Filesystem: Block allocation

Youjip Won

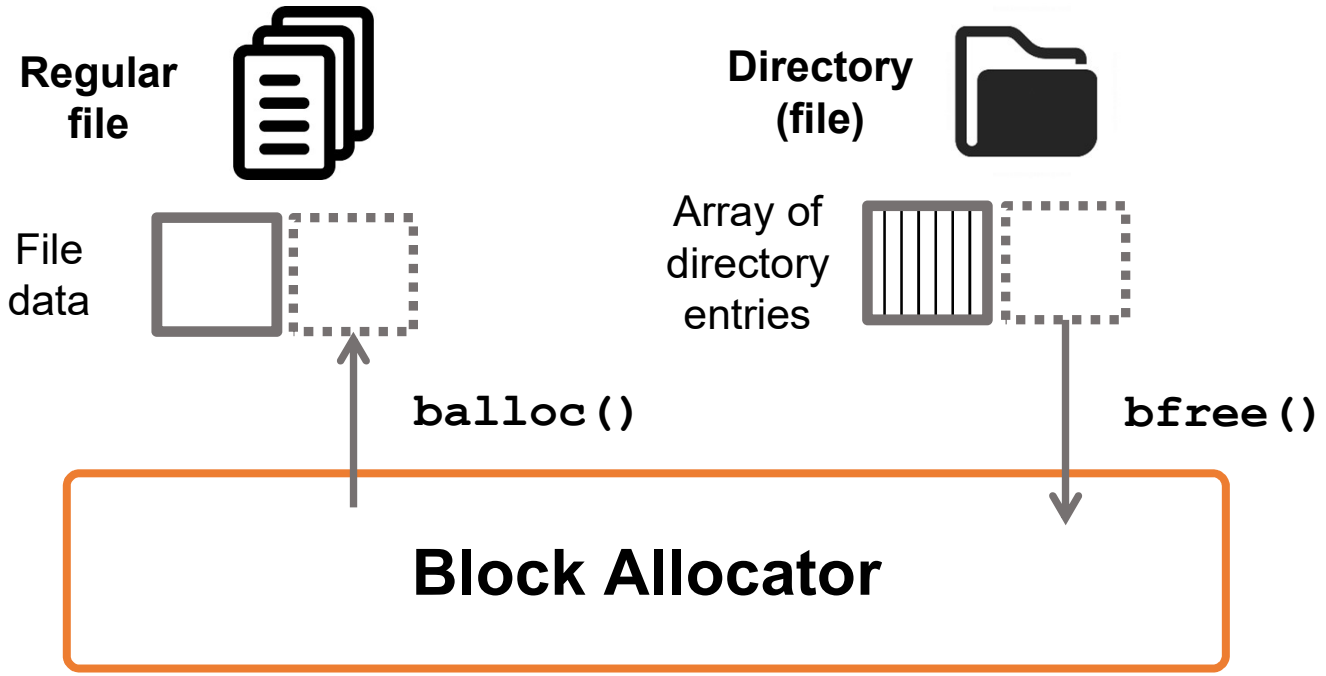
KAIST EE

Contents

- Concepts
- Block allocator
- Code: Block allocator
  - `balloc()`, and `bfree()`
- Address of file block
- Code
  - `bmap()`

