.

✴ Freeing Of Disk Blocks :- The algorithm used for freeing of disk block is *free()* and is quite similar to *ifree()*.

If the *Super Block*’s “*Free List* of blocks” is not full, means there is room to keep the “freed block”, the *Kernel* keeps free block on the list as usual and increments the global “count of free blocks in *File System*”. But if the “*Free List* of blocks” on *Super Block* is <u>full</u> (means there is no room to keep freed block), then *Kernel* writes “present free block list on *Super Block*” to this freed block and keeps it in the “*Free List*”. Obviously this block becomes the only content of “*Free List*” now onwards and it has array of free blocks inside it (which was just written to it).


"free list of blocks" on Super Block
127 | 123 | 120 | 118 | 112 | 109 | 107 | 103 | 101
These are not indexes. They are elements inside.

↓

Suppose we block 130 is freed, as list is full, there is no room to keep block 130.

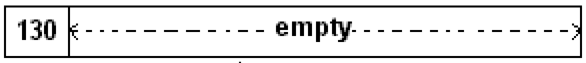
↓

Kernel writes all above “full” list inside the freed block 130 and keeps 130 on *Free List* (which is now empty)

↓

So after keeping 130 on list

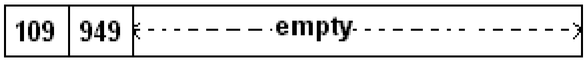
↓



Thus block number 130 becomes the only member of *Free List*.

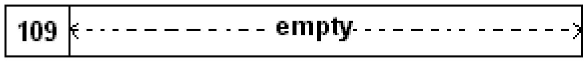
But again note that, inside 130, there is "full list" of first figure.

✷ Figure Of Assigning A Free Block ^



This is the "*Free List* of blocks" on *Super Block*. Currently there are 2 free blocks no. 109 & 949 on the list. Note that, inside these blocks there are yet two array of free blocks, one each.

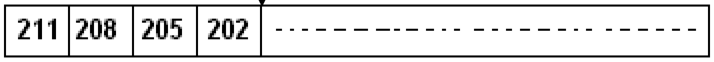
Now suppose block 949 gets assigned to a process by *alloc()* algorithm, then figure becomes



Now suppose, block 109 is assigned to some another process by *alloc()* algorithm..
Then initially list will be empty, because *Kernel* removes block no. 109.

Kernel now reads block 109, inside it, *Kernel* finds array of free blocks, because 109 works as link to this array. The array elements are say 211, 208, 205, 202 and so on.

These "read contents" from block number 109 now become new contents of "*Free List*" on *Super Block*.



The "new" free block list on *Super Block*.

✷ Comparison Between Inode & Block Algorithms : As seen up till now, *ialloc()* & *alloc()* algorithms are logically similar and also the *ifree()* & *free()* algorithms.

The major difference is that, system keeps data structure "link list" for blocks but there is no such list for *inodes*. Why?

There are 3 reasons...

1) As *inode* hat "file type" field, *Kernel* can easily determine whether the *inode* is free or not. There is no such mechanism for block to determine, whether the block is free or not. Thus for "block", *Kernel* needs some external entity to determine. This external entity is the "link list". If a block has some "link list" linked array, then it is free, else it is not free.
2) *Inode* has smaller size than that of disk block. Obviously keeping link list on *inode* is not possible. But it is possible to keep link list on block.
3) End users mostly need free blocks continuously (to write data on them) rather than *inodes*. (As *inodes* are more commonly used by programmers rather than by end users). Obviously more crowding is towards block and thus searching for free block is more critical than searching for free *inode*.

✷ Other Types Of Files ^ Up till now we saw information about "regular file" and "*directory file*". But *Kernel* supports 2 more types of files...

1) **Pipe :** They differ from regular files, because data on them is "transient". Means once data is read from them, cannot be read again. Another thing is that, you have to read data from pipes in the

same order in which it was written. Means data from pipes cannot be read from anywhere (say from any byte). Conceptually *Kernel* keeps data on pipe just like on regular file. But regular file may use direct blocks & indirect blocks, but pipes use only direct blocks. Obviously pipes are of too small capacity than that of regular file. Pipe is also called as ***fifo*** (i.e. "first in first out"), just like queue.

2) **Special files:** This includes "device files" either "character device special" or "block device special". As they are devices, *inodes* of them do not have any "Table of disk addresses" like data files. Instead *inodes* of them have 2 numbers
   a) Major device number.
   b) Minor device number.

   The major device number indicates the type of device (such as terminal, disk, etc) while the minor device number indicates the unit number of the device.

✷ At the last, algorithms described in this chapter are internal to the *Kernel* and thus are not visible to the user. These algorithms are "low level" algorithms for *Kernel*, to which *Kernel* uses them inside major "file subsystem manipulating APIs" like c*reat()*, *open()*, *link()*, *unlink()*, etc.