File System: buffer cache

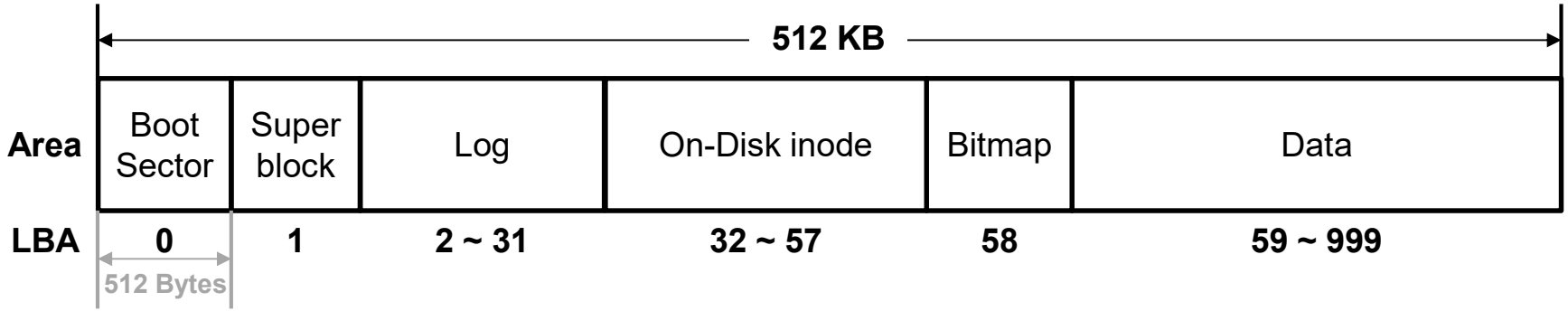
Youjip Won

KAIST EE

Contents

- Overview
- Buffer cache layer
- Code: Buffer cache
- Logging layer
- Log design
- Code: Logging

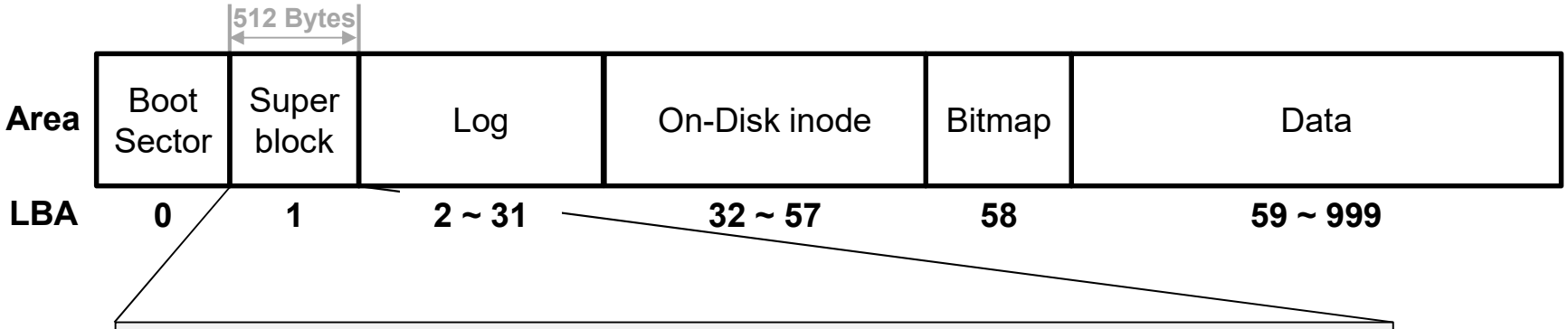
File System Layout of xv6



- Block 0: Boot sector, what we covered at Chapter 2
- Block 1: Superblock. (metadata about file system)
- Blocks starting at 2
  - Log
  - inodes
  - bitmap
  - data

Superblock in xv6 - Implementation

- A block of metadata describing the file system

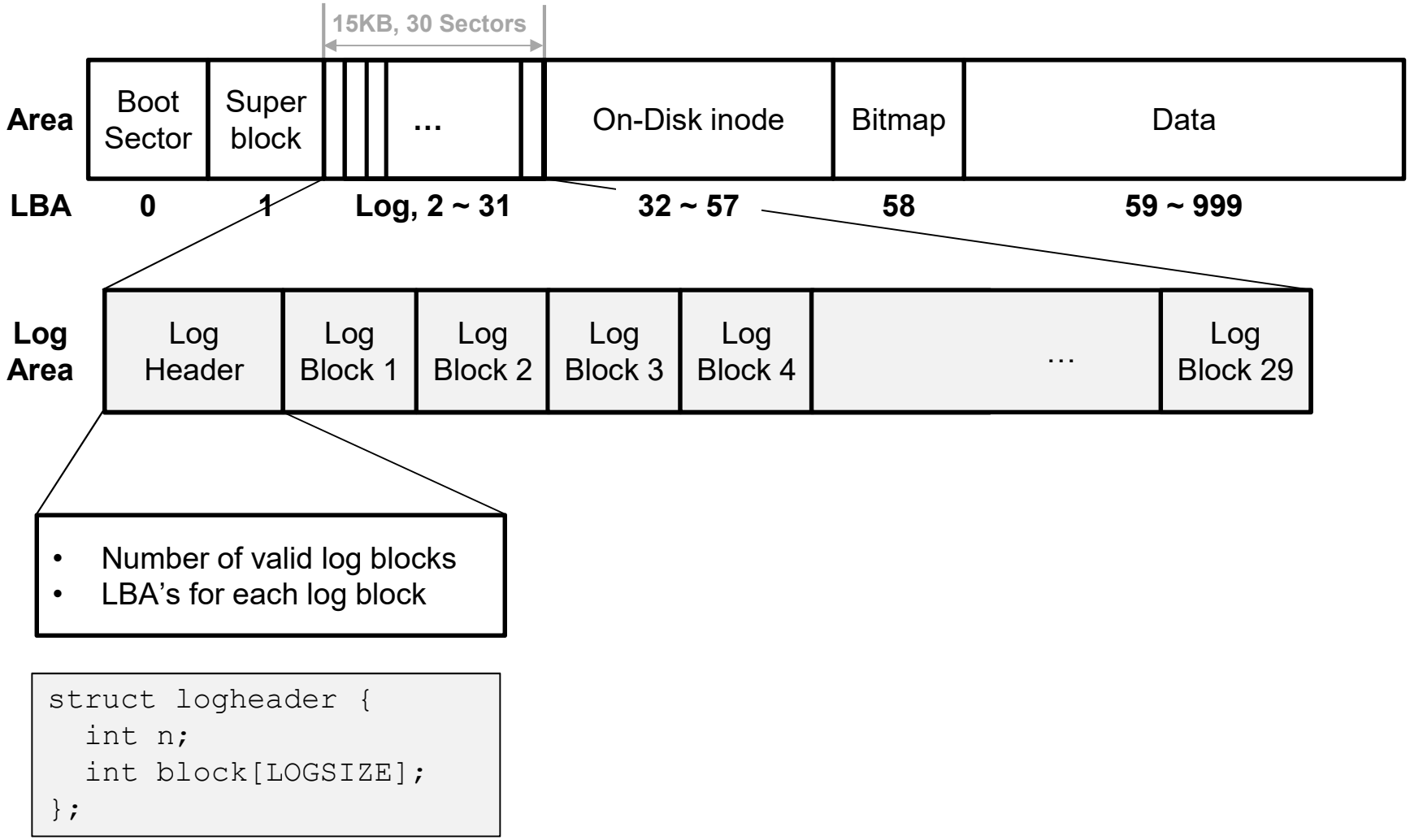



```
struct superblock {
  uint size;          // Size of file system image (blocks)
  uint nblocks;       // Number of data blocks
  uint ninodes;       // Number of inodes.
  uint nlog;          // Number of log blocks
  uint logstart;      // Block number of first log block
  uint inodestart;    // Block number of first inode block
  uint bmapstart;     // Block number of first free map block
};
```

```
(gdb) p sb
$1 = {size = 1000, nblocks = 941, ninodes = 200, nlog = 30, logstart = 2, inodestart = 32, bmapstart = 58}
```

