3rd Chapter

THE BUFFER CACHE

If we see figure of the *Kernel* on page 15, we will able to realize that *File System* and block devices (such as floppy or hard disk) do not interact directly. Instead there is an intermediate step known as ***Buffer Cache***. The *buffer* word actually means a "*shock absorber*", really this buffer too, functions like a shock absorber of repetitive disk operations.

We know that data is stored in a "*file*" and as a storage medium, "*file*" is stored on a disk. As a disk is made up of blocks, they are called as "Block Devices". Whenever user accesses a file (say wants to open a file to read), *Kernel* does not directly show the disk file to user, instead *Kernel* brings file data into main memory and then memory layout of the same file is shown to the user. (Though user has a feel, that he/she is looking a file actually on disk). Similarly when user wants to write to a file, *Kernel* brings the file into memory, user actually changes memory copy of that file and when says "save", *Kernel* puts back the changed copy of that file to disk.

Though above process looks simple and can be explained in single paragraph, the actual process is much more complicated and multi-stepped

From the previous chapter we know that every file has its unique *Inode* on *Inode List* and the total information about the *File System* is in *Super Block* of the *File System*. These two things (i.e. *Super Block* and *Inode List*) are also stored on disk in *File System* layout structure (as given in figure on page 19). So to read information about a file, *Kernel* has to bring these two things to memory for reading and examining purpose.

So when a file is brought to memory (either for reading or writing) from disk, actually 3 things are brought :

a) *Super Block* (which has disk layout)
b) *Inode List* (which has the concerned file's layout) and…
c) The actual file data.

Out of these 3 things, first two i.e. a) & b) are not actual file data, but necessary to access the specific file. These two items are called as ***Auxiliary Data***.

A s *Kernel* puts back the file data (after completion of task) to disk again *Kernel* also put back these 2 things to disk from where to he took them previously.

Now we will pay some attention to hardware. The main memory (i.e. *RAM*) is the "*electronic free space*" made available for executing programs. Obviously all main memory related operations are very fast. Conversely the disks (i.e. block devices) are magnetic media and require movement of their heads repetitively which is for more slower than the *RAM* operations.

So, though it is possible for *Kernel* to bring data directly from the disk and to put the data directly on the disk, there will be very large disk operations (i.e. head movements) which will ultimately result in poor speed of *Operating System*. So *Kernel* tries to keep disk operations as less as possible by keeping pool of internal data buffers called as *Buffer Cache*. *Note that, Buffer Cache contains the data of recently used disk blocks*.

If you refer back to the figure of *Kernel* on page 15, when *Kernel* reads data from a file on disk, it first keeps the read data on *Buffer Cache* and then reads from that. If the data is already present on buffer, *Kernel* does not access the disk but directly reads data from the buffer. If the data is not on the buffer, then & then only *Kernel* reads data from disk. Similarly, when *Kernel* wants to write data to disk, first it writes the same data to buffer (***Why ?*** This is provision that if *Kernel* again wants to read the written data, it does not have to do disk operations. Instead it can immediately read it back from the buffer) and then decides, whether the data is really writable of it is just a transient data which is going to be soon overwritten and hence not needed to write. If the data is immediate writable, it writes it on the disk and if not, it puts in buffer marked as "delay-write", so it can be overwritten or if not overwritten, then can be written to disk after some time (hence the word "delay").

So by this way, using *Buffer Cache*, *Kernel* tries to reduce read and write disk operations of the head of the disk and speeds up the system performance.

Now an interesting question may arise, "*Adding a Buffer Cache stage, we add something new between file sub-system and block devices. Then how we can say that process of data access becomes faster ?*"

The answer lies in the answer of another question, "*What is Buffer Cache exactly means ?*". <u>*Buffer Cache* is nothing but some reserved space in *RAM*</u>. As buffer is *RAM*, the process is faster.

During system installation, the system administrator gives possible number of buffers to installer and installer program gives this number to *Operating System*. Obviously the number of buffers will depend upon the *RAM* hardware installed on the machine.

So again mention that *Buffer Cache* is software product, though dependant on size of *RAM* hardware, it is created by a program in *Operating System*. *Buffer Cache* is not a single entity.

Actually buffer consists of 2 parts……

a) Buffer Header and
b) Memory Area.

Buffer depends upon 2 *Kernel* data structures……

a) Free List Of Buffers
b) Hash Queue Of Buffers.

[Note : This does not mean that there are 2 copies of buffers on these two data structures. There is only one copy of buffer, but it can be accessed by 2 ways:-

1) By *Linked List* method
2) By *Hash Queue* method.

★ *Some Important Properties Of Buffer*

! During System initialization *Kernel* allocates memory spaces for given number of buffers. The allocated memory obviously depends on the *RAM* hardware.

! There is *one-to-one mapping* of buffer header & buffer's data arrays (i.e. the *Memory Area* of Buffer) which are the two parts of buffer. The word "*one-to-one mapping*" means, that <u>*One buffer header corresponds to one data array*</u>.

! We said that *Kernel* reads data from disk into buffer. Obviously buffer must have the address number of disk from which it reads data. This address is not the physical location of data on disk. (Because if it is, then a disastrous situation may occur when reading *1 GB* file into *32 MB RAM*). The address is a logical block number (which we described on page 18) which is corresponding to physical address of data on the disk.

