The Buffer Layer

xv6-rev7

Carmi Merimovich

CS School, Tel Aviv Academic College

January 12,2017

Buffer layer routines

- bp = bread(dev,nblock);
- bwrite(bp);
- brelse(bp);
- nblock = balloc();
- bfree(nblock);

struct buf

```
3501  struct buf {
        int flags; //B_BUSY, B_VALID, B_DIRTY
        uint dev;
        uint sector;
        struct buf *prev;
        struct buf *next;
        struct buf *qnext;
        uchar data[512];
      };
```

First looks

- We begin by examining:
  - bread.
  - bwrite.
- The driver layer supplies the routine iderw(bp):
  - If B_VALID is clear, iderw will read the block.
  - If B_DIRTY set, iderw will write the block.

bread

```
4101  struct buf bread(uint dev, uint sector) {
        struct buf *b;

        b = bget(dev, sector);
        if (!(b->flags & B_VALID))
          iderw(b);
        return b;
      }
```

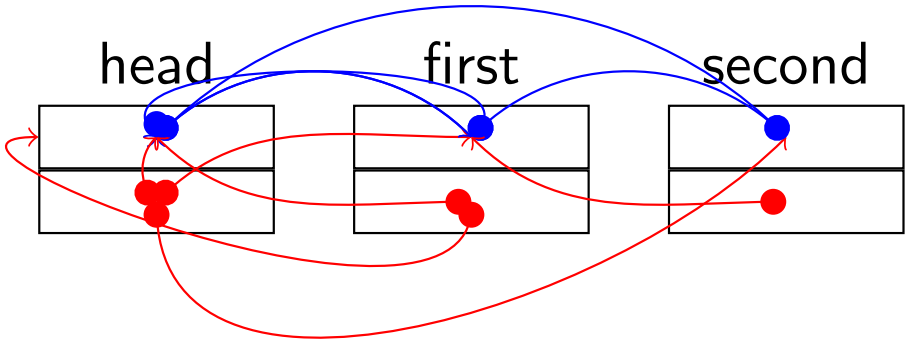
bwrite

```
4113  void bwrite(struct buf *b) {
        if ((b->flags & B_BUSY) == 0)
          panic("bwrite");
        b->flags |= B_DIRTY;
        iderw(b);
      }
```