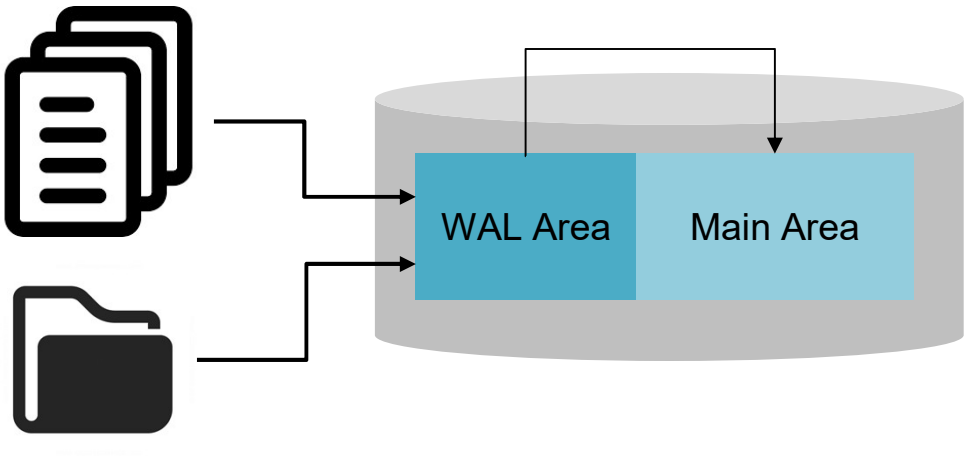
Log (Write-Ahead-Log)

- An area for consistency guarantee




```
struct logheader {
  int n;
  int block[LOGSIZE];
};
```

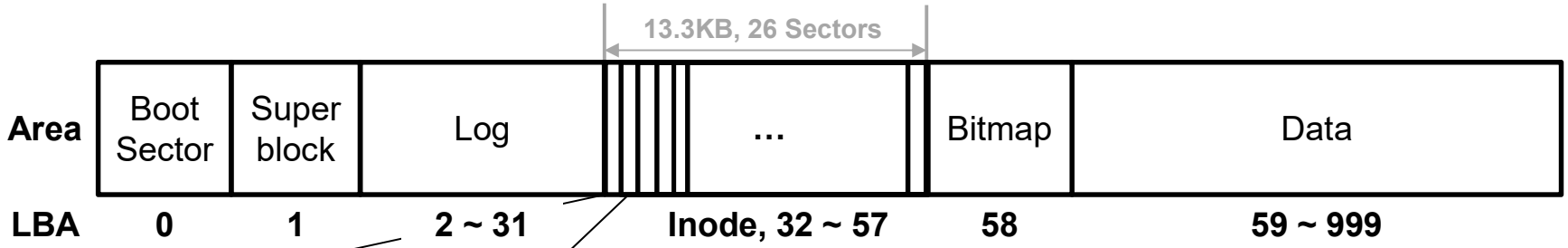
In-memory representation of the log area

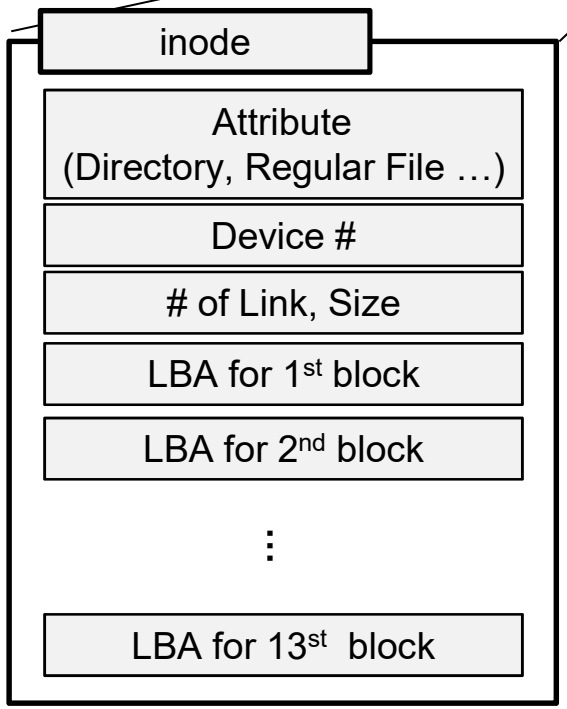



```
struct log {
  struct spinlock lock;
  int start;
  int size;
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev;
  struct logheader lh;
};
```

