19CS2106S
Lecture Notes
Buffer cache Algorithms & Logging
Session – 5

Session No:5

List of Topics:	Learning Outcomes:	Questions Answered from This session :
1. Buffer Cache Algorithms :bread,bwrite & breada 2. Looging	1. Implement Disk Block Read and write algorithms :bread and bwrite 2. Implement log_write algorithm	1. How disk blocks are read and written 2. Need for logging

SESSION NUMBER: 05

Session Outcome: 1. understand and explore the Design of Buffer Cache allocation algorithms.
2. Apply Log design and Logging.

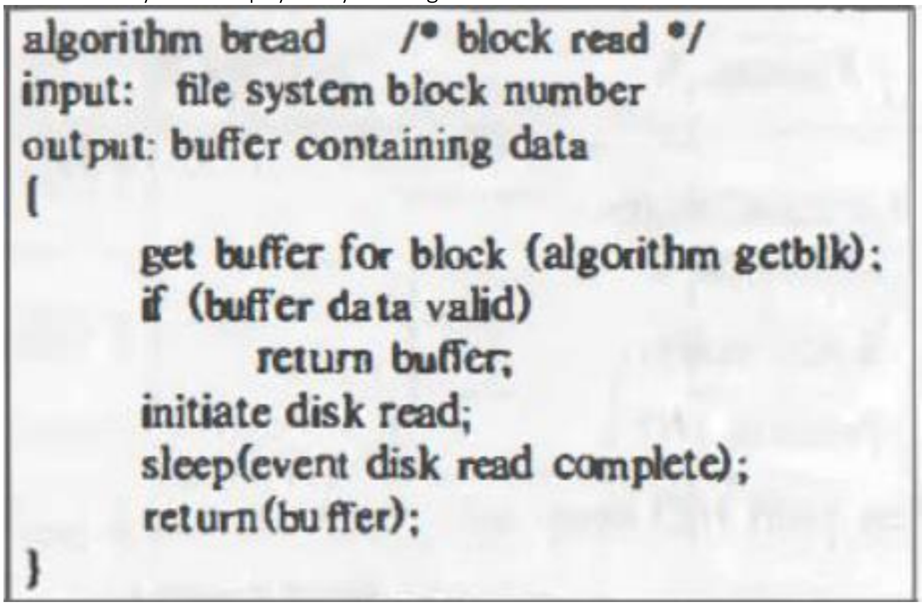
Time (min)	Topic	BTL	Teaching - Learning Methods	Active Learning Methods
10	Buffer Cache allocation algorithms: bread, breada, bwrite	2	Lecture & Discussion	
10	Log design, Logging	3	Lecture & Discussion	
20	Xv6 functions: bread, bwrite. Design and Implementation of log.c	3	Lecture & Discussion	LTC
5	Conclusion & Summary	-		

References

1. Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing. CH 3, [3.3, 3.4] Page No [54,55,56],
2. Russ Cox, Frans Kaashoek, Robert Morris, xv6: a simple, Unix-like teaching operating system", Revision https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf CH 6 Page No [79,80],
3. 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 T3 Sheet No [44,45],[47]

READING AND WRITING DISK BLOCKS:

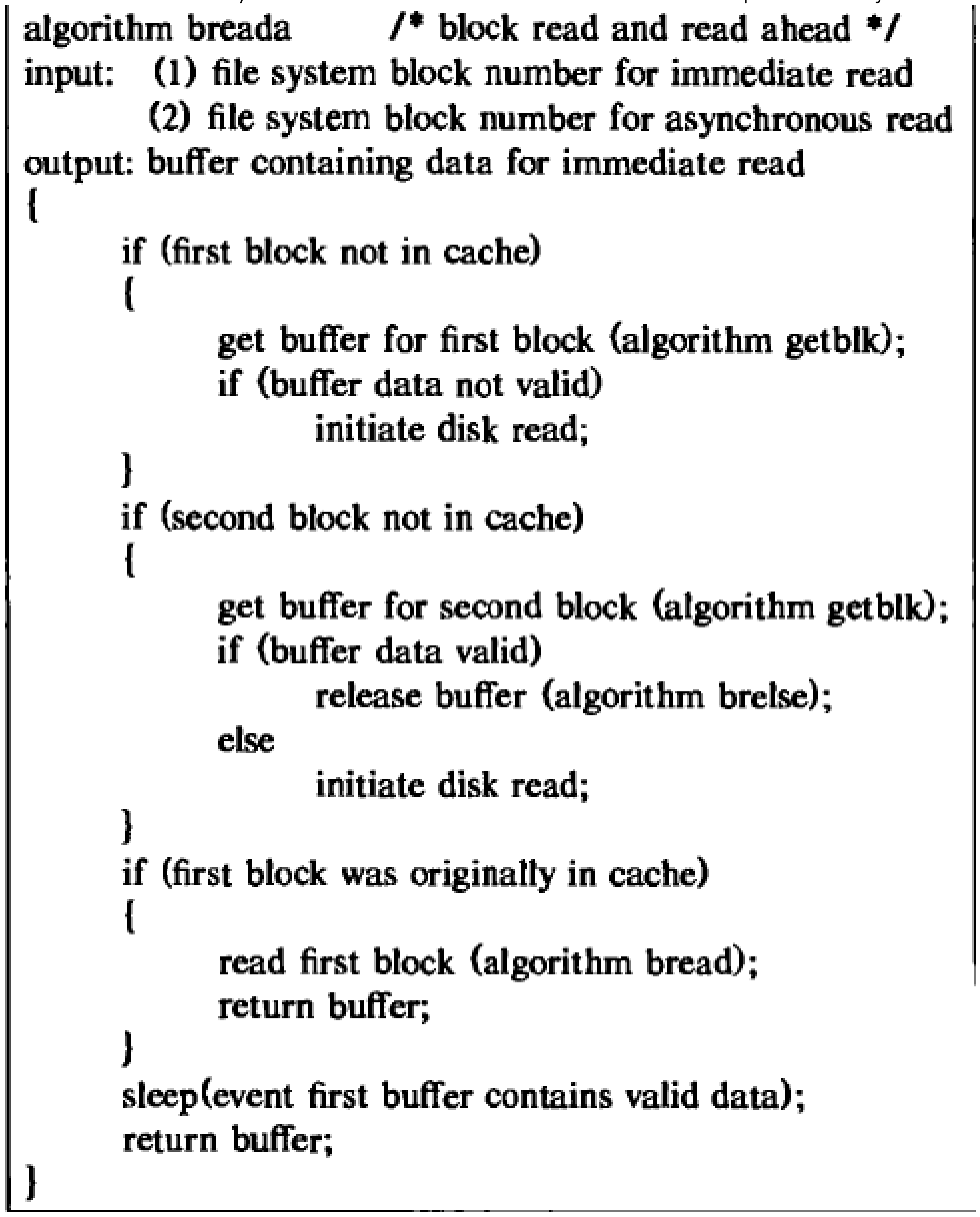
To read a disk block, a process uses algorithm getblk to search for it in the buffer cache. If it is in the cache, the kernel can return it immediately without physically reading the block from the disk



```
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);
}
```

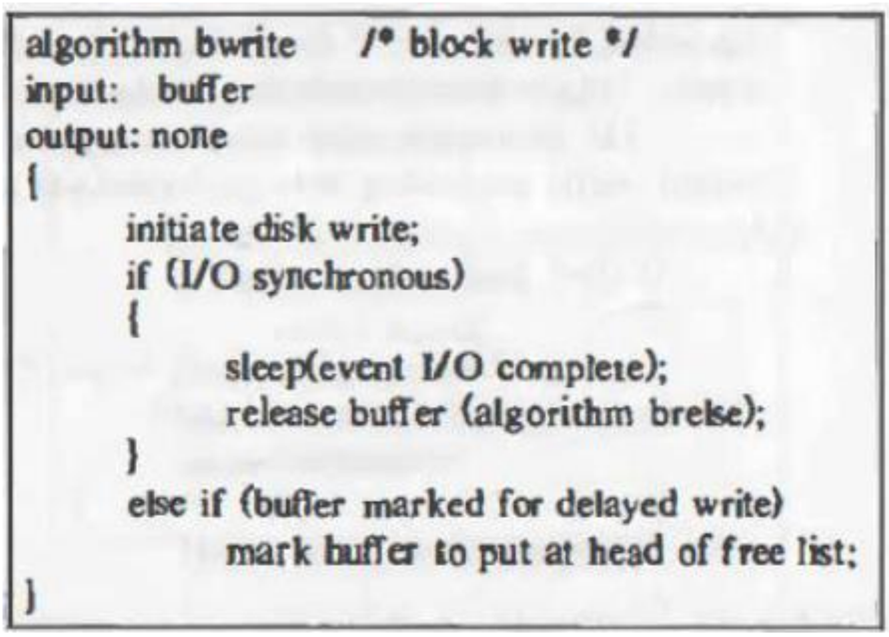
If it is not in the cache, the kernel calls the disk driver to "schedule" a read request and goes to sleep awaiting the event that the I/O completes. The disk driver notifies the disk controller hardware that it wants to read data, and the disk controller later transmits the data to the buffer. Finally, 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. higher-level kernel modules (such as the file subsystem)may anticipate the need for a second disk block when a process reads a filesequentially. The modules request the second I/O asynchronously in the hope that the data will be in memory when needed, improving performance. To do this, the kernel executes the block read-ahead algorithm *breada*. The kernel checks if the first block is in the cache and, if it is not there, invokes the disk driver
to read that block. If the second block is not in the buffer cache, the kernel instructs the disk driver to read it asynchronously. Then the process goes to sleep awaiting the event that the I/O is complete on the first block. When it awakens, it returns the buffer for the first block, and does not care when the I/0 for the second block completes. When the I/O for the second block does complete, the disk controller interrupts the system; the interrupt handler recognizes that the I/O was asynchronous and releases the buffer (algorithm *brelse).* If it would not release the buffer, the buffer would remain locked and, therefore, inaccessible to all processes. It is impossible to unlock the buffer beforehand, because I/O to the buffer was
active, and hence the buffer contents were not valid. Later, if the process wants to read the second block, it should find it in the buffer cache, the I/O having completed in the meantime. If, at the beginning of *breada,* the first block was in the buffer cache, the kernel immediately checks if the second block is in the cache and proceeds as just described.



```
algorithm breada        /* block read and read ahead */
input:  (1) file system block number for immediate read
        (2) file system block number for asynchronous read
output: buffer containing data for immediate read
{
        if (first block not in cache)
        {
                get buffer for first block (algorithm getblk);
                if (buffer data not valid)
                        initiate disk read;
        }
        if (second block not in cache)
        {
                get buffer for second block (algorithm getblk);
                if (buffer data valid)
                        release buffer (algorithm brelse);
                else
                        initiate disk read;
        }
        if (first block was originally in cache)
        {
                read first block (algorithm bread);
                return buffer;
        }
        sleep(event first buffer contains valid data);
        return buffer;
}
```