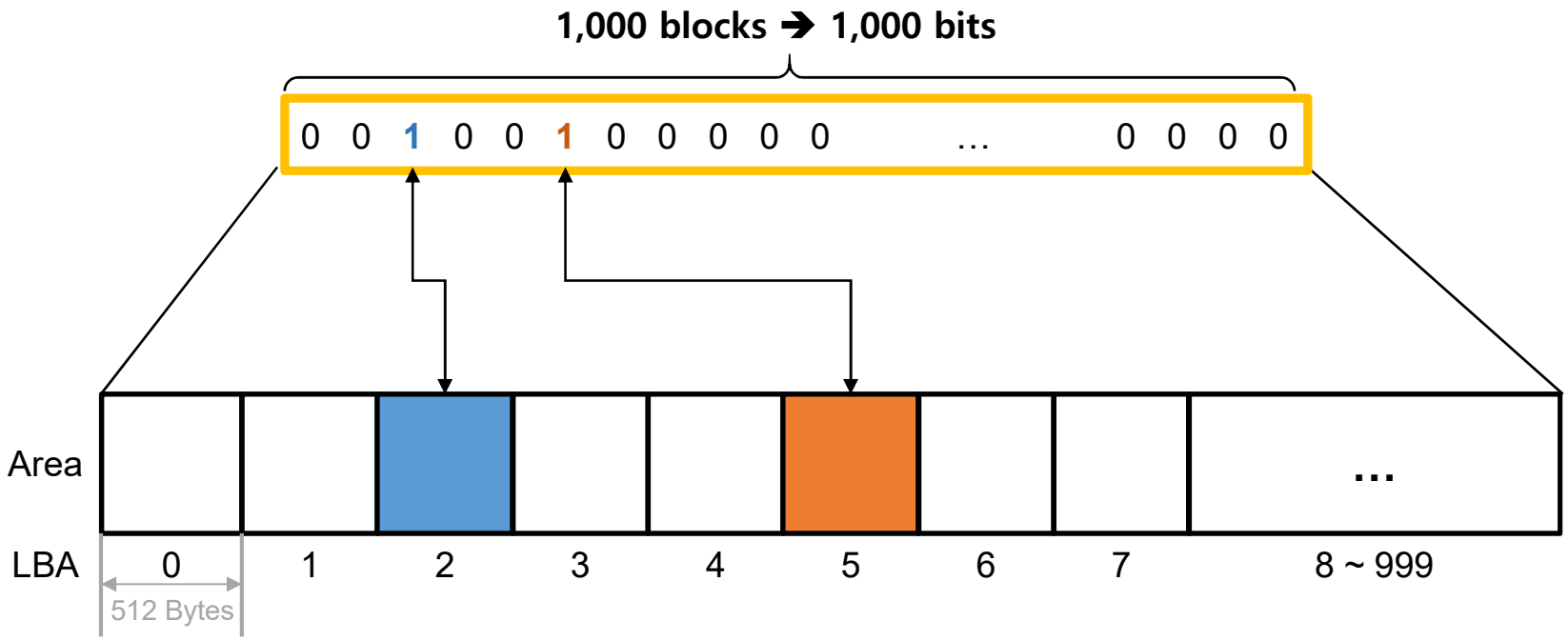
Concepts

- File and directory content is stored in disk blocks.
- The blocks must be allocated from block allocator.