Inode structure

- Inode describes a file

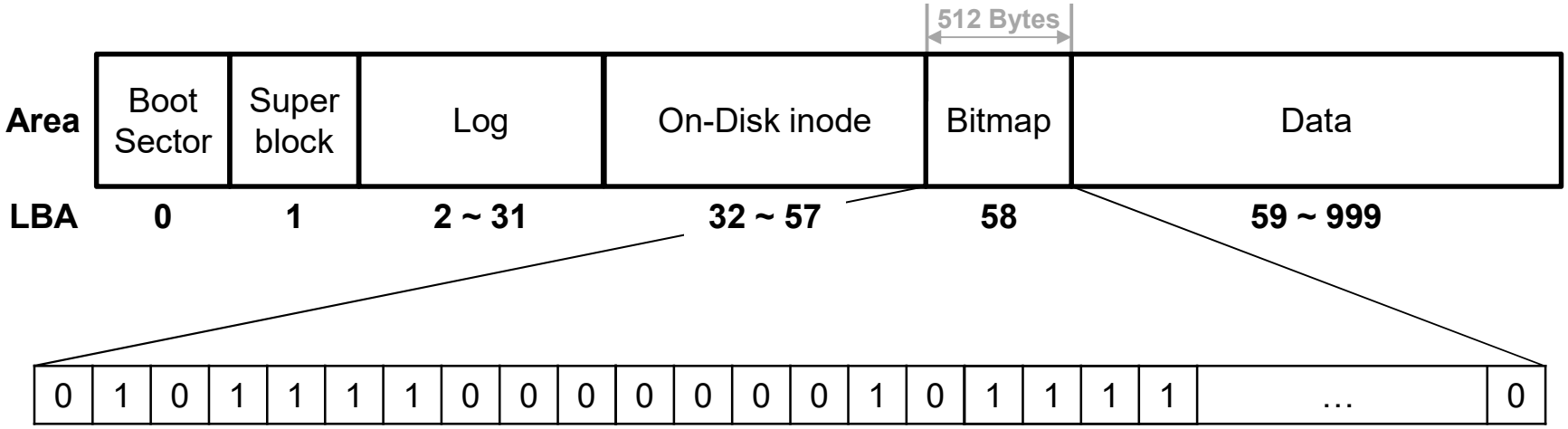





```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number
  short minor;          // Minor device number
  short nlink;          // Number of links to
inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block
addresses
};
```

Bitmap

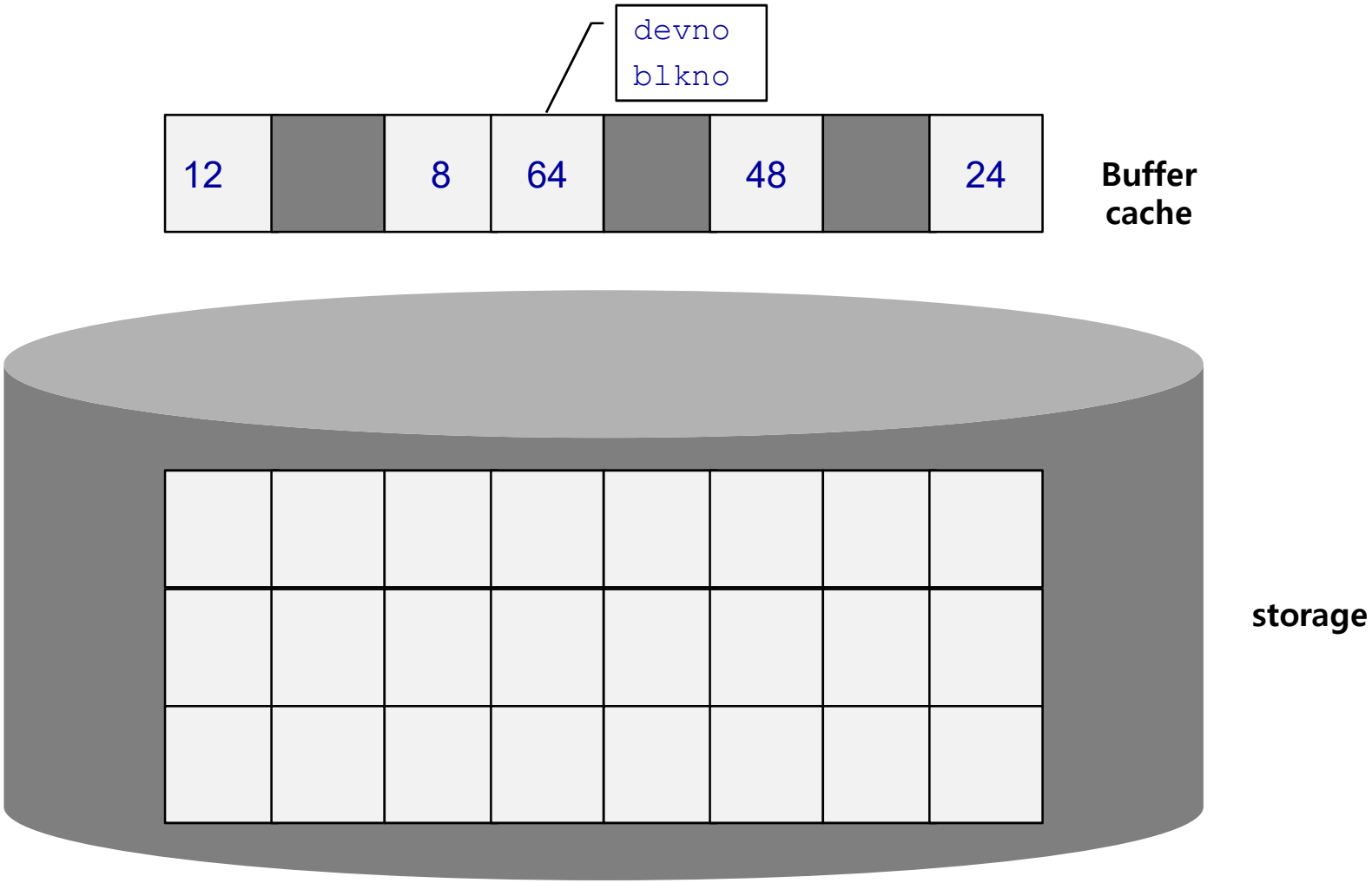
- Bitmap represents the state of the usage of blocks in filesystem



  - Each bit in bitmap represents the state of the block allocation.
  - 512byte can represent 4096 blocks (512 Bytes = 512 * 8 bit = 4096 bit).