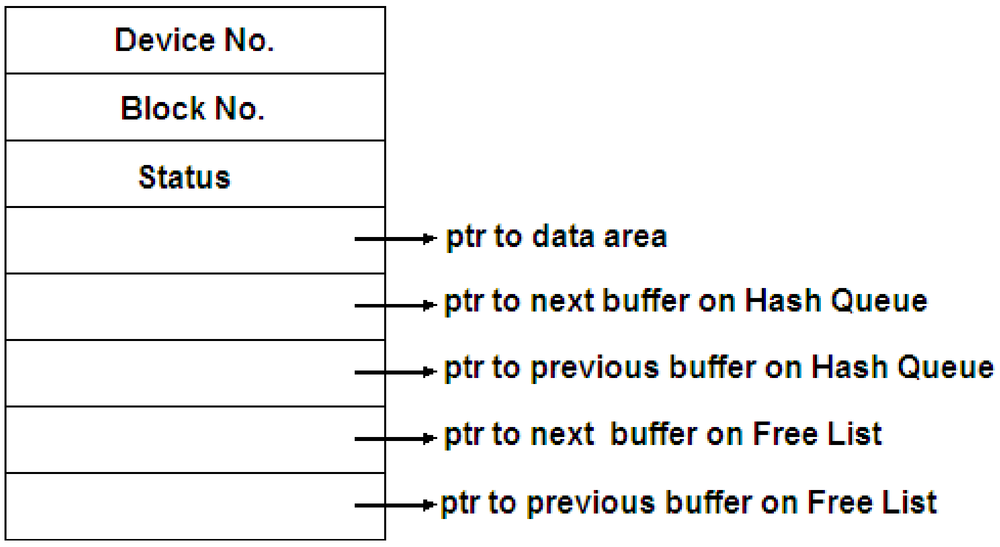
! Thus in other words we can say that, "<u>*Buffer is a memory copy of a disk block*</u>". So if a data block is actually (physically) from 1002nd to 1400th block, then it may be present in data block of *File System* from 502nd block to 900th logical block. (See the data block area of *File System* in figure on page 19). Now, when *Kernel* reads data from the disk, it will not read from 1002 to 1400, but reads from 502 to 900.

! The mapping of read data from disk is <u>not permanent</u> (if permanent memory will get wasted until system shuts down), <u>but temporary</u>. Means when the same buffer is used to read another block of data from the disk, the old contents are wiped off.

! <u>Now a very important point, one block of data can be mapped into one buffer at a time</u>. This is mercy, because, suppose one disk block is read into two buffers ***A*** & ***B***. *Kernel* shows data for buffer ***A*** to user. User looks at it and decides to do some another task by minimizing the contents of ***A*** buffer. Then after some time user re-opens buffer ***A*** for writing. But meanwhile, accidentally, as contents of ***A*** & ***B*** are same, *Kernel* puts buffer ***B*** and allow some another process to read buffer ***A*** which is in demand of same data. User writes to buffer ***B*** (as user has no knowledge of ***A*** & ***B***) and saves changes to ***B***. Now actually when the data is to be written on disk, it is supposed that data of buffer ***A*** should be written to disk. So *Kernel* writes data from ***A*** to disk. Now think when user re-opens the file to see the changes made by him. They are lost! This happens because ***A*** was not used by user to write changes.

<u>To avoid condition like above</u>, these must be one buffer for one data block.

The Buffer Header


Device No.
Block No.
Status
ptr to data area
ptr to next buffer on Hash Queue
ptr to previous buffer on Hash Queue
ptr to next buffer on Free List
ptr to previous buffer on Free List

The Structure Of Buffer Header

✷ The device number field of buffer header contains logical device number given by *File System* to a corresponding physical device unit number.

✷ The block number field of buffer header contains block number of the data block on disk. This field is unique for each buffer.

✷ The status field of buffer header contains the current status of the buffer. Thus this field may contain one or <u>combination of</u> following condition (means can be "<u>or</u>" ed) ……….

- ! The buffer is currently in "***Locked***" status (or say *busy* status)
- ! The buffer is currently in "***Un-locked***" status (or say free status)
- ! The buffer is currently in "***Valid***" status. Means, now it contains some valid data.
- ! The buffer is currently in "***Delayed - Write***" status. Means the *Kernel* must write data in this buffer to the disk, before assigning it as free to some other process.
- ! The buffer is currently in "***Reading***" status. Means the buffer at present reading data from the disk.
- ! The buffer is currently in "***Writing***" status. Means the buffer is at present writing data to the disk.
- ! The buffer is currently in "***Reserved***" status. Means some process has already reserved this buffer and is waiting for this buffer to become free (which is currently in use by another process).

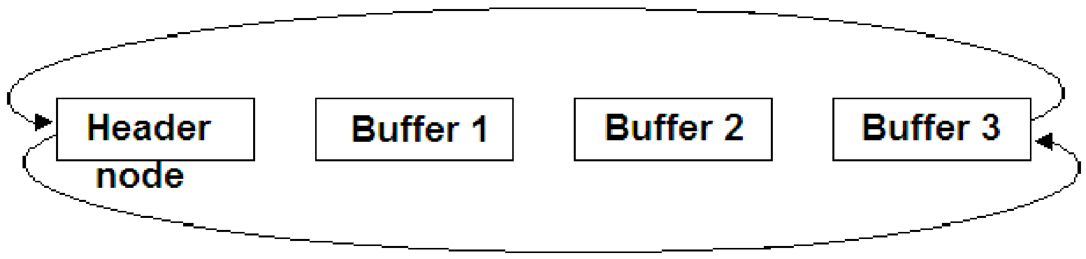
- ✷ The pointer to data area field is a memory block whose size must be as big as the size of one disk block.
- ✷ The pointer to next buffer on *Hash Queue* field is pointer to the next (i.e. next in sequence) buffer present on the *Hash Queue* of buffers.
- ✷ The pointer to previous buffer on *Hash Queue* field is the pointer to the previous (i.e. previous in sequence) buffer present on the *Hash Queue* of buffers.
- ✷ The pointer to next buffer on *Free List* field is the pointer to next (i.e. next in sequence) buffer present on *Free List* of buffers.
- ✷ The pointer to previous buffer on *Free List* field is the pointer to previous (i.e. previous in sequence) buffer present on *Free List* of buffers.

