19CS2106S
Lecture Notes
File System and Buffer cache Allocation Algorithms
Session – 4

Session No:4

List of Topics:	Learning Outcomes:	Questions Answered from This session :
1. File System 2. Buffer Cache Allocation Algorithms :Getblk, Brelse	1. Understand the basic File System 2. Understand the need for Buffer Cache 3. Implement Block Allocation algorithms :Bget, Brelse	1. What a file system is : 2. Need for Buffer cache 3. Illustrate getblk, brelse.

Session plan:

Session Outcome: Understand and Explore the Design of File Subsystem and Buffer Cache Algorithms.

Time (min)	Topic	BTL	Teaching - Learning Methods	Active Learning Methods
10	An Overview of the File Subsystem, File System Layout	2		
20	Buffer Cache allocation algorithms: getblk, brelse.	3	Lecture & Discussion	
10	Xv6 functions: bget, brelse. Design and Implementation of bio.c	3	Lecture & Discussion	LTC
5	Conclusion & Summary	-		

References

1. Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing. CH 2.2.1 Page No [22 – 24, 44, 46]
2. Frans Kaashoek, Robert Morris, and Russ Cox, The xv6 source code booklet (draft) (revision 11). https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf Sheet No [44]

File System:

A file system is a logical collection of files on a partition or disk. A partition is a container for information and can span an entire hard drive if desired.

Your hard drive can have various partitions which usually contain only one file system, such as one file system housing the **/file system** or another containing the **/home file system**.

One file system per partition allows for the logical maintenance and management of differing file systems.

Everything in Unix is considered to be a file, including physical devices such as DVD-ROMs, USB devices, and floppy drives.

Directory Structure:

Unix uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.

A Unix file system is a collection of files and directories that has the following properties –

- It has a root directory (**/**) that contains other files and directories.
- Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an **inode**.
- By convention, the root directory has an **inode** number of **2** and the **lost+found** directory has an **inode** number of **3**. Inode numbers **0** and **1** are not used. File inode numbers can be seen by specifying the **-i option** to **ls command**.
- It is self-contained. There are no dependencies between one filesystem and another.

File System Layout:

Boot Block	Super Block	Inode List	Data Blocks

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.

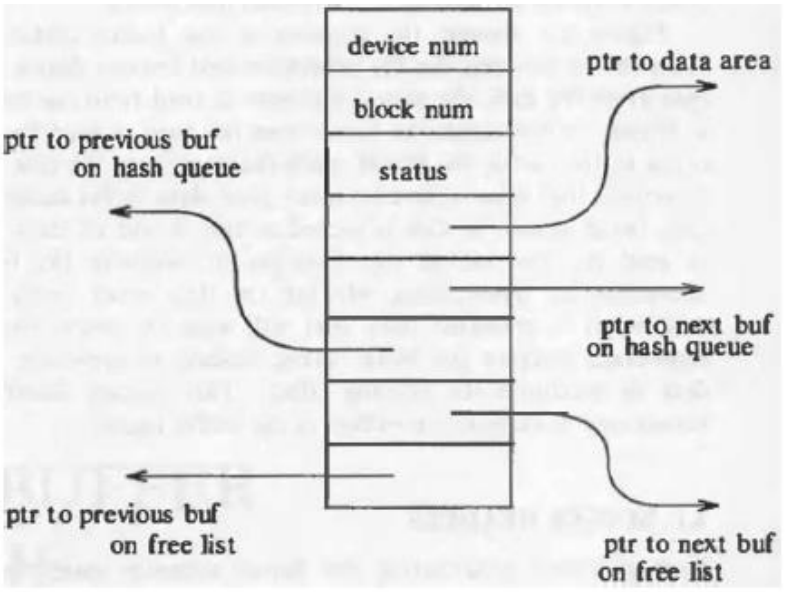
If two buffers were to contain data for one disk block, the kernel would not know which buffer contained the current data and could write incorrect data back to disk. For example, suppose a disk block maps into two buffers, A and B. If the kernel writes data first into buffer A and then into buffer B, 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 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.