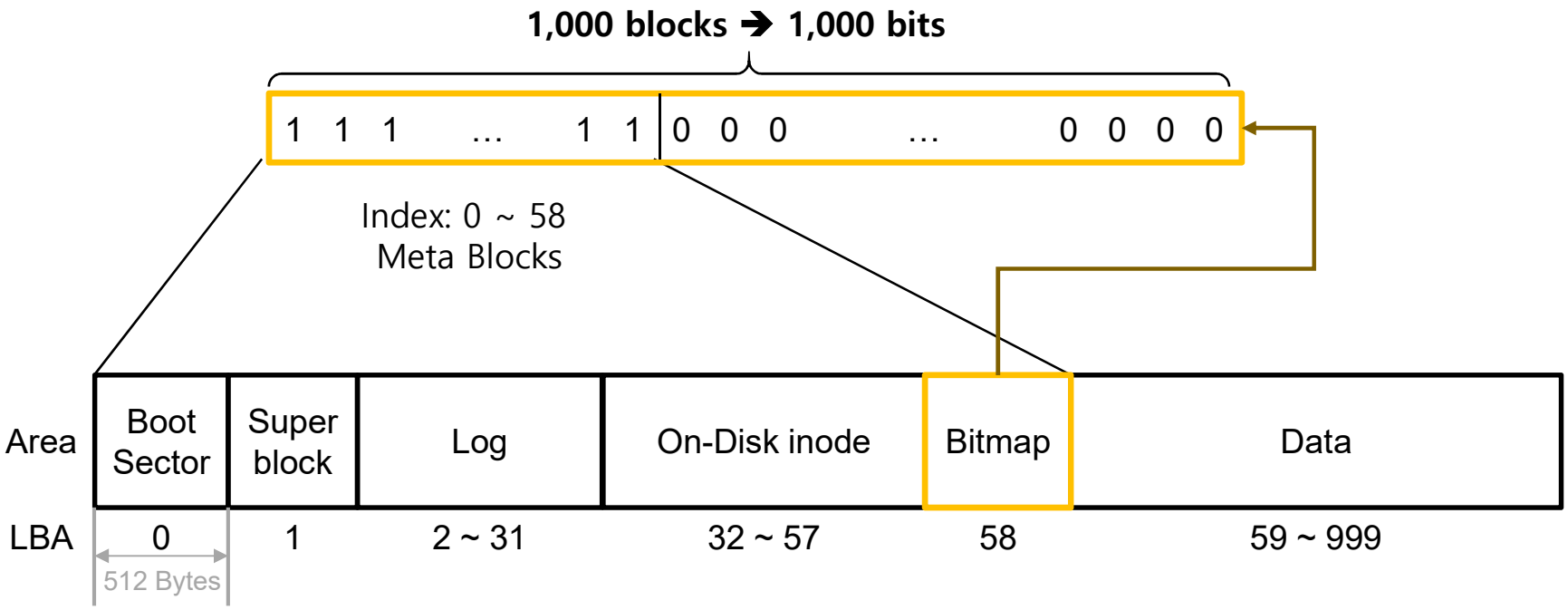
Block allocator - bitmap

- xv6's block allocator maintains a free bitmap on disk, with one bit per block.
- Each bit represents the status of a block located at the same index in the disk.
- A zero bit indicates that the corresponding block is free; a one bit indicates that it is in use.



Bitmap block

- Bitmap content is stored in the disk block, which is called bitmap block.
- The bits corresponding to the boot sector, superblock, log blocks, inode blocks, and bitmap blocks are always set.



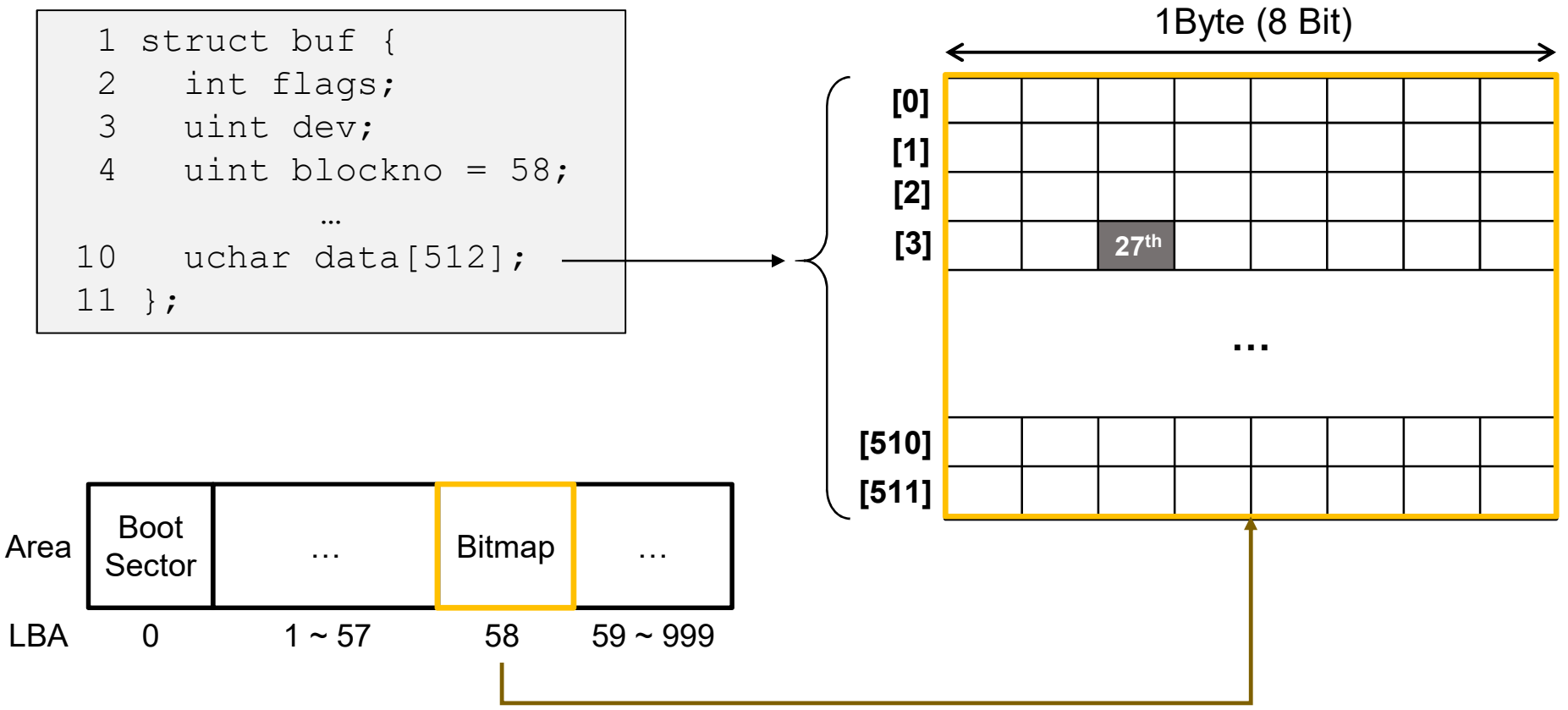
`balloc()` & `bfree()`

- The block allocator provides two functions.
  - `balloc()`: Mark the bit as in use to allocate a new disk block.
  - `bfree()`: Unset the corresponding bit to free the block.
- The unit of allocation is a block (512 bytes).
- xv6 uses the interface of the buffer cache and logging layers to read or update the bitmap block.
  - `struct buf`
  - `bread()` and `brelse()`
  - `log_write()`

Read or update the bitmap block content.