Out of above 5 pointers, the pointers for buffer on *Hash Queue* and the pointers for buffer on *Free List* are used by "***Buffer Allocation Algorithm***" to maintain the overall structure of buffer pool.

Memory Area Of Buffer

The *Memory Area of Buffer* is the part which contains the actual data. This part is not different from the buffer header. But it is actually the 4th field of the buffer header. This field, as explained before is at least as large as one disk block and it contains the readable or writable data of the buffer. This field is identified by buffer header as a pointer which is used in buffer allocation algorithm to read or write the data.

Free List Of Buffers



Kernel maintains a *Free List* of buffers in memory. This list is a "*doubly linked list implemented as circular list*".

✷ ***Note that :*** following discussion, the term header node, means header node of the list (i.e. not the buffer header). And the term buffer header, means part of buffer (page 38) not the header node of list.

When the system is booted this list is created in memory with a header node indicating its "beginning". The header node is then followed by list of buffers kept in a specific order. The order or sequence of buffers is maintained by "***Least Recently Used Algorithm***" (i.e. LRU algorithm). The number of buffers is as given by system administrator on the time of system installation or the number of buffers depends on number of disk blocks present in *File System*. Thus one buffer is allocated for one logical disk block. The meaning of *LRU* algorithm means when *Kernel* allocates a buffer for a disk block, the same buffer will not be used for another block, until all other remaining buffers in the list are used more recently. Now when its time that *Kernel* wants a free buffer, it takes that buffer which is nearer to header node of the list. But when *Kernel* does not want a free buffer but wants a specific buffer (having data in it), then it can take it from middle of the list. *Kernel* identifies this specific buffer (i.e. the buffer having data) because it matches the block number (that it wants to read or write) in the *File System* with the block number on the buffer's header (see figure on page 38). When match is perfectly found identification completes.

In the both conditions, i.e. whether *Kernel* chooses a free buffer or a specific buffer, it pulls out that buffer from the list. After its removal, list joins the cut ends as if removed buffer was never in the list. [Recall the mechanism of removal of a node from any position in a linked list from the topic of data structure].

After removing the buffer, now *Kernel* is ready to use that removal buffer. If buffer is a "free", (empty or having such data that can be overwritten) then it usually writes on it. If buffer is "filled" (means having valid data on it), then it either only reads the data or make changes after changes are made, changes are written on it. This reading or changing depends on the user, whether he wants just to read or wants to make changes the data.

After completion of working with the buffer, its now time to put the buffer again on the list. [Recall the mechanism of addition of a node at any position in a *Linked List* from the topic of Data Structure]. Now a question is that*, where to put this buffer in the list?* There are 2 possibilities…

1. If *Kernel*'s mechanism of reading/writing the buffer has an error, then such buffer is attached to the beginning of list [i.e. just after the header node].

2. I f *Kernel*'s mechanism of reading/writing the buffer is successful (i.e. no errors), then such buffer is attached to the end of list.

Means in any circumstances, the buffer is never put in the middle of the list.

Now suppose our list contains *buffer 1* to say buffer ----, *buffer 49* contains some valid data. As *Kernel* starts removing buffer (mainly from beginning and rarely from middle), the *buffer 49* starts getting closer & closer to header node and *Kernel* puts its used buffer to the end of the list (as seen above), so we can say that "*buffers closer to the header node are not recently used by Kernel, while the buffers at the end of the list are more recently used*". This is the way by which *Kernel* maintains *LRU* algorithm of the list.

! ***A buffer can be present on Free List, only if it is un-locked*****.**

Hash Queues Of Buffers

After completing the discussion of "*How Kernel maintains the buffer pool*", now come back to the searching mechanism of *Kernel* for a specific buffer in the list.

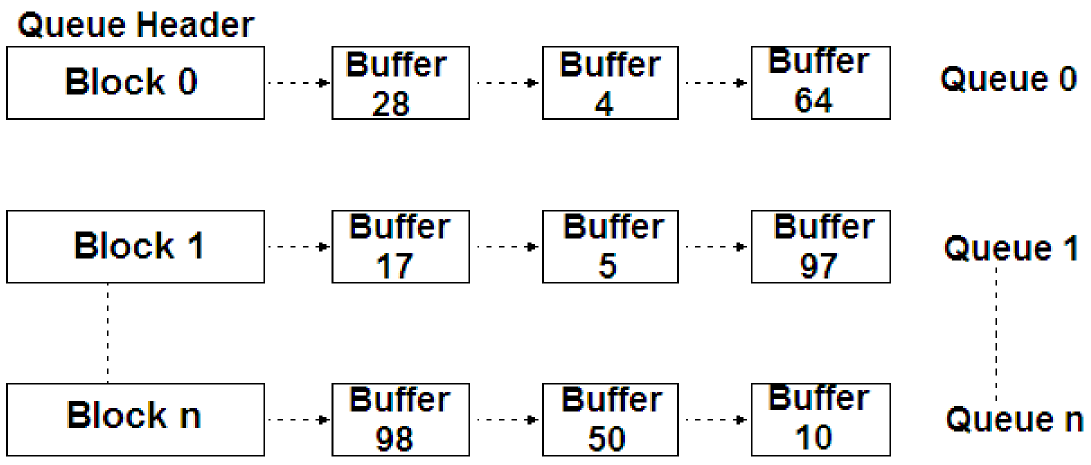
In the middle of page 40, we had a very short discussion of "*How Kernel matches the disk block numbers from the File System with that on buffer header of the buffer*".

This topic needs some detailed explanation, because number of buffer on the list is depending on system administrator's value or on the disk size. So there may be very large number of buffer on list. Once created, numbers of buffers are never become smaller. Because though *Kernel* removes a buffer (where we may think that list becomes shorter), this is for time being, because *Kernel*, later, is going to put that buffer again on the list. So list size is going to be constant always.

