Lecture 31: Device driver and block I/O in xv6

Mythili Vutukuru

IIT Bombay

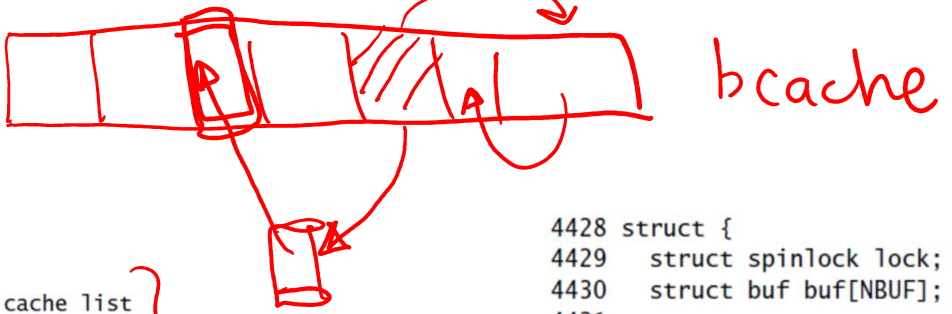
https://www.cse.iitb.ac.in/~mythili/os/

File systems and I/O in xv6

- Multiple layers of abstraction in file systems
  - System call implementations (open, read, write)
  - Operations on file system data structures (inodes, files, directories)
  - Block I/O layer (in-memory cache of disk blocks)
  - Device driver (communicates with hard disk to read/write blocks)
- This lecture and next: overview of these various layers in the xv6 file system

Disk blocks and buffers

- Disk maintains data as 512-byte blocks
- Disk buffer (struct buf) = copy of disk block in memory
- Buffer cache (bcache) is an array of disk buffers
  - Pointers across buffers create a linked list, most recently used buffers at head
- Reading a block from disk: assign buffer for the block number in buffer cache, device driver sends read request to disk controller, disk controller raises interrupt when data is ready, data copied from disk controller into buffer cache(VALID flag set after data is read)
- Writing a block to disk: first write into buffer in buffer cache, device driver copies data from buffer to disk controller, disk controller raises interrupt when write completes (DIRTY flag is set until disk is updated)