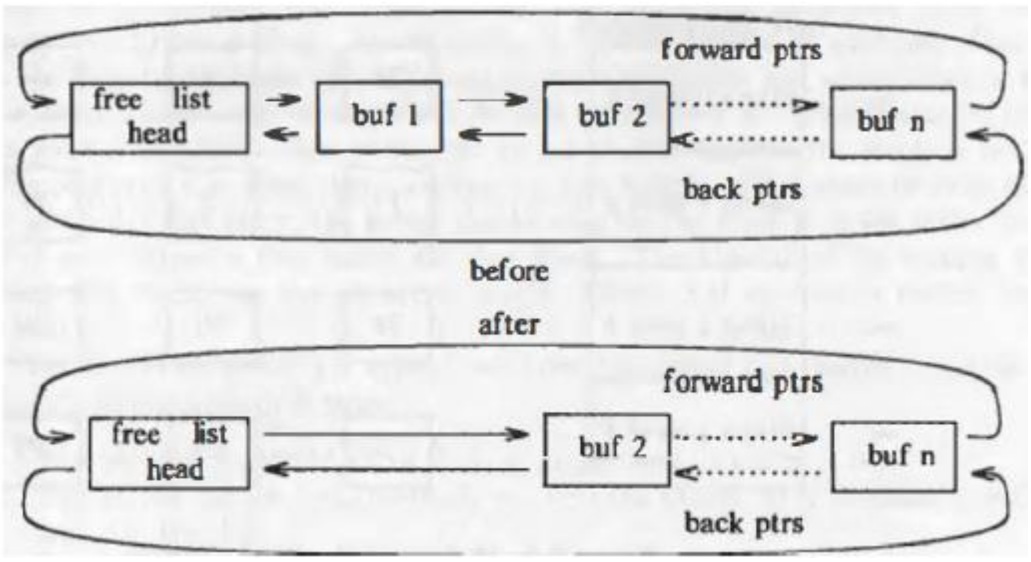
- The status of a buffer is a combination of the following conditions:
- The buffer is currently locked
- The buffer contains valid data
- The kernel must write the buffer contents to disk before reassigning the buffer; this condition is known as "delayed-write"
- The kernel is currently reading or writing 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.


device num
block num
status
ptr to data area
ptr to previous buf on hash queue
ptr to next buf on hash queue
ptr to previous buf on free list
ptr to next buf on free list

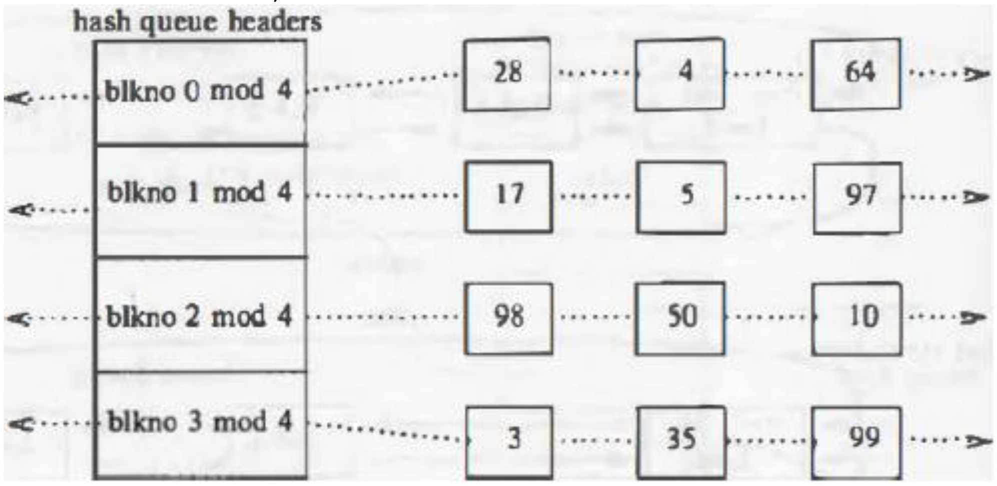
STRUCTURE OF THE BUFFER POOL:

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



Free List 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 .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 Below 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).