Then, while searching for a specific block on list, *Kernel* may have to scan all the list entirely, which may be very time consuming.

To avoid this time consuming task, *Kernel* maintains another data structure called as ***Hash Queue***. So when *Kernel* accesses a disk block, it searches for a buffer with matching appropriate "*device - block number* combination" that on the disk block with that on the buffer header. Instead of searching the entire buffer list, it organizes buffers into separate "queues" which are hashed on a "*hash function*" of device number & block number. The *Kernel* then links buffers on *Hash Queue* with that on list. *Splitting of all buffers into which number of Hash Queues?* is depending upon system administrator who gives desired number of *Hash Queue*s during the process of system installation.

Figure : On next page.



Here "***n***" is the number that is given by system administrator at the time of system installation. So if there are 1000 buffers (i.e. because there are 1000 blocks on hard disk) and administrator gives "*n*" as 20, then obviously there will be 50 buffers in one *Hash Queue*. (Because 1000 buffers are divided into 20 *Hash Queues*) on each queue. Which buffer should present on which queue is dependant on "*hash function*" used by *Kernel*. So if *Kernel* wants to access 353rd block, then it first executes *hash function*, gets the number of queue to search and also gets the position of required buffer on that queue. E.g. ! Suppose "*hash function*" is division of required block with number of buffer in each queue. So 353/50 gives division 7 & remainder 3, so the required block is on 8th queue (because 7 is completed and there is remainder, so 7+1=8) and the location of 353rd is 3rd (i.e. remainder) in the 8th queue. This *hash function*'s example is just for understanding. Because it assumes that you have just 1 device having 1000 blocks. In practice, this "*hash function*" is not so easy because *UNIX* supports many devices, many disks and disk size may be very big.

This is the reason for which only block number is not enough for *hash function*. It also requires device number too.

It is obvious that every buffer must be present on *Hash Queue*, but which buffer should present on which queue, depends upon "*hash function*". As we already stated that "there must be one & only one buffer for one block", the buffer for that block is also present on one & only one *Hash Queue*, and will present only once.

Now as there are two data structures, i.e. *Free List* & *Hash Queue*, a single buffer may be present both on *Free List* and on *Hash Queue*. Thus *Kernel* has got 2 opportunities to search for a buffer. If buffer is free (i.e. empty/garbage), then device number/block number are not significant. Then *Kernel* just takes any free buffer nearer to the header node of the list. But when *Kernel* wants a specific buffer, then device number & block number are very significant. Here *Kernel* uses *hash function* to get number of queue & location at that buffer on queue, then enters into appropriate queue and then finally removes it from the queue. The "*buffer putting*" mechanism is also similar to above mechanism.

In next section, we are going to see, by which actual mechanism *Kernel* gets the buffer. We also are going to see *what conditions Kernel should check, before accessing a buffer* and so on……

Scenarios For Getting a Buffer

If we again look at the figure of *Kernel* on page 15, we found that *Buffer Cache* is manipulated by the functions located in "*File Subsystem*". If a process wants to read data from a file, then *Kernel* first looks in *File System* that which *File System* contains the required file. Then it looks in "data block" area of *File System* to see which block contains the respective data. Now *Kernel* gets the device number & block number of the respective data. Then *Kernel* searches "the *Hash Queue* of buffer" in memory for this "device-number & block-number" combination (obviously by using a *hash function*) and when it finds the match, it picks up that buffer. But if this specific "*device number - block number*" combination is not in *Hash Queue*, *Kernel* leaves the *Hash Queue* and starts looking in *Free List*, from which it takes a free buffer from the list.

Similarly, when *Kernel* wants to write a file to disk, it first searches the *Hash Queue* for respective combination of "*device number - block number*", if it finds here, then it picks up that buffer and further uses it for writing. If it does not found the "device number & block number" combination buffer in *Hash Queue*, it leaves the *Hash Queue* and picks up a free buffer from *Free List* of buffers.

Above algorithm is used by *Kernel* for reading or writing of the disk blocks. As all above process requires the most important 2 stepped approach….

1 Get the respective block from the system.
2 Get the respective buffer (either from *Hash Queue* or from *Free List*) corresponding to respective block.

The *UNIX* terminology calls this algorithm as ***getblk*** – which allocates requested buffer from the buffer pool.

The algorithm ***gelblk*** is like follows….

```
getblk algorithm
input : 1) File System number
        2) Block number
{/* Algorithm begins */
while (buffer not found)
        {/* while begins */
        if (block is in Hash Queue)
                {/* main if begins */
                /* Scenario 5 */
                if (buffer is busy)
                        {
                        sleep (event : until specific buffer becomes free) ;
                        continue ; /* back to while loop */
                        }
                        /* Scenario 1 */
                        if (buffer is not busy or now becomes free and hence in
                                                                     Free List)
                        {
                        mark the buffer as busy ;
                        now remove the buffer from Free List ;
                return (buffer) ;
                }
            } /* main if ends */
            else /* block is not present in Hash Queue */
            { /* main else begins */
            /* Scenario 4 */
            if (as well as buffer is also not on Free List or Free List is
                                                              empty )
            {
            sleep (event : until any buffer become free ) ;
            continue ; /* back to while loop */
            }
            After becoming buffer free, thus now Free List, then remove it
            from Free List ;
            /* Scenario 3 */
            if (buffer is in Free List but marked as "delayed-write")
            {
            Asynchronous writing of such buffer to disk ;
            continue ; /* back to while loop */
            }
            /* Scenario 2 */
            remove buffer from old Hash Queue ;
            put buffer on new Hash Queue ;
            return (buffer) ; /* common step */
            } /* main else complete */
        } /* while complete */
    } /* Algorithm complete */
    output : a locked buffer which can be now used.
```

If we look at this algorithm, we found that there are 5 scenarios (or situations) for getting/retrieval of a buffer.