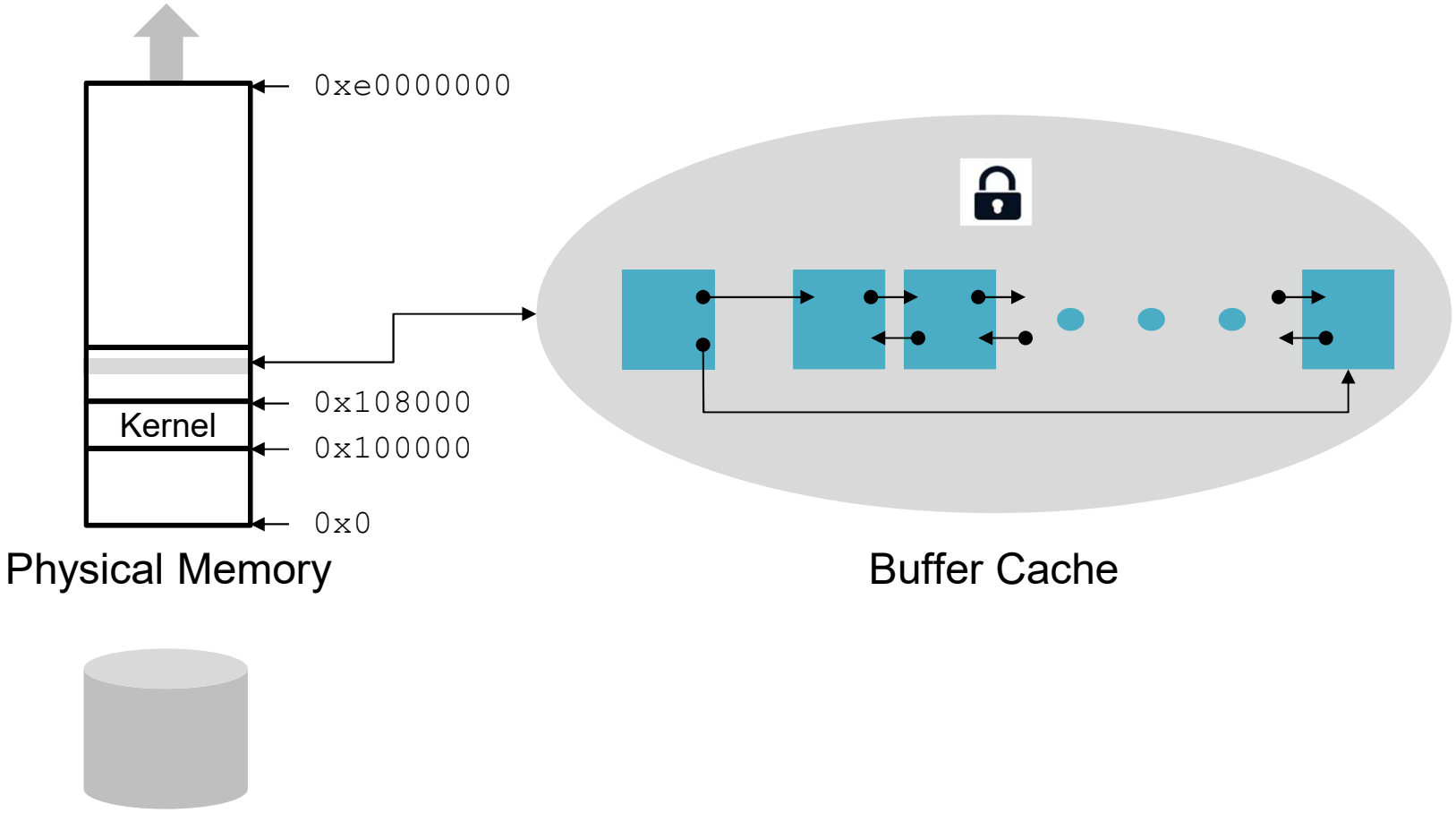
Buffer cache

- Use part of the memory as storage.



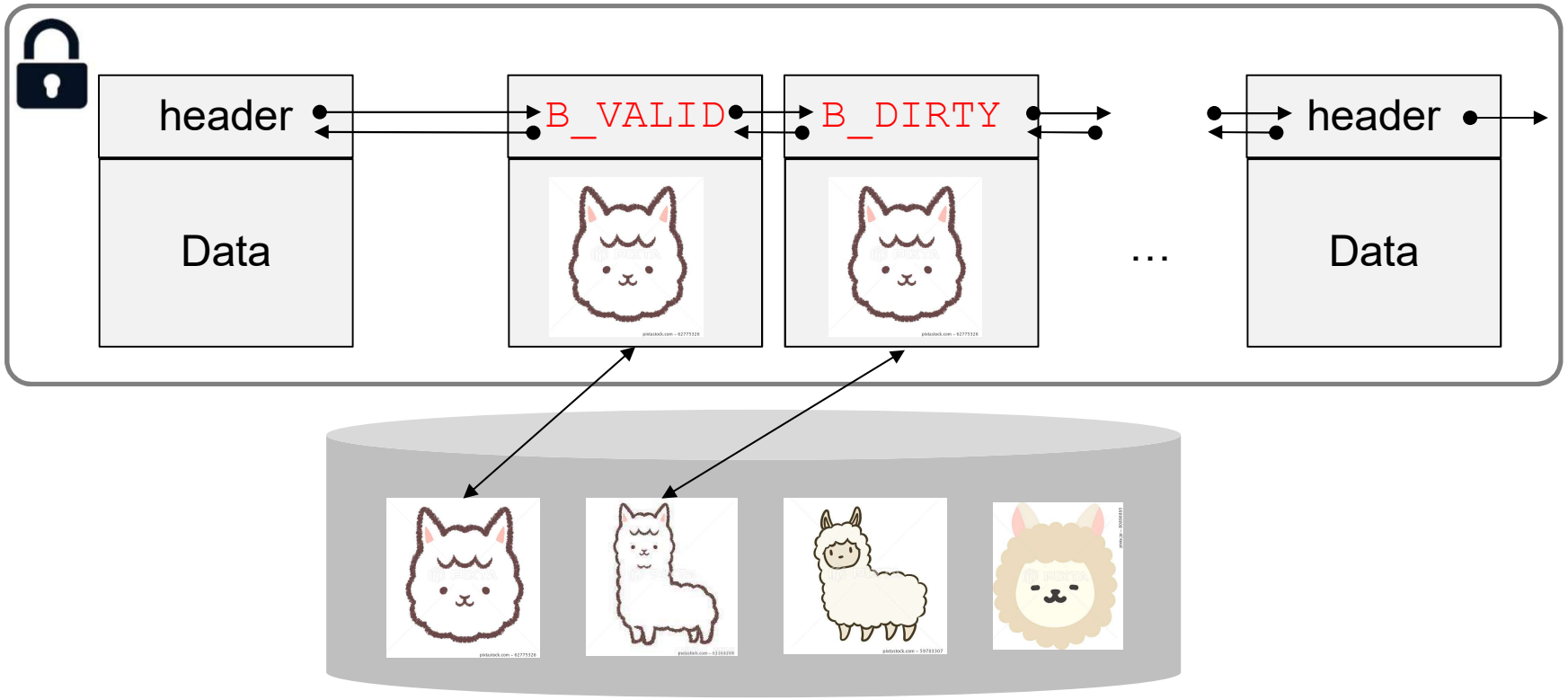
Buffer Cache Location at the Memory – Version 3

Rest are mapped for device.



