xv6©-rev10
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)
Inode Layer I

Carmi Merimovich

Tel-Aviv Academic College

January 20, 2017

(memory) inode

```
4073 #define NDIRECT 12

4162 struct inode {
      uint dev; // Device number
      uint inum; // Inode number
      int ref; // Reference count
      struct sleeplock lock;
      int valid;

      short type; // copy of disk inode
      short major;
      short minor;
      short nlink;
      uint size;
      uint addrs[NDIRECT+1];
    };
```

(memory) inode pool

```
struct {
 struct spinlock lock;
 struct inode inode[NINODE];
} icache;
```

Implemented Functions

- iget(dev,inum).
- idup(ip).
- ialloc(dev,type).
- iput(ip).
- ilock(ip).
- iunlock(ip).
- iunlockput(ip).
- iupdate(ip).
- readi(ip,buf,off,length)
- writei(ip,buf,off,length)
- itrunc(ip);

iget

- An inode is identified by its device number and inode number.
- There will be at most one (memory) inode specifying a pair (device number, inode number).
- iget returns a pointer to a memory inode given its id.
- If this memory inode already exists, it is used, and its refcnt increased.
- If there is no such memory inode, it is allocated, and its id is set.
- (The actual inode values are not in memory yet!).

iget(dev, inum) logic

1. Search the icache vector for inode matching dev and num.
   1.1 If found, increase its reference count and returns it.
   1.2 If not found, use an empty entry in the icahche vector, and fill it with metadata.

iget(1): Search for the inode

```
static struct inode *iget(uint dev, uint inum) {
 struct inode *ip, *empty;

 acquire(&icache.lock);

 empty = 0;
 for(ip=&icache.inode[0];ip<&icache.inode[NINODE];ip++) {
  if (ip->ref > 0 && ip->dev == dev && ip->inum == inum){
   ip->ref++;
   release(&icache.lock);
   return ip;
  }
  if (empty == 0 && ip->ref == 0) // Remember empty slot.
   empty = ip;
 }

 if (empty == 0)
  panic("iget:␣no␣inodes");
```

iget(2): setup inode meta information

```
 ip = empty;
 ip->dev    = dev;
 ip->inum   = inum;
 ip->ref    = 1;
 ip->flags = 0;
 release(&icache.lock);

 return ip;
}
```

idup

```
struct inode *idup(struct inode *ip) {
 acquire(&icache.lock);
 ip->ref++;
 release(&icache.lock);
 return ip;
}
```

Discuss:

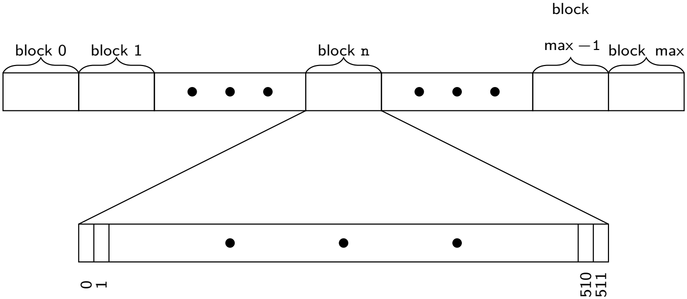
```
 iget(ip->dev, ip->inum);
```

We aim at ialloc.

Beforehand we need information about the buffer layer.

Reason for the buffer layer

Abstract disk structure:



- Disk operations are ALWAYS on block boundaries.
- Each block is 512 bytes.
- This structure is factory formatted nowadays.

We use the buffer layer for R/W

```
3501 struct buf {
      int flags;
      uint dev;
      uint blockno;
      struct sleeplock lock;
      uint refcnt;
      struct buf *prev;
      struct buf *next;
      struct buf *qnext;
      uchar data[BSIZE];
     };
```

```
4063 struct superblock {
      uint size; // Size of file syste
      uint nblocks; // Number of data
      uint ninodes; // Number of inod
      uint nlog; // Number of log blo
      uint logstart; // Block number
      uint inodestart; // Block numbe
      uint bmapstart; // Block number
     };
```

- readsb(dev, &sb).
- buf = bread(dev,sector).
- log_write(buf).
- brelse(buf).

- balloc(dev).
- bfree(dev,sector).

IBLOCK(

The (disk) `inode`

- The (disk) inodes are continuous on the disk.
- Each (disk) inode has a number.