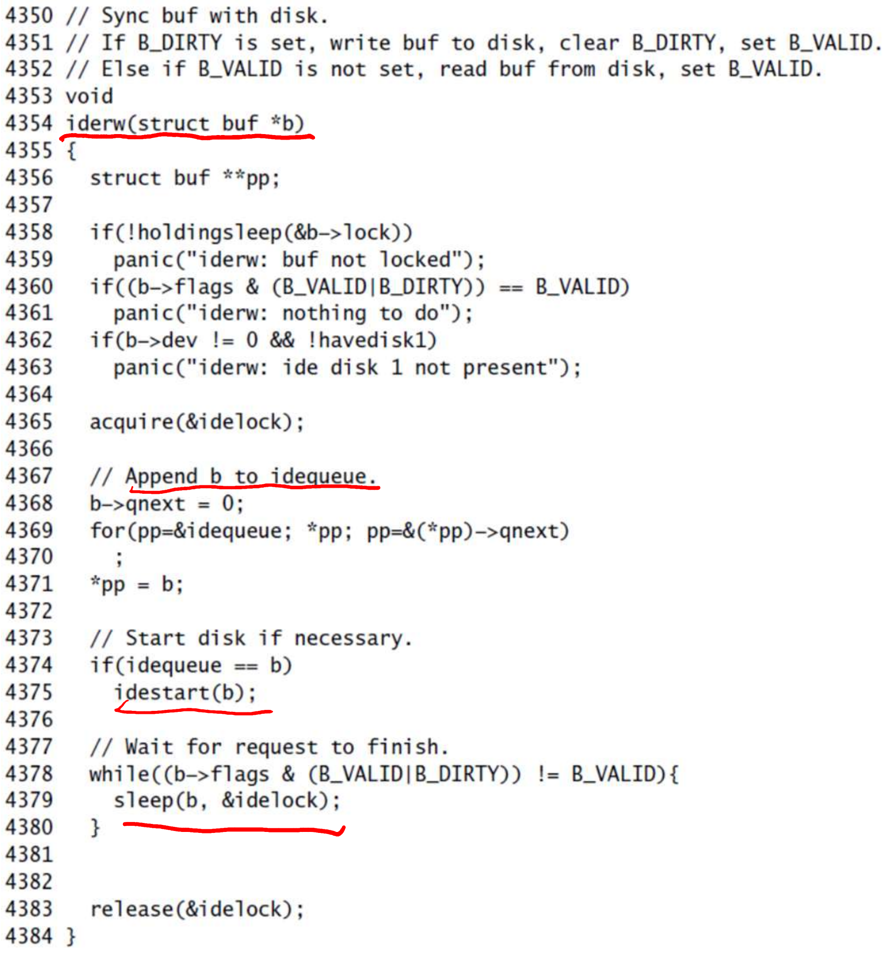


```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

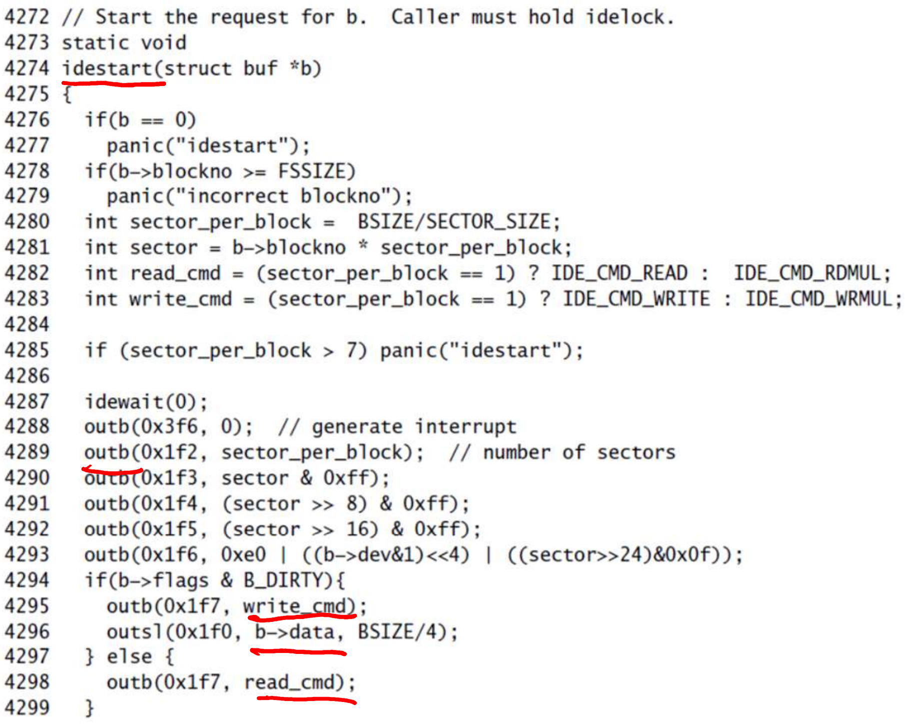
```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

Device driver (1)

- Process that wishes to read/write calls iderw function, buffer as argument
  - If buffer is dirty, write request. If buffer is not valid, read request
  - Requests added to queue, function idestart issues requests one after another
  - Process sleeps until request completes
- Communication with disk controller registers via in/out instructions