Buffer Cache Entry - Flags

- Flags of buffer head indicates the state of the buffer cache entry
  - `B_VALID`: Buffer contains a copy of the block
  - `B_DIRTY`: Buffer content has been modified and needs to be written to the disk
  - If not set, buffer cache entry is unused

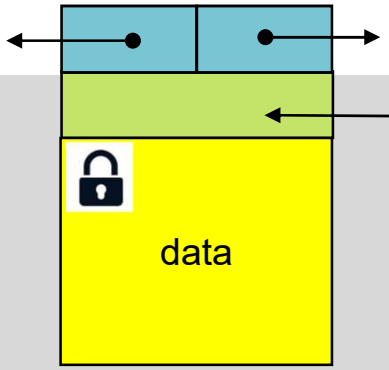


Buffer cache layer

- Buffer cache
  - Cache disk block in memory.
  - Ensure that only one copy of a block is in memory.
  - Ensure that only one kernel thread at a time accesses that copy.

- API's
  - `bget` : Scan the buffer cache and find the matching buffer .
  - `bread`: Obtain a buffer containing a copy of a block, lock the buffer.
  - `bwrite`: Write the modified buffer to the appropriate location on the disk.
  - `brelse`: unlock the buffer.

Buffer

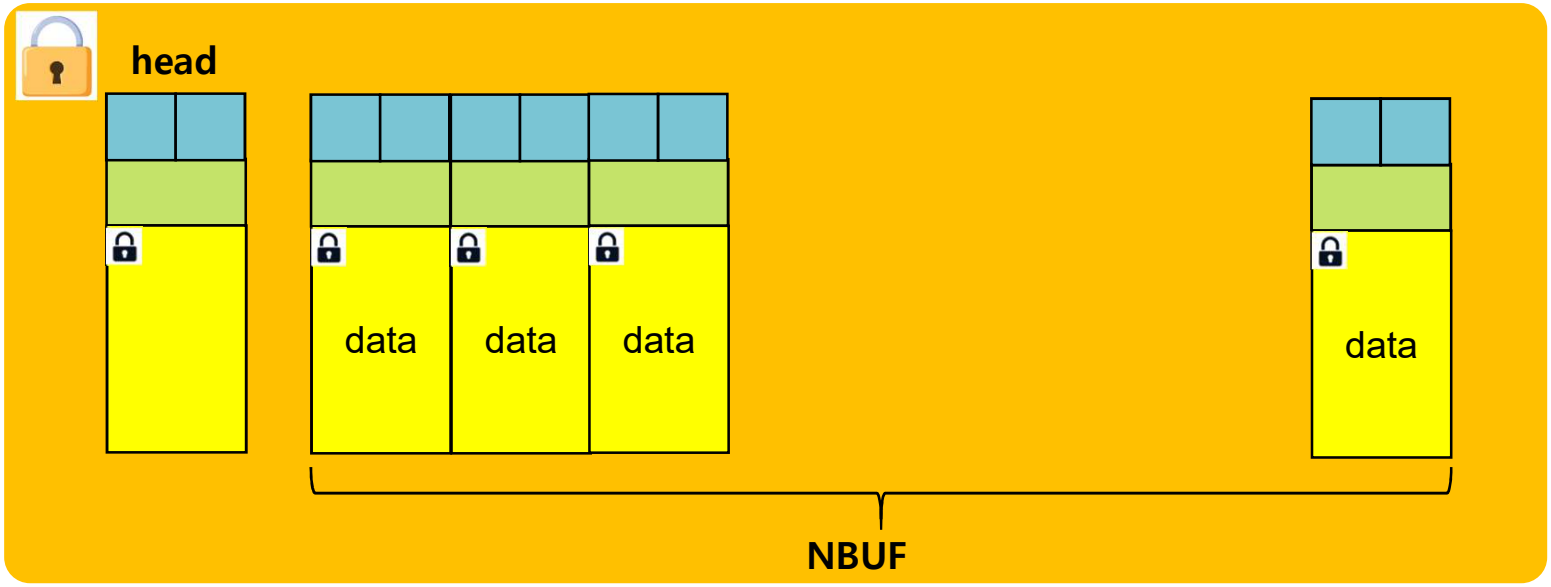


- flag
- device number
- blockno
- lock

```
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
```

Buffer cache: bcache




```
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;
```

Buffer cache mechanism

- The buffer cache uses a per-buffer sleep-lock.
  - Only one thread at a time uses each buffer.
  - `bread` returns a locked buffer.
  - `bget` locks the buffer.
  - `brelse` releases the lock.
- Buffer cache has a fixed number of buffer.
  - If the file system asks for a block that is not in cache, buffer cache recycles a buffer currently holding some other block.
  - LRU scheme for victim selection