Locating block numbers

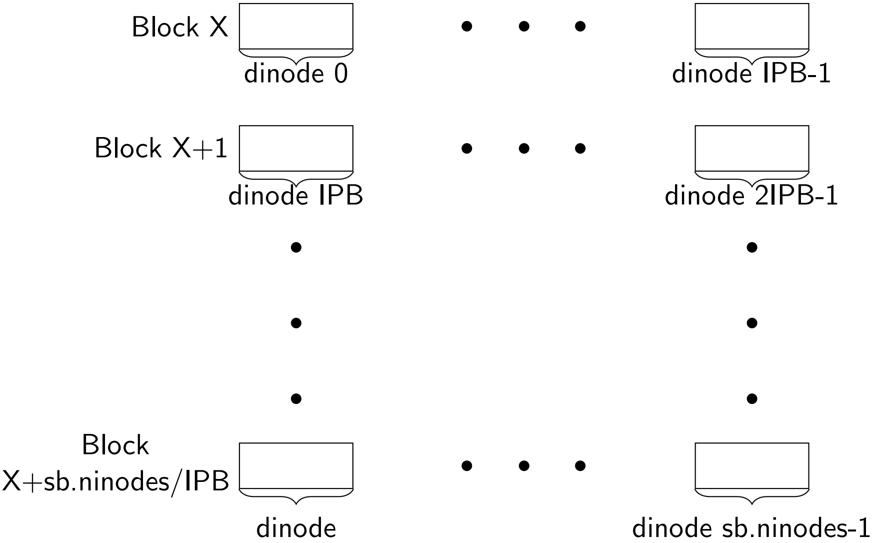
```
#define BSIZE 512 // block size

#define IPB (BSIZE / sizeof(struct dinode))

// Block containing inode i
#define IBLOCK(i,sb)  (i / IPB + sb.inodestart)

// Bitmap bits per block
#define BPB (BSIZE*8)

// Block containing bit for block b
#define BBLOCK(b, sb) (b / BPB + sb.bmapstart)
```

disk inodes

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

- The disk inodes are continuous on disk.
- The IBLOCK(inum,sb) macro returns the block number containing inode num.

disk inode area (on disk)



Block X

dinode 0

dinode IPB-1

Block X+1

dinode IPB

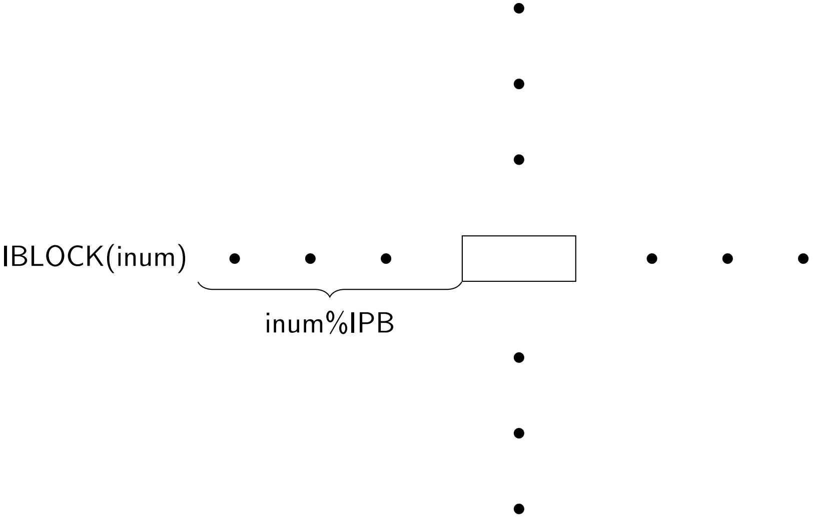
dinode 2IPB-1

Block X+sb.ninodes/IPB

dinode

dinode sb.ninodes-1

pinpointing inode inum



ialloc(dev,type) logic

1. Search, on disk, for unused disk inode. If not found, panic (which is a bit harsh).
2. return iget(dev,inum).

Observe:

- In this case, iget returns inode which is NOT I_VALID.
- Since the inode was free, necessarily ref in the memory inode will be set to one.

reading a block from disk

- The sequence to read from disk is:

```
bp = bread(dev, nblock);
⋮
Use bp->data
⋮
brelse(bp);
```

- The buffer layer has an inernal lock mechanism.
- Thus, if we read a block, others attempting to read it will go SLEEPING.
- Do not hold bp unnecessarily,
- As soon as the use of bp− >data is finished, brelse the block.