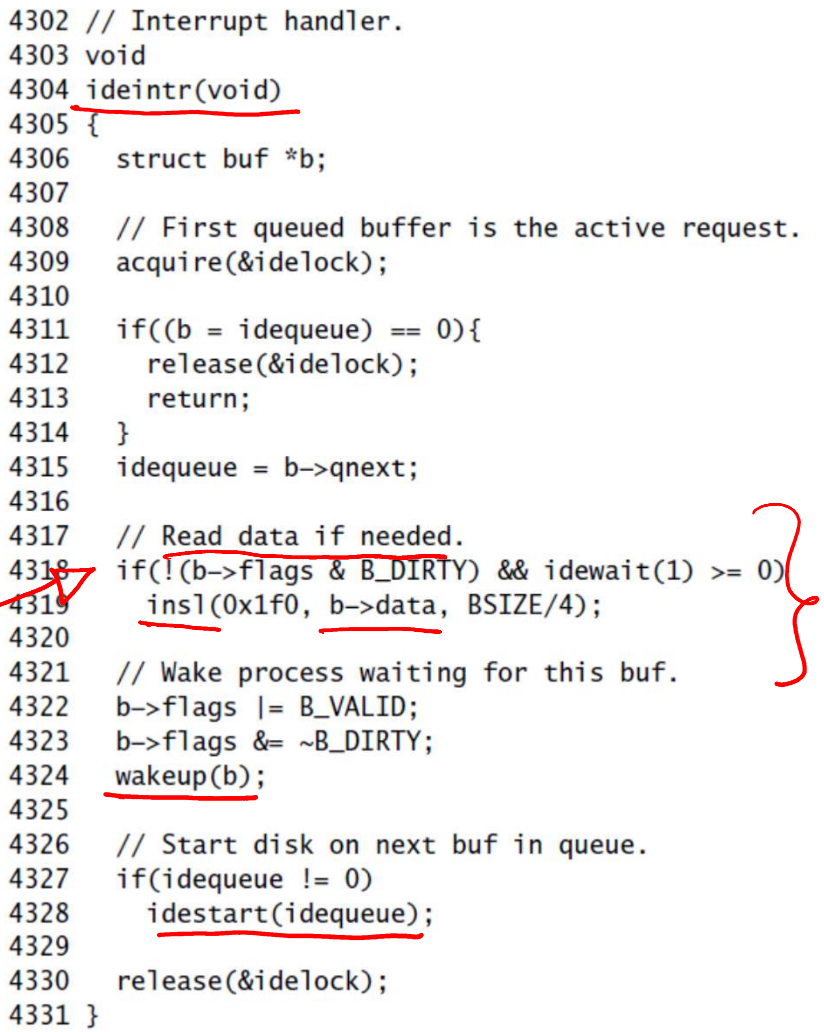
```
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!holdingsleep(&b->lock))
    panic("iderw: buf not locked");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }


  release(&idelock);
}
```



```
// Start the request for b.  Caller must hold idelock.
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");
  if(b->blockno >= FSSIZE)
    panic("incorrect blockno");
  int sector_per_block =  BSIZE/SECTOR_SIZE;
  int sector = b->blockno * sector_per_block;
  int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ :  IDE_CMD_RDMUL;
  int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;

  if (sector_per_block > 7) panic("idestart");

  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, sector_per_block);  // number of sectors
  outb(0x1f3, sector & 0xff);
  outb(0x1f4, (sector >> 8) & 0xff);
  outb(0x1f5, (sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, write_cmd);
    outsl(0x1f0, b->data, BSIZE/4);
  } else {
    outb(0x1f7, read_cmd);
  }
```

Device driver (2)

