THE BUFFER CACHE:

The kernel maintains files on mass storage devices such as disks, and it allows processes to store new information or to recall previously stored information.

When a process wants to access data from a file, the kernel brings the data into main memory where the process can examine it, alter it, and request that the data be saved in the file system again.

The kernel could read and write directly to and from the disk for all file system accesses, but system response time and throughput would be poor because of the slow disk transfer rate.

The kernel therefore attempts to minimize the frequency of disk access by keeping a pool of internal data buffers, called the ***buffer cache***, which contains the data in recently used disk blocks.

The position of the buffer cache module in the kernel architecture is in between the file subsystem and (block) device drivers.

When reading data from the disk, the kernel attempts to read from the buffer cache. If the data is already in the cache, the kernel does not have to read from the disk.

If the data is not in the cache, the kernel reads the data from the disk and caches it, using an algorithm that tries to save as much good data in the cache as possible.

Similarly, data being written to disk is cached so that it will be there if the kernel later tries to read it.

The kernel also attempts to minimize the frequency of disk write operations by determining whether the data must really be stored on disk or whether it is transient data that will soon be overwritten.

Higher-level kernel algorithms instruct the buffer cache module to pre-cache data or to delay-write data to maximize the caching effect.

BUFFER HEADERS:

During system initialization, the kernel allocates space for a number of buffers, configurable according to memory size and system performance constraints.

A buffer consists of two parts: a memory array that contains data from the disk and a buffer header that identifies the buffer.

The data in a buffer corresponds to the data in a logical disk block on a file system, and the kernel identifies the buffer contents by examining identifier fields in the buffer header.

The buffer is the in-memory copy of the disk block; the contents of the disk block map into the buffer, but the mapping is temporary until the kernel decides to map another disk block into the buffer. A disk block can never map into more than one buffer at a time.

If two buffers were to contain data for one disk block, the kernel would not know which buffer contained the current data and could write incorrect data back to disk.

For example, suppose a disk block maps into two buffers, A and B. If the kernel writes data first into buffer A and then into buffer B, the disk block should contain the contents of buffer B if all write operations completely fill the buffer.

However, if the kernel reverses the order when it copies the buffers to disk, the disk block will contain incorrect data.

The buffer header (Figure 2.1) contains a device number field and a block number field specifies the file system and the block number of the data on disk and uniquely identify the buffer.

Note: The buffer cache is a software structure that should not be confused with hardware caches that speed memory references.

The device number is the logical file system number, not a physical device (disk) unit number.

The buffer header also contains a pointer to a data array for the buffer, whose size must be at least as big as the size of a disk block, and a status field that summarizes the current status of the buffer.

The status of a buffer is a combination of the following conditions:

- The buffer is currently locked
- The buffer contains valid data
- The kernel must write the buffer contents to disk before reassigning the buffer; this condition is known as "delayed-write"
- The kernel is currently reading or writing the contents of the buffer to disk
- A process is currently waiting for the buffer to become free

The buffer header also contains two sets of pointers, used by the buffer allocation algorithms to maintain the overall structure of the buffer pool.

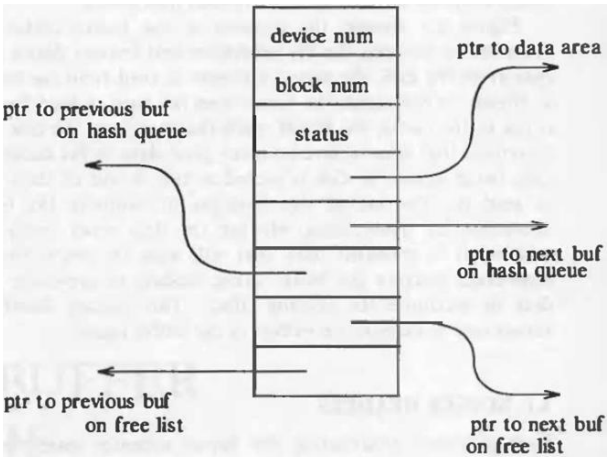

Figure 2.1 Buffer Header

STRUCTURE OF THE BUFFER POOL:

The kernel caches data in the buffer pool according to a least recently used algorithm: after it allocates a buffer to a disk block, it cannot use the buffer for another block until all other buffers have been used more recently.

The kernel maintains a free list of buffers that preserves the least recently used order. The free list is a doubly linked circular list of buffers with a dummy buffer header that marks its beginning and end (Figure 2.2).