1) If the required "*device number - block number*" combination buffer is present on *Hash Queue*, also buffer is not busy, and also it is on "*Free List*".
2) If the required "*device number - block number*" combination buffer is not on *Hash Queue*, hence *Kernel* chooses a free buffer from "*Free List*".
3) If the required "*device number - block number*" combination buffer is not on *Hash Queue*, hence *Kernel* looks in *Free List*, there is a usable buffer but it is marked as "*delayed-write*". So *Kernel* must first write this buffer to disk.
4) If the required "*device number - block number*" combination buffer is neither on *Hash Queue* nor on *Free List* (i.e. list is empty).
5) If the required "device number – block number" combination buffer is present on *Hash Queue*, but currently it is busy (as it is being used by some other process).

Now we will discuss these scenarios in more detail….

1) Scenario 1 : When *Kernel* needs a buffer for a block, in advance it know the device number & block number combination (***How ?*** that will be discussed in next chapter). It uses these two numbers as input parameters and calls *getblk ( )* function. Then as it is not having the required buffer right now, obviously it enters into the *while* loop. In *while* loop it calls *hash function* gets the number of proper *Hash Queue* number from this function and starts searching that queue for such a buffer whose header has matching "device number – block number" combination. If buffer is found, then it checks whether the buffer is busy or free. If the buffer is free, it marks that buffer as "busy" and then switches to "*Free List*". As buffer is free it is also present on the *Free List*. (Note that : All buffers are always present on one of the *Hash Queues*, but *Free List* only have "free buffers" and "delayed write" buffers. So "busy" buffer is never present on *Free List*). Now it removes the selected buffer from the *Free List*. The step of marking the buffer as "busy" is important for preventing other processes from accessing it when they are in *Kernel Mode*. (It is obvious that accessing buffer is a *Kernel*'s task So whichever process wants to access buffer is obviously doing so in *Kernel*'s context). Now the return statement in function returns this buffer to *Kernel*. (See algorithm for *scenario 1* on page 43 from line 16 to 22).

Though description of *scenario 1* ends here, it is interesting to see what happens to this buffer after *Kernel* gets it!

After getting the buffer, *Kernel* may read data from the block into the buffer (if user just wants to read) or *Kernel* may write data to this buffer (if user wants to write or to read & write together) and then either writes buffered data directly to disk or may mark such buffer for "delayed-write" and then put it on *Free List*. As seen in first scenario, no other process can access the same buffer same time. Thus there may or may not be many processes slept waiting for this buffer to become free. Thus when *Kernel* finishes with this buffer for current process, it calls the algorithm *brelease ( )* and using this algorithm it releases (i.e. frees or unlocks) the buffer and immediately wakes up all slept processes. (Note that : among the slept processes some may require "this" specific buffer or some may require any free buffer if *Free List* is empty). Now there occurs fighting between all awaken processes for the buffer and highest priority process wins and takes the buffer by the same method as in *getblk ( )*. So obviously after taking this buffer, the "*awaken highest priority process*" again marks it as "busy" and again all remaining processes go to sleep. This goes on & goes on but be sure that at some time, every waiting/slept process has sure chance to get the buffer.

Now coming back to *brelease ( )*, its algorithm is something like follows :

```
01 :    brelease algorithm
02 :    input : locked (or busy) buffer
03 :    { /* algorithm begins */
04 :            wake up all processes which are waiting for "any" buffer ;
05 :            wake up all processes which are waiting for "this" buffer ;
06 :            raise processor execution level to block interrupts ;
07 :            if (buffer contents are valid and buffer is not old i.e. delayed write or of
                                                                          different block)
08 :                    put the buffer at the end of the Free List ;
09 :            else /* if any error occurs */
10 :                    put the buffer at beginning of Free List ;
11 :            low the processor execution level to allow interrupts ;
12 :            unlock or free or release the buffer ;
13 :    }
14 : output : none
```

The algorithm is quite easy to understand. Most of the steps already discussed above. Important thing in this algorithm is "*raising & lowering of process execution level*". Not all processes are synchronous (means another process waits until the completion of one process), but some may be asynchronous (means another process starts its execution before the completion of prior process). If such asynchronous process causes "*disk interrupt*" or calls *getblk ( )* while some another process is already using *Hash Queue* or *Free List* of buffers, then the buffer pointers may get corrupted. To avoid this and thus to maintain the consistency of buffer pool, *Kernel* raises processor execution level to block disk interrupts. And when *Kernel* finishes manipulating with buffer pool, it again lowers the processor execution level.

2) Scenario 2 : While searching the *Hash Queue* for required buffer or free buffer, if it fails to find the buffer on *Hash Queue*, *Kernel* returns to *Free List* and removes first buffer from the *Free List*. Now this buffer is not for required "device-block" combination, but it is actually for some other "device-block" combination and thus present both on *Free List* and *Hash Queue*.

A s *Kernel* takes such a buffer which is not for required "device-block" combination, but for some other "device-block" combination, after removing it from list, *Kernel* wipes of "device-block" number on its buffer header and replaces it with the new required "device-block" combination. Then *Kernel* puts this buffer on correct *Hash Queue*. Now *Kernel* uses this buffer as described before but keeps no record of its original "device-block" combination & data which it already wiped off. So if a process wants that "old" block (which *Kernel* wiped off) buffer, the process can not find it any where and thus allocates a new buffer (i.e. free) from the list as described in *scenario 1*.

The method of releasing buffer is same as explained before on page 45 by *brelease ( )* algorithm. (The code for *scenario 2* is on page 44, from line 40 to 43).