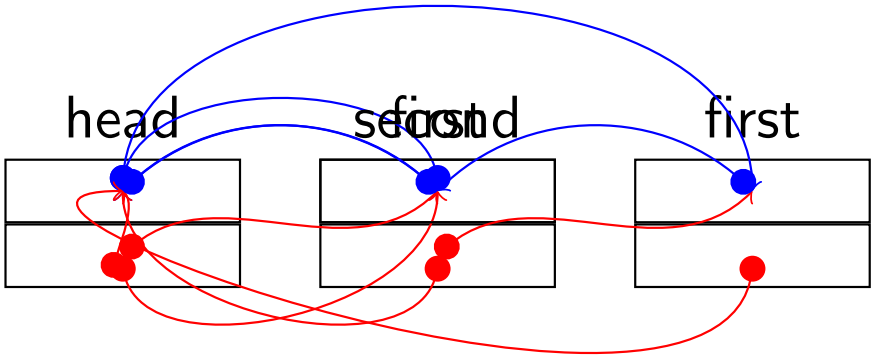
The buffer cache

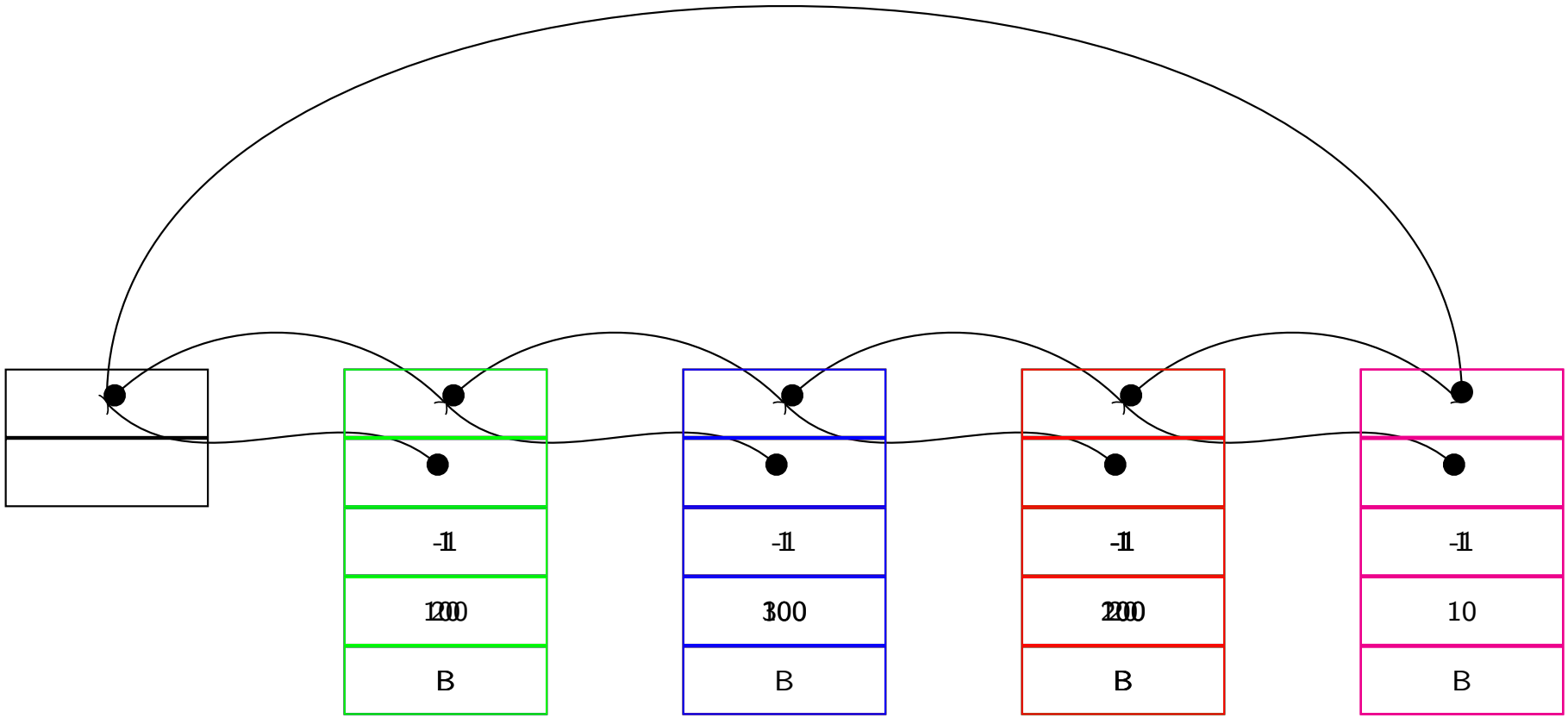
```
4028  struct {
        struct spinlock lock;
        struct buf buf[NBUF];

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

Cache list, tail insertion



Cache list, head insertion




-1
B
-1
300
B
-1
B
-1
10
B

Cache initialization

```
⋮

4051    // Create linked list of buffers
    bcache.head.prev = &bcache.head;
    bcache.head.next = &bcache.head;
    for (b = bcache.buf; b < bcache.buf+NBUF; b++){
     b->next = bcache.head.next;
     b->prev = &bcache.head;
     b->dev = -1;
     bcache.head.next->prev = b;
     bcache.head.next = b;
    }
  }
```

bget (1)

```
4065 static struct buf *bget(uint dev, uint sector) {
      struct buf *b;
      acquire(&bcache.lock);
     loop:
      // Is the sector already cached?
      for (b = bcache.head.next; b != &bcache.head;
                                      b = b->next) {
       if (b->dev == dev && b->sector == sector){
        if (!(b->flags & B_BUSY)){
         b->flags |= B_BUSY;
         release(&bcache.lock);
         return b;
        }
        sleep(b, &bcache.lock);
        goto loop;
       }
      }
```

bget (2)