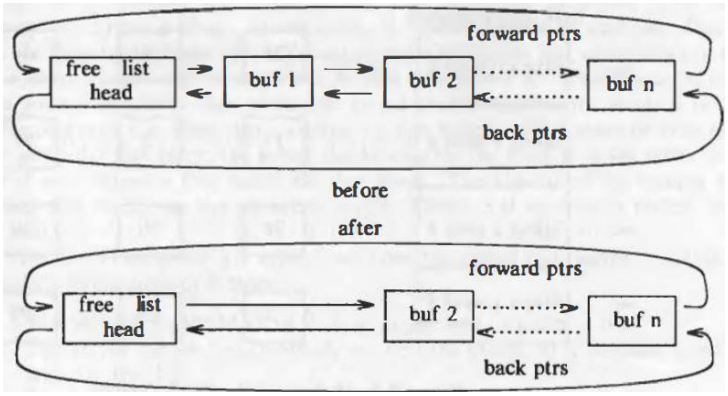

Figure 2.2 Free Lists of Buffers

Every buffer is put on the free list when the system is booted. The kernel takes a buffer from the head of the free list when it wants any free buffer, but it can take a buffer from the middle of the free list if it identifies a particular block in the buffer pool.

In both cases, it removes the buffer from the free list. When the kernel returns a buffer to the buffer pool, it usually attaches the buffer to the tail of the free list, occasionally to the head of the free list (for error cases), but never to the middle.

As the kernel removes buffers from the free list, a buffer with valid data moves closer and closer to head of the free list (Figure 2.2). Hence, the buffers that are closer to the head of the free list have not been used as recently as those that are further from the head of the free list.

When the kernel accesses a disk block, it searches for a buffer with the appropriate device-block number combination. Rather than search the entire buffer pool, it organizes the buffers into separate queues, hashed as a function of the device and block number.

The kernel links the buffers on a hash queue into a circular, doubly linked list, similar to the structure of the free list. The number of buffers on a hash queue varies during the lifetime of the system.

The kernel must use a hashing function that distributes the buffers uniformly across the set of hash queues, yet the hash function must be simple so that performance does not suffer. System administrators configure the number of hash queues when generating the operating system.

Figure 2.3 shows buffers on their hash queues: the headers of the hash queues are on the left side of the figure, and the squares on each row are buffers on a hash queue. Thus, squares marked 28, 4, and 64 represent buffers on the hash queue for "blkno 0 mod 4" (block number 0 modulo 4).

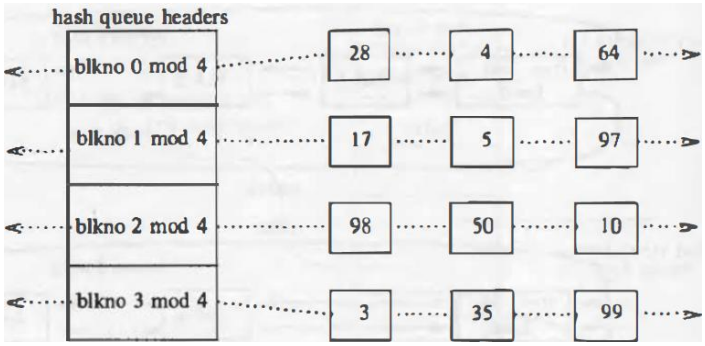

Figure 2.3 Buffers on Hash Queues

The dotted lines between the buffers represent the forward and back pointers for the hash queue. Each buffer always exists on a hash queue, but there is no significance to its position on the queue.

As stated above, no two buffers may simultaneously contain the contents of the same disk block; therefore, every disk block in the buffer pool exists on one and only one hash queue and only once on that queue. However, a buffer may be on the free list as well if its status is free.

SCENARIOS FOR RETRIEVAL OF A BUFFER:

High-level kernel algorithms in the file subsystem invoke the algorithms for managing the buffer cache. The high-level algorithms determine the logical device number and block number that they wish to access when they attempt to retrieve a block.

For example, if a process wants to read data from a file, the kernel determines which file system contains the file and which block in the file system contains the data.

When about to read data from a particular disk block, the kernel checks whether the block is in the buffer pool and, if it is not there, assigns it a free buffer.

When about to write data to a particular disk block, the kernel checks whether the block is in the buffer pool, and if not, assigns a free buffer for that block.

The algorithms for reading and writing disk blocks use the algorithm **getblk** to allocate buffers from the pool.

```
algorithm getblk
input:  file system number
        block number
output: locked buffer that can now be used for block
{
        while (buffer not found)
        {
                if (block in hash queue)
                {
                        if (buffer busy)  /* scenario 5 */
                        {
                                sleep (event buffer becomes free);
                                continue; /* back to while loop */
                        }
                        mark buffer busy; /* scenario 1 */
                        remove buffer from free list;
                        return buffer;
                }
```

```
                else /* block not on hash queue */
                {
                        if (there are no buffers on free list) /* scenario 4 */
                        {
                                sleep (event any buffer becomes free);
                                continue; /* back to while loop */
                        }
                        remove buffer from free list;
                        if (buffer marked for delayed write) { /*  scenario 3 */
                        asynchronous write buffer to disk;
                        continue; /* back to while loop */
                        }
                        /* scenario 2 -- found a free buffer */
                        remove buffer from old hash queue;
                        put buffer onto new hash queue;
                        return buffer;
                }
        }
}
```