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;
}
```

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.
the implementation of bread and bwrite algorithms in xv6 is shown here.

```
struct buf *
bread(uint dev, uint sector)
{
struct buf * b;
b=bget(dev,sector);
If(!(b->flags & B_valid))
iderw(b);
}
```

```
Void bwrite(struct buf *b)
{
If (b->flags & B_BUSY)==0)
   panic("bwrite");
   b->flags |= B_DIRTY;
   iderw(b);
}
```

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. o 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.

LOGGING

The purpose of a file system is to: Organize and store data, Support sharing of data among users and applications ,and Ensure persistence of data after a reboot.

The file system needs to support crash recovery and a restart must not corrupt the file system or leave it in an inconsistent state One of the most interesting problems in file system design is crash recovery.

xv6 implements file system fault tolerance through a simple logging mechanism

• System calls do not directly write file system data structures , Instead: A system call first writes a description of all the disk writes that it wishes to perform to a log on the disk. It then writes a special commit record to the log to specify that it contains a complete operation .Next it copies the required writes to the on-disk file system data structures. Finally, it deletes the log

- In case of a reboot, the file system performs recovery by looking at the log file .
- If the log contains the commit record, the recovery code copies the required writes to the on-disk data structures .
- If the log does not contain a complete operation, it is ignored and deleted.
- If the crash occurs before the commit record, the log will be ignored, and the state of the disk will stay unmodified .
- If the crash occurs after the commit record, then the recovery will replay all of the operation's writes, even repeating them if the crash occurred during the write to the on-disk data structure .
- In both cases, the correctness of the file system is preserved: Either all writes are reflected on the disk or none
- The log resides at a fixed location at the end of the disk . It consists of a header block and a set of data blocks . The header block contains an array of sector numbers, one for each of the logged data blocks . Count of logged blocks . The header block is written to after a commit . The count is set to zero once all logged blocks have been reflected in the file system. The count will be zero in case of a crash before a commit .The count will be non-zero in case of a crash after a commit

```
begin_trans();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
commit_trans();
```

- begin_trans: Waits until it obtains exclusive use of the log •
- log_write:
  - Appends the block's new content to the log on the disk
  - Leaves the modified block in the buffer cache so that subsequent reads of the block during the transaction will yield the updated state
  - Records the block's sector number in memory to find out when a block is written multiple times during a transaction and overwrite the block's previous copy in the log •
- commit_trans:
  - Writes the log's header block to disk, updating the count
  - Calls install_trans to copy each block from the log to the relevant location on the disk
  - Sets to count in the log header to zero

the implementation of Log_write in XV6 is shown here.

```
void
log_write(struct buf *b)
{
int i;

if (log.lh.n >= LOGSIZE || log.lh.n >= log.size − 1)
panic("too big a transaction");
if (log.outstanding < 1)
panic("log_write outside of trans");

acquire(&log.lock);
for (i = 0; i < log.lh.n; i++) {
if (log.lh.block[i] == b−>blockno) // log absorbtion
break;
}
log.lh.block[i] = b−>blockno;
if (i == log.lh.n)
log.lh.n++;
b−>flags |= B_DIRTY; // prevent eviction
release(&log.lock);
}
```

```
// Return a locked buf with the contents of the indicated block.
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b;
}

// Write b's contents to disk.  Must be locked.
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```