- Access the Nth bit ➔ (N%8)th bit of `data[`$\lfloor N/8 \rfloor$`]`
- ex) 27th bit ➔ 3rd bit of `data[3]`

```
 1 struct buf {
 2   int flags;
 3   uint dev;
 4   uint blockno = 58;
             …
10   uchar data[512];
11 };
```



uint balloc(uint dev)

- Allocate a new block and return the allocated block number.

  ① Read the bitmap block and find a free block.

  ② Mark the bit as in-use.

  ③ Log the updated bitmap block.

  ④ Initialize the block.

  ⑤ Return the block number.

`balloc()`: read the bitmap blocks.

- For efficiency, the scanning is split into two pieces.
  - xv6 does not need to read all bitmap blocks to find a free block.
  - The outer loop reads each block of bitmap blocks.
  - The inner loop checks all bits in a single bitmap block.

```
56 static uint
57 balloc(uint dev)
58 {
59   int b, bi, m;
60   struct buf *bp;
61
62   bp = 0;
63   for(b = 0; b < sb.size; b += BPB){
64     bp = bread(dev, BBLOCK(b, sb));
65     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
                                    …
75     brelse(bp);
76   }
77   panic("balloc: out of blocks");
78 }
```

Outer loop

Inner loop

balloc(): outer loop

- struct superblock sb: super block content
  - sb.size : total number of blocks in disk.
- BPB : the number of bits in a block. ➔ 512 byte X 8 bit
  - The number of blocks that can be covered by a single bitmap block.

```
56 static uint
57 balloc(uint dev)
58 {
59   int b, bi, m;
60   struct buf *bp;
61
62   bp = 0;
63   for(b = 0; b < sb.size; b += BPB){
64     bp = bread(dev, BBLOCK(b, sb));
                                  …
75     brelse(bp);
76   }
77   panic("balloc: out of blocks");
78 }
```

`balloc()`: outer loop (Cont.)

- `#define BBLOCK(b, sb) (b/BPB + sb.bmapstart)`
  - `sb.bmapstart` : block number of the first bitmap block = 58
  - Return the block number of the bitmap block containing the bitmap for the block of block number `b`.

```
56 static uint
57 balloc(uint dev)
58 {
59   int b, bi, m;
60   struct buf *bp;
61
62   bp = 0;
63   for(b = 0; b < sb.size; b += BPB){
64     bp = bread(dev, BBLOCK(b, sb));
                                        …
75     brelse(bp);
76   }
77   panic("balloc: out of blocks");
78 }
```

Read a single bitmap block.

`balloc()`: find a free block.

- The inner loop checks all bits in a single bitmap block if each block in the bitmap is free or not.
- `bi`: bit index = 0 ~ BPB - 1.

```
56 static uint
57 balloc(uint dev)
58 {
59   int b, bi, m;
60   struct buf *bp;
61
62   bp = 0;
63   for(b = 0; b < sb.size; b += BPB){
64     bp = bread(dev, BBLOCK(b, sb));
65     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
66       m = 1 << (bi % 8);
67       if((bp->data[bi/8] & m) == 0){  // Is block free?
                                    …
78 }
```

`balloc()`: find a free block. (Cont.)

- Look for a block whose bitmap bit is zero, indicating that the block is free.
- Bitwise & operation isolates (bi % 8)th bit from `data[`$\lfloor bi/8 \rfloor$`]`.

ex) bi = 27

```
m = 1 << 3 = 0b 0000 1000
bp->data[3] = 0b 0101 1111
```

bitwise &

```
bp->data[3] & m = 0b 0000 1000
```

```
56 static uint
57 balloc(uint dev)
58 {
59   int b, bi, m;
60   struct buf *bp;
61
62   bp = 0;
63   for(b = 0; b < sb.size; b += BPB){
64     bp = bread(dev, BBLOCK(b, sb));
65     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
66       m = 1 << (bi % 8);
67       if((bp->data[bi/8] & m) == 0){  // Is block free?
                                    …
78 }
```

`balloc()`: mark the free block as in use.