3) Scenario 3 : Like scenario 2, here too, *Kernel* can not found required buffer in *Hash Queue*. So *Kernel* returns to *Free List*. The required buffer on *Free List* is not of "free" nature but of "delayed-write" nature. In such cases *Kernel* removes it from *Free List* and then immediately writes it contents to disk. This process of *Kernel*'s writing of "delayed-write" buffer to disk, is of <u>asynchronous</u> nature. Now while doing this asynchronous write, at the same time it tires to allocate another buffer from the *Free List*. When asynchronous writing completes, *Kernel* releases the buffer and keeps this buffer at the head of *Free List*, (not at the end of *Free List*). ***Why ?*** we saw that in *scenario 3 Kernel* is doing 2 tasks simultaneously. One, it is writing "delayed write" buffer to disk asynchronously and Second, it is trying to allocated a buffer form the *Free List*. Now suppose it can not found the buffer in *Free List*, then it can use that buffer which it keeps at the head. As the buffer is used just now for delayed writing, we might thing that this is most recently used buffer and hence we also expect that it should be kept at the "tail" of the *Free List*. But this is not the story! Though buffer is used "right now", it was claimed for used in the past, hence marked as "old". Hence it should not be considered "most recent used" and hence it is kept at the head and not at tail of *Free List* (line 34-35).

After completion of "asynchronous writing" of "delayed-write" buffer, it is marked as "old" and kept at the beginning of *Free List*.

4) ***Scenario 4 :*** Like scenario 2, here also *Kernel* can not found required buffer on *Hash Queue*. And it returns to *Free List*. But in this scenario, now *Free List* is also empty. So there is no way for the *Kernel* to get the buffer. In such condition *Kernel* tells the process (which requires this buffer) that it can not give buffer and so the process goes to sleep on the event of *waiting for the buffer to become free*. It is obvious that absence of buffer on both, i.e. on *Hash Queue* and on *Free List*, indicates that some other process is already using that buffer and hence marked as "Locked". So when this "buffer using" process frees or unlocks the buffer by *brelease ( )* algorithm, then & then only the previous process (which went to sleep) can access that released buffer. So the "buffer using" process unlocks or frees or releases the buffer and awakes all processes which went to sleep.

Now we may expect that our "buffer needing" process immediately grab the released buffer. But this is not what we expect. Because some other process may have claimed for the same buffer before our "buffer needing" process. So on priority level our "buffer needing" process is on "low priority". So though the buffer is released, our "buffer needing" process may or may not get it immediately as we expect. So to ensure that the buffer is released and no other process is claimed on that, *Kernel* again goes to *Hash Queue* and continues the search again. To verify that, buffer is really free, the continue statement is *scenario 4* on line 30 does this job. (The code for *scenario 4*, is from line 26 to 33).

5) Scenario 5 : The code for scenario 5 is from lines 09 to 14. Though the code looks small & simple, it actually is quiet complicated.

To explain this scenario we will take example of 3 processes i.e. *process A*, *process B*, *process C*.

Suppose right now, *process A* is looking for buffer by *getblk ( )* algorithm. So *Kernel* is right now working for *process A*.

Process A, searches buffer, found it on *Hash Queue*, utilizes it as per its need, but before releasing it, some I/O is going on and thus went to sleep, keeping the buffer "locked" (i.e. busy).

As *process A* sleeps, *Kernel* now schedules *process B*, which is also waiting for the same buffer. So *process B* uses *getblk ( )*, searches *Hash Queue*, found the buffer. But as the buffer is "locked", *process B* went to sleep on the event *of waiting for the buffer to become free*.

Eventually, when I/O completes, *process A* wakes up, releases the buffer and wakes up all other processes which are waiting for this buffer to become free.

So *process B* wakes up. Hence again we may expect that, *process B* was already claimed for this buffer and thus should immediately use the released buffer. But no! the *process B* must again to go top of the loop and starts searching right from the beginning. The *continue* statement on line 13 does this job. This extra work for *process B* is made for 2 reasons :

A) to verify that, buffer is really free. Because some time it may happen, that a process may be scheduled before *process B* and process c may went to sleep (due to some other reason like I/O interrupts) keeping the buffer locked. So *process B* must verify that the buffer is really free.

B) Similarly, if process c scheduled before *process B* and during operation, process c may allocate the buffer for some another block (as seen in scenario 2). So when *process B* wakes up, found the buffer, but now this buffer is "wrong buffer" (means not the actual buffer for which *process B* was waiting). If is uses immediately it will corrupt the data. So this is important for *process B*, not only to check whether the buffer is really free, but also to check whether the buffer really contains "*device - block number* combination" for which it was sleeping.

So continuing the loop goes on until *process B* gets the valid buffer with required "device-block" combination or a free buffer which it can allocated to required "device-block" combination as explained in *scenario 2*.

In short, *Kernel* guarantees you to give buffer to a process, now or later. This guarantee is given on the basis that *Kernel* always allocates buffer during the execution of process's system calls and frees it before returning from the executing process. Exception is "*mount*" system call, which keeps the buffer locked (not up till its termination) up till the next "*umount*" call.

Reading And Writing Disk Blocks

As we know from the previous discussion about buffer, the buffer allocation (*getblk ( )*) and release (*brelease ( )*) is mainly done for file I/O in the *file subsystem*. So the major target of buffer algorithm is for reading & writing of disk blocks.

! ***Reading Of Disk Blocks :*** To read disk block, a process switches to *Kernel Mode*, then *Kernel* uses *getblk ( )* algorithm to search for corresponding buffer in buffer pool. If it is in the pool with valid data, then *Kernel* immediately returns the buffer to the process without actual reading the block from the disk. Buffer is there, but data on buffer is not valid. But if buffer with valid data is not in the pool, then *Kernel* calls disk driver and requests it for file I/O and the process goes to sleep until I/O gets completed. The disk driver then notifies the actual disk hardware that it wants to read the data. Now the disk hardware i.e. head starts moving to read the data and after reading transmits the read data to the buffer. Finally the disk controller hardware makes interrupt to microprocessor telling it that I/O is complete. Also this disk controller's interrupt awakes the slept process. Now the buffer is not empty, it is filled with data and now the awaken process got the data. When process's task with the data (such as viewing), user closes the process during termination, process tells *Kernel* release the buffer and *Kernel* uses *brelease ( )* to release the buffer so that other processes can use it. [Here *Kernel* releases the buffer].