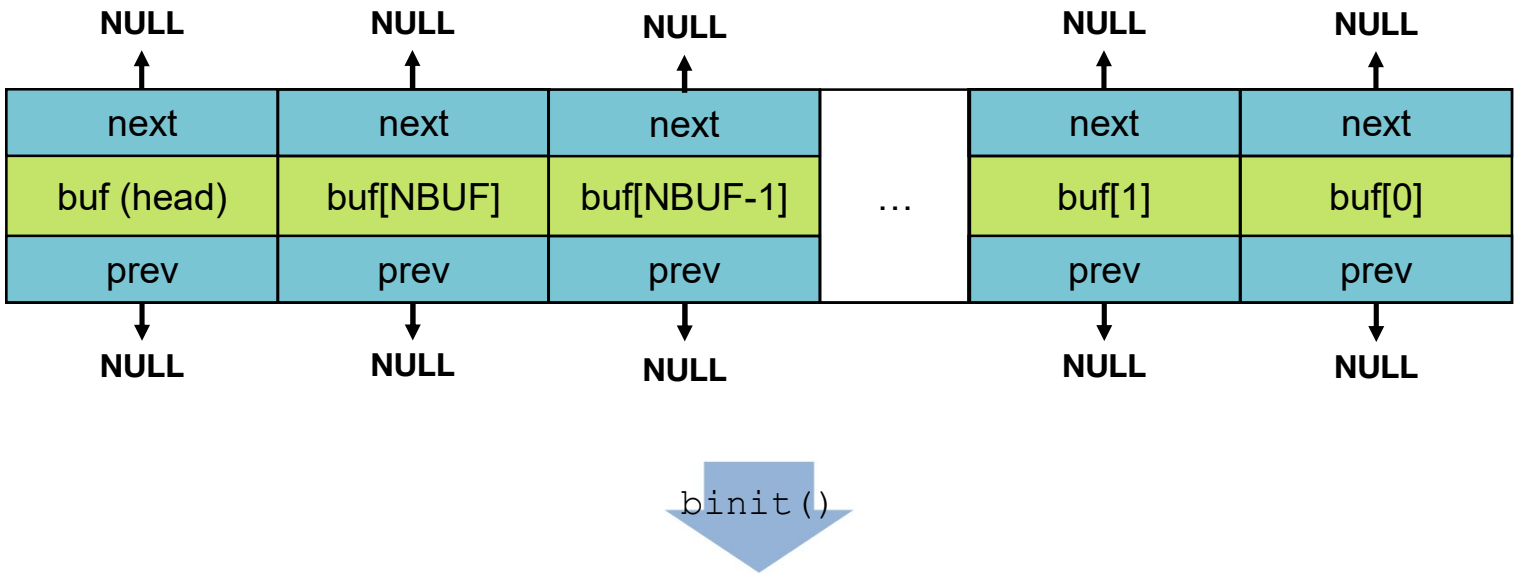
Buffer Cache - Initialization

- Buffer cache is already allocated when kernel is loaded (at data section).
- `binit()` is called at `main()` and initializes the list.

```
main(void)
{
  …
  binit();
  …
}
```

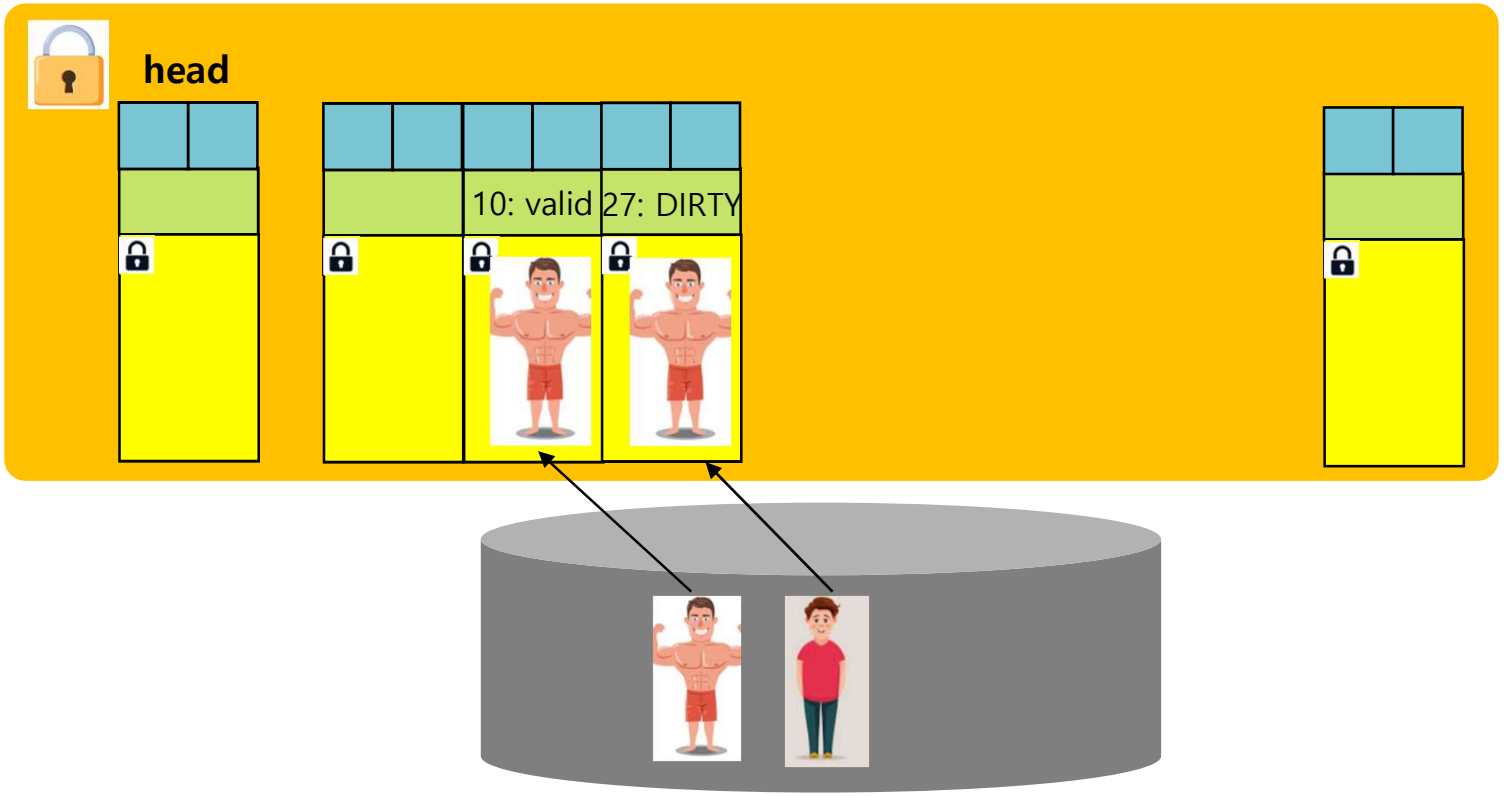
```
void
binit(void)
{
  struct buf *b;
  initlock(&bcache.lock, "bcache");
  // Create linked list of buffers
  bcache.head.prev = &bcache.head;
  bcache.head.next = &bcache.head;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    initsleeplock(&b->lock, "buffer");
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}
```