There are five typical scenarios the kernel may follow in getblk to allocate a buffer for a disk block:

1. The kernel finds the block on its hash queue, and its buffer is free.
2. The kernel cannot find the block on the hash queue, so it allocates a buffer from the free list.
3. The kernel cannot find the block on the hash queue and, in attempting to allocate a buffer from the free list, finds a buffer on the free list that has been marked "delayed write". The kernel must write the "delayed write" buffer to disk and allocate another buffer.
4. The kernel cannot find the block on the hash queue, and the free list of buffers is empty.
5. The kernel finds the block on the hash queue, but its buffer is currently busy.

When searching for a block in the buffer pool by its device-block number combination, the kernel finds the hash queue that should contain the block.

It searches the hash queue, following the linked list of buffers until (in the first scenario) it finds the buffer whose device and block number match those for which it is searching. The kernel checks that the buffer is free and, if so, marks the buffer "busy" so that other processes cannot access it.

The kernel then removes the buffer from the free list, because a buffer cannot be both busy and on the free list. If other processes attempt to access the block while the buffer is busy, they sleep until the buffer is released.

Figure 2.4 depicts the first scenario, where the kernel searches for block 4 on the hash queue marked "blkno 0 mod 4." Finding the buffer, the kernel removes it from the free list, leaving blocks 5 and 28 adjacent on the free list.

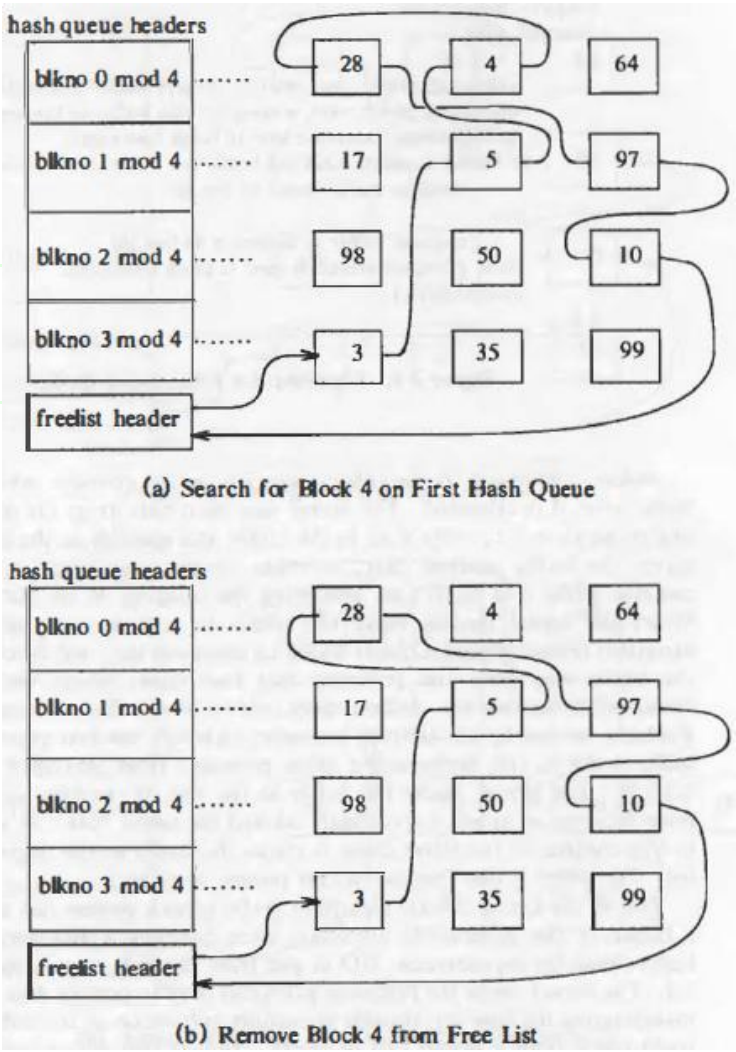

Figure 2.4 Scenario 1 in Finding a Buffer: Buffer on Hash Queue

In the second scenario in algorithm getblk, the kernel searches the hash queue that should contain the block but fails to find it there. Since the block cannot be on another hash queue because it cannot "hash" elsewhere, it is not in the buffer cache.

So the kernel removes the first buffer from the free list instead; that buffer had been allocated to another disk block and is also on a hash queue.

If the buffer has not been marked for a delayed write, the kernel marks the buffer busy, removes it from the hash queue where it currently resides, reassigns the buffer header's device and block number to that of the disk block for which the process is searching, and places the buffer on the correct hash queue.