Here we may expect that above process completes the reading of a file and this disk controller will sit quietly until next request. But doing so will reduce the system performance. To make the reading of a file more faster, *file subsystem* anticipates the need of possible next block to read. (*How this anticipation of possible next block is done?* This will be seen in 5th chapter). This is particularly important when a process wants to read the file sequentially (i.e. either by "down arrow key" or by "page-down key" or by "downward mouse scrolling").

In such cases *Kernel* requests hardware controller for reading of second block asynchronously means reading of first block is still going on i.e. yet not completed). This process keeps second block in its own buffer (Note that : getting of second buffer is same as the getting of first buffer. Means *getblk ( )* is again called, but now asynchronously). So that if process (i.e. user) looks for next data, the data will be in buffer (i.e. in memory) well in advance. Obviously as reading of first block is still going on, process goes to sleep waiting for first block's I/O completion. As soon as user scrolls next, first block's I/O completes, process awakens, returns the buffer of first block and does not care about the I/O completion of second block (which is going on in background asynchronously). As process does not care about I/O completion of second block, it is the duty of hardware controller to do further action. So when second block's I/O completes, it interrupts the microprocessor. Here the interrupt handler automatically recognizes that this I/O was asynchronous and itself releases the buffer (by *brelease ( )*) [Here hardware controller released the buffer] which allows buffer to get kept in the *Free List* & in *Hash Queue*. This step is important, because if buffer remains locked no other processes could access it. now the process which was looking the file sequentially, looks for this second buffer, (which is now in *Free List* & in *Hash Queue* due to the work of interrupt handler) it easily finds that buffer in buffer pool and reading goes on.

So this "asynchronous" mechanism for reading second buffer is of vital importance for the system performance. The heart of this mechanism is the *Anticipation Capacity Of Kernel To Read The Next* Buffer though process has not demanded for it. this is done by "***Block Read Ahead***" algorithm. In short it is termed as *breada ( )*. The algorithm is as follows….

```
01 :    breada algorithm /* reading of block & of ahead block */
02 :    input : 1) The block number to read immediately (first block)
03 :            2) The block number to read asynchronously (second block)
04 :    {
05 :            if (first block is not in buffer pool)
06 :            {
07 :            get buffer for first block by getblk ( ) ;
08 :            if (buffer data is not valid)
09 :                    initiate hardware controller for disk reading ;
10 :            }
11 :            if (second block is not in buffer pool)
12 :            {
13 :            get buffer for second block by getblk ( ) ; /* asynchronous */
14 :            if (buffer data is valid)
15 :                    release buffer by brelease ( ) ;
16 :            else /* means buffer data is not valid */
17 :                    initials hardware controller for disk reading ;
18 :            }
19 :            if (first block is already in buffer pool)
20 :            {
21 :            read first block by bread ( ) ;
22 :            return (buffer) ;
23 :            }
24 :            sleep (event : first buffer contains valid data, means first block is still
                          not released i.e. it is valid, when reading completes it will
                                                                         become invalid) ;
25 :            return (buffer) ;
26 :    }
27 : output : Buffer containing data for first block.
```

The algorithm is not much difficult to understand.

! The algorithm has 2 parameters, the present block number (i.e. the first block) which process wants to read write now. And the second parameter is block number that process would read after first block.

! First *Kernel* looks for first block's buffer in buffer pool, if it is in pool, then it enters in the algorithm from line 19 to 23. Where it reads the block's buffer by *bread ( )* algorithm (we will see it shortly) and after completion of reading returns the buffer.

! If first block is not in buffer pool, then, it uses *getblk ( )* algorithm and go through respective scenarios as explained before. Ultimately get the buffer and looks for data in it. if it gets the buffer but the data on it is invalid (scenario 2 – "old" block), then *Kernel* starts disk reading by telling hardware controller.

! Now if second block's buffer is not in the buffer pool, then *Kernel* again starts searching for buffer by *getblk ( )* algorithm, but this time asynchronously (as reading of first block is going on in background). Ultimately gets the buffer for second block. Looks for valid data on the buffer. If data is valid releases the buffer by *brelease ( )*. Here a question may arise *why not to be read?* For second buffer *Kernel* just releases it so that it will be kept in buffer pool. It does not read it by *bread ( )* because this is not the process wants now! Process is now reading the first block and the second block is *Kernel*'s anticipation for the user that may read the second block next time, so it is better to keep buffer of second block in memory, well in advance!

! If the data on the second block's buffer is invalid, then it proceeds on explained for "invalid data on first buffer". So it initiates disk reading by telling hardware controller.

! See there is no return statement for second buffer's if block. It shows that *Kernel*'s duty is just to get the second block's buffer and release it to keep in buffer pool.

! There is one situation for which there is no "if" block in algorithm. That situation is *if (second block's buffer is already present in buffer pool)*. This situation is deliberately omitted, because task of keeping 2nd buffer in pool is already done and as the target is reading of first block and not the reading of second block. So if above situation arises, *Kernel* skips to *sleep ( )* statement (line 24). As reading of valid data on first block is going on, *Kernel* can not read second block's buffer and hence process sleeps waiting for completion of first block's buffer reading.

So it is clear from above algorithm, that *Kernel* when looking for first block's buffer, it actually looks for two buffers, reads first buffer and keeps second buffer in buffer pool for possible next reading.