Buffer 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:

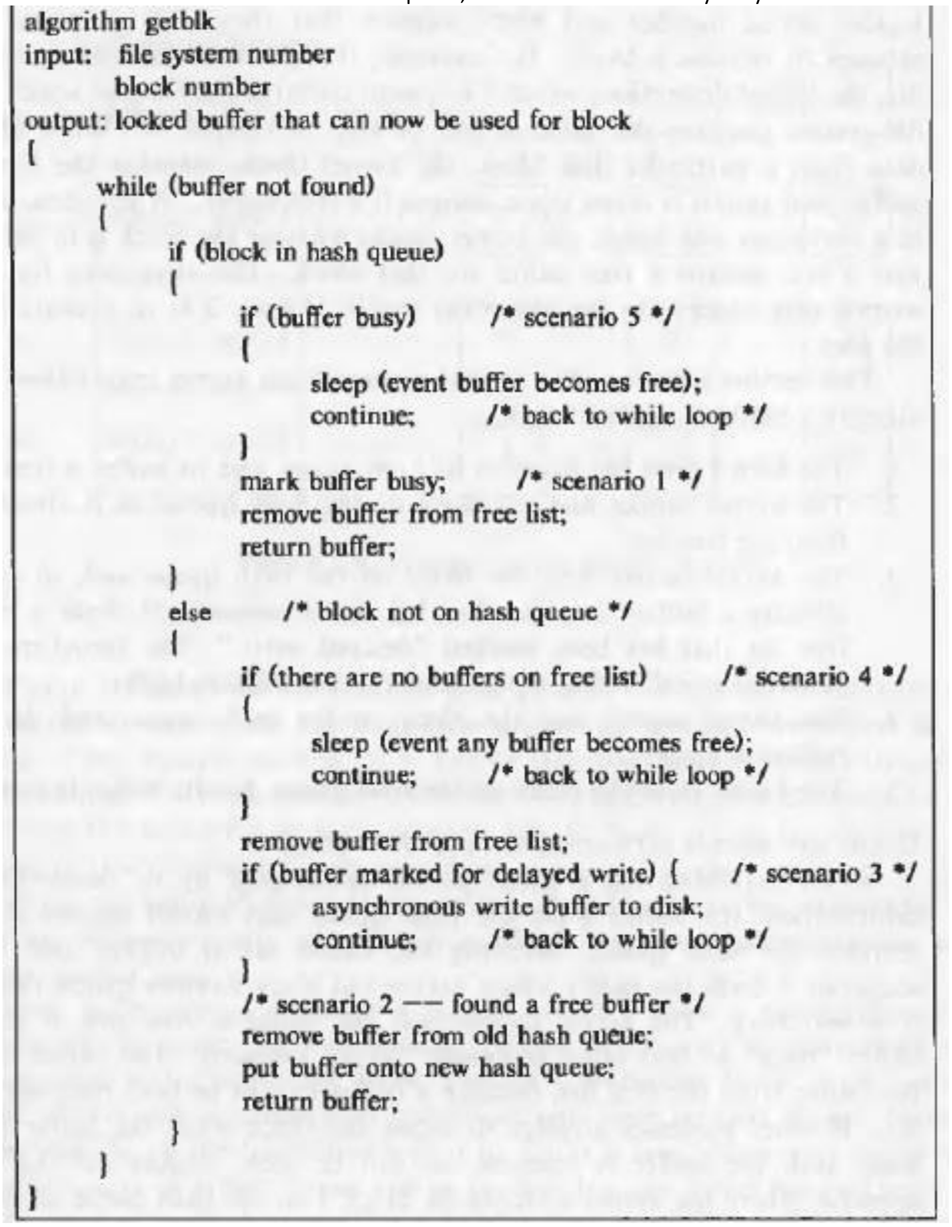
High-level kernel algorithms in the file subsystem invoke the algorithms for managing the buffer cache. The high-level algorithms determine the logical device number and block number that they wish to access when they attempt to retrieve a block.

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.