- If `balloc()` finds a free block, it sets the corresponding bit.

```
56 static uint
57 balloc(uint dev)
58 {
                                        …
67      if((bp->data[bi/8] & m) == 0){  // Is block free?
68         bp->data[bi/8] |= m;  // Mark the block as being in use.
69         log_write(bp);
70         brelse(bp);
71         bzero(dev, b + bi);
72         return b + bi;
73      }
                                      …
```

`balloc()`: log the updated bitmap block.

- Register the buffer cache entry of the bitmap block at the in-memory log structure.
- Then, release the buffer cache entry.

```
56 static uint
57 balloc(uint dev)
58 {
                                  …
67       if((bp->data[bi/8] & m) == 0){  // Is block free?
68          bp->data[bi/8] |= m;  // Mark block in use.
69          log_write(bp);
70          brelse(bp);
71          bzero(dev, b + bi);
72          return b + bi;
73       }
                                  …
```

`balloc()`: zero the block and return block number.

- Lastly, zero the block and return the block number.

```
56 static uint
57 balloc(uint dev)
58 {
                                  …
67       if((bp->data[bi/8] & m) == 0){  // Is block free?
68          bp->data[bi/8] |= m;  // Mark block in use.
69          log_write(bp);
70          brelse(bp);
71          bzero(dev, b + bi);
72          return b + bi;
73       }
                                  …
```

`void bfree(int dev, uint b)`

- Free a disk block.

  ① Unset the corresponding bit.

  ② Log the updated bitmap block.

`bfree()`: read the bitmap block.

- `#define BBLOCK(b, sb) (b/BPB + sb.bmapstart)`
  - Return the block number of the bitmap block containing the bitmap for the block of block number `b`.
- `bi`: bit index = 0 ~ BPB - 1.

```
81 static void
82 bfree(int dev, uint b)
83 {
84   struct buf *bp;
85   int bi, m;
86
87   bp = bread(dev, BBLOCK(b, sb));
88   bi = b % BPB;
89   m = 1 << (bi % 8);
90   if((bp->data[bi/8] & m) == 0)
91     panic("freeing free block");
                          …
95 }
```

`bfree()`: unset the bit and log the updated block.

- Unset the corresponding bit.
- Register the buffer cache entry of the bitmap block at the in-memory log structure.
- Then, release the buffer cache entry.

```
81 static void
82 bfree(int dev, uint b)
83 {
                          …
92   bp->data[bi/8] &= ~m;
93   log_write(bp);
94   brelse(bp);
95 }
```