The kernel uses the buffer but has no record that the buffer formerly contained data for another disk block. A process searching for the old disk block will not find it in the pool and will have to allocate a new buffer for it from the free list.

In the third scenario in algorithm getblk, the kernel also has to allocate a buffer from the free list. However, it discovers that the buffer it removes from the free list has been marked for "delayed write," so it must write the contents of the buffer to disk before using the buffer.

The kernel starts an asynchronous write to disk and tries to allocate another buffer from the free list. When the asynchronous write completes, the kernel releases the buffer and places it at the head of the free list. The buffer had started at the end of the free list and had traveled to the head of the free list.

READING AND WRITING DISK BLOCKS:

To read a disk block (**Figure 2.5**), a process uses algorithm getblk to search for it in the buffer cache. If it is in the cache, the kernel can return it immediately without physically reading the block from the disk.

```
algorithm bread    /* block read */
input:  file system block number
output: buffer containing data
{
        get buffer for block (algorithm getblk);
        if (buffer data valid)
                return buffer;
        initiate disk read;
        sleep(event disk read complete);
        return(buffer);
}
```

Figure 2.5 Algorithm for Reading Disk Block

If it is not in the cache, the kernel calls the disk driver to "schedule" a read request and goes to sleep awaiting the event that the I/O completes. The disk driver notifies the disk controller hardware that it wants to read data, and the disk controller later transmits the data to the buffer.

Finally, the disk controller interrupts the processor when the I/O is complete, and the disk interrupt handler awakens the sleeping process; the contents of the disk block are now in the buffer. The modules that requested the particular block now have the data; when they no longer need the buffer they release it so that other processes can access it.

The algorithm for writing the contents of a buffer to a disk block is similar (**Figure 2.6**). The kernel informs the disk driver that it has a buffer whose contents should be output, and the disk driver schedules the block for I/O.

```
algorithm bwrite    /* block write */
input:  buffer
output: none
{
        initiate disk write;
        if (I/O synchronous)
        {
                sleep(event I/O complete);
                release buffer (algorithm brelse);
        }
        else if (buffer marked for delayed write)
                mark buffer to put at head of free list;
}
```

Figure 2.6 Algorithm for Writing a Disk Block

If the write is synchronous, the calling process goes to sleep awaiting I/O completion and releases the buffer when it awakens. If the write is asynchronous, the kernel starts the disk write but does not wait for the write to complete. The kernel will release the buffer when the I/O completes.

A delayed write is different from an asynchronous write. When doing an asynchronous write, the kernel starts the disk operation immediately but does not wait for its completion. For a "delayed write," the kernel puts off the physical write to disk as long as possible; then, recalling the third scenario in algorithm getblk, it marks the buffer "old" and writes the block to disk asynchronously.

ADVANTAGES AND DISADVANTAGES OF THE BUFFER CACHE:

Use of the buffer cache has several advantages and, unfortunately, some disadvantages:

- The use of buffers allows uniform disk access, because the kernel does not need to know the reason for the I/O. Instead, it copies data to and from buffers, regardless of whether the data is part of a file, an inode, or a super block.
- The system places no data alignment restrictions on user processes doing I/O, because the kernel aligns data internally.
    - Hardware implementations frequently require a particular alignment of data for disk I/0, such as aligning the data on a two-byte boundary or on a four-byte boundary in memory. Without a buffer mechanism, programmers would have to make sure that their data buffers were correctly aligned.
- Use of the buffer cache can reduce the amount of disk traffic, thereby increasing overall system throughput and decreasing response time.
    - Processes reading from the file system may find data blocks in the cache and avoid the need for disk I/O.
    - The kernel frequently uses "delayed write" to avoid unnecessary disk writes, leaving the block in the buffer cache and hoping for a cache hit on the block.
- The buffer algorithms help insure file system integrity, because they maintain a common, single image of disk blocks contained in the cache. If two processes simultaneously attempt to manipulate one disk block, the buffer algorithms (getblk for example) serialize their access, preventing data corruption.
- Reduction of disk traffic is important for good throughput and response time, but the cache strategy also introduces several disadvantages.
    - Since the kernel does not immediately write data to the disk for a delayed write, the system is vulnerable to crashes that leave disk data in an incorrect state.
- Use of the buffer cache requires an extra data copy when reading and writing to and from user processes.
    - A process writing data copies the data into the kernel, and the kernel copies the data to disk; a process reading data has the data read from disk into the kernel and from the kernel to the user process.
        - When transmitting large amounts of data, the extra copy slows down performance, but when transmitting small amounts of data, it improves performance because the kernel buffers the data (using algorithms getblk and delayed write) until it is economical to transmit to or from the disk.