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



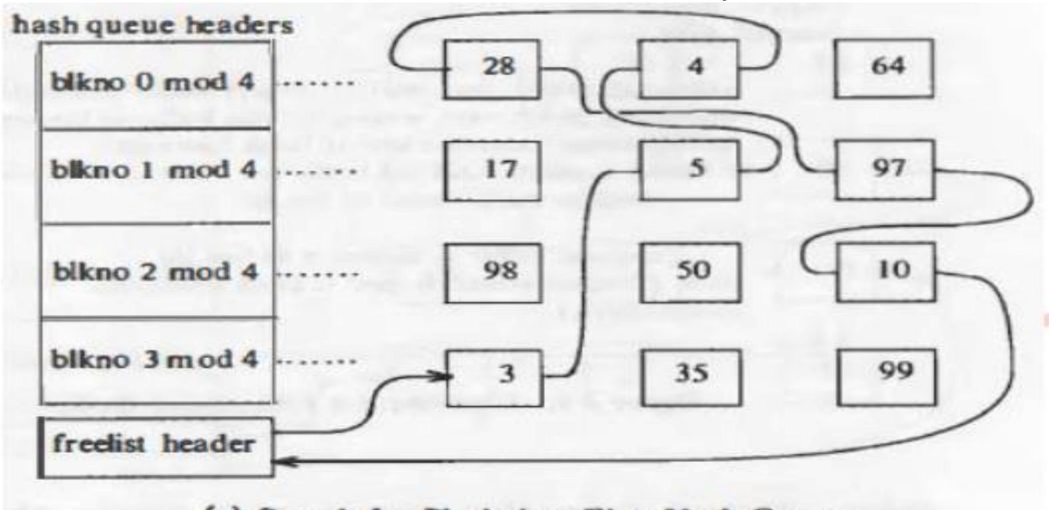
```
algorithm getblk
input:  file system number
        block number
output: locked buffer that can now be used for block
{
    while (buffer not found)
    {
        if (block in hash queue)
        {
            if (buffer busy)        /* scenario 5 */
            {
                sleep (event buffer 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;
        }
    }
}
```

Algorithm :Getblk

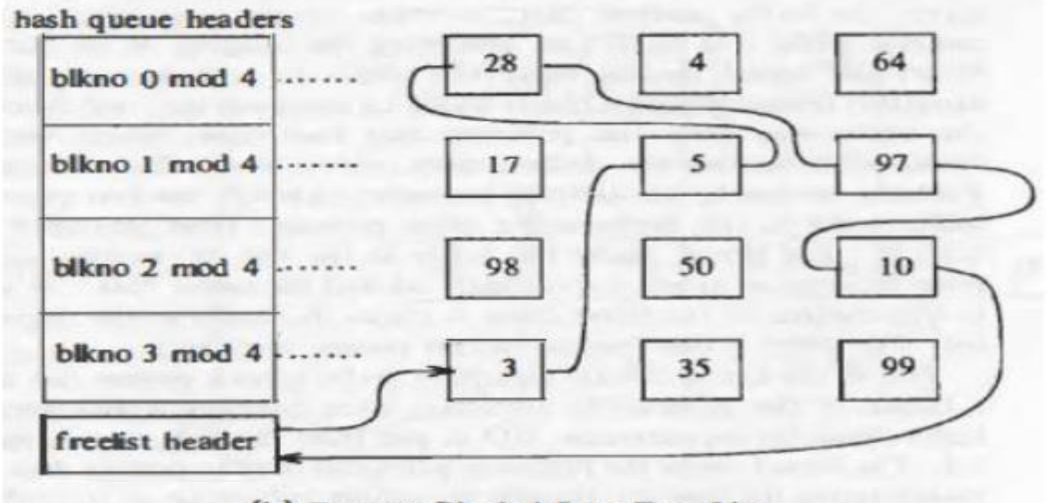
When searching for a block in the buffer pool by its device-block number combination, the kernel finds the hash queue that should contain the block.

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.



(a) Search for Block 4 on First Hash Queue



(b) Remove Block 4 from Free List

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.

```
// Look through buffer cache for block on device dev.
// If not found, allocate a buffer.
// In either case, return locked buffer.
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);

  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached; recycle an unused buffer.
  // Even if refcnt==0, B_DIRTY indicates a buffer is in use
  // because log.c has modified it but not yet committed it.
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->flags = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```