LBA of file block

Inode

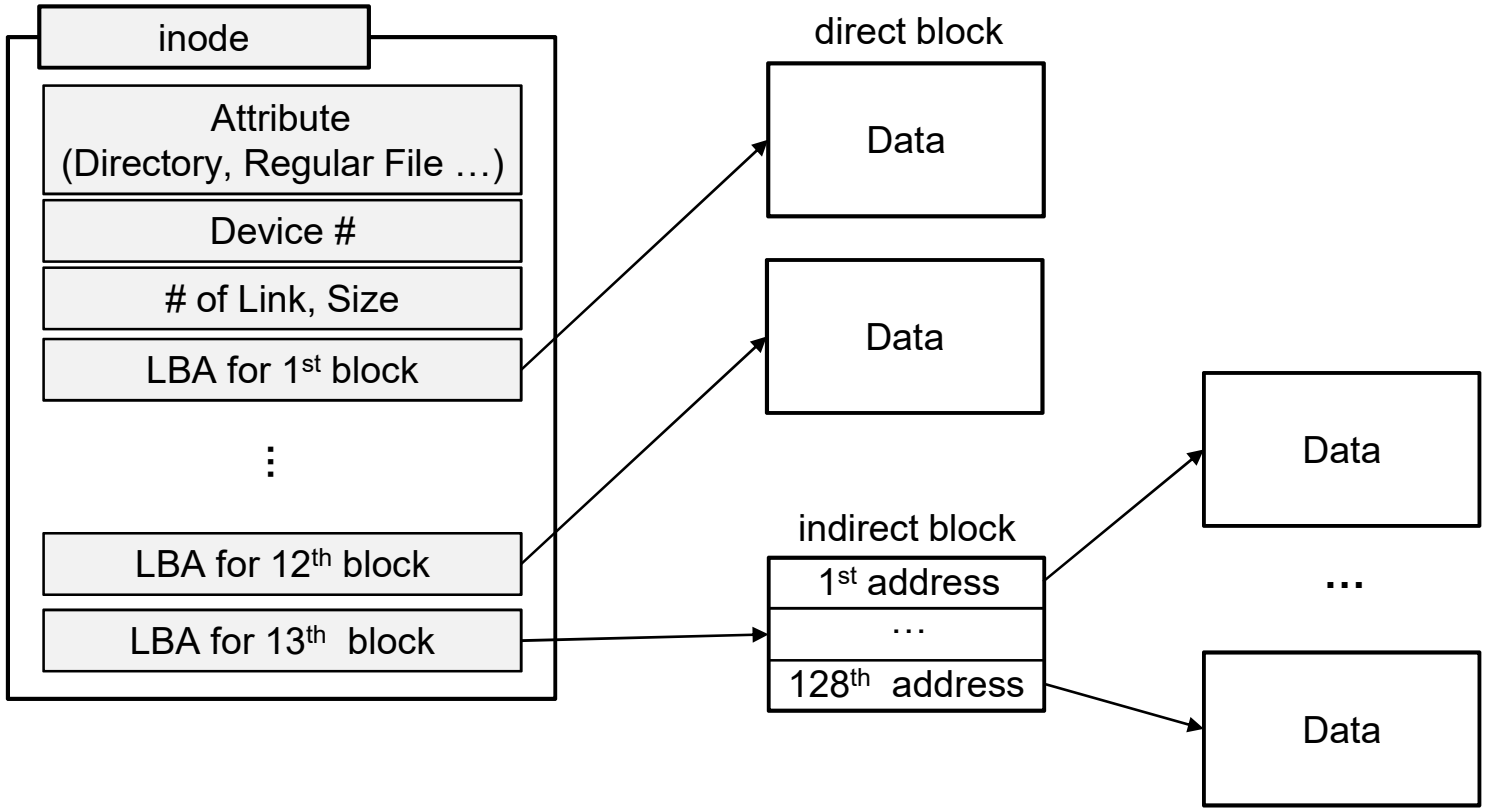
- Data structure to represent the attribute of file
  - file type: T_FILE (regular file), T_DIR (directory), or T_DEV (device file)
  - the number of links, file size
  - **locations of the file blocks : LBAs for 13 blocks**

```
#define NDIRECT 12

struct inode {
  …
  uint addrs[NDIRECT+1];   // File block addresses
};
```