The algorithm for reading any block's buffer is very easy….

```
bread algorithm /* reading of buffer of a block */
input : the required block number to read
{
        get buffer for required block by getblk ( ) ;
        if (buffer data is valid)
                return (buffer);
        initiate hardware controller for disk reading ;
        sleep (event : till hardware controller reading) ;
        return (buffer) ;
}
output : buffer containing valid data
```

Everything from above algorithm is already explained in previous discussion.

Writing Of Disk Buffer

When *Kernel* wants to write buffer contents to disk, it first informs the disk driver (i.e. the hardware controller) that right now I am having such a buffer whose contents should be written to disk. So now the disk driver stimulates the head of the disk and head goes to desired block (actual physical block) corresponding to the buffer and starts writing buffer contents on to that block.

If the process of writing is synchronous, the process which calls above operation, goes to sleep on the event of waiting for completion of this disk I/O. When I/ O completes, process gets awakened which then releases the buffer [Here the process releases the buffer]. If the process of writing is asynchronous, then *Kernel* starts disk writing as above but does not wait for this I/O completion. After I/O completion *Kernel* itself releases the buffer [Here *Kernel* releases the buffer].

But as explained before is scenarios of *getblk ( )*, there is a situation of "delayed write", in which *Kernel* does not write the data immediately to the disk. Instead it marks the buffer as "old" and keeps it at beginning of *Free List*. Now *Kernel* hopes that some process will use this buffer. But note that, though the buffer is kept at head of the *Free List*, *Kernel* does not allow any process to allocate this buffer for some other block. So such "old" marked buffer remains for the same block whose *device - block number* combination is on it. If such a time comes, that this is the only possible buffer for a process, then & then only *Kernel* first writes its data to the disk (as explained before) and then releases the buffer so the respective can use it. Now the process, if needs, can allocate this buffer to some another block by wiping its "device-block" number combination and putting new combination on it. It is clear from this explanation that, *Kernel* waits "as long as possible" for writing "delayed write" buffer to disk.

Here it may seem initial similarity between "asynchronous write" & "delayed write". But actually there is a big conceptual difference between the two. In "asynchronous write" operation, *Kernel* starts disk operations (i.e. disk driver) immediately but does not wait for is completion. While in "delayed write" operation *Kernel* waits as long as possible for actual disk operation and ultimately by *scenario 3*, it uses *getblk ( )*, marks the buffer "old" and then when time comes, it writes it contents to disk asynchronously. When I/O completes, now its hardware controller that release the buffer and put it of the beginning *Free List* as it is "old".

Concluding, we can say that, putting of buffer the buffer pool is done by *brelease ( )* algorithm which his used by *Kernel* in 3 situations…

a) When a process calls *brelease ( )* directly or

b) When a process calls *Kernel* (in *Kernel Mode*) and then *Kernel* itself calls *brelease ( )*

c) Or When hardware controller (i.e. the interrupt handler) calls *Kernel* and then *Kernel* executes *brelease ( )*.

Out of these 3 situations, situation a) is usually synchronous. Situation c) is usually asynchronous and situation b) is conditional, means may be of synchronous or of asynchronous type as per the need. This is because situation b) is handled by *Kernel* itself.

We might be interested in the algorithm of "writing buffer to the disk". The algorithm is termed as ***bwrite*** and is quite similar to ***bread***. The algorithm is as follows….

```
bwrite algorithm /* Writing of buffer contents to disk */
input : the buffer that we wanted to write
{
        Initiate disk hardware controller ;
        if (I/O is synchronous)
        {
                sleep (event : until I/O completes)
                release the buffer by brelease ( ) ;
        }
        if (I/O is asynchronous & buffer is "delayed write")
        mark the buffer as "old" & put it at beginning of Free List ;
}
output : no output i.e. no return value.
```

1) The use of buffer allows uniform disk access. Means the same buffering mechanism is there for any type of data, which may be data from a file or data from an *Inode* or data from a *Super Block*. This makes the system design more modular & simpler. *Kernel* even does not need to know whether I/O is for reading or writing. Instead it will just copy data from disk to buffer or from buffer to disk.
2) In other operating systems, for good I/O performance, data needs to be aligned. Means say alignment on two-byte boundary or say four-byte boundary in memory. So programmer must keep aware themselves about thid data alignment while doing I/O. <u>But in *UNIX*</u>, as everything is file and as every file is just stream of bytes, there is no need of such data alignment. *Kernel* does this alignment internally. The *Buffer Cache* makes *UNIX* system to portable even on these machines which have strict address space alignment in memory. So the programmers need not to worry about special memory alignment.
3) As explained before, algorithms like "***block read ahead***" can reduce the number of hardware operations and thus mark ably decrease the system response time. Also unnecessary disk writing is avoid by "delayed write" mechanism.
4) As number of disk blocks & number of buffers are corresponding in nature, if two processes simultaneously try to access same buffer, the algorithm prevent them doing this and thus prevent data corruption.

1) Large capacity disk will obviously have large number of blocks and thus obviously have large number of buffer in memory. So this leads to some memory constraints. Thus less amount of memory remain for executing the processes. Thus processes now start swapping or paging of memory, which may further reduce the system performance.
2) A s *Kernel*, by using buffer algorithms, decides which data to be written immediately and which data to be written "delayed", a user giving write command (i.e. saving the file), is <u>never sure</u> about the time when the data will be actually get written.
3) As we had seen, during read operation a data copy is made in memory (i.e. in buffer) and then written to the disk. So for this is a very good approach to reduce the disk operations (i.e. movement of head on disk surface). And this gives fantastic fast performance for small amount of data. <u>But when large amount of data</u> is to be read or is to be written, then large amount of memory gets occupied due to large data copies in buffers, which reduces the amount of free memory for running processes, slowing down the system performance.