Isolating dinode in buffer

$$\mathtt{bp} = \mathtt{bread}(\mathtt{dev}, \mathtt{IBLOCK}(\mathtt{inum}));$$

$$\mathtt{struct\ dinode}\ \ *\,\mathtt{dip} = ((\mathtt{structdinode}*) \qquad \mathtt{bp}- > \mathtt{data}) + (\mathtt{inum}\%\mathtt{IPB});$$

Use dip$- >$

$\vdots$

$\mathtt{brelse}(\mathtt{bp});$

ialloc

```
struct inode *ialloc(uint dev, short type) {
 struct buf *bp;
 struct dinode *dip;

 for (int inum = 1; inum < sb.ninodes; inum++) {
  bp = bread(dev, IBLOCK(inum,sb));
  dip = (struct dinode*)bp->data + inum%IPB;
  if (dip->type == 0) { // a free inode
   memset(dip, 0, sizeof(*dip));
   dip->type = type;
   log_write(bp); // mark it allocated on the disk
   brelse(bp);
   return iget(dev, inum);
  }
  brelse(bp);
 }
 panic("ialloc:␣no␣inodes");
}
```

iupdate()

```c
void iupdate(struct inode *ip) {
  struct buf *bp;
  struct dinode *dip;

  bp = bread(ip->dev, IBLOCK(ip->inum,sb));
  dip = (struct dinode*)bp->data + ip->inum%IPB;
  dip->type  = ip->type;
  dip->major = ip->major;
  dip->minor = ip->minor;
  dip->nlink = ip->nlink;
  dip->size  = ip->size;
  memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
  log_write(bp);
  brelse(bp);
}
```

acquiresleep

```
3901 struct sleeplock {
      uint locked; // Is the lock held?
      struct spinlock lk; // spinlock protecting this sle

      char *name; // Name of lock.
      int pid; // Process holding lock
     };

4622 void acquiresleep(struct sleeplock *lk) {
      acquire(&lk->lk);
      while (lk->locked) {
       sleep(lk, &lk->lk);
      }
      lk->locked = 1;
      lk->pid = myproc()->pid;
      release(&lk->lk);
     }
```

releasesleep

```
3901 struct sleeplock {
      uint locked; // Is the lock held?
      struct spinlock lk; // spinlock protecting this sle

      char *name; // Name of lock.
      int pid; // Process holding lock
     };

4634 void releasesleep(struct sleeplock *lk) {
      acquire(&lk->lk);
      lk->locked = 0;
      lk->pid = 0;
      wakeup(lk);
      release(&lk->lk);
     }
```

ilock()

```
void ilock(struct inode *ip) {
 struct buf *bp;
 struct dinode *dip;

 if (ip == 0 || ip->ref < 1)
  panic("ilock");

 acquiresleep(&ip->lock);
```

ilock (2)

```
  if (ip->valid==0) {
   bp = bread(ip->dev, IBLOCK(ip->inum,sb));
   dip = (struct dinode*)bp->data + ip->inum%IPB;
   ip->type  = dip->type;
   ip->major = dip->major;
   ip->minor = dip->minor;
   ip->nlink = dip->nlink;
   ip->size  = dip->size;
   memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
   brelse(bp);
   ip->valid =;
   if (ip->type == 0)
    panic("ilock:␣no␣type");
  }
}
```

iunlock

```
void iunlock(struct inode *ip) {
 if (ip == 0 || !holdingsleep(&ip->lock) || ip->ref
  panic("iunlock");

 releasesleep(&ip->lock);
}
```

iput

```
5358 void iput(struct inode *ip) {
 acquiresleep(&ip->lock);
 if (ip->valid == 1 && ip->nlink == 0) {
  acquire(&icache.lock);
  int r = ip->ref;
  release(&icache.lock);
  if (r == 1) {
   itrunc(ip);
   ip->type = 0;
   iupdate(ip);
   ip->valid=0;
  }
 }
 releasesleep(&ip->lock);
 acquire(&icache.lock);
 ip->ref--;
 release(&icache.lock);
}
```