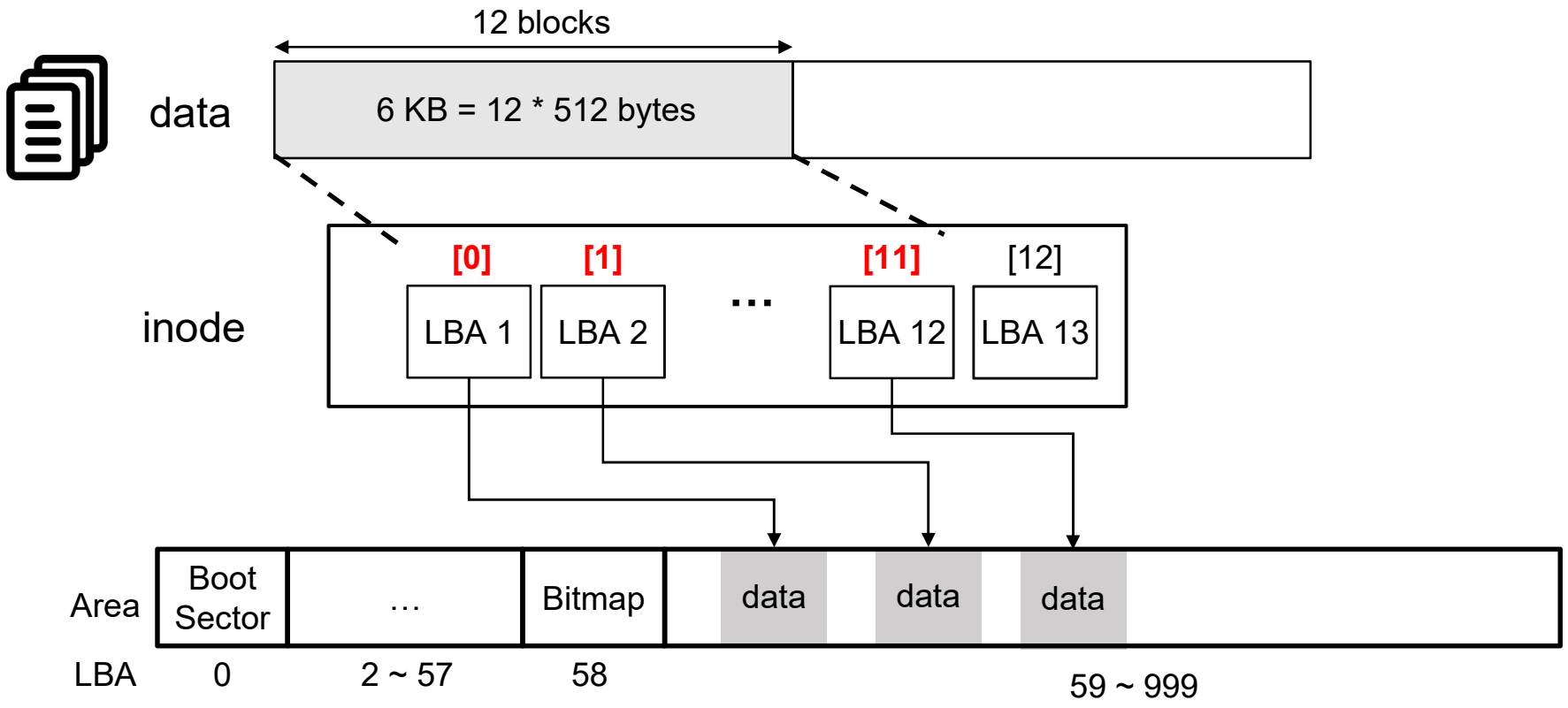
Direct and indirect block

- The maximum size of a file is 70 Kbytes.
  - (12 direct blocks + 1 indirect block * 128 blocks) * 512 bytes