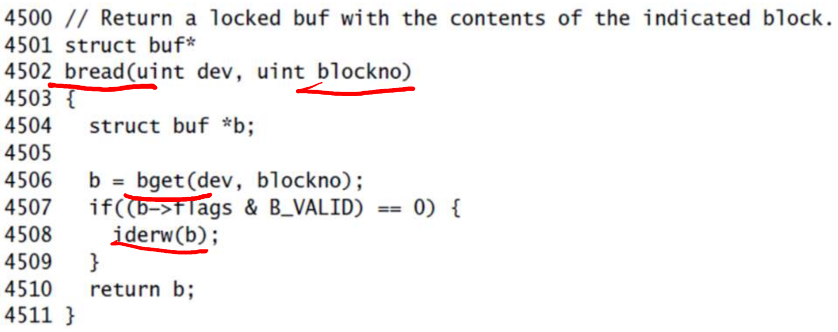
- When disk controller completes read/write operation, it raises an interrupt
  - Data is read from disk controller into buffer using “in” instruction
  - Process sleeping for data is woken up
  - Next request from queue is issued
- No support for DMA in x86. With DMA, data is copied by disk controller into memory buffers directly before raising interrupt
  - Interrupt handler need not copy data using I/O instructions



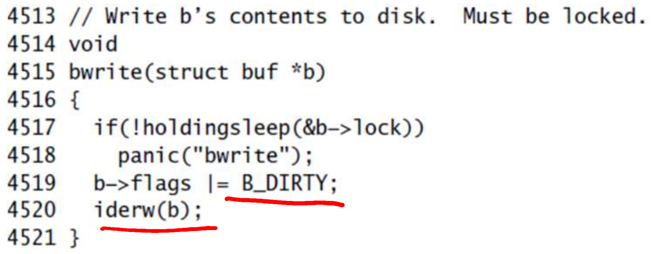
```
// Interrupt handler.
void
ideintr(void)
{
  struct buf *b;

  // First queued buffer is the active request.
  acquire(&idelock);

  if((b = idequeue) == 0){
    release(&idelock);
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, BSIZE/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}
```

Disk buffer cache: block read/write (1)

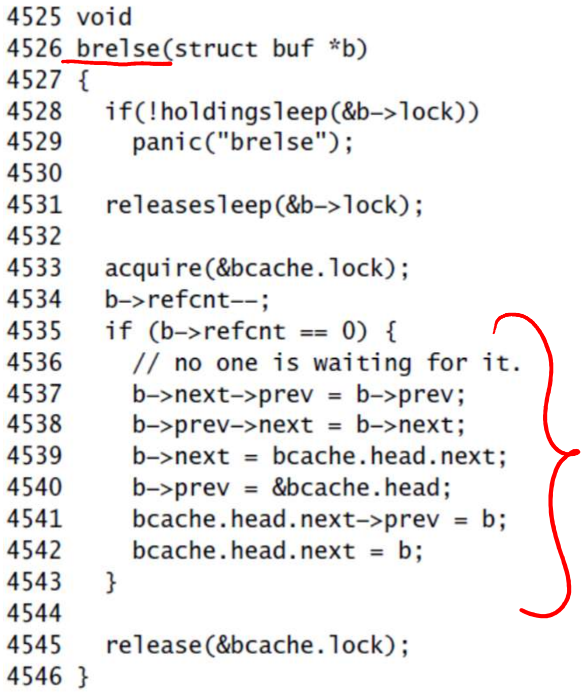
- All processes access disk via buffer cache only
- Only copy of disk block in cache, only one process can access it at a time
- Process calls “bread” to read a disk block, which calls function bget
  - Function bget returns buffer if it already exists in cache and no other process using it
  - If valid buffer not returned by bget, read from disk
- Process calls “bwrite” to write a block to disk: set dirty bit and request device driver to write
- When done with block, process calls brelse to release block, moves to head of list (most recently used)



```
// 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;
}
```