```
  // Not cached; recycle some non busy and clean bu
  for (b = bcache.head.prev; b != &bcache.head;
                                  b = b->prev){
    if((b->flags & B_BUSY) == 0 &&
       (b->flags & B_DIRTY) == 0){
      b->dev = dev;
      b->sector = sector;
      b->flags = B_BUSY;
      release(&bcache.lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

brelse

```
4124  void brelse(struct buf *b) {
       if ((b->flags & B_BUSY) == 0)
        panic("brelse");

       acquire(&bcache.lock);

       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;

       b->flags &= ~B_BUSY;
       wakeup(b);
       release(&bcache.lock);
      }
```

bitmap

Looking for cleared bit in a byte, the set it.

```
uchar c;

if ((c & 1) == 0)   {c |= 1; return 0};
if ((c & 2) == 0)   {c |= 2; return 1;)
if ((c & 4) == 0) {c |= 4 return 2;}


if ((c & 128)  == 0))  {c |= 128; return 7;}
```

Find cleared bit in a loop

```
bit 0:    1 << 0
bit 1:    1 << 1

bit 7:    1 << 7

 for (bi=0; bi < 8; bi++) {
  m = 1<<bi;
  if ((c & m) == 0) {
   c |= m;
   return (bi);
  }
 }
```

Large bitmap

```
uchar c[512];

 for (bi =0; bi < 4096; bi++) {
  m = 1 << (b i% 8);
  if ((c[bi/8] & m) == 0) {
    c[bi/8] |= m;
    return (bi);
  }
 }
```

Even larger bitmap

```
uchar c0[512],c1[512];

 for (bi = 0; bi < 4096; bi++) {
  m = 1 << (bi % 8);
  if ((c0[bi/8] &  m) == 0} {
   c0[bi/8] |= m;
   return (bi);
  }
 }
 for (bi = 0; bi < 4096; bi++) {
   m = 1 << (bi % 8);
   if ((c1[bi/8] & m) == 0} {
    c1[bi/8] |= m;
    return (4096+bi);
  }
 }
```

Really long bitmap

```
uchar c[2][512];

 for (b = 0; b < 4096*2; b += 4096) {
  block = b/4096;
  for (bi = 0; bi < 4096; bi++) {
   m = bi % 8;
   if ((c[block][bi/8] & m) == 0} {
    c[block][bi/8] |= m;
    return (b+bi);
   }
  }
 }
```

Bitmap not necessarily multiple of 4096

```
#define NUMOFBITS 8000
uchar c[2][512];

  for (b = 0; b < NUMOFBITS; b += 4096) {
   block = b/4096;
   for (bi = 0; bi < 4096 && b+bi < NUMOFBITS; bi++)
    if ((c[block][bi/8] & m) == 0} {
     c[block][bi/8] |= m;
     return (b+bi);
    }
   }
  }
```

balloc

```
4453  static uint balloc(uint dev) {
       struct superblock sb;
       struct buf *bp = 0;
       readsb(dev, &sb);
       for (int b = 0; b < sb.size; b += BPB){
        bp = bread(dev, BBLOCK(b, sb.ninodes));
        for (int bi = 0; bi < BPB && b + bi < sb.size; bi++) {
         int m = 1 << (bi % 8);
         if ((bp->data[bi/8] & m) == 0){ // Is block free?
          bp->data[bi/8] |= m; // Mark block in use.
          log_write(bp);
          brelse(bp);
          bzero(dev, b + bi);
          return b + bi;
         }
        }
        brelse(bp);
       }
       panic("balloc: out of blocks");
```

bitmap, clearing bit

```
uchar c;

 m = 1 << bi;
 c &= ~m;
```

bitmap, clearing bit

```
uchar c[512];

 m = 1 << (bi % 8);
 c[bi/8] &= ~m;
```

bitmap, clearing bit

```
uchar c[2][512];

 block = b/4096;
 bi = b % 4096;
m = 1 << (bi % 8);
 c[block][bi/8] &= ~m;
```

bfree

```
4480  static void bfree(int dev, uint b) {
       struct buf *bp;
       struct superblock sb;
       int bi, m;

       readsb(dev, &sb);
       bp = bread(dev, BBLOCK(b, sb.ninodes));
       bi = b % BPB;
      m = 1 << (bi % 8);
       if ((bp->data[bi/8] & m) == 0)
        panic("freeing free block");
       bp->data[bi/8] &= ~m;
       log_write(bp);
       brelse(bp);
      }
```