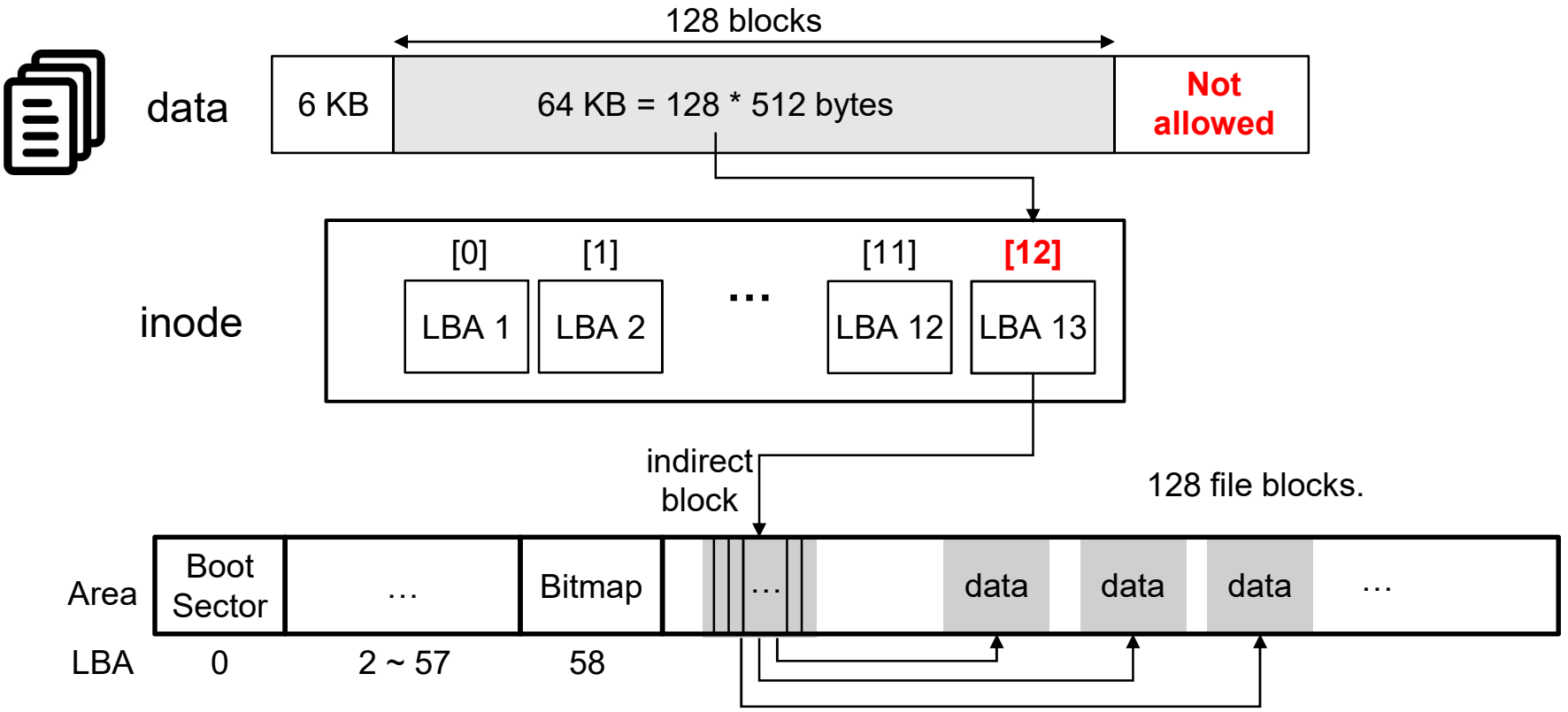
Direct block

- xv6's inode structure contains addresses of 12 direct blocks, which store the file data.
- The addresses are located at the first 12 entries of `addrs`.



Indirect block

- The last entry of `addrs` points to an indirect block.
- The content of indirect block is an array of addresses of file blocks.
- A single indirect block can cover the 128 file blocks.



uint bmap(struct inode *ip, uint bn)

- Return the disk block address of nth file block in inode.

```
#define NDIRECT 12
#define NINDIRECT 128
```

① `if (bn < NDIRECT)`

Return the disk block address of nth file block.

② `if (bn >= NDIRECT && bn < NDIRECT + NINDIRECT)`

Read the indirect block. Then, return the corresponding address in the indirect block.

③ `if (bn >= NDIRECT + NINDIRECT)`

System fails due to the access of invalid range.

- If there is no nth block, `bmap()` allocates one.
  - If the block address is 0, it denotes that there is no block.

`bmap()`: return the address of the file block.

- If `bn` is smaller than `NDIRECT`,
  - return the block address at index `bn` in `ip->addrs`.
- If the block address is 0, it allocates the new block.

```
#define NDIRECT 12
#define NINDIRECT 128
```

```
372 static uint
373 bmap(struct inode *ip, uint bn)
374 {
375   uint addr, *a;
376   struct buf *bp;
377
378   if(bn < NDIRECT){
379     if((addr = ip->addrs[bn]) == 0)
380       ip->addrs[bn] = addr = balloc(ip->dev);
381     return addr;
382   }
                              …
400 }
```

bmap(): Check the range.

- If `bn` is equal to or larger than `NDIRECT`,
  - it calculates the index of entry in the indirect block.
- If the index exceeds `NINDIRECT`, xv6 fails.

```
#define NDIRECT 12
#define NINDIRECT 128
```

```
372 static uint
373 bmap(struct inode *ip, uint bn)
374 {
                                        …
383   bn -= NDIRECT;
384
385   if(bn < NINDIRECT){
                                        …
397   }
398
399   panic("bmap: out of range");
400 }
```

`bmap()`: Read indirect block.

- To find the LBA in indirect block, it loads the indirect block.
- The block address of indirect block is located at index `NDIRECT` in `ip->addrs`.

```
372 static uint
373 bmap(struct inode *ip, uint bn)
374 {
                                        …
383   bn -= NDIRECT;
384
385   if(bn < NINDIRECT){
386     // Load indirect block, allocating if necessary.
387     if((addr = ip->addrs[NDIRECT]) == 0)
388       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
389     bp = bread(ip->dev, addr);
390     a = (uint*)bp->data;
                                        …
400 }
```

```
#define NDIRECT 12
```

bmap(): Allocate new indirect block.

- If the block address of indirect block is 0, allocate a new indirect block.

```
372 static uint
373 bmap(struct inode *ip, uint bn)
374 {
                                        …
383   bn -= NDIRECT;
384
385   if(bn < NINDIRECT){
386     // Load indirect block, allocating if necessary.
387     if((addr = ip->addrs[NDIRECT]) == 0)
388       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
389     bp = bread(ip->dev, addr);
390     a = (uint*)bp->data;
                                        …
400 }
```

`bmap()`: return the block address in indirect block.

- Return the block address at index `bn` in the indirect block.
- If the address is 0, allocated new block.

```
372 static uint
373 bmap(struct inode *ip, uint bn)
374 {
                                        …
389     bp = bread(ip->dev, addr); // Read indirect block.
390     a = (uint*)bp->data;
391     if((addr = a[bn]) == 0){
392       a[bn] = addr = balloc(ip->dev);
393       log_write(bp);
394     }
395     brelse(bp);
396     return addr;
397   }
398
399   panic("bmap: out of range");
400 }
```

Summary

- Block allocator
- Bitmap & bitmap block
- Code: `balloc()` and `bfree()`

- Address of file block
- Direct & indirect block
- Code: `bmap()`