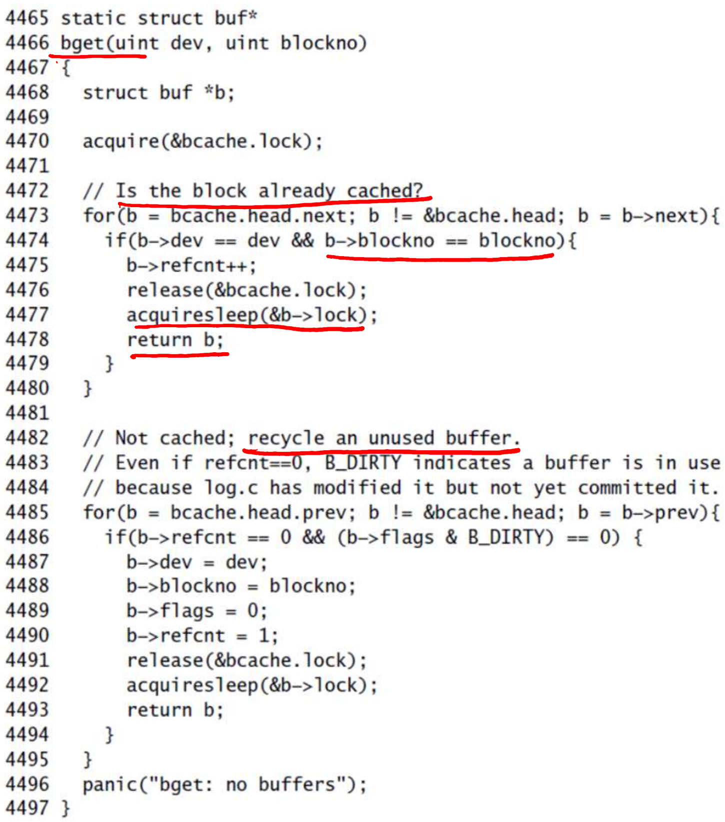
```
// Write b's contents to disk.  Must be locked.
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```



```
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  acquire(&bcache.lock);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }

  release(&bcache.lock);
}
```

Disk buffer cache: block read/write (2)

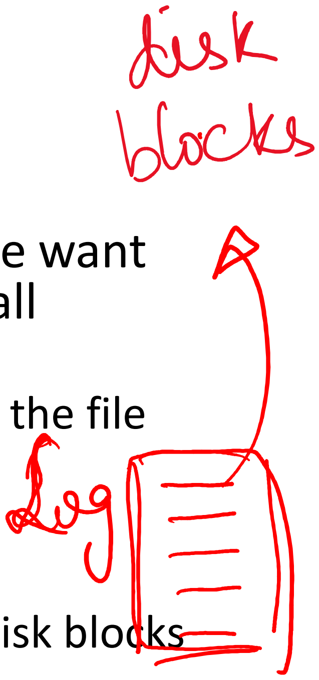
- Function bget returns pointer to a disk block if it exists in the cache
  - Ensures only one process at a time accesses a disk buffer
- If block in cache and another process using it, sleep until the block is released by the other process
- If block not in cache, find a least recently used non-dirty buffer and recycle it to use for this block
- Two goals achieved by buffer cache
  - Recently used disk blocks stored in memory for future use
  - Disk blocks modified by one process at a time



```
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");
}
```

Logging layer (overview)

- A system call can change multiple blocks at a time on disk, and we want atomicity in case the system crashes during a system call. Either all changes are made or none is made
  - Example: we do not want disk block added to the inode of a file but the file data not yet written to it
- Logging ensures atomicity by grouping disk block changes into transactions
  - Every system call starts a transaction in the log, writes all changed disk blocks in the log, and commits the transaction
  - Later, the log installs the changes in the original disk blocks one by one
  - If crash happens before log is written fully, no changes made
  - If crash happens after log entry is committed, log entries are replayed when system restarts after crash
- In xv6, changes of multiple system calls are collected in memory and committed to log together. Actual changes happen to disk blocks only after the group transaction commits
  - Process must call “log_write” instead of “bwrite” during system call



Summary

- Device driver in xv6 communicates with disk controller using in/out instructions to read/write disk blocks
  - Simple driver with no DMA capability
- Buffer cache stores all recently read disk blocks in memory, and synchronizes access to disk blocks across processes
- All blocks changed in a system call are logged on disk and changes are installed atomically
- Next: File system code translates system calls into block read/write operations