`binit()` - initialize the list of buffers



buffer state

- Buffer has two state bits
  - `B_VALID`: Buffer contains a copy of the block
  - `B_DIRTY`: Buffer content has been modified and needs to be written to the disk

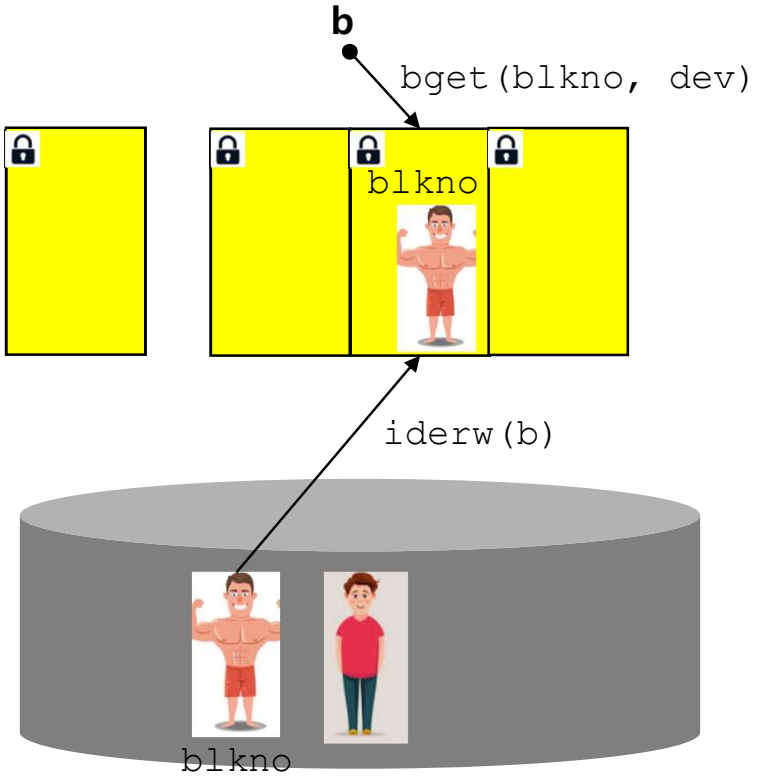


bread()

- read '`blockno`' from the '`dev`'.
- `bget`: scan the buffer cache and find the matching buffer with '`blockno`' from the '`dev`'.
- If buffer does not exist, read the disk block of `blockno`' from the '`dev`' into buffer pointed by `b`.

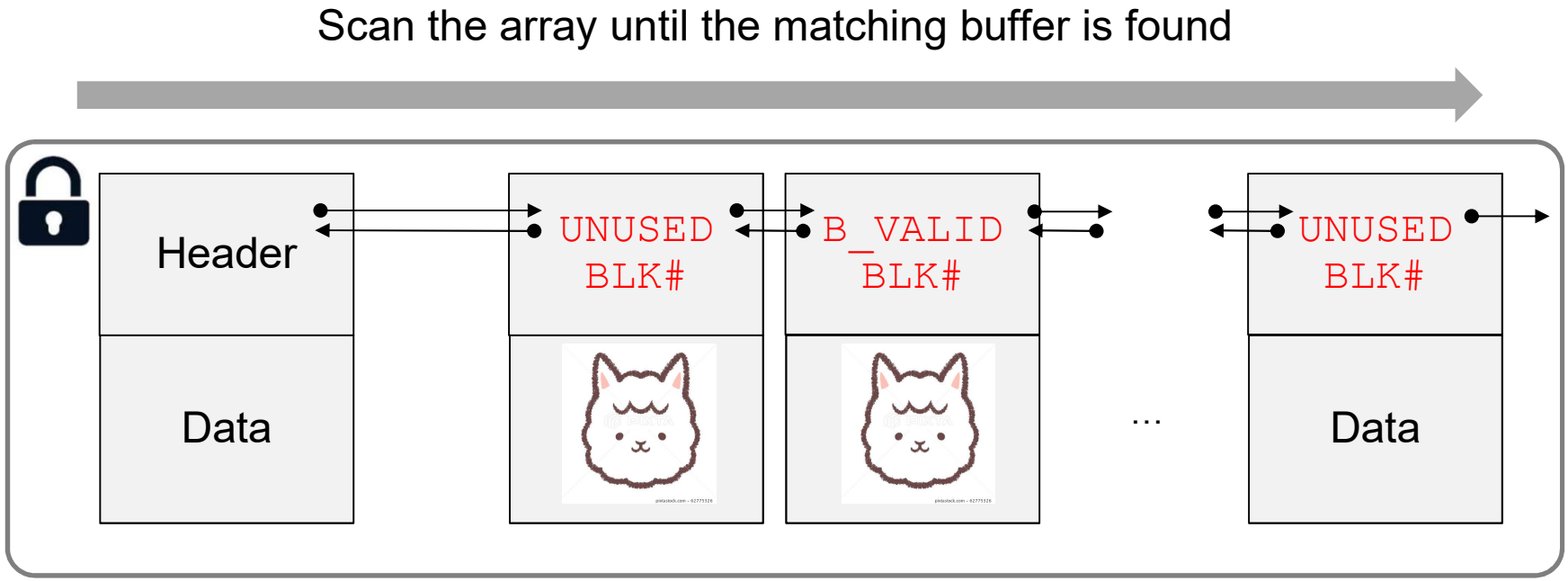
```
bread(uint dev, uint blockno)
{
  struct buf *b;

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



struct buf * bget(uint dev, uint blockno)

- Find the matching buffer by scanning.
  - [Cache Hit] Return the locked Buffer.
  - [Cache Miss] Return the locked Buffer which is newly allocated.



struct buf * bget(uint dev, uint blockno)

- Find the matching buffer by scanning.

```
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;
    }
  }
```

Note that we scan from the beginning of the MRU List.

struct buf * bget(uint dev, uint blockno)

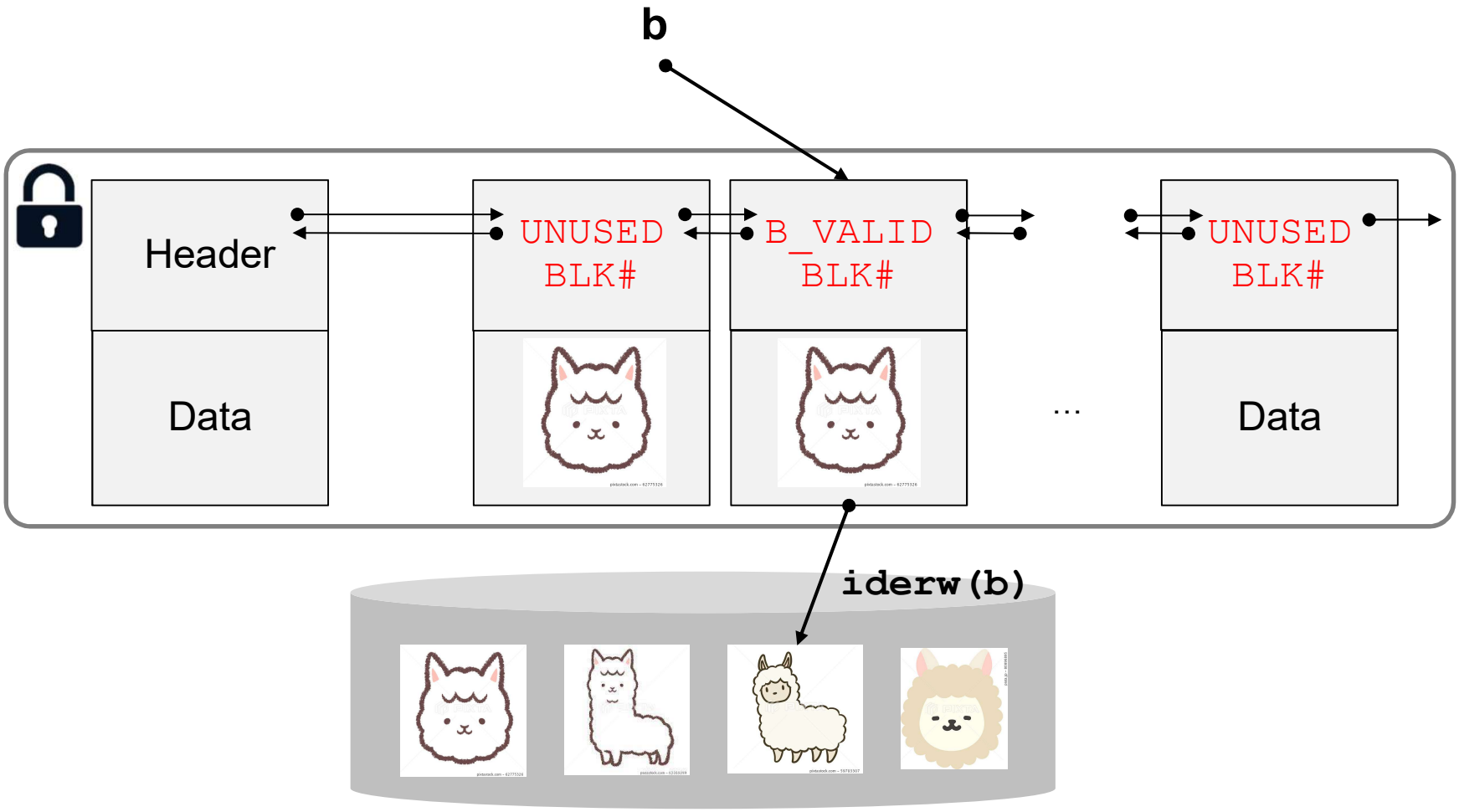
- Find the matching buffer by scanning.

```
  // 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");
}
```

Note that we scan from
the end of the MRU List
(the beginning of the LRU LIst).

void bwrite(struct buf *b)

- Write the modified buffer to the appropriate location on the disk.



void bwrite(struct buf *b)

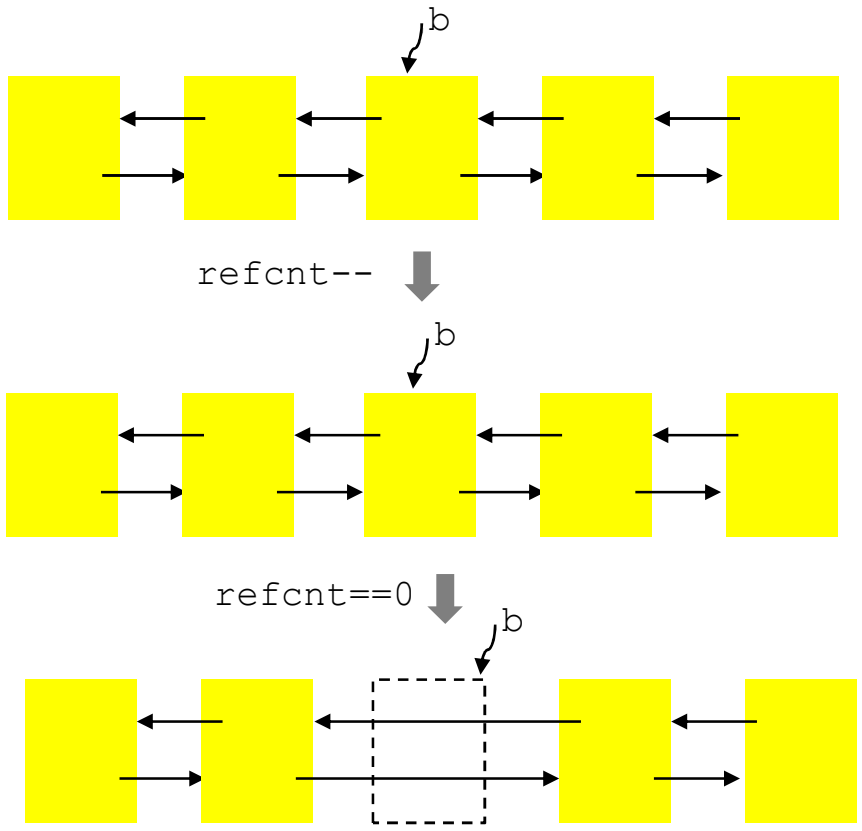
- Write the modified buffer to the appropriate location on the disk.

```
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

brelse()

- Decrease the reference counter.
- When reference counter is 0, remove it from the buffer cache and add it to the free buffer list.

```
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);
}
```



Summary

- State of the buffer: `B_VALID, B_DIRTY`
- Locks
  - Buffer lock
  - Buffer cache lock
- API's
  - `bread()`
  - `bget()`
  - `bwrite()`
